Tartalomjegyzék

Oldal

1. Bevezetés

Az operációkutatás egy olyan tudományág, melynek feladata, hogy a gyakorlati élet különböző problémacsoportjaihoz olyan optimumszámítási modelleket generáljon, melyek az adott problémacsoportokat leírják, valamint ezekhez a konstruált modellekhez olyan eljárásokat dolgozzon ki, amik az optimális megoldást meghatározzák.

Egy operációkutatatási probléma megoldása során először a gyakorlati probléma egy matematikai vagy matematikai formalizmussal megadott modelljét kell tehát megalkotni, azután az abból adódó feladatokat megoldani. A modellekben a döntés-előkészítés folyamán optimalizálás történik, azaz vagy a ráfordítási költségeket kell minimalizálni vagy a nyereséget maximalizálni a feltételek teljesülése mellett.

Ezeket a modelleket többféle szempont szerint lehet osztályozni. Többek között beszélhetünk olyanokról, melyekben a feltételek lineáris egyenlőségek és egyenlőtlenségek formájában adottak. Ekkor a célfüggvény szerint megkülönböztetünk lineáris és hiperbolikus programozási modelleket.

Lineáris programozási modellnek vagy feladatnak nevezzük azokat a szélsőérték feladatokat, melyek a feltételek mellett, keresik egy szintén *lineáris* célfüggvény maximumát vagy minimumát. Azon modelleket melyek célfüggvényei *nem-lineáris*ak, hiperbolikus feladatoknak nevezzük.

Lineáris és hiperbolikus feladatok megoldására különböző technikák léteznek. Ezekben a módszerekben nagyméretű mátrixok kezelésére van szükség, melyek gyakran a sok zérus elem mellett csak kevés értékes elemet tartalmaznak. Ezeket a kétdimenziós mátrixok általánosan m*n alakúak, ahol m a sorok száma, n pedig az oszlopok száma. A mátrix elemeit pedig általában kétdimenziós tömb formájában adjuk meg. Azokat a mátrixokat, amelyek sok zérus elemet tartalmaznak **ritka mátrixok**nak nevezzük. Ha alkalmazásunkban direkt módon tároljuk ezeket a struktúrákat és úgy dolgozunk velük, akkor nagyon sok processzor időt emésztünk föl, valamint helytakarékosság szempontjából is kissé pazarló. A feladatok megoldása során a tárigény csökkenthető azáltal, hogy a mátrix(ok) zéró elemeit nem tároljuk. Ennek kiküszöbölésére szolgálnak a különböző tárolási sémák, melyek közül néhányat

részletezni is fogok diplomamunkámban. Ezen tárolási módokkal kizárólag a nem zérus elemeket és azok helyét tároljuk.

Cél az, hogy elkerüljük a nulla elemekkel történő műveleteket.

A ritka mátrixokat két csoportba sorolhatjuk. Beszélhetünk reguláris és irreguláris mátrixokról:

- **Reguláris struktúrájú ritka mátrixok:** Olyan mátrix, melynek nem-zérus elemeit könnyen kódolhatjuk és kiértékelhetjük.
- **Irreguláris ritka mátrixok:** Ezen mátrixok nem nulla elmei úgy helyezkednek el, hogy azokat nem könnyű kódolni és feldolgozni. Ezek általában expliciten tárolódnak adatstruktúrákban, amik elősegítik az alap operátorok végrehajtását és az elérés megvalósítását.

2. Ritka vektorok

2.1 Ritka vektorok tárolása

A ritka vektorokat tárolhatjuk teljes hosszukon, ekkor ezt a tárolási módot explicit formának nevezzük. Bár ez a tárolási mód eléggé pazarló, mégis van néhány előnye. Ilyen például az egyszerűsége; közvetlenül és gyorsan el tudjuk érni az egyes elemeket a tömb indexeken keresztül, és a tárolási követelményeket is előre ismerjük. Ezen kívül, a műveleteket is könnyű megtervezni. Azonban ha nagy méretű a vektorunk és nagy számban tartalmaz zéró elemet, akkor jobb megoldás más tárolási formát alkalmazni.

Hatékonyabb megoldás, ha csak a vektor nem-nulla elemeit tároljuk. Mindezt általában (*Érték*, *Pozíció*) párokkal tesszük, ahol *Érték* tömbben a ritka vektor nem-nulla elemei tárolódnak, és a *Pozíció* tömbben, pedig az *Érték* tömb által tartalmazott elemek ritka vektorbeli indexei szerepelnek. Egy számítógépes programban általában külön fenntartunk egy egész és egy valós tömböt, amelyek legalább a vektorban lévő elemek számával megegyező hosszúságúak. Ezt a tárolási módot tömörített vagy csomagolt tárolásnak nevezzük.

Sorszám	1	2	...	k
Érték	j_1	j_2	…	j_k
Pozíció	a_{j1}	a_{j2}	...	a_{jk}

Tábla 2.1 Ritka vektorok tárolása

Példa:

$\mathbf{v} = [1, 0, 0, -3, 1]^T$

$\mathbf{Z} = \{0, 3, 4\}$, és $|Z| = 3$

Hossz	3		
Érték	1.0	-3.0	1.0
Pozíció	0	3	4

A tárolás rendezett, ha az indexek monotonak, általában növekvők, mint ahogy a példában is láthattuk, egyébként rendezetlen. Általában nem szükséges megtartani a ritka vektor rendezettségét.

Azt a folyamatot, mikor egy explicit vektort konvertálunk tömör formába, becsomagolásnak nevezzük. Ennek az ellentétes művelete a kicsomagolásnak, mikor a tömörített vektort visszaállítjuk teljes hosszúságú vektorrá.

2.2 Ritka vektort tartalmazó műveletek

A legtöbb ritka vektort magába foglaló műveletnél a műveletek elvégzésénél használunk egy **w** munka tömböt, melynek minden elemét inicializálni kell. Ha a **w** tömb mérete nagy, akkor a gyakori inicializálás számottevő processzor időt emészt fel. Szerencsére, a számítások végrehajtásánál mindig nyomon tudjuk követni a pozíciókat. Ezeken a pozíciókon, amik a nem-nulla elemeket tartalmazzák, könnyen tárolhatunk újra zéró elemeket. Ezzel elkerülhetjük, hogy minden iterációban újra kelljen inicializálni a nagy munka tömböket és így időt spórolunk meg.

3. Ritka mátrixok

3.1 Ritka mátrixok tárolása

3.1.1 A tárolási forma minden eleme egy egyszerű mátrix elem

3.1.1.1 Koordináta séma (Coordinate Storage - COO)

A legszélesebb körben és leggyakrabban használt mód a tárolásra. Rendezett hármast *(Row, Column, Value)* használ a tárolásra. A következő példa illusztrálja legjobban a tárolási sémát.

A mátrixunk egy 5*5-ös mátrix, mely 11 nem-nulla elemet tartalmaz:

$$A = \begin{pmatrix} 0.1 & 0.0 & 0.0 & 1.2 & 0.0 \\ 0.0 & 0.0 & 2.5 & 0.0 & 0.6 \\ 1.8 & 0.0 & 0.8 & 0.0 & 0.0 \\ 0.5 & 0.0 & 0.0 & 0.9 & 1.4 \\ 0.0 & 0.7 & 0.0 & 0.2 & 0.0 \end{pmatrix} \qquad (3.1)$$

A koordinátaséma esetében a ritka mátrix tárolására három tömböt használunk. Ezek a következők: két egész típusú tömböt, a sor és oszlop indexek tárolására, illetve egy valós tömböt, mely a mátrix nem-nulla elemeinek tárolására szolgál.

Az *A* mátrix reprezentációja a következőképpen néz ki:

Példa:

Sorszám	1	2	3	4	5	6	7	8	9	10	11
Sor	1	3	4	5	2	3	1	4	5	2	4
Oszlop	1	1	1	2	3	3	4	4	4	5	5
Érték	0.1	1.8	0.5	0.7	2.5	0.8	1.2	0.9	0.2	0.6	1.4

Tábla 3.1 Koordinátaséma ritka mátrix tárolására

A Táblában minden tömb oszlop szerint rendezett. Általában, ha a mátrix sűrűsége kevesebb, mint 0.55, akkor egy tárolási sémánál kevesebb memóriára van szükség, mint a teljes mátrix tárolásánál. Ha egy rendezett háromszög mátrixnál m=1000, akkor a pontosság miatt a valós adatok kiterjesztett típusúak lesznek és 10 byte-on tárolódnak, míg az indexek tárolására valós típust használunk, melynek mérete 4 byte. Ekkor a mátrix teljes tárolásakor lefoglalódik $m \times m \times 10 = 1000 \times 1000 \times 10 \approx 10$MB memória. A következő táblázat megmutatja, hogy a koordinátaséma esetében a tároláshoz szükséges memória mennyiség hogyan függ a tárolni kívánt mátrix sűrűségétől.

Sűrűség	0.05	0.15	0.025	0.35	0.45	**0.55**	0.65	0.75	0.85
≈ MB	0.9	2.7	4.5	6.3	8.1	**9.9**	11.7	13.5	15.3

Tábla 3.2 Memória követelmények a koordinátasémánál

A Tábla 3.2 megmutatja, hogy 0.55-ös sűrűség alatt memóriatakarékos megoldás, ha a koordinátasémát alkalmazzuk tároláskor és a mátrixok nem teljes méretében tároljuk.

Az elemek beszúrását és törlését könnyebb elvégezni mikor ezt a tárolási formát alkalmazzuk, míg az elemek közvetlen elérése költségesebb művelet, mert a tároláshoz használt három tömböt szekvenciálisan végig kell vizsgálni ahhoz, hogy egy-egy elemet

megkeressünk.

A közvetlen elérés teljesítményének javítására D.E.Knuth [1] javasolta a „belépési pontok” használatát soroknál és oszlopoknál egyaránt. A pont a sor és oszlop első nem nulla elemére mutat. Ezekre két további tömböt használunk: *KS* (a következő, ugyanabban a sorban lévő nem-nulla elem sorszáma), *KO* (a következő, ugyanabban az oszlopban lévő nem-nulla elem sorszáma). Ez a két tömb megkönnyíti számunkra a sorban vagy az oszlopban következő nem-nulla elem keresését, a három tömb vizsgálata közben.

Példa:

Sorszám	1	2	3	4	5	6	7	8	9	10	11
KS	7	6	8	9	10	0	0	11	0	0	0
KO	2	3	0	0	6	0	8	9	0	11	0

Tábla 3.3 *KS* és *KO* tömbök

Lehet még tovább tökéletesíteni a közvetlen elérést, ehhez még két további tömbre van szükségünk, melyek a következők: *ES* és *EO*. Mindkét vektor hossza 5. Minden sorban és oszlopban megmutatják az adott sor és oszlop első elemének sorszámát. Jelen esetben a két tömb a következőképpen néz ki:

ES	1	5	2	3	4
EO	1	4	5	7	10

Tábla 3.4 *ES* és *EO* tömbök

A séma a következőképpen működik:

Először is vegyük a *EO* tömb 4. elemét: *EO*[4]=7. Ez mutatja meg nekünk, hogy az Érték tömb 7. helyén mi áll: *Érték*[7]=1.2, amely az 1. sorban van (*Sor*[7]=1). Majd a *KO* tömb 7. pozícióján 8-as áll, vagyis *KO*[7]=8, ami azt jelenti, hogy a 4. oszlopban a következő elem az *Érték*[8]=0.9. A *KO*[8]=9, vagyis a 0.9-et a 4. oszlopban az *Érték*[9]=0.2 követi. *KO*[9]=0, mely szerint, a 4. oszlopban a 0.2-es érték után nem szerepel további elem.

Analóg módon működik oszlopokra is.

3.1.1.2 Tömörített oszlop tárolás (Compressed Column Storage - CCS)

Ezen tárolási forma esetében, minden ritka oszlop-vektor ugyanabban a valós tömbben szekvenciálisan tárolódik. A tömbben lehetnek rendezetten és rendezetlenül az elemek. A tároláshoz 3 tömböt használunk ennél a tárolási módnál.

Ezek a következők:

(1) *Érték:* egy valós tömb, mely a mátrix nem-nulla elemit tartalmazza.

(2) *Sor_ind:* egy egész típusú tömb a mátrix nem-nulla eleminek sor indexeit tartalmazza.

(1) *Oszlop_ptr:* egész típusú tömb, mely a mátrix oszlopaiban szereplő első nem nulla elemek *Érték* tömbbeli indexeit tartalmazza.

Ha **Érték(k)=a_{ij}**, akkor **Sor_ind=i** és **Oszlop_ptr(j) ≤ k < Oszlop_ptr(j+1)**.

Példa:

Sorszám	1	2	3	4	5	6	7	8	9	10	11
Sor_ind	1	3	4	5	2	3	1	4	5	2	4
Érték	0.1	1.8	0.5	0.7	2.5	0.8	1.2	0.9	0.2	0.6	1.4

Sorszám	1	2	3	4	5
Oszlop_ptr	1	4	5	7	10

Tábla 3.5 Az *A (3.1)* mátrix CCS formája

A CCS nagyon hatékony a memória használatban. A metódusnak van néhány súlyos vesztesége: az első vesztesége, hogy nem nyújt adatstruktúrát a mátrix sorainak közvetlen elérésére, illetve az új elemek beszúrása is nehézkes.

Az első probléma orvoslására szolgál CRS tárolási forma.

3.1.1.3 Tömörített sor tárolás (Compressed Row Storage - CRS)

Az egyik széles körben használt, hatékony tárolási forma a CRS. Ez a tárolási módszer analóg módon működik a CCS-el. A reprezentációhoz itt is 3 vektort használunk: egy valós típusú (*Érték)*, és két egész típusú tömböt (*Oszlop_ind*, *Sor_ptr):*

(1) *Érték:* a mátrix nem-nulla eleminek tárolására szolgál.

(2) *Oszlop_ind:* az Érték tömbben szereplő értékekhez tartozó oszlopindexeket fogja tartalmazni.

(3) Sor_ptr: elemi indexek, melyek megmutatják, hogy az Érték tömbben hol kezdődik egy sor.

Ha **Érték(k)=a_{ij}**, akkor **Oszlop_ind=j** és **Sor_ptr(i) ≤ k < Sor_ptr(i+1)**.

Példa:

Sorszám	1	2	3	4	5	6	7	8	9	10	11
Oszlop_ind	1	4	3	5	1	3	1	4	5	2	4
Érték	0.1	1.2	2.5	0.6	1.8	0.8	0.5	0.9	1.4	0.7	0.2

Sorszám	1	2	3	4	5
Sor_ptr	1	3	5	7	10

Tábla 3.6 Az *A (3.1)* mátrix CRS formája

Ha a mátrixunk szimmetrikus, akkor elég csak a felső trianguláris részt tárolnunk.

3.1.1.4 Ritka diagonális (Sparse Diagonal – DIA)

Ennél a tárolási módnál mindössze két tömbre van szükségünk. Egy *Érték* és egy *Diag_ind* tömbre:

(1) *Érték*: egy kétdimenziós valós típusú elemeket tartalmazó tömb, melyben a sorok száma l= min(m, n), ahol az eredeti mátrix m × n-es; oszlopainak száma pedig megegyezik a nem zéró diagonálisok számával.

(2) *Diagind:* egész típusú elemeket tartalmazó tömb, melynek hossza megegyezik a nem zéró diagonálisok számával; *Diagind(j) = i*, ami azt jelenti, hogy az *Érték* tömb *j*-ik oszlopa tartalmazza az eredeti mátrix *i*-ik átlóját.

$$A = \begin{pmatrix} 11 & 0 & 13 & 0 & 0 \\ 21 & 0 & 0 & 24 & 0 \\ 31 & 31 & 33 & 0 & 35 \\ 0 & 42 & 0 & 44 & 0 \\ 0 & 0 & 53 & 0 & 55 \end{pmatrix} \qquad (3.2)$$

Az *B* mátrix reprezentációja a következőképpen néz ki:

Példa:

$$\textit{Érték} = \begin{pmatrix} * & * & 11 & 13 \\ * & 21 & 0 & 24 \\ 31 & 32 & 33 & 35 \\ 42 & 0 & 44 & * \\ 53 & 0 & 55 & * \end{pmatrix} \qquad \textit{Diagind} = \begin{pmatrix} -2 & -1 & 0 & 2 \end{pmatrix}$$

Tábla 3.7 Példa DIA-ra

Diagind(j) = i és i = 0, akkor az *Érték* kétdimenziós tömb j-ik oszlopa tartalmazza az A mátrix fődiagonálisát; továbbá:

- Ha m ≤ n, akkor:
  - i < 0, ekkor az *Érték* tömb j-ik oszlopában az első |i| elem nem használt, és a következő m - |i| elem az i-ik diagonális az eredeti mátrixban a fődiagonális alatt.
  - i > 0, akkor az *Érték* tömb j-ik oszlopában az első min(m, n-i) elem a i-ik diagonális az eredeti mátrixban a fődiagonális fölött.

- Ha m > n, akkor:
    - i < 0, akkor az *Érték* tömb j-ik oszlopában az első n – min(n, m - |i|) elem az i-ik diagonális a főátló fölött az eredeti mátrixban, és a megmaradó elemek nem használtak.
    - i > 0, akkor az *Érték* tömb j-ik oszlopában az első n – i elem az i-ik diagonális a főátló felett az eredeti mátrixban, és a fennmaradó elemek nem használtak.

Ha a mátrix szimmetrikus, akkor elég csak a mátrix alsó (vagy felső) trianguláris diagonálisait tárolni.

3.1.1.5 Ellpack/Itpack (ELL)

Maxnz változó azt mutatja meg, hogy maximum mennyi elem lehet a mátrix soraiban.

A tároláshoz két tömböt használunk:

(1) *Érték:* ez egy $m \times maxnz$ méretű kétdimenziós valós típusú tömb, melynek i. sorában a mátrix nem-nulla elemei lesznek. A sorokat szekvenciálisan töltjük föl. Ha nincs az adott sorban több nem-nulla elem, akkor nullákkal pótoljuk, ha szükséges.

(2) *Oszlopind:* ez egy $m \times maxnz$ méretű kétdimenziós egész típusú elemeket tartalmazó tömb, mely az Érték tömbben lévő elemekhez határozza meg, hogy az eredeti mátrixban milyen oszlopindex tartozik. Ahol az Érték tömbben 0 szerepel, azokhoz az előttük lévő elem oszlopindexét rendeli.

Legyen a mátrixunk a (3.2)-es mátrix. A következő példa jól szemlélteti a tárolási formát:

Példa:

$$Érték = \begin{pmatrix} 11 & 13 & 0 & 0 \\ 21 & 24 & 0 & 0 \\ 31 & 32 & 33 & 35 \\ 42 & 44 & 0 & 0 \\ 53 & 55 & 0 & 0 \end{pmatrix}$$

$$Oszlopdiag = \begin{pmatrix} 1 & 3 & 3 & 3 \\ 1 & 4 & 4 & 4 \\ 1 & 2 & 3 & 5 \\ 2 & 4 & 4 & 4 \\ 3 & 5 & 5 & 5 \end{pmatrix}$$

Tábla 3.8 Példa ELL tárolásra

3.1.1.6 Jagged Diagonal Storage (JDS)

Szükségünk van egy P permutációs mátrixra mely a mátrix sorait permutálja, mégpedig úgy, hogy csökkenő sorba rendezi a mátrix sorait, a sorokban szereplő elemek száma szerint. A D*iagptr* fogja reprezentálni a permutációkat. Legyen $\tilde{A} = PA$, vagyis a B mátrix permutált mátrixa.

A tároláshoz négy tömbre van szükségünk:

(1) *Érték*: Valós típusú elemeket tartalmazó tömb, mely a nem-nulla értékeket fogja tárolni, mégpedig úgy, hogy oszloponként haladunk.

(2) *Oszlopind*: Egy egész típusú tömb, amely az *Érték* tömbben szereplő elemekhez tartozó eredeti mátrixbeli oszlopindexeket tartalmazza.

(3) *Perm*: Ez egy egész típusú permutációs tömb, az átszervezett sorok indexeit tartalmazza.

(4) *Diagptr*: Egy egész típusú tömb, mely az eltolt mátrix első sorának nem-nulla elemeinek *Érték* tömbbeli sorszámát tartalmazza.

Példa:

Legyen a kiinduló mátrixunk az A (3.2) mátrix. Ezt a mátrixot átalakítjuk úgy, hogy minden sorában a nem-nulla elemeket balra húzzuk. Így a következőt kapjuk:

$$\begin{pmatrix} 11 & 13 & 0 & 0 & 0 \\ 21 & 24 & 0 & 0 & 0 \\ 31 & 31 & 33 & 35 & 0 \\ 42 & 44 & 0 & 0 & 0 \\ 53 & 55 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix}$$

$$P = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Majd permutáljuk a sorokat a P permutációs mátrix szerint:

$$\tilde{A} = \begin{pmatrix} 31 & 31 & 33 & 35 & 0 \\ 21 & 24 & 0 & 0 & 0 \\ 11 & 13 & 0 & 0 & 0 \\ 42 & 44 & 0 & 0 & 0 \\ 53 & 55 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} 3 \\ 2 \\ 1 \\ 4 \\ 5 \end{matrix}$$

Majd a 4 tömb feltöltése:

Sorszám	1	2	3	4	5	6	7	8	9	10	11	12
Érték	31	21	11	42	53	32	24	13	44	55	33	35
Oszlopind	1	1	1	2	3	2	4	3	4	5	3	5

Sorszám	1	2	3	4	5
Perm	3	2	1	4	5

Sorszám	1	2	3	4
Diagptr	1	6	11	12

Tábla 3.9 Az *A* (3.2) mátrix JDS tárolása

3.1.1.7 Skyline

Ezzel a tárolási sémával csak trianguláris mátrixokat tárolhatunk. A tároláshoz két tömböt használunk:

(1) *Érték:* Ez, mivel a mátrix elmeit fogja tartalmazni, így valós típusú. Az elemek halmazát fogja tartalmazni, minden sorból az első nem-nulla elemtől a diagonálisig, ha a mátrix alsó trianguláris. Ha felső trianguláris, akkor minden oszlop első nem-nulla elemétől a diagonális elemig fogja tartalmazni.

(2) *Mutató:* Egy egész típusú tömb, melynek dimenziója $m + 1$, ahol m az alsó trianguláris mátrix sorainak a száma (felső trianguláris esetén pedig az oszlopok száma). *Mutató(i) – Mutató(1) + 1* mutatja alsó trianguláris esetén az i. sor első nem-nulla elem helyét az *Érték* tömbben. *Mutató(m+1) = nnz + Mutató(1),* ahol *nnz* az *Érték* tömb elemeinek a száma.

Példa:

Legyen a kiinduló mátrixunk a következő:

$$A = \begin{pmatrix} 1 & -1 & -3 & 0 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{pmatrix} \qquad (3.3)$$

Sorszám	1	2	3	4	5	6	7	8	9	10	11	12
Érték	1	-2	5	4	-4	0	2	7	8	0	0	-5

Sorszám	1	2	3	4	5	6
Mutató	1	2	4	5	9	13

Tábla 3.10 Az *A* alsó trianguláris részének Skyline tárolása

Sorszám	1	2	3	4	5	6	7	8	9	10	11
Érték	1	-1	5	-3	0	4	6	7	4	0	-5

Sorszám	1	2	3	4	5	6
Mutató	1	2	4	7	9	12

Tábla 3.11 Az *A* felső trianguláris részének Skyline tárolása

3.1.1.8 Láncolt lista

Négy adat struktúrát használunk a tároláshoz. Ezek közül az első, az *Érték* tömb, mely a mátrix nem-nulla elemeit tartalmazza (oszlop folytonosan). A második tömb egy *Link* nevű tömb, mely minden *Érték* tömbbeli elemhez tárolja az adott oszlopban utána következő elem *Érték* tömbbeli indexét. Ha egy elem az oszlop utolsó nem-nulla eleme, akkor a hozzá tartozó *Link* tömbbeli érték *Nil* lesz. A következő adat struktúra a *Sor* tömb, mely minden *Érték* tömbben szereplő elemhez tartalmazza az elem mátrixbeli sor indexét. Végül definiálunk egy mutatót, mely minden oszlop első nem-nulla elemére mutat, ennek a mutatónak a neve: *Fej*.

A következő példa reprezentálja a tárolási módot:

Példa:

Sorszám	1	2	3	4	5	6	7	8	9	10	11
Sor	1	3	4	5	2	3	1	4	5	2	4
Érték	0.1	1.8	0.5	0.7	2.5	0.8	1.2	0.9	0.2	0.6	1.4
Link	2	3	*Nil*	*Nil*	6	*Nil*	8	9	*Nil*	11	*Nil*

Sorszám	1	2	3	4	5
Fej	1	4	5	7	10

Tábla 3.12 Az *A* (3.1) mátrix láncolt lista tárolása

A harmadik oszlop helyreállítása a következő módon történik: a *Fej* tömbből le tudjuk olvasni, hogy az adott oszlop melyik elemen kezdődik. *Fej[3] = 5*, tehát az *Érték* tömb 5. eleme lesz a 3. oszlop kezdő eleme, vagyis a 2.5-as érték. Ezután a *Sor[5] =2,* ami azt mutatja, hogy a 2.5 eredeti helye a mátrix harmadik oszlopának második sorában van. A *Link[5] = 6* pedig megadja, hogy a 2.5-ös érték után az adott oszlopban, az *Érték* tömb 6. eleme szerepel. Majd az *Érték* tömb 6. eleméhez, ami jelen esetben 0.8, vesszük a *Sor* és *Link* tömbök hatodik elemét. Mivel a *Link* tömb 6. eleme *Nil*, így a 3. oszlopban a 0.8 után nem szerepel nem-nulla elem.

3.1.2 Állandó méretű blokkokat tartalmazó adat struktúrák

3.1.2.1 Fixed-Size Blocking

Ebben a megközelítésben a mátrixot néhány mátrix összegeként írjuk fel. Ezekből néhány tartalmazza a mátrix tömör blokkjait. Például a következő A mátrixot bontsuk fel az A_{12} és az A_{12} mátrixokra úgy, hogy $A = A_{12} + A_{11}$, ahol A_{12} tartalmazza az 1×2 –es

blokkokat, az A_{11} pedig a maradékot. A következő példa illusztrálja mindezt:

$$\begin{pmatrix} x & x & x & & \\ & x & & x & \\ x & x & & & \\ & & & x & x \\ & & x & x & x \end{pmatrix} = \begin{pmatrix} x & x & & & \\ & & & & \\ x & x & & & \\ & & & x & x \\ & & x & x & \end{pmatrix} + \begin{pmatrix} & & & x & \\ & x & & x & \\ & & & & \\ & & & & \\ & & & & x \end{pmatrix}$$

$$A = A_{12} + A_{11}$$

3.1.2.2 Block Compressed Row Storage (BCRS)

Ezen tárolási formánál a mátrixot blokkokra bontjuk szét, tehát a mátrix minden eleme egy LB × LB – s tömör blokk, ahol LB tipikusan egy kicsi szám, mely kisebb, mint 10 és az LB fogja megadni a mátrix blokkjainak dimenzióját. A blokkok lefedik a mátrix összes nem-nulla elemét, természetesen helyenként tartalmazhatnak zéró elemeket is.

A blokkok tárolására két lehetőség van. Az egyik, hogy minden blokk elemét folyamatosan tároljuk úgy, hogy például ha LB = 2, akkor minden blokk elemre tároljuk először az (1, 1)-es helyen lévő elemet, majd az (2, 1) –t, aztán (1, 2)-t, és végül (2, 2)-t. Ezért egy blokk tárolásához, LB^2 memória helyre van szükség. A másik mód, mikor folyamatosan tároljuk minden blokk (1, 1)-es helyen lévő elemét, majd a (2, 1)-esen lévőt, és így tovább. A két megoldási mód közül az első a kényelmesebb.

Ha tudjuk a blokkban az első nem-nulla elem oszlop indexét, akkor az összes többinek fogjuk tudni az oszlopindexét. Más szóval csak egy memória índirekció szükséges ahhoz, hogy a blokk összes elemének helyét ismerjük.

Legyen A a kiindulási mátrixunk, mely tetszőleges $m \times k$ méretű mátrix lehet, ahol m és k egész számok. Az $\tilde{A}$ jelölje azt a mátrixot, melynek elemei a blokkok lesznek. Ennek mérete $m_b \times k_b$, ahol m_b jelöli az $\tilde{A}$ mátrix egyetlen sorában helyet foglaló blokkok számát és k_b pedig az egy oszlopban lévőkét. A tárolási forma igényli a következő tömböket:

(1) *Érték*: ez a tömb tárolja a mátrix nem-nulla blokkjait, hossza egyenlő a nem-zéró blokkok számával.

(2) *Blokk_ind*: egy egész típusú elemeket tartalmazó tömb, melynek hossza megegyezik a blokkok számával, és a blokkok $\tilde{A}$ -beli oszlop indexeit tartalmazza.

(3) *Blokk_ptr*: egy egész típusú elemeket tároló tömb, hossza eggyel nagyobb, mint a sorok száma. Minden sorban az első blokk *Érték* tömbbeli indexét tárolja. Utolsó elemének értéke: összeadjuk a mátrix nem-nulla blokkjainak számát a *Blokk_ptr* tömb első elemével.

A tárolási séma hatékonysága közvetlenül függ a blokkok méretétől.

Példa:

Legyen most a következő mátrixunk a következő:

$$A = \begin{pmatrix} 11 & 12 & 0 & 0 & 15 & 16 \\ 21 & 22 & 0 & 0 & 25 & 26 \\ 0 & 0 & 33 & 0 & 35 & 36 \\ 0 & 0 & 43 & 44 & 45 & 46 \\ 51 & 52 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 0 \end{pmatrix} \qquad (3.3)$$

$$\tilde{A} = \left(\begin{array}{cc|cc|cc} 11 & 12 & 0 & 0 & 15 & 16 \\ 21 & 22 & 0 & 0 & 25 & 26 \\ \hline 0 & 0 & 33 & 0 & 35 & 36 \\ 0 & 0 & 43 & 44 & 45 & 46 \\ \hline 51 & 52 & 0 & 0 & 0 & 0 \\ 61 & 62 & 0 & 0 & 0 & 0 \end{array}\right)$$

$$Érték(1) = \begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix} \qquad Érték(2) = \begin{pmatrix} 15 & 16 \\ 25 & 26 \end{pmatrix}$$

$$Érték(3) = \begin{pmatrix} 33 & 0 \\ 43 & 44 \end{pmatrix} \qquad Érték(4) = \begin{pmatrix} 35 & 36 \\ 45 & 46 \end{pmatrix}$$

$$Érték(5) = \begin{pmatrix} 51 & 52 \\ 61 & 62 \end{pmatrix}$$

Sorszám	1	2	3	4	5
Blokk_ind	1	3	2	3	1
Blokk_ptr	1	3	5	6	

Az **első** tárolási formánál az *Érték* tömb elemei a memóriában a következőképpen fognak elhelyezkedni:

(11, 21, 12, 22, 15, 25, 16, 26, 33, 43, 0, 44, 35, 45, 36, 46, 51, 61, 52, 62)

A **második** tárolási formánál pedig a következőképpen tárolódnak az elemek:

(11, 15, 33, 35, 51, 21, 25, 43, 45, 61, 12, 16, 0, 36, 52, 22, 26, 44, 46, 62)

Tábla 3.13 Az *A* mátrix BCRS tárolása

Ha a mátrix szimmetrikus, akkor elég csak a felső (vagy alsó) trianguláris részt tárolni.

3.1.2.3 Variable Block Compressed Sparse Row Strorage (VCBR)

Legyen A a kiindulási mátrixunk, mely tetszőleges $m \times k$ méretű mátrix lehet, ahol m és k egész számok. Az $\tilde{A}$ jelölje azt a mátrixot, melynek elemei a blokkok lesznek. Ennek mérete $m_b \times k_b$, ahol m_b jelöli az $\tilde{A}$ mátrix egyetlen sorában helyet foglaló blokkok számát és k_b pedig az egy oszlopban lévőkét. A VCBR adat struktúrát a következőképpen definiálhatjuk. A mátrixot nem egyforma méretű blokkokra tagoljuk, ahol a blokkok tartalmazzák az összes nem-nulla elemet.

Minden blokkot úgy tárolunk, mintha egy tömör mátrix lenne. Egy valós, és három, négy, vagy öt egész típusú tömböt használunk a mátrix tárolásához:

(1) *Érték:* skalár típusú tömb, melynek hossza egyenlő a mátrix nem-nulla elemeinek számával. Sor folytonosan haladunk végig a blokkokon, és vesszük a blokkok elemeit oszlop folytonosan.

(2) Ind: egész típusú elemeket tároló tömb, melynek hossza eggyel nagyobb, mint a mátrix blokkjainak száma. A tömb i-ik helyén a mátrix i-ik blokkjának (1, 1) indexű helyén szereplő elem, *Érték* tömbbeli helyét tartalmazza. Az utolsó elemet úgy

számítjuk, hogy a blokk elemeinek számához hozzáadunk egyet.

(3) Blokk_ind: egy egész típusú elemeket tartalmazó tömb, melynek hossza megegyezik a blokkok számával, és a blokkok $\tilde{A}$ -beli oszlop indexeit tartalmazza.

(4) Sor_ptr: egész típusú elemeket tartalmaz, mérete: $m_b + 1$. Haladunk sor folytonosan az $\tilde{A}$ mátrixban, és minden sor első nem-nulla blokkjának (1, 1) indexű helyén álló elemének, A mátrixbeli sor indexét tárolja. Az utolsó elemének kiszámítása: m + Sor_ptr(1).

(5) Oszlop_ptr: egész típusú elemeket tartalmaz, hossza $k_b + 1$. Haladunk oszlop folytonosan az $\tilde{A}$ mátrixban, és minden oszlop első nem-nulla blokkjának (1, 1) indexű helyén álló elemének, *A* mátrixbeli oszlop indexét tárolja. Az utolsó eleme pedig: k + Oszlop_ptr(1).

(6) Blokk_ptr: egész típusú, mutatókat tartalmazó tömb, hossza $m_b + 1$. Minden blokksor első nem nulla blokkjának (1, 1)-es indexű elemének a *Blokk_ind*-beli indexe. Utolsó eleme pedig a *Blokk_ind* utáni elemre mutat.

Példa:

$$A = \begin{pmatrix} 4 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 \\ 1 & 5 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 6 & 1 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 9 & 3 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 3 & 4 & 5 & 10 & 4 & 3 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 13 & 4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 3 & 11 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 7 & 0 & 0 \\ 8 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25 & 3 \\ -2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 12 \end{pmatrix} \qquad (3.4)$$

$$\tilde{A} = \left(\begin{array}{cc|ccc|c|ccc|cc} 4 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 1 \\ 1 & 5 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & -1 \\ \hline 0 & 0 & 6 & 1 & 2 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 7 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 2 & 9 & 3 & 0 & 0 & 0 & 0 & 0 \\ \hline 2 & 1 & 3 & 4 & 5 & 10 & 4 & 3 & 2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 4 & 13 & 4 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 3 & 3 & 11 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 7 & 0 & 0 \\ \hline 8 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 25 & 3 \\ -2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 12 \end{array}\right)$$

Sorszám	1	2	3	4	5	6	7	8	9	10	11	12	13
Érték	4	1	2	5	1	2	-1	0	1	-1	6	2	-1
Sorszám	14	15	16	17	18	19	20	21	22	23	24	25	26
Érték	1	7	2	2	1	9	2	0	3	2	1	3	4
Sorszám	27	28	29	30	31	32	33	34	35	36	37	38	39
Érték	5	10	4	3	2	4	3	0	13	3	2	4	11
Sorszám	40	41	42	43	44	45	46	47	48	49	50	51	
Érték	0	2	3	7	8	-2	4	3	25	8	3	12	

Sorszám	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Ind	1	5	7	11	20	23	25	28	29	32	35	44	48	52

Sorszám	1	2	3	4	5	6	7	8	9	10	11	12	13
Blokk_ind	1	3	5	2	3	1	2	3	4	3	4	1	5

Sorszám	1	2	3	4	5	6
Sor_ptr	1	3	6	7	10	12
Oszlop_ptr	1	3	6	7	10	12
Blokk_ptr	1	4	6	10	12	14

Tábla 3.14 Példa BCRS tárolásra

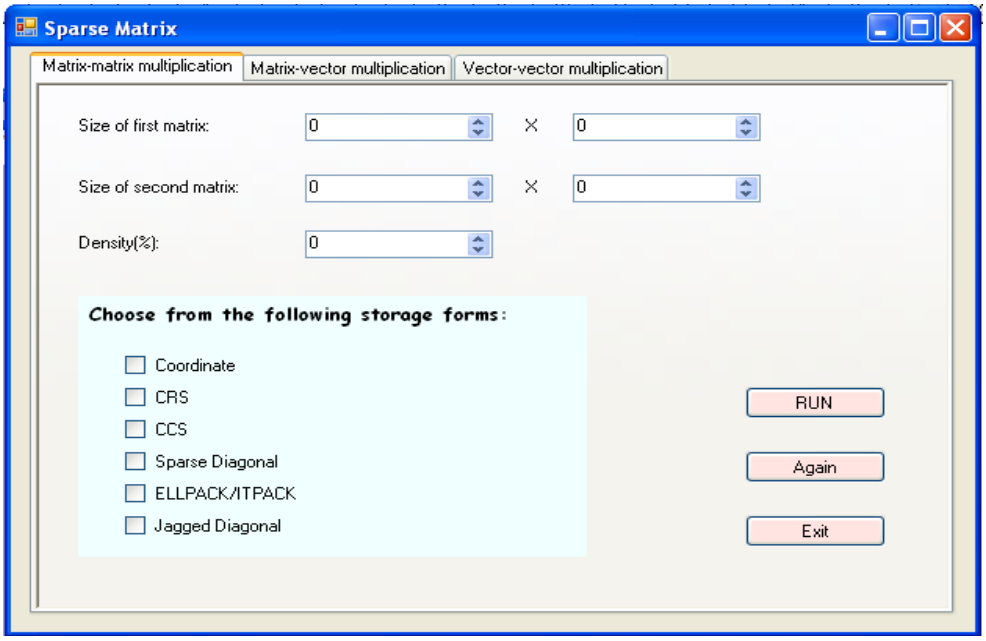
4. Alkalmazás

Programomban a fentiekben említett tárolási formák közül a megvalósítottak:

- Koordináta séma,
- Tömörített sor tárolás,
- Tömörített oszlop tárolás,
- Ritka diagonális,
- ELLPACK,
- Jagged diagonális.

4.1 Interfész bemutatása

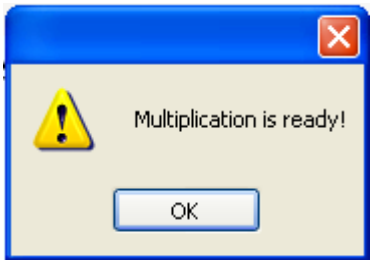
Az alkalmazás felülete a következőképpen mutat:



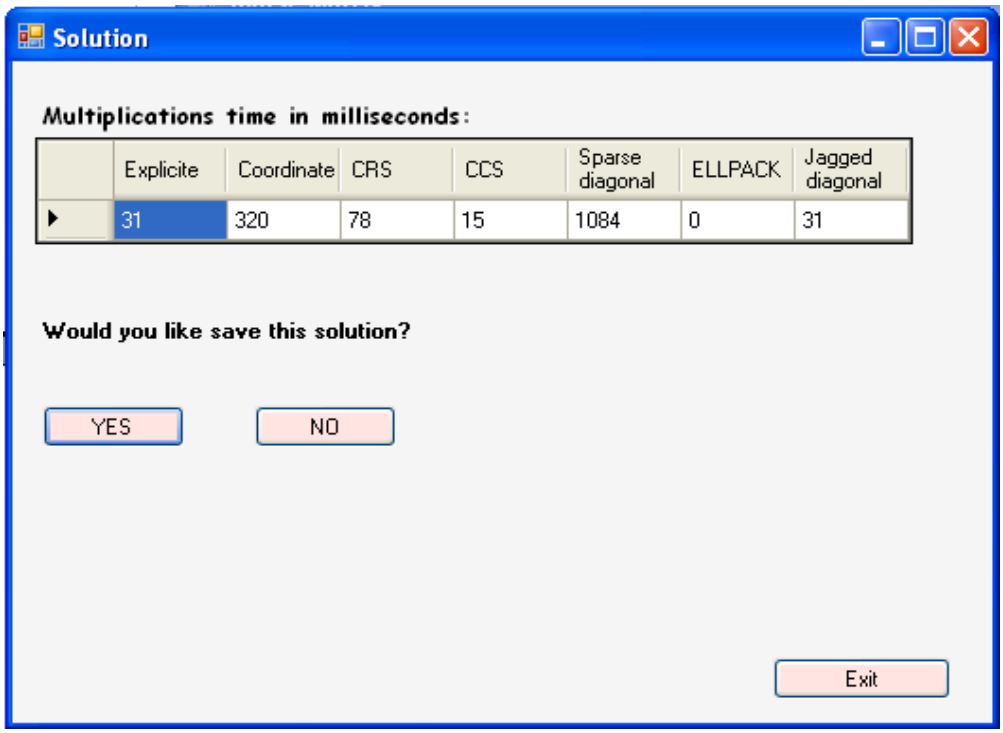
Látható, hogy három tab oldalt tartalmaz, melyek külön-külön, rendre a mátrix-mátrix, mátrix-vektor és a vektor-vektor szorzást valósítják meg. Minden oldalon tetszőleges 1 és 1000 közötti szám választható a mátrix(ok) és/vagy vektor(ok) méretének, sor × oszlop

alakban. A méretek után a kezelendő mátrix(ok) és/vagy vektor(ok) sűrűségét választhatjuk meg. Ezt követően a tárolási sémák közül tetszőleges számút jelölhetünk tesztelésre. A „RUN” gombbal indítható el a számolás. Az „Again” feliratú gomb lenyomásával lenullázódik a felület minden egész típusú értéke, valamint a ckeckboxok jelölései törlődnek. Az „Exit” gombbal kiléphetünk a programból.

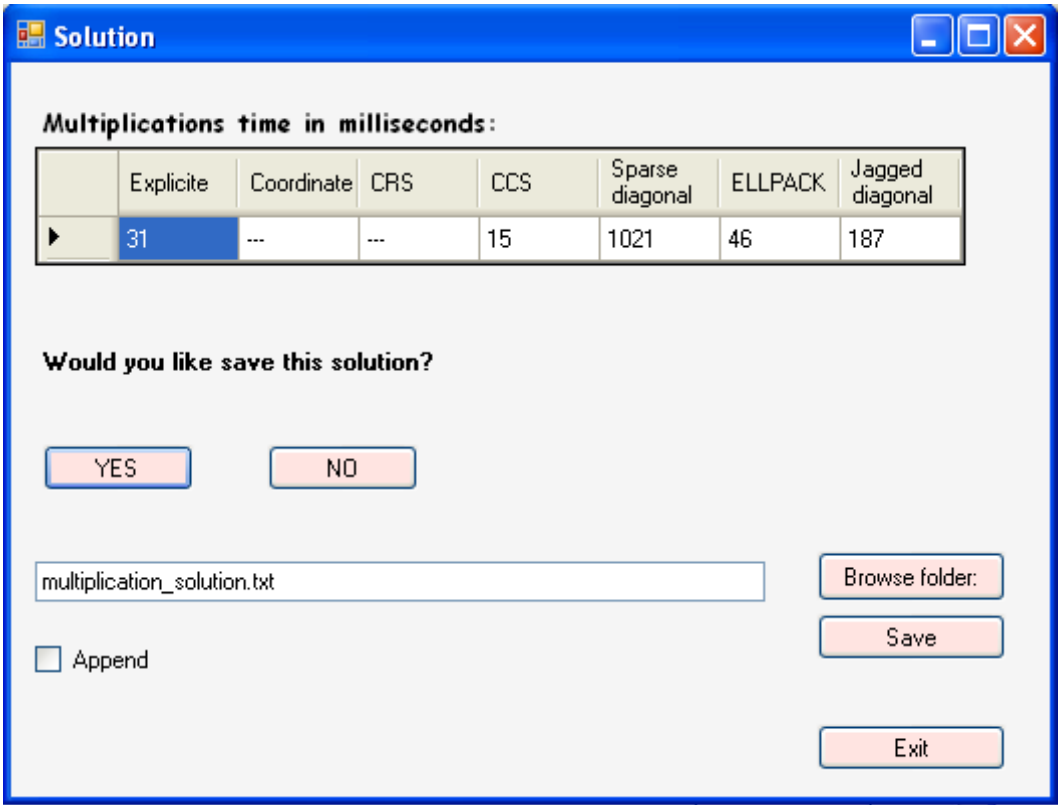
A „RUN” címkéjű gombra klikkelve elindul a számolás. A szorzás elvégzése után, a következő MessageBox-ot kapjuk:



Ez az ablak figyelmezteti a felhasználót, hogy a szorzást/szorzásokat elvégezte a program. Az OK gombra kattintva láthatjuk az eredmény formon a szorzásokhoz szükséges időtartamot milliszekundumban:

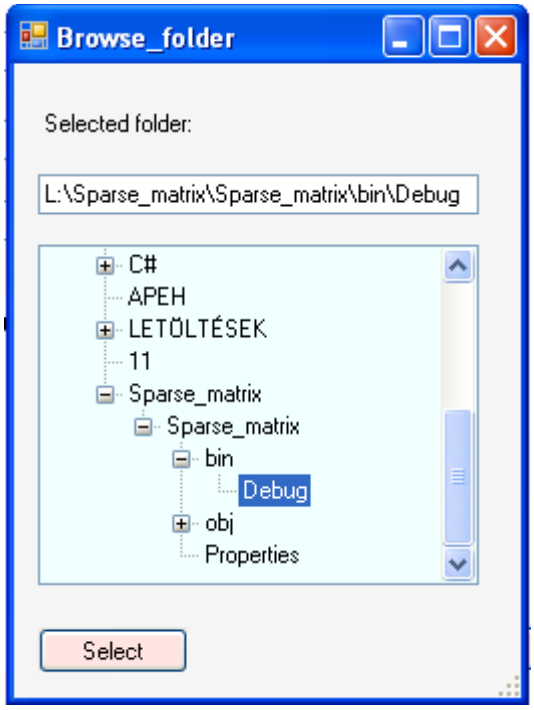


A táblázatban látható számítási eredményeket a felhasználó mentheti is, ekkor az IGEN gomba klikkelve, az alábbi módon változik a form felülete:



Ha az „Append” címkéjű checkboxot bejelöljük, akkor egy már létező fájlhoz fűzi hozzá a kapott eredményeket, egyébként pedig felülírja ha már létezett a fájl.

Majd a „Browse folder” gombra klikkelve, egy újabb ablakot kapunk, melynek segítségével az eredmény kiírására szolgáló fájl útvonalát határozhatjuk meg. Ezen felület itt látható:



Ez az kiválasztott elérési út fog megjelenni a fentebb látható felület szövegmezőjében. Majd a SAVE gombra klikkelve mentjük az adatokat

.

4.2 Implementáció

A tárolási formák megvalósításához egy ősosztályt és hat alosztályt használok. Az ősosztályom neve Storage_forms, melynek alosztályai lesznek a Coordinate, CRS, CCS, Ell, Jagged, Sparsediag osztályok, ezek a megfelelő sémákat, míg a Storage_forms a direkt tárolást fogja megvalósítani.

Minden osztályban megtalálható két példányszintű adattag **(value, index),** és egy osztályszintű **(time).** Az előbbiek a vektorok tárolását végzik. A **value** tartalmazza a vektor nem-nulla elemeit, míg az **index** tömbben minden **value**-beli elem vektorbeli indexét tárolom. Ezeket a mátrix-vektor szorzásnál használom. A **time**-ba kerül a szorzáshoz szükséges időmennyiség milliszekundumban. Ezek mivel minden osztályban megtalálhatók, így nem

fogom minden osztály leírásánál megemlíteni.

Minden osztályban találhatunk két konstruktort, melyek közül az egyik a mátrix-mátrix szorzáshoz tartozik, ekkor a mátrixok méretén kívül a két struktúrát is megkapja paraméterként. Míg a másik a mátrix-vektor szorzás választásánál fut le. Paraméterei a mátrix, a vektor és ezek méretei. Minden osztályban az első konstruktorból a *szorozmm* metódust hívjuk meg, míg a másodikból a *szorozmv* függvényt.

4.2.1 Coordinate osztály

A fent említetteken kívül 6 példányszintű adattagja van: **value1, row1, column1, value2, row2, column2.**

Konstruktorok: A **value[1/2]**, **row[1/2]**, **column[1/2],** *Arraylist* típusú adattagok, melyek rendre a mátrix elemeinek értékeit, sor- és oszlopindexeit tartalmazzák. Szekvenciálisan haladok a mátrix elemein, és ha nem-nulla értékűt találok, akkor a **value** listához hozzáadom ezt az értéket, sorindexét a **row** tömbhöz, oszlopindexét pedig a **column**-hoz adom. Ha a mátrix-vektor szorzást választotta a felhasználó, akkor a vektor tárolását is hasonlóképpen kell megvalósítani, azzal a különbséggel, hogy oszlopindexet nem tárolunk. Majd a **szorozmm** és **szorozmv** metódusok végzik el a mátrix-mátrix és a mátrix-vektor szorzásokat.

szorozmm metódus működése:

Ez az *integer* visszatérési típusú függvény, mely a mátrix-mátrix szorzást végzi. Két darab segéd tömböt használok, melyek hossza megegyezik az első mátrix oszlopainak számával. Ezek a **vectr** és a **vectc** tömbök, melyek közül az előbbi az első mátrix sorait fogja tárolni, míg a másik, a második mátrix oszlopait fogja. Minden sort minden oszloppal szorzok és szorzatukat, a **szorzat** kétdimenziós tömbben tárolom.

Egy for ciklus megy végig az első mátrix sorainak indexein, majd egy belső ciklusban a *k* változó segítségével pedig a **row1** értékein haladok. Vizsgálom, hogy a **row1** elemei közül melyek egyeznek meg az aktuális sorindexszel. Ha egyezést találok, akkor veszem a **column1**-ben a *k*-ik elemet, amely megmutatja, hogy a **value1** tömb megfelelő elemét a **vectr**

segédtömbbe hová helyezzük (column1[k]-ik helyre). Majd a másik mátrix oszlopait kell sorban előállítanunk. A második mátrix oszlopindexein haladva, vizsgálom a **column2** értékeit egy *l* változó segítségével, és ha egy elem értéke megegyezik az aktuális oszlopindexszel, akkor a **row2** tömb *l*-ik eleme által meghatározott helyre kerül a **vectc** tömbbe a megfelelő **value2-**beli érték. Majd elvégzem a szorzásokat, melyeknek eredményét a **szorzat** tömb megfelelő indexű pozícióira helyezek.

A for ciklusok előtt egy változóban tárolom a pontos időt, majd a szorzások végrehajtása után ismét lekérem az aktuális időt, melyek különbségéből kiszámolom, hogy mennyi a mátrix-mátrix szorzáshoz szükséges időtartam (milliszekundumokban). Ezt a mennyiséget tárolom a time változóban.

szorozmv metódus működése:

Paraméterként megkapja a mátrix és a vektor méreteit, valamint a mátrixot és a vektort. Azokat a segédtömböket használja, mint az előző metódus, illetve a sorokat is ugyanúgy állítja elő, melyeket a **vectc** tömb által tartalmazott értékekkel szorzunk. A **vectc** tömb elemeit kezdetben nullára állítjuk, majd az **index** tömb által meghatározott indexű helyekre a **value** tömb megfelelő elemeit tesszük, tehát előállítunk egy munka tömböt. A szorzáshoz szükséges időmennyiséget hasonlóképpen számítjuk, mint a másik metódusban, és ugyanabban a **time** adattagban tároljuk, mivel a felhasználó vagy mátrix-vektor, vagy mátrix-mátrix szorzást választhat, egyszerre a kettőt nem, így egymás szorzáshoz szükséges idejét sem befolyásolhatják.

4.2.2 CRS osztály

Ez az osztály fogja a „Tömörített sor tárolás” nevű tárolási formát megvalósítani. Hat példányszintű *Arraylist típusú* adattagja van az osztálynak: **value1, colind1, rowptr1, value2, colind2, rowptr2.**

Konstruktorok:

(1) public CompressedRow(int[,] matrix1, int[,] matrix2, int r, int c, int c2) :

base(matrix1, matrix2, r, c, c2):

Ez akkor fut le, ha a felhasználó a mátrix-mátrix szorzást választotta. Sor-folytonosan haladok a mátrix elemein, ha nem-nulla elemet találok, akkor annak értéke belekerül a megfelelő **value1** tömbbe, és a **colind1**-be pedig az elem oszlopindexét tesszük. Ha az oszlopindex 0, akkor a a sorok első nem-nulla elemének **value1** tömbbeli indexét a **rowptr1**-be helyezzük. Ezt megismételjük a másik mátrixra is.

A tárolás után a kövekező metódust hívom meg:

public int szorozmm(int r, int c, int c2):

A metódus visszatérési értéke a szorzás közben eltelt időtartam. Paraméterei a mátrixok méretei lesznek. Két segédtömböt használok, a **vectr-**et és a **vectc**-t. A **vectr**-ben tárolom az első mátrix sorait, a **vect-**ben pedig a másik oszlopait. A mátrixok szorzata pedig a **szorzat** kétdimenziós tömbbe kerül.

A vektorok inicializálása után egy ciklusban megyek végig a **rowptr1** elemein. Mindig két egymást követő elemet vizsgálok benne, melyek a **value1**-ben kijelölik az egy sorba tartozó értékeket. Ezeket az értékeket a hozzájuk tartozó **colind1**-ben tárolt indexű helyekre teszek. Ezzel előállítjuk az első mátrix sorait. A másik mátrix oszlopainak előállítása pedig úgy történik, hogy egy külső ciklusban haladok a mátrix oszlopindexein, majd az előbbiekhez hasonlóan jelölöm ki a sorokat a mátrixban. Mindig egy sorát vizsgálom, és az egy sorban található értékek közül azt amelyiknek oszlopindexe megegyezik a külső ciklus ciklusváltozójának értékével, azt beleteszem a **vectc**-be. Majd elvégzem a szorzásokat, melyek eredményeként feltöltődik a **szorzat** tömb.

A számítások kezdetekor tárolom az aktuális időt, és a végrehajtás után ismét lekérdezem a pontos időt. Majd a kettő különbségével megkapom a szorzáshoz szükséges időmennyiséget.

(2) public CompressedRow(int[,] matrix, int[] vector, int r, int c, int r2) : base(matrix, vector, r, c, r2):

Ahhoz, hogy lefusson, a felhasználónak a mátrix-vektor tab oldalt kell választania. A mátrix tárolása ugyanolyan elven történik, mint ahogy azt az előző konstruktorban leírtam. A tároláshoz az 1-es indexűadattagokat használom. A vektor tárolása pedig a fentebb ismertetett Coordinate osztály második konstruktorában szerepel.

A tárolások megvalósítása után a a következő metódust hívom meg:

public int szorozmv(int r, int c, int r2):

Működése hasonlatos az előzőleg említett metódussal, azzal a különbséggel, hogy a második mátrix oszlopainak kinyerése helyett egy munka tömbbe, amit szintén a **vectc**-vel azonosítok, visszaállítom az eredeti vektorom, és ezzel szorzok meg minden sort. Az eredmény egy vektor lesz. A szorzási időt minden esetben ugyanúgy számolom, úgy ahogy fentebb már leírtam.

4.2.3 CCS osztály

A "Tömörített oszlop tárolás"-t valósítja meg. Adattagjai a következők: **value1, rowind1, colptr1, value2, rowind2, colptr2.**

Az osztály ***konstruktorai:***

(1) public CompressedColumn(int[,] matrix1, int[,] matrix2, int r, int c, int c2)
: base(matrix1, matrix2, r, c, c2):

A mátrix elemein oszlopfolyatonosan haladok végig. Minden elemet vizsgálok, és ha értéke nem nulla, akkor a **value1-**hez hozzáveszem, valamint ezen elemem sorindexét a **rowind1**-hez teszem. Ha az oszlopindex zéró, akkor az oszlop első eleménét tárolom az előbbi módon, és **value1-**beli indexét a **colptr1-**hez hozzáveszem. Ugyanezt a folyamatot megismétlem a másik mátrixra is.

A konstruktor végén pedig meghívom a kövekező metódust:

public int szorozmm(int r, int c, int c2):

Ez a metódus fogja itt is a tényleges szorzást elvégezni. A segéd tömbök, melyek a sorokat és oszlopokat fogják tárolni, valamint a **szorzat** tömb ebben a metódusban is megjelenik. A tömbök inicializálása után, egy for ciklusban megyünk az első mátrix sorindexein végig. Egy belső ciklusban pedig két egymást követő **colptr1**-beli értéket veszünk, melyek a **value1** tömbben fognak egy oszlopot kijelölni, illetve annak nem-nulla elemeit. Ezekhez az elemekhez vizsgáljuk a velük megegyező indexű **rowind1**-beli elemeket, melyek értékei ha megegyeznek az aktuális sorindexszel, akkor az érték a **vectr rowind1** által meghatározott indexű helyére kerül. A sor előállítása után vissza kell állítani a másik mátrix összes oszlopvektorát, amit a **colptr2** segítségével teszek meg. Mivel minden egymást követő **colptr2**-ben szereplő a **value2**-ben egy intervallumot jelöl ki, mely éppen egy oszlop elemeit jelenti, így elég végig menni a **colptr2** struktúrán, és a **value2**-ből kinyerni az aktuális oszlop megfelelő értékeit. A tömbök előállítása után, összeszorozva őket megkapjuk a **szoroz** tömb egy elemét. Az összes sor és oszlop előállítása és szorzása után megkapjuk a két mátrix szorzatát. A szorzáshoz szükséges idő tárolása után már csak az eredmény kiiratása, illetve ha van még bejelölt tárolási forma melyben szükséges kiszámítani az időtartamot, akkor azok elvégzése van hátra.

(2) public CompressedColumn(int[,] matrix, int[] vector, int r, int c, int r2)
: base(matrix, vector, r, c, r2):

A mátrix tárolását ugyanúgy konstruálom,m int az előző konstruktornál, jellemzői az 1-es indexű adattagokba kerülnek. A vektor tárolása, pedig ugyanolyan módon történik, mint azt a fentebbi két osztálynál már láthattuk.
A konstruktor végén a szokásos public int szorozmv(int r, int c, int r2) metódus meghívásával számítjuk ki a két struktúra szorzatát, és a szorzáshoz szükséges időt. A szorzat előállításánál a mátrixból a sorokat, a *szorzatmm* metódus ismertetésénél leírt móddal megegyezően végzem.

4.2.4 ELL osztály

Az “Ellpack” tömörítési formát implementálja. A szokásos adattagokon kívül a következőkkel dolgozik: **value1, colind1, value2, colind2, max1, max2.** Ezek közül az első négy kétdimenziós tömb, míg az utolsó kettő egész típusú változók.

Konstruktorai:

(1) public Ell(int[,] matrix1, int[,] matrix2, int r, int c, int c2)
: base(matrix1, matrix2, r, c, c2):

A mátrix soraiban szereplő nem-nulla értékek száma szerint csökkenő sorrendbe kell rendezni a sorokat. A **max1,** és **max2** adattagok fogják tárolni, hogy a mátrixok egy sorban maximum mennyi nem-nulla elemet tárolnak. Egy segéd változót hazsnálok, mely számolja az egyes sorokban szereplő nem zérus értékeket, és ezt hasonlítom a **max1** értékekkel. Ha a segéd változó értéke nagyobb, akkor értékül adom a **max1** adattagnak. Ezen változók segítségével határozom meg, hogy a **value1** és **colind1** kétdimenziós tömbök hány oszlopból állnak. Majd két ciklus segítségével haladok a mátrixon sor folytonosan, és minden sor nem-nulla értékeit szekvenciálisan belerakom a **value1** tömbbe. Ha egy sorban nincs több nem zéró elem, akkor a maradék helyeket feltöltöm nullákkal. A **value1** tömb feltöltésével párhuzamosan történik a **colind1** tömb feltöltése. Minden esetben, ha nem-nulla elemhez érek a mátrixban, az érték tárolása mellett az elem oszlopindexét is tároljuk a **colind1**-ben. Ha nincs több nem-nulla elem egy sorban, akkor az utoljára letárolt oszlopindexet fogom minden későbbi pozícióba elhelyezni. Ugyanezeket a lépéseket megismételve a második mátrixra, megvalósítottuk a tárolást. Majd a *szorozmm* metódus meghívása következik:

public int szorozmm(int r, int c, int c2):

A szokásos segédtömböket használjuk az eredeti mátrixok sorainak és oszlopainak előállítására. Az első mátrix sorainak visszaállítása úgy történik, hogy megyek végig egy for

ciklusban a sorindexeken, és a **value1** tömb aktuális sorának elemeit vizsgálom. Ha az aktuális elem nem nulla, akkor a **vectr**-ben a helye a **colind1** tömbben hozzátartozó index lesz. Oszlopok visszaállításánál egy külső for ciklus segítségével haladok a **colind2** tömb oszlopindexein, majd egy belső ciklusban pedig ugyanezen tömb sorainak elemeit vizsgálom. Ha a sor elemének értéke egyenlő a külső ciklus változójával, és nem egyezik meg a sorban előtte álló elem értékével, akkor beleteszem a **vectc** segéd vektoromba a belső ciklus változója által meghatározott helyre. A teljes oszlopvektor visszaállítása után a szorzást elvégezzük, majd ugyanezt a műveletet elvégezzük úgy, hogy minden sort szorzunk minden oszloppal. A szorzáshoz szükséges időmennyiséget pedig tárolom a már említett **time** változóban.

(2) public Ell(int[,] matrix, int[] vector, int r, int c, int r2)
: base(matrix, vector, r, c, r2)*:*

A másik konstruktorhoz valósítja meg a mátrix tárlását, valamint már ismertettem a vektor tárolását is. Majd a két adatstruktúra szorzását a *public int szorozmv(int r, int c, int r2)* metódus végzi, melyben a mátrix minden sorát úgy állítunk vissza, mint ahogy azt a *szorozmm* függvényben tettük.

4.2.5 Sparsediag osztály

Adattagjai a következők: van két egész típusú kétdimenziós tömb **(value1, value2),** valamit két Arraylist típusú struktúra **(diagind1, diagind2),** és két egész változó **(diag1, diag2).**

Konstruktorai:

(1) public Sparsediag(int[,] matrix1, int[,] matrix2, int r, int c, int c2)
: base(matrix1, matrix2, r, c, c2):

public void feltölt(int[,] matrix1, int[,] matrix2, int r):

A *feltolt* metódus meghívásával implementálom a tárolási formát, illetve adom meg a példányszintű adattagok értékeit. Három segéd *Arraylist* típusú struktúrát használok:

- *diagonal:* rendre megkapja értékül a mátrix összes diagonálisát, úgy hogy alulról haladunk a mátrixban felfele.
- *v1:* ezen struktúra fogja az első mátrix összes diagonálisát tartalmazni, tehát az összes *diagonal* által felvett értéket tárolja. Még azokat az átlókat is, melyek csak zéró elemet tartalmaznak.
- *v2:* ugyanazt a célt szolgálja, mint a *v1,* csak nem az első, hanem a második mátrix diagonálisait tárolja.

A *diagonal* dinamikus tömb feltöltésekor, az aktuális átlóbeli elemeket beletesszük, mégpedig úgy, hogy ha a főátló alatt helyezkedik el, akkor annyi -1-es értéket veszünk fel, amennyivel alatta van, és a -1-ek után kerülnek be az átló értékei. Ha a főátló felett van, akkor előbb az aktuális átló elemei kerülnek a *diagonal*ba, majd annyi -1-es érték, amennyivel a főátló felett van. Ha a *diagonal* struktúrát feltöltöttük, akkor hozzáadjuk a *v1* struktúrához. Minden átló hozzáadásakor számoljuk a benne lévő -1-ket. Ha főátló alatti átló elemei kerültek a *diagonal*ba, akkor -1-esek darabszámát szorozni kell -1-el, és ez az érték kerül a **diagind1**-be. Ha főátló feletti, akkor csak a darabszámot adjuk hozzá.
A *v1* kétdimenziós segédtömb feltöltése után azokat az oszlopokat melyek csak 0 és -1-es értékeket tartalmaznak, töröljük a struktúrából, illetve a hozzá tartozó pozitív vagy negatív értéket pedig a **diagind1**-ből. Ha minden ilyen átlót töröltünk, akkor a *v1* értékeit másoljuk a **value1**-be. Ugyanezt megismételve felötöljük a **value2** és **diagind2** adatstruktúrákat.

A mátrix-mátrix szorzást a következő metódus végzi:

public int szorozmm(int r):

Ha a **value1** tömbben benne van az az összes átló, azok is melyek csak zéró elemeket tartalmaznak, akkor a mátrix egy-egy sorát úgy nyerhetjük ki, ha vesszük a **value1**

kétdimenziós tömb egy-egy sorának nem -1-es értékű elemeit. Mivel a **value1** tömb nem tartalmazza az összes átlót, így nem is fogja egy sorban a mátrix teljes sorát tartalmazni. Minden hiányzó elem helyére nullát teszünk a **vectr** tömbben. A **diagind1** vektor segít abban, hogy megtudjuk mely átlók hiányoznak a struktúrából.

Oszlopok előállítása: ha az összes átlót tartalmazná a **value2** adatstruktúra, akkor annak azon mellékátlói, melyek nem tartalmaznak -1-es értéket, lesznek az eredeti mátrix oszlopai. Itt is a hiányzó értékek helyére nullát írunk a **vectc** segédtömbben. Majd a szorzások elvégzésével előállítható a két mátrix szorzata.

4.2.6 Jagged osztály

A Jagged diagonális tárolási sémát implementálja, melyhez a következő adattagok szükségesek, a már fent említetteken kívül: hat Arraylist típusú, ezek a **value1, colind1, diagptr1, value2, colind2, diagptr2**; valamint két egész típusú egy dimenziós tömb: **perm1, perm2.**

Konstruktorai:

(1) public Jagged(int[,] matrix1, int[,] matrix2, int r, int c, int c2)
: base(matrix1, matrix2, r, c, c2):

A konstruktorból csak három metódushívás történik, ből kettő a két mátrix tárolását valósítja meg, míg a harmadik a szorzást végzi. Metódusok:

public void feltölt(int[,] matrix, int r, int c):

Négy segédtömböt használok a megvalósításhoz: *rows, col1, copym1, copyc1.*

- *rows:* hossza megegyezik a sorok számával. A mátrix egy-egy soraiban lévő nem-nulla elemek számát tárolja.
- *col1:* a mátrix elemeinek oszlopindexeit tartalmazza, segít majd a *copyc1* feltöltéséhez.

- *copym1:* a mátrix elemeit eltoltan tárolja.
- *copyc1:* a mátrix elemeinek oszlopindexeit tartalmazza eltoltan.

A *rows* és a *col1* struktúrák feltöltése egyszerre történik. Majd a *col1* segítségével feltöltöm a másik két struktúrát. Ezt követően a *rows* használatával elvégzem a mátrix sorainak permutációját. És feltöltöm a tárolás megvalósításához deklarált adattagokat. Majd a szorzást a már ismert metódus végzi:

public int szorozmm(int r, int c, int c2):

Az osztály adattagjai közül a **diagptr1** elemei fogják az eltolt mátrix oszlopainak első elemeit meghatározni. Mindig két **diagptr1** által meghatározott elem közötti értékeket vizsgálom a **value1** adatstruktúrában. Ezen elemekhez nézem a **perm1**-ben hozzájuk tartozó értékeket, melyek ha megegyeznek az aktuális sorindexszel, akkor belekerülnek a **vectr** segédtömbbe, a **colind1** által jelölt helyre. Minden sorhoz elő kell állítani a másik mátrix oszlopait, ez a következőképpen történik: itt is két **diagptr2** által meghatározott **value2-**beli elemeket vizsgálok. Ha a hozzájuk tartozó oszlopindex megegyezik az aktuálissal, akkor a **vectc**-ben az értékhez tartozó **perm2** által meghatározott helyre kerül. Ezen vektorok segítségével elvégezem a két mátrix szorzását.

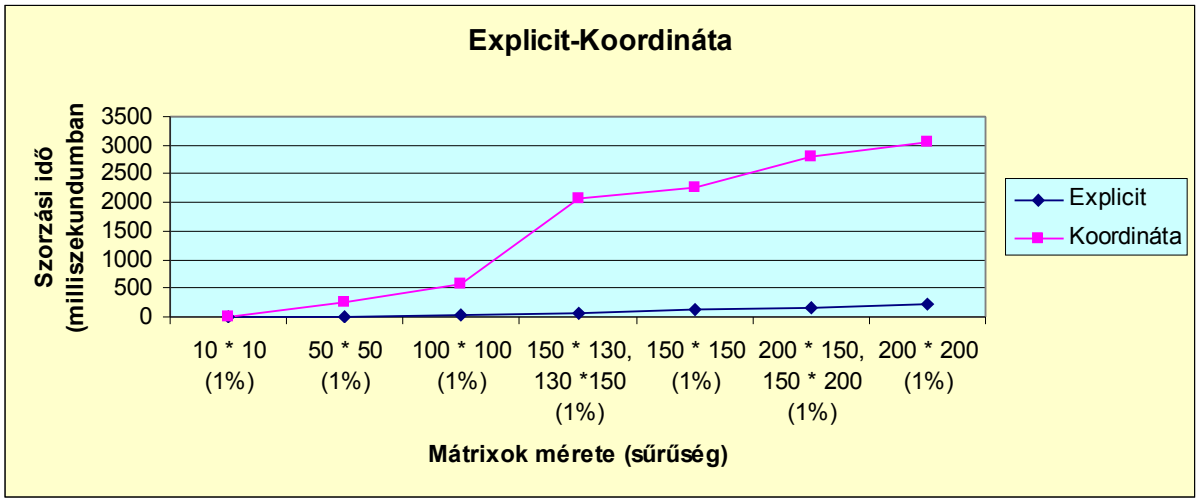
A másik konstruktora hasonlóképpen valósítja meg a mátrix tárolását, illetve a vektor tárolási módja is fentebb már le lett írva. A szorzás is hasonlóképpen működik a *szorozmm* metódus ismertetésénél leírtakkal, azzal a különbséggel, hogy itt minden sort szorzok a vektorral.

5. Alkalmazás tesztelése

5.1 Mátrix-mátrix szorzás tesztelése

Ebben a fejezetben az általam írt program tesztelésének jelentősebb eredményeiről szeretnék írni. Egyrészt, hasonlítani kívánom, minden tárolási formában elvégzett szorzáshoz szükséges időmennyiséget az explicit tárolással tárolt mátrixok szorzásánál mért idővel. Másrészt néhány tárolási sémát is összevetek.

Először a Koordináta sémát hasonlítom az explicit tárolással. Ennek eredménye az alábbi grafikonon látható:

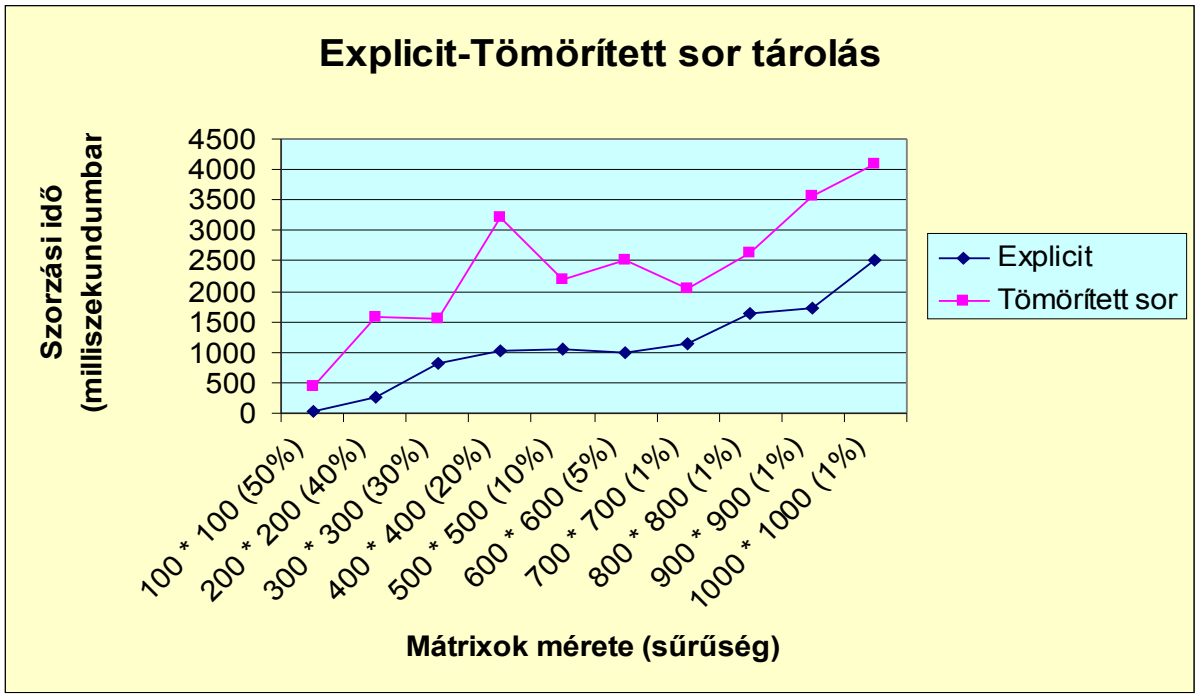


1. ábra Explicit és Koordináta sémák hasonlítása (mátrix-mátrix)

Bár az ábrán csak maximum 200 × 200-as mátrixok esetén látható a szorzáshoz szükséges időmennyiség, de nagyobb méretű mátrixok esetén is ekkora mértékű időbeli növekedés figyelhető meg a Koordináta séma esetében. Tesztemben csak 1% sűrűségű adatstruktúrákat vizsgáltam, de nagyobb százalék mellett, még több időt igényel a szorzási eredmény kiszámítása ezen tárolási séma esetén. Mivel a programom csak maximum 1000 × 1000-es mátrixokra futtatható, így nem tudom, hogy ettől nagyobb méretű struktúrák esetén hogyan változik ez az érték. Elképzelhető, hogy több ezer sorból és oszlopból generált

mátrixoknál már hatékonyabbá válik a Koordináta séma használata, és érdemesebb ezt alkalmazni, nem pedig a direkt tárolást.

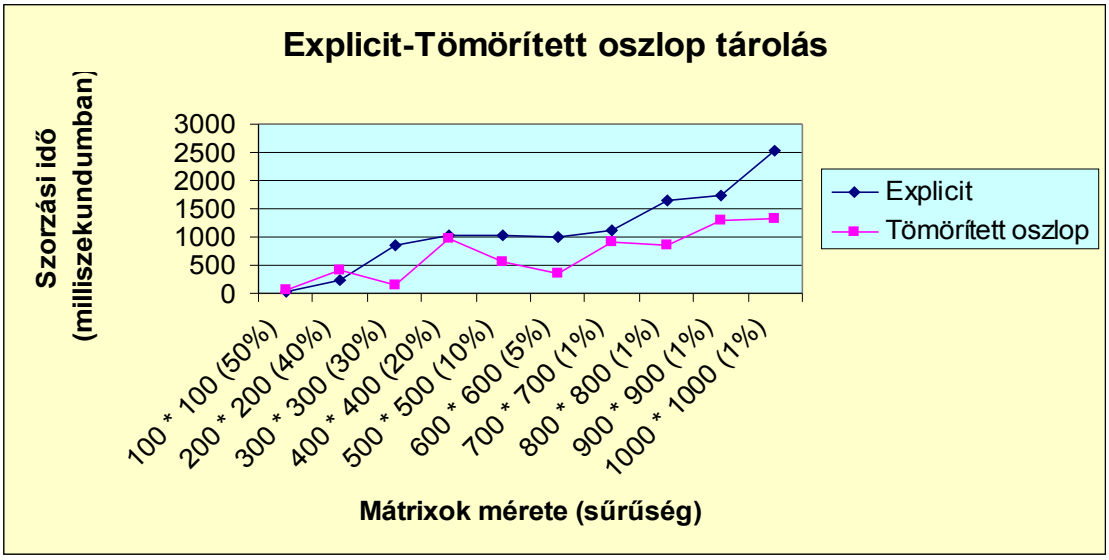
A következő vizsgált séma a Tömörített sor tárolás:



2. ábra Explicit és CRS sémák hasonlítása (mátrix-mátrix)

Teszteléskor fokozatosan növeltem a mátrixok méretét, és ezzel együtt csökkentettem a sűrűséget. Érzékelhető, hogy ez a séma már valamivel hatékonyabb a Koordináta sémától, azonban 1000 × 1000-es struktúrák esetén még ez sem válik igazán hatákonnyá.

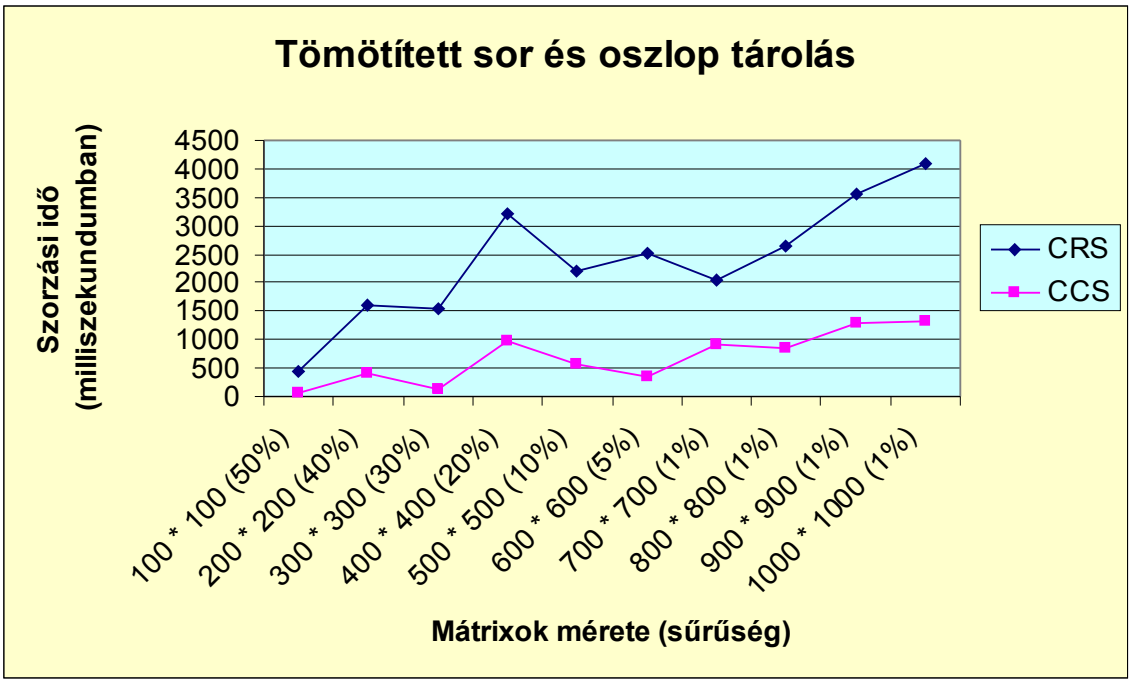
A következő vizsgált tárolási séma a Tömörített oszlop tárolás:



3. ábra Explicit és CCS sémák hasonlítása (mátrix-mátrix)

Kezdetben egészen kicsi és viszonylag nagy sűrűségű mátrixokra futtattam az alkalmazást. Ezekben az esetekben még gyorsabbnak bizonyult a direkt tárolási mód. Azonban miután növelni kezdtem a méretet és csökkenteni a sűrűségi mutatót, egyre hatékonyabbá vált a CCS séma. A grafikonon is látható, hogy 200 × 200-as mátrixoknál még a direkt tárolási móddal történő szorzás gyorsabb. 400 × 400-as struktúráknál a szorzáshoz szükséges időmennyiség egyenlő mindkét esetben. Ettől nagyobb mátrixok esetében viszont már a CCS mindig gyorsabb az explicitnél.

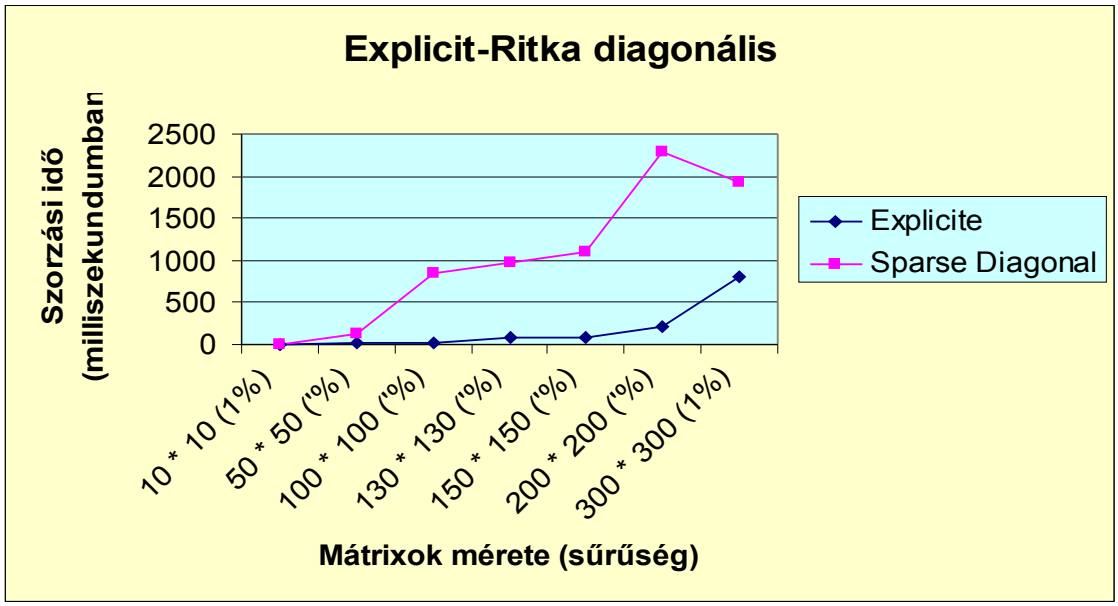
Már a tárolási formák bemutatásakor is említettem, hogy a CRS-el teljesen analóg módon működik a CCS séma, ám ennek ellenére érdemes megfigyelni, hogy az analógia és a hasonló implementálás ellenére mennyire szembetűnő a szorzás elvégzéséhez szükséges időbeli különbség.



4. ábra CRS és CCS sémák hasonlítása (mátrix-mátrix)

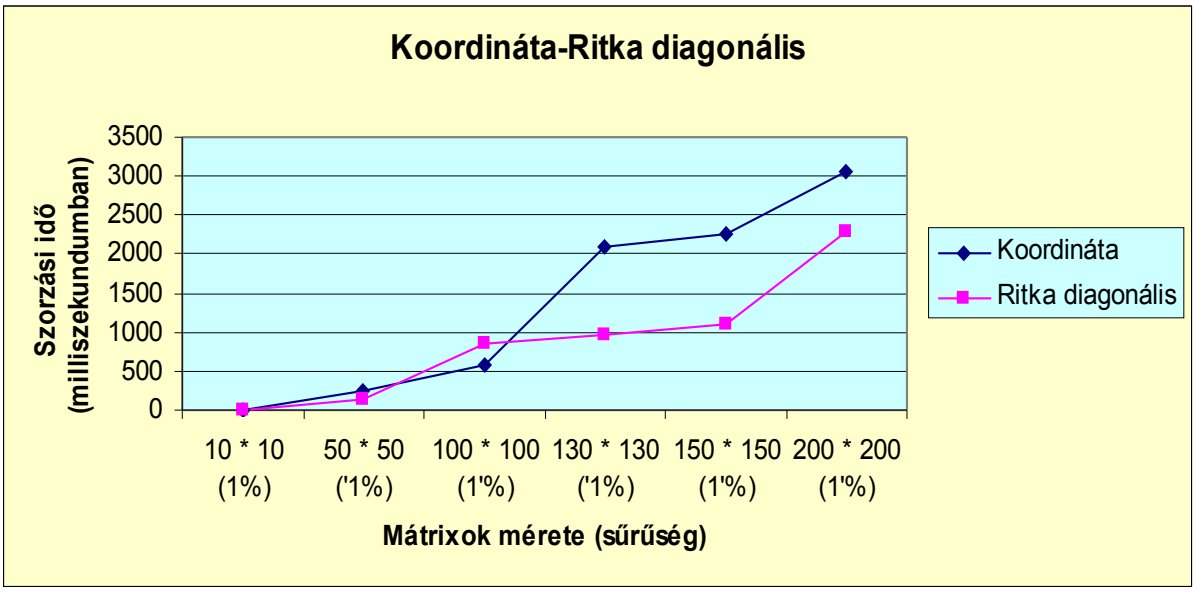
Minden egyes tesztesetnél hatalmas időbeli különbségek észlelhetők, melyeknek okát nem tudom. Lényegesen gyorsabb a CCS a CRS tárolási sémánál.

A következő tesztelt séma a Ritka diagonális, mely szintén nem válik igazán hatékonnyá.



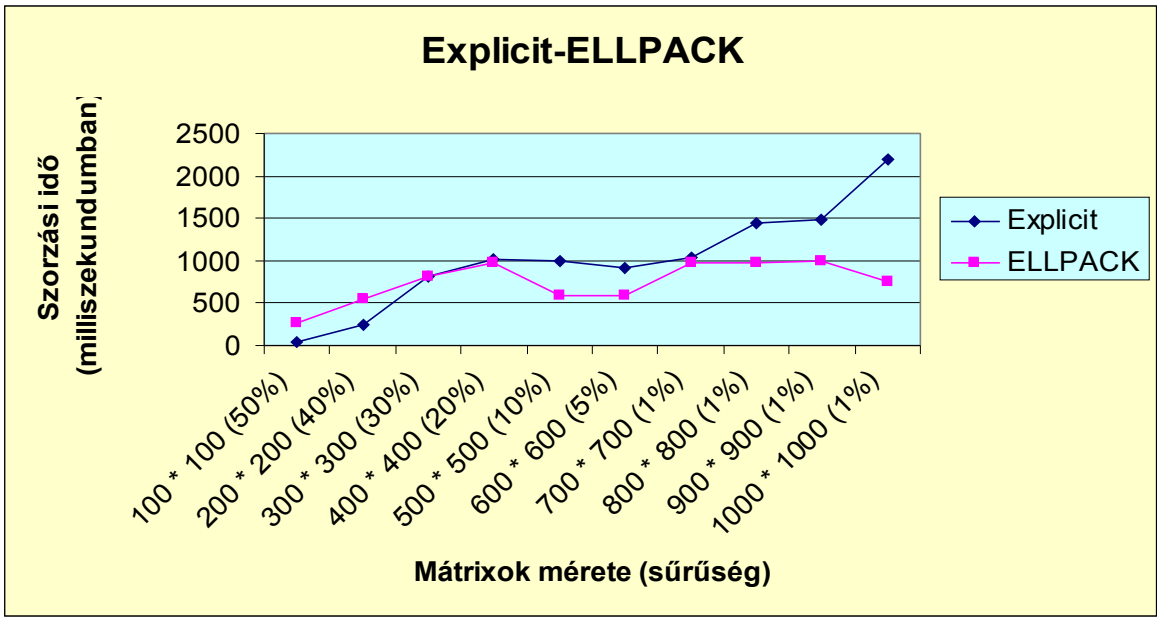
5. ábra Explicit és Ritka diagonális sémák hasonlítása (mátrix-mátrix)

Ez a séma sem válik hatékonnyá még maximum 1000 × 1000-es mátrixok esetén, azonban összehasonlítva a Koordináta séma hatékonyságával, azt az eredményt kapjuk, hogy attól gyorsabban számítja ki a struktúrák szorzatát. A hasonlítás eredménye az alábbi grafikonon látható:



6. ábra Koordináta és Ritka diagonális sémák hasonlítása (mátrix-mátrix)

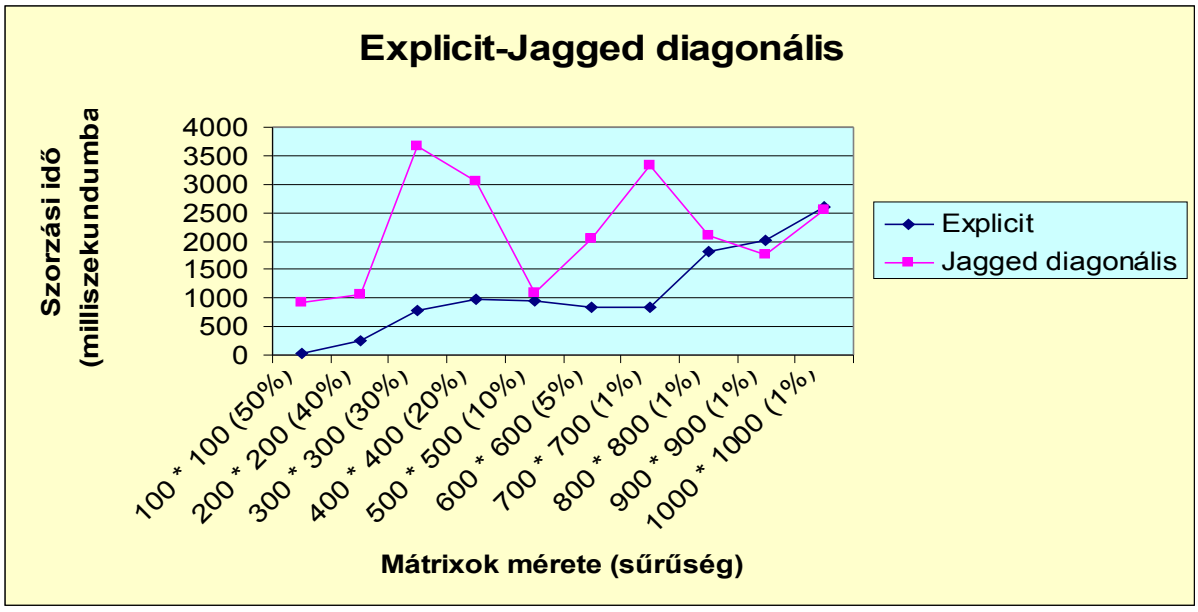
Az ezt követő tesztelés alá vetett tárolási séma az ELLPACK séma, melynek eredményei:



7. ábra Explicit és ELLPACK sémák hasonlítása (mátrix-mátrix)

A grafikont vizsgálva szembetűnő, hogy kis méretű, kis ritkasági együtthatójú mátrixok esetén még a direkt tárolás a hatékonyabb. 300 × 300-as méretű és 30%-os sűrűségű mátrixok esetén a két tárolási séma közel azonos hatékonysággal számítja ki a szorzái eredményt. Ettől nagyobb méretű és kisebb sűrűségi együtthatójú struktúrák esetén azonban lényegesen gyorsabbá válik az ELLPACK séma.

A következő tárolási séma amely implementálásra került, a Jagged diagonális, melynek tesztelése során a következő eredményeket kaptam:

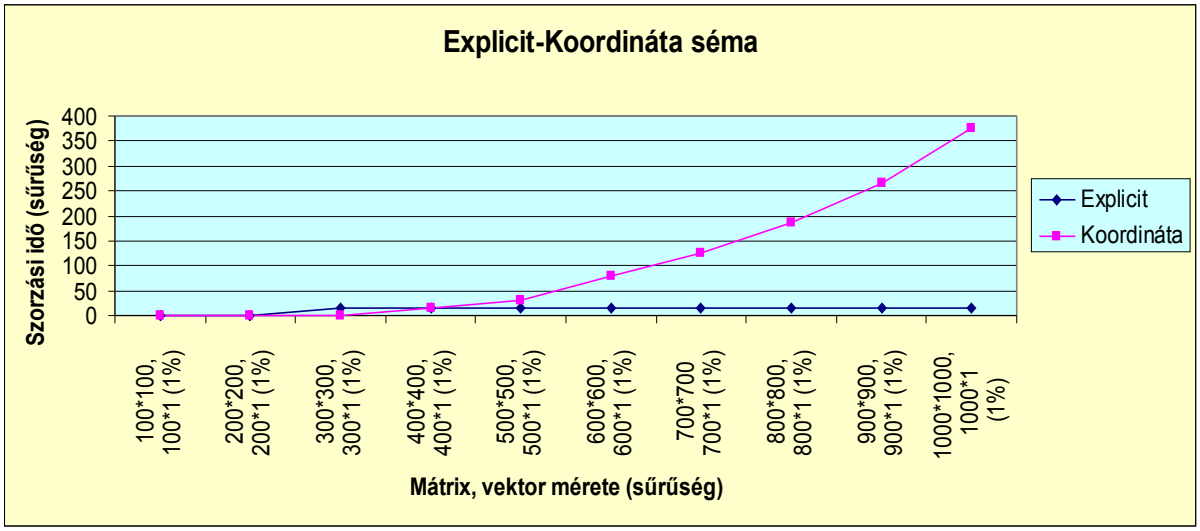


8. ábra Explicit és Jagged diagonális sémák hasonlítása (mátrix-mátrix)

Erre a sémára is jellemző, hogy majd csak 1000-től több sort és oszlopot tartalmazó mátrixok esetén válik igazán a direkt tárolástól gyorsabbá. Már 900 × 900-as és 1000 × 1000-es méret esetén is kicsivel gyorsabbnak mondható ez a tárolási mód. Megfigyelhetjük, hogy a 800 × 800-as mátrix méretig nagyon kiugró értékek születtek, ám ettől a ponttól kezdve csak kis értékbeli ingadozás mutatkozik.

5.2 Mátrix-vektor szorzás tesztelése

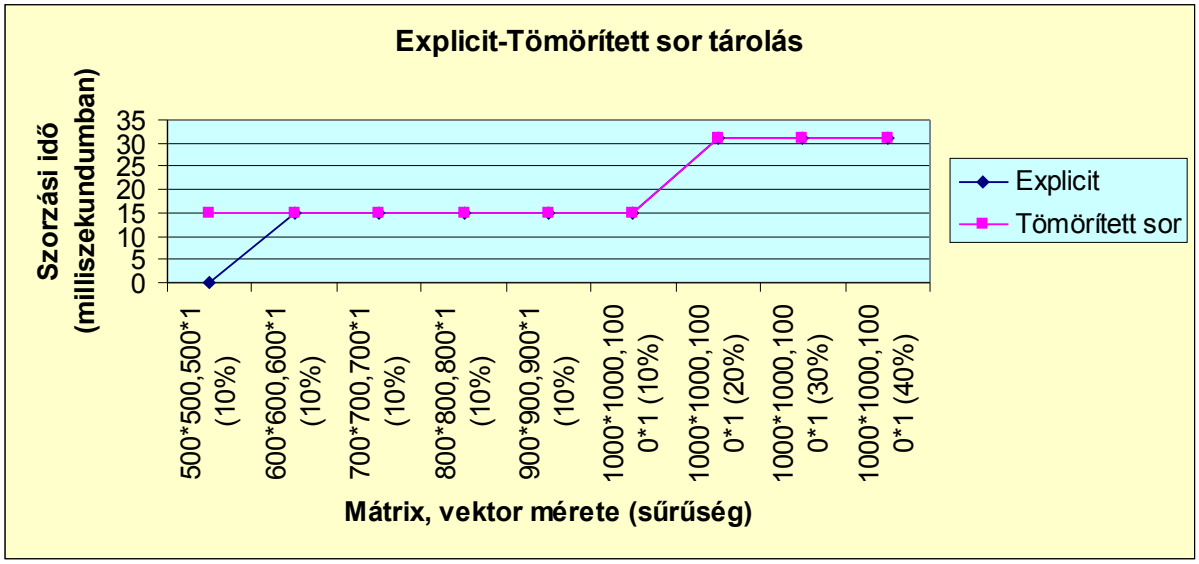
Először itt is a Koordináta sémát teszteltem, mely, látható, hogy szintén nem igazán hatékony, ugyanúgy, mint mátrix-mátrix szorzás esetén. Az eredmény a következő:



9. ábra Explicit és Koordináta sémák hasonlítása (mátrix-vektor)

Direkt tárolásnál tetszőleges méreteket választva sem kaptam 15 milliszekdundumnál nagyobb időmennyiséget, viszont a Koordináta séma egészen hatákonynak bizonyult 400 × 400-as méretű mátrixokig, ám ha ettől nagyobbnak választottam a méretet, akkor már az explicit tárolás vált lényegesen gyorsabbá.

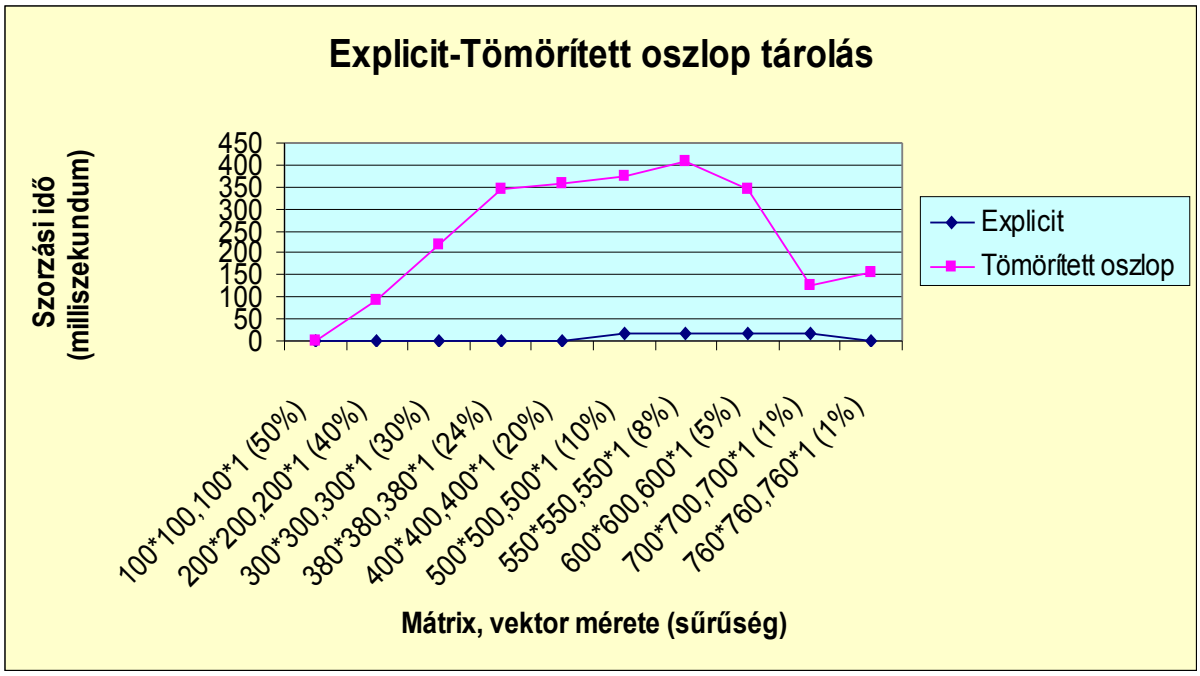
A következő vizsgált a CRS séma:



10. ábra Explicit és CRS sémák hasonlítása (mátrix-vektor)

Számos esetben teszteltem a sémát, de jelentősebb eltérést nem mutatott a direkt tárolástól. Ha átlagosan nézem a szorzáshoz szükséges időintervallumot, akkor a két séma esetében közel azonos értékeket kapok. Elképzelhető, hogy még nagyobb adatstruktúrák esetén vizsgálva megfigyelhető lenne, hogy elválik a két tárolási mód, és hatákonyabbá valik a CRS séma.

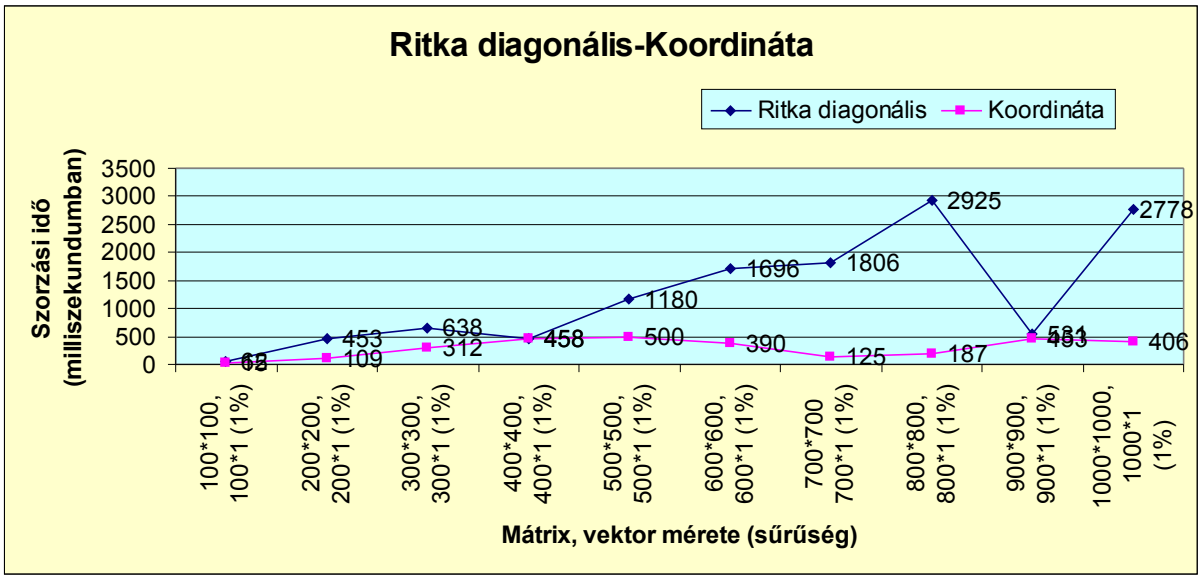
A Tömörített oszlop tárolási sémánál az alábbi eredményeket kaptam:



11. ábra Explicit és CCS sémák hasonlítása (mátrix-vektor)

Várakozásaimmal ellentétben ennek a tárolási sémának a használata sem gyorsítja meg a mátrix-vektor szorzást 1000 × 1000-esnél kisebb vagy éppen ekkora méretű mátrixok esetén. Úgy gondoltam, ha a mátrix-mátrix szorzásnál egészen hatékony ez a séma, akkor ennél a szorzásnál is azzá válik, de a tesztesetek mást mutatnak.

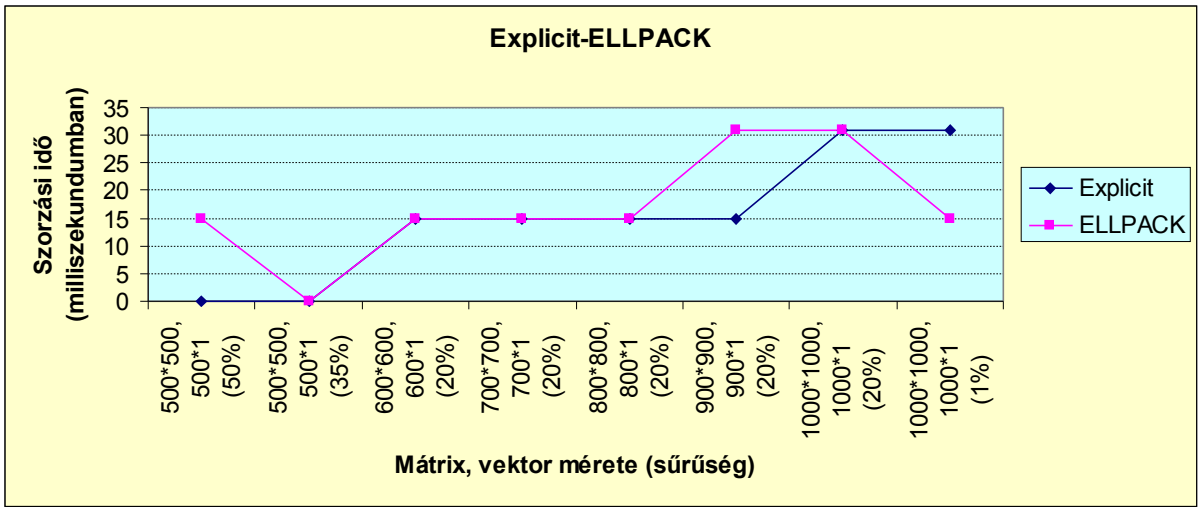
Mivel a mátrix-mátrix szorzásnál a két legkevésbé hatékony tárolási módnak a Koordináta séma és a Ritka diagonális bizonyult, így most ezt a két sémát hasonlítom össze, mivel feltételezem, hogy a Ritka diagonális most is jelentősen lassabb a direkt tárolásnál. Az összevetés eredménye:



12. ábra Ritka diagonális és Koordináta sémák hasonlítása (mátrix-vektor)

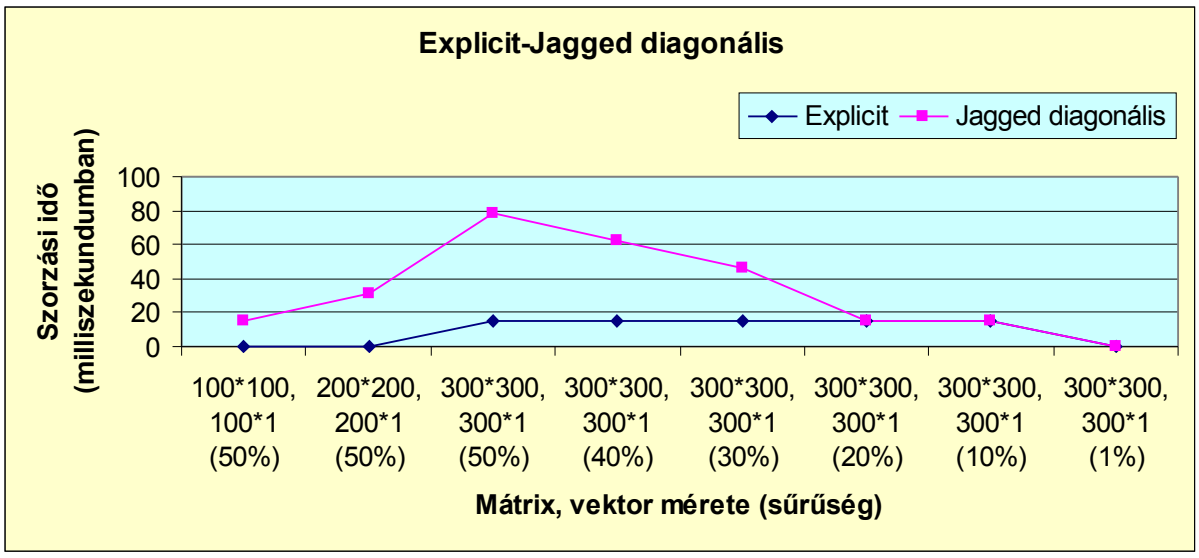
Míg mátrix-mátrix szorzás esetén a Ritka diagonális forma bizonyult gyorsabbnak, addig a mátrix-vektor szorzás esetében az ábrán is látható, hogy a Koordináta séma. Annak ellenére van különbség, hogy a mátrix-mátrix és a vektorral történő szorzás esetén ugyanúgy történik a mátrix sorainak visszaállítása. Mindezek mellett a két séma esetén a munkavektorok elkészítése is azonos módon megy.

A következő vizsgált séma az ELLPACK, mely a mátrix-vektor szorzás esetében eddig a leghatékonyabb tárolási forma a CRS mellett. Igazán jelentős időbeli különbség nincs az Explicit és az ELLPACK sémák között, ám 800 × 800-as méretű mátrixoktól nagyobb méretűekre, többszöri futtatás után azt az eredményt láttam, hogy egyre többször, kevesebb idő szükséges a szorzás elvégzéséhez az ELLPACK sémában tárolt mátrixoknál, mint a direkten tároltaknál. Az alábbi ábrán minden méret és sűrűség mellett a leggyakoribb eredmények láthatók:



13. ábra Explicit és ELLPACK sémák hasonlítása (mátrix-vektor)

Az utolsó tesztelt séma a Jagged diagonális:



14. ábra Explicit és Jagged diagonális sémák hasonlítása (mátrix-vektor)

Elsőként viszonylag nagy sűrűségi együtthatójú mátrixokra teszteltem a sémát. A sűrűség változtatása nélkül a mátrixok méretét növelve megfigyelhetjük, hogy a szorzáshoz szükséges időmennyiség gyorsabb mértékben növekedett a Jagged diagonális tárolási sémánál, mint a direkt tárolásnál. Majd a méretet változatlanul hagyva, fokozatosan csökkenteni kezdtem a sűrűséget, melynek eredménye az igényelt idő csökkenése. Ám a kis

sűrűség és a vizsgálatok során tesztelt egyre nagyobb mátrixok esetén sem juthatunk arra a következtetésre, hogy ez a tárolási mód lényegesen hatékonyabbá válna 1000 × 1000-es vagy attól kisebb mátrixokra. Számos esetben gyorsabb a direkt tárolásnál, de ennek ellenkezője is megfigyelhető még.

5.3 Vektor-vektor szorzás tesztelése

Ennek a szorzási formának a tesztelése során arra figyelhetünk fel, hogy igazán eltérés nem mutatkozik a sűrített és a direkt tárolás között. 1000 × 1000-es mátrixok esetén egészen 10%-os sűrűségig mindkét tárolási módnál a szorzási idő 0 milliszekundum. Mivel a méretet már nem tudta növelni, mert az alkalmazásomban a limit 1000, így a sűrűséget kezdtem egyenletesen emelni. Ekkor nagyon kis mértékben nőtt a sűrített tárolásnál végzett szorzáshoz szükséges idő. Viszont mivel a tárolási formának nem nagy sűrűségi együtthatójú vektorok esetén kell hatékonyan működnie, így 50%-os sűrűségi mutatótól nagyobb értékeket nem is vizsgáltam. Ekkor a legtöbb esetben 31 milliszekundum volt az az időmennyiség, mely a szorzások elvégzéséhez kellett. Nagy valószínűséggel több ezres méretű és kis sűrűségi mutatójú vektorok esetén elválik a két tárolási mód egymástól, és hatékonyabbá válik a sűrített tárolás.

6. Összefoglalás

Diplomamunkám célja a különböző rikta mátrix tárolási sémák ismertetése, valamint egy oktatási célú szoftver fejlesztése, melyben az ismertetett sémák közül néhány kiválasztott tesztelése végezhető el.

Alkalmazásom tesztelése során arra a következtetésre jutottam, hogy a jelenleg implementálásra került tárolási sémák közül a leghatékonyabb az ELLPACK séma. Mind a mátrix-mátrix, mind a mátrix-vektor szorzásoknál gyorsabb a direkt, és a többi tárolási módnál.

Bár az irodalmakban olvasott gyorsítási lehetőségeket alkalmaztam a programomban, mégis látható, hogy néhány séma egyáltalán nem válik hatékonnyá 1000 × 1000-es vagy attól kisebb méretű mátrixok esetén. Így a szoftver tökéletesítése, bővítése, a tárolási módok hatékonyabbá tétele további munkát igényel. Valamint jövőbeli fejlesztés tárgyát képezi az interfész fejlesztése is.

7. Irodalomjegyzék

[1] Bajalinov Erik – Imreh Balázs, „Operációkutatás”, POLYGON Szeged, 2001

[2] Bajalinov,E.B., ”Linear-Fractional Programming: Theory, Methods, Applications and Software”, Kluwer Academic Publishers, 2003.

[3] www.mathematik.hu-berlin.de/~gaggle/EVENTS/2006/BRENT60/presentations/Shahadat%20Hossain%20-%20On%20effic..

[4] www.ir.iit.edu/publications/downloads/goharian_ITCC01.pdf

[5] www.ir.iit.edu/publications/downloads/SCI-Journal-CameraReady-Goharian.pdf

[6] http://www.csit.fsu.edu/~gallivan/courses/NLA2/set2.pdf

[7] http://www.capsl.udel.edu/courses/eleg652/2006/homework/hmk1.pdf

[8] www.tacc.utexas.edu/~eijkhout/Articles/2004-buttari-spmvp.pdf

[9] www.nag.co.uk/numeric/FN/manual/pdf/c04/c04int_fn04.pdf

[10]

http://www.intel.com/software/products/mkl/docs/webhelp/appendices/mkl_appa_smsf.html

[11] http://delivery.acm.org/10.1145/1070000/1062299/p230-silva.pdf?key1=1062299&key2=1101005711&coll=GUIDE&dl=GUIDE&CFID=18381832&CFTOKEN=84444510

[12] www.mgnet.org/mgnet/papers/SparKer/sparker3.ps.gz

[13] http://www.cs.utk.edu/~dongarra/etemplates/node383.html

[14] http://192.18.109.11/816-0652-10/816-0652-10.pdf

[15] http://192.18.109.11/816-0652-10/816-0652-10.pdf

[16] http://www.cs.utk.edu/~dongarra/etemplates/node373.html

[17] http://www.cs.utk.edu/~dongarra/etemplates/node375.html

[18] http://www.netlib.org/linalg/html_templates/node94.html

[19] http://www.cs.utk.edu/~dongarra/etemplates/node377.html

[20] http://www.hlrs.de/people/sunil/PUBS/ICCS06.pdf

[21] http://www.capsl.udel.edu/courses/eleg652/2006/homework/hmk1.pdf

[22] http://www.ida.liu.se/~chrke/sparamat/iwpc.pdf

[23] www10.informatik.uni-erlangen.de/Teaching/Courses/STiSC/2006-11-07/matrix.pdf

[24] www.tacc.utexas.edu/~eijkhout/Articles/2004-buttari-spmvp.pdf

8. Ábrák és táblázatok jegyzéke

ÁBRÁK:

Oldal

TÁBLÁZATOK:

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek Dr Bajalinov Erik tudományos főmunkatársnak a diplomamunkám megírásához nyújtott segítségéért.

Tisztelettel: Zabos Katalin