Sparse Matrices

Introduction

In economic applications it's not uncommon to have matrices with a large number of zero-valued elements and, because C/C++ stores zeros in the same way it stores any other numeric value, these elements can use memory space unnecessarily and can sometimes require extra computing time. Examples of applications that use sparse matrices: graph stored as adjacency matrices, optimization problems using linear algebra and sparse linear equation systems.

Sparse matrices provide a way to store data that has a large percentage of zero elements more efficiently. While full matrices internally store every element in memory regardless of value, sparse matrices store only the nonzero elements and their position. Using sparse matrices can significantly reduce the amount of memory required for data storage.

There are two main advantages of using sparse matrices:

- greatly reduced memory requirements
- faster computations by eliminating operations on zero elements

Because sparse matrices store both element values and position for non-zero elements, they require more storage space if number of such elements isn’t much smaller than zero elements. Another drawback of using a sparse matrix is that it doesn’t provide direct access to elements. Basic operations are also more difficult to implement compared to normal matrices.

These tradeoffs must be well considered when deciding whether to use a normal or a sparse matrix. Usually sparse matrices are used when dealing with large volumes of data that contain between 0.15% and 3% non-zero elements.

Storage

When storing a sparse matrix we need to consider two kinds of information:

- the values for non-zero elements

- positioning of these element in the matrix

There are many ways to store a sparse matrix in memory:

a) Using three arrays for line indices, column indices and values. The first two arrays can be replaced by a single array containing the index of the elements in the linearized matrix.
b) By grouping the indices and value triplets in a structure and storing them into an array or a linked list.
c) Using a bit array for marking the positioning of the non-zero elements in the linearized matrix and a separate array for values.

In the following sections we'll use the first form (we'll use three dynamically allocated arrays for storing line indices, column indices and values). For easier manipulation all the information related to a matrix is grouped into a C structure:

```
struct MatriceRara
{
      // vectorii pentru stocarea liniei/coloanei si valorii
      int *Linie, *Coloana, *Valoare;

      // numarul de elemente nenule din matrice
      int nrElemente, nrLinii, nrColoane;
};
```

Conversions

The conversions between normal matrices and sparse matrices and vice versa are straightforward:

```
// Conversie matrice normala -> matrice rara
MatriceRara Conversie(int **mat, int m, int n)
{
      MatriceRara matR;

      // determinarea numarului de elemente
      int i, j, nrElemente = 0, indiceMatR = 0;
      for (i = 0; i < m; i++)
            for (j = 0; j < n; j++)
                  if (mat[i][j] != 0)
                        nrElemente++;

      // initializare matrice rara
      matR.Linie = new int[nrElemente];
      matR.Coloana = new int[nrElemente];
      matR.Valoare = new int[nrElemente];
      matR.nrElemente = nrElemente;
      matR.nrLinii = m;
```

```
	matR.nrColoane = n;

	// copierea valorilor nenule
	for (i = 0; i < m; i++)
		for (j = 0; j < n; j++)
			if (mat[i][j] != 0)
			{
				matR.Linie[indiceMatR] = i;
				matR.Coloana[indiceMatR] = j;
				matR.Valoare[indiceMatR] = mat[i][j];

				indiceMatR++;
			}

	return matR;
}

// Conversie matrice rara -> matrice normala
int ** Conversie(MatriceRara& matR)
{
	// declarare si alocare matrice
	int **mat;
	mat = new int*[matR.nrLinii];
	for (int i = 0; i < matR.nrLinii; i++)
	{
		mat[i] = new int[matR.nrColoane];
		for (int j = 0; j < matR.nrColoane; j++)
			mat[i][j] = 0;
	}

	// copiere elemente nenule
	for (int i = 0; i < matR.nrElemente; i++)
		mat[matR.Linie[i]][matR.Coloana[i]] = matR.Valoare[i];

	// returnare rezultat
	return mat;
}
```

Basic operations

One way to implement matrix operations on sparse matrices is to simulate direct access to elements using Get/Set functions and use the traditional algorithms. This is the simplest method but has the drawback that is very slow because we must consider all elements (even non-zero ones) and use the sequential access functions that need *O(nrZ)* time to complete.

A better solution is to develop specialized algorithms that use the particularities of the sparse matrices to provide a fast execution time. In order to speed up the operations we’ll assume that elements inside the sparse matrix are sorted according to (line, column); all operations presented here preserve this property on the matrices.

Transpose

The algorithm to compute the transpose of a sparse matrix is very simple. All we need to do is swap the columns and line indices and sort the resulted arrays in order to preserve the mentioned property:

```
// Calculeaza transpusa unei matrici stocate ca matrice rara
MatriceRara Transpusa(const MatriceRara& mat)
{
	// initializare rezultat
	MatriceRara R;
	R.Linie = new int[mat.nrElemente];
	R.Coloana = new int[mat.nrElemente];
	R.Valoare = new int[mat.nrElemente];
	R.nrElemente = mat.nrElemente;
	R.nrLinii = mat.nrColoane;
	R.nrColoane = mat.nrLinii;

	// copiere date si inversare linie-coloana
	for (int i = 0; i < mat.nrElemente; i++)
	{
		R.Linie[i] = mat.Coloana[i];
		R.Coloana[i] = mat.Linie[i];
		R.Valoare[i] = mat.Valoare[i];
	}

	// sortare rezultat dupa linie, coloana
	int temp;
	for (int i = 0; i < R.nrElemente; i++)
		for (int j = i + 1; j < R.nrElemente; j++)
			if (R.Linie[i] > R.Linie[j] ||
			R.Linie[i] == R.Linie[j] && R.Coloana[i] > R.Coloana[j])
			{
				temp = R.Linie[i]; R.Linie[i] = R.Linie[j];
				R.Linie[j] = temp;
				temp = R.Coloana[i]; R.Coloana[i] = R.Coloana[j];
				R.Coloana[j] = temp;
			}

	return R;
}
```

Addition

The algorithm for sparse matrix addition has three steps:

- check that the matrix sizes are equal
- determine the number of elements in the resulted matrix and allocate memory for them
- compute the value for all elements and store them in the resulted matrix

Because the matrices are sorted by (line, column) we can do only one pass to determine the number of elements and to compute the values. In the process we take care to eliminate zero elements in the result.

```
// Aduna doua matrici rare
MatriceRara Adunare(const MatriceRara& M1, const MatriceRara& M2)
{
	MatriceRara R = {NULL, NULL, NULL, 0, 0, 0};

	if (M1.nrLinii != M2.nrLinii && M1.nrColoane != M2.nrColoane)
	{
		cout << "Eroare: Matricele trebuie sa aiba dimensiuni egale."
		<< endl;
		return R;
	}

	// determinarea numarului de elemente al matricei rezultat
	int i = 0, j = 0, nr = 0;
	while (i < M1.nrElemente && j < M2.nrElemente)
	{
		if (M1.Linie[i] < M2.Linie[j])
			i++;
		else if (M1.Linie[i] > M2.Linie[j])
			j++;
		else
		{
			if (M1.Coloana[i] < M2.Coloana[j])
				i++;
			else if (M1.Coloana[i] > M2.Coloana[j])
				j++;
			else
			{
				if (M1.Valoare[i] + M2.Valoare[j] == 0)
					nr = nr + 2; // element comun nul
				else
					nr = nr + 1; // element comun nenul

				i++; j++;
			}
		}
	}

	// numarul de elemente al rezultatului
	nr = M1.nrElemente + M2.nrElemente - nr ;

	// initializare rezultat
	R.Linie = new int[nr];
	R.Coloana = new int[nr];
	R.Valoare = new int[nr];
	R.nrElemente = nr;
	R.nrLinii = M1.nrLinii;
	R.nrColoane = M1.nrColoane;

	// calculare rezultat
	int k = 0;		// indice in matricea rezultat
	i = 0; j = 0;
	while (k < R.nrElemente)
	{
		if (M1.Linie[i] < M2.Linie[j])
		{
			R.Linie[k] = M2.Linie[i];
			R.Coloana[k] = M2.Coloana[i];
			R.Valoare[k] = M2.Valoare[i];
			i++; k++;
		}
		else if (M1.Linie[i] > M2.Linie[j])
		{
			R.Linie[k] = M1.Linie[j];
			R.Coloana[k] = M1.Coloana[j];
			R.Valoare[k] = M1.Valoare[j];
			j++; k++;
		}
		else
		{
			if (M1.Coloana[i] < M2.Coloana[j])
			{
				R.Linie[k] = M2.Linie[i];
				R.Coloana[k] = M2.Coloana[i];
				R.Valoare[k] = M2.Valoare[i];
				i++; k++;
			}
			else if (M1.Coloana[i] > M2.Coloana[j])
			{
				R.Linie[k] = M1.Linie[j];
				R.Coloana[k] = M1.Coloana[j];
				R.Valoare[k] = M1.Valoare[j];
				j++; k++;
			}
			else
			{
				if (M1.Valoare[i] + M2.Valoare[j] == 0)
				{
					i++; j++;    // element comun nul
				}
				else
				{
					R.Linie[k] = M1.Linie[i];;
					R.Coloana[k] = M1.Coloana[i];
					R.Valoare[k] = M1.Valoare[i] +
				M2.Valoare[j];
					i++; j++; k++; // element comun nenul
				}
			}
		}
	}

	return R;
}
```

Multiplication

Matrix multiplication is done in two phases. In the first phase we simulate the operation in order to determine the number of elements in the result set and in the second one we compute the result.

The operation is executed only if the matrices have compatible sizes.

In order to compute the value of one element in the result set we do the following:

- we initialize the current value with 0
- for every element in the current column from the first matrix
    a. we search for a corresponding element in the second matrix
    b. if we find such element we add the product to the current value
- if the current value is non-zero we store it in the result matrix

```
// Inmultirea a doua matrici rare
MatriceRara Inmultire(MatriceRara& M1, MatriceRara& M2)
{
        MatriceRara R = {NULL, NULL, NULL, 0, 0, 0};

        if (M1.nrColoane != M2.nrLinii)
        {
                cout << "Eroare: Dimensiunile matricelor nu corespund." << endl;
                return R;
        }

        // determinarea numarului de elemente al rezultatului (simulare inmultire)
        int nr = 0, suma;

        for (int i = 0; i < M1.nrLinii; i++)
                for (int j = 0; j < M2.nrLinii; j++)
                {
                        suma = 0;

                        // calcul valoare element
                        for (int col = 0; col < M1.nrColoane; col++)
                                for (int i1 = 0; i1 < M1.nrElemente; i1++)
                                        // daca exista in M1
                                        if (M1.Linie[i1] == i && M1.Coloana[i1] == col)
                                                for (int i2 = 0; i2 < M2.nrElemente;
i2++)
                                                        if (M2.Linie[i2] == col &&
                                                        M2.Coloana[i2] == j)  // si in M2
                                                                suma += M1.Valoare[i1] *
M2.Valoare[i2];
                        if (suma) nr++;
                }


        // initializare rezultat
        R.Linie = new int[nr];
        R.Coloana = new int[nr];
        R.Valoare = new int[nr];
        R.nrElemente = nr;
        R.nrLinii = M1.nrLinii;
        R.nrColoane = M2.nrColoane;

        // calcul rezultat
        int indice = 0;
        for (int i = 0; i < M1.nrLinii; i++)
                for (int j = 0; j < M2.nrLinii; j++)
                {
                        suma = 0;

                        // calcul valoare element
                        for (int col = 0; col < M1.nrColoane; col++)
                                for (int i1 = 0; i1 < M1.nrElemente; i1++)
                                        // daca exista in M1
                                        if (M1.Linie[i1] == i && M1.Coloana[i1] == col)
                                                for (int i2 = 0; i2 < M2.nrElemente;
i2++)
                                                        if (M2.Linie[i2] == col &&
M2.Coloana[i2] == j)  // si in M2
                                                                suma += M1.Valoare[i1] *
M2.Valoare[i2];
                        if (suma)
                        {
                                R.Linie[indice] = i;
                                R.Coloana[indice] = j;
                                R.Valoare[indice] = suma;
                                indice++;
                        }
                }
        return R;
}
```

Exercises

1. Write a function to eliminate all elements with values less than a threshold from a sparse matrix.
2. Write functions to read a sparse matrix from the console and to write the result in both forms (sparse and normal).
3. Write a function to multiply a sparse matrix with a constant. Use this function in conjunction with the addition to simulate sparse matrix subtraction.
4. Write a function to multiply a sparse matrix by a line vector.
5. Give a solution to eliminate the necessity of the first pass in the matrix multiplication algorithm.
6. An airline company operates direct flights between *n* cities. Write an application to determine the minimum number of flight segments between any two cities.

Solution:

We model the problem as an undirected graph with cities represented as vertexes and flight segments as edges and we store the graph as an adjacency matrix A (using a sparse matrix). In this case A^k will contain all flight routes of length *k* between cities.