

TDK-dolgozat

Tar Péter

Pannon Egyetem

Műszaki Informatikai Kar

Rendszer- és Számítástudományi Tanszék

Szorzat alakú bázis-inverz előállításának
korszerű technikái

Tar Péter

Mérnök Informatikus MSc szak, II. évf.

Témavezető: Dr. Maros István, egyetemi tanár

Tartalomjegyzék

1. Bevezetés	4
2. Irodalmi áttekintés	6
2.1. Lineáris programozási feladatok	7
2.1.1. Standard alak	7
2.1.2. Számítási alakok	7
2.2. A simplex módszer	9
2.2.1. Feladatok beolvasása, az MPS formátum	9
2.2.2. Előfeldolgozás és skálázás	9
2.2.3. Számítási alak előállítás	10
2.2.4. Induló bázis megtalálása	10
2.2.5. A bázis feldolgozása	11
2.2.6. Megengedettség ellenőrzése	12
2.2.7. Primál - Duál módszer kiválasztása	12
2.2.8. Pricing	12
2.2.9. A pivot elem kiválasztása	13
2.2.10. Áttérés új bázisra	13
2.2.11. Utőfeldolgozás	14
2.3. Nagyméretű feladatok jellemzői	14
2.3.1. Ritkás tárolású adatszerkezetek	14
2.4. A szorzat alakú bázis-inverz (PFI)	15
2.4.1. Szomszédos bázisok	15
2.4.2. Műveletek a szorzat alakokkal	17
2.5. Konkurens rendszerek	18
2.5.1. GLPK	18
2.5.2. IBM ILOG CPLEX	19
2.5.3. CLP	19
2.5.4. FICO XPRESS	20

3. A PanOpt keretrendszer	21
3.1. Tervezési kérdések	22
3.1.1. Adatszerkezetek nagyméretű feladatok kezelésére	24
3.1.2. A rendszer paraméterei	26
3.2. Implementációs kérdések	27
3.2.1. Kódolási konvenciók	27
3.3. A megvalósítás elemei	29
4. A szorzat alakú bázis-inverz megvalósítása	30
4.1. A bázis fej feldolgozása	31
4.2. Szingularitás kezelése	32
4.3. Triangularizálás	33
4.3.1. Iteratív kiválasztás	34
4.3.2. Láncolt listás kiválasztás	36
4.4. Nem trianguláris rész feldolgozása	38
4.5. Numerikus pontosság kezelése	39
4.6. A bázis-inverz ellenőrzése	40
4.7. Futási eredmények	40
5. Összefoglalás	42
Jelölésjegyzék	43
Irodalomjegyzék	44

Ábrák jegyzéke

3.1. A simplex algoritmus felépítése	23
3.2. A PanOpt keretrendszer modellje	25
3.3. A projekt struktúrája	28
4.1. Az aktív részmátrix trianguláris felbontása	34
4.2. Új nemnulla elemek keletkezése szorzás során	38

1. fejezet

Bevezetés

A gyakorlatban számos rendszermodellezési és optimalizálási technika létezik, amelyek alapjai lineáris programozási feladatok megoldására vezethetők vissza. A szimplex eljárás nagyméretű feladatok megoldására alkalmas, azonban annak hatékonysága nagymértékben függ a legkorszerűbb algoritmikusváltozatok és implementációs technikák használatától [1].

A piacon több szimplex alapú optimalizáló keretrendszer is található, azonban mindegyik rendelkezik valamiféle hiányossággal. Ilyen lehet, hogy külső függvénykönyvtárként nem használható, nem rendelkezik grafikus felülettel, nehezen konfigurálható, és talán a legfontosabb, hogy kutatási, továbbfejlesztési célra nem használható. Ezt a forráskód tömörsége, vagy a modularitás hiánya okozza.

Munkám során egy, a szimplex módszeren alapuló, új optimalizáló keretrendszer fejlesztésével foglalkoztam. Az új rendszer célja, hogy egyaránt kutatási és piaci felhasználóságra, valós életbeli feladatok megoldására, valamint új algoritmikus változatok kipróbálására és tesztelésére alkalmas legyen. Kiemelt fontosságú, hogy a rendszer stabilan és felhasználóbarátan működjön. A feladat elvégzéséhez alaposan áttekintem a szimplex módszer részleteit, és az egyes részek funkcionális szerepét. Megvizsgáltam, hogy milyen tervezési és implementációs kérdéseket vetnek fel a rendszerkövetelmények, és hogy azok miként valósíthatóak meg hatékonyan. A dolgozatban ezen rendszer fejlesztésének egy modulját, a szorzat alakú bázis-inverz megvalósítását részletesen ismertetem [2].

Az invertáló modul kritikus fontosságú szerepet tölt be a rendszerben, így a gyorsaság és a korszerű algoritmikus változatok használata elengedhetetlen. Az iterációk során a vektorműveletek elvégzésének minősége az inverz stabilitásától függ, tehát a numerikus pontosság fenntartása kiemelten fontos [3, 4].

További terveim közt szerepel további algoritmikus választási lehetőségek ki-

dolgozása, mérések elvégzése az algoritmikus változatok között és az invertálás párhuzamosítási lehetőségeinek vizsgálata is. Bizonyos esetekben a különböző invertálási módszerek eltérő stabilitású inverz alakot szolgáltatnak, amelyek párhuzamos kiszámításával csökkenthetjük az invertálással töltött időt, vagy növelhetjük az inverz numerikus pontosságát.

Céljam egy hosszú távú, több évre szóló csapatmunkát vetítenek elő olyan résztvevőkkel, akik egyaránt jártasak a korszerű programozási technikák használatában és megvalósításában, valamint kellő matematikai és optimalizálási ismeretekkel rendelkeznek. Ennek a csoportnak részeseként dolgoztam munkám során. Saját eredményeimnek tekintem a tervezési és implementációs megoldások kialakításában való aktív részvételt, valamint a szorzat alakú bázis-inverz és a pivotálás megvalósítását.

Munkám utolsó fejezetében egy rövid összefoglalás, jelölésjegyzék, valamint a tudományos hivatkozások jegyzéke található. A könnyebb érthetőség, és a követhetőség céljából a tárgyalásban többes szám első személyt használok.

2. fejezet

Irodalmi áttekintés

Ebben a fejezetben, a korábban publikált eredményekre és a téma szakirodalmára hivatkozva bemutatjuk a lineáris programozási feladatokat és a szimplex eljárás alapjait. Nagyobb részletességgel tárgyaljuk a dolgozat saját eredményeihez tartozó részt, a szorzat alakú bázis-inverz megvalósításához kapcsolódó irodalmi hátteret, valamint a nagyméretű feladatok sajátosságait. A fejezet utolsó szakaszában áttekintjük a konkurens rendszereket, azok előnyeit és hátrányait.

A lineáris programozási feladatok hatékony megoldása számos iparág számára kulcsfontosságú, hiszen ez a legelterjedtebb optimalizálási technológia. Számos esetben nem lineáris rendszereket is lineáris közelítésű modellekkel írnak le a kutatók, majd azoknak, mint optimalizálási feladatoknak a megoldását alkalmazzák a gyakorlatban. A lineáris programozási feladatok megoldásának két, lényegesen eltérő megközelítése létezik jelenleg az irodalomban. Az egyik a szimplex módszer[1], amellyel munkámban részletesen foglalkozunk, a másik pedig a belső pontos módszerek. Mindkét technikával nagyméretű feladatok megoldhatók, mégsem jelenthetjük ki, hogy az egyik módszertan hatékonyabb a másikonál, mert a megoldás hatékonysága (a pontosság és a megoldási idő) nagy mértékben feladatfüggő. Az elmúlt évtizedekben bebizonyosodott, hogy mindkét eljárásnak van létjogosultsága különböző feladatok megoldása esetén.

A szimplex algoritmus fontossága miatt manapság is kedvelt és izgalmas kutatási területet alkot. Az újabb algoritmikus változatok kidolgozásával és az egyre fejlettebb implementációk kialakításával évtizedről évtizedre egyre komplexebb rendszerek globális optimalizálására nyílik lehetőség.

2.1. Lineáris programozási feladatok

Ahhoz, hogy megismerhessük a simplex módszert, először össze kell foglalnunk a legfontosabb tudnivalókat a lineáris programozási (LP) feladatokról. Egy LP feladat megoldása során adott n változó és m feltétel, valamint egy célfüggvény, amelyet minimalizálni, vagy maximalizálni akarunk. Az LP feladatok több formában leírhatók. A leggyakrabban használatosak a standard alak (standard form), az általános alak (general form), és a számítási alakok (computational form #1, #2).

2.1.1. Standard alak

A standard alakban n nem negatív változó, $x_1, x_2, \dots, x_n$ (másként döntési változó) függvényében minimalizálunk (vagy maximalizálunk) egy célfüggvényt úgy, hogy a változók kielégítsenek m lineáris feltételt. A célfüggvényben a döntési változók költség együtthatókkal (c_j) számított lineáris kombinációja szerepel. A feltételeket szintén a döntési változók lineáris kombinációja adja (egy a_j^i együttható az i feltétel j változójának együtthatója), amelyek egyenlők egy konstans, jobboldal értékkel (b_i). A standard alakú LP probléma tehát:

$$\text{minimize} \quad c_1 x_1 + \dots + c_n x_n \quad (2.1)$$

$$\text{s.t. :} \quad a_1^i x_1 + \dots + a_n^i x_n = b_i, i = 1, \dots, m \quad (2.2)$$

$$x_j \geq 0, j = 1, \dots, n \quad (2.3)$$

ami ekvivalens az alábbi írásmóddal:

$$\text{minimize} \quad \mathbf{c}^T \mathbf{x} \quad (2.4)$$

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

$$\mathbf{x} \geq \mathbf{0}, \quad (2.6)$$

ahol $\mathbf{A} \in \mathbb{R}^{m \times n}$; $\mathbf{c}, \mathbf{x} \in \mathbb{R}^n$ és $\mathbf{b} \in \mathbb{R}^m$.

A célfüggvény értékét z -vel szokás jelölni, noha ez nem része a probléma definíciónak. Tehát $z = \mathbf{c}^T \mathbf{x}$.

2.1.2. Számítási alakok

A gyakorlatban az optimalizálási feladatok nem standard alakban állnak rendelkezésre, hiszen számos folyamat, vagy művelet egyenlőtlenségekkel van modellezve,

2.1. LINEÁRIS PROGRAMOZÁSI FELADATOK

valamint a változók is rendelkeznek egyéni alsó és felső korlátokkal is. Ilyen lehet például a rendelkezésre álló mennyiség, vagy egyéb triviális fizikai korlát. Ahhoz hogy mindezek mellett optimalizálást hajtsunk végre, definiálni kell, hogy milyen formára konvertáljuk a feladatot, amelyen a megoldási algoritmusunk működni tud. Ezeket az alakokat hívjuk számítási alakoknak. Több ilyen is fellelhető a szakirodalomban. Általában elmondható, hogy minden feltételhez bevezetünk egy új, logikai változót. A logikai változók meghatározása a feltételek típusától ($=, \geq, \leq, \neq$) függ. Az i feltételhez tartozó logikai változót általában y_i -vel jelöljük.

A változók és a korlátozó feltételek az alábbi típusok szerint egyértelműen csoportosíthatók:

2.1. táblázat. A változók típusai

Megengedett értékek	Típus	Elnevezés
$y_i, x_i = 0$	0	Fix
$0 \leq y_i, x_i \leq u_j$	1	Korlátos
$0 \leq y_i, x_i \leq \infty$	2	Nem negatív
$-\infty \leq y_i, x_i \leq \infty$	3	Szabad
$-\infty \leq y_i, x_i \leq 0$	4	Nem pozitív

2.2. táblázat. A feltételekhez tartozó logikai változók típusai

Logikai változó típusa	Feltétel típusa
0	Egyenlőség
1	$\leq$ és $\geq$
2	$\leq$
3	Kötetlen
4	$\geq$

A 2.1. táblázatban ismertetett típusoknál a változók korlátait nem muszáj 0-ba eltolni, kötetlenebb formák tetszőleges (véges) alsó, vagy felső korlátot megengednek. Munkánk során mi is ilyen rugalmas korlátokkal dolgozunk. Az általános alakunk tehát, amire a későbbiekben támaszkodni fogunk:

$$\text{minimize} \quad \mathbf{c}^T \mathbf{x} \quad (2.7)$$

$$\text{s.t. :} \quad \mathbf{Ax} + \mathbf{Iy} = \mathbf{b} \quad (2.8)$$

$$\text{Minden változó a 0-4 típusok valamelyikébe tartozik.} \quad (2.9)$$

2.2. A szimplex módszer

A szimplex módszer napjaink egyik leghatékonyabb eljárása LP feladatok megoldására. A felépítése moduláris, számos algoritmikus változata létezik. Munkánk során a módszer két kitüntetett moduljával foglalkoztunk, a szorzat alakú bázis-inverz, valamint a pivot elem választásának korszerű technikáival.

A szimplex módszer lényege az, hogy beolvasott feladaton egy kezdő bázisból elindulva olyan lépéseket hajtson végre, hogy szomszédos bázisokra (1 oszlopvektorban eltérő) való lépések során egyre közelebb kerüljünk az optimális megoldáshoz. Mindezt addig kell folytatnunk, ameddig el nem értük az optimális megoldást, vagy ki nem derült, hogy a probléma nem korlátos, vagy a megadott feltételek mellett nem megoldható.

2.2.1. Feladatok beolvasása, az MPS formátum

Az LP feladatok megoldása a számítástechnikai módszerek tekintetében nagy múltra tekint vissza, kialakulása a lyukkártyás időkhez köthető. Számos formátum létezik, a legrégebbi és érdekes módon a legelterjedtebb az MPS fájlformátum, amely magában hordozza a lyukkártyás idők rideg formátumát és szigorú szintaktikáját. Bár a formátum régimódi, a téma fontossága és a konzisztencia fenntartása miatt az ipar a formátumot a mai napig életben tartotta, sőt, ezen alapulnak a nagy feladatok megoldására specializálódott szoftverek is. A modernizáció során kialakult a Free MPS formátum is, amely nagyban hasonlít a az eredeti MPS formátumhoz (Fixed MPS), de kötetlenebb változóneveket és sorokat engedélyez. Sajnos a Free MPS formátum a történelmi vonatkozások miatt kevésbé elterjedt, ezért egy megoldó szoftverrendszernek a szigorú MPS formátumot feltétlenül be kell tudnia olvasni. A beolvasás mellett az MPS formátumba való írásnak is számos alkalmazása lehet, például a Branch-and-Bound eljárásoknál, a megoldás, vagy módosított feladatok (például az előfeldolgozás utáni állapot) tárolásánál.

2.2.2. Előfeldolgozás és skálázás

Az ipari méretű feladatok előállítását általában valamilyen LP modell generáló szoftverrel végzik, hiszen akár több tízezer változót és feltételt tartalmazhatnak. Ekkora méretű feladatok számos olyan egyszerűsítési lehetőséget rejtenek magukban, amelyek kiaknázásával a feladat mérete (szélsőséges esetekben akár drasztikusan is) lecsökkenthető. Ezt a folyamatot előfeldolgozásnak (Presolve) hívjuk, amelynek

során az egyszerűsítési lehetőségeket iteratíván keressük, amíg a probléma redukciója jelentős. Ez a módszer a tesztek sorrendje, valamint az előfeldolgozó paramétereire miatt heurisztikus, de általában ennek ellenére gyors és hatékony.

Skálázás (Scaling) során a feltételeket leíró mátrixot egy megfelelően megválasztott sor és oszlopvektorral megszorozva a feladatban szereplő nemnulla elemeket közelebb hozhatjuk egymáshoz. Erre elsősorban a numerikus pontosság fenntartása miatt van szükség. Például egy ipari feladat esetében, ha különböző feltételekben különböző mértékegységeket használunk ugyanahhoz a változóhoz, az az invertálás és az iterációk során nagyobb numerikus hibákhoz vezethet, mintha azonos nagyságrendekkel dolgoznánk.

2.2.3. Számítási alak előállítás

Egy beolvasott feladatban a változók és a feltételek az MPS formátumból adódóan számos formátumot ölthetnek, nincs semmiféle megkötés arra vonatkozóan, hogy az adatok standard alakban, vagy bármilyen más, jól definiált alakban legyenek. Ez a modell generáló szoftvereknek nagyobb szabadsági fokot jelent, viszont az algoritmus a feladatok alakjától függően kis mértékben változhat. A megoldás során a 2.1.2 fejezetben tárgyalt alakok alapján egy jól definiált formára kell hozni a feladatot a beolvasás után annak érdekében, hogy azok egységesen kezelhetők legyenek. Mindemellett az MPS-ből beolvasott feladatokban a logikai változók nem szerepelnek, azok típusát is meg kell határozni.

2.2.4. Induló bázis megtalálása

Ahhoz, hogy a feladat iteratív megoldásának nekikezdhessünk, szükségünk van egy kiinduló bázisra, amelyből szomszédos bázisokra való áttéréssel az algoritmus elindulhat az optimális megoldás felé. Fontos, hogy kiindulásképp olyan vektorokat határozzunk meg, amelyek teljes rangú mátrixot, így bázist alkotnak. A kiindulási bázis meghatározására általában három lényegesen eltérő módszert használnak:

Logikai bázis: A legriválisabb megoldás nem szinguláris bázis választására a logikai bázis. Több változata létezik a nembázis változók kiindulási értékének megválasztásától függően.

Trianguláris bázis: Mivel trianguláris mátrix inverzét könnyű meghatározni általában trianguláris kiindulási bázist keresünk. Empirikus tapasztalatok alapján a potenciális trianguláris kezdőbázisok közül azok a bázisok vannak a legkö-

zelebb az optimális megoldáshoz, amelyek a lehető legtöbb strukturális változót tartalmaznak. Ilyen bázis meghatározására több strukturális és numerikus eljárás is létezik, viszont szem előtt kell tartani, hogy a kezdőbázis meghatározásának nagyon gyorsnak kell lennie, hiszen ez a lépés még az inicializáció része.

Részben definiált bázis: Az iteratív megoldási folyamat közepén előfordulhat, hogy el kívánjuk menteni az aktuális állapotot, majd később innen szeretnénk folytatni a megoldást. Ez például Branch-and-Bound algoritmusok során gyakran előfordul. Az ilyen módon lementett majd beolvasott bázisváltozóknak szerepelniük kell a megoldás folytatásakor a kezdőbázisban. Amennyiben az így definiált bázisváltozók száma kisebb, mint a bázis mérete, a további változók meghatározása a fenti módszerek valamelyikével történhet.

2.2.5. A bázis feldolgozása

A bázissal explicit formában való számítás sajnos annyira műveletigényes, hogy nagyméretű feladatok megoldhatatlanná válnak. Ennek kiküszöbölése érdekében két technika terjedt el, a szorzat alakú bázis-inverz használata, vagy az LU dekompozíció. Bár az egyik módszer a bázis inverzén, a másik pedig magán a bázison alapszik, a simplex algoritmus szempontjából mindegyik kiválóan használható. Mindkét módszer napjainkban használatos, feladatfüggő, hogy melyik bizonyul hatékonyabbnak. Általában az LU alak kiszámítása költségesebb, viszont numerikusan stabilabb tulajdonságokkal rendelkezik, így ritkábban kell a faktorizációt újra elvégezni, mint a szorzat alaknál az újrainvertálást.

A szorzat alakú bázis-inverz alapja, hogy az bázis inverzét olyan ritkás mátrixok szorzataként írja le, amelyek egyetlen oszlopban térnek el az egységmátrixtól. A reprezentáció lényege, hogy a szomszédos bázisra való áttérés mindössze egy új ilyen mátrix felvételét igényli. A Szorzat alak előállításának számos algoritmikus változata létezik, amelyek egyaránt eltérhetnek a kiszámítás idejében, valamint az inverz numerikus pontosságában és stabilitásában egyaránt, általában körülbelül 50 iterációnként érdemes újrainvertálást végezni.

Az LU dekompozíció lényege, hogy egy alsó és egy felső trianguláris mátrix szorzatára dekomponálja a bázist. Mivel a vektorral való szorzás mind egy alsó, mind egy felső trianguláris mátrixszal gyorsan és hatékonyan elvégezhető, nagyméretű feladatok esetén ez az alak igen hatékony műveletvégzés szempontjából. Kiszámítása, valamint az iterációk során az LU forma fenntartása algoritmikusan

igen bonyolult és műveletigényes, viszont numerikus pontossága és stabilitása jobb. Körülbelül 150 iterációnként érdemes a faktorizációt újra elvégezni.

2.2.6. Megengedettség ellenőrzése

Egy megoldás megengedett, ha a változók olyan értékeket vesznek fel, hogy kielégítik a feladat feltételeit. Ahhoz, hogy meg tudjuk állapítani, hogy megengedett megoldásnál vagyunk-e, meg kell határoznunk a bázisváltozók értékét. A nem bázisban lévő változók mindig megengedett értéken szerepelnek, hiszen azok értéke a többitől függetlenül változtatható.

Az iterációk során az algoritmus egyik lehetséges kimenetele az, hogy megállapítja, hogy a feladatnak nincs megengedett megoldása. Erre ebben a lépésben derülhet fény, ilyen esetben az algoritmus futása véget ér.

2.2.7. Primál - Duál módszer kiválasztása

A szimplex algoritmusnak két változata létezik, a primál és a duál algoritmus. Egyértelműen nem lehet megállapítani, hogy melyik algoritmust kell használni, a feladattól függ, hogy melyik a megfelelő. Az egyes iterációs lépések között áttérhetünk egyik algoritmusról a másikra, hiszen a primál és a duál probléma, valamint a primál és a duál algoritmus is analóg egymással olyan tekintetben, hogy az iterációk során a globális optimum felé törekszik. Ha az egyik módszer konvergenciája az iterációk során lelassul, vagy leáll, érdemes lehet áttérni menet közben a másik algoritmusra.

Mindkét algoritmus esetén két fázist különböztetünk meg annak függvényében, hogy az aktuális bázismegoldás megengedett-e. Mivel a kezdőbázishoz tartozó bázismegoldás általában nem megengedett, így a megoldó algoritmus többnyire első fázisból indul. Amennyiben az iterációk során elérünk egy megengedett megoldást, átkerülünk a második fázisba. Speciális esetekben az algoritmus visszakerülhet első fázisba, ezt általában numerikus hibák okozzák.

2.2.8. Pricing

Minden iterációban egy báziscsere végrehajtásával térünk át egy új bázisra. A báziscsere során első fázisban az nem megengedettség mértékét kell csökkentenünk, második fázisban pedig a megengedettség fenntartása mellett a célfüggvény értékét javítani. Ahhoz, hogy a báziscserét végrehajthassuk, meg kell határozni a bázisból

kilépő, és a bázisba belép változót. A primál algoritmusnál a belépő, duálnál pedig kilépő változó meghatározását hívjuk pricing-nak. Amennyiben a pricing már nem tud változót kiválasztani, ha első fázisban vagyunk a problémának nincs megoldása, második fázis esetén pedig megtaláltuk az optimális megoldást.

Számos gyakorlatban is használatos pricing eljárás létezik. Az eljárásokra jellemző, hogy minél körültekintőbb kiválasztást végeznek, annál nagyobb erőforrást igényelnek a megoldás során. A pricing eljárások heurisztikusak, egyikről sem állapítható meg, hogy garantáltan képes a legjobb változó kiválasztására.

2.2.9. A pivot elem kiválasztása

A pricing során meghatározott változó oszlopában (duál esetén sorában) olyan értéket kell keresnünk, amely megfelelő generálóelem az új bázisra való áttéréshez. Ezt pivot elem választásnak nevezzük. Ezen elem kiválasztása szintén nem csak az algoritmustól függ, hanem attól is, hogy éppen melyik fázisban vagyunk. Amennyiben sikerül olyan belépésre jelölt változót találnunk, amelynek értékét úgy tudjuk módosítani (alsó korlátról felső korlátra, vagy felső korlátról alsó korlátra), hogy báziscsere nélkül megengedett megoldásnál maradjunk, akkor "bound flip" lépésről beszélünk. Egy ilyen lépés sokkal kevésbé költséges, mint a báziscsere, hiszen a bázist magát nem kell frissíteni. Amennyiben

2.2.10. Áttérés új bázisra

Miután meghatároztuk a pivot elemet és új bázisra kell térnünk, ahhoz hogy újrakezdhesük az iterációt, frissíteni kell bizonyos adatokat. A bázis tárolásának függvényében a PFI, vagy az LU alaknak megfelelő módosításokat el kell végeznünk, hogy az új iterációban már a módosult bázissal dolgozhassunk. A többi adatot (például a bázismegoldás értékét, a szimplex szorzót, stb.) is frissíthetjük, így nem kell minden iterációban újraszámolni őket, viszont a numerikus hibák felhalmozódhatnak. Az új bázisra való áttérés után újabb iteráció veszi kezdetét, ami van a bázis újrainvertálásának / faktorizálásának lépésével kezdődik, vagy a következő lépésben használatos algoritmus (primál / duál, illetve első, vagy második fázis) kiválasztásával.

2.2.11. Utófeldolgozás

Ahhoz, hogy a felhasználó számára az algoritmus értelmezhető eredményt nyújtson, az előfeldolgozás és skálázás során történt módosításokat vissza kell állítani. Erre két módszert alkalmaznak. Az egyik, hogy az előfeldolgozás során végzett lépések ellentettjét fordított sorrendben elvégezzük a probléma megoldása után, a másik pedig az, hogy az optimálisnak talált bázissal futtatjuk az algoritmust az eredeti feladaton. Ha az iterációk során azt kapjuk, hogy a megoldás nem korlátos, vagy nincs megengedett megoldás, akkor az utófeldolgozást nem kell elvégezni.

2.3. Nagyméretű feladatok jellemzői

A szimplex algoritmus, mint általában minden optimalizáló algoritmus igazán olyan esetekben érdekes, amikor nagyméretű feladatokat kell megoldani. Az implementációs megoldások, a numerikus problémák, valamint az algoritmikus változatok közötti különbségek akkor szignifikánsak, ha kellően sok iterációt kell végeznünk egy nagyobb adathalmazon. Természetesen emellett a kis feladatokat sem szabad elhanyagolni, azok helyes megoldása egy megoldó számára alapvető követelmény.

Empirikus tapasztalatok alapján elmondhatjuk, hogy a nagyméretű feladatok ritkás tulajdonságúak. A ritkáság azt jelenti, hogy az A mátrixban a nemnulla elemek száma sokkal kisebb, mint a mátrix mérete. Minél nagyobb feladatot vizsgálunk, a ritkás tulajdonság annál szembetűnőbb.

2.3.1. Ritkás tárolású adatszerkezetek

Nagyméretű, ipari feladatoknál a problémát leíró A mátrix rengeteg nulla elemet tartalmaz [5]. Ezt ritkáságnak nevezzük. Mindez abból adódik, hogy egy-egy változó általában körülbelül 10 nemnulla értékkel rendelkezik, hiszen az iparban felírt modellek nem olyan komplexek, hogy minden paramétert figyelembe véve írják le egy-egy összefüggést. Az adatok ritkás tárolása mellett a működés során úgy kell megválasztani a tárolási stratégiákat, hogy minimalizáljuk az adatok másolását, valamint a keresést.

Néhány esetben azonban, amikor tudjuk, hogy a nem nulla elemek száma egy adatszerkezetben magas, akkor nem előnyös a ritkás tárolási mód, így fel kell készülni a hagyományos (sűrű) tárolás támogatására is. A szimplex algoritmus futása során előfordulhat, hogy a ritkás vektorunkat hagyományos, sűrű vektorrá kell ala-

kítani, vagy épp fordítva. Ezeket a műveleteket scatter-nek és gather-nek hívják a szakirodalomban.

A vektor mellett a másik alapvető adatszerkezet a mátrix. A különböző algoritmusok számítási igényei úgy elégíthetők ki a leghatékonyabban, ha a mátrixban az elemeket sor-, valamint oszlopfolytonosan is tároljuk. Általában mind a sor, mind az oszlopvektorok ritkásak, és mivel a simplex algoritmus során a mai modern architektúrákon a processzor teljesítménye szűkebb keresztmetszet, mint a memória, ezért a plusz tárolás általában kifizetődő.

2.4. A szorzat alakú bázis tárolás

(Product Form of the Inverse - PFI)

Az algoritmus futása során számos alkalommal van szükség különböző vektoroknak a bázissal való transzformáltjára. Ennek az explicit kiszámítása olyannyira időigényes, hogy nagyobb méretű feladatok megoldhatatlanok ilyen megközelítéssel. Mivel a megoldás során a bázisra explicit módon nincs szükségünk, csak a vele való műveletek elvégzésére, ezért tárolhatjuk más alakban is. Erre irányuló technika a szorzat alakú bázis-inverz tárolás, ami lehetővé teszi nagyobb méretű feladatok megoldását is [2].

A bázissal a simplex algoritmus során jellemzően két műveletet végezhetünk. Ez a bázis előlről, vagy hátulról történő megszorozása egy vektorral. Ez a művelet átírható úgy, hogy a bázis helyett annak inverze szerepeljen a képletben, ez lehetővé teszi az inverz alapú bázis kezelésének lehetőségét.

2.4.1. Szomszédos bázisok

Két, $\mathbf{A}$ oszlopvektoraiból álló bázist szomszédosnak nevezünk, ha azok mindössze egy oszlopban térnek el. Jelölje a bázist $\mathbf{B} = [\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_{p-1}, \mathbf{b}_p, \mathbf{b}_{p+1}, \dots, \mathbf{b}_m]$, ahol $\mathbf{b}_i$ egy oszlopa $\mathbf{A}$ -nak, azaz $\mathbf{b}_i = \mathbf{a}_{k_i}$.

A szomszédos bázis, ha $\mathbf{b}_p$ helyére belép egy $\mathbf{a}$ vektor, így $\mathbf{B}_a = [\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_{p-1}, \mathbf{a}, \mathbf{b}_{p+1}, \dots, \mathbf{b}_m]$ lesz. Mivel $\mathbf{b}_p$ egy m dimenziós vektor, ezért felírható, hogy:

$$\mathbf{a} = \sum_{i=1}^m v_i \mathbf{b}_i \quad (2.10)$$

2.4. A SZORZAT ALAKÚ BÁZIS-INVERZ (PFI)

Mivel $\mathbf{B}$ bázis (nem szinguláris), ezért amennyiben $v_p \neq 0$:

$$\mathbf{b}_p = \frac{1}{v_p} \mathbf{a} - \sum_{i \neq p} \frac{v_i}{v_p} \mathbf{b}_i \quad (2.11)$$

ami $\mathbf{B}_a$ -val kifejezve $\mathbf{b}_p = \mathbf{B}_a \boldsymbol{\eta}$, ahol:

$$\boldsymbol{\eta} = \left[-\frac{v_1}{v_p}, \dots, -\frac{v_{p-1}}{v_p}, \frac{1}{v_p}, -\frac{v_{p+1}}{v_p}, \dots, -\frac{v_m}{v_p} \right] \quad (2.12)$$

A belépő, új vektor pozícióját (jelen esetben p -t) pivot pozíciónak szokás nevezni. Amennyiben veszünk egy egységmatrixot, majd annak i oszlopát az előbb felírt $\boldsymbol{\eta}$ vektorra cseréljük, akkor egy elemi transzformációs mátrixot (ETM) kapunk.

$$\mathbf{E} = \begin{bmatrix} 1 & & \eta_1 & & \\ & \ddots & \vdots & & \\ & & \eta^p & & \\ & & \vdots & \ddots & \\ & & \eta^m & & 1 \end{bmatrix}$$

Ennek segítségével felírhatjuk formálisan a szomszédos bázisra való áttérést, ami a szorzat alakú bázis-inverz előállításának és használatának alapját képezi:

$$\mathbf{B}_a^{-1} = \mathbf{E} \mathbf{B}^{-1} \quad (2.13)$$

Amennyiben rendelkezünk egy bázis inverzével, akkor a fenti műveletnek köszönhetően egyszerű transzformációk segítségével áttérhetünk egy másik bázis inverzére. Ahhoz, hogy egy tetszőleges bázis inverzét felírjuk, találunk kell egy olyan bázist, aminek ismerjük az inverzét. Az egységmatrix kiindulásképpen kiválóan alkalmas, hiszen annak az inverze önmaga, $\mathbf{I}^{-1} = \mathbf{I}$. Ez után $\mathbf{I}$ oszlopait egyesével lecserélhetjük a $\mathbf{b}_i, i = 1, \dots, m$ oszlopokra. Miután az összes oszlopot kicseréltük, azt kapjuk, hogy:

$$\mathbf{B}^{-1} = \mathbf{E}_m \mathbf{E}_{m-1} \dots \mathbf{E}_1 \quad (2.14)$$

$$= \mathbf{E}_i \mathbf{B}_{i-1}^{-1}, \text{ ahol } i = 1, \dots, m; \mathbf{B}_0^{-1} = \mathbf{I} \quad (2.15)$$

$\mathbf{E}_i$ a $\mathbf{b}_i$ -hez tartozó elemi transzformációs mátrix. Ez a bázis inverzének szorzat alakja, amelyet PFI (Product Form of the Inverse) néven lelhetünk fel a szakirodalomban.

2.4. A SZORZAT ALAKÚ BÁZIS-INVERZ (PFI)

A bázis fej

Ahhoz, hogy a szimplex megoldó kommunikálni tudjon a szorzat alakú bázis-inverzt előállító modullal, át kell tudnia adni a bázisvektorokat. Ez a leghatékonyabban úgy történhet, ha egy m elemű vektorban átadjuk a bázisban szereplő vektorok indexét, ezt nevezzük bázis fejnek, $\mathcal{B} = \{k_1, \dots, k_m\}$.

Ha n a változók száma, akkor egy ha $k_i \leq n$ index egy strukturális változót jelöl, amennyiben $n < k_i \leq n + m$, akkor pedig egy logikai változót. A logikai változók explicit módon nincsenek tárolva a problémában, hiszen lényegi információt nem tárolnak.

2.4.2. Műveletek a szorzat alakkal

A probléma megoldása során szükséges műveletek az $\alpha = \mathbf{B}^{-1}\mathbf{a}$ (FTRAN - Forward TRANSformation) és a $\pi^T = \mathbf{h}^T\mathbf{B}^{-1}$ (BTRAN - Backward TRANSformation) műveletek. A megoldó algoritmus során sem a bázisra, sem a bázis inverzére nincsen szükség explicit módon, csupán erre a két műveletre. A műveletek formális leírásához feltételezzük, hogy a PFI formában k darab elemi transzformációs mátrix van.

Az FTRAN műveletet kiszámítása a szorzat alakkal rekurzióval hajtható végre:

$$\alpha_0 = \mathbf{a} \quad (2.16)$$

$$\alpha_i = \mathbf{E}_i \alpha_{i-1}, \quad i = 0, \dots, k \quad (2.17)$$

$$\alpha = \alpha_k \quad (2.18)$$

A BTRAN hasonlóképpen:

$$\mathbf{h}_0^T = \mathbf{h}^T \quad (2.19)$$

$$\mathbf{h}_i^T = \mathbf{h}_{i-1}^T \mathbf{E}_{k-i+1}, \quad i = 0, \dots, k \quad (2.20)$$

$$\pi^T = \mathbf{h}_k^T \quad (2.21)$$

A szimplex algoritmus lépésről lépésre szomszédos bázisokon haladva keresi az optimális megoldást, így a bázis frissítése is elengedhetetlen. A szorzat alak talán legnagyobb előnye, hogy felépítéséből adódóan a frissítés nem más, mint egy újabb ETM felvétele, hiszen a módszer a szomszédos bázisokból való építkezésen alapul. Így a frissítés képlete egyszerűen:

$$\bar{\mathbf{B}}^{-1} = \mathbf{E}_{k+1} \mathbf{B}^{-1} = \mathbf{E}_{k+1} \mathbf{E}_k \dots \mathbf{E}_1; \quad (2.22)$$

2.5. Konkurens rendszerek

Az LP feladatok megoldására a mai napig számos rendszert kifejlesztettek már, viszont egy olyat sem találhatunk, amellyel kényelmesen és könnyedén kipróbálhatnánk új algoritmikus változatokat. Léteznek nyílt forráskódú rendszerek, mint például a COIN-OR CLP vagy a GLPK, viszont forráskódjuk módosítása azok bonyolultsága vagy a megjegyzések hiánya miatt nagyon nehéz. Számos piaci megoldó is létezik, azonban azok zárt forráskóddal rendelkeznek, így kutatási célokra nem használhatók.

A nagyméretű feladatokra alkalmazható megoldók mellett létezik számos egyszerűbb megvalósítás is. Ezek általában oktatási, vagy demonstrációs célra használatos szoftverek, amelyek többnyire csak kézzel megoldható méretű feladatok megoldására alkalmasak. Ilyen például a PHPSimplex, vagy a LiSimplex, azonban ilyen rendszerek nem alkalmasak sem algoritmusok tesztelésére, sem hatékonysági mérések végzésére.

Összességében elmondható, hogy számos LP feladatok megoldására készült szoftver fellelhető, azonban egyik sem elégíti ki az általunk fejlesztett keretrendszer követelményeit. Az alábbiakban a teljesség igénye nélkül röviden bemutatjuk a négy talán leggyakrabban használt optimalizáló rendszert.

2.5.1. GLPK

A Gnu Linear Programming Kit (GLPK) [6, 7] talán az egyik legnépszerűbb LP megoldó. Fejlesztése 2000 körül kezdődött, jelenleg a 4.45-ös verzió a legfrissebb, C nyelven íródott. A rendszerhez tartozó API igen bő és jól dokumentált, azonban sok optimalizálási algoritmus közül paraméterezéssel nem választhatunk. Különös tekintettel munkánkra, a szorzat alakú bázis-inverz megvalósítására elmondható, hogy a GLPK-ban nincs lehetőség a bázis kezelésének paraméterezésére. A dokumentáció, amely a legfrissebb változathoz tartozik, nem tesz említést a szorzat alakú bázis-inverzről és az LU dekompozícióról sem. A rendszer ilyen tekintetben fix beállításokat használ, LU dekompozícióval.

A GLPK az alábbi komponensekből áll:

- Primál és duál szimplex algoritmus
- Belső pontos primál és duál algoritmus
- Branch-and-cut eljárás
- A GNU MathProg nyelv támogatása

- Külső programok által használható API
- Önmagában futtatható LP/MIP megoldó

Összességében elmondható, hogy a GLPK oktatási célra, valamint nagyméretű feladatok megoldására is alkalmas, azonban nagyon nagy (több mint 100 változós feladatok) esetén az üzleti szoftverek jobb eredményeket adnak.

2.5.2. IBM ILOG CPLEX

Az IBM üzleti megoldó szoftvere elsősorban nagyvállalatok számára kínál üzleti optimalizációs megoldásokat. Három változata is megtalálható jelenleg a piacon, a CPLEX CP Optimizer, a CPLEX Optimization Studio és CPLEX Optimizer [8, 9]. A csomagok mindegyike tartalmaz matematikai modelleket felíró komponenseket és megoldókat egyaránt. A termék bár rendelkezik akadémiai licenszsel, természetesen zárt forráskóddal rendelkezik, így kutatási környezetben, például algoritmikus módosítások kipróbálására, vagy különböző eljárások összehasonlítására nem használható. Vállalatok számára, nagyméretű feladatok megoldására, vagy döntéstámogatásra kiválóan alkalmas.

2.5.3. CLP

A COmputational INfrastructure for Operations Research (COIN-OR) számos nyílt forráskódú optimalizáló projektet fog össze, legyen az akár modellező rendszer, akár megoldó, akár egy interfész. A COIN-OR LP (CLP) [10, 11] egy általános célú simplex megoldó. Egyaránt használható külső függvénykönyvtárként, vagy önálló alkalmazásként. Fejlesztését John Forrest nevéhez kapcsolódik, C++ nyelven íródott. Elsősorban külső függvénykönyvtárként történő használatra szánták, de önálló optimalizáló szoftverként is megállja a helyét. Kisebb feladatok megoldására, valamint akadémiai környezetben oktatási célokra alkalmas.

Mindemellett sajnálatos módon a projekt struktúrája és a forrás kódok állapota miatt új algoritmus illesztése a CLP-hez külső segítség nélkül szinte kivitelezhetetlen lenne számunkra. A moduláris felépítés és a dokumentáció hiánya miatt kutatási célokra hatékonyan nem alkalmazható. Ezt többek között az is illusztrálja, hogy a CLP napjainkban nem mutat jelentős fejlődést, a hivatalos felhasználói dokumentációja 2004-es verzió.

2.5.4. FICO XPRESS

A FICO XPRESS Optimization Suite [12, 13] a CPLEX-hez hasonlóan nagy piaci részesedésű vállalatok optimalizálására szakosodott. Egyaránt alkalmas LP, MILP és MINLP feladatok megoldására. Rendelkezik grafikus felhasználói felülettel, modellező eszközökkel, és számos más modullal. Az optimalizáló komponens konzolos felülettel, C++, Java és .NET API-val is rendelkezik, de a zárt forráskódú, így algoritmikus fejlesztések nem megvalósíthatóak. Rendelkezik viszont akadémiai partner programmal, valamint nagyméretű feladatok hatékony megoldására is képes.

3. fejezet

A PanOpt keretrendszer

Ebben a fejezetben bemutatjuk munkánk tárgyát, a PanOpt keretrendszert. Rendszerünk nevét a Pannon Egyetem Optimalizáló Rendszere (Pannon Optimizer) elnevezés rövidítéséül kapta. A projekt számára számos kutatási-fejlesztési célt tűzünk ki:

- Kutatási célra használható rendszer kiépítése:
 - Moduláris felépítés, hogy egyes algoritmikusan eltérő változatok a többi rész módosítása nélkül kipróbálhatóak legyenek.
 - Tesztek és mérési eredmények végzésének támogatása, hogy az algoritmusok összehasonlítására.
 - Letisztult, egyszerű interfészek használata a könnyű kiterjeszthetőség és az egyszerű továbbfejlesztés érdekében.
- Piaci értelemben helyt álló keretrendszer kifejlesztése:
 - Zárt forráskódú, de szubrutinként használható törzsalkalmazás.
 - Kiterjesztések támogatása publikus interfészekkel.
 - Unit tesztek használata a minőségi kódok garanciájaként.
 - Grafikus felület a problémák betöltésének, a megoldó paraméterezésének és az eredmények értékelésének támogatására.
 - Numerikus pontosság biztosítása a futás során.

Az alábbi fejezetekben elsőként bemutatjuk a keretrendszerünkben megvalósított LP probléma modellt, majd a céljaink alapján kidolgozott tervezési és implementációs kérdéseket. Ezek után bemutatjuk szorzat alakú bázis-inverz előállítására és használatára kidolgozott és megvalósított modult. A tárgyalásban bemutatjuk a

megvalósított modullal végzett futási eredményeket ?? táblázatban található teszt-feladatokon, valamint összehasonlítjuk a különböző invertálási algoritmusok által adott eredményeket.

3.1. Tervezési kérdések

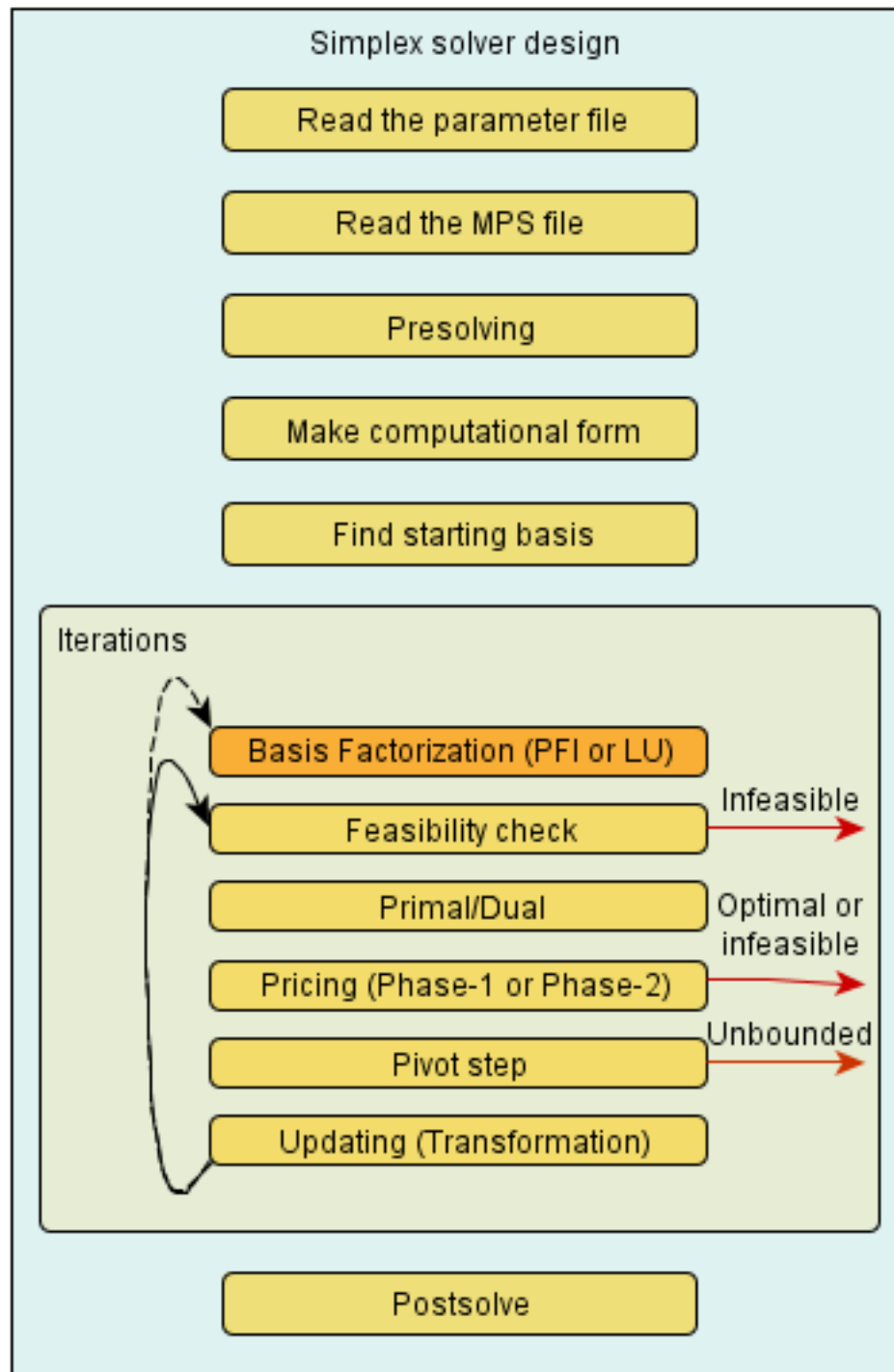
Egy optimalizáló szoftverrendszer esetében kulcsfontosságú a tervezési lépések helyessége, hiszen egy rosszul, vagy nem elég körültekintően felépített keretrendszer képtelen nagyméretű ipari feladatok hatékony megoldására. Mindemellett a helytelen tervezés azt is eredményezheti, hogy újabb modulok nehezen csatlakoztathatók, és a meglévő részek között is erős függés van, tehát a továbbfejlesztés, vagy az átalakítás is problémát jelenthet.

A tervezési lépések során a két legfontosabb tényező a szimplex algoritmus lépéseinek és a nagyméretű feladatok sajátosságainak figyelembe vétele volt. A végrehajtás szempontjából az egyes algoritmikus egységek sorrendjét kell figyelembe venni. Minden algoritmikus egység egy modult reprezentál a rendszerben, amely a többi modul megvalósításától független, célja az adott feladat minél hatékonyabb ellátása. A szimplex módszerben azonosított modulokat a 3.1. ábrán mutatjuk be.

Mint azt a 2. fejezetben tárgyaltuk, a nagyméretű feladatok jellegzetességeinek minél hatékonyabb kihasználása esszenciális egy hatékony keretrendszer kifejlesztése szempontjából. Ez az adatok ritkás tárolása mellett azt is jelenti, hogy minimalizáljuk a szoftver működése során az adatok másolását, valamint a keresést.

Ezen megfontolások alapján az elemi műveletvégzés elemeit különválasztottuk a szimplex módszer algoritmikus elemeitől, így a keretrendszer két részre osztható, egy lineáris algebrai kernelre, valamint egy szimplex komponensre, amely tartalmazza az egyes modulokat, mint például a szorzat alakú bázis-inverz előállító modult, a kezdőbázis keresést, vagy az előfeldolgozást. A kernel feladata a szimplex számára elemi műveletek biztosítása, mint például az addVector vagy az dotProduct, hiszen algoritmizálás szempontjából ezen műveleteknek csak a helyes végeredménye számít, a kiszámítás módja nem.

A rendszer felépítése a 3.2. ábrán látható. Mivel a rendszer egésze nagyon összetett, ezért az ábrán csak a legfontosabb elemek szerepelnek, azok publikus interfészével. A szimplex komponensben foglal helyet a Model osztály, ami magas szintű hozzáférést biztosít az LP feladathoz az egyes modulok számára. A Model a problémát leíró adatokon (Együtthatómátrix, célfüggvény, stb.) kívül tartalmazza a megoldást végig kísérő legfontosabb számítási adatokat is. Ilyen például a bá-



3.1. ábra. A szimplex algoritmus felépítése

ziszváltozókat leíró bázis fej, a változók értékei, illetve a változók és a feltételek típusai.

A megoldási folyamatot a Simplex osztály vezérli. Itt történik a különböző moduloknak a meghívása a megfelelő sorrendben és paraméterekkel. A felhasználó számára nem kell mást tenni, mint a megfelelő paraméter fájlal ellátott megoldónak meghívni a solve() metódusát, majd értelmezni a kapott kimenetet.

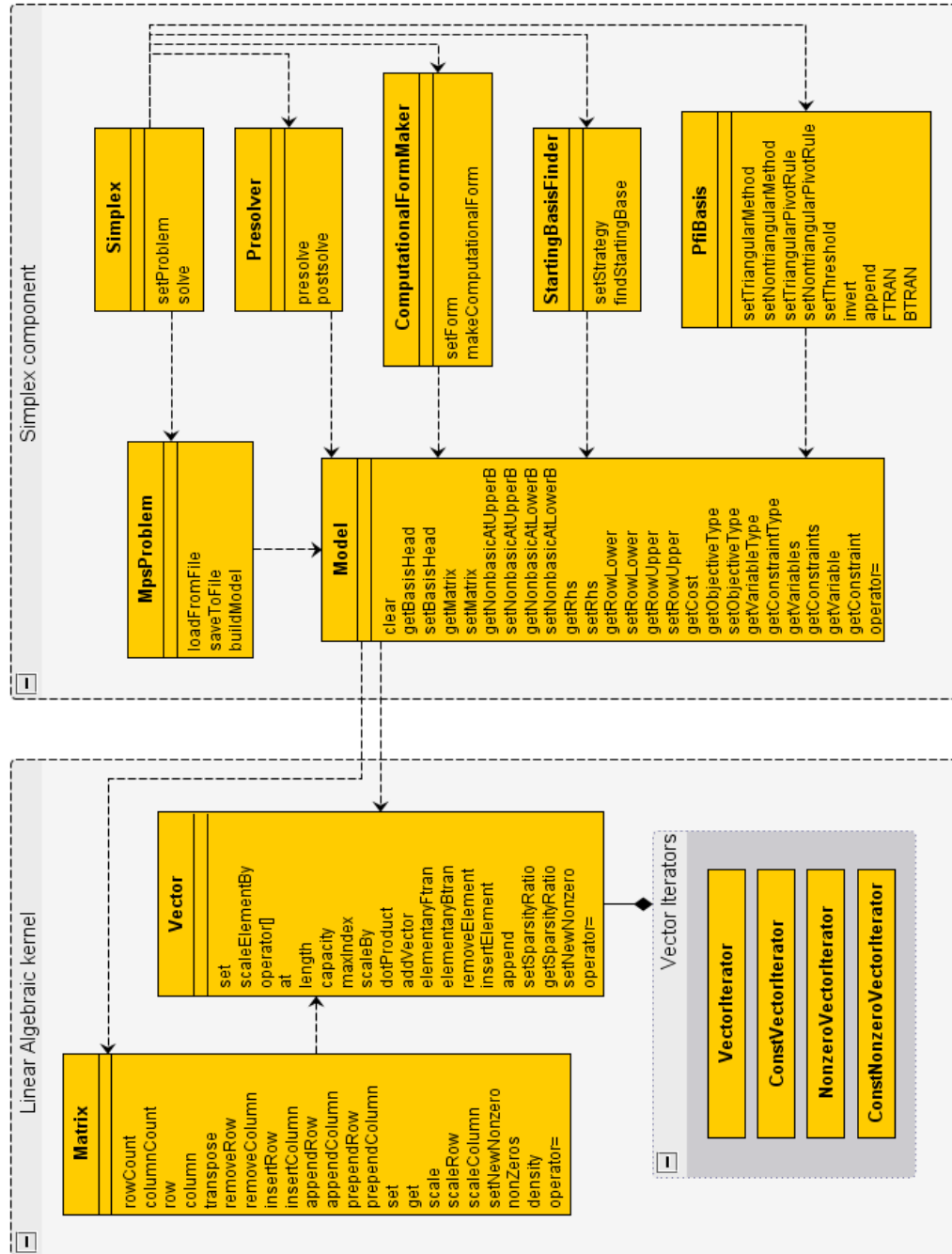
3.1.1. Adatszerkezetek nagyméretű feladatok kezelésére

A 2.3.1. fejezetben tárgyalt megfontolások alapján az adatszerkezetek kialakításánál elengedhetetlen volt a ritkás adatok speciális tárolásának támogatása. Természetesen bizonyos esetekben (például a bázis fej, vagy a redukált költségek tárolására) nem célszerű a ritkás adatszerkezetek használata, hiszen jellemzően telített adathalmazzal dolgozunk. Mivel a szimplex rendszerek alapvető adatszerkezete a vektor, valamint azt szinte minden más osztály használja, így megvalósításának minősége az egész szoftverre hatással van.

A ritkás vektorok adatszerkezete úgy épül fel, hogy a vektorok nem tárolják a nulla értékű elemeket, csak azokat amelyek nem nullák. Ezt úgy tehetjük meg, hogy minden elemhez két értéket tárolunk, az egyik az a nem nulla értékű elemünk indexe, a másik pedig a hozzá tartozó érték. Ez az adatszerkezet nagyon hatékony, ha a ritkasság mértéke nagy, viszont az elemek egy ritkás vektorban nincsenek sorba rendezve. Ennek következtében, ha le szeretnénk kérdezni egy adott indexhez tartozó értéket, akkor végig kell járnunk a regisztrált elemeket, majd ha megtaláljuk a keresett indexhez tartozó értéket, visszatérhetünk azzal, ha nem, akkor pedig nullával.

A belső tárolásnak transzparensen kell történnie a program többi egysége számára. Előfordulhat, hogy amennyiben egy adatszerkezetben a műveletvégzések során elszaporodnak a nem nulla elemek, a tárolás hatékonysága érdekében a rendszer képes dinamikusan átkonvertálni a ritkás vektorokat sűrűre. A Vector osztály paraméterezhető olyan tekintetben, hogy milyen ritkassági fok elérése esetén váltson át az adatszerkezet ritkásról sűrű tárolásra. Természetesen az interfészben elérhető műveletek mindkét tárolási módon azonos eredményt szolgáltatnak, különbség csak a háttérben meghúzódó műveletekben és idejében lehet.

A mátrix adatszerkezetet elsősorban a probléma együtthatómátrixának tárolására használjuk. Mivel a probléma a megoldás során általában konstans értékek halmaza (ritka esetben változhat), ezért a tapasztalat azt mutatja, hogy az egyes



3.2. ábra. A PanOpt keretrendszer modellje

elemek módosításával járó járulékos műveletek elvégzése sem jelent sok plusz adminisztrációs számítás.

3.1.2. A rendszer paraméterei

Ahhoz hogy a PanOpt rendszer kutatási célra, algoritmusok összehasonlítására alkalmas legyen, lehetőséget kell biztosítani a felhasználóknak, hogy kiválaszthassa a futtatás során a hogy melyik modul milyen módszer alapján működjön. Egyes modulok, mint például az előfeldolgozás, vagy a skálázás teljes egészében kikapcsolhatóak kell, hogy legyenek.

A rendszer számára a paramétereket egy paraméter fájlban tároljuk. Ez a gyakorlatban is jól bevett módszer. Mivel sok paraméter definiálható a rendszerhez, ezért a fájl alapú leírás azt is lehetővé teszi, hogy beállítások elvégzése nélkül, a felhasználók egymás között könnyedén oszthassák meg tapasztalataikat, eredményeiket. Természetesen a megoldandó problémát kivéve minden paraméternek a forráskódban van definiálva alapértelmezett érték is, így ha a paraméter fájl hiányzik, vagy hibás, akkor is lehet futtatni az algoritmust.

Az rendszer szintű paraméterek (amelyek az egész algoritmusra hatással vannak) közül a két legfontosabb az *EPSILON* érték, ami az elemek tárolására használt adatszerkezetek abszolút toleranciáját határozza meg, valamint az, hogy primál, vagy duál algoritmussal dolgozunk.

Ahhoz, hogy a rendszer felhasználói minden esetben azt az eredményt kaphassák, amit elvárnak a rendszertől különféle kimeneti szinteket kell biztosítanunk. Amennyiben egy szimplexhez nem értő felhasználója van a rendszernek, nem szabad megjeleníteni nagy mennyiségű információt az iterációk során kapott értékekről, viszont optimalizálási tudással rendelkező felhasználóknak meg kell adni a lehetőséget a modulok működésének elemzésére. Ennek támogatására elkülöníthetünk felhasználó és a fejlesztői szinteket, valamint azokon belül is több szintet a kimenet részletességét illetően. Minden szinten *debug*, *info*, *warning*, *error* típusú üzenetet közölhetünk a felhasználóval. Természetesen a részletesebb kimenetet választóknak meg kell jeleníteni minden olyan információt, amelyet alacsonyabb kimeneti szinthez társítottunk.

3.2. Implementációs kérdések

Az alábbiakban bemutatjuk az implementációra vonatkozóan kidolgozott kódolási konvenciókat, valamint a numerikus kérdéseket. A konvenciók alapját több rendszerfejlesztési, és C++ programozási ajánlás összevetésével állítottuk elő. A fejezetben emellett még bemutatjuk a megvalósított modul fordításához és futtatásához szükséges eszközöket és beállításokat.

3.2.1. Kódolási konvenciók

A fejlesztés elkezdése előtt áttekintettünk számos C++ ajánlást a forráskódok rendezését és felépítését tekintve, amelyekből egy saját fejlesztési útmutatót állítottunk össze. Ez kiterjed a forráskódok szervezésére, az elnevezésekre, valamint a tesztelési módszerekre is.

A projekt forráskód struktúrája

- Minden forrás fájt JavaDoc-stílusú annotációkkal lássunk el.
- A nem használt és az elavult fájlokat a JavaDoc-nak megfelelően archiváljunk és töröljük az *svn/trunk*-ból.
- Amennyire lehetséges kövessük ezt a struktúrát, ettől csak indokolt esetben térjünk el, de akkor is csak értelmes mappanevekkel.
- Az *src/* mappa tartalmát osszuk jól elkülöníthető, funkcionális modulokra. Amennyire lehetséges kövessük ezt a struktúrát, ettől csak indokolt esetben térjünk el, de akkor is csak értelmes mappanevekkel.

Fájl, osztály és projekt elnevezési konvenciók

- A projekt és az alprojektek elnevezése camel-case legyen, nagy kezdőbetűvel. (A camel-case annyit jelent, hogy a szavak egybeírva, de szavanként nagy kezdőbetűvel alkotnak egy sztringet. Ettől csak a legelső kezdőbetű térhet el.)
- Az osztályokat header fájlokban deklaráljuk, és forrásfájlokban definiáljuk, a fájlok neve egyezzen meg a kapcsolódó osztályokéval, elnevezésük kis kezdőbetűs camel-case legyen.
- A header fájlokra *.h* a forrásfájlokra *.cpp* kiterjesztést használjunk.
- A unit tesztek *ut_* a funkcionális tesztek *ft_* prefixet használjunk.
- Minden osztályt pontosan egy unit teszt teszteljen, amelynek neve a prefix kivételével legyen megegyező.

```
PanOpt/  
  doc/  
    Doxyfile  
  diagrams/  
  examples/  
    ProofOfConcept/  
      main.cpp  
    SolveManyProblems/  
    ...  
  input/  
    mps/  
    bad_mps/  
    netlib/  
  src/  
    linalg/  
    simplex/  
    ...  
  tests/  
  ut_PfiBasis/  
    ut_PfiBasis.cpp  
    ut_PfiBasis.h  
  ut_Vector/  
  ...
```

3.3. ábra. A projekt struktúrája

- Az elevezések ne tartalmazzanak sem a struktúrára sem egyéb szempontra redundáns információt.
- A befejezetlen, vagy javításra szoruló kódrészleteket annotáljuk *//TODO*, *//FIXME* megjegyzésekkel.

Kód formázási konvenciók

- A forráskódban angol nevek szerepeljenek.
- Minden változó, függvény, és konstans neve legyen beszédes.
- Kerüljük a rövidítések használatát.
- Csoportosítsuk a public, protected, private függvényeket és adattagokat.
- Az osztályok adattagjait és statikus adattagjait *m_* és *sm_* előtagokkal lássuk el.

Kimeneti üzenetek formázása

- A működés során *DEBUG*, *INFO*, *WARNING*, és *ERROR* típusú üzenetet jelenítsünk meg a felhasználónak.
- Legyenek paraméterezhetők a kiírások a részletesség szerint, valamint különböztessük meg a felhasználói és a fejlesztői szinteket.
- A különböző modulok üzenetei legyenek egységenként ki-, bekapcsolhatóak.

3.3. A megvalósítás elemei

Amint azt a 3.2. ábrán bemutattuk, a PanOpt rendszer két fő komponensből áll. A szorzat alakú bázis-inverz modul a Vector és a Model osztályokat használja közvetlenül. A Model segítségével érhetjük el a megoldandó LP feladatot, valamint az invertálás során használt ritkás adatszerkezetekre a Vector osztályt használjuk.

Ahhoz hogy az inverzt előállíthassuk, szükségünk van számos belső adatszerkezetre is. Ezek dinamikusan lefoglalt és felszabadított *std::vector<T>* vektorok, ahol T tetszőleges template paraméter. Ilyenre van szükség például a sorokhoz és oszlopokhoz tartozó nemnulla számlálók adminisztrálására, vagy az előállított elemi transzformációs mátrixok tárolására.

Az invertálás paramétereit *enum* adattagokban tároljuk, így egy új módszer implementálásakor csak fel kell vennünk egy újabb értéket valamelyik felsorolásból. Az így felvett új érték esetén végrehajtandó kódot pedig a felsoroláshoz tartozó *switch-case* szerkezetben kell egy új elágazásban elhelyeznünk. A kód többi része az invertáláson belüli moduláris felépítésnek köszönhetően nem igényel módosítást.

4. fejezet

A szorzat alakú bázis-inverz megvalósítása

A szorzat alakú bázis-inverz megvalósításánál két szempontot kell figyelembe venni. Az egyik, hogy maga az inverz előállítása gyors és hatékony legyen, a másik pedig az, hogy az inverz minősége a lehető legjobb legyen. Az inverz minősége alatt értjük azt, hogy a szimplex megoldó futása során az inverzzel végzett műveletek numerikus szempontból mennyire pontosak, valamint azt, hogy ezen műveletek elvégzése mennyi időt igényel.

A bázis inverzének megvalósításakor a 2.4. fejezetben bemutatott képlet hatékonyabban alkalmazhatjuk, ha minden ETM előállítása után a fennmaradó részt frissítjük a kapott ETM-el FTRAN operációk segítségével. Ennek eredményeképp a következő lépés során nincs szükség a $\mathbf{v}_i = \mathbf{B}_{i-1}^{-1} \mathbf{b}_i$ képlet alkalmazására, hiszen a következő ETM előállításához szükséges vektorok így adóttak. Fontos megjegyezni, hogy az eredeti mátrix bázishoz tartozó strukturális változóinak oszlopait le kell másolnunk ahhoz, hogy ezt a műveletet végrehajthassuk, különben a feladat $\mathbf{A}$ mátrixát módosítanánk, ami a feladat szempontjából konstans kell, hogy legyen, hiszen arra számos modulnak szüksége van.

Az inverzben a tárolás és a számítás hatékonyságát tovább növelhetjük, ha nem teljes $\mathbf{E}$ mátrixokat tárolunk, hanem egy $\mathbf{E}_i$ mátrix helyett csak a benne szereplő η vektort és annak oszlopindexét tároljuk.

4.1. A bázis fej feldolgozása

Ahogy a 2.4.1. fejezetben bemutattuk, a bázis fejjel tároljuk a bázisváltozók indexét a problémában. Az indexeken kívül azok sorrendje is fontos, a bázis fej meghatározza, hogy melyik változó hányadik helyen szerepel. Mivel az invertálás során a bázisfej módosításra kerül, az PFI alak előállításának egyik eredménye az új bázis fej. Ennek két oka lehet:

- Az egyik, hogy a bázist egy induló bázis kereső algoritmus kimeneteként kaptuk meg, ekkor a bázisoszlopok sorrendje invertálás során nagy mértékben megváltozhat a hatékony inverz előállítása érdekében. Természetesen az invertálást az egyes oszlopokon olyan sorrendben is elvégezhetnénk, ahogy a bázis fejben szerepelnek, viszont ez szinte biztosan nagyon rossz minőségű inverzt alkotna.
- A bázisfej megváltozásának másik oka a szingularitás. A szimplex iterációk során előfordulhat, hogy a numerikus hibák következtében előáll egy szinguláris bázis. Ekkor az invertálás nem végezhető el. A művelet során fel kell derítenünk a nem megfelelő pozíciókat, majd azok helyére más változót kell találnunk, hogy a bázis teljes ranggal rendelkezzen. A szingularitás kérdésével részletesebben a 4.2. fejezetben foglalkozunk.

Az invertálás során azok a 2.4.1. fejezetben tárgyaltak miatt a logikai változók nem játszanak szerepet. Ezt könnyű belátni, hiszen a logikai változókhoz tartozó oszlopvektor egy egységvektor, így az abból generált η vektor is az. Ennek következményeképp a kapott elemi transzformációs mátrix egy egységmátrix lesz, az egységmátrixszal való szorzás viszont nem változtathat meg egy másik mátrixot, így az ilyen ETM-ek elhagyhatóak a $\mathbf{B}^{-1} = \mathbf{E}_m \mathbf{E}_{m-1} \dots \mathbf{E}_1$ alakból.

A fentiek alapján megállapíthatjuk, hogy az invertálás során csak azokra az oszlopokra van szükségünk amelyek nem logikai változóhoz tartoznak. Azonban ahhoz, hogy a bázisunkban ne lépjen fel szingularitás, a logikai változókhoz tartozó sorokon nem pivotálhatunk, így ezeket a sorokat már a bázisfej beolvasásakor figyelmen kívül hagyhatjuk. Könnyen látható, hogy a fennmaradó rész is egy négyzetes mátrix, ezt aktív részmátrixnak hívjuk. Mivel a logikai változók egyetlen lehetséges pivot pozíciója az oszlopukban szereplő egyes, ezért a bázis fejben ezeket a változókat a megfelelő pozícióra kell rendeznünk. A továbbiakban az invertálás során ezekkel a változókkal nem foglalkozunk.

Ahhoz, hogy a PFI alakot előállíthassuk, le kell másolnunk az aktív részmátrixot. Ennek az az oka, hogy egy oszlopon történő pivot pozíció kiválasztásához szükségünk van annak a bázissal való szorzatára. Implementációs szempontból ez hatékonyan úgy végezhető el, ha egy oszlop kiválasztása után a fennmaradó részt megszorozzuk a újonnan kapott ETM-el, így nem dolgozhatunk az eredeti $\mathbf{A}$ mátrixon, hiszen annak oszlopaiban konstans értékek szerepelnek. Az aktív részmátrix feldolgozása során arra is szükség van, hogy az egyes sorokat hatékonyan bejárhasuk. Mivel a ritkás oszloptárolás miatt egy sor kikeresése nagyon költséges, ezért az adatokat sorfolytonosan is tárolnunk kell. A redundancia a memóriahasználatot nem növeli meg jelentősen, viszont a sorfolytonos tárolás szerint az egy sorban lévő nemnulla elemek bejárása nagyságrendekkel gyorsabb a kereséses megoldásnál, és egy LP megoldó során napjainkban egyébként is a számítási idő a szűk keresztmetszet, memóriából általában elegendő rendelkezésre áll.

A robusztus működés érdekében gondolnunk kell arra az esetre is, ha a bázis fejbe valamilyen okból hibás adat kerül. Ez megjelenhet negatív oszlopindexben, a maximális indexnél nagyobb indexben, vagy olyan esetben, ha egy oszlop kétszer, vagy többször szerepel a bázisfejben. Ekkor ezeket a pozíciókat a bázis fej beolvasásakor figyelmen kívül kell hagynunk, különben az invertáló algoritmus nagy valószínűséggel címzési hibával leállna, vagy felesleges számításokat végezne a szinguláris esetben.

4.2. Szingularitás kezelése

Az inverzáltási folyamat során ETM-ek sorozatát kell előállítanunk, azonban előfordulhat, hogy olyan oszlopok szerepelnek a bázisban, amelyek felírhatók a többi lineáris kombinációjaként. Ekkor szinguláris bázisról beszélünk. Amennyiben a bázis strukturálisan szinguláris, akkor a szingularitás megléte a numerikus hibáktól, és a pivotálás sorrendjétől független. Ez a jelenség a nemnulla elemek mintázatából felismerhető, szimplex iterációk során nem jellemző. Előfordulhat az az eset is, amikor az ETM-ek előállítása során keletkező numerikus hibák miatt egyes elemek értéke megváltozik, így szingularitás lép fel.

Amennyiben szinguláris bázist invertálunk, az invertálási folyamat végén néhány oszlop megmarad, nem tudunk rajtuk érvényes (nem nulla) pivot pozíciót választani. Az invertálási algoritmus ilyenkor leáll, a szingularitás mértéke azon oszlopok száma lesz, amelyekből nem tudunk η vektort előállítani. A fel nem töltött pivot pozíciók helyére az új bázis fejbe a nekik megfelelő logikai változók kerülnek.

4.3. Triangularizálás

Ahhoz hogy a korábbiak alapján ismertetett numerikus stabilitást fenntarthassuk, valamint az inverzben lévő nemnulla elemek száma megközelítse a bázisban lévő nemnulla elemek számát, strukturális átrendezésre, triangularizálásra van szükség. Az átrendezés a bázisban lévő sorok és oszlopok logikai felcserélését jelenti, az eredményre negatív következménnyel nem lehet.

Új nemnulla elemek akkor generálódhatnak, ha egy olyan $\mathbf{a}$ oszlopvektoron végzünk $\mathbf{E}_i \mathbf{a}$ műveletet, amelynek az $\mathbf{E}_i$ mátrix pivot pozíciójához tartozó értéke nem nulla. Ekkor az $\mathbf{E}_i$ -ben szereplő η vektor és az $\mathbf{a}$ vektor összege új nemnulla elemeket generálhat. Bizonyos esetekben a numerikus értékek függvényében nulla értékek is generálódhatnak, ez nem gyakori, azonban az implementációnak ezt az esetet is kezelnie kell.

Amennyiben egy trianguláris mátrixot invertálunk, az $\mathbf{E}_i \mathbf{a}$ művelet nem generálhat új nemnulla elemet, hiszen a már beválasztott oszlopok frissítésével nem kell foglalkoznunk. Az aktív részmátrixot három részre oszthatjuk. Egy felső trianguláris részre, egy alsó trianguláris részre, valamint a fennmaradó, nem trianguláris részre. Azért kell két különálló trianguláris részt kezelnünk, mert a sorok és a oszlopok alapján is azonosíthatunk trianguláris részeket a bázisban. Egy példa az így megkapható trianguláris felépítésre a 4.1. ábrán látható. A sor szerint triangularizálható oszlopokat és sorokat R (Row triangular) oszlopoknak és R soroknak hívjuk. Rendre az oszlopok alapján kiválasztott részeket C (Column triangular), a középső nem trianguláris részt pedig M (Middle) betűkkel jelöljük. A továbbiakban a tömör írásmód érdekében ezeket a rövidítéseket használjuk.

A trianguláris részek megtalálásához, az aktív részmátrix minden sorára (r_i az i sorhoz tartozó számláló) és oszlopára (c_i az i oszlophoz tartozó számláló) egy számlálót kell bevezetnünk, ami a nem nulla elemek számát reprezentálja. Ezeket a számlálókat az aktív részmátrix felépítésekor inicializáljuk.

A triangularizáció azon alapul, hogy a 4.1. ábrán bemutatott RR rész megtalálásához szükség van egy olyan i sorra az aktív részmátrixban, amelynek az r_i sorszámolója 1. Ellenkező esetben az RR rész trianguláris formája nem állítható elő úgy, hogy annak főátlójában nemnulla elemek szerepelnek. A szinguláris eseteket, azaz amikor a főátlóban nulla szerepel, az RR és CC blokkokba nem soroljuk be, hiszen azokon a pivotálás nem lehetséges.

A triangularizálás két módszerrel végezhető el. Az egyik egy iteratív keresésen alapul, a másik pedig összetettebb, láncolt listás adatszerkezeteken. Az alábbiakban

	R			M			C		
R	X								
		X							
	X	X	X						
M			X	X	X				
		X		X		X			
			X		X		X		
			X	X		X	X		
C	X						X	X	
		X		X	X			X	X
	X		X			X		X	X

4.1. ábra. Az aktív részmátrix trianguláris felbontása

bemutatjuk mindkét eljárás megvalósítását, valamint a ?? fejezetben bemutatjuk a különböző eljárások által elért eredményeket.

4.3.1. Iteratív kiválasztás

Az felső trianguláris (RR) rész megtalálásának elve a következő:

1. Legyen $R_{num} = 0$ (a felépített RR rész méretét jelöli).
2. Legyen $i = 0$, $found = 0$ (A $found$ változó egy elem megtalálását jelöli az iterációban).
3. Ha $r_i = 1$:
 - (a) Válasszuk ki az i sort, a benne lévő egyetlen nem nulla elem indexe legyen j .
 - (b) Pivotaljunk az (i, j) elemen.
 - (c) A j oszlop minden nemnulla elemének sorában csökkentsük a sorszámláló értékét egyel, valamint legyen $c_j = 0$.
 - (d) Legyen $R_{num} = R_{num} + 1$, $found = 1$.
4. Legyen $i = i + 1$;
5. Ha $found = 1$, és $i = r_{max}$, ahol r_{max} az aktív részmátrix sorainak száma, akkor álljon le az algoritmus.
6. Lépünk a 2.-es lépésre.

A fenti algoritmus segítségével iteratív módon addig járunk körbe a sorszámlálón, amíg tudunk sort kiválasztani az RR rész bővítésére. Amennyiben egy teljes

iteráció végrehajtódik úgy, hogy nem találunk megfelelő sorszámlálót, az algoritmus leáll. Ekkor minden érvényes (nem nulla) sorszámláló legalább 2, valamint R_{num} tárolja a felső trianguláris rész méretét.

Az alsó trianguláris (CC) rész megállapításához hasonló eljárást használhatunk. Fontos azonban, hogy míg az RR résznél egy sor megtalálásakor pivotálást hajthattunk végre, addig a CC részben csak feljegyeznünk kell a triangularizáció során előálló pivotsorrendet. Ennek az oka, hogy amint a 4.1. ábrán is látható, a fennmaradó nem triangularizálható rész oszlopaiban a CM részben is lehetnek nemnulla elemek. Amennyiben a pivotálást az MM rész feldolgozása előtt elvégeznénk, ezek az elemek miatt frissíteni kellene bizonyos oszlopokat az MM részben, amely további plusz műveletvégzést, így számítási költséget jelentene. Mindez elkerülhető, ha az CC rész feldolgozását elvégezzük, de a pivotálási műveleteket csak az MM rész feldolgozása után hajtjuk végre. Az oszlopok feldolgozásának sorrendje ekkor azok megtalálásával ellentétes, így a CC részbe feljegyzett oszlopokon ellenkező irányban kell pivot lépést végrehajtani.

Az alsó trianguláris részt azonosító algoritmus a következő:

1. Legyen $C_{num} = 0$ (a felépített CC rész méretét jelöli).
2. Legyen $j = 0$, $found = 0$ (A found változó egy elem megtalálását jelöli az iterációban).
3. Ha $c_j = 1$:
 - (a) Válasszuk ki az j oszlopot, a benne lévő egyetlen nem nulla elem indexe legyen i .
 - (b) Adjuk hozzá az (i, j) pozíciót a pivotálási láncolt lista elejéhez. (Így a pivotálás majd sorban végezhető)
 - (c) Az i sor minden nemnulla elemének sorában, ahol az oszlopszámláló pozitív, annak értékét csökkentjük egyel, valamint legyen $r_i = 0$.
 - (d) Legyen $C_{num} = C_{num} + 1$, $found = 1$.
4. Legyen $j = j + 1$;
5. Ha $found = 1$, és $j = c_{max}$, ahol c_{max} az aktív részmátrix oszlopainak száma, akkor álljon le az algoritmus.
6. Lépünk a 2.-es lépésre.

A fenti algoritmus segítségével iteratív módon addig járunk körbe az oszlopszámlálókat, amíg tudunk sort kiválasztani az CC rész bővítésére. Fontos, hogy az implementáció során az RR részt azonosítsuk először, hiszen annak invertálása

kevesebb adminisztrációs adatszerkezetet és művelete igényel. Az algoritmus végeztével minden érvényes (nem nulla) sor- és oszlopszámláló legalább 2, valamint C_{num} tárolja az alsó trianguláris rész méretét. Megjegyzésként továbbá megemlíthetjük, hogy $r_{max} = c_{max}$.

Mivel az iteratív kiválasztások során új nemnulla elemek nem generálódhatnak, ezért a ritkasság az algoritmusok során változatlan marad. Ennek következtében, mivel egy iteráción belül több sort, vagy oszlopot is kiválaszthatunk, a feladatok ritkassága jellemzően alacsony, és a ciklusváltozó léptetése igen gyors művelet, ezért általában 8-10 iteráció alatt képes az algoritmus az egyes trianguláris részek azonosítására.

4.3.2. Láncolt listás kiválasztás

Az iteratív megoldás során előfordulhat, hogy sok sor-, vagy oszlopszámlálót meg kell vizsgálnunk, mire egy megfelelő értéket találunk. Ez nagyméretű feladatok esetében akár néhány tízezer nemnulla számláló megvizsgálását is jelentheti iterációnként.

Ahhoz, hogy elkerülhessük ezt a jelenséget, vezessünk be láncolt listákat a következők szerint. Minden sor lista az azonos sorszámlálójú sorokat, valamint minden oszloplistá az azonos oszlopszámlálójú oszlopokat tartalmazza. Jelölje r_{list_i} az i nemnullát tartalmazó sorok listáját, valamint c_{list_j} a j nemnullát tartalmazó oszlopok listáját (Például r_{list_2}). Ahhoz, hogy a listákat inicializálhassuk, elsőként szükség van a maximális sor-, illetve oszlopszámláló meghatározására. A listák számának meghatározása után be kell járunk minden sorszámlálót és az adott sort hozzá kell adnunk a neki megfelelő listához. Oszlopok esetében annyiban tér el az inicializálási folyamat, hogy az oszloplistákat csak az RR rész meghatározása után inicializáljuk. Ennek az az oka, hogy az RR rész felépítésekor a beválasztott oszlopok számlálóját nullázzuk, így ezeket az oszlopok az oszloplistákban nem kell figyelembe vennünk.

Ez után az inicializáló lépés után nekiláthatunk a triangularizáló algoritmusnak, ami sorok esetében az alábbi lépéseket hajtja végre:

1. Legyen $R_{num} = 0$ (a felépített RR rész méretét jelöli).
2. Ha r_{list_1} üres az algoritmus leáll.
3. Válasszuk ki r_{list_1} első elemét, jelölje ezt a sort k , a sorban lévő egyetlen nem nulla elem indexe legyen j .
4. Pivotaljunk az (k, j) elemen.

4.3. TRIANGULARIZÁLÁS

5. A j oszlop minden nemnulla elemének sorában csökkentjük a sorszámláló értékét egyel, és a sort adjuk hozzá az csökkentett sorszámlálóhoz tartozó listához, töröljük ki r_{list_1} első elemét, valamint legyen $c_j = 0$.
6. Legyen $R_{num} = R_{num} + 1$.
7. Lépünk a 2.-es lépésre.

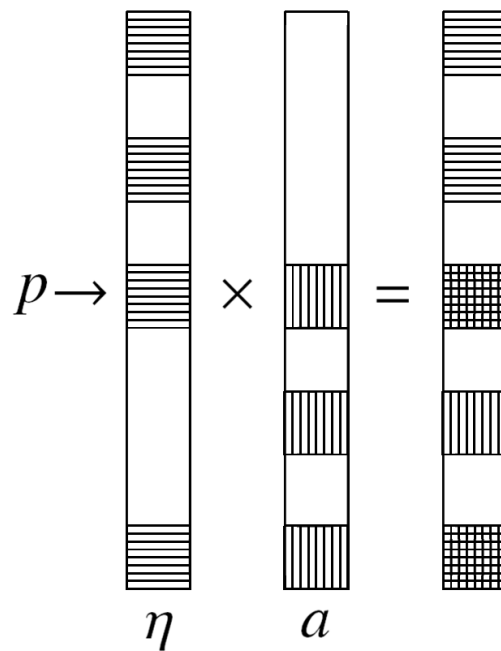
Az alsó trianguláris rész megállapítása a felső trianguláris rész azonosítása után történik. Az alsó trianguláris rész azonosítása nagyrészt a fent leírt algoritmusok elemeiből felépíthető:

1. Legyen $C_{num} = 0$ (a felépített CC rész méretét jelöli).
2. Ha c_{list_1} üres az algoritmus leáll.
3. Válasszuk ki c_{list_1} első elemét, jelölje ezt az oszlopot k , az oszlopban lévő egyetlen nem nulla elem indexe legyen j .
4. Adjuk hozzá az (j, k) pozíciót a pivotálási láncolt lista elejéhez. (Így a pivotálás majd sorban végezhető)
5. A j sor minden nemnulla elemének oszlopában csökkentjük az oszlopszámláló értékét egyel, és az oszlopot adjuk hozzá az csökkentett oszlopszámlálóhoz tartozó listához, töröljük ki c_{list_1} első elemét, valamint legyen $r_j = 0$.
6. Legyen $C_{num} = C_{num} + 1$.
7. Lépünk a 2.-es lépésre.

Az algoritmus végén C_{num} változó tárolja a C részbe választott oszlopok számát. Mivel az algoritmusok során csak az r_{list_1} és c_{list_1} listák érdekesek, ezért a magasabb számlálókhoz tartozó listákból a számlálók csökkentése során nem szükséges a sorok, illetve oszlopok eltávolítása. Bár ezzel a megfontolással az 1-nél magasabb számlálójú listák érvényessége megváltozik olyan tekintetben, hogy a lista által jelölt vektorokban a nemnullák száma kisebb, vagy egyenlő lehet, mint a listához rendelt szám. Ennek a megfontolásnak a segítségével számítási teljesítményt spórolhatunk meg úgy, hogy az algoritmus helyes végrehajtását nem befolyásoljuk. Az algoritmusok végrehajtása után felszabadíthatjuk a listák tárolására használt adat-szerkezeteket, azokra a továbbiakban nincs szükség.

4.4. Nem trianguláris rész feldolgozása

A középső, nem trianguláris rész (MM) feldolgozására számos módszert alkalmazhatunk. Ennek a résznek az invertálása jelentős fontossággal bír az invertálás és az inverzzel való műveletvégzés szempontjából, hiszen az egyes η vektorok előállítása után a még be nem választott oszlopokat meg kell szoroznunk az újonnan kapott ETM-el. Amennyiben van olyan oszlop, amely megváltozik a szorzás során (az ETM-ben szereplő η vektorhoz tartozó együttható a szorzás során nem nulla), akkor új nem nulla elemek keletkezhetnek az oszlopban. Ezt a jelenséget feltöltődés (fill-in) nevezzük. Az új nemnulla elemek keletkezését a 4.2. ábrán mutatjuk be. Az $\mathbf{E}\mathbf{a}$ művelet eredményeként, ha $\mathbf{E}$ -ben az η vektor pozíciója p , a kapott vektorban minden olyan pozíción nemnulla elem fog szerepelni, hol az η vagy az $\mathbf{a}$ vektorban nemnulla volt.



4.2. ábra. Új nemnulla elemek keletkezése szorzás során

A nem trianguláris rész invertálását több módszer alapján is elvégezhetjük. Az inverzben lévő értékek numerikus pontossága, valamint a feltöltődés mértéke nagyan függ a kiválasztott módszertől. Az alkalmazható eljárások közül felsorolunk néhányat:

- Keresünk egy tetszőleges nem nulla pivot elemet.
- Keresünk egy feltételezhetően jó pivot elemet a sor- és oszlopszámlálók alapján.

- Blokk trianguláris formát alakítunk ki, majd a főátlóban lévő elemeket választjuk pivot elemeknek. [5, 14, 15]
- Blokk trianguláris formát alakítunk ki, majd a blokkokban "threshold pivoting" eljárással dolgozunk.
- Kényszerített triangularizációt használunk. [16, 17]

Munkánk során a tetszőleges pivot elem választást, valamint az említett két blokk trianguláris eljárást valósítottuk meg.

4.5. Numerikus pontosság kezelése

A szimplex algoritmus futása során több olyan műveletet is kell végeznünk, ami numerikus hibához vezethet. Pontosnak sajnos csak azokat az értékeket tekinthetjük, amelyek a bemeneti fájlból beolvasásra kerültek. Minden transzformált érték magában hordozhatja a numerikus pontatlanságot [3]. A legkritikusabb műveletek azonban az invertáláshoz és a pivot elem kiválasztásához is szorosan kapcsolódnak. Amennyiben a szorzat alakú bázis-inverzben olyan η vektorokat tárolunk, amelyek értékei pontatlanok (ez olyankor fordulhat elő, ha az η vektort előállító pivot elem nagyon nagy, vagy nagyon kicsi), akkor az FTRAN és BTRAN műveletek eredményeiben is megjelenik majd a numerikus hiba. Több ilyen egymásutánja esetén a pontatlanság egyre növekszik.

Ahhoz, hogy kezelni tudjuk a numerikus hibát abszolút, vagy relatív toleranciával kell dolgoznunk. Abszolút tolerancia esetén két értéket azonosnak tekintünk, ha azok egy ϵ értéknél kisebb eltéréssel rendelkeznek, és ϵ egy előre beállított konstans paraméter. Relatív hibakezelés esetén az ϵ érték nagysága függ az adott vektor normájának, vagy valamilyen más értéknek a nagyságától, a hiba mértéke annak függvényében történik [4].

A ritkás tárolásnak köszönhetően különös fontosságú, hogy azok a nulla közeli elemek, amelyek az abszolút, vagy relatív hibakorlátként kiszámított ϵ értéknél kisebbek, ne kerüljenek tárolásra. Ezzel egyrészt csökkenthetjük a számítás és a memóriaigényt, másrészt pedig a hiba elhagyásával megakadályozzuk annak halmozódását. Erre a vektor implementációját alkotó osztály nyújt támogatást.

A numerikus kérdések a pivot elem választásánál is nagy hangsúlyt kapnak. Rossz pivot elemnek kell tekintenünk a kedvezőtlen numerikus tulajdonságokkal rendelkező pivot elemet. A numerikus hibák drasztikus növekedésének elkerülése

végezt a gyakorlatban a legtöbb esetben érdemesebb egy, az optimum szempontjából kevésbé kedvező, viszont sokkal jobb numerikus kondícióval rendelkező pivot elemet választani, hogy a numerikus stabilitást fenntarthassuk.

4.6. A bázis-inverz ellenőrzése

A bázis-inverz helyességére három különböző módszert alkalmazhatunk. Mivel $\mathbf{B}\mathbf{B}^{-1} = \mathbf{I}$, ezért:

- A bázishoz tartozó oszlopvektorokkal végzett FTRAN művelet eredménye egy egységvektor kell hogy legyen. Az összes ilyen vektor együttesen az egységmátrixot kell hogy adja.
- A bázis oszlopokhoz tartozó értékekből képzett sorvektorok végzett BTRAN művelet eredménye egy transzponált egységvektor kell hogy legyen. Az összes ilyen vektor együttesen az egységmátrixot kell hogy adja.

A harmadik lehetőség, amivel ellenőrzést végeztünk, azon alapul, hogy a bázisvektorokhoz tartozó redukált költségek értéke mindig nulla kell, hogy legyen. Ennek képlete: $c_j - \mathbf{c}_B^T \mathbf{B}^{-1} \mathbf{a}_j$, ahol $\mathbf{c}_B^T$ a célfüggvény bázishoz tartozó részét jelöli.

A futtatások alapján megállapíthatjuk, hogy sikerült helyes inverz alakot előállítanunk, ezen tesztek sikeres eredménnyel lefutottak a netlib feladatain.

4.7. Futási eredmények

A megvalósított eljárásokat generált nem trianguláris bázisokkal teszteltük a netlib feladatain. A triangularizációt nem használó esetekben a futási idő és a feltöltődés is jelentős mértékű volt, az inverz sokszor elérte a bázisban szereplő nemnullák számának többszörösét.

Az iterációkon alapuló és a láncolt listás triangularizációval azt tapasztaltuk, hogy mindkét eljárás nagyjából azonos eredményt ad. Az iteratív eljárás során általában maximum 8-10 iteráció alatt sikerült triangularizálni a trianguláris részeket. A netlib legnagyobb feladatán, a stocfor3-on egy teljes trianguláris bázis invertálása az iteratív eljárással 0.1, míg a listás eljárással 0.12 másodperc volt többszöri futtatás átlagos idejét számítva. Összességében megállapítható, hogy bár az egyik eljárás egyszerűsége kompenzálja a másik hatékonyságát, az iterálási idő nagyjából megegyezik a láncolt listák által igényelt plusz adminisztrációs műveletek végrehajtásával.

4.7. FUTÁSI EREDMÉNYEK

Nem trianguláris bázison a tetszőleges pivot elem választó eljárás bár gyors, numerikusan nagyon pontatlan eredményt szolgáltat. A bázis egyes numerikusan érdekesebb feladatnál (agg, pivot87) annyira rossz eredményt szolgáltatott, hogy használhatatlan lenne a szimplex algoritmus számára. Az is előfordult viszont, hogy mindössze néhány százalékos feltöltődés mellett invertálta bizonyos feladatokra a generált bázis, és minden teszt sikeres eredménnyel lefutott.

A blokk triangularizáció már sokkal jobb eredményeket mutatott. Az implementációval sikerült elérni, hogy minden feladatra a tesztek kielégítő PFI alakot kapjunk, valamint az invertálás 0.8 másodperc alatt végzett a legnagyobb feladaton is úgy, hogy a nem triangularizálható rész mérete több mint 12000 sort tartalmazott.

Jelenlegi implementációk legnagyobb eredményének azt tekinthetjük, hogy megalkottunk egy olyan paraméterezhető invertáló modult, amely a kutatók, fejlesztők számára egyszerű felületet biztosít új ötleteik, algoritmusaik kipróbálására, azonnali mérésére és összehasonlítására más eljárásokkal.

5. fejezet

Összefoglalás

A dolgozat készítése során részt vettem egy olyan optimalizáló keretrendszer kidolgozásában, amely kutatási és piaci célokra egyaránt hatékonyan felhasználható. Segítségével nagyméretű, valós életből vett feladatokat oldhatunk meg, melyhez paraméterezhető felületet és stabil, robosztus működést biztosít. A megvalósítás során törekedtünk egy megfelelő interfész kidolgozására, amellyel később könnyedén illeszthetünk tetszőleges grafikus felhasználói felületet a meglévő keretrendszerhez.

Szakmai tevékenységem kezdeti szakaszán elengedhetetlen volt, hogy részletekbe menően áttekintsük a szimplex módszer működését és strukturális felépítését. Megismerkedtem a tradicionális szimplex működésével és a teljesítményét javító fejlett eljárásokkal egyaránt.

Megvizsgáltam, hogy milyen tervezési és implementációs kérdéseket vet fel a követelményeknek megfelelő megvalósítás, és hogy azok miként valósíthatóak meg hatékonyan. Munkám a fejlesztésben a szorzat alakú bázis-inverz korszerű megvalósítását. A modul szimplexben betöltött szerepe miatt nagy súllyal jelenik meg a felhasznált erőforrások tekintetében a teljes futás során, ezért a gyorsaságot, a numerikus pontosságot és a lehető legjobb megoldások használatát is szem előtt tartottam. A megvalósítás hatékonyságát a netlib feladatain végzett mérési eredményekkel igazoltam.

Hosszú távú céljaim között szerepel, hogy további összehasonlításokat végezzek további invertálási technikák között, valamint hogy a megvalósított tesztkörnyezet segítségével újabb technikákat dolgozzak ki. Mindemellett részt kívánok venni az optimalizáló szoftverrendszer további fejlesztésében, hogy elkészülhessen egy olyan program, amely versenyképes a jelenlegi piacon található termékekkel.

Jelölésjegyzék

η	Szomszédos bázisra való áttérést leíró vektor, 19. o.
$\mathcal{B}$	Bázis fej, 20. o.
B	Bázis, 19. o.
E	Elemi transzformációs mátrix, 20. o.
a_j^i	Változó együtthatója, 7. o.
b_i	Feltételhez tartozó jobboldal együttható, 7. o.
c_i	Egy oszlopvektor nem nulla elemeinek száma, 36. o.
c_j	Változóhoz tartozó költség együttható, 7. o.
k_i	A bázis fej i eleme, 20. o.
m	Feltételek (sorok) száma, 7. o.
n	Változók (oszlopok) száma, 7. o.
r_i	Egy sorvektor nem nulla elemeinek száma, 36. o.
x_j	Döntési változó, 7. o.
y_i	Az i feltételhez tartozó logikai változó, 8. o.
z	A célfüggvény értéke., 7. o.
BTRAN	Vektor szorzása a bázis inverzával hátulról, 21. o.
ETM	Elemi transzformációs mátrix, 20. o.
FTRAN	Vektor szorzása a bázis inverzával előlről, 21. o.
PFI	A bázis inverzének szorzat alakja, 20. o.

Irodalomjegyzék

- [1] I. Maros. *Computational techniques of the simplex method*. Kluwer Academic Publishers, 2003.
- [2] 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.
- [3] Hartung Ferenc. *Bevezetés a numerikus analízisbe*. Veszprémi Egyetemi Kiadó, Veszprém, 2004.
- [4] Virágh János. *Numerikus Matematika*. JATEPress, Szeged, 1997.
- [5] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford Science Publications, 1986.
- [6] Glpk (gnu linear programming kit). <http://www.gnu.org/software/glpk/>.
- [7] Andrew Makhorin. *GNU Linear Programming Kit Reference Manual*, 2010.
- [8] Ibm ilog cplex optimization studio. <http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/>.
- [9] *User's Manual for CPLEX*, 2009.
- [10] Coin-or linear program solver. <http://www.coin-or.org/Clp/>.
- [11] Robin Lougee-Heimer John Forrest, David de la Nuez. *CLP User Guide*, 2004.
- [12] Fico xpress optimization suite. <http://www.fico.com/en/Products/DMTools/Pages/FICO-Xpress-Optimization-Suite.aspx>.
- [13] *Xpress-Optimizer Reference manual*, 2009.

- [14] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):140–160, 1972.
- [15] Alex Pothén and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, 1990.
- [16] E. Hellerman and D.C. Rarick. The partitioned preassigned pivot procedure (p4). *Sparse Matrices and Their Applications*, pages 67–76, 1972.
- [17] A. M. Erisman, R. G. Grimes, J. G. Lewis, and Jr. W. G. Poole. A structurally stable modification of hellerman-rarick’s p4 algorithm for reordering unsymmetric sparse matrices. *SIAM Journal on Numerical Analysis*, 22:369–385, 1985.