

TDK-dolgozat

Varga Virág



**Pannon Egyetem
Műszaki Informatikai Kar**



**A simplex induló bázisának meghatározása korszerű
módszerekkel**

Készítette: Varga Virág, Mérnök Informatikus MSc, II. évf.

Témavezető: Dr. Maros István, Egyetemi tanár

Pannon Egyetem, Műszaki Informatikai Kar,
Rendszer és Számítástudományi Tanszék

2012

Tartalomjegyzék

1. Bevezetés	1
2. Lineáris programozás	3
2.1. Alkalmazási területek	3
2.2. Módszerek LP feladatok megoldására	6
2.3. Formális leírás	7
3. A szimplex módszertan	11
3.1. Definíciók és jelölések	12
3.2. A szimplex algoritmus	14
3.2.1. Megvalósíthatóság és optimalitás	14
3.2.2. Az algoritmus leírása	16
4. Kezdőbázis keresés	19
4.1. A kezdőbázis meghatározás szerepe	19
4.2. Logikai bázis	20
4.3. Crash bázisok	21
4.3.1. Hagyományos crash eljárás (szimbolikus crash)	21
4.3.2. LTSF crash eljárás	23
4.3.3. ADG crash eljárás	25
4.4. Egyéb kezdőbázis kereső módszerek	26
4.4.1. Parciális kezdőbázis	26
4.4.2. A CPLEX bázis	27
4.4.3. Szakító algoritmus	27
5. A PanOpt keretrendszer	29
5.1. Szimplex módszertan nagy feladatoknál	29
5.2. Technikai háttér	30
5.3. A kezdőbázis keresés implementálása	31
5.3.1. Egy kezdőbázis kereső algoritmus általános felépítése	33

5.3.2.	Segéd eljárások	34
5.3.3.	A kezdőbázis minőségének megítélése	37
5.3.4.	A kezdőbázis kereső modul felépítése	40
5.3.5.	A logikai bázis speciális tulajdonságai	41
5.3.6.	A crash bázisok speciális tulajdonságai	42
5.4.	Tesztelés	43
6.	Összefoglalás	49
	Irodalomjegyzék	50

Táblázatok jegyzéke

2.1. Az LP feladatok standard jelölései	8
2.2. Változók típusai	8
3.1. Az x_k változó lehetséges elmozdulásának mértéke	15
3.2. Az x_k változó optimalitási feltétele	16
4.1. Sor prioritások (<i>Row Priority, RP</i>)	23
4.2. Oszlop prioritások (<i>Column Priority, CP</i>)	24
5.1. A logikai bázis teszteredményei 1.	44
5.2. A logikai bázis teszteredményei 2.	45
5.3. A crash bázis teszteredményei 1.	46
5.4. A crash bázis teszteredményei 2.	47

Ábrák jegyzéke

4.1. A hagyományos crash algoritmus	22
4.2. Az LTSF crash algoritmus	25
4.3. Az ADG crash algoritmus	28
5.1. Az architektúra vázlata	32
5.2. Legfontosabb adatszerkezetek	32
5.3. A kezdőbázis kereső modul felépítése	41

1. fejezet

Bevezetés

Napjaink egyik legfontosabb szakterülete az operációkutatás, másképpen a döntéshozatal tudománya: egy gyakorlatban megfogalmazódott probléma megoldása, melynek során a *mit* és *hogyan* kérdésekre a bizonyítottan optimális választ keressük.

Az operációkutatás kifejezés megszületése a II. világháború idejére tehető, ahol az *operáció* szó hadműveletet jelentett. Számos harc és háború kimenetele függött attól, hogy az egyes csapatok hogyan voltak képesek megszervezni az egységek élelmiszer- és fegyverellátását, a harci útvonalat vagy akár a felderítő radarok elhelyezését. Az ilyen jellegű döntések súlyossága megkövetelte, hogy a korábbi döntéshozatali folyamatokkal ellentétben ne egy ember által intuitív módon kijelölt megoldást iteráljanak, hanem a globális optimumot legyenek képesek meghatározni.

Az operációkutatás során a megfelelő szaktudással rendelkező személyek a gyakorlati problémának egyszerűsített, de minden lényeges tényezőt magában foglaló matematikai modelljét írják fel. Ezután a szakterület gondosan kidolgozott matematikai módszertanjaival oldják meg a feladatot, megkapva egy/az optimális megoldást. Amennyiben a probléma nem megoldható, úgy ennek tényéről kapnak tájékoztatást. Végül az eredményeket a feladat eredeti megfogalmazásának figyelembevételével prezentálják a döntéshozó testületnek.

A XXI. századra a fenti folyamat jelentős mértékben automatizálódott: különböző programok segítségével a matematikai modell akár generálható is, amelyre aztán különféle megoldó szoftverek hívhatóak meg. Ezek a programok számos követelménynek kell, hogy megfeleljenek: megbízhatóság, sebesség, rendkívül nagy mennyiségű adatok tárolása, numerikus pontosság és így tovább.

Számos szoftver található, amely a fenti megoldó folyamatot képes elvégezni. Ezek nagy része fizetős program formájában elérhető, amelyek ugyan képesek több

1. Bevezetés

specifikus probléma megoldására is, azonban komoly összegekbe kerülnek. Továbbá mind az üzleti, mint a szabadon elérhető alkalmazások hátránya a már meglévő, hosszú évek során felhalmozódott óriási kódbázis, amely jelentős mértékben hátráltatja a fejlesztőket, hogy az időközben publikált eredményeket és algoritmusokat integrálni legyenek képesek.

Dolgozatom célja, hogy néhány kollégámmal egyetemben létrehozzuk a Pannon Egyetem saját szoftverét, a *PanOpt* optimalizáló rendszert. A program elsősorban az operációkutatás egy speciális területével foglalkozik, az ún. *lineáris programozással*. A fenti általános jellegű követelmények megtartása mellett fontos hangsúlyt fektetünk az új ötletek implementálására és a modern megvalósítási technikák felhasználására. A tervezési fázisok és implementációs kérdések közös kidolgozása után saját feladatom a **kezdőbázis kereső** részrendszer kidolgozása és megvalósítása.

Ahogy az idevonatkozó szakirodalom is bizonyítja, a hatékony kezdőbázis-keresés fontos lépés a megoldási folyamat során. Mindezek ellenére nem létezik egységes álláspont ennek kivitelezésére, hiszen az egyes algoritmusokat heurisztikus alapon, több év vagy évtized fejlesztési tapasztalatai hívták életre. Dolgozatom során a korszerű LP rendszerekben alkalmazott kezdőbázis-kereső algoritmusokat sorakoztatom fel és hasonlítom össze. Az elméleti áttekintés mellett korszerű implementációt alkalmazva számos statisztikai eredmény segíti a kiértékelést. Mindezek mellett egy általánosan jól teljesítő verzió meghatározása is fontos szempont a PanOpt rendszer számára, szem előtt tartva, hogy a saját implementáció felvegye a versenyt az elterjedt szoftverek hasonló funkciójának teljesítményével.

2. fejezet

Lineáris programozás

Az operációkutatás egyik alappillére a lineáris programozás (LP), újabb nevén a lineáris optimalizálás. A felírt matematikai modellben a kérdéses változók egy *változóvektorban* foglalhatóak össze. Mivel az LP feladatok optimalizálni kívánt kifejezésében ezek mind egyes fokszámon szerepelnek, így a *célfüggvény* a változók különböző együtthatókkal vett *lineáris kombinációjával* adott. Ezek mellett számos korlátozási feltétel adható meg, mind külön az egyes változókra, mind azok bármely módon vett lineáris kombinációjára.

Ugyan nehézségi fokát tekintve az LP feladatok mondhatóak az operációkutatás „legkönnyebb” lépcsőjének, jelentősége azonban pont abban rejlik, hogy számos más módszertan (egészértékű programozás, nemlineáris programozás, játékelmélet, stb.) is felhasználja. Így rendkívül fontos, hogy minél hatékonyabb implementációt biztosítsunk. Ahogy a későbbiekben is látni fogjuk, egy LP megoldó sikerének kulcsa lehet a hatékony részalgoritmusok mellett a megfelelő adattárolás és a rendszerek rugalmas és gyors működése.

Jelen fejezetben a lineáris programozási feladatok lehetséges alkalmazási területeinek megismerésével és a formális leírásával foglalkozunk.

2.1. Alkalmazási területek

Lineáris programozással az élet számos területén találkozhatunk. Szállítmányozási, logisztikai feladatokra könnyen felírhatóak LP modellek, mint ahogy az egészségügyben, a mezőgazdaságban, az olajiparban, vegyiparban és egyéb ipari ágazatokban, telekommunikációban, nemzetvédelemben vagy a pénzügyekben is rengeteg példát találhatunk.

2. Lineáris programozás

Az optimalizálni kívánt célfüggvény és annak megoldása, a vezetőség és a menedzsment számára általában a költségek minimalizálását vagy a profit maximalizálását jelentik.

Az üzleti tervet tehát a gazdasági környezet által meghatározott peremfeltételek figyelembevételével kell meghozni. A legtöbb esetben maga az LP modell képes leírást - valamint a hozzá tartozó szoftverek: megoldást - biztosítani. Bonyolultabb folyamatok esetében pedig az LP feladatokra épülő modellek képesek segítséget nyújtani.

Példa LP feladatra

Gyakori az olyan üzleti cél felírása, melynek során különböző termékek vagy folyamatok mennyiségi eloszlását kell meghatározni, hogy közben a profit maximális lehessen. Ezek az úgy nevezett *termékösszetételi* (product mix) feladatok.

Tekintsük a következőt. Egy kisvállalkozásként indított pékség különféle péksüteményeket képes gyártani: kenyeret (K), zsemlet (ZS), óriásperecet (P) és lepényt (L). Ezeket sorrendben a következő egységáron tudja eladni: 120 Ft, 24 Ft, 80 Ft, 80 Ft. Naponta az alábbi alapanyagokat tudja kedvezményesen beszerezni: liszt, víz, só, cukor, élesztő, alma. Prezentáljuk azt az üzleti tervet, amely meghatározza, hogy a rendelkezésre álló készletek alapján mely termékből hányat kell gyártani a profit maximalizálásához.

A fenti adatok a részletes alapanyagmennyiségekkel kiegészítve egy táblázatban foglalhatóak össze.

		Kenyér	Zsemle	Óriásperec	Lepény	Korlát
Bevétel	[Ft]	120	24	80	70	
Liszt	[dkg]	30	1	2	0.5	100
Víz	[dl]	1	0.5	0.05	0.05	10
Só	[g]	2	0.1	10	0.05	1000
Cukor	[g]	2	0.5	0.01	0.5	5
Élesztő	[g]	30	0.1	0	0.1	2
Alma	[dkg]	0	0	0	2	100

A példa megoldására az egyik legelterjedtebb, nyílt forráskódú megoldót, a *GLPSOL: GLPK LP/MIP Solver, v4.43* szoftvert használtam [1], mivel könnyen olvasható modellező nyelvvel (GMPL/GNU MathProg) [2] rendelkezik.

2.1. Alkalmazási területek

A feladat egyszerű modellje a következőképpen néz ki:

```
# Problem
# This finds the optimal solution for maximizing profit

/* Decision variables */
var x1 >=0; /* K */
var x2 >=0; /* ZS */
var x3 >=0; /* P */
var x4 >=0; /* L */

/* Objective function */
maximize z: 120*x1 + 24*x2 + 80*x3 + 70*x4;

/* Constraints */
s.t. Flour : 30*x1 + x2 + 2*x3 + 0.5*x4 <= 100;
s.t. Water : x1 + 0.5*x2 + 0.05*x3 + 0.05*x4 <= 10;
s.t. Salt : 2*x1 + 0.1*x2 + 10*x3 + 0.05*x4 <= 1000;
s.t. Sugar : 2*x1 + 0.5*x2 + 0.01*x3 + 0.5*x4 <= 5;
s.t. Yeast : 30*x1 + 0.1*x2 + 0.1*x4 <= 2;
s.t. Apple : 2*x4 <= 100;

end;
```

A megoldás a glpsol paranccsal, a bemeneti és kimeneti fájlok megadásával történik.

```
$ glpsol -m Pekseg.mod -o Pekseg.sol
```

Végül az eredmény a Pekseg.sol fájlból kiolvasható.

```
Status:      OPTIMAL
Objective:   z = 4452.261307 (MAXimum)
```

No.	Row name	St	Activity	Lower bound	Upper bound
1	z	B	4452.26		
2	Flour	NU	100		100
3	Water	B	2.8392		10
4	Salt	B	477.839		1000
5	Sugar	NU	5		5
6	Yeast	B	0.904523		2
7	Apple	B	18.0905		100

2. Lineáris programozás

No.	Column name	St	Activity	Lower bound	Upper bound
1	x1	NL	0	0	
2	x2	NL	0	0	
3	x3	B	47.7387	0	
4	x4	B	9.04523	0	

A döntéshozók számára prezentált adatok így a következők lesznek (a modellezés során a beszerzési költségtől eltekintettünk, kiindulva abból, hogy a készlet már rendelkezésre áll):

- Az elérhető maximális profit 4452 Ft.
- Nagyjából 47.7 darab óriásperec és 9 darab lepény sütése szükséges.
- A felhasznált mennyiségek: 100 dkg liszt, 2.8 dl víz, 477.8 g só, 5 g cukor, 0.9 g élesztő és 18 dkg alma.

A fentiekben egy egyszerűsített példát vizsgáltunk. A valóságban nagyságrendekkel több termék és feltétel fordulhat elő, valamint figyelembe kell venni, hogy adott változóknak esetenként csak egész érték mellett lehet értelmük. ¹

2.2. Módszerek LP feladatok megoldására

A lineáris egyenlőtlenség rendszerek az 1940-es évek utolsó felében kerültek az fokozott érdeklődés középpontjába. Az USA-beli Dantzig és a Szovjetunió-beli Kantorovich munkája tette a lineáris programozást vezető kutatási témává, számos lehetséges alkalmazási területet felsorakoztatva [3].

Dantzig nevéhez köthető a hagyományos, majd később az újratervezett **szimplex módszer** (*revised simplex method*) megtervezése, amely alkalmas lineáris megkötésekkel megfogalmazott, lineáris kifejezés maximumának vagy minimumának meghatározására. Mivel az optimalizálási probléma számos területen alkalmazható, így megszületett a **lineáris programozás** kifejezés.

Az 1960-1970-es években az ipari feladatok megjelenésével újabb kérdések merültek fel: nagy méretű, ritkás mátrixok kezelésének problémája [4]. A témával sokan foglalkoztak, s mára számos ajánlás és módszer született meg az alkalmazandó

¹Ez esetben már ún. vegyes egész lineáris programozásról beszélhetnénk, amelynek bonyolultsága meghaladja a hagyományos LP feladatokét.

2.3. Formális leírás

adatstruktúrákkal, a szoftveres implementáció sajátosságaival és a gépi korlátok figyelembevételével kapcsolatban. Ezek a mai napig virágzó kutatási területet jelentenek.

Az 1980-as évekre a szimplex módszertan vizsgálata és javításai mellett újabb kutatások indultak más megközelítéseket alkalmazva. Ez eredményezte az úgynevezett **belső pontos módszerek** (*interior point method* megjelenését, *IPM*), mely főként Karmarkar munkásságához köthető [5].

Ugyan az IPM számos jelentős sikereket ért el, ezek tovább sarkallták a szimplex módszertan kutatóit újabb bővítmények és gyorsítások beépítésére. A szimplex módszertan jellegzetességeiből adódóan minden valószínűség szerint a jövőben is meg fogja tartani vezető szerepét, mivel az operációkutatás más feladatosztályaiban is hatékonyan felhasználható.

Az LP feladatok kutatásának evolúciójáról többek közt Maros és Mitra cikke ad áttekintést [6].

2.3. Formális leírás

Az LP feladatok formális leírása a következő. A megfogalmazott modell legfontosabb komponense maga a **célfüggvény**, melyben az optimalizálni kívánt kifejezés kerül leírásra. Szokásos jelölése z -vel történik. Tartalmazza az optimalizálás irányát (minimalizálás vagy maximalizálás), valamint az $\mathbf{x}$ ($\in \mathbb{R}^n$) változóvektor $\mathbf{c}$ ($\in \mathbb{R}^n$) célfüggvényegyütthatókkal történő lineáris kombinációját. Másképpen megfogalmazva: azt fejezi ki, hogy az optimumot az egyes változók értéke mennyiben befolyásolja.

$$\text{min/max} \quad z = c_0 + \mathbf{c}^T \mathbf{x} = c_0 + c_1 x_1 + \dots + c_n x_n = c_0 + \sum_{j=1}^n c_j x_j \quad (2.1)$$

Amennyiben m darab korlátozó feltétellel rendelkezik a feladat, úgy azokat összefoglalva az $\mathbf{A} \in \mathbb{R}^{m \times n}$ **együtthatómátrix**szal szokás leírni, amelyekhez mind alsó ($\mathbf{L} \in \mathbb{R}^m$), mind felső ($\mathbf{U} \in \mathbb{R}^m$) korlát hozzárendelhető.

$$\mathbf{L} \leq \mathbf{Ax} = \mathbf{a}_1 x_1 + \dots + \mathbf{a}_n x_n = \sum_{j=1}^n \mathbf{a}_j x_j \leq \mathbf{U} \quad (2.2)$$

Az együtthatómátrix szemantikai jelentősége abban rejlik, hogy a modellezés során azonosított változók tetszőleges lineáris kombinációjára figyelembe veendő feltételeket egységesen képes kezelni.

2. Lineáris programozás

Végül magukra a változókra individuális korlát adható ($\mathbf{l} \in \mathbb{R}^n$ és $\mathbf{u} \in \mathbb{R}^n$).
(Amennyiben az explicit megadás hiányzik, úgy végtelennel számolhatunk.)

$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \quad (2.3)$$

Könnyen belátható, hogy megfelelő átalakításokkal minden LP feladat a következő ekvivalens, *standard formára* hozható.

$$\min \quad z = \mathbf{c}^T \mathbf{x} \quad (2.4)$$

$$\text{s.t.} \quad \mathbf{Ax} = \mathbf{b} \quad (2.5)$$

$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \quad (2.6)$$

A standard számítási formák egyes jelöléseit a 2.1. táblázat foglalja össze.

Jelölés	Leírás
$z \in \mathbb{R}$	Célfüggvényérték
$\mathbf{x} \in \mathbb{R}^n$	Változókat tartalmazó vektor
$\mathbf{c} \in \mathbb{R}^n$	Célfüggvényegyütthatók vektora
$\mathbf{A} \in \mathbb{R}^{m \times n}$	Együtthatómátrix (korlátok alapján)
$\mathbf{b} \in \mathbb{R}^m$	RHS (<i>Right Hand Side</i>) értékek vektora
$\mathbf{l} \in \mathbb{R}^n$	Változók alsó korlátja
$\mathbf{u} \in \mathbb{R}^n$	Változók felső korlátja

2.1. táblázat. Az LP feladatok standard jelölései

A változókat a 2.2. táblázat szerint szokás megkülönböztetni.

Alsókorlát		Felsőkorlát	Hivatkozás	Típus
l_j	$\leq x_j \leq$	u_j	Fix	0
l_j	$\leq x_j \leq$	u_j	Korlátos	1
l_j	$\leq x_j \leq$	∞	Plusz	2
$-\infty$	$\leq x_j \leq$	∞	Szabad	3
$-\infty$	$\leq x_j \leq$	u_j	Mínusz	4

2.2. táblázat. Változók típusai

Áttérés standard formára

Miközben az eredeti egyenlőtlenségekről áttértünk a 2.5. egyenlőségre, a következőt alkalmaztuk. Vegyünk fel m darab új változót a feladathoz, jelöljük ezeket az $\mathbf{y} \in \mathbb{R}^m$ vektorral. Ha a korlátozó feltételek minden k . sorához hozzáadjuk 1-es együtthatóval az adott indexű y változót, úgy y_k típusának meghatározása után egyenlőségekhez jutunk. Ezután felüldefiniáljuk $\mathbf{x}$ -t és $\mathbf{A}$ -t.

$$\mathbf{x} = \begin{bmatrix} \mathbf{x} \\ \mathbf{y} \end{bmatrix}$$
$$\mathbf{A} = \begin{bmatrix} \mathbf{A} \mathbf{I} \end{bmatrix}$$

Természetesen $\mathbf{l}$ és $\mathbf{u}$ is kibővül az egyenlőtlenségek eredeti alakjától függően. A célfüggvényegyütthatók vektora 0 elemekkel töltődik fel. Ezzel eljutottunk a 2.4-2.6. alakhoz.

A feladat eredeti $\mathbf{x}$ változóit **strukturális**, az imént hozzáadott $\mathbf{y}$ változókat **logikai változóknak** nevezzük.

3. fejezet

A szimplex módszertan

Az LP feladatok megoldására több módszertan létezik, ahogy azt az előző fejezet 2.2. részében is láthattuk. Ugyan a tervezett *PanOpt keretrendszer* célja egy általános operációkutatási megoldó szoftver, a szimplex módszertant tekintjük kiindulási alapnak. Így a program jelenlegi fázisában - és ezzel a dolgozat hátralévő részében - a kutatásainkat a szimplex módszertan köré fókuszáljuk.

Mivel természetesen a modern kor követelményeinek való megfelelés is cél, így munkánk során ritkás szimplex algoritmusokkal (*sparse simplex*, *SSX*) dolgozunk. Ezzel nagy méretű, általában modellező szoftverek kimeneteként generált, fokozottan ritkás jellemzőkkel bíró, gyakran numerikusan instabil feladatok megoldása is elérhetővé válik.

A szimplex megoldási folyamatának lényege, hogy az LP problémát egy komplex poliéderrel (politóp) reprezentálja, amelyben a lehetséges megoldások csúcsokként szerepelnek. Egy kezdőcsúcsot kiválasztva és a politóp élein sétálva a jelenlegi célfüggvényérték rontása nélkül eljutunk az optimális megoldásig, amennyiben az létezik.

A gyakorlatban több szituáció felléphet, amikor egy ideig lényeges javítások nélkül halad az algoritmus, illetve esetenként beragadhat a következő lépés irányának meghatározása. Ezek kiküszöbölésére a hagyományos szimplex algoritmust újabb pivotáló szabályokkal szokás kiegészíteni.

Mindezeket figyelembe véve a szimplex algoritmus megfelelően hatékony, és - amennyiben a megfelelő ciklikusságmegelőző lépések megtörténnek -, képes garantálni a globális optimum megtalálását.

A témakör részletes tárgyalását megtalálhatjuk többek közt Maros [7] és [8] munkáiban.

3.1. Definíciók és jelölések

Az LP feladat **megoldásának** nevezzük azt az $x \in \mathbb{R}^n$ vektort, amely a 2.5 általános korlátokat kielégíti. Egy $x \in \mathbb{R}^n$ vektor **megengedett** (megvalósítható, *feasible*) **megoldás**, ha megoldás és a 2.6 individuális korlátoknak is megfelel. A lehetséges megoldások halmazát $\mathcal{X}$ -szel jelöljük:

$$\mathcal{X} = \{\mathbf{x} : \mathbf{Ax} = \mathbf{b}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\}$$

A megengedett megoldások tehát egy konvex halmazt határoznak meg. Amennyiben $\mathcal{X}$ üres, úgy a probléma nem megoldható (*infeasible*). Egy megengedett $\mathbf{x}^*$ megoldás **optimális**, amennyiben

$$\mathbf{c}^T \mathbf{x}^* \leq \mathbf{c}^T \mathbf{x}, \forall \mathbf{x} \in \mathcal{X}.$$

Ha az optimum nem egyedi, úgy a problémának **alternatív optima** is létezik. Az optimális megoldások konvex lineáris kombinációi is optimális megoldások egyben.

Az $\mathbf{A}$ mátrix rangjára vonatkozóan, könnyen belátható annak értéke: mivel a standard formára való áttéréskor (lásd: 2.3. rész) hozzáadtunk a mátrixhoz m darab független változót ($y_k : k = 1, 2, \dots, m$), ezáltal $\mathbf{A}$ biztosan teljes rangú és annak értéke m . Ez azt jelenti, hogy létezik $\mathbf{A}$ -ban olyan m darab oszlopvektor ($\mathbf{a}_{k_1}, \dots, \mathbf{a}_{k_m}$), amelyek egy nemszinguláris $\mathbf{B} \in \mathbb{R}^{m \times m}$ mátrixot alkotnak. Az ilyen jellegű $\mathbf{B}$ mátrixot **bázis**nak hívjuk. **Bázisfejnek** nevezzük a $\mathbf{B}$ -t alkotó $\mathbf{A}$ -beli oszlopvektorok indexeinek rendezett halmazát: $\mathcal{B} = \{k_1, \dots, k_m\}$. A megmaradó (nembázis) változók indexhalmazának jelölése: $\mathcal{R} = \mathcal{N} \setminus \mathcal{B}$.

Amennyiben $\mathcal{B}$ -t az $\mathbf{A}$ mátrix alsó indexébe írjuk ($\mathbf{A}_{\mathcal{B}}$), úgy arra a részmátrixra hivatkozunk, amely az $\mathbf{A}$ -ból $\mathcal{B}$ által leírt sorrendben vett oszlopokból áll. Hasonló módon értelmezhető $\mathbf{A}_{\mathcal{R}}$ is. Így

$$\mathbf{A} = [\mathbf{A}_{\mathcal{B}} \mid \mathbf{A}_{\mathcal{R}}].$$

Az $\mathbf{x}$ és $\mathbf{c}$ vektorok szintén felbonthatóak:

$$\mathbf{c} = \begin{bmatrix} \mathbf{c}_{\mathcal{B}} \\ \mathbf{c}_{\mathcal{R}} \end{bmatrix} \quad \text{és} \quad \mathbf{x} = \begin{bmatrix} \mathbf{x}_{\mathcal{B}} \\ \mathbf{x}_{\mathcal{R}} \end{bmatrix}. \quad (3.1)$$

A jelölés tovább egyszerűsíthető, amennyiben $\mathbf{A}_{\mathcal{B}}$ -t $\mathbf{B}$ -vel, $\mathbf{A}_{\mathcal{R}}$ -t $\mathbf{R}$ -rel helyettesítjük:

$$\mathbf{A} = [\mathbf{B} \mid \mathbf{R}]. \quad (3.2)$$

3.1. Definíciók és jelölések

Az előzőeket behelyettesítve a 2.5 egyenletbe:

$$\mathbf{B}\mathbf{x}_{\mathcal{B}} + \mathbf{R}\mathbf{x}_{\mathcal{R}} = \mathbf{b}, \quad (3.3)$$

és amennyiben létezik $\mathbf{B}^{-1}$, úgy

$$\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_{\mathcal{R}}). \quad (3.4)$$

Ezekből jól látszik, hogy a bázisváltozók *függő*, a nembázis-változók *független* változók olyan értelemben, hogy a nembázis változók egyértelműen meghatározzák a bázisváltozók értékeit.

A megoldások során a nembázis-változók rendszerint l_k értéket vesznek fel, bár az 1-es és 4-es típusú változók felső korlátjukon is állhatnak. Ennek jelölésére érdemes bevezetni az $\mathcal{U}$ és $\mathcal{L}$ indexhalmazt:

$$\mathcal{U} = \{k : k \in \mathcal{R}, \quad x_k = u_k\}$$

$$\mathcal{L} = \{k : k \in \mathcal{R}, \quad x_k = l_k\}.$$

Ebben az esetben a 3.4. egyenletből kiindulva, $\mathbf{x}_{\mathcal{B}}$ értékei a következőképpen alakulnak:

$$\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}(\mathbf{b} - \sum_{k \in \mathcal{U}} u_k \mathbf{a}_k - \sum_{k \in \mathcal{L}} l_k \mathbf{a}_k). \quad (3.5)$$

A $\mathcal{B}$, $\mathcal{U}$ és $\mathcal{L}$ halmazoknak megfelelő $\mathbf{x}_{\mathcal{B}}$ -t **alapmegoldásnak** hívjuk, amelyben tehát a nembázis-változók valamelyik korlátjukon vannak. Amennyiben $\mathbf{x}_{\mathcal{B}}$ megvalósítható, azaz $\mathbf{l}_{\mathcal{B}} \leq \mathbf{x}_{\mathcal{B}} \leq \mathbf{u}_{\mathcal{B}}$, úgy **megengedett alapmegoldásnak** (*basic feasible solution, BFS*) nevezzük.

Ha az $\mathcal{U}$ halmaz üres, úgy az összes lehetséges alapmegoldásnak a száma:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}.$$

A lineáris programozás egyik fő elméleti tétele kimondja, hogy amennyiben egy problémának létezik optimális megoldása, úgy létezik legalább egy **optimális alapmegoldása** (*optimal basic solution*) [7]. Így tehát elegendő csak az alapmegoldások között keresni az optimálisat.

3.2. A szimplex algoritmus

A **szimplex módszertan** egy iteratíván működő numerikus algoritmus, amely - bizonyos megkötések mellett - véges számú lépés után a következő válaszok valamelyikét szolgáltatja:

1. A problémának nincs megvalósítható megoldása.
2. A probléma nem korlátos.
3. Egy optimális megoldás megtalálva.

Két bázis szomszédosnak mondható, amennyiben csak egy oszlopvektorukban különböznek egymástól. Minden iterációban a szimplex módszer egy szomszédos bázisra lép át, miközben a célfüggvényérték javul (vagy esetleg változatlan marad).

A szimplex algoritmus iteratív voltának megértéséhez először a megvalósíthatóság és az optimalitás feltételeit kell áttekinteni.

3.2.1. Megvalósíthatóság és optimalitás

A könnyebbesség kedvéért írjuk fel újra a megfogalmazott feladatot, azaz a 2.4-2.6. lineáris egyenletrendszert.

$$\min \quad z = \mathbf{c}^T \mathbf{x} \quad (2.4)$$

$$\text{s.t.} \quad \mathbf{Ax} = \mathbf{b} \quad (2.5)$$

$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \quad (2.6)$$

Kiindulásképpen feltételezzük, hogy adott egy feasible bázis, amit a 3.4. egyenlet szerint felírhatunk:

$$\mathbf{x}_B = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_R). \quad (3.4)$$

Vizsgáljuk a célfüggvényérték alakulását a bázis- és nembázis-változók értéke alapján.

$$z = \mathbf{c}^T \mathbf{x} = \mathbf{c}_B^T \mathbf{x}_B + \mathbf{c}_R^T \mathbf{x}_R \quad (3.6)$$

Ami a 3.4. egyenletből:

$$z = \mathbf{c}_B^T \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_R) + \mathbf{c}_R^T \mathbf{x}_R \quad (3.7)$$

És végül átcsoportosítva:

$$z = \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{b} + (\mathbf{c}_R^T - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{R}) \mathbf{x}_R \quad (3.8)$$

3.2. A szimplex algoritmus

Megkülönböztetett figyelmet érdemel a nembázis-változók együtthatója. Az x_k -ra vonatkozó ún. **redukált költség** (*reduced cost*):

$$d_k = c_k - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{a}_k \quad (3.9)$$

Vizsgáljuk a célfüggvényérték elmozdulását az x_k nembázis-változó értékének megváltozásának hatására. Jelöljük t -vel x_k megváltozásának mértékét. Ahogy a 3.7. egyenletből is látható, z visszavezethető kizárólag a nembázis-változók értékeire. Ezek alapján következik, hogy a célfüggvényérték a td_k értékkel fog változni:

$$z(t) = z + td_k \quad (3.10)$$

Ahogy már korábban megelőlegeztük, a szimplex algoritmus a megvalósítható megoldásokat iterálja, amíg optimálishoz nem ér. Ebből következik, hogy már meglévő jó eredményt soha nem fog elrontani. Így a következő feltételnek is teljesülnie kell:

$$l_k \leq x_k + t \leq u_k \quad (3.11)$$

azaz x_k új értéke x_k individuális korlátain belülre kell hogy essen.

Emlékeztető. A szimplex módszertan bázismegoldások közül válogat. Ez azt jelenti, hogy minden nembázis-változó valamelyik korlátján található.

Összefoglalva egy táblázatban a különböző típusú nembázis-változók lehetséges elmozdulásait, a következő hat esettel kell számolni: ¹

x_k típusa	értéke	Feltétel $x_k + t$ -re vonatkozóan	t lehetséges értéke
0	$l_k = u_k$	$x_k + t = l_k$	$t = 0$
1	l_k	$l_k \leq x_k + t \leq u_k$	$0 \leq t \leq u_k - l_k$
1	u_k	$l_k \leq x_k + t \leq u_k$	$l_k - u_k \leq t \leq 0$
2	l_k	$l_k \leq x_k + t$	$0 \leq t$
3	0	$-\infty \leq x_k + t \leq \infty$	$-\infty \leq t \leq \infty$
4	u_k	$x_k + t \leq u_k$	$t \leq 0$

3.1. táblázat. Az x_k változó lehetséges elmozdulásának mértéke

¹A 3-as típusú, azaz szabad változók a megoldáskeresés során ideiglenesen felvehetik a 0 értéket. Ilyenkor superbázis-változóként kell rájuk tekinteni.

3. A szimplex módszertan

A 3.1. táblázat tehát aszerint lett kialakítva, hogy az egyes nembázis-változók miképpen mozdulhatnak el úgy, hogy továbbra is megengedett értékre kerüljenek.

A 3.10. egyenlet alapján látható, hogy mivel minimalizálási feladatról van szó, így z értéke az esetben csökkenthető, ha a td_k kifejezés negatív. Másképpen megfogalmazva: z értéke nem csökkenthető tovább, azaz **optimális**, ha *nem* található olyan nembázis-változó, melyre a td_k kifejezés negatív. A 3.2. táblázat összefoglalja az imént megfogalmazott **optimalitási feltételt**.

x_k típusa	értéke	t_k lehetséges értéke	d_k elvárt értéke	Megjegyzés
0	$l_k = u_k$	$t = 0$	tetszőleges	
1	l_k	$0 \leq t \leq u_k - l_k$	$0 \leq d_k$	
1	u_k	$l_k - u_k \leq t \leq 0$	$d_k \leq 0$	
2	l_k	$0 \leq t$	$0 \leq d_k$	$k \in \mathcal{U}$
3	0	$-\infty \leq t \leq \infty$	$d_k = 0$	
4	u_k	$t \leq 0$	$d_k \leq 0$	$k \in \mathcal{L}$

3.2. táblázat. Az x_k változó optimalitási feltétele

3.2.2. Az algoritmus leírása

A szimplex módszertan alapvetően a következő lépésekből áll.

1. **Kezdőbázis keresés.** Induló bázis meghatározása. Triviális eset: logikai bázis, azaz a logikai változókból alkotott mátrix kiválasztása.
2. **Első fázis.** A kezdőbázis iterálása a megvalósíthatósági feltétel kielégítéséig, azaz amíg az egyes változók mind megfelelnek az individuális korlátaiknak.
 - (a) **Measure of infeasibility.** Annak a mérőszámnak a meghatározása, amely leírja, hogy az egyes bázisban szereplő változók milyen mértékben sértik az individuális korlátjaikat, azaz aktuális értékük mennyire esik túl a felső-, vagy esik alá az alsó határuknak. Amennyiben ez az érték 0, úgy áttérés a második fázisra.
 - (b) **Pricing.** A bázisba újonnan bekerülő változó meghatározása. Kiindulás az 1-es fázisú redukált költségek alapján, az individuális korlátokat sértő változók közül történő választással. Feltétel: a measure of infeasibility értékének javítása (csökkentése).

3.2. A szimplex algoritmus

(c) **Ratio test.** A bázisból kikerülő változó meghatározása. Feltételek: 1. a bekerülő változó megengedett értéket vegyen fel a bázisban, 2. a kilépő változó megengedett értéket vegyen fel a bázison kívülre kerülve, 3. minden egyéb bázisváltozó, amely eddig megengedett értéken volt, maradjon is megengedett.

(d) **Iteráció.** Visszatérés a fázis első lépésére.

3. **Második fázis.** A feasible bázismegoldások iterálása az optimalitási feltétel kielégítéséig.

(a) **Pricing.** A bázisba újonnan bekerülő változó meghatározása. Kiindulás a 2-es fázisú redukált költségek alapján, az optimalitási feltételt sértő változók közül történő választással. Feltétel: a célfüggvényérték javítása vagy helybenhagyása. Többféle kiválasztási szabály létezik, legelterjedtebb a Dantzig-szabály alkalmazása. Amennyiben minden változó optimális értéken van, úgy az eljárás megáll „optimális megoldás” állapottal.

(b) **Ratio test.** A bázisból kikerülő változó meghatározása. Kiindulás a bekerülő változó együtthatómátrixbeli a_q oszlopának nemnulla elemei alapján. Feltételek: 1. a bekerülő változó megengedett értéket vegyen fel a bázisban, 2. a kilépő változó megengedett értéket vegyen fel a bázison kívülre kerülve, 3. minden egyéb bázisváltozó maradjon megengedett értéken. Két speciális eset: 1. A pricing által kijelölt változó mégsem lép be a bázisba, viszont átlép a másik korlátjára (*bound flip*) 2. Ha nem található a fenti feltételeket kielégítő kimenő változó és bound flip sem hajtható végre, akkor az eljárás megáll „nem korlátos” állapottal.

(c) **Update.** A célfüggvényérték, majd a bázisváltozók értékeinek frissítése. Kilépő változó és belépő változó értékeinek frissítése.

(d) **Iteráció.** Visszatérés a fázis első lépésére.

Bár elméleti szempontból a szimplex algoritmus a fentiek szerint működik, a gyakorlatban a számítógépes adatábrázolásból adódó numberikus problémák miatt előfordulhat, hogy a második fázisból a feladat az iterációk során visszaesik első fázisba. Az ilyen jellegű problémákat kiküszöbölendő egy modern implementációnak számításba kell vennie a megfelelő toleranciákkal történő számítást.

4. fejezet

Kezdőbázis keresés

Szimplex módszertan lényege, hogy a kezdetben valamely módon meghatározott megoldásból kiindulva, iterációnként egyet-egyet lépve halad az optimális megoldás felé. Könnyen felismerhető tehát, hogy a futási időt és ezzel a hatékonyságot leginkább befolyásoló tényező az **iterációs szám**.

A másik lényegi megfontolás mindezekből következően a **kezdőbázis** megválasztásának kérdése. Amennyiben egy jó minőségű megoldásból indulunk ki, úgy könnyen felgyorsíthatjuk a későbbi megoldási fázisokat is.

Kezdőbázis meghatározására több elterjedt módszer létezik. Alapvető követelményként megfogalmazható, hogy magának a keresésnek gyorsnak kell lennie, illetve a későbbi iterációs számot a lehető legjobban csökkenteni kell. Más szavakkal az a módszer a jobb, amelynél a probléma megoldása kevesebb erőforrást igényel, beleértve magának a kezdőbázisnak az előállítását is.

4.1. A kezdőbázis meghatározás szerepe

Ahogy az LP feladatok és a szimplex módszertan rövid történeti áttekintésében is láthattuk (2.2. rész), a operációkutatás szakterületén történő vizsgálatok a mai napig jelentős mértékben továbbra is a szimplex algoritmusra koncentrálnak. Az ilyen jellegű kutatások fontosabb célpontjai a következők [6]:

- degeneráltság és ciklikusság az LP feladatokban
- anti-degeneráltságot célzó szabályok
- a teljes táblázattranzformáció tömörítése (Revised Simplex Method, **RSM**)
- ritkás adattárolás és specializált algoritmusrészletek (Sparse Simplex Method, **SSX**)

4. Kezdőbázis keresés

- előfeldolgozási technikák bevezetése
- pricing és ratio test kiválasztási szabályainak hatékonyságjavítása

Ahogy a 3.2.2. részben a szimplex algoritmus leírása során már láthattuk, a megoldási folyamat a kezdőbázis meghatározásával kezdődik. A további lépések ebből kiindulva fognak az iterációk során előbb egy individuális korlátokat kielégítő, majd optimális megoldást eredményezni. Könnyen belátható ezek alapján, hogy a később optimálisnak ítélt megoldáshoz minél közelebbi kezdőbázis az iterációs számot drasztikusan lecsökkentheti. Emellett egy megfelelően megválasztott kezdőbázis az egyes és kettes fázis számára kedvező numerikus tulajdonságokat is eredményezhet.

A kezdőbázis-keresés tehát fontos lépés a megoldási folyamat során. Mindezek ellenére - a fentebb felsorolt területekhez viszonyítva -, mellőzött kutatási témáról van szó. Dolgozatom során Maros [7] és [9] művét vettem kiindulásnak.

4.2. Logikai bázis

A logikai bázist másképpen triviális bázisnak is nevezik. Nevét onnan kapta, hogy a bázisba kizárólag logikai változók kerülnek be, így $\mathbf{B} = \mathbf{I}$.

Az ilyen jellegű bázisok ugyan végtelenül egyszerűnek tűnnek, mégis érdemes fontolóra venni használatukat a következő okokból kifolyólag:

1. az előállítás pillanatok alatt megtörténik
2. $\mathbf{B}^{-1}$ -t rögtön ismerjük ($\mathbf{I}^{-1} = \mathbf{I}$)
3. az első iterációk gyorsak ezzel a bázisinverzzel számolva, és a modern implementációkban alkalmazott bázisinverz tárolás során is hatékonyan használható.

Amennyiben logikai bázist választunk, úgy a $\mathcal{B}$ halmaz meghatározása után a $\mathcal{B}$ és $\mathcal{U}$ halmazok a következőképpen alakulhatnak:

Alsó logikai bázis Az 1-es típusú változók alsó korlátra kerülnek, azaz

$$\mathbf{x}_{\mathcal{R}} = \mathbf{l}_{\mathcal{R}} \quad \text{és} \quad \mathbf{x}_{\mathcal{B}} = \mathbf{b} - \sum_{k \in \mathcal{L}} l_k \mathbf{a}_k .$$

Felső logikai bázis Az 1-es típusú változók felső korlátra kerülnek, azaz

$$\mathbf{x}_{\mathcal{R}} = \mathbf{u}_{\mathcal{R}} \quad \text{és} \quad \mathbf{x}_{\mathcal{B}} = \mathbf{b} - \sum_{k \in \mathcal{U}} u_k \mathbf{a}_k .$$

4.3. Crash bázisok

Kevert logikai bázis Minimalizálási feladat esetén a $c_j < 0$, maximalizálási feladat esetén a $c_j > 0$ együtthatójú 1-es típusú, nembázis-változók kerülnek felső korlátjukra, ellenkező esetben alsó korlátjukra.

A három féle logikai bázis közös jellemzői:

- A 0-s típusú változók fix értéküket veszik fel.
- A 2-es típusúak alsó-, a 4-es típusúak felső korlátjukra kerülnek.
- A 3-as típusú változók ideiglenesen a 0 értéket veszik fel, és szuperbázis-változónak tekintendők.

A logikai bázis generálása az individuális korlátokat tehát figyelmen kívül hagyja, ami a legtöbb esetben azt eredményezi, hogy az algoritmus sokáig az első fázisban kénytelen lépegetni.

Habár a logikai bázis számos kedvező tulajdonsággal rendelkezik, a későbbiekben eredményezett magas iterációszáma miatt a többi kezdőbáziskereső módszerrel szemben kevésbé használt algoritmus lett. Mindezek mellett több feladat során mégis igencsak kedvezően teljesít.

4.3. Crash bázisok

A crash bázisoknak két fő jellemzőjük van. Egyrészt a lehető legnagyobb mértékű triangularitás elérése, másrészt **minél több strukturális változó** beválasztása a bázisba. Ez utóbbi irányvonalat gyakorlati tapasztalatok eredményezték.

Nincs általános szabály arra, hogy a felső vagy az alsó **trianguláris** rész kialakítása hasznosabb-e, de mindkét esetben számos előnyös tulajdonságot kapunk:

1. a minden iterációban használt bázis inverz előállítása nem jár feltöltődéssel,
2. nagyon gyors műveletek (gyors előállítani és gyors vele számolni),
3. nagyfokú pontosság.

A feladat ilyenkor nem más, mint egy megfelelő algoritmus megvalósítása, amely a legnagyobb trianguláris részt megtalálja az **A** mátrixban.

4.3.1. Hagyományos crash eljárás (szimbolikus crash)

Nagyjából minden crash eljárás a következőkből indul ki.

4. Kezdőbázis keresés

Első lépésben a logikai változókból kialakított bázist választjuk megoldásnak, azaz **I**-t. Minden további lépésben **I** oszlopait cseréljük le egy-egy strukturális változó oszlopára úgy, hogy közben a triangularitást megőrizzük. Az algoritmus leírását a 4.1. ábra tartalmazza.

1. Amennyiben $\mathbf{A}_S$ -sel jelöljük a strukturális változók oszlopaiból képzett mátrixot, úgy $\mathbf{A} = [\mathbf{A}_S \mid \mathbf{I}]$.
2. Legyen R_i a sorokban, C_j az oszlopokban szereplő nemnulla elemek száma az $\mathbf{A}_S$ mátrixot tekintve.
3. $M(i, j)$ kiválasztása:
 - (a) $R_i = \min_k \{R_k\}$
 - (b) Ha $R_i = 1$, j megválasztása egyértelmű,
 - (c) amúgy - tekintettel a $\mathbf{B}$ mátrixban lévő nemnulla elemek számára -, válasszuk a $\min_l \{C_l\}$ elemet.
4. Ha megvan a pivot elem, akkor azt megjegyezzük, és a neki megfelelő sort és oszlopot a további számítások során figyelmen kívül hagyjuk. A triangularitás biztosítása végett nemcsak a pivot elem oszlopát, hanem a pivot sorral érintkező összes oszlopot (ahol a metszetben nemnulla elem áll), ki kell hagyni a további számításokból.
5. Az eljárást addig folytatjuk a 3. pontra visszatérve, amíg vagy a sorok, vagy az oszlopok el nem fogynak.

4.1. ábra. A hagyományos crash algoritmus

A fenti algoritmus nem minden esetben talál teljes bázist, de a maradék pivot pozíciók automatikusan az egységvektorokkal lesznek feltöltve.

Az eljárást *szimbolikus* crashnek nevezik, mivel az $\mathbf{A}$ mátrix nemnulla elemeinek értékétől függetlenül választ kezdőbázist.

A meghatározott bázis minősége több összetevőből adódik: denzitás, megvalósíthatóság, célfüggvény értéke, stb. Ezek a szempontok együttesen határozhatják meg az egyes sor- és oszlopválasztások további befolyásolásának módját, amennyiben versenyhelyzet adódna.

4.3.2. LTSF crash eljárás

Az úgynevezett *LTSF crash* eljárás az előzőekben megismert szimbolikus crash algoritmus egy továbbfejlesztett változata, amely a triangularitás megtartása mellett a legnagyobb feasibility-ású kezdőbázist keresi (**L**ower **T**riangular **S**ymbolic crash designed to **F**easibility, alsó trianguális szimbolikus crash, megvalósíthatóságra tervezve) [7].

A különbség a hagyományos algoritmushoz képest abban rejlik, hogy tovább finomítja a nem egyértelmű helyzetekben a választás mikéntjét. Két fő koncepcióval rendelkezik:

- a szűk individuális korlátokkal rendelkező logikai változók eltávolítása és
- a tág individuális korlátokkal rendelkező strukturális változók bevétele.

Ezek kezeléséhez *prioritásfüggvényeket* célszerű bevezetni. A *sortípus* a megfelelő sorban szereplő logikai változó típusára vonatkozik, míg az *oszloptípus* kifejezés az adott strukturális változó típusára.

RPF	(<i>Row Priority Function</i>)	Sor prioritás függvény
CPF	(<i>Column Priority Function</i>)	Oszlop prioritás függvény

A hagyományos crash algoritmusban megismert sor- és oszlopválasztási szabály módosul. Míg az előbbit az *RPF*, utóbbit a *CPF* fogja meghatározni.

Sortípus	Prioritás	Megjegyzés
0	3	A legnagyobb prioritásúak az egyenlőségek
1	2	Tartomány típusú sorok
2	1	„≤” vagy „≥” típusú sorok
3	0	A legalacsonyabb prioritásúak a szabad sorok

4.1. táblázat. Sor prioritások (*Row Priority, RP*)

A legkedvezőbb esetben tehát egy 0-s típusú logikai változót cserélünk le egy 3-as típusú strukturális változóra: ezzel a lépéssel a logikai változó ki-, a strukturális változó pedig bekerül a bázisba.

A konkrét értékekhez néhány további fogalom bevezetésére van szükség.

4. Kezdőbázis keresés

Oszloptípus	Prioritás	Megjegyzés
0	0	A legalacsonyabb prioritásúak a fixált változók
1	1	1-es típusú változók
2	2	2-es és 4-es típusú változók
3	3	A legnagyobb prioritásúak a szabad változók

4.2. táblázat. Oszlop prioritások (*Column Priority, CP*)

RT(i)	Az i . sor logikai változójának típusa.
RC(i)	Az A_S mátrix aktív oszlopaait vizsgálva az i . sorban lévő nemnulla elemek száma.
RI(i)	Az i . sort metsző aktív oszlopok indexeinek halmaza.
CT(j)	A j . oszlop típusa.
CC(j)	Az A_S mátrix aktív sorait vizsgálva a j . oszlopban lévő nemnulla elemek száma.
AR	Az aktív sorok indexeinek halmaza.

Ezek alapján a pivotelem sorindexe (r) a következőképpen számolandó:

$$RPF(i) = RP(RT(i)) - 10 \times RC(i)$$

Az egyenletben a 10 varázsszámnak (*magic number*) tűnhet. Ez egy apró gyakorlati megfontolás, amely segít a különböző $RPF(i)$ értékeket jobban elkülöníteni egymástól. Ezután felírható r kiszámítása is:

$$r = \max_{i \in AR} \{RPF(i)\}$$

Hasonlóan számolható j oszlopindex $CPF(j)$ függvény alapján, azzal a feltétellel, hogy csak azokra az oszlopokra értelmezett, amelyeknek az r . sorral vett metszetében nemnulla elem áll.

$$CPF(j) = CP(CT(j)) - 10 \times CC(j)$$

Ezekből a k . oszlopot választjuk:

$$k = \max_{j \in RI(r)} \{CPF(j)\}$$

Ez alapján egy konkrét lépés végrehajtható. Az algoritmus teljes vázlata a 4.2. ábrán látható.

1. Amennyiben A_S -sel jelöljük a strukturális változók oszlopaiból képzett mátrixot, úgy $A = [A_S \mid I]$.
2. Felírjuk a prioritástáblákat.
3. A fix változókat jelölő oszlopokat és a szabad változókat tartalmazó sorokat kizárjuk. A_S megmaradó részét *aktív*nak hívjuk
4. A bázist az egyésmátrixszal inicializáljuk: $B = I$
5. A_S aktív részére kiszámítjuk az $RC(i)$ és a $CC(j)$ értékeket.
6. Kijelöljük a sort RPF alapján.
7. Amennyiben a sorválasztás sikerül:
 - (a) Kijelöljük az oszlopot CPF alapján.
 - (b) Újradefiniáljuk A_S aktív részét és frissítjük a megfelelő sor- és oszlop-számlálókat.
 - (c) Visszatérünk a sorkiválasztási lépéshez.
8. A Crash(LTSF) eljárás véget ér.

4.2. ábra. Az LTSF crash algoritmus

4.3.3. ADG crash eljárás

Az úgynevezett *ADG crash* algoritmus szintén a crash eljárások családjába tartozik. Az LTSF crashtól eltérően nem az individuális korlátok betartását célozza meg, hanem a szimplex algoritmus egyik gyenge pontjára, a degeneráltságra koncentrálnak (**AntiDeGeneracy Strategy**).

Egy bázismegoldást akkor nevezünk **degeneráltnak**, ha legalább egy bázisváltozó a korlátján van. A degeneráltság mértékét D -vel szokás jelölni, értéke a korláton lévő bázisváltozók számosságából adódik.

A degeneráltság azért kritikus pont a szimplex algoritmus során, mert megnöveli annak az esélyét, hogy érdemi javítás nélküli iterációs lépés történjen.

Megjegyzés. A második fázis *pricing* lépése után a következő szituáció léphet fel. Amennyiben a kiválasztott oszlop tartalmaz olyan nemnulla elemet, amely korláton lévő bázisváltozóhoz tartozik, úgy a hagyományos *ratio test* egyrészt ezt a bázisváltozót fogja megjelölni kilépő változónak, másrészt minden érték (belépő változó értéke, kilépő változó értéke, célfüggvényérték, többi bázisváltozó értéke) változat-

4. Kezdőbázis keresés

lan marad. A ratio test részletes tárgyalása megtalálható többek közt Maros [7] művében.

Az ADG crash eljárás célja olyan kezdőbázis meghatározása, amely elébe megy a degeneráció okozta problémáknak, de közben a triangularitást megtartja.

A bázisváltozók értékeit a 3.4. egyenlet alapján kaphatjuk meg.

$$\mathbf{x}_B = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_R). \quad (3.4)$$

Tekintsük azt a feladatosztályt, amelyben a modell csak 0, 1, 2 és 3 típusú változókkal rendelkezik, és minden alsó korlát értéke 0.

Megjegyzés. Könnyen belátható, hogy bármely LP feladat ekvivalens átalakításokkal ilyen formára hozható. Elnevezése: CF-1 (**Computational Form #1**). Használata nagy mértékben elterjedt, mivel a csökkenti az algoritmusok leírásának bonyolultságát.

Indítsuk az ADG algoritmust alsó logikai bázisról. Ebben az esetben a bázisváltozók értéke a következőképpen alakul:

$$\beta = \mathbf{B}^{-1}\mathbf{b}$$

Az ADG minden lépés során olyan soron próbál pivotálni, amelyre $b_i \neq 0$, mivel így a frissítés során feltöltődés fog fellépni, ami csökkenti a degenerációt.

Az ADG algoritmus szöveges leírása a 4.3. ábrán látható.

Abban az esetben ha a megoldandó feladat az eredeti individuális korlátokat tartalmazza, úgy az ADG eljárás ebben a formában nem alkalmazható. A probléma megoldására további kutatások szükségesek.

4.4. Egyéb kezdőbázis kereső módszerek

Az előző alfejezetek során a ma legismertebb kezdőbázis kereső stratégiákat mutattam be. Következzenek most, felsorolás szintjén az indulóbázis meghatározásának további lehetséges módjai.

4.4.1. Parciális kezdőbázis

Számos esetben elképzelhető, hogy a feladatmegoldás speciális körülményeit figyelembe véve, a modellező szoftver vagy szakértő ajánlással élne a kezdőbázisba beválasztandó strukturális változókkal kapcsolatban.

4.4. Egyéb kezdőbázis kereső módszerek

Ez az ajánlás lehet explicit módon meghatározott (melyik változó melyik bázispozíciót foglalja el), vagy akár csak egy lista, amelynek változóit prioritással kell kezelni.

A parciális kezdőbázis alkalmazása önmagában nem alkalmazandó keresési stratégia, inkább a már megismert algoritmusok paraméterezéséről lehet szó.

4.4.2. A CPLEX bázis

Bixby nevéhez köthető a CPLEX optimalizációs stúdió [10], aki 1991-ben több-részes cikksorozatban mutatta be a rendszerében alkalmazott technikákat [11]. A CPLEX kezdőbázis keresési stratégiája hasonlít az LTSF crash eljárásához abban a tekintetben, hogy a változókat szintén prioritáslista alapján kezeli, és hogy ezen a listán a tág individuális korlátokkal rendelkező változók előre kerülnek. További hasonlóság a triangularitás megtartása. Egyik alapvető különbség, hogy a CPLEX algoritmusa a célfüggvény együtthatókat is aktívan felhasználja.

Bixby az általános keresési eljárásokkal ellentétben nem az optimálisához legközelebb eső kezdőbázis megtalálására törekszik (a keresés költségessége és heurisztikus jellege miatt), hanem az előreláthatóan felesleges lépések kiküszöbölésére.

Bixby tapasztalatai alapján a CPLEX főként kis és közepes méretű feladatokon a leghatékonyabb, azaz ott képes a legtöbb iterációs szám-csökkenést elérni.

4.4.3. Szakító algoritmus

Gould és Reid nagy ritkás rendszerek számára dolgozott ki ígéretes algoritmust a kezdőbázis meghatározására [12]. Céljuk, hogy elfogadható számítási igény mellett azt a bázist határozzák meg, amely leginkább betartja az individuális korlátokat (csökkentve ezzel a szimplex első fázisának szükségletét). Eljárásuk a „szakító algoritmus” *tearing algorithm* nevet kapta.

A fő koncepció abban áll, hogy az eredeti A együtthatómátrixot blocktrianguláris részmátrixokra bontja, majd azokra kis LP feladatokként tekint. Ezek a részfeladatok az eredeti, teljes mátrixot transzformáló, egyszerűbb szimplex algoritmussal megoldhatóak.

Az algoritmus ugyan kiváló eredményeket ért el közepes méretű feladatokon, használata mégsem terjedt el a modern szoftverek implementációiban.

4. Kezdőbázis keresés

1. Amennyiben $\mathbf{A}_S$ -sel jelöljük a strukturális változók oszlopaiból képzett mátrixot, úgy $\mathbf{A} = [\mathbf{A}_S \mid \mathbf{I}]$.
2. Tekintsük $\mathcal{H}_1$ halmazt, melyre: $\mathcal{H}_1 = \{i : \beta_i \neq 0, k_i \in \mathcal{I}\}$
3. Az oszlopválasztás a strukturális változók közül történik. Számoljuk ki, hogy a k . oszlopban hány nemnulla elem van, ahol a jobboldal értéke sem 0.

$$j_k = \sum_{\substack{i \in \mathcal{H}_1 \\ a_k^i \neq 0}} 1$$

4. Az oszlopkiválasztás $j_q = \max_k j_k$ alapján történik. Amennyiben a maximum nem egyértelmű, akkor célszerű a több nemnullát tartalmazó oszlop kiválasztása. Ha ez a maximális érték 0, úgy az algoritmus megáll.
5. Amennyiben a sorválasztás nem egyértelmű, azaz $j_k \geq 1$, úgy több lehetőség is van a versenyhelyzet feloldására. Mivel a pivot elem kiválasztásakor nemcsak a pivot oszlopa és sora kerül ki a további számításokból, hanem a triangularitás biztosítása miatt a pivot sorral nemnulla elemmel érintkező oszlopok is, így célszerű a kevesebb nemnulla elemmel rendelkező sor preferálása.
6. β frissítése - általában elég csak szimbolikusan, azaz a konkrét értékek nem, csak a nulla-nemnulla pozíciók számítanak. A pontos számítás menete: legyen $\mathbf{E}$ egységmátrix, majd a q . oszlopát cseréljük le a pivot oszlopra; ezután $\beta = \mathbf{E}\beta$
7. Újradefiniáljuk $\mathbf{A}_S$ aktív részét és frissítjük a megfelelő sor- és oszlopszámlálókat.
8. Visszatérés $\mathcal{H}_1$ definiálásához.

4.3. ábra. Az ADG crash algoritmus

5. fejezet

A PanOpt keretrendszer

A Pannon Egyetem Rendszer- és Számítástudományi Tanszékén működő Optimalizálási Kutatócsoport által tervezett *PanOpt* rendszer egy általános jellegű megoldó szoftver. A projekt első fázisában a több egymással együttműködő modul megtervezése, a koncepciók és konvenciók kialakítása volt elsődleges, majd a különböző részegységek megvalósítása.

A szoftver jelenleg elsősorban nagy méretű, ritkás LP feladatok szimplex algoritmus alapú megoldására képes: a feladat beolvasásával kezdődően, az előfeldolgozás, a kezdőbázis keresés, majd az iterációk után különböző részletességű megoldás-kiírató lehetőségek állnak rendelkezésre.

Fontos, hogy az egyes részek megfelelően paraméterezhetőek legyenek, valamint hogy jól kidolgozott interfészen legyen képesek kommunikálni.

A fejezet elején röviden áttekintem a keretrendszer legfontosabb tulajdonságait, majd ezt követően az általam megvalósított kezdőbázis kereső algoritmusokat vizsgálom megvalósítási kérdések és teszteredmények alapján.

5.1. Szimplex módszertan nagy feladatoknál

Az eredeti szimplex algoritmus nagyszerű felismeréseken alapul, és elméletben képes tetszőleges LP feladat megoldására. Fontos azonban felismerni, hogy mivel magának a feladatosztálynak a kimerítő kutatásait az ipari szükségletek hívták életre, így a gyakorlatban is kellően hatékony megoldásra van szükség.

A gépi ábrázolás végeessége numerikus problémákat indukált, a nagy méretű, ritkás együtthatómátrixszal rendelkező feladatok fel hatalmas tárhelyigénnyel bírtak. Mindemellett az összes érték iterációnként történő frissítése a feladat méretének növekedésével óriási futási időket eredményezett.

5. A PanOpt keretrendszer

Az ilyen jellegű megfontolások miatt előbb az „újragondolt” szimplex algoritmus (*revised simplex method*), majd a ritkás szimplex (*sparse simplex*) került kidolgozásra. Ezek lényegi elgondolásai a következők:

- A teljes mátrixtranszformáció kiszámolásának elhagyása
- Részszerkezetek során kapott értékek felhasználása és karbantartása
- Ritkás tulajdonságot kezelni képes adatszerkezetek (ritkás vektorok, mátrixok, azokhoz megfelelő operátorok implementálása)
- Toleranciák használata az algoritmus különböző pontjain

Az imént felsorolt tulajdonságok nélkül ma már elképzelhetetlen egy szimplex megoldó szoftver készítése. A *PanOpt* rendszer tervezésekor természetesen mind ezeket, mind számos egyéb jellemzőt szem előtt tartottunk.

Egy LP megoldó szoftver validálásának *de facto* módja az ún. *NETLIB* feladatoknak a megoldása [13]. Ez az internetes, LP feladatokat tartalmazó adatbázis különféle méretű és tulajdonságú modelleket tartalmaz, amelyeknek sikeres és gyors megoldása belépőt jelenthet a versenyképes szimplex implementációk közé.

Ennek szellemében a később ismertetett tesztadatok a *NETLIB*-ből jól ismert feladatokra fognak épülni.

5.2. Technikai háttér

A *PanOpt* keretrendszer több résztvevős projekt, amelyben nemcsak a naprakész hatékony működés integrálása fontos a modulok tulajdonosai számára, hanem a kutatói tevékenységből adódóan kísérletezésre, könnyű kódátszervezésre vagy -újrahasználásra is gyakran szükség van.

Az implementálás során a csoport minden résztvevője együtt kialakított, írásban rögzített elvek mentén haladt, így biztosítva a közöt munka gördülékenységét.

Programozási nyelv

Programozási nyelvnek a C++-t választottuk [14]: így nem kellett egy operációs rendszer mellett sem elköteleznünk magunkat. Másik fontos szempont volt, hogy kellően gyors és alacsonyszintű, de mégis jól áttekinthető kód születhessen. A C++ a nyelvvel szemben támasztott minden követelményünknek megfelelt, és így modern, objektumorientált környezetbe helyezhettük algoritmusainkat.

Függőség

A keretrendszer tervezésekor fontos szempontnak találtuk, hogy az elkészült szoftver önállóan futtatva is képes legyen működni, valamint külső függvénykönyvtárként is elérhető legyen. Használata további külső komponenseket nem igényel.

Adattárolás

A hatékony adattárolásra saját típusokat vezettünk be az *algebrai magban*: ritkás vektor (*sparse vector*), sűrű vektor (*dense vector*), valamint az adott jellegű vektorból felépülő *mátrix* típust.

Általában óriási mennyiségű adatról van szó már egy átlagos méretű ipari feladat esetében is. Ezek nagy része valamely más program kimeneteként áll elő, és csak kevés nemnulla elemet tartalmaz. Ezek alapján adódik, hogy a 0 elemek explicit módon ne legyenek tárolva - ezt hívják ritkás tárolásnak. (A ritkás tárolás legkönnyebben láncolt listával oldható meg, amelyben a nemnulla elemek vannak fel-fűzve, értékük, sor- és oszlopindexeik szerint.) Mindemellett a vektorok tárolási stratégiája legyen transzparens az API-t használó felé, azaz egy tetszőleges elemre való hivatkozáskor a vektor értelmes adatot adjon vissza, attól függetlenül, hogy az explicit módon tárolva van-e.

A program vázlata

A program váza a legfontosabb részegységekkel az 5.2. ábrán látható.

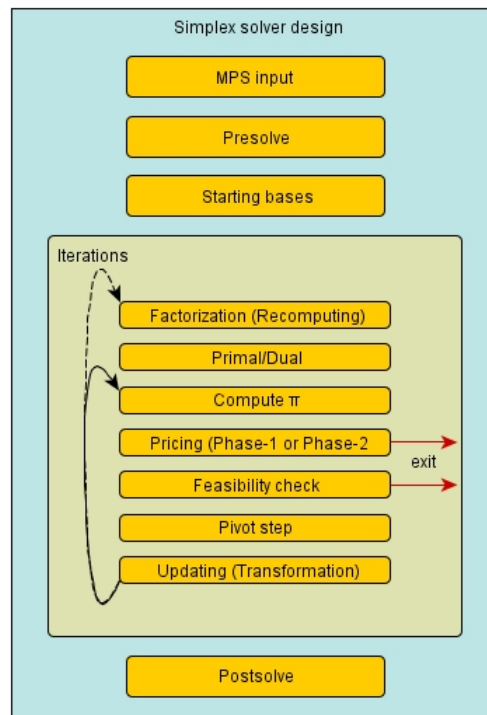
Jól látható, hogy a program moduljai szorosan korrelálnak a 3.2.2. részben tárgyaltakkal, ahogy a simplex módszertan alapvető lépéseit mutattam be.

Egy osztály publikus adattagjai és függvényei adják az adott modul programozói interfészét, másnéven *API*-jét (Application Programming Interface). A program legfontosabb osztályainak diagramjait az 5.2. ábra tartalmazza.

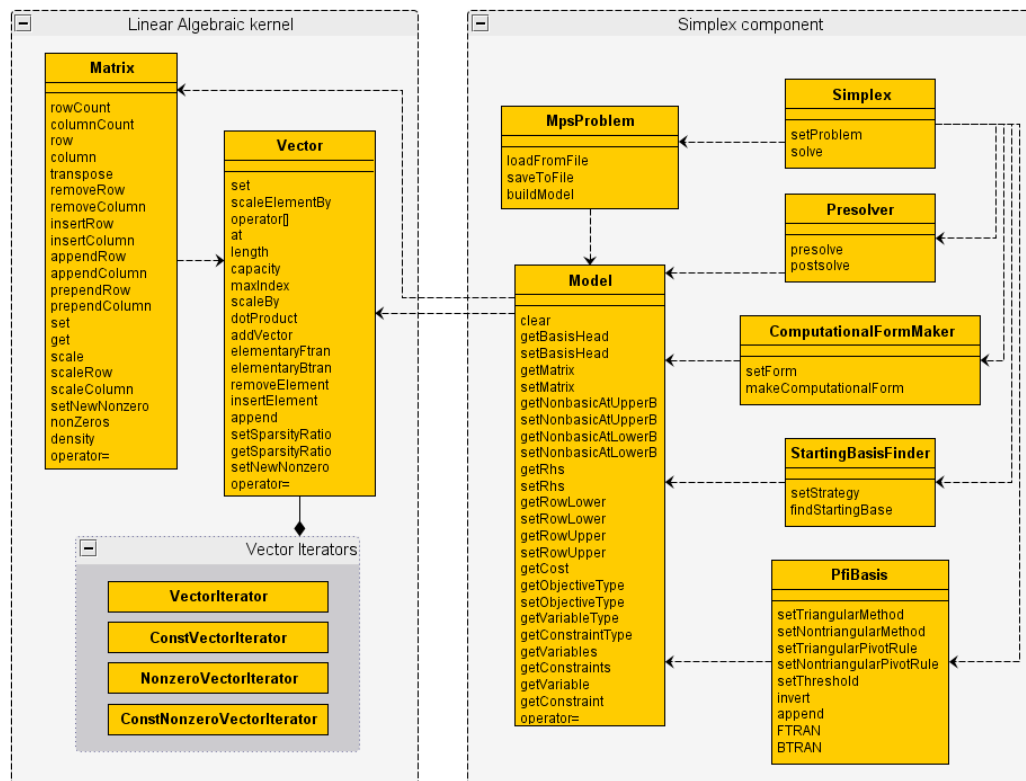
5.3. A kezdőbázis keresés implementálása

A kezdőbázis kereső részmodul alapvető feladata, hogy a probléma modelljét megkapva futása végeztével ezt **bázisfejet** szolgáltatson. A bázisfej rész részéből áll: bázisban lévő változók indexeinek rendezett listája, felsőkorláton lévő nembázis változók indexeinek halmaza, alsókorláton lévő nembázis változók indexeinek halmaza.

5. A PanOpt keretrendszer



5.1. ábra. Az architektúra vázlata



5.2. ábra. Legfontosabb adatszerkezetek

5.3. A kezdőbázis keresés implementálása

Működését annak elkezdődése után már nem befolyásolják külső elemek, azaz minden szükséges paramétert a modul inicializálásakor kell átadni.

Mivel a dolgozat célja több különböző keresési stratégia összehasonlítása, ezért rugalmasan kell megoldani a futások során az egyes algoritmusok kiválasztását.

A modul létrehozása során fokozottan ügyelni kell az egységbezárásra: az implementációs részletek (algoritmusok egymáshoz való viszonya, belső adattárolás, saját használatú metódusok) rejtve maradjanak a modul használója számára.

5.3.1. Egy kezdőbázis kereső algoritmus általános felépítése

A különböző kezdőbázis kereső stratégiák elméleti áttekintése, majd implementációs tapasztalatai alapján a következő általános forma írható fel az egyes algoritmusok felépítésére.

1. **Constructor** Inicializálás paraméterben kapott adatokkal. Számos változó-hivatkozást célszerű elmenteni ebben a lépésben, még olyan áron is, ha ezzel memóriát pazarolunk. A mai tendenciák alapján fel kell ismerni, hogy már nem a memória, hanem a sebesség a szűk keresztmetszet.
2. **Preinitialization** [Opcionális] Futási előtti utolsó lehetőség néhány paraméter beállítására. Tipikusan ebben a lépésben specializálhatóak az egyes algoritmusverziók.
3. **Initialization** [Opcionális, bár szinte mindig elvárt] Logikai bázis felépítése, amely később minden algoritmusnak helyesen inicializált változókat biztosít. Ez a gyakorlatban azt jelenti, hogy egyrészt minden strukturális változónak lesz definiált értéke, másrészt az általában oszlopkicserélésre épülő algoritmusok számára elkészül a kezdeti bázismegoldás.
4. **Select & Execute** Pontos keresési módszer kiválasztása, futtatás. Ebben a lépésben történik a valós munka, azaz a kezdőbázis meghatározása a rendelkezésre álló adatok alapján, a kijelölt módszert alkalmazva.
5. **Update** A feladat modelljében a bázisfej (bázispozíciók, valamint alsó- és felső korláton lévő nembázis-változók listája) frissítése.

Nyilvánvalóan logikai bázis alkalmazásakor az utolsó lépésekkel rövidül a kereső modul feladata.

További megjegyzés, hogy amennyiben több különböző eljárást szeretnénk egymás után (egymást eredményeit felhasználva) lefuttatni, úgy a *Select & Execute* és

Update lépések ismétlésével tudjuk ezt elérni. Fontos felismerni azonban, hogy egy részeredményekről indított algoritmusnak másképp kell viselkednie, mint ahogy a hagyományos leírások tartalmazzák.

Mivel a kutatásaink során több ilyen igény is felmerült, így minden logikai bázistól eltérő kereső algoritmus módosításra került, hogy kezelhessék az expliciten adott parciális bázisról indítást.

5.3.2. Segéd eljárások

Az algoritmusok implementálása során több helyen is megjelentek hasonló funkcionalitású kódrészletek. Bár a sebességfokozás érdekében a lelkes programozó könnyen a kódok kiszervezése ellen emelné szavát, ez könnyelmű döntésnek bizonyulna.

Korai optimalizálás (**early optimization**) néven ismert az a jelenség, amikor főként sebesség- vagy tárhelykritikus rendszerek készítésekor a programozók már a fejlesztés korai szakaszában kisebb-nagyobb kódoptimalizálásokat építenek szoftverükbe. Ugyan a cél nemes, azonban már közepesen hosszú távon általában kiderülnek káros hatásai: a kód nehezebben átlátható és a még gyakran felmerülő változtatásokat rosszul kezeli.

Megjegyzés. Donald Knuth-nak, a legendás A számítógép-programozás művészete c. és egyéb más jelentős könyvek szerzőjének (aki többek közt a *T_EX* megalkotója is) tulajdonítják a híres idézetet: „Early optimization is the root of much evil.” Ez magyarul nagyjából annyit tesz, hogy a korai optimalizáció minden ördögi dolognak a forrása.

A fenti megfontolások alapján a gyakran felmerülő kódrészleteket külön függvényekbe emeltem ki.

AdjustVariableByType

A függvény célja, a változók értékeinek beállítása egy adott irányvonal alapján, de a típusok figyelembevételével.

A függvény bemenete egy ún. `Variable::STATE`, amelynek lehetséges értékei: `UNDECIDED`, `BASIC`, `NONBASIC_AT_LB`, `NONBASIC_AT_UB`, `SUPERBASIC`.

Erre a módszerre tipikusan logikai bázis esetén lehet szükség, amikor a nem-bázis változókat valamely értékükön szeretnénk elhelyezni. Mivel a simplex mód-

5.3. A kezdőbázis keresés implementálása

szertan bázismegoldásokon iterál, ezért a nembázis változók valamely korlátukon kell hogy álljanak.

Futásának eredményeképpen azok a nembázis változók, amelyekre értelmezhető a paraméterben megadott kérdés, a paraméterben megjelölt korlátjukra kerülnek. A többi nembázis változó szintén értelmes és megengedett korlátra kerül.

CalculateNonZeros

Ahogy a kezdőbázis kereső stratégiák elméleti tárgyalásakor megfigyelhettük, a triangularitás kialakításakor gyakran oszlop- és sorszámlálókkal dolgoztak az algoritmusok. Ezek nem jelentettek mást, mint az adott oszlopban vagy sorban lévő nemnulla elemek számát.

Ennél a függvénynél két fontos szempontot kell figyelembe venni. Az első az az egyszerű figyelmeztetés, hogy bár adott esetben a pseudo kódok a teljes sorokon, oszlopokon történő iterálásra buzdíthatnak, a nagy ritkás mátrixok tulajdonságaiból adódóan ezt a műveletet semmiképpen se szó szerint végezzük. A nemnulla elemeken való végigiterálásokra a `Vector::NonzeroIterator` segédosztály használható, az eredeti formában tárolt vektorokra pedig egyszerűen lekérdezhető a nemnulla elemek száma (`Vector::nonZeros()`).

A másik lényeges pont a parciális kezdőbázis miatt történt módosítások figyelembevétele. A foglalt sorok és oszlopok kiesnek a további számítások alól, így az egyes számlálók kialakításakor ezek ellenőrzése is szükséges.

UpdateBasisFromPivots

A különböző kezdőbázis keresések többsége egyesével halad pivotálás során, így a végén egy maximum m hosszúságú listába lesznek pivot elemek felfűzve. Minden pivot elem egy sor- és egy oszlopindexből áll össze.

A pivot vektorból a következőképpen olvashatjuk ki az eredményeket: végigmeigyünk a vektor elemein. Minden elem első tagja (`first`) azt a sort fogja kijelölni, amely bázisfej pozíción bekerül a változó a bázisba. Az elemek második (`second`) tagja a változót azonosítja az eredeti változólistában betöltött indexének értékével.

Az alábbi kódrészlet a valós kódból származik:

```
inline void SbfSuper::updateBasisFromPivots(const std::vector< std
    ::pair<int, int> >& pivots)
{
```

```

m_basisStructVarCount += pivots.size();

unsigned int i, j;
for (i = 0; i < pivots.size(); i++) {
    j = sb_basisHead[pivots.at(i).first];
    adjustVariableByType(&(sb_variables[j]), Variable::
        NONBASIC_AT_LB);
    sb_basisHead[pivots.at(i).first] = pivots.at(i).second;
    sb_variables[pivots.at(i).second].setState(Variable::BASIC);
}
}

```

ReconfigureFreeVariables

Ennek a függvénynek az a szerepe, hogy egy adott bázisfej alapján kigyűjtse, hogy mely változókat és mely bázispozíciókat szabad még felhasználni.

A bázisfej a bázis lévő változók indexeit tartalmazza. Megfelelő tárolás mellett az index értékéből megmondható, hogy strukturális vagy logikai változót jelöl-e. Amennyiben strukturális változóról van szó, úgy az tekinthető a parciális bázisból jövő információnak. A neki megfelelő elemet a bázisfejben foglaltként kell megjeleníteni, mint ahogy magát a strukturális változót is.

Rövid kódrészlet a valós kódból:

```

void StartingBasisFinder::reconfigureFreeVariables()
{
    m_freeBasisPositions = vector<bool>(orig_basisSize, true);
    m_freeStructVars = vector<bool>(orig_structVarCount, true);
    uint temp;
    for (uint i = 0; i < orig_basisSize; i++) {
        temp = m_basisHead.at(i);
        /* If in the basis there is a structural variable,
         * so the index is lesser than the number of structural
           variables */
        if (temp < orig_structVarCount) {
            /* Set the variable busy */
            m_freeStructVars[temp] = false;
            /* Set the basis position busy */
            m_freeBasisPositions[i] = false;

            /* We should not only deactivate the pivot row and the pivot
               column,

```

5.3. A kezdőbázis keresés implementálása

```
    * but all columns intersecting with pivot rows */
    const Vector& v = orig_problem->getMatrix().row(i);
    Vector::ConstNonzeroIterator vi;
    for (vi = v.beginNonzero(); vi != v.endNonzero(); vi++) {
        uint j = vi.getIndex();
        m_freeStructVars[j] = false;
    }
}
}
```

5.3.3. A kezdőbázis minőségének megítélése

A szimplex algoritmus kutatása legfőképpen az iterációk egyes lépéseinek gyorsítására koncentrál. Bár a kezdőbázis nagy mértékben képes befolyásolni a későbbi első és második fázisban szükséges lépések számát, még sincs jól definiált eljárásrendszer meghatározva ennek a választási stratégiának a mikéntje.

Általános elvárásként az fogalmazható meg, hogy a kezdőbázis keresés célja egy *kellően jó* bázis megtalálása, amelyről indítva a szimplex iterációs lépéseket a teljes megoldásra szánt futási idő csökken (beleszámolva magának a kezdőbázis előállításához szükséges számítási időt).

A további cél mindezek mellett már nem egyértelmű. Míg néhány kutató a későbbi optimálishoz legközelebb eső kezdőbázis azonosítására törekszik, addig mások az iterációs lépések „kezére játsza”, az előre láthatóan nem kellően hatékony lépések bekövetkezésének elkerülésére törekszik.

Fontos felismerni, hogy az egyes kezdőbázis kereső technikáknak közös jellemzője, hogy empirikus és heurisztikus információkra épít. A különböző jellegű LP feladatoknál más és más módszer hozhatja a legjobb eredményt.

Amennyiben az LP megoldó szoftver önálló futásra is képes (abban az értelemben, hogy részletes paraméterlista nélkül, alapértelmezett értékeket alkalmazva indul), úgy egy általánosan jól teljesítő kezdőbázis kereső stratégiát is meg kell jelölni.

Az egy kezdőbázisok minőségének összehasonlítására az alábbi jellemzők vizsgálatát javaslom.

Előállítás költsége

Az a futási idő, amely a kezdőbázis előállításához szükséges. Ez a mérőszám óvatosan kezelendő, mivel figyelembe kell venni a későbbi teljes megoldási folyamat idejét. Célszerű a másodperces adatok mellett a teljes futási időre vetített százalékos arányt is kifejezni.

Strukturális változók száma

A korábbi fejlesztési tapasztalatok alapján általánosan elmondható, hogy a szimplex által megtalált optimális megoldás több strukturális változót tartalmazott. Ezek alapján valószínűsíthető, hogy a több strukturális változót felsorakoztató kezdőbázis szerencsésebb választásnak bizonyulhat.

Triangularitás

A triangularitás kedvező tulajdonság, mivel a szimplex egyik kulcsfontosságú változója a $\mathbf{B}^{-1}$ mátrix. A trianguláris mátrixok inverze, amennyiben az létezik, szintén trianguláris mátrix. Ezen túlmenően, a főátlóban lévő pivot elemek választásával a mátrix átrendezésére sincs szükség, mivel az indexek megtartják a helyüket. Ennek eredményeképpen a trianguláris kezdőbázis átadása a szimplex számára nagyfokú könnyebbséget jelent, mivel egyszerűbb lesz vele számolni.

Mindezek mellett numerikusan is kedvező formáról van szó, valamint a későbbi bázisok mátrixain is lassabban fog mutatkozni feltöltődés.

Megjegyzés. *A feltöltődés (fill-in) azért kedvezőtlen folyamat, mivel minél sűrűbb a mátrix, azaz minél több nemnulla eleme van, annál több lehetőség adódik az iterációk oszlop- és sorválasztásai során, valamint értelemszerűen annál több adattal kell számolni.*

Degeneráltság

A degeneráltság mértékét az mutatja, hogy a bázisban szereplő változók közül hány darab áll valamelyik korlátján. Ahogy az ADG algoritmus leírásánál (a 4.3. rész) már láthattuk, a degeneráltság azért veszélyes, mert fokozza annak kockázatát, hogy érdemi javítás nélküli iterációs lépés történjen.

Amennyiben az iterációt végző modulok nem vagy csak kis mértékben foglalkoznak a degeneráció okozta problémával, úgy a szimplex hatékonyságának megtartása érdekében célszerű minél kevésbé degenerált kezdőbázisról indulni.

Kiterjeszthetőség

Kiterjeszthetőség alatt a kezdőbázis kereső algoritmus azon tulajdonságát értjük, hogy a futás során mennyire képes már meglévő adatokkal (parciális kezdőbázis) dolgozni, illetve befejeztével - eredményeire építve - milyen lehetőség van további algoritmusok futtatására.

Megengedettség

Felidézve a szimplex algoritmus működését (3.2.2. rész), a kezdőbázis keresés után az első fázis feladata megengedett, azaz az individuális korlátokat nem sértő bázis előállítása.

Amennyiben a kezdőbázis keresés a megengedetthez minél közelebbi bázist eredményez, úgy jelentősen csökkenthető vagy a legjobb esetben, akár el is hagyható az első fázis. Ez azért tekinthető fontos szempontnak, mert algoritmikus szempontból az első fázis tartalmazza a leginkább számításigényes részeket.

Későbbi futás iterációszáma

Ez a minőségi jellemző arra utal, hogy a többi részmodul állandó értékű paraméterezése mellett, milyen iterációszámra van szükség az optimális megoldás megtalálásáig. Általános célnak tekinthető minden kezdőbázis kereső módszer számára, hogy ezt az iterációszámot a lehető lejjebb vigye.

Kutatásaink során ez az a pont, amikor fel kell ismerni, hogy a kezdőbázis keresés vizsgálata azért is nehéz feladat, mert ugyan a szimplex első kötelező lépése, a kiértékelés csak akkor lehet sikeres, ha már minden egyéb, az algoritmusban később szereplő részrendszer helyesen működik, és képes a legkülönbözőbb tulajdonságokkal rendelkező feladatok megoldására is.

Távolság az optimális megoldástól

A szimplex algoritmus végeztével célszerű megvizsgálni, hogy a kezdőbázis által beválasztott változók közül hány került végül az optimális megoldás bázisába. Érdekes adat és a heurisztikus eljárások javítását célozhatja, bár egyes kutatók szerint az erre való optimalizálás feleslegesen sok energiát fogyaszt, érdemi eredmények nélkül.

5.3.4. A kezdőbázis kereső modul felépítése

A kezdőbázis kereső modult az implementálás korai szakaszaiban egy darab header és egy hozzá tartozó forrásfájl jelentette. Mivel a különböző módszerek nem egymás után, szekvenciális jelleggel, hanem többségében egy futásra értelmezett kizárólagosan működik, így egy idő után - az egyre több, adott stratégiához nem tartozó kódok megszületésével - a fájlok és módszerek áttekinthetősége folyamatosan csökkent.

Még a *PanOpt* rendszer tervezésének korai fázisában megfogalmazódott egyik távlati cél a párhuzamosítás megvalósítása volt. A kezdőbázis keresés egy egyszeri és nem túl hosszú feladat, így a párhuzamosítás mikéntjét elsősorban abban láttuk, hogy egyszerre több stratégiát indítunk útjára, és a leggyorsabban végző vagy leginkább kedvező tulajdonságokkal rendelkező megoldást delegáljuk tovább a szimplex algoritmusnak.

Ezeket a megfontolásokat figyelembe véve a kezdőbázis kereső modult az alábbiak szerint bontottam részekre. A keresés menedzselését végző `StartingBasisFinder` osztályt megtartott, ami a *PanOpt* többi komponense számára transzparens átszervezést jelentett. Feladata a beállított stratégiák alapján a kiválasztott algoritmus felparaméterezése, futtatása, majd a kezdeti beállításától függően a részeredményre újabb algoritmus meghívása. Amennyiben több algoritmust futtat, úgy a kötött változók és bázispozíciók frissítése is a feladatai közé tartozik. Kérésre a `printLevel` megadásával információkat ír ki az elkészített bázisról.

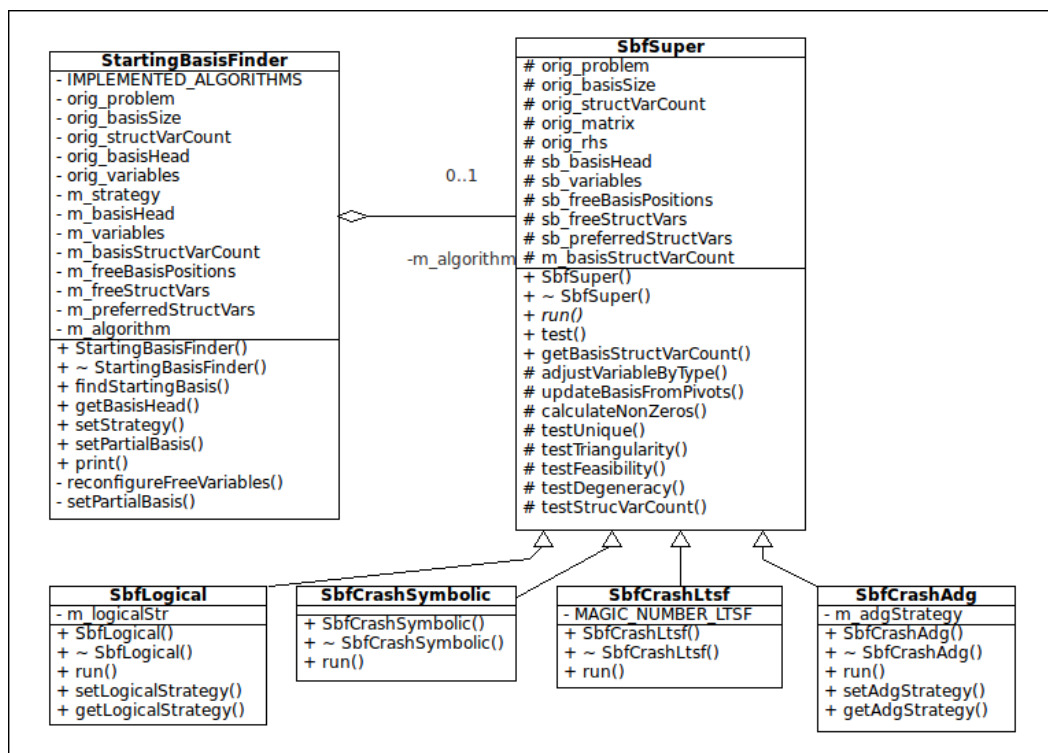
Ahhoz, hogy az egyes algoritmusokat közösen lehessen kezelni és már meglévő kódrészleteket (lásd: 5.3.2. rész) lehessen használni, létrehoztam egy közös ősosztályt, `SbfSuper` néven. Ezzel biztosítva van, hogy a kódok között könnyű átjárhatóság érdekében azonos módon hivatkozzanak a feladat eredeti adataira, illetve eljárásuk közben az adatok tárolása is egységesen történjen. Az imént hivatkozott segéd eljárások kódjainak nagy része szintén itt kapott helyet. A harmadik fontos jellemzője a `test` metódus, amely számos kisebb eljárást tartalmaz, és az alábbiak szerint paraméterezhető.

```
enum TestType { TEST_UNIQUE = 1, TEST_TRINAGULARITY = 2,
    TEST_DEGENERACY = 4, TEST_FEASIBILITY = 8,
    TEST_STRUCTVARCOUNT = 16 };
```

Az egyes algoritmusok implementációja külön osztályokba került: `SbfLogical`, `SbfCrashSymbolic`, `SbfCrashLtsf` és `SbfAdg`. Az osztályok

5.3. A kezdőbázis keresés implementálása

egymáshoz való viszonyát - a szignatúrák feltüntetése nélkül -, az 5.3. UML ábrán láthatjuk.



5.3. ábra. A kezdőbázis kereső modul felépítése

5.3.5. A logikai bázis speciális tulajdonságai

A logikai bázis implementációjánál az általános algoritmushoz képest egy kiegészítésre volt szükség: a logikai bázis finomhangolása, azaz hogy alsó-, felső- vagy kevert logikai bázist készítsen-e.

```
enum LogicalStrategy { LOWER_LOGICAL, UPPER_LOGICAL,
    MIXED_LOGICAL };
```

Első lépésben gondoskodni kell arról, hogy a bázison kívüli változók, amelyek jelen esetben a strukturális változók, értéket kapjanak. Ehhez az 5.3.2. részben ismertett `adjustVariableByType` függvényt használtam. A preferált típus alsó logikai bázis esetén az alsókorlát, felső logikai bázis esetén a felső korlát, kevert logikai bázis esetén pedig a kedvezőbb célfüggvényértéket biztosító korlát.

A bázisba a logikai változók természetes sorrendjükben kerülnek be.

5.3.6. A crash bázisok speciális tulajdonságai

A crash bázisokra jellemző a sor- és oszlopszámlálók intenzív használata. Ezek inicializálására az 5.3.2. részben ismertetett `calculateNonZeros` eljárást használtam, amely képes figyelembe venni a már fixált bázispozíciókat és fixált bázisváltozókat.

A sor- és oszlopszámlálók karbantartása az implementációnak kulcsfontosságú kérdése. Amennyiben a pszeudo kód ajánlásait követjük, úgy a sor- és oszlopkiválasztások után a kizárások azonosításával újra kellene definiálni az A_s mátrix aktív részét, és azok újraszámolni a sor- és oszlopszámlálókat. Ezek nagyságrendileg sok számítást jelent.

Ehelyett hatékonyan használható az a megközelítés, hogy a számlálókat listába szervezzük, majd azok folyamatos frissítését vállaljuk. A kizárt oszlopokat és sorokat legegyszerűbb úgy jelölhetjük, ha a nekik megfelelő számlálók értékét 0-ra állítjuk, hisz így bármely crash algoritmust választva, azok biztosan nem kerülnek pivot pozícióba.

Már korábban említettük a redundáns adattárolást, amely a memóriagazdálkodással nagyvonalúan bántva a gyorsabb adatelérésekkel rövidebb futási időt eredményez. Így a crash algoritmusok megvalósításánál a számlálóvektorok használatával egyidőben gyakran egyéb, fordított adattárolást is használtam.

Szimbolikus és LTSF crash

Erre kiváló példa, hogy a szimbolikus és LTSF crash során a sorszámlálókat nemcsak sor $\rightarrow$ számláló irányban (számlálók értékeit tartalmazó vektor), hanem számláló $\rightarrow$ sor irányban (map, melynek kulcsa a számláló, értéke pedig az akkora sorszámlálóval rendelkező sorok indexeinek halmaza) is tároljuk.

A tesztelés során ez a fajta gyorsítás sokszor javulást eredményezett.

A számlálók karbantartásának főbb lépései:

1. **Inicializálás** Számlálók inicializálása a `calculateNonZeros` függvény segítségével, amely képes mind a vektoros ábrázolás, mind a map-es adattárolás feltöltésére
2. **Sorválasztás** Az iterációk elején a minimális sorszámlálók keresése a map legkisebb kulcsához tartozó (azaz a map legelején lévő) halmazból egy sor-index kiválasztása. Ebben a lépésben ügyelni kell arra, hogy az megüresedett

5.4. Tesztelés

halmazok kikerüljenek a map-ből. Ezzel együtt, ha egy kulcshoz nem tartozik halmaz, akkor azt is el kell távolítani a map-ből.

3. **Frissítés** Amikor a pivot elem kiválasztásra kerül, akkor nemcsak az adott sor- és oszlopszámlálót kell nullára állítani, hanem minden olyan oszlop számlálóját is, amelyik a pivot sorral való metszetében nemnulla elemet tartalmaz. Ezen kívül a pivot oszlop eltávolításával a sorszámlálók vektorát és a map-et is frissíteni kell azokra a sorokra nézve, amelyek nemnulla elemmel rendelkeztek az adott oszlopindex által kijelölt pozícióban.

ADG crash

Az ADG crash implementációja jellegéből adódóan eltér az előbb említett crash implementációktól. Fontos pontja, hogy itt már némi numerikus számításra is szükség van. Ezek elvégzésére a *PanOpt* rendszer *Numerical* típusát és annak metódusait kell használni, mivel az a rendszerben alapértelmezett adatábrázolásokkal szemben (pl. *double*) a nagy méretű, ritkás LP feladatok jellemzőire fel van készítve, különböző toleranciák használatával.

Az ADG megvalósítása során a pszeudo kódtól érdemes eltérni a tekintetben, hogy a H_1 halmaz tárolása és az abban való keresés programozási szempontból nagyobb költséget generál, mintha a halmazba való tartozás feltételét alkalmanként, helyileg vizsgálnánk.

5.4. Tesztelés

Mivel a modul elsődleges feladata, hogy versenyképes implementációt nyújtson a *PanOpt* rendszer számára, így a megvalósított kezdőbázis kereső algoritmusokat legfőképpen futási idő szempontjából vizsgáltam.

A korszerű C++ implementáció, a ritkás adatszerkezetek és a redundáns adattárolás segítségével sikerült olyan eljárásokat előállítani, amelyek kellően gyors futási idővel sikeresen megoldják a kezdőbázis keresés feladatát.

A tesztelés során a már hivatkozott NETLIB adatbázisban szereplő feladatokat használtam. Közülük pár fontosabb tesztelési adat az alábbiakban látható.

5. A PanOpt keretrendszer

Name	Time	SumT	It	Time	SumT	It	Time	SumT	It
AFIRO.MPS	0	0	18	0	0	18	0	0.01	18
SC50A.MPS	0	0.01	48	0	0.01	48	0	0	48
ADLITTLE.MPS	0	0.01	110	0	0.01	110	0	0.02	110
AGG.MPS	0	0.23	132	0	0.16	132	0	0.14	132
BEACONFD.MPS	0	0.05	111	0	0.05	111	0	0.04	111
AGG3.MPS	0	0.34	163	0	0.35	163	0	0.33	163
AGG2.MPS	0	0.35	157	0.01	0.34	157	0	0.34	157
BLEND.MPS	0	0.01	95	0	0.01	95	0	0.01	95
BOEING2.MPS	0	0.07	192	0	0.1	287	0	0.06	199
BANDM.MPS	0	0.97	856	0	0.85	856	0	0.79	856
BORE3D.MPS	0	0.14	240	0	0.1	185	0	0.11	185
BRANDY.MPS	0	0.31	441	0	0.24	441	0	0.32	441
BOEING1.MPS	0	0.97	1006	0	1.26	1261	0	0.95	949
FIT1D.MPS	0	0.38	1362	0	0.3	1273	0	0.3	1336
ETAMACRO.MPS	0	0.92	789	0	0.88	777	0	0.93	789
FORPLAN.MPS	0	0.25	404	0	0.18	311	0	0.27	404
FFFFFF800.MPS	0	2.78	1291	0	2.77	1291	0	2.43	1291
GROW15.MPS	0	2.27	840	0	2.53	936	0	2.24	867
GROW7.MPS	0	0.23	310	0	0.25	385	0	0.22	324
ISRAEL.MPS	0	0.19	359	0	0.18	359	0	0.19	359
KB2.MPS	0	0.01	111	0	0.02	103	0	0.02	105
LOTFI.MPS	0	0.07	243	0	0.07	243	0	0.07	243
RECIPE.MPS	0	0.02	50	0	0.01	58	0	0.01	44
SC105.MPS	0	0.04	105	0	0.04	105	0	0.03	105
SC205.MPS	0	0.19	246	0	0.16	246	0	0.18	246
SC50B.MPS	0	0.01	50	0	0.01	50	0	0.01	50
SCAGR25.MPS	0	1.44	740	0	1.46	740	0	1.42	740
SCAGR7.MPS	0	0.04	157	0	0.05	157	0	0.04	157
SCFXM1.MPS	0	0.55	532	0	0.56	532	0	0.54	532
GROW22.MPS	0	5.84	1180	0	6.98	1445	0	5.63	1316

5.1. táblázat. A logikai bázis teszteredményei 1.

5.4. Tesztelés

Name	Time	SumT	It	Time	SumT	It	Time	SumT	It
SCFXM2.MPS	0	3.59	1062	0	3.45	1062	0	3.55	1062
SCORPION.MPS	0	0.46	372	0	0.45	372	0	0.46	372
GANGES.MPS	0	14.33	1628	0	12.22	1345	0	15.06	1587
SCSD1.MPS	0.01	0.08	206	0	0.07	206	0	0.07	206
SCSD6.MPS	0	0.15	372	0	0.15	372	0	0.15	372
SCRS8.MPS	0	1.62	928	0	1.57	928	0	1.62	928
SCTAP1.MPS	0	0.28	391	0	0.28	391	0	0.28	391
BNL1.MPS	0	15.07	4739	0	16.36	4739	0	14.77	4739
SCFXM3.MPS	0	11.01	1599	0	10.45	1599	0	9.67	1599
SEBA.MPS	0	1.1	641	0	4.1	2234	0	1.11	641
SHARE1B.MPS	0	0.13	351	0	0.13	351	0	0.12	351
SHARE2B.MPS	0	0.03	133	0	0.03	133	0	0.03	133
SCTAP2.MPS	0	9.32	1415	0	9.39	1415	0	9.46	1415
SHIP04L.MPS	0	0.83	740	0	0.82	740	0	0.82	740
SHIP04S.MPS	0	0.56	517	0	0.57	517	0	0.57	517
SHELL.MPS	0	1.29	743	0	1.66	954	0	1.29	743
SCSD8.MPS	0	1.53	1073	0	1.79	1073	0	1.57	1073
SHIP08S.MPS	0	2.34	663	0	2.34	663	0	2.35	663
SHIP08L.MPS	0.01	4.02	1162	0	4.01	1162	0	3.99	1162
SCTAP3.MPS	0	21.36	1818	0	21.32	1818	0	19.87	1818
SHIP12S.MPS	0	8.61	1251	0	8.65	1251	0	8.75	1251
STANDATA.MPS	0	0.12	99	0	0.34	361	0	0.12	99
STANDGUB.MPS	0	0.13	99	0	0.34	361	0	0.12	99
STANDMPS.MPS	0	0.67	476	0	0.71	506	0	0.67	476
STAIR.MPS	0	1.67	624	0	2.1	667	0	1.87	624
STOCFOR1.MPS	0	0.01	81	0	0.01	81	0	0.01	81
VTP _B AS E.MPS	0	0.18	373	0	0.13	285	0	0.17	373
SHIP12L.MPS	0	14.07	2007	0	13.68	2007	0	14.06	2007
TUFF.MPS	0	3.86	3924	0	7.25	7450	0	3.86	3924
SIERRA.MPS	0	5.47	702	0	30.87	3593	0	5.51	702
WOODW.MPS	0	11.37	1968	0	9.63	1968	0	8.91	1968

5.2. táblázat. A logikai bázis teszteredményei 2.

5. A PanOpt keretrendszer

Name	Struct	Time	SumT	It	Struct	Time	SumT	It	Struct	Time	SumT	It
AFIRO.MPS	0	0	0.01	7	13	0.01	0.01	10	7	0	0	15
SC50A.MPS	0	0.02	0.02	13	35	0.02	0.03	24	9	0	0.01	43
ADLITTLE.MPS	0	0.01	0.02	67	19	0.02	0.03	125	32	0	0.02	120
AGG.MPS	0	0.05	0.29	192	59	0.06	0.28	139	75	0.01	0.32	180
BEACONFD.MPS	0	0.05	0.09	89	103	0.06	0.1	89	67	0	0.04	84
AGG3.MPS	0	0.11	0.54	222	87	0.07	0.37	183	109	0.01	0.36	179
AGG2.MPS	0	0.11	0.52	211	89	0.1	0.5	190	116	0.02	0.36	176
BLEND.MPS	0	0.01	0.03	86	27	0.01	0.03	78	8	0	0.01	127
BOEING2.MPS	0	0.05	0.13	223	4	0.05	0.15	273	36	0	0.07	191
BANDM.MPS	0	0.05	0.33	326	193	0.05	0.65	573	116	0	0.87	684
BORE3D.MPS	0	0.06	0.16	149	139	0.08	0.13	90	0	0	0.1	185
BRANDY.MPS	0	0.06	0.23	217	48	0.05	0.33	370	0	0	0.28	365
BOEING1.MPS	0	0.08	1.37	964	8	0.09	1.26	1177	115	0.01	1.04	918
FIT1D.MPS	0	0.09	0.62	1954	1	0.08	0.47	1675	0	0.01	0.3	1336
ETAMACRO.MPS	0	0.12	1.26	864	210	0.12	2	1692	24	0.01	1.24	1055
FORPLAN.MPS	0	0.06	0.27	319	76	0.06	0.29	348	7	0	0.24	380
FFFFF800.MPS	0	0.16	2.09	907	107	0.16	1.82	795	178	0.02	3.08	1396
GROW15.MPS	0	0.1	2.25	794	300	0.1	2.27	794	0	0	2.22	867
GROW7.MPS	0	0.05	0.24	241	140	0.04	0.24	241	0	0	0.22	324
ISRAEL.MPS	0	0.03	0.27	410	13	0.05	0.3	497	92	0	0.19	336
KB2.MPS	0	0.02	0.03	95	18	0	0.01	69	0	0	0.01	105
LOTFI.MPS	0	0.05	0.11	167	53	0.06	0.12	239	49	0	0.08	271
RECIPE.MPS	0	0.04	0.04	30	62	0.02	0.03	19	0	0	0.02	44
SC105.MPS	0	0.03	0.05	28	75	0.03	0.05	47	19	0	0.03	98
SC205.MPS	0	0.06	0.12	33	148	0.05	0.16	105	37	0	0.13	191
SC50B.MPS	0	0	0.01	3	35	0.01	0.03	21	5	0	0.01	51
SCAGR25.MPS	0	0.11	1.13	525	202	0.12	0.77	353	176	0.01	1.38	712
SCAGR7.MPS	0	0.02	0.06	108	58	0.04	0.06	121	50	0	0.04	154
SCFXM1.MPS	0	0.1	0.46	318	120	0.1	0.58	434	111	0.01	0.55	499
GROW22.MPS	0	0.15	6.54	1281	440	0.14	6.82	1281	0	0.01	5.67	1316

5.3. táblázat. A crash bázis teszteredményei 1.

5.4. Tesztelés

Name	Struct	Time	SumT	It	Struct	Time	SumT	It	Struct	Time	SumT	It
SCFXM2.MPS	0	0.19	2.45	636	240	0.19	2.88	778	223	0.02	3.64	1046
SCORPION.MPS	0	0.09	0.36	161	169	0.1	0.43	240	76	0	0.41	331
GANGES.MPS	0	0.21	8.59	807	663	0.37	11.69	1150	471	0.06	10.93	1169
SCSD1.MPS	0	0.08	0.14	136	74	0.07	0.12	144	1	0.01	0.08	222
SCSD6.MPS	0	0.13	0.34	394	139	0.14	0.37	462	9	0	0.27	574
SCRS8.MPS	0	0.17	1.41	681	380	0.18	1.76	780	77	0.01	1.83	1020
SCTAP1.MPS	0	0.1	0.3	271	179	0.1	0.24	156	154	0.01	0.25	276
BNL1.MPS	0	0.2	16.24	4278	191	0.21	17.38	4761	185	0.02	15.9	4399
SCFXM3.MPS	0	0.26	6.96	938	360	0.29	8.18	1133	335	0.04	11.01	1548
SEBA.MPS	0	0.17	1.48	729	507	0.18	2.8	1490	8	0.01	1.28	710
SHARE1B.MPS	0	0.04	0.18	319	46	0.05	0.17	324	64	0	0.15	380
SHARE2B.MPS	0	0.02	0.04	104	25	0.03	0.05	104	24	0	0.04	129
SCTAP2.MPS	0	0.33	4.22	612	616	0.33	3.62	510	521	0.07	6.79	1020
SHIP04L.MPS	0	0.23	0.75	449	66	0.23	0.8	493	261	0.02	0.37	281
SHIP04S.MPS	0	0.18	0.51	294	66	0.19	0.64	416	240	0.02	0.39	304
SHELL.MPS	0	0.23	1.09	528	403	0.16	1.17	569	2	0	1.36	783
SCSD8.MPS	0	0.28	2.38	846	393	0.29	2.31	814	15	0	2.69	1271
SHIP08S.MPS	0	0.32	1.6	424	130	0.32	2.84	749	368	0.05	1.5	407
SHIP08L.MPS	0	0.47	3.24	825	130	0.46	4.03	1004	407	0.09	2.03	533
SCTAP3.MPS	0	0.36	10.59	880	848	0.47	7.94	633	682	0.12	14.9	1242
SHIP12S.MPS	0	0.4	4.19	487	194	0.41	8.03	1059	586	0.08	4.75	629
STANDATA.MPS	0	0.15	0.58	407	135	0.15	0.44	298	5	0	0.23	219
STANDGUB.MPS	0	0.16	0.58	407	136	0.16	0.45	298	5	0	0.23	219
STANDMPS.MPS	0	0.16	1.54	881	135	0.16	0.86	481	113	0.01	0.84	556
STAIR.MPS	0	0.09	1.74	416	135	0.09	1.77	476	59	0.01	2.39	699
STOCFOR1.MPS	0	0.02	0.03	84	32	0.02	0.03	60	8	0	0.03	73
VTP _{BASE} .MPS	0	0.04	0.4	589	56	0.05	0.27	433	30	0	0.3	588
SHIP12L.MPS	0	0.62	8.63	1071	194	0.63	10.36	1399	641	0.16	7.01	988
TUFF.MPS	0	0.1	1.76	1537	149	0.1	1.09	905	0	0	3.32	3924
SIERRA.MPS	0	0.36	19.08	2365	455	0.38	9.64	1148	651	0.07	12.21	2179
WOODW.MPS	0	0.6	9.14	1737	368	0.53	8.79	1917	34	0.02	9.48	2124

5.4. táblázat. A crash bázis teszteredményei 2.

6. fejezet

Összefoglalás

Kutatásaim során az LP feladatok szimplex módszertan alapú megoldásával foglalkoztam. Társaimmal együtt megterveztük és implementáltuk a Pannon Egyetem saját megoldó szoftverét, a *PanOpt* keretrendszert. Egyéni feladatomban a kezdőbázis kereső modul implementálása volt.

A szakterület korábbi tapasztalataiból kiindulva ismertettem a legelterjedtebb módszereket, és a *PanOpt* rendszerhez igazítva, azokat algoritmusait módosítottam, hogy nem támaszkodjon a speciális computational formhoz, hanem az eredeti individuális korlátokkal dolgozzon.

A feladatom másik része ezek után a megfelelő megvalósítás volt, melynek során az implementációs sajátosságokat figyelembe véve újabb módosításokat tettem.

Eredményül elkészült egy olyan moduláris kezdőbázis kereső környezet, amely az eddig piaci termékekkel ellentétben már alkalmas további ötletek könnyű integrálására. Elkészítettem továbbá négy különböző algoritmust, amelyek tovább paraméterezhetőek. Mivel futási eredményük felveszi a versenyt az elérhető szoftverek hasonló funkcionalitású részegységeivel, így kijelenthetjük, hogy a *PanOpt* keretrendszer számára megfelelő modul született.

Irodalomjegyzék

- [1] GNU Project. GLPK (GNU Linear Programming Kit), 2012. 04. 14.
<http://www.gnu.org/software/glpk/> .
- [2] GNU Project. GLPK/GMPL (MathProg), 2012. 04. 14.
http://en.wikibooks.org/wiki/GLPK/GMPL_%28MathProg%29 .
- [3] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.
- [4] D. J. Rose, R. A. Willoughby, editor. *Sparse Matrices and their Applications, Proceedings*. Plenum Press, 1972.
- [5] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.
- [6] I. Maros, G. Mitra. *Simplex Algorithms, in Advances in Linear and Integer Programming*. Oxford University Press, 1996.
- [7] I. Maros. *Computational Techniques of the Simplex Method*. Springer, 2002.
- [8] I. Maros. *Operation Research 1 (MSc in Information Engineering)*. Pannon Egyetem, 2009.
- [9] N. Gülpinar, G. Mitra, I. Maros. Creating Advanced Bases For Large Scale Linear Programs Exploiting Embedded Network Structure. *Computational Optimization and Applications*, 2002.
- [10] IBM. IBM ILOG CPLEX Optimizer, 2012. 04. 14.
<http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/> .
- [11] R. E. Bixby. Implementing the Simplex Method: The Initial Basis. *ORSA Journal on Computing*, 4:3:267:284., 1992.

IRODALOMJEGYZÉK

- [12] N. I. M. Gould, J. K. Reid. New Crash Procedures for Large Systems of Linear Constraints. *Mathematical Programming*, 45:475–501, 1989.
- [13] Netlib. Netlib collection of mathematical software, papers, and databases. , 2012. 04. 14.
<http://www.netlib.org/lp/data/> .
- [14] B. Stroustrup. *A C++ programozási nyelv*. Kiskapu Könyvkiadó, Budapest, Magyarország, 2001.