

TDK dolgozat

Smidla József



**Pannon
Egyetem**



Gyorsított teljes oszlop kiválasztás tervezése és implementációja a szimplex módszerhez

Készítette: Smidla József

II. évfolyam

Témavezető: Dr. Maros István

egyetemi tanár

Műszaki Informatikai Kar, Rendszer- és
Számítástudományi Tanszék

Tartalomjegyzék

1. Irodalmi áttekintés	5
1.1. Lineáris programozási feladatok	5
1.2. Szimplex módszer	6
1.3. Szorzat alapú bázis reprezentáció	10
1.4. Oszlop kiválasztása a szimplex módszerben	12
1.4.1. Megengedettségi feltételek	13
1.4.2. Optimalitási feltételek	13
1.4.3. Oszlop választó stratégiák	14
1.4.4. Redukált költségek számolása	16
2. Gyorsított teljes oszlop kiválasztás	18
2.1. Primál első fázisú szimplex szorzó frissítése	18
2.2. Sorfolytonos bázis inverz reprezentáció	20
2.3. Gyorsított teljes oszlop kiválasztás	30
2.4. Mérési eredmények	31
2.4.1. A két BTRAN implementáció összehasonlítása	32
2.4.2. A sorfolytonos BTRAN és az első fázisú szimplex szorzó frissítése	32
2.4.3. Redukált költségek kiszámítási módjai	32
2.4.4. Újrainvertálások gyakorisága	41
3. Összefoglalás	44
Irodalomjegyzék	45

Ábrák jegyzéke

1.1. A szimplex módszer folyamat ábrája	10
2.1. Algoritmus a $\phi^T(\mathbf{B} - \bar{\mathbf{B}})$ kiszámolására	21
2.2. Oszlopfolytonos η reprezentációt alkalmazó BTRAN algoritmus . .	22
2.3. Példa az oszlopfolytonos reprezentációt alkalmazó BTRAN-ra . . .	23
2.4. A (2.3) példában ábrázolt η vektorok sorfolytonosan tárolva	25
2.5. Példa az L_η és L_a tömbök kapcsolatára	25
2.6. Az $L_a[\maxIndex]$ elemeinek új listákba fűzése	26
2.7. Új elemmel bővült az $\mathbf{a}$ vektor, ezért L_a -t is ki kell egészíteni	26
2.8. Az $\mathbf{a}$ vektor 2. elemét átírjuk, viszont most L_a -t nem kell kiegészíteni	27
2.9. A sorfolytonos BTRAN algoritmushoz szükséges összes adatszer- kezet	27
2.10. Sorfolytonos bázis inverz reprezentációt alkalmazó BTRAN algo- ritmus	28
2.11. Példa a halom indexelésére	31

Táblázatok jegyzéke

1.1. Változó típusok	7
1.2. Oszlop kiválasztási feltételek az első fázisban	13
1.3. Oszlop kiválasztási feltételek a második fázisban	14
2.1. A sorfolytonos BTRAN algoritmus futási ideje az első 39 feladatra .	33
2.2. A sorfolytonos BTRAN algoritmus futási ideje az utolsó 30 feladatra	34
2.3. A sorfolytonos BTRAN algoritmus által végrehajtott szorzások száma az első 39 feladatra	35
2.4. A sorfolytonos BTRAN algoritmus által végrehajtott szorzások száma a második 30 feladatra	36
2.5. Csak az első fázisú szimplex szorzó frissítésének, majd a sorfolytonos BTRAN algoritmus alkalmazásának hatása az oszlop kiválasztó futási idejére, az első 39 feladat esetén	37
2.6. Csak az első fázisú szimplex szorzó frissítésének, majd a sorfolytonos BTRAN algoritmus alkalmazásának hatása az oszlop kiválasztó futási idejére, az utolsó 30 feladat esetén	38
2.7. A halom alkalmazásának hatása az első 39 feladat oszlopkiválasztási idejére	39
2.8. A halom alkalmazásának hatása az utolsó 30 feladat oszlopkiválasztási idejére	40
2.9. Újrainvertálás 500 iterációnként oszlopfolytonos, illetve sorfolytonos BTRAN algoritmussal	42
2.10. Újrainvertálás 500 iterációnként oszlopfolytonos, illetve sorfolytonos BTRAN algoritmussal	43

Bevezetés

Az operációkutatás napjainkra az alkalmazott matematika egy fontos részterületévé nőtt ki magát. Nélkülözhetetlen segítséget nyújt számos területen, mint például az ipar, pénzügyi szektor, hadászat. A lineáris programozás (újabban lineáris optimalizálás) az operációkutatáson belül központi szerepet játszik: Nem csak egy önállóan alkalmazható tudományág, hanem az operációkutatás több megoldó módszerében felhasználható, a nemlineáris programozástól a játékelméletig. Éppen ezért a lineáris programozási feladatok megoldó módszereinek kutatása rendkívül fontos. Jelenleg két fő megoldó módszer létezik: A belső pontos algoritmus és a szimplex módszer. Jelen dolgozatban ez utóbbival foglalkozunk.

A szimplex módszernek számos implementációja létezik, melyek minőségben és hozzáférhetőségben jelentősen eltérnek egymástól. A célunk egy helyesen működő, hatékony megoldó szoftver fejlesztése, mely több módon is felhasználható lesz: Önálló programként és függvénykönyvtárként is. Fontosnak tartjuk, hogy az implementáció jól dokumentált, áttekinthető és egyszerűen módosítható, fejleszthető legyen, ezért a modularitásra is különös hangsúlyt fektetünk.

Az algoritmus hatékonyságát befolyásolják a használt adatszerkezetek, és az egyes rész algoritmusok hatékonysága, pontossága. Jelen dolgozatban a szimplex módszer áttekintése után az oszlop kiválasztó modulra koncentrálnunk. Ennek a modulnak a működése nagy mértékben kihat a fő algoritmus iterációinak számára, így az oszlop kiválasztás a hatékonyság szempontjából ígéretes kutatási terület.

Az első fejezetben ismertetjük a lineáris programozási feladatokat, majd ezt követően felvázoljuk a szimplex módszer működésének elvét, és röviden bemutatjuk az egyes rész algoritmusok lényegét. Külön foglalkozunk a szorzat alapú bázis inverz reprezentációval. A fejezet végén részletesebben ismertetjük az oszlop kiválasztáshoz szükséges elméleti alapokat, és az egyes stratégiákat. A második fejezetben ismertetünk három gyorsítási technikát a teljes oszlopkiválasztáshoz: Korábban már ismert volt, hogy a primál második fázisában hogyan lehet gyorsan kiszámolni a megváltozott szimplex szorzót, a fejezet elején bemutatjuk, hogy ezt a feladatot hogy oldhatjuk meg első fázisban. A következő szakaszban ismertetünk egy hatékony BTRAN algoritmust, mely az első fejezetben bemutatott bázis inverz ábrázolási technika egyik műveletét valósítja meg, ez az algoritmus erősen kötődik az oszlopkiválasztáshoz. A harmadik szakaszban bemutatjuk, hogy a teljes oszlopkiválasztás milyen módon gyorsítható fel. Végül ismertetjük a mérési eredményeket, ahol azt vizsgáltuk, hogy a három bemutatott technika segítségével mely feladatoknál mekkora hatékonyság növekedést értünk el.

1. fejezet

Irodalmi áttekintés

Az alábbi fejezetben ismertetjük a lineáris programozási feladatokat, valamint a szimplex módszer működését [7].

1.1. Lineáris programozási feladatok

Egy lineáris programozási feladatban az optimalizálandó problémát lineáris kifejezésekkel, lineáris egyenlőtlenségekkel kell megadni, mivel a megoldó módszerek ezeket használják fel a megoldás során. A feladat megoldását numerikus értékek reprezentálják, ehhez szükséges egy döntési változókat tartalmazó n dimenziós vektor, ezt jelöljük $\mathbf{x}$ -el. Minden változóra meg kell adni, hogy azok milyen intervallumban vehetnek fel értékeket. A változók alsó korlátait az n dimenziós $\mathbf{l}$, a felső korlátokat pedig az $\mathbf{u}$ vektor tartalmazza. Az $\mathbf{l}$ tartalmazhatja a $-\infty$ -t, az $\mathbf{u}$ pedig a $+\infty$ -t is, ezzel lehetővé tesszük, hogy egy változónak ne legyen alsó, illetve felső korlátja.

$$l_j \leq x_j \leq u_j, j = 1 \dots n \quad (1.1)$$

A feladat tartalmaz egy célfüggvényt, amely a döntési változók lineáris kombinációja. Ezt az értéket kell minimalizálni, vagy maximalizálni. A dolgozat hátralévő részében az egyszerűbb tárgyalhatóság kedvéért minimalizálási feladatokról lesz szó. A célfüggvényben szereplő változók együtthatóit a $\mathbf{c}$ n dimenziós vektor tartalmazza, a célfüggvény ez alapján a következő:

$$\min z = \sum_{i=1}^n c_i x_i \quad (1.2)$$

A feladatok tartalmaznak még korlátozásokat is, ezeket az $\mathbf{x}$ vektorból képzett lineáris egyenlőtlenségekkel adjuk meg: Az $\mathbf{x}$ vektor komponenseinek lineáris kombinációjára felírhatunk egy alsó, illetve egy felső korlátot. Jelölje m ezen korlátozások számát, így a korlátozásokat a következő módon írhatjuk fel:

$$\begin{aligned} L_1 &\leq x_1 a_1^1 + \dots + x_n a_n^1 \leq U_1 \\ &\vdots \end{aligned}$$

$$\begin{aligned}
L_i &\leq x_1 a_1^i + \dots + x_n a_n^i \leq U_i \\
&\vdots \\
L_m &\leq x_1 a_1^m + \dots + x_n a_n^m \leq U_m
\end{aligned}$$

ahol az $\mathbf{L}$ és $\mathbf{U}$ m dimenziós vektorok tartalmazzák az egyes korlátozások alsó és felső korlátait, az a_j^i pedig az i korlátozáson belül a j változó együtthatója, ahol $1 \leq i \leq m$ és $1 \leq j \leq n$. Ezen együtthatókat az $\mathbf{A}^{m \times n}$ együttható mátrix tárolja. A változók, a célfüggvény, és a korlátozások együtt alkotják a lineáris programozási modellt:

$$\min z = \mathbf{c}^T \mathbf{x} \quad (1.3)$$

$$s.t. \quad \mathbf{L} \leq \mathbf{Ax} \leq \mathbf{U} \quad (1.4)$$

$$\mathbf{l} \leq \mathbf{x} \leq \mathbf{u} \quad (1.5)$$

A feladat megoldásának nevezzük azt az $\mathbf{x}$ vektort, amely eleget tesz a (1.4) egyenlőtlenség rendszernek. Megengedett megoldásnak nevezzük azt az $\mathbf{x}$ megoldást, amely eleget tesz a (1.5)-nek. A megengedett megoldások halmazát $\mathcal{X}$ -el jelöljük:

$$\mathcal{X} = \{\mathbf{x} : \mathbf{L} \leq \mathbf{Ax} \leq \mathbf{U}, \mathbf{l} \leq \mathbf{x} \leq \mathbf{u}\} \quad (1.6)$$

Amennyiben $\mathcal{X} = \emptyset$, a feladatnak nincsen megoldása.

Optimálisnak mondjuk azt az $\mathbf{x} \in \mathcal{X}$ megoldást, amelyre igaz, hogy bármely $\mathbf{x}' \in \mathcal{X}$ esetén: $\mathbf{c}^T \mathbf{x} \leq \mathbf{c}^T \mathbf{x}'$.

1.2. Szimplex módszer

Az alábbi szakaszban ismertetjük a szimplex módszer működését. Először áttekintjük, hogy a feladatot milyen formában kell ábrázolni, mi a megoldó algoritmus működésének elve, majd röviden kitérünk az egyes modulokra. A szimplex módszernek matematikai szempontból két fő változata létezik: A primál és a duál módszer. Jelen dolgozatban csupán a primál módszerrel foglalkozunk.

A (1.3) - (1.5)-ben ismertetett általános feladatlírást átalakíthatjuk egyszerűbb, úgynevezett számítási formába úgy, hogy a kapott modell pontosan azt a feladatot írja le, mint az eredeti. Az átalakítás célja kettős: Egyrészt egyszerűbbé teszi az algoritmusok tárgyalását, másrészt az implementált programkód is egyszerűsödik. Itt megjegyezzük, hogy az általunk implementált szoftver az eredeti, általános formát használja.

Számítási formából több fajta létezik, a közös tulajdonságuk, hogy minden korlátozáshoz hozzáadunk egy logikai változót úgy, hogy az egyenlőtlenségeket egyenlőségekké alakíthassuk át. A logikai változókat az m dimenziós $\mathbf{y}$ vektor tartalmazza. Bizonyos esetekben szükséges a változók alsó- és felső korlátainak, valamint a korlátozások alsó- és felső korlátainak transzformálása is. A transzformálás után kapott számítási alak a következő módon néz ki:

$$\min z = \mathbf{c}^T \mathbf{x} \quad (1.7)$$

$$\text{s.t.} \quad \mathbf{Ax} + \mathbf{Iy} = \mathbf{b} \quad (1.8)$$

$$\mathbf{l} \leq \mathbf{x}, \mathbf{y} \leq \mathbf{u} \quad (1.9)$$

A változók korlátai a jelen dolgozatban használt számítási formában a következők lehetnek:

Megengedett tartomány	Típus	Elnevezés
$y_i, x_i = 0$	0	Fix
$0 \leq y_i, x_i \leq u_j$	1	Korlátozott
$0 \leq y_i, x_i \leq +\infty$	2	Nemnegatív
$-\infty \leq y_i, x_i \leq +\infty$	3	Szabad

1.1. táblázat. Változó típusok

A továbbiakban az együtttható mátrix alatt az

$$\mathbf{A} := [\mathbf{A}|\mathbf{I}]$$

mátrixot értjük, valamint az

$$\mathbf{x} := [\mathbf{x}|\mathbf{y}]$$

vektor tartalmazza a változókat.

A szimplex módszerhez az $\mathbf{A}$ mátrixot felosztjuk egy $\mathbf{B}$ bázisra, és egy bázison kívüli oszlopvektorokat tartalmazó $\mathbf{R}$ mátrixra:

$$\mathbf{A} = [\mathbf{B}|\mathbf{R}]$$

Vezessünk be néhány index halmazt. Az összes változó indexeinek halmazát jelölje $\mathcal{N}$. Minden változóhoz tartozik 1-1 oszlopvektor az $\mathbf{A}$ mátrixból. Azt mondjuk, hogy egy változó a bázisban van, ha a hozzá tartozó oszlopvektor benne van $\mathbf{B}$ -ben. Az ilyen változókat bázisváltozóknak nevezzük, a többi pedig nembázisváltozóknak. A bázisváltozók indexeit tartalmazza a $\mathcal{B}$ halmaz, a nembázis változók indexeit pedig az $\mathcal{R}$. Jelölje továbbá $\mathcal{U}$ azon nembázis változók indexeit, amelyek a felső korlátjukon vannak.

Ezek alapján az $\mathbf{x}$ vektor felbontható bázisváltozókra, illetve nembázis változókra:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_{\mathcal{B}} \\ \mathbf{x}_{\mathcal{R}} \end{bmatrix}$$

Hasonlóan, a $\mathbf{c}$ is felbontható:

$$\mathbf{c} = \begin{bmatrix} \mathbf{c}_{\mathcal{B}} \\ \mathbf{c}_{\mathcal{R}} \end{bmatrix}$$

Megmutatható, hogy az optimális megoldásban a nembázis változók a korlátaikon vannak, a bázisváltozók értékeit pedig a nembázis változók és a bázis összetétele határozzák meg:

$$\beta = \mathbf{x}_B = \mathbf{B}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}_R) \quad (1.10)$$

Szükségünk lesz még a következő jelölésre is: A bázis inverzének és az együtt-ható mátrix j oszlopának szorzata:

$$\alpha_j = \mathbf{B}^{-1}\mathbf{a}_j \quad (1.11)$$

Két bázist szomszédosnak nevezünk akkor, ha csak egy oszlopvektorukban különböznek egymástól. Egyik bázisról a másikra egy báziscserével juthatunk át. A szimplex módszer lényege, hogy egy kezdő bázisból kiindulva, báziscserék véges sorozatán keresztül keresi az optimális megoldást. A megoldás optimalitása könnyen ellenőrizhető, továbbá annak globális voltát a szimplex módszer garantálja. Általában az algoritmus indulásakor a kapott megoldások nem megengedettek, ezért ekkor a primál első fázis dolgozik a feladaton: Ennek lényege, hogy megtaláljon egy megengedett megoldást. Ezt követően életbe lép a primál második fázis, amely amellett, hogy fenntartja a megengedettséget, az optimumot keresi.

A megoldó algoritmus minden iterációban megváltoztatja egy nem bázis változó értékét, a legtöbb esetben ezt a változót behozzuk a bázisba. Legyen x_q a kiválasztott nem bázis változó, θ pedig az az érték, amennyivel x_q értéke megváltozik. Ekkor a bázisváltozók értékei az alábbi összefüggés szerint változnak meg:

$$\bar{\beta} = \beta - \theta\alpha_q \quad (1.12)$$

A θ értékét a hányados teszt során határozzuk meg. Számítása során figyelembe kell venni, hogy az általa okozott elmozdulások miatt a megoldás ne váljon nem megengedetté.

Az alábbiakban röviden ismertetjük az implementált megoldó algoritmus főbb lépéseit. Ezen lépések és a köztük lévő sorrend az (1.1) ábrán láthatóak.

MPS beolvasás

A feladatot többnyire valamilyen szabványos szöveges fájlban tároljuk. Ma számos elterjedt formátum létezik, ezek közül az MPS az, amelyet minden LP megoldó rendszernek tudnia kell olvasni. Az MPS formátumot a lyukkártyás számítógépek korában fejlesztették ki, ezért az első modelleket lyukkártyákon tárolták, ez pedig egy meglehetősen merev tárolási módot eredményezett. Később, mikor a technikai fejlődés lehetővé tette, hogy más formátumokat használjanak, már túl sok modell létezett MPS-ben, ezért ez maradt az LP modellek szabványos formátuma. Az MPS beolvasó feladata a beolvasáson kívül a fájl hibáinak felderítése, valamint statisztikák számítása. Ennek a modulnak az eredménye a nyers modell, amely már a memóriában található, készen arra, hogy a következő modulok számításokat végezzenek rajta.

Előfeldolgozó

Az előfeldolgozó modul logikailag nem szükséges ahhoz, hogy a megoldó algoritmus lefusson. Feladata a beolvasótól kapott modell egyszerűsítése, valamint javítása. Mivel sok modell úgy keletkezik, hogy azt valamilyen algoritmus generálja,

előfordulhat, hogy a modell redundáns, vagy akár üres korlátozásokat tartalmaz. Az algoritmusnak fel kell ismernie a mátrix eltávolítható oszlopait, illetve sorait úgy, hogy a modell által leírt feladat ne változzon meg. A megoldó algoritmus végén a kapott megoldást újra el kell látni az eltávolított vektorokkal, hiszen a felhasználó számára úgy kell reprezentálni a megoldást, hogy ne legyen zavaró az előfeldolgozó munkája.

Ennek a modulnak van egy külön, szintén opcionális része, a skálázó. Ennek feladata, hogy a numerikusan nehezen kezelhető feladatokat egyszerűbbé tegye.

Kezdőbázis keresés

Ennek a modulnak az a feladata, hogy meghatározzon egy kezdő bázist, amelyről a megoldó algoritmus elindul. Eerre számos lehetőség van, a legegyszerűbb kezdő bázis az egységmátrix, azaz csak logikai változókat választunk be a bázisba, ezt logikai bázisnak nevezzük. Az ideális kezdő bázis kereső megtalálja az optimális bázist, ám jelenleg ilyen módszer nem ismert. Az egyes kezdőbázis kereső algoritmusokat elsősorban azzal a céllal tervezik, hogy az általuk adott bázis közelebb legyen az optimumhoz, mint a logikai bázis, valamint valamilyen egyéb előnyt biztosítsanak.

A következő modulok minden iterációban lefutnak egyszer, tehát implementálásukkor különösen ügyelni kell arra, hogy minél gyorsabbak legyenek.

Megengedettség teszt

Az aktuális bázis meghatározza a bázis változók értékeit, ahogy azt a (1.10) mutatja. Ellenőrizni kell, hogy a bázisváltozók értékei megfelelő határokon belül vannak-e, azaz hogy a megoldás megengedett, vagy sem. Ez a modul az ellenőrzésen túl, abban az esetben, ha ez utóbbi áll fenn, feltölt bizonyos adatszerkezeteket, amelyeket az első fázisú algoritmusok használnak. Elvileg ennek a modulnak csak az első fázisban van értelme, ám a gyakorlat azt mutatja, hogy ellenőrzésre a második fázisban is szükség van, hiszen a numerikus hibák miatt előfordulhat, hogy egy megengedett megoldás nem megengedetté válik.

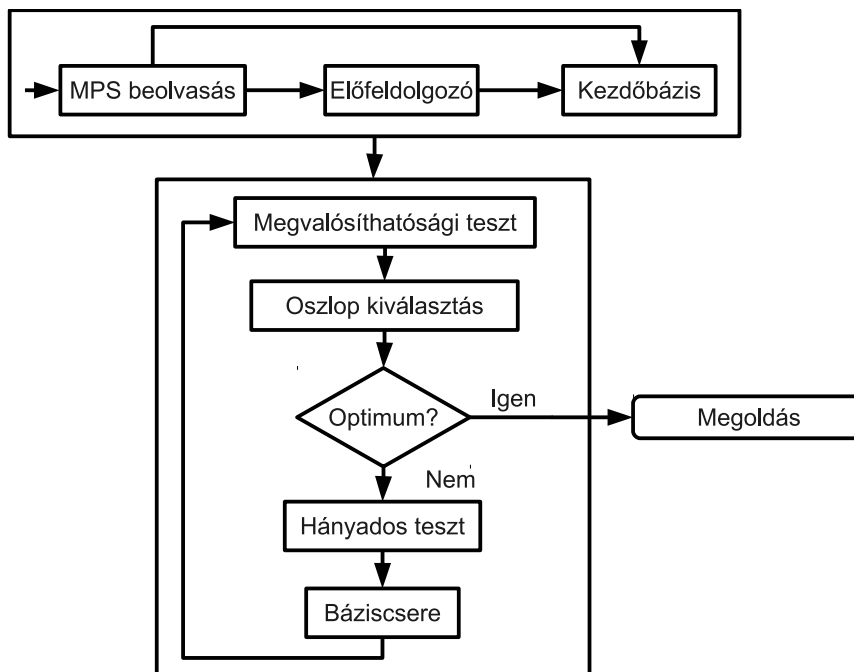
Oszlop kiválasztás

A primál szimplex algoritmusban ez a modul kiválaszt egy nem bázis változót, hogy azt a megoldó algoritmus más részében behozzuk a bázisba. Olyan változót kell választani, amely behozásával javítani lehet az aktuális célfüggvényt. Dolgozatunk ennek elméleti és implementációs kérdéseivel foglalkozik.

Hányados teszt

Az oszlop kiválasztó által javasolt bejövő változóhoz meg kell határozni, hogy az mely bázis változót váltsa a bázisban. A feladatot úgy kell megoldani, hogy második fázisban a megengedettségi feltételek ne sérüljenek, első fázisban pedig mérsekelyjük azok megszegését. Előfordulhat, hogy a csere nem hajtható végre, hanem

1.3. SZORZAT ALAPÚ BÁZIS REPREZENTÁCIÓ



1.1. ábra. A szimplex módszer folyamat ábrája

csupán elég a oszlop kiválasztó által kiválasztott változó értékét egyik korlátjáról a másikra átbillenteni, ezt bound flip-nek hívjuk.

Báziscsere

A modul feladata a báziscsere végrehajtása, illetve egyéb járulékos adminisztratív feladatok ellátása, mint például a bázisváltozók értékeinek aktualizálása. Bizonyos időközönként a bázis teljes újra invertálása is itt történik meg. A hatékonyság növelése érdekében ez a modul összevonható a megengedettség vizsgálattal.

1.3. Szorzat alapú bázis reprezentáció

A szimplex algoritmus során gyakran van szükség arra, hogy egy vektort balról, illetve jobbról megszorozzuk a $\mathbf{B}^{-1}$ mátrixal:

$$\alpha = \mathbf{B}^{-1} \mathbf{a} \quad (1.13)$$

$$\pi^T = \mathbf{h}^T \mathbf{B}^{-1} \quad (1.14)$$

A feladat megoldható úgy, hogy $\mathbf{B}^{-1}$ -et minden iterációban frissítjük, majd elvégezzük a szükséges szorzásokat, ám $\mathbf{B}^{-1}$ karbantartása rendkívül időigényes. Ennek a problémának a megoldására dolgozták ki először a szorzat alapú bázis inverz technikát [3, 7]. A módszer lényege, hogy a $\mathbf{B}^{-1}$ felírható elemi transzformációs mátrixok, azaz ETM-ek szorzataként: Tároljuk ezeket az ETM-eket egy listában, majd

1.3. SZORZAT ALAPÚ BÁZIS REPRESENTÁCIÓ

bázisváltáskor egy új ETM-el bővítjük a listát. Egy ETM gyorsan meghatározható. (1.13) számolásakor ezen ETM-eket felhasználva az ún. FTRAN algoritmust, (1.14) alkalmazásakor pedig az ún. BTRAN algoritmust alkalmazzuk.

Legyen $\mathbf{B}$ az eredeti bázis, $\mathbf{B}_\alpha$ pedig egy ezzel szomszédos bázis, a két mátrix a p oszlopvektorban térnek el egymástól, hiszen $\mathbf{B}$ p oszlopát kicseréltük egy $\mathbf{a}$ oszlopvektorral:

$$\begin{aligned}\mathbf{B} &= [\mathbf{b}_1, \dots, \mathbf{b}_{p-1}, \mathbf{b}_p, \mathbf{b}_{p+1}, \dots, \mathbf{b}_m] \\ &\Downarrow \\ \mathbf{B}_\alpha &= [\mathbf{b}_1, \dots, \mathbf{b}_{p-1}, \mathbf{a}, \mathbf{b}_{p+1}, \dots, \mathbf{b}_m]\end{aligned}$$

Az új ETM megalkotásához szükségünk lesz a $\mathbf{v} = \mathbf{B}^{-1}\mathbf{a}$ vektorra, ezt korábban a hányados teszt során már kiszámoltuk. Képezzük $\mathbf{v}$ -ből az $\boldsymbol{\eta}$ vektort az alábbi módon:

$$\boldsymbol{\eta} = \left[-\frac{v_1}{v_p}, \dots, -\frac{v_{p-1}}{v_p}, \frac{1}{v_p} - \frac{v_{p+1}}{v_p}, \dots, -\frac{v_m}{v_p} \right]$$

A keresett $\mathbf{E}$ ETM a következő:

$$\mathbf{E} = \begin{bmatrix} 1 & & \eta^1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix}$$

A kapott $\mathbf{E}$ mátrix segítségével $\mathbf{B}_\alpha^{-1}$ a következő módon számolható ki:

$$\mathbf{B}_\alpha^{-1} = \mathbf{E}\mathbf{B}^{-1}$$

Az i báziscsere után a bázis inverz a következő:

$$\mathbf{B}_i^{-1} = \mathbf{E}_i\mathbf{E}_{i-1} \dots \mathbf{E}_1,$$

ahol

$$\mathbf{B}_0^{-1} = \mathbf{I}$$

Az FTRAN és BTRAN műveletekhez elegendő csupán az egyes ETM-ekből az $\boldsymbol{\eta}$ vektorokat, illetve az azokhoz tartozó p indexeket tárolni.

FTRAN

Az FTRAN (Forward Transformation) műveletet a (1.13) képlethez használhatjuk. Tegyük fel, hogy az aktuális bázis inverzet s darab ETM szorzataként tároljuk. Tudjuk, hogy

$$\boldsymbol{\alpha} = \mathbf{B}^{-1}\mathbf{a} = \mathbf{E}_s\mathbf{E}_{s-1} \dots \mathbf{E}_1\mathbf{a} \quad (1.15)$$

$$\boldsymbol{\alpha}_0 = \mathbf{a}$$

$$\boldsymbol{\alpha}_i = \mathbf{E}_i\boldsymbol{\alpha}_{i-1}, i = 1, \dots, s$$

1.4. OSZLOP KIVÁLASZTÁSA A SZIMPLEX MÓDSZERBEN

$$\alpha_s = \alpha$$

Az $\alpha = \mathbf{E}\mathbf{a}$ műveletet a következő módon végezhetjük el:

$$\alpha = \mathbf{E}\mathbf{a} = \begin{bmatrix} 1 & & \eta^1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_p \\ \vdots \\ a_m \end{bmatrix} = \begin{bmatrix} a_1 + \eta^1 a_p \\ \vdots \\ \eta^p a_p \\ \vdots \\ a_m + \eta^m a_p \end{bmatrix}$$

Látható, hogy ha egy adott ETM esetén $a_p = 0$, akkor az adott ETM kihagyható az (1.15)-ből.

BTRAN

A BTRAN (Backward Transformation) műveletet pedig a (1.14) képlethez használhatjuk. Továbbra is s darab ETM-el ábrázoljuk a bázis inverzt. Tudjuk, hogy

$$\alpha^T = \mathbf{a}^T \mathbf{B}^{-1} = \mathbf{a}^T \mathbf{E}_s \mathbf{E}_{s-1} \dots \mathbf{E}_1 \quad (1.16)$$

$$\alpha_0^T = \mathbf{a}^T$$

$$\alpha_i^T = \alpha_{i-1}^T \mathbf{E}_{s-i+1}, i = 1, \dots, s$$

$$\alpha_s^T = \alpha^T$$

Az $\alpha^T = \mathbf{a}^T \mathbf{E}$ műveletet a következő módon végezhetjük el:

$$\alpha^T = \mathbf{a}^T \mathbf{E} = \begin{bmatrix} a_1, \dots, a_p, \dots, a_m \end{bmatrix} \begin{bmatrix} 1 & & \eta^1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix}$$

$$\alpha^T = \mathbf{a}^T \mathbf{E} = \begin{bmatrix} a_1, \dots, \sum_{i=1}^m a_i \eta^i, \dots, a_m \end{bmatrix}$$

1.4. Oszlop kiválasztása a szimplex módszerben

Ahogy azt az előző szakaszban említettük, jelen dolgozatban a primál algoritmussal foglalkozunk, így az oszlop kiválasztó modult is ennek megfelelően vizsgáljuk. Primál algoritmusnál az oszlop kiválasztó feladata az, hogy a nem bázis változók közül kiválasszon egyet, amelyet be kell hozni a bázisba. Amennyiben ez a modul nem talál megfelelő változót, úgy az első fázisban jelzi a feladat nem megengedtségét, második fázisban pedig az optimum elérését.

1.4.1. Megengedettségi feltételek

Az algoritmus akkor van az első fázisban, ha a bázisváltozók értékei nem megengedett értékekkel rendelkeznek. Ekkor az algoritmus feladata az, hogy olyan bázist találjon, valamint a nembázis változók értékeit olyan határra tegye, hogy a bázisváltozók értékei elfogadhatóak legyenek [6].

Jelölje $\mathcal{M}$ és $\mathcal{P}$ az alsó korlát alatti, illetve felső korlát feletti bázisváltozók bázison belüli indexeit. Az első fázishoz tartozó célfüggvényben összegezzük a bázisváltozóknak azok alsó vagy felső korlátaitól való eltéréseit:

$$w = \sum_{i \in \mathcal{M}} \beta_i - \sum_{i \in \mathcal{P}} (\beta_i - v_i)$$

Amennyiben $w < 0$, úgy az aktuális megoldás nem megengedett, hiszen van olyan bázisváltozó, amely vagy a felső korlátja felett, vagy az alsó korlátja alatt van. Az oszlop kiválasztó feladata olyan nembázis változó kiválasztása, hogy annak bázisba hozásával w értéke nőjön. Minden nembázis változóhoz kiszámíthatunk egy első fázisú redukált költség értéket:

$$d_j = \sum_{i \in \mathcal{M}} \alpha_j^i - \sum_{i \in \mathcal{P}} \alpha_j^i \quad (1.17)$$

Ennek a redukált költségnek a jelentése a következő: Amennyiben az x_q nembázis változó értékét t -vel növeljük, de a t értéke akkora, hogy $\mathcal{M}$ és $\mathcal{P}$ változatlan marad, úgy a w értéke $-td_q$ -val változik: $\Delta w = -td_q$. Nemnegatív t esetében az oszlop kiválasztónak olyan x_q változót kell választani, hogy $d_q < 0$, nem pozitív t -nél viszont pozitív d_q -t kell keresni. Az (1.2) táblázatban összefoglaltuk, hogy milyen változók esetén milyen értéket vehet fel a redukált költség, hogy a változót kiválaszthassuk.

Típus	Érték	d_j	Elmozdulás	Megjegyzés
0	$x_j = 0$	Bármilyen	0	Nem lép bázisba
1	$x_j = 0$	< 0	+	$j \in \mathcal{U}$
1	$x_j = u_j$	> 0	-	
2	$x_j = 0$	< 0	+	
3	$x_j = 0$	$\neq 0$	+/-	

1.2. táblázat. Oszlop kiválasztási feltételek az első fázisban

Abban az esetben, ha az oszlop választó nem talál olyan nembázis változót, amelyet a bázisba behozva w értéke nő, és $w < 0$, az algoritmus megáll, mivel a feladat nem megoldható.

1.4.2. Optimalitási feltételek

A második fázisban a bázisváltozók értékei megengedett értékeken állnak, azaz $\mathcal{M} \cup \mathcal{P} = \emptyset$. Ez esetben az oszlop választó feladata olyan nembázis változó kiválasztása, hogy annak bázisba hozásával a megoldás közelebb kerüljön az optimumhoz.

1.4. OSZLOP KIVÁLASZTÁSA A SZIMPLEX MÓDSZERBEN

A második fázisban is kiszámolhatunk minden $j \in \mathcal{R}$ nembázis változóhoz egy redukált költséget, a következő képlet alapján:

$$d_j = c_j - \mathbf{c}_B \mathbf{B}^{-1} \mathbf{a}_j \quad (1.18)$$

A bázisváltozókhoz tartozó redukált költségek a 0 értéket veszik fel. Ezen érték jelentése a következő: Amennyiben az x_q változó értékét megváltoztatjuk θ -val, úgy a feladat célfüggvényének értéke a következő módon változik:

$$\bar{z} = z + \theta d_q \quad (1.19)$$

Az alsó korláton lévő nembázis változókat csak növelhetjük, a felső korláton lévőket csak csökkenthetjük, a korláttal nem rendelkezőket pedig bármely irányba mozgathatjuk, valamint a fix változókat nem változtathatjuk. Az (1.3) táblázatban összefoglaltuk, hogy egy nembázis változó mikor alkalmas arra, hogy az oszlop választáskor kiválasszuk.

Típus	Érték	d_j	Elmozdulás	Megjegyzés
0	$x_j = 0$	Lényegtelen	$t = 0$	$j \in \mathcal{U}$
1	$x_j = 0$	< 0	$0 \leq t \leq u_k$	
1	$x_j = u_j$	> 0	$-u_k \leq t \leq 0$	
2	$x_j = 0$	< 0	$t \geq 0$	
3	$x_j = 0$	$\neq 0$	$-\infty \leq t \leq \infty$	

1.3. táblázat. Oszlop kiválasztási feltételek a második fázisban

A megoldás akkor optimális, ha nincs olyan nembázis változó, amelyet eleget tesz az (1.3) táblázatban felsorolt feltételek egyikének: Ebben az esetben, ha egy nembázis változót behozunk a bázisba, nem javíthatunk a célfüggvényen.

1.4.3. Oszlop választó stratégiák

Első és második fázisban a (1.4.1), (1.4.2)-ban ismertetett szempontok alapján ki kell választani egy megfelelő javító változót, hogy azt a bázisba behozva közelebb kerüljünk a megengedett megoldáshoz, vagy az optimumhoz. Több javító változó létezésekor bármelyik kiválasztható, a szimplex algoritmus logikailag helyesen fog működni. Ám az oszlop választó által alkalmazott kiválasztási szabály nagyban befolyásolja az optimalizálási algoritmus futási idejét, hiszen ez a választás kihat az iterációk számára. Ezért az egyik elvárás az oszlop választó modullal szemben az, hogy úgy válasszon bejövő változót, hogy minél közelebb kerüljünk az optimumhoz. Másodszor, ahogy azt a (1.4.1) és (1.4.2) szakaszokban láthattuk, a redukált költségek kiszámítása nagyon számítás igényes. A második elvárás éppen ezért az, hogy az oszlop választó minél gyorsabban hajtsa végre a bejövő változó kiválasztását.

Az oszlop választás sebességének növelésére, valamint a választás jóságának fokozására is léteznek különféle technikák.

Dantzig

A legelső oszlop választó stratégiát Dantzig javasolta. Lényege, hogy azt a javító változót kell választani, amelynek a redukált költsége a legtávolabb van nullától. Ennek oka az, hogy a redukált költség értéke szerepel a célfüggvény értékének megváltozásában. Az elvárás tehát az, hogy a célfüggvény értéke minél gyorsabban konvergáljon az optimumhoz. Sajnos a módszer hátránya, hogy a célfüggvény értéke a (1.19) szerint változik, ahol a θ értékének is szerepe van. Ezt pedig a hánados tesztben kapjuk meg, osztási műveletek eredményeként, ahol az x_q kiválasztott nem bázis változó α_q vektorának nem nulla értékeivel kell osztani. Amennyiben az α_q komponensei nagyok, úgy a θ kicsi lesz, így a célfüggvény értéke csekély mértékben módosul. A normalizált oszlop választási stratégiák ezt a hátrányt igyekeznek kiküszöbölni.

Részleges oszlop választás

A Dantzig stratégiát eredetileg úgy alkalmazták, hogy az összes nem bázis változóra kiszámolták a redukált költségeket, majd ezek közül választottak egyet. Ezt a módszert teljes oszlop kiválasztásnak nevezzük. Nagy számú nem bázis változó esetén ez jelentős többlet munkát igényel. Az OSA-60 nevű feladat 10 280 korlátozást és 232 966 változót tartalmaz. Tehát teljes oszlop választás alkalmazása esetén minden iterációban 232 966 változó redukált költségét kell számolni, ami a számolásban szereplő skalár-szorzat miatt nagyon megnöveli a futási időt. Ezt a módszert alkalmazva, egy Intel(R) Core(TM)2 Duo CPU P7370, 2 GHz-es processzorral, 3 GiB DDR3-as memóriával felszerelt számítógépen ennek a feladatnak a megoldása 473.35 másodpercig tartott, ebből az oszlop választó futási ideje 454.3 másodperc. A szimplex 8975 iterációval oldotta meg a feladatot, tehát átlagosan az oszlop kiválasztó egy iterációra 0.05 másodpercet szánt.

Tudjuk azonban, hogy ha találunk legalább egy javító változót, akkor azt kiválaszthatjuk. Ezért nagy változó szám esetén nem feltétlenül éri meg, hogy minden változót megvizsgáljunk, elég csak egy részükre koncentrálni minden iterációban. Ezt a módszert hívjuk részleges oszlop választásnak. Az egyik lehetséges módszer, hogy felosztjuk a változókat n darab partícióra, és minden iterációban kiválasztunk egy partíciót, és csak a benne szereplő változókat vizsgáljuk meg. Amennyiben nem találunk javító változót, úgy a következő partícióra ugrunk. A módszer tovább finomítható, hogy azt is kikötjük, hogy egy partícióban egy iterációban legalább m változót kell átnézni, tehát nem feltétlenül kell egy partíciót sem teljesen megvizsgálni. Az előbbi feladatot megoldottuk úgy, hogy a változókat 100 partícióra osztottuk, és minden partícióban legalább 100 változót néztünk végig 1-1 iterációban. Ebben az esetben a teljes futási idő 39.75 másodperc, amiből a oszlop választó ideje 16.19 másodperc. Ám az iterációk száma 11 407-re nőtt, ekkor az oszlop választó egy iteráción átlagosan 0.00142 másodpercig dolgozott, ami 35x-ös gyorsulást jelent.

Használhatunk olyan stratégiát is, hogy az oszlop választó az első megtalált javító változót válassza ki. A fenti modellt ekkor 60.72 másodperc alatt oldottuk meg, ahol az oszlop választó 13.05 másodpercig futott. A megnövekedett futási időt az okozta, hogy 21 061-re nőtt az iterációk száma. Bár az oszlop választó egy iteráción ekkor már csak 0.00062 másodpercig dolgozott, az általa meghozott

döntés minősége romlott, így a megoldó algoritmusnak tovább kellett dolgoznia, hogy eljusson az optimális megoldásig.

1.4.4. Redukált költségek számolása

Az alábbiakban röviden ismertetjük, hogy a primál első, illetve második fázisban hogyan számolhatóak hatékonyan az egyes redukált költségek.

Első fázisú redukált költségek

Az első fázisú redukált költségek definíciója (1.17) szerint a következő:

$$d_j = \sum_{i \in \mathcal{M}} \alpha_j^i - \sum_{i \in \mathcal{P}} \alpha_j^i$$

Látható, hogy a j redukált költség számolásához a definíció alapján szükségünk van az $\alpha_j = \mathbf{B}^{-1} \mathbf{a}_j$ vektorra. Mivel egy iterációban több redukált költséget is számolunk, ezért nem célszerű az előző képlet alapján előállítani az egyes α_j vektorokat, mivel ez nagyban rontja az algoritmus hatékonyságát. A probléma megoldására vezessük be a $\mathbf{h} \in \mathbb{R}^m$ vektort, amelynek komponensei a következők:

$$h_i = \begin{cases} 1, & \text{ha } i \in \mathcal{M} \\ -1, & \text{ha } i \in \mathcal{P} \\ 0, & \text{egyébként} \end{cases}$$

Ez alapján a j változó redukált költsége:

$$d_j = \mathbf{h}^T \alpha_j = \mathbf{h}^T \mathbf{B}^{-1} \mathbf{a}_j$$

Mivel minden redukált költség esetén a $\mathbf{h}^T \mathbf{B}^{-1}$ szorzat ugyanaz, ezért vezessük be az első fázisú szimplex szorzót:

$$\phi^T = \mathbf{h}^T \mathbf{B}^{-1}$$

A redukált költségeket ezek alapján így is kiszámíthatjuk:

$$d_j = \phi^T \mathbf{a}_j$$

Ennek köszönhetően egy redukált költséghez csupán egy skalárszorzásra van szükség, így a számítás nagyságrendekkel gyorsabb lesz. Ezen szorzások előkészítő lépése pedig egy BTRAN művelet, azaz a bázis inverzet ábrázoló ETM-ek és a $\mathbf{h}^T$ alapján meg kell határozni a ϕ^T -t.

A még gyorsabb számoláshoz érdemes az alábbi megközelítést használni:

$$\mathbf{d}_R^T = \phi^T \mathbf{R} \quad (1.20)$$

Amennyiben $\mathbf{R}$ sorfolytonosan is tárolva van, így a redukált költségek kiszámításához elegendő ennek a mátrixnak a megfelelő sorait súlyozottan összeadni.

Második fázisú redukált költségek

A második fázisú redukált költségek definíciója (1.18) szerint a következő:

$$d_j = c_j - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{a}_j$$

Hasonlóan az első fázishoz, bevezethetjük a második fázisú szimplex szorzót:

$$\boldsymbol{\pi}^T = \mathbf{c}_B^T \mathbf{B}^{-1} \quad (1.21)$$

Így a redukált költségek az alábbi módon is számolhatóak:

$$d_j = c_j - \boldsymbol{\pi}^T \mathbf{a}_j \quad (1.22)$$

Ennek a szimplex szorzónak az előnye ugyanaz, mint az első fázis esetén, azaz mielőtt kiszámolnánk a redukált költségeket, az előtt elvégzünk egy BTRAN-t, hogy megkapjuk $\boldsymbol{\pi}^T$ -t, majd skalárszorzatok és egy kivonás segítségével gyorsan megkaphatjuk az eredményeket.

Ismert, hogy ha a bázis p oszlopát kicseréljük egy másik oszlopra, akkor az új második fázisú szimplex szorzót ($\bar{\boldsymbol{\pi}}$) kiszámíthatjuk az alábbiak szerint is:

Legyen

$$\bar{\boldsymbol{\rho}}^p = \mathbf{e}_p^T \mathbf{B}^{-1}$$

$$\bar{\boldsymbol{\pi}}^T = \boldsymbol{\pi}^T + d_q \bar{\boldsymbol{\rho}}^p, \quad (1.23)$$

ahol d_q az előző iterációban kiválasztott redukált költség.

Mivel az $\mathbf{e}_p$ vektor egyetlen nem nulla elemet tartalmaz, a $\bar{\boldsymbol{\rho}}^p$ sorvektor számolása alkalmas BTRAN implementáció használata mellett hamarabb elvégezhető, mintha (1.21) szerint számolnánk ki a szimplex szorzót.

Közvetlenül a redukált költségeket is frissíthetjük az alábbi módon:

$$\bar{d}_j = d_j - d_q \bar{\boldsymbol{\rho}}^p \mathbf{a}_j$$

Mivel $\bar{\boldsymbol{\rho}}^p$ általában kevesebb nem nulla elemet tartalmaz, mint $\boldsymbol{\pi}^T$, ezért célszerű az alábbiak szerint frissíteni a redukált költségeket:

$$\bar{\mathbf{d}}_R^T = \mathbf{d}_R^T - d_q \bar{\boldsymbol{\rho}}^p \mathbf{R}$$

Hasonlóan az első fázishoz, amennyiben $\mathbf{R}$ sorfolytonosan is tárolva van, a redukált költségekhez itt is néhány sor súlyozott összegzésére van szükség.

2. fejezet

Gyorsított teljes oszlopkiválasztás

2.1. Primál első fázisú szimplex szorzó frissítése

Ahogy azt az előző fejezetben láttuk, a primál második fázisra korábban is ismert volt egy képlet (1.23), amely segítségével a második fázisú szimplex szorzót frissíthettük. Az alábbiakban bemutatjuk, hogy ezt hogy végezhetjük el az első fázisban is. A szimplex szorzót egy adott $\mathbf{h}$ vektor és az aktuális $\mathbf{B}^{-1}$ bázis inverz alapján számoljuk ki:

$$\phi^T = \mathbf{h}^T \mathbf{B}^{-1} \quad (2.1)$$

Áttérünk a $\mathbf{B}$ -ről a $\bar{\mathbf{B}}$ bázisra, a $\mathbf{h}$ vektor pedig $\bar{\mathbf{h}}$ -ra változik. Legyen

$$\Delta \mathbf{h} = \bar{\mathbf{h}} - \mathbf{h}. \quad (2.2)$$

Feltételezzük, hogy $\Delta \mathbf{h}$ a $\bar{\mathbf{h}}$ -hoz képest kevesebb nem nulla elemet tartalmaz, ezért azt várjuk el, hogy $\Delta \mathbf{h}$ segítségével kevesebb művelettel kaphassuk meg az új szimplex szorzót.

A feladat az, hogy $\bar{\phi}^T = \bar{\mathbf{h}}^T \bar{\mathbf{B}}^{-1}$ értékét ϕ^T , $\Delta \mathbf{h}$ és $\bar{\mathbf{B}}^{-1}$ alapján határozzuk meg. Tudjuk, hogy

$$\bar{\phi}^T = \bar{\mathbf{h}}^T \bar{\mathbf{B}}^{-1} = (\mathbf{h} + \Delta \mathbf{h})^T \bar{\mathbf{B}}^{-1} = \mathbf{h}^T \bar{\mathbf{B}}^{-1} + \Delta \mathbf{h}^T \bar{\mathbf{B}}^{-1} \quad (2.3)$$

A $\Delta \mathbf{h}^T \bar{\mathbf{B}}^{-1}$ művelet elvégzése a $\Delta \mathbf{h}$ ritkassága miatt gyorsan elvégezhető, viszont a $\mathbf{h}^T \bar{\mathbf{B}}^{-1}$ szorzat kiszámítása akár lassabb is lehet, mintha közvetlenül $\bar{\mathbf{h}}$ -ból számolnánk az új szimplex szorzót, hiszen $\bar{\mathbf{h}}$ vektor kevesebb nem nulla elemet tartalmaz mint $\mathbf{h}$. Ezért $\mathbf{h}^T \bar{\mathbf{B}}^{-1}$ kiszámítására más módszerre van szükség.

Az (2.1) átrendezéséből azt kapjuk, hogy

$$\phi^T \mathbf{B} = \mathbf{h}^T$$

Legyen $\hat{\phi}$ az a vektor, amire $\mathbf{h}^T \bar{\mathbf{B}}^{-1} = (\phi + \hat{\phi})^T$, ezt rendezzük $\mathbf{h}^T$ -ra:

$$(\phi + \hat{\phi})^T \bar{\mathbf{B}} = \mathbf{h}^T,$$

tehát

$$(\phi + \hat{\phi})^T \bar{\mathbf{B}} = \phi^T \mathbf{B}$$

2.1. PRIMÁL ELSŐ FÁZISÚ SZIMPLEX SZORZÓ FRISSÍTÉSE

$$\begin{aligned}
 \phi^T \bar{\mathbf{B}} + \hat{\phi}^T \bar{\mathbf{B}} &= \phi^T \mathbf{B} \\
 \hat{\phi}^T \bar{\mathbf{B}} &= \phi^T \mathbf{B} - \phi^T \bar{\mathbf{B}} \\
 \hat{\phi}^T \bar{\mathbf{B}} &= \phi^T (\mathbf{B} - \bar{\mathbf{B}}) \\
 \hat{\phi}^T &= \phi^T (\mathbf{B} - \bar{\mathbf{B}}) \bar{\mathbf{B}}^{-1}
 \end{aligned} \tag{2.4}$$

A (2.3), (2.2), valamint (2.4) alapján bázisváltás, és $\mathbf{h}$ -nak $\Delta \mathbf{h}$ -val való változása után

$$\bar{\phi}^T = \phi^T + (\phi^T (\mathbf{B} - \bar{\mathbf{B}}) + \Delta \mathbf{h}^T) \bar{\mathbf{B}}^{-1}. \tag{2.5}$$

Mivel $\mathbf{B}$ és $\bar{\mathbf{B}}$ egy oszlopvektorukban különböznek egymástól, ezért $\mathbf{B} - \bar{\mathbf{B}}$ egy olyan mátrix, amelynek egy oszlopát kivéve minden eleme 0. Emiatt $\phi^T (\mathbf{B} - \bar{\mathbf{B}})$ egy olyan vektor, amelynek egy kivételével minden komponense nulla. Tehát $\phi^T (\mathbf{B} - \bar{\mathbf{B}})$ kiszámolásához egy vektor kivonásra, valamint egy skalárszorzatra van szükség.

A fentiek ismeretében már az első fázisú redukált költségeket is frissíthetjük, hasonlóan, mint ahogy a második fázisban a (1.20) képletet alkalmaztuk. Az első fázisú szimplex szorzó változását jelöljük így:

$$\Delta \phi^T = (\phi^T (\mathbf{B} - \bar{\mathbf{B}}) + \Delta \mathbf{h}^T) \bar{\mathbf{B}}^{-1}.$$

Ekkor az első fázisú redukált költségek változása a következő: Tudjuk, hogy

$$\mathbf{d}_R^T = \phi^T \mathbf{R},$$

tehát

$$\bar{\mathbf{d}}_R^T = \phi^T \mathbf{R} + \Delta \phi^T \mathbf{R} = \mathbf{d}_R^T + \Delta \phi^T \mathbf{R} \tag{2.6}$$

A szimplex szorzó számolása előtt ellenőrizni kell, hogy a $\phi^T (\mathbf{B} - \bar{\mathbf{B}}) + \Delta \mathbf{h}^T$ vektor ritkásabb-e, mint a $\mathbf{h}$, amennyiben igen, akkor ezt a képletet érdemes használni.

A $\phi^T (\mathbf{B} - \bar{\mathbf{B}})$ alkalmas vektor reprezentációk mellett három, egymás utáni ciklussal is implementálható. Mivel a feladat együttható mátrixa rendkívül kevés nem nullát tartalmaz, ezért csak ezen értékeket tároljuk: Minden vektor esetén (idx, v) párokat tárolunk, ahol az idx index azt adja meg, hogy a v érték hányadik komponense a vektornak. Erre az ábrázolási módra ritkás reprezentációval fogunk hivatkozni. A $\mathbf{B}$ oszlopvektorai ilyen módon kerülnek tárolásra. Ez lassulást okozna akkor, ha a mátrix egy adott elemét külön le kell kérdezni, hiszen ehhez egy $O(n)$ bonyolultságú keresés szükséges. Szerencsére ilyen eset nem fordul elő, az egyes oszlopok használatakor minden oszlopon belüli nem nullát felhasználunk, így az egyes algoritmusok implementálásakor úgy járunk el, hogy egyenként végighaladunk ezeken az értékeken. Ettől még hatékonyabb is lesz a program, hiszen nem kell a rengeteg 0-val is foglalkozni. A ϕ szimplex szorzót viszont a nullákkal együtt ábrázoljuk, mivel ezen vektor esetén szükség van az egyes komponensek külön-külön való lekérdezésére. Mivel ebből a vektorból csak egy létezik, ezért ez a mai számítógépeken nem fog észrevehető memóiafogyasztást eredményezni.

A ritkás reprezentációnál általában nincs biztosítva az, hogy az egyes komponensek index szerint növekvő sorrendben kövessék egymást, hiszen ennek biztosítása túl nagy többlet terhet róna az egyes vektor kezelő algoritmusokra. Ám az együttható mátrix vektorai esetén kivételt tehetünk, hiszen ez egy statikus mátrix,

azaz a szimplex futása során az előfeldolgozó lépések után nem változtatunk rajta semmit. Ezt kihasználva a mi implementációnkban az egyes együttható mátrix vektorokon belüli (idx, v) párokat az idx indexek szerint növekvő sorrendbe rendeztük. A gyors rendezés érdekében a rendezendő elemek számától függően szelekciós, illetve leszámlálós rendezést használtunk: Csak az egyik megoldás használata érezhetően lassabb, mintha minden vektor esetén kiválasztanánk a méretéhez legmegfelelőbb rendező algoritmust. A két technika alkalmazásával viszont alig érezhető a rendezési idő. Megjegyezzük, kipróbáltuk, hogy változik-e a teljes rendezés ideje, ha az $O(n^2)$ -es szelekciós és $O(n)$ -es leszámlálós rendező mellé berakunk valamilyen $O(\log_2)$ -es algoritmust, ám azt tapasztaltuk, hogy nem történt észrevehető javulás.

A $\phi^T(\mathbf{B} - \bar{\mathbf{B}})$ kiszámolásához tehát azt kell kihasználni, hogy $\mathbf{B}$ és $\bar{\mathbf{B}}$ csupán egy vektorban térnek el egymástól, jelöljük ezeket a vektorokat $\mathbf{b}$ -vel és $\bar{\mathbf{b}}$ -vel. A második tény, amit kihasználhatunk, hogy $\mathbf{b}$ és $\bar{\mathbf{b}}$ ritkás reprezentációval vannak ábrázolva, valamint a tárolt (idx, v) párok index szerint növekvő sorrendben adóttak. Végül, tudjuk, hogy ϕ összes komponensét tároljuk, beleértve a nullákat is. Az ezeket kihasználó algoritmus az összefésülő rendezésre épül: Végignézzük $\mathbf{b}$ és $\bar{\mathbf{b}}$ nem nulla elemeit, mégpedig úgy, hogy k -val jelöljük $\mathbf{b}$, és l -el $\bar{\mathbf{b}}$ aktuális (idx, v) párjának sorszámát. Az algoritmus feladata az, hogy összegezze a $\sum_{i=1}^n \phi_i(b_i - \bar{b}_i)$ értékeket. Kezdetben k és l 1-el egyenlő. Megnézzük, hogy k és l hogy viszonyulnak egymáshoz: Ha egyenlőek, akkor ez a két változó olyan (idx, v) párokra hivatkozik a két vektoron belül, ahol az indexek egyenlőek. Ekkor a két értékkel elvégezzük a kivánást, majd az eredményt megszorozzuk ϕ_{idx} elemével, végül k -t és l -t egyel növeljük. Amennyiben k kisebb, mint l , akkor az azt jelenti, hogy k -val $\mathbf{b}$ -nek a $(idx1, v1)$, míg l -el a $\bar{\mathbf{b}}$ -nek egy $(idx2, v2)$ párjára hivatkozunk, ahol $idx1 < idx2$. Ez annyit jelent, hogy a $\phi_{idx1}(v1 - 0)$ műveletet kell elvégezni, majd növeljük k -t. A harmadik eset pedig az, mikor $k > l$, ekkor az előző eset mintájára a $\phi_{idx2}(0 - v2)$ -t kell kiszámolni és megnöveljük l -t. Az algoritmus pseudo kódja a (2.1) ábrán látható. A vektorok dimenziószáma n , továbbá jelölje $c(\mathbf{a})$ az $\mathbf{a}$ ritkás reprezentációval tárolt vektor (idx, v) párjainak számát, végül $nz(\mathbf{a}, i)$ jelölje az $\mathbf{a}$ vektor i (idx, v) párját.

Természetesen az (2.5) képlet kihasználásához olyan BTRAN implementációra van szükség, amely kihasználja azt, hogy a műveletet olyan vektoron végezzük el, amely kevés nem nullát tartalmaz.

2.2. Sorfolytonos bázis inverz reprezentáció

Korábban láthattuk, hogy az oszlop kiválasztó algoritmus felhasználja a BTRAN műveletet. Célszerű ezt az algoritmust minél hatékonyabban implementálni. Az alábbiakban bemutatunk két implementációt is, majd a mérések során összehasonlítjuk a kettőt.

Elevenítsük fel a (1.16) alapján a BTRAN képletét:

$$\alpha^T = \mathbf{a}^T \mathbf{B}^{-1} = \mathbf{a}^T \mathbf{E}_s \mathbf{E}_{s-1} \dots \mathbf{E}_1$$

A fentiek alapján következő egyszerű algoritmust alkalmazhatjuk: A számolás

Bemenet: $\mathbf{b}$, $\bar{\mathbf{b}}$ és ϕ vektorok
Kimenet: s skalár

```

 $s := 0$ 
 $k := 1$ 
 $l := 1$ 
ciklus amíg  $k < c(\mathbf{b})$  és  $l < c(\bar{\mathbf{b}})$ 
    ha  $k = l$  akkor
        Legyen  $(idx1, v1) := nz(\mathbf{b}, k)$ 
        Legyen  $(idx2, v2) := nz(\bar{\mathbf{b}}, l)$ 
         $s := s + \phi_{idx1}(v1 - v2)$ 
         $k := k + 1$ 
         $l := l + 1$ 
    egyébként ha  $k < l$  akkor
        Legyen  $(idx, v) := nz(\mathbf{b}, k)$ 
         $s := s + \phi_{idx}v$ 
         $k := k + 1$ 
    egyébként
        Legyen  $(idx, v) := nz(\bar{\mathbf{b}}, l)$ 
         $s := s - \phi_{idx}v$ 
         $l := l + 1$ 
elágazás vége
ciklus vége
ciklus amíg  $k < c(\mathbf{b}_p)$ 
    Legyen  $(idx, v) := nz(\mathbf{b}, k)$ 
     $s := s + \phi_{idx}v$ 
     $k := k + 1$ 
ciklus vége
ciklus amíg  $l < c(\bar{\mathbf{b}}_p)$ 
    Legyen  $(idx, v) := nz(\bar{\mathbf{b}}, l)$ 
     $s := s - \phi_{idx}v$ 
     $l := l + 1$ 
ciklus vége

```

2.1. ábra. Algoritmus a $\phi^T(\mathbf{B} - \bar{\mathbf{B}})$ kiszámolására

2.2. SORFOLYTONOS BÁZIS INVERZ REPREZENTÁCIÓ

Bemenet: $\mathbf{a}$
Kimenet: α
ciklus $i := s$ -től 1 -ig
 $d := 0$
 ciklus minden $\eta_i^j \neq 0$ -ra:
 $d := d + a_j \eta_i^j$
 ciklus vége
 $a_{p_i} := d$
ciklus vége
 $\alpha := \mathbf{a}$

2.2. ábra. Oszlopfolytonos η reprezentációt alkalmazó BTRAN algoritmus

során elég az ETM-ekből az η vektorokat, illetve az ETM-ekhez tartozó p indexeket tárolni. Használjuk a következő jelöléseket:

- s : A tárolt ETM-ek száma
- η_i : Az i ETM-hez tartozó η vektor
- p_i : Az i ETM-hez tartozó p index
- $\mathbf{a}$: Az algoritmus bemenete, n dimenziós vektor

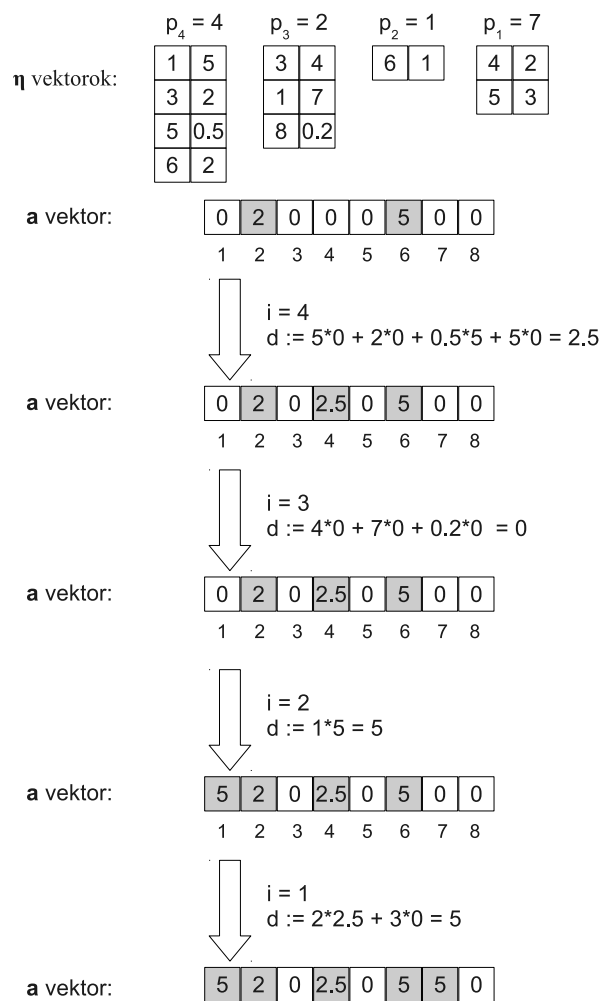
Feltételezzük, hogy az $\mathbf{a}$ vektor ún. direkt alakban adott, azaz tároljuk a nulla és nem nulla elemeit egyaránt. Ennek a tárolásnak az az előnye, hogy $O(1)$ idő alatt lekérdezhetőek a vektor elemei. A vektor így nyilván több memóriát foglal, mintha csak a nem nullákat tárolnánk, viszont ebben az esetben ez nem okoz problémát. Az egyes η_i vektorok viszont ritkás alakban adóttak, azaz mindegyik vektor esetén egy (index, érték) listát használunk a nem nulla elemek tárolására.

A BTRAN során skalárszorzatokat számolunk ki az egyes η_i és az $\mathbf{a}$ vektorok között, majd az eredményt visszaírjuk az $\mathbf{a}$ vektor p_i pozíciójára:

A (2.2) algoritmus futási ideje attól függ, hogy mennyi η vektor van, illetve az egyes vektorok mennyi nem nullát tartalmaznak. Az algoritmus hátránya az, hogy a belső ciklusban az $a_j \eta_i^j$ szorzat egyik tagja gyakran 0, emiatt d ekkor nem fog változni, azaz rengeteg szorzást feleslegesen hajtunk végre. A másik hátrány az, hogy az algoritmus nem lesz gyorsabb, ha a bemenő $\mathbf{a}$ vektor kevesebb nem nullát tartalmaz. Ezt illusztráltuk a (2.3) ábrán: Van olyan iteráció is, ahol az $\mathbf{a}$ vektor nem is változik meg. Indokolt tehát egy olyan algoritmust tervezni, amely mentes az előző kettő hátránytól.

A hatékonyság növeléséhez más módon kell tárolni az η vektorokat. Ezen vektorokból továbbra is a nem nullákat, illetve azok vektoron belüli indexeit tároljuk. Az η és $\mathbf{a}$ vektorok m elemet tartalmaznak. Használjunk egy L_η nevű, m elemű tömböt, mégpedig úgy, hogy ez a tömb 1-1 listára hivatkozik. Egy lista (index, érték) párokat tartalmaz a következő módon: A j lista tartalmazza az η vektorok azon elemeinek indexeit és értékeit, ahol $\eta_i^j \neq 0$. A listák létrehozásakor ügyelni kell arra, hogy egy listán belül az elemek az index szerint sorrendbe legyenek rendezve. Ez könnyen teljesíthető: Új ETM létrehozásakor egy új η vektort kell hozzáadni a

2.2. SORFOLYTONOS BÁZIS INVERZ REPREZENTÁCIÓ



2.3. ábra. Példa az oszlopfolytonos reprezentációt alkalmazó BTRAN-ra

2.2. SORFOLYTONOS BÁZIS INVERZ REPRESENTÁCIÓ

listához, ez úgy történik, hogy a vektor nem nulla elemeit a megfelelő listák elejére fűzzük. Láncolt listák használata esetén ez egy elem esetén $O(1)$ időben elvégezhető. Ekkor nyilván a később létrehozott η vektor elemei a listák elejére kerülnek. Erre az adatszerkezetre mutat példát a (2.4) ábra.

Az új adatszerkezetre épülő BTRAN első lépésben a bemeneti $\mathbf{a}$ vektorból ki-gyűjti a nem nulla elemek értékeit, illetve indexeit. Ha az $a_j \neq 0$, akkor feljegyezzük, hogy az $L_\eta[j]$ láncolt lista legelső elemére majd szükségünk lesz: Erre használunk egy másik tömböt, az L_a -t, amely s elemű (azaz annyi eleme van, ahány η vektort tárolunk), és szintén láncolt listákat tartalmaz. A tömb j láncolt listája olyan pointereket tartalmaz, amelyek az L_η tömb láncolt listáinak elemeire hivatkoznak: Ha az $a_j \neq 0$, akkor megnézzük, hogy $L_\eta[j]$ lista első eleme hányas η_i vektorhoz tartozik. A következő lépésben az L_a i listájához hozzáfűzünk egy mutatót, amely az $L_\eta[j]$ első elemére mutat. Egy példán keresztül ezt mutatjuk be a (2.5) ábrán.

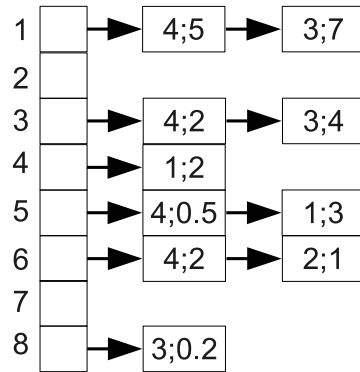
Ebből az adatszerkezetből kiolvasható, hogy az $\mathbf{a}$ -t először melyik η_i vektorral kell szorozni: Megkeressük az L_a tömb legutolsó elemét, amely tárol láncolt listát, ennek az indexét jelöljük $maxIndex$ -el. Végighaladunk az $L_a[maxIndex]$ tömb elemein, és ezen elemekből kiolvashatjuk, az $\eta_{maxIndex}$ vektor mely elemeit kell szorozni az $\mathbf{a}$ vektor megfelelő elemeivel. Ennek hatására az egyes szorzatok eredményei nem nullák lesznek, tehát nem végzünk el felesleges szorzást. Az $L_a[maxIndex]$ láncolt lista elemeit vagy töröljük, vagy áthelyezzük a tömb egy másik láncolt listájába. Amennyiben az aktuális láncolt lista elem olyan L_η -beli elemre hivatkozik, amelynek van következő eleme, akkor lekérdezzük ezt a következő elemet, majd az alapján, hogy ez az elem melyik η_k -hoz tartozik, az eredeti $L_a[maxIndex]$ lista elemét a k listához fűzzük hozzá. Ellenkező esetben pedig nincs hozzáfűzés. Ezt illusztrálja a (2.6) ábra.

A skalárszorzás eredményét el kell tárolni az $a_{p_{maxIndex}}$ elemében, majd az L_a -hoz új elemet kell keresni, mégpedig az $L_\eta[p_{maxIndex}]$ lista azon első elemét, ahol az index a lehető legnagyobb, de kisebb, mint $maxIndex$. A megfelelő ilyen elem kiválasztása alkalmas láncolt lista adatszerkezet mellett $O(\log_2)$ időben megvalósítható. Ezt mutatja be az (2.7) ábra. Végül L_a -ben megkeressük a $maxIndex$ -nél következő kisebb sorszámú listát, és a fenti algoritmust újra kezdjük.

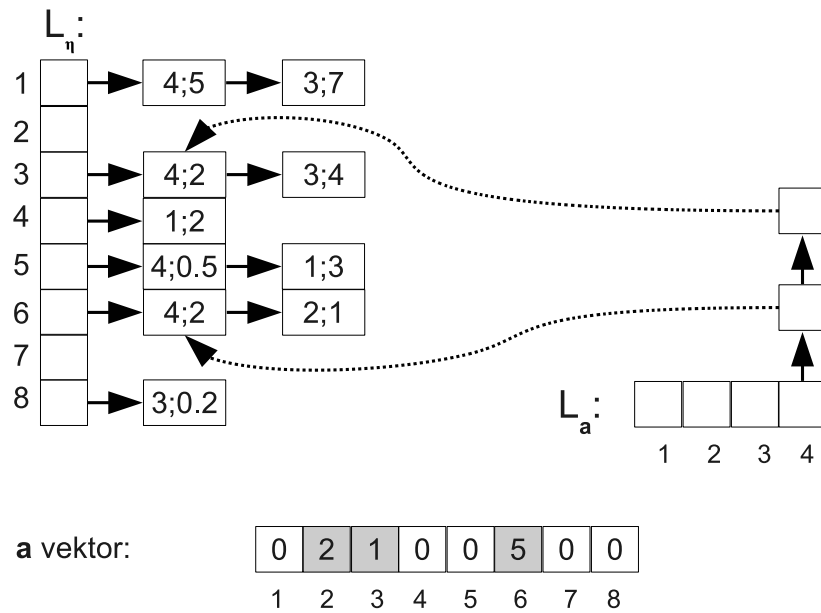
Figyelni kell arra az esetre, mikor a skalárszorzat eredménye 0. Ekkor törölni kell L_a -ból azt a mutatót, amely olyan L_η -beli lánc elemere mutat, amely a $p_{maxIndex}$ -edik láncolt listához tartozik. Ekkor csinálhatjuk azt, hogy megkeressük ezt a mutatót, ám ennek bonyolultsága $O(m)$. Ennél gyorsabban célt érhetünk, hogy ha $\mathbf{a}$ minden nem nulla eleméhez redundáns módon eltároljuk a megfelelő L_a -beli mutatót, így a keresett hivatkozás $O(1)$ időben előállítható. Ezt a redundáns mutatót tárolhatjuk egy L_d nevű tömbben.

A (2.7) ábrán látható példában $maxIndex$ következő értéke 3 lesz. Az $L_a[3]$ lista egyetlen elemet tartalmaz, ez pedig egy hivatkozás az η_3^3 -ra. Így egy szorzást hajtunk végre, szemben az oszlopfolytonos algoritmussal, ahol az η_3 vektornál 3 szorzás lenne, abból kettő eredménye 0. Mivel $p_3 = 2$, ezért az eredményt az a_2 -ben mentjük el. Láthatjuk, hogy a_2 korábban sem 0 volt, ezért most nem kell $L_\eta[2]$ -ben új elemet keresni, hogy L_a -ba beszúrhassuk (2.8).

2.2. SORFOLYTONOS BÁZIS INVERZ REPREZENTÁCIÓ

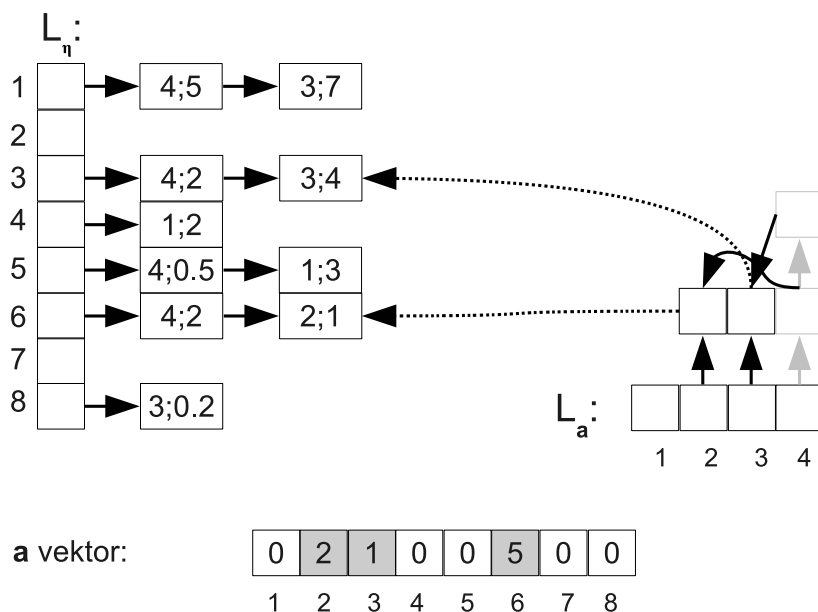


2.4. ábra. A (2.3) példában ábrázolt η vektorok sorfolytonosan tárolva

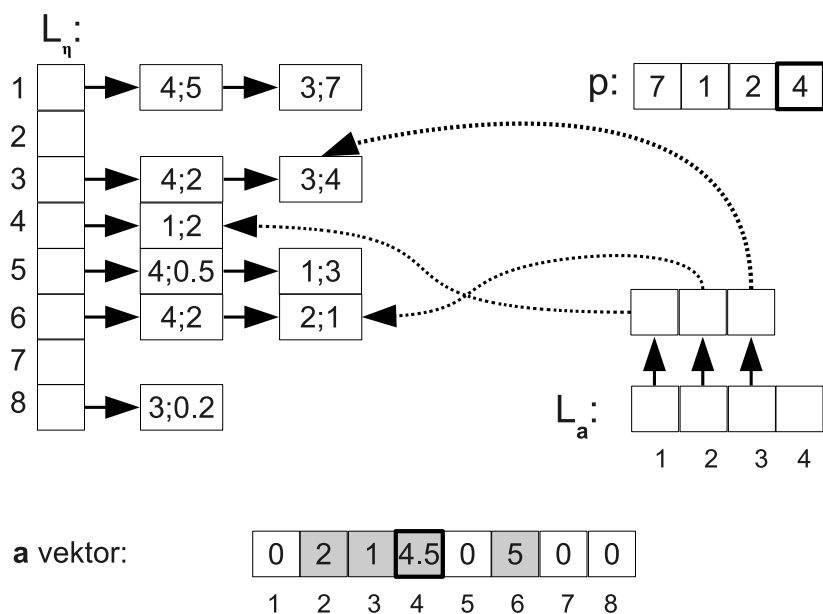


2.5. ábra. Példa az L_η és L_a tömbök kapcsolatára

2.2. SORFOLYTONOS BÁZIS INVERZ REPREZENTÁCIÓ

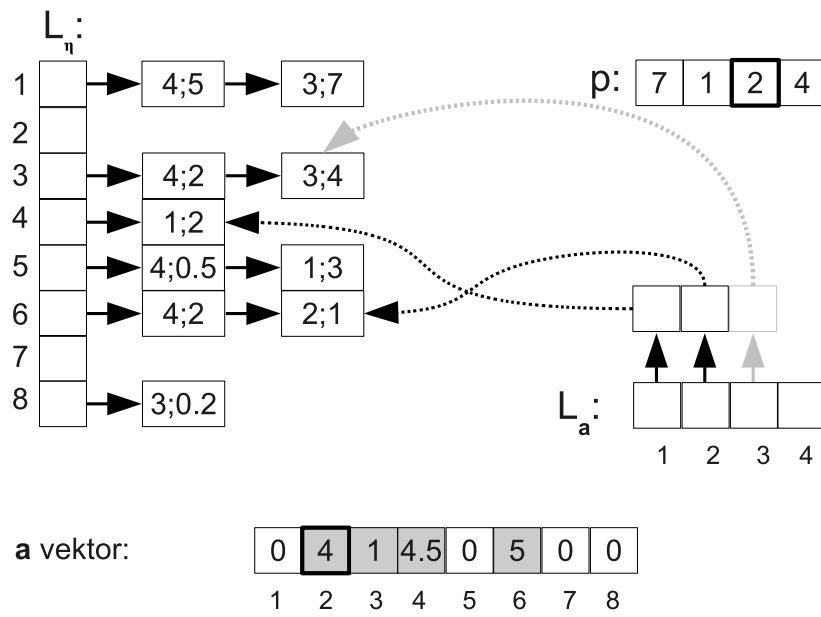


2.6. ábra. Az $L_a[\maxIndex]$ elemeinek új listákba fűzése

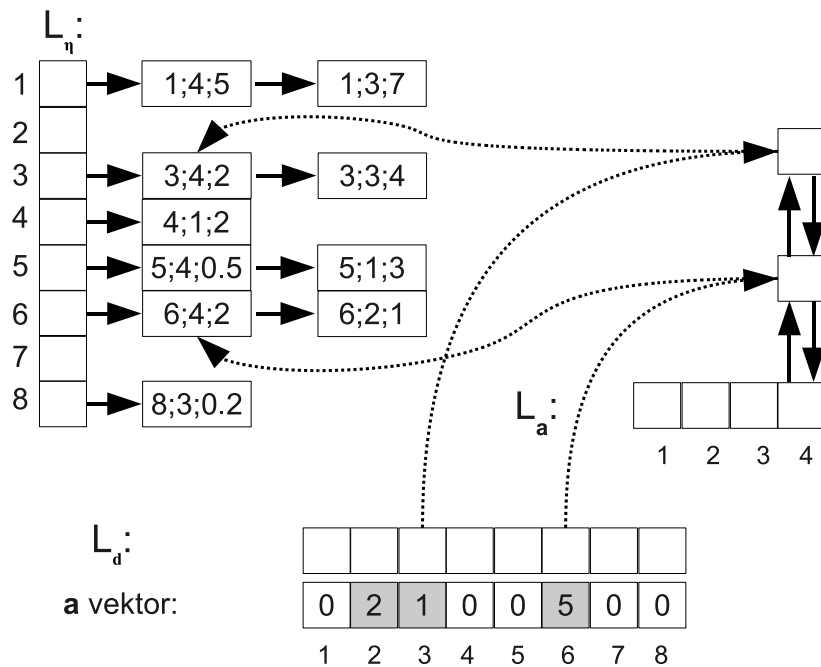


2.7. ábra. Új elemmel bővült az **a** vektor, ezért L_a -t is ki kell egészíteni

2.2. SORFOLYTONOS BÁZIS INVERZ REPREZENTÁCIÓ



2.8. ábra. Az a vektor 2. elemét átírjuk, viszont most L_a -t nem kell kiegészíteni



2.9. ábra. A sorfolytonos BTRAN algoritmusához szükséges összes adatszerkezet

2.2. SORFOLYTONOS BÁZIS INVERZ REPREZENTÁCIÓ

Bemenet: $\mathbf{a}$

Kimenet: α

Inicializálás

$maxIndex := 0$

ciklus minden $a_j \neq 0$ -ra:

ha $L_\eta[j]$ nem üres **akkor**

$(i, v) := L_\eta[j]$ lista első eleme

$L_a[i]$ -be (i, v) -re mutató pointer beszúrása

ha $maxIndex < i$ **akkor**

$maxIndex := i$

elágazás vége

elágazás vége

ciklus vége

Skaláris szorzatok

ciklus amíg $maxIndex > 0$

$d := 0$

ciklus minden $L_a[maxIndex]$ elem által mutatott (i, v) -re

 Legyen $L_\eta[j]$ az a láncolt lista, amelyben benne van (i, v)

$d := d + va_j$

ha $kov(i, v)$ létezik **akkor**

 Legyen $(i', v') := kov(i, v)$

 Szúrjunk be mutatót (i', v') -re $L_a[i']$ -be

elágazás vége

 Töröljük (i, v) -t

ciklus vége

ha $a_{p_{maxIndex}} = 0$ és $d \neq 0$ **akkor**

ha létezik $L_\eta[p_{maxIndex}]$ -ben (i, v) , ahol $i < maxIndex$ **akkor**

(i, v) -re mutató pointer beszúrása $L_a[i]$ -be

elágazás vége

elágazás vége

ha $a_{p_{maxIndex}} \neq 0$ és $d = 0$ **akkor**

L_a -ból töröljük azt a mutatót, amely olyan (i, v) -re mutat,
 amely $L_\eta[p_{maxIndex}]$ -ben van

elágazás vége

$a_{p_{maxIndex}} := d$

$maxIndex := maxIndex - 1$

ciklus amíg $maxIndex > 0$ és $L_a[maxIndex]$ üres

$maxIndex := maxIndex - 1$

ciklus vége

ciklus vége

$\alpha := \mathbf{a}$

2.10. ábra. Sorfolytonos bázis inverz reprezentációt alkalmazó BTRAN algoritmus

Implementációs megfontolások

Az alábbiakban ismertetünk néhány implementálással kapcsolatos meggondolást, amelyekkel a sorfolytonos BTRAN algoritmus még hatékonyabbá tehető.

Az L_η tömb listáit érdemes tömbként megvalósítani. Egy ilyen tömb esetén tároljuk, hogy mennyi elem van benne (azaz mekkora a mérete), illetve, hogy még mennyi elemet szűrhatunk a végére anélkül, hogy a tömböt újra kellene foglalnunk. A szimplex algoritmus során időnként szükség van arra, hogy a felhalmozódott ETM-eket eldobjuk, és ezeket kevesebb, de ezekkel ekvivalens ETM-ekkel helyettesítsük. Ezt hívjuk újra invertálásnak. Ennek oka egyrészt a numerikus stabilitás növelése, másrészt az ETM-ek száma befolyásolja az FTRAN és BTRAN algoritmusok futási idejét. Egy újrainvertáláskor tehát először L_η minden listáját üríteni kell, hogy majd újabb értékekkel töltsük fel azokat. Amennyiben a fenti tömbös megvalósítást használjuk, úgy elég csupán a tömb méretét tároló változót nullázni. Ekkor a tömböt nem szabadítjuk fel, egy ideig megőrzi a korábbi értékeket, csupán egy gyors logikai törlést hajtunk végre. Az új elemek pedig majd felülírják a régi-eket. Ennek a módszernek az az előnye, hogy a szimplex futásának elején ezek a tömbök beállnak valamekkora kapacitásra, később pedig nagyon ritkán fordul elő az, hogy új elem beszúrásakor egy ilyen tömböt újra kell foglalni. Ráadásul szükség van egy olyan műveletre, hogy a listában egy elem utáni elemet lekérdezzük, ez egy pointer növeléssel elvégezhető.

Az L_a tömbben valódi láncolt listákat kell tárolni, ugyanis mint ahogy azt láthattuk, előfordulhat, hogy egy ilyen láncolt lista valamelyik elemét törölni kell. Tömbös megvalósítás esetén ez azt jelentené, hogy a törölt elem utáni elemeket előrébb kell mozgatni, ez viszont költséges, a láncolt listából való törlés pedig $O(1)$ komplexitású. A láncolt lista bővítését, illetve az abból való törlést a C/C++ programnyelvekben memóriefoglaló, illetve memóriefelszabadító utasításokkal véghez-
zük el. Ám ha ezen utasításokat túl sűrűn használjuk, akkor az érezhetően lassítja a programot. Ezért a mi implementációnkban ezt a problémát úgy oldottuk meg, hogy mikor új elemet kell a láncolt listához fűzni, akkor ezt memóriefoglaló utasítással lefoglaltuk, és beillesztettük a listába. Ám felszabadításkor a pointert, amit a C/C++-os felszabadító utasítás kapott volna, egy gyűjtő tömb végéhez adtuk. A következő alkalommal, mikor új elemmel kellett kibővíteni az egyik láncolt listát, akkor megnéztük, hogy van-e felhasználható elem ebben a gyűjtő tömbben. Amennyiben igen, úgy ebből a tömbből választottunk ki egy pointert, egyébként pedig meghívtuk a memóriefoglaló utasítást. Ennek az eredménye az lett, hogy a program futásának a legelején lefoglalunk annyi elemnek memóriát, amennyire később szükség lehet, és a későbbi BTRAN hívásoknál ezekből az elemekből gazdálkodunk. Így a BTRAN által meghívott tényleges memóriefoglalások és felszabadítások száma elhanyagolható.

Kérdés, hogy egy adott L_η -beli (index, érték) párról hogy dönthetjük el hatékonyan, hogy ez melyik $L_\eta[i]$ listában található. Ez megoldható úgy, hogy újfent redundanciát alkalmazunk, mégpedig úgy, hogy ezt a kettést ellátjuk még egy újabb sor indexel is. A hatékony implementációhoz szükséges adatszerkezeteket teljes részletességben a (2.9) ábra szemlélteti.

2.3. Gyorsított teljes oszlopkiválasztás

Ahogy azt korábban láttuk, megéri az együtthatómátrixot sorfolytonosan is ábrázolni, hiszen így gyorsan kiszámolhatjuk a redukált költségeket. Ráadásul mivel ez a mátrix statikus, ezért ennek bővítésével sem kell foglalkozni. Az első fázisban (2.6), második fázisban pedig (1.20) segítségével ezen mátrix néhány sorának súlyozott összegzésével frissíthetjük a megváltozott redukált költségek értékeit. A következő lépésben ki kell választani azt a redukált költséget, amelyik a (1.2), illetve a (1.3) táblázatokban foglalt feltételeknek megfelel, és a legnagyobb az abszolút értéke. A frissített redukált költségek között a frissítés során kiválaszthatunk egy ilyen redukált költséget, ám figyelembe kell venni azokat is, amelyek nem változtak meg. Erre több is lehetőség van: Egy lineáris bonyolultságú szélsőérték kiválasztó algoritmussal végignézzük a redukált költségeket, majd kiválasztjuk a legmegfelelőbbet. Abban az esetben, ha iterációnként kevés elem változik, viszont nagyon sok redukált költség van, ez a módszer nem biztos, hogy kifizetődő.

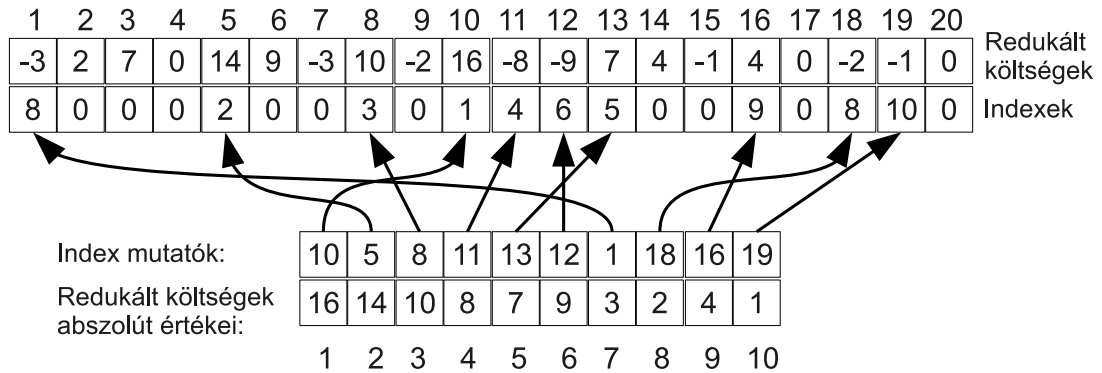
Az előzőekben felvázolt esetre használhatjuk a halom adatszerkezetet [8]. Ez egy olyan n elemű tömb, ahol az i elem nagyobb, vagy egyenlő, mint a $2i$, illetve $2i + 1$ elemek, és $1 \leq i \leq n$. Ezt az adatszerkezetet előszeretettel alkalmazzák prioritási sorok hatékony kezelésére, ugyanis ahhoz, hogy lekérdezzük a maximális elemet, nem kell az egész tömböt rendezni, elég, ha a tömb rendelkezik az előbbi tulajdonsággal. Egy halom adatszerkezet az alábbi műveletekkel rendelkezik:

- Beszúr(a): Beszúrja az a elemet a halomba, bonyolultsága: $O(\log_2(n))$
- Töröl(i): Törli az i elemet a halomból, bonyolultsága: $O(\log_2(n))$
- Módosít(i, a): Módosítja az i elemet a -ra, bonyolultsága: $O(\log_2(n))$
- Max: Visszaadja a legnagyobb elemet a halomból, bonyolultsága: $O(1)$

Ezek az algoritmusok jól ismertek, így jelen dolgozatban részletes ismertetésüktől eltekintünk.

Minden változó redukált költségéről eltároljuk, hogy javító vektorhoz tartozik-e, vagy sem. A halomban csak a javító változók redukált költségeinek abszolút értékét tároljuk. Egy redukált költség frissítésekor 4 eset lehetséges:

1. A redukált költség javító oszlophoz tartozik, és a frissítés után is ahhoz tartozik. Ekkor a Módosít műveletet kell alkalmazni a halmon, azaz a megváltozott redukált költséget a halomban is módosítani kell.
2. A redukált költség javító oszlophoz tartozik, de a frissítés után már nem. Ekkor a Töröl műveletre lesz szükségünk.
3. A redukált költség nem javító oszlophoz tartozik, viszont frissítés után már javítónak számít. Ekkor a redukált költséget be kell szűrni a halomba a Beszúr művelettel.
4. A redukált költség frissítés előtt, illetve az után sem tartozik javító vektorhoz, emiatt ekkor nem kell semmilyen halom műveletet végrehajtani.



2.11. ábra. Példa a halom indexelésére

Látható, hogy a Töröl és Módosít műveleteknek szükségük van arra, hogy a kérdéses redukált költség hányadik a halom által használt tömbben. Ez úgy oldható meg hatékonyan, hogy, minden redukált költségre tartozik egy index változó, amely ezt az információt tárolja. A halomban pedig valójában (redukált költség abszolút értéke, index mutató) ketteseket tárolunk, ahol az index mutató egy olyan memória pointer, amely az előző indexre hivatkozik. Ezt illusztrálja a (2.11) ábra. A Beszúr, Töröl és Módosít műveletek néhány halombeli elemet felcserélnek. Ekkor a felcserélt elemekben tárolt mutatókon keresztül módosítani kell az egyes indexeket. Egy ilyen módosítás $O(1)$ bonyolultságú, így végső soron a halom adminisztrációja hatékonyan megoldható.

2.4. Mérési eredmények

A bemutatott újításokat a NETLIB feladat adatbázisban szereplő feladatokon teszteltük, valamint annak nagy mérete miatt az OSA-60 nevű feladatot is felhasználtuk. A NETLIB-ben lévő feladatok közül nem mindegyiken futtattuk a programot, mivel a jelenlegi implementáció még nincs abban az állapotban, hogy minden feladatot jól meg tud oldani. Egy adott feladaton a programot adott beállítások mellett 5x futtattuk le, majd az egyes időméréseket átlagoltuk, hogy a futtatások közötti idő ingadozásokat kiküszöböljük. Minden esetben logikai bázisról indítottuk a megoldó programot, ahol minden nem bázis változó az alsó korlátról indult, valamint 30 iterációnként újra invertáltuk a bázist és összesen 20 000 iterációt engedélyeztünk. A mérések eredményei között nem közöljük a teljes szimplex algoritmus futási idejét, mivel vizsgálatunk csupán az oszlop kiválasztóra irányul, így csak annak gyorsítását mutatjuk be.

Ellenőriztük, hogy a bemutatott feladatokat a megoldó program helyesen oldja-e meg, erre a GLPK és a Coin-OR programcsomagot használtuk fel [2, 5, 1, 4].

2.4.1. A két BTRAN implementáció összehasonlítása

Megvizsgáltuk, hogy az oszlopfolytonos és sorfolytonos BTRAN algoritmusok közül melyik a hatékonyabb. Mértük az egyes futási időket, valamint számoltuk, hogy melyik mennyi szorzást végez el a program futása során. A (2.1) (2.4) táblázatok azt az esetet mutatják be, mikor a bemutatott újítások közül csak a sorfolytonos BTRAN-t engedélyeztük, tehát az első fázisú szimplex szorzó megváltozását nem használtuk ki. Látható, hogy a sorfolytonos BTRAN akár lassabb is lehet, mint az oszlopfolytonos, mivel több adminisztrációs feladatot kell ellátnia, hiába hajt végre kevesebb szorzást.

2.4.2. A sorfolytonos BTRAN és az első fázisú szimplex szorzó frissítése

Megvizsgáltuk, hogy önmagában az első fázisú szimplex szorzó (2.5) szerinti frissítése, majd a redukált költségek (2.6) szerinti számolása milyen hatással van az oszlop kiválasztó futási idejére. Ezt reprezentálja a (2.5) és (2.6) táblázat 4. oszlopja. Ezt követően bekapcsoltuk a sorfolytonos BTRAN algoritmust is, ennek a hatása látható az említett két táblázat 6. oszlopában. Ez utóbbi algoritmus hatását a táblázatok 4. oszlopához hasonlítottuk.

2.4.3. Redukált költségek kiszámítási módjai

Végezetül engedélyeztük a halom adatszerkezet használatát az oszlop kiválasztó algoritmusban, azaz az utolsó mérésben minden, a dolgozatban bemutatott újítást bekapcsoltunk. Ennek eredményét mutatják a (2.7) és (2.8) táblázatok. A CYCLE nevű feladatnál a 7,08-as gyorsulási érték annak köszönhető, hogy a halom nélküli algoritmus nem állt meg 20 000 iteráció után, a halom segítségével viszont 4979 iteráció után megtalálta az optimális megoldást. Ez annak köszönhető, hogy a halom adatszerkezet miatt nem pontosan ugyanazok az oszlopok kerülnek kiválasztásra.

2.4. MÉRÉSI EREDMÉNYEK

Feladat	Oszlopfolytonos BTRAN futási ideje [μs]	Sorfolytonos BTRAN futási ideje [μs]	Gyorsítás
25FV47	2379929	1450473	1,64
ADLITTLE	579	579	1
AGG	1182	901	1,31
AGG2	1258	836	1,50
AGG3	1472	941	1,56
BANDM	34540	30928	1,12
BEACONFD	497	397	1,25
BLEND	666	508	1,31
BNL1	318481	299334	1,06
BOEING1	15001	11409	1,31
BOEING2	1888	1809	1,04
BORE3D	1331	1137	1,17
BRANDY	10480	9589	1,09
CAPRI	23017	18465	1,25
CYCLE	1459860	439211	3,32
CZPROB	45765	55651	0,82
D2Q06C	18107581	10487130	1,73
DEGEN2	100144	69562	1,44
DEGEN3	4260184	2687882	1,58
ETAMACRO	12802	9426	1,36
FFFFFF800	40401	37744	1,07
FIT1D	12482	14236	0,88
STOCFOR	306	281	1,09
STOCFOR2	86643	39866	2,17
STOCFOR3	9410647	2711823	3,47
FIT2D	146583	197014	0,74
FORPLAN	9957	6812	1,46
GANGES	34372	18993	1,81
GFRD-PNC	3883	5515	0,70
GREENBEA	3936794	3654752	1,08
GREENBEB	3840772	3285698	1,17
GROW15	60963	35873	1,70
GROW22	115616	62055	1,86
GROW7	6980	4966	1,41
ISRAEL	10463	7088	1,48
KB2	672	627	1,07
LOTFI	2365	2040	1,16
MAROS-R7	2380411	3667316	0,65
QAP08	18269627	7362965	2,48

2.1. táblázat. A sorfolytonos BTRAN algoritmus futási ideje az első 39 feladatra

2.4. MÉRÉSI EREDMÉNYEK

Feladat	Oszlopfolytonos BTRAN BTRAN futási ideje [μ s]	Sorfolytonos BTRAN BTRAN futási ideje [μ s]	Gyorsítás
SC105	847	609	1,39
SC205	5192	3107	1,67
SCAGR25	39286	26992	1,46
SCAGR7	1468	1602	0,92
SCFXM1	8662	7984	1,08
SCFXM2	34880	31809	1,10
SCFXM3	76478	72047	1,06
SCORPION	5491	5681	0,97
SCRS8	23361	23841	0,98
SCTAP1	5941	6407	0,93
SCTAP2	39462	54722	0,72
SCTAP3	177647	165376	1,07
SCSD1	2371	1890	1,25
SCSD6	4963	2934	1,69
SCSD8	21170	11722	1,81
SHARE1B	4706	4967	0,95
SHARE2B	1261	1264	1,00
SHELL	8927	13883	0,64
SHIP04L	3683	3122	1,18
SHIP04S	3085	2808	1,10
SHIP08L	9692	10910	0,89
SHIP08S	6918	7120	0,97
SHIP12L	24060	21759	1,11
SHIP12S	17518	18149	0,97
STAIR	351261	235702	1,49
STANDMPS	3430	3540	0,97
TRUSS	4035627	1963604	2,06
WOOD1P	13136	12548	1,05
WOODW	81249	70095	1,16
OSA-60	2037215	3250980	0,63

2.2. táblázat. A sorfolytonos BTRAN algoritmus futási ideje az utolsó 30 feladatra

2.4. MÉRÉSI EREDMÉNYEK

Feladat	Oszlopfolytonos BTRAN szorzásainak száma	Sorfolytonos BTRAN szorzásainak száma	Csökkenés [%]
25FV47	405215557	37188052	9,18
ADLITTLE	38358	7673	20
AGG	242851	6433	2,65
AGG2	243381	2517	1,03
AGG3	295097	3980	1,35
BANDM	3747792	876978	23,40
BEACONFD	61779	3346	5,42
BLEND	67954		
BNL1	37353137	5544319	14,84
BOEING1	2205243	148827	6,75
BOEING2	289314	19763	6,83
BORE3D	183081	15486	8,46
BRANDY	1070538	264667	24,72
CAPRI	2347852	356280	15,17
CYCLE	265530539	4216050	1,59
CZPROB	4955822	929488	18,76
D2Q06C	2883693905	209943881	7,28
DEGEN2	12889050	1918212	14,88
DEGEN3	717012845	72909393	10,17
ETAMACRO	1560205	114302	7,33
FFFFF800	4809477	1106148	23,00
FIT1D	1037685	457868	44,12
STOCFOR	30116	1636	5,43
STOCFOR2	20801529	362952	1,74
STOCFOR3	1552637669	14208580	0,92
FIT2D	12053790	5465053	45,34
FORPLAN	1172904	190863	16,27
GANGES	4466158	111266	2,49
GFRD-PNC	322884	54529	16,89
GREENBEA	566324733	66719487	11,78
GREENBEB	552635721	59085641	10,69
GROW15	12314665	1215449	9,87
GROW22	25202536	2028558	8,05
GROW7	969716	149289	15,40
ISRAEL	2196130	158728	7,23
KB2	68867	9255	13,44
LOTFI	250173	20637	8,25
MAROS-R7	272868600	104574289	38,32
QAP08	2005517520	239995389	11,97

2.3. táblázat. A sorfolytonos BTRAN algoritmus által végrehajtott szorzások száma az első 39 feladatra

2.4. MÉRÉSI EREDMÉNYEK

Feladat	Oszlopfolytonos BTRAN szorzásainak száma	Sorfolytonos BTRAN szorzásainak száma	Csökkenés [%]
SC105	124001	8347	6,73
SC205	976762	56950	5,83
SCAGR25	4927335	668192	13,56
SCAGR7	122636	28998	23,65
SCFXM1	879931	132970	15,11
SCFXM2	3673610	560811	15,27
SCFXM3	7833289	1254485	16,01
SCORPION	550460	69766	12,67
SCRS8	2276443	441430	19,39
SCTAP1	578170	92485	16,00
SCTAP2	4059676	906946	22,34
SCTAP3	20926576	2261723	10,81
SCSD1	260306	29914	11,49
SCSD6	575871	37641	6,54
SCSD8	4286028	178976	4,18
SHARE1B	449085	103588	23,07
SHARE2B	119094	22591	18,97
SHELL	665532	166349	24,99
SHIP04L	348857	20991	6,02
SHIP04S	278627	17944	6,44
SHIP08L	1025587	124007	12,09
SHIP08S	665838	58028	8,72
SHIP12L	2624606	189388	7,22
SHIP12S	1837713	162867	8,86
STAIR	55870503	9878431	17,68
STANDMPS	294822	29363	9,96
TRUSS	815992201	47067670	5,77
WOOD1P	1990021	592929	29,80
WOODW	11053295	1514677	13,70
OSA-60	119345933	26898345	22,54

2.4. táblázat. A sorfolytonos BTRAN algoritmus által végrehajtott szorzások száma a második 30 feladatra

2.4. MÉRÉSI EREDMÉNYEK

Feladat	Eredeti idő [μ s]	Szimplex szorzó frissítése [μ s]	Gyorsulás	Sorfolytonos BTRAN [μ s]	Gyorsulás
25FV47	3972728	3635129	1,09	2476800	1,60
ADLITTLE	1864	1832	1,02	1877	0,99
AGG	3744	3479	1,08	2813	1,33
AGG2	4755	4612	1,03	4020	1,18
AGG3	5077	5130	0,99	4425	1,15
BANDM	67871	60660	1,12	53269	1,27
BEACONFD	2155	2180	0,99	1969	1,09
BLEND	1754	1706	1,03	1568	1,12
BNL1	764595	671708	1,14	662262	1,15
BOEING1	44029	58942	0,75	52065	0,85
BOEING2	5926	8676	0,68	7705	0,77
BORE3D	4597	6763	0,68	5694	0,81
BRANDY	22599	23505	0,96	24910	0,91
CAPRI	52703	37404	1,41	30725	1,72
CYCLE	4880579	4728725	1,03	3936213	1,24
CZPROB	326963	268958	1,22	253770	1,29
D2Q06C	28859124	27729649	1,04	19072428	1,51
DEGEN2	170787	182969	0,93	141382	1,21
DEGEN3	6049991	4471772	1,35	3054841	1,98
ETAMACRO	36540	29482	1,24	23807	1,53
FFFFF800	106666	74409	1,43	73790	1,45
FIT1D	240670	240381	1,00	241829	1,00
STOCFOR	1144	1215	0,94	1062	1,08
STOCFOR2	202822	228268	0,89	163995	1,24
STOCFOR3	16574047	17565795	0,94	9840406	1,68
FIT2D	24610912	25568760	0,96	25577969	0,96
FORPLAN	26348	21481	1,23	19082	1,38
GANGES	122594	113484	1,08	83696	1,46
GFRD-PNC	15223	60614	0,25	56395	0,27
GREENBEA	11838026	8685838	1,36	7057554	1,68
GREENBEB	11613164	8847727	1,31	7536872	1,54
GROW15	107064	106319	1,01	82637	1,30
GROW22	200669	198635	1,01	150756	1,33
GROW7	15621	15349	1,02	13277	1,18
ISRAEL	20424	20400	1,00	17142	1,19
KB2	1781	1707	1,04	1690	1,05
LOTFI	8280	5139	1,61	4740	1,75
MAROS-R7	6450337	4600447	1,40	3630438	1,78
QAP08	21382601	19806458	1,08	12769135	1,67

2.5. táblázat. Csak az első fázisú szimplex szorzó frissítésének, majd a sorfolytonos BTRAN algoritmus alkalmazásának hatása az oszlop kiválasztó futási idejére, az első 39 feladat esetén

2.4. MÉRÉSI EREDMÉNYEK

Feladat	Eredeti idő [μ s]	Szimplex szorzó frissítése [μ s]	Gyorsulás	Sorfolytonos BTRAN [μ s]	Gyorsulás
SC105	2058	2071	0,99	1822	1,13
SC205	9595	9480	1,01	7523	1,28
SCAGR25	65895	57403	1,15	47334	1,39
SCAGR7	3605	3817	0,94	3575	1,01
SCFXM1	24915	23648	1,05	20135	1,24
SCFXM2	100701	80879	1,25	64791	1,55
SCFXM3	220601	149283	1,48	113336	1,95
SCORPION	15138	11012	1,37	9080	1,67
SCRS8	79717	76883	1,04	75511	1,06
SCTAP1	21998	15756	1,40	14713	1,50
SCTAP2	170429	160746	1,06	142175	1,20
SCTAP3	699719	2041821	0,34	1872921	0,37
SCSD1	10791	9323	1,16	8748	1,23
SCSD6	26760	23106	1,16	22920	1,17
SCSD8	97184	103464	0,94	105867	0,92
SHARE1B	12236	10592	1,16	9636	1,27
SHARE2B	3161	3133	1,01	3350	0,94
SHELL	69512	42022	1,65	38269	1,82
SHIP04L	29820	31765	0,94	29457	1,01
SHIP04S	21730	22087	0,98	20120	1,08
SHIP08L	99821	104527	0,95	99710	1,00
SHIP08S	48988	40896	1,20	37149	1,32
SHIP12L	208980	190853	1,09	174559	1,20
SHIP12S	113283	96169	1,18	84183	1,35
STAIR	440988	218858	2,01	150382	2,93
STANDMPS	17372	18524	0,94	13751	1,26
TRUSS	8196333	8652147	0,95	6400663	1,28
WOOD1P	178188	88821	2,01	85469	2,08
WOODW	572450	474553	1,21	439489	1,30
OSA-60	115745543	109296980	1,06	120876957	0,96

2.6. táblázat. Csak az első fázisú szimplex szorzó frissítésének, majd a sorfolytonos BTRAN algoritmus alkalmazásának hatása az oszlop kiválasztó futási idejére, az utolsó 30 feladat esetén

2.4. MÉRÉSI EREDMÉNYEK

Feladat	Eredeti idő [μ s]	Szimplex szorzó frissítése, sorfolytonos BTRAN, halom [μ s]	Gyorsulás
25FV47	3972728	2463278	1,61
ADLITTLE	1864	2183	0,85
AGG	3744	2310	1,62
AGG2	4755	3259	1,46
AGG3	5077	3735	1,36
BANDM	67871	52440	1,29
BEACONFD	2155	1838	1,17
BLEND	1754	2352	0,75
BNL1	764595	601833	1,27
BOEING1	44029	44150	1,00
BOEING2	5926	3861	1,53
BORE3D	4597	3674	1,25
BRANDY	22599	19892	1,14
CAPRI	52703	19123	2,76
CYCLE	4880579	689587	7,08
CZPROB	326963	209063	1,56
D2Q06C	28859124	20213855	1,43
DEGEN2	170787	142331	1,20
DEGEN3	6049991	2977360	2,03
ETAMACRO	36540	20922	1,75
FFFFF800	106666	57270	1,86
FIT1D	240670	277751	0,87
STOCFOR	1144	1159	0,99
STOCFOR2	202822	79286	2,56
STOCFOR3	16574 047	3556426	4,66
FIT2D	24610912	30854991	0,80
FORPLAN	26348	27392	0,96
GANGES	122594	49350	2,48
GFRD-PNC	15223	90908	0,17
GREENBEA	11838026	5506801	2,15
GREENBEB	11613164	5853011	1,98
GROW15	107064	59361	1,80
GROW22	200669	191160	1,05
GROW7	15621	13030	1,20
ISRAEL	20424	10806	1,89
KB2	1781	1504	1,18
LOTFI	8280	6662	1,24
MAROS-R7	6450337	3214767	2,01
QAP08	21382601	5894575	3,63

2.7. táblázat. A halom alkalmazásának hatása az első 39 feladat oszlop kiválasztási idejére

Feladat	Eredeti idő [μs]	Szimplex szorzó frissítése, sorfolytonos BTRAN, halom [μs]	Gyorsulás
SC105	2058	2043	1,01
SC205	9595	7673	1,25
SCAGR25	65895	40238	1,64
SCAGR7	3605	3743	0,96
SCFXM1	24915	15234	1,64
SCFXM2	100701	40854	2,46
SCFXM3	220601	69893	3,16
SCORPION	15138	7065	2,14
SCRS8	79717	45689	1,74
SCTAP1	21998	10451	2,10
SCTAP2	170429	52859	3,22
SCTAP3	699719	73945	9,46
SCSD1	10791	7733	1,40
SCSD6	26760	18991	1,41
SCSD8	97184	67863	1,43
SHARE1B	12236	9693	1,26
SHARE2B	3161	2583	1,22
SHELL	69512	17431	3,99
SHIP04L	29820	13954	2,14
SHIP04S	21730	10603	2,05
SHIP08L	99821	48930	2,04
SHIP08S	48988	20862	2,35
SHIP12L	208980	83802	2,49
SHIP12S	113283	31886	3,55
STAIR	440988	64868	6,80
STANDMPS	17372	12525	1,39
TRUSS	8196333	6456077	1,27
WOOD1P	178188	89401	1,99
WOODW	572450	278046	2,06
OSA-60	115745543	103289163	1,12

2.8. táblázat. A halom alkalmazásának hatása az utolsó 30 feladat oszlop kiválasztási idejére

2.4.4. Újrainvertálások gyakorisága

Az alábbi tesztben 500 iterációnként végeztük el az újrainvertálást. Két esetet mutatunk be a (2.9) és (2.10) táblázatokon: Az első esetben az oszlopfolytonos, a második esetben a sorfolytonos BTRAN algoritmust alkalmaztuk. A táblázatokban az egyes BTRAN algoritmusok futási idejei láthatóak. A mérések alapján a sorfolytonos BTRAN többnyire jobban tűri azt, ha ritkábban invertálunk újra, bár itt is akad néhány kivétel. A kivételes eseteknél azért lassabb az új BTRAN, mert túl sok nem nullával kell dolgoznia, ez pedig nagy mértékben növeli az algoritmus adminisztrációs idejét.

Feladat	Oszlopfolytonos BTRAN [μ s]	Sorfolytanos, BTRAN [μ s]	Gyorsulás
25FV47	8060334	3379788	2,38
ADLITTLE	1015	841	1,21
AGG	1591	512	3,11
AGG2	1836	717	2,56
AGG3	2247	821	2,74
BANDM	150447	52645	2,86
BEACONFD	616	285	2,16
BLEND	988	665	1,49
BNL1	1520707	479839	3,17
BOEING1	60040	29010	2,07
BOEING2	2100	1124	1,87
BORE3D	4069	1350	2,99
BRANDY	53662	23333	2,30
CAPRI	78873	22693	3,48
CYCLE	3663705	415783	8,81
CZPROB	85925	60956	1,41
D2Q06C	46620189	14477500	3,22
DEGEN2	596678	256510	2,33
DEGEN3	10912475	3241923	3,37
ETAMACRO	26546	4991	5,32
FFFFF800	126197	31202	4,04
FIT1D	111626	129452	0,86
STOCFOR	409	231	1,77
STOCFOR2	594099	41339	14,37
STOCFOR3	49314393	1085805	45,42
FIT2D	1141319	1405958	0,81
FORPLAN	32441	3783	8,47
GANGES	72746	12831	5,66
GFRD-PNC	29733	175270	0,17
GROW15	294298	159487	1,85
GROW22	583537	478865	1,22
GROW7	73876	37440	1,97
ISRAEL	41727	24409	1,71
KB2	1092	751	1,45
LOTFI	2780	2596	1,07
MAROS-R7	4233239	1508421	2,81
QAP08	17837546	8561152	2,08

2.9. táblázat. Újrainvertálás 500 iterációként oszlopfolytonos, illetve sorfolytanos BTRAN algoritmussal

Feladat	Oszlopfolytonos BTRAN [μ s]	Sorfolytonos, BTRAN [μ s]	Gyorsulás
SC105	1961	934	2,10
SC205	25713	8971	2,87
SCAGR25	104901	31247	3,36
SCAGR7	3391	7182	0,47
SCFXM1	20810	7449	2,79
SCFXM2	54374	16309	3,33
SCFXM3	110004	26021	4,23
SCORPION	6399	3123	2,05
SCRS8	53692	21012	2,56
SCTAP1	13613	4704	2,89
SCTAP2	64440	14723	4,38
SCTAP3	98417	21028	4,68
SCSD1	5004	3013	1,66
SCSD6	13891	19724	0,70
SCSD8	201832	29949	6,74
SHARE1B	15372	8448	1,82
SHARE2B	2268	1346	1,68
SHELL	13645	4575	2,98
SHIP04L	8146	2784	2,93
SHIP04S	6995	2304	3,04
SHIP08L	26770	7696	3,48
SHIP08S	14279	4881	2,93
SHIP12L	39865	13783	2,89
SHIP12S	26242	8171	3,21
STAIR	182464	116689	1,56
STANDMPS	10914	2621	4,16
TRUSS	13183839	4417018	2,98
WOOD1P	33782	12267	2,75
WOODW	186160	58478	3,18
OSA-60	2101893	2748056	0,76

2.10. táblázat. Újrainvertálás 500 iterációnként oszlopfolytonos, illetve sorfolytonos BTRAN algoritmussal

3. fejezet

Összefoglalás

A dolgozatban bemutattuk a szimplex algoritmus alapjait, majd részletesen ismertettük a szorzat alapú bázis inverz reprezentációt, majd az oszlop kiválasztás elméleti alapjait. Kidolgoztunk három technikát, amely segítségével bizonyos esetekben gyorsíthatunk az oszlop kiválasztó modulon. A bemutatott gyorsításokat 69, széles körben ismert feladaton teszteltük. A mérések rávilágítottak arra, hogy bár mindhárom technika alkalmazása észrevehetően javít a hatékonyságon, vannak feladatok, ahol ronthatunk is.

További terveink, hogy megvizsgáljuk azokat a feladatokat, ahol nem sikerült javulást elérni, hogy a bemutatott technikákat továbbfejlesztve az implementált szimplex algoritmust még hatékonyabbá tegyük.

Irodalomjegyzék

- [1] COIN-OR Linear Program Solver. <http://www.coin-or.org/Clp/>.
- [2] GLPK (GNU Linear Programming Kit). <http://www.gnu.org/software/glpk/>.
- [3] George B. Dantzig and Wm. Orchard-Hays. The product form for the inverse in the simplex method. *Mathematical Tables and Other Aids to Computation*, 8(46):64–47, 1954.
- [4] Robin Lougee-Heimer John Forrest, David de la Nuez. *CLP User Guide*, 2004.
- [5] Andrew Makhorin. *GNU Linear Programming Kit Reference Manual*, 2010.
- [6] I. Maros. A general phase-i method in linear programming. *European Journal of Operational Research*, 23:64–77, 1986.
- [7] I. Maros. *Computational techniques of the simplex method*. Kluwer Academic Publishers, 2003.
- [8] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson. *Algoritmusok*. Műszaki Könyvkiadó, 2003.