1 Оглавление

ВВЕДЕНИЕ

Тема моей работы: «применение алгоритмов теории графов при проверке изоморфизма полурешеток». Теория графов и теория полугрупп являются активно развивающимися областями. В 1736 Леонад Эйлер формулирует знаменитую задачу о кёнигсбергских мостах. Этот момент можно считать рождением теории графов. Теория полугрупп более молодая область, чем теория графов. Первые работы в этой области появились только в 20-х и 30-х годах 20 века[1][9]. Обе теории богаты как результатами так и вопросами. Одна из проблем является общей для этих теорий. Это проблема изоморфизма. В общем случае в обоих теориях остается нерешенной проблемой.

Актуальность мой работы может обосновать огромной областью приложений теории графов и теории полугрупп. Вот несколько областей, в которых активно используется теория графов:

1. Компьютерная химия или хемоинформатика[2].
2. В коммуникационных и транспортных системах. В том числе, для маршрутизации данных в Интернете.
3. Экономика[3].
4. Логистика.
5. Схемотехника[4].
6. Информатика и программирование.

Теория полугрупп, в свою очередь, применяется в следующих областях[9]:

1. Дифференциальная геометрия;
2. функциональный анализ;
3. теория графов;
4. теория алгоритмов;
5. абстрактная теория автоматов и др.

Граф сам по себе достаточно простой объект. Неформально его можно определить как точки соединенные линиями со стрелками или без них. Но с помощью графов можно описывать очень сложные совокупности объектов и взаимосвязи между ними. Например, граф путей сложных химических реакций, сеть дорог, печатные платы и микросхемы. Графы с одной стороны хорошо изученный объект, для которого существуем много самых разных теорем,

алгоритмов и инвариантов, но с другой графы имеют множество не решенных проблем. Из известных это задача коммивояжера, задача о клике, изоморфизм графов.

Во многих областях требуется выяснить являются ли графы «похожими». Это так называемая проблема изоморфизма графов. Все существующие алгоритмы проверки графов на изоморфизм работают за экспоненциальное время и неизвестно существует ли алгоритм, который будет иметь полиномиальную сложность и решать данную задачу. Чаще всего графы являются не изоморфными. В этом случае находят некоторую характеристику графа, которая должна совпадать у изоморфных графов и если она не совпадает, то заключают, что графы не изоморфны. Эти характеристики обычно находить гораздо проще, чем проверять графы на изоморфизм напрямую.

Для полугрупп и полурешеток также есть понятие изоморфизма. Проверка алгебраических систем на изоморфизм еще более нетривиальная задача. В этой работе рассмотрены полугруппы, являющиеся полурешетками, для проверки изоморфизма которых можно применить известные алгоритмы проверки изоморфизма графов. Описано, как для таких полугрупп можно найти соответствующий им граф. Этот граф может оказаться деревом, и в этом случае для проверки изоморфизма полугрупп можно применить известные алгоритмы проверки изоморфизма деревьев. В работе сформулирован и доказан критерий того, в каком случае граф полурешетки является деревом. Далее обосновывается выбор алгоритма проверки изоморфизма деревьев, описан этот алгоритм, представлена программа, реализующая его. Для того чтобы применить выбранный алгоритм для проверки изоморфизма полурешеток, необходимо сначала полурешетке сопоставить дерево. Для этого также разработан и реализован необходимый алгоритм. Созданная в итоге программа для двух полурешеток, заданных таблицами Кэли, работает следующим образом: она выводит структуру соответствующих полурешеткам деревьев, каноническое имя полученных деревьев, проверяет изоморфизм деревьев, а значит, и полурешеток. При этом выбор и реализация алгоритмов являются эффективными, программа в течение нескольких секунд определяет изоморфизм полурешеток с трехзначным числом элементов.

Данный алгоритм может быть применен и для произвольных полурешеток, но в этом случае результат не всегда верный: для изоморфных полурешеток ответ будет верный, но для неизоморфных может быть дан ошибочный результат. В работе рассмотрено, какое кодовое слово выдается произвольной полурешетке, показано, что это кодовое слово может служить инвариантом для проверки изоморфизма такой полурешетки.

Еще один из видов графов, для которых существует алгоритм (отличающийся от алгоритмов полного перебора) – планарные графы. Поэтому в работе решен вопрос о том, является ли граф произвольной полурешетки планарным.

После этого рассматриваются произвольные полурешетки и описывается алгоритм проверки изоморфизма таких полурешеток. Для этого описываются различные инварианты теории графов, которые можно успешно применить для полурешеток, рассмотрен вопрос о полноте представленной системы инвариантов.

Созданная в итоге программа для двух произвольных полурешеток, заданных таблицами Кэли, дает информацию о графах (их инварианты) и определяет, изоморфны ли они. В случае изоморфизма выдается биективное отображение элементов этих полурешеток.

По данной теме уже были написаны две статьи [10].

1 ГРАФЫ И АЛГОРИТМЫ НА ГРАФАХ

1.1 Основные понятия теории графов

Во введении уже было приведено неформальное определение графа. Определим граф формально. А также введем несколько других определений из теории графов, которые пригодятся в дальнейшем.

Граф G — это упорядоченная пара (V, E), где V — это не пустое множество вершин, а E — множество пар вершин (подмножество множества V×V), называемых ребрами. Граф может быть ориентированным или неориентированным.

Ориентированный граф (орграф) — это граф, у которого ребра имеют направления от одной вершины к другой. Ребра орграфа называют дугами.

Далее в основном будем работать только с орграфами.

Пусть в графе есть дуга e = (u, v). Дугу e будем называть инцидентной вершинам u и v, а вершины u и v смежными. Вершину u назовем началом дуги e: u=beg(e), а вершину v — концом дуги e: v=end(e). Также вершину u будем называть непосредственно предшествующей вершине v, а вершину v непосредственно следующей за вершиной u. Степень полузахода вершины v — это количество вершин, которые непосредственно предшествуют вершине v, а степень полуисхода — это количество вершин, которые непосредственно следуют за вершиной v.

Корнем орграфа будем называть вершину со степенью полузахода 0. Лист, наоборот, вершина со степенью полуисхода 0. Например, если граф состоит из одной вершины, то данная вершина будет одновременно и корнем и листом.

Путь или маршрут в графе — это последовательность вершин и ребер v_1, e_1, u_1, v_2, e_2, u_2, …, v_n, e_n, u_n такая, что $e_i=(v_i, u_i)$ для всех i от 1 до n. Если $v_1=u_n$, то маршрут называется замкнутым. Цепь маршрут, все ребра которого различны. Для орграфов цепь называется орцепью. Цикл — замкнутая цепь. Для орграфов цикл называется контуром.

Неориентированный граф является связным, если в нём есть путь из каждой вершины в любую другую. Для орграфов обычно вводят два определения связности: сильной и слабой. Орграф сильно-связан, если в нём существует ориентированный путь из каждой вершины в любую другую. Орграф слабо-связан если будет связным неориентированный граф полученный из ориентированного графа заменой ориентированных ребер на неориентированные. Под тем что орграф связен будем понимать, что он связан слабо.

Деревом называют связный граф, не содержащий циклов.

Инвариант — это свойство, которое сохраняется для изоморфных объектов. Это определение подходит и для графов, и для полугрупп.

Два графа G_1, G_2 называются изоморфными, если их множества вершин равны и существует биекция f из G_1 в G_2 такая, что для любых двух вершин u, v из графа G_1 верно, что ребро (u, v) в графе G_1 есть тогда и только тогда, когда есть ребро (f(u), f(v)) в графе G_2.

Все данные и другие определения из теории графов можно найти в следующей литературе: [6][7][8].

1.2 Реализация графов в программе

Граф (точнее орграф) в программе реализован в виде класса на C++. Реализация графа представлена в двух вариантах SimpleDigraph и Digraph<T>. SimpleDigraph — это граф определенный на целых числах от 0 до n-1, где n — количество вершин в графе. Обычно граф определяют, либо с помощью списка смежности, либо с помощью матрицы смежности. Каждый из способов имеет свои плюсы и минусы:

Задача	Оптимальный вариант
Проверка на вхождение ребра (x, y) в граф	Матрица смежности
Определение степени вершины	Списки смежности
Объем памяти для небольших графов	Матрица смежности (с небольшим преимуществом)
Вставка или удаления ребра	Матрица смежности
Обход графа	Списки смежности

Данная таблица из книги «Алгоритмы. Руководство по разработке» Стивена Скиена[5] с небольшими изменениями.

Я решил с комбинировать оба варианта. При этом SimpleDigraph будет оптимально справляется почти со всеми задачами, но будут и минусы:

1. Большее использование памяти;
2. Задача удаления вершин и ребер из графа.

Обе проблемы не важны для задач данной работы. Оперативная память сейчас у персональных компьютеров в избытке. А возможность удаления вершин и ребер из графа я убрал. SimpleDigraph имеет следующий интерфейс:

1. SimpleDigraph(n) — конструктор. Создает граф из n вершин без ребер;
2. add_node() — добавляет вершину в граф и возвращает её номер;
3. add_edge(u, v) — добавляет в граф дугу (u, v);
4. add_edges(edges) — добавляет сразу несколько дуг в граф;
5. is_edge(u, v) — проверяет есть ли дуга (u, v) в графе;
6. successors(u) — возвращает массив вершин, непосредственно следующих из вершины u;
7. predecessors(u) — возвращает массив вершин, непосредственно предшествующих вершине u;
8. number_of_nodes() — возвращает количество вершин в графе;
9. number_of_edges() — возвращает количество ребер в графе;
10. is_connected() — проверяет является ли граф связным;
11. is_tree() — проверяет является ли граф деревом;
12. find_root() — находит один корень графа;
13. shortest_path_length(u, v) — находит длину кратчайшего пути от вершины u до вершины v. Можно находить как ориентированный, так и неориентированный кратчайший путь;
14. is_node_exist(u) — проверяет есть ли в графе вершина под номером u;
15. operator==(g2), operator!=(g2) — проверяет (не) равен ли данный граф графу g2. Под равенством здесь понимается следующее. Два графа $G_1=(V_1, E_1)$, $G_2=(V_2, E_2)$ равны если равны их множества вершин и ребер равны: $V_1=V_2$, $E_1=E_2$.

Следующие переменные и функции недоступны для пользователя (private секция класса):

1. n — количество вершин;
2. m — количество ребер;
3. adj_matrix — матрица смежности;
4. _successors, _predecessors — списки вершин, которые непосредственно следуют/предшествуют вершинам графа;
5. _shortest_path_length, _undirected_shortest_path_length — матрицы, в которых запоминаются расстояния от одной вершины до другой;
6. resize(n) — изменяет размер графа;

Класс SimpleDigraph на языке C++ выглядит следующим образом:

```
class SimpleDigraph {
  int n = 0;
  std::vector<std::vector<int>> adj_matrix;
  void resize(int n);
  std::vector<std::vector<int>> _shortest_path_length,
      _undirected_shortest_path_length;

public:
  explicit SimpleDigraph(int n = 0);
  int add_node();
  void add_edge(int u, int v);
  void add_edges(std::vector<std::pair<int, int>> edges);
  bool is_edge(int u, int v);
  std::vector<int> successors(int u);
  std::vector<int> predecessors(int u);
  int number_of_nodes();
  int number_of_edges();
  bool is_tree_with_root(int root);
  int shortest_path_length(int u, int v, bool undirected);
  int shortest_path_length(int u, int v);
  bool is_node_exist(int v);
  void throw_exception_if_node_does_not_exist(int u);
};
```

Ниже приводится реализация каждого из методов:

```
SimpleDigraph::SimpleDigraph(int an) : m(0) { resize(an); }

void SimpleDigraph::resize(int new_n) {
  n = new_n;
  if (adj_matrix.size() != n) {
    adj_matrix.resize(n);
    _successors.resize(n);
    _predecessors.resize(n);
```

```
    _shortest_path_length.resize(n);
    _undirected_shortest_path_length.resize(n);
    for (int i = 0; i < n; i++) {
      adj_matrix[i].resize(n);
      _shortest_path_length[i].resize(n, -1);
      _undirected_shortest_path_length[i].resize(n, -1);
      _shortest_path_length[i][i] =
_undirected_shortest_path_length[i][i] = 0;
    }
  }
}

int SimpleDigraph::add_node() {
  n++;
  resize(n);
  return n - 1;
}

bool SimpleDigraph::is_node_exist(int v) const { return 0 <= v && v <
n; }

void SimpleDigraph::add_edge(int u, int v) {
  assert(is_node_exist(v) && is_node_exist(u));
  if (is_edge(u, v))
    return;
  adj_matrix[u][v] = 1;
  _successors[u].push_back(v);
  _predecessors[v].push_back(u);
  m++;
}

void SimpleDigraph::add_edges(vector<pair<int, int>> edges) {
  for (auto &p : edges) {
    add_edge(p.first, p.second);
  }
}

bool SimpleDigraph::is_edge(int u, int v) const {
  assert(is_node_exist(v) && is_node_exist(u));
  return adj_matrix[u][v];
}

vector<int> &SimpleDigraph::successors(int u) {
  assert(is_node_exist(u));
  return _successors[u];
}

vector<int> &SimpleDigraph::predecessors(int u) {
  assert(is_node_exist(u));
  return _predecessors[u];
}

int SimpleDigraph::number_of_nodes() const { return n; }

int SimpleDigraph::number_of_edges() const { return m; }

pair<bool, int> SimpleDigraph::find_root() const {
  for (int i = 0; i < n; i++) {
```

```
    if (_predecessors[i].size() == 0) {
      return {true, i};
    }
  }
  return {false, 0};
}

bool SimpleDigraph::is_connected() const {
  if (n <= 1)
    return true;
  queue<int> q;
  vector<bool> used(n);
  q.push(0);
  int k = 0;
  while (!q.empty()) {
    int w = q.front();
    q.pop();
    used[w] = true;
    k++;
    for (int i = 0; i < n; i++) {
      if ((adj_matrix[w][i] || adj_matrix[i][w]) && !used[i]) {
        q.push(i);
      }
    }
  }
  if (k < n)
    return false;
  return true;
}

bool SimpleDigraph::is_tree() const {
  if (number_of_nodes() - 1 == number_of_edges() && is_connected())
    return true;
  return false;
}

int SimpleDigraph::shortest_path_length(int u, int v, bool undirected)
{
  assert(is_node_exist(v) && is_node_exist(u));
  if (undirected && _undirected_shortest_path_length[u][v] >= 0) {
    return _undirected_shortest_path_length[u][v];
  }
  if (!undirected && _shortest_path_length[u][v] >= 0) {
    return _shortest_path_length[u][v];
  }
  queue<pair<int, int>> q;
  vector<bool> used(n);
  q.emplace(u, 0);
  while (!q.empty()) {
    int w, d;
    tie(w, d) = q.front();
    q.pop();
    used[w] = true;
    if (undirected) {
      _undirected_shortest_path_length[u][w] =
          _undirected_shortest_path_length[w][u] = d;
    } else {
      _shortest_path_length[u][w] = d;
```

```
    }
    if (w == v) {
      return d;
    }
    for (int i = 0; i < n; i++) {
      if ((adj_matrix[w][i] || (undirected && adj_matrix[i][w])) && !
used[i]) {
        q.emplace(i, d + 1);
      }
    }
  }
  return -1;
}

bool SimpleDigraph::operator==(const SimpleDigraph &g2) const {
  if (number_of_nodes() != g2.number_of_nodes())
    return false;
  if (number_of_edges() != g2.number_of_edges())
    return false;
  if (number_of_nodes() <= 1)
    return true;
  for (int u = 0; u < n; u++) {
    for (int v = 0; v < n; v++) {
      if (is_edge(u, v) != g2.is_edge(u, v))
        return false;
    }
  }
  return true;
}

bool SimpleDigraph::operator!=(const SimpleDigraph &g2) const {
  return !operator==(g2);
}
```

Digraph<T> в качестве типа вершин может использовать произвольный тип T. Например, Digraph<string> — граф вершины которого являются строками, а Digraph<int> — граф целых чисел (причем числа могут идти в любом порядке, а не 0, ..., n-1, как в SimpleDigraph). Отчасти Digraph<T> — это оболочка над SimpleDigraph, но с дополнительными возможностями. Помимо тех методов, которые есть для SimpleDigraph, у Digraph<T> есть несколько дополнительных:

1. nodes() — возвращает множество всех вершин графа;
2. edges() — возврщает все ребра графа;
3. to_string() — преобразует граф в строку по правилу описанному ниже;
4. from_stream, from_string, from_file — методы, которые позволяют получать граф из разных источников.

   Класс Digraph<T> на языке C++:

```
template <class T> class Digraph {
  std::set<T> _elements;
  std::map<T, int> id;
  std::vector<T> name;
  std::map<T, std::vector<T>> _predecessors, _successors;
  SimpleDigraph simple_digraph;

public:
  Digraph();
  void add_node(T v);
  void add_nodes(std::vector<T> nodes);
  const std::set<T> &nodes();
  bool is_node(T v) const;
  void add_edge(T u, T v);
  bool is_edge(T u, T v);
  void add_edges(std::vector<std::pair<T, T>> edges);
  std::vector<std::pair<T, T>> edges();
  const std::vector<T> &successors(T v);
  const std::vector<T> &predecessors(T v);
  int number_of_nodes() const;
  int number_of_edges() const;
  bool is_tree() const;
  std::pair<bool, T> find_root() const;
  int shortest_path_length(T u, T v, bool undirected = false);
  const SimpleDigraph &get_simple_digraph();
  const std::map<T, int> &get_id();
  T get_name_by_id(int aid);
  std::string to_string();
  static Digraph<T> from_stream(std::istream &is);
  static Digraph<T> from_string(std::string str);
  static Digraph<T> from_file(std::string filename);
  bool operator==(const Digraph<T> &g2);
  bool operator!=(const Digraph<T> &g2);
};
```

Реализация методов данного класса:

```
template <class T> Digraph<T>::Digraph() {}

template <class T> void Digraph<T>::add_node(T v) {
  if (is_node(v))
    return;
  id[v] = simple_digraph.add_node();
  name.push_back(v);
  _elements.insert(v);
}

template <class T> bool Digraph<T>::is_node(T v) const {
  return _elements.find(v) != _elements.end();
}

template <class T> void Digraph<T>::add_nodes(vector<T> nodes) {
  for (T v : nodes)
    add_node(v);
}
```

```
template <class T> const set<T> &Digraph<T>::nodes() { return
_elements; }

template <class T> Digraph<T> Digraph<T>::from_stream(std::istream
&is) {
  Digraph<T> g;
  while (!is.eof()) {
    string line;
    getline(is, line);
    if (line.empty())
      continue;
    istringstream iss(line);
    T u, v;
    iss >> u;
    g.add_node(u);
    while (iss >> v) {
      g.add_edge(u, v);
      u = v;
    }
  }
  return g;
}

template <class T> Digraph<T> Digraph<T>::from_string(std::string str)
{
  stringstream ss(str);
  return Digraph<T>::from_stream(ss);
}

template <class T> Digraph<T> Digraph<T>::from_file(std::string
filename) {
  ifstream f(filename);
  return Digraph<T>::from_stream(f);
}

template <class T> const vector<T> &Digraph<T>::successors(T v) {
  return _successors[v];
}

template <class T> const vector<T> &Digraph<T>::predecessors(T v) {
  return _predecessors[v];
}

template <class T> int Digraph<T>::number_of_nodes() const {
  return simple_digraph.number_of_nodes();
}

template <class T> int Digraph<T>::number_of_edges() const {
  return simple_digraph.number_of_edges();
}

template <class T> bool Digraph<T>::is_tree() const {
  return simple_digraph.is_tree();
}

template <class T> pair<bool, T> Digraph<T>::find_root() const {
  auto p = simple_digraph.find_root();
  if (p.first == false) {
```

```
    return {false, T()};
  }
  return {p.first, name[p.second]};
}

template <class T>
int Digraph<T>::shortest_path_length(T u, T v, bool undirected) {
  if (!is_node(u) || !is_node(v))
    return -1;
  return simple_digraph.shortest_path_length(id[u], id[v],
undirected);
}

template <class T> bool Digraph<T>::is_edge(T u, T v) {
  if (!is_node(u) || !is_node(v))
    return false;
  return simple_digraph.is_edge(id[u], id[v]);
}

template <class T> void Digraph<T>::add_edge(T u, T v) {
  if (is_edge(u, v))
    return;
  if (!is_node(u))
    add_node(u);
  if (!is_node(v))
    add_node(v);
  simple_digraph.add_edge(id[u], id[v]);
  _successors[u].push_back(v);
  _predecessors[v].push_back(u);
}

template <class T> void Digraph<T>::add_edges(vector<pair<T, T>>
edges) {
  for (auto p : edges) {
    add_edge(p.first, p.second);
  }
}

template <class T> vector<pair<T, T>> Digraph<T>::edges() {
  vector<pair<T, T>> res;
  for (T u : _elements) {
    for (T v : _elements) {
      if (is_edge(u, v)) {
        res.emplace_back(u, v);
      }
    }
  }
  return res;
}

template <class T> std::string Digraph<T>::to_string() {
  stringstream ss;
  // ss << "nodes " << _elements << endl;
  map<T, bool> used;
  for (auto e : edges()) {
    used[e.first] = used[e.second] = true;
    ss << e.first << " " << e.second << endl;
  }
```

```
  for (T n : _elements) {
    if (!used[n]) {
      ss << n << endl;
    }
  }
  return ss.str();
}

template <class T> const SimpleDigraph
&Digraph<T>::get_simple_digraph() {
  return simple_digraph;
}
template <class T> const std::map<T, int> &Digraph<T>::get_id()
{ return id; }

template <class T> bool Digraph<T>::operator==(const Digraph<T> &g2) {
  return get_id() == g2.get_id() &&
         get_simple_digraph() == g2.get_simple_digraph();
}

template <class T> bool Digraph<T>::operator!=(const Digraph<T> &g2) {
  return !operator==(g2);
}
```

Хранится и вводится граф в виде списка путей, например:

```
0 1 2
1 3
```

Здесь строка «0 1 2» означает, что в графе есть путь 0 → 1 → 2. Граф из одной вершины задается так: «0».

После того как классы графов были написаны, были добавлены тесты, которые проверяют корректность работы методов.

1.3 Проверка деревьев на изоморфизм

Наиболее простым случаем графа является дерево. Для деревьев есть простой алгоритм проверки на изоморфизм. Опишем его.

Входные данные: дерево T и вершина r, являющаяся корнем дерева T.

Результатом выполнения алгоритма будет строка. Обозначим результат выполнения алгоритма A для дерева T с корнем r так: A (T, r).

Шаг 1. Если дерево состоит всего из одной вершины, то сопоставим ему строку "01".

Шаг 2. Если в дереве более одной вершины, то рассматриваем вершины, являющиеся детьми корня дерева: r_1, …, r_n. Для каждой из этих вершин строим деревья T_1, …, T_n, имеющие корнями вершины r_1, …, r_n .

Шаг 3. Для каждого из полученных деревьев T1, …, Tn запускается алгоритм A.

Замечание. В результате первых трех шагов всем концевым (висячим) вершинам дерева сопоставляется строка "01".

Шаг 4. Находим вершины дерева, расстояние от которых до корня максимально и которым последовательность еще не сопоставлена.

Шаг 5. Пусть u — такая вершина. Для вершины u находим всех детей. Для этих детей ранее были найдены соответствующие им последовательности S_1, …, S_n,.

Шаг 6. Строки S_1, …, S_n располагаем в лексикографическом порядке (как слова в словаре). Получаем упорядоченный набор строк s_1, …, s_n. Тогда вершине u сопоставим строку: "0s_1...s_n1".

Шаг 7. Если u — корень дерева T, то алгоритм останавливаем. Если нет, то переходим к шагу 4.

Результатом работы алгоритма будет строка «0s_1...s_n1», где s_1, …, s_n — последовательности, которые сопоставлены детям корня r дерева T.

Данный алгоритм не очень сложный, если искать только строки деревьев и сравнивать их. Небольшая сложность может возникнуть при поиске биекции. Далее приведена реализация данного алгоритма на языке C++ с возможностью поиска биекции:

```
template <class T> class TreeEncoder {
  Digraph<T> &g;
  std::map<T, std::string> s;
  std::pair<bool, T> r;
  void go();
  void go(const T &v);
  bool _good = false, computed = false;
public:
  TreeEncoder(Digraph<T> &ag);
  string code();
  string code_of_node(T v);
  bool good();
};


template <class T> class TreeIso {
  Digraph<T> &g1, &g2;
  TreeEncoder<T> e1, e2;
  std::map<T, T> f;
  int _is_iso = -1;
  bool f_computed = false;
```

```
public:
  TreeIso(Digraph<T> &ag1, Digraph<T> &ag2);
  bool is_iso();
  const std::map<T, T> &get_biection();
};


template <class T> TreeEncoder<T>::TreeEncoder(Digraph<T> &ag) : g(ag)
{}

template <class T> void TreeEncoder<T>::go() {
  r = g.find_root();
  if (!r.first || !g.is_tree()) {
    _good = false;
    return;
  }
  _good = true;
  if (g.number_of_nodes() == 0)
    return;
  go(r.second);
  computed = true;
}

template <class T> void TreeEncoder<T>::go(const T &v) {
  string t = "1";
  vector<string> a;
  for (const T &u : g.successors(v)) {
    go(u);
    a.push_back(s[u]);
  }
  sort(a.begin(), a.end());
  for (string &tt : a) {
    t += tt;
  }
  t += "0";
  s[v] = t;
}

template <class T> string TreeEncoder<T>::code() {
  if (!computed)
    go();
  return s[r.second];
}

template <class T> string TreeEncoder<T>::code_of_node(T v) {
  if (!computed)
    go();
  return s[v];
}

template <class T> bool TreeEncoder<T>::good() {
  if (!computed)
    go();
  return _good;
}

template <class T>
```

```
TreeIso<T>::TreeIso(Digraph<T> &ag1, Digraph<T> &ag2)
    : g1(ag1), g2(ag2), e1(ag1), e2(ag2) {}

template <class T> bool TreeIso<T>::is_iso() {
  if (_is_iso >= 0)
    return _is_iso;
  if (!e1.good() || !e2.good()) {
    _is_iso = 0;
    return false;
  }
  _is_iso = e1.code() == e2.code() ? 1 : 0;
  return _is_iso;
}

template <class T> const std::map<T, T> &TreeIso<T>::get_biection() {
  if (_is_iso < 0) {
    is_iso();
  }
  if (!_is_iso || f_computed) {
    return f;
  }
  T r1 = g1.find_root().second;
  T r2 = g2.find_root().second;
  f[r1] = r2;
  queue<T> q;
  q.push(r1);
  while (!q.empty()) {
    T v1 = q.front();
    q.pop();
    T v2 = f[v1];
    vector<pair<T, string>> a1, a2;
    for (const T &u1 : g1.successors(v1)) {
      a1.emplace_back(u1, e1.code_of_node(u1));
    }
    for (const T &u2 : g2.successors(v2)) {
      a2.emplace_back(u2, e2.code_of_node(u2));
    }
    sort(a1.begin(), a1.end(),
         [](auto &a, auto &b) { return a.second < b.second; });
    sort(a2.begin(), a2.end(),
         [](auto &a, auto &b) { return a.second < b.second; });
    for (int i = 0; i < a1.size(); i++) {
      f[a1[i].first] = a2[i].first;
      q.push(a1[i].first);
    }
  }
  f_computed = true;
  return f;
}
```

1.4 Проверка произвольных графов на изоморфизм

Если графы не являются деревьями и не сработает ни один из инвариантов, то необходимо использовать общий алгоритм проверки графов на изоморфизм. Вот его описание.

Пусть V — подмножество вершин 1-го графа, а U — подмножество вершин 2-го графа. На протяжении всего алгоритма подграф состоящий из вершин V и подграф состоящий из вершин U изоморфны. Сначала V = {0}. А для U последовательно перебираются все вершины x: U={x}, предполагая, что эти вершины совпадут при изоморфизме, т. е. что f(0) = x.

Далее добавляем вершину k в множество V={0, 1, …, k-1}. Перебираем все не использованные вершины y 2-го графа, чтобы выполнялось условие: если есть ребро из вершин 0, 1, …, k-1 в вершину k или из вершины k в одну из вершин 0, 1, …, k-1, то тоже самое должно быть верно для вершины y и вершин f(0), …, f(k-1). Если ни одной такой вершины не найдено, то возвращаемся на шаг назад. На данном шаге мы можем использовать последовательности из инварианта 3 для вершин k и y. Если эти последовательности не совпадают, то вершины k и v не могут совпасть, если графы изоморфны.

Если хотя бы раз |V|=n, то графы изоморфны и f искомая биекция. Иначе графы не изоморфны.

Реализация данного алгоритма:

```
template <class T, class U> class DigraphIso {
  Digraph<T> orig_g1;
  Digraph<U> orig_g2;
  SimpleDigraph g1, g2;
  std::set<int> vertices;
  int n = -1;    // количество вершин в графах g1, g2
  int g1_k = -1; // подмножество вершин g1 {0, 1, ..., g1_k}
  std::set<int> g2_s; // Подмножество вершин графа g2, которое
  // соотвествует множеству вершин грфафа g1 {0, 1, ..., g1_k}
  vector<int> f; // биекция: i (вершина g1) -> f[i] (вершина g2)
  int ans = -1;
  bool match(int v);
  void go();
  std::map<T, U> f2;
  std::function<bool(T, U)> match2;
  std::map<T, U> initial_bijection;

public:
  DigraphIso(Digraph<T> ag1, Digraph<U> ag2,
             std::function<bool(T, U)> amatch2 = [](T x, U y) { return
true; });
  bool is_iso();
  void set_initial_biection(std::map<T, U> b) {
    initial_bijection = b;
    ans = -1;
  }
  std::map<T, U> get_biection();
};
```

```
template <class T, class U>
DigraphIso<T, U>::DigraphIso(Digraph<T> ag1, Digraph<U> ag2,
                             std::function<bool(T, U)> amatch2)
    : orig_g1(ag1), orig_g2(ag2), g1(ag1.get_simple_digraph()),
      g2(ag2.get_simple_digraph()), match2(amatch2) {}

template <class T, class U> bool DigraphIso<T, U>::match(int v) {
  // Подмножество g1 {0, 1, ..., g1_k - 1} изоморфно
  // подножеству графа g2 g2_s,
  // которое равно {f[0], f[1], ..., f[g1_k - 1]}
  // На данном шаге нужно проверить может ли подмножество g1
  // {0, 1, ..., g1_k - 1, g1_k} быть изоморфно
  // подмножеству g2 {f[0], f[1], ..., f[g1_k - 1], v}
  if (!match2(orig_g1.get_name_by_id(g1_k),
orig_g2.get_name_by_id(v)))
    return false;
  for (int u = 0; u < g1_k; u++) {
    if (g1.is_edge(u, g1_k) && !g2.is_edge(f[u], v)) {
      // cout << 1 << ": " << u << " " << g1_k << " "
      //      << f[u] << " " << v << endl;
      return false;
    }
    if (g1.is_edge(g1_k, u) && !g2.is_edge(v, f[u])) {
      // cout << 2 << endl;
      return false;
    }
  }
  return true;
}

template <class T, class U> void DigraphIso<T, U>::go() {
  // cout << "go begin\n";
  // cout << "g2_s: ";
  // for (int i : g2_s) cout << i << " ";
  // cout << endl;
  if (g2_s.size() == n) {
    ans = 1;
    return;
  }
  g1_k += 1;
  set<int> t;
  set_difference(vertices.begin(), vertices.end(), g2_s.begin(),
g2_s.end(),
                 inserter(t, t.end()));
  // cout << "t: ";
  // for (int i : t) cout << i << " ";
  // cout << endl;
  for (int v : t) {
    if (ans >= 0)
      return;
    if (match(v)) {
      // cout << g1_k << " match " << v << endl;
      auto g1_k_name = orig_g1.get_name_by_id(g1_k);
      auto v_name = orig_g2.get_name_by_id(v);
      auto it = initial_bijection.find(g1_k_name);
      if (!(it != initial_bijection.end() && it->second != v_name)) {
        f[g1_k] = v;
```

```
        f2[g1_k_name] = v_name;
        g2_s.insert(v);
        go();
        g2_s.erase(v);
      }
    } else {
      // cout << g1_k << " don't match " << v << endl;
    }
  }
  g1_k -= 1;
}

template <class T, class U> bool DigraphIso<T, U>::is_iso() {
  if (ans >= 0)
    return ans;
  if (g1.number_of_nodes() != g2.number_of_nodes()) {
    return false;
  }
  n = g1.number_of_nodes();
  if (n == 0) {
    return true;
  }
  f.resize(n);
  for (int i = 0; i < n; i++)
    vertices.insert(i);
  go();
  // cout << "ans " << ans << endl;
  if (ans < 0)
    ans = 0;
  return ans;
}

template <class T, class U> map<T, U> DigraphIso<T, U>::get_biection()
{
  return f2;
}
```

2 ПОЛУРЕШЕТКИ

2.1 Определения

Бинарная операция * на множестве A — это отображение из A2 в A. Результат бинарной операции записывают в виде x*y.

Бинарная операция * является ассоциативной, если для любых элементов x, y, z выполняется равенство: (x*y)*z=x*(y*z).

Бинарная операция * является коммутативной, если для любых элементов x, y выполняется равенство: x*y=y*x.

Бинарная операция * является идемпотентной, если для любого элемента x выполняется равенство: x*x=x.

Полугруппа — это множество с определенным на нем ассоциативной бинарной операцией.

Полурешетка — это полугруппа, операция которой коммутативна и идемпотентна.

Определение инварианта было дано в главе о графах.

Две полурешетки S1 и S2 изоморфны если равны их множества и существует биекция.

Полурешетки $(S_1,*)$ и $(S_2, \cdot)$ изоморфны, если существует биекция f из S_1 в S_2 такая, что для любых элементов a, b полурешетки S_1 выполняется равенство $f(a*b)=f(a)\cdot f(b)$.

Эти и другие определения можно найти в [1][9].

2.2 Представление полурешеток в программе

Полурешетки аналогично графам реализованы в виде двух классов SimpleSemilattice и Semilattice<T>. У SimpleSemilattice есть следующие публичные методы:

1. add_element() — добавляет элемент в полурешетку и возвращает номер этого элемента;
2. size() — возвращает количество элементов в полурешетке;
3. inf(a, b) — находит произведение элементов полурешетки;
4. set_inf(a, b, val) — устанавливает произведение элементов a, b полурешетки в val;
5. is_element(a) — проверяет есть ли в полурешетке элемент под номером a;
6. is_valid() — проверяет действительно ли это полурешетка;
7. is_associative() — проверка ассоциативности;
8. is_commutativity() — проверка коммутативности;
9. is_idempotence() — проверка идемпотентности;

Класс SimpleSemilattice на C++:

```
class SimpleSemilattice {
  std::vector<std::vector<int>> op;
  void resize(int new_n);

public:
```

```
  explicit SimpleSemilattice(int new_n = 0);
  int add_element();
  int size();
  int inf(int a, int b);
  void set_inf(int a, int b, int val);
  bool is_element(int a);
  void throw_exception_if_element_does_not_exist(int a);
  bool is_valid();
  bool is_associative();
  bool is_commutativity();
  bool is_idempotence();
};
```

Реализация методов данного класса:

```
SimpleSemilattice::SimpleSemilattice(int an) : n(0) { resize(an); }

void SimpleSemilattice::resize(int new_n) {
  assert(new_n >= 0);
  if (op.size() != new_n) {
    op.resize(new_n);
    for (int i = 0; i < new_n; i++) {
      op[i].resize(new_n);
      if (i >= n) {
        op[i][i] = i;
      }
    }
  }
  n = new_n;
}

int SimpleSemilattice::add_element() {
  resize(++n);
  return n - 1;
}

int SimpleSemilattice::size() { return n; }

int SimpleSemilattice::inf(int a, int b) {
  assert(is_element(a) && is_element(b));
  return op[a][b];
}

void SimpleSemilattice::set_inf(int a, int b, int val) {
  assert(is_element(a) && is_element(b) && is_element(val));
  op[a][b] = val;
}

bool SimpleSemilattice::is_element(int a) { return 0 <= a && a < n; }

bool SimpleSemilattice::is_valid() {
  // cout << endl << is_associative() << endl;
  // cout << endl << is_commutativity() << endl;
  // cout << endl << is_idempotence() << endl;
  return is_associative() && is_commutativity() && is_idempotence();
}
```

```
bool SimpleSemilattice::is_associative() {
  for (int x = 0; x < n; x++) {
    for (int y = 0; y < n; y++) {
      for (int z = 0; z < n; z++) {
        if (op[x][op[y][z]] != op[op[x][y]][z]) {
          return false;
        }
      }
    }
  }
  return true;
}

bool SimpleSemilattice::is_commutativity() {
  for (int x = 0; x < n; x++) {
    for (int y = 0; y < n; y++) {
      if (op[x][y] != op[y][x]) {
        return false;
      }
    }
  }
  return true;
}

bool SimpleSemilattice::is_idempotence() {
  for (int x = 0; x < n; x++) {
    if (op[x][x] != x) {
      // cout << endl << x << " " << op[x][x] << endl;
      return false;
    }
  }
  return true;
}
```

Класс Semilattice<T> аналогичен SimpleSemilattice, но элементы могут быть для произвольного типа T и имеет несколько дополнительных функций:

1. from_stream, from_string, from_file — методы, которые позволяют получать полурешетку из разных источников.
2. to_string() — представление полурешетки в виде одной строки.

Класс Semilattice<T> на языке C++:

```
template <class T> class Semilattice {
  std::set<T> _elements;
  std::map<T, int> id;
  std::vector<T> name;
  SimpleSemilattice simple_semilattice;

public:
  void add_element(T a);
  template <class U> void add_elements(U elems);
  bool is_element(T a);
```

```
  int size();
  std::set<T>& elements();
  T inf(T a, T b);
  void set_inf(T a, T b, T c);
  void throw_exception_if_element_does_not_exist(T a);
  bool is_valid();
  bool is_associative();
  bool is_commutativity();
  bool is_idempotence();
  std::string to_string();
  static Semilattice<T> from_stream(std::istream &is);
  static Semilattice<T> from_file(std::string fn);
  static Semilattice<T> from_string(std::string s);
};
```

Реализация методов класса:

```
template <class T> void Semilattice<T>::add_element(T a) {
  if (is_element(a))
    return;
  id[a] = simple_semilattice.add_element();
  name.push_back(a);
  _elements.insert(a);
}

template <class T> std::set<T>& Semilattice<T>::elements() { return
_elements; }

template <class T> bool Semilattice<T>::is_element(T a) {
  return _elements.find(a) != _elements.end();
}

template <class T> int Semilattice<T>::size() {
  return simple_semilattice.size();
}

template <class T> T Semilattice<T>::inf(T a, T b) {
  assert(is_element(a) && is_element(b));
  auto res = simple_semilattice.inf(id[a], id[b]);
  return name[res];
}

template <class T> void Semilattice<T>::set_inf(T a, T b, T c) {
  assert(is_element(a) && is_element(b) && is_element(c));
  simple_semilattice.set_inf(id[a], id[b], id[c]);
}

template <class T>
Semilattice<T> Semilattice<T>::from_stream(std::istream &is) {
  Semilattice<T> s;
  vector<T> a;
  T t;
  set<T> elems;
  while (is >> t) {
    a.push_back(t);
    elems.insert(t);
  }
```

```
  int n = sqrt(a.size());
  if (n * n != a.size()) {
    return s;
  }
  if (elems.size() != n) {
    return s;
  }
  for (T el : elems) {
    s.add_element(el);
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      s.set_inf(s.name[i], s.name[j], a[i * n + j]);
    }
  }
  return s;
}

template <class T> Semilattice<T>
Semilattice<T>::from_string(std::string s) {
  stringstream ss(s);
  return from_stream(ss);
}

template <class T> Semilattice<T>
Semilattice<T>::from_file(std::string fn) {
  ifstream f(fn);
  return from_stream(f);
}

template <class T> bool Semilattice<T>::is_valid() {
  return simple_semilattice.is_valid();
}

template <class T> bool Semilattice<T>::is_associative() {
  return simple_semilattice.is_associative();
}

template <class T> bool Semilattice<T>::is_commutativity() {
  return simple_semilattice.is_commutativity();
}

template <class T> bool Semilattice<T>::is_idempotence() {
  return simple_semilattice.is_idempotence();
}

template <class T> string Semilattice<T>::to_string() {
  stringstream ss;
  int n = size();
  for(T a: _elements) {
    for(T b:_elements) {
      ss << inf(a, b) << " ";
    }
    ss << '\n';
  }
  return ss.str();
}
```

2.3 Граф полурешетки

Полурешетки интересны тем, что для каждой такой полугруппы можно построить граф.

Пусть множество M частично упорядочено. Любое частично упорядоченное множество можно представить как ориентированный граф, в котором дуга (a, b) между парой элементов означает $a < b$ и нет такого элемента c, что $a < c$ и $c < b$. Заметим, что не любой ориентированный граф является представлением частично упорядоченного множества. Чтобы ориентированный граф представлял частично упорядоченное множество, необходимо и достаточно чтобы в нем не было циклов. В математической литературе частично упорядоченные множества обычно изображаются в виде неориентированных графов, при этом подразумевается, что предшествующие элементы расположены ниже последующих. Поэтому, если в этих схемах правильно заменить ребра на дуги, то все дуги окажутся направленными снизу вверх. Обозначим полученный в результате граф символом .

Напомним, что частично упорядоченное множество называется нижней полурешеткой, если всякое двухэлементное его подмножество {a,b} имеет точную нижнюю (обозначают inf(a,b)) грань. Двойственным образом дается определение верхней полурешетки (верхнюю грань двухэлементного подмножества обозначают sup(a,b)). Частично упорядоченное множество, являющееся и нижней и верхней полурешеткой, называется решеткой.

Поясним, почему коммутативную полугруппу идемпотентов называют полурешеткой. Этот термин оправдан тем, что если рассмотреть на полурешетке S отношение естественного частичного порядка (заданного формулой: $e \le f \Leftrightarrow e \cdot f = f \cdot e = e$), то для любых e, f из S произведение $e \cdot f$ будет равно inf(e, f); и обратно, если M – частично упорядоченное множество, в котором любые два элемента имеют точную нижнюю грань, то операция, заданная условием $a \cdot b = \inf(a,b)$, превращает M в коммутативную полугруппу идемпотентов.

Так как любой коммутативной полугруппе идемпотентов $(S, \cdot)$ соответствует частично упорядоченное множество $(S, \le)$, то любую такую полугруппу можно изобразить в виде графа. По построенному таким образом графу легко искать

произведение элементов полурешетки: из вершин, соответствующих данным элементам, спускаемся вниз по ребрам до их пересечения. Элемент, стоящий на пересечении, является произведением элементов.

Возникает вопрос: является ли деревом полученный граф? Оказывается, нет. Дело в том, что для некоторых несравнимых элементов a и b коммутативной полугруппы идемпотентов $(S,\cdot)$ может существовать элемент c, отличный от элементов a и b такой, что $c = \sup(a,b)$. Тогда $c = \sup(a,c) = \sup(b,c)$. В этом случае в графе, соответствующем данной полугруппе, возникнет цикл.

Опишем коммутативные полугруппы идемпотентов $(S,\cdot)$, графы которых являются деревьями.

Пусть M – частично упорядоченное множество. Нижним конусом некоторого элемента a этого множества называется множество $L_a=\{x \in M \mid x \le a\}$. Частичный порядок на множестве M называется полулинейным снизу, если для каждого элемента a этого множества ее нижний конус линейно упорядочен.

Предложение. Представление коммутативной полугруппы идемпотентов $(S, \cdot)$ в виде графа является деревом, тогда и только тогда, когда частичный порядок, заданный формулой: $e \le f \Leftrightarrow e \cdot f = f \cdot e = e$, является полулинейным снизу.

Доказательство.

Пусть для коммутативной полугруппы идемпотентов $(S, \cdot)$ ее граф является деревом. Предположим, что порядок не является полулинейным снизу. Это означает, что существует элемент a полугруппы, для которого ее нижний конус не является линейно упорядоченным. Это, в свою очередь, означает, что существуют элементы из нижнего конуса этой полугруппы, которые не являются сравнимыми элементами. Пусть b и c – такие элементы. Но тогда $a = \sup(a,c) = \sup(a,b)$, поэтому в этом случае в графе будет замкнутый маршрут a…b…d…c… a , где $d = \inf(b,c)$, на месте многоточий находятся вершины, располагающиеся в графе между соответствующими вершинами. Из этого маршрута можно выделить цикл, что противоречит определению дерева.

Пусть теперь частичный порядок в коммутативной полугруппе идемпотентов $(S, \cdot)$, заданный формулой: $e \le f \Leftrightarrow e \cdot f = f \cdot e = e$, является полулинейным снизу. Предположим, что соответствующий полугруппе граф не является деревом. Это означает, что в нем есть цикл. Из алгоритма построения

дерева из этого следует, что для некоторых различных элементов полугруппы есть точные нижние и точные верхние грани и они отличны от самих элементов. Пусть b и c – такие элементы, a = sup(b,c), d = inf(b,c). Но тогда частичный порядок не является полулинейным снизу, так как нашлись элементы b и c, которые не являются сравнимыми.

Из предложения следует, что проверку изоморфизма полурешеток, для которых частичный порядок, заданный формулой: $e \le f \Leftrightarrow e \cdot f = f \cdot e = e$, является полулинейным снизу, можно произвести с помощью проверки изоморфизма соответствующих им деревьев.

Для деревьев существует эффективный алгоритм проверки изоморфизма, на основании которого и был реализован соответствующий алгоритм для полурешеток.

Созданная в итоге программа работает следующим образом: по таблицам Кэли двух полугрупп строятся соответствующие им деревья, а затем проверяется изоморфизм построенных деревьев.

Рассмотрим алгоритм построения графа полурешетки. Для этого сначала находится корень графа r. Он будет равен произведению всех элементов полурешетки $r=x_1*x_2*...*x_n$.

Затем находятся вершины x смежные с корнем. Вершина будет смежна с корнем, если r=r*x и не существует элемента y (отличного от r и x) такого, что r=r*y и y=y*x.

Аналогично находятся остальные вершины.

Граф являющийся графом некоторой полурешетки будем называть s-графом.

Реализация данного алгоритма на C++:

```
template <class T> Digraph<T> to_digraph(Semilattice<T> s) {
  // int n = s.size();
  auto elems = s.elements();
  T root = *begin(elems);
  Digraph<T> g;
  for (T el : elems) {
    g.add_node(el);
    root = s.inf(root, el);
  }
  set<T> level = {root};
  while (level.size() > 0) {
    set<T> new_level;
    for (T x : level) {
      for (T y : elems) {
        if (y != x && s.inf(x, y) == x) {
          bool flag = true;
```

```
          for (T z : elems) {
            if (z != x && z != y && x == s.inf(x, z) && z == s.inf(z,
y)) {
              flag = false;
              break;
            }
          }
          if (flag) {
            g.add_edge(x, y);
            new_level.insert(y);
          }
        }
      }
    }
    level = new_level;
  }
  return g;
}
```

2.4 Обратное построение полурешетки по её графу

Из графа полурешетки можно обратно получить полурешетку. Это похоже на то как это было описано выше. По построенному графу из вершин, соответствующих данным элементам, спускаемся вниз по ребрам до их пересечения.

В функцию inf_for_digraph передаютюся орграф g и две его вершины v, u. Заметим, что если inf существует то, он долен находится на пересечении двух каких-нибудь путей от root до v и от root до u. Перебор все таких путей и нахождение их пересечений займет слишком много времени для больших графов. Вместо этого будем находить вершины путей находящихся на одном расстояние до корня. Предположим, что вершина u находится дальше от корня, чем v. Найдем все смежные и предшествующие вершины для вершины u. Для этих вершин сделаем тоже самое и так далее, до тех пор пока вершины не будут на том же расстоянии до корня, что и вершина v. Далее нужно посмотреть есть ли вершина v среди найденых предков вершины u, находящихся на том же расстоянии от корня, что и v. Есть есть, то ответ inf(v, u) = v. Иначе находим всех предков вершин u, v уровнем ниже и снова проверяем их пересечение.

Реализация данного алгоритма нахождения операции inf для двух вершин графа полурешетки на C++:

```
template <class T> T inf_for_digraph(Digraph<T> g, T u, T v) {
  T root = find_root(g);
  int du = g.shortest_path_length(root, u);
```

```
  int dv = g.shortest_path_length(root, v);
  set<T> us = {u}, vs = {v};
  bool flag = false;
  while (1) {
    while (us.size() > 0 && (flag || du > dv)) {
      set<T> new_us;
      for (T x : us) {
        auto p = g.predecessors(x);
        new_us.insert(begin(p), end(p));
      }
      us = new_us;
      du--;
      flag = false;
    }
    while (vs.size() > 0 && dv > du) {
      set<T> new_vs;
      for (T x : vs) {
        auto p = g.predecessors(x);
        new_vs.insert(begin(p), end(p));
      }
      vs = new_vs;
      dv--;
      flag = false;
    }
    if (vs.size() == 0 || us.size() == 0) {
      return T();
    }
    set<T> uv_intersection;
    set_intersection(begin(us), end(us), begin(vs), end(vs),
                     inserter(uv_intersection,
uv_intersection.begin()));
    if (uv_intersection.size() > 1) {
      return T();
    } else if (uv_intersection.size() == 1) {
      return *begin(uv_intersection);
    } else {
      flag = true;
    }
  }
}
```

Если написать функцию, которая бы по заданному орграфу и двум его вершинам могла вычислять их inf, то создать полуршетки по орграфу будет тривиальным:

```
emplate <class T> Semilattice<T> to_semi(Digraph<T> g) {
  Semilattice<T> s;
  if (g.number_of_nodes() == 0) {
    return s;
  }
  if (g.number_of_nodes() == 1) {
    s.add_element(*g.nodes().begin());
    return s;
  }
  s.add_elements(g.nodes());
```

```
  for (T v : g.nodes()) {
    for (T u : g.nodes()) {
      s.set_inf(v, u, inf_for_digraph(g, v, u));
    }
  }
  return s;
}
```

2.5 Проверка на изоморфизм полурешеток, графы которых являются деревьями

Граф полурешетки может быть деревом. В этом случаю можно применить быстрый алгоритм проверки деревьев на изоморфизм, который был описан выше. Данный алгоритм дает значительное преимущество в скорости над общим алгоритмом.

Данный алгоритм предназначается для деревьев. Но реализован он таким образом, что программа применима также и для полурешеток, графы которых деревьями не являются. На таких графах программа ищет все пути, ведущие от корня до концевых вершин дерева, при этом одна и та же вершина может появиться в структуре графа несколько раз. При этом, если программа применяется к графам изоморфных полурешеток, то канонические имена будут совпадать, если же полурешетки не изоморфны, то канонические имена могут совпадать. Приведем пример таких полурешеток.

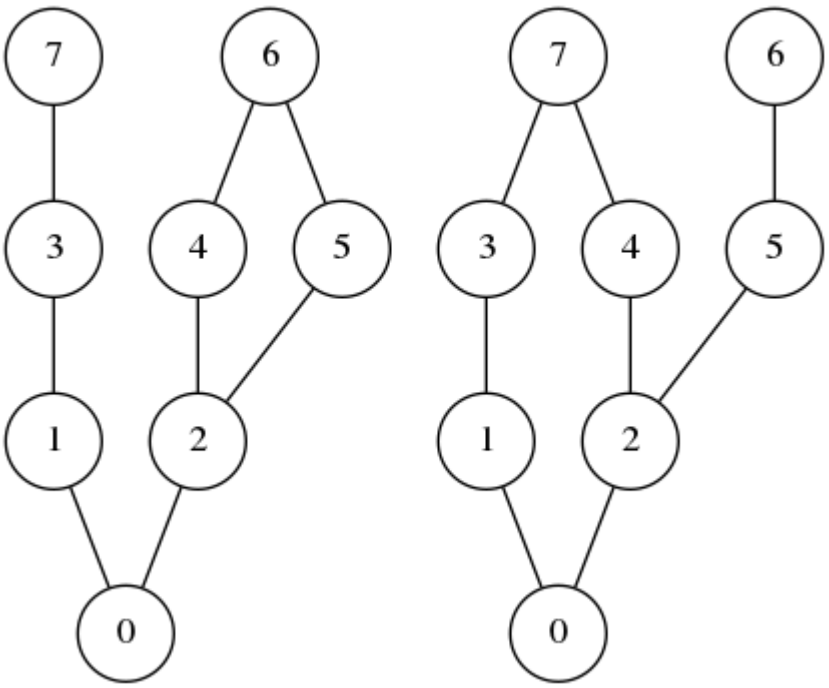


Рисунок 1 — Пример неизоморфных полурешеток, для которых совпадают канонические имена

Для того чтобы проверить, изоморфны ли две полугруппы, являющиеся полурешетками, необходимо сначала найти соответствующие им деревья.

Для начала находим корень полурешетки, для этого достаточно найти inf от всех элементов полурешетки, то есть root = inf(x1, x2, …, xn). Далее перебираем элементы полурешетки x ≠ root и inf(x, root) и выбираем среди них только те, для которых не существует элемента y ≠ root и y ≠ x и root = inf(root, y) и y = inf(y, x). Все найденные элементы x будут вершинами смежными с root. Аналогично находятся вершины смежные к только что найденным и так далее пока все элементы не будут исчерпаны. Код на языке C++:

```
template <class T> Digraph<T> to_digraph(Semilattice<T> s) {
  auto elems = s.elements();
  T root = *begin(elems);
  Digraph<T> g;
  for (T el : elems) {
    g.add_node(el);
    root = s.inf(root, el);
  }
  set<T> level = {root};
  while (level.size() > 0) {
    set<T> new_level;
    for (T x : level) {
      for (T y : elems) {
        if (y != x && s.inf(x, y) == x) {
          bool flag = true;
          for (T z : elems) {
            if (z != x && z != y && x == s.inf(x, z) && z == s.inf(z,
y){
              flag = false;
              break;
            }
          }
          if (flag) {
            g.add_edge(x, y);
            new_level.insert(y);
          }
        }
      }
    }
    level = new_level;
  }
  return g;
}
```

Приведем пример работы программы для двух полурешеток – S_1 и S_2, состоящих из 5 элементов, заданных с помощью таблиц Кэли Для каждой полугруппы сначала выводится ее структура (как дерева), затем выводится ее каноническое имя (строка), затем канонические имена сравниваются и выдается ответ: 1 или 0.

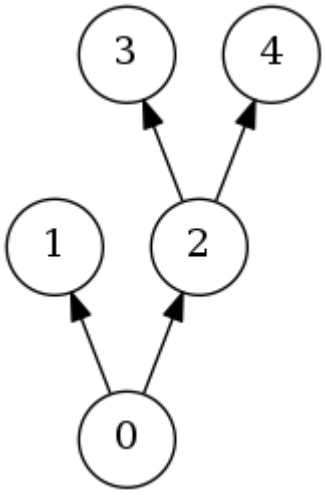
Для полурешеток, заданных данными таблицами, программа выдает результат, представленный ниже.

Полурешетки переданные в программу:

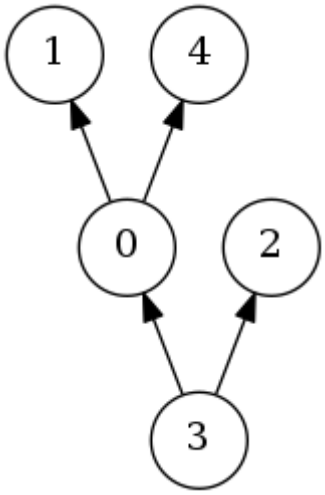
	0	1	2	3	4
0	0	0	0	0	0
1	0	1	0	0	0
2	0	0	2	2	2
3	0	0	2	3	2
4	0	0	2	2	4

	0	1	2	3	4
0	0	0	3	3	0
1	0	1	3	3	0
2	3	3	2	3	3
3	3	3	3	3	3
4	0	0	3	3	4

Граф первой полурешетки:



Граф второй полурешетки:



Вывод программы:

```
Проверка полурешеток s1 и s2 на изоморфизм.
Полурешетка s1:
0 0 0 0 0
0 1 0 0 0
0 0 2 2 2
0 0 2 3 2
0 0 2 2 4
Полурешетка s2:
0 0 3 3 0
0 1 3 3 0
3 3 2 3 3
3 3 3 3 3
0 0 3 3 4
Преобразование полурешеток s1, s2 в графы g1, g2... 0.001076 sec.
Граф g1:
0 1
0 2
2 3
2 4
Граф g2:
0 1
0 4
3 0
3 2
Количество вершин в графе g1: 5
Количество вершин в графе g2: 5
Количество ребер в графе g1: 4
Количество ребер в графе g2: 4
Корень в графе g1: 0
Корень в графе g2: 3
Является ли граф g1 деревом? 1
Является ли граф g2 деревом? 1
Оба графа - деревья.
Код графа g1: '0010010111'
Код графа g2: '0010010111'
Строки равны. Графы изоморфны.
Полурешетки имзоморфны? 1
Полное время проверки полурешеток на изоморфизм: 0.002897 sec.
```

Так как канонические имена деревьев совпадают, то делается вывод об изоморфизме деревьев, а значит, и полугрупп.

Для проверки времени работы данной программы было создана программа, генерирующая полурешетки по бинарным деревьям. Для двух таких полурешеток из 200 элементов программа работает менее 5 с.

2.6 Решение вопроса, является ли граф полурешетки планарным

Кроме деревьев, еще одним интересным классом графов, для которых вопрос проверки изоморфизма является решенным, являются планарные графы, к примеру, соответствующий алгоритм описан в статье. Поэтому стоит выяснить вопрос о планарности графа произвольной полурешетки.

Известен критерий планарности графа: граф планарен тогда и только тогда, когда в нем отсутствуют подграфы, гомеоморфные одному из графов K_5 или $K_{3,3}$.

Но граф произвольной полурешетки, в отличие от графа полурешетки, для которой частичный порядок, заданный формулой: $e \leq f \Leftrightarrow e \cdot f = f \cdot e = e$, является полулинейным снизу, не всегда планарный. Пример полурешетки, граф которой не является планарным, приведен на рисунке ниже:

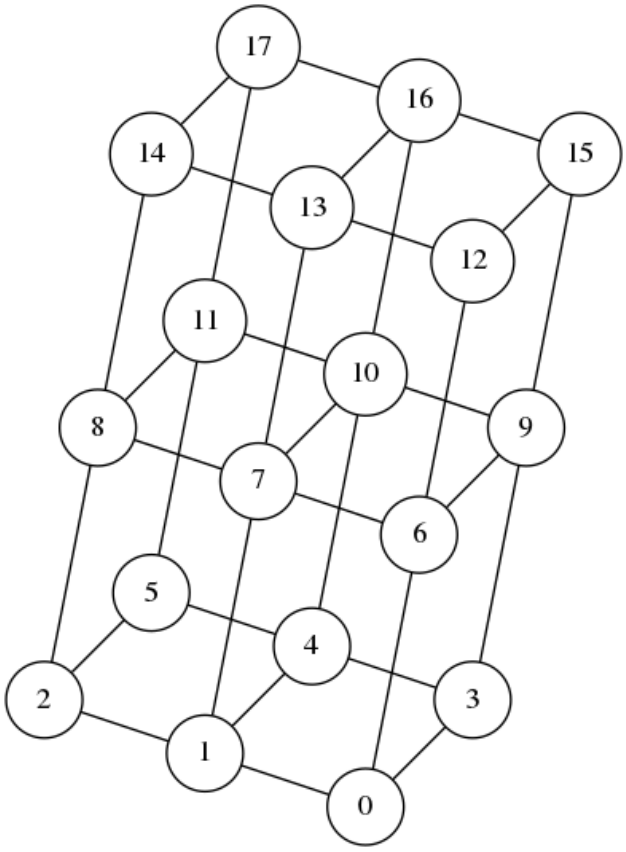


Рисунок 2 - Полурешетка, граф которой не является планарным

В соответствующей такому графу полурешетке элемент 0 является точной нижней гранью, элемент 17 – точной верхней гранью. Изображенный на рисунке граф планарным не является, так как есть подграф этого графа, гомеоморфный двудольному графу $K_{3,3}$. В одной доле этого графа расположим вершины 13, 11, 10, в другой – вершины 12, 16, 7.

Заметим, что при удалении всего одного ребра (7, 10) в указанном графе он становится планарным.

2.7 Инварианты для графа, изображающего полурешетку

Предварительно рассмотрим для графа следующую систему инвариантов. Будем называть графы, изображающие полурешетки, s-графами. Сопоставим ориентированному графу неориентированный граф.

1. Число вершин графа.
2. Число ребер графа.

3. Следующий инвариант является упорядоченной последовательностью, составленной следующим образом.

Пусть v является вершиной s-графа. Пусть этой вершине в полурешетке соответствует элемент a. Вершине v сопоставим три числа и одну последовательность. Опишем их.

a) Первое число – длина кратчайшей цепи от вершины до корня.
b) Второе число – число вершин графа, удовлетворяющих следующему условию: им соответствует в полурешетке элемент b, такой, что b = inf (a, b) и нет такого элемента c, отличного от a и b, что c = inf (a, c) и b = inf (c, b). (В соответствующем ориентированном графе это полустепень захода вершины).
c) Третье число – число вершин графа, удовлетворяющих следующему условию: им соответствует в полурешетке элемент b, такой, что a = inf (a, b) и нет такого элемента c, отличного от a и b, что a = inf (a, c) и c = inf (c, b). (В соответствующем ориентированном графе это полустепень исхода вершины).
d) Сопоставим вершине упорядоченную по возрастанию последовательность длин цепей от вершины v до всех концевых вершин. Такая последовательность всегда будет состоять из t элементов, где t – это число концевых вершин. Если вершина концевая, то один из элементов последовательности будет равен 0.

В результате каждой вершине v графа сопоставляется последовательность следующего вида: k, p, r, (f1, …, ft). Упорядочим последовательности, сопоставленные вершинам графа лексикографически, по возрастанию, начиная с первого числа.

В итоге получим последовательность: k1, p1, r1, (f11, …, f1t); k2, p2, r2, (f21, …, f2t);…; km, pm, rm, (fm1, …, fmt) (здесь m – число вершин графа).

Полученная в результате последовательность будет являться инвариантом графа. Назовем его инвариант 3.

Данная система инвариантов полной не является. На рисунке 3 приведен пример неизоморфных полурешеток, для которых инварианты 1, 2, 3 совпадают, кроме того, проверка для них алгоритмом, созданным для деревьев, также дает одинаковый результат.

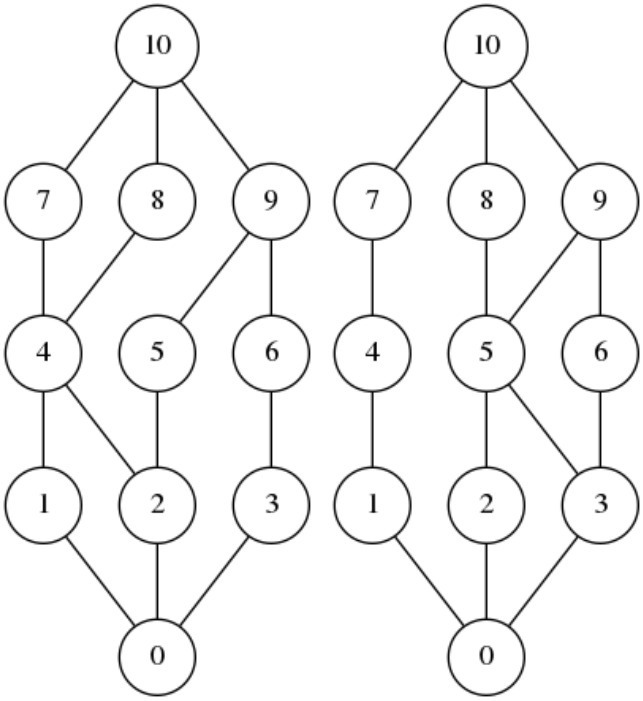


Рисунок 3 - Графы неизоморфных полурешеток, которым соответствуют одинаковые инварианты 1,2,3 и совпадают канонические имена.

У графов на рисунке 3 одинаковое количество вершин и ребер, совпадают описанные ранее последовательности. Но графы не являются изоморфными, так как, к примеру, во втором графе вершины степени 3 соединены цепью 0 –1 – 4 – 7 – 10 три из этих вершин имеют степень два, а в первом графе такой цепи нет.

Код, который находит инвариант 3 графа:

```
template <class T> class Inv3 {
  bool is_empty = false;
  Digraph<T> g;
  T root;
  std::vector<T> leaves;
  void find_leaves();
  std::map<T, std::string> inv3_for_node;
  std::string full_inv3;
  std::string compute_inv3_for_node(T v);
  std::string compute_full_inv3();

public:
  Inv3();
```

```
  explicit Inv3(Digraph<T> ag);
  std::string get_inv3_for_node(T v);
  std::string get_full_inv3();
};

template <class T> Inv3<T>::Inv3() : is_empty(true), root() {}

template <class T> Inv3<T>::Inv3(Digraph<T> ag) : root() {
  if (ag.number_of_nodes() == 0) {
    is_empty = true;
    return;
  }
  g = ag;
  find_leaves();
  root = find_root(g);
  for (T v : g.nodes()) {
    inv3_for_node[v] = compute_inv3_for_node(v);
  }
  full_inv3 = compute_full_inv3();
}

template <class T> void Inv3<T>::find_leaves() {
  for (T v : g.nodes()) {
    if (g.successors(v).size() == 0) {
      leaves.push_back(v);
    }
  }
}

template <class T> string Inv3<T>::get_inv3_for_node(T v) {
  return inv3_for_node[v];
}

template <class T> string Inv3<T>::compute_inv3_for_node(T v) {
  if (is_empty)
    return "";
  stringstream ss;
  ss << g.shortest_path_length(root, v, true) << ", "
     << g.predecessors(v).size() << ", " << g.successors(v).size() <<
", (";
  vector<int> t;
  for (int i = 0; i < leaves.size(); i++) {
    t.push_back(g.shortest_path_length(leaves[i], v, true));
  }
  sort(t.begin(), t.end());
  for (int i = 0; i < t.size(); i++) {
    ss << t[i];
    if (i < t.size() - 1)
      ss << ", ";
  }
  ss << ")";
  return ss.str();
}

template <class T> string Inv3<T>::get_full_inv3() { return full_inv3;
}

template <class T> string Inv3<T>::compute_full_inv3() {
```

```
  if (is_empty)
    return "";
  vector<string> arr;
  for (T v : g.nodes()) {
    arr.push_back(get_inv3_for_node(v));
  }
  sort(arr.begin(), arr.end());
  string full_inv3;
  for (string x : arr) {
    full_inv3 += x + "; ";
  }
  return full_inv3;
}
```

2.8 Проверка полурешеток на изоморфизм

Собрав воедино все, что было написано о графах и полурешетках выше, можно сформулировать алгоритм, который будет проверять произвольные полурешетки на изоморфизм.

1. Находим для для полурешеток S и S' соответствующие им графы так, как это было описано в [10].

2. Сравниваем число вершин и ребер в графах. Если они не совпадают для двух графов, то ответ: полурешетки не изоморфны. Если они совпадают, то переходим к шагу 3.

3. Если графы G и G' являются деревьями, то применяем ранее описанный алгоритм проверки изоморфизма деревьев.

4. Если графы не являются деревьями, то преобразуем каждый из них в два графа G_1 и G_2, G'_1 и G'_2. Опишем, как строятся графы G_1 и G_2. Рассмотрим максимальные подграфы графа G, имеющие корнем вершину, являющуюся корнем графа G и лишь одну вершину, смежную с корнем. Будем называть такие подграфы ветвями s-графа. Смотрим, нет ли в пересечении каких- нибудь из ветвей вершины, кроме корня. Если есть, то добавляем к одному из них ребра и вершины другого, которых нет в первом. Полученный в результате подграф также будем называть ветвью графа. Граф G преобразуем в два графа G_1 и G_2, корень графа G становится единственной общей вершиной и корнем этих подграфов. Граф G_1 является связным подграфом графа G, который содержит ветви графа, не содержащие циклов. Граф G_2 является связным подграфом графа G, который состоит из ветвей, содержащих циклы.

5. Аналогичным образом получаем графы G'_1 и G'_2 из графа G'.

6. Проверяем на изоморфизм графы G_1 и G'_1, которые являются деревьями. Если графы не являются изоморфными, то ответ: полурешетки не изоморфны. Если графы изоморфны, то переходим к шагу 6.

7. Рассматриваем графы G_2 и G'_2. Сначала проверяем их алгоритмом для деревьев. Если канонические имена графов совпадают, то переходим к шагу 7. Если нет, то ответ: полурешетки не изоморфны.

8. Находим графов и инварианты 3. Сравниваем полученные последовательности. последовательности не совпадают для двух графов, то ответ: полурешетки не изоморфны. Если они совпадают, то переходим к шагу 8.

9. Ищем изоморфизм графов. При этом сопоставляем вершинам одного графа вершины другого, если у них совпадают последовательности. В итоге перебор возникает только для тех вершин, для которых совпадает инвариант 4. Если изоморфизм найден, то ответ: полурешетки изоморфны, если нет, то ответ: полурешетки не изоморфны.

Реализация описанного выше алгоритма на языке C++ с дополнительным выводом, который можно включать/отключать при запуске программы:

```
template <class T> bool is_isomorphic(Semilattice<T> s1,
Semilattice<T> s2) {

  clock_t t, start_time = clock();

  auto end_of_log = [&](bool res) {
    semi_log << "Полурешетки имзоморфны? " << res << endl;
    semi_log << "Полное время проверки полурешеток на изоморфизм: "
             << ((float)(clock() - start_time) / CLOCKS_PER_SEC) << "
sec."
             << endl;
  };

  if (s1.size() != s2.size()) {
    semi_log << "Полурешетки не изоморфны, т.к. состоят из различного
"
                "количества переменных."
             << endl;
    end_of_log(0);
    return false;
  }

  if (s1.size() == 0) {
    semi_log << "Путсые полурешетки изоморфны." << endl;
    end_of_log(1);
    return true;
  }
```

```
  if (s1.size() == 1) {
    semi_log << "Полурешетки, состоящие из одного элемента,
изоморфны." << endl;
    end_of_log(1);
    return true;
  }

  auto tree_is_isomorphic_with_log = [&](Digraph<T> &g1, string
g1_name,
                                         Digraph<T> &g2, string
g2_name) {
    cout << "Проверка графов " << g1_name << " и " << g2_name
         << " на изоморфизм с помощью алгоритма проверки деревьев на
изоморфизм"
         << endl;
    string s1 = encode_tree(g1);
    string s2 = encode_tree(g2);
    bool res = s1 == s2;
    semi_log << "Код графа " << g1_name << ": '" << s1 << "'" << endl;
    semi_log << "Код графа " << g2_name << ": '" << s2 << "'" << endl;
    if (res) {
      semi_log << "Строки равны следовательно графы изоморфны." <<
endl;
    } else {
      semi_log << "Строки не равны следовательно графы не изоморфны."
<< endl;
    }
    return res;
  };

  if (log_mode) {
    semi_log << "Проверка полурешеток s1 и s2 на изоморфизм." << endl;
    semi_log << "Полурешетка s1:\n" << s1.to_string();
    semi_log << "Полурешетка s2:\n" << s2.to_string();
    semi_log << "Преобразование полурешеток s1, s2 в графы g1, g2...
";
    t = clock();
  }

  Digraph<T> g1 = to_digraph(s1), g2 = to_digraph(s2);

  if (log_mode) {
    float dt = ((float)(clock() - t) / CLOCKS_PER_SEC);
    semi_log << dt << " sec." << endl;
    semi_log << "Граф g1:\n" << g1.to_string();
    semi_log << "Граф g2:\n" << g2.to_string();
    t = clock();
  }

  int n1 = g1.number_of_nodes(), n2 = g2.number_of_nodes();

  if (log_mode) {
    semi_log << "Количество вершин в графе g1: " << n1 << endl;
    semi_log << "Количество вершин в графе g2: " << n2 << endl;
  }

  if (n1 != n2) {
    if (log_mode) {
```

```
      semi_log
          << "В графах различное количество вершин, поэтому
полурешетки не "
             "изоморфны."
          << endl;
    }
    return false;
  }

  int m1 = g1.number_of_edges(), m2 = g2.number_of_edges();

  if (log_mode) {
    semi_log << "Количество ребер в графе g1: " << m1 << endl;
    semi_log << "Количество ребер в графе g2: " << m2 << endl;
  }

  if (m1 != m2) {
    if (log_mode) {
      semi_log << "В графах различное количество ребер, поэтому
полурешетки не "
                  "изоморфны."
               << endl;
    }
    return false;
  }

  T r1 = find_root(g1), r2 = find_root(g2);

  if (log_mode) {
    semi_log << "Корень в графе g1: " << r1 << endl;
    semi_log << "Корень в графе g2: " << r2 << endl;
  }

  bool g1_is_tree = g1.is_tree_with_root(r1);
  bool g2_is_tree = g2.is_tree_with_root(r2);

  if (log_mode) {
    semi_log << "Является ли граф g1 деревом? " << g1_is_tree << endl;
    semi_log << "Является ли граф g2 деревом? " << g2_is_tree << endl;
  }

  if (g1_is_tree != g2_is_tree) {
    if (log_mode) {
      semi_log
          << "Один из графов - дерево, а другой нет, поэтому
полурешетки не "
             "изоморфны."
          << endl;
    }
    return false;
  }

  if (g1_is_tree && g2_is_tree) {
    if (log_mode) {
      semi_log << "Оба графа - деревья." << endl;
      bool res = tree_is_isomorphic_with_log(g1, "g1", g2, "g2");
      end_of_log(res);
      return res;
```

```
    } else {
      return tree_is_isomorphic(g1, g2);
    }
  }

  if (log_mode) {
    float dt = ((float)(clock() - t) / CLOCKS_PER_SEC);
    semi_log << "... " << dt << " sec." << endl;
    semi_log << "Поиск графов G1 и G2... ";
    t = clock();
  }

  Digraph<T> g1_G1, g1_G2, g2_G1, g2_G2;
  tie(g1_G1, g1_G2) = find_G1_and_G2_graphs(g1);
  tie(g2_G1, g2_G2) = find_G1_and_G2_graphs(g2);

  if (log_mode) {
    float dt = ((float)(clock() - t) / CLOCKS_PER_SEC);
    semi_log << dt << " sec." << endl;
    semi_log << "Граф g1_G1:\n" << g1_G1.to_string();
    semi_log << "Граф g1_G2:\n" << g1_G2.to_string();
    semi_log << "Граф g2_G1:\n" << g2_G1.to_string();
    semi_log << "Граф g2_G2:\n" << g2_G2.to_string();
    semi_log << "Проверка графов g1_G1, g2_G1 и g1_G2, g2_G2 на
изоморфизм с "
                "помощью алгоритма для деревьев... ";
    t = clock();
  }

  bool res1 = tree_is_isomorphic(g1_G1, g2_G1);
  bool res2 = tree_is_isomorphic(g1_G2, g2_G2);

  if (log_mode) {
    float dt = ((float)(clock() - t) / CLOCKS_PER_SEC);
    semi_log << dt << " sec." << endl;
  }

  if (log_mode) {
    bool res1 = tree_is_isomorphic_with_log(g1_G1, "g1_G1", g2_G1,
"g2_G1");
    if (!res1) {
      end_of_log(0);
      return false;
    }
    bool res2 = tree_is_isomorphic_with_log(g1_G2, "g1_G2", g2_G2,
"g2_G2");
    if (!res2) {
      end_of_log(0);
      return false;
    }
  } else {
    if (!res1 || !res2) {
      return false;
    }
  }

  if (log_mode) {
    semi_log << "Поиск инварианта 3 для графов g1_G2 и g2_G2... ";
```

```
    t = clock();
  }

  Inv3<T> inv3_for_g1_G2(g1_G2);
  Inv3<T> inv3_for_g2_G2(g2_G2);

  if (log_mode) {
    float dt = ((float)(clock() - t) / CLOCKS_PER_SEC);
    semi_log << dt << " sec." << endl;
    semi_log << "Инвариант 3 для каждой из вершин графа g1_G2:" <<
endl;
    for (T v : g1_G2.nodes()) {
      semi_log << v << ": '" << inv3_for_g1_G2.get_inv3_for_node(v) <<
"'"
               << endl;
    }
    semi_log << "Инвариант 3 для каждой из вершин графа g1_G2:" <<
endl;
    for (T v : g2_G2.nodes()) {
      semi_log << v << ": '" << inv3_for_g2_G2.get_inv3_for_node(v) <<
"'"
               << endl;
    }
    semi_log << "Инвариант 3 для графа g1_G2: '"
             << inv3_for_g1_G2.get_full_inv3() << "'" << endl;
    semi_log << "Инвариант 3 для графа g2_G2: '"
             << inv3_for_g2_G2.get_full_inv3() << "'" << endl;
  }

  bool iseqinv3 =
      inv3_for_g1_G2.get_full_inv3() ==
inv3_for_g2_G2.get_full_inv3();

  if (!iseqinv3) {
    if (log_mode) {
      semi_log
          << "Инварианты графов не совпадают, значит полурешетки не
изоморфны."
          << endl;
      end_of_log(0);
    }
    return false;
  }

  if (log_mode) {
    semi_log << "Проверка графов на изоморфизм с помощью общего
алгоритма... ";
    t = clock();
  }

  DigraphIso<T, T> digiso(g1_G2, g2_G2, [&](T v, T u) {
    return inv3_for_g1_G2.get_inv3_for_node(v) ==
           inv3_for_g2_G2.get_inv3_for_node(u);
  });
  // digiso.set_initial_biection({{r1, r2}});
  bool res = digiso.is_iso();

  if (log_mode) {
```

```
    float dt = ((float)(clock() - t) / CLOCKS_PER_SEC);
    semi_log << dt << " sec." << endl;
  }

  if (log_mode && res) {
    semi_log << "Графы изоморфны. Биекция:\n";
    map<T, T> b = digiso.get_biection();
    for (auto p : b) {
      semi_log << p.first << " <-> " << p.second << endl;
    }
  }

  if (log_mode) {
    end_of_log(res);
  }

  return res;
}
```

2.9 Генераторы больших полурешеток

Для проверки данного алгоритма на больших полурешетках и сравнении с другими алгоритмами проверки графов на изоморфизм были написаны несколько генераторов полурешеток определенного вида.

3 ДЕТАЛИ РЕАЛИЗАЦИИ И РЕЗУЛЬТАТЫ

3.1 Выбор языка программирования. Система автоматизации сборки проекта. Тестирование

Все языки программирования имеют свои достоинства и недостатки. На каких-то разработка идет быстро и легко, а на каких-то долго и с затруднениями. На одних языках программы получаются быстрее, а на других медленнее. Скриптовые языки программирования скорее всего плохой выбор для решения нашей задачи, т.к. задача проверки на изоморфизм имеет высокую сложность и поэтому программа будет работать медленно для полурешеток большого порядка. Из компилируемых языков был выбран язык программирования C++, так как для него написана богатая библиотека шаблонов STL, которая поможет в реализации многих алгоритмов. Для C++ есть огромное множество инструментов, упрощающих разработку программ.

Программные проекты обычно состоят из огромного количества файлов. Каждый файл исходного кода необходимо сначала перевести в объектный файл, а затем собрать все объектные файлы в один исполняемый файл. Чтобы не делать это вручную были придуманы системы автоматизации сборки проекта. Наиболее известные системы автоматизации сборки: make, cmake, scons, apache ant, autotools. Для работы были выбраны системы cmake и make. Cmake использовался собственно для сборки программы, а make для выполнения различных команд (запуск сборки, автоматическое тестирование и т.п.).

Тестирование важный элемент разработки программ. После того как написан код выполняющий некоторую задачу, необходимо проверить действительно ли эта задача выполняется так как нужно. Для этого можно запускать программу с разными входными данными и вручную проверять вывод. Делать это после каждого изменения программного кода утомительно и нерационально. Эффективнее будет написать заранее все необходимые тесты и проверки, а потом запускать их одним нажатием кнопки. Для C++ есть несколько систем автоматического или модульного тестирования: boost::test, UnitTest++, CxxTest, CppUnit. Я выбрал CxxTest по следующим причинам:

1. CxxTest не требует установки.
2. Нет зависимостей от других библиотек.
3. Библиотека проста в использовании.
4. Высокая скорость работы.

3.2 Поиск всех полурешеток малого порядка

В интернете я смог найти лишь одну программу, с помощью которой можно найти все полурешетки порядка от 1 до 8. Программа называется GAP (Groups, Algorithms, Programming) и найти её можно в интернете по следующему адресу: http://www.gap-system.org/. Для GAP есть пакет smallsemi (https://www.gap-system.org/Packages/smallsemi.html), который позволяет работать с полугруппами порядка от 1 до 8. Полугрупп порядка 9 слишком много, а количество полугрупп порядка даже 10 неизвестно. Из всех полугрупп я выделил лишь те, что являются полурешетками. Их оказалось значительно меньше, чем полугрупп:

Количество полугрупп	Количество полуршеток
1	1
4	1
18	2
126	5
1 160	15
15 973	53
836 021	222
1 843 120 128	1 078

Найденные полурешетки были записаны в файл small_semilattices.txt и были использованы для тестирования алгоритма.

3.3 Проверка всех полурешеток малого порядка на изоморфизм

Полурешеток малого порядка, как выяснилось, не много. Ниже в таблице представлены результаты проверки этих полурешеток с помощью программы:

Порядок полугруппы	Количество полугрупп	Количество полурешеток	Количество деревьев	Максимальное количество полурешеток с совпадающим инвариантом 3	Максимальное количество полурешеток с совпадающим каноническим кодом	Максимальное количество полурешеток с совпадающим и инвариантом 3 и каноническим кодом
1	1	1	1	1		
2	4	1	1	1		
3	18	2	2	1		
4	126	5	4	1		
5	1 160	15	9	1		
6	15 973	53	20	1		
7	836 021	222	48	1		
8	1 843 120 128	1 078	115	2		

Из таблицы можно заключить, что инвариант 3 для полурешеток порядка от 1 до 7 является инвариантом. Это показывает эффективность построенного инварианта.

Полуерешетки 8 порядка, у которых совпадают инвариант 3 выглядят похожими или даже изоморфными, но это не так. Легко показать, что графы этих полурешеток не изоморфны. В полурешетке слева есть путь длины 3 от корня до верны степени 3. В полурешетке справа такого пути нет.

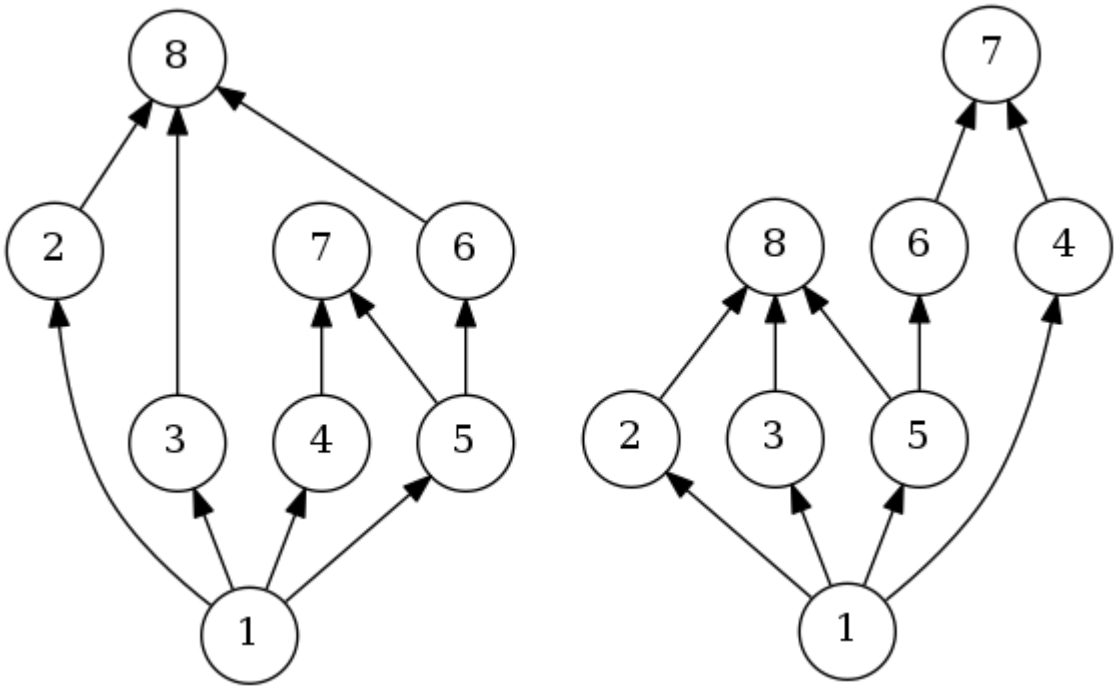


Рисунок 4 - Пример не изоморфных полурешеток, у которых совпадает инвариант 3

3.4 Сравнение с другими программами проверки графов на изоморфизм

С помощью поисковых систем Google и Яндекс я смог найти всего одну программу для проверки графов на изоморфизм – nauty. Также с помощью поисковика можно наткнуться на некоторые алгоритмы для проверки графов на изоморфизм и почти всегда без реализации на каком-нибудь языке программирования. Для C++ есть собрание библиотек классов Boost. Среди них есть библиотека для работы с графами, с помощью которой можно проверять графы на изоморфизм. На крупнейшем веб-сервисе для хостинга IT-проектов GitHub я нашел несколько реализаций различных алгоритмов проверки графов на изоморфизм. Наиболее популярные из найденых алгоритмов я выберу для сравнения с написанной мною программой. Найти какие-либо программы для работы с полурешетками не удалось.

3.5 Веб-приложение

К программе был написан интуитивно понятный интерфейс в виде небольшого веб-приложения. Данное приложение также позволяет легко продемонстрировать возможности программы. Веб-приложение состоит из клиентской и серверной части. Сервеная часть написана на node.js с помощью фреймворка koa. Клиенская часть состоит из нескольких веб-страничек динамически взаимодействующих с сервером с помощью технологии ajax.

Запускается веб-приложение с помощью команды make runapp из корня проекта. Для запуска необходима программа node.js версии 7 или выше. Если вы запустили сервер локально, то можно перейти по ссылки из вывода: localhost:3497.

Если использовать данное веб-приложение, то можно получить визуализацию графов в выводе программы.

Далее на рисунках представлена различные моменты работы веб-приложения.

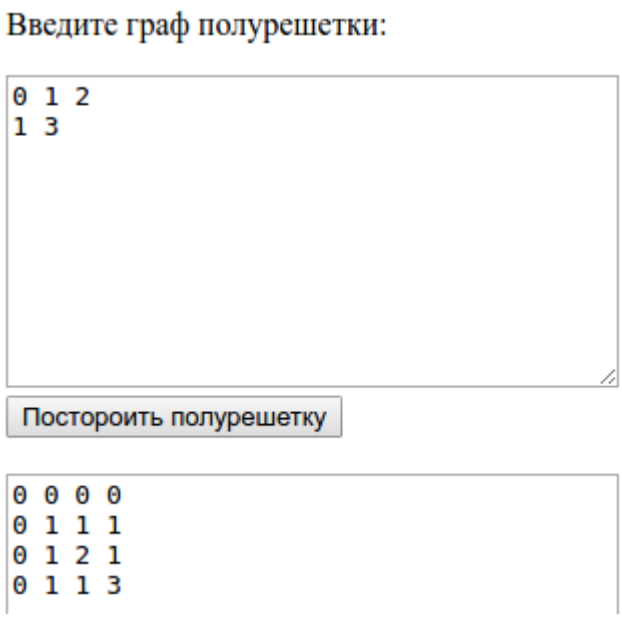


Рисунок 5 - Преобразование графа полурешетки в полурешетку

Проверка полурешеток на изоморфизм

example 1 | example 2 | example 3 | example 4 | example 5 | example 6

Введите полурешетку S_1:

```
0 0 0 0 0
0 1 1 0 0
0 1 2 3 0
0 0 3 3 0
0 0 0 0 4
```

Введите полурешетку S_2:

```
0 1 2 4 4
1 1 4 4 4
2 4 2 4 4
4 4 4 3 4
4 4 4 4 4
```

Проверить на изоморфизм

Проверить на изоморфизм с подробным выводом

Рисунок 6 - Начальная страница после запуска веб-приложения

```
Проверка полурешеток s1 и s2 на изоморфизм.
Полурешетка s1:
0 0 0 0 0
0 1 1 0 0
0 1 2 3 0
0 0 3 3 0
0 0 0 0 4
Полурешетка s2:
0 1 2 4 4
1 1 4 4 4
2 4 2 4 4
4 4 4 3 4
4 4 4 4 4
Преобразование полурешеток s1, s2 в графы g1, g2... 0.000964 sec.
Граф g1:
0 1
0 3
0 4
1 2
3 2
```

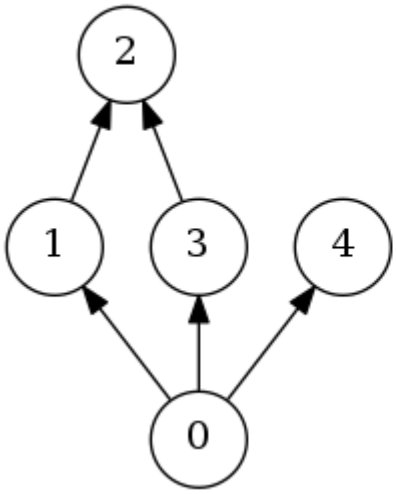


Рисунок 7 - Визуализация графов в выводе программы

ЗАКЛЮЧЕНИЕ

С графами традиционно тесно связан ряд математических объектов, таких как бинарные отношения, упорядоченные множества, некоторые виды полугрупп, алфавитные коды и другие. Эти взаимосвязи рассматриваются, например, в работах [7,8]. В данной статье описан алгоритм, позволяющий установить или опровергнуть изоморфизм двух конечных полурешеток, используя взаимосвязь этого специального вида полугрупп и графов. Алгоритм эффективно реализован для полурешеток порядка n=?

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ляпин Е. С. Полугруппы. — М.: Физматлит, 1960. — 592 с.
2. Г. С. Яблонский, В. И. Быков, А. Н. Горбань, Кинетические модели каталитических реакций, Новосибирск: Наука (Сиб. отделение), 1983.- 255 с.
3. Ерёменко А. О. Использование теории графов при решении задач в экономике // Прогрессивные технологии и экономика в машиностроении : сборник трудов VII Всероссийской научно-практической конференции для студентов и учащейся молодежи, г. Юрга, 7-9 апреля 2016 г. : в 2 т. — Томск : Изд-во ТПУ. — 2016. — Т. 2. — С. 279-281.
4. Курейчик В. М., Глушань В. М., Щербаков Л. И. Комбинаторные аппаратные модели и алгоритмы в САПР. М.: Радио и связь, 1990. 216 с.
5. «Алгоритмы. Руководство по разработке» Стивена Скиена
6. Дистель Р. Теория графов Пер. с англ.
7. Харари Ф. Теория графов
8. Оре О. Теория графов
9. клиффорд престон алгебраическая теория полугрупп
10. Л. В. Зяблицева, С. А. Пестов. Применение алгоритмов проверки изоморфизма графов в теории полугрупп. – Архангельск: Вестник Северного (Арктического) федерального университета. Серия: Естественные науки. 2016. № 4. с. 69 - 74.