Debreceni Egyetem

Informatika Kar

Intervallumokon végzett számítások

Témavezető:

Dr. habil. Nagy Benedek

Egyetemi tanár

Készítette:

Bécsi Zoltán József

PTML

Debrecen

2009

Tartalomjegyzék

Bevezetés

Amikor valaki fog egy ceruzát, és húz egy vonalat a papírlapon, a figura pontokat és vonalakat tartalmaz. Szűkítsük le a teret egy 1 dimenziós vonalra. A figura intervallumokat tartalmaz, és persze pontokat. Mi lenne, ha ehhez a természetes koncepcióhoz készítenénk egy számító egységet? A hagyományos számítógépek számos bitet használnak (általában 2 hatványait), ezeket bájtokba szervezve, és a számítási folyamat egy vagy több bájtot használ egy számítási lépésben. Egy bájt hivatkozhat egy természetes számra 0 és 2^k-1intervallumon belül. Elméletileg a számítógép hatékonysága erősen függ a *k* értékétől. A dolgozaton belül megpróbálok rámutatni, ha a hivatkozott bájt nem egy véges egész értékre vagy egy valós értékre hivatkozik adott intervallumon belül, hanem egy intervallum értékre, az sokkal nagyobb hatékonyságot jelent a számításban. Sajnos a jelenlegi technikai lehetőségek nem teszik egyenlőre lehetővé, hogy készítsünk egy ilyen számítógépet, ami intervallumokkal foglalkozik, de a szakdolgozatban megpróbálok megvalósítani egy programot, amivel a mai számítógépek korlátain belül (nincs végtelen sok valós szám 0 és 1 között) szimulálni lehet egy ilyen számítógépet.

Mint korábban már utaltam arra a tényre, hogy az intervallumok fogalma mennyire természetes az embereknek, kézenfekvő, hogy bizonyos műveletek elvégzése sokkal egyszerűbb, gyorsabb és komfortosabb volna, ha implementálnánk őket intervallumokra.

Lássunk néhány példát, hol vehetnénk hasznát az intervallumoknak:

- a különböző, két-, vagy többértékű logikák ábrázolására,
- vagy a SAT problémák lineáris időben való megoldására.

A szakdolgozatban egy olyan programot írok, amellyel lehet szimulálni a következő logikákat:

- klasszikus kétértelmű logika / Boolean
- Háromértékű Heyting logika, Lukasiewicz logika
- Belnap négyértékű logika
- Fuzzy logika / tetszőleges érték 0 és 1 között

valamint szimulálni tud egy analóg számítógépet, mellyel lineáris időben megoldhatóak a SAT problémák.

De miért is jó intervallumokkal számolni? Mert az így ábrázolt feladatok értelmezéséhez nem kell magasszintű szakmai ismeret, a diákok könnyebben megértik a logikai feladatokat, és az eredmény egyből leolvasható az ábráról.

A dolgozatban bemutatom a különböző logikákat, prezentálom a működésüket a programmal, bemutatom azt az újelvű számítási módszert, amivel lineáris időben megoldható a SAT probléma, bemutatom a programot működés közben, és végül magának a program fejlesztésének fontosabb lépéseit ismertetem.

1. Általánosságban az intervallum-számítógépekről

Az intervallum számítógépek olyan architektúrák, melyek bitek vagy más egységek helyett intervallumokon dolgoznak.

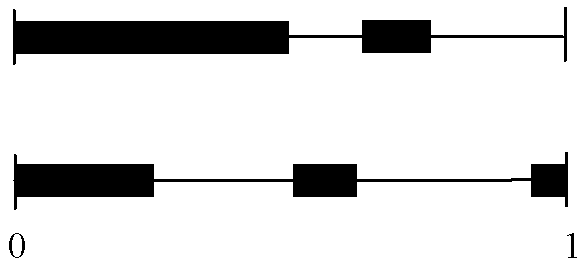
Az intervallumok működését a 0 és 1 közé eső alulról zárt intervallumokra írták le. Azaz az intervallum minden pontja a [0,1) intervallum része.

Az, hogy egy intervallumba beletartozik-e egy pont, azt az intervallum karakterisztikus függvénye adja meg.

A fenti leírásból kitűnik, hogy minden intervallumnak potenciálisan végtelen sok pontja van. Azaz, az intervallum számítógépek egyszerre végtelen sok logikai értékkel dolgoznak. Ebből következik, hogy a számítási ereje jóval nagyobb, mint a hagyományos számítógépeké.

1.1 Intervallum értékek és műveletek

Ahogy korában már említettem, az intervallum-értékek véges uniói a [0,1) intervallum nem egybefüggő, balról zárt, jobbról nyitott szub-intervallumai.



1.1.1 ábra példa az intervallum-értékek vizuális prezentációjára

Az intervallum-értékű számítási rendszerben intervallum értékeket használunk a [0,1] felett. Ezen intervallumok iteratív definíciója:

Minden egyes intervallum [a,b] ($0 \le a \le b \le 1$) egy atomi intervallum, amely minden x pontot tartalmaz a és b között ($a \le x \le b$). a és b speciális megválasztásával ($a=b$) megkapjuk a [0,1] intervallum 1 pontját, ami maga egy atomi intervallum.

Két intervallum-érték uniója és különbsége is intervallum-érték. (két intervallum-érték uniója tartalmazza az összes pontot azokból, amelyek legalább az egyik intervallum-értékben előfordulnak, A és B intervallumok különbsége pedig azokat a pontokat tartalmazza, melyek A-ban előfordulnak, de B intervallum-érték nem tartalmazza.)

Minden intervallum-érték megkonstruálható atomi intervallum-értékekből csak az unió és a különbségképzés műveletét használva.

Van két speciális intervallum-érték, az üres és a teljes intervallum. Jelöljük az üres halmazt ⊥-vel, a teli halmazt pedig ⊤-vel.

1.1.2 ábrb Az üres és teljes intervallum-értékek

A logikai operátorokról kicsit később fogok beszélni, majd az intervallum alapú logikánál.

2. Klasszikus és többértékű logikák, a Fuzzy logika

2.1 Klasszikus Boole-logika

Boole angol matematikus nevével fémjelzett logika, amit a legrégebb óta használunk. A Boole-logika szerint minden kijelentés vagy csak igaz, vagy csak hamis lehet. Ezzel meg is van a legnagyobb hátránya, mivel nem tud mit kezdeni az olyan állításokkal, mint pl.: „Én most hazudok”. Ugyebár, ha hazudok, azzal igazat mondtam, tehát nem hazudtam.

Alant látható a Boole-logika igazságtáblája.

Név	1. változó	2. változó	Negáció	Konjunkció	Diszjunkció	Implikáció
Jele	A	B	A	$A \wedge B$	$A \vee B$	$A \rightarrow B$
értékek	0	0	1	0	0	1
	0	1	1	0	1	1
	1	0	0	0	1	0
	1	1	0	1	1	1

2.1.1 táblázat, a Boole-logika igazságtáblája

Mivel az igaz-hamis értékeket bináris számokkal vagy logikai áramkörök feszültségszintjeivel is azonosíthatjuk, a Boole-algebra elmélete rengeteg gyakorlati alkalmazással bír a villamosmérnöki szakma és a számítógéptudomány területén.

A programban a Boole-logika szimulálására a következő módszert alkalmaztam:

- Ha hamis, akkor üres intervallum
- Ha igaz, akkor pedig a teljes intervallum a változó értéke.

2.1.1 ábra Igaz-Hamis értékek a programban 1

Dióhéjjban ennyit a Boole-logikáról.

2.2 Heyting 3 értékű logikája

Az első többértékű logika, ami próbálta követni az intuicionista utat (az intuicionisták szerint egy állítés, melyet még nem bizonyítottak, nem rendelkezik igazságértékkel, sőt léteznek se nem igaz, se nem hamis állítások, melyeket lehetetlen bizonyítani vagy cáfolni). Mint korábban megemlítettem, a Boole-logika nem tudott mit kezdeni bizonyos állításokkal, olyanokkal, amik lehetnek, igazak is meg hamisak is, tehát egyértelműen nem eldönthetők. Heyting azt gondolta, hogy legyen egy harmadik érték, egy köztes állapot, az ½, ami nem teljesen igaz, de nem is teljesen hamis.

A	¬A
1	0
½	0
0	1

A→B	1	½	0
1	1	½	0
½	1	1	0
0	1	1	1

2.2.1 táblázat, a Heyting-logika igazságtáblája negációra és implikációra

A programban a Helyting-logika szimulálására a következő módszert alkalmaztam:

- Ha hamis, akkor üres intervallum
- Ha ½, akkor értelemszerűen a [0,1/2] intervallumot használtam
- Ha igaz, akkor pedig a teljes intervallum a változó értéke.

2.2.1 ábra Heyting-logika értékei a programban

2.3 Belnap 4 értékű logikája

Belnap a következő értékeket alkalmazta a rendszerében:

- Igaz
- Ismeretlen
- Mindkettő
- És végül hamis

Belnap logikáját általában lehetséges ellentmondó tudás reprezentálására, valamint ismeret megvizsgálásának és frissítésének problémájára használták.

Ez a rendszer szintén alkalmazva van logikai programok tanulási folyamatában.

A	¬A
t	f
u	u
±	±
f	t

A∧B	t	u	±	f
t	t	u	±	f
u	u	u	f	f
±	±	f	±	f
f	f	f	f	f

A∨B	t	U	±	f
t	t	t	t	t
u	t	u	t	u
±	t	t	±	±
f	t	u	±	f

2.3.1 táblázat, a Belnap-logika igazságtáblája

A programban a Belnap-logika szimulálására a következő módszert alkalmaztam:

- Ha hamis, akkor üres intervallum
- Ha ismeretlen, akkor a [0,1/2] intervallumot használtam
- Ha mindkettő, akkor a [1/2,1] intervallumot használtam
- Ha igaz, akkor pedig a teljes intervallum a változó értéke.

2.3.1 ábra Belnap-rendszer értékei a programban

2.4 Łukasiewicz többértékű logikája

Élete során Łukasiewicz készített 3- és 4értékű logikát is. Később kiterjesztette a rendszert tetszőlegesen sok (n≥2) igazságértékűvé, egészen a megszámlálható végtelenig.

Łukasiewicz eredetileg az implikációt és a negációt definiálta. A Łukasiewicz-stílusú logikának szintén van & és + művelete is. Ezek az alábbiak szerint vannak definiálva:

¬A=1-A

A & B = max(A + B – 1,0)

A + B = min(A + B,1)

$$A \to B = \begin{cases} 1, & \text{ha } B \geq A \\ 1 - A + B, & \text{ha } A \geq B \end{cases}$$

Ezek a típusú műveletek szintén használhatók véges sok értékű logikai rendszerekben, amelyekben az értékek a következők (minden *k*>2 egész esetén):

0=0/(*k*-1),1/(*k*-1),…, (*k*-1)/ (*k*-1)=1. (k értékű Łukasiewicz-rendszerben, ha *k*=2, visszakapjuk a klasszikus műveleteket)

A konjukciónak és diszjunkciónak szintén vannak megfelelői a Łukasiewicz-típusú logikában, ezek nevei „kötött produktum" és „kötött szum", tiszteletből.

A programban, mikor kiválasztja a felhasználó, hogy a Łukasiewicz-típusú logikát akarja használni, megadja az elemek mennyiségét (*k*), majd egyszerűen csak azt kell beírni, hogy hányadik értékre hivatkozik (0-*k*).

2.4.1 ábra 5 elemű Łukasiewicz -rendszer értékei a programban

2.5 Fuzzy- logika

Gödel 1932-ben mutatta be rendszerét végtelen sok igazságértékkel. A következő négy műveletet definiálta változóira:

$$A \to B = \begin{cases} 1, \text{ha } A \leq B \\ 0 \text{ egyébként} \end{cases}$$

$$\neg A = \begin{cases} 1, \text{ha } A = 0 \\ 0 \text{ egyébként} \end{cases}$$

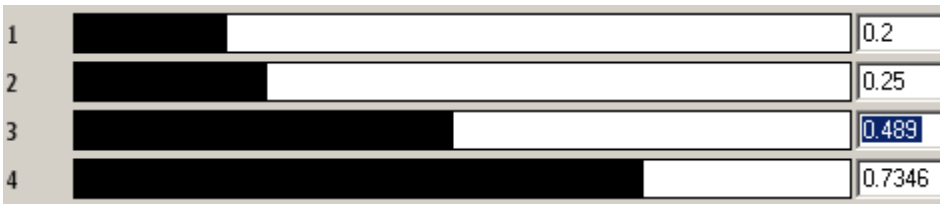
$$A \wedge B = \min(A, B)$$

$$A \vee B = \max(A, B)$$

Most már láthatjuk, hogy Heyting rendszere speciális esete Gödel Fuzzy-rendszerének, ahol mindössze 3 igazságértéket engedélyezünk.
Léteznek véges sok igazságértékű variációi is a Gödel-rendszernek, ahol az igazságértékek lekorlátozódnak *m*/*k* variációra (*m*>1, 0≤*k*≤*m*), a műveletek pedig megegyeznek az eredeti rendszerben levővel. Az intuíciós logikában a klasszikus törvény, a dupla negáció ¬¬A→A

nem működik az “A” összes lehetséges értékével. Mivel Gödel rendszere is egy intuicionista rendszer a lánctörvénnyel: (A →B) ∨ (B→A).

A programban a felhasználónak elég beírni az értéket, a program generál egy tartományt a Fuzzy-változónak.



2.5.1 ábra néhány Fuzzy-érték a programban

2.6 Intervallum-értékű logika

Ez a logika meglehetősen újkeletű, kitalálása dr. Nagy Benedek nevéhez fűződik.

Az Intervallum-értékű logikában az igazságértékek ponthalmazok vagy nem összefüggő intervallumok lehetnek a (0,1] intervallum fölött. A (0,1] tartomány minden pontja vagy benne van 1 intervallumban, vagy nincs.

Név	Negáció	Konjunkció	Diszjunkció	Implikáció
Jele	¬A	A ∧ B	A ∨ B	A→B
értékek	$\overline{\lvert A\rvert} = [0,1]\backslash\lvert A\rvert$	$\lvert A\rvert \cap \lvert B\rvert$	$\lvert A\rvert \cup \lvert B\rvert$	$\overline{\lvert A\rvert} \cup \lvert B\rvert, \overline{\lvert A\rvert\backslash\lvert B\rvert}$
Halmazelmélet	|A| komplementere	Metszet	Unió	Különbségek komplementere

2.6.1 táblázat, Az intervallum alapú logika alap logikai műveletei

Most pedig bemutatom az Intervallum-értékű logika logikai műveleteinek vizuális implementációit

Negáció:

2.6.1. ábra Példa negációra a programban

Mint látható, A és ¬A egymásnak komplementerei.
Konjunkció, diszjunkció és implikáció:

2.6.2. ábra Példa logikai műveletekre a programban

Vizuálisan jobban érzékelhető, hogy a műveletek valóban megfelelnek a halmazműveleteknek.

Van még pár nem logikai művelet, melyek definiálása még hátravan. Ezek már nem igazán az intervallum-értékű logikához tartoznak, inkább az intervallum-értékű számításokhoz kellenek.

Először is az első hossz (First Length), aminek FLength a neve. Ez a funkció egy valós értéket rendel az intervallumunkhoz, ami az első intervallum-elem hossza.

FLength(A)=b-a amennyiben A tartalmazza az (a,b) intervallumot, és nincs egy másik intervallum (a,c), melyre igaz, hogy $c \geq b$, és A nem tartalmaz olyan x pontot, melyre igaz, hogy $x \leq a$.
FLength(A)=0 minden más esetben.

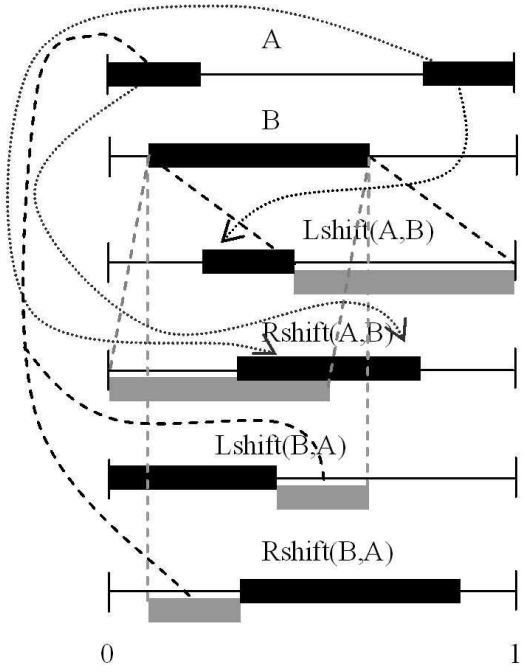
E művelet segítségével megvalósíthatunk két másik eltoló műveletet. Mindkettőnek 2 operandusa van. A balra toló operátor eltolja az első intervallum-értéket a második operandus első intervallum-elemének hosszával (FLength), és levágja azt a részt, ami kitolódik a [0,1] intervallumból. Ennek ellentéte a jobbra toló operator, mely értelemszerűen jobbra tolja el az első intervallum-elemet a második operandus első hosszával, de itt a másik oldalon bekúszik az, ami kitolódik a [0,1] intervallumból.

Matematikailag:

LShift(A,B)(x)=A(x+FLength(B)) ha $0 \leq x+FLength(B) \leq 1$, minden más esetben 0.

RShift(A,B)(x)=A(frac(x-FLength(B))), ahol $x<1$. A frac() függvény megadja a frakcionális részét egy valós értéknek. frac(x)=x-int(x), ahol int(x) a legnagyobb egész szám, ami nem nagyobb x-nél.

A 2.6.3-as ábrán vizuálisan látható, hogyan is zajlik le a jobbra és balra eltolás folyamata.



2.6.3. ábra A jobbra-balra tolás művelete

Van még egy operator, ami meg tudja sokszorozni az intervallum-értékeket, a Product.

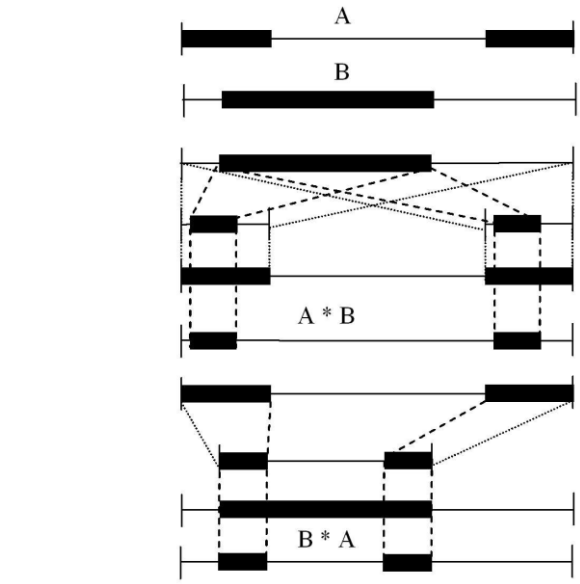
Legyen A intervallum-érték, ami tartalmaz *k* intervallum-elemet a_{i1},a_{i2} ($1 \le i \le k$) és B, ami *l* elemet tartalmaz a_{i1},a_{i2} ($1 \le i \le l$). Aztán legyen C=A*B értéke a következő:
C elemeinek a száma legyen $k \times l$. Ehhez a művelethez használhatunk kettős indexelést a C elemeihez. Legyen az *ij*-edik komponens kezdő- és végpontja $a_{i1}+b_{j1}(a_{i2}-a_{i1})$ és $a_{i1}+b_{j2}(a_{i2}-a_{i1})$. Egy végpont hozzátartozik az intervallumhoz, ha az eredeti végpontok hozzátartoztak az eredeti A és B intervallum-értékeihez.

Mint a definícióból látható, ez a produktum meglehetősen hasonlít a Cartesian-produktumhoz, az indexek ugyanazon elv alapján működnek. C *ij*-edik komponense az A *i*-edik és B *j*-edik komponenséből keletkezik.

A*B elemeinek a hossza megegyezik A elemei hosszának az összegével és B elemei hosszának az összegével.

A 2.6.5. ábrán vizuálisan szemlélhető a Produktumképzés folyamata.
A 2.6.4. ábrán pedig a program által generált produktum.

2.6.4. ábra Produktum képzés a programban



2.6.5. ábra Produkt képzés folyamata

Na de mire is használhatóak az intervallumok?
Azt már bemutattam, hogy a különféle logikai rendszerek hatékonyan vizualizálhatóak intervallumokkal, látványos eredményeket adva, de van egy másik lehetőség, ahol szintén nagyon hatékonyan lehet alkalmazni az intervallumokat problémamegoldásra, mégpedig a SAT problémák.

3. SAT probléma

A SAT probléma logikai formulák kielégíthetőségét vizsgálja. Elvárás, hogy a logikai formula konjunktív normálformában legyen. Ekkor, ha a formula *n* atomból (logikai változóból) áll, akkor legrosszabb esetben 2^n próbálkozásból megállapítható, hogy kielégíthető-e a formula.
A SAT probléma jól ismert NP-teljes probléma, ami azt jelenti, hogy nem oldható meg polinomiális időben.

A konjunktív normálformák olyan nulladrendű logikai formulák, melyekben csak változók és az $\neg$, $\vee$, $\wedge$ operátorok fordulnak elő.

Tekintsük meg az alábbi konjunktív normálforma alakú formulát:

$(a \vee b \vee c) \wedge (\neg b \vee c \vee \neg d) \wedge (\neg a \vee \neg e \vee d)$

A változókat a normálforma atomjainak nevezzük, a fenti példában *a,b,c,d,e*

A változókat vagy negáltjaikat összefoglaló néven a normálforma literáljainak nevezzük, jelen esetben ezek *a, b,* $\neg b$*, c,* $\neg d$*, d,* $\neg e$*.* A negálatlan literálokat pozitív literáloknak, a negáltakat negatívnak is szokás jelölni.

A literálok diszjunkciói pedig a klózok (*clause*, angolul mellékmondat). A fenti példa klózai:

$a \vee b \vee c$, $\neg b \vee c \vee \neg d$ és $\neg a \vee \neg e \vee d$.

3.1 SAT probléma megoldása intervallum-értékű logikával

Most pedig bemutatom, hogyan lehet lineáris időben megoldani a SAT problémát.

A Klasszikus (digitális) számítások a 2 értékű logikán alapulnak.

Az intervallum értékű számítások intervallum-értékű logikán alapulnak. Az intervallum-értékű számítógépeken az adategység a bájt helyett az intervallum-érték.

A SAT intervallumokkal való megoldásának lépései az alábbiakban lettek definiálva:

- Legyen n a változók száma
- Legyen [0,1/2) a FIRSTHALF konstans értéke.

A számítási sor: FIRSTHALF - ból indulva:

- $S_0,\ldots, S_n$, ($n \in \mathbf{N}$), ahol S_0 = FIRSTHALF,
- minden $i<n$: $S_i = op(S_k, S_j)$, ahol $k,j<i$.
- és op egy Boole operátor, a shift vagy a product. (negációnál csak egy (az első) argumentumot használjuk)

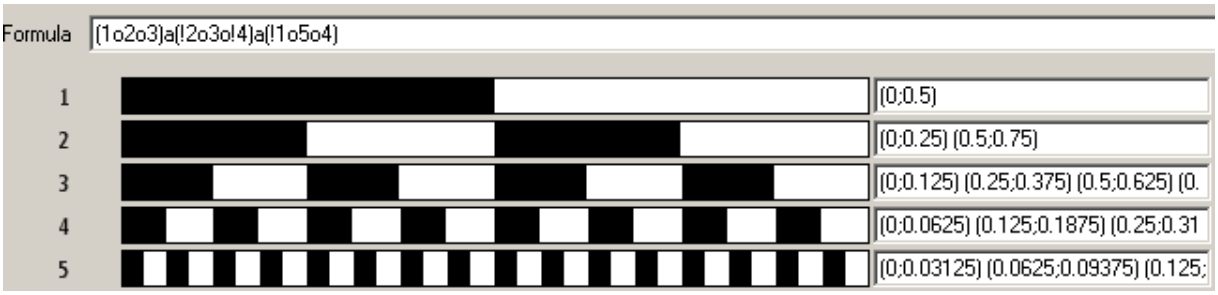
- Minden lehetséges kombináció elkészítése:
- product, negáció és unió
  (diszjunkció) műveletekkel.
  Ezek száma lineáris a változók számában mérve:
- $A_{i+1}=A_i*[0,1/2)\vee\neg A_i*[0,1/2)$
- Így az *i.* változó értéke:

$$A_i = \bigcup_{j=0}^{2^{i-1}-1} \left[\frac{j}{2^{i-1}}, \frac{2j+1}{2^i}\right)$$

Hogy könnyebben érthető legyen, jöjjön egy kis grafikus prezentáció!

Vegyük mégegyszer a bevezetőben írt formulát:

$(a \vee b \vee c) \wedge (\neg b \vee c \vee \neg d) \wedge (\neg a \vee \neg e \vee d)$

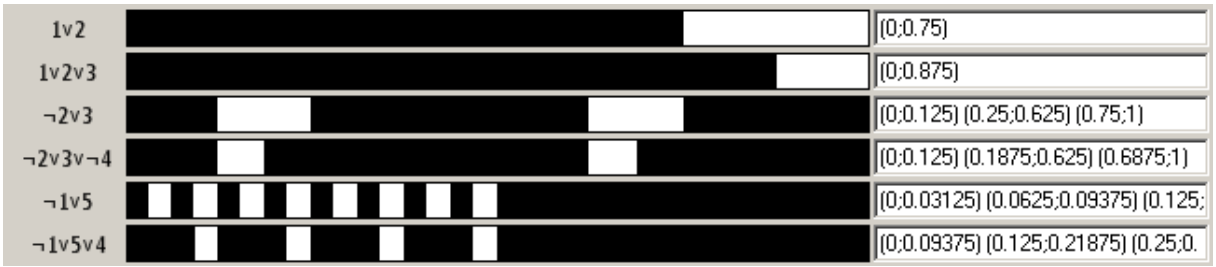


3.1.1. ábra Változók felvétele a programban

A 3.1.1. ábrán látható a program működés közben. némi magyarázatra szorulnak a jelölések, de bevezetőnek annyit, hogy a változók neve betűk helyett szám, hogy miért, arról később.

Ami viszont látszik a képernyőn, hogy az első változó a FirstHalf értéket vette fel, ami a [0,1/2] intervallum. A következő változó pedig $A_1*[0,1/2)\vee\neg A_1*[0,1/2)$, és így tovább. Az utolsó intervallum 2^{n-1} darab intervallum-elemet tartalmaz.

A változók felvétele után nem marad más hátra, mint kiértékelni a klózokat, és azok eredményein is elvégezzük a klózok közötti műveleteket. Ezt a lépést láthatjuk a 3.1.2. és a 3.1.3. ábrán.



3.1.2. ábra Kiértékelt klózok a programban

3.1.3. ábra Végeredmény a programban

Amennyiben a végeredmény tartalmaz legalább 1 intervallum-elemet, van megoldása a formulának, ha pedig nem tartalmaz egy intervallum-elemet sem, nem elégíthető ki a formula.

4. A program elkészítése

Eddig láthatóak voltak a különféle logikák és a SAT probléma megoldása is a programból vett képekről, most pedig következik, hogyan is készült el az a program, amivel mindez prezentálható.

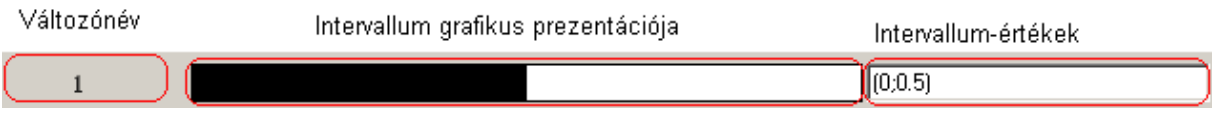
A program fejlesztésének a főbb lépései a következők voltak a velük szemben támaszott követelményekkel:

1. Az intervallum-értékek megjelenítéséért felelős komponens. Legyen képes értelmezni az előre beállított működési módnak megfelelő intervallum-értékeket, jelenítse meg azt grafikusan és írja is ki az intervallum-elemek értékeit.
2. A kezelőfelület. Legyen átlátható, egyszerű, felhasználóbarát. adjon módot arra, hogy a felhasználó kézzel felvehessen változókat, tudja definiálni a változók értékeit, tudjon formulákat beírni, adjon lehetőséget a formulák kiszámítására, mentési- és betöltési lehetőség.
3. Mentés-betöltés

A program Microsoft C# 2008 Express Edition fejlesztői környezetben és programnyelven lett elkészítve, mely tanulási célokra ingyenesen letölthető.

4.1 IntervalPanel komponens fejlesztése

A program alapköve a megjelenítés mellett sok egyéb dologért is felelős `IntervalPanel` komponens. Maga a komponens 3 fő részből áll.



4.1.1. ábra IntervalPanel komponens és részei

A 4.1.1 ábra szemlélteti a komponens felépítését.

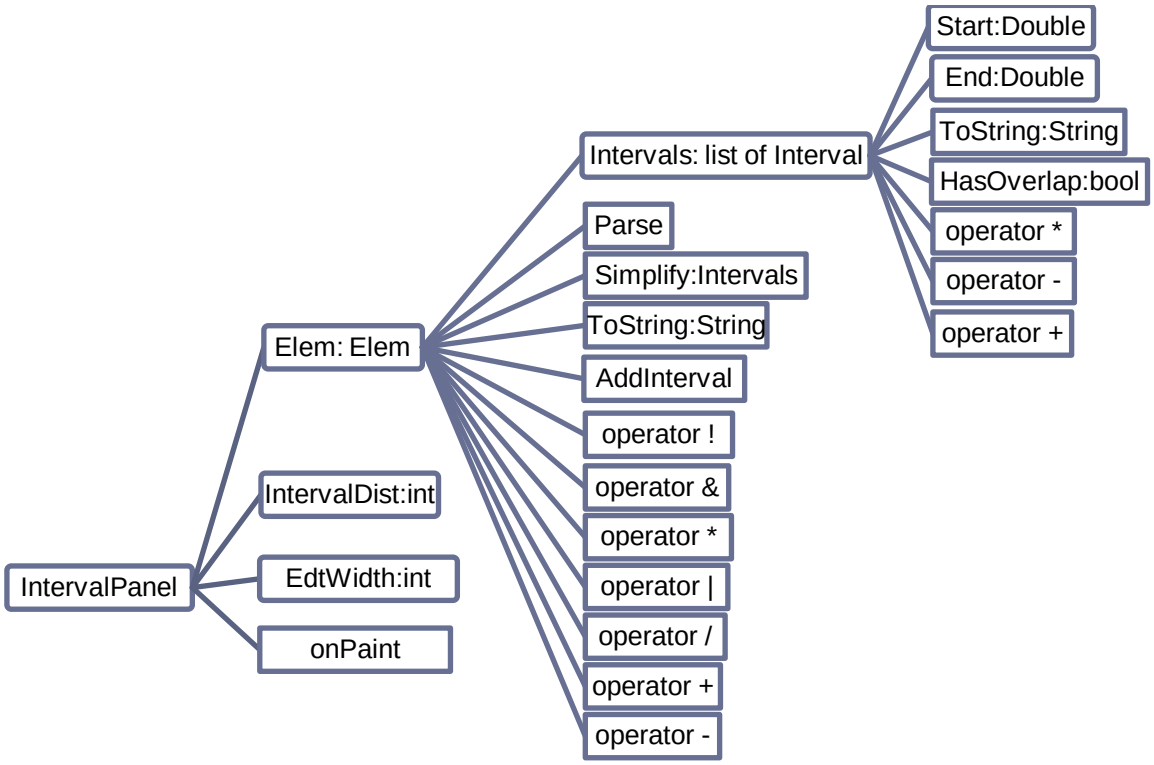
Balra a változó neve szerepel, hogy a felhasználó tudja, melyik intervallum-érték szerepel az adott panelen.

Középen az intervallum-érték grafikus prezentációja.

És végül a jobb oldalon egy szerkesztőmező, ahol egyrészt megjelennek az intervallum-érték intervallum-elemei, másrészt, ha a felhasználó maga akar beírni értékeket, akkor itt megteheti, amit ezután a komponens értelmez és megjelenít.

Az `IntervalPanel` komponens legfontosabb eleme az `Elem` osztály (angol Element szóból). Amikor a felhasználó beírja az intervallum-értékeket, akkor tulajdonképpen ez az osztály értelmezi a felhasználói inputot, és az `IntervalPanel` komponens ezt jeleníti meg.

Az `Elem` osztály az intervallum-érték, elemei pedig az `Interval` osztály típusú lista elemei.



4.1.1. táblázat `IntervalPanel` komponens architektúrája

A 4.1.1. táblázat egy kivonat a program szerkezetéről, látható, ahogy az objektumok egymásba vannak ágyazva.

4.1.1 Interval osztály fejlesztése

Az ismertetést kezdeném talán a legalján, az `Interval` osztállyal.

Ez a legegyszerűbb osztály a programban, van egy kezdő és egy végértéke (Start, End). Hogy később megkönnyítse a munkát, leszármaztattam az `IComparable` interfészből, ehhez implementálni kellett a `CompareTo()` metódust, így meg lehet hívni később a `Sort()` függvényt, ami egy kicsit később kiderül, miért hasznos. De most visszatérve a `CompareTo()` metódusra, a kezdőpontok alapján végződik az összehasonlítás.

Van egy statikus HasOverlap nevű bool típusú függvénye, bemenő paraméterei két `Interval` típusú változó. Azért lett statikus, mert ugyan az `Interval` típushoz tartozik, de így rugal-masabban lehet használni, bármikor megadok neki két intervallumot, és megmondja, hogy van-e köztük átfedés vagy nincs.

A ToString() függvény visszaadja stringként az intervallum értékét, attól függően, hogy milyen mód lett beállítva a programban.

3 operátora felül van írva, hogy így kényelmesebben lehessen használni a különböző logikai és nem logikai műveletek során. A – operator a LShift műveletnek felel meg. A + operátor a RShift. Végül van még a * operátor, melyet a produktum művelet közben lehet használni.

4.1.2 Elem osztály fejlesztése

Följebb lépve az `Elem` osztály található. Első és legfontosabb változója az Intervals, ami `Interval` típusú lista. Ez tartalmazza az intervallum-elemeket, és mivel lista típusú, akármennyi eleme lehet (max. memóriakorlát).

Következő függvénye a Parse(string), mely a string típusú paramétert értelmezi. Formai megkötésként az intervallum-értékek az alábbi „(;)” formában kell legyenek, vagyis kerek zárójelben pontosvesszővel elválasztva a kezdő és a végértéket. A nemzetközi decimális elválasztójelek közti különbséget figyelembe véve még a metódus elején egy konverzió hajtódik végre, ami a , és a . jeleket helyettesíti a helyi nyelvi beállításokban található decimális elválasztójellel.

```
pc_text = pc_text.Replace(",", culture.NumberFormat.NumberDecimalSeparator);
```

majd ezek után a “(“ és “)” karakterek közötti értéket megfelelteti az intervallum-elem kezdeti és végértékének.

```
double mn_start = double.Parse(s.Substring(0, s.IndexOf(";")), culture),
       mn_end   = double.Parse(s.Substring(s.IndexOf(";") + 1), culture);
AddIntervall(mn_start, mn_end);
```

Az AddInterval mindössze hozzáadja az intervallum-elemet az intervallum-listához.

A Simplify egyszerűsíti az intervallum-elemeket, ha átfedés van köztük, vagy csak egymásba érnek, akkor összevonja őket. A működése során végigszalad az összes intervallumon két egymásba ágyazott ciklussal és megvizsgálja az egyes elempárokat, hogyan viszonyulnak egymáshoz:

```
  private void Simplify()
  {
    Intervalls.Sort();

    for (int i = Intervalls.Count - 2; i >=0 ;i-- )
    {
      for (int j = Intervalls.Count - 1; j > i; j--)
        if ((Intervalls[i].End >= Intervalls[j].Start))
        {
          Intervalls[i].End = Math.Max(Intervalls[j].End, Intervalls[i].End);
          Intervalls.RemoveAt(j);
        }
    }
```

```
}
```

Itt látható, hogy a Sort() metódus használatával nekem nem kellett foglalkozni a sorbarendezéssel, mert a Sort() egy beágyazott QuickSort eljárást használ. Mivel már az intervallum-elemek sorba vannak rendezve a kezdőértékük szerint, csak azt kell megvizsgálni visszafelé, hogy van-e olyan elem, melynek a vége nagyobb, mint egy később következő elem eleje. Ha van ilyen, akkor azok összevonhatók, és a későbbi elem törölhető.

A ToString() függvény itt is szerepel. Hasznos, mivel nem csak a szerkesztőmezőbe írja ki az intervallum értékét, hanem remekül használható a mentésnél. Végigfut az intervallum-elemeken, meghívja azok ToString() függvényét, és összefűzi őket.

```
public override string ToString()
{
  StringBuilder s = new StringBuilder();
  for (int i = 0; i < Intervalls.Count; i++)
    s.Append(Intervalls[i].ToString() + " ");
  return s.ToString();
}
```

Az operátorok közül kicsit részletezném a konjunkciót, mivel az volt a legbonyolultabb, a többit csak futólag említem meg.

Az intervallum-logika konjunkcióját implementáltam az “&” operator helyére, mivel a számítástechnikában is ez a jele.

```
public static Elem operator&(Elem po_elem1, Elem po_elem2)
{
  Elem mo_elem=new Elem();
  double mn_start=0,mn_end=0;

  foreach (Intervall mo_int1 in po_elem1.Intervalls)
  {
    foreach (Intervall mo_int2 in po_elem2.Intervalls)
    {
      if (Intervall.HasOverlap(mo_int1, mo_int2))
      {
        if(mn_start!=0)
          mn_start = Math.Min(Math.Max(mo_int1.Start, mo_int2.Start), mn_start);
        else
          mn_start = Math.Max(mo_int1.Start, mo_int2.Start);
        mn_end = Math.Max(Math.Min(mo_int1.End, mo_int2.End), mn_end);
        mo_elem.AddIntervall(mn_start, mn_end);
        mn_start = mn_end = 0;
      }
      else
      {
        if ((mn_start != 0) && (mn_end != 0))
        {
          mo_elem.AddIntervall(mn_start, mn_end);
          mn_start = mn_end = 0;
        }
      }
    }
  }
  if ((mn_start != 0) && (mn_end != 0))
  {
    mo_elem.AddIntervall(mn_start, mn_end);
    mn_start = mn_end = 0;
  }
  mo_elem.Simplify();
  return mo_elem;
}
```

A külső ciklus végigmegy az első intervallum paraméter összes elemén, a beágyazott ciklus pedig a második intervallum paraméter összes elemén és megvizsgálja, van-e átfedés a két

aktuális intervallum-elem között. Itt használtam fel az `Interval` osztály `HasOverlap()` függvényét, amit már korábban említettem. Ha van átfedésük, akkor van közös metszetük is. Ha van közös metszet, akkor mivel a konjunkció halmazelméletileg a metszet, így az operátor visszatérő intervallum-értékéhez hozzá lehet tenni a közös metszetet. A metódus végén meghívjuk a `Simplify()` függvényt és az új, közös metszetet tartalmazó intervallum készen áll a visszaadásra.

A „|” operátor szintén a C és C alapú nyelvek diszjunkciója, így egyértelmű volt, hogy mire lesz használva, az intervallumok diszjunkciójára.

A „*” operátor a multiplikáció jele, így az intervallumoknál is a multiplikáció lesz a funkciója.

A „!” negációt jelent, így ennek is megmaradt a jelentése.

A „/” mivel az intervallumokon egyenlőre nincs értelmezve az osztás, így ő lett az implikáció.

A „+” és „–“ jelek a jobbra- és balratolás műveletei.

4.1.3 IntervalPanel osztály fejlesztése

Az `IntervalPanel.IntervalDist` egy statikus változó, mely azt jelenti, hogy a képernyőn hány pixel távolságra kezdődjön az intervallum-érték grafikus prezentációja, az IntervalPanel.EdtWidth statikus attributum pedig a TextBox méretét adja meg. Ezzel a két statikus változóval biztosítható, hogy az egymás alá kerülő `IntervalPanel`-ek mindig egyformán nézzenek ki, egymás alatt legyenek elvágólag.

Az `IntervalPanel.OnPaint()` pedig a megjelenítésért felelős metódus.

Balról jobbra haladva a megjelenítéssel, először a komponens kiírja az intervallum-panel feliratát

```
// Text of IntervalPanel
pe.Graphics.DrawString(Text, Font, brush_Black, r, style);
```

Majd ezután megrajzolja az üres intervallumot, rajzol köré egy fekete keretet, és bele az intervallum-elemeket

```
// White Rectangle
r=new Rectangle(IntervallDist,0,Width-IntervallDist-EdtWidth-2,Height);
pe.Graphics.FillRectangle(brush_White,r);
// Black frame arounding the White Rectangle
r.Width -= 1;
r.Height-=1;
pe.Graphics.DrawRectangle(pen_Black,r);
for (int i = 0; i <= Elem.Intervalls.Count - 1; i++)
{
  int mn_width = Width - IntervallDist-EdtWidth-2;
  pe.Graphics.FillRectangle(
    brush_Black,
    new Rectangle(
      IntervallDist+(int)(mn_width*Elem.Intervalls[i].Start),
      0,
      (int)(mn_width*Intervall.FLength(Elem.Intervalls[i])),
      Height
    )
```

```
  );
}
```

Ezek után jön az edit mező megjelenítése

```
// EditBox
edt.Left = Width - EdtWidth;
edt.Top = 0;
edt.Width = EdtWidth;
```

és végül az intervallum-elemek valós értékei bekerülnek az edit mezőbe.

```
edt.Text = Elem.ToString();
```

Ezzel készen is van egy komponens, ami mindent tud, amire szüksége van, hogy intervallumokat jelenítsen meg, ismeri az alapvető műveleteket, használatra készek.

4.2 A felhasználói felület fejlesztése

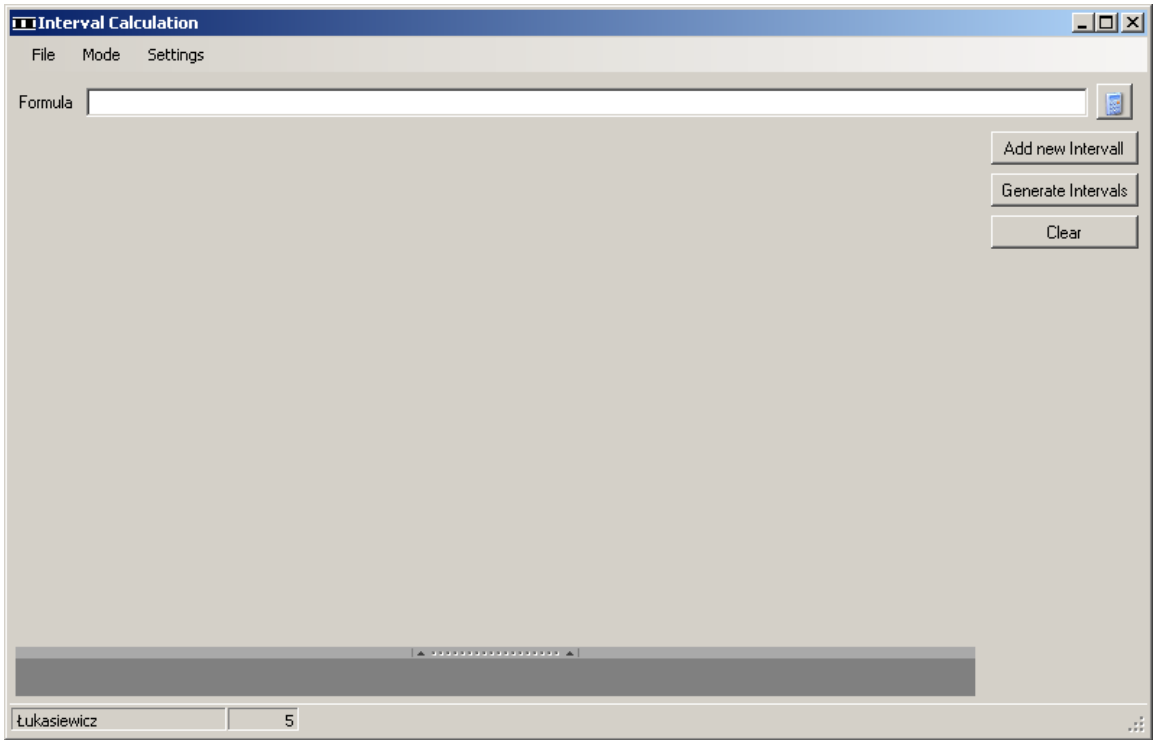
A felhasználói felület fejlesztésekor igyekeztem a leghatékonyabb, adatbevitelt legjobban támogató felületet megalkotni, ugyanakkor egyszerű, áttekinthető legyen a felhasználó számára.

A képernyő a 4.1.2. ábrán látható. Szokványos Windows Form, legfelül menüvel, melyben a felhasználó elérhet olyan funkciókat, hogy Mentés, Betöltés, Üzemmódok kiválasztása, vagy a Beállítások menüpont.

Rögtön a menüsor alatt található a formula bevitelére szolgáló editmező. Ide írhatja a felhasználó a formulát, amit számoltatni akar. Annyit megjegyeznék, hogy a hatékonyabb adatbevitel érdekében nem betűket használtam az egyes változók megnevezésére, hanem számokat, így talán kényelmesebb a felhasználónak beírni az adatokat. Ezzel, valamint az operátorok betűvel való megfeleltetésével a felhasználónak nem kell ismernie rengeteg billentyűkombinációt, nem kell szenvednie speciális karakterek begépelésével, hanem csak természetes úton be lehet gépelni az adatokat
Ehhez még listáznám az elérhető operátorokat:

Művelet	Jelölés	Jelölés a programban
Konjunkció	$\wedge$	A, a
Diszjunkció	$\vee$	O, o
Implikáció	$\subseteq$	I, i
Multiplikáció	*	M, m
LShift		L, l
RShift		R, r

Kis és nagybetű nem számít.



4.1.2. ábra A felhasználói felület

Közvetlenül a formula beviteli mező mellett található a Kalkulátor ikonnal rendelkező gomb, mely a számolást indítja. A számítás eredménye függ a választott üzemmódtól. Én most csak a SAT üzemmód számítását ismertetem.

Először is klózonként értékelem ki a formulát, zárójeltől zárójelig. Ha nincs adott nevű változó felvéve, akkor general neki egyet a program a korábban említett FirstHalf érték mintájára. Itt megjegyezném, hogy számomra egyszerűbb volt a 2 hatványait használni, az eredmény ugyanaz.

```
    private void Parse()
    {
      string mc_text,s;
      int mn_pos1, mn_pos2; // mn_pos1 means the start index, mn_pos2 means the end index
      mc_text = edt_formula.Text.Replace(" ", "");
      mc_text = mc_text.ToUpper(); // Just to avoid the problems between small and capital
letters
      Elem mo_result = new Elem();
      Elem mo_elem1=null, mo_elem2=null;
      while (mc_text.Contains("("))
      {
        mn_pos1 = mc_text.IndexOf("(");
        mn_pos2 = mc_text.IndexOf(")");
        char mc_operand;
        if (mn_pos1 > 0)
          mc_operand = mc_text[mn_pos1 - 1];
        else mc_operand = ' ';
        s = mc_text.Substring(mn_pos1 + 1, mn_pos2 - mn_pos1 - 1);
        mc_text = mc_text.Substring(mn_pos2 + 1);

        if (mo_elem1 == null)
          mo_elem1 = ParseClause(s); // parsing the clause
        else
        {
          mo_elem2 = ParseClause(s); // parsing the clause
```

```
          Elem mo_tmp = new Elem();
          char mc_sign=' ';
          mo_tmp = Calculator(mo_elem1, mo_elem2, GetOperand(mc_operand,ref mc_sign));
          mo_tmp.Name = "(" + mo_elem1.Name + ")" + mc_sign.ToString() + "(" + mo_elem2.Name +
")";
          mo_elem1 = new Elem();
          mo_elem1 = mo_tmp;
        }
      }
      AddElem(mo_elem1, mo_elem1.Name, pnl_result);
    }
```

A `Parse()` függvény vagdossa szét a formulát klózokra, majd meghívja a `ParseClause()` függvényt. Ezt kicsit később részletesen ismertetem. Ez a függvény visszatér a klóz kiszámított intervallum-értékével, amit az `mo_elem1` változóban helyezek el, ha az még üres (első klóz értéke kerül bele). Ha már nem üres, akkor az `mo_elem2` változóba kerül az értéke, majd a két klóz közötti műveleti jelet értelmezve (`GetOperand()`) az újabb eredmény ismét az `mo_elem1` változóba kerül, így az mindig a legfrissebb számítási eredményt fogja tartalmazni, mindig felhasználva az előző számítás eredményét.

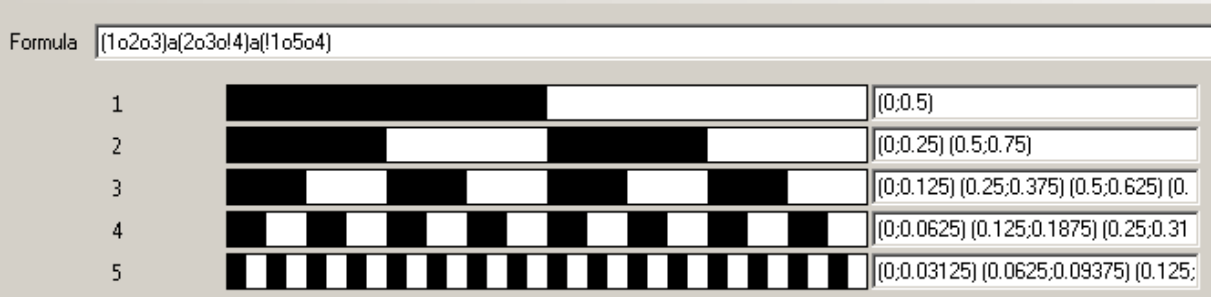
Most következzen a `ParseClause()` függvény. A terjedelme miatt csak részletekben ismertetem.

```
    private Elem ParseClause(string pc_s)
```

Input paraméterként megkapj a a zárójelek közötti stringet (klózt).

```
      // Firstly create the IntervallPanels
      for (int i = 0; i < pc_s.Length; i++)
      {
        if (char.IsNumber(pc_s[i]))
        {
          mc_elem1.Append(pc_s[i]);
          if (((i < pc_s.Length - 1) && (char.IsLetter(pc_s[i + 1]))) || (i == pc_s.Length -
1))
          {
            mo_elem = FindElem(mc_elem1.ToString());
            if (mo_elem == null)
            {
              mo_elem = AddElem(Convert.ToString(pnl_variables.Controls.Count + 1),
Convert.ToString(pnl_variables.Controls.Count + 1), pnl_variables);
              mc_elem1.Remove(0, mc_elem1.Length);
              int mn_interval = (int)Math.Pow(2, pnl_variables.Controls.Count);
              for (int j = 0; j < mn_interval; j += 2)
              {
                mo_elem.AddIntervall((double)j / mn_interval, (double)(j + 1) / mn_interval);
              }
            }
            else
              mc_elem1.Remove(0, mc_elem1.Length);
          }
        }
      }
```

Végigszalad egy ciklus a string karakterein, és megkeresi az összes változót a klózban (ha a karakter szám, akkor változónév). Ezek után megkeresi a már meglevő változók között, hogy létezik-e már az adott nevű változó vagy meg kell kreálni. Ha nem létezik, megcsinálja.



4.1.3. ábra A formula változói és azok értékei

Ezek után még egyszer végigfut a ciklus a string összes karakterén, és most már megkeresi a negációt, a változókat és az operátorokat.

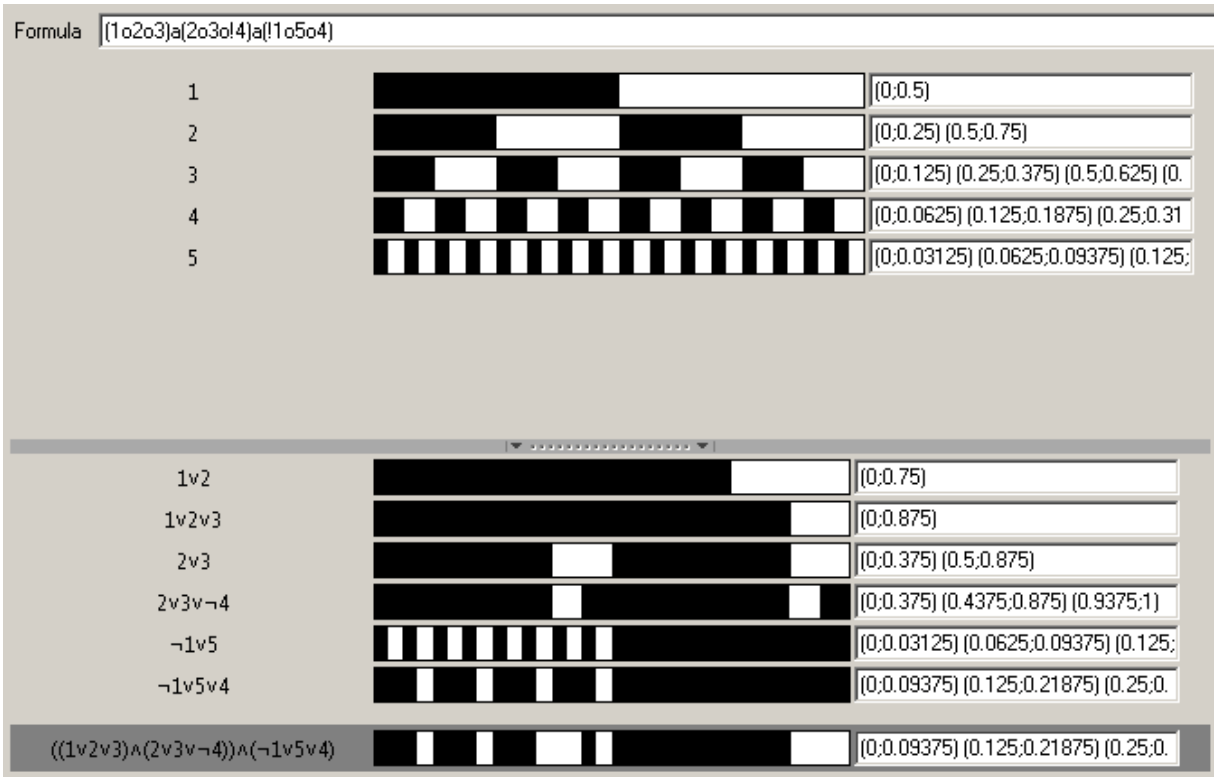
```
if (pc_s[i] == '!')
{
  ml_neg = true;
}
else if (char.IsNumber(pc_s[i]))
{
  if (ml_first) mc_elem1.Append(pc_s[i]);
  else mc_elem2.Append(pc_s[i]);
}
else if (char.IsLetter(pc_s[i])) // operand
{
  mc_operand = pc_s[i];
}
```

Megvan a negáció, megvan az elem neve, már csak meg kell keresni az elemet a korábban kreált intervallum-értékek között, és el kell végezni a negálást, amit nagyon egyszerű a felülírt „!” operátorral. Még apró finomítás, hogy a változó megjelenő feliratába beszúrjuk a ¬ szimbólumot (Unicode)

```
mo_elem1 = FindElem(mc_elem1.ToString()).Elem;
mc_elem1.Remove(0, mc_elem1.Length);
if (ml_neg)
{
  Elem mo_tmp = new Elem();
  mo_tmp = !mo_elem1;
  mo_elem1 = mo_tmp;
  mo_elem1.Name = '\u00ac' + mo_elem1.Name;
}
ml_neg = false;
```

Ugyanezt a műveletet elvégezzük a következő változóra is, annyi különbséggel, hogy annak az értéke mostantól az `mo_elem2` változóba kerül, mert ez lesz a második operandus.

Megvan két változónk, megvan a művelet, a változók már negálva vannak, ha úgy szerepelnek a formulában, már csak el kell végezni a műveletet. Ez eléggé hasonlít a klózok eredményeinek összevonására, annyi különbséggel, hogy itt a egyes részműveletek eredményeit is hozzáadjuk a képernyőn alapértelmezetten nem látható részeredményeket tartalmazó panelhez.



4.1.4. ábra A formula változói és (rész)eredményei

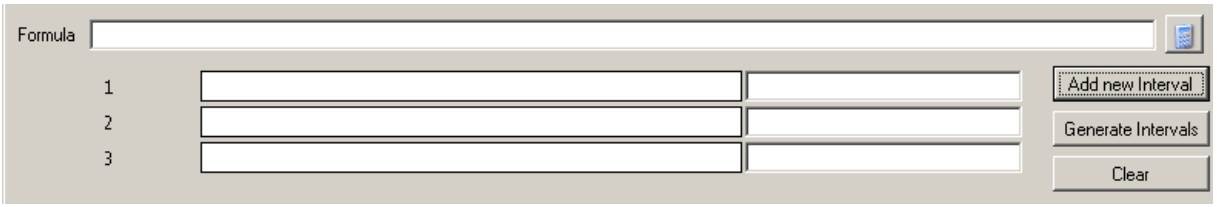
A 4.1.4. ábrán látható a legalsó sötéttel kiemelt sávban a formula megoldása intervallumokkal. Ha az intervallum nem üres, akkor a formula kielégíthető. Ha üres, nincs megoldása a formulának.

Itt említeném meg, hogy ez a `Parse()` és `ParseClause()` függvény nem csak a SAT üzemmód esetén működik. A felhasználó felvehet tetszőleges változókat, értéket adhat nekik, vagy véletlenszerűen generál értékeket, majd beírja a formulát, amit ki akar számolni, és a program kiszámolja a végeredményt. Arra viszont ügyelni kell, hogy a formulában azok a változónevek szerepeljenek, amiket a felhasználó előzőleg fölvett, különben a FirstHalf alapú generált értékekkel fog számolni. A 4.1.5. ábra mutatja, amint a felhasználó beírt értékeket, és a program azokkal számolt a formula kiértékelése során.

4.1.5. ábra A felhasználó által beírt változók, és a velük végzett műveletek

Felhasználó által definiált változókat bevinni a rendszerbe kétféleképpen lehet.

- A felhasználó hozzáad intervallumokat az Add new Interval gombbal, és kézzel beírja az értékeket (4.1.6. ábra)
- A felhasználó hozzáad intervallumokat a Generate Intervals gomb megnyomásával, és akkor a program generál véletlenszerűen intervallumokat. (4.1.7. ábra)



4.1.6. ábra Hozzáadott üres intervallumok

4.1.7. ábra Hozzáadott generált intervallumok

A Clear gomb megnyomásával természetesen törlődnek az intervallum-értékek.

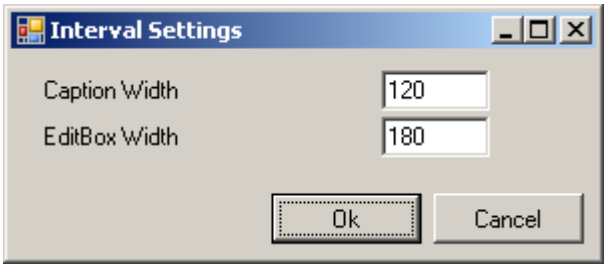
A középen levő ezüstszürke sáv, aminek hivatalosan Splitter a neve, nem a Windows beépített splitter komponense. Ez annyival tud többet annál, hogy a közepén található gomb megnyomásával a felhasználó megjelenítheti vagy eltüntetheti a részeredményeket, nem kell bajlódni az egérrel történő húzogatással. Ez nem az én művem, hanem az Internetről szabadon letölthető CollapsibleSplitter komponens.

Szükség volt még egy indikátorra, ahol a felhasználó visszajelzést kap a programtól, hogy éppen milyen módot állított be. A 4.1.8. ábrán látható StatusStrip nevű komponens

4.1.8. ábra Indikátor, mely visszajelez a felhasználónak, az éppen beállított üzemmódról

Pár szót ejtenék még a Beállításokról, mely elérhető a Settings menüpont alatt.

A képernyő a 4.1.9. ábrán látható.



4.1.9. ábra A program beállításai

A Caption Width az `IntervalPanel` komponens bal oldalán szereplő felirat helyének szélessége. Ha túl hosszú lenne a felirat, a felhasználó megnövelheti ennek a méretét, vagy csökkentheti, ha indokolatlanul nagy hely áll a felirat rendelkezésére.

Az EditBox Width pedig az `IntervalPanel` jobb oldalán álló edit mező szélessége. Jelentőssége megint csak abban van, hogy a felhasználó maga döntheti el, mi számára a fontosabb, nagyobb területen látni a feliratot, az intervallum-értékek grafikus prezentációja a lényeges, vagy pedig az intervallum-elemek valós értékeire kíváncsi.

A fájl mentésénél Ini formátumot használtam. Elsőre meglepetésként ért, hogy a Microsoft nem implementálta ennek a formátumnak a támogatását a C# programozási nyelvbe, holott ez a Windows egyik alapja, de aztán jobban belegondolva logikus, mivel a C# egy platformfüggetlen programozási nyelv, ami annyit jelent, hogy a lefordított kódot a Java Virtual Machine-hoz hasonlóan egy virtuális számítógép értelmezi és futtatja, így ha Linux vagy MacOS vagy bármilyen más operációs rendszer alá létezik ez a virtuális számítógép, akkor azon az operációs rendszeren is lehet futtatni a programot. Így viszont ez a remekül használható fájlformátum nem támogatott más operációs rendszerek által. Szerencsére van rá mód, hogy a Windows kernel32.dll-ben megírt függvényeket használhassuk. Ezt szerencsére más megtette helyettem, és megírt egy rövid IniFile.cs fájlt, ami tartalmazza ezt a dll hívást, az inifile kreálást, és a section-olvasást, -írást.

```
using System;
using System.Runtime.InteropServices;
using System.Text;

namespace Ini
{
  /// Create a New INI file to store or load data
  public class IniFile
  {
    public string path;

    [DllImport("kernel32")]
    private static extern long WritePrivateProfileString(string section,
        string key, string val, string filePath);
    [DllImport("kernel32")]
    private static extern int GetPrivateProfileString(string section,
             string key, string def, StringBuilder retVal,
        int size, string filePath);

    /// INIFile Constructor.
    public IniFile(string INIPath)
    {
      path = INIPath;
```

```
    }
    /// Write Data to the INI File
    public void IniWriteValue(string Section, string Key, string Value)
    {
      WritePrivateProfileString(Section, Key, Value, this.path);
    }

    /// Read Data Value From the Ini File
    public string IniReadValue(string Section, string Key)
    {
      StringBuilder temp = new StringBuilder(255);
      int i = GetPrivateProfileString(Section, Key, "", temp, 255, this.path);
      return temp.ToString();

    }
  }
}
```

Mindössze ennyi, és használható a jól megszokott ini formátum az adatok mentésére.

A struktúra, amit használok, meglehetősen egyszerű.

Lementem a programmódot, a formulát, és ha a program módja nem SAT, akkor a változókat is.

```
[Elems]
Mode=4
Formula=(!1o2o3)a(2o!4o5)a(1o!2o4)
Count=5
0=0.5
1=0.25 0.75
2=0.125 0.375 0.625 0.875
3=0.0625 0.1875 0.3125 0.4375 0.5625 0.6875 0.8125 0.9375
4=0.03125 0.09375 0.15625 0.21875 0.28125 0.34375 0.40625 0.46875 0.53125
0.59375 0.65625 0.71875 0.78125 0.84375 0.90625 0.96875
[Names]
0=1
1=2
2=3
3=4
4=5
```

Itt látható egy példa egy elmentett fájlról. Az `[Elems]` szekcióban található a program működési módja, a formula, ha szükséges, a változók száma, és azok intervallum-értékei, valamint a `[Names]` szekcióban az egyes változók címkéi, vagy feliratai.

4.3 A fejlesztés során használt fejlesztőkörnyezet, alkalmazott konvenciók

Korábban már említésre került, hogy a program Microsoft Visual Studio 2008 Express Edition fejlesztő környezetben készült, mely tanulásra, magánhasználatra ingyenes és szabadon letölthető, ha a felhasználó rendelkezik Microsoft Live hozzáféréssel, ami megint

csak ingyenes. Az alkalmazott .Net FrameWork a 2.0-ás változat. Azért döntöttem emellett, mert ez nagyobb valószínűséggel van a felhasználók számítógépén feltelepítve, mint a legújabb 3.5-ös verzió, azonkívül semmi okot nem találtam, ami miatt a legfrissebb verziót kelljen használnom.

A programozás során alkalmazott névkonvenció megfelel a Hungarian notation néven elhíresült programozási névkonvenciónak. Mint oly sok mindennek, ennek is van magyar vonatkozása, kitalálója a magyar származású Charles Simonyi, aki dolgozott a Xerox PARC nevű cégnél, később pedig vezető programtervező lett a Microsoftnál.

A paraméterek p-betűvel kezdődnek, a lokális változók m-el. A privát változók y, a publikus változók x előtagot kapnak.

A második karakter utal a változó típusára:

- n numeric, szám
- l logical, Boolean
- c character, karakter vagy string
- o object, összetett objektumok

vagyis `ParseClause(string pc_s)` függvény bemenő parameter pc_s. p, mint paraméter, s, mert string, és végül a neve.

Összegzés

Remélem, sikerült a dolgozatban bemutatnom, mennyire hatékony az intervallum-értékű számítás, az összes logika szimulálható vele, méghozzá vizuálisan, ami segíthet megérteni a logikát a kevésbé szakértő egyéneknek is, vagy a gyermekeknek a tanulában. De kíválóan reprezentálhatóak vele grafikusan a különféle számítások, mint például a SAT, ráadásul nem csak szemlélteti a megoldást, hanem egy nagyon hatékony algoritmus, amihez ráadásul lineáris számú intervallum-érték elegendő.

Ami a programot illeti, a későbbiekben szeretném képessé tenni a PSpace-teljes QSAT probléma megoldására is, megpróbálom később a műveleti sorrendet is implementálni, valamint nem tartom elképzelhetetlennek azt sem, hogy az oktatásban a tanárok prezentációkra, a diákok tanulásra és a feladatok ellenőrzésére felhasználják.

Irodalomjegyzék

Logic and Theory of Algorithms - Fourth Conference on Computability in Europe 2008 –Dr. Benedek Nagy and Ákos Tajti: Solving Tripartite Matching by Interval-valued Computation in Polynomial Time

Logical Approaches to Computational Barriers – Second Conference on Computability in Europe 2006 – Dr. Benedek Nagy and Sándor Vályi: Solving a PSPACE-Complete Problem by a Linear Interval-Valued Computation

Dr. Benedek Nagy: Reasoning by Intervals

Dr. Benedek Nagy: A General Fuzzy Logic Using Intervals

Dr. Benedek Nagy and Sándor Vályi: Visual reasoning by generalized interval-values and interval temporal logic

Dr. Benedek Nagy and Sándor Vályi: Interval-valued Computations and their Connection with PSPACE

Dr. Benedek Nagy: An Interval-valued Computing Device

Pásztorné Varga Katalin – Várterész Magda: A matematikai logika alkalmazásszemléletű tárgyalása

http://en.wikipedia.org/wiki/Heyting_algebra

http://hu.wikipedia.org/wiki/Boole-algebra

http://hu.wikipedia.org/wiki/G%C3%B6del