МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЕТ
по курсовой работе
по дисциплине «Алгоритмы и структуры данных»
Тема: «ГРАФЫ»

Студенты гр. 3312 ____________________ Поляков А.И Половникова А.С.

Преподаватель ______________________________ Колинько П.Г.

Санкт-Петербург
2024

Цель работы

Целью выполнения курсовой работы «Графы» является исследование алгоритмов на графах.

Задание (Вариант 9)

Отыскание клики наибольшей мощности в неориентированном графе

Математическая формулировка задачи в терминах теории множеств

Дан неориентированный граф G=(V,E), где V — множество вершин, а E — множество рёбер. Граф задан с помощью матрицы смежности A, где aij∈ E для i,j∈ V. Необходимо найти максимально большое подмножество вершин C⊆V такое, что для любых двух вершин u,v∈C существует ребро e∈E соединяющее u и v. Формально: ∀u,v∈C:(u,v)∈E.

Обоснование выбора матрицы смежности для представления графа

Использование матрицы смежности как структуры данных для хранения графа удобно и эффективно, особенно в задаче нахождения максимальной клики. Основные причины этого выбора следующие:

1. **Простота и наглядность**
    - Матрица смежности является понятным способом описания связей между вершинами.
    - В ней сразу видно, есть ли ребро между двумя вершинами (через значения элементов матрицы).
2. **Удобство поиска смежных вершин**
    - Матрица позволяет легко определить, с какими вершинами связана каждая отдельная вершина графа.
    - Это важно при поиске клики, где необходимо проверять наличие рёбер между всеми вершинами множества.
3. **Быстрая проверка наличия ребра**
    - Проверить, соединены ли две вершины, можно за O(1), обращаясь к соответствующему элементу матрицы.
    - Это сокращает время на выполнение базовых операций при обработке графа.
4. **Поддержка алгоритмов**
    - Алгоритмы для поиска клик и других задач на графах часто используют матрицу смежности благодаря её удобству для выполнения массовых операций, например, умножения матриц.

    - Это делает реализацию более универсальной и эффективной.
5. **Проверка свойств клики**
    - Матрица позволяет легко проверять, образуют ли вершины подмножества клику.
    - Все необходимые данные для проверки связности вершин находятся прямо в матрице.

Таким образом, использование матрицы смежности обусловлено её простотой, скоростью доступа к информации о рёбрах и возможностью эффективной работы алгоритмов, что делает её оптимальным выбором для задачи поиска максимальной клики.

Описание алгоритма и оценка его временной сложности

Алгоритм:

1. **Инициализация:**
    - Создается граф, представленный матрицей смежности.
    - Вводится количество вершин графа.
    - Генерируется случайный граф с ребрами на основе матрицы смежности.
2. **DFS для поиска клик:**
    - Реализован рекурсивный метод BruteForce, который использует алгоритм обхода в глубину (DFS) для перебора возможных клик.
    - Используется массив ‘U’ для отслеживания посещенных вершин.
    - При обнаружении клики, информация о ней выводится на экран.
3. **Вывод результата:**
    - После завершения поиска выводится информация о найденной клике максимальной мощности.

Временная сложность:

1. Создание графа

В методах CreateRandGraph() и CreateGraph() происходит заполнение матрицы смежности для графа. Давайте оценим временную сложность этих операций:

- В методе CreateRandGraph():
    - Цикл с переменными i и j проходит по всем возможным парам вершин, то есть два вложенных цикла, каждый из которых имеет длину N, где N — количество вершин.
    - Время выполнения для каждого элемента в матрице — O(1), так как присваивание значения в матрице и проверка условия генерации случайного числа — операции с постоянной сложностью.

    - В общем, сложность этого метода будет O(N2), так как мы проходим по всем парам вершин.
- В методе CreateGraph():
    - В этом методе используется заранее заданная матрица 5×5. Процесс её копирования — это фиксированное количество операций.
    - Время выполнения будет O(1), так как мы просто копируем небольшую матрицу размером 5×5 в матрицу графа.

Время выполнения:

- CreateRandGraph(): O(N2)
- CreateGraph(): O(1)

2. DFS (BruteForce)

Метод BruteForce() использует рекурсию для поиска клики максимального размера. Важный момент здесь — рекурсивный вызов и количество итераций для каждой вершины.

- Для каждого уровня рекурсии мы выбираем вершину из множества вершин V, и для каждой вершины мы проверяем её смежность с уже выбранными вершинами.
- В худшем случае, при поиске самой большой клики, будет N уровней рекурсии, так как максимальный размер клики равен N.
- На каждом уровне рекурсии, для каждой вершины, нам нужно проверить её смежность с предыдущими вершинами. Это можно сделать за O(N), так как для каждой вершины мы смотрим на её связи с уже выбранными вершинами.

Таким образом, на каждом уровне рекурсии выполняется O(N) операций, и поскольку в худшем случае рекурсия может быть глубиной N, сложность будет примерно O(N^2).

Время выполнения: O(N^2) для одного уровня рекурсии, но так как максимальная глубина рекурсии может быть N, общая сложность будет O(N^3).

3. Итоговая сложность

Теперь, объединим все этапы:

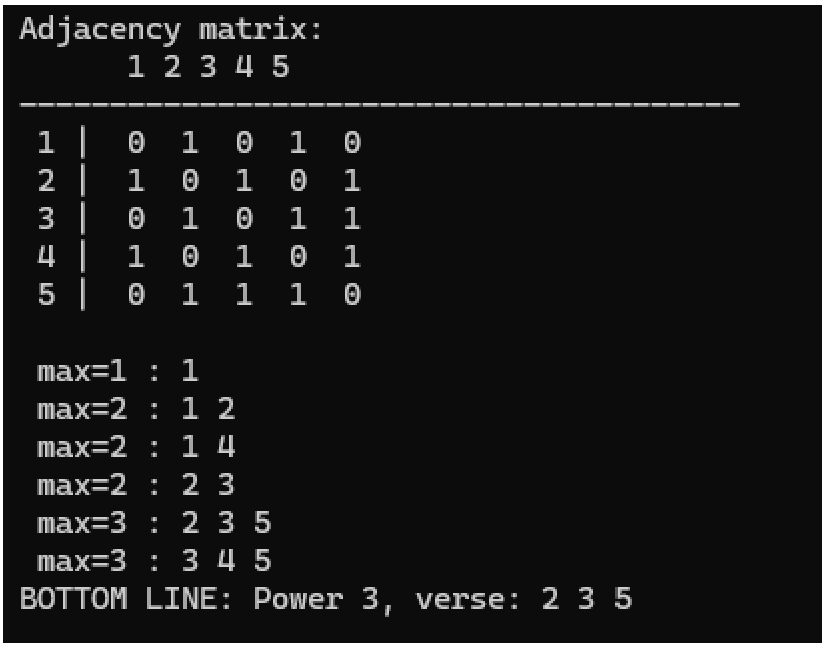
- **Создание графа** — максимальная сложность для этого этапа будет O(N^2) (для метода CreateRandGraph()).
- **DFS (BruteForce)** — сложность этого метода будет O(N^3).

Итоговая временная сложность: Наиболее затратной операцией является рекурсивный метод BruteForce(), который выполняется за O(N^3), и это определяет итоговую временную сложность всей программы.

Таким образом, программу можно оценить как квадратичную относительно количества вершин графа.

Результаты работы программы

Результаты контрольных тестов представлены на рисунках 1, 2, 3.



```
Adjacency matrix:
      1 2 3 4 5
----------------------------------------
 1 |  0  1  0  1  0
 2 |  1  0  1  0  1
 3 |  0  1  0  1  1
 4 |  1  0  1  0  1
 5 |  0  1  1  1  0

 max=1 : 1
 max=2 : 1 2
 max=2 : 1 4
 max=2 : 2 3
 max=3 : 2 3 5
 max=3 : 3 4 5
BOTTOM LINE: Power 3, verse: 2 3 5
```

Рисунок 1. Результат с заготовленными данными

```
Enter the number of vertices (max 20): 8

Adjacency matrix:
      1 2 3 4 5 6 7 8
----------------------------------------
 1 |  0  1  0  1  1  1  1  1
 2 |  1  0  1  1  1  1  1  0
 3 |  0  1  0  1  0  1  1  0
 4 |  1  1  1  0  1  1  1  1
 5 |  1  1  0  1  0  0  1  1
 6 |  1  1  1  1  0  0  1  1
 7 |  1  1  1  1  1  1  0  1
 8 |  1  0  0  1  1  1  1  0

 max=1 : 1
 max=2 : 1 2
 max=3 : 1 2 4
 max=4 : 1 2 4 5
 max=5 : 1 2 4 5 7
 max=5 : 1 2 4 6 7
 max=5 : 1 4 5 7 8
 max=5 : 1 4 6 7 8
 max=5 : 2 3 4 6 7
BOTTOM LINE: Power 5, verse: 1 2 4 5 7
```

Рисунок 2. Результат 1 с автогенерацией

```
Enter the number of vertices (max 20): 18

Adjacency matrix:
      1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
----------------------------------------
 1 |  0  1  0  1  1  1  1  1  1  1  1  1  1  0  1  0  1  1
 2 |  1  0  0  1  1  1  1  0  1  1  1  1  1  1  1  0  1  1
 3 |  0  0  0  1  1  1  0  1  1  1  1  1  0  1  1  1  1  0
 4 |  1  1  1  0  1  1  0  1  1  1  1  1  1  1  1  1  0  1
 5 |  1  1  1  1  0  1  1  1  0  1  0  1  1  1  0  1  1  1
 6 |  1  1  1  1  1  0  1  1  1  1  1  1  1  1  1  1  1  0
 7 |  1  1  0  0  1  1  0  1  1  1  1  1  1  1  0  1  1  1
 8 |  1  0  1  1  1  1  1  0  1  1  1  1  1  1  1  0  1  1
 9 |  1  1  1  1  0  1  1  1  0  1  1  1  1  1  1  0  1  0
 10|  1  1  1  1  1  1  1  1  1  0  1  1  1  0  0  1  1  1
 11|  1  1  1  1  0  1  1  1  1  1  0  0  0  1  1  1  1  1
 12|  1  1  1  1  1  1  1  1  1  1  0  0  1  1  0  1  1  1
 13|  1  1  0  1  1  1  1  1  1  1  0  1  0  1  1  1  0  1
 14|  0  1  1  1  1  1  1  1  1  0  1  1  1  0  1  1  1  1
 15|  1  1  1  1  0  1  0  1  1  0  1  0  1  1  0  1  1  1
 16|  0  0  1  1  1  1  1  0  0  1  1  1  1  1  1  0  0  0
 17|  1  1  1  0  1  1  1  1  1  1  1  1  0  1  1  0  0  1
 18|  1  1  0  1  1  0  1  1  0  1  1  1  1  1  1  0  1  0

 max=1 : 1
 max=2 : 1 2
 max=3 : 1 2 4
 max=4 : 1 2 4 5
 max=5 : 1 2 4 5 6
 max=6 : 1 2 4 5 6 10
 max=7 : 1 2 4 5 6 10 12
 max=8 : 1 2 4 5 6 10 12 13
 max=8 : 1 2 4 5 10 12 13 18
 max=8 : 1 2 4 6 9 10 12 13
 max=8 : 1 2 5 6 7 10 12 13
 max=8 : 1 2 5 6 7 10 12 17
 max=8 : 1 2 5 7 10 12 13 18
 max=8 : 1 2 5 7 10 12 17 18
 max=8 : 1 2 6 7 9 10 11 17
 max=8 : 1 2 6 7 9 10 12 13
 max=8 : 1 2 6 7 9 10 12 17
 max=8 : 1 4 5 6 8 10 12 13
 max=8 : 1 4 5 8 10 12 13 18
 max=8 : 1 4 6 8 9 10 12 13
 max=8 : 1 5 6 7 8 10 12 13
 max=8 : 1 5 6 7 8 10 12 17
 max=8 : 1 5 7 8 10 12 13 18
 max=8 : 1 5 7 8 10 12 17 18
 max=8 : 1 6 7 8 9 10 11 17
 max=8 : 1 6 7 8 9 10 12 13
 max=8 : 1 6 7 8 9 10 12 17
 max=8 : 3 4 6 8 9 11 14 15
 max=8 : 3 6 8 9 11 14 15 17
BOTTOM LINE: Power 8, verse: 1 2 4 5 6 10 12 13
```

Рисунок 3. Результат 2 с автогенерацией

Выводы

В рамках курсовой работы "Графы" были исследованы алгоритмы, направленные на решение задачи поиска клик наибольшей мощности в неориентированных графах. Основным алгоритмом, примененным в работе, является алгоритм "перебор с возвратом". Этот метод позволяет систематически проверить все возможные комбинации вершин с целью обнаружения клик наибольшей мощности.

Список использованных источников

Колинько П. Г. Пользовательские структуры данных: Методические указания по дисциплине “Алгоритмы и структуры данных, часть 1”. - СПб.: СПбГЭТУ “ЛЭТИ”, 2024. - 64 с. (вып.2309).

Приложение. Исходные тексты программ.

Основная программа

```
#include <iostream>
#include "Graph.h"

int main()
{
    char choose;
    int N;
    Graph A(0);

    std::cout << "#=================#" << std::endl;
    std::cout << "1 - autogeneration" << std::endl;
    std::cout << "2 - ready asset" << std::endl;
    std::cout << "#=================#" << std::endl;

    std::cin >> choose;

    switch (choose)
    {
    case '1':
        system("cls");
        std::cout << "Enter the number of vertices (max 20): ";
        std::cin >> N;

        if (N > 0 && N <= 20)
        {
            A = Graph(N);
            A.CreateRandGraph();
```

```
            A.PrintGraph();
            A.BruteForce(1, A);

            std::cout << "\nBOTTOM LINE: Power " << A.maxv << ", verse: ";
            for (auto i = 0; i < A.maxv; ++i)
                std::cout << A.ans[i] << " ";
        }
        else
            std::cerr << "Error: Number of vertices must be between 1 and 20." <<
std::endl;

        break;

    case '2':
        system("cls");
        A = Graph(5);
        A.CreateGraph();
        A.PrintGraph();
        A.BruteForce(1, A);

        std::cout << "\nBOTTOM LINE: Power " << A.maxv << ", verse: ";
        for (auto i = 0; i < A.maxv; ++i)
            std::cout << A.ans[i] << " ";

        break;

    default:
        std::cerr << "Error: Incorrect selection." << std::endl;
        break;
    }

    std::cout << std::endl;
    return 0;
}
```

“Graph.h”

```
#pragma once
#include <iostream>
#include <vector>
#include <map>
#include <set>


class Graph
{
private:
    int N;
    std::vector<std::vector<int>> matrix;
    std::vector<int> K;
    std::vector<int> U;
public:
    int maxv = 0;

    std::vector<int> ans;

    Graph(int N);

    Graph();

    Graph(const Graph&) = delete;

    Graph(Graph&&) = delete;
```

```
        void addEdge(int v1, int v2);

    void PrintGraph();

    void BruteForce(int currentDepth, Graph& parentGraph);

    void CreateRandGraph();

    void CreateGraph();

    Graph& operator=(const Graph& other);

    ~Graph();
};
```

“Graph.cpp”

```
#include "Graph.h"


Graph::Graph(int N) : N(N), matrix(N, std::vector<int>(N, 0)), U(N, 1), K(N), ans(N)
{}

Graph::Graph() : N(1), matrix(1, std::vector<int>(1, 0)), U(1, 1), K(1), ans(1) {}

void Graph::addEdge(int v1, int v2)
{
    matrix[v1][v2] = 1;
    matrix[v2][v1] = 1;
}

void Graph::PrintGraph()
{
    if (N < 21)
    {
        std::cout << "\nAdjacency matrix:";
        std::cout << "\n       ";

        for (int i = 0; i < N; ++i)
            std::cout << i + 1 << " ";

        std::cout << "\n----------------------------------------";

        for (int i = 0; i < N; ++i)
        {
            if (i + 1 >= 10)
                std::cout << "\n " << i + 1 << "| ";
            else
                std::cout << "\n " << i + 1 << " | ";
            for (int j = 0; j < N; ++j)
                std::cout << " " << matrix[i][j] << " ";
        }
        std::cout << "\n";
    }
}

void Graph::BruteForce(int currentDepth, Graph& parentGraph)
{
    int vertex, startIndex, neighbor;
    if (currentDepth == 1)
        startIndex = 0;
    else
        startIndex = K[currentDepth - 2] + 1;
```

```
    for (vertex = startIndex; vertex < N; vertex++)
        if (U[vertex])
        {
            K[currentDepth - 1] = vertex;
            neighbor = 0;
            while ((neighbor < currentDepth) && (K[neighbor] < N) && (vertex < N) &&
matrix[K[neighbor]][vertex])
                neighbor++;

            if (neighbor + 1 == currentDepth)
            {
                if (currentDepth > maxv)
                {
                    maxv = currentDepth;
                    for (int i = 0; i < currentDepth; ++i)
                        ans[i] = K[i] + 1;
                }
                if (currentDepth == maxv)
                {
                    std::cout << '\n' << " max=" << maxv << " : ";

                    for (int i = 0; i < maxv; ++i)
                        std::cout << (K[i] + 1) << " ";
                }

                U[vertex] = 0;
                parentGraph.BruteForce(currentDepth + 1, *this);
                U[vertex] = 1;
            }
        }
}

void Graph::CreateRandGraph()
{
    for (auto i = 0; i < N; ++i)
    {
        U[i] = 1;
        for (auto j = i; j < N; ++j)
        {
            if (j == i)
                matrix[i][j] = 0;
            else
                matrix[i][j] = matrix[j][i] = rand() % 15 > 2;
        }
    }
}

void Graph::CreateGraph()
{
    int tempmatrix[5][5] = {
        {0, 1, 0, 1, 0},
        {1, 0, 1, 0, 1},
        {0, 1, 0, 1, 1},
        {1, 0, 1, 0, 1},
        {0, 1, 1, 1, 0}
    };

    for (auto i = 0; i < N; ++i)
    {
        U[i] = 1;
        for (auto j = 0; j < N; ++j)
        {
            matrix[i][j] = tempmatrix[i][j];
        }
    }
```

```
}

Graph& Graph::operator=(const Graph& other)
{
    if (this == &other)
        return *this;

    N = other.N;
    maxv = other.maxv;

    matrix = other.matrix;
    K = other.K;
    U = other.U;
    ans = other.ans;

    return *this;
}

Graph::~Graph() {}
```