Московский авиационный институт
(Национальный исследовательский университет)
факультет “Информационные технологии и прикладная математика”
кафедра “Математическая кибернетика”

КУРСОВАЯ РАБОТА

по курсу «Дискретная математика»
2-й семестр

Тема: Теория графов, алгебраические структуры, теория алгоритмов. Построение базы циклов (сокращенной системы циклов) графа на основе поиска в глубину.

Студент: Сикорский Александр Александрович

Группа: М8О-108Б

Руководитель: Н.С. Алексеев

Оценка: ______________

Дата: ______________

Часть III. Программная реализация алгоритмов

Алгоритм построение базы циклов (сокращенной системы циклов) графа должен работать на основе поиска в глубину. Поиском в глубину называют такой рекурсивный способ обхода графа, при котором от стартовой вершины мы пытаемся пройти как можно глубже в граф. Если при переборе ребер, исходящих от вершины, встречается ребро, ведущее в еще не исследованную вершину, то алгоритм запускается от этого ребра. После мы возвращаемся и продолжаем перебирать другие ребра. Сам по себепоиск в глубину работает за линейное время.

Для того, чтобы найти у графа базис циклов, нужно комбинировать поиск в глубину и его модифицированную версию. Вообще говоря, внести изменения нужно в оба алгоритма, но второй из них будет использоваться для поиска цикла в графе.

Обратимся к остовному дереву. Мы знаем, что остовным деревом (каркасом) графа называется такой ацикличный и связный подграф нашего графа, в который входят все вершины графа. Вспомним, что при добавлении любого ребра, не входящего в каркас, мы получим граф с единственным циклом. Этот цикл и будет входить в базис циклов графа.

Перебрав все не используемые в каркасе вершины, мы получим все циклы, которые и будут составлять базис циклов графа.

Перейдем к реализации:

Как найти остовное дерево? Обыкновенный поиск в глубину (depth first search, далее DFS) не справится, его нужно немного видоизменить. По определению в каркас входят все вершины графа. Так что когда рассматриваемое в DFS ребро ведет в еще не использованную вершину, добавим её в остовное дерево:

```
if (f != prev){
   tree[prev].push_back(f);
   tree[f].push_back(prev);
}
```

У нас есть массив (вектор) **used**, в котором содержится информация о том, была ли посещена вершина с индексом *i*. Таким образом в DFS мы полностью обойдем граф и построим остовное дерево.

Теперь нужно найти ребра, которые не входят в остовное дерево. Это те ребра, при добавлении которых мы будем получать циклы, образующие базис циклов графа. Эти ребра представляют собой дополнение множества ребер каркаса до множества ребер всего графа. Находятся они с помощью прохода по всем ребрам и определения тех, которые не входят в каркас.

После того, как мы получили список ребер, не входящих в остовное дерево, мы можем по одному добавлять их к нему и запускать для каждого получившегося графа DFS с поиском цикла. Он заполняет вектор вершин, входящих в цикл и завершается после того, как цикл замкнулся. Теперь нужно развернуть вектор вершин, входящих в цикл и отсортировать этот вектор. Цикл найден. Такие шаги повторяются для всех ребер, не входящих в каркас. В итоге мы находим все циклы, образующие базис.

Описание программы и инструкции:

Программа считывает матрицу смежности из файла, где первым числом задается количество вершин графа, а дальше следует сама матрица.

```
5
0 1 0 0 0
1 0 1 1 1
0 1 0 1 0
0 1 1 0 1
0 1 0 1 0
```

После считывания программа преобразовывает матрицу смежности к списку смежности. Это нужно для удобства и оптимизации. Подумаем, какую пространственную сложность составит хранение матрицы смежности для графа. Так как в случае матрицы память выделяется на все, даже ненулевые элементы, то хранение такой матрицы будет занимать порядка *O(n * n)* памяти, где *n* - количество вершин в графе. Понятно, что далеко не в каждом случае все вершины соединены со всеми. И при большом количестве вершин, например, [illegible] , это будет занимать уже очень много места.

Список смежности решает эту проблему. Это вектор векторов целых чисел, который представляет собой список *ребер* графа. То есть мы не храним несуществующие ребра, что мы вынуждены делать в случае матрицы смежности. Это особенно эффективно в том случае, когда матрица графа сильно разрежена. Вывод программы адаптирован для использования с системой ГРАФОИД. Программа выводит надпись Text: , что позволяет ГРАФОИДу понять, что дальше будет происходить вывод ответа. Далее в отдельных строках выводятся циклы, входящие в базис циклов графа. При этом используется 0-индексация (нумерация вершин начинается с нуля). Вывод осуществляется в тот же файл, в котором хранится матрица смежности. В данном случае файл называется *matrix.txt.*

Оценим вычислительную сложность алгоритма:

Поиск в глубину работает за линейное время, поиск в глубину с нахождением графа работает тоже за линейное время. Самое нагруженное место в алгоритме это поиск ребер, не входящих в каркас, их перебор и запуск для каждого из них поиска в глубину с поиском цикла. В этом месте имеем три вложенных цикла, в самой глубине и работает DFS для поиска цикла. Получаем сложность *O*( ).

Пример прикладной задачи:

Алгоритм поиска базиса циклов графа может быть использован для расчета параметров электрических цепей по законам Кирхгофа. При составлении уравнений токов и напряжений нужно находить базис циклов графа. В этом месте и поможет запрограммированный алгоритм.

Заключение

В ходе выполнения курсовой работы были решены 12 задач по различным разделам курса "Дискретная математика".

Кроме того был изучен вопрос о различных методах поиска путей и маршрутов в графах. Была написана и отлажена программа, реализующая Построение базы циклов (сокращенной системы циклов) графа на основе поиска в глубину. Программа написана на языке программирования С++. Программа обеспечивает связь по установленному формату с системой ГРАФОИД, разработанной на кафедре 805, что дает возможность обеспечить графический интерфейс при ее использовании. Эта программа является основным результатом курсового проектирования.

Приложение 1

Текст программы

```
#include <bits/stdc++.h>
using namespace std;

vector<char> cl;
vector<int> p;
int cycle_st, cycle_end;


bool dfs_cl (int v, vector<vector<int>>& g) {
   cl[v] = 1;

   for (size_t i=0; i<g[v].size(); ++i) {
      int to = g[v][i];

      if (cl[to] == 0 && g[v][i] != p[v]) {
         p[to] = v;
         if (dfs_cl (to, g))  return true;
      }

      else if (cl[to] == 1 && g[v][i] != p[v]) {
            cycle_end = v;
            cycle_st = to;
            return true;
      }
   }
   cl[v] = 2;
   return false;
}

void dfs(int f, int prev, const vector<vector<int>>& graph, vector<int>& used, vector<vector<int>>&
```

```
tree, int& cnt) {

    if (used[f] != -1)
        return;

    used[f] = cnt;

    if (f != prev){
        tree[prev].push_back(f);
        tree[f].push_back(prev);
    }

    prev = f;
    cnt++;

    for( int to: graph[f] ) {
        dfs(to, prev, graph, used, tree,  cnt);
    }

}

int main(int argc, char *argv[]) {
    ifstream fin(argv[1]);
    int size;
    fin >> size;
    int matrix[size][size];

    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++) {
            fin >> matrix[i][j];
        }
    }

    vector<vector<int>> g(size);
    fin.close();

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (matrix[i][j] == 1)
                g[i].push_back(j);
        }
    }

    vector<vector<int>> tree(size);
    vector<int> used(size);
    used.assign(size, -1);
```

```
int f = 0;
int cnt = 0;
int prev = 0;

dfs(f, prev, g, used, tree, cnt);
vector<vector<int>> unused(size);
vector<vector<int>> cycles;

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

    vector<int> vertexes_a(size);
    vertexes_a.assign(size, -1);

    vector<int> vertexes_b(size);
    vertexes_b.assign(size, -1);

    vector<int> difference;

    for (int x : g[i]) {
        vertexes_a[x] = 1;
    }

    for (int x : tree[i]) {
        vertexes_b[x] = 1;
    }

    for (int j = 0; j < size; j++) {
        if (vertexes_a[j] != vertexes_b[j])
            difference.push_back(j);
    }

    for (int x: difference) {
        vector<vector<int>> temp_g = tree;
        temp_g[i].push_back(x);
        temp_g[x].push_back(i);
        p.assign (size, -1);
        cl.assign (size, 0);
        cycle_st = -1;

        for (int a = 0; a < size; ++i) {
            if (dfs_cl (a, temp_g))
                break;
        }

        vector<int> cycle;
```

```
            cycle.push_back (cycle_st);

            for (int v=cycle_end; v!=cycle_st; v=p[v])
                cycle.push_back (v);

            cycle.push_back (cycle_st);
            reverse (cycle.begin(), cycle.end());
            sort (cycle.begin(), cycle.end() - 1);
            cycles.push_back(cycle);
        }
    }

    sort(cycles.begin(), cycles.end());
    ofstream fout;
    fout.open(argv[1]);
    fout.clear();

    fout << size;
    fout << '\n';

    for(int i = 0; i < size; i++) {
        for(int j = 0; j < size; j++) {
            fout << matrix[i][j] << ' ';
        }
        fout << '\n';
    }

    fout << '\n';
    fout << "Text:";
    fout << '\n';

    for (int i = 0; i < cycles.size(); i++) {
        if (cycles[i+1] == cycles[i])
            continue;
        fout << "cycle: \n";
        for (int j = 0; j < cycles[i].size() - 1; j++)
            fout << cycles[i][j] << ' ';

        fout << '\n';
    }
    fout.close();
}
```

Приложение 2

Тестовые примеры

5

0 1 0 0 0

1 0 1 1 1

0 1 0 1 0

0 1 1 0 1

0 1 0 1 0

Text:

cycle:

1 2 3

cycle:

1 2 3 4

6

0 1 1 0 0 0

1 0 1 0 0 0

1 1 0 0 1 0

0 0 0 0 1 1

0 0 1 1 0 1

0 0 0 1 1 0

Text:

cycle:

0 1 2

cycle:

3 4 5

9

0 1 0 0 0 1 0 0 0

1 0 1 0 1 0 0 0 0

0 1 0 1 0 0 0 0 0

0 0 1 0 1 0 0 0 1

0 1 0 1 0 1 0 1 0

1 0 0 0 1 0 1 0 0

0 0 0 0 0 1 0 1 0

0 0 0 0 1 0 1 0 1

0 0 0 1 0 0 0 1 0

Text:

cycle:

0 1 2 3 4 5

cycle:

1 2 3 4

cycle:

3 4 5 6 7 8

cycle:

4 5 6 7