Debreceni Egyetem

Informatika Kar

C++ ALAPÚ MPS FELDOLGOZÓ ALKALMAZÁS

Témavezető:

Bekéné Rácz Anett

Egyetemi tanársegéd

Készítette:

Mátyus Veronika

Gazdaságinformatikus

Debrecen

2011

Tartalom

1 Bevezetés

„This is not maths in theory,

but math in the real world.”

-OR Society [13.]-

Az alkalmazott tudományoknak sok ága ismert, attól függően, hogy a világ mely része kiemelkedően fontos az adott gyakorlati tevékenység szempontjából. Alkalmazott matematika esetén az alaptudomány a matematika. Az operációkutatás pedig az alkalmazott matematikának, azaz ága, ami bizonyos feladatok és eljárások optimalizálásával foglalkozik [14.].

E tudományág a második világháborút követően terjedt el a gazdaságban, mely hatására egyre több és eltérőbb területeken alkalmazzák. Népszerűségét döntést segítő algoritmusainak köszönheti, melyek első lépésben a problémát inicializálják, majd modellezik a feladat lényegét. A felvázolt modell megoldását manapság már számítógépes programok végzik, úgynevezett solverek. Az eljárás sikerességét a kapott eredmény realizálása mutatja.

Egy solver lehetővé teszi, hogy a felállított modellek megoldásához szükséges számítási algoritmusok egy szoftver formájában bármikor rendelkezésünkre álljanak. Ráadásul a számítás folyamatát ez lényegesen megkönnyíti, könnyen, gyorsan újra paraméterezhetőek és pontosak, megbízhatóak a számításaik. A legismertebb solver az Excel Solver bővítménye és a LinGo, melyek elsősorban oktatási célúak és kisebb modelleknél alkalmazhatóak. Az iparban legelterjedtebb professzionális solverek egyike az IBM CPLEX csomagja, melyhez modellező környezetet is kínál (ILOG CPLEX Optimization Studio) [15.] Egy másik jellemzően elterjedt program csomag az úgynevezett GUROBI [16.] Léteznek különböző modellező környezetek, de ezek többsége vagy a CPLEX-et vagy a GUROBI-t használja optimalizáló „motor”-ként.

A Debreceni Egyetemen elindult egy kutatás, amely keretében kifejleszteünk egy új optimalizáló programot, melynek megvalósítása eltér az eddig létező solverektől, a számítások időigénye új technológiákkal csökkenthető. Dolgozatom az új solver beolvasó modulját mutatja be.

Az elkészítést megelőzően tisztázni kellett, hogy az adott tesztfájlok milyen formátumban és hol érhetőek el, így a tárgyalás első fejezetében részletesen ismertetem az adatbázisok

felépítését és az előforduló formátumok lehetőségét. Ezáltal indoklom is miért az MPS formátum lett az alapértelmezett bemeneti formátum.

2 Teszt-adatbázisok

Manapság az internet globális hálózatának köszönhetően eme szükséges adatokat könnyedén letölthetjük egy tesztfájl-adatbázisból. A különböző specifikus probléma csoportoknak külön gyűjteményeik vannak, így szükség estén könnyedén találunk hiteles és jól felhasználható forrásokat. A következő példákat az [1.] oldalról válogattam ki:

Specifikus probléma	Adatbázis
MILP (Mixed Integer Linear Programming) Vegyes egészértékű lineáris programozási feladatok:	MIPLIB 2003, MILPlib, CORAL, MIPlib
MIQP (Mixed Integer Quadratic Programming) Vegyes egészértékű kvadratikus programozási feladatok.	MIQPlib
Stochastic Programming Sztochasztikus programozási feladatok	POSTS
Transportation Programming Szállítási feladatok	Fixed Charge Transportation Benchmark
QP (Quadratic Programming) Kvadratikus programozási feladatok	Maros and Mészáros's problem set
Nonlinear Programming Nemlineáris programozási feladatok	Collection from Tango Project „La Cumparsita"

A való élethez legjobban közelítő eseteket a NETLIB adatbázisban találhatjuk, melyet elérhetünk a [2.] helyről. A fájlok egyedi felhasználhatóságuk szerint vannak tömörítve, általában MPS formátumban. Az eredeti forrás fájlok, a tömörítő és a kicsomagoló program is megtalálható a NETLIB honlapján. Választhatjuk az AMPL formátum letöltését is. A különböző formátumú fájlok átfogó listája látható a COAP (Collection of Optimization Problems) weblapján [3.].

A NETLIB főleg a lineáris programozási (LP) feladatok tesztelésére nyújt lehetőséget. A való életről alkotott LP problémák több különböző forrásból származnak. A példák MPS formátumban érhetőek el, amely része a CUTEr (Constrained and Unconstrained Testing Environment, revisited) által használt SIF (Standard Input Format,) formátumnak. Így a NETLIB bázis további érdekes példákat kínál azoknak, akiknek az optimalizáló csomagjukban CUTEr interfész van.

A [4.] forrásban láthatjuk, hogy a CUTEr egy sokoldalú tesztkörnyezet az optimalizáló és lineáris algebra solverekhez. A csomag tartalmaz egy teszt fájl gyűjteményt, melyet a Fortran 77, Fortran 90/95 (Formular Translating System) és Matlab eszközök segítségével készítettek, hogy támogassa a tervezést, új és már létező solverek összehasonlítást és fejlesztését.

A tesztproblémák úgy nevezett szabványos bemeneti formátumban (SIF) íródtak. Lineáris és nem lineáris problémákat egyaránt tartalmaz. Ebből a formátumból a Fortran 77-be egy dekóder konvertálja át. Más interfészek is alkalmasak a már létező csomagokhoz, mint a MINOS, SNOPT, filterSQP, Knitro, stb. A CUTEr előnye, hogy elérhető számos UNIX platformon, beleértve a Linuxot. Hozzáférhető és könnyen kezelhető heterogén hálózaton is.

Minden feltöltött tesztfájl rendelkezik egy névvel, legalább egy elérhetőséggel (ahonnan letölthető) és egy besorolással.*(lásd az 1.ábrát).*

name	files available	CUTE classification
25FV47	ascii file, gzipped file	LLR2-AN-1571-822
80BAU3B	ascii file, gzipped file	LLR2-AN-9799-2263
ADLITTLE	ascii file, gzipped file	LLR2-AN-97-56
AFIRO	ascii file, gzipped file	LLR2-AN-32-27

1. ábra A NETLIB osztályozása – példa [2.]

2.1 A CUTE osztályozási rendszere

Az osztályozás menetét a [5.] leírás alapján ismertetem. A harmadik oszlopban szereplő osztályozási sémát Hock és Schittkowski dolgozták ki, 1981-ben. Minden egyes problémát az alábbi sztring segítségével jellemzik:

XXXr-XX-n-m

A jelölt értékek egyike sem tartalmazhat üres részt és csak nagybetűvel lehet megadni karakteres értéket. A következőekben ismertetem, mely helyeken mely betűk és egészek interpretálhatóak és azoknak mik a jelentésük.

Az első X a probléma célfüggvényének típusát határozza meg:

Kód	Célfüggvény típusa
N	nincs definiálva
C	konstans
L	lineáris
Q	quadratikus
S	négyzet összeg
O	a fentiek közül egyik sem

A második karakter a feltételek típusát jelöli:

Kód	Kényszer típus
U	a problémának nincsenek kényszer feltételei
X	csak fix változók vannak
B	korlátolt változók
N	egy hálózat szomszédossági mátrixát reprezentálja
L	lineáris feltételek
Q	kvadratikus feltételek
O	egyéb

A harmadik karakter a probléma simaságát jelzi. Két lehetőség van:

Kód	Simaság
R	probléma folytonos, az első és második deriváltja létezik és a végtelenbe tart
I	szabálytalan

A negyedik karakter egy egész szám lesz (r), ami a legmagasabb derivált fokát mutatja. Csak 0, 1, 2 értékeket vehet fel.

A kötőjelet közvetlenül egy betű követi, mely az adott problémának az elsődleges célját és/vagy okát határozza meg.

Kód	Célja/Oka
A	Kísérleti, amely kifejezetten algoritmus - tesztelésre lett kifejlesztve
M	Egy modellezett feladat része, ahol a tényleges megoldást sosem használták a valós életben
R	A megoldást a való életben alkalmazzák, de nem algoritmusok tesztelésére

A következő karakter azt mutatja, hogy a modell tartalmaz-e explicit belső változókat:

Kód	Van benne?
Y	Igen
N	Nem

A második és a harmadik kötőjel közötti szimbólum lehetséges értékei:

Kód	Változók száma
V	a felhasználó választja ki
n	fix, pozitív egész szám

A harmadik kötőjel után a feltételek számát határozzuk meg. A rögzített változókat és az egyedi korlátokat (BOUND) nem ide soroljuk.

Kód	Feltételek száma
V	a felhasználó választ
m	nem negatív egész szám

2.2 Lehetséges formátumok

Ebben a fejezetben szeretném ismertetni a fájl letöltésekor választható leggyakoribb formátumok (az AMPL, GAMS és az MPS) jellegzetességeit és sajátosságait.

AMPL

Angol neve elárulja rendeltetését (A Modelling Language for Mathematical Programming). A formátum hivatalos holapján [6.], elérhetőek a legfontosabb és legfrissebb információk, melyből megtudhatjuk, hogy egyaránt használható lineáris és nem lineáris problémák modelljének megalkotására, akár diszkrét, akár folytonos változóról legyen szó. A Bell Laboratories fejlesztette ki. Előnye a rugalmas és kényelmes kezelésében áll. Gyors és könnyen irányítható. Ráadásul sok solver is szívesen dolgozik vele, mint például: CPLEX, BPMPD, LAMPS, OSL, OPT, stb.

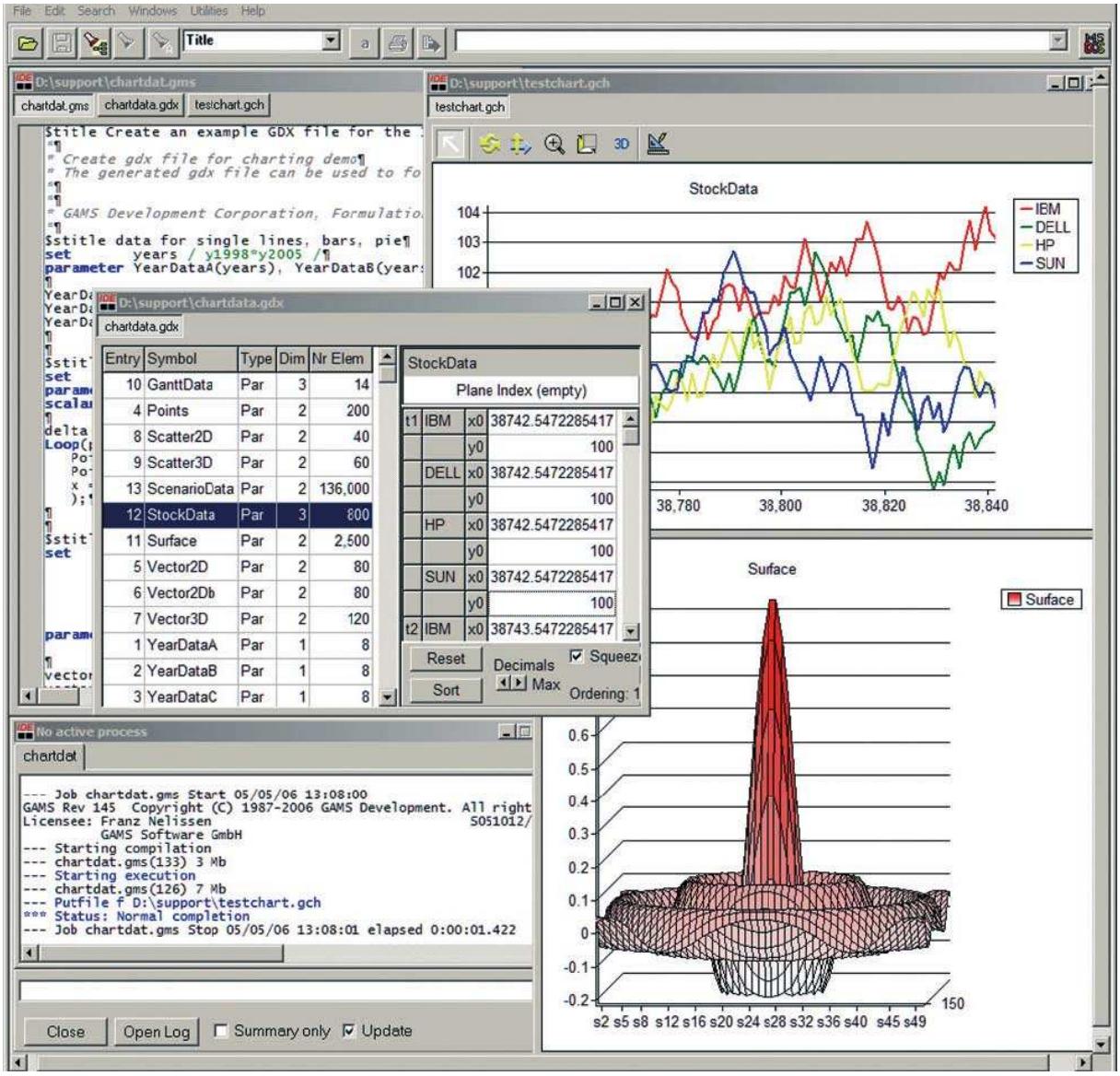
Manapság már léteznek más módozatai is, mint az AT&T Mathematical Programming Language, vagy az Algebraic Modelling Programming Language. Lényegében ugyanazt csinálják, csak más nevet kaptak. A nyelv fejlesztése még ma is történik a világkülönböző pontjain (Ausztráliában, NASA egyik laboratóriumában, Seattle-ben, stb.) *(Forráskód példa a függelékben: Forráskód 1.)*

GAMS

(General Algebraic Modelling System) A használatához szükséges információkat a [8.] ismerteti, mely szerint e formátum nem más, mint egy magas szintű modellező rendszer kifejezetten optimalizálásra és matematikai programozásra fejlesztve. Alkalmazására főleg komplex, nagyméretű és dinamikus modellek esetén kerül sor.

Egyszerű beállításainak köszönhetően inkább a modellezésre lehet összpontosítani. Különösen hasznos a nagy, komplex és sok módosítást igénylő modellek kezelésében. A modellezést rendkívül kompakt és természetes úton végzi. A formulákat a felhasználó könnyen és gyorsan tudja változtatni. Rugalmas és teljes mértékben hordozható. Egyik solverről képes a másikra váltani és nem lineáris problémát konvertálhatunk lineárissá. Könnyű fejleszteni, dokumentálni. Hasonlít a leggyakrabban használt programozási nyelvekhez. Így akinek már van programozási tapasztalata, annak a GAMS használta barátságos. *(Forráskód példa a függelékben: Forráskód 2.)*

Az GAMS rendszere egy felhasználóbarát modellező környezetet ad, melyben kevesebb programozói tapasztalattal rendelkező személyek is könnyen generálhatnak GAMS kódokat: *(lásd a 2. ábra)*



2. ábra GAMS felhasználói felülete [17.]

MPS

(Mathematical Programming System) A [10.] leírás szerint, lineáris programozási és vegyes egészértékű programozási problémák esetén a választás egyértelműen az MPS formátumra esik. Minden kereskedelmi LP solver elfogadja az MPS formátumot és a nyílt forráskódú COIN-OR (Computational Infrastructure for Operations Research) gyűjtemény is támogatja [11.].

Az MPS egy oszlop-orientált megközelítés, melyben a modell összes komponense azonosítót kap, melyek alapján történik a feldolgozás. Olyan régi formátum, hogy még lyukkártyákra tervezték. NETLIB-en található tesztfájlok nagy része csak ebben a formátumban érhető el, ezért esett erre a formátumra a választás. A továbbiakban szeretném részletesebben bemutatni az MPS formátum működési elvét.

3 MPS formátum

A legjobb leírást a [18.] könyvben találtam, melynek leíró logikáját és struktúráját átvettem. A magyarosítás pontosításában a [12.] szakdolgozat segített.

A LP problémák két különböző reprezentációja ismert. A modell készítés utolsó fázisában a modellező rendszerek vagy az egyedi feldolgozásra szánt programok által előállított végső modellt többnyire olyan formátumban tárolják, amelyet más gépek is értelmezni tudnak. Ezt nevezzük külső reprezentációnak. E fájl beolvasása egy solverbe belső reprezentációvá konvertálja a tárolt formátumot.

A leggyakrabban használt külső reprezentáció az MPS formátum. A fejezet további részében ennek felépítését mutatom be.

3.1 A formátum felépítése

Az MPS formátumot 1950-ben az IBM fejlesztette ki, hogy a matematikai programozási feladatok megadására egységes tárolási formát használjanak. Ekkoriban minden szükséges adatot lyukkártyákon tároltak. Minden kártyán egyszerre 80 karaktert (betűt, decimális számot, szimbólumot) lehetett ábrázolni. A természetes bemeneti egység 80 karakter hosszú volt.

Az egységes értelmezés érdekében több szabványt is jóváhagytak, melynek eredményeként a formátum túl merevvé vált. Érdekes, hogy pont e tulajdonsága miatt vált ennyire népszerűvé. Kompatibilitásának köszönheti, hogy még ma is használják, hiszen minden komolyabb rendszer könnyen fordítja és a solverek által elfogadott legalább 2 külső reprezentációból, az egyik általában MPS. Az sem szokatlan, hogy bizonyos programok egy közbeeső fájlt készítenek és a solver azt használja közvetlen bemenetnek.

Működési elve semmit sem változott az évek során. A nem nulla értékeket csak az előfordulásukkal együtt tárolja, így csökkenti a beolvasott adatok mennyiségét. A rekordokat 6 szigorúan pozícionált mezőre tagolja:

	Mező -1	Mező - 2	Mező -3	Mező -4	Mező -5	Mező -6
Hely	2-3	5-12	15-22	25-36	40-47	50-61
Tartalom	Mutató	Név	Név	Érték	Név	Érték

Felépítése függőlegesen strukturált és szekciókra tagolható. Minden szekció egy szekció azonosítóval kezdődik, amely az első pozícióról indul (1 pozíció = 1 karakter) és az alábbi sztringek egyikét tartalmazza:

- NAME (a probléma neve)
- ROWS (sorok, feltételek)
- COLUMNS (oszlopok, változók)
- RHS (right-hand side - jobb oldali értékek)
- RANGES (tartományok - opcionális)
- BOUNDS (korlátok - opcionális)
- ENDATA (végjel)

A szekciók ebben a sorrendben követik egymást. Eltérő hosszúságukból adódóan más pozícionálást igényelnek.

Az MPS formátumot egyetlen feladat tárolására tervezték. Minden egységnek (korlátnak, változónak, stb.) egy maximum 8 karakteres egyedi azonosítóval kell rendelkeznie (név). Ezzel a szimbolikus azonosítással a cél, a tartomány, és a különböző korlát függvények kiválóan megkülönböztethetők.

Ezeket a szekciókat az alábbi módon definiáljuk:

NAME

(1-4 pozíció) Ez a probléma definíció első rekordja. Ezt megelőzően minden információt ignorál a beolvasó. Ugyanezen sorban a matematikai programozási probléma neve következik a 15-22. pozición. Az implementáció engedi, hogy a cím bármilyen hosszúságú legyen, akár a kártya végéig is tarthat

ROWS

(1-4 pozíció) Ebben a szekcióban minden kényszerfeltétel azonosítóját (nevét) és relációját definiáljuk. A későbbiekben (pl.: a COLUMNS szekcióban) csak az itt megjelent egyenletekre hivatkozhatunk. Az alábbi jelölések használhatóak:

Feltétel típus	Kód	Jelentés
≤	L	kisebb vagy egyenlő
≥	G	nagyobb vagy egyenlő
=	E	egyenlő
Cél	N	cél függvény
Szabad	N	nem kötött feltétel

A kód a 2-3. pozíciókon jelenhet meg. A névnek a Mező - 2-ben kell megjelennie. A sor azonosítója nem tartalmazhat speciális karaktereket (szóköz, +, -, =, *).

COLUMNS

(1-7. pozíció) E szakaszban 2 dolog is történik. Először definiáljuk a változók neveit, majd meghatározzuk az együtthatóit a különböző feltételekben, beleértve a cél függvényt is. Általában csak a nem nulla értékek vannak megadva, de előfordulhat, hogy néhány nulla érték is megjelenik, így nem garantált, hogy minden érték nem nulla.

Az elemeket bármilyen sorrendben megadhatjuk, de hibának számít, ha megadjuk egy oszlop elemeit, aztán egy másikét, majd újra az előzőét.

Az adatok az alábbi struktúrát követik:

- Mező-2: Oszlop neve (maximum 8 karakter)
- Mező-3: Sor neve (melyeket már definiáltunk a ROWS szekcióban)
- Mező-4: Az együttható értéke

Van lehetőségünk, hogy ugyanazon változóhoz 2 elemet is definiáljunk egy rekordban, ez a lehetőség választható. Ebben az esetben a folytatás:

- Mező-5: Sor név
- Mező-6: Együttható értéke

Azaz [változónév] [feltételnév] [együttható] [feltételnév] [együttható] módon.

Ez a struktúra ismétlődik a következő szekcióig.

RHS

(1-3 pozíció) Ez a szekció adja meg a jobb oldali korlátokat. Minden vektornak meg van adva a saját neve. Az adat felépítése a következő:

- Mező-2: RHS vektor neve
- Mező-3: Sor név
- Mező-4: RHS érték
- Mező-5-t és Mező-6-t használhatjuk az előzőhöz hasonló módon. A sorrend nem kötött.

RANGES

(opcionális – 1-6 pozíció) Egy tartományt az alábbi módon definiálhatunk:

$$L_i \leq \sum_{j=1}^{n} a_j^i x_j \leq U_i$$

ahol L_i és U_i véges értékek. A tartomány értéke $R_i = L_i - U_i$. Az L_i és U_i értékek közül egy definiálva van az RHS szekcióban, melyet a táblázatban b_i –vel jelölök. Az R_i értékét a RANGE részben adjuk meg, a ROW részben megadott tulajdonságokkal együtt. Az L_i és U_i korlátok a következőek lehetnek:

Row típus	R_i előjele	Alsó határ	Felső határ
L(≤)	+ or -	b_i - $\|R_i\|$	b_i
G (≥)	+ or -	b_i	b_i + $\|R_i\|$
E (=)	+	b_i	b_i + $\|R_i\|$
E (=)	-	b_i - $\|R_i\|$	b_i

Az adattagok definiálása az RHS részben hasonló módon történik. Az egyetlen különbség a RANGES sztring a név mezőben.

- Mező-2: tartomány típusú megkötés neve
- Mező-3: Sor név
- Mező-4: Tartomány érték (R_i)

- Mező-5-t és Mező-6-t hasonló módon használhatjuk, mint a COLUMNS-ban. A kényszer tartományok értékének megadási sorrendje a RANGES vektoron belül nem kötött.

BOUNDS

(opcionális – 1-6 pozíció) A korlátok definíciója kicsit eltér az eddigiektől. Egyetlen korlát névvel az alsó és felső korlátot is definiálni tudjuk. Azaz egy korlátnévnek több mint 1 értéke van, például, amikor véges alsó és felső határt definiálunk. Az MPS formátumban sok alapértelmezett érték van, melyet nem kell újra meghatározni. Ugyanakkor hiányzik belőle minden eszköz, hogy új típusokat alkossunk.

Az adatok struktúrája a következő:

- Mező-1: Korlát típus
- Mező-2: BOUNDS vektor neve
- Mező-3: Változó név
- Mező-4: Korlát értéke
- Mező-5-t és Mező-6-t nem használjuk, üresnek kell lennie, esetleg kommentet tartalmazhat.

A korlát típusa 2 karakter hosszú:

Típus	Leírás	Jelentés
LO	Alsó korlát	$l_j \leq x_j (\leq +\infty)$
UP	Felső korlát	$(0 \leq) x_j \leq u_j (\leq +\infty)$
FX	rögzített értékű	$x_j = l_j$ (amely = u_j)
FR	korlátlan értékű	$-\infty \leq x_j \leq +\infty$
MI	alulról nem korlátos	$-\infty \leq x_j (\leq 0)$
PL	felülről nem korlátos	$(0 \leq) x_j \leq +\infty$

Az alapértelmezett értékek zárójelben vannak megadva.

ENDATA

(1-6 pozíció) Egyedül áll és ez jelzi az LP probléma végét. Az utána következő karakterek nem kerülnek beolvasásra.

Az MPS definíciója során a legsajátságosabb dolog a célfüggvény irányának a megadása, ugyanis nincs állandó alapértelmezett iránya, esetleg a kommentek között találhatunk róla valamilyen információt. Ám egyes solverek maximumot, míg mások minimumot számolnak. Előfordulhat, hogy számítás előtt rákérdez a program. Mindenesetre könnyű konvertálni az egyikből a másikba. Egyszerűen a célfüggvény együtthatókat be kell szorozni -1-gyel.

3.2 MPS-ből LP

A fenti leírást követve bármilyen MPS formátumban megadott LP feladat modelljét könnyen felírhatjuk matematikai szimbólumokkal is. Például a [10.] linken található MPS fájlból

```
NAME          TESTPROB
ROWS
 N  COST
 L  LIM1
 G  LIM2
 E  MYEQN
COLUMNS
    XONE      COST                 1   LIM1                 1
    XONE      LIM2                 1
    YTWO      COST                 4   LIM1                 1
    YTWO      MYEQN               -1
    ZTHREE    COST                 9   LIM2                 1
    ZTHREE    MYEQN                1
RHS
    RHS1      LIM1                 5   LIM2                10
    RHS1      MYEQN                7
BOUNDS
 UP BND1      XONE                 4
 LO BND1      YTWO                -1
 UP BND1      YTWO                 1
ENDATA
```

a következő LP problémát írhatjuk fel:

Célfüggvény:	$X + 4*Y + 9*Z \rightarrow \min$
Lim1:	$X + Y <= 5$
Lim2:	$X + Z >= 10$
Myeqn:	$-Y + Z = 7$
Korlátok:	$X \leq 4, -1 \leq Y \leq -1$

4 MPS-fájl feldolgozó modul

Ebben a részben bemutatom az általam készített beolvasó tervezésének és létrehozásának folyamatát, melynek funkciója, hogy az MPS formátumban elérhető adatokat beolvassa a számítógép memóriájába, ezzel előállítva a modell belső reprezentációját a későbbi feldolgozáshoz. A program C ++ nyelven íródott.

4.1 Memória kezelés

Korábban már bemutattam hogyan épül fel egy LP modell egy MPS formátumból *(lásd 5.2 fejezet*). Az ott felépített LP modellen szemléltetem, hogy az adatok tárolására milyen adatszerkezetet választottam.

Első felvetés az volt, hogy tároljuk az adatokat egy mátrixban, tömbök segítségével. Például az alábbi LP modellből a következő táblázatot állíthatjuk elő:

Célfüggvény:	X + 4*Y + 9*Z → min
Lim1:	X + Y <= 5
Lim2:	X + Z >= 10
Myeqn:	-Y + Z = 7
Korlátok:	X ≤ 4, -1 ≤ Y ≤ -1

Változók / Feltételek	X	Y	Z
Célfüggvény	1	4	9
Lim1	1	1	0
Lim2	1	0	1
Myeqn	0	-1	1

Mely táblázatból a következő mátrixot hozzuk létre:

$$M = \begin{pmatrix} 1 & 4 & 9 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & -1 & 1 \end{pmatrix}$$

Minden egyes feltétel és változó a neve alapján kap egy indexet, melyek segítenek a számítógépen ábrázolni és tárolni a mátrixot:

Változók / Feltételek		X	Y	Z
		0	1	2
Célfüggvény	0	1	4	9
Lim1	1	1	1	0
Lim2	2	1	0	1
Myeqn	3	0	-1	1

Az így kiosztott indexszámok segítségével a mátrix elemei közvetlenül elérhetővé válnak:

$$m_{0\,0} = 1$$

$$m_{3\,1} = -1$$

$m_{i\,j} = c_{i\,j}$, ahol

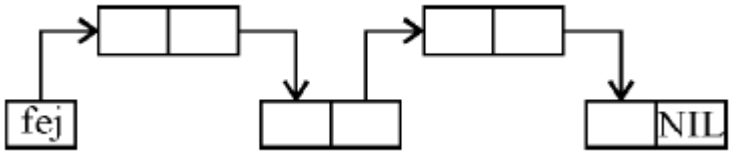
- $m_{i\,j}$ az M mátrix i-edik sorának j-edik oszlopában lévő eleme
- $c_{i\,j}$ az i-edik feltétel j-edik változójának az együtthatója (coefficient).

A következő probléma a modellek méretéből adódott. Míg a fent leírt példa tárolásához elegendő egy 4x3 mátrix, addig a NETLIB adatbázisban találkoztam olyan példákkal melyek egy 5167x2171-es mátrixba fértek volna bele. Az együtthatók 2 dimenziós tömbben tárolása memória pazarlást okozott volna, a nulla elemek miatt. Ugyanis már ebben a példában is az M mátrixban háromszor szerepel a nulla elem. A NETLIB-ben és való életben jellemzően olyan problémákkal találkozunk, melynek feltétel mátrixa egy ritka mátrix.

Nulla együttható érték akkor keletkezik, amikor az adott változó nem szerepel a szóban forgó feltételben. Nagyobb modellek esetén a mátrixban előforduló nullák gyakorisága is nagyobb, így az adatok közül csak a nem nulla elemek kerülnek tárolásra. Azaz az MPS fájlból beolvasott adatokat inkább valamilyen ritka mátrix reprezentációban ajánlatos tárolni.

A ritka mátrix tömbös megvalósításához pedig előre szükséges tudni, hogy mennyi nem nulla elem fordul elő a modellben.

Ezen okokból adódóan célszerűnek láttam egy dinamikus adatszerkezetet választani, méghozzá az egyirányú láncolt listát:

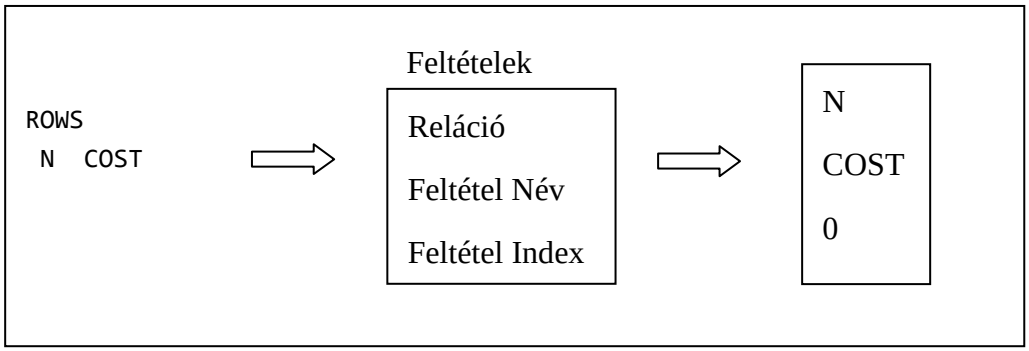


3. ábra Egyirányba láncolt lista (linked list)[29.]

Ez után el kellett döntenem, hogy hány láncolt listára lesz szükség.

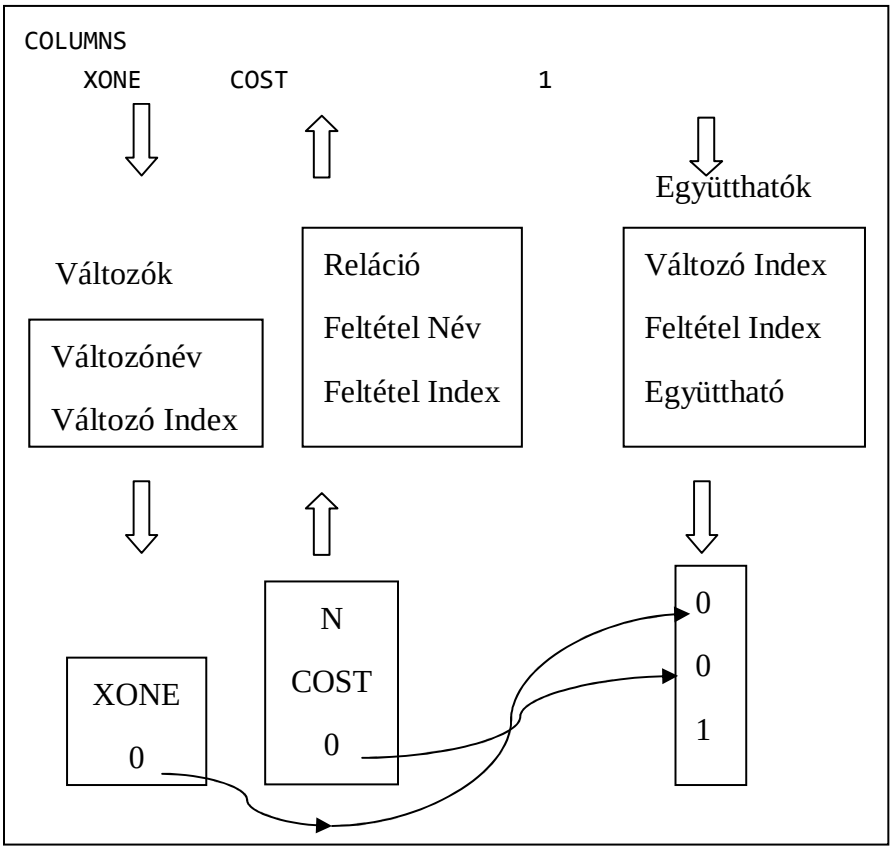
A ROWS szekcióban definiálódnak a feltételek, mely részleg befejeztével tudjuk, hogy hány sorból áll a mátrix.

```
ROWS
 N  COST
 L  LIM1
 G  LIM2
 E  MYEQN
```



A COLUMNS szekcióban találkozunk először a változók neveivel, melyek segítségével azonosítjuk az oszlopot. Amennyiben a változót egy korábban már tárolt feltétel név követi, akkor a listában tárolt index segítségével meghatározzuk a mátrix sorát. Az így kapott indexek egyértelműen definiálják, hogy az olvasásban soron következő érték melyik együtthatóra vonatkozik.

```
COLUMNS
    XONE      COST                1   LIM1                1
    XONE      LIM2                1
```



Azaz 3 struktúrát hozunk létre:

1. Condition (Feltételek):
2. Variable (Változók)
3. Coefficient (Együtthatók)

Az MPS formátumú reprezentáció értelmezésére a legkézenfekvőbb megoldást a formális nyelvek nyújtották.

4.2 Formális nyelvek

A [19.]-ben leírt történelmi háttér és fogalmi magyarázatok segítettek megérteni az alap összefüggéseket. A könyv szerint az igény kielégítése, hogy az emberek helyett teljes egészében gépek végezzék a fordítási feladatokat, már az 1950 –es évektől foglalkoztatja az

emberiséget. Ez 1956-ban egy új tudomány, a matematikai nyelvészet megjelenéséhez vezetett. Az új terület lényege a gépies szellemi munka kiváltása.

A formális szó a külső megjelenésre, alaki és formai jellemzőkre utal, mely aspektusban a látható, nyilvánvaló dolgokat elemezzük és közben a tartalmi összefüggések háttérbe szorulnak. A formális nyelvek elmélete nevéhez hűen az írott szöveg számítógépes értelmezésével foglalkozik.

Algoritmusa akkor indul el, ha már előre definiáltuk a véges karakterkészletét, a szintér elemeit. Az így képzett halmazt nevezzük a nyelv alfabetájának. Ez alapján bináris és szöveges fájlokat is értelmezhetünk. A bináris fájlok lehetnek például a memória adatszerkezetek leképzései, a szöveges fájlok esetén megkülönböztetjük a természetes (magyar, német, angol, stb.) és mesterséges (Java, C, stb.) nyelveket.

A mesterséges nyelvek az ember és gép közti kommunikációt szolgálják. Nyelvtani szabályaik egyszerűek, fogalmi körük erősen szűkített, mondataik egyértelműek és általában nem ismernek kivételeket. A gépi nyelvek fordítása ezen okok miatt sokkal könnyebben kivitelezhető.

A nyelvtan, azaz a grammatika leírja a szavak és speciális karakterek lehetséges sorrendjét, szerkezetét és törvényszerűségeit. A grammatikát formálisan egy négyes határozza meg

$$G = (N, \sum, P, S), \text{ ahol}$$

- N a grammatikai szimbólumok véges halmaza
- $\sum$ az alfabeta, a nyelv karakterkészletének véges halmaza
- P levezetési szabályok összessége, szintén véges halmaz
- S mondatszimbólum

A grammatikai szimbólumok segítségével definiálhatjuk a nyelvet és általuk levezethetjük, vagy akár generálhatjuk is egy nyelv mondatait.

Azon jelsorozatokat, melyekben a $\sum$ elemein kívül grammatikai szimbólumok is szerepelnek mondatszerű formáknak nevezzük.

Mondatszerű formák egymásutánja egy levezetést alkot. A levezetési szabályok a következő alakúak:

$$\alpha \rightarrow \beta$$

ahol α és β két jelsorozat, a → azt jelenti, hogy a baloldali formából a nyelvtan levezetési szabályai szerint a jobboldali pontosan egy lépésben levezethető. α-nak legalább egy grammatikai szimbólumot tartalmaznia kell, míg β tetszőleges. Azaz egy szabály csak akkor alkalmazható, ha a karaktersorozat egy részsorozata megegyezik a helyettesítési szabály baloldalával. Egyezés esetén az adott sorozat lecserélődik a szabály jobb oldalára. A levezetés történhet több, meg nem határozott számú lépésben.

Minden levezetés egy speciális, egyetlen grammatikai szimbólumból kezdődik, a mondat szimbólumból. Minden levezetés innen indul el.

Amikor a kapott jelsorozaton egyetlen levezetési szabály sem alkalmazható, azaz csakis nyelvi szimbólumokat tartalmaz, a mondatszerű forma egyben mondat is. Ezeket nevezzük a nyelv terminális szimbólumainak, vagy a program tokenjeinek.

Amennyiben a mondatszerű forma tartalmaz grammatikai szimbólumot, akkor a levezetést még folytatni kell, a mondat generálása még nem fejeződött be, nem terminálódott. Ezt a formát nem terminális szimbólumoknak nevezzük.

A magyar nyelv egy egyszerűbb változatát például az alábbi szabályok definiálhatják:

<Szöveg> → {**<Mondat>**}
<Mondat> → **<Alany>** + **<Cselekvés>** + **<.>**
<Cselekvés> → **<Állítmány>**
<Cselekvés> → **<Tárgy>** + **<Állítmány>**
<Alany> → *<Főnév>*
<Tárgy> → *<Főnév>* + *<t>*
<Állítmány> → *<Ige>*

Jelmagyarázat: A terminális és a nem terminális szimbólumokat < > jelek közé tesszük és a nem terminálisok vastag betűvel vannak szedve. A { } az ismétlődésre utal. A + jel az egymás utáni felsorolást jelenti. A [] szögletes zárójel az opcionalitás jele, azaz a benne foglalt fogalom egyszer vagy egyszer sem jelenik meg. *(A szabályok és az ábra értelmezése a [20.]-ből származik)*

A fenti jelölés rendszert alkalmazva az MPS fordítóra az alábbi szabályokat fogalmaztuk meg:

<MPSFile> → <Name> + <Rows> + <Columns> + <Rhs> + <Ranges> + <Bounds> + <Endata>

<Name> → <*NAME*> + <*Sztring*>

<Rows> → <*ROWS*> + {<RowsData>}
[<RowsData> → <*E*> + <*Sztring*>]
[<RowsData> → <*N*> + <*Sztring*>]
[<RowsData> → <*L*> + <*Sztring*>]
[<RowsData> → <*G*> + <*Sztring*>]

<Columns> → <*COLUMNS*> + {<ColumnsData>}
<ColumnsData> → <*Sztring*>+<*Sztring*>+<*Float*>[+<Sztring>+<Float>]

<Rhs> → {RhsData}
<RhsData> → <Sztring> + <Sztring> + <Float>
<Ranges> → <RANGES> + {<RangesData>}
<RangesData> → <Sztring> + <Sztring> + <Float>
<Bounds> → <BOUNDS> + {<BoundsData>}
[<BoundsData> → <UP> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <LO> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <FX> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <FR> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <MI> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <PL> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <BV> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <UI> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <LI> + <*Sztring*> + <*Sztring*> + <*Float*>]
[<BoundsData> → <SC> + <*Sztring*> + <*Sztring*> + <*Float*>]
<Endata> → <ENDATA>

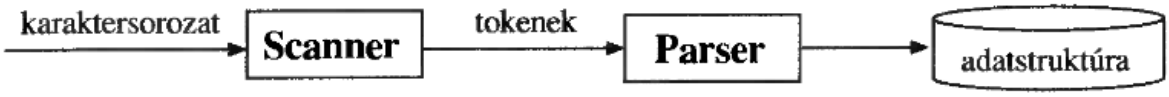
Az MPS feldolgozóban csak a bekeretezett szabályok kerültek implementálásra. A BOUND, a RANGES és az RHS szekció kezelését a későbbiekben tervezzük megvalósítani.

4.3 A program működése

A tervezés szakasz befejeztével az implementáció következett, amely alapjaiban a [20.] munka szerkezetét és gondolatmenetét követi. Ebben a fejezetben ezt fogom bemutatni.

A kód felépítése az általános beolvasók struktúráját követi, melyet a 4. ábra szemléltet. A karakterekes bementi állományon először a Scanner (lexikális elemző) hajt végre műveletet. Ez az értelmezés első lépése, mely során a szavakat és lexikális szimbólumokat felismeri, illetve a felesleges részeket (megjegyzések, üres karakterek) eldobja. A lexikális elemző kimenete a beazonosított terminális szimbólum (token) és annak tartalma. A token egyértelműen azonosít egy speciális jelet, a tartalom a feldolgozáshoz pontosítja, hogy miről

van szó. A tokeneket az értelmező (Parser) dolgozza fel, elvégzi a nyelvtani helyettesítéseket, továbbá ellenőrzi, hogy a mondat helyes eleme-e a nyelvnek.



4. ábra Egy általános beolvasó felépítése [17.]

MPS formátum beolvasása (Scanner)

A beolvasó programhoz a tokeneket előállító Scannert és a nyelvtani szabályokat értelmező Parser-t kell megírnunk. A Scanner feladata a bemeneti karaktersorozat összetartozó elemeinek (tokeneknek) az azonosítása. Egy token lehet egy speciális karakter (például *), egy kulcsszó (if, for,stb.), egy konstans(123), vagy akár egy változó vagy függvény neve. Az MPS formátumban a kulcsszavak a következőek: NAME, ROWS, E, N, L, G, COLUMNS, RHS, RANGES, BOUNDS, UP, LO, FX, FR, MI, PL, BV, UI, LI, SC.

A lehetséges tokeneket egy felsorolás típussal adjuk meg:

```
enum Token {
        EOF_TOKEN,
        MPS_NAME_TOKEN,
        ROWS_TOKEN,
        EQUALITY_ROWS_TOKEN,
        NON_CONSTRAINING_ROWS_TOKEN,
        LESS_THAN_ROWS_TOKEN,
        GREATER_THAN_ROWS_TOKEN,
        COLUMNS_TOKEN,
        RHS_TOKEN,
        RANGES_TOKEN,
        BOUNDS_TOKEN,
        UPPER_BOUND_TOKEN,
        LOWER_BOUND_TOKEN,
        FIXED_TOKEN,
        FREE_TOKEN,
        FREE_NEGATIVE_TOKEN,
        FREE_POSITIVE_TOKEN,
        BINARY_TOKEN,
        UPPER_INTEGER_TOKEN,
        LOWER_INTEGER_TOKEN,
        SEMI_CONTINOUS_TOKEN,
        ENDATA_TOKEN,
```

```
        NUMBER_TOKEN,
        WORD_TOKEN
};
```

A kulcsszavakat pedig táblázatok segítségével kapcsoljuk a token-azonosítókhoz:

```
//-------------------------------------------------------------
// Kulcsszavak táblázata
//-------------------------------------------------------------
Keyword keywords[] = {
       { "NAME",      MPS_NAME_TOKEN },
       { "ROWS",      ROWS_TOKEN },
       { "E",         EQUALITY_ROWS_TOKEN },
       { "N",         NON_CONSTRAINING_ROWS_TOKEN },
       { "L",         LESS_THAN_ROWS_TOKEN },
        { "G",             GREATER_THAN_ROWS_TOKEN },
        { "COLUMNS",       COLUMNS_TOKEN },
        { "RHS",   RHS_TOKEN },
        { "RANGES",        RANGES_TOKEN },
        { "BOUNDS",        BOUNDS_TOKEN },
        { "UP",    UPPER_BOUND_TOKEN },
        { "LO",    LOWER_BOUND_TOKEN },
        { "FX",    FIXED_TOKEN },
        { "FR",    FREE_TOKEN },
        { "MI",    FREE_NEGATIVE_TOKEN },
        { "PL",    FREE_POSITIVE_TOKEN },
        { "BV",    BINARY_TOKEN },
        { "UI",    UPPER_INTEGER_TOKEN },
        { "LI",    LOWER_INTEGER_TOKEN },
        { "SC",    SEMI_CONTINOUS_TOKEN },
        { "ENDATA",        ENDATA_TOKEN }
};
```

A Scanner az elválasztó jelekig (szóköz, tabulátor, új sor) gyűjti az egymás utáni karaktereket, majd megvizsgálja, hogy kulcsszó-e, vagy pedig a programozó által megadott név. A Scanner mindig az aktuális karaktert tárolja, illetve előreolvas a fájlban és megnézi, hogy mi a következő karakter. Ezáltal ismeri fel, hogy az aktuális karakter a token utolsó karaktere. Amíg a token nem áll össze, a hozzá tartozó karakterek a token_buffer karaktertömbbe kerülnek.

A Match() eljárás az aktuális tokent a várt tokennel veti össze, illeszkedés esetén a következő tokenre lép, eltéréskor viszont hibát jelezve leáll.

A GetReal() az illeszkedésvizsgálat speciális formája, amely lebegő pontos számot vár és a számként értelmezhető karakter sorozatot számmá alakítja át. Elemző programunk Scanner osztályának lelke a GetToken() függvény, amely a fájlból a következő azonosítható karaktersorozattal, valamint annak megfelelő tokennel tér vissza. Az eljárás addig olvas, amíg egy összetartozó egységet fel nem ismer.

A szóközök átlépése után, először a speciális karaktereket vizsgáljuk. Jelen esetben csak egy speciális karakter van, mégpedig a csillag (*), mely a megjegyzés kezdetét jelöli és az adott sor végéig tart:

```
virtual int IsCommentBegin( char c ) { return c == '*'; }
virtual int IsCommentEnd( char c)  { return c == '\n'; }
```

Az illesztés a GetToken() eljárásban található:

```
// Megjegyzéseket eldobjuk
if ( IsCommentBegin( curr_char ) ) {
        curr_char = Read( );
        while ( !IsCommentEnd( curr_char ) ) curr_char = Read( );
}
```

A betűvel induló elemek karaktereit addig gyűjtjük, amíg nem betűt kapunk (például szóközt) és ekkor megvizsgáljuk, hogy az idáig összeálló karaktersorozat azonos-e valamelyik kulcsszóval. Ha nem, akkor csak változó név lehet, amelyet a GetName() eljárással kérdezhetünk le.

A Scannner kimenete a tokenek sorozata, amelyet a Parser a nyelvtani szabályoknak megfelelően dolgoz fel. Az értelmezőt rekurzív ereszkedő stratégiával készítettük el. Ez azt jelenti, hogy minden nyelvtani szabályhoz egy függvényt írunk, amely a jobb oldali elemet illeszti. A terminális illesztése a Scanner-től kapott token és a nyelvtani szabály alapján várható token összehasonlításából áll. Ha megegyeznek, minden rendben van, továbblépünk. Eltérés esetén a fájl nem felel meg a nyelvtani szabályoknak.

Először az első nyelvtani szabály elemző rutinját írtuk meg. Egy MPS fájlban feltételeket, relációkat, változókat és együtthatókat sorolhatunk fel.

```
//-----------------------------------------------------------------
// MPS Parser
//-----------------------------------------------------------------
void MPSParser :: ParseFile( ) {  // <Name> + <Rows> + <Columns> + <Rhs>
//-----------------------------------------------------------------
        GetToken();
        for( ; ; ) {
                switch( GetCurrentToken() ) {
                        case MPS_NAME_TOKEN:            ParseName(); break;
                        case ROWS_TOKEN:                ParseRows(); break;
                        case COLUMNS_TOKEN:             ParseColumns(); break;
                        case RHS_TOKEN:                 goto end;break;
                }
        }
```

Amikor arra a következtetésre jutunk, hogy az MPS_NAME_TOKEN következik, akkor a Scanner osztály Match() eljárásával ellenőrizzük, hogy valóban az jött-e és rögtön a következő token feldolgozásába kezdünk.

```
//-----------------------------------------------------------------
void MPSParser :: ParseName() {  //NAME + {Sztring}
//-----------------------------------------------------------------
        Match( MPS_NAME_TOKEN );                //Token ellenőrzés
        strcpy(mps->FileName,GetName());        //Nev beolvasasa
}
//-----------------------------------------------------------------
void MPSParser :: ParseRows() { // ROWS + { RowsData }
//-----------------------------------------------------------------
        Match( ROWS_TOKEN );
        ParseRowsData();
}
//-----------------------------------------------------------------
void MPSParser :: ParseColumns() { // COLUMNS + { ColumnsData }
//-----------------------------------------------------------------
        Match( COLUMNS_TOKEN );
        ParseColumnsData();
}
```

MPS formátum értelmezése (Parser)

A ParseRowsData() eljáráson belül is szükség van egy kulcsszó illesztésre, hiszen minden feltételnek egy speciális karakterben meg van adva a relációja. A következő tokenből a

feltétel nevét tudjuk kiolvasni, melynek ismeretével létrehozzuk a feltételek láncolt listáját. A láncolás a COLUMNS_TOKEN beolvasásával befejeződik.

```
//-------------------------------------------------------------
void MPSParser :: ParseRowsData() {       // r + Condition
//-------------------------------------------------------------
        char            r[2], name[10];
        while(GetCurrentToken()!= COLUMNS_TOKEN){
                //Parser r
                switch( GetCurrentToken() ) {
                    case EQUALITY_ROWS_TOKEN:         strcpy(r,GetName());break;
                    case NON_CONSTRAINING_ROWS_TOKEN:  strcpy(r,GetName());break;
                    case LESS_THAN_ROWS_TOKEN:        strcpy(r,GetName());break;
                    case GREATER_THAN_ROWS_TOKEN:     strcpy(r,GetName());break;
                    default:                      cout << "Rossz reláció"; return;
                }
                //Parser Condition name
                strcpy(name,GetName());
                mps->AddCondition(r, name);
        }
}
```

Az AddCondition (char *Reláció*, char *Feltételnév*) eljárás fűzi hozzá a listához az új elemeket.

```
//---------------------------------------------------------------------
void MPS :: AddCondition(char rel[1], char *nm){  //feltétel hozzáadása
//---------------------------------------------------------------------
        Condition*      NewCond;
        NewCond=(struct Condition*)malloc(sizeof(struct Condition));
        NewCond->SetN(nm);

        char c;
        c = rel[0];
        NewCond->setR(c);

        if(cond_first == NULL){
                        condNum++;
                        NewCond->idx=0;

                        cond_first=NewCond;
                        conditerator=NewCond;
                        conditerator->condNext = NULL;
                }
        else {
```

```
                        NewCond->idx = condNum;
                        condNum++;

                        NewCond->condNext=conditerator->condNext;
                        conditerator->condNext=NewCond;
                        conditerator=NewCond;
                }
        }
```

A reláció tulajdonság beállítása egy setR(char *Reláció*) eljárással történik. A relációknak csak négy fajtája megengedett (E, N, L, G) *(lásd 3.1 fejezet)*, melyeket egy felsorolás típussal adunk meg, eltérés esetén hibát jelez.

```
enum Relation { E, N, L, G};

//=============================================================
struct Condition {
//=============================================================
       Relation r;
        char condName[10];
        int idx;
        Condition* condNext;

        void setR(char rel){
                switch (rel){
                case 'E':       r = E;break;
                case 'N':       r = N;break;
                case 'L':       r = L;break;
                case 'G':       r = G;break;
                default:printf("Rossz relacio\n");
                }
        }
        void SetN(char *nm){
                strcpy(condName,nm);
        }
};
```

A ParseColumns() eljárás két láncolt listát is generál, a változókét és az együtthatókét:

```
//=============================================================
struct Variable {
//=============================================================
        char varName[10];
        int idx;
        Variable* varNext;
};

//=============================================================
struct Coefficient {
//=============================================================
        int condIdx;
        int varIdx;
        float data;
        int coefIdx;
        Coefficient* coefNext;
};
```

Ezen szekció minden egyes sora mindig egy változónévvel kezdődik, amely ha még nem létezik a változók listájában, akkor felfűzésre kerül az AddVariable(char *VáltozóNév*) metódussal:

```
//-------------------------------------------------------------------
void MPS :: AddVariable(char *name){      //változó hozzáadása
//-------------------------------------------------------------------
        Variable*       NewVar;
        NewVar=(struct Variable*)malloc(sizeof(struct Variable));
        strcpy(NewVar->varName, name);

        if(var_first == NULL){
                        varNum++;
                        NewVar->idx=0;

                        var_first=NewVar;
                        variterator=NewVar;
                        variterator->varNext = NULL;
        }
        else {
                        GetFirstVar();
                        while(variterator->varNext != NULL)
                                variterator=variterator->varNext;

                        NewVar->idx= varNum;
                        varNum++;
```

```
                        NewVar->varNext=variterator->varNext;
                        variterator->varNext=NewVar;
                        variterator=NewVar;
                }

}
```

A meglétét a FindVar(char *VáltozóNév*) függvénnyel ellenőrizhetjük, amely egy nem negatív számmal tér vissza, ha az adott név már szerepel a Variable listában, hiány esetén -1-gyel. Az így kapott szám a változó indexe, azaz a mátrix oszlop indexe. Ezt az indexet a vidx segédváltozóba tároljuk.

```
//---------------------------------------------------------------------
int MPS :: FindVar(char* name){          //A változó indexét adja vissza
//---------------------------------------------------------------------
        GetFirstVar();
        int vi=-1;

        while(variterator != NULL){
                if(strcmp(name, variterator->varName) == 0)
                {vi = variterator->idx;
                 return vi;
                }
                else
                        GetVarNext();
         }
        return vi;
}
```

A második token feltételnév lesz, amelynek az indexére szükségünk van a ritkamátrix felépítéséhez. A FindCond(char *FeltételNév*) függvény a megadott feltétel indexével tér vissza, ha az adott feltétel létezik, egyébként -1-gyel. Ezt az értéket ideiglenesen a cidx fogja felvenni.

```
//A feltétel indexét adja vissza
//---------------------------------------------------------------------
int MPS :: FindCond(char* nm){
//---------------------------------------------------------------------
        GetFirstCond();
        int ci=-1;

         do{
```

```
                if(strcmp(nm, conditerator->condName) == 0)
                        {ci =conditerator->idx;
                        return ci;
                        }
                else
                        GetCondNext();
         }while(conditerator != NULL);
        return ci;
}
```

A feltételt egy együttható követi, melynek láncolását az AddCoefficient(int *VáltozóIndex*, int *FeltételIndex*, float *EgyütthatóÉrték*) eljárás végzi. A struktúrában tároljuk az együttható értékét és a korábban meghatározott vidx és cidx indexeket, létrehozva ezzel a ritka mátrixot.

```
//együttható hozzáadása
//---------------------------------------------------------------------
void MPS :: AddCoefficient(int vidx, int cidx,  float data){
//---------------------------------------------------------------------
        Coefficient     *NewCoef;

        NewCoef = (struct Coefficient*)malloc(sizeof(struct Coefficient));

        NewCoef->varIdx = vidx;
        NewCoef->condIdx = cidx;
        NewCoef->data = data;

        if(coef_first == NULL){
                        NewCoef->coefIdx = coefNum;
                        coefNum++;

                        coef_first=NewCoef;
                        coefiterator=NewCoef;
                        coefiterator->coefNext = NULL;
                }
        else {
                        NewCoef->coefIdx=coefNum;
                        coefNum++;

                        NewCoef->coefNext=coefiterator->coefNext;
                        coefiterator->coefNext=NewCoef;
                        coefiterator=NewCoef;
                }
}
```

A folytatás két módon lehetséges: az együtthatót vagy változó vagy feltétel követi (azaz vagy új sor kezdődik vagy a régi folytatódik). Ezért az együttható után beolvasott tokenen végrehajtjuk a FindCond(char *FeltételNév*) műveletet. Ha ennek eredménye -1, akkor új sor kezdődik és az adott token egy változónév vagy az RHS_TOKEN. Egyébként az eredmény nem negatív, ekkor a sor folytatódik és az ötödik token számmá konvertálható értékű lesz, melyre ismét meghívjuk az AddCoefficient(int *VáltozóIndex*, int *FeltételIndex*, float *Együttható Érték*) függvényt, amelynek paramétere a mátrixban a korábban meghatározott vidx (oszlop index) és a most kapott új cidx (sor index) által definiált helyre kerül.

```
//-------------------------------------------------------------
void MPSParser :: ParseColumnsData() {    // vált + felt + data { + felt + data }
//-------------------------------------------------------------
        char    vname[10],* cname;
        float   data;
        int             cidx, vidx;

        strcpy(vname,GetName());

        while(GetCurrentToken() != RHS_TOKEN){
                //Parser Variable
                vidx = -1;
                //Létezik-e már egy ilyen nevű változó
                vidx = mps->FindVar(vname);
                //Ha nem, akkor adjunk hozzá egyet
                if(vidx == -1){
                        mps->AddVariable(vname);
                        vidx = mps->FindVar(vname);}
                //Parser Condition
                cidx = 0;
                // Az adott token változónév-e, vagy RHS_TOKEN
                while(cidx != -1 && GetCurrentToken() != RHS_TOKEN){
                        cname = GetName();
                        //A következő sztring feltétel-e?
                        cidx = mps->FindCond(cname);

                        //Ha  igen, bővítse az együtthatók listáját
                        if (cidx > -1){
                                data = GetReal();
                                mps->AddCoefficient(vidx, cidx, data);
                                }
                        else // Ha nem, akkor a sztring változónév
                                strcpy(vname,cname);
                }
```

```
        }
}
```

Az RHS_TOKEN beolvasása után a beolvasó megáll és a PrintMatrix() parancsra kiírja a létrehozott mátrixot.

```
end:
        cout << mps->FileName;
        cout << "\n A ritka matrix: \n";
        //Kiiratja a mátrixot
        mps->PrintMatrix();

//---------------------------------------------------------------------
void MPS :: PrintMatrix(){              //Ritka mátrix elemeinek kiiratása
//---------------------------------------------------------------------
        GetFirstCoef();

        do{
                cout << coefiterator->coefIdx << ". Coef " << coefiterator-
>condIdx << " " << coefiterator->varIdx <<" " << coefiterator->data << "\n";
                GetNextCoef();
        }while(coefiterator != NULL);
}
```

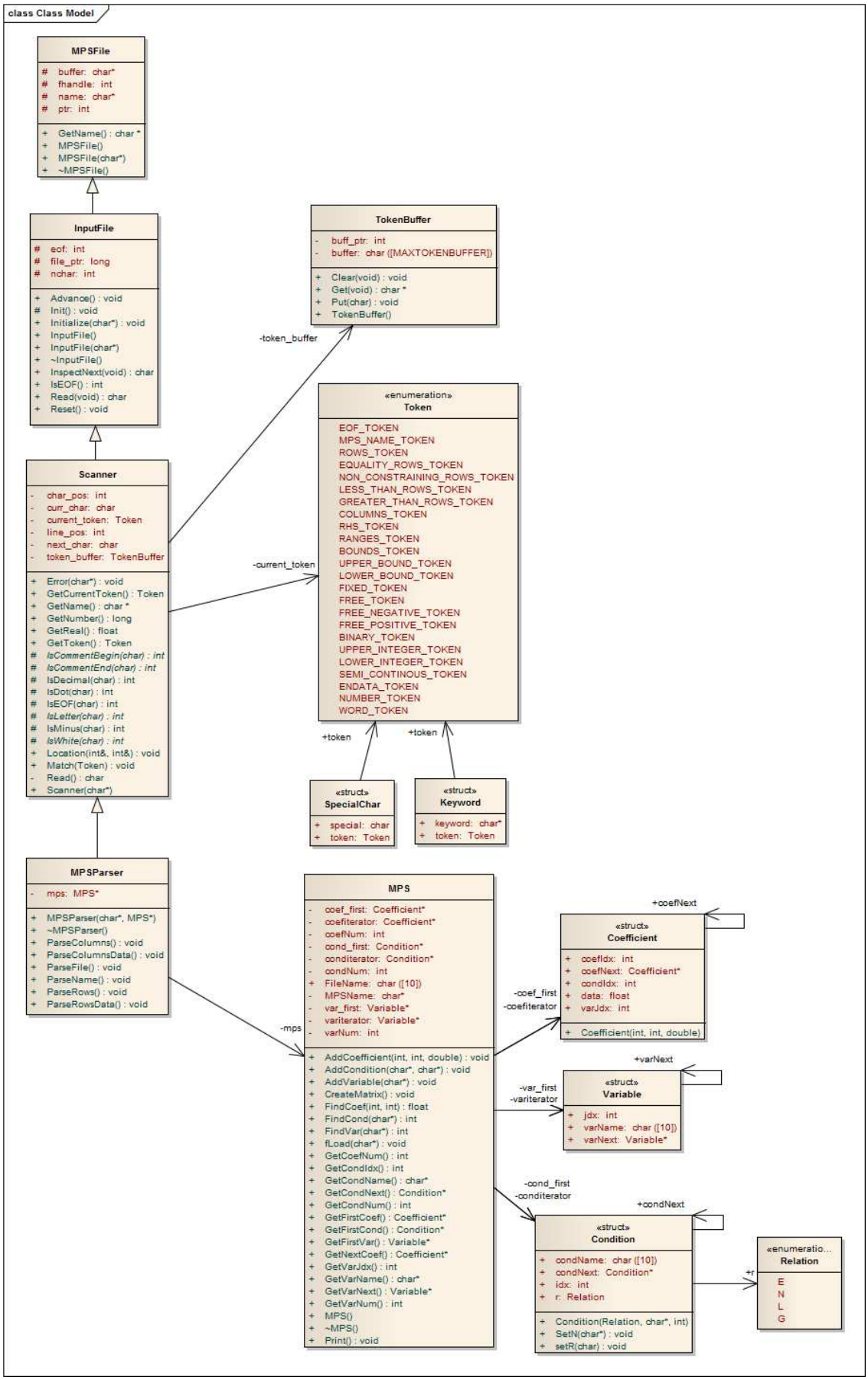
Készítettem továbbá egy kereső algoritmust, a FindCoef(int *FeltételIndex*, int *VáltozóIndex*)-et, amely a kiírt mátrix elemeinek közvetlen elérését teszi lehetővé. A felhasználó megadhatja a feltétel és a változó indexszámát, melyet a kereső paraméterként kap meg, visszatérési értéke pedig az indexekkel lokalizált együttható értéke lesz.

```
//---------------------------------------------------------------------
float MPS :: FindCoef(int c, int v){ // együttható megkeresése
//---------------------------------------------------------------------
        float coef=0;
        GetFirstCoef();

        do{
                if ( coefiterator->condIdx == c && coefiterator->varIdx == v)
                        coef = coefiterator->data;      //talalt
                GetNextCoef();                          //keri a kovetkezot
        }while(coefiterator != NULL && coef == 0);
        return coef;
}
```

Ezen a ponton lehetőségünk van az opció ismételt futtatására, az újabb kereső folyamat indítása az „i” karakter megadására történik, egyébként a program befejeződik.

```
// A szükséges együttható megkeresése & kiíratása

                int cond, var;
                float coef;
                char kov;

                do{
                cout << "Feltétel szama:\n";
                cin >> cond;
                cout << "Valtozo szama:\n";
                cin >> var;

                coef = mps->FindCoef(cond,var);
                cout << "Az egyutthato:" << coef <<"\n";

                cout << "Szeretnel meg egyutthatora keresni? (i/n) \n";
                cin >> kov;
                }while(kov == 'i' || kov == 'I');
}
```

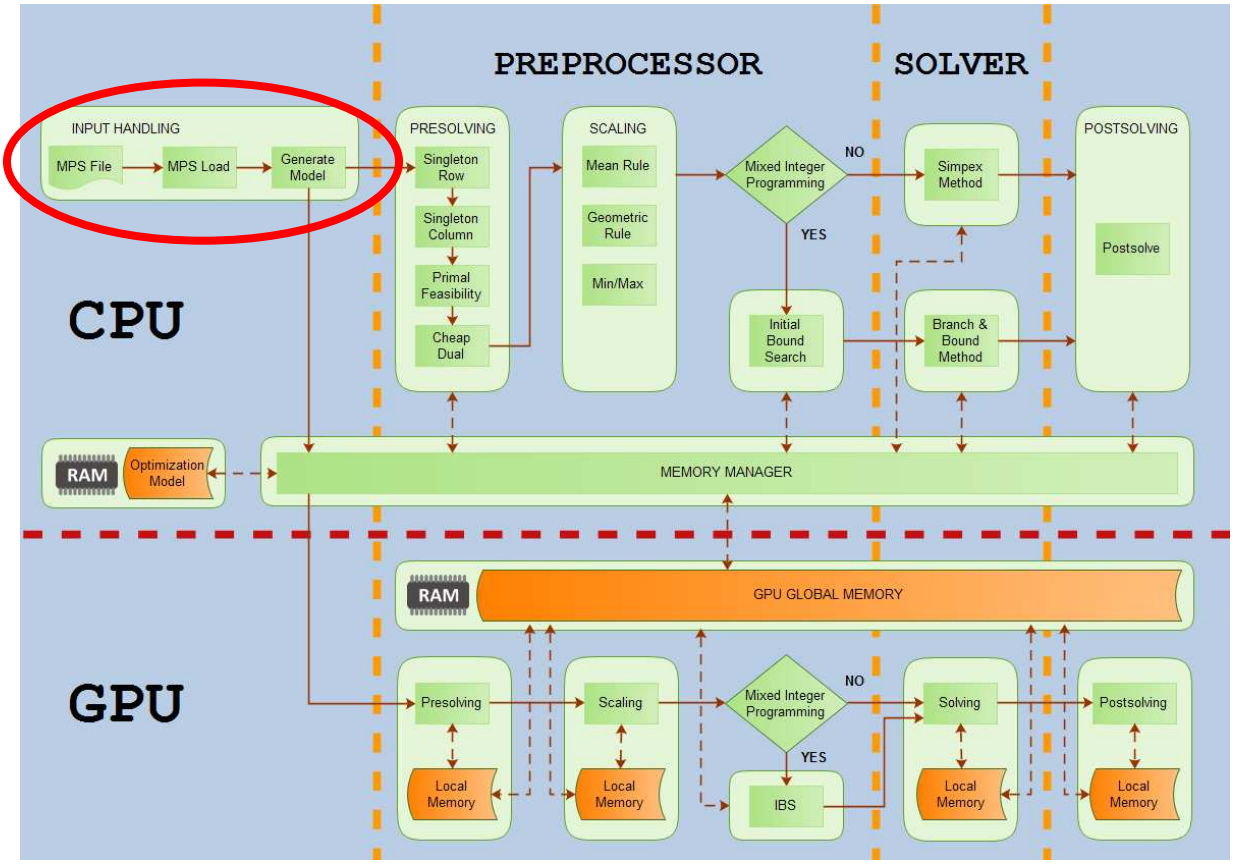
Az elkészített szoftver végleges osztály- és tartalmazási diagramja a következő ábrán látható:


class Class Model
MPSFile
buffer: char*
fhandle: int
name: char*
ptr: int
+ GetName() : char *
+ MPSFile()
+ MPSFile(char*)
+ ~MPSFile()
InputFile
eof: int
file_ptr: long
nchar: int
+ Advance() : void
# Init() : void
+ Initialize(char*) : void
+ InputFile()
+ InputFile(char*)
+ ~InputFile()
+ InspectNext(void) : char
+ IsEOF() : int
+ Read(void) : char
+ Reset() : void
TokenBuffer
- buff_ptr: int
- buffer: char ([MAXTOKENBUFFER])
+ Clear(void) : void
+ Get(void) : char *
+ Put(char) : void
+ TokenBuffer()
-token_buffer
«enumeration»
Token
EOF_TOKEN
MPS_NAME_TOKEN
ROWS_TOKEN
EQUALITY_ROWS_TOKEN
NON_CONSTRAINING_ROWS_TOKEN
LESS_THAN_ROWS_TOKEN
GREATER_THAN_ROWS_TOKEN
COLUMNS_TOKEN
RHS_TOKEN
RANGES_TOKEN
BOUNDS_TOKEN
UPPER_BOUND_TOKEN
LOWER_BOUND_TOKEN
FIXED_TOKEN
FREE_TOKEN
FREE_NEGATIVE_TOKEN
FREE_POSITIVE_TOKEN
BINARY_TOKEN
UPPER_INTEGER_TOKEN
LOWER_INTEGER_TOKEN
SEMI_CONTINOUS_TOKEN
ENDATA_TOKEN
NUMBER_TOKEN
WORD_TOKEN
Scanner
- char_pos: int
- curr_char: char
- current_token: Token
- line_pos: int
- next_char: char
- token_buffer: TokenBuffer
+ Error(char*) : void
+ GetCurrentToken() : Token
+ GetName() : char *
+ GetNumber() : long
+ GetReal() : float
+ GetToken() : Token
# IsCommentBegin(char) : int
# IsCommentEnd(char) : int
# IsDecimal(char) : int
# IsDot(char) : int
# IsEOF(char) : int
# IsLetter(char) : int
# IsMinus(char) : int
# IsWhite(char) : int
+ Location(int&, int&) : void
+ Match(Token) : void
- Read() : char
+ Scanner(char*)
-current_token
+token
+token
«struct»
SpecialChar
+ special: char
+ token: Token
«struct»
Keyword
+ keyword: char*
+ token: Token
MPSParser
- mps: MPS*
+ MPSParser(char*, MPS*)
+ ~MPSParser()
+ ParseColumns() : void
+ ParseColumnsData() : void
+ ParseFile() : void
+ ParseName() : void
+ ParseRows() : void
+ ParseRowsData() : void
-mps
MPS
- coef_first: Coefficient*
- coefiterator: Coefficient*
- coefNum: int
- cond_first: Condition*
- conditerator: Condition*
- condNum: int
+ FileName: char ([10])
- MPSName: char*
- var_first: Variable*
- variterator: Variable*
- varNum: int
+ AddCoefficient(int, int, double) : void
+ AddCondition(char*, char*) : void
+ AddVariable(char*) : void
+ CreateMatrix() : void
+ FindCoef(int, int) : float
+ FindCond(char*) : int
+ FindVar(char*) : int
+ fLoad(char*) : void
+ GetCoefNum() : int
+ GetCondIdx() : int
+ GetCondName() : char*
+ GetCondNext() : Condition*
+ GetCondNum() : int
+ GetFirstCoef() : Coefficient*
+ GetFirstCond() : Condition*
+ GetFirstVar() : Variable*
+ GetNextCoef() : Coefficient*
+ GetVarJdx() : int
+ GetVarName() : char*
+ GetVarNext() : Variable*
+ GetVarNum() : int
+ MPS()
+ ~MPS()
+ Print() : void
+coefNext
«struct»
Coefficient
+ coefIdx: int
+ coefNext: Coefficient*
+ condIdx: int
+ data: float
+ varJdx: int
+ Coefficient(int, int, double)
-coef_first
-coefiterator
+varNext
«struct»
Variable
+ jdx: int
+ varName: char ([10])
+ varNext: Variable*
-var_first
-variterator
-cond_first
-conditerator
+condNext
«struct»
Condition
+ condName: char ([10])
+ condNext: Condition*
+ idx: int
+ r: Relation
+ Condition(Relation, char*, int)
+ SetN(char*) : void
+ setR(char) : void
«enumeratio...
Relation
E
N
L
G
+r

5. ábra Az MPS beolvasó modul osztály- és tartalmazási diagramja

4.4 A program alkalmazási területe

A Debreceni Egyetem Alkalmazott Matematika és Valószínűségszámítás tanszékén folyamatban van egy új optimalizáló szoftver fejlesztése, mely az operációkutatási feladatok feldolgozási folyamatát támogatja. Az új solverben az eddigiektől eltérő megközelítést és technológiát alkalmaznak. A kutatás célul tűzte ki, hogy a könyvtárfejlesztések végeztével az új technológiát összehasonlítják a meglévőekkel.



6. ábra A solver folyamat ábrája

A feldolgozó több modulból áll majd, melynek kidolgozását több ember is végezheti. Ezen a projekten belül az én feladatom volt az Input handling modul elkészítése (a 6. ábrán piros körrel bekarikázva), melynek célja, hogy az alapértelmezett input formátumban, az MPS-ben megadott adatok belekerüljenek a memóriába. Szakmailag az algoritmusok összehasonlítása akkor elfogadott, ha valamely adatbázis, vagy problémagyűjtemény tesztfájljai képezik az összevetés alapját. Ezért vált szükségessé egy MPS beolvasó modul elkészítése.

5 Összefoglalás

A program elkészítése során lehetőségem volt tapasztalatot szerezni a programkészítés folyamatáról. Magát a programozást több lépés is megelőzte, mint az információgyűjtés, a tervezés, a formátumok, teszt-adatbázisok megismerése, beolvasási technológiák tanulmányozása, kritériumok és követelmények meghatározása. Utána kezdődött el a legnehezebb rész, a megvalósítás.

Az új nyelvtani szabályok megalkotása jelentették a legnagyobb kihívást. Az elméleti háttér megértése, áttekintése sokat jelentett a feladat kódjának elkészítésében, és a programozási nyelv ismerete is alapszükségletnek bizonyult. Az implementációt több tesztelés követte, melyben NETLIB-ről származó, nagyméretű, MPS formátumú fájlokra is ellenőriztük a program működését, ez lehetővé tette, hogy tökéletesítsük és finomítsuk a végleges szabályokat. Ezen fájlok beolvasása a nagy méret ellenére sem igényelt túl sok időt.

Bár az MPS egy viszonylag régi technológián, egy lyukkártyán tárolt formátumon alapul, tapasztalataim szerint mégis jelentős fontossággal bír a mai optimalizáló szoftverek világában. A kutatók újításaik hatékonyságának vizsgálatára a szakmailag elfogadott adatbázisok tesztfájljait használják, melyek elsősorban MPS formátumban érhetőek el. A kereskedelemi szoftverekkel való összehasonlítást lehetővé teszi, hogy a ma forgalomban lévő solverek többsége ezt az input formátumot is támogatja. Számunkra is a későbbi összevetések miatt volt fontos, hogy a készülőben lévő optimalizáló szoftver is tudjon MPS fájlokkal dolgozni.

6 Irodalomjegyzék

Internetes források:

[1.] http://plato.asu.edu/sub/testcases.html

[2.] ftp://ftp.numerical.rl.ac.uk/pub/oldwww/cute/netlib.html

[3.] http://www.math.ufl.edu/~hager/coap/testcases.html

[4.] http://www.cuter.rl.ac.uk/

[5.] http://www.cuter.rl.ac.uk/Problems/classification.shtml

[6.] http://www.ampl.com/

[7.] http://www.ampl.com/EXAMPLES/MCDONALDS/diet1.mod

[8.] http://www.g8.ams.com/

[9.] http://www.gams.com/docs/example.htm

[10.] http://en.wikipedia.org/wiki/MPS_(format)

[11.] http://www.coin-or.org/

[12.] http://ganymedes.lib.unideb.hu:8080/dea/bitstream/2437/3719/1/Dolgozat.pdf

[13.] http://www.learnaboutor.co.uk

[14.] http://hu.wikipedia.org/wiki/Oper%C3%A1ci%C3%B3kutat%C3%A1s

[15.] http://www.ibm.com/developerworks/university/teachingtopics/or_ms.html

[16.] http://www.gurobi.com/

[17.] http://www.software4research.com/index.php?language=en&products_id=19&store_link=product_info

Könyvek, cikkek, tanulmányok

[18.] Maros I., *„Computational Techniques of the Simplex Method”*, Kluwer Academic Publishers, 2003.

[19.] Bach I., *„Formális nyelvek: Egyetemi tankönyv*” , Typotex, 2005.

[20.] Szirmay-Kalos L., Antal Gy. és Csonka F., *„Háromdimenziós grafika, animáció és játékfejlesztés”*, ComputerBooks, 2004.

[21.] Dantzig, G. B., *„Linear programming”*, Operations Research, 2002. 50: 42-47, INFORMS

[22.] Pardalos, P.M., Resende M.G.C., *„Handbook of Applied Optimization”*, Oxford Universoty Press, 2002.

[23.] Dömösi P., Fazekas A., Horváth G., Mecsei Z., *„Formális nyelvek és automaták”*, Debreceni Egyetem, egyetemi jegyzet, 2003.

[24.] Winston W. L., *„Operációkutatás: Módszerek és alkalmazások”*, Aula kiadó, 2003.

[25.] Benkő T., Kuzmina J., Moré G., Tóth B., *„A C++ programozási nyelv alkalmazásokkal”*, Budapesti Műszaki Egyetem Mérnöktovábbképző Intézet, 1995.

[26.] Kernighan B. W., Ritchie M. D., *„A C programozási nyelv”*, Műszaki kiadó, 2006.

[27.] Raffai Mária, *„Az operációkutatás, mint döntés előkészítés”*, NOVADAT, 1994.

[28.] Ferenczi Zoltán, *„Alkalmazott operációkutatás”*, NOVADAT, 1997.

[29.] Csordás A., Mohai G., *„Adatszerkezetek és algoritmusok”*, Debreceni Egyetem, egyetemi jegyzet, 2000.

[30.] Dewhurst S. C., *„C++ hibaelhárító”*, Kiskapu, 2003.

[31.] Alexandrescu A., Sutter H. *„C++ kódolási szabályok”*, Kiskapu, 2005.

[32.] Stroustrup B., *„A C++ programozási nyelv”*, Kiskapu, 2001.

[33.] Kondorosi K., László Z., Szirmay-Kalos L., *„Objektum-orientált szoftverfejlesztés”*, ComputerBooks, 2007.

7 Ábrák jegyzéke

8 Függelék

Forráskód 1.:

AMPL forráskód példa a [7.] forrásból származik:

```
set NUTR ordered;
set FOOD ordered;
param cost {FOOD} >= 0;
param f_min {FOOD} >= 0, default 0;
param f_max {j in FOOD} >= f_min[j], default Infinity;
param n_min {NUTR} >= 0, default 0;
param n_max {i in NUTR} >= n_min[i], default Infinity;
param amt {NUTR,FOOD} >= 0;
# ---------------------------------------------------------
var Buy {j in FOOD} integer >= f_min[j], <= f_max[j];
# ---------------------------------------------------------
minimize Total_Cost:  sum {j in FOOD} cost[j] * Buy[j];
minimize Nutr_Amt {i in NUTR}: sum {j in FOOD} amt[i,j] * Buy[j];
# ---------------------------------------------------------
```

Forráskód 2.:

GAMS Forráskód példa a [9.] forrásból származik:

```
subject to Diet {i in NUTR}:
   n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
SETS
       I   canning plants   / SEATTLE, SAN-DIEGO /
       J   markets          / NEW-YORK, CHICAGO, TOPEKA / ;
  PARAMETERS
       A(I)  capacity of plant i in cases
         /    SEATTLE     350
              SAN-DIEGO   600  /
       B(J)  demand at market j in cases
         /    NEW-YORK    325
              CHICAGO     300
              TOPEKA      275  / ;
  TABLE D(I,J)  distance in thousands of miles
                    NEW-YORK       CHICAGO      TOPEKA
      SEATTLE          2.5           1.7          1.8
      SAN-DIEGO        2.5           1.8          1.4  ;
  SCALAR F  freight in dollars per case per thousand miles  /90/ ;
  PARAMETER C(I,J)  transport cost in thousands of dollars per case ;
            C(I,J) = F * D(I,J) / 1000 ;
  VARIABLES
```

```
     X(I,J)  shipment quantities in cases
     Z       total transportation costs in thousands of dollars ;
POSITIVE VARIABLE X ;
EQUATIONS
     COST        define objective function
     SUPPLY(I)   observe supply limit at plant i
     DEMAND(J)   satisfy demand at market j ;
COST ..        Z  =E=  SUM((I,J), C(I,J)*X(I,J)) ;
SUPPLY(I) ..   SUM(J, X(I,J))  =L=  A(I) ;
DEMAND(J) ..   SUM(I, X(I,J))  =G=  B(J) ;
MODEL TRANSPORT /ALL/ ;
SOLVE TRANSPORT USING LP MINIMIZING Z ;
```

9 Köszönetnyilvánítás

Ez a dolgozat nem jöhetett volna létre témavezető tanárom, Bekéné Rácz Anett, támogató munkássága és türelme nélkül. Köszönöm a rám fordított órákat és a türelmes magyarázatokat.

Továbbá köszönettel tartozok Megyesi Zoltán tanár úrnak, akinek utat mutató utasításai nélkül nem tudtam volna elkészíteni a programot.

Végül hálával tartozom családomnak, akik lehetővé tették, hogy elegendő időt szánhassak a dolgozat elkészítéséhez.