

Sudoku: Molekulák és reakciók

Pintér Gergő

Konzulens:

Dr. Németh Zsolt

ÓBUDAI EGYETEM
NEUMANN JÁNOS INFORMATIKAI KAR
2012. november 10.

Tartalomjegyzék

Tartalomjegyzék	2
1 Bevezetés.....	4
2 Kémiai modell	4
2.1 A kémiai kalkulus	5
2.2 Magasabb-rendű kémiai nyelv	5
2.3 Prímszám keresés kémiai nyelven.....	6
3 Sudoku.....	6
3.1 Kitöltési stratégiák.....	8
3.1.1 Triviális lépés	8
3.1.2 Kizárás	8
3.1.3 Kiválasztás	8
3.1.4 Következtetés	9
3.1.5 X-szárnny	9
3.1.6 Kardhal	10
3.1.7 Tippelés	10
4 Megoldási módszerek	11
4.1 Rekurzio	11
4.2 Brute force	11
4.3 197 karakteres Python kód	11
4.4 Perl háromsoros	12
4.5 Deklaratív megoldások	13
4.6 Megoldó módszerek természeti analógián.....	13
4.6.1 Mesterséges méh kolónia algoritmus.....	13
4.6.2 Szimulált hűtés	14
5 Kémiai megoldás.....	14
5.1 Adatok kémiai reprezentációja	14
5.2 Prímkód.....	16
5.3 Kémiai algoritmus.....	17
5.4 Implementációs részletek	19
5.4.1 HOCL kód.....	19
6 Összegzés.....	20
7 Irodalomjegyzék.....	21

1 Bevezetés

Ben Laurie szerint a sudoku „egy DoS¹ támadás az emberi értelem ellen”². Igaza lehet, hiszen ott van mindenütt, futótűzként terjedt szét a világon és a népszerűsége azóta is töretlen, az újságok, magazinok rejtvényrovatainak állandó szereplője lett, játsszák papíron ceruzával, számítógépen, és telefonon is. Egyszerű, olcsó, szórakoztató, ugyanakkor elgondolkodtató játék és legalább ilyen népszerű probléma a megoldó algoritmusok íróinak is, hogy elkészítsék a leggyorsabb, legrövidebb, vagy valamely paradigmára vagy nyelvre illesztett legszebb, legelegánsabb megoldó algoritmust.

Ez a dolgozat is egy újabb megoldást ad, miközben igyekszik bemutatni a megoldáshoz használt modellt és összevetni a más paradigmákkal.

2 Kémiai modell

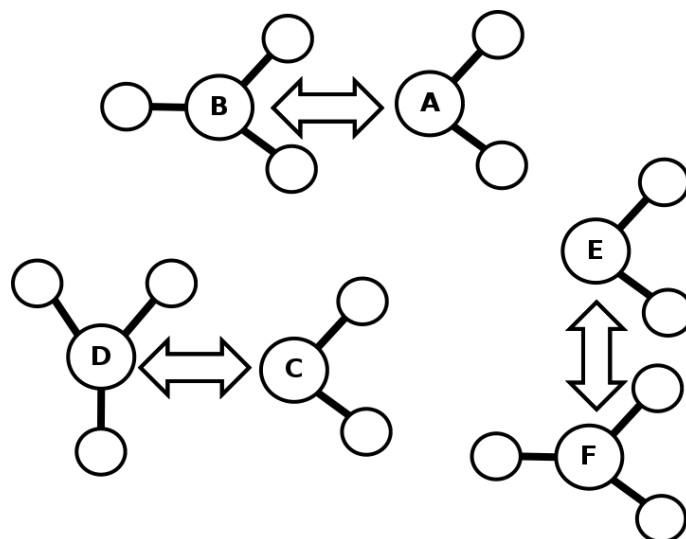
A számítási feladatokat általában valamilyen elemi lépések sorozataként modellezzük, azonban ennek nem kell feltétlenül így lennie, mind a „lépés”, mind a sorozat mesterségesen létrehozott absztrakció és nem a számítás lényegéből adódik. Lehetne például a számítást elképzelni adatok és műveletek reakciójaként is: egy kémiai oldat adatokat és műveleteket reprezentáló molekulákat tartalmaz, melyek reakcióba lépnek a megfelelő reakciós szabályoknak megfelelően és így az adatokon műveletek végezhetőek(1).

Tehát maga a számítás reakciókon keresztül valósul meg és ezek a reakciók indeterminisztikusan és párhuzamosan mennek végbe. Nem lehet megmondani, hogy melyik molekula melyik molekulával fog reakcióba lépni, csak azt, hogy képesek-e reakcióra (2). Továbbá, két különböző lefutás során a molekulák akár teljesen más sorrendben reagálhatnak egymással, mint egy korábbi futás közben történt. Ettől lesz a folyamat indeterminisztikus. Ez a szemlélet rendkívül távol áll a Neumann-i elvű végrehajtással, ahol a definiált utasítások minden futás során abban a sorrendben hajtódnak végre, amilyen sorrendben leírták őket. Párhuzamosság pedig azt jelenti, hogy míg A molekula reakcióba lép B molekulával, ettől teljesen függetlenül C molekula reakcióba léphet D molekulával, és így tovább, ahogyan a 1. ábra is szemlélteti.

A molekulák lehetnek aktívak és passzívak, az aktív molekulák a műveletek a passzívak pedig az adatok egy oldatban. Ha egy oldat tartalmaz aktív molekulát, akkor azt mondjuk, hogy az oldat is aktív és egészen pontosan addig aktív, míg vannak benne reakcióképes molekulák. Ha nincs több ilyen, akkor azt mondjuk, hogy az oldat elérte a stabil állapotát, azaz inertté vált, tartalma pedig a számítás eredménye(3).

¹ Denial of Service, azaz szolgáltatás megtagadás. A DoS támadás lényege, hogy annyira lefoglaljunk valamit, hogy ne legyen képes ellátni a feladatát.

² Sudoku is “a denial of service attack on human intellect.”



1. ábra - Párhuzamosan végbemenő kémiai reakciók

2.1 A kémiai kalkulus

A fentebb vázolt elképzelés természetesen működésképtelen egy precíz végrehajtási modell nélkül. A kémiai számítás matematikai alapját a γ -kalkulus teremti meg, ahhoz hasonlóan, mint a funkcionális paradigmaét a λ -kalkulus. A γ -kalkulus alapvető adatszerkezete a multihalmaz³ (multiset), mely a kémiai oldatot jelenti, ez foglalja magában az adatok és műveletek molekuláit (4).

A γ -absztrakció egy reakcióra képes molekula, ami felhasznál egy molekulát és előállít egy újat. A végrehajtás egyszerre tekintető a kémiai reakciók Brown-mozgásához⁴ hasonlóan és a multihalmazok magasabb rendű, asszociatív és kommutatív újraírásaként (4). A γ kifejezéseket, azaz molekulákat a multihalmazban az asszociatív és kommutativitást jelentő „,” jel választja el(4).

A γ -kalkulus több hasonlóságot mutat a λ -kalkulussal, mivel mindkettő a szabályok magasabb rendű újraírása. De míg a λ kifejezések fa-szerű struktúrájúak, addig a γ kifejezéseknek multihalmaz-szerű, asszociatív és kommutatív viselkedést kölcsönöz a „,” operátor(4).

2.2 Magasabb-rendű kémiai nyelv

A magasabb-rendű kémiai nyelv (Higher-Order Chemical Language, HOCL) a magasabb-rendű kémiai modellt leíró gamma kalkulus megvalósítása, és mint ilyen legfontosabb szerepe, hogy egy jól használható és olvasható nyelvi eszközkészletet biztosítson. Mindezt a nyelv egyetlen utasítás-kifejezéssel teszi meg, ami a **replace P by M if C in <>**, ahol P a minta (pattern) amit le akarunk cserélni az M-mel jelölt kifejezésre, a C az a feltétel (condition) aminek teljesülésekor a csere megtörténik, és a <> jelek között pedig az oldalon elhelyezkedő adatok vannak(5).

³ Olyan halmaz, amelyben egy elem többször is szerepelhet.

⁴ „A Brown-mozgás a gázokban és folyadékokban lebegő (szuszpendált) részecskék szüntelenül zajló, véletlenszerű mozgása.” (31)

A kémiai nyelvben a molekula valamilyen érték, pár vagy oldalt. A molekulákat vesszővel választjuk el egymástól. Az oldatok bármilyen molekulát, akár újabb oldatot is tartalmazhatnak, míg a párok olyan speciális oldatok, amiknek csak két eleme lehet, s ezek sorrendje is kötött. Azonban arra is igaz az, ami az oldatokra, hogy tartalmazhatnak további oldatokat vagy párokat, így az oldatok tetszőleges „mélységben” egymásba ágyazhatók.

2.3 Prímszám keresés kémiai nyelven

Példaként a kémiai nyelv működésére bemutatom, hogyan lehet egy tartományból kiszűrni azokat a számokat, amik nem prímek. Eredményként az oldatban csak prímszámok maradnak, így az algoritmus prímszítaként működik. Ennek HOCL kódja látható az 1. forráskódban.

```
chemprimesieve = replace X,Y by Y if (X=1) or ((X mod Y=0) and (Y!=1)) in  
  <1, 2, 3, 4, 5, 6, 7, 8, 9, 9, chemprimesieve>
```

1. forráskód - Kémiai prímszita

Mivel az illesztés során x és y bármilyen szám lehet nem elég a két szám maradékát vizsgálni, arról is meg kell bizonyosodni, hogy y nem 1, mert bármely N -re igaz, hogy $N \bmod 1 = 0$, és ez elrontaná az eredményt. Ha pedig x értéke 1, akkor a feltétel további részének vizsgálata nélkül x , y molekulák lecserélhetők y -ra, azaz x eltávolítható az oldatból. Amennyiben előre garantált lenne, hogy az oldatban nem lehet 1-es, akkor ezekre az óvintézkedésekre nem lenne szükség, de a megoldás így teljesebb. A szabálynak, ami szintén egy molekulaként jelenik meg a modellben is benne kell lennie az oldatban, ezért van benne a „chemprimesieve” molekula, ami magát az eljárást reprezentálja.

Ezen kívül az algoritmus és a kémiai modell tulajdonságait ismerve észrevehető, hogy az oldalt inertté válása után minden prímszám egyszer fog szerepelni, továbbá belátható, hogy az eredményhalmaz elemei csak abban az esetben lesznek prímek, ha a számok 1-től folytonosan szerepelnek, különben csak annyit tudunk meg róluk, hogy nem volt osztójuk az oldatban. Ilyenre példa, ha a 3-as nem szerepelne az oldatban, ekkor a 9-es bent maradna.

Ám ebből a példából már látható, hogy a kémiai modellben megírt algoritmus felépítéséhez a szokásostól eltérő gondolkodásra van szükség.

3 Sudoku

A sudoku egy részlegesen kitöltött 9x9-es mátrix, ahol a probléma ennek teljes kitöltése oly módon, hogy egy-egy szám csak egyszer szerepelhet minden sorában, oszlopában és a 9 darab 3x3-as részmátrixban is. A dolgozat során sudokun mindig a 9x9-es változatot értem, az ettől való eltérést külön jelzem, ilyen eltérő méretű a 16x16-os vagy 25x25-ös.

Általánosítva egy N-ed rendű sudoku $N^2 \times N^2$ -es mátrix, ami N területre oszlik, és 1-től N^2 -ig kell feltölteni a számokkal (6). Az általánosított sudoku NP teljes probléma⁵ (7).

És pont ez, a részleges kitöltöttség adja a lényegét, egy üres mátrix feltöltése ugyanis a sudoku szabályainak megfelelően sem jelent kihívást. Felmerül a kérdés, hogy ha egy üres tábla nem jelent kihívást, viszont egy kulcsokkal, „segítségekkel” teletűzdelt pálya igen, akkor hol a határ? Mettől sudoku egy sudoku? A válaszhoz előbb azt kell tisztázni miért is fontos ez. Ha túl kevés kulcsot tartalmaz a feladvány, akkor több lehetséges megoldása is létezik. Ezért sem jelent problémát egy üres tábla kitöltése, mert a 6.670.903.752.021.072.936.960, azaz $\sim 6,67 \cdot 10^{21}$ szabályoknak megfelelő kitöltés (8) közül bármelyik „jó” és némelyik olyan egyszerű, hogy néhány szabály betartásával mechanikusan felírható. Egy sudoku feladvány attól lesz nehéz, hogy általában csak egy megoldása van és az előre megadott kulcsok segítségével, vagy épp az ezek okozta korlátozások miatt pontosan ezt kell megtalálni.

Egészen a közelmúltig nyitott volt az a probléma, hogy mennyi az a legkevesebb kulcs, ami még egyértelműen meghatározhat egy sudoku feladványt. Ezt a határszámot, amitől még egy feladvány csupán egy megoldással rendelkezik Dublinban találták meg, és nem más, mint a 17(9), egy ilyen feladvány látható a 3. ábrán.

	3	8				2	4	
1	9			6			7	
2	5		8		3			
	8			7	6			1
5	7						8	6
3			1	8			9	
			9		7		2	8
	2			5			6	7
	1	3				9	5	

2. ábra – Egy sudoku feladvány, a továbbiakban a példák erre vonatkoznak

⁵ Az NP-nehez problémák esetében a paraméterek növekedésével a megoldáshoz szükséges idő nem lineárisan, hanem exponenciálisan nő. Az NP-nehez problémáknak nincs ismert polinom idejű megoldásuk.

			8		1			
						4	3	
5								
				7		8		
						1		
	2			3				
6							7	5
		3	4					
			2			6		

3. ábra - Egy 17 kulcsos sudoku feladvány (9)

3.1 Kitöltési stratégiák

Egy sudoku megoldása során különféle stratégiákat alkalmazunk, még ha nem is vesszük ezt észre. Ebben a szakaszban felsorolom ezeket a stratégiákat azzal a céllal, hogy a megoldó algoritmus(ok) működése érthetőbb legyen.

3.1.1 Triviális lépés

Triviális lépés akkor fordul elő, amikor egy sorban, oszlopban, vagy területen nyolc cella már ki van töltve, így a kilencedik beírása különösebb nehézség nélkül megoldható. Sőt olyan rendkívül egyszerű feladványok is léteznek, amik kizárólag ilyen triviális lépésekkel is megoldhatók. Ilyenre példa a 4. ábra feladványa, ahol a 81-ből 21 cella üres.

1	9	6	3		5	4	8	2
4	2	7		1	8	3	5	6
	5	3	2		4		7	9
6	1	2	7	4		8	3	5
9	4	8	5		1			
	7	5		8	2		4	1
5				2	6	7	9	
7	8	9	1	5		2	6	
	6	4	8	9	7	5	1	

4. ábra - Kizárólag triviális lépésekkel is megoldható feladvány (10)

3.1.2 Kizárás

Az egyik legalapvetőbb stratégia, hogy kiválasztok egy számot, ami viszonylag sokszor szerepel, ezek kizárnak sorokat, oszlopokat és területeket (3x3-as részmátrixokat) míg egy cellába egyértelműen be nem írható egy a szám(11).

3.1.3 Kiválasztás

Kiválasztok egy minél jobb területet, vagyis amiben minél több szám már a helyére került és a fennmaradókat a környező sorok és oszlopok beírt értékeinek segítségével tovább szűkíttem, míg egyértelműen be nem lehet írni egy számot (11).

3.1.4 Következtetés

Ennek a stratégiának a lényege, hogy nem egy szám pontos helyét, hanem csak annak sorát vagy oszlopát próbáljuk kikövetkeztetni és ezt az információt felhasználni az előbbi stratégiák kombinációjával, hogy végül újabb szám kerüljön a helyére (11).

3.1.5 X-szárny

Ha a fenti stratégiák egyike sem képes már újabb szám helyét egyértelműen meghatározni, akkor sem feltétlenül kell a vak-szerencsére bízni a következő lépést. Vannak még olyan speciális helyzetek, amikben biztos a lépés helyessége, ilyen az X-szárny (X-Wings) is (lásd 5. ábra).

Minden mezőhöz meghatározható, hogy milyen számok írhatók bele, ha a fenti stratégiák valamelyike még működne, akkor a meghatározás után azt tapasztalnánk, hogy van olyan cella, amibe csak egy szám írható, tehát értéke egyértelműen meghatározható. Ebben az esetben ilyen lehetőség már nincs, minden cellához legalább két szám tartozik, viszont ezekből még mindig lehet találni olyan mintákat, mik segítenek biztos lépést tenni.

Ilyen minta az X-szárny is, ami akkor alakul ki, amikor két vonalban (sor, oszlop) két szabad hely van egymás alatt és mindbe beírható ugyanaz a szám(12), az 5. ábra esetében ez a hatos. Az X-szárny feloldása pedig a következő: mivel mind a négy helyre beírható a hatos, ezek a helyek egymástól függenek, egy zárt függőségi kört alkotnak. Ha ezekből bármelyik helyre beírok egy hatost, az kizárja a sorában és oszlopában levő két helyet és csak a vele átellenesen levő marad szabad a hatos számára. A nem szomszédos csúcsokat összekötve pedig egy X-et kapunk és pont ez az, amiről a módszer a nevét kapta(12).

9	_{8 6}	₂	₂	5	1	7	3	_{8 6}
1	_{4 6}	7	3	9	8	2	_{4 6}	5
5	_{4 3 8}	_{2 3}	₂	7	6	_{4 8}	9	1
8	1	_{6 9}	7	2	4	3	5	_{6 9}
2	_{4 3}	_{4 3 9}	1	6	5	_{8 9}	_{4 8}	7
_{4 6}	7	5	9	8	3	_{4 6}	1	2
_{4 6}	2	1	5	3	7	_{8 6 9}	_{4 6 8 9}	_{4 6 8 9}
7	5	8	6	4	9	1	2	3
3	9	_{4 6}	8	1	2	5	7	_{4 6}

5. ábra - X-szárny helyzet (12)

3.1.6 Kardhal

A kardhal (Swordfish) az X-szárnyhoz (3.1.5. fejezet) hasonló minta, de valamivel bonyolultabb annál, hat vagy hosszabb zárt függőségi láncot jelent (lásd 6. ábra és 7. ábra). A megoldáshoz hasonlóan járhatunk el, mint az X-szárny esetében. Kiválasztunk egy cellát és beleírjuk a megfelelő értéket, innentől kezdve a zárt függőség miatt ez már egyértelműen meghatározza, hogy melyik cellába lehet még azonos értékeket beírni(13). A hatsúcsos (6. ábra) esetében vagy a zöld, vagy a narancsszínű cellákba írhatók négyes számok, a nyolcsúcsos (7. ábra) esetében pedig a zöld vagy a piros cellák tölthetők föl hármas értékekkel.

1	9	5	3	6	7	2	4	8									
↓ ²	7	8	↓ ^{1 2}	5	↓ ¹	3	6	9									
3	↓ ²	6	↓ ²	9	8	1	5	7		↓							
↓ ^{2 6}	↓ ^{1 2}	3	7	8	↓ ¹	5	9	↓ ²									
7	↓ ^{1 2}	9	↓ ^{2 3}	5	↓ ^{2 3}	6	7	1		↓							
5	8	4	9		6	7	1	↓ ^{2 3}									
8	3	2	5	4	9	6	7	1									
9	↓ ⁴	7	↓ ⁸	1	3	↓ ⁸	2	5		↓							
↓ ^{4 6}	5	1	↓ ⁶	7	2	9	↓ ³	↓ ³									

6. ábra - Hatsúcsos kardhal 4-es számra és a feloldása (13)

	↓ ³				↓ ³												
	↓				↓												
					↓ ³			↓ ³									
	↓ ³			↓ ³				↓									
				↓				↓									
				↓ ³				↓									
								↓ ³									

7. ábra - Nyolcsúcsos kardhal 3-as számra (14)

Mind az X-szárny, mind a kardhal biztos megoldást jelenthet adott esetekben, ha a kizárás (3.1.2), kiválasztás (3.1.3), következtetés (3.1.4) már nem működik, azonban a megoldó algoritmusok ezekre legtöbbször ezekre nincsenek felkészítve a bonyolultságuk miatt.

3.1.7 Típpelés

Ha a fenti stratégiák egyike sem képes már újabb szám helyét egyértelműen meghatározni, legfeljebb szűkíteni a lehetőségeket, akkor használható megoldás a típpelés. A megmaradt lehetősége közül választunk egyet úgy, hogy helyessége nem megerősített. Abban az

esetben, ha rosszul választottunk valószínűleg a sudokut nem sikerül majd megoldani és a hiba kijavításához vissza kell vonni minden olyan lépést, amit a tippelés után tettünk.

4 Megoldási módszerek

Ebben a szakaszban felsorolok néhány algoritmus tulajdonságot, amelyek vizsgálati szempontok lesznek a bemutatott megoldó módszerek esetében.

4.1 Rekurzió

Rekurciónak hívjuk azt a módszert, amikor egy függvény önmagát meghívja, ahhoz hogy ez működjön a problémára meg kell találni azt a legegyszerűbb esetet, amikor a megoldás magától értetődő és találni kell egy olyan véges, ismétlődő egyszerűsítési folyamatot, amellyel eljuthatunk ehhez a legegyszerűbb esethez, és minden következő lépésnél feltesszük, hogy az már ez a legegyszerűbb eset lesz(15).

```
int faktorialis(int n)
{
    if(n <= 1)
        return 1;
    return n * faktorialis(n-1);
}
```

2. forráskód - Faktoriális számítás rekurzív implementációja C nyelven (16)

A 2. forráskódban látható egy egyszerű példa a rekurzív függvényekre. A függvénytörzs első sora vizsgálja, hogy a legegyszerűbb eset bekövetkezett-e, ha igen a következő sorban ott a megoldás rá, a harmadik sorban pedig n csökkentésével haladunk a legegyszerűbb eset felé.

4.2 Brute force

A brute force (nyers erő) alapú megoldások lényege, hogy nem próbálja bonyolult stratégiákkal szűkíteni a keresési teret, hanem végig próbálja annak minden elemét, míg meg nem találja a megoldást. A megoldás előnye, hogy a megoldáshoz szükséges idő nem függ a feladvány nehézségétől (17), viszont jelentősen függ a keresési tér nagyságától.

Képzeljünk el egy számszörös lakatot, ha 3 számjegyet használ, 1.000 lehetséges kombináció létezik, ennek végigpróbálásához T idő kell. Ha 4 számjegyet használna, 10.000 kombináció lenne, és ezt végig próbálni 10T ideig tartana. A sudoku esetében a lehetséges kombinációk száma $\sim 6,67 \cdot 10^{21}$. Belátható, hogy a brute force algoritmus nem egy hatékony algoritmus, viszont garantáltan megtalálja a megoldást és a kérdés csak az, hogy érdemes-e megvárni.

4.3 197 karakteres Python kód

Ez az egyik legrövidebb kód, amit találtam, de ebben persze benne vannak a nem túl beszédes, egykarakteres metódus és változónevek illetve a szóközők és sortörések mellőzése onnan, ahonnan csak lehet. A 3. forráskódban lényegi változtatás nélkül közlöm a tippelés mentes, rekurzív sudoku megoldó algoritmust, de beillesztettem a feladványt reprezentáló karakterláncot. A 4. forráskódban a könnyebb értelmezhetőség miatt kicsit áttördeltem a kódot.

```
def S(s): i=s.find('0')
    if i<0:print s;return
    for v in '123456789':
        if sum(v==s[j]and(i/9==j/9or i%9==j%9or(i%9/3==j%9/3and i/27==j/27))for j in
range(81))==0:S(s[:i]+v+s[i+1:])
S('0300010000060000505000009830800063020000500009038000607140000090200008000004000
30')
```

3. forráskód - 197 karakteres sudoku megoldó Python⁶ nyelven (18)

```
def Solve(input):
    i=input.find('0')
    if i<0: print input; return
    for value in '123456789':
        if sum(value ==input[j] and ( i/9==j/9 or i%9==j%9 or (i%9/3==j%9/3 and
i/27==j/27) ) for j in range(81) ) == 0:
            Solve(input[:i]+ value +input[i+1:])
Solve('030001000006000050500000983080006302000050000903800060714000009020000800000
400030')
```

4. forráskód - (18) módosított változata

A kód bemenetként egy karakterláncot kap, ami 81 számot tartalmaz, a sudoku mátrixot soronként összeillesztve, az üres helyeket pedig nullával jelöli. Működése pedig a következő: első lépésben keres egy üres cellát, ha nem talál, akkor kilép, befejezte a működését. Utána következik a beilleszthetőségi vizsgálat, ha a kiválasztott üres cellába illeszteni kívánt szám nem ütközik a sorában, oszlopában és a területében akkor mehet az illesztés. Az illesztésről azt kell tudni, hogy Python nyelvben a karakterlánc nem változtatható, ezért veszi a kijelölt helyig tartó részét, hozzákapcsolja a beillesztendő számot és a kijelölt hely utáni karaktertől a karakterlánc végéig tartó részt.

A vizsgálat érdekes módon egy összeg képzésével, a „sum” függvény segítségével valósul meg. Lényege, hogy összeadja a kiválasztott cella sorában, oszlopában és 3x3-as területében található azon számokat, amik az éppen beillesztésre kiválasztott számmal megegyeznek. Ha ez az összeg nulla, akkor nem volt ilyen, tehát nem volt ütközés és a szám bírható, majd pedig a módosult karakterláncra újra meghívja önmagát az eljárás.

Az eredeti kódot iterációról iterációra elemezve arra jutottam, hogy a megoldás megtalálása után nem áll le azonnal az algoritmus, hanem hibás számítási ágakkal tovább dolgozik, és ez jelentős időtöbbletet jelent. Ha itt megszakítanánk az algoritmust, akkor körülbelül a felére csökkenne a végrehajtási idő, bár ez nem lenne valami elegáns eljárás.

4.4 Perl háromsoros

Egy másik (5. forráskód) igen tömör megoldás Perl nyelven született és csupán háromból áll igaz, ehhez olyan fokú zsugorítás kellett, hogy teljesen olvashatatlaná teszi a kódot. Ezt részletesen nem elemzem, annyit azonban megjegyeznék, hogy ez is egy rekurzív megoldás. A szerző részletesen taglalja a működését és olvasható formában is közli a honlapján (19).

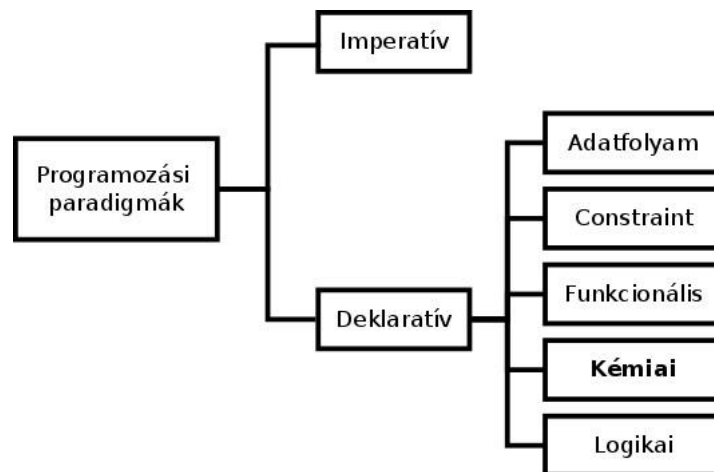
⁶ Python 2.7.x verzióval hibátlanul lefutott, de a 3.x-es verzióval nem.

```
use integer;@A=split//,<>;sub R{for$i(0..80){next if$A[$i];my%t=map{$_/9
==$_/9||$_%9==$_%9||$_/27==$_/27&&$_%9/3==$_%9/3?$A[$_]:0=>1}0..80;R($A[
$i]=$_)for grep{!$_}1..9;return$A[$i]=0}die@A}R
```

5. forráskód - Perl háromsoros (19)

4.5 Deklaratív megoldások

A deklaratív paradigma lényege, azt írjuk le, hogy *mit* kell a programnak tenni ahelyett, hogy azt mondanánk meg, hogy *hogyan* tegye, a hogyan ebben az esetben a nyelv implementációjára marad(20). A kémiai modell is a deklaratív paradigmába tartozik, ezt szemlélteti a 8. ábra.



8. ábra - Kémiai modell elhelyezése a paradigmák között(20)

Egy logikai paradigmájú, Prolog nyelvű megoldás látható a Wikipédián (21), illetve egy másik logikai megoldás a (22)-ben. Funkcionális megoldásra, azon belül is Haskell nyelvűekre pedig összegyűjtöttek egy csokorral a <http://www.haskell.org/haskellwiki/Sudoku> oldalon, csoportosítva tulajdonságaik szerint, némelyikhez pedig részletes értelmezés is tartozik.

4.6 Megoldó módszerek természeti analógián

Egyes megoldásokat valamilyen természeti analógia ihletett, ilyen a megoldást mézelő méhekkal kereső (23) módszer, a szimulált hűtés alapú megoldás (24) de ide tartozik a kémiai modell is.

4.6.1 Mesterséges méh kolónia algoritmus

A mesterséges méh kolónia algoritmus (Artificial Bee Colony Algorithm) (25) a mézelő méhek élelemkeresési módszerét utánözva ad egy természeti analógián alapuló optimalizációs algoritmust. A Scholarpedia cikke így foglalja össze az algoritmus lényegét:

Az algoritmus a méhek három csoportját tartalmazza, dolgozókat, akik egy adott élelemforráshoz tartoznak, figyelőket, akik a dolgozók méhtáncát⁷ figyelik és a felderítőket, akik véletlenszerűen kutatnak élelemforrás után. Kezdetben a felderítők összegyűjtik az

⁷ „A kaptárba visszatérő felderítő méh a tánc segítségével tudatja fajtársaival az általa talált táplálékforrás minőségét, irányát és távolságát.” (34)

élelemforrásokat, aztán a dolgozók elkezdik kitermelni a nektárt, míg mindet össze nem gyűjtik, és az a dolgozó, akinek az élelemforrása kiapadt felderítő lesz.

Az algoritmusban az élelemforrások jelképezik a probléma lehetséges megoldásait, a lelőhely nektármennyisége pedig az adott megoldás jóságát mutatja. A dolgozók száma pontosan egyenlő a lelőhelyek számával, mivel egy méh csak egy forráshoz tartozhat.

Az algoritmus véletlenszerűen talál egy megoldást, majd iterációk során folyamatosan javítja azt és halad egy jobb megoldás felé a szomszédos keresési mechanizmusnak köszönhetően, ami kiejti a gyengébb megoldásokat (26).

4.6.2 Szimulált hűtés

A szimulált hűtés (Simulated Annealing, SA) által kiindulási alapnak tekintett fizikai folyamat két részből áll. Először olvadáspontig melegítjük a fémeket, ekkor ugyanis a részecskék elrendeződése véletlenszerű lesz (27). Majd lassan lehűtjük annyira, hogy „megfagy”, vagyis onnantól kezdve nem történik további változás. A rendszer ekkor éri el a minimum energiaszintet, s a megféleltetés révén ekkor kaptuk meg a keresett, globális minimumértéket (28).

Alapállapotban a részecskék szigorú rácsszerkezetben vannak és az energiaszintjük minimális. Felmelegítés során az energiaszint akkor éri el maximális szintet mikor a fém megolvadt. Megolvasztás után a minimális energiaszintet csak akkor éri el ismét, ha a hűtési folyamat kellően lassú. Különben metastabil állapotba kerül, azaz egy többletenergiát hordozó állapotba, ami a megféleltetés alapján egy lokális optimum(27).

A fizikai jelenség megféleltetése az algoritmus számára a következő: Az optimalizációs probléma megoldásai jelképezik a fizikai rendszer állapotait, míg a megoldás „jósa” az állapot energiájának felel meg (27).

5 Kémiai megoldás

Ebben a fejezetben bemutatom a kémiai megoldás során alkalmazott adatrepresentációt, a megoldó algoritmus működését és korlátait.

5.1 Adatok kémiai reprezentációja

A kémiai modellben nincsenek a hagyományos értelemben vett adatstruktúrák (például tömbök, listák), így külön problémát jelent az adatok reprezentációja egy olyan feladat során, ahol fontos információtartalommal bír már az adatok egymáshoz viszonyított elhelyezkedése is. A kémiai modellben molekulák és oldatok vannak, illetve a pár, ami egy olyan speciális oldat, aminek csak két eleme lehet.

Az oldatok multihalmazok, olyan adatszerkezetek, ahol az elemek még csak sorrendiséggel sem rendelkeznek, annyi információ áll rendelkezésre egy molekuláról, hogy melyik oldatban található. A kémiai paradigmától szintén idegen, hogy egy multihalmaz (oldat) elemeinek a számát lekérdezzük, különösen igaz ez, ha egy adott értékű elem számáról van szó. Például

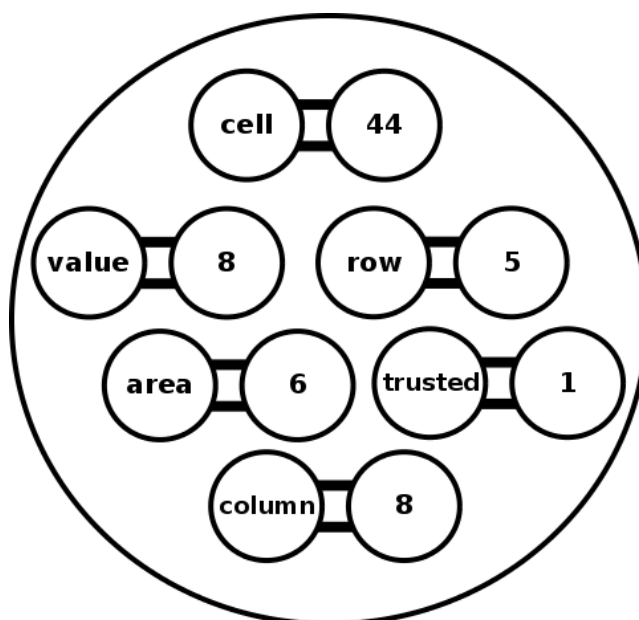
az üres cellákat reprezentáló molekulák számát szeretnénk megtudni, ami pedig belátható, hogy fontos információ lehet a mátrix kitöltésénél.

Ahhoz tehát, hogy megállapíthassam, hogy egy cellába mely számok írhatók be tudnom kell róla, hogy melyik sorba, oszlopba és területbe tartozik. Ezeket akár ki is lehetne számolni minden alkalommal, de én explicit beleírtam a cellát reprezentáló oldatba, lásd 9. ábra és 6. forráskód.

Egy cellát egy oldat jelképez, amiben a következő információkat hordozó molekulák vannak: (i) azonosító (szám 1-81-ig), (ii) a cella értéke, (iii) a sor, amibe tartozik, (iv) az oszlop, amibe tartozik, (v) a terület, amibe tartozik, és végül (vi) egy állapotjelző, ami azt jelzi, hogy az adott érték előre magadott volt-e. Ezek az információk párokban vannak tárolva, ahol a pár két molekuláját kulcs-érték párként használtam az információk megjelenítésére.

Találtam egy megoldást, ami a feladványhoz értékek megjelölését egy érdekes módon oldja meg nevezetesen, hogy e cellák értéke negatív (29). Ekkor persze a számítások előtt abszolút értékét kell venni a cellának, viszont egyetlen összehasonlítással megállapítható, hogy a szám az alap feladvány része volt-e. Ezt a módszert a dolgozatom során is használhatnám, megtakarítanék vele egy-egy párt a cellákat reprezentáló oldatban, cserébe abszolút értékekkel kellene végeznem a számításokat.

Ezen kívül szükséges volt még tárolni, hogy egy-egy csoport (sor, oszlop, terület) milyen számokat tartalmaz illetve, hogy hány üres cella található benne. Ennek megoldását a prímkódnak keresztelt megoldással oldottam meg, amit az 0. szakaszban ismertetek.



9. ábra - Egy cellát leíró oldat, benne az tulajdonságait megadó párokkal

<cell:44, value:8, row:5, column:8, area:6, trusted:1>

6. forráskód - Egy cellát leíró oldalt HOCL kódja

5.2 Prímkód

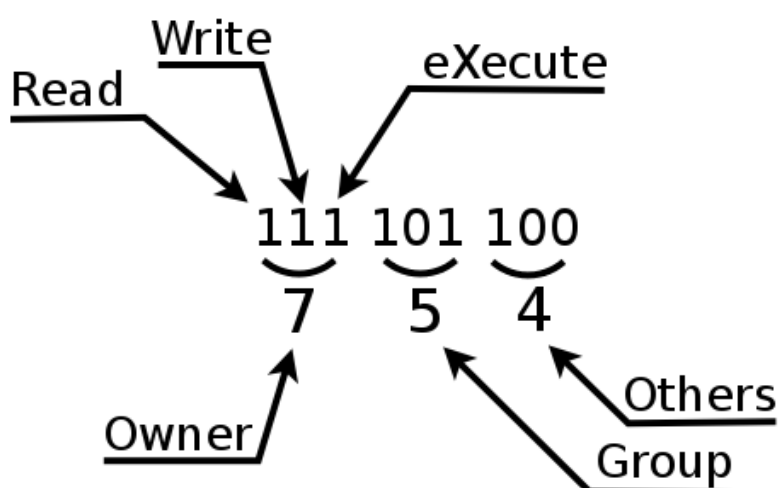
A sudokuhoz számokat használunk 1-től 9-ig, de igazából a jeleknek nincs jelentőségük, tetszőleges eltérő kilenc karaktert, jelet vagy kódot használhatunk az egyes értékek megkülönböztetésére, ettől a szabályok és a megoldás menete nem változik. Ezért a dolgozatban használt megoldásban le is cseréltem őket, úgy hogy az első kilenc prímszámot használom a tartalom megkülönböztetésére az 1. táblázatnak megfelelően.

szám	kód	szám	kód	szám	kód
1	2	4	7	7	17
2	3	5	11	8	19
3	5	6	13	9	23

1. táblázat - A megoldásban használt szám-kód párok

Miért is volt erre szükség? Azért, mert a megoldás során szükséges tudnom, hogy egy adott cellába milyen számok írhatók be és milyenek nem. A HOCL egyetlen mintaillesztésen alapuló csere (replace) utasításával viszont nem volt lehetséges, hogy összegyűjtssem a kiválasztott cella sorába, oszlopába és 3x3-as területébe tartozó már beírt számokat. Nem értelmezett az oldat számossága, a „tartalmaz-e” művelet, mellyel megállapítható lenne, hogy egy oldat tartalmazz-e az adott számot (például a Java nyelv magas szintű láncolt lista megvalósításában egy hasonló ellenőrzés egyetlen metódushívással megoldható).

Olyan megoldásra volt tehát szükségem, ami könnyen, egyetlen művelettel képes szolgáltatni ezt az összetett információt. Az ötletet az unix rendszerekben fájl jogosultságok leírására használt módszer (lásd 10. ábra) adta. Persze a bináris „flag” megoldás sem volt tökéletes, mert maszkolásra van szükség hozzá, amihez pedig nyelvi támogatás kell.



10. ábra - Unix rendszerek jogosultság leírása flagekkel

Hasonló hatást lehet viszont elérni a prímszámokkal, és ehhez csak szorzásra és osztásra (maradékképzés) van szükség. A prímkód működését egy példán keresztül mutatom be.

Vegyük az 9. ábra - Egy cellát leíró oldat, benne az tulajdonságait megadó párokkal látható cellát, sorszáma 44, és értéke 8, amit a 1. táblázatban látható módon 19-ként ábrázolok. Ez a cella az 5. sorba tartozik, amiben jelenleg a következő számok vannak: 5, 6, 7, és 8 (lásd 2. ábra). Az 5. sor foglaltság jelzéséhez e számok prímkódjainak szorzatát használom, azaz a $11 \cdot 13 \cdot 17 \cdot 19 = 46.189$ értéket.

Ahhoz, hogy ellenőrizsem, hogy ebben a sorban benne van-e a 8-as, csak a $46.189/19$ hányados maradékát kell vizsgálnom. Ha nulla, tehát osztható 19-el, akkor a sor tartalmazza a 8-ast, különben nem. Mindezt pedig a prímszámok természete biztosítja számomra, miszerint egy prímszám csak egyel és önmagával osztható. Több prímszám szorzata pedig csak az adott prímszámokkal lesz osztható.

De ugyanezt használom az üres helyek megállapítására is. Minden csoportban 9 cella van, amiket beszámozok (N. sor 1., 2., ... 9. eleme), társítom hozzá a prímszámokat, és ha például az 5. sor 3., 4., 5., 6., és 7. helye üres, akkor az 5. sorhoz tartozó üres helyeket (lásd 2. ábra – Egy sudoku feladvány, a továbbiakban a példák erre vonatkoznak) jelző kód 85.085 lesz.

Önmagában persze nem a prímkód jelenti a kémiai megoldás lényegét, ez csupán egy összetevője, ami azért volt szükséges, mivel a molekuláknak egy oldatban nincs sem sorszáma, sem pedig sorrendjük és közvetlenül nem is érhetők el. Maga a prímkód alkalmazható bármely más elvű módszer esetében is, hogy jelent-e előnyt vagy hátrányt az alkalmazása más elvű megoldás során az egy külön vizsgálat tárgya lehetne.

5.3 Kémiai algoritmus

A kémiai modell egy indeterminisztikus végrehajtású számítási modell, emiatt két különböző futtatás ugyanazon az adatokon is két különböző lépéssorozatként érhet véget. Éppen emiatt nem mondható meg, hogy melyik szám melyik után kerül beillesztésre, ahhoz hasonlóan mintha egy ember egymás után kétszer töltené ki ugyanazt a sudokut, de már nem emlékezne (és nem is akarna emlékezni) arra, hogy milyen lépések sorozatával haladt, így valószínűleg két eltérő úton jut majd el a megoldáshoz. Ez szöges ellentétben áll az imperatív nyelveken adott megoldással, ahol a programozó által előre megadott utasítások sorozataként áll össze a megoldó algoritmus, ennek megfelelően a kitöltés azonos sorrendben megy végbe (természetesen kivétel, ha ezt szándékosan meggátolták az algoritmus véletlenszerűsítésével).

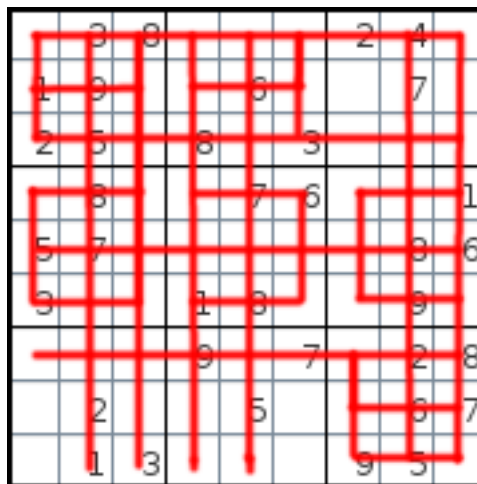
A kémiai algoritmus működése a következő lépéssorozatban foglalható össze:

1. A megoldó algoritmus az oldatban szabadon levő molekulák közül elkap egy számot a még fel nem használt számok halmazából.
2. Ezt kövően elkap egy üres cellát is.
3. Ellenőrzi, hogy a kiválasztott szám beírható-e a kiválasztott cellába (nincs-e azonos szám a cella sorában, oszlopában vagy területében).
4. Ha beírható, akkor a cella értéke a kiválasztott szám lesz és törli a számot a fel nem használtak halmazából, továbbá frissítésre kerülnek a beírhatóság megállapításához

használt értékek (a prímkód miatt ez csupán egy szorzást jelent a kiválasztott számmal).

5. Ha nem volt beírható, akkor pedig új szám és cella páros kerül kiválasztásra.

Ezen algoritmus nem garantálja, hogy a szám globálisan is megfelelő helyre kerül, csupán az aktuális állásnak megfelelően egy „nem feltétlenül rossz” helyre helyezi a kiválasztott számot. Vegyük a példafeladványt és tegyük fel, hogy a kiválasztott szám, ami be akarunk írni a nyolcas. Ekkor kizárjuk (3.1.2. fejezet) a már beírt nyolcasokkal azokat a cellákat, ahova biztosan nem kerülhet nyolcas. Az eredményt a 11. ábra mutatja, amin jól látszik, hogy öt olyan cella marad ahova a jelenlegi (a kezdeti) felállítás szerint nyolcast lehet írni.



11. ábra - Nyolcasokra lefedett tábla

A fent vázolt kémiai algoritmus számára ezek a cellák egyenértékűek, így véletlenszerűen választ közülük, ennek következtében 0,2 a valószínűsége, hogy (ebben a lépésben) biztosan jó cellába kerül a kiválasztott nyolcas. Összességében viszont az öt lehetséges hely közül háromba nyolcas való, így a lépés helyességének valószínűsége 0,6-ra módosul, vagyis annak kisebb a valószínűsége, hogy a lépés hibás lesz. Természetesen ez csak a példában vázolt lépésre igaz, ebből az összes lépésre általánosítani nem szabad.

Abban az esetben pedig, ha egy szám rossz helyre került, előbb vagy utóbb egy másik számot nem lehet majd beírni. Ez úgy vehető észre, hogy a fel nem használt számok halmazában lesz még szám, ám nem lesz hozzá a beírási szabályoknak megfelelő szabad cella.

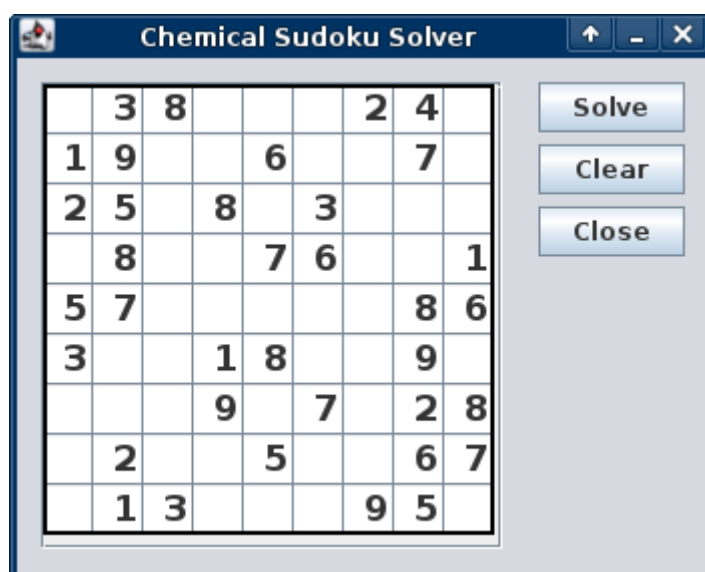
Ezután a következőket lehet tenni:

1. Törölöm az összes algoritmus által beírt számot, visszatérek az elejére és előlről kezdem a megoldást és újfent csak remélem, hogy ezúttal minden a helyére kerül (ehhez szükséges a feladványt jelentő kulcsokat megjelölése).
2. Pontosítom a beírási szabályokat, hogy több azonos lépés közül a biztosat válassza és követem is ezeket a lépéseket, hogy egy visszavonás során csak azokat töröljem, amik tippeléssel kerültek a helyükre.
3. Javítási kísérlet nélkül feladom, és a „Nem sikerült megoldani.” üzenettel térek vissza.

A második bonyolultabb illesztési szabályokat jelent, viszont megszabadít az elsőnél fennálló, igen összetett karbantartási feladatokról, amik az után lépnek fel, hogy töröltem a számokat. Továbbá a kémiai paradigma jellegéből adódóan azt sem lehet egyszerűen megmondani, hogy a megoldó szabály mikor válik tehetetlenné, és így mikor kell a hibajavító szabályt futtatni. Illetve a harmadik lehetőség a feladás, jelenleg az algoritmus ezt teszi.

5.4 Implementációs részletek

A HOCL Interpretert Java nyelven írták(2), így a felhasználói felületet én is Java nyelven írtam, és ebbe kódba került beágyazásra az interpreter is. A grafikus felhasználói felület rendkívül egyszerű, biztosít egy beviteli táblázatot, a sudoku kezdőállapotának beviteléhez, egy gombot a táblázat törléséhez, egy gombot a megoldás indításához és egyet a kilépéshez, ahogy ezt a 12. ábra is mutatja.



12. ábra - Kémiai sudoku megoldó program felhasználói felülete

5.4.1 HOCL kód

A 7. forráskód mutatja a megoldó algoritmus HOCL kódját a korábban ismertetett **replace P by M if C** formulával és a változók a Prolog nyelv konvenciójának megfelelően nagybetűvel szerepelnek. Az főoldat tartalmát itt nem közöltem, tartalmazza a 81 darab cellát reprezentáló oldatot a 6. forráskódhoz hasonlóan, a 27 darab ütközésetektáláshoz használt oldatot (9 sor, 9 oszlop és 9 3x3-as terület) illetve a fel nem használt számokat, amiből a példafeladványban 45 darab van.

```
solvesudoku = replace <cell:X,value:1,row:R,column:C,area:A,trusted:0>,
    <row:R,value:J>, <column:C,value:K>, <area:A,value:L>, N
by <cell:X, value:N, row:R, column:C, area:A, trusted:0>,
    <row:R, value:J*N>, <column:C, value:K*N>, <area:A, value:L*N>
    if (J mod N)*(K mod N)*(L mod N)!=0
```

7. forráskód – Sudoku megoldó algoritmus HOCL nyelven

A kód maga az 5.3. fejezetben felvázolt algoritmust valósítja meg. Működése a következő: a solvesudoku aktív molekula „elkap” egy passzív cellamolekulát, tehát egy az aktív molekula reakcióba lép egy passzívvval. Ez a reakció akkor következik be, ha a passzív molekula illeszkedik arra a mintára, hogy értéke 1⁸, azaz üres cella, tartalmaz „cell”, „row”, „column” és „area” első elemű párokat és ezek második elemeinek értékét az X, R, C, és A változókba kötöm le, későbbi feldolgozás céljából. Tehát ezekre az értékekre nincs megkötés, későbbi vizsgálatok folyamán lesz rájuk szükség. Szintén lekötök egy fel nem használt számot az N változóba.

Ezeket cserélem le a következőkre: egy cellára, amiben a változás csak a value, ami felveszi N értékét, a sor, oszlop és terület oldatokra, amik értéke N-szeresére nő. Látható, hogy N nem kerül vissza! Maga a csere pedig akkor történik meg, ha $J \bmod N$, $K \bmod N$ és $L \bmod N$ értékek szorzata nem nulla. J, K, és L a prímkódokat tárolják, amik azt mutatják meg, hogy az adott sorban, oszlopban és területben milyen számok vannak. Ezen moduló művelet értéke akkor nulla, ha a prímszámok szorzataként képzett prímkód tartalmazza (szerepel benn szorzó tényezőként) a vizsgált N értéket. A három művelet szorzatát vizsgálom csak egybe nullára, mivel ha bármely tényező nulla, az egész szorzat is nulla lesz, és ez azt jelenti, hogy a szám a kiválasztott cellába nem írható be.

6 Összegzés

A dolgozat célja a viszonylag kevésbé ismert kémiai programozási modell bemutatása. A kémiai modell viszonylag új, az elmúlt két évtizedben jelent meg. Beleillik azokba a törekvésekbe, amelyek szakítani próbálnak a szinte egyeduralkodó Neumann elvű számítási modellel, amely a számítást elemi lépések sorozataként írja le. Még a napjainkban elterjedt párhuzamos és konkurens modellek is többnyire ennek változatai, többszálú megvalósításai.

A kémiai modellt és a HOCL nyelvet egy egyszerű és elterjedt, ugyanakkor megfelelő kísérletezési alapot biztosító példán, a sudokun keresztül mutattam be. A cél a kémiai modell elhelyezése más megoldási nyelvek és megoldási stratégiák között. Ezekkel összehasonlítva a kémiai modell deklaratív jellegénél fogva jóval tömörebb és érthetőbb leírást ad, ugyanakkor más deklaratív modellekhez képest transzparensen párhuzamos végrehajtást biztosít explicit nyelvi elemek nélkül.

⁸ Azért 1 az üres cella jelölése, mert az a szorzás egységeleme, így a prímkódok előállításának értékvizsgálat nélkül szorozhatok minden releváns számot, és az üres helyek egyesei nem változtatják az eredményt.

7 Irodalomjegyzék

1. *Higher-order Chemical Model of Computation*. **Banâtre, Jean-Pierre, Radenac, Yann és Fradet, Pascal**. CiteSeerX: 10.1.1.65.1446.
2. **Rajcsányi, Vilmos**. *Interpreter Higher Order Chemical Language (HOCL)*. 2011.
3. *A Generalized Higher-Order Chemical Computation Model*. **Banatre, Jean-Pierre, Fradet, Pascal és Radenac, Yann**. hely nélkül. : Electr. Notes Theor. Comput. Sci., 2006., 135. kötet.
4. *Higher-order Chemical Programming Style*. **Banâtre, Jean-Pierre, Fradet, Pascal és Radenac, Yann**. 2004.
5. **Vilmos Rajcsányi, Vilmos és Németh, Zsolt**. *The chemical machine: an interpreter for the Higher*. hely nélkül. : Springer, 2011. 978-3-642-29736-6.
6. *Solving Sudoku in Reconfigurable Hardware*. **Skliarova, Iouliia, Vallejo, Tiago és Sklyarov, Valery**.
7. *Complexity and Completeness of Finding Another Solution and Its Application to Puzzles*. **YATO, Takayuki és SETA, Takahiro**.
8. *Enumerating possible Sudoku grids*. **Felgenhauer, Bertram és Dresden, Tu**. 2005.
9. *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem*. **McGuire, Gary, Tugemann, Bastian és Civario, Gilles**. hely nélkül. : CoRR, 2012., abs/1201.0749. kötet.
10. **Brouwer, Andries E**. Solving sudokus - Baby steps. [Online] 2006. 5 31. [Hivatkozva: 2012. 11 8.] <http://homepages.cwi.nl/~aeb/games/sudoku/solving2.html>.
11. **Khan, Raza**. Sudoku: Rules and Strategies. [Online] 2006. [Hivatkozva: 2012. 11 4.] <http://www.altafkhan.com/ib/sudoku-rules-and-strategies.htm>.
12. **Tomlinson, Howard, Oakley, David és Phillips, John**. Solving Sudoku : X-Wings. *Sudoku of the day*. [Online] Astraware Limited. [Hivatkozva: 2012. 11 9.] <http://www.sudokuoftheday.com/pages/techniques-8.php>.
13. —. Solving Sudoku : Swordfish. *Sudoku of the day*. [Online] Astraware Limited. [Hivatkozva: 2012. 11 9.] <http://www.sudokuoftheday.com/pages/techniques-9.php>.
14. **guggilam, swaroop**. Sudoku Solving Techniques - Swordfish. *blogger.com*. [Online] 2011. 5 11. [Hivatkozva: 2012. 11 9.] http://swaroopg92.blogspot.hu/2011/05/sudoku-solving-techniques_9477.html.
15. Rekurzió. *www.wiki.prog.hu*. [Online] 2010. 6 23. [Hivatkozva: 2012. 11 10.] <http://wiki.prog.hu/wiki/Rekurzi%C3%B3>.
16. **Wikipédia**. Rekurzió. *Wikipédia*. [Online] 2012. 9 4. [Hivatkozva: 2012. 11 10.] <http://hu.wikipedia.org/wiki/Rekurzi%C3%B3>.
17. —. Sudoku algorithms. *Wikipédia*. [Online] [Hivatkozva: 2012. 11 4.] http://en.wikipedia.org/wiki/Sudoku_algorithm.

18. **Rushil**. Implement a Non-Guessing Sudoku Solver. <http://codegolf.stackexchange.com/>. [Online] 2011. 10 8. [Hivatkozva: 2012. 11 8.] <http://codegolf.stackexchange.com/questions/355/implement-a-non-guessing-sudoku-solver>.
19. **von der Burg, Edmund**. Sudoku solver in three lines explained. [Online] 2005. 6 2. [Hivatkozva: 2012. 11 8.] <http://www.ecclestoat.co.uk/2005/06/sudoku-solver-in-three-lines-explained>.
20. **Wikipédia**. Declarative programming. *Wikipédia*. [Online] [Hivatkozva: 2012. 11 4.] http://en.wikipedia.org/wiki/Declarative_programming.
21. —. Sudoku algorithms. *Wikipédia*. [Online] [Hivatkozva: 2012. 11 9.] http://en.wikipedia.org/wiki/Sudoku_algorithms#Example_of_a_Sudoku_solver_.28in_Prolog.29.
22. *Compiling and Executing Declarative Modeling*. **Cipriano, Raffele, Dovier, Agostino és Mauro, Jacopo**.
23. *Solving Sudoku Puzzles Using Improved Artificial Bee Colony Algorithm*. **Pacurib, Jaysonne A., Seno, Glaiza Mae M. és Yusiong, John Paul T.** hely nélkül. : IEEE Computer Society, 2009.
24. *An FPGA-based Sudoku Solver based on Simulated Annealing methods*. **Malakonakis, Pavlos, és mtsai**. 2009.
25. *Artificial Bee Colony Algorithm and Its Application to Generalized Assignment Problem*. **Baykasoulu, Adil, Özbakır, Lale és Tapkan, Pınar**. 2007.
26. **Karaboga, Dervis**. Artificial bee colony algorithm. *www.scholarpedia.org*. [Online] 2012. [Hivatkozva: 2012. 11 7.] http://www.scholarpedia.org/article/Bee_colony_algorithm.
27. *Simulated Annealing*. **Aarts, Emile, Korst, Jan és Michiels, Wil**. hely nélkül. : Springer US, 2005.
28. *Optimization by Simulated Annealing*. **Kirkpatrick, Scott, Gelatt Jr., D. és Vecchi, Mario P.** hely nélkül. : Science, 1983.
29. **Abrams, Brad**. Sudoku solver in C#. *MSDN Blogs*. [Online] 2005. 09. [Hivatkozva: 2012. 11 8.] <http://blogs.msdn.com/b/brada/archive/2005/09/25/sudoku.aspx>.
30. **Mazzocchio, Daniele**. *www.kernel-panic.it*. [Online] 2005. 9 30. [Hivatkozva: 2012. 11 9.] <http://www.kernel-panic.it/software.html#funproj>.
31. **Wikipédia**. Brown-mozgás. *Wikipédia*. [Online] [Hivatkozva: 2012. 11 10.] <http://hu.wikipedia.org/wiki/Brown-mozg%C3%A1s>.
32. —. Dancing Links. *Wikipédia*. [Online] [Hivatkozva: 2012. 11 8.] http://en.wikipedia.org/wiki/Dancing_Links.
33. *Dancing links*. **Knuth, Donald E**. 2000.
34. **Wikipédia**. Méhtánc. *Wikipédia*. [Online] [Hivatkozva: 2012. 11 7.] <http://hu.wikipedia.org/wiki/M%C3%A9ht%C3%A1nc>.

35. Visszalépéses keresés (algoritmus). *www.wiki.prog.hu*. [Online] 2010. 9 1. [Hivatkozva: 2012. 11 10.] http://wiki.prog.hu/wiki/Visszal%C3%A9p%C3%A9ses_keres%C3%A9s_%28algoritmus%29.