**Debreceni Egyetem**
**Informatikai Kar**

# Megoldáskeresés Sokoban játékban

**Témavezető:**

Dr. Halász Gábor

*egyetemi docens*

**Készítő:**

Horváth Attila

*programtervező informatikus (B.sc)*

Debrecen
2009.

# Tartalomjegyzék

**Bevezetés** **5**
Célkitűzés . . . . . 5
A játékról . . . . . 5

**1. Állapottér-reprezentáció** **7**
1.1. Állapottér . . . . . 7
1.1.1. A labirintus . . . . . 7
1.1.2. Kényszerfeltételek . . . . . 8
1.2. Kezdőállapot . . . . . 8
1.3. Célállapotok halmaza . . . . . 8
1.4. Operátor . . . . . 8
1.4.1. Az operátor hatása . . . . . 9
1.4.2. Az operátor alkalmazásának előfeltétele . . . . . 9
1.5. Implementáció . . . . . 10
1.5.1. Az operátor implementációja . . . . . 13
1.6. Az állapottér-reprezentációs gráf . . . . . 14
1.6.1. Az elágazási tényező . . . . . 14
1.6.2. Az állapottér mérete . . . . . 15

**2. A kereső algoritmus** **17**
2.1. Kritériumok . . . . . 17
2.2. IDA* . . . . . 17
2.2.1. Az IDA* implementációja . . . . . 19
Zárógondolatok . . . . . 19
2.3. A heurisztika . . . . . 20
2.3.1. A relaxált feladat . . . . . 21
2.3.2. A hozzárendelési feladat . . . . . 21
2.3.3. Büntető függvény . . . . . 22

**3. Holtpontok** **25**
3.1. Halott mezők . . . . . . . . . . 25
3.2. Párosítási probléma . . . . . . . . . . 26
3.3. Korall probléma . . . . . . . . . . 27
3.4. Alagutak . . . . . . . . . . 28

**Összegzés** **29**
Zárógondolatok . . . . . . . . . . 29
A mellékelt program . . . . . . . . . . 29
Köszönetnyilvánítás . . . . . . . . . . 30

**Irodalomjegyzék** **31**

**Ábrák jegyzéke** **33**

# Bevezetés

> *"Mindenki úgy tekint a számítógépre, mint egy eszközre, holott az nem egy eszköz, hanem egy kapu. Kapu egy másik világra, egy olyan világra, melynek peremvidékeit csak most kezdjük felfedezni."*
>
> Alan Dean Foster

## Célkitűzés

A dolgozat első felében a játék egy lehetséges absztrakt matematikai modellje kerül bemutatásra. Ebben az *állapottér-reprezentáció*ban az egyszerűség lesz az irányadó, csupán primitív adatszerkezeteket tartalmaz majd. Ezáltal biztosított, hogy szinte bármely programnyelven megvalósítható. Az implementáció során azonban érdemes növelni az absztrakciós szintet s alkalmas adattípusokra cserélni a modell egyes elemeit. Sőt számos további információ tárolására is szükség lesz egy valóban hatékony rendszer felépítéséhez.

A dolgozat második részében szó esik a (mellékelt programban is) használt kereső algoritmus tulajdonságairól, illetve egy haladó szintű *Solver* kifejlesztéséhez elengedhetetlen technikákról. Utóbbiak egyrészt ésszerű javítások, olyan vágások a keresési gráfban, melyek jelentősen csökkentik a kiterjesztett csomópontok számát, másrészt olyan trükkök, melyekkel a kereső motort a „megfelelő irányba" tereljük. Bemutatásra kerülnek a lehetséges "iránymutatók" is, azaz a hatékony és kevésbé hatékony heurisztikák.

## A játékról

A Sokoban egy ismert és népszerű logikai játék, melyet *Imabajasi Hirojuki* japán programozó alkotott meg 1980-ban. A cél és a szabályok egyszerűek. A *labirintus*ban található összes *ládá*t el kell juttatnunk egy-egy *célmező*re az *emberke* segítségével. Az

emberke az aktuális pozíciójával négyszomszédos mezőre léphet, illetve odébb tolhat egy vele négyszomszédos ládát, ha az üres mezőre kerül.

Az évek során több változat is megjelent a piacon, újabb kiadások láttak napvilágot. Ezek részben módosították az eredeti koncepciót, a legtöbb esetben azonban csak egyszerű klónokról van szó.

A Sokoban a bonyolultságelmélet témakörébe tartozik. A játék megoldása bizonyítottan NP-nehézségű. Egy kezdetleges reprezentációval és keresővel, a kiterjesztett csomópontok száma exponenciálisan növekszik a keresési gráf szintjének függvényében. A mesterséges intelligenciával foglalkozók számára tehát komoly kihívást jelent a probléma, hiszen a nagy elágazási faktor miatt nem kezelhető a szokványos keretek között. Létfontosságú a minél hatékonyabb reprezentáció, a felesleges állapotok elkerülése, a holtpontok kiküszöbölése, releváns vágások a keresési gráfban, hatékony heurisztika, valamint egy kevésbé tárigényes, de elég gyors kereső algoritmus is. Az egyik legjobb keresőrendszer a YASC ([1]) nevű szoftver. A kutatások azt is megerősítették, hogy a Sokoban *PSPACE-teljes* ([2]).

A szakirodalomban konvencióvá vált az 1. ábrán látható jelölésrendszer. Egyszerűsége miatt bármilyen konzolos környezetben megjeleníthető.

| **objektum** | **jel** |
|---|---|
| emberke | @ |
| láda | $ |
| falmező | # |
| üresmező | ␣ |
| célmező | . |
| emberke egy célmezőn | + |
| láda egy célon | * |

1. ábra. Jelölési konvenciók

# 1. fejezet

# Állapottér-reprezentáció

*„A féltudás győzedelmesebb, mint az egész tudás: egyszerűbbnek tudja a dolgokat, mint amilyenek s ezért megfoghatóbban és meggyőzőbben alkotja meg róluk a véleményét.”*

Friedrich Nietzsche

## 1.1. Állapottér

Legyen a továbbiakban $n$ a labirintusban található figurák száma.

A probléma világának állapotait rendezett elem $2 * n$-esok írják le. Legyen

$$A \subseteq A_{11} \times \cdots \times A_{2n},$$

ahol

$$A_{ij} \in \{\, 0, 1, 2, \dots \},\qquad i = 1, 2, j = 1, \dots, n.$$

Az első oszlopvektor írja le a figura sorát és oszlopát, a többi oszlopvektor pedig a ládák pozicióját a táblán.

### 1.1.1. A labirintus

Legyen $T$ a labirintust leíró mátrix. Ekkor

$$T = (T(i,j))_{s \times o} = \begin{cases} 0, & \text{ha az } (i,j) \text{ mező üres,} \\ 1, & \text{ha az } (i,j) \text{ mező fal,} \\ 2, & \text{ha az } (i,j) \text{ mező cél.} \end{cases}$$

ahol $s$ a labirintus sorainak, $o$ pedig oszlopainak száma.

### 1.1.2. Kényszerfeltételek

A kényszerfeltétel az alábbi részfeltételek konjukciójaként írható fel.

Az emberke és a ládák nem állhatnak falmezőkön:

$$(K1) \qquad \forall i \, [T(a_{1i}, a_{2i}) \neq 1] \, , \qquad i = 1, 2, j = 1, \ldots, n.$$

Az állapotmátrix oszlopvektorai páronként különböznek, azaz az emberke és a ládák nem lehetnek azonos mezőkön.

$$(K2) \qquad \forall i \forall j \, [a_{1i} = a_{1j} \wedge a_{2i} = a_{2j} \subset i = j] \, , \qquad i = 1, 2, j = 1, \ldots, n.$$

## 1.2. Kezdőállapot

Kezdőállapot lehet minden

$$k = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \end{pmatrix}$$

alakú mátrix, amely kielégíti a fentebb ismertetett *kényszerfeltételeket*.

## 1.3. Célállapotok halmaza

A célállapotok halmaza:

$$C = \{ \, (a_{11}, \ldots, a_{2n}) \in A : \text{célfeltétel}(a_{11}, \ldots, a_{2n}) \, \},$$

ahol a célfeltételt megadó formula:

$$\forall i \, [T(a_{1i}, a_{2i}) = 2] \, , \qquad i = 2, 3, \ldots, n.$$

## 1.4. Operátor

Egyetlen operátort definiálunk, amely az adott állapotban az emberkével a paraméterként kapott irányba lép. Az egyes irányokat egész számpárokkal fogjuk jelölni. Legyen

$$\Delta = (\delta_{ij})_{2 \times 4} = \begin{pmatrix} 0 & -1 & 0 & 1 \\ -1 & 0 & 1 & 0 \end{pmatrix}$$

az a mátrix, amelynek oszlopvektorai megadják, hogy az emberkével az adott irányba lépve mennyit kell hozzáadni az emberke sor-, illetve oszlopkoordinátájához, hogy az operátor működése helyes legyen.

### 1.4.1. Az operátor hatása

A fentiek alapján az operátort az alábbi módon adhatjuk meg:

$$\text{lép}\colon \text{dom}(\text{lép}) \to A,$$

ahol

$$\text{dom}(\text{lép}) \subseteq A \times D, \qquad D = \{\,1, 2, 3, 4\,\}.$$

Az 1, 2, 3, 4 számok jelentsék rendre a balra, fel, jobbra, le irányokat.

Továbbá vezessük be az alábbi segédpredikátumokat.

$$(P1) \qquad \text{LÁDA}(a, (x, y)) = \forall i\,[2 \le i \le 4 \supset (a_{1i} = x \wedge a_{2i} = y)]$$

Igaz, ha az adott mezőn láda van.

$$(P2) \qquad \text{FAL}((x, y)) = [T(x, y) = 1]$$

Igaz, ha az adott mező falmező.

$$(P3) \qquad \text{ÜRES}(a, (x, y)) = \neg\text{LÁDA}(a, (x, y)) \wedge \neg\text{FAL}((x, y))$$

Igaz, ha az adott mezőn nincs láda és nem fal, azaz egy *léphető* mező.

Az operátor hatása az $(a_{11}, \ldots, a_{2n}) \in A$ állapot és $d \in D$ irány esetén:

$$\text{lép}((a_{11}, \ldots, a_{2n}), d) = (a'_{11}, \ldots, a'_{2n})$$

$$a'_{11} = a_{11} + \delta_{1d}$$
$$a'_{21} = a_{21} + \delta_{2d}$$

az emberke új poziciója, valamint

$$a'_{ij} = \begin{cases} a_{ij} + \delta_{1d}, & \text{ha } i = 1 \wedge a_{ij} = a'_{11} \wedge a_{2j} = a'_{21}, \qquad i = 1, 2 \\ a_{ij} + \delta_{2d}, & \text{ha } i = 2 \wedge a_{1j} = a'_{11} \wedge a_{ij} = a'_{21}, \qquad j = 1, 2, \ldots, n \\ a_{ij}, & \text{egyébként.} \end{cases}$$

### 1.4.2. Az operátor alkalmazásának előfeltétele

A formula az alábbi módon írható fel:

$$\text{ÜRES}(a, (a_{11} + \delta_{1d}, a_{21} + \delta_{2d})) \vee [\text{LÁDA}(a, (a_{11} + \delta_{1d}, a_{21} + \delta_{2d})) \supset$$
$$\supset \text{ÜRES}(a, (a_{11} + 2 * \delta_{1d}, a_{21} + 2 * \delta_{2d}))]$$

Azaz ellenőrizzük, hogy az emberke előtti mező üres-e, vagy ha láda van ott, akkor az aktuális irányban a láda utáni mezőnek kell üresnek lennie.

## 1.5. Implementáció

Lássuk a fenti állapottér-reprezentáció *Java* nyelvű megvalósítását:

```
package sokoban;

import java.util.Arrays;
import java.util.EnumSet;
import java.util.Formatter;

public class Sokoban implements Cloneable {

    private int[][] labirintus;

    private int[][] poziciok;

    private int LadakSzama;

    public int getLadakSzama() {
        return ladakSzama;
    }

    public int[][] getPoziciok() {
        return poziciok;
    }

    public int getKoltseg(){
        return 1;
    }

    public static Sokoban AlkalmazOperatort(Irany irany,
                                            Sokoban allapot) {
        allapot.alkalmaz(irany);
        return allapot;
    }

    public Sokoban(int[][] labirintus, int[][] poziciok) {
        this.labirintus = labirintus;
        this.poziciok = poziciok;
    }

    public boolean isCelAllapot() {
        for(int i = 1; i < poziciok[0].length; ++i) {
            if (labirintus[poziciok[0][i]][poziciok[1][i]] != 2) {
                return false;
            }
        }
        return true;
```

```
}

public boolean isAlkalmazhato(Irany irany) {
    if(labirintus[poziciok[0][0] + irany.getDx()]
                 [poziciok[1][0] + irany.getDy()] == 1) {
        return false;
  }
    if((isUres(poziciok[0][0] + irany.getDx(), poziciok[1][0]
       + irany.getDy()))
    ||(!isLada(poziciok[0][0] + irany.getDx(), poziciok[1][0]
       + irany.getDy())
    ||  isUres(poziciok[0][0] + 2 * irany.getDx(), poziciok
       [1][0] + 2 * irany.getDy()))) {
        return true;
    }
    return false;
}

public void alkalmaz(Irany irany) {
    int [][] ujPoziciok = new int[2][];

    for (int i = 0; i < poziciok.length; i++) {
        ujPoziciok[i] = poziciok[i].clone();
    }
    ujPoziciok[0][0] = poziciok[0][0] + irany.getDx();
    ujPoziciok[1][0] = poziciok[1][0] + irany.getDy();
    for (int i = 0; i < poziciok.length; i++) {
        for (int j = 1; j < poziciok[0].length; j++) {
            if( i == 0 && poziciok[i][j] == ujPoziciok[0][0]
            && poziciok[1][j] == ujPoziciok[1][0]) {
                ujPoziciok[i][j] += irany.getDx();
            }
            else if( i == 1 && poziciok[0][j] == ujPoziciok
                [0][0] {
            && poziciok[1][j] == ujPoziciok[1][0])
                ujPoziciok[i][j] += irany.getDy();
            }
        }
    }
    poziciok = ujPoziciok;
}

public boolean isLada(int x, int y) {
    for(int i=1; i<poziciok[0].length; i++) {
        if(poziciok[0][i] == x && poziciok[1][i] == y) {
            return true;
        }
    }
    return false;
```

```
    }

    public boolean isUres(int x, int y){
        if(!isLada(x, y) && labirintus[x][y] != 1){
            return true;
        }
        return false;
    }

    public EnumSet<Irany> getOperatorok() {
        EnumSet<Irany> operatorok = EnumSet.allOf(Irany.class);
        for (Irany irany :Irany.values()) {
            if(!isAlkalmazhato(irany)){
              operatorok.remove(irany);
            }
        }
        return operatorok;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        } else if (o == null || ! getClass().equals(o.getClass()))
            return false;
        Sokoban masik = (Sokoban) o;
        return Arrays.deepEquals(poziciok, masik.poziciok);
    }

    @Override
    public int hashCode() {
        return Arrays.deepHashCode(poziciok);
    }

    @Override
    public Sokoban clone() {
        Sokoban copy = null;
        try {
            copy = (Sokoban) super.clone();
        } catch(CloneNotSupportedException e) {}
        copy.poziciok = this.poziciok.clone();
        return copy;
    }

    @Override
    public String toString() {
        Formatter formatter = new Formatter();
        for (int i = 0; i < labirintus.length; i++) {
            for (int j = 0; j < labirintus[0].length; j++) {
```

```
                int k = 0;
                boolean azonos = false;
                for(k = 0; k < poziciok[0].length; k++) {
                    if(azonos = poziciok[0][k] == i && poziciok
                        [1][k] == j)
                        break;
                }
                if(azonos)
                    if (k == 0)
                        formatter.format(" E ");
                    else
                        formatter.format(" G ");
                else
                    formatter.format("%2d ", labirintus[i][j]);
            }
            formatter.format("\n");
        }
        return formatter.toString();
    }
}
```

Az osztály megvalósítja továbbá a kereső rendszer helyes működéséhez nélkülözhetetlen metódusokat. Ilyen a felüldefiniált *equals()*, *hashCode()* és *clone()* is. Valamint konzolos környezetben használható, karakterlánc reprezentációt megvalósító *toString()* metódus is megvalósításra (felüldefiniálásra) került.

### 1.5.1. Az operátor implementációja

Az operátort megvalósító osztály:

```
package sokoban;

public enum Irany {

    BALRA(0,-1),
    FEL(-1,0),
    JOBBRA(0,1),
    LE(1,0);

    private int dx;

    public int getDx() {
        return dx;
    }

    private int dy;
```

```
        public int getDy() {
            return dy;
        }

        private Irany(int dx, int dy){
            this.dx = dx;
            this.dy = dy;
        }
    }
```

Tehát minden irány (operátor) egy-egy enumkonstans, melynek két attributúma rendre a reprezentált irány sor- és oszlopnövekménye. Az osztálynak a modellben a $\delta$ mátrix felel meg.

## 1.6. Az állapottér-reprezentációs gráf

A gráfról az alábbi egyszerű megállapításokat tehetjük:

- A gráf köröket és hurkokat is tartalmaz. Ezek a problémák egyrészt egy hatékonyabb reprezentációval, másrészt közvetlen ellenőrzéssel *kiküszöbölhetőek*.
- Egy operátor alkalmazása után az újonnan előállt állapotra mindig alkalmazható annak *névleges* „inverze”. Ezt az operátort alkalmazva azonban nem feltétlenül jutunk a kiinduló állapothoz. Szükséges és elegendő feltétel, hogy egyszerű (tolás nélküli) lépésről legyen szó. Így az *inverz operátor* megnevezés matematikailag nem korrekt [1].

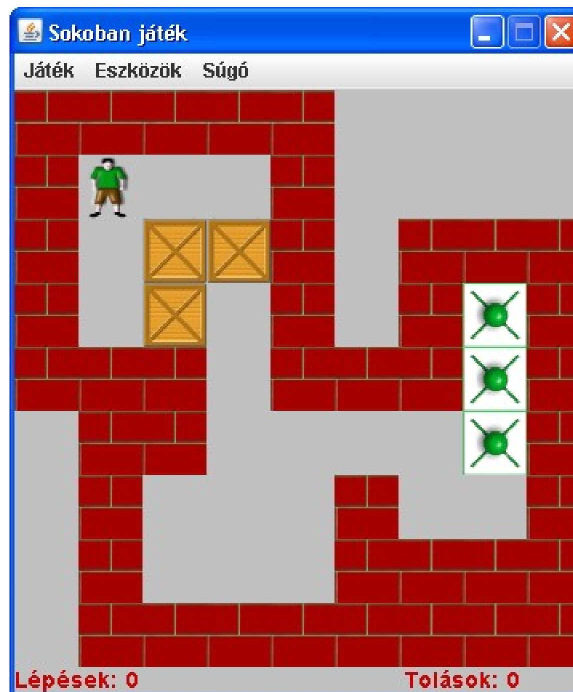
### 1.6.1. Az elágazási tényező

A fenti megállapításokból következik, hogy minden *érvényes* állapotra alkalmazható legalább egy operátor, felső korlát pedig természetesen az előzőekben definiált négy lehet.

[1] Könnyen konstruálható olyan reprezentáció, ahol az inverz operátor egyértelmű.

### 1.6.2. Az állapottér mérete

A csomópontok számának (elméleti) felső korlátját $m$ elem $k$-ad osztályú ismétlés nélküli kombinációinak száma adja, ahol $m$ a léphető mezők, $f$ pedig a figurák száma. Példaként tekintsük az alábbi ábrán található labirintust.




1.1. ábra. Példa a felső korlát számításhoz
Itt $K = \binom{24}{4} = 10624$.

# 2. fejezet

# A kereső algoritmus

„Van közte jó út, rossz út, de csak egyvalaki tudja megmondani, milyen az a bizonyos út: aki járja.

Peter Marshall

## 2.1. Kritériumok

Az algoritmus kiválasztásakor az alábbi értékelési szempontokat vehetjük figyelembe:

- **Teljesség:** A rendszernek minden esetben meg kell találnia a megoldást.
- **Optimalitás:** Több megoldás létézése esetén az optimálisat kell megtalálni.

Ezen feltételek tükrében kézenfekvő választásnak tűnik az A* algoritmus. Azonban e kereső *memóriabonyolultsága* a feltárt élek és csúcsok számával arányos. A mi problémánk jellegzetes tulajdonsága azonban épp a hatalmas keresési tér. Így az A* algoritmus egy fejlettebb változatát kell választanunk, mely szintén teljes és optimális, azonban kevésbé memóriaigényes.

## 2.2. IDA*

Az **IDA*** (**I**terative **D**eepening **A*** algorithm - iteratívan mélyülő A* algoritmus) az *optimális* és a *visszalépéses* keresés tulajdonságainak ötvözésével adódik. Adott $c$ költséghatárig járjuk be a reprezentációs gráfot visszalépéses kereséssel, eközben meg kell határoznunk egy $c'$ költséghatárt is arra az esetre, ha az eddig feltárt csúcsok között

nincs *terminális*. Ebben az esetben meg kell ismételnünk a keresést, ahol $c$ új értéke $c'$ lesz. Egészen addig folytatjuk az eljárást, amíg $c'$ végtelen nem lesz.
Lássuk az algoritmus pszeudokódját:

```
Program IDA*
1.  NYÍLT ← {s}
2.  c ← f(s)
3.  c′ = ∞
4.  while ¬üres(NYÍLT) loop
5.       n ← pop(NYÍLT)
6.       if cel(n) then
7.          return n
8.       endif
9.       if f(n) ≤ c then
10.          NYÍLT ← NYÍLT ∪ Γ(n)
11.      else
12.         c′ ← min(c′,f(n))
13.      endif
14.      if üres(NYÍLT) then
15.         if c′ = ∞
16.           return nincs megoldas
17.         endif
18.         c ← c′
19.         c′ ← ∞
20.         NYÍLT ← {s}
21.      endif
22.  endloop
23.  return nincs megoldas
end
```

A NYÍLT halmaz egy verem adatszerkezet, így a *pop()* művelet a szokásos módon működik, valamint $\Gamma$ az $n$ csomópont kiterjesztésével nyert csomópontok halmaza, $s$ a kezdőcsúcs. Az $f$ függvénynek monoton növekvő az $s$ kezdőcsúcsból induló utak mentén. Ahhoz, hogy az IDA* elnevezés jogos legyen, az $f$ függvénynek az alábbi alakúnak kell lennie: $f(n) = g(n) + h(n)$. Itt $g(n)$ egy, az $n$ csúcsba vezető út költsége, $h(n)$ pedig egy *megengedő* heurisztika, azaz bármely állapot esetén a heurisztika függvény alulról becsli az optimális megoldás hosszát.

### 2.2.1. Az IDA* implementációja

Lássuk a Java forráskódot!

```
package sokobansolver.keresok;

import java.util.LinkedList;
import sokobansolver.sokoban.Allapot;
import sokobansolver.sokoban.Heurisztika;

public class IDACsillag {

    int koltsegKorlat;
    int kovetkezokoltsegKorlat;
    int kiterjesztve;
    long keresesIdeje;
    Heurisztika heurisztika;

    public int getKiterjesztve() {
        return kiterjesztve;
    }

    public long getKeresesIdeje() {
        return keresesIdeje;
    }

    public IDACsillag(Heurisztika heurisztika) {
        this.heurisztika = heurisztika;
    }

    public Csomopont kereses(Allapot allapot) {
        Csomopont startCsomopont = new Csomopont(allapot,
           heurisztika),
                  csomopont = null;
        LinkedList<Csomopont> nyiltak = new LinkedList<Csomopont
           >();
        nyiltak.addFirst(startCsomopont);
        koltsegKorlat = csomopont.getTeljesKoltseg();
        koltsegKorlat = Integer.MAX_VALUE;
        keresesIdeje = System.currentTimeMillis();

        while ((csomopont = nyiltak.poll()) != null) {
            if (csomopont.getAllapot().isCelAllapot()) {
                break;
            }
            Csomopont gyerek = null;
            if (csomopont.getTeljesKoltseg() <= koltsegKorlat) {
```

```
                while ((gyerek = csomopont.getKovGyerek()) != null
                    ) {

                    nyiltak.addLast(gyerek);
                    ++kiterjesztve;
                }
            } else {
                kovetkezokoltsegKorlat = Math.min(
                    kovetkezokoltsegKorlat, csomopont.
                    getTeljesKoltseg());
            }

            if (nyiltak.isEmpty()) {
                if (kovetkezokoltsegKorlat == Integer.MAX_VALUE) {
                    csomopont = null;
                    break;
                }

                koltsegKorlat = kovetkezokoltsegKorlat;
                kovetkezokoltsegKorlat = Integer.MAX_VALUE;
                nyiltak = new LinkedList<Csomopont>();
                nyiltak.addFirst(startCsomopont);
            }
        }
        keresesIdeje = System.currentTimeMillis() - keresesIdeje;
        return csomopont;
    }
}
```

A while-ciklusban történik az aktuális csomópont kiterjesztése.

## 2.3. A heurisztika

A keresőnk hatékonyságát döntően befolyásolja a heurisztika minősége. Egy becslésre van szükségünk, amely megadja a hátralévő lépések számát. Határozzuk meg a ládák és egy hozzájuk rendelt célmező távolságát! Kézenfekvőnek tűnhet az ún. *Manhattan-távolság*:

$$M((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|,$$

ahol $(x_1, y_1)$ és $(x_2, y_2)$ a két pozíció.

Ez a függvény azonban nem érzékeny a labirintus sajátosságaira, nem veszi figyelembe az üres- és falmezőket. Sőt a fenti képlet alapján az is látható, hogy ez egy statikus információ, azaz nem függ az aktuális állapottól sem. Így fenn áll a veszélye annak, hogy a fenti metrika „félreinformál”.

### 2.3.1. A relaxált feladat

Az 1.4.2. pontban megfogalmazott előfeltételből tetszőleges számú részfeltételt elhagyva az eredeti problémához tartozó relaxált feladato(ka)t kapjuk. Ezen feladatok optimális megoldásainak költségei rendszerint pontosabban becsülhetőek, s ezek alsó korlátjai az eredeti megoldás optimális költségének, azaz megengedő heurisztikaként használhatóak. Definiáljunk egy olyan $\tau$ távolság függvényt, amely úgy határozza meg $(x_1, y_1)$ és $(x_2, y_2)$ pontok közötti optimális út költségét, hogy közben nem veszi figyelembe az útbaeső ládákat, azokat üresmezőnek tekinti. E függvénynek is meg van az az előnye, hogy a keresés előtt kiszámítható, azonban a fenti Manhattan-távolsággal ellentétben figyelembe veszi a labirintus sajátosságait. Kétféleképpen adhatjuk meg a függvényt:

- Egyszerűen $(x_1, y_1)$ és $(x_2, y_2)$ között vesszük az optimális távolságot. Ez esetben az eredeti feladatban a tolások számát adja meg a függvény. Hátránya, hogy előfordulhat, hogy elvileg is lehetetlen az egyik pontból eljutni a másikba, nemcsak az útban lévő ládák miatt, hanem az emberke nem fér a ládához kezdetben vagy útközben.

- Az előző pontban ismertetett problémát kiküszöbölendő, eleve úgy keressük az optimális utat, hogy figyelembe vesszük az emberkét is. Máshogy fogalmazva, a labirintusból (ideiglenesen) eltávolítjuk a többi ládát és célmezőt, s az előálló problémát egyszerűen megoldjuk úgy, hogy az emberkét a láda négyszomszédos mezőire tesszük, amennyiben ez lehetséges. Így akár az is előfordulhat a relaxált feladatban, hogy egy mezőtől egy célmezőre négy optimális út vezet. Itt már használhatjuk a Manhattan-távolságot heurisztikaként.

### 2.3.2. A hozzárendelési feladat

Tehát minden mezőhöz tartozik minden célmezőre nézve egy vagy több optimális út. Tekintsük most azokat a mezőket, amelyeken láda van. Feltesszük természetesen, hogy ugyanannyi láda van, mint célmező. Keressünk egy olyan párosítást, amelyben minden ládához pontosan egy célmező tartozik és az $\tau$ szerinti összköltség minimális!

Adott tehát $m$ láda és $m$ célmező. Az eredménymátrix szerkezete az alábbi:

$$x_{ij} = \begin{cases} 1, & \text{ha az } i\text{-edik ládát a } j\text{-edik célmezőhöz párosítjuk,} \\ 0, & \text{egyébként,} \end{cases}$$

valamint a költségmátrix:

$$c_{ij} = \tau((x_i, y_i), (x_j, y_j)), \qquad i, j = 1, 2, \ldots, m.$$

Ezekkel a jelölésekkel a *hozzárendelési feladat*ot az alábbi modell írja le.

$$\sum_{t=1}^{m} x_{it} = 1 \qquad i = 1, 2, \ldots, m$$
$$\sum_{t=1}^{m} x_{tj} = 1 \qquad j = 1, 2, \ldots, m$$
$$x_{ij} \in \{0, 1\} \qquad i, j = 1, 2, \ldots, m$$

$$\sum_{i=1}^{m} \sum_{j=1}^{m} c_{ij} x_{ij} = z \to \text{min}$$

A minimalizálási feladat megoldása az $E_{m \times m}$ egységmátrix egy permutációja. A probléma megoldása a mellékelt programban a *magyar módszer*rel történik ([9]).

### 2.3.3. Büntető függvény

A minimális összköltségű párosítás meghatározása után szükség van heurisztikánk további finomítására. A párosításba bekerült utakat a keresés során folyamatosan ”aktualizálni” kell. Definiálunk egy ún. büntető függvényt, amely két egyszerű feltétel alapján növelheti a heurisztikánkat.

Ha az út mentén két láda egymás mellett van, akkor az emberkének legalább 6 lépésre van szüksége ahhoz, hogy eltávolítsa ezt a bizonyos akadályt. Ezt a problémát a szakirodalom *linear conflict*-nek nevezi ([3]).

#### A büntető függvény első része

A fentiek alapján az alábbi módon definiálhatjuk büntető függvényünk első felét. Legyen először is a fenti torlódási jelenségnek az indikátor függvénye:

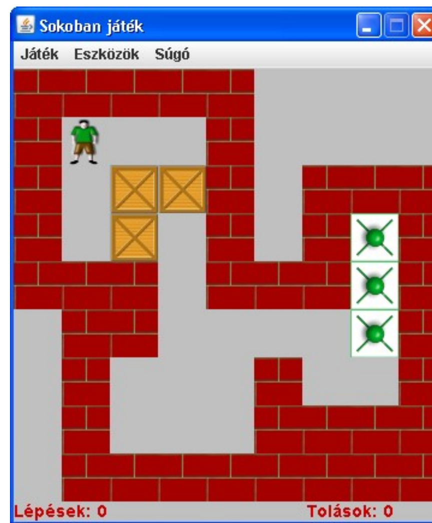
$$\mathcal{I}_L(x_1, y_1), (x_2, y_2)) = \begin{cases} 1, & \text{ha } (x_1, y_1) \text{ és } (x_2, y_2) \text{ ládák,} \\ 0, & \text{egyébként,} \end{cases}$$

a fenti alapján pedig

$$B_1(P^*) = 6 * \sum_{t=1}^{m-1} \mathcal{I}_L(P_i^*, P_{i+1}^*),$$

ahol $m$ a ládák száma és $P^* = \{(x_i, y_i)\}_{i=1}^{m}$ az optimális út.

A 3.1. ábrán látható $(3, 3)$ mezőn található láda optimális útjaihoz hozzáadódik $6$ büntetőpont, hiszen a $(4, 3)$ (vagy a $(3, 4)$ [1]) koordinátájú láda akadályozza.

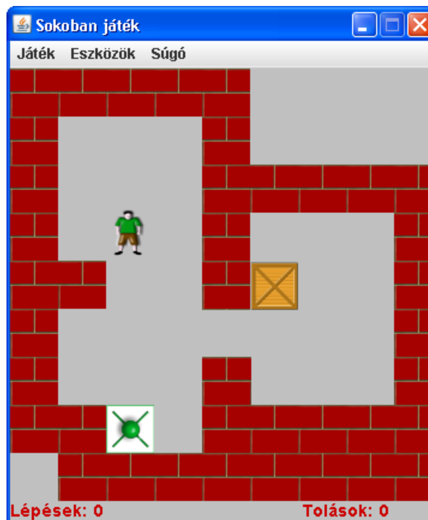



2.1. ábra. Példa a büntető függvény első részéhez

[1] Mindkét mező lehet optmiális úton.

### A büntető függvény második része

A másik feltétel egy egyszerű megfigyelésből adódik. Egy ún. *szoba* „bejárati falai” mentén elhelyezkedő ládák elmozdítása a fallal ellentétes irányba igen nehézkes. Ekkor egy lehetséges út adódik, s az nem változik a keresés során. Az alábbi ábrán látható labirintusban a ládát balra elmozdítani csak úgy tudjuk, ha azt fentről megközelítve átvisszük a másik szobába és ott megkerülve visszatoljuk a kívánt helyre. Ezt a felfedezést használva megkímélhetjük a keresőt a felesleges kiterjesztésektől s heurisztikánkat is finomíthatjuk ([3]).




2.2. ábra. Példa a büntető függvény második részéhez

# 3. fejezet

# Holtpontok

*„Meg kell tanulnod eldobni a felesleges dolgokat.”*

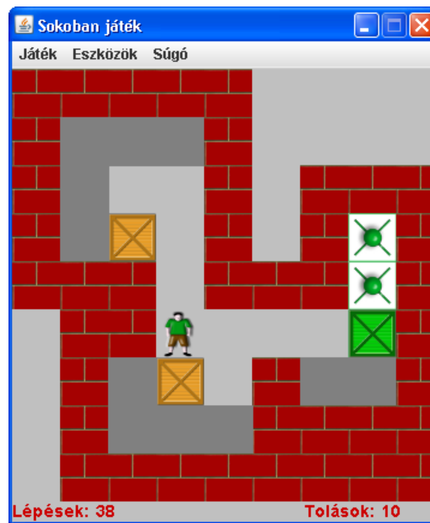
OSHO

Az eddigiekben igyekeztünk minél hatékonyabb algoritmust és hozzá tartozó heurisztikát találni. A továbbiakban más oldalról közelítjük a problémát. Arra fogunk törekedni, hogy a felesleges állapotokat elkerüljük, valamint az ún. *holtpont*okat eltávolítsuk az adatbázisból. Ezzel időt és tárhelyet takarítunk meg.
Tekintsünk át néhány típusát a holtpontoknak!

## 3.1. Halott mezők

Azokat a mezőket a labirintusban, amelyekről nem lehet elmozdítani a ládát *halott mezők*nek hívjuk. Ezek tipikusan sarokmezők, esetenként fal mellettiek. a 3.1. ábrán látható labiritusban a sötétszürke mezők ilyenek. Ide szükségtelen ládát tolni, hiszen innen nem érhető el célállapot. A heurisztika tárgyalásakor említettük, hogy előre meghatározzuk minden egyes mezőre a célmezőkhöz vezető optimális utakat. Így már a keresés előtt ismertek ezek a távolságok[1]. Egy hatékony kereső motornak az ilyen mezőre vezető operátorokat figyelmen kívűl kell hagynia, hiszen az ilyen állapotokból nyert újabb állapotok is holtpontok. Az előbbi műveletet *vágás*nak nevezzük.

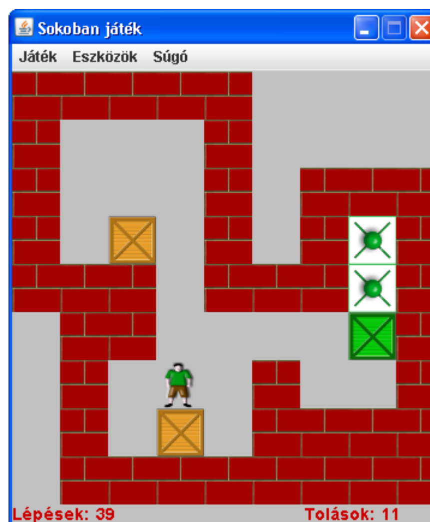
[1] Így felesleges erre írnunk egy külön eljárást, hiszen az előfeldolgozó szakaszban nyilvánvalóvá válik, melyik mezőről nem érhető el egy célmező sem. Ezek lesznek a halott mezők.




3.1. ábra. Példa a ”halott mezők” problémához

## 3.2. Párosítási probléma

Az előbbi probléma általánosítása a következő: A hozzárendelési feladat létezésének alapfeltétele, hogy legalább egy lehetséges párosítás legyen. Ha ez nem teljesül az egész probléma megoldhatatlan, a keresést felesleges folytatni. Egy példa a 3.2. ábrán látható.



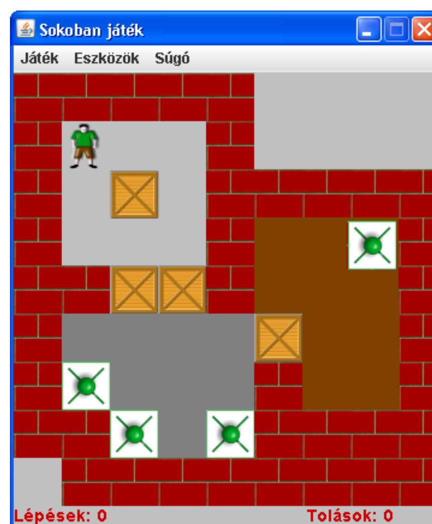

3.2. ábra. Példa a párosítási problémára

## 3.3. Korall probléma

A 3.3. ábrán látható labirintusban sötétszürkével, ill. barnával jelöltük az ún. korall mezőket. Egy haladó szintű Solvernek képesnek kell lennie az ilyen objektumok felismerésére. Ezek tulajdonképpen ládákkal és/vagy falakkal határolt egybefüggő „nem-fal” mezők. Egy ilyen korall létezése esetén a keresőnek arra kell törekednie, hogy megszüntesse ezek létezését, így amikor a kiterjesztés műveletét végzi egy állapoton, csak azokkal az operátorokkal kell dolgoznia, amelyek egyébként is alkalmazhatóak, és egy ládát egy ilyen korall mezőre kell tolnia. Ez csökkentheti az előállított állapotok számát, azonban veszélyeztetheti a megoldás optimalitását.
Kétféle definíció képzelhető el:

- Az egyikben ládákat tartalmazó mezőket nem engedünk meg. A szakirodalom ezt a típust *I-coral*nak nevezi. A 3.3. ábrán látható az e típus szerinti jelölés.

- A másik típus neve *PI-coral*. Ennek használata hatékonyabb, ám nehezebben implementálható. Figyelmet kell arra is fordítanunk, hogy a ládák célmezőkön vannak-e, hiszen akkor nem feltétlenül szükséges a fentebb leírtak betartása. E-szerint a definíció szerint a 3.3. ábrán egy színnel kellett volna jelölni a korall mezőket, amikbe most beleértendő a láda is.



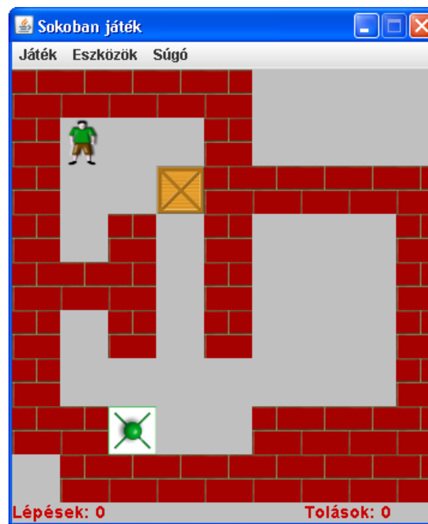

3.3. ábra. Példa a korall problémára

## 3.4. Alagutak

Gyakoriak a labirintusban olyan összefüggő mező-csoportok, melyek tulajdonságait kihasználva hatékonyabb keresőt írhatunk. Egy ilyen speciális mező-csoport az *alagút*. Ennek jellemző tulajdonsága, hogy olyan mezők alkotják, melyek kevésbé jelentősek a kereséskor. Gyakran előfordul, hogy ezeket egyszerűen ignorálhatjuk. Vessünk egy pillantást a 3.4. ábrára!
A $(6, 2)$ és $(6, 3)$ mezők egy alagutat alkotnak. Látható, hogy ebben a példában az alagút mezői a keresés szempontjából egyáltalán nem fontosak. A labirintus így ekvivalens azzal a labirintussal, melyben a fentebb említett mezők falak.

Egy másik típusú alagutat alkotnak a $(4, 4)$, $(5, 4)$ és $(6, 4)$ mezők. Ez a típus más okból fontos, mint az előbbi bekezdésben említett. Ezek a mezők fontosak ugyan a keresés szempontjából, de jelentőségük kevesebb, mint a többié. A 3.4. ábrán található ládát szükségtelen az alagút mezőire tolni. Ebben az esetben egy olyan speciális operátort kell definiálnunk, mely a ládát a $(7, 4)$-re tolja.

Az alagutak detektálásakor körültekintően kell eljárnunk. Előfordulhat, hogy egyes mezők célmezők, ilyenkor természetesen meg kell engednünk, hogy láda kerüljön ide. A legtöbb esetben az alagutat körülvevő mezőket is meg kell vizsgálnunk.




3.4. ábra. Példa az alagút problémára

# Összegzés

*„A számítógép döntően abban különbözik az embertől, hogy csak válaszolni tud, kérdezni nem.”*

Lukács György

## Zárógondolatok

Összességében elmondhatjuk, hogy ügyes reprezentációval és hatékony keresővel jelentős eredményeket érhetünk el. A hatalmas keresési tér rákényszerít minket arra, hogy minél több ésszerű javítást eszközöljünk, s minél több tárgyspecikfikus tudást használjunk fel. Azonban a nagyobb labirintusok még így is kívül vannak a megoldhatóság körén. Tehát közel sem megoldott még a Sokoban probléma. Sokáig lesz még szakdolgozatok és PhD-tézisek témája.

Az újabb kutatások rámutattak, hogy az absztrakciós szint növelésével, a probléma egy merőben új megközítésével hatékonyabb kereső rendszer alkotható. Valójában ez egy új megoldáskeresési stratégia, miszerint részproblémákra kell bontanunk az eredeti, bonyolult problémát, s azokat megoldani. A részproblémák a labirintus dekompozíciójából nyerhetőek. Két csoport különböztethető meg: a *szobák* és az *alagutak*. E téma túlmutat a dolgozat keretein, de az mindenképp látszik, hogy a jövő kutatásai e gondolatok szellemében folynak majd.

## A mellékelt program

A *SokobanSolver* nevű szoftver alkalmas játékra és megoldáskeresésre is.
A CD-mellékleten megtalálható a program forrása és egy futtatható *jar* fájl is. A dolgozatban bemutatott összes elemet tartalmazza, s többet is annál, hiszen képes a labirintust lebontani szobákra és alagutakra. Ebből azonban csak az alagutak detektálását használja a kereső, az előző fejezetben ismertetett „alagút probléma” kezelésére.

A program képes egy egyszerű struktúrájú bejegyzésekből álló szöveges állományból labirintusokat gyűjteni. Egy példán keresztül vizsgáljuk meg egy bejegyzés szerkezetét:

```
Parameters(1) : 9 9 0 2

#####
# @ #
# $ #####
#   #   #
##$$#   #
#    $  #
#.   #  #
##...####
#######
```

A zárójelek között a labirintus sorszáma, a kettőspont után az oszlopok és sorok száma szerepel. A másik kettő paraméter számunkra most nem fontos, hiszen az a labirintus felépítésére vonatkozik. Más paraméterekkel olyan labirintus adható meg, amelynél egy „fél” lépés is engedélyezett.

## Köszönetnyilvánítás

Köszönetet szeretnék mondani Dr. Halász Gábor egyetemi docensnek, hogy értékes tanácsaival segítette munkámat, Vári Szalai Zsuzsannának, aki részt vállalt az idegen nyelvű szakirodalom feldolgozásában, valamint Kiss Gábornak, aki a grafikus elemek elkészítésében segített.

# Irodalomjegyzék


[1] YASC (Yet Another Sokoban Clone)
Lásd: http://sokoban-yasc.en.softonic.com/

[2] Joseph C. Culberson, Sokoban is PSPACE-complete. Technical Report TR 97-02, Dept. of Computing Science, University of Alberta, 1997.
Lásd: http://web.cs.ualberta.ca/~joe/Preprints/Sokoban

[3] Andreas Junghanns, Jonathan Schaeffer, Sokoban: A Challenging Single-Agent Search Problem, University of Alberta

[4] Andreas Junghanns, Jonathan Schaeffer, Sokoban: Improving the search with relevance cuts, University of Alberta

[5] Dr. Várterész Magdolna, Mesterséges Intelligencia 1 előadások (fóliák)
Lásd:http://www.inf.unideb.hu/~varteres/mi1folia/

[6] Mesterséges Intelligencia (Szerk. Futó Iván), Aula Kiadó, 1999

[7] Stuart Russell, Peter Norvig. Mesterséges intelligencia modern megközelítésben. (2. kiadás), Panem Kiadó Kft., 2005.

[8] Jeszenszky Péter segédanyagai
Lásd: http://www.inf.unideb.hu/~jeszy

[9] Imreh Balázs, Operációkutatás, elektronikus jegyzet

[10] Wettl Ferenc, Mayer Gyula, Sudár Csaba. LaTeX kezdőknek és haladóknak. Panem, 1998.

# Ábrák jegyzéke


1. Jelölési konvenciók . . . . . . . . 6

1.1. Példa a felső korlát számításhoz . . . . . . . . 15

2.1. Példa a büntető függvény első részéhez . . . . . . . . 23
2.2. Példa a büntető függvény második részéhez . . . . . . . . 24

3.1. Példa a ”halott mezők” problémához . . . . . . . . 26
3.2. Példa a párosítási problémára . . . . . . . . 26
3.3. Példa a korall problémára . . . . . . . . 27
3.4. Példa az alagút problémára . . . . . . . . 28