Sparse Matrix

This C++ file is a brilliant showcase of how to efficiently handle data structures that are mostly "empty." It demonstrates a powerful technique for representing and manipulating both **sparse matrices** and **polynomials**. The core idea is to save memory by only storing the meaningful data—the non-zero elements in a matrix or the terms with non-zero coefficients in a polynomial.

Instead of using large, wasteful 2D arrays or fixed-size arrays, the code uses a compact, 1D array of structs. Each struct holds a piece of essential information: the row, column, and value for a matrix element, or the coefficient and exponent for a polynomial term.

The highlight of the file is the addition algorithm. For both sparse matrices and polynomials, it implements a highly efficient, merge-like process. It iterates through two compact arrays simultaneously, comparing indices (rows/columns or exponents) to correctly combine them into a third compact array representing the sum. This approach is a cornerstone of efficient data structure manipulation. The entire file is written in a procedural, C-style, using structs and manual memory management with malloc and free.

Of course, Yash! Here's a detailed breakdown of your file on sparse representations.

This C++ file is a brilliant showcase of how to efficiently handle data structures that are mostly "empty." It demonstrates a powerful technique for representing and manipulating both **sparse matrices** and **polynomials**. The core idea is to save memory by only storing the meaningful data—the non-zero elements in a matrix or the terms with non-zero coefficients in a polynomial.

Instead of using large, wasteful 2D arrays or fixed-size arrays, the code uses a compact, 1D array of structs. Each struct holds a piece of essential information: the row, column, and value for a matrix element, or the coefficient and exponent for a polynomial term.

The highlight of the file is the addition algorithm. For both sparse matrices and polynomials, it implements a highly efficient, merge-like process. It iterates through two compact arrays simultaneously, comparing indices (rows/columns or exponents) to correctly combine them into a third compact array representing the sum. This approach is a cornerstone of efficient data structure manipulation. The entire file is written in a procedural, C-style, using structs and manual memory management with malloc and free.

**Sparse Matrix Representation (C-style)**

This code defines a memory-efficient way to represent a **sparse matrix** (a matrix with mostly zero values). Instead of storing a full M x N grid, it only stores the non-zero elements along with their row and column indices. 📝

* struct element{ int i; int j; int x; };

This defines the blueprint for a single non-zero element. It stores its location (i for row, j for column) and its value (x).

* struct Sparse{ ... struct element \*ele; };

This structure represents the entire sparse matrix. It holds the dimensions (m, n), the total number of non-zero elements (num), and a pointer ele to a dynamically allocated array of element structs.

* s->ele = (struct element \*)malloc(s->num\*sizeof(struct element));

This is the key to memory efficiency. The code allocates just enough memory on the heap to hold only the non-zero elements, avoiding waste.

* if(i==s.ele[k].i && j==s.ele[k].j){ printf("%d ", s.ele[k++].x); }

The display function reconstructs the full matrix view. It iterates through every possible (i, j) position. If the current position matches the coordinates of the next non-zero element in our compact list (s.ele[k]), it prints that element's value; otherwise, it prints a 0.

**Addition of two sparse matrices**

This section implements a function to add two sparse matrices. The algorithm is highly efficient because it works directly with the compact representations, never needing to create the full matrices in memory. It merges the two lists of non-zero elements into a new list representing the sum. ➕

* struct Sparse \*add(struct Sparse \*s1, struct Sparse \*s2) { ... }

The add function takes pointers to two sparse matrices and returns a pointer to a new sparse matrix that will contain their sum.

* while (i < s1->num && j < s2->num) { ... }

This is the main merge loop. It uses three counters: i for s1, j for s2, and k for the sum matrix. It continues as long as there are elements left in both input matrices.

* if (s1->ele[i].i < s2->ele[j].i) sum->ele[k++] = s1->ele[i++];

The logic compares the locations of the elements. If the element from s1 comes first (i.e., is in a smaller row number), it's copied to the sum, and only s1's counter i moves forward.

* else { ... sum->ele[k++].x = s1->ele[i++].x + s2->ele[j++].x; }

If two non-zero elements are at the exact same position (i and j are the same, and rows and columns are the same), their values are added together, and the result is stored in the sum matrix. Both i and j are then incremented.

* while (i < s1->num) sum->ele[k++] = s1->ele[i++];

After the main loop finishes, one of the matrices might still have elements left. These final two while loops are crucial for copying any remaining elements into the sum matrix.

**Polynomial representation and adding two polynomials**

This code uses the same sparse representation technique to efficiently store and add polynomials. A polynomial is treated as a collection of terms, where each term has a coefficient and an exponent. Only terms with non-zero coefficients are stored. 🧑‍🏫

* struct Term { int coeff; int exp; };

This struct defines a single term of a polynomial, storing its coefficient (coeff) and its exponent (exp).

* struct Poly { int n; struct Term \*terms; };

This represents the entire polynomial. It stores the number of non-zero terms (n) and a pointer to a dynamic array of those Terms.

* while (i < p1->n && j < p2->n) { ... }

This is the core merge logic for adding two polynomials, p1 and p2. It's almost identical to the sparse matrix addition but compares exponents instead of row/column indices.

* if (p1->terms[i].exp > p2->terms[j].exp)

The algorithm compares the exponents of the current terms from each polynomial. If the term from p1 has a higher exponent, it's copied to the sum, and p1's counter i is advanced.

* else { ... sum->terms[k++].coeff = p1->terms[i++].coeff + p2->terms[j++].coeff; }

If the exponents are the same, the coefficients are added together, and the new term is stored in the sum. Both counters i and j are then moved forward. This correctly combines like terms (e.g., 3x2+5x2=8x2).