МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» им.В.И.УЛЬЯНОВА (ЛЕНИНА)

Кафедра вычислительной техники

Отчет по лабораторной работе № 4
по дисциплине «Алгоритмы и структуры данных, часть 1»
Тема: «Графы»

Студенты гр. 9306 ЕвдокимовО.В.Кныш С.А. Павельев М.С.

Преподаватель Манерагена Валенс

Санкт-Петербург
2020

Содержание

Цель

Исследование алгоритмов для работы с графами.

Задание

Обнаружение всех элементарных циклов ориентированного графа.

Математическая формулировка задачи в терминах теории множеств.

Если элементарный путь это — последовательность вершин, в которой все вершины различны, то элементарный цикл это элементарный путь, в котором все вершины, кроме начальной и конечной, различны. Следовательно нам требуется найти все такие пути.

Давайте попробуем оценить количество таких циклов в орграфе. N(n) – здесь и далее количество вершин графа.

1)Количество циклов длинной 1(петли): n

2)Количество циклов длинной n: n*(n-1)*(n-2)...2*1 т.е. n!

3)Количество циклов длинной n-1: n*(n-1)*(n-2)...(n-n+2)

4) и так далее

Таким образом, грубо оценивая, мы получаем O(n!), что очень большое число. Было принято решение отсеивать циклы идентичные при цикличном сравнении, так как фактически они являются одинаковыми циклами с разной вершиной начала. Тогда у каждого цикла длинной n будет по n идентичных ему циклов, т.е. всего таких n^2, аналогично для циклов длинной n-1,n-2…. Тогда общее количество, грубо оценивая, (n!/n^2), что все ещё я валяется очень большим числом.

Выбор и обоснование способа представления данных.

В качестве структуры данных для хранения графа, была выбрана матрица(массив) смежности, она очень удобна для хранения ориентированных графов, позволяет удобно генерировать тесты.

Циклы хранится в виде списка списков(векторов в терминах C++). Хранение каждого цикла в виде списка нужно для экономии ресурсов(которых и так мало), поскольку цикл может быть как длинной 1 так и длинной n, практически равно вероятно. Все циклы хранятся в виде списка, так как заранее мы не можем знать сколько их у нас будет, причем как мы уже знаем это число может быть достаточно большим.

Описание алгоритма и оценка его временной сложности.

Фактически был реализован алгоритм перебора с возвратом, мы проходим по всем возможным путям, возвращаясь в места, где было сделано ветвление.

Сначала посмотрим на сложность алгоритма циклического сравнения. И проверки цикла на наличие в списке циклов.

При сравнении двух циклов, для каждого элемента первого проходим по всем элементам втором, соответственно сложность O(n^2).

При проверки цикла на наличие, для каждого цикла из списка мы проводим проверку с сложность n^2, следовательно в общем случае сложность O(n!).

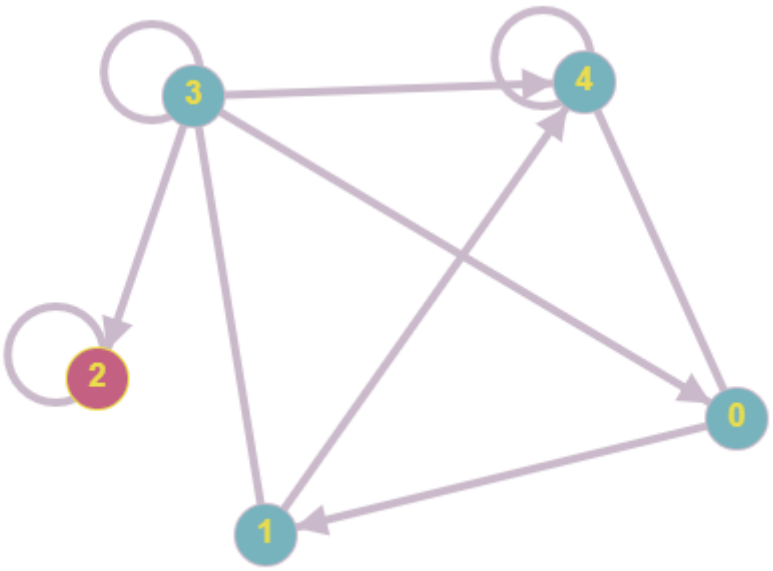
Теперь рассмотрим сложность алгоритма перебора с возвратом.

В общем случае мы проходим по всем возможным путям, что является n*(n-1)*(n-2)...(n-k+1)(k - длинна пути), что можно сравнить с O(n!). И для малого количество путей выполняем операцию проверки на наличие сложностью O(n!). Тогда сложность всего алгоритма O((n!)^2).

Набор тестов и результаты проверки алгоритма на ЭВМ.

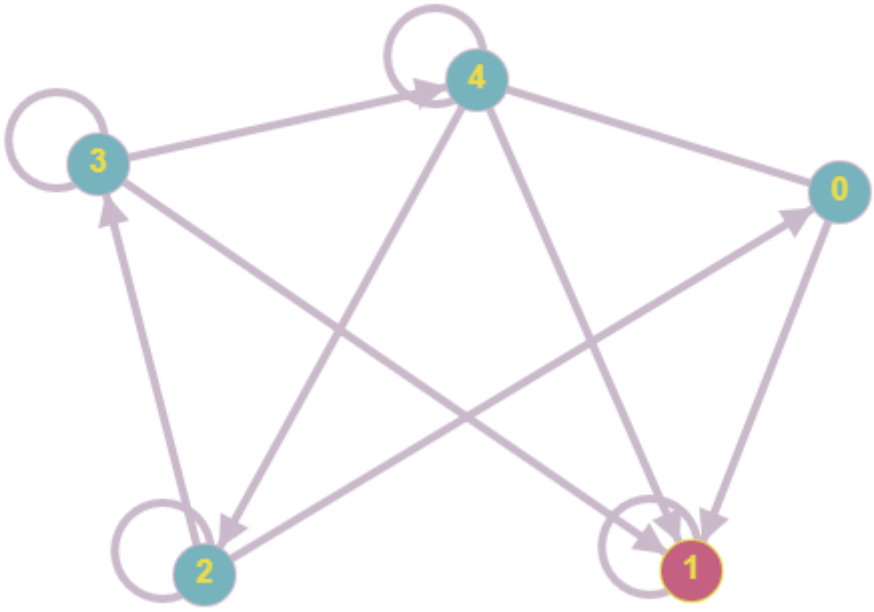
```
0 1 0 0 1
0 0 0 1 1
0 0 1 0 0
1 1 1 1 1
1 0 0 0 1

Cycles:
0-1-3-0
0-1-3-4-0
0-1-4-0
0-4-0
1-3-1
2-2
3-3
4-4
```



```
0 1 0 0 1
0 1 0 0 0
1 0 1 1 0
0 1 0 1 1
1 1 1 0 1

Cycles:
0-4-0
0-4-2-0
1-1
2-2
2-3-4-2
3-3
4-4
```



Выводы

Задача поиска всех элементарных циклов, является крайне сложной. Скорее всего кем-то уже был разработан алгоритм эффективнее нашего, но не был найден нами, однако остаётся проблема хранения всех циклов, из-за их огромного количества. Однако и нами был проведен анализ задачи, подсчёт количества циклов и реализован и оттестирован алгоритм с маленькими оптимизациями.

Код программы

```
#include<iostream>
#include<time.h>
#include<stack>
#include<vector>

const int maxSize = 10;

class Graph
{
private:
	int m_size;
	int arr[maxSize][maxSize];
	std::vector<std::vector<int>> cycles;

	bool cyclic_cmpr_of_vectors(std::vector<int> a, std::vector<int>
b)
	{
		bool res = true;
		size_t n = a.size();
		int i;
		for (i = 0; i < n; ++i)
		{
			if (a[0] == b[i])
			{
				for (int j = 0; j < n; ++j)
				{
					if (a[j] != b[(i + j) % int(n)])
```

```
                                    res = false;
                        }
                        break;
                    }
                }
                if (i == n)
                    res = false;
                return res;
        }

        bool cycle_already_added(std::vector<int> that)
        {
                bool flag = false;
                for (int i = 0; i < cycles.size(); ++i)
                    if (cycles[i].size() == that.size() &&
cyclic_cmpr_of_vectors(cycles[i], that))
                        flag = true;
                return flag;
        }

public:
        Graph(int, int);
        void print_matrix();
        void DFS_cycle(int v, int par, int color[], int parent[],int);
        int getSize();
        void print_cycles();
};

Graph::Graph(int size,int max)
{
        for (int i = 0; i < maxSize; ++i)
        {
                for (int j = 0; j < maxSize; ++j)
                {
                    arr[i][j] = 0;
                }
        }

        m_size = size + (rand() % (max - size));
        for (int i = 0; i < m_size; ++i)
        {
                for (int j = 0; j < m_size; ++j)
                {
                    if (rand() % 2 == 0)
                    {
                        arr[i][j] = 1;
                    }
                }
        }
}

void Graph::print_matrix()
{
        for (int i = 0; i < m_size; ++i)
        {
                for (int j = 0; j < m_size; ++j)
                {
                    std::cout << arr[i][j] << " ";
                }
                std::cout << std::endl;
        }
}

void Graph::DFS_cycle(int v, int prnt, int color[], int parent[],int
start)
{
```

```
	if (color[v] == 1)
	{
		if (v == start)
		{
			std::vector<int> temp;
			int cur = prnt;
			temp.push_back(v);
			while (cur != v)
			{
				temp.push_back(cur);
				cur = parent[cur];
			}
			if (!cycle_already_added(temp))
			{
				cycles.push_back(temp);
			}
		}
		return;
	}

	parent[v] = prnt;

	color[v] = 1;

	for (int i = 0; i < m_size; ++i)
	{ // проверяем для нее все смежные вершины
		if (arr[v][i] == 1)
			DFS_cycle(i, v, color, parent,start);
	}
	color[v] = 0;
}

int Graph::getSize()
{
	return m_size;
}

void Graph::print_cycles()
{
	for (int i = 0; i < cycles.size(); ++i)
		if (cycles[i][0] != -1)
		{
			std::cout << cycles[i][0] << "-";
			for (int j = (int)cycles[i].size() - 1; j > 0; --j)
			{
				std::cout << cycles[i][j] << "-";
			}
			std::cout << cycles[i][0] << "\n";
		}
}

int main()
{
	srand(time(0));

	Graph G(5,6);

	G.print_matrix();

	for (int i = 0; i < G.getSize(); ++i)
	{
		int* color = new int[G.getSize() * sizeof(int)];
		int* parent = new int[G.getSize() * sizeof(int)];

		for (size_t j = 0; j < G.getSize(); ++j)
		{
```

```
                color[j] = -1;
                parent[j] = -1;
            }

            G.DFS_cycle(i, -1, color, parent,i);

            delete[] color;
            delete[] parent;
        }
        std::cout << "\nCycles:\n";

        G.print_cycles();

        return 0;
}
```