Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЁТ**

По лабораторной работе №8

По курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Обходграфа в ширину»

**Выполнили**

**студенты группы 23ВВВ2:**

Пырков Д. А.

Родионов А. А.

**Приняли:**

Митрохин М. А.

Юрова О. В.

Пенза 2024

**Цель работы:** изучить и реализовать процедуру обхода графа в ширину для графов, представленных в матричном виде и в виде списков смежности, а также разработать алгоритм обхода графов в ширину с использованием очереди, построенной на основе структуры данных «список» и оценить время работы двух реализаций алгоритмов обхода в ширину.

**Задание 1:**

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Выведите матрицу на экран. Для сгенерированного графа осуществите процедуру обхода в ширину, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки С++.
2. (\*) Реализуйте процедуру обхода в ширину для графа, представленного списками смежности.

\* **Задание 2:**

1. Для матричной формы представления графов реализуйте алгоритм обхода в ширину с использованием очереди, построенной на основе структуры данных «список», самостоятельно созданной в лабораторной работе № 3.
2. Оцените время работы двух реализаций алгоритмов обхода в ширину (использующего стандартный класс **queue** и использующего очередь, реализованную самостоятельно) для графов разных порядков.

**Ход работы**

**Задание 1**

**1.** **Обход в ширину для матричной формы представления графов**

1. Функция breadth\_search инициирует процесс обхода:

void breadth\_search(int size, int\_fast8\_t\*\* graph\_pointer) {

// Создаем массив для отметки посещенных вершин

bool\* visited = new bool[size];

// Инициализируем все вершины как непосещенные

for (int i = 0; i < size; i++) {

visited[i] = false;

}

// Проходим по всем вершинам

for (int i = 0; i < size; i++) {

// Если вершина еще не посещена, запускаем BFS из нее

if (!visited[i]) {

BFS(i, size, graph\_pointer, visited);

}

}

delete[] visited;

}

1. Основной алгоритм обхода в ширину в функции BFS:

void BFS(int v, int size, int\_fast8\_t\*\* graph\_pointer, bool\* visited) {

// Создаем очередь для хранения вершин

std::queue<int> Q;

// Добавляем начальную вершину в очередь и помечаем её как посещенную

Q.push(v);

visited[v] = true;

// Пока очередь не пуста

while (!Q.empty()) {

// Извлекаем вершину из начала очереди

v = Q.front();

Q.pop();

// Выводим номер вершины

printf("%d ", v + 1);

// Просматриваем всех соседей текущей вершины

for (int i = 0; i < size; i++) {

// Если есть ребро и вершина еще не посещена

if (graph\_pointer[v][i] == 1 && !visited[i]) {

// Добавляем вершину в очередь и помечаем как посещенную

Q.push(i);

visited[i] = true;

}

}

}

}

Алгоритм работает следующим образом:

1. Создается массив visited для отслеживания посещенных вершин.
2. Для каждой непосещенной вершины запускается BFS:
   * Вершина добавляется в очередь
   * Помечается как посещенная
   * Пока очередь не пуста:
     + Извлекается вершина из начала очереди
     + Обрабатывается (выводится)
     + Все непосещенные соседи добавляются в очередь и помечаются как посещенные
3. Использование очереди гарантирует, что сначала будут обработаны все вершины на текущем уровне (расстоянии) от начальной вершины, прежде чем перейти к следующему уровню.
4. Процесс продолжается, пока не будут посещены все достижимые вершины.

Этот алгоритм обеспечивает обход графа "волнами", постепенно удаляясь от начальной вершины, что и является сутью поиска в ширину.

**Результат работы программы**

На рисунке 1 приведён результат работы программы №1.1.

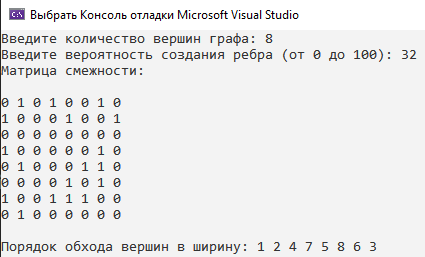


Рисунок 1 – результат работы программы №1.1

**2.** \***Обход в ширину для графа, представленного списками смежности**

1. Подготовка:

bool\* visited = (bool\*)calloc(size, sizeof(bool)); // Массив для отметки посещённых вершин

int\* queue = (int\*)malloc(size \* sizeof(int)); // Очередь для хранения вершин

int front = 0, rear = 0; // Указатели начала и конца очереди

1. Начало обхода:

queue[rear++] = start; // Помещаем начальную вершину в очередь

visited[start] = true; // Отмечаем её как посещённую

1. Основной цикл обхода:

while (front < rear) { // Пока очередь не пуста

int current = queue[front++]; // Извлекаем вершину из очереди

printf("%d ", current + 1); // Выводим её

Node\* temp = adj\_list[current]; // Получаем список смежных вершин

while (temp != NULL) { // Перебираем все смежные вершины

int adj\_vertex = temp->vertex;

if (!visited[adj\_vertex]) { // Если вершина ещё не посещена

queue[rear++] = adj\_vertex; // Добавляем её в очередь

visited[adj\_vertex] = true; // Отмечаем как посещённую

}

temp = temp->next;

}

}

1. Обработка несвязных компонент:

for (int i = 0; i < size; i++) {

if (!visited[i]) { // Если найдена непосещённая вершина

printf("Обход компоненты, начиная с вершины %d: ", i + 1);

BFS(adj\_list, size, i); // Запускаем BFS для новой компоненты

}

}

Алгоритм работает следующим образом:

1. Начинает с указанной вершины;
2. Посещает все смежные с ней вершины;
3. Затем посещает вершины, смежные с только что посещёнными;
4. Продолжает этот процесс, пока не обойдёт все достижимые вершины;
5. Если остались непосещённые вершины, запускает новый обход для каждой компоненты связности.

Это обеспечивает обход графа "по уровням", где сначала посещаются все вершины на расстоянии 1 от начальной, затем на расстоянии 2 и т.д.

**Результат работы программы**

На рисунке 2 приведён результат работы программы №1.2.

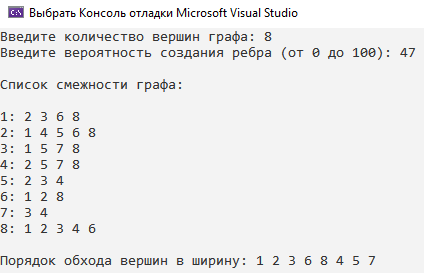


Рисунок 2 – результат работы программы №1.2

\***Задание 2**

**1. Обход в ширину с использованием очереди, построенной на основе структуры данных «список»**

1. Основные шаги алгоритма:

void BFS(int\_fast8\_t\*\* graph, int size, int start) {

// Создаём массив для отметки посещённых вершин

int\* visited = (int\*)calloc(size, sizeof(int));

// Создаём пустую очередь

Node\* queue = NULL;

// Отмечаем начальную вершину как посещённую

visited[start] = 1;

// Добавляем начальную вершину в очередь

enqueueBFS(&queue, start);

// Выводим начальную вершину

printf("%d ", start + 1);

1. Основной цикл обхода:

while (queue != NULL) {

// Извлекаем вершину из очереди

int current = dequeue(&queue);

// Просматриваем все смежные вершины

for (int i = 0; i < size; i++) {

// Если есть ребро и вершина не посещена

if (graph[current][i] == 1 && !visited[i]) {

visited[i] = 1; // Отмечаем как посещённую

enqueueBFS(&queue, i); // Добавляем в очередь

printf("%d ", i + 1); // Выводим номер вершины

}

}

}

1. Вспомогательные функции для работы с очередью:

* enqueueBFS: добавляет элемент в конец очереди
* dequeue: удаляет и возвращает первый элемент очереди

Алгоритм работает следующим образом:

1. Начинает с указанной вершины;
2. Посещает все смежные с ней вершины;
3. Затем посещает вершины, смежные с только что посещёнными;
4. Продолжает этот процесс, пока не обойдёт все достижимые вершины.

Это обеспечивает обход графа "волнами" - сначала посещаются все вершины на расстоянии 1, затем на расстоянии 2 и т.д.

**Результат работы программы**

На рисунке 3 приведён результат работы программы №2.1.

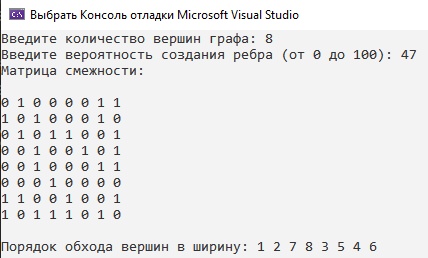


Рисунок 3 – результат работы программы №2.1

**2. Оценка времени работы двух реализаций алгоритмов обхода в ширину**

В программе время работы алгоритмов оценивается следующим образом:

Для каждой реализации BFS:

* Засекается начальное время с помощью clock() перед выполнением алгоритма
* После завершения алгоритма засекается конечное время
* Вычисляется разница между конечным и начальным временем
* Результат переводится в миллисекунды путём деления на CLOCKS\_PER\_SEC и умножения на 1000

Что касается производительности:

Стандартный класс queue обычно работает быстрее по нескольким причинам:

1. Оптимизация:

* std::queue оптимизирован разработчиками стандартной библиотеки
* Использует эффективные внутренние структуры данных
* Имеет оптимизированные реализации операций вставки и удаления

1. В ручной реализации:

* При добавлении элемента (manually\_enqueueBFS) каждый раз происходит проход до конца списка (O(n))
* Используется динамическое выделение памяти для каждого нового узла
* Менее эффективное управление памятью

1. Структура данных:

* std::queue обычно реализован на основе двусторонней очереди (deque)
* Ручная реализация использует односвязный список, что менее эффективно для операций добавления

Для улучшения производительности ручной реализации можно было бы:

* Хранить указатель на конец очереди
* Использовать кольцевой буфер
* Применить пред-выделение памяти

**Результат работы программы**

На рисунке 4 приведён результат работы программы №2.2.



Рисунок 4 – результат работы программы №2.2

**Выводы**

Был изучен и реализован алгоритм обхода графа в ширину над графами в матричном виде и в виде списков смежности, реализован алгоритм обхода в ширину с использованием очереди, построенной на основе структуры данных «список», а также было оценено время работы двух реализаций алгоритмов обхода в ширину.