Московский авиационный институт

(Национальный исследовательский университет)

Факультет прикладной математики и физики

Курсовой проект

по курсу

«Методы и языки программирования»

II семестр

Задание 7

Выполнила: Довженко А.

Группа: М8О-107Б-16, №5

Проверили: Ридли А.Н.

Ридли М.К.

Оценка:

Дата:

Москва

2017г.

Задача:

Составить программу на Си с процедурами и функциями для обработки прямоугольных разреженных матриц элементами комплексного типа, которая:

1. Принимает на вход матрицы различного размера, представленные во входном текстовом файле в обычном формате (по строкам), с одновременных размещением ненулевых элементов в разреженной матрице в соответствии с заданной схемой.
2. Печатает введенную матрицу в обычном виде и во внутреннем представлении согласно заданной схеме размещения.
3. Выполняет необходимые преобразования разреженных матриц (или вычисления над ними) путем обращения к соответствующим процедурам и функциям.
4. Печатает результат задания.

Схема размещения:

Цепочка ненулевых элементов в векторе A со строчным индексированием

M:

Индекс начала 1-ой строки в массиве A	Индекс начала 2-ой строки	…	Индекс начала N-ой строки

A:

Номер столбца	Значение	Индекс следующего ненулевого элемента этой строки	…	

В случае использования динамических структур индексы заменяются соответствующими ссылками.

Преобразование:

Умножить разреженную матрицу на вектор-столбец и вычислить количество ненулевых элементов.

Использование программы:

Программа собирается командой «$make$». Также можно использовать команду «$gcc - std = c99 - ostart *.c$» . Эти способы необходимо производить в корне проекта.

Запускается программа командой «./start»(можно использовать «valgrind ./start», чтобы просматривать используемую программой память).

На вход программа принимает размер матрицу NxM. То есть сначала программа принимает числа n и m. Затем считывает матрицу и матрицу-столбец. После этого программа печатает матрицы в двух представлениях и наконец выводит результат(ы) выполнения задания.

Введение:

Разреженная матрица — это матрица с преимущественно нулевыми элементами. В противном случае, если бóльшая часть элементов матрицы ненулевые, матрица считается плотной.

Нет единства в определении того, какое именно количество ненулевых элементов делает матрицу разрежённой. Разные авторы предлагают различные варианты. Для матрицы порядка n число ненулевых элементов:

- есть O(n). Такое определение подходит разве что для теоретического анализа асимптотических свойств матричных алгоритмов;
- в каждой строке не превышает 10 в типичном случае;
- ограничено $n^{1+\delta}$, где δ< 1.
- таково, что для данного алгоритма и вычислительной системы имеет смысл извлекать выгоду из наличия в ней нулей.

В силу того, что определить разреженную матрицу строго не получается, то разреженными матрицами можно считать именно те, для которых применение специализированных алгоритмов обработки и/или хранения является оправданным.

Огромные разрежённые матрицы часто возникают при решении дифференциальных уравнений именно численными методами.

При хранении и преобразовании разрежённых матриц в компьютере бывает полезно, а часто и необходимо, использовать специальные алгоритмы и структуры данных, которые учитывают разрежённую структуру матрицы. Операции и алгоритмы, применяемые для работы с обычными, плотными матрицами, применительно к большим разрежённым матрицам работают относительно медленно и требуют значительных объёмов памяти. Однако разрежённые матрицы могут быть легко сжаты путём записи только своих ненулевых элементов, что снижает требования к компьютерной памяти.

Организация программы:

Управляющая программа, организующая ввод и вывод, реализована в $main.c$, а реализация матриц и функции над ними – в файлах $matrix.h$ и $matrix.c$.

matrix.h, matrix.c:

Матрица имеет элементы типа elem_type – это комплексный тип.

typedef complex double elem_type;

```
typedef struct _node_A Node_A;
typedef struct _node_M Node_M;
typedef struct _matrix *Matrix;


struct _node_A {
   int column;
   elem_type elem;
   struct _node_A *next;
}; //Node_A

struct _node_M {
   Node_A *begin_row; // = NULL, если в строке только 0
   struct _node_M *next;
}; //Node_M

struct _matrix {
   int n;
   int m;
   int entries;
   Node_M *head_M;
}; //*Matrix;
```

Векторы A и M представляют собой списки: узел списка A содержит номер столбца, значение элемента и указатель на следующий элемент в списке A (и, соответственно, в строке), по сути мы имеем список A для каждой строки; узел списка M содержит указатель на начало каждой строки (список A строки), если строка нулевая, указатель == NULL, и указатель на следующий узел списка M. Структура Matrix создана для удобства выполнения задачи, в принципе, можно было обойтись и без нее. Она содержит n, m – количество строк и столбцов матрицы соответственно, entries – количество ненулевых элементов и указатель на начало списка M.

Matrix matrix_create(int n, int m);

Создание матрицы mat, размером $n \times m$.

void matrix_destroy(Matrix *mat);
Освобождение памяти, выделенной под матрицу *mat.

void elem_set(Matrix mat, int i, int j, elem_type value);
Устанавливает в матрицу mat элемент val на место i-й строки и j-го столбца.

int nonzero(Matrix mat);
Вспомогательная функция, возвращающая количество ненулевых элементов в матрице.

elem_type elem_get(Matrix mat, int i, int j);
Возвращает элемент матрицы mat на месте i-й строки и j-го столбца.

void matrix_print_outside(Matrix mat);
Печатает матрицу mat в нормальном представлении.

void matrix_print_inside(Matrix mat);
Печатает матрицу mat в компьютерном представлении.

void matrix_multiply(Matrix mat1, Matrix mat2, Matrix res);
Перемножает матрицы mat1 и mat2, помещая результат в матрицу res.

main.c:

Программа управляется через этот файл. Принимает на вход 2 числа: n и m –размер матрицы. Затем считывается матрица и матрица-столбец в стандартном строчном формате. После этого выводится матрица-столбец в нормальном представлении и в компьютерном представлении, затем выводится матрица в нормальном представлении и в представлении компьютера. Матрицы перемножаются и выводится результат в нормальном представлении и в представлении компьютера. Выводится количество ненулевых элементов в результирующей матрице.

Вывод:

Разреженные матрицы широко используются в математике и некоторых областях программирования. С разреженными матрицами удобно и необходимо работать, используя компактные схемы хранения только ненулевых элементов матрицы. Поскольку хранится будут только ненулевые элементы, то и необходимо хранить индекс этих ненулевых элементов, указывающий на расположение их в матрице. Так что вместо расходования памяти на нулевые элементы, расходуется память на хранение индексов.

Задачи на моделирование, задачи линейной алгебры и простое решение дифференциальных уравнений – примеры того, что помогают решать разреженные матрицы.

Существует огромное количество схем представления разреженных матриц, которые придумываются и по сей день. И, наверное, каждая чем-то хороша, так как задачи бывают разные. Где-то нужен быстрый доступ к элементу, где-то необходимо быстро вносить изменения в матрицу, ну а где-то минимальное потребление памяти.

Чтобы рассчитать эффективность схемы, мы составим уравнение $sizeof(struct*) * N + (sizeof(int) + sizeof(complex\ double) + sizeof(struct*)) * k * m * n < m * n * sizeof(complex\ double)$, где $k -$ плотность матрицы

Зная, что sizeof(complex double) равно 16 байт, sizeof(int) – 4 байта, sizeof(struct *) – 8 байт получаем уравнение:

$8 * n + (4 + 16 + 8) * k * m * n < m * n * 16$, где $k -$ плотность матрицы

После преобразований выходит:

$$k < \frac{4}{7}$$

Это значит, что приблизительно схема эффективна до 57% ненулевых элементов в матрице.

Код программы:

Makefile

```
CC = gcc
LDFLAGS =-std=c99
CFLAGS =-c -std=c99
SOURCES = main.o matrix.o
EXECUTABLE = start

all: $(EXECUTABLE)


$(EXECUTABLE): $(SOURCES)
	$(CC)$(LDLAGS)$(SOURCES) -o $@

.c.o:
	$(CC)$(CFLAGS) $< -o $@

clean:
	-rm -f *.o start
```

matrix.h

```
#ifndef _MATRIX_H_
#define _MATRIX_H_

#include <complex.h>

#define MEMORY_LIMIT 1
#define ERROR_GETTING_ELEMENT 2

typedef complex double elem_type;

typedef struct _node_A Node_A;
typedef struct _node_M Node_M;
typedef struct _matrix *Matrix;

Matrix matrix_create(int n, int m);
void matrix_destroy(Matrix *mat);
elem_type elem_get(Matrix mat, int i, int j);
void matrix_print_inside(Matrix mat);
void matrix_print_outside(Matrix mat);
int nonzero(Matrix mat);
void elem_set(Matrix mat, int i, int j, elem_type value);
void matrix_multiply(Matrix mat1, Matrix mat2, Matrix res);

#endif // __MATRIX_H__
```

matrix.c

```
#include <complex.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

#include "matrix.h"

struct _node_A {
   int column;
   elem_type elem;
   struct _node_A *next;
}; //Node_A

struct _node_M {
   Node_A *begin_row; // = NULL, если в строке только 0
   struct _node_M *next;
}; //Node_M

struct _matrix {
   int n;
   int m;
   int entries;
   Node_M *head_M;
}; //*Matrix;
```

```
Node_A *node_A_create(void);
Node_M *node_M_create(void);
void node_A_add_last(Node_M *node_M, Node_A *new_node);
void node_A_add_before(Node_M *node_M, Node_A *after_node_A, Node_A *new_node);
void node_M_add_last(Matrix mat, Node_M *new_node);



Matrix matrix_create(int n, int m)
{
   Matrix mat = (Matrix) calloc(1, sizeof(*mat));
   if (!mat) {
      fprintf(stderr, "Error: no memory\n");
      exit(MEMORY_LIMIT);
   }

   mat->n = n;
   mat->m = m;
   mat->entries = 0;

   Node_M *node_M;
   for(int i = 0; i < n; ++i) {
      node_M = node_M_create();
      node_M_add_last(mat, node_M);
      node_M = node_M->next;
   }

   return mat;
}

Node_A *node_A_create(void)
{
   Node_A *new_node = (Node_A *) calloc(1, sizeof(Node_A));
   if (!new_node) {
      fprintf(stderr, "Error: no memory\n");
      exit(MEMORY_LIMIT);
   }
   new_node->next = NULL;
   return new_node;
}

Node_M *node_M_create(void)
{
   Node_M *new_node = (Node_M *) calloc(1, sizeof(Node_M));
   if (!new_node) {
      fprintf(stderr, "Error: no memory\n");
      exit(MEMORY_LIMIT);
   }
   new_node->next = NULL;
   new_node->begin_row = NULL;
   return new_node;
}
```

```
void node_A_add_last(Node_M *node_M, Node_A *new_node)
{
   Node_A *tmp = node_M->begin_row;
   if(!tmp) {
      node_M->begin_row = new_node;
      new_node->next = NULL;
      return;
   }
   while(tmp->next) {
      tmp = tmp->next;
   }
   tmp->next = new_node;
   new_node->next = NULL;
}

void node_A_add_before(Node_M *node_M, Node_A *after_node_A, Node_A *new_node)
{
   Node_A *tmp = node_M->begin_row;
   if(node_M->begin_row == after_node_A) {
      new_node->next = node_M->begin_row;
      node_M->begin_row = new_node;
      return;
   }
   while(tmp && (tmp->next != after_node_A)) {
      tmp = tmp->next;
   }
   if(tmp) {
      new_node->next = tmp->next;
      tmp->next = new_node;
   }
}

void node_M_add_last(Matrix mat, Node_M *new_node)
{
   Node_M *tmp = mat->head_M;
   if(!tmp) {
      mat->head_M = new_node;
      new_node->next = NULL;
      new_node->begin_row = NULL;
      return;
   }
   while(tmp->next) {
      tmp = tmp->next;
   }
   tmp->next = new_node;
   new_node->next = NULL;
   new_node->begin_row = NULL;
}

void matrix_destroy(Matrix *mat)
{
```

```
    Node_M *node_M = (*mat)->head_M;
    while(node_M) {
        Node_M *tmp = node_M->next;


        Node_A *node_A = node_M->begin_row;
        while(node_A) {
            Node_A *tmp_A = node_A->next;
            free(node_A);
            node_A = tmp_A;
        }

        free(node_M);
        node_M = tmp;
    }

    free(*mat);
    *mat = NULL;
}

elem_type elem_get(Matrix mat, int i, int j)
{
    if (!(mat->head_M)) {
        fprintf(stderr, "Error: impossible getting element\n");
        exit(ERROR_GETTING_ELEMENT);
    }

    Node_M *row = mat->head_M;
    while(i > 1) {
        row = row->next;
        --i;
    }

    if(!row) {
        fprintf(stderr, "Error: impossible getting element\n");
        exit(ERROR_GETTING_ELEMENT);
    }

    if(row->begin_row == NULL) { //если в строке только 0
        return 0;
    } else {
        for(Node_A *helper = row->begin_row; helper->next; helper = helper->next) {
            if(helper->column == j) {
                return helper->elem;
            }
        }
        return 0;
    }
}
```

```
void matrix_print_outside(Matrix mat)
{
   for (int i = 1; i <= mat->n; ++i) {
      for (int j = 1; j <= mat->m; ++j) {
         elem_type num = elem_get(mat, i, j);
         printf("(%.2lf + %.2lfi)", creal(num), cimag(num));
      }
      printf("\n");
   }
   printf("\n");
}

void matrix_print_inside(Matrix mat)
{
   Node_M *M_print = mat->head_M;
   for (int i = 0; M_print; ++i, M_print = M_print->next) {
      printf("%d row address: %p\n", i+1, &M_print->begin_row);
      Node_A *A_print = M_print->begin_row;
      if(!A_print) {
         printf("NULL\n");
      } else {
         for (; A_print; A_print = A_print->next) {
            printf("Column: %d, value: (%.2lf + %.2lfi), address next element %p\n", A_print->column,
creal(A_print->elem), cimag(A_print->elem), &A_print->next);
         }
      }
   }
   printf("\n");
}



int nonzero(Matrix mat)
{
   return mat->entries;
}



void elem_set(Matrix mat, int i, int j, elem_type value)
{
   if(value != 0) {
      mat->entries++;
      Node_M *node_M = mat->head_M;
      for(; i > 1; --i) {
         node_M = node_M->next;
      }

      Node_A *node_A = node_M->begin_row;
      bool insert_last = true;
      while(node_A) {
```

```
            if(node_A->column < j) {
                node_A = node_A->next;
            } else {
                Node_A *new_node = node_A_create();
                node_A_add_before(node_M, node_A, new_node);
                new_node->column = j;
                new_node->elem = value;
                insert_last = false;
                break;
            }

            if(insert_last) {
                node_A = node_A_create();
                node_A_add_last(node_M, node_A);
                node_A->column = j;
                node_A->elem = value;
            }
        }
    }
}



void matrix_multiply(Matrix mat1, Matrix mat2, Matrix res)
{
    for (int i = 1; i <= mat1->n; ++i) {
        for (int j = 1; j <= mat2->m; ++j) {
            elem_type s = 0;
            for (int k = 1; k <= mat1->m; ++k) {
                elem_type num_mat1 = elem_get(mat1, i, k);
                elem_type num_mat2 = elem_get(mat2, k, j);
                s += num_mat1 * num_mat2;
            }
            elem_set(res, i, j, s);
        }
    }
}
```

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <complex.h>

#include "matrix.h"

int main(void)
{
    int n, m;
    double x = 0.0, y = 0.0;
    printf("Enter the size of the input matrix:\n");
    scanf("%d %d", &n, &m);
```

```
    Matrix mat = matrix_create(n, m);
    Matrix column = matrix_create(m, 1);

    printf("Enter the matrix:\n");
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            scanf("%lf %lf", &x, &y);
            elem_type elem = x + y * I;
            elem_set(mat, i, j, elem);
        }
    }
    printf("Enter the matrix-column:\n");
    for (int i = 1; i <= m; ++i) {
        scanf("%lf %lf", &x, &y);
        elem_type elem = x + y * I;
        elem_set(column, i, 1, elem);
    }

    Matrix mult = matrix_create(n, 1);
    printf("Column:\n");
    matrix_print_outside(column);
    printf("Internal representation:\n");
    matrix_print_inside(column);
    printf("Matrix:\n");
    matrix_print_outside(mat);
    printf("Internal representation:\n");
    matrix_print_inside(mat);
    matrix_multiply(mat, column, mult);
    printf("Result:\n");
    matrix_print_outside(mult);
    printf("Internal representation:\n");
    matrix_print_inside(mult);
    printf("Nonzero elements: %d\n", nonzero(mult));
    matrix_destroy(&mult);
    matrix_destroy(&column);
    matrix_destroy(&mat);
    return 0;
}
```