Условие

Разреженная (содержащая много нулей) матрица хранится в форме 3-х объектов:

- вектор A содержит значения ненулевых элементов;
- вектор JA содержит номера столбцов для элементов вектора A;
- вектор IA, в элементе Nk которого находится номер компонент в A и JA, с которых начинается описание строки Nk матрицы A.

Вектор-столбец хранится в 2х объектах:

- вектор B, содержащий значения ненулевых элементов
- вектор IB, параллельный вектору B, содержащий индексы ненулевых элементов

1. Смоделировать операцию умножения матрицы и вектора-столбца, хранящихся в приведенных выше форматах, с получением результата в форме хранения вектора.
2. Произвести операцию умножения, применяя стандартный алгоритм работы с матрицами.
3. Сравнить время выполнения операций и объем памяти при использовании этих 2-х алгоритмов при различном проценте заполнения матриц.

2. Описание ТЗ

2.1 Исходные данные и результаты

Входные данные

1. Матрица (ручной ввод / координатный ввод / рандомное заполнение некоторого процента)
2. Вектор - столбец (ручной ввод / координатный ввод / рандомное заполнение некоторого процента)
3. Целое число — номер команды (от 0 до 10; См. Описание задачи);

Выходные данны

1. Введенная матрица в нормальном виде
2. Введенная матрица в разреженном представлении
3. Введенный вектор в нормальном виде
4. Введенный вектор в разреженном представлении
5. Результат произведения матрицы на вектор в нормальном виде если размер < 30x30 иначе - в разреженном формате
6. Характеристика сравнения способов перемножения матриц

2.2 Описание задачи, реализуемой программой

Программа реализует интерфейс для умножения вектора-столбца на матрицу, позволяющий выполнить следующие действия:

> Номер перечисления соответствует номеру команды в программе

0. Выход
1. Задать множители (матрицу и вектор)
2. Отобразить матрицу в нормальном виде
3. Отобразить матрицу в представлении CSR
4. Отобразить вектор в нормальном виде
5. Отобразить вектор в формате CSR
6. Выполнить умножение обычным способом
7. Выполнить умножение в формате CSR
8. Показать результаты сравнения методов

2.3 Способ обращения к программе

Взаимодействие с программой происходит через консольный интерфейс, входные данные вводятся пользователем с клавиатуры Запуск программы из рабочей директории:

```
./app.exe
```

2.4 Описание возможных аварийных ситуаций и ошибок пользователя

Аварийные ситуации могут возникнуть только в случае невозможности выделения памяти.

> В программе предусмотренна защита от неправильного ввода, поэтому она не завершится аварийно в этом случае.

3. Внутренние структуры данных

Представление матрицы

```
typedef struct
{
    int **data;
    size_t nrows;
    size_t ncols;
} matrix_t;
```

Представление разреженной матрицы

```
typedef struct
{
    int *val;
    size_t *col;
    size_t *row_offset;
    size_t nnz;
```

```
    size_t nrows;
    size_t ncols;
} csr_matrix;
```

Представление разреженного вектора

```
typedef struct
{
    int *val;
    size_t *i;
    size_t nnz;
    size_t len;
} csr_vector;
```

4. Описание алгоритма

Алгоритм умножения разреженной матрицы на разреженный вектор на число:

1. Создание массива для быстрого поиска индексов вектора (расширение разреженного вектора в нормальный вид)
2. Обход матрицы по строкам
3. Обход ненулевых элементов строки
4. Если существует ненулевой элемент вектора, у которого номер строки равен номеру столбца очередного ненулевого элемента матрицы, то результат их умножения прибавляется к очередному элементу результирующего вектора
5. Добавление ненулевого результата в итоговый вектор

5. Основные функции

`return_code` - тип, описывающий код возврата `matrix_t` - тип, описывающий матрицу (См. структуры данных) `csr_matrix` - тип, описывающий разреженную матрицу (См. структуры данных) `csr_vector` - тип, описывающий разреженный вектор (См. структуры данных)

Команда для задания множителей

```
return_code process_set_multipliers(matrix_t *m, csr_matrix *sm, matrix_t
*v, csr_vector *sv);
```

Команда для печати матрицы в нормальной форме

```
```

```
void process_print_matrix(const matrix_t *m);
```

Команда для печати матрицы в разреженном виде

```
void process_print_spare_matrix(const csr_matrix *sm);
```

Команда для печати вектора в нормальной форме

```
void process_print_vector(const matrix_t *m);
```

Команда для печати вектора в разреженном вде

```
void process_print_spare_vector(const csr_vector *sv);
```

Команда для классического умножения

```
void process_multiply(const matrix_t* m, const matrix_t* v);
```

Команда для умножения матрицы на вектор в разреженном виде

```
void process_multiply_spare_mv(const csr_matrix *sm, const csr_vector
*sv);
```

Команда для вывода статистики по производительности для заданных множителей

```
void process_show_stat(const matrix_t *m, const csr_matrix *sm, matrix_t
*v, const csr_vector *sv);
```

6. Тесты

Описание теста	Ввод	Вывод

Описание теста	Ввод	Вывод
Не удалось выделить память в ходе исполнения программы	-	Error: can't allocate memory
Ввод некорректной опции выбора	-23	Невалидное значение. Попробуйте еще раз
Ввод некорректной размерности матрицы	-1	Невалидное значение. Попробуйте еще раз
Вывод матрицы до того, как она была задана	2	MATRIX IS UNDEFINED
Вывод матрицы до того в CSR представлении, как она была задана	3	MATRIX IS UNDEFINED
Вывод вектора до того, как он был задан	4	VECTOR IS UNDEFINED
Вывод вектора в CSR представлении до того,в CSR представлении как он был задан	5	VECTOR IS UNDEFINED
Выполнение операции умножения до того, как были заданы множители	6	MATRIX OR VECTOR IS UNDEFINED
Выполнение операции разреженного умножения до того, как были заданы множители	7	MATRIX OR VECTOR IS UNDEFINED
Выполнение сравнения эффективности до того, как были заданы множители	8	MATRIX OR VECTOR IS UNDEFINED
Умножение с результатом < 30x30	1; [Матрица]; [Вектор]; 6|7	[Результат умножения в нормальном виде]
Умножение с результатом > 30x30	1; [Матрица]; [Вектор]; 6|7	[Результат умножения в разреженном виде]

7. Оценка эффективности

Для каждой контрольной точки мною было проведено 10 замеров, а затем взят средний результат.

[10 x 10, 0%]

	Time (nanoseconds)	Memory (bytes)
Common method	200	488
Chang & Gustavson	1400 (+600.00%)	168 (-65.57%)

[10 x 10, 10%]

	Time (nanoseconds)	Memory (bytes)
Common method	200	488
Chang & Gustavson	700 (+250.00%)	300 (-38.52%)

[10 x 10, 20%]

	Time (nanoseconds)	Memory (bytes)
Common method	300	488
Chang & Gustavson	700 (+133.33%)	432 (-11.48%)

[10 x 10, 30%]

	Time (nanoseconds)	Memory (bytes)
Common method	400	488
Chang & Gustavson	500 (+25.00%)	564 (+15.57%)

[10 x 10, 40%]

	Time (nanoseconds)	Memory (bytes)
Common method	900	488
Chang & Gustavson	100 (-88.89%)	696 (+42.62%)

[10 x 10, 50%]

	Time (nanoseconds)	Memory (bytes)
Common method	300	488
Chang & Gustavson	700 (+133.33%)	828 (+69.67%)

[10 x 10, 60%]

	Time (nanoseconds)	Memory (bytes)
Common method	500	488
Chang & Gustavson	700 (+40.00%)	960 (+96.72%)

[10 x 10, 70%]

	Time (nanoseconds)	Memory (bytes)
Common method	400	488

	Time (nanoseconds)	Memory (bytes)
Chang & Gustavson	600 (+50.00%)	1092 (+123.77%)

[10 x 10, 80%]

	Time (nanoseconds)	Memory (bytes)
Common method	600	488
Chang & Gustavson	600 (+0.00%)	1224 (+150.82%)

[10 x 10, 90%]

	Time (nanoseconds)	Memory (bytes)
Common method	600	488
Chang & Gustavson	700 (+16.67%)	1356 (+177.87%)

[10 x 10, 100%]

	Time (nanoseconds)	Memory (bytes)
Common method	400	488
Chang & Gustavson	700 (+75.00%)	1488 (+204.92%)

[50 x 50, 0%]

	Time (nanoseconds)	Memory (bytes)
Common method	7800	10248
Chang & Gustavson	700 (-91.03%)	488 (-95.24%)

[50 x 50, 10%]

	Time (nanoseconds)	Memory (bytes)
Common method	9400	10248
Chang & Gustavson	1400 (-85.11%)	3548 (-65.38%)

[50 x 50, 20%]

	Time (nanoseconds)	Memory (bytes)
Common method	8000	10248
Chang & Gustavson	2600 (-67.50%)	6608 (-35.52%)

[50 x 50, 30%]

	Time (nanoseconds)	Memory (bytes)
Common method	8000	10248
Chang & Gustavson	4700 (-41.25%)	9668 (-5.66%)

[50 x 50, 40%]

	Time (nanoseconds)	Memory (bytes)
Common method	8000	10248
Chang & Gustavson	7000 (-12.50%)	12728 (+24.20%)

[50 x 50, 50%]

	Time (nanoseconds)	Memory (bytes)
Common method	8300	10248
Chang & Gustavson	9400 (+13.25%)	15788 (+54.06%)

[50 x 50, 60%]

	Time (nanoseconds)	Memory (bytes)
Common method	8000	10248
Chang & Gustavson	11200 (+40.00%)	18848 (+83.92%)

[50 x 50, 70%]

	Time (nanoseconds)	Memory (bytes)
Common method	7800	10248
Chang & Gustavson	11800 (+51.28%)	21908 (+113.78%)

[50 x 50, 80%]

	Time (nanoseconds)	Memory (bytes)
Common method	8100	10248
Chang & Gustavson	11500 (+41.98%)	24968 (+143.64%)

[50 x 50, 90%]

	Time (nanoseconds)	Memory (bytes)
Common method	8000	10248
Chang & Gustavson	11100 (+38.75%)	28028 (+173.50%)

[50 x 50, 100%]

	Time (nanoseconds)	Memory (bytes)
Common method	7900	10248
Chang & Gustavson	10200 (+29.11%)	31088 (+203.36%)

[100 x 100, 0%]

	Time (nanoseconds)	Memory (bytes)
Common method	31000	40448
Chang & Gustavson	1000 (-96.77%)	888 (-97.80%)

[100 x 100, 10%]

	Time (nanoseconds)	Memory (bytes)
Common method	31800	40448
Chang & Gustavson	4200 (-86.79%)	13008 (-67.84%)

[100 x 100, 20%]

	Time (nanoseconds)	Memory (bytes)
Common method	31500	40448
Chang & Gustavson	10700 (-66.03%)	25128 (-37.88%)

[100 x 100, 30%]

	Time (nanoseconds)	Memory (bytes)
Common method	31400	40448
Chang & Gustavson	18700 (-40.45%)	37248 (-7.91%)

[100 x 100, 40%]

	Time (nanoseconds)	Memory (bytes)
Common method	32000	40448
Chang & Gustavson	28800 (-10.00%)	49368 (+22.05%)

[100 x 100, 50%]

	Time (nanoseconds)	Memory (bytes)
Common method	31700	40448

	Time (nanoseconds)	Memory (bytes)
Chang & Gustavson	39300 (+23.97%)	61488 (+52.02%)

[100 x 100, 60%]

	Time (nanoseconds)	Memory (bytes)
Common method	30700	40448
Chang & Gustavson	44700 (+45.60%)	73608 (+81.98%)

[100 x 100, 70%]

	Time (nanoseconds)	Memory (bytes)
Common method	40500	40448
Chang & Gustavson	55600 (+37.28%)	85728 (+111.95%)

[100 x 100, 80%]

	Time (nanoseconds)	Memory (bytes)
Common method	31200	40448
Chang & Gustavson	44500 (+42.63%)	97848 (+141.91%)

[100 x 100, 90%]

	Time (nanoseconds)	Memory (bytes)
Common method	30700	40448
Chang & Gustavson	41400 (+34.85%)	109968 (+171.88%)

[100 x 100, 100%]

	Time (nanoseconds)	Memory (bytes)
Common method	31000	40448
Chang & Gustavson	37400 (+20.65%)	122088 (+201.84%)

[250 x 250, 0%]

	Time (nanoseconds)	Memory (bytes)
Common method	187200	251048
Chang & Gustavson	2100 (-98.88%)	2088 (-99.17%)

[250 x 250, 10%]

	Time (nanoseconds)	Memory (bytes)
Common method	242100	251048
Chang & Gustavson	32700 (-86.49%)	77388 (-69.17%)

[250 x 250, 20%]

	Time (nanoseconds)	Memory (bytes)
Common method	196000	251048
Chang & Gustavson	65700 (-66.48%)	152688 (-39.18%)

[250 x 250, 30%]

	Time (nanoseconds)	Memory (bytes)
Common method	203600	251048
Chang & Gustavson	124200 (-39.00%)	227988 (-9.19%)

[250 x 250, 40%]

	Time (nanoseconds)	Memory (bytes)
Common method	192100	251048
Chang & Gustavson	192900 (+0.42%)	303288 (+20.81%)

[250 x 250, 50%]

	Time (nanoseconds)	Memory (bytes)
Common method	192800	251048
Chang & Gustavson	257800 (+33.71%)	378588 (+50.80%)

[250 x 250, 60%]

	Time (nanoseconds)	Memory (bytes)
Common method	188900	251048
Chang & Gustavson	291200 (+54.16%)	453888 (+80.80%)

[250 x 250, 70%]

	Time (nanoseconds)	Memory (bytes)
Common method	208800	251048
Chang & Gustavson	323300 (+54.84%)	529188 (+110.79%)

[250 x 250, 80%]

	Time (nanoseconds)	Memory (bytes)
Common method	216500	251048
Chang & Gustavson	280100 (+29.38%)	604488 (+140.79%)

[250 x 250, 90%]

	Time (nanoseconds)	Memory (bytes)
Common method	187100	251048
Chang & Gustavson	260300 (+39.12%)	679788 (+170.78%)

[250 x 250, 100%]

	Time (nanoseconds)	Memory (bytes)
Common method	190300	251048
Chang & Gustavson	231100 (+21.44%)	755088 (+200.77%)

[500 x 500, 0%]

	Time (nanoseconds)	Memory (bytes)
Common method	734200	1002048
Chang & Gustavson	2100 (-99.71%)	4088 (-99.59%)

[500 x 500, 10%]

	Time (nanoseconds)	Memory (bytes)
Common method	725700	1002048
Chang & Gustavson	90400 (-87.54%)	304688 (-69.59%)

[500 x 500, 20%]

	Time (nanoseconds)	Memory (bytes)
Common method	728600	1002048
Chang & Gustavson	246400 (-66.18%)	605288 (-39.59%)

[500 x 500, 30%]

	Time (nanoseconds)	Memory (bytes)
Common method	736900	1002048

	Time (nanoseconds)	Memory (bytes)
Chang & Gustavson	441700 (-40.06%)	905888 (-9.60%)

[500 x 500, 40%]

	Time (nanoseconds)	Memory (bytes)
Common method	732000	1002048
Chang & Gustavson	712100 (-2.72%)	1206488 (+20.40%)

[500 x 500, 50%]

	Time (nanoseconds)	Memory (bytes)
Common method	721000	1002048
Chang & Gustavson	1027900 (+42.57%)	1507088 (+50.40%)

[500 x 500, 60%]

	Time (nanoseconds)	Memory (bytes)
Common method	757300	1002048
Chang & Gustavson	1139400 (+50.46%)	1807688 (+80.40%)

[500 x 500, 70%]

	Time (nanoseconds)	Memory (bytes)
Common method	724600	1002048
Chang & Gustavson	1148200 (+58.46%)	2108288 (+110.40%)

[500 x 500, 80%]

	Time (nanoseconds)	Memory (bytes)
Common method	724700	1002048
Chang & Gustavson	1102000 (+52.06%)	2408888 (+140.40%)

[500 x 500, 90%]

	Time (nanoseconds)	Memory (bytes)
Common method	741600	1002048
Chang & Gustavson	983600 (+32.63%)	2709488 (+170.40%)

[500 x 500, 100%]

	Time (nanoseconds)	Memory (bytes)
Common method	737500	1002048
Chang & Gustavson	877900 (+19.04%)	3010088 (+200.39%)

[1000 x 1000, 0%]

	Time (nanoseconds)	Memory (bytes)
Common method	2947100	4004048
Chang & Gustavson	4200 (-99.86%)	8088 (-99.80%)

[1000 x 1000, 10%]

	Time (nanoseconds)	Memory (bytes)
Common method	3076000	4004048
Chang & Gustavson	387800 (-87.39%)	1209288 (-69.80%)

[1000 x 1000, 20%]

	Time (nanoseconds)	Memory (bytes)
Common method	2904300	4004048
Chang & Gustavson	972400 (-66.52%)	2410488 (-39.80%)

[1000 x 1000, 30%]

	Time (nanoseconds)	Memory (bytes)
Common method	2892700	4004048
Chang & Gustavson	1779200 (-38.49%)	3611688 (-9.80%)

[1000 x 1000, 40%]

	Time (nanoseconds)	Memory (bytes)
Common method	2902000	4004048
Chang & Gustavson	2875000 (-0.93%)	4812888 (+20.20%)

[1000 x 1000, 50%]

	Time (nanoseconds)	Memory (bytes)
Common method	3157100	4004048
Chang & Gustavson	4485500 (+42.08%)	6014088 (+50.20%)

[1000 x 1000, 60%]

	Time (nanoseconds)	Memory (bytes)
Common method	3230700	4004048
Chang & Gustavson	5149000 (+59.38%)	7215288 (+80.20%)

[1000 x 1000, 70%]

	Time (nanoseconds)	Memory (bytes)
Common method	2961700	4004048
Chang & Gustavson	4738200 (+59.98%)	8416488 (+110.20%)

[1000 x 1000, 80%]

	Time (nanoseconds)	Memory (bytes)
Common method	2900400	4004048
Chang & Gustavson	4412700 (+52.14%)	9617688 (+140.20%)

[1000 x 1000, 90%]

	Time (nanoseconds)	Memory (bytes)
Common method	2952100	4004048
Chang & Gustavson	4176400 (+41.47%)	10818888 (+170.20%)

[1000 x 1000, 100%]

	Time (nanoseconds)	Memory (bytes)
Common method	3583200	4004048
Chang & Gustavson	4018100 (+12.14%)	12020088 (+200.20%)

[2500 x 2500, 0%]

	Time (nanoseconds)	Memory (bytes)
Common method	20491000	25010048
Chang & Gustavson	23300 (-99.89%)	20088 (-99.92%)

[2500 x 2500, 10%]

	Time (nanoseconds)	Memory (bytes)
Common method	22173200	25010048

	Time (nanoseconds)	Memory (bytes)
Chang & Gustavson	2784400 (-87.44%)	7523088 (-69.92%)

[2500 x 2500, 20%]

	Time (nanoseconds)	Memory (bytes)
Common method	20593700	25010048
Chang & Gustavson	6722300 (-67.36%)	15026088 (-39.92%)

[2500 x 2500, 30%]

	Time (nanoseconds)	Memory (bytes)
Common method	20477900	25010048
Chang & Gustavson	12164400 (-40.60%)	22529088 (-9.92%)

[2500 x 2500, 40%]

	Time (nanoseconds)	Memory (bytes)
Common method	23064600	25010048
Chang & Gustavson	22904700 (-0.69%)	30032088 (+20.08%)

[2500 x 2500, 50%]

	Time (nanoseconds)	Memory (bytes)
Common method	19224800	25010048
Chang & Gustavson	26300200 (+36.80%)	37535088 (+50.08%)

[2500 x 2500, 60%]

	Time (nanoseconds)	Memory (bytes)
Common method	18607300	25010048
Chang & Gustavson	29042100 (+56.08%)	45038088 (+80.08%)

[2500 x 2500, 70%]

	Time (nanoseconds)	Memory (bytes)
Common method	18649500	25010048
Chang & Gustavson	29038800 (+55.71%)	52541088 (+110.08%)

[2500 x 2500, 80%]

	Time (nanoseconds)	Memory (bytes)
Common method	19156900	25010048
Chang & Gustavson	28593400 (+49.26%)	60044088 (+140.08%)

[2500 x 2500, 90%]

	Time (nanoseconds)	Memory (bytes)
Common method	18642300	25010048
Chang & Gustavson	25711800 (+37.92%)	67547088 (+170.08%)

[2500 x 2500, 100%]

	Time (nanoseconds)	Memory (bytes)
Common method	18575200	25010048
Chang & Gustavson	22656200 (+21.97%)	75050088 (+200.08%)

[5000 x 5000, 0%]

	Time (nanoseconds)	Memory (bytes)
Common method	73535000	100020048
Chang & Gustavson	23100 (-99.97%)	40088 (-99.96%)

[5000 x 5000, 10%]

	Time (nanoseconds)	Memory (bytes)
Common method	73744800	100020048
Chang & Gustavson	8911300 (-87.92%)	30046088 (-69.96%)

[5000 x 5000, 20%]

	Time (nanoseconds)	Memory (bytes)
Common method	73570300	100020048
Chang & Gustavson	23762700 (-67.70%)	60052088 (-39.96%)

[5000 x 5000, 30%]

	Time (nanoseconds)	Memory (bytes)
Common method	73770100	100020048
Chang & Gustavson	45573100 (-38.22%)	90058088 (-9.96%)

[5000 x 5000, 40%]

	Time (nanoseconds)	Memory (bytes)
Common method	73403500	100020048
Chang & Gustavson	72816200 (-0.80%)	120064088 (+20.04%)

[5000 x 5000, 50%]

	Time (nanoseconds)	Memory (bytes)
Common method	73360600	100020048
Chang & Gustavson	102243800 (+39.37%)	150070088 (+50.04%)

[5000 x 5000, 60%]

	Time (nanoseconds)	Memory (bytes)
Common method	73609700	100020048
Chang & Gustavson	115994600 (+57.58%)	180076088 (+80.04%)

[5000 x 5000, 70%]

	Time (nanoseconds)	Memory (bytes)
Common method	73215100	100020048
Chang & Gustavson	115281700 (+57.46%)	210082088 (+110.04%)

[5000 x 5000, 80%]

	Time (nanoseconds)	Memory (bytes)
Common method	73903300	100020048
Chang & Gustavson	111742900 (+51.20%)	240088088 (+140.04%)

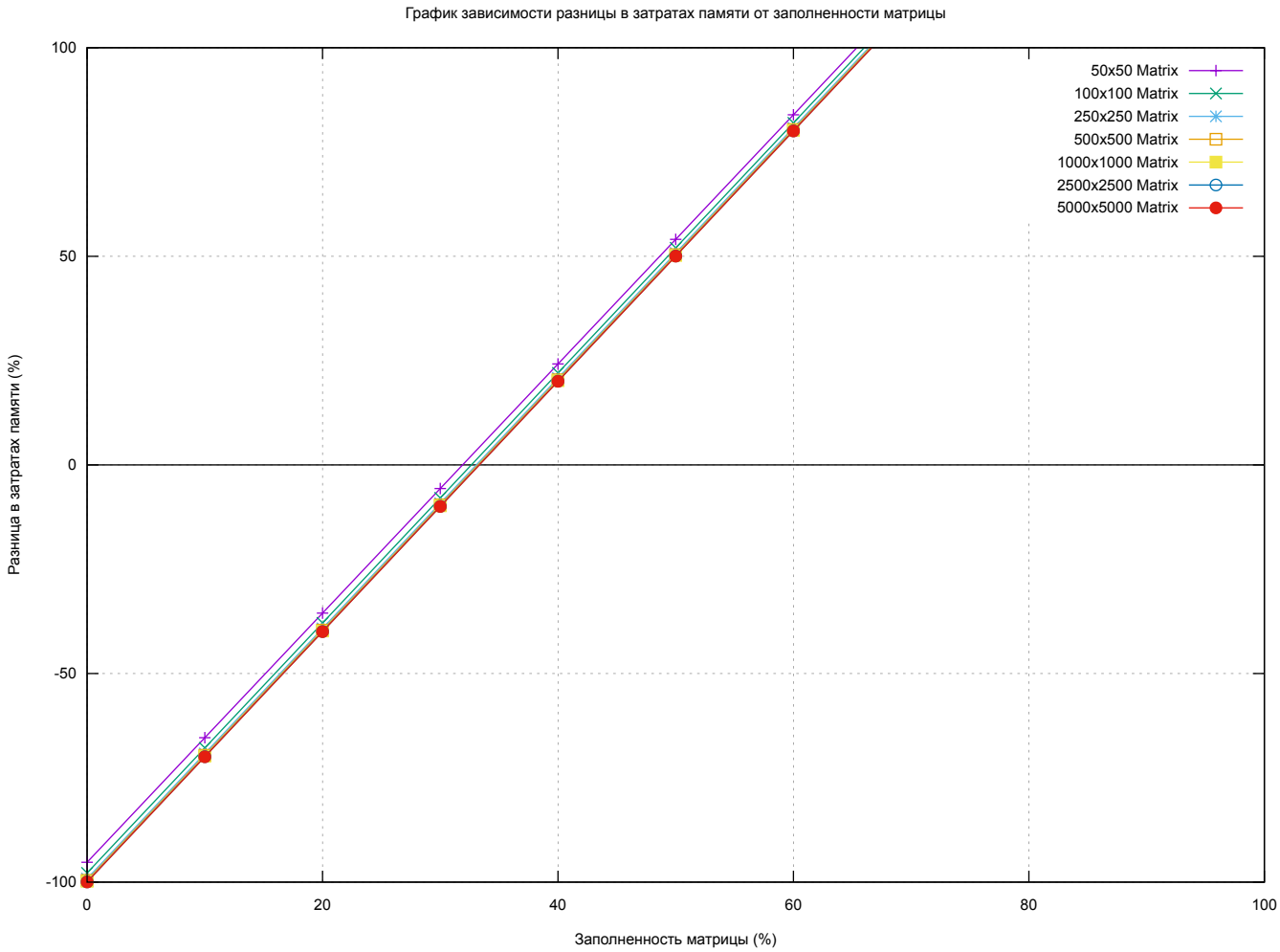
[5000 x 5000, 90%]

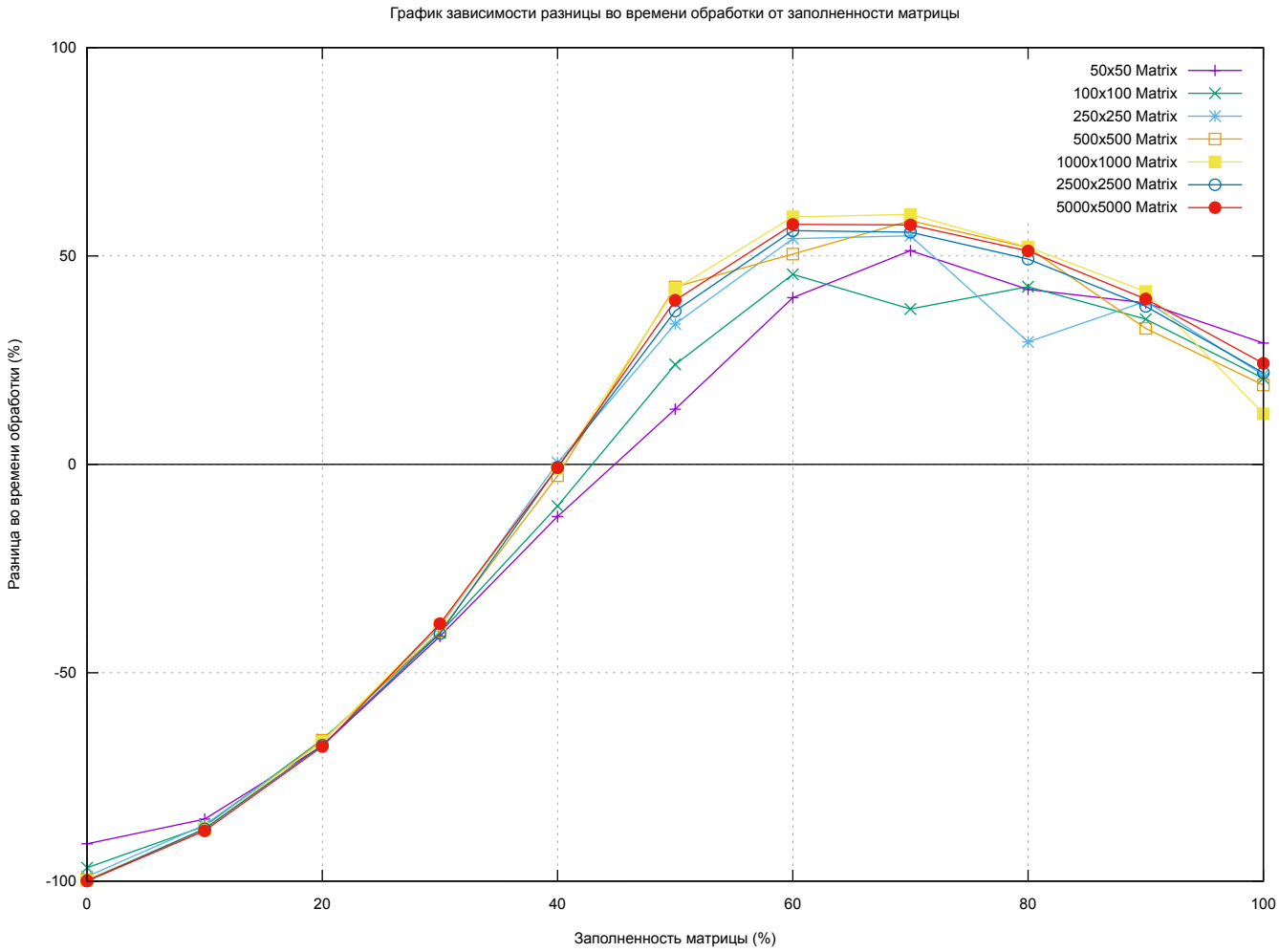
	Time (nanoseconds)	Memory (bytes)
Common method	73466600	100020048
Chang & Gustavson	102617600 (+39.68%)	270094088 (+170.04%)

[5000 x 5000, 100%]

	Time (nanoseconds)	Memory (bytes)
Common method	73162800	100020048

	Time (nanoseconds)	Memory (bytes)
Chang & Gustavson	90893600 (+24.23%)	300100088 (+200.04%)





Таким образом можно заметить, что процент зполненности при котором разреженное представление матрицы начинает требовать больше памяти, и больше времени на обработку, чем обычное составляет `~30%` и `~40%` соответственно

8. Контрольные вопросы

1. Что такое разреженная матрица, какие схемы хранения таких матриц Вы знаете?

Разреженная матрица - матрица, которая содержит большое количество нулевых элементов. Наиболее распространенные виды представления разреженной матрицы:

CSR (Compressed sparse row):

- `Data` - массив с ненулевыми элементами матрицы в порядке обхода по строкам
- `Col` - массив с номерами столбцов матрицы, параллельный массиву ненулевых элементов
- `Row offset` - массив, в котором для каждой строки матрицы, хранится индекс элемента из `Data`, который является первым ненулевым элементом этой строки

CSC (Compressed sparse column):

- `Data` - массив с ненулевыми элементами матрицы в порядке обхода по столбцам
- `Row` - массив с номерами строк матрицы, параллельный массиву ненулевых элементов

- `Col offset` - массив, в котором для каждого столбца матрицы, хранится индекс элемента из `Data`, который является первым ненулевым элементом этого столбца

2. Каким образом и сколько памяти выделяется под хранение разреженной и обычной матрицы?

Количество памяти под разреженную матрицу = `NZ * el + iw * (N + NZ)`, где

> NZ - количество ненулевых элементов в матрице
> el - размер одного элемента матрицы
> N - количество строк матрицы
> iw - вес индекса

Количество памяти под обычную матрицу = `el * N * M`, где

> el - размер одного элемента матрицы
> N - количество строк матрицы
> M - количество столбцов матрицы

3. Каков принцип обработки разреженной матрицы?

Обрабатываются только ненулевые элементы, а нулевые - игнорируются

4. В каком случае для матриц эффективнее применять стандартные алгоритмы обработки матриц? От чего это зависит?

Стандартные алгоритмы обработки работают быстрее, если матрица заполнена более чем на `~40%`, потому что на этом а по памяти стандартное представление начинает выигрывать уже на `~30%` заполненности. Эффективность стандартных алгоритмов зависит от заполненности матрицы.

9. Выводы

Алгоритмы работы с большими разреженными матрицами могут дать существенные приросты в скорости обработки матриц и существенно снизить затраты памяти на их хранение, при малом количестве значащих элементов, но начинают замедлять обработку если матрица заполнена более чем на `~40%`, и требуют больше памяти, если матрица заполнена более чем на `~30%`. Поэтому программисты должны ответственно подходить к решению использовать особое представление для работы с разреженными матрицами, так как этот подход не только усложняет код и создает новые просторы для ошибок, но может и ухудшить производительность при неверном применении. Так же стоить отметить, что при работе с небольшими матрицами, также нет смысла реализовывать разреженную обработку матриц, ведь это добавит дополнительные трудности и усложнит разработку, что абсолютно того не стоит.