

Debreceni Egyetem

Informatika Kar

**Heurisztikus megoldáskereső algoritmus
a Sokoban játékra**

Témavezető:

Dr. Halász Gábor egyetemi docens

Információ Technológia Tanszék

Készítette:

Szabados Szabolcs

Programozó-matematikus szak

Debrecen

2009

Tartalomjegyzék

1. Bevezetés	3
2. A C# nyelv	6
3. A program telepítése és használata	8
4. Heurisztikus megoldáskereső algoritmusok	10
5. Állapottér reprezentáció	12
6. A program	17
6.1 1. Generáció	17
6.2 2. Generáció	19
6.3 3. Generáció	23
7. Összegzés	36
8. Felhasznált irodalom	38

1. Bevezetés

Téma

A Sokoban egy olyan fejtörő, ahol a játékosnak labirintusban kell dobozokat tologatnia a helyükre. Egyszerre csak egy dobozt lehet mozgatni, és csak tolni lehet, húzni nem.

Nagyon sok játék tekinthető Sokobannak, ha azt vesszük alapul, hogy a Sokobanban egy irányítható karakter ládákat tologat egy labirintusban. Az alábbi szempontok mentén születtek új verziók:

- **Alternatív játéktér:** A játék alapverziója négyzetekből álló játéktérrel („csempézést”) használ. Számos későbbi változat viszont azonos szabályok mellett a labirintust más idomokkal fedi le. A Hexoban hatszögeket, a Trioban egyenlő oldalú háromszögeket használ.
- **Több karakter:** Az Interlock és a Multiban nevű verziókban a játékos több karaktert irányíthat.
- **Alternatív célok:** Sok klónban van nehezítés: például a Block-o-Maniában a dobozok különböző színűek, és az a cél, hogy minden dobozt ugyanolyan színű helyre toljunk. A Sokomind Plus is hasonló, csak itt már minden egyes doboz különböző színű. Az Interblockban és a Sokolorban is különböző színűek a dobozok, de itt az a cél, hogy a hasonló színűek határosak legyenek. A Cyber Boxban minden szinten van egy kilépő négyzet, és a feladat az, hogy ezt elérjük.

Plusz játékelemek: A Sokonex, Xsok, Cyberboy és a Block-o-Mania új elemeket adott hozzá a játékhoz, például teleportokat, üregeket, mozgó blokkokat vagy egyirányú utakat.

Tudományos megközelítés:

A Sokoban a bonyolultságelmélet tárgykörébe tartozik. A játék megoldása bizonyítottan NP-nehézségű, mivel egy olyan mozgásszervező probléma része, ahol egyszerre több dobozt is lehet tolni vagy húzni. Érdekes továbbá a mesterséges intelligencia kutatóinak is, mert a Sokoban hasonlítható egy robottervezéshez is, ami dobozokat mozgat egy raktárban. További kutatások megerősítették, hogy a Sokoban PSPACE-teljes, azaz polinom időben megoldható (hivatkozással Joseph C. Culberson dolgozatára, melynek elérhetősége a felhasznált irodalmak között megtalálható).

A Sokoban nem csak a pillanatnyilag lehetséges lépések magas száma (branching factor) miatt nehéz (ami a sakkhöz hasonlítható, de még mindig sokkal kisebb, mint a go), hanem számos pályánál a keresőfák mélysége miatt is; sok pálya több mint ezer „tolást” követel meg. Tapasztalt emberi játékosok leginkább a heurisztikájukban bíznak; ők általában gyorsan eldobják a haszontalan vagy redundáns lehetséges utakat a játékból és felismerik azokat a mintákat és részcélokat, amelyek drasztikusan lecsökkentik a keresések számát.

Néhány Sokoban-probléma megoldható automatikusan a single-agent kereső algoritmussal, mint például az IDA, felfejlesztve sok technikával, amik a doménspecifikus tudást használják. Ezt használja a Rolling Stone, egy Sokoban megoldó program is, amit az Albertai Egyetem GAMES (Game-playing, Analytical methods, Minimax search and Empirical Studies) csoportja fejlesztett ki. A komplexebb pályák még így is kívül esnek a megoldhatóság körén a legjobb automatikus megoldószoftverek számára is.

Célok

A célom egy olyan heurisztika illetve megoldáskereső algoritmus megtalálása, mellyel a programom a lehető legrövidebb időn belül képes legyen az optimális megoldás megtalálására, azaz a legkevesebb lépésből tudja helyükre juttatni az összes ládát. Ehhez segítséget nyújtanak leírások, szakkönyvek és persze maga a játék, melynek számos internetes változata létezik, melyeket felhasználtam a dolgozat megírásához.

A dolgozat felépítése

Négy különböző logikai részre bontom a dolgozatot:

1. rész:

Itt található C# nyelv rövid ismertetése, valamint a megoldáskereső program telepítésébe, használatába nyújtok rövid betekintést.

2. rész:

Ebben a részben rövid áttekintést szeretnék nyújtani a heurisztikus megoldáskereső algoritmusokról, heurisztikus függvényekről és az általam alkalmazott heurisztikáról.

3. rész:

Ez a rész tartalmazza a probléma állapottér reprezentációját és állapottér gráfját.

4. rész:

Magát a megoldás kereső algoritmust és heurisztikát ismerteti kódrészletekkel és leírásokkal. Továbbá itt található az azon kódrészletek és okfejtések is, melyek a programom evolúciója során létrejöttek.

Terveim

A dolgozatom programozási részét C# nyelven készítettem, mert már több mint három éve dolgozom ezzel a nyelvvel egy multinacionális cégnél, így ezt a nyelvet ismerem leginkább. Valamint érdekesnek találtam a probléma OO megközelítését. Tanulmányaim elvégzése után szeretném más iparágakban is megmérettetni magam, valamint nyelvtudásomat fejleszteni.

2. A C# nyelv

A C# a Microsoft által a .NET keretrendszer részeként fejlesztett objektumorientált programozási nyelv. A nyelv alapjául a C++ és a Java szolgált. A C# -ot úgy tervezték, hogy meglegyen az egyensúly a fejlesztő nyelvi szabadsága és a gyors alkalmazásfejlesztés lehetősége között. Bár a Mono project szinte tökéletes C# fordítót állított elő (amely a nyelv legújabb 3.5 verzióját is támogatja), a C# Windows operációs rendszereken kívüli használata kevésbé elterjedt, mivel az osztálykönyvtárakat szolgáltató .NET Framework portolása más rendszerek alá még gyermekcipőben jár. Ugyanakkor a Mono az utóbbi években rohamos fejlődésnek indult, amit elősegített a Microsoft és a Novell (a Mono "anyacége") együttműködése. Microsoft Silverlight technológiájának Linux illetve Macintosh rendszerekkel kompatibilis változatának – a Moonlight -nak - készítésekor.

Nyelvi lehetőségek

A C# az a programozási nyelv, amely a legközvetlenebb módon tükrözi az alatta működő minden .NET programot futtató .NET keretrendszert, valamint erősen függ is attól: nincsen nem menedzsel, natív módban futó C# program. A primitív adattípusai objektumok, a .NET típusok megfelelői. Szemétgyűjtést használ, valamint az absztrakcióinak többsége (osztályok, interfészek, delegáltak, kivételek...) a .NET futtatórendszert használja közvetlen módon.

A C vagy C++ nyelvhez hasonlítva a C# több korlátozást és továbbfejlesztést is tartalmaz. A lehetőségei közül néhány:

- A mutatók és a nem ellenőrzött aritmetika csak egy speciális, nem biztonságos módban (*unsafe mode*) használható. A legtöbb objektum hozzáférés csak biztonságos hivatkozásokon keresztül tehető meg, és az aritmetikai műveletek debug módban túlsordulás szempontjából ellenőrzöttek.
- Az objektumok nem szabadíthatók fel közvetlen módon, ehelyett a szemétgyűjtő szabadítja fel őket, mikor már nincs rájuk hivatkozás. Ez a módszer kizárja a nem létező objektumokra való hivatkozás lehetőségét.
- A destruktorkok nem elérhetőek. A legközelebbi megfelelőjük az *IDisposable* interfész, ami a *using* blokkal együtt kikényszerítheti az azonnali felszabadítást. A finalizerek szintén rendelkezésre állnak, de nem váltanak ki azonnali felszabadítást.

- A nyelv csak egyszeres öröklődést támogat, de egy osztály több interfészt is megvalósíthat.
- A C# sokkal típusosabb, mint a C++. Az egyetlen implicit konverzió a biztonságos konverzió úgy, mint az egészek tágabb intervallumba konvertálása vagy a leszármazott osztályok alaposztályba konvertálása. Nincs implicit konverzió az egészek és a logikai típus (boolean) között, a felsorolás tagok és az egészek között. Nincsenek *void* mutatók (bár az *Object* osztályra mutató mutatók hasonlóak), valamint bármely, a felhasználó által definiált implicit konverziót explicit módon meg kell jelölni.
- A tömbdeklaráció szintaxisa eltérő (*int[] a = new int[5]* az *int a[5]* helyett).
- A felsorolás adattagjai a saját névterükben helyezkednek el.
- A C# 1.x nem rendelkezik *template*-ekkel, de a C# 2.0 már rendelkezik *generikus* típusokkal.
- Tulajdonságok (*Properties*) használhatók, amelyek úgy tesznek lehetővé kódfuttatást mezők beállításakor és olvasásakor, mintha mezőhozzáférés történne.
- Teljes reflexió elérhető.

Kódkönyvtárak

A legtöbb programozási nyelvtől eltérően a C# megvalósítások nem rendelkeznek önálló, eltérő osztály- vagy függvénykönyvtárakkal. Ehelyett a C# szorosan kötődik a .NET keretrendszerhez, amittől a C# kapja a futtató osztályait és függvényeit. A .NET keretrendszer osztálykönyvtárat tartalmaz, ami a .NET nyelvekből felhasználható egyszerű feladatok (adatrepresentáció és szövegmanipuláció) végrehajtásától kezdve a bonyolult (dinamikus ASP.NET weblapok generálása, XML feldolgozás és reflexió) feladatokig. A kód névterekbe van rendezve, mely a hasonló funkciót ellátó osztályokat fogja össze. Például System.Drawing a grafikai, System.Collections az adatstruktúra és System.Windows.Forms a Windows Forms funkciókat fogja össze.

További rendezési szint a szerelvény (assembly). Egy szerelvény állhat egy fájlból, vagy több összelinkelt fájlból (az al.exe segítségével), ami több névteret és objektumot tartalmazhat. A különböző feladatokhoz szükséges osztályokat szerelvények (például System.Drawing.dll, System.Windows.Forms.dll) hivatkozásával vagy a központi könyvtár (mscorlib.dll a Microsoft megvalósításában) használatával érhetik el a programok.

3. A program telepítése és használata

Támogatott operációs rendszerek:

Windows 2000 Service Pack 4; Windows Server 2003 Service Pack 1; Windows XP Service Pack 2

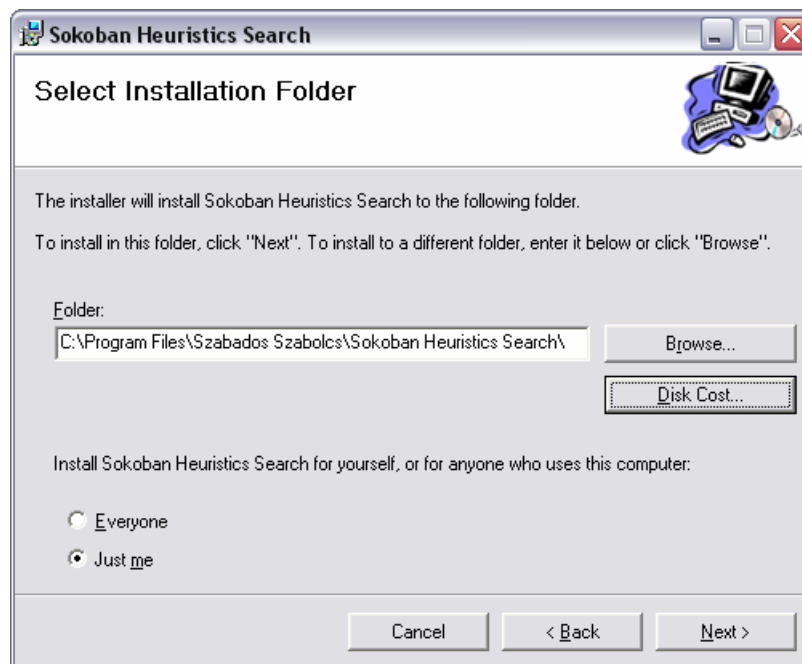
Rendszerkövetelmény:

Pentium III 600 MHz vagy gyorsabb processzor (1 GHz vagy gyorsabb ajánlott.)

Minimum 732 Kb szabad merevlemez kapacitás. Ha a .NET framework is telepítésre szorul, akkor plusz 280 MB – ra van szükség.

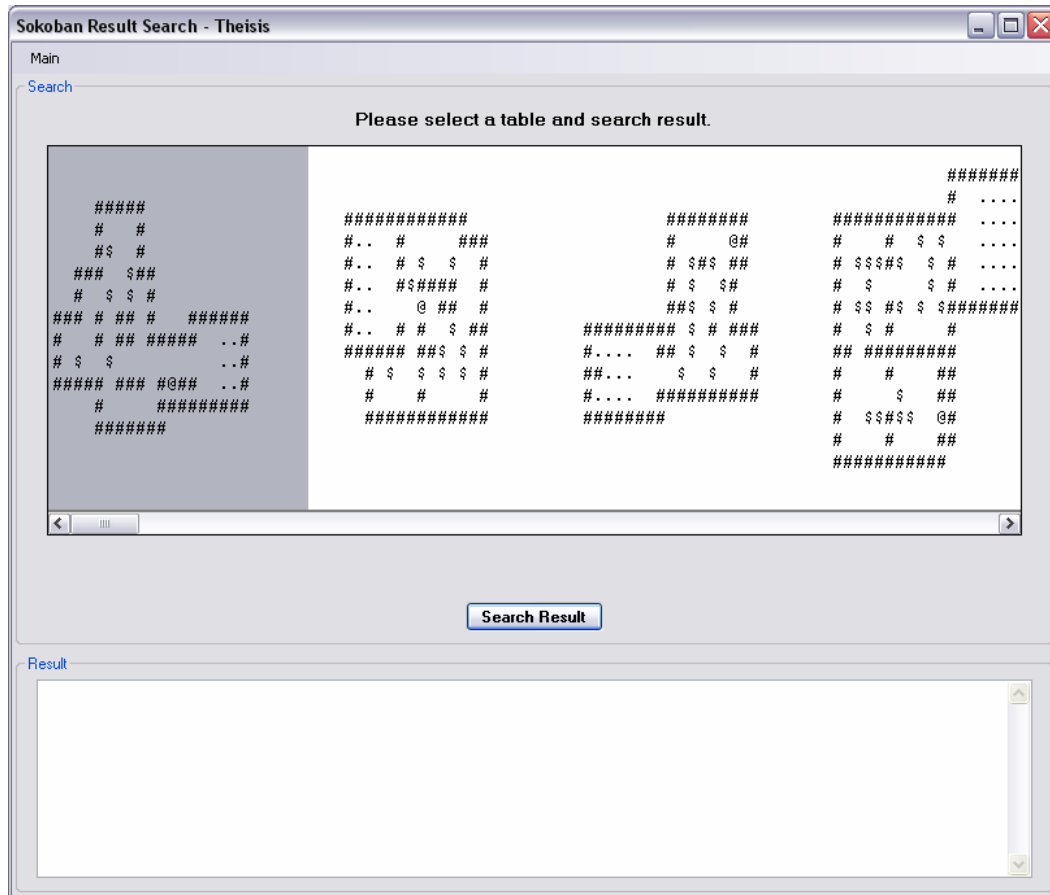
Telepítés:

A telepítő CD-t behelyezve a telepítés azonnal elindul, ha ez nem történne meg nyissuk meg a CD tartalmát, majd kattintsunk kétszer a setup.exe állományra. Ezután mindössze annyi dolgunk van, hogy a telepítő által feltett kérdésekre értelemszerűen válaszoljunk és hajtsuk végre a telepítést. A telepítő, ha az nem áll rendelkezésre, akkor telepíti a .Net framework keretrendszert is, amely telepítése külön is kezdeményezhető a CD-n található dotnetfx.exe állomány futtatásával.



A program használata:

A következő képernyőn válasszuk ki a pályát, melyre a megoldáskeresést le szeretnénk futtatni, majd kattintsunk a 'Search Result' feliratú gombra és ezzel kezdetét veszi a megoldáskeresés.



Amennyiben a program megtalálja az optimális megoldást, a fenti képernyő alsó részén található szövegdobozban kiírja a játékos lépéseinek irányát, illetve a megoldás megtalálására fordított időt is, valamint a játékos lépéseinek számát.

4. Heurisztikus megoldáskereső algoritmusok

Ezen algoritmusok közös jellemzője, hogy probléma specifikus tudást alkalmaznak a hatékonyabb megoldáskereső érdekében. Ezen algoritmusok kulcseleme a heurisztikus függvény, mely becslést ad az adott n csomóponttól a célig vezető út költségére.

Ilyen algoritmusok például:

- A mohó legjobbat először keresés (greedy best-first search), ami azt a csomópontot értékeli ki legelőször, melyet legközelebbinek ítél a célállapothoz.
- A* keresés, mely a csomópontokat úgy értékeli ki, hogy összegzi az adott csomópontig megtett út költségét a célhoz vezető út becsült költségével.

Én ez utóbbi A* keresést alkalmaztam a probléma megoldás érdekében. Az ilyen (heurisztikus) megoldáskereső algoritmusok természetéből fakadóan a sikerem feltétele a megfelelő heurisztikus függvény megtalálása volt. Több, például a nyolcas kirakó játéknál már sikerrel alkalmazott heurisztikus függvény között válogathattam.

Például:

- A rossz helyen lévő, ez esetben, ládák száma.
- Manhattan-távolság, ez esetben, a ládák a saját célhelyeiktől mért Manhattan-távolságainak összege.

A választásban fontos szerepet játszott az effektív elágazási tényező (effective branching factor), ami a heurisztikus függvény minősítésének egyik lehetséges módja. Amennyiben az A* algoritmus által kifejtet összes csomópont száma egy adott problémára N , és a mélysége d , akkor a b^* annak a d mélységű kiegyensúlyozott fának az elágazási tényezőjével egyezik meg, amely $N + 1$ csomópontot tartalmazna.

Ebből adódóan:

$$N + 1 = 1 + b^* + (b^*)^2 + \dots + (b^*)^d$$

Például ha az A* algoritmus egy 5 mélységben fekvő megoldást 52 csomópont kifejtésével talál meg, akkor az effektív elágazási tényező 1,92. Ez a tényező a probléma esetek függvényében változhat, de a megfelelően nehéz problémák esetében állandó.

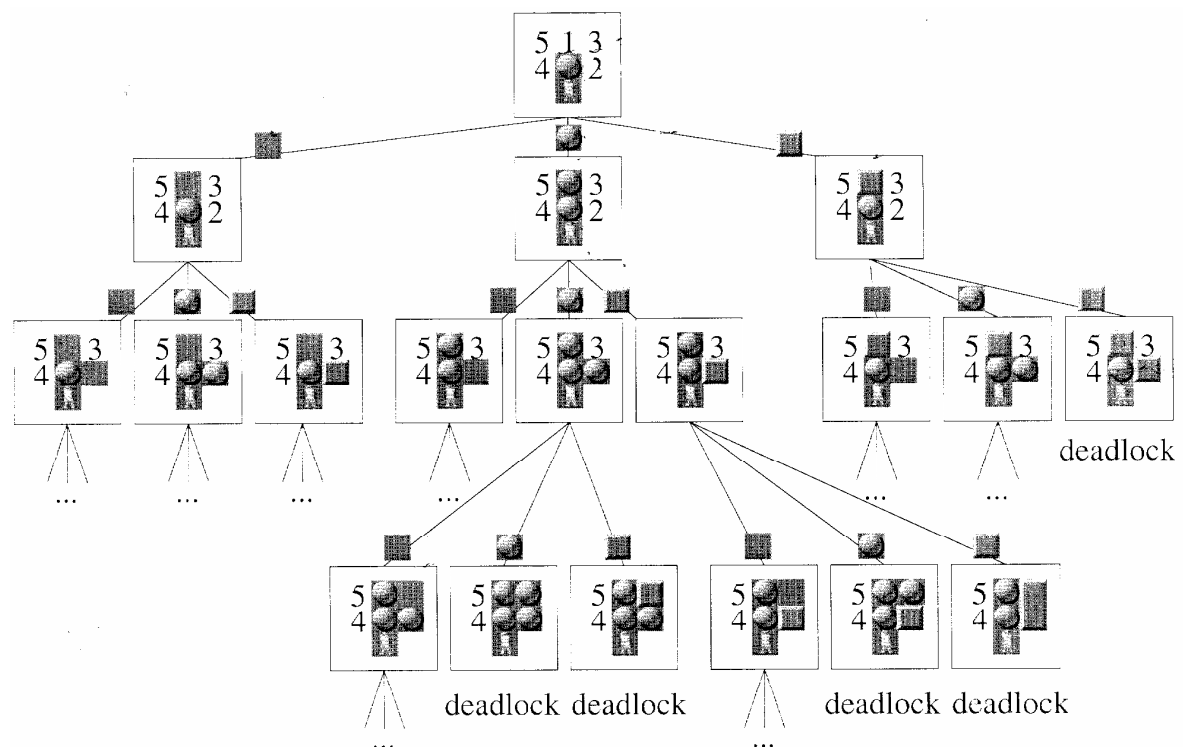
Egy megfelelően megválasztott heurisztikus függvény elágazási tényezője egy 1 körüli érték.

A kérdés az volt, hogy melyik heurisztikus függvényt válasszam.

A választásom az előzőekben felsoroltakon kívüli, egy harmadik, általam jobbnak ítélt heurisztikus függvényre esett, melynek lényege, hogy az előzőleg megtett lépések számát kombinálja össze a ládák a saját célhelyeiktől mért tényleges távolságának összegével.

Itt kell még megemlítenem az ismételt állapotok elkerülését, amire szintén figyelmet kellett, hogy fordítsak. Ennek lényege, hogy az egyszer, egy már másik úton megtalált, kifejtett csomópontot, még egyszer ismételtelen ne fejtsem ki, mert ebben az esetben a keresési fa végtelen lehetett volna. Ezek mellett ki kell még térnem az úgynevezett deadlock állapotokra, melyek felismerése és kiküszöbölése nagyban befolyásolta a megoldáskeresés hatékonyságát. Ezek alatt a deadlockok alatt, az olyan ládapozíciókat értem, melyekben egy vagy több láda megreked és elmozdíthatatlanná válik.

Lássunk néhányat:



5. Állapottér reprezentáció

Tulajdonság:

A pálya melyik pozícióján, milyen objektum található.

Állapottér:

$$\begin{array}{ccc} (1,1) & (1,2) & (i,j) \\ p_{1,1} & p_{1,2} & p_{i,j} \end{array}$$

Ahol: $p_{i,j}$ a pálya egyes pozícióit jelöli és $i=1,\dots; j=1,\dots$

$H_{i,j}=\{0,1,2,3,4,5,6\}$ és $i=1,\dots; j=1,\dots$

Ahol:

3, 0, 1, 2, 4, 5, 6, rendre a falat, a szabad mezőt, a játékost, a ládát, a célmezőt, játékost a célmezőn, a ládát a célmezőn jelöli.

$A \subseteq H_{1,1} \times \dots \times H_{i,j}; a=(p_{1,1}, p_{1,2}, \dots, p_{i,j}) \supseteq A; i=1,\dots; j=1,\dots$

Ahol 'A' az állapotteret jelöli.

Kényszerfeltétel (a): csak egyetlen játékos létezhet és a ládák számának meg kell egyeznie a célmezők számával, egy mezőn csak egyetlen objektum lehet:

$$\begin{aligned} & \exists i \exists j \forall k \forall l (p_{i,j}=1 \wedge p_{k,l}=1 \supset k=i \wedge l=j) \wedge \forall i \forall j \forall k \forall l ((p_{i,j}=2 \vee p_{i,j}=4 \vee p_{i,j}=5 \vee p_{i,j}=6) \\ & \wedge (p_{k,l}=2 \vee p_{k,l}=4 \vee p_{k,l}=5 \vee p_{k,l}=6) \supset k \neq i \vee l \neq j) \wedge \forall i \forall j \exists k \exists l (p_{i,j}=2 \wedge p_{k,l}=4 \supset k \neq i \wedge l \neq j) \\ & \wedge \exists i \exists j \forall k \forall l (p_{i,j}=2 \wedge p_{k,l}=4 \supset k \neq i \wedge l \neq j) \text{ ahol } i, l=1,\dots; j, k=1,\dots \end{aligned}$$

$A=\{a \mid a \in (H_{1,1} \times \dots \times H_{i,j}) \wedge \text{kényszerfeltétel(a)}\}; i=1,\dots; j=1,\dots$

Kezdőállapot:

$$\forall i \forall j (p_{i,j}=3 \vee p_{i,j}=0 \vee p_{i,j}=1 \vee p_{i,j}=2 \vee p_{i,j}=4 \vee p_{i,j}=5 \vee p_{i,j}=6) \wedge \text{kényszerfeltétel(a)} \text{ és } i=1,\dots; j=1,\dots$$

Célállapot:

$$(\forall i \forall j (p_{i,j}=1 \vee p_{i,j}=0 \vee p_{i,j}=6 \vee p_{i,j}=3)) \wedge (\exists k \exists l \forall m \forall n (p_{k,l}=1 \wedge p_{m,n}=1 \supset k=m \wedge l=n))$$

$i, k, m=1, \dots; j, l, n=1, \dots$

Operátorok:

$$O = \{\text{Fel}, \text{Le}, \text{Jobbra}, \text{Balra}\}$$

Fel operátor: egy, paramétere van a honnan. A honnan pozíción található objektumot mozgatja felfelé ha az megfelel az előfeltételeknek.

$$\begin{aligned} \text{Dom (Fel)} = \{ a \mid a \in A \wedge \\ & ((p_{i,j}=1 \wedge (p_{i-1,j}=0 \vee p_{i-1,j}=4)) \\ & \vee (p_{i,j}=5 \wedge (\wedge (p_{i-1,j}=0 \vee p_{i-1,j}=4)) \\ & \vee (p_{i,j}=2 \wedge (p_{i-1,j}=0 \vee p_{i-1,j}=4) \wedge (p_{i+1,j}=1 \vee p_{i+1,j}=5)) \\ & \vee (p_{i,j}=6 \wedge (p_{i-1,j}=0 \vee p_{i-1,j}=4) \wedge (p_{i+1,j}=1 \vee p_{i+1,j}=5))) , i=1, \dots; j=1, \dots \} \end{aligned}$$

$$\text{Rng(Fel)}: a' = \text{Fel}(a) \in A \text{ ahol}$$

$$\begin{aligned} \text{ha } (p_{i,j}=1 \wedge p_{i-1,j}=0) \in a \text{ akkor } (p_{i-1,j}'=1 \wedge p_{i,j}'=0) \in a'; \\ \text{ha } (p_{i,j}=1 \wedge p_{i-1,j}=4) \in a \text{ akkor } (p_{i-1,j}'=5 \wedge p_{i,j}'=0) \in a'; \\ \text{ha } (p_{i,j}=5 \wedge p_{i-1,j}=0) \in a \text{ akkor } (p_{i-1,j}'=1 \wedge p_{i,j}'=4) \in a'; \\ \text{ha } (p_{i,j}=5 \wedge p_{i-1,j}=4) \in a \text{ akkor } (p_{i-1,j}'=5 \wedge p_{i,j}'=4) \in a'; \end{aligned}$$

$$\begin{aligned} \text{ha } (p_{i,j}=2 \wedge p_{i-1,j}=0 \wedge p_{i+1,j}=1) \in a \text{ akkor } (p_{i-1,j}'=2 \wedge p_{i,j}'=1 \wedge p_{i+1,j}'=0) \in a'; \\ \text{ha } (p_{i,j}=2 \wedge p_{i-1,j}=4 \wedge p_{i+1,j}=1) \in a \text{ akkor } (p_{i-1,j}'=6 \wedge p_{i,j}'=1 \wedge p_{i+1,j}'=0) \in a'; \\ \text{ha } (p_{i,j}=2 \wedge p_{i-1,j}=0 \wedge p_{i+1,j}=5) \in a \text{ akkor } (p_{i-1,j}'=2 \wedge p_{i,j}'=1 \wedge p_{i+1,j}'=4) \in a'; \\ \text{ha } (p_{i,j}=2 \wedge p_{i-1,j}=4 \wedge p_{i+1,j}=5) \in a \text{ akkor } (p_{i-1,j}'=6 \wedge p_{i,j}'=1 \wedge p_{i+1,j}'=4) \in a'; \\ \text{ha } (p_{i,j}=6 \wedge p_{i-1,j}=0 \wedge p_{i+1,j}=1) \in a \text{ akkor } (p_{i-1,j}'=2 \wedge p_{i,j}'=5 \wedge p_{i+1,j}'=0) \in a'; \\ \text{ha } (p_{i,j}=6 \wedge p_{i-1,j}=4 \wedge p_{i+1,j}=1) \in a \text{ akkor } (p_{i-1,j}'=6 \wedge p_{i,j}'=5 \wedge p_{i+1,j}'=0) \in a'; \\ \text{ha } (p_{i,j}=6 \wedge p_{i-1,j}=0 \wedge p_{i+1,j}=5) \in a \text{ akkor } (p_{i-1,j}'=2 \wedge p_{i,j}'=5 \wedge p_{i+1,j}'=4) \in a'; \\ \text{ha } (p_{i,j}=6 \wedge p_{i-1,j}=4 \wedge p_{i+1,j}=5) \in a \text{ akkor } (p_{i-1,j}'=6 \wedge p_{i,j}'=5 \wedge p_{i+1,j}'=4) \in a'; \end{aligned}$$

Le operátor: egy, paramétere van a honnan. A honnan pozíción található objektumot mozgatja lefelé, ha az megfelel az előfeltételeknek.

$$\begin{aligned} \text{Dom (Le)} = \{ a \mid a \in A \wedge \\ & ((p_{i,j}=1 \wedge (p_{i+1,j}=0 \vee p_{i+1,j}=4)) \\ & \vee (p_{i,j}=2 \wedge (p_{i-1,j}=1 \vee p_{i-1,j}=5) \wedge (p_{i+1,j}=0 \vee p_{i+1,j}=4)) \\ & \vee (p_{i,j}=6 \wedge (p_{i-1,j}=5 \vee p_{i-1,j}=1) \wedge (p_{i+1,j}=0 \vee p_{i+1,j}=4)) \\ & \vee (p_{i,j}=5 \wedge (p_{i+1,j}=0 \vee p_{i+1,j}=2))) , i=1, \dots; j=1, \dots \} \end{aligned}$$

Rng(Le): $a' = \text{Le}(a) \in A$ ahol

- ha $(p_{i,j}=1 \wedge p_{i+1,j}=0) \in a$ akkor $(p_{i+1,j}'=1 \wedge p_{i,j}'=0) \in a'$;
- ha $(p_{i,j}=1 \wedge p_{i+1,j}=4) \in a$ akkor $(p_{i+1,j}'=5 \wedge p_{i,j}'=0) \in a'$;
- ha $(p_{i,j}=5 \wedge p_{i+1,j}=0) \in a$ akkor $(p_{i+1,j}'=1 \wedge p_{i,j}'=4) \in a'$;
- ha $(p_{i,j}=5 \wedge p_{i+1,j}=4) \in a$ akkor $(p_{i+1,j}'=5 \wedge p_{i,j}'=4) \in a'$;
- ha $(p_{i,j}=2 \wedge p_{i+1,j}=0 \wedge p_{i-1,j}=1) \in a$ akkor $(p_{i+1,j}'=2 \wedge p_{i,j}'=1 \wedge p_{i-1,j}'=0) \in a'$;
- ha $(p_{i,j}=2 \wedge p_{i+1,j}=4 \wedge p_{i-1,j}=1) \in a$ akkor $(p_{i+1,j}'=6 \wedge p_{i,j}'=1 \wedge p_{i-1,j}'=0) \in a'$;
- ha $(p_{i,j}=2 \wedge p_{i+1,j}=0 \wedge p_{i-1,j}=5) \in a$ akkor $(p_{i+1,j}'=2 \wedge p_{i,j}'=1 \wedge p_{i-1,j}'=4) \in a'$;
- ha $(p_{i,j}=2 \wedge p_{i+1,j}=4 \wedge p_{i-1,j}=5) \in a$ akkor $(p_{i+1,j}'=6 \wedge p_{i,j}'=1 \wedge p_{i-1,j}'=4) \in a'$;
- ha $(p_{i,j}=6 \wedge p_{i+1,j}=0 \wedge p_{i-1,j}=1) \in a$ akkor $(p_{i+1,j}'=2 \wedge p_{i,j}'=5 \wedge p_{i-1,j}'=0) \in a'$;
- ha $(p_{i,j}=6 \wedge p_{i+1,j}=4 \wedge p_{i-1,j}=1) \in a$ akkor $(p_{i+1,j}'=6 \wedge p_{i,j}'=5 \wedge p_{i-1,j}'=0) \in a'$;
- ha $(p_{i,j}=6 \wedge p_{i+1,j}=0 \wedge p_{i-1,j}=5) \in a$ akkor $(p_{i+1,j}'=2 \wedge p_{i,j}'=5 \wedge p_{i-1,j}'=4) \in a'$;
- ha $(p_{i,j}=6 \wedge p_{i+1,j}=4 \wedge p_{i-1,j}=5) \in a$ akkor $(p_{i+1,j}'=6 \wedge p_{i,j}'=5 \wedge p_{i-1,j}'=4) \in a'$;

Jobbra operátor: egy, paramétere van a honnan. A honnan pozíción található objektumot mozgatja jobbra, ha az megfelel az előfeltételeknek.

Dom (Jobbra) = $\{a \mid a \in A \wedge$
 $((p_{i,j}=1 \wedge (p_{i,j+1}=0 \vee p_{i,j+1}=4))$
 $\vee (p_{i,j}=2 \wedge (p_{i,j-1}=1 \vee p_{i,j-1}=5) \wedge (p_{i,j+1}=0 \vee p_{i,j+1}=4))$
 $\vee (p_{i,j}=6 \wedge (p_{i,j-1}=5 \vee p_{i,j-1}=1) \wedge (p_{i,j+1}=0 \vee p_{i,j+1}=4))$
 $\vee (p_{i,j}=5 \wedge (p_{i,j+1}=0 \vee p_{i,j+1}=4))) , i=1, \dots; j=1, \dots \}$

Rng(Jobbra): $a' = \text{Jobbra}(a) \in A$ ahol

- ha $(p_{i,j}=1 \wedge p_{i,j+1}=0) \in a$ akkor $(p_{i,j+1}'=1 \wedge p_{i,j}'=0) \in a'$;
- ha $(p_{i,j}=1 \wedge p_{i,j+1}=4) \in a$ akkor $(p_{i,j+1}'=5 \wedge p_{i,j}'=0) \in a'$;
- ha $(p_{i,j}=5 \wedge p_{i,j+1}=0) \in a$ akkor $(p_{i,j+1}'=1 \wedge p_{i,j}'=4) \in a'$;
- ha $(p_{i,j}=5 \wedge p_{i,j+1}=4) \in a$ akkor $(p_{i,j+1}'=5 \wedge p_{i,j}'=4) \in a'$;
- ha $(p_{i,j}=2 \wedge p_{i,j+1}=0 \wedge p_{i,j-1}=1) \in a$ akkor $(p_{i,j+1}'=2 \wedge p_{i,j}'=1 \wedge p_{i,j-1}'=0) \in a'$;
- ha $(p_{i,j}=2 \wedge p_{i,j+1}=4 \wedge p_{i,j-1}=1) \in a$ akkor $(p_{i,j+1}'=6 \wedge p_{i,j}'=1 \wedge p_{i,j-1}'=0) \in a'$;
- ha $(p_{i,j}=2 \wedge p_{i,j+1}=0 \wedge p_{i,j-1}=5) \in a$ akkor $(p_{i,j+1}'=2 \wedge p_{i,j}'=1 \wedge p_{i,j-1}'=4) \in a'$;
- ha $(p_{i,j}=2 \wedge p_{i,j+1}=4 \wedge p_{i,j-1}=5) \in a$ akkor $(p_{i,j+1}'=6 \wedge p_{i,j}'=1 \wedge p_{i,j-1}'=4) \in a'$;
- ha $(p_{i,j}=6 \wedge p_{i,j+1}=0 \wedge p_{i,j-1}=1) \in a$ akkor $(p_{i,j+1}'=2 \wedge p_{i,j}'=5 \wedge p_{i,j-1}'=0) \in a'$;
- ha $(p_{i,j}=6 \wedge p_{i,j+1}=4 \wedge p_{i,j-1}=1) \in a$ akkor $(p_{i,j+1}'=6 \wedge p_{i,j}'=5 \wedge p_{i,j-1}'=0) \in a'$;
- ha $(p_{i,j}=6 \wedge p_{i,j+1}=0 \wedge p_{i,j-1}=5) \in a$ akkor $(p_{i,j+1}'=2 \wedge p_{i,j}'=5 \wedge p_{i,j-1}'=4) \in a'$;
- ha $(p_{i,j}=6 \wedge p_{i,j+1}=4 \wedge p_{i,j-1}=5) \in a$ akkor $(p_{i,j+1}'=6 \wedge p_{i,j}'=5 \wedge p_{i,j-1}'=4) \in a'$;

Balra operátor: egy, paramétere van a honnan. A honnan pozíción található objektumot mozgatja balra, ha az megfelel az előfeltételeknek.

$$\begin{aligned} \text{Dom (Balra)} = & \{ a \mid a \in A \wedge \\ & ((p_{i,j}=1 \wedge (p_{i,j-1}=0 \vee p_{i,j-1}=4)) \\ & \vee (p_{i,j}=2 \wedge (p_{i,j+1}=1 \vee p_{i,j+1}=5) \wedge (p_{i,j-1}=0 \vee p_{i,j-1}=4)) \\ & \vee (p_{i,j}=6 \wedge (p_{i,j+1}=5 \vee p_{i,j+1}=1) \wedge (p_{i,j-1}=0 \vee p_{i,j-1}=4)) \\ & \vee (p_{i,j}=5 \wedge (p_{i,j-1}=0 \vee p_{i,j-1}=4))) \}, i=1, \dots; j=1, \dots \} \end{aligned}$$

$$\text{Rng(Balra): } a' = \text{Balra}(a) \in A \text{ ahol}$$

$$\begin{aligned} \text{ha } (p_{i,j}=1 \wedge p_{i,j-1}=0) \in a \text{ akkor } (p_{i,j-1}'=1 \wedge p_{i,j}'=0) \in a'; \\ \text{ha } (p_{i,j}=1 \wedge p_{i,j-1}=4) \in a \text{ akkor } (p_{i,j-1}'=5 \wedge p_{i,j}'=0) \in a'; \\ \text{ha } (p_{i,j}=5 \wedge p_{i,j-1}=0) \in a \text{ akkor } (p_{i,j-1}'=1 \wedge p_{i,j}'=4) \in a'; \\ \text{ha } (p_{i,j}=5 \wedge p_{i,j-1}=4) \in a \text{ akkor } (p_{i,j-1}'=5 \wedge p_{i,j}'=4) \in a'; \end{aligned}$$

$$\begin{aligned} \text{ha } (p_{i,j}=2 \wedge p_{i,j-1}=0 \wedge p_{i,j+1}=1) \in a \text{ akkor } (p_{i,j-1}'=2 \wedge p_{i,j}'=1 \wedge p_{i,j+1}'=0) \in a'; \\ \text{ha } (p_{i,j}=2 \wedge p_{i,j-1}=4 \wedge p_{i,j+1}=1) \in a \text{ akkor } (p_{i,j-1}'=6 \wedge p_{i,j}'=1 \wedge p_{i,j+1}'=0) \in a'; \\ \text{ha } (p_{i,j}=2 \wedge p_{i,j-1}=0 \wedge p_{i,j+1}=5) \in a \text{ akkor } (p_{i,j-1}'=2 \wedge p_{i,j}'=1 \wedge p_{i,j+1}'=4) \in a'; \\ \text{ha } (p_{i,j}=2 \wedge p_{i,j-1}=4 \wedge p_{i,j+1}=5) \in a \text{ akkor } (p_{i,j-1}'=6 \wedge p_{i,j}'=1 \wedge p_{i,j+1}'=4) \in a'; \\ \text{ha } (p_{i,j}=6 \wedge p_{i,j-1}=0 \wedge p_{i,j+1}=1) \in a \text{ akkor } (p_{i,j-1}'=2 \wedge p_{i,j}'=5 \wedge p_{i,j+1}'=0) \in a'; \\ \text{ha } (p_{i,j}=6 \wedge p_{i,j-1}=4 \wedge p_{i,j+1}=1) \in a \text{ akkor } (p_{i,j-1}'=6 \wedge p_{i,j}'=5 \wedge p_{i,j+1}'=0) \in a'; \\ \text{ha } (p_{i,j}=6 \wedge p_{i,j-1}=0 \wedge p_{i,j+1}=5) \in a \text{ akkor } (p_{i,j-1}'=2 \wedge p_{i,j}'=5 \wedge p_{i,j+1}'=4) \in a'; \\ \text{ha } (p_{i,j}=6 \wedge p_{i,j-1}=4 \wedge p_{i,j+1}=5) \in a \text{ akkor } (p_{i,j-1}'=6 \wedge p_{i,j}'=5 \wedge p_{i,j+1}'=4) \in a'; \end{aligned}$$

Ahol: $i=1, \dots; j=1, \dots$

Heurisztika:

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

ahol $g(n)$ az előzőleg megtett lépések száma, $h(n)$ pedig a ládák célhelyektől mért tényleges távolságának összegével.

Deadlock:

Olyan ládapozíciókat értem, melyekben egy vagy több láda megreked és elmozdíthatatlanná válik.

$$\exists i \exists j (p_{i,j}=2 \wedge ((p_{i,j+1}=3 \wedge p_{i-1,j}=3) \vee (p_{i,j+1}=3 \wedge p_{i+1,j}=3)) \vee (p_{i,j-1}=3 \wedge p_{i+1,j}=3) \vee (p_{i,j-1}=3 \wedge p_{i-1,j}=3)) \in a$$

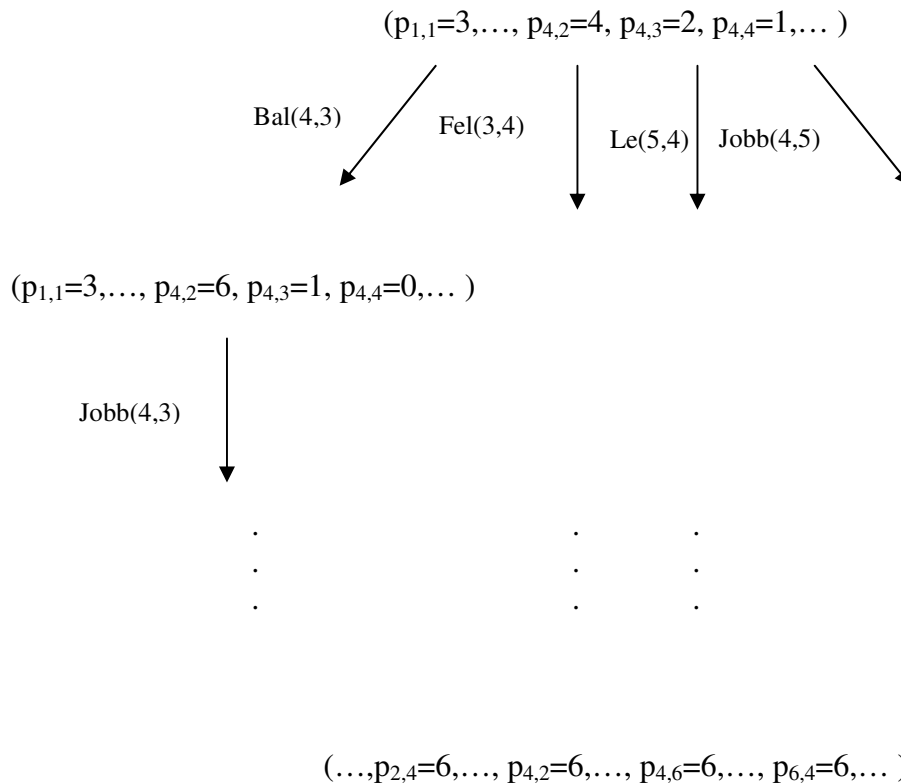
$$\exists i \exists j (p_{i,j}=2 \wedge (((p_{i,j+1}=3 \vee p_{i,j+1}=2 \vee p_{i,j+1}=6) \wedge (p_{i-1,j}=3 \vee p_{i-1,j}=2 \vee p_{i-1,j}=6) \wedge (p_{i-1,j+1}=3 \vee p_{i-1,j+1}=2 \vee p_{i-1,j+1}=6)) \vee ((p_{i,j+1}=3 \vee p_{i,j+1}=2 \vee p_{i,j+1}=6) \wedge (p_{i+1,j}=3 \vee p_{i+1,j}=2 \vee p_{i+1,j}=6) \wedge (p_{i+1,j+1}=3 \vee p_{i+1,j+1}=2 \vee p_{i+1,j+1}=6)) \vee ((p_{i,j-1}=3 \vee p_{i,j-1}=2 \vee p_{i,j-1}=6) \wedge (p_{i-1,j}=3 \vee p_{i-1,j}=2 \vee p_{i-1,j}=6) \wedge (p_{i-1,j-1}=3 \vee p_{i-1,j-1}=2 \vee p_{i-1,j-1}=6)) \vee ((p_{i,j-1}=3 \vee p_{i,j-1}=2 \vee p_{i,j-1}=6) \wedge (p_{i+1,j}=3 \vee p_{i+1,j}=2 \vee p_{i+1,j}=6) \wedge (p_{i+1,j-1}=3 \vee p_{i+1,j-1}=2 \vee p_{i+1,j-1}=6)))) \in a$$

$$\exists i \exists j (p_{i,j}=2 \wedge (((p_{i,j+1}=3 \vee p_{i,j+1}=2 \vee p_{i,j+1}=6) \wedge p_{i-1,j}=3 \wedge p_{i+1,j+1}=3) \vee ((p_{i,j+1}=3 \vee p_{i,j+1}=2 \vee p_{i,j+1}=6) \wedge p_{i+1,j}=3 \wedge p_{i-1,j+1}=3) \vee ((p_{i+1,j}=3 \vee p_{i+1,j}=2 \vee p_{i+1,j}=6) \wedge p_{i+1,j-1}=3 \wedge p_{i,j+1}=3) \vee ((p_{i+1,j}=3 \vee p_{i+1,j}=2 \vee p_{i+1,j}=6) \wedge p_{i,j-1}=3 \wedge p_{i+1,j+1}=3)))$$

Állapottér gráf:

Például, ha a pálya a következő módon néz ki:

3	3	3	3	3	3	3
3	3	3	4	3	3	3
3	3	3	2	3	3	3
3	4	2	1	2	4	3
3	3	3	2	3	3	3
3	3	3	4	3	3	3
3	3	3	3	3	3	3



Ahol: $i=1..7, j=1..7$

6. A program

A következő részben szeretném bemutatni, hogy a programom milyen fejlődésen ment át a számos változat megírása alatt, hogy milyen technikákkal szerettem volna redukálni a megoldás megtalálására fordított időt. Ezen technikákat kódrészleteken keresztül kívánom prezentálni, amelyek közül a kettes és hármas generáció teljes forráskódja fellelhető a CD-mellékleten.

1. Generáció

Ez a verzió még nagyon kezdetleges volt, nem mértem fel, hogy a pálya méretének, illetve a ládák számának növekedésével milyen mértékben növekszik a megoldás megtalálására fordított idő. Nem volt céloom továbbá az sem, hogy az optimális megoldást próbálja meg megtalálni az algoritmusom. A kód mindössze egy körfigyelést tartalmazott a megoldáskeresés gyorsítása érdekében. Amiért mégis fontosnak tartottam írni erről a generációról az volt, hogy itt szembesültem először a probléma igazi bonyolultságával, és hogy mekkora állapotteret eredményezhet egy rosszul megkonstruált algoritmus.

Körfigyelés:

```
private bool volte(Lada l)
{
    if (!_Allapotok.ContainsKey(l.JatekosHelye))
    {
        _Allapotok[l.JatekosHelye] = new Hashtable();
    }
    else
    {
        if
        (((Hashtable)_Allapotok[l.JatekosHelye]).ContainsKey(l.LadaHelye))
        return true;
    }

    ((Hashtable)_Allapotok[l.JatekosHelye])[l.LadaHelye] = "";

    return false;
}
```

Jól látható milyen hibát követtem el ebben a körfigyelésben azzal, hogy egy állapotot akkor tekintettem ismétlődőnek, ha egy láda a pálya egy adott pozícióján már megtalálható volt úgy, hogy a játékoskal egy bizonyos irányból toltuk az adott pályapozícióra. Ez természetesen nem igaz, ugyanis az én megközelítemben az jellemez egyértelműen egy állapotot, hogy a pályán a játékos, valamint az összes láda hol található.

Ebből következik, hogy azonos ládaállás annyi egymástól különböző állapotban jelenhet meg, ahány üres mező van a pályán. Tehát következésképpen a játékos is tekinthető lának. Amiből viszont az következik, hogy egy megoldás megtalálásához szükséges kiértékelt állapotok száma megközelítheti az n elem k -ad osztályú ismétlés nélküli kombinációinak számát. Azaz:

$$C_k^n = \frac{n \cdot (n-1) \cdot \dots \cdot (n-k+1)}{k \cdot (k-1) \cdot \dots \cdot 1} = \binom{n}{k}$$

Noha elkövettem ezt a hibát, mégis beláttam, hogy ha a játékos minden egyes lépése nyomán létrehozok egy új állapotot, akkor a kiértékelésre váró állapotok száma végesen elszabadulhat. Az állapotok számának drasztikus növekedését úgy próbáltam elkerülni, hogy csak a ládákkal léptem, ezen lépéseknek előfeltételeként azt határoztam meg, hogy a ládák lépéseinek irányával ellentétes irányból megközelíthetők legyenek a játékoskal. Ennek érdekében két szélességi keresést vezettem be, az egyik a ládákkal lépett a másik minden lépés előtt a játékoskal megpróbálta megközelíteni a kérdéses lánat. Ez rendkívül költséges megoldásnak bizonyult, mivel bármelyik lánával próbáltam meg lépni lefutott egy szélességi keresés, hogy a játékoskal elérhető-e a láda. Ezenkívül magával hozta az optimális megoldás elvesztésének problémáját is. Hiszen nem vettem figyelembe, hogy egy tolás mennyi költséggel jár, azaz mennyit kell lépni előtte a játékoskal. Ezt a kettéválasztást mindvégig megtartottam, persze a későbbi verziókban jelentős mutáción esett át, ezért tekinthető ez a generáció a következők ösének.

Végezetül lássuk a célállapot figyelést, ami azért költséges, mert egy ciklust kellett bevezetnem, hogy megvizsgáljam, hogy minden lánapozíció egyben célmező e.

```

public bool CeleE(Hashtable cel) {
    foreach (Point p in _LadakHelye) {
        if (!cel.ContainsKey(p)) return false;
    }
    return true;
}

```

2. Generáció

Ez a generáció már lényegesen hatékonyabb volt, mint az előző, több vizsgálatot vezettem, a körfigyelést kijavítottam, új megoldás keresési stratégiát vezettem be, hatékonyabb adatszerkezeteket kezdtem alkalmazni. Ami viszont továbbra is megmaradt, az a heurisztikus függvény hiánya, és hogy továbbra is minden körben lefuttattam egy szélességi keresést: ha egy ládával próbálok lépni, a láda elérhető-e a játékkal.

Körfigyelés:

```

public bool volte(Lada l) {
    bool volt;
    if (!Allapotok.ContainsKey(l.JatekosHelye)) {
        Allapotok[l.JatekosHelye] = new List<Hashtable>();
    }
    else {
        foreach (Hashtable allapot in
            ((List<Hashtable>)Allapotok[l.JatekosHelye])) {
            volt = true;
            foreach (Point p in l.LadakHelye) {
                if (!allapot.ContainsKey(p)) { volt = false;
                break; }
            }
            if (volt) return true;
        }
    }

    Hashtable h = new Hashtable();
    foreach (Point p in l.LadakHelye) {
        h[p] = "";
    }
    ((List<Hashtable>)Allapotok[l.JatekosHelye]).Add(h);
    return false;
}

```

A helyességnek viszont ára volt. Ahhoz ugyanis, hogy az összes ládahelyet megvizsgáljam ciklusokat kellett bevezetnem, ami lassította a körfigyelést, valamint arra a következtetésre kellett jutnom, hogy a hashtábla C# -ban egy elég lassú adatszerkezet, mikor új adatokat kell beszúrni. A program más részein viszont sikerült jelentős gyorsulást elérnem a megfelelő adatszerkezet kiválasztásával. Ez ugyan programozástechnikai dolog, de mivel a mesterséges intelligenciánál az idő fontos tényező, illetve a következő kódrészletek megértésének szempontjából sem elhanyagolható, ezért kitérek rá.

Az egyes állapotok tárolására integer tömböket vezettem be. Ezen tömbök egyes bitjei hordoztak információt arról, hogy a pálya adott pozícióján található-e objektum vagy sem. Ezért meglehetősen gyors lett a pálya játékosal történő bejárása a bitműveletek miatt, valamint az itt először megjelenő deadlock figyelés is hatékony lett tőle, amit a lentebb található 'LehetE' metódusba integráltam, ami megmodja azt is, hogy egy ládával lehet-e lépni, azaz a célmező üres e. A hátránya az volt, hogy külön nyilván kellett tartanom, hogy a ládák hol találhatók, mivel egy egyesből nem lehetett megállapítani, hogy az fal vagy láda. A másik hátulütője a dolognak, hogy a pálya maximum harminckettő egység széles lehetett.

```
public bool LehetE(Point p,int irányX,int irányY,Hashtable cel,int i)
{
    Point hely = new Point();
    hely.X = p.X + irányX; hely.Y = p.Y + irányY;
    if(SzelessegiLada.Uzemmod)
        if(cel.ContainsKey(hely)) return false;
        //A célpozíció a táblán belül van e
        if (p.X + irányX > 31 || p.Y + irányY > 31
            || p.X + irányX < 0 || p.Y + irányY < 0) return false;
        //A láda elért-e a játékosal
        if ((_Palya[p.X - irányX] & (1 << (p.Y - irányY))) != 0) return
false;
        //Megmondja hogy a célmező üres e
        if ((_Palya[hely.X] & (1 << (hely.Y))) != 0) return false;
        //Deadlock figyelés
        if (!cel.ContainsKey(hely)) {
            if (irányX != 0) {
                if (((_Palya[p.X + (irányX << 1)] & (1 << p.Y)) != 0) &&
                    ((_Palya[p.X + (irányX << 1)] & (1 << (p.Y + 1))) != 0) &&
                    ((_Palya[p.X + irányX] & (1 << (p.Y + 1))) != 0)) return
false;
                if (((_Palya[p.X + (irányX << 1)] & (1 << p.Y)) != 0) &&
                    ((_Palya[p.X + (irányX << 1)] & (1 << (p.Y - 1))) != 0) &&
                    ((_Palya[p.X + irányX] & (1 << (p.Y - 1))) != 0)) return
false;
            }
            else {
```

```

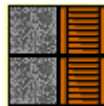
        if (((_Palya[p.X] & (1 << (p.Y + (iranyY << 1)))) != 0) &&
            ((_Palya[p.X + 1] & (1 << (p.Y + (iranyY << 1)))) != 0) &&
            ((_Palya[p.X + 1] & (1 << (p.Y + iranyY))) != 0)) return
false;

        if (((_Palya[p.X] & (1 << (p.Y + (iranyY << 1)))) != 0) &&
            ((_Palya[p.X - 1] & (1 << (p.Y + (iranyY << 1)))) != 0) &&
            ((_Palya[p.X - 1] & (1 << (p.Y + iranyY))) != 0)) return
false;

        }
    }
    return true;
}

```

A deadlock figyelést, mint ahogy az a fenti kódrészletből is kiderül, csak akkor alkalmaztam, ha a láda célpozíciója nem célmező, valamint ez a deadlock figyelés korántsem teljes még, mert csak a következő alakú mintákra illeszt:



Ahol a barnával színezett mezők a ládák, ez a szám legalább egy, de lehet négy is, a szürkék pedig a falak, amelyek száma maximum három.

A fenti kódrészletben látszik, hogy figyelek egy bizonyos üzemmódot. Nos, ez az üzemmód a kód gyorsaságának kulcsa.

```

public Lada KiterjesztLada(Lada l) {
    List<Point> ladak = l.LadakHelye;
    SzelessegiLada kereso;
    int szint = l.Szint;
    Point hely=new Point(); Lada uj;
    int ig = ladak.Count;

    for (int i = 0; i < ig; ++i) {
        if (!_Uzemmod) {
            Hashtable cel;
            foreach (DictionaryEntry de in _CelMezok) {
                if ((l.Palya[((Point)de.Key).X] & (1 <<
                    ((Point)de.Key).Y)) == 0) {
                    cel = new Hashtable();
                    cel[de.Key] = "";
                    kereso = new SzelessegiLada(l, cel,
ladak[i]);

                    uj = kereso.Kereses();
                    l.LadakHelye = ladak;
                    _Uzemmod = true;
                    if (uj != null) {

```

```

        hely = uj.LadakHelye[0]; uj.LadakHelye
= new List<Point>();
uj.LadakHelye[i] = hely;
        if (uj.CeleE(_CelMezok))
            return uj;
        if (!volte(uj)) NyiltLada.Insert (1,
uj);
    }
    }
    hely = ladak[i];
}
else hely = ladak[i];

    if (l.LehetE(hely, -1, 0, _CelMezok, i)) {
        uj = new Lada(l.Palya, l.JatekosHelye, ladak,
l.Iranyok); uj.Szint = szint;
        if (uj.Fel(hely, i) && !volte(uj)) {
            if (uj.CeleE(_CelMezok)) return uj;
NyiltLada.Add(uj);
        }
    }
    if (l.LehetE(hely, 1, 0, _CelMezok, i)) {
        uj = new Lada(l.Palya, l.JatekosHelye, ladak,
l.Iranyok); uj.Szint = szint;
        if (uj.Le(hely, i) && !volte(uj)) {
            if (uj.CeleE(_CelMezok)) return uj;
NyiltLada.Add(uj);
        }
    }
    if (l.LehetE(hely, 0, 1, _CelMezok, i)) {
        uj = new Lada(l.Palya, l.JatekosHelye, ladak,
l.Iranyok); uj.Szint = szint;
        if (uj.Jobbra(hely, i) && !volte(uj)) {
            if (uj.CeleE(_CelMezok)) return uj;
NyiltLada.Add(uj);
        }
    }
    if (l.LehetE(hely, 0, -1, _CelMezok, i)) {
        uj = new Lada(l.Palya, l.JatekosHelye, ladak,
l.Iranyok); uj.Szint = szint;
        if (uj.Balra(hely, i) && !volte(uj)) {
            if (uj.CeleE(_CelMezok)) return uj;
NyiltLada.Add(uj);
        }
    }
}
return null;
}

```

Tehát mielőtt az egyes ládákkal normál módban megpróbálnék jobbra, balra, fel vagy le lépni, létrehozok egy másik keresőt, ami természetesen már nem normál módu lesz, és lefuttatok vele minden ládára és minden célmezőre egy szélességi keresést, hogy a ládával valamelyik célmező elérhető e. Ha igen, akkor az így előállított új állapotot a nyitott állapotok elejére teszem, az összes többit csak utána. Ebből adódóan nem csak,

hogy egyből egy célmezőre ugrok, de még a nyitott állapotok listájának elejére is kerül az új állapot. Ezáltal továbbra sem lesz optimális a megoldás, de egy megoldás megtalálására fordított idő drasztikusan csökken.

3. Generáció

Ezt tekintem a programom végleges verziójának, noha az előző generáció bizonyíthatóan gyorsabb volt, mégis ez a legteljesebb mind a programozás technika, mind pedig a mesterséges intelligencia szempontjából. Az integer tömböt, mint nyelvi elemet az állapotok ábrázolására megtartottam, de számos új, a megoldás megtalálását gyorsító adatstruktúrát is bevezettem. Ilyen például, hogy a célállapotokat egy stringben is tároltam. A C# nyelvi sajátossága, hogy egy karaktert két bájtól tárol, ezt kihasználva egy célpozíció x koordinátáit, egy karakter felső bájton tárol, az y koordinátáját pedig az alsó bájton tároltam. Ezt a stringet még a program indulásakor létrehozom és a karaktereket növekvő sorrendbe rendezem. Az egyes láda pozíciókkal ugyan így teszek, és csak annyi a feladatom, hogy a ládapozíciók rendezettségét mindvégig fenntartsam. Ezáltal a célállapotok vizsgálatából a ciklus eltűnik, és a vizsgálat egy egyszerű, két string összehasonlítására redukálódik. Ezt mutatják be a következő kódrészletek.

Célállapot vizsgálat:

```
/// <summary>
/// Returns true if the current state is the goal state.
/// </summary>
private bool GoalState(string boxesPositions)
{
    return _goalState.Equals(boxesPositions);
}
```

A láda mozgatása utáni rendezettség fenntartása beszűrőrendezéssel:

```
/// <summary>
/// Replaces the old box position with the new box position and keeps
/// the sorting of box positions.
/// </summary>
private string InsertPosition(short newPosition, short oldPosition)
{
    char[] c = _currentBoxesPositions.ToCharArray();
    int index = Array.BinarySearch<char>(c, (char)oldPosition);
    c[index] = (char)newPosition;
    if (index < c.Length - 1 && _currentBoxesPositions[index + 1] <
        newPosition)
    {
        //if the new positin is higher that the latest...
    }
}
```

```

        do
        {
            c[index] = c[index + 1];
            ++index;
        } while (index < c.Length - 1 &&
_currentBoxesPositions[index + 1] < newPosition);
    }
    else if (index > 0 && _currentBoxesPositions[index - 1] >
newPosition) {
        //if the new positin is lower that the latest...
        do{
            c[index] = c[index - 1];
            --index;
        } while (index > 0 && _currentBoxesPositions[index - 1] >
newPosition);
    }
    //insert new position
    c[index] = (char)newPosition;
    return new string(c);
}

```

A ládapozíciók rendezett stringjét a körfigyelésben is felhasználtam. Ezáltal itt is megszűnt a ciklus, mint lassító tényező, következő kódrészletben ezt láthatjuk.

```

/// <summary>
/// Adds new state to history.
/// </summary>
/// <returns>Returns true if history already contains the new
state.</returns>
private bool AddToHistroy(short playerPosition, string state)
{
    bool result = _history.ContainsKey(state);
    if (result)
    {
        if (!(result &= (_history[state][playerPosition >> 8] & (1
<< ((byte)playerPosition))) != 0))
            _history[state][playerPosition >> 8] |= (1 <<
((byte)playerPosition));
    }
    else
    {
        _history[state] = new int[32];
        _history[state][playerPosition >> 8] |= (1 <<
((byte)playerPosition));
    }

    return result;
}

```

A `_history` egy dictionary, ami a hashtable generikus változata. Ebből a kódrészletből az látszik, hogy a hashtable kulcsai rendre a dobozpozíciók rendezett stringjei, az értékek pedig integer tömbök a játékos pozícióknak. Így ábrázolom, hogy egy ládaállás mellett a játékos a pálya mely pozícióján állt.

Mint azt már korábban említettem az integer tömb is megmaradt az egyes állapotok tárolására. A pálya játékkal történő bejárásán viszont sokat egyszerűsítettem és ez által is gyorsítottam úgy, hogy ez a bejárás minden állapotra már csak egyszer fut le. Tehát nem járom be a pályát minden ládával történő lépés előtt, csak egy listát készítek a pálya elérhető pozícióiról és minden egyes pozíciónál megjegyzem az odajutáshoz szükséges lépések irányát is. A következő metódus végzi ezt a bejárást.

```

/// <summary>
/// The player walks around the table in the current state.
/// </summary>
private string[,] PlayerWalks(short playerPosition)
{
    short[] openStates = new short[_emptyPositionsCount];
    string[,] tourDirections = new
    string[_emptyTable.Length, 32];
    int index = 1;

    _tourTable = new int[_currentTable.Length];
    Array.Copy(_currentTable, _tourTable, _emptyTable.Length);

    openStates[0] = playerPosition;
    _tourTable[playerPosition >> 8] |= (1 << (byte)playerPosition);

    for (int i = 0; i < index; i++)
    {
        for (int j = 0; j < _directions.Length; j++)
        {
            short position = (short)(openStates[i] +
            _directions[j]);
            if ((_tourTable[position >> 8] & (1 <<
            (byte)position)) == 0)
            {
                tourDirections[position >> 8, (byte)position] =
                tourDirections[openStates[i] >> 8, (byte)openStates[i]] + ","
                + _tourDirections[j];
                _tourTable[position >> 8] |= (1 <<
                (byte)position);
                openStates[index] = position;
                ++index;
            }
        }
    }

    return tourDirections;
}

```

Ez a bejárás nem más valójában, mint egy mintára lefuttatott flood fill algoritmus, ahol a minta a pálya. A kódrészletben látható az is, hogy fix méretű tömböt alkalmazok a nyitott állapotok tárolására, ami úgy jöhet létre, hogy már a keresés elején összeszámolom az üres, falat nem tartalmazó mezőket. Teszem ezt azért, mert a C#-ban a lista, mint dinamikus adatszerkezet szintén meglehetősen lassú beszúrásnál.

Ezt az integer tömböt továbbá felhasználtam már az előző verzióban is megjelenő, de itt már kibővített deadlock állapotok vizsgálatára. A következő metódus e vizsgálatokat mutatja be.

```
/// <summary>
/// Returns true if the new state is a deadlock.
/// </summary>
private bool DeadLock(short newBoxPosition, short directionX, short
directionY){
    //initialize return value
    bool deadlock = false;

    if (!GoalPosition(newBoxPosition))
    {
        //decompose position to x and y coordinates
        byte x = (byte)(newBoxPosition >> 8);
        byte y = (byte)newBoxPosition;

        if (directionX > 0)
        {
            //check dead locks if step down
            deadlock = ((_currentTable[x] & (1 << y + 1)) > 0)
            && ((_currentTable[x + 1] & (1 << y + 1)) > 0) &&
            ((_currentTable[x + 1] & (1 << y)) > 0);
            deadlock |= ((_currentTable[x] & (1 << y - 1)) > 0)
            && ((_currentTable[x + 1] & (1 << y - 1)) > 0) &&
            ((_currentTable[x + 1] & (1 << y)) > 0);

            deadlock |= ((_emptyTable[x] & (1 << y - 1)) > 0)
            && ((_emptyTable[x + 1] & (1 << y)) > 0);

            deadlock |= ((_emptyTable[x] & (1 << y + 1)) > 0)
            && ((_emptyTable[x + 1] & (1 << y)) > 0);

            deadlock |= ((_currentTable[x + 1] & (1 << y)) >
            0) && ((_emptyTable[x] & (1 << y + 1)) > 0) &&
            ((_emptyTable[x + 1] & (1 << y - 1)) > 0);

            deadlock |= ((_currentTable[x + 1] & (1 << y)) > 0)
            && ((_emptyTable[x] & (1 << y - 1)) > 0) &&
            ((_emptyTable[x + 1] & (1 << y + 1)) > 0);
        }
        else
        if (directionX < 0)
        {
            //check dead locks if step up
            deadlock = ((_currentTable[x] & (1 << y + 1)) > 0)
            && ((_currentTable[x - 1] & (1 << y + 1)) > 0) &&
            ((_currentTable[x - 1] & (1 << y)) > 0);

            deadlock |= ((_currentTable[x] & (1 << y - 1)) > 0)
            && ((_currentTable[x - 1] & (1 << y - 1)) > 0) &&
            ((_currentTable[x - 1] & (1 << y)) > 0);

            deadlock |= ((_emptyTable[x] & (1 << y + 1)) > 0)
            && ((_emptyTable[x - 1] & (1 << y)) > 0);
        }
    }
}
```

```

        deadlock |= ((_emptyTable[x] & (1 << y - 1)) > 0)
        && ((_emptyTable[x - 1] & (1 << y)) > 0);

        deadlock |= ((_currentTable[x - 1] & (1 << y)) > 0)
        && ((_emptyTable[x] & (1 << y + 1)) > 0) &&
        ((_emptyTable[x - 1] & (1 << y - 1)) > 0);

        deadlock |= ((_currentTable[x - 1] & (1 << y)) > 0)
        && ((_emptyTable[x] & (1 << y - 1)) > 0) &&
        ((_emptyTable[x - 1] & (1 << y + 1)) > 0);
    }
    else
    if (directionY > 0)
    {
        //check dead lokcks if step right
        deadlock = ((_currentTable[x] & (1 << y + 1)) > 0)
        && ((_currentTable[x - 1] & (1 << y + 1)) > 0) &&
        ((_currentTable[x - 1] & (1 << y)) > 0);

        deadlock |= ((_currentTable[x] & (1 << y + 1)) > 0)
        && ((_currentTable[x + 1] & (1 << y + 1)) > 0) &&
        ((_currentTable[x + 1] & (1 << y)) > 0);

        deadlock |= ((_emptyTable[x] & (1 << y + 1)) > 0)
        && ((_emptyTable[x - 1] & (1 << y)) > 0);

        deadlock |= ((_emptyTable[x] & (1 << y + 1)) > 0)
        && ((_emptyTable[x + 1] & (1 << y)) > 0);

        deadlock |= ((_emptyTable[x - 1] & (1 << y)) > 0)
        && ((_currentTable[x] & (1 << y + 1)) > 0) &&
        ((_emptyTable[x + 1] & (1 << y + 1)) > 0);

        deadlock |= ((_emptyTable[x + 1] & (1 << y)) > 0) &&
        ((_currentTable[x] & (1 << y + 1)) > 0) &&
        ((_emptyTable[x - 1] & (1 << y + 1)) > 0);
    }
    else
    if (directionY < 0)
    {
        //check dead lokcks if step left
        deadlock = ((_currentTable[x] & (1 << y - 1)) > 0)
        && ((_currentTable[x - 1] & (1 << y - 1)) > 0) &&
        ((_currentTable[x - 1] & (1 << y)) > 0);

        deadlock |= ((_currentTable[x] & (1 << y - 1)) > 0)
        && ((_currentTable[x + 1] & (1 << y - 1)) > 0) &&
        ((_currentTable[x + 1] & (1 << y)) > 0);

        deadlock |= ((_emptyTable[x] & (1 << y - 1)) > 0)
        && ((_emptyTable[x - 1] & (1 << y)) > 0);

        deadlock |= ((_emptyTable[x] & (1 << y - 1)) > 0)
        && ((_emptyTable[x + 1] & (1 << y)) > 0);

        deadlock |= ((_emptyTable[x - 1] & (1 << y)) > 0)
        && ((_currentTable[x] & (1 << y - 1)) > 0) &&
        ((_emptyTable[x + 1] & (1 << y - 1)) > 0);
    }

```

```

        deadlock |= ((_emptyTable[x + 1] & (1 << y)) > 0)
        && ((_currentTable[x] & (1 << y - 1)) > 0) &&
        ((_emptyTable[x - 1] & (1 << y - 1)) > 0);
    }
    return deadlock;
}

```

Ezek a deadlock vizsgálatok az összes általam ismert mintát kiszűrnek, amiket már az állapottér reprezentációban is leírtam. Ez az ott leírtaktól annyiban tér el, hogy a mintaillesztések számának csökkentése érdekében lépésirányonként szétbontottam a vizsgálatokat.

Továbbá annyi magyarázattal szeretnék élni, hogy itt már két int tömb is megjelenik, ezek közül az `_emptyTable` tartalmazza a falakat, a `_currentTable` pedig a ládákat. Az is látható, hogy eltűnt az üzemmód vizsgálat, tehát itt már nem futtatok le minden körben egy szélességi keresést, hogy van-e elérhető célpozíció az egyes ládák számára, hanem a gyorsaság növelése és az optimális megoldás megtalálása érdekében csak a heurisztikámra hagyatkozom, amit majd később fejtek ki. A ládával történő lépés validálását is egy másik metódusba szerveztem ki:

```

/// <summary>
/// Returns true if the destination position is empty.
/// </summary>
private bool ValidStep(short to, short playerPosition)
{
    return ((_currentTable[(to >> 8)] & (1 << (byte)to)) == 0 &&
        (_currentTable[(playerPosition >> 8)] & (1 <<
        (byte)playerPosition)) == 0);
}

```

Ez a metódus mindössze annyit vizsgál, hogy a célpozíció üres e, és hogy a lépés irányával ellentétes pozíció elérté vált-e a játékkal történő bejárás során.

További heurisztikus információ gyanánt, bevezettem egy további int tömböt is, amely azon pályapozíciókat tartalmazza, amelyekről kiindulva, függetlenül a többi láda elhelyezkedésétől, nem elérhető egyetlen célmező sem. Ennek a tömbnek az inicializálását is már a keresés előtt megteszem. Következzék most e tömb inicializálása, majd a vizsgálat, hogy a ládával történő lépés célpozíciója ‘rossz pozíció e’.

A tömb inicializálása:

```
/// <summary>
/// Gets the list of wrong positions starting from a corner.
/// </summary>
private List<short> SearchWrongPosition(int[] table, int[]
goalPositions, short position)
{
    //Depth-first search
    if ((table[position >> 8] & (1 << (byte)position)) == 0)
    {
        //Creates new instance of wrong places
        List<short> wrongPlaces = new List<short>();
        //Adds the first element to wrong places
        wrongPlaces.Add(position);
        for (int i = 0; i < wrongPlaces.Count; i++)
        {
            //If a goal position is available...
            if ((goalPositions[wrongPlaces[i] >> 8] & (1 <<
(byte) wrongPlaces[i]))) == 0)
                return null;
            //Inits coordinates
            byte x = (byte)(wrongPlaces[i] >> 8);
            byte y = (byte)wrongPlaces[i];
            //Steps to the right
            byte positionX = x;
            byte positionY = (byte)(y + 1);
            short pos = (short)((positionX << 8) | positionY);
            if ((table[positionX] & (1 << positionY)) == 0 &&
(table[x] & (1 << (y - 1))) == 0 && wrongPlaces.IndexOf(pos) == -1)
                wrongPlaces.Add(pos);
            //Steps to the left
            positionX = x;
            positionY = (byte)(y - 1);
            pos = (short)((positionX << 8) | positionY);
            if ((table[positionX] & (1 << positionY)) == 0 &&
(table[x] & (1 << (y + 1))) == 0 && wrongPlaces.IndexOf(pos) == -1)
                wrongPlaces.Add(pos);
            //Steps up
            positionX = (byte)(x + 1);
            positionY = y;
            pos = (short)((positionX << 8) | positionY);
            if ((table[positionX] & (1 << positionY)) == 0 &&
(table[x - 1] & (1 << (y))) == 0 && wrongPlaces.IndexOf(pos) == -1)
                wrongPlaces.Add(pos);
            //Steps down
            positionX = (byte)(x - 1);
            positionY = y;
            pos = (short)((positionX << 8) | positionY);
            if ((table[positionX] & (1 << positionY)) == 0 &&
(table[x + 1] & (1 << (y))) == 0 && wrongPlaces.IndexOf(pos) == -1)
                wrongPlaces.Add(pos);
        }

        return wrongPlaces;
    }
    return null;
}
```

Ez az inicializálás, ahogy az látszik, a pálya bejárása mélységi kereséssel a pálya egyes sarkaiból kiindulva. Azért a sarkokból, mert a sarkok bizonyosan rossz pozíciók, tehát nem elérhető belőlük egyetlen célmező sem. A sarkokból elért pozíciók közül pedig azt tekintem rossznak, amelyek olyan úton helyezkednek el, amelyek nem vezetnek célpozícióba, hanem csak egy másik sarokba.

A vizsgálat:

```
/// <summary>
/// Returns true if any goal positions cannot be accessed from the new
box position.
/// </summary>
private bool WrongPlace(short newBoxPosition)
{
    return (_wrongPlaces[newBoxPosition >> 8] & (1 <<
((byte)newBoxPosition))) != 0;
}
```

Most pedig következzen a heurisztikus függvényem bemutatása, amitől az optimális megoldás megtalálását és a kiterjesztett állapotok számának drasztikus csökkenését vártam. Mint ahogy azt már az állapottér reprezentációban is leírtam, ebben a függvényben a játékos által korábban megtett lépések számát kombináltam össze a ládák célhelyektől mért tényleges távolságának összegével. Ahol ez a tényleges távolság azon legrövidebb út hossza, melyet egy pályapozícióról bejárva valamely célmezőhöz jutunk. Ezen utak hosszát még a tényleges keresés előtt inicializálom, azaz letárolom minden pályapozíció esetén, hogy hány lépésből lehet elérni az egyes célmezőket. Ez persze nem azonos a tényleges bejárással, ugyanis itt a pálya akadályként mindössze a falakat tartalmazza, ládákat nem. Tehát ezek a távolságok az akadálymentes legrövidebb távolságok minden pályapozícióra és minden célmezőre.

Az inicializálást a következő metódusban implementálom. Ez a metódus egy szélességi keresést tartalmaz garantálva a legrövidebb utakat, és ami a pálya játékosal történő bejárásához hasonlóan egy flood fill algoritmus, amit minden egyes célmezőre lefuttatok.

```

/// <summary>
/// Gets the map of the distances without any obstacles for each goal
state.
/// </summary>
public override int[, ] GetNoObstacleDistance(int[] table, string
goalPositions, int emptyPositionsCount)
{
    int[, ] result = new int[table.Length, 32,
goalPositions.Length];
    for (int i = 0; i < goalPositions.Length; i++)
    {
        int[] tableClone = new int[table.Length];
        Array.Copy(table, tableClone, table.Length);
        short[,] openStates = new short[emptyPositionsCount, 2];
        int index = 1;
        openStates[0, 0] = (short)goalPositions[i];
        tableClone[goalPositions[i] >> 8] |= (1 <<
(byte)goalPositions[i]);
        for (int j = 0; j < index; j++)
        {
            for (int k = 0; k < _directions.Length; k++)
            {
                short position = (short)(openStates[j, 0] +
_directions[k]);
                if ((tableClone[position >> 8] & (1 <<
(byte)position)) == 0)
                {
                    tableClone[position >> 8] |= (1 <<
(byte)position);
                    openStates[index, 0] = position;
                    openStates[index, 1] =
(short)(openStates[j, 1] + 1);
                    result[position >> 8, (byte)position, i] =
openStates[index, 1];
                    ++index;
                }
            }
        }
    }

    return result;
}

```

A kérdéses heurisztikus függvény pedig a következő.

```

/// <summary>
/// Inserts new state into the open states.
/// </summary>
private void HeuristicsInsert(State insertingState)
{
    insertingState.HeuristicsValue +=
_distancesMap[insertingState.BoxesPositions[0] >> 8,
(byte)insertingState.BoxesPositions[0], 0] + 1;
    for (int i = 1; i < _goalState.Length; i++)
    {
        insertingState.HeuristicsValue +=
_distancesMap[insertingState.BoxesPositions[i] >> 8,
(byte)insertingState.BoxesPositions[i], i]
+ _distancesMap[insertingState.BoxesPositions[i] >> 8,
(byte)insertingState.BoxesPositions[i], i - 1];
    }
}

```

```

        insertingState.HeuristicsValue *=
insertingState.Directions.Length;
        _openStates.Insert(insertingState);
    }

```

Tehát végigmegyünk a ládapozíciókat tartalmazó stringen, és bízva benne, hogy a rendezettségnek köszönhetően az adott i -edik, ládapozícióhoz az i -edik célpozíció van a legközelebb, összeadom a legrövidebb távolságokat, amivel az adott ládapozíciókról a hozzájuk tartozó célmező elérhető. Valamint hozzáadom azon utak hosszát, amelyeket bejárva az előző ládapozíciókhoz tartozó célmezőktől az aktuális ládapozíciókhoz jutunk. Végül ezt az összeget megszorozom a játékosal eddig megtett lépések számával azért, hogy az előzőleg megtett lépéseknek kellő súlya legyen a heurisztikus érték meghatározásakor. Sajnos ez a heurisztika nem biztosítja az optimális megoldást. A probléma okát pedig abban látom, hogy jelen tudásom szerint nem tudom meghatározni, hogy melyik célpozícióra melyik láda kerüljön. Amennyiben ezt meg tudnám tenni, a probléma egy egyszerű utazó ügynök problémára redukálódna, és bonyolult, lassú megoldáskeresést eredményező heurisztikák nélkül is meg lehetne találni az optimális megoldást. Azaz könnyű lenne gyors és optimális heurisztikát konstruálni.

Ezen okfejtés után szeretnék még kitérni egy pár szó erejéig a nyitott állapotokat tartalmazó adatszerkezetemre, ami nem más, mint egy általam konstruált bináris halom, amit egy vektorral valósítottam meg. Így nem kell külön nyilvántartanom az egyes állapotoknál a szülő illetve a gyermek állapotokat, mert minden egyes csomópont szülője, kivéve a gyökér csomópontot, a vektor $(i - 1) / 2$ pozícióján helyezkedik el, ahol i az adott állapot vektorbeli indexe. Ebből adódóan egy adott csomópont esetén könnyedén meg lehet határozni a szülőjének vagy valamelyik gyermekének indexét. A bináris halom további nagy előnye, hogy a legkisebb vagy legnagyobb értékkel rendelkező állapot, attól függően, hogy miként definiáljuk a halmot, mindig a gyökér elem, tehát ez egy elsőbbségi sor.

Új elemet mindig a vektor végére szúrok be, majd megvizsgálom, hogy kisebb-e az értéke, mint a szülőjének, ha igen akkor felcserélem őket. Ezt mindaddig folytatom, amíg létezik szülő csomópont és annak értéke nagyobb, mint az újonnan beszúrt. A gyökérelem törlése úgy történik, hogy felülírom a vektor legutolsó elemével, majd ezt az új gyökérelemet felcserélem azon legkisebb értékű gyermekével, melynek értéke kisebb, mint az új gyökérelem. Ezt mindaddig folytatom, amíg létezik olyan gyermek elem amelynek értéke kisebb, mint a végéről áthelyezett csomópont értéke.

Végezetül pedig következzen a keresés motorja, a kiterjesztés metódusa:

```
/// <summary>
/// Extends the start state and searches the solution.
/// </summary>
private State Expand()
{
    int counter = 0;
    while (!_openStates.IsEmptyTree)
    {
        ++counter;
        State currentState = _openStates.PopLeastElement();

        if (GoalState(currentState.BoxesPositions))
            return currentState;

        _currentBoxesPositions = currentState.BoxesPositions;
        _currentTable = new int[_emptyTable.Length];
        Array.Copy(_emptyTable, _currentTable,
            _emptyTable.Length);

        for (int i = 0; i < _currentBoxesPositions.Length; i++)
        {
            _currentTable[_currentBoxesPositions[i] >> 8] |= 1 <<
(byte)_currentBoxesPositions[i];
        }

        string[,] directions =
PlayerWalks(currentState.PlayerPosition);
        for (int i = 0; i < _currentBoxesPositions.Length; i++)
        {
            for (int j = 0; j < _directions.Length; j++)
            {
                short newPosition =
(short)(_currentBoxesPositions[i] + _directions[j]);
                short inversePosition =
(short)(_currentBoxesPositions[i] + _directionsInverse[j]);

                if (ValidStep(newPosition, inversePosition) &&
!WrongPlace(newPosition))
                {
                    _currentTable[_currentBoxesPositions[i] >>
8] ^= (1 << (byte)_currentBoxesPositions[i]);
                    _currentTable[newPosition >> 8] |= (1 <<
(byte)newPosition);

                    if (!DeadLock(newPosition,
(short)(_directions[j] >> 8), (byte)_directions[j]))
                    {
                        _currentTable[newPosition >> 8] ^= (1
<< (byte)newPosition);

                        _currentTable[_currentBoxesPositions[i] >> 8] |= (1 <<
(byte)_currentBoxesPositions[i]);

                        if ((_tourTable[inversePosition >> 8]
& (1 << (byte)inversePosition)) != 0)
                        {
                            string boxesPositions =
InsertPosition(newPosition, (short)_currentBoxesPositions[i]);
```

```

                                if
(!AddToHistroy((short)_currentBoxesPositions[i], boxesPositions))
                                {
                                    HeuristicsInsert(new
State(boxesPositions, (short)_currentBoxesPositions[i],
                                currentState.Directions +
directions[inversePosition >> 8, (byte)inversePosition] + ", "
                                + _tourDirections[j]));
                                }
                                }
                                else
                                {
                                    _currentTable[newPosition >> 8] ^= (1
<< (byte)newPosition);

                                _currentTable[_currentBoxesPositions[i] >> 8] |= (1 <<
(byte)_currentBoxesPositions[i]);
                                }
                                }
                                }

                                }

                                }
                                return null;
}

```

A metódus egy szélességi keresést valósít meg, ami a következőképp működik:

- 1) Kiveszem a legkisebb elemet a nyílt állapotok közül, ha van még egyébként nincs megoldás.
- 2) Megvizsgálom az állapotot, hogy célállapot-e, ha igen akkor visszaadom a megoldást egyébként 3.
- 3) Lemásolom a falakat tartalmazó tömböt, és elhelyezem benne a ládákat, majd bejárom ezt a játékossal, meghatározva ezáltal, hogy mely ládák, mely irányokból érhetők el.
- 4) Veszem a következő ládát, ha van még egyébként 1.
 - a) Veszem a következő irányt, ha van még egyébként 4.
 - i) Megvizsgálom, hogy valid-e a lépés, azaz ez a célmező üres-e, és hogy megközelíthető-e a játékossal, a lépés irányával ellentétes irányból. Ha nem akkor a).
 - ii) Megvizsgálom, hogy a célmező szerepel-e a 'rossz' ládapozíciók között, ha nem akkor a).

- iii) Megvizsgálom, hogy az újonnan kialakuló állapot deadlock e, ha igen akkor
 - a) egyébként iv).
- iv) Beszúrom az új állapotot.

7. Összegzés

Úgy gondolom, hogy azirányú törekvésem, hogy optimális és gyors heurisztikát találjak, nem járt sikerrel. Indokolható ez a nem megfelelően választott heurisztikával, illetve a probléma összetettségével. A véglegesnek ítélt verzió heurisztikája ugyanis már megközelíti az optimális megoldást, de csak a kisebb pályák esetén éri azt el, és megfelelően gyors. Lent található egy táblázat, amelyből látható, hogy a pálya méretének és a dobozok számának növekedése miként befolyásolja az ismétlés nélküli kombinációk, ez esetben a csomópontok számát, amiből már következtetni lehet mi számít kis pályának. Ez természetesen egy idealizált érték, mert a falak és a többi láda ezt a számot csökkenti. Az értékekből észrevehető, hogy nem elegendő egy hatalmas pálya, hogy sok lehetséges állapot legyen, mivel egy nagy pálya kisszámú ládával kevés kombinációt eredményez. Ebből következően én kis pályának a kis méretű pályát kevés ládával vagy nagy kiterjedésű pályát kevés ládával tekintetem. Az első pálya például ilyen, melynek adatati a táblázat első három sorában található. Azon pályáknál, ahol az időmezőben végtelen található, memóriaproblémákba ütköztem. A lépések irányainak folyamatos tárolása ugyanis meglehetősen költségesnek bizonyult. Ez a memóriaprobléma akkor sem oldódott meg, amikor a lépések irányi helyett csak a megtett lépések számát tároltam. Az interneten és a könyvekben történő kutatások is arra világítottak rá, hogy ezt a problémát a mesterséges intelligencia kutatók újra és újra előveszik és a tudomány aktuális állása szerint próbálják megoldani. A külső források is arra engednek következtetni, hogy ezen témakör kutatóinak sem sikerült még megfelelő heurisztikus függvényt kifejleszteniük. Ezek a tények engem is arra fognak ösztönözni, hogy noha a harmadik generációt a dolgozatom szempontjából véglegesnek tekintem, én is újra és újra elővegyem a problémát és megpróbáljam más megközelítésből megvizsgálni a tudomány illetve a saját tudásom aktuális állása szerint megoldani.

n	k	C	t
56	2	1540	<1
56	4	367290	<1
56	7	231917400	3
133	2	8778	<1

133	7	124397910208	51
133	12	3,836E+16	$+\infty$
133	17	1,232E+21	$+\infty$
157	2	12246	<1
157	7	407340975756	64
157	12	3,043E+17	$+\infty$
157	17	2,450E+22	$+\infty$
157	22	3,879E+26	$+\infty$

Ahol n azon mezők száma ahová lépni lehet, k pedig a dobozok száma plusz a játékos.
A t pedig a programom számára egy megoldás megtalálásához szükséges idő másodpercben kifejezve.

8. Felhasznált irodalom

Stuart Russell- Peter Norvig: Mesterséges intelligencia modern megközelítésben

Internet: <http://hu.wikipedia.org/>

<http://hu.wikipedia.org/>

ai.kaist.ac.kr/~jkim/cs570-2000/Resource/Group4Final.ppt

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.41&rep=rep1&type=pdf>

(Joseph C. Culberson – Sokoban is PSPACE-complete)