Глава 1

Графы и их инварианты

1.1 Определения

Определение 1 *Граф, или неориентированный граф G — это упорядоченная пара $G = (V, E)$, где V — это не пустое множество вершин или узлов, а E — множество пар (в случае неориентированного графа — неупорядоченных) вершин, называемых ребрами.*

Определение 2 *Ориентированный граф (сокращенно орграф) G — это упорядоченная пара $G = (V, A)$, где V — не пустое множество вершин или узлов, и A — множество (упорядоченных) пар различных вершин, называемых дугами или ориентированными ребрами.*

Определения пути, цикла, цепи и связанные с ними:

Определение 3 *Пусть G — неориентированный граф. Путём в G называется такая конечная или бесконечная последовательность ребер и вершин*

$$S = (..., a_0, E_0, a_1, E_1, ..., E_{n-1}, a_n, ...),$$

что каждые два соседних ребра E_{i-1} и E_i имеют общую вершину a_i. Таким образом, можно написать

$$..., E_0 = (a_0, a_1), E_1 = (a_1, a_2), ..., E_n = (a_n, a_{n+1}), ...$$

Отметим, что одно и тоже ребро может встречаться в пути несколько раз. Если нет ребер, предшествующих E_0, то a_0 называется начальной вершиной, а если нет ребер, следующих за $E_{(n-1)}$, то a_n называется конечной

вершиной. Любая вершина, принадлежащая двум соседним рёбрам, называется внутренней. Так как рёбра и вершины в пути могут повторяться, внутренняя вершина может оказаться начальной или конечной вершиной. Если начальная и конечная вершины совпадают, путь называется циклическим. Путь называется цепью, а циклический путь — циклом, если каждое его ребро встречается не более одного раза (вершины могут повторяться). Не циклическая цепь называется простой цепью, если в ней никакая вершина не повторяется. Цикл с концом a_0 называется простым циклом, если a_0 не является в нём промежуточной вершиной и никакие другие вершины не повторяются.

Определение 4 *Связный граф — граф, между любой парой вершин которого существует как минимум один путь.*

Определение 5 *Компонента связности графа G — максимальный связный подграф графа G.*

Матрица смежности — это один из способов представления графа:

Определение 6 *Матрицей смежности $A = ||\alpha_{i,j}||$ графа $G = (V, E)$ называется матрица $A_{[V \times V]}$, в которой $\alpha_{i,j}$ — количество рёбер, соединяющих вершины v_i и v_j, причём при $i = j$ каждую петлю учитываем дважды, если граф не является ориентированным, и один раз, если граф ориентирован.*

Определение 7 *Кликой неориентированного графа называется подмножество его вершин, любые две из которых соединены ребром.*

Максимальная клика — это клика, которая не может быть расширена путём включения дополнительных смежных вершин, то есть нет клики большего размера, включающей все вершины данной клики.

Определение 8 *Если v_1, v_2 — вершины, а $e = (v_1, v_2)$ — соединяющее их ребро, то говорят, что вершины v_1, v_2 инцидентные ребру e.*

Определение 9 *Степень вершины в теории графов — количество рёбер графа G, инцидентных вершине x. Обозначается $deg(x)$.*

Определение 10 *Граф называется однородным (регулярным), если степени всех его вершин равны. Если степени вершин однородного графа равны k, то граф называют k-однородным.*

Определение 11 *Изоморфизмом графов $G = (V_G, E_G)$ и $H = (V_H, E_H)$ называется биекция между множествами вершин графов $f\colon\ V_G \to V_H$ такая, что любые две вершины u и v графа G смежны тогда и только тогда, когда вершины $f(u)$ и $f(v)$ смежны в графе H.*

Определение 12 *Дерево — связный граф, не содержащий циклов.*

Определение 13 *Корень дерева — выбранная вершина дерева; в орграфе — вершина с нулевой степенью захода.*

1.2 Примеры инвариантов

Определение 14 *Инвариант графа — некоторое обычно числовое значение или упорядоченный набор значений, одинаковый для изоморфных графов.*

- ◇ Диаметр графа $diam(G)$ — длина кратчайшего пути (расстояние) между парой наиболее удаленных вершин.
- ◇ Индекс Винера — величина $w = \sum_{\forall i,j} d(v_i, v_j)$, где $d(v_i, v_j)$ — минимальное расстояние между вершинами v_i и v_j.
- ◇ Индекс Рандича — величина $r = \sum_{\forall i,j} \frac{1}{\sqrt{d(v_i)d(v_j)}}$.
- ◇ Минимальное число вершин, которое необходимо удалить для получения несвязного графа.
- ◇ Минимальное число ребер, которое необходимо удалить для получения несвязного графа.
- ◇ Обхват графа — число ребер в составе минимального цикла.
- ◇ Определитель матрицы смежности.

- ◇ Плотность графа $\varphi(G)$ — число вершин максимальной по включению клики.

- ◇ Упорядоченный по возрастанию или убыванию вектор степеней вершин $s(G) = (d(v_1), d(v_2), \ldots, d(v_n))$ — при использовании переборных алгоритмов определения изоморфизма графов в качестве предположительно-изоморфных пар вершин выбираются вершины с совпадающими степенями, что способствует снижению трудоемкость перебора. Использование данного инварианта для k-однородных графов не приводит к снижению вычислительной сложность перебора, так как степени всех вершин подобного графа совпадают: $s(G) = (k, k, \ldots, k)$.

- ◇ Упорядоченный по возрастанию или убыванию вектор собственных чисел матрицы смежности графа (спектр графа). Сама по себе матрица смежности не является инвариантом, так как при смене нумерации вершин она претерпевает перестановку строк и столбцов.

- ◇ Число вершин $n(G) = |A|$ и число дуг/ребер $m(G) = |V|$.

- ◇ Число компонент связности графа $\kappa(G)$.

- ◇ Характеристический многочлен матрицы смежности.

- ◇ Хроматическое число $\chi(G)$ — минимальное число цветов, в которые можно раскрасить вершины графа G так, чтобы концы любого ребра имели разные цвета.

Глава 2

Полугруппы и полурешетки

Определение 15 *Бинарная операция $*$ на множестве S — это функция*

$$f : S^2 \to S.$$

Значение бинарной операции для $x, y \in S$ обозначают $x * y = f(x, y)$.

Определение 16 *Пусть $*$ — бинарная операция определённая на множестве S, а $x, y, z \in S$ — произвольные элементы множества S. Тогда если выполняется равенство*

$$x * y = y * x,$$

то операция $*$ называется коммутативной. Если выполняется

$$x * (y * z) = (x * y) * z,$$

то операция $*$ называется ассоциативной. Если выполняется

$$x * x = x,$$

то операция $*$ называется идемпотентной.

Определение 17 *Полугруппа — множество с заданной на нём ассоциативной бинарной операцией $(S, *)$.*

Определение 18 *Полурешетка — это полугруппа, бинарная операция которой коммутативна и идемпотентна.*

Для полурешетки можно определить частичный порядок

$$x \leq y \Leftrightarrow x * y = x.$$

2.1 Таблица Кэли

Таблица Кэли — таблица, которая описывает структуру конечных алгебраических систем путём расположения результатов операции в таблице, напоминающей таблицу умножения. Названа в честь английского математика Артура Кэли. Таблица имеет важное значение в дискретной математике, в частности, в теории групп. Таблица позволяет выяснить некоторые свойства группы, например, является ли группа абелевой, найти центр группы и обратные элементы элементов группы.

В высшей алгебре таблицы Кэли могут также использоваться для определения бинарных операций в полях, кольцах и других алгебраических структурах.

Простой пример таблицы Кэли для группы $1, -1$ с обычным умножением:

$\times$	1	-1
1	1	-1
-1	-1	1

2.2 Граф полурешетки

Каждой полурешетке можно сопоставить ориентированный граф. Этот граф можно построить следующим образом.

Пусть полурештка состоит из элементов $0, 1, \ldots, n-1$ и задана таблицей Кэли.

Шаг 1. Находим корень дерева $root$:

$$root = 0 \cdot 1 \ldots \cdot (n-1).$$

Шаг 2. Находим смежные с $root$ вершины. Для этого перебираем все элементы x полурешётки и отбираем те из них, которые удовлетворяют условиям: $x \neq root$, $x \cdot root = root$, не существует элемента z, такого, что $z \neq x$, $z \neq root$, $z \cdot root = root$, $z \cdot x = z$. Это и будут смежные с $root$ вершины.

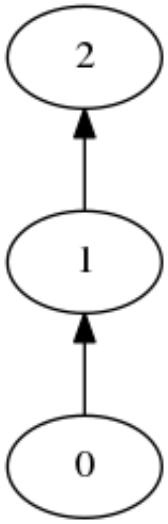
Шаг 3. Аналогично находим смежные вершины с найденными и т.д. пока не исчерпаются все вершины.

2.3 Примеры

Рассмотрим примеры полурешеток с такой таблицей Кэли:

×	0	1	2
0	0	0	0
1	0	1	1
2	0	1	2

Граф, который будет ей будет соответствовать:



Глава 3

Полный исходный код

3.1 digraph.h

```
 1 #ifndef GRAPH_H
 2 #define GRAPH_H
 3
 4 #include <bits/stdc++.h>
 5 using namespace std;
 6
 7 class digraph {
 8   int n = 0;
 9   vector<vector<int>> adj_list;
10   void resize(int new_n);
11   void load(string file_name);
12 public:
13   digraph(string fn);
14   digraph(int an);
15   int count_vertices();
16   bool is_edge(int a, int b);
17   void add_edge(int a, int b);
18   vector<pair<int, int>> edges();
19   void print();
20   int degree(int v);
```

```
21   vector<int> output_vertices(int k);
22 };
23
24 #endif /* end of include guard: GRAPH_H */
```

3.2 digraph.cpp

```
 1 #include <bits/stdc++.h>
 2 #include "digraph.h"
 3 using namespace std;
 4
 5 void digraph::resize(int new_n) {
 6   n = new_n;
 7   adj_list.resize(n);
 8   for (auto v : adj_list) {
 9     v.clear();
10   }
11 }
12
13 void digraph::load(string file_name) {
14   ifstream fin(file_name);
15   string line;
16   getline(fin, line);
17   istringstream iss(line);
18   iss >> n;
19   resize(n);
20   while (!fin.eof()) {
21     getline(fin, line);
22     if (line.empty()) continue;
23     istringstream iss(line);
24     char c;
25     iss >> c;
26     int v, u;
```

```
27      if (c == 'l') {
28        iss >> v;
29        while (iss >> u) add_edge(v, u);
30      } else if (c == 'p') {
31        iss >> v;
32        while (iss >> u) {
33          add_edge(v, u);
34          v = u;
35        }
36      }
37    }
38  }
39
40  digraph::digraph(string fn) {
41    load(fn);
42  }
43
44  digraph::digraph(int an) : n(an) {
45    adj_list.resize(n);
46  }
47
48  int digraph::count_vertices() {
49    return n;
50  }
51
52  bool digraph::is_edge(int a, int b) {
53    assert(a < n);
54    for (auto v : adj_list[a]) {
55      if (v == b) return true;
56    }
57    return false;
58  }
59
```

```
60 void digraph::add_edge(int a, int b) {
61   assert(a < n && b < n);
62   if (is_edge(a, b)) return;
63   adj_list[a].push_back(b);
64 }
65
66 vector<pair<int, int>> digraph::edges() {
67   vector<pair<int, int>> es;
68   for (int v = 0; v < n; v++) {
69     for (int u : adj_list[v]) {
70       es.push_back(make_pair(v, u));
71     }
72   }
73   return es;
74 }
75
76 void digraph::print() {
77   cout << "Count of vertices: " << n << '\n';
78   cout << "Degree of vertices:\n";
79   for (int v = 0; v < n; v++) {
80     cout << "\tdeg(" << v << ") = " << degree(v) << endl;
81   }
82   cout << "Edges:\n";
83   for (int v = 0; v < n; v++) {
84     for (int u : adj_list[v]) {
85       cout << '\t' << v << " -> " << u << '\n';
86     }
87   }
88 }
89
90 int digraph::degree(int v) {
91   return adj_list[v].size();
92 }
```

```
93
94 vector<int> digraph::output_vertices(int v) {
95   return adj_list[v];
96 }
```

3.3 graph_iso.cpp

```
 1 #include <bits/stdc++.h>
 2 #include "digraph.h"
 3 using namespace std;
 4
 5 class DigraphIso {
 6   digraph g1, g2;
 7   set<int> vertices;
 8   int n = -1; // количество вершин в графах g1, g2
 9   int g1_k = -1; // подмножество вершин g1 {0, 1, ..., g1_k}
10   set<int> g2_s; // Подмножество вершин графа g2, которое
11   // соотвествует множеству вершин грфафа g1 {0, 1, ..., g1_k}
12   vector<int> f; // биекция: i (вершина g1) -> f[i] (вершина g2)
13   int ans = -1;
14   bool match(int v);
15   void go();
16 public:
17   DigraphIso(digraph ag1, digraph ag2);
18   bool is_iso();
19   vector<int> biection();
20 };
21
22 DigraphIso::DigraphIso(digraph ag1, digraph ag2)
23   : g1(ag1), g2(ag2) {}
24
25 bool DigraphIso::match(int v) {
26   // Подмножество g1 {0, 1, ..., g1_k - 1} изоморфно
```

```
27   // подножеству графа g2 g2_s,
28   // которое равно {f[0], f[1], ..., f[g1_k - 1]}
29   // На данном шаге нужно проверить может ли подмножество g1
30   // {0, 1, ..., g1_k - 1, g1_k} быть изоморфно
31   // подмножеству g2 {f[0], f[1], ..., f[g1_k - 1], v}
32   for (int u = 0; u < g1_k; u++) {
33     if (g1.is_edge(u, g1_k) && !g2.is_edge(f[u], v)) {
34       // cout << 1 << ": " << u << " " << g1_k << " "
35       //      << f[u] << " " << v << endl;
36       return false;
37     }
38     if (g1.is_edge(g1_k, u) && !g2.is_edge(v, f[u])) {
39       // cout << 2 << endl;
40       return false;
41     }
42   }
43   return true;
44 }
45
46 void DigraphIso::go() {
47   // cout << "go begin\n";
48   // cout << "g2_s: ";
49   // for (int i : g2_s) cout << i << " ";
50   // cout << endl;
51   if (g2_s.size() == n) {
52     ans = 1;
53     return;
54   }
55   g1_k += 1;
56   set<int> t;
57   set_difference(
58     vertices.begin(), vertices.end(),
59     g2_s.begin(), g2_s.end(),
```

```
60      inserter(t, t.end())
61    );
62    // cout << "t: ";
63    // for (int i : t) cout << i << " ";
64    // cout << endl;
65    for (int v : t) {
66      if (ans > 0) return;
67      if (match(v)) {
68        // cout << g1_k << " match " << v << endl;
69        f[g1_k] = v;
70        g2_s.insert(v);
71        go();
72        g2_s.erase(v);
73      } else {
74        // cout << g1_k << " don't match " << v << endl;
75      }
76    }
77    g1_k -= 1;
78  }
79
80  bool DigraphIso::is_iso() {
81    if (ans >= 0) return ans;
82    if (g1.count_vertices() != g2.count_vertices()) {
83      return false;
84    }
85    n = g1.count_vertices();
86    f.resize(n);
87    for (int i=0; i<n; i++) vertices.insert(i);
88    go();
89    cout << "ans " << ans << endl;
90    if (ans < 0) ans = 0;
91    return ans;
92  }
```

```
 93
 94 vector<int> DigraphIso::biection() {
 95   return f;
 96 }
 97
 98 int main(int argc, char **argv) {
 99   if (argc < 3) {
100     cout << "few arguments" << endl;
101     return 1;
102   }
103   digraph g1(argv[1]), g2(argv[2]);
104   cout << "# 1.digraph:\n";
105   g1.print();
106   cout << "# 2.digraph:\n";
107   g2.print();
108   DigraphIso iso(g1, g2);
109   cout << "Is isomorph?: " << iso.is_iso() << endl;
110   if (iso.is_iso()) {
111     vector<int> f = iso.biection();
112     cout << "biection:\n";
113     for (int i=0; i<g1.count_vertices(); i++) {
114       cout << '\t' << i << " -> " << f[i] << '\n';
115     }
116   }
117 }
```