ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет вычислительной техники и информатики
Кафедра прикладной матиматики и информатики

НА КОНКУРС НА ЛУЧШУЮ РАБОТУ СТУДЕНТОВ ПО РАЗДЕЛУ
«Техническая кибернетика, информатика и вычислительная техника»

СТУДЕНЧЕСКАЯ НАУЧНАЯ РАБОТА

На тему: "Исследование методов организации данных в задачах разбиения графов больших размерностей"

Выполнила
ст. гр. ПО-01а Краснокутская М.В.
Руководитель
ст. пр. кафедры ПМИ Костин В.И.

Донецк - 2005

РЕФЕРАТ

Отчет содержит 44 с., 9 рис., 1 таблицу., 2 приложения, 9 источников.

Объект исследования – методы организации данных в задачах разбиения графов больших размерностей.

Цель работы – разработать программу, реализующую разные методы организации данных в задачах разбиения графов больших размерностей.

Метод исследования – ознакомление с алгоритмами разбиения графов и определения собственных значений, особенностями их реализации для графов больших размерностей, анализ способов организации данных для представления разреженных матриц больших размеров, разработка программной реализации этих способов, построение диаграммы зависимости времени умножения матрицы на вектор от размерности матрицы и степени ее разреженности.

Результат работы – программная реализации способов организации данных для представления разреженных матриц больших размеров, построение диаграммы зависимости времени умножения матрицы на вектор от размерности матрицы и степени ее разреженности.

ГРАФ, РАЗБИЕНИЕ ГРАФА, СОБСТВЕННЫЕ ЧИСЛА, СОБСТВЕННЫЕ ВЕКТОРА, ПАРАЛЛЕЛЬНЫЕ ВЫЧИСЛЕНИЯ

СОДЕРЖАНИЕ

ВВЕДЕНИЕ....4
1 ПОСТАНОВКА ЗАДАЧИ О РАЗБИЕНИ ГРАФА....6
2 АЛГОРИТМ СПЕКТРАЛЬНОЙ БИСЕКЦИИ....8
3 НАХОЖДЕНИЕ СОБСТВЕННЫХ ВЕКТОРОВ....12
3.1 Общие сведения о собственных векторах....12
3.2 Обзор алгоритмов....13
3.2.1 Определение наибольшего собственного значения методом итераций....15
3.2.1 Определение наименьшего собственного значения методом итераций....17
3.2.1 Определение промежуточных собственных значений методом итераций....17
4 ИССЛЕДОВАНИЕ МЕТОДОВ ОРГАНИЗАЦИИ ДАННЫХ....21
ВЫВОДЫ....29
ПЕРЕЧЕНЬ ССЫЛОК....30
ПРИЛОЖЕНИЕ А. ЛИСТИНГ ПОРОГРАММЫ....31
ПРИЛОЖЕНИЕ Б. ФАЙЛЫ РЕЗУЛЬТАТОВ....41

ВВЕДЕНИЕ

Параллельные вычисления позволяют значительно повысить эффективность и скорость обработки информации при решении современных задач. Такие задачи возникают в электромеханике, при оптимизации сложных систем, при прогнозировании погоды, моделировании разнообразных технических и природных процессов.

Одна из основных проблем, которая возникает в каждом параллельном вычислении, это распределение обработки данных между процессорами. Решением может быть использование математической модели, в основе которой лежит граф потоков данных (ГПД). Программа представляется набором вычислительных узлов-подзадач, которые имеют фиксированное количество информационных входов и выходов. Каждая подзадача выполняется на отдельном процессоре. Узлы-подзадачи – вершины графа потоков данных, а информационные потоки между ними – ребра графа. Оптимальное распределение обработки данных между процессорами минимизирует время выполнения всех вычислений. Задача распределения обработки данных на процессоры сводится к задаче разбиения графа. Необходимо разбить граф потоков данных так, чтобы количество связей между подграфами было минимальным. Практический опыт показал, что качество распределения задач между процессорами сильно влияет на производительность, что обусловило значительный интерес к алгоритмам разбиения графов [1].

К сожалению разбиение графа – NP-сложная задача, и поэтому все известные алгоритмы разбиения являются эвристическими и дают приближенный к оптимальному результат. Однако, не смотря на это ограничение было разработано много алгоритмов разбиения графов, дающих высококачественное разбиение за малое время [2]. Среди наиболее известных можно выделить алгоритм спектральной бисекции [1].

При программной реализации любого из этих алгоритмов встает задача выбора типа данных для представления информации о графе. Существуют

различные способы внутреннего представления графов в оперативной памяти ЭВМ, в том числе в виде списков (массивов) вершин и ребер, списков (массивов) смежности, матриц смежности, а также в виде комбинаций этих структур хранения. Выбор внутреннего представления оказывает решающее влияние на эффективность выполнения различных операций над графами [3].

Целью данной работы является исследование методов организации данных в задачах разбиения графов больших размерностей методом спектральной бисекции.

1 ПОСТАНОВКА ЗАДАЧИ О РАЗБИЕНИ ГРАФА

Исходным объектом для задач разбиения служит неориентированный граф. Пространством решений служит множество всевозможных разбиений графа на непересекающиеся подграфы. На разбиения графа могут накладываться ограничения D. Для различных задач, естественно, возможны различные ограничения.

Задачи декомпозиции имеют следующий набор основных критериев:

- число внешних соединений между подграфами;
- число подграфов.

Первый критерий определяется функционалом внешних связей. Смысл второго критерия очевиден: он просто равен числу подграфов разбиения. Постановки задач могут варьироваться в зависимости от свойств графа, которые задаются моделируемым объектом.

Пусть задан неориентированный взвешенный граф G(V,E) порядка n, где $V=\{v_1,\ldots,v_n\}$ – множество вершин; $E \subseteq V\times V$ – множество ребер.

Требуется определить разбиение множества вершин V графа G(V,E) на k – подмножеств $(V_1,\ldots,V_k)$ таким образом, чтобы для частей графа $G_1(V_1,E_1),\ldots, G_k(V_k,E_k)$ выполнялись следующие требования:

$$V_i \cap V_j = \varnothing, \text{ для } \forall\ i \neq j, \text{ где } i,j = \overline{1,k};$$

$$\bigcup_{i=1}^{k} V_i = V; \tag{1.1}$$

$$|V_1|=n_1,\ldots, |V_k|=n_k,\ n_1+\ldots+n_k=n,$$

Сечением разбиения $C(V_1,\ldots,V_k)$ будем называть совокупность ребер, соединяющих вершины, которые принадлежат разным подграфам.

В качестве критерия оптимальности Q, определяющего эффективность биразбиения (бисекции) $(V_1,\ldots,V_k)$ будем рассматривать вес сечения - сумма всех ребер сечения:

$$Q(V_1,V_2,\ldots,V_k) = \frac{1}{2}\sum_{L=1}^{k-1}\sum_{i\in E_L}\sum_{j\notin E_L} 1 \to min \qquad (1.2)$$

В этом случае оптимальным k-разбиением является решение $(V_1^*,\ldots,V_k^*)$ экстремальной задачи (1.2), то есть разбиение $(V_1^*,\ldots,V_k^*)$ с минимальным весом сечения $C(V_1^*,\ldots,V_k^*)$.

Частным случаем задачи k-разбиения является задача (1.1) бисекции графа – когда граф разбивается на два подграфа (k=2). В этом случае решение задачи разбиения графа (V_1,V_2) будем называть бисекцией.

Система требований (1.2), предъявленных к разбиению $(V_1,\ldots,V_k)$, определяет область поиска D задачи разбиения графа. Данная задача относится к задачам переборного типа и общее число допустимых решений |D| можно вычислить из выражения:

$$\frac{n!}{n_1!\cdot n_2!\cdot \ldots \cdot n_k!\cdot t!} \qquad (1.3)$$

где t – общее число подграфов, имеющих одинаковые размерности [4].

2 АЛГОРИТМ СПЕКТРАЛЬНОЙ БИСЕКЦИИ

Алгоритм спектральной бисекции основан на использовании собственных векторов и чисел. Этот метод получил большое внимание, так как дает хорошее соотношение между универсальностью, качеством и эффективностью. Идея использования собственных векторов для разбиения графов появилась в начале 70-ых в работах Донатана (Donath) и Хофмана (Hoffman) [5], Файдлера (Fieder) [6].

Граф G представлен набором вершин V и набором граней E. V_i соответствует вершине с индексом i, $E_{i,j}$ означает грань между V_i и V_j, и Σ_{Vi} и Σ_{Eij} означают суммы по вершинам и граням соответственно. В алгоритме спектральной бисекции в соответствии каждой вершине графа ставится переменная x такая, что $x_i = \pm 1$, таким образом, чтобы сумма всех x_i была равна 0. Первое условие подразумевает разбиение на два различных набора. Второе требует чтобы наборы были равного размера, учитывая четность исходного количества. Назовем вектор **x** вектор-индикатор, так как он показывает принадлежность каждой вершины к конкретному набору.

Определим функцию:

$$f(x) = \frac{1}{4} \sum_{Eij} (x_i - x_j)^2 \tag{2.1}$$

Она определяет число граней, пересекающихся между наборами (сечение), $(x_i - x_j)^2$ не ничего не вносит в сумму, если x_i и x_j имеют одинаковый знак, и вносит 4, если они имеют противоположный знак.

Далее преобразуем f(x) к матричной форме, так чтобы решение было более очевидным. Вначале, определим матрицу смежности:

$$A_{ij} = \begin{cases} 1, & \text{если} \ (V_i, V_j) \in E \\ 0, & \text{иначе} \end{cases} \tag{2.2}$$

Определим матрицу степени $D = diag(d_i)$, где d_i - это степень V_i, то есть число граней, смежных вершине V_i.

Преобразуем все следующим образом:

$$\sum_{Eij}(x_i - x_j)^2 = \sum_{Eij}(x_i^2 + x_j^2) - \sum_{Eij} 2x_i x_j \qquad (2.3)$$

$$\sum_{Eij} 2x_i x_j = \sum_{Vi}\sum_{Vj} x_i A_{ij} x_j = \sum_{Vi} x_i \sum_{Vj} A_{ij} x_j = x^T A x \qquad (2.4)$$

$$\sum_{Eij}(x_i^2 + x_j^2) = \sum_{Eij} 2 = 2|E| = \sum_{Vi} x_i^2 d_i = x^T D x \qquad (2.5)$$

Здесь x^T – транспонированный вектор x.

Получаем:

$$\sum_{Eij}(x_i - x_j)^2 = x^T D x - x^T A x = x^T (D - A) x \qquad (2.6)$$

Теперь определим матрицу Лапласа для графа G:

$$L_{ij} = \begin{cases} -1, & \text{если } (V_i, V_j) \in E, \; i \neq j \\ d_i, & \text{если } i = j \\ 0, & \text{иначе} \end{cases} \qquad (2.7)$$

Отсюда, $L = D - A$

$$f(x) = \frac{1}{4} x^T L x \qquad (2.8)$$

Соединяя это с ограничениями на вектор x, мы определяем дискретную проблему деления пополам:

$$\text{Минимизировать: } \frac{1}{4} x^T L x \qquad (2.9)$$
$$\text{При условии: } x^T I = 0, \quad x_i = \pm 1$$

Где I – это n-мерный вектор $(1,1,\ldots,1)^T$.

Так как разбиение графа – NP-трудная задача, необходимо ослабить ограничения дискретности на **x** и сформулировать новую непрерывную задачу:

$$\text{Минимизировать: } \frac{1}{4} x^T L x \qquad (2.10)$$
$$\text{При условии: } x^T I = 0, \quad x^T x = n$$

Эта непрерывная задача - только приближение к дискретной, и значения, определяющие ее решение, должны быть отображены назад к ± 1 в соответствии с некоторой соответствующей схемой. Идеально, когда решение близко к ± 1 .

Если u^1, u^2 … – нормализованные собственные векторы L с соответствующими собственными значениями $\lambda_1 < =\lambda_2 <= \lambda_3 \ldots$, то матрица L имеет следующие свойства:

- L – симметрична;
- u^i попарно ортогональны;
- $u^1 = n^{-0.5} 1$, $\lambda_1 = 0$;
- Если граф замкнутый, то только λ_1 принимает нулевое значение.

Затем выразим X в терминах собственных векторов L: $x = \Sigma \alpha_i u^i$, где α_i – вещественные константы, такие, что $\Sigma(\alpha_i)^2 = n$. Второе свойство гарантирует, что это всегда возможно. Заменой для x мы получаем функцию, для минимизации, зависящую от собственного значения матрицы Лапласа λ_2.

$f(x) = 0.25(\alpha_2^2 \lambda_2 + \alpha_3^2 \lambda_3 + \ldots + \alpha_n^2 \lambda n)$ начиная с $\lambda_1 = 0$.

Очевидно, что

$(\alpha_2^2 + \alpha_3^2 + \ldots + \alpha_n^2)\ \lambda_2 <= (\alpha_2^2\ \lambda_2 + \alpha_3^2\ \lambda_3 + \ldots + \alpha_n^2\ \lambda n)$ учитывая упорядоченность собственных величин $f(x) >= n\ \lambda_2/4$.

Мы можем минимизировать $f(x) = n\ \lambda_2/4$, выбирая $\mathbf{x} = \sqrt{n} u_2$. Такой вектор x удовлетворяет также ограничению (2.9): $x^T I = u_2^T u^i = 0$.

Полученный вектор x – решение непрерывной задачи. Остается решить задачу отображения вектора x к дискретному разделению. Для этого находится медиана значений x_i, и затем отображается вершины больше значения медианы в один набор, меньше - в другой. Если несколько вершин имеют значение медианы, то они распределяются, не нарушая равновесия. Это решение - самая близкая дискретная точка к непрерывному оптимуму.

3 НАХОЖДЕНИЕ СОБСТВЕННЫХ ВЕКТОРОВ

3.1 Общие сведения о собственных векторах

Пусть даны квадратная матрица A и ненулевой вектор-столбец u, умножив матрицу A на вектор u, получим вектор-столбец y:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ . & . & . & . \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \quad u = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

т.е.

$$y = A u \tag{3.1}$$

Если координаты y_i , ($i = 1,2,\dots n$) вектора y пропорциональны соответствующим координатам u_i вектора u с коэффициентом пропорциональности λ, то есть если $y_i = \lambda u_i$ и, следовательно,

$$y = \lambda u, \tag{3.2}$$

то ненулевой вектор-столбец u называется собственным вектором матрицы A, а коэффициент пропорциональности λ - собственным значением матрицы A. Так как $y = A u$ и $y = \lambda u$, то очевидно, что

$$A u = \lambda u, \tag{3.3}$$

Таким образом, если выполняется условие (3.3), то вектор u является собственным вектором матрицы A, соответствующим ее собственному значению λ.[7]

3.2 Обзор алгоритмов

При определении собственных значений и собственных векторов матриц решается одна из двух задач: определение всех собственных значений и принадлежащих им собственных векторов матриц или определение одного или нескольких собственных значений и принадлежащих им собственных векторов.

Первая задача состоит в развертывании характеристического определителя в многочлен n-й степени (т. е. в определении его коэффициентов) с последующим вычислением собственных значений λ_1, λ_2, …, λ_n, и, наконец, в определении координат собственного вектора $u^T = (u_1, u_2, ..., u_n)$. Данный метод не очень подходит для решения поставленной задачи для графов большой размерности, так как приводит к решению алгебраического уравнения высокой степени:

$$(-1)^n (\lambda^n - p_1 \lambda^{n-1} + p_2 \lambda^{n-2} - ... + (-1)^n p_n) = 0 \qquad (3.4)$$

n – размерность матрицы – число вершин графа;

p_i – коэффициентов характеристического многочлена;

λ определяется из уравнения (3.4) и принимает n значений λ_1, λ_2, … , λ_n, среди которых могут быть и равные.

Вторая задача заключается в определении собственных значений λ (одного или нескольких) итерационными методами без предварительного развертывания характеристического определителя.

Методы первой задачи являются точными, т. е. если их применить для матриц, элементы которых заданы точно (рациональными числами), и точно проводить вычисления (по правилам действий с обыкновенными дробями), то в результате будет получено точное значение коэффициентов характеристического многочлена, и координаты собственных векторов окажутся выраженными точными формулами через собственные значения.

Методы решения второй задачи — итерационные, здесь собственные значения получаются как пределы некоторых числовых последовательностей, так же как и координаты принадлежащих им собственных векторов. Поскольку эта методы не требует вычисления коэффициентов характеристического многочлена, они менее трудоемки [7].

Ниже рассматриваются некоторые итерационные методы нахождения собственных значений матрицы.

3.2.1 Определение наибольшего собственного значения методом итераций

На рисунке 3.1 показана блок-схема простейшего итерационного метода отыскания наибольшего собственного значения системы (3.3)

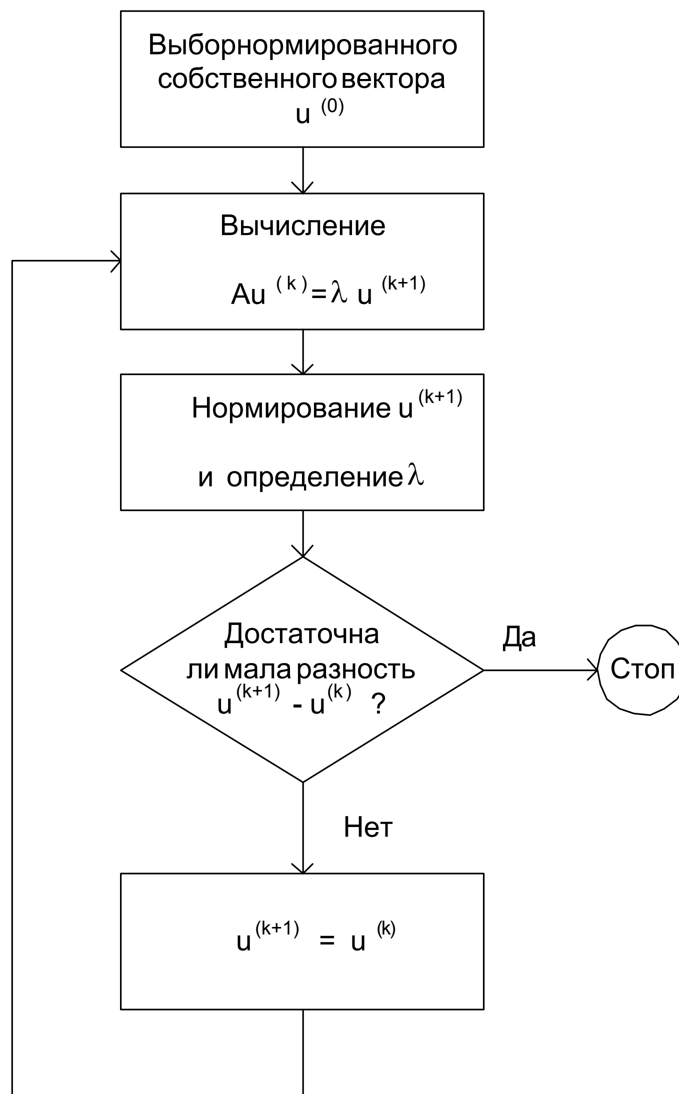

Рисунок 3.1 – Блок-схема алгоритма итерационного метода решения задач на собственные значения

Процедура начинается с пробного нормированного вектора $u^{(0)}$. Этот вектор умножается слева на матрицу A, и результат приравнивается

произведению постоянной (собственное значение) и нормированному вектору $u^{(1)}$. Если вектор $u^{(0)}$ совпадает с вектором $u^{(1)}$, то счет прекращается. В противном случае новый нормированный вектор используется в качестве исходного и вся процедура повторяется. Если процесс сходится, то постоянный множитель соответствует истинному наибольшему собственному значению, а нормированный вектор — соответствующему собственному вектору. Быстрота сходимости этого итерационного процесса зависит от того, насколько удачно выбран начальный вектор. Если он близок к истинному собственному вектору, то итерации сходятся очень быстро. На быстроту сходимости влияет также и отношение величин двух наибольших собственных значений. Если это отношение близко к единице, то сходимость оказывается медленной.

Например, возьмем матрицу

$$\begin{bmatrix} 10 & 5 & 6 \\ 5 & 20 & 4 \\ 6 & 4 & 30 \end{bmatrix},$$

и начальное приближение $u^{(0)} = (1,\ 0,\ 0)$. Результаты применения данного метода приведены в таблице 3.1

Таблица 3.1 – Результаты применения метода итераций

Номер итерации	Собственное значение λ	Собственный вектор		
		u_1	u_2	u_3
1		1	0	0
2	26	0.61923	0.66923	1
3	36.392	0.42697	0.56278	1
4	34.813	0.37583	0.49954	1
5	34.253	0.35781	0.46331	1
6	34	0.34984	0.44280	1
7	33.870	0.34580	0.43121	1
8	33.800	0.34362	0.42466	1
9	33.760	0.34240	0.42094	1

Продолжение таблицы 3.1

10	33.738	0.34171	0.41884	1
11	33.726	0.34132	0.41765	1
12	33.719	0.34110	0.41697	1
13	33.714	0.34098	0.41658	1
14	33.712	0.34091	0.41636	1

Получаем наибольшее собственное значение $\lambda_1 = 33.712$, и соответствующий ему собственный вектор $u_1 = (0.34091, 0.41636, 1)$.

3.2.1 Определение наименьшего собственного значения методом итераций

Для определения наименьшего собственного значения необходимо предварительно умножить исходную систему на матрицу, обратную A:

$$A^{-1} A u = \lambda A^{-1} u . \tag{3.6}$$

Если обе части этого соотношения умножим на $\frac{1}{\lambda}$, то получим

$$\frac{1}{\lambda} u = A^{-1} u . \tag{3.7}$$

Ясно, что это уже иная задача на собственное значение, для которой оно равно $\frac{1}{\lambda}$, а рассматриваемой матрицей является A^{-1}. Максимум $\frac{1}{\lambda}$ достигается при наименьшем λ. Таким образом, описанная выше итерационная процедура может быть использована для определения наименьшего собственного значения новой системы.

3.2.1 Определение промежуточных собственных значений методом итераций

Найдя наибольшее собственное значение, можно определить следующее за ним по величине, заменив исходную матрицу матрицей,

содержащей лишь оставшиеся собственные значения. Используем для этого метод, называемый методом исчерпывания. Для исходной матрицы А с известным наибольшим собственным значением λ_1 и собственным вектором u_1 можно воспользоваться принципом ортогональности собственных векторов, т. е. записать

$$\begin{aligned} u_i^T u_j &= 0 \quad \text{при} \quad i \neq j \\ u_i^T u_j &= 1 \quad \text{при} \quad i \neq j \end{aligned} . \tag{3.8}$$

Если образовать новую матрицу A^* в соответствии с формулой (3.8)

$$A^* = A - \lambda_1 u_1 u_1^T \tag{3.9}$$

то ее собственные значения и собственные векторы будут связаны соотношением

$$A^* u_i = \lambda_i u_i \tag{3.10}$$

Из приведенного выше выражения для матрицы A^* следует, что

$$A^* u_i = A u_i - \lambda_i u_1 u_1^T u_i \tag{3.11}$$

Здесь при i = 1 свойство ортогональности позволяет привести правую часть к виду

$$A u_1 = \lambda_1 u_1 \tag{3.12}$$

Но по определению собственных значений матрицы A это выражение должно равняться нулю. Следовательно, собственное значение λ_1 матрицы A^* равно нулю, а все другие ее собственные значения совпадают с собственными значениями матрицы A. Таким образом, матрица A^* имеет собственные значения 0, λ_2, λ_3, … , λ_n, и соответствующие собственные векторы u_1 , u_2 , …, u_n. В результате выполненных преобразований наибольшее собственное значение λ_1 было изъято, и теперь, чтобы найти следующее наибольшее собственное значение λ_2, можно применить к матрице A^* обычный итерационный метод. Определив λ_2 и u_2, повторим весь процесс, используя новую матрицу A^{**}, полученную с помощью A^*, λ_2 и u_2. Хотя на первый взгляд кажется, что этот процесс должен быстро привести к цели, он имеет существенные недостатки. При выполнении каждого шага погрешности в определении собственных векторов будут сказываться на точности определения следующего собственного вектора и вызывать накопление ошибок. Поэтому описанный метод вряд ли применим для нахождения более чем трех собственных значений, начиная с наибольшего или наименьшего [8].

Другой способ нахождения промежуточных собственных значений и векторов заключается в следующем. Если у матрицы A порядка n известен собственный вектор u_1, отвечающий собственному значению λ_1, то задачу о б определении остальных собственных векторов и значений можно свести к аналогичной задаче для некоторой матрицы A_1 порядка n – 1.

Для перехода к матрице A_1 обозначим первую строку матрицы A буквой r и условимся нормировать собственные векторы u_k матрицы A (определенные с точностью до скалярного множителя) так, чтобы их первая координата равнялась единице. Обозначим $B = A - u_1 r$; тогда из первого условия вытекает, что первая строка матрицы B состоит из одних нулей. Из того, что

$$r u_k = \lambda_k$$

$$A u_k = \lambda_k u_k$$

следует, что

$$B(u_k - u_1) = Au_k - u_1 r u_k - Au_1 + u_1 r u_1 =$$
$$= \lambda_k u_k - \lambda_k u_1 - \lambda_1 u_1 + \lambda_1 u_1 = \lambda_k (u_k - u_1),$$

то есть

$$B(u_k - u_1) = \lambda_k (u_k - u_1). \tag{3.13}$$

Обозначим через y_k столбец высоты n-1, получающийся из $u_k - u_1$ отбрасыванием верхнего нулевого элемента; через A_1 обозначим матрицу порядка n-1, получающуюся из B зачеркиванием первой строки и первого столбца. Тогда из формулы (3.13) следует, что $A_1 y_k = \lambda y_k$, то есть y_k $(k = 2, 3,\ldots,n)$ – собственный вектор матрицы A_1, отвечающий собственному значению λ_k; вектор y_1 – нулевой. Найдя вектор y_k, надо к нему сверху приписать нуль, после чего к полученному вектору прибавить u_1, тогда мы получим u_k; при этом u_k нормируется так, чтобы $ru_k = \lambda_k$, тогда из формулы (3.13) легко получить, что $Au_k = \lambda_k u_k$ [9].

Данный метод лучше подходит для определения собственных векторов и значений матрицы Лапласа, так как он понижает ее размерность и нам известен $u_1 = n^{-0.5} 1$, $\lambda_1 = 0$.

4 ИССЛЕДОВАНИЕ МЕТОДОВ ОРГАНИЗАЦИИ ДАННЫХ

Как уже говорилось, при реализации алгоритма встает задача выбора типа данных для представления информации о графе.

Задание графов с помощью матриц удобно для алгоритмов, использующие матричные вычисления (например, алгоритм спектральной бисекции). Однако, при обработке графа большой размерности (n=1000, 10000), матрицы занимают слишком много памяти. Следует учесть, что матрицы графов потоков данных довольно разрежены, т. е. матрицы содержат много нулей.

Для алгоритма спектральной бисекции и рассмотренного выше алгоритма для нахождения собственных значений основной операцией является умножение матрицы Лапласа на вектор. Рассмотрим два представления матрицы при программной реализации алгоритма. Первый способ – двумерный массив, второй – массив динамических массивов (первый элемент динамического массива – число ненулевых элементов в строке матрицы)

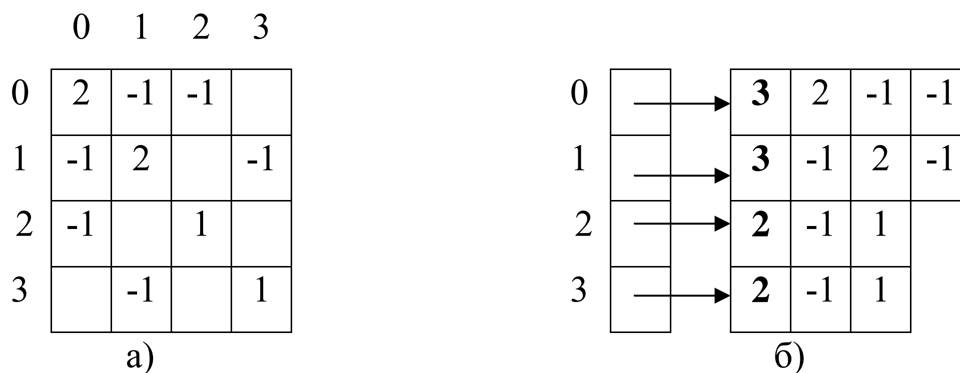

Рисунок 4.1 – Способы представления матрицы; а) двумерный массив; б) массив динамических массивов

Представляя матрицу Лапласа двумерным массивом мы занимаем n^2 единиц памяти. Массив динамических массивов занимает $2n+m$ единиц памяти, где $m \le n^2$ – число ребер графа.

Найдем, при каких значениях m выгоднее использовать вторую структуру (когда она занимает меньше памяти):

$$2n + m \le n^2;$$

$$m \le n^2 - 2n.$$

Рассмотрим программную реализацию этих способов представления матриц (полный листинг программы приведен в Приложении А).

```
BYTE *m_matrix;
double *m_LaplasMatrix;

m_matrix = (BYTE*)LocalAlloc(LPTR, m_vertixes*m_vertixes);

m_LaplasMatrix =
     (double*)LocalAlloc(LPTR, m_vertixes*m_vertixes*sizeof(double));
```

Рисунок 4.2 – Реализация двумерного массива на языке С++

На рисунке 4.2 показана реализация двумерного массива на языке С++. Переменная m_matrix содержит указатель на двумерный массив, содержащий матрицу смежности для графа, m_vertixes задает число вершин графа, m_LaplasMatrix - указатель на двумерный массив, содержащий матрицу Лапласа. Объем занимаемой памяти – $(m_vertex)^2 \cdot 1$ байт $+ (m_vertex)^2 \cdot 8$ байтов $= (m_vertex)^2 \cdot 9$ байтов (1 байт – размер типа BYTE, 8 байтов – размер типа данных double).

```
WORD **m_lists;
double **m_LaplasLists;

m_lists = (WORD**)malloc(m_vertixes*sizeof(WORD*));
for(int i=0; i<m_vertixes; i++){
      int k=0;
      for(int j=0; j<m_vertixes; j++){
          if(isEdge(i, j)) k++;
      }
      m_lists[i] = (WORD*)LocalAlloc(LPTR, (k+1)*sizeof(WORD));
}

m_LaplasLists = (double**)malloc(m_vertixes*sizeof(double*));

for(int i=0; i<m_vertixes; i++){
   m_LaplasLists[i] =
        (double*)LocalAlloc(LPTR, (m_lists[i][0])*sizeof(double));
}
```

Рисунок 4.3 – Реализация массива динамических массивов на языке C++

На рисунке 4.3 показана реализация массива динамических массивов на языке C++. Переменная m_mlists содержит указатель на массив указателей. Каждый из m_mlists[i] – указатель на динамический массив, содержащий список смежности для графа. Первый элемент каждого динамического массива содержит число смежных вершин - k. Функция isEdge(i, j) определяет, есть ли ребро между i-ой и j-ой вершинами.

Переменная m_LaplasLists содержит указатель на массив указателей. Каждый из m_LaplasLists[i] – указатель на динамический массив, содержащий ненулевые элементы матрицы Лапласа. Объем занимаемой памяти :

$$(4\ \text{байта} + 2\ \text{байта}) \cdot m_vertixes + 2\ \text{байта} \cdot m_edges +$$
$$4\ \text{байта} \cdot m_vertixes + 8\ \text{байта} \cdot m_edges =$$
$$(22\, m_vertixes + 10\, m_edges)\ \text{байтов},$$

где m_edges – число ребер графа;

4 байта – размер указателя;

2 байта – размер типа данных WORD;

8 байтов – размер типа данных double.

Рассмотрим когда массив динамических массивов занимает меньше памяти чем двумерный массив:

$$22\,m_vertixes + 10\,edges \le 9\,m_vertixes^2;$$

$$m_edges < 0.9m_vertixes^2 - 2{,}2\,m_vertixes \tag{4.1}$$

Рассмотрим само умножение матрицы на вектор. На рисунке 4.4 приведено умножение двумерного массива на вектор. Доступ к элементам двумерного массива и вектора происходит по индексу, цикл идет по всем элементам массивов.

```
double buf;
double * new_vect;

new_vect = (double*)LocalAlloc(LPTR, m_vertixes*sizeof(double));
for(int i=0; i<m_vertixes; i++){
    new_vect[i] = 1;
}
for(i=0;  i<m_vertixes; i++)
{
      buf = 0;
      for(int j=0; j<m_vertixes; j++)
      {
           buf += m_LaplasMatrix[i*m_vertixes+j] * vector[j];
      }
     new_vect[i] = buf;
}
```

Рисунок 4.4 – Реализация умножения двумерного массива на вектор на языке С++

На рисунке 4.5 приведено умножение массива динамических массивов на вектор. В отличие от первого случая, здесь не осуществляется проход по всем элементам вектора. В умножении участвуют только те элементы, которые соответствуют ненулевым элементам строки матрицы. Для разреженных матриц это существенно сокращает время умножения.

```
double buf;
double * new_vect;
new_vect = (double*)LocalAlloc(LPTR, m_vertixes*sizeof(double));
for(int i=0; i<m_vertixes; i++){
      buf = 0;
      for(int j=0; j<m_lists[i][0]; j++){
          buf+=vector[m_lists[i][j+1]]*(m_LaplasLists[i][j]);
      }
     new_vect[i] = buf;
}
```

Рисунок 4.5 – Реализация умножения массива динамических массивов на вектор на языке С++

На рисунках 4.6, 4.7, 4.8 приведены диаграммы зависимости времени умножения матриц от степени разреженности матрицы для матриц размерностью 10×10, 100×100 и 1000×1000 соответственно. Для вычисления каждой точки графика задавалась размерность матрицы, число ненулевых элементов. По этим данным случайным образом создавалось 10 матриц. Каждая матрица умножалась на вектор несколько раз (10000, 1000, 100) в зависимости от размера матрицы. Для построения диаграммы бралось среднее время умножения всех этих матриц на вектор. Вычисления проводились на компьютере с процессором Intel Celeron с тактовой частотой 2,93 GHz и оперативной памятью 1GB. Файлы результатов приведены в приложении Б.

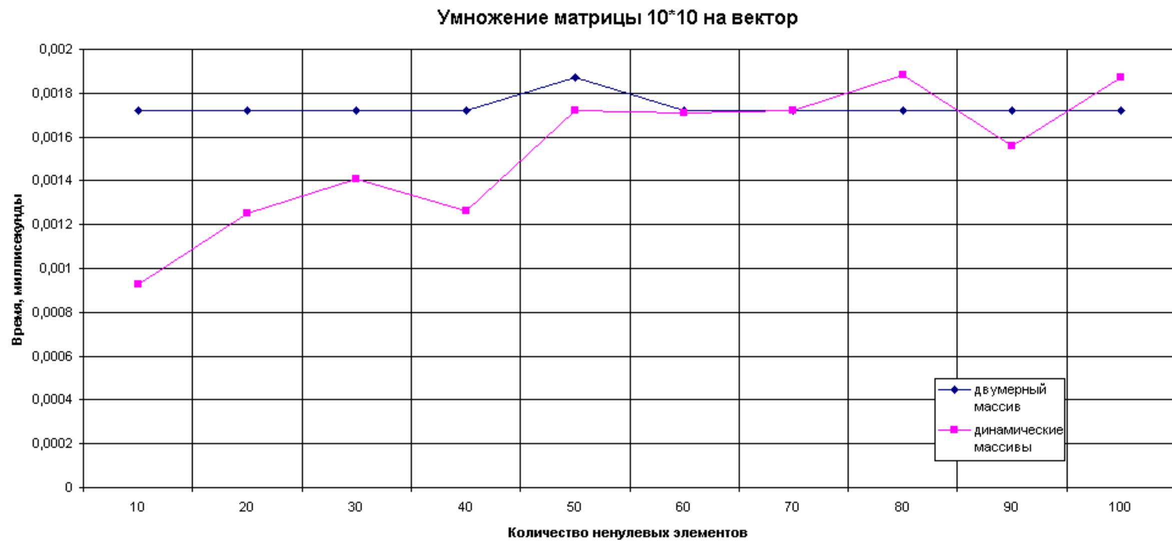

Рисунок 4.6 – Время умножения матрицы 10×10 на вектор

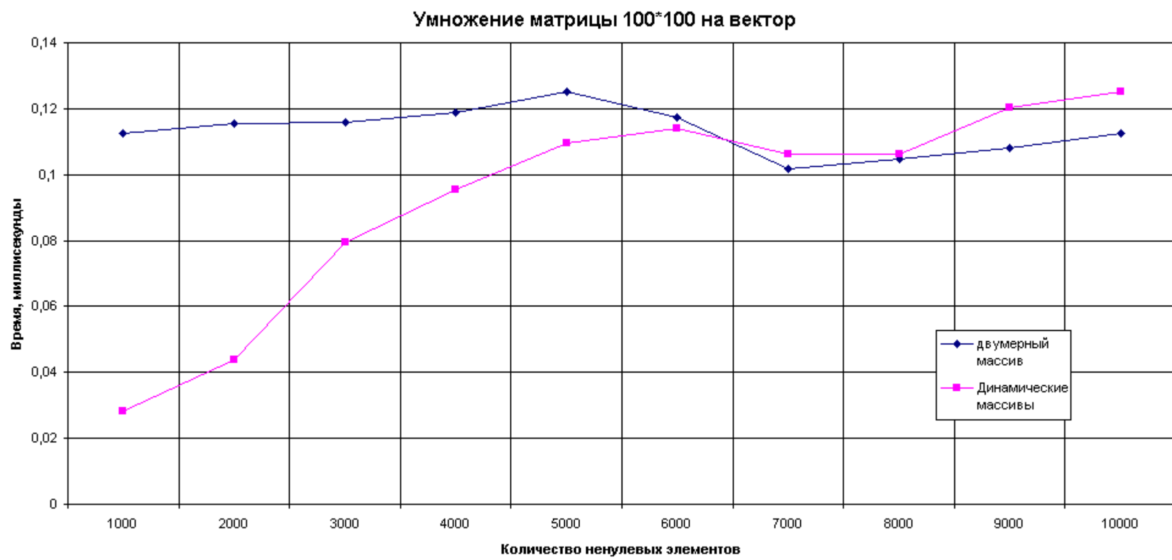

Рисунок 4.7 – Время умножения матрицы 100×100 на вектор

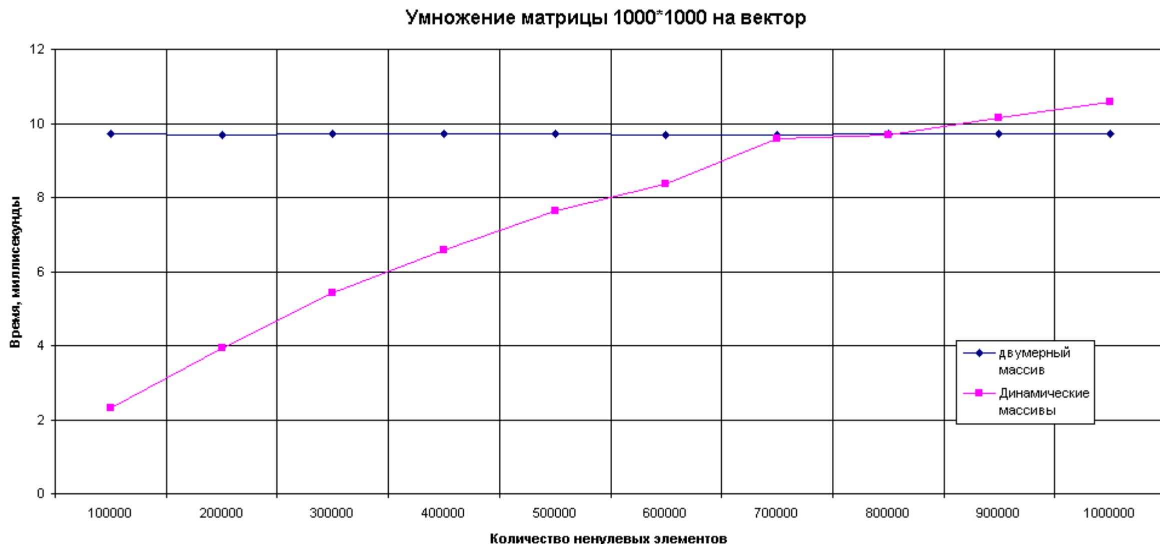

Рисунок 4.8 – Время умножения матрицы 1000×1000 на вектор

Как видно, степень разреженности матрицы мало влияет на скорость умножения, когда матрица задана двумерным массивом. При реализации матрицы с помощью массива динамических массивов видно, что с чем больше число ненулевых элементов в матрице, тем больше время умножения. Причем до определенного значения это время меньше соответствующего, для двумерного массива. Так для матрицы 10×10 – это приблизительно 65 ненулевых элементов, для матрицы 100×100 – 650, для матрицы 1000×1000 – 7000. То есть можно сказать что при количестве нулевых элементов больше 35% использование массива динамических массивов является более оптимальным по времени. Определим, какой дожжен быть размер матрицы, чтобы использование данного способа задания матрицы было также оптимально по занимаемому объему памяти. Воспользуемся формулой (4.1)

$$(1-0.35)m_vertixes < 0.9m_vertixes^2 - 2.2m_vertixes$$

$$m_vertixes > 15$$

То есть для матриц размерность больше 15×15 и с количеством нулевых элементов больше 35% использование массива динамических

массивов является оптимальным и по времени и по занимаемому объеме памяти.

ВЫВОДЫ

В ходе данной работы были рассмотрены алгоритмы разбиения графов и определения собственных значений. Рассматривались особенности их реализации для графов больших размерностей. Были проанализированы два способа организации данных для представления разреженных матриц больших размеров. Была разработана программная реализация этих двух способов, построены диаграммы зависимости времени умножения матрицы на вектор от размерности матрицы и степени ее разреженности.

Представление матрицы двумерным массивом проще в реализации. Объем занимаемой памяти и время умножения матрицы на вектор при такой реализации – постоянны. Такой способ оптимально подходит для неразреженных матриц с большим числом ненулевых элементов. Разреженные матрицы лучше представлять массивом динамических массивов, так как при такой реализации объем занимаемой памяти и время умножения матрицы на вектор зависит от размерности и степени разреженности графа.

ПЕРЕЧЕНЬ ССЫЛОК

1. B. Hendricson and R. Leland. Multidemensional spectral load balancing. Sandia National Laboratories, Albuquerque, NM, 1993
2. Bradford L. Chamberlain. Graph Partitioning Algorithms for DistributingWorkloads of Parallel Computations. 1998
3. AGraph: библиотека классов для работы с помеченными графами http://www.caravan.ru/~alexch/AGraph
4. Д. И. Батищев, Н. В. Старостиным. Методические указания по проведению лабораторных работ «задачи декомпозиции графов» по курсу «Эволюционно-генетические алгоритмы решения оптимизационных задач» для студентов факультета ВМК специальности «Прикладная информатика». Нижегородский государственный университет, 2001. – 13с.
5. W. Donath and A. Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. IBM Technical Disclosure Bulletin, 1972
6. M. Fiedler. Algebraic connectivity of graphs. Czechoslovak Math., 1973
7. Н. И. Данилина, Н. С. Дубровская, О. П. Кваша, Г. Л. Смирнов. Вычислительная математика. – М.: Высшая школа, 1985. – 472с.
8. Т. Шуп. Решение инженерных задач на ЭВМ: Практическое руководство. Пер. с англ. – М.: Мир, 1982. – 238 с.
9. А. Д. Мышкис. Математика для втузов. Специальные курсы. М.: Наука, 1971. – 632 с.

ПРИЛОЖЕНИЕ А. ЛИСТИНГ ПОРОГРАММЫ

```
// Graph.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "Graph.h"
#include "GraphList.h"
#include <conio.h>
#include "DynamicArray.h"
#include <Mmsystem.h>


#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

// The one and only application object

CWinApp theApp;

using namespace std;

double do_matrix_graph(int n_n, int n_rarity);
double do_list_graph(int n_n, int n_rarity);
CStdioFile file1, file2;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
      int nRetCode = 0;


      // initialize MFC and print and error on failure
if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
{
      // TODO: change error code to suit your needs
      cerr << _T("Fatal Error: MFC initialization failed") << endl;
            nRetCode = 1;
      }
      else
      {
      double time;
        CString str;

      file1.Open("logfile_matrix.txt",        CFile::modeCreate        |
CFile::modeWrite| CFile::modeNoTruncate);
            file1.SeekToEnd();
            for(int i=10; i<=1000; i*=10){
                str.Format("\nGraph contains %d vertixes\n", i);
                file1.WriteString(str);
                  for(int r=10; r<=100; r+=10){
                        int edges = (int)i*i*r/100;
                        str.Format("Graph contains %d edges\n", edges);
                        file1.WriteString(str);
                        time=0;
                        for(int j=0; j<10; j++){
                              time+=do_matrix_graph(i, r);
                        }
                        time/=10;
                str.Format("%f\n", time);
```

```
                              file1.WriteString(str);
                              file1.Close();
                              file1.Open("logfile_matrix.txt",
CFile::modeCreate | CFile::modeWrite| CFile::modeNoTruncate);
                              file1.SeekToEnd();
                        }
                  }
            file1.Close();

                  file2.Open("logfile_list.txt",      CFile::modeCreate      |
CFile::modeWrite| CFile::modeNoTruncate);
                  file2.SeekToEnd();
                  for(int i=10; i<=1000; i*=10){
                        str.Format("\nGraph contains %d vertixes\n", i);
                      file2.WriteString(str);
                        for(int r=10; r<=100; r+=10){
                              int edges = (int)i*i*r/100;
                              str.Format("Graph contains %d edges\n", edges);
                              file2.WriteString(str);
                              time=0;
                              for(int j=0; j<10; j++){
                                    time+=do_list_graph(i, r);
                              }
                              time/=10;
                              str.Format("%f\n", time);
                              file2.WriteString(str);
                              file2.Close();
                              file2.Open("logfile_list.txt", CFile::modeCreate
| CFile::modeWrite| CFile::modeNoTruncate);
                              file2.SeekToEnd();
                        }
                  }
                  file2.Close();
            }
            return nRetCode;
      }
      double do_matrix_graph(int n_n, int n_rarity){
                  char c;
                  CGraph g;
                        int n = n_n;
                        float rarity = n_rarity;
                        CString str;
                        g.CreateGraph(n, rarity);
                  g.CreateLaplasMatrix();
                        double * vector;
                      vector = (double*)LocalAlloc(LPTR, n*sizeof(double));
                      for(int i=0; i<n; i++){
                          vector[i] = i%3;
                        }
                        int rounds = 100000/n;
                        WORD t1 = GetTickCount();
                  double * mass;
                        for(i=0; i<rounds; i++){
                               mass = g.MultiplyLaplasMatrix(vector);
                        }
                        WORD t2 = GetTickCount();
                        LocalFree( vector );
                        LocalFree(mass);
                  return (double)(t2-t1)/rounds;

      }

      double do_list_graph(int n_n, int n_rarity)
      {
```

```
char c;
      CGraphList g;
      int n = n_n;
      float rarity = n_rarity;
      CString str;
int edges = (int)n*n*rarity/100;
      g.CreateGraph(n, rarity);
      g.CreateLaplasMatrix();
      double * vector;
    vector = (double*)LocalAlloc(LPTR, n*sizeof(double));
    for(int i=0; i<n; i++){
        vector[i] = i%3;
      }
int rounds = 100000/n;
      WORD t1 = GetTickCount();
double * mass;
      for(i=0; i<rounds; i++){
      mass = g.MultiplyLaplasMatrix(vector);
      }
      WORD t2 = GetTickCount();
LocalFree( vector );
      LocalFree(mass);
return (double)(t2-t1)/rounds;

}
```

```
// CGraph.h : header file
// CGraph class
class CGraph
{
public:   CGraph();          // protected constructor used by dynamic creation
// Attributes
public:
        BYTE *m_matrix;
        double *m_LaplasMatrix;
        int m_vertixes;
        float m_rarity;
        int m_edges;
// Operations
public:
   void CreateGraph(int n, float rarity); /* sparse matrix */
        void PrintGraph();
        void PrintGraph(CStdioFile * log);
        bool isRowCovered(int row);
        bool isAllRowsCovered();
        bool isEdge(int i, int j);
        void addEdge(int i, int j);
        void CreateLaplasMatrix();
        void PrintLaplasMatrix();
        void PrintLaplasMatrix(CStdioFile * file);
        double * MultiplyLaplasMatrix(double * vector);
// Implementation
```

```
public:
            virtual ~CGraph();
};
// GraphList.h : header file
#include "CGraph.h"
#include "DynamicArray.h"
// CGraphList
class CGraphList : public CGraph
{
// Construction
public:
            CGraphList():CGraph(){};
   void CreateGraph(int n, float rarity);
            void PrintGraph();
            void PrintGraph(CStdioFile* log);
            void CreateLaplasMatrix();
            void PrintLaplasMatrix();
            void PrintLaplasMatrix(CStdioFile* log);
            double* MultiplyLaplasMatrix(double * vector);

// Attributes
public:
  WORD **m_lists;
  double **m_LaplasLists;
// Implementation
public:
            virtual ~CGraphList();
};

// CGraph.cpp : implementation file
//
#include "stdafx.h"
#include "Graph.h"
#include "CGraph.h"
#include <iostream.h>
#include <stdlib.h>
/////////////////////////////////////////////////////////////////////////////
// CGraph
CGraph::CGraph()
{
            m_matrix = NULL;
            m_LaplasMatrix = NULL;
```

```
		m_vertixes = 0;
		m_edges = 0;
		m_rarity = 0;
}
CGraph::~CGraph(){
		if(m_matrix != NULL) LocalFree(m_matrix);
		if(m_LaplasMatrix != NULL) LocalFree(m_LaplasMatrix);
}
void CGraph::CreateGraph(int n, float rarity) /* sparse matrix */{
		m_vertixes = n;
		m_rarity = rarity;
		int items = 0;
		int i=0, j=0;
   m_matrix = (BYTE*)LocalAlloc(LPTR, m_vertixes*m_vertixes);
		for(i=0; i<m_vertixes; i++)
			for(j=0; j<m_vertixes; j++)
				m_matrix[i*m_vertixes+j]=0;
		float c = (float)m_vertixes/(float)RAND_MAX; //RAND_MAX 0x7fff 32767
		m_edges = (int)m_vertixes*m_vertixes*rarity/100;
		cout << n <<" vertixes    " << "rarity: "<< rarity << "% = " << m_edges << " edges"<<endl;
   int ddd = (RAND_MAX-2)*c;
   for(items=0; items<m_vertixes; items++){
			do{
			  j = rand()*c;
			}while(j==items);
			if(!isEdge(items, j)){
				addEdge(items, j);
			}
		}
		while(items < m_edges){
			// индексы ячейки матрицы смежности графа
			i = rand()*c;
			j = rand()*c;
			if(i==j) continue;
			addEdge(i, j);
			items++;
		}
return;
}

// возвращет 1 если ребро между i-ой и j-ой
// вершинами существует, иначе - 0
```

```
bool CGraph::isEdge(int i, int j){
	return m_matrix[i*m_vertixes+j];
}
// Добавляет ребро между i-ой и j-ой
// вершинами
void CGraph::addEdge(int i, int j){
	m_matrix[i*m_vertixes+j] = 1;
	m_matrix[j*m_vertixes+i] = 1;
}
//Печатает граф в виде списков смежности
void CGraph::PrintGraph()
{
	cout<<endl;
	cout<<"Graph:"<<endl;

	cout << endl;
	for(int i=0; i<m_vertixes; i++){
		cout << i <<"->";
		for(int j=0; j<m_vertixes; j++){
			if(isEdge(i, j)) cout <<j<<" ";
		}
	cout << endl;
	}
	return;
}
//Печатает граф в виде списков смежности
// в файл
void CGraph::PrintGraph(CStdioFile *log)
{
	char end[2];
	end[0] = '\n';
	end[1] = 0;
	log->WriteString("\nGraph:");
	log->WriteString(end);
	char s[5];
	for(int i=0; i<m_vertixes; i++){
		itoa(i, s, 5);
		cout << s;
		log->WriteString(s);
		log->WriteString(" -> ");
		for(int j=0; j<m_vertixes; j++){
			if(isEdge(i, j)){
```

```
                        itoa(j, s, 5);
                        log->WriteString(s);
                        log->WriteString(" ");
                    }
                }
                log->WriteString(end);
        }
        return;
}
//Создает матрицу Лапласа для графа
void CGraph::CreateLaplasMatrix(){
        m_LaplasMatrix = (double*)LocalAlloc(LPTR, m_vertixes*m_vertixes*sizeof(double));
        int di;
        for(int i=0; i<m_vertixes; i++)
        {       di=0;
                for(int j=0; j<m_vertixes; j++)
                {
                  m_LaplasMatrix[i*m_vertixes+j] = - m_matrix[i*m_vertixes+j];
                   if(m_matrix[i*m_vertixes+j]==1) di++;;
                }
                m_LaplasMatrix[i*m_vertixes+i] = di;
        }
}
// Печатает матрицу Лапласа
void CGraph::PrintLaplasMatrix(){
        cout<<endl<<endl;
        cout<<"Laplas Matrix:"<<endl;
        CString s;
        for(int i=0;  i<m_vertixes; i++)
        {       for(int j=0; j<m_vertixes; j++)
                {
                 s.Format("%6.2f ", m_LaplasMatrix[i*m_vertixes+j]);
                  cout << s;
                }
        cout << endl;
        }
}
// Печатает матрицу Лапласа
void CGraph::PrintLaplasMatrix(CStdioFile * log){
        char end[2];
        end[0] = '\n';
        end[1] = 0;
```

```
	log->WriteString("\nLaplas matrix:");
	log->WriteString(end);
	CString s;
	for(int i=0;  i<m_vertixes; i++)
	{	for(int j=0; j<m_vertixes; j++)
		{
	s.Format("%6.2f ", m_LaplasMatrix[i*m_vertixes+j]);
	log->WriteString(s);
		}
		log->WriteString(end);
	}}
// Умножает матрицу Лапласа на вектор
double* CGraph::MultiplyLaplasMatrix(double * vector){
	double buf;
	double * new_vect;
	new_vect = (double*)LocalAlloc(LPTR, m_vertixes*sizeof(double));
   for(int i=0; i<m_vertixes; i++){
		new_vect[i] = 1;
	}
	for(i=0;  i<m_vertixes; i++)
	{	buf = 0;
		for(int j=0; j<m_vertixes; j++)
		{
	buf += m_LaplasMatrix[i*m_vertixes+j] * vector[j];
		}
    new_vect[i] = buf;
	}
	return new_vect;
}
// GraphList.cpp : implementation file
//

#include "stdafx.h"
#include "Graph.h"
#include "GraphList.h"
#include <iostream.h>
#include <malloc.h>
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

```
///////////////////////////////////////////////////////////////////////
// CGraphList
CGraphList::~CGraphList(){}
void CGraphList::CreateGraph(int nv, float rarity){
        CGraph::CreateGraph(nv, rarity);
        m_lists = (WORD**)LocalAlloc(LPTR, m_vertixes*sizeof(WORD*));
        for(int i=0; i<m_vertixes; i++){
                int k=0;
                for(int j=0; j<m_vertixes; j++){
                   if(isEdge(i, j)) k++;
                }
                m_lists[i] = (WORD*)LocalAlloc(LPTR, (k+1)*sizeof(WORD));
                m_lists[i][0]=k;
                k=1;
                for(j=0; (j<m_vertixes)&&(k<=nv); j++){
                  if((isEdge(i, j))||(i==j)) {
                          m_lists[i][k] = j;
                          k++;
                        };      }       }       }
void CGraphList::PrintGraph(){
        for(int i=0; i<m_vertixes; i++){
                cout << i << " -> ";
                for(int j=1; j<=m_lists[i][0]; j++)
                {
                        cout<<m_lists[i][j]<<" ";
                }
                cout << "\n";
        }}
void CGraphList::PrintGraph(CStdioFile* log){}
void CGraphList::CreateLaplasMatrix(){
        m_LaplasLists = (double**)LocalAlloc(LPTR, m_vertixes*sizeof(double*));
        int r = sizeof(double*);
        int n;
        double item;
        for(int i=0; i<m_vertixes; i++){
                m_LaplasLists[i] = (double*)LocalAlloc(LPTR, (m_lists[i][0])*sizeof(double));
                n = m_lists[i][0];
                for(int j=0; j<n; j++){
               item = m_lists[i][j+1];
                        if(item==i) item = n-1;
                        else item = -1;
```

```
                        m_LaplasLists[i][j]=item;
                    }           }}
void CGraphList::PrintLaplasMatrix(){
        for(int i=0; i<m_vertixes; i++){
                int k=0;
                for(int j=0; j<m_vertixes; j++){
                        if(j==m_lists[i][k+1]){
                                cout << m_LaplasLists[i][k]<<" ";
                                k++;
                        }else{
                                cout << "0 ";
                        }                       }
                cout <<"\n";
        }}
void CGraphList::PrintLaplasMatrix(CStdioFile* log){}
double* CGraphList::MultiplyLaplasMatrix(double * vector){
        double buf;
        double * new_vect;
        new_vect = (double*)LocalAlloc(LPTR, m_vertixes*sizeof(double));
        for(int i=0; i<m_vertixes; i++){
                buf = 0;
                for(int j=0; j<m_lists[i][0]; j++){
                        buf+=vector[m_lists[i][j+1]]*(m_LaplasLists[i][j]);
                }
   new_vect[i] = buf;
        }
 return new_vect;
}
```

ПРИЛОЖЕНИЕ Б. ФАЙЛЫ РЕЗУЛЬТАТОВ

Файл logfile_matrix.txt

```
Двумерный массив
Graph contains 10 vertixes
Graph contains 10 edges
0.000930 milliseconds
Graph contains 20 edges
0.001250 milliseconds
Graph contains 30 edges
0.001410 milliseconds
Graph contains 40 edges
0.001260 milliseconds
Graph contains 50 edges
0.001720 milliseconds
Graph contains 60 edges
0.001710 milliseconds
Graph contains 70 edges
0.001720 milliseconds
Graph contains 80 edges
0.001880 milliseconds
Graph contains 90 edges
0.001560 milliseconds
Graph contains 100 edges
0.001870 milliseconds

Graph contains 100 vertixes
Graph contains 1000 edges
0.028100 milliseconds
Graph contains 2000 edges
0.043800 milliseconds
Graph contains 3000 edges
0.079600 milliseconds
Graph contains 4000 edges
0.095300 milliseconds
Graph contains 5000 edges
0.109400 milliseconds
Graph contains 6000 edges
0.114000 milliseconds
Graph contains 7000 edges
0.106300 milliseconds
```

```
Graph contains 8000 edges
0.106200 milliseconds
Graph contains 9000 edges
0.120500 milliseconds
Graph contains 10000 edges
0.125000 milliseconds

Graph contains 1000 vertixes
Graph contains 100000 edges
2.311000 milliseconds
Graph contains 200000 edges
3.921000 milliseconds
Graph contains 300000 edges
5.422000 milliseconds
Graph contains 400000 edges
6.591000 milliseconds
Graph contains 500000 edges
7.628000 milliseconds
Graph contains 600000 edges
8.375000 milliseconds
Graph contains 700000 edges
9.591000 milliseconds
Graph contains 800000 edges
9.671000 milliseconds
Graph contains 900000 edges
10.158000 milliseconds
Graph contains 1000000 edges
10.562000 milliseconds
```

Файл logfile_lists.txt

Массив динамических массивов

```
Graph contains 10 vertixes
Graph contains 10 edges
0.000930 milliseconds
Graph contains 20 edges
0.001250 milliseconds
Graph contains 30 edges
0.001410 milliseconds
Graph contains 40 edges
0.001260 milliseconds
Graph contains 50 edges
```

```
0.001720 milliseconds
Graph contains 60 edges
0.001710 milliseconds
Graph contains 70 edges
0.001720 milliseconds
Graph contains 80 edges
0.001880 milliseconds
Graph contains 90 edges
0.001560 milliseconds
Graph contains 100 edges
0.001870 milliseconds

Graph contains 100 vertixes
Graph contains 1000 edges
0.028100 milliseconds
Graph contains 2000 edges
0.043800 milliseconds
Graph contains 3000 edges
0.079600 milliseconds
Graph contains 4000 edges
0.095300 milliseconds
Graph contains 5000 edges
0.109400 milliseconds
Graph contains 6000 edges
0.114000 milliseconds
Graph contains 7000 edges
0.106300 milliseconds
Graph contains 8000 edges
0.106200 milliseconds
Graph contains 9000 edges
0.120500 milliseconds
Graph contains 10000 edges
0.125000 milliseconds

Graph contains 1000 vertixes
Graph contains 100000 edges
2.311000 milliseconds
Graph contains 200000 edges
3.921000 milliseconds
Graph contains 300000 edges
5.422000 milliseconds
Graph contains 400000 edges
6.591000 milliseconds
```

```
Graph contains 500000 edges
7.628000 milliseconds
Graph contains 600000 edges
8.375000 milliseconds
Graph contains 700000 edges
9.591000 milliseconds
Graph contains 800000 edges
9.671000 milliseconds
Graph contains 900000 edges
10.158000 milliseconds
Graph contains 1000000 edges
10.562000 milliseconds
```