

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Кафедра «Вычислительная техника»

ОТЧЁТ

По лабораторной работе №9

По курсу «Логика и основы алгоритмизации в инженерных задачах»

На тему «Поиск расстояний в графе»

Выполнили

студенты

группы

23ВВВ4:

Святов И.Ю.

Епинин Д.В.

Приняли:

Юрова О.В.

Деев М.В.

Пенза 2024

Общие сведения:

Поиск расстояний – довольно распространенная задача анализа графов.

Для поиска расстояний можно использовать процедуры обхода графа. Для этого при каждом переходе в новую вершину необходимо запоминать, сколько шагов до нее мы сделали. При этом вектор, который хранил информацию о посещении вершин становится вектором расстояний. Довольно просто модернизировать для поиска расстояний в графе алгоритм обхода в ширину, т.к. этот алгоритм проходит вершины по уровням удаленности, то для не ориентированного графа для вершин каждого следующего уровня глубины расстояние от исходной вершины увеличивается на 1. Удалённость в данном случае понимается как количество ребер, по которым необходимо прейти до достижения вершины.

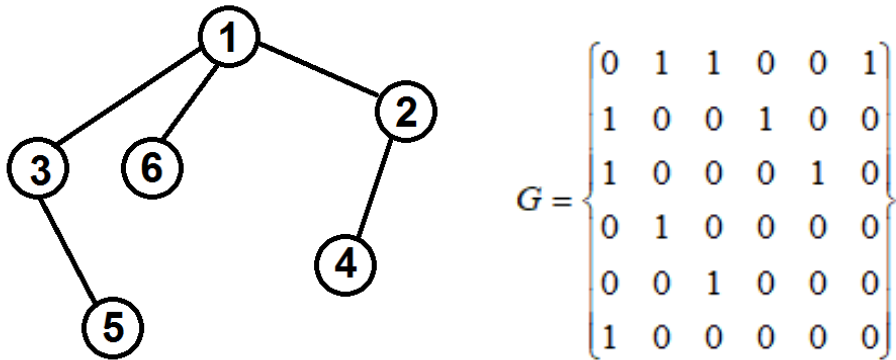


Рисунок 1 – Граф

Таким образом, можно предложить следующую реализацию алгоритма обхода в ширину.

Задание:

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа G . Выведите матрицу на экран.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));
    const int G = 5; // Размер графа
    int M[G][G];      // Матрица смежности

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0; // Нет петли
            }
        }
    }
}
```

```

        else {
            M[i][j] = 0; // Изначально нет ребер
        }
    }
}
// Генерация случайных рёбер для неориентированного графа
for (int i = 0; i < G; i++) {
    for (int j = i + 1; j < G; j++) {
        M[i][j] = M[j][i] = rand() % 2; // Случайное ребро между i и j
    }
}
// Вывод матрицы смежности на экран
printf("Матрица смежности:\n");
for (int i = 0; i < G; i++) {
    for (int j = 0; j < G; j++) {
        printf("%d ", M[i][j]);
    }
    printf("\n");
}
return 0;
}

```

2. Для сгенерированного графа осуществите процедуру поиска расстояний, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки C++.

Ответ:

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <queue>

//const int G = 5; // Размер графа
//int M[G][G]; // Матрица смежности

void bfs(int startVertex, int G, int** M) {
    std::queue<int> q; // Очередь для BFS
    bool* visited = (bool*)malloc(G * sizeof(bool));
    int* distance = (int*)malloc(G * sizeof(int));
    for (int i = 0; i < G; i++) {
        visited[i] = false;
    }
    // bool visited[G] = { false }; // Массив для отслеживания посещенных вершин
    //int distance[G]; // Массив для хранения расстояний
    for (int i = 0; i < G; ++i) {
        distance[i] = -1; // Изначально расстояние до всех вершин неопределено
    }

    visited[startVertex] = true;
    distance[startVertex] = 0;
    q.push(startVertex);

    while (!q.empty()) {
        int currentVertex = q.front();
        q.pop();
    }
}

```

```

        // Смотрим соседей текущей вершины
        for (int i = 0; i < G; i++) {
            if (M[currentVertex][i] == 1 && !visited[i]) {    // Если есть ребро и не
посещена
                visited[i] = true;
                distance[i] = distance[currentVertex] + 1;    // Увеличиваем расстояние
                q.push(i);    // Добавляем в очередь
            }
        }
    }

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины %d:\n", startVertex);
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    int G;
    int start;
    printf("Введите размер: ");
    scanf("%d", &G);
    int** M = (int**)malloc(G * sizeof(int*));
    for (int i = 0; i < G; i++) {
        M[i] = (int*)malloc(G * sizeof(int));
    }

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0;    // Нет петли
            }
            else {
                M[i][j] = 0;    // Изначально нет ребер
            }
        }
    }

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2;    // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Запуск BFS от вершины 0 (можно выбрать любую)

```

```

    bfs(0, G, M);
    for (int i = 0; i < G; i++) {
        free(M[i]);
    }
    free(M);
    return 0;
}

```

3. *Реализуйте процедуру поиска расстояний для графа, представленного списками смежности.

Ответ:

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <queue>

#define G 5 // Размер графа
#define MAX_QUEUE_SIZE 100

typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

typedef struct Graph {
    Node* adjLists[G];
} Graph;

// Создание графа
Graph* createGraph() {
    Graph* graph = (Graph*)malloc(sizeof(Graph)); // Явное приведение типа
    for (int i = 0; i < G; i++) {
        graph->adjLists[i] = NULL;
    }
    return graph;
}

// Добавление рёбер в граф
void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = (Node*)malloc(sizeof(Node)); // Явное приведение типа
    newNode->vertex = dest;
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Для неориентированного графа добавляем обратное направление
    newNode = (Node*)malloc(sizeof(Node)); // Явное приведение типа
    newNode->vertex = src;
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

// Освобождение памяти
void freeGraph(Graph* graph) {
    for (int i = 0; i < G; i++) {
        Node* temp = graph->adjLists[i];
        while (temp) {
            Node* next = temp->next;

```

```

        free(temp);
        temp = next;
    }
}
free(graph);
}

// Реализация BFS
void bfs(Graph* graph, int startVertex) {
    int queue[MAX_QUEUE_SIZE];    // Очередь
    int front = 0, rear = 0;      // Указатели на начало и конец очереди
    int visited[G] = { 0 };      // Массив для отслеживания посещенных вершин
    int distance[G];              // Массив для хранения расстояний

    // Инициализация расстояний
    for (int i = 0; i < G; ++i) {
        distance[i] = -1; // Изначально расстояние до всех вершин неопределено
    }

    // Начнем с начальной вершины
    visited[startVertex] = 1;
    distance[startVertex] = 0;
    queue[rear++] = startVertex; // Добавляем начальную вершину в очередь

    while (front < rear) {
        int currentVertex = queue[front++]; // Получаем текущую вершину из очереди

        // Смотрим соседей текущей вершины
        Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (!visited[adjVertex]) { // Если сосед не посещен
                visited[adjVertex] = 1;
                distance[adjVertex] = distance[currentVertex] + 1; // Увеличиваем
                расстояние
                queue[rear++] = adjVertex; // Добавляем в очередь
            }
            temp = temp->next;
        }
    }

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины %d:\n", startVertex);
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    // Создание графа
    Graph* graph = createGraph();

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            if (rand() % 2) { // Случайное ребро между i и j
                addEdge(graph, i, j);
            }
        }
    }
}

```

```

    }
}

// Вывод списков смежности на экран
printf("Списки смежности:\n");
for (int i = 0; i < G; i++) {
    printf("%d: ", i);
    Node* temp = graph->adjLists[i];
    while (temp) {
        printf("%d ", temp->vertex);
        temp = temp->next;
    }
    printf("\n");
}

// Запуск BFS от вершины 0 (можно выбрать любую)
bfs(graph, 0);

// Освобождение памяти
freeGraph(graph);

return 0;
}

```

Задание 2*

1. Реализуйте процедуру поиска расстояний на основе обхода в глубину.

Ответ:

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

const int G = 5; // Размер графа
int M[G][G];     // Матрица смежности
int visited[G];  // Массив для отслеживания посещенных вершин
int distance[G]; // Массив для хранения расстояний

// Функция для выполнения обхода в глубину
void dfs(int vertex, int dist) {
    visited[vertex] = 1;           // Отмечаем вершину как посещенную
    distance[vertex] = dist;       // Устанавливаем расстояние до вершины

    for (int i = 0; i < G; i++) {
        if (M[vertex][i] == 1 && !visited[i]) { // Проверяем наличие ребра и
            посещенность
                dfs(i, dist + 1);               // Рекурсивно вызываем dfs для соседней вершины
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
}

```

```

srand(time(NULL));

// Инициализация матрицы смежности
for (int i = 0; i < G; i++) {
    for (int j = 0; j < G; j++) {
        if (i == j) {
            M[i][j] = 0; // Нет петли
        }
        else {
            M[i][j] = 0; // Изначально нет ребер
        }
    }
}

// Генерация случайных рёбер для неориентированного графа
for (int i = 0; i < G; i++) {
    for (int j = i + 1; j < G; j++) {
        M[i][j] = M[j][i] = rand() % 2; // Случайное ребро между i и j
    }
}

// Вывод матрицы смежности на экран
printf("Матрица смежности:\n");
for (int i = 0; i < G; i++) {
    for (int j = 0; j < G; j++) {
        printf("%d ", M[i][j]);
    }
    printf("\n");
}

// Инициализация массивов
memset(visited, 0, sizeof(visited));
memset(distance, -1, sizeof(distance)); // Установка -1 для обозначения недоступных
вершин

// Запуск DFS от стартовой вершины (например, 0)
dfs(0, 0);

// Вывод расстояний от стартовой вершины
printf("Расстояния от вершины 0:\n");
for (int i = 0; i < G; i++) {
    if (distance[i] != -1) {
        printf("До вершины %d: %d\n", i, distance[i]);
    }
    else {
        printf("До вершины %d: недоступно\n", i);
    }
}

return 0;
}

```

2. Реализуйте процедуру поиска расстояний на основе обхода в глубину для графа, представленного списками смежности.

Ответ:


```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <vector>

const int G = 5; // Размер графа
std::vector<int> adj[G]; // Списки смежности
int visited[G]; // Массив для отслеживания посещенных вершин
int distance[G]; // Массив для хранения расстояний

// Функция для выполнения обхода в глубину
void dfs(int vertex, int dist) {
    visited[vertex] = 1; // Отмечаем вершину как посещенную
    distance[vertex] = dist; // Устанавливаем расстояние до вершины

    for (int i = 0; i < adj[vertex].size(); i++) {
        int neighbor = adj[vertex][i]; // Получаем соседа
        if (!visited[neighbor]) { // Проверяем посещенность
            dfs(neighbor, dist + 1); // Рекурсивно вызываем dfs для соседней вершины
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            if (rand() % 2) { // Случайное ребро между i и j
                adj[i].push_back(j);
                adj[j].push_back(i);
            }
        }
    }

    // Вывод матрицы смежности на экран
    printf("Списки смежности:\n");
    for (int i = 0; i < G; i++) {
        printf("%d: ", i);
        for (size_t j = 0; j < adj[i].size(); j++) {
            printf("%d ", adj[i][j]);
        }
        printf("\n");
    }

    // Инициализация массивов
    memset(visited, 0, sizeof(visited));
    memset(distance, -1, sizeof(distance)); // Установка -1 для обозначения
    недоступных вершин

    // Запуск DFS от стартовой вершины (например, 0)
    dfs(0, 0);
}

```

```

// Вывод расстояний от стартовой вершины
printf("Расстояния от вершины 0:\n");
for (int i = 0; i < G; i++) {
    if (distance[i] != -1) {
        printf("До вершины %d: %d\n", i, distance[i]);
    }
    else {
        printf("До вершины %d: недоступно\n", i);
    }
}

return 0;
}

```

3. Оцените время работы реализаций алгоритмов поиска расстояний на основе обхода в глубину и обхода в ширину для графов разных порядков.

Использует обход в ширину:

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <vector>
#include <ctime> // Для использования clock()
#include <time.h> // Для использования clock()

#define G 1000 // Размер графа
int M[G][G]; // Матрица смежности
bool visited[G]; // Массив для отслеживания посещенных вершин

void bfs(int start) {
    int q[G];
    int front = 0, rear = 0;
    visited[start] = true;
    q[rear++] = start;

    while (front != rear) {
        int current = q[front++];
        printf("%d ", current);

        for (int i = 0; i < G; i++) {
            if (M[current][i] == 1 && !visited[i]) {
                visited[i] = true;
                q[rear++] = i;
            }
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
}

```

```

srand(static_cast<unsigned int>(time(NULL)));

// Инициализация матрицы смежности
for (int i = 0; i < G; i++) {
    for (int j = 0; j < G; j++) {
        M[i][j] = 0; // Изначально нет рёбер
    }
}

// Генерация случайных рёбер для неориентированного графа
for (int i = 0; i < G; i++) {
    for (int j = i + 1; j < G; j++) {
        M[i][j] = M[j][i] = rand() % 2; // Случайное ребро между i и j
    }
}

// Вывод матрицы смежности на экран
printf("Матрица смежности:\n");
for (int i = 0; i < G; i++) {
    for (int j = 0; j < G; j++) {
        printf("%d ", M[i][j]);
    }
    printf("\n");
}

// Измерение времени выполнения обхода в ширину
clock_t start_time = clock(); // Запоминаем время начала
// Обход в ширину, начиная с вершины 0
printf("Обход в ширину: ");
bfs(0);
printf("\n");
clock_t end_time = clock(); // Запоминаем время окончания

// Вычисляем и выводим время выполнения
double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
printf("Время выполнения: %f секунд\n", time_taken);

return 0;
}

```

```
Консоль отладки Microsoft Visual Studio
17 518 519 522 523 525 526 529 532 535 536 540 542 543 545 546 552 553 557 559 561 564 566 568 573 576 577 579 583 584 5
89 590 591 594 596 599 600 602 606 611 612 614 616 617 618 625 627 628 629 631 632 636 638 639 640 641 644 645 646 647 6
48 649 650 652 653 654 657 658 661 662 664 665 666 667 668 672 674 676 678 679 681 683 684 687 688 689 690 691 693 694 6
95 696 697 699 700 701 702 705 706 707 708 709 713 718 719 722 724 725 726 729 734 735 738 739 740 741 742 744 745 746 7
49 751 752 753 754 756 757 759 760 763 765 774 775 778 781 782 783 785 788 789 790 791 793 794 796 797 799 801 803 804 8
05 808 812 813 814 815 816 818 822 824 825 828 832 833 834 839 843 846 847 848 849 851 855 858 859 860 861 862 863 864 8
66 868 872 873 875 879 881 882 883 884 889 897 902 904 906 909 911 913 914 916 920 923 927 928 932 934 935 936 940 942 9
43 946 948 949 950 951 952 953 956 958 962 964 966 967 968 969 970 971 973 976 977 979 982 983 984 985 987 988 989 990 9
93 994 996 997 5 10 12 16 20 25 30 31 34 35 38 52 55 57 58 61 62 63 67 70 76 79 83 84 85 86 87 90 92 93 96 97 103 105 10
8 110 113 124 130 134 135 136 140 147 149 152 157 159 163 167 175 176 177 181 184 185 191 193 196 199 209 212 219 220 22
2 228 232 240 241 242 245 255 256 258 260 272 274 276 290 292 295 296 299 304 310 311 316 318 321 322 325 333 335 338 34
1 352 354 356 358 362 363 366 370 378 380 382 383 390 392 394 399 400 405 406 410 413 423 428 430 437 440 444 451 452 45
4 455 458 466 471 478 481 486 490 491 493 494 496 502 506 520 527 528 530 533 537 539 541 547 548 555 560 562 563 567 57
0 575 581 582 587 592 603 604 607 613 615 620 624 633 634 635 651 656 659 663 669 670 671 673 675 677 685 703 711 712 71
4 715 716 717 720 721 723 728 730 736 737 748 761 762 766 768 770 771 772 773 776 779 780 795 800 806 811 820 823 826 82
9 831 835 837 842 844 871 874 878 880 886 888 891 893 903 908 915 919 922 929 930 933 939 941 944 947 955 957 959 965 97
2 975 978 986 992 998 4 6 29 45 49 64 81 88 94 106 111 137 139 144 164 165 170 187 189 192 195 202 205 208 211 214 223 2
50 263 264 277 279 285 287 298 319 320 324 332 342 345 361 364 369 388 395 409 424 426 431 438 447 468 480 483 485 492 4
95 499 503 524 538 544 554 565 569 572 585 588 593 595 598 622 626 643 660 680 686 710 727 732 743 755 758 784 802 807 8
09 830 838 840 845 854 865 867 869 876 885 892 894 895 896 898 901 905 907 910 917 931 954 963 974 981 991 995 14 22 66
91 98 100 154 169 178 203 206 229 231 246 251 262 266 271 286 291 334 340 343 351 371 375 376 384 403 415 417 418 448 48
4 497 513 531 558 571 574 580 586 597 601 610 623 637 642 682 692 698 747 764 767 769 792 798 810 817 841 850 856 857 87
0 887 899 900 912 921 926 937 938 960 980 27 32 41 42 44 71 80 116 117 133 148 188 294 302 309 313 355 377 404 416 504 5
34 578 608 619 630 733 786 787 821 836 853 877 890 999 68 72 254 357 605 609 621 731 827 852 918 924 925 961 283 387 419
420 434 521 549 550 556 750 777 819 9 23 221 551 655 945 186 704
Время выполнения: 0,108000 секунд
C:\Users\dimav\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe (процесс 14604) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно...
```

Использует обход в глубину:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <vector>
#include <ctime> // Для использования clock()
const int G = 1000; // Размер графа
std::vector<int> adj[G]; // Списки смежности
int visited[G]; // Массив для отслеживания посещенных вершин
int distance[G]; // Массив для хранения расстояний

// Функция для выполнения обхода в глубину
void dfs(int vertex, int dist) {
    visited[vertex] = 1; // Отмечаем вершину как посещенную
    distance[vertex] = dist; // Устанавливаем расстояние до вершины

    for (int i = 0; i < adj[vertex].size(); i++) {
        int neighbor = adj[vertex][i]; // Получаем соседа
        if (!visited[neighbor]) { // Проверяем посещенность
            dfs(neighbor, dist + 1); // Рекурсивно вызываем dfs для соседней
            // вершины
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(static_cast<unsigned int>(time(NULL))); // Инициализация генератора
    // случайных чисел
```

```

clock_t start = clock(); // Начало измерения времени

// Генерация случайных рёбер для неориентированного графа
for (int i = 0; i < G; i++) {
    for (int j = i + 1; j < G; j++) {
        if (rand() % 2) { // Случайное ребро между i и j
            adj[i].push_back(j);
            adj[j].push_back(i);
        }
    }
}

// Вывод списков смежности на экран
printf("Списки смежности:\n");
for (int i = 0; i < G; i++) {
    printf("%d: ", i);
    for (size_t j = 0; j < adj[i].size(); j++) {
        printf("%d ", adj[i][j]);
    }
    printf("\n");
}

// Инициализация массивов
memset(visited, 0, sizeof(visited));
memset(distance, -1, sizeof(distance)); // Установка -1 для обозначения
недоступных вершин

// Запуск DFS от стартовой вершины (например, 0)
dfs(0, 0);

// Вывод расстояний от стартовой вершины
printf("Расстояния от вершины 0:\n");
for (int i = 0; i < G; i++) {
    if (distance[i] != -1) {
        printf("До вершины %d: %d\n", i, distance[i]);
    }
    else {
        printf("До вершины %d: недоступно\n", i);
    }
}

clock_t end = clock(); // Конец измерения времени
double elapsed_time = double(end - start) / CLOCKS_PER_SEC; // Вычисление
прошедшего времени

printf("Время выполнения программы: %.6f секунд\n", elapsed_time);

return 0;
}

```

```
Консоль отладки Microsoft Visual Studio
До вершины 975: 976
До вершины 976: 977
До вершины 977: 973
До вершины 978: 979
До вершины 979: 980
До вершины 980: 978
До вершины 981: 982
До вершины 982: 981
До вершины 983: 983
До вершины 984: 986
До вершины 985: 984
До вершины 986: 987
До вершины 987: 985
До вершины 988: 989
До вершины 989: 988
До вершины 990: 990
До вершины 991: 992
До вершины 992: 996
До вершины 993: 991
До вершины 994: 997
До вершины 995: 993
До вершины 996: 994
До вершины 997: 995
До вершины 998: 999
До вершины 999: 998
Время выполнения программы: 91,032000 секунд
C:\Users\dimav\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe (процесс 16688) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно...
```

Задание: убрать лишние вектор visited. И второе, в списках смежности добавить динамическое выделение памяти.

Ответ 1:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <queue>

//const int G = 5; // Размер графа
//int M[G][G]; // Матрица смежности

void bfs(int startVertex, int G, int** M) {
    std::queue<int> q; // Очередь для BFS
    int* distance = (int*)malloc(G * sizeof(int));

    for (int i = 0; i < G; ++i) {
        distance[i] = -1; // Изначально расстояние до всех вершин неопределено
    }

    distance[startVertex] = 0;
    q.push(startVertex);

    while (!q.empty()) {
        int currentVertex = q.front();
        q.pop();
```

```

        // Смотрим соседей текущей вершины
        for (int i = 0; i < G; i++) {
            if (M[currentVertex][i] == 1 && distance[i] == -1) { // Если есть
ребро и не посещена
                distance[i] = distance[currentVertex] + 1; // Увеличиваем
расстояние
                q.push(i); // Добавляем в очередь
            }
        }
    }

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины %d:\n", startVertex);
    for (int i = 0; i < G; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }
    free(distance);
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    int G;
    int start;
    printf("Введите размер: ");
    scanf("%d", &G);
    int** M = (int**)malloc(G * sizeof(int*));
    for (int i = 0; i < G; i++) {
        M[i] = (int*)malloc(G * sizeof(int));
    }

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0; // Нет петли
            }
            else {
                M[i][j] = 0; // Изначально нет ребер
            }
        }
    }

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2; // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {

```

```

        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Запуск BFS от вершины 0 (можно выбрать любую)
    bfs(0, G, M);
    for (int i = 0; i < G; i++) {
        free(M[i]);
    }
    free(M);
    return 0;
}

```

Ответ 2:

```

#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
#include <queue>
#include <time.h>

typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

typedef struct Graph {
    int numVertices; // Количество вершин
    Node** adjLists; // Списки смежности
} Graph;

// Создание графа
Graph* createGraph(int vertices) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numVertices = vertices;
    graph->adjLists = (Node**)malloc(vertices * sizeof(Node*));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
    }
    return graph;
}

// Добавление рёбер в граф
void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = dest;
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    // Для неориентированного графа добавим обратное направление
    newNode = (Node*)malloc(sizeof(Node));
    newNode->vertex = src;
    newNode->next = graph->adjLists[dest];
}

```



```

    graph->adjLists[dest] = newNode;
}

// Освобождение памяти
void freeGraph(Graph* graph) {
    for (int i = 0; i < graph->numVertices; i++) {
        Node* temp = graph->adjLists[i];
        while (temp) {
            Node* next = temp->next;
            free(temp);
            temp = next;
        }
    }
    free(graph->adjLists);
    free(graph);
}

// Реализация BFS
void bfs(Graph* graph, int startVertex) {
    int* queue = (int*)malloc(graph->numVertices * sizeof(int));
    int front = 0, rear = 0;
    int* distance = (int*)malloc(graph->numVertices * sizeof(int));

    for (int i = 0; i < graph->numVertices; ++i) {
        distance[i] = -1; // Изначально расстояние до всех вершин неопределено
    }

    distance[startVertex] = 0;
    queue[rear++] = startVertex; // Добавляем начальную вершину в очередь

    while (front < rear) {
        int currentVertex = queue[front++]; // Получаем текущую вершину из очереди

        Node* temp = graph->adjLists[currentVertex];
        while (temp) {
            int adjVertex = temp->vertex;
            if (distance[adjVertex] == -1) { // Если сосед не посещен

                distance[adjVertex] = distance[currentVertex] + 1; // Увеличиваем
расстояние
                queue[rear++] = adjVertex; // Добавляем в очередь
            }
            temp = temp->next;
        }
    }

    // Вывод расстояний от стартовой вершины
    printf("Расстояния от вершины %d:\n", startVertex);
    for (int i = 0; i < graph->numVertices; i++) {
        if (distance[i] != -1) {
            printf("До вершины %d: %d\n", i, distance[i]);
        }
        else {
            printf("До вершины %d: недоступно\n", i);
        }
    }
}

```

```

        free(queue);
        free(distance);
    }

int main() {
    setlocale(LC_ALL, "Russian");

    int numVertices;
    printf("Введите количество вершин графа: ");
    scanf("%d", &numVertices);

    // Создание графа с динамически выделенной памятью
    Graph* graph = createGraph(numVertices);
    srand(time(NULL)); // Инициализация генератора случайных чисел

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < numVertices; i++) {
        for (int j = i + 1; j < numVertices; j++) {
            if (rand() % 2) { // Случайное решение о добавлении рёбер
                addEdge(graph, i, j);
            }
        }
    }

    // Вывод списков смежности на экран
    printf("Списки смежности:\n");
    for (int i = 0; i < numVertices; i++) {
        printf("%d: ", i);
        Node* temp = graph->adjLists[i];
        while (temp) {
            printf("%d ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }

    // Запуск BFS от вершины 0 (можно выбрать любую)
    bfs(graph, 0);

    // Освобождение памяти
    freeGraph(graph);

    return 0;
}

```

Вывод: в ходе выполнения лабораторной работы были выполнены работы с графами, научились генерировать матрицы смежности и осуществлять обход в ширину различными методами. Оценили время работы этих методов и сделали вывод о том, какой из них более эффективен для решения конкретных задач.