МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теоретических основ компьютерной безопасности и криптографии

# ОТНОШЕНИЕ ЭКВИВАЛЕНТНОСТИ И ОТНОШЕНИЕ ПОРЯДКА

ОТЧЁТ
ПО ДИСЦИПЛИНЕ
«ПРИКЛАДНАЯ УНИВЕРСАЛЬНАЯ АЛГЕБРА»

студента 3 курса 331 группы
специальности 10.05.01 Компьютерная безопасность
факультета компьютерных наук и информационных технологий
Бородина Артёма Горовича

Преподаватель
профессор, д.ф.-м.н. _____ В. А. Молчанов
подпись, дата

Саратов 2022

## СОДЕРЖАНИЕ

## 1 Постановка задачи

**Цель работы** — изучение основных свойств бинарных отношений и операций замыкания бинарных отношений.

**Порядок выполнения работы:**

**1.** Разобрать определения отношения эквивалентности, фактор-множества. Разработать алгоритмы построения эквивалентного замыкания бинарного отношения и системы представителей фактор-множества.

**2.** Разобрать определения отношения порядка и диаграммы Хассе. Разработать алгоритмы вычисления минимальных (максимальных) и наименьших (наибольших) элементов и построения диаграммы Хассе.

**3.** Разобрать определения контекста и концепта. Разработать алгоритм вычисления решетки концептов.

## 2 Теоретические сведения по рассмотренным темам с их обоснованием

**Определение.** Бинарное отношение $\varepsilon$ на множестве $A$ называется *отношением эквивалентности (эквивалентностью)*, если оно рефлексивно, симметрично и транзитивно.

Для обозначения эквивалентности $\varepsilon$ используется инфиксная запись с помощью символа $\equiv$: вместо $(a,\, b) \in \varepsilon$ пишут $a \equiv b(\varepsilon)$ или просто $a \equiv b$.

Срезы $\varepsilon(a)$ называются *классами эквивалентности* по отношению $\varepsilon$ и обозначаются символом $[a]$. Множество всех таких классов эквивалентности $\{\, [a] : a \in A \,\}$ называется *фактор-множеством* множества $A$ по эквивалентности $\varepsilon$ и обозначается $A/\varepsilon$.

**Определение.** Подмножество $T \subset A$ называется *полной системой представителей классов* эквивалентности $\varepsilon$ на множестве $A$, если:

1) $\varepsilon(T) = A$,

2) из условия $t_1 \equiv t_2(\varepsilon)$ следует $t_1 = t_2$.

Классы эквивалентности $[t] \in A/\varepsilon$ могут быть отождествлены со своими представителями $t$, и фактор-множество $A/\varepsilon$ может быть отождествлено с множеством $T$.

**Определение.** Бинарное отношение $\omega$ на множестве $A$ называется *отношением порядка (порядком)*, если оно рефлексивно, антисимметрично и транзитивно.

Множество $A$ с заданным на нём отношением порядка $\leq$ называется *упорядоченным множеством* и обозначается $A = (A,\, \leq)$.

**Определение.** Элемент $a$ упорядоченного множества $(A,\, \leq)$ *называется*:

1. *Минимальным*, если $(\forall\, x \in A)\ x \leq a \implies x = a$,
2. *Максимальным*, если $(\forall\, x \in A)\ a \leq x \implies x = a$,
3. *Наименьшим*, если $(\forall\, x \in A)\ a \leq x$,
4. *Наибольшим*, если $(\forall\, x \in A)\ x \leq a$.

Упорядоченное множество $A = (A, \leq)$ наглядно представляется *диаграммой Хассе*, которая представляет элементы множества $A$ точками плоскости и пары $a \lessdot b$ представляет линиями, идущими *вверх* от элемента $a$ к элементу $b$.

**Алгоритм построения диаграммы Хассе конечного упорядоченного множества** $A = (A, \leq)$.

1. В упорядоченном множестве $A = (A, \leq)$ найти множество $A_1$ всех минимальных элементов и расположить их в один горизонтальный ряд (это первый уровень диаграммы).

2. В упорядоченном множестве $A \setminus A_1$, найти множество $A_2$ всех минимальных элементов и расположить их в один горизонтальный ряд над первым уровнем (это второй уровень диаграммы). Соединить отрезками элементы этого ряда с покрываемыми ими элементами предыдущего ряда.

3. В упорядоченном множестве $A \setminus (A_1 \cup A_2)$ найти множество $A_3$ всех минимальных элементов и расположить их в один горизонтальный ряд над вторым уровнем (это третий уровень диаграммы). Соединить отрезками элементы этого ряда с покрываемыми ими элементами предыдущих рядов.

4. Процесс продолжается до тех пор, пока не выберутся все элементы множества $A$.

**Определение.** Подмножество $X$ упорядоченного множества $(A, \leq)$ называется:

1. *Ограниченным сверху*, если найдется такой элемент $a \in A$, что $x \leq a$ для всех $x \in X$; в этом случае элемент $a$ называется *верхней гранью* множества $X$; если для множества $X$ существует наименьшая верхняя грань, то она обозначается символом $\sup X$ и называется *точной верхней гранью* множества $X$; в случае $\sup X \in X$ значение $\sup X$ является наибольшим элементом множества и обозначается $\max X$;

2. *Ограниченным снизу*, если найдется такой элемент $a \in A$, что $a \leq x$ для всех $x \in X$; в этом случае элемент $a$ называется нижней гранью множества $X$; если для множества $X$ существует наибольшая нижняя грань, то она обозначается символом $\inf X$ и называется *точной нижней гранью* множества $X$; в случае $\inf X \in X$ значение $\inf X$ является наименьшим элементом множества и обозначается $\min X$.

**Определение.** Порядок $\leq$ на множестве $A$ называется:

1. *Линейным*, если любые два элемента этого множества сравнимы, т.е. выполняется $(\forall\, a, b \in A)\,(a \leq b \vee b \leq a)$;

2. *Полным*, если его любое непустое подмножество имеет точную верхнюю и точную нижнюю грани;

3. *Решеточным*, если для всяких $a, b \in A$ существуют $\sup\{a, b\}$ и $\inf\{a, b\}$, которые обозначаются соответственно $a \vee b,\ a \wedge b$ и называются также *объединением* и *пересечением* элементов $a, b$.

Множество с заданным на нем линейным порядком называется *линейно упорядоченным множеством* или *цепью*.

Множество с заданным на нем решеточным порядком называется *решеточно упорядоченным множеством* или *решеткой*.

Бинарное отношение $\rho \subset G \times M$ между элементами множеств $G$ и $M$ можно рассматривать как базу данных с множеством объектов $G$ и множеством атрибутов $M$. Такая система называется контекстом и определяется следующим образом:

**Определение.** *Контекстом* называется алгебраическая система $K = (G, M, \rho)$, состоящая из множества *объектов* $G$, множества *атрибутов* $M$ и бинарного отношения $\rho \subset G \times M$, показывающего $(g, m) \in \rho$, что объект $g$ имеет атрибут $m$.

**Определение.** Упорядоченная пара $(X, Y)$ замкнутых множеств $X \in Z_{f_G}$, $Y \in Z_{f_M}$ (где $Z_{f_G}$ и $Z_{f_M}$ - системы замыканий множеств $G$ и $M$), удовлетворяющих условиям $\varphi(X) = Y$, $\psi(Y) = X$, называется *концептом* контекста $K = (G, M, \rho)$. При этом компонента $X$ называется *объёмом* и компонента $Y$ - *содержанием* концепта $(X, Y)$.

Также для составления алгоритма вычисления решётки концептов нам понадобится **алгоритм вычисления системы замыканий** на множестве $G$:

1. Рассматриваем множество $G \in Z_{f_G}$.

2. Последовательно перебираем все элементы $m \in M$ и вычисляем для них $\psi(\{m\}) = \rho^{-1}(m)$.

3. Вычисляем все новые пересечения множества $\psi(\{m\})$ с ранее полученными множествами и добавляем новые множества к $Z_{f_G}$. Аналогично вычисляется система замыканий на множестве $M$.

## 3 Результаты работы

### 3.1 Алгоритм построения эквивалентного замыкания бинарного отношения

**Описание алгоритма построения эквивалентного замыкания бинарного отношения.**

**Вход:** матрица бинарного отношения $\rho$.

**Выход:** матрица замкнутого относительно свойств рефлексивности, симметричности, транзитивности (т.е. отношение эквивалентности) бинарного отношения $\rho$.

**Метод:** к бинарному отношению $\rho$ поочерёдно применить операции рефлексивного, затем симметричного, затем транзитивного замыканий.

**Псевдокод алгоритма построения эквивалентного замыкания бинарного отношения.**

```
equivalence_closure(<matrix> binaryRelation)
{
  return transitive_closure(
            symmetric_closure(
              reflexive_closure(binaryRelation)));
}
```

Листинг 1: Псевдокод алгоритма.

Функции построения рефлексивного, симметричного и транзитивного замыкания были описаны в предыдущей лабораторной работе.

**Код программы, реализующей алгоритм построения эквивалентного замыкания бинарного отношения.**

```
map<int, set<int>>
  equivalence_closure(map<int, set<int>> binaryRelation)
{
  return transitive_closure(
            symmetric_closure(
              reflexive_closure(binaryRelation)));
}
```

Листинг 2: Код программы.

**Результат тестирования программы построения эквивалентного замыкания бинарного отношения.**

Для демонстрации работы программы рассмотрим произвольное бинарное отношение $\delta$.

Как видно, исходное бинарное отношение не является отношением эквивалентности. Построим эквивалентное замыкание этого бинарного отношения:

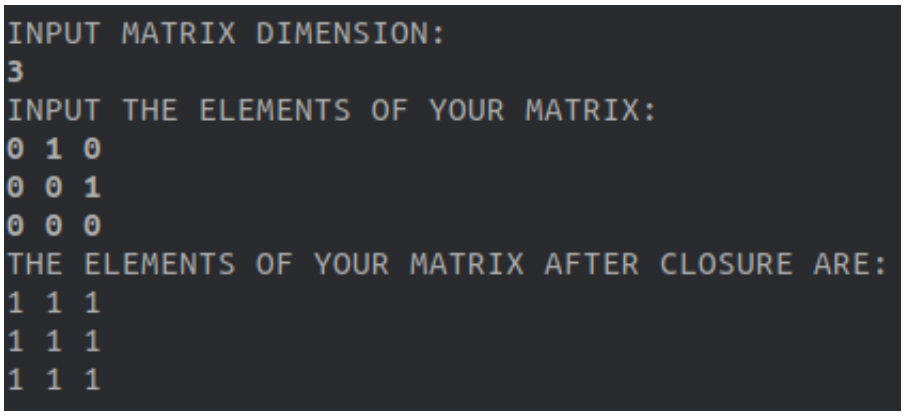


Рисунок 1 – Матрица бинарного отношения $\delta$ после эквивалентного замыкания.

Получившееся бинарное отношение $\delta$ является замкнутым относительно свойств рефлексивности, симметричности и транзитивности, что видно по матрице этого бинарного отношения.

**Оценка временной сложности алгоритма эквивалентного замыкания бинарного отношения.**

Поскольку алгоритм эквивалентного замыкания использует функции, реализованные в прошлой лабораторной работе, то воспользуемся уже полученными оценками: $T(reflexive\_closure,\ n) = O(n \log n),\ T(symmetric\_closure,\ n) = O(n^2 \log n),\ T(transitive\_closure, n) = O(n^4 \log n)$. Операции замыкания применяются последовательно, значит, $T(equivalence\_closure,\ n) = O(n \log n + n^2 \log n + n^4 \log n) = O(n^4 \log n)$.

### 3.2 Алгоритм нахождения системы представителей по заданному бинарному отношению

**Описание алгоритма нахождения системы представителей по заданному бинарному отношению.**

**Вход:** матрица бинарного отношения $\rho$.

**Выход:** система представителей заданного бинарного отношения $\rho$.

**Метод:** для получения системы представителей бинарного отношения $\rho$ оно должно являться эквивалентностью. Поэтому сначала бинарное отношение $\rho$ подвергается эквивалентному замыканию. Затем по замкнутому отношению $\rho$ строится фактор-множество этого отношения. Из каждого класса эквивалентности в построенном фактор-множестве выбирается свой представитель.

**Псевдокод алгоритма нахождения системы представителей по заданному бинарному отношению.**

```
get_system_of_representatives(<matrix> binaryRelation)
{
  systemOfRepresentatives[], factorSet[[]];
  factorSet[[]] = get_equivalence_classes(binaryRelation);
  for i in range(factorSet.size())
    systemOfRepresentatives.push_back(
                              getRepresentative(factorSet[i]));

  return systemOfRepresentatives;
}
```

Листинг 3: Псевдокод алгоритма.

Стоит заметить, что понятию класса эквивалентности в ориентированном графе соответствует понятие компоненты сильной связности. Для поиска компонент сильной связности по заданной матрице бинарного отношения используются вспомогательные функции *get_equivalence_classes*, *get_component_DFS* и *order_DFS*.

**Код программы, реализующей алгоритм нахождения системы представителей по заданному бинарному отношению.**

```
void orderDFS(vector<vector<int>> adjVector, int vertexNum)
{
  visited[vertexNum] = true;
```

```
4
5   for (int i = 0; i < adjVector[vertexNum].size(); ++i)
6     if (!visited[adjVector[vertexNum][i]])
7       orderDFS(adjVector, adjVector[vertexNum][i]);
8
9   exitOrder.push_back(vertexNum);
10 }
11
12 void getComponentDFS(vector<vector<int>> adjMatrixTransposed,
13                      int vertexNum)
14 {
15   visited[vertexNum] = true;
16   component.push_back(vertexNum);
17
18   for (int i = 0; i < adjMatrixTransposed[vertexNum].size(); ++i)
19     if (!visited[adjMatrixTransposed[vertexNum][i]])
20       getComponentDFS(adjMatrixTransposed,
21                       adjMatrixTransposed[vertexNum][i]);
22 }
23
24 vector<vector<int>> getEquivalenceClasses(
25   vector<vector<int>> adjMatrix)
26 {
27   int i;
28   vector<vector<int>> adjVectorTransposed, connectedComponents;
29   vector<vector<int>> adjVector =
30     convertMatrixToAdjVector (adjMatrix);
31   visited.assign(matrixDimension, false);
32
33   for (i = 0; i < matrixDimension; ++i)
34     if (!visited[i])
35       orderDFS(adjVector, i);
36
37   adjVectorTransposed =
38     convertMatrixToAdjVector (transposeMatrix (adjMatrix));
39   visited.assign (matrixDimension, false);
40   reverse (exitOrder.begin (), exitOrder.end ());
41
42   for (i = 0; i < matrixDimension; ++i)
43   {
44     if (!visited[exitOrder[i]])
```

```
45      {
46        getComponentDFS (adjVectorTransposed, exitOrder[i]);
47        connectedComponents.push_back (component);
48        component.clear ();
49      }
50    }
51    return (connectedComponents);
52  }
53
54  void getSystemOfRepresentatives (vector<vector<int>> adjMatrix)
55  {
56    int i, j;
57    vector<int> systemOfRepresentatives;
58    vector<vector<int>> connectedComponents =
59      getEquivalenceClasses (adjMatrix);
60
61    cout << "FACTOR SET OF THIS BINARY RELATION IS:\n";
62
63    for (i = 0; i < connectedComponents.size (); ++i)
64    {
65      cout << "{ ";
66      for (j = 0; j < connectedComponents[i].size(); ++j)
67        cout << connectedComponents[i][j] + 1 <<
68          (j == connectedComponents[i].size() - 1 ? " " : ", ");
69      cout << "}" <<
70        (i == connectedComponents.size() - 1 ? "." : ", ");
71    }
72
73    for (i = 0; i < connectedComponents.size(); ++i)
74      systemOfRepresentatives.push_back(
75        getMin (connectedComponents[i]));
76
77    cout <<
78      "\nSYSTEM OF REPRESENTATIVES OF THIS BINARY RELATION IS:\n";
79
80    for (i = 0; i < systemOfRepresentatives.size(); ++i)
81      cout << "{ " << systemOfRepresentatives[i] + 1 << " }" <<
82        (i == systemOfRepresentatives.size() - 1 ? "." : ", ");
83  }
```

Листинг 4: Код программы и вспомогательных функций.

**Результат тестирования программы нахождения системы представителей по заданной матрице бинарного отношения.**

Для демонстрации работы программы рассмотрим бинарное отношение $\delta$, заданное матрицей.

Для заданного бинарного отношения $\delta$ получим систему представителей:

```
INPUT MATRIX DIMENSION:
5
INPUT THE ELEMENTS OF YOUR MATRIX:
0 1 0 0 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 1
0 0 0 1 0
THE ELEMENTS OF YOUR MATRIX AFTER CLOSURE ARE:
1 1 1 0 0
1 1 1 0 0
1 1 1 0 0
0 0 0 1 1
0 0 0 1 1

FACTOR SET OF THIS BINARY RELATION IS:
{ 4, 5 }, { 1, 2, 3 }.
SYSTEM OF REPRESENTATIVES OF THIS BINARY RELATION IS:
{ 4 }, { 1 }.
```

Рисунок 2 – Система представителей бинарного отношения $\delta$.

Программа сначала осуществила эквивалентное замыкание заданного бинарного отношения $\delta$, затем построила фактор-множество и выбрала представителя из каждого класса эквивалентности.

**Оценка временной сложности алгоритма построения системы представителей заданного бинарного отношения.**

Рассмотрим алгоритм выделения компонент сильной связности, при помощи которого происходит построение фактор-множества:

1. Запустить серию обходов в глубину графа G, которая возвращает вершины в порядке увеличения времени выхода **tout**, т.е. некоторый список **order**.
2. Построить транспонированный граф $G^T$. Запустить серию обходов в глубину этого графа в порядке, определяемом списком **order** (а именно, в обратном порядке, т.е. в порядке уменьшения времени выхода). Каждое множество вершин, достигнутое в результате очередного запуска обхода, и будет очередной компонентой сильной связности.

Временная сложность такого алгоритма равна $O(m + n)$, поскольку алгоритм представляет собой два обхода в глубину. Однако после получения списка **order**, необходимо построить транспонированный граф $G^T$, что увеличивает временную сложность алгоритма до $O(n^2)$ (если предварительно не сохранять транспонированную матрицу бинарного отношения).

### 3.3 Алгоритм построения диаграммы Хассе по заданному числу

**Описание алгоритма построения диаграммы Хассе по заданному числу.**

**Вход:** натуральное число $n$ и два символа 'y'/'n', учитывающих само число $n$ и 1 в качестве делителей.

**Выход:** диаграмма Хассе отношения делимости для делителей заданного числа $n$, а также минимальные, максимальные и наименьшие, наибольшие элементы полученного отношения порядка.

**Метод:** сначала необходимо получить все делители рассматриваемого числа $n$ (с учётом или без учёта 1 и $n$). В множестве делителей найти минимальные элементы и расположить их на первом уровне диаграммы Хассе. Затем искать минимальные элементы в множестве делителей без множества элементов, расположенных на первом уровне диаграммы Хассе (в последующих случаях, – исключая элементы, расположенные на предыдущих уровнях диаграммы) и расположить их на втором уровне диаграммы, сохранить связи, указывающие на отношения делимости между уровнями диаграммы. Повторять процедуру, пока множество делителей не будет пусто. Для определения того, какие элементы являются минимальными, максимальными и наименьшими, наибольшими, просматриваются самый верхний и самый нижний уровень диаграммы Хассе. Если только один элемент содержится на нижнем уровне диаграммы Хассе, то он будет и минимальным, и наименьшим. Если же на нижнем уровне содержится более одного элемента, то наименьший элемент отсутствует, а минимальными будут являться все элементы, расположенные на нижнем уровне. Аналогичные рассуждения проводятся для верхнего уровня и для максимального и наибольшего элемента.

**Псевдокод алгоритма построения диаграммы Хассе по заданному числу.**

```
1 division_Hasse_diagram(natural_number)
2 {
3   divisors[] = get_divisors(natural_number);
4   currentMins[], HasseDiagram[[]];
5
6   while (!divisors.empty())
7   {
```

```
    currentMins = get_mins(divisors);
    HasseDiagram.push_back(currentMins);
    remove_elements(divisors, currentMins);
  }
  make_diagram_connections(HasseDiagram);
}
```

Листинг 5: Псевдокод алгоритма.

В качестве вспомогательных используются функции *getDivisors*, возвращающая список делителей заданного числа $n$, *getMins*, возвращающая список минимальных (в соответствии с отношением делимости) делителей числа $n$ на текущей итерации, *removeMins*, удаляющая из множества делителей множество минимальных делителей, полученных на текущей итерации, а также *getHasseConnections*, формирующая связи между уровнями диаграммы Хассе.

**Код программы построения диаграммы Хассе по заданному числу.**

```
vector<int> getDivisors(int numberX)
{
  int i, numberOfDivisors = 0;
  char includeOne = '.', includeX = '.';
  vector<int> divisors;

  for (i = 2; i * i < numberX; ++i)
    if (numberX % i == 0)
      ++numberOfDivisors;

  while (includeOne != 'y' && includeOne != 'n')
  {
    cout << "INCLUDE ONE ? (y/n):\n";
    cin >> includeOne;
  }
  if (includeOne == 'y')
    divisors.push_back(1);
  while (includeX != 'y' && includeX != 'n')
  {
    cout << "INCLUDE " << numberX << " ? (y/n):\n";
    cin >> includeX;
  }
  if (includeX == 'y')
```

```
24      divisors.push_back(numberX);
25    if (numberOfDivisors == 0 &&
26        includeOne == 'n' &&
27        includeX == 'n')
28    {
29      cout << "YOU CAN'T REMOVE THE NUMBER ITSELF
30        AND NUMBER ONE FOR A PRIME NUMBER.";
31      exit(0);
32    }
33
34    for (i = 2; i <= numberX / 2; ++i)
35      if (numberX % i == 0)
36        divisors.push_back(i);
37
38    return (divisors);
39  }
40
41  vector<int> getMins(vector<int> divisors)
42  {
43    int i;
44    map<int, vector<int>> divisibilityRelation;
45    vector<int> mins;
46
47    for (int divisor : divisors)
48    {
49      vector<int> placeholder;
50      divisibilityRelation[divisor] = placeholder;
51    }
52    for (int divisor : divisors)
53    {
54      for (int divisorToCheck : divisors)
55        if (divisor != divisorToCheck &&
56            divisorToCheck % divisor == 0)
57          divisibilityRelation[divisor].push_back(divisorToCheck);
58    }
59    for (int divisor : divisors)
60    {
61      bool found = false;
62      for (auto divisElements : divisibilityRelation)
63        for (i = 0; i < divisElements.second.size(); ++i)
64        {
```

```
65          if (divisor == divisElements.second[i])
66          {
67            found = true;
68            break;
69          }
70          if (found)
71            break;
72        }
73      if (!found)
74        mins.push_back(divisor);
75    }
76    return (mins);
77  }
78
79  void removeMins (vector<int>& divisors, vector<int> mins)
80  {
81    int i;
82
83    for (int min : mins)
84    {
85      auto it = divisors.cbegin();
86
87      for (i = 0; i < divisors.size(); ++i)
88        if (divisors[i] == min)
89          divisors.erase (it + i);
90    }
91  }
92
93  void getHasseConnectionsMachinerie(
94    vector<vector<pair<int, int>>>& HasseLevelConnections,
95    vector<int> lowerLevel, vector<int> upperLevel)
96  {
97    int i;
98    vector<pair<int, int>> levelRelations;
99
100   for (int lowerLevelElt : lowerLevel)
101   {
102     for (int upperLevelElt : upperLevel)
103       if (upperLevelElt % lowerLevelElt == 0)
104         levelRelations.push_back(
105           make_pair(lowerLevelElt, upperLevelElt));
```

```
106   }
107   HasseLevelConnections.push_back(levelRelations);
108 }
109
110 vector<vector<pair<int, int>>> getHasseConnections(
111   vector<vector<int>> HasseLevels)
112 {
113   int i;
114   vector<vector<pair<int, int>>> HasseLevelConnections;
115
116   for (i = 0; i < HasseLevels.size() - 1; ++i)
117     getHasseConnectionsMachinerie(
118       HasseLevelConnections,
119       HasseLevels[i], HasseLevels[i + 1]);
120
121   return (HasseLevelConnections);
122 }
123
124 void HasseDiagram ()
125 {
126   int numberX, i;
127   vector<int> divisors, mins;
128   vector<vector<int>> HasseLevels;
129   vector<vector<pair<int, int>>> HasseLevelConnections;
130
131   cout << "INPUT YOU NUMBER:\n";
132   cin >> numberX;
133
134   divisors = getDivisors (numberX);
135
136   while (!divisors.empty ())
137   {
138     mins = getMins (divisors);
139     HasseLevels.push_back (mins);
140     removeMins (divisors, mins);
141   }
142   HasseLevelConnections = getHasseConnections (HasseLevels);
143   printHasseDiagram (HasseLevels, HasseLevelConnections);
144 }
```

Листинг 6: Код программы.

**Результат тестирования программы нахождения диаграммы Хассе по заданному числу.**

Для демонстрации работы программы рассмотрим её выход для заданного числа 60. Само число 60 не будем включать в множество делителей.

```
INPUT YOU NUMBER:
60
INCLUDE ONE ? (y/n):
y
INCLUDE 60 ? (y/n):
n
12 20 30
4 6 10 15
2 3 5
1
(4, 12), (4, 20), (6, 12), (6, 30), (10, 20), (10, 30), (15, 30).
(2, 4), (2, 6), (2, 10), (3, 6), (3, 15), (5, 10), (5, 15).
(1, 2), (1, 3), (1, 5).
LEAST ELEMENT: 1.
MINIMAL ELEMENT: 1.
GREATEST ELEMENT: NONE.
MAXIMAL ELEMENTS: 12, 20, 30.
```

Рисунок 3 – Построение диаграммы Хассе отношения делимости для делителей числа 60 (без учёта самого числа 60).

В качестве выхода работы программы была сформирована диаграмма Хассе отношения делимости на множестве делителей числа 60 (без учёта самого числа 60). Ниже идут пары вида $(x, y)$, указывающие на связь элементов между уровнями диаграммы. Также, поскольку само число 60 не рассматривалось как делитель, то наибольший элемент отсутствует, в то время как максимальными элементами являются элементы, расположенные на верхнем уровне диаграммы.

**Оценка временной сложности алгоритма построения диаграммы Хассе по заданному числу.**

Сначала в соответствии с псевдокодом рассмотрим операцию получения всех делителей заданного числа $n$. Для числа $n$ необходимо проверить, делят ли его все числа от 2 до $n/2$ (рассмотрение 1 и $n$ зависит от ввода пользователя). Таким образом, операция получения делителей числа $n$ имеет сложность: $O(n/2) = O(n)$.

Перейдём к рассмотрению содержимого цикла *while*. Первая операция – получение минимальных элементов относительно отношения делимости в

текущем множестве делителей. В нашем случае для каждого делителя $d_i$ из множества делителей $D$ числа $n$ ищется элемент $d_j$ из множества $D$ такой, что $d_j$ делит $d_i$. Если такой элемент $d_j$ для элемента $d_i$ не найден, то $d_i$ признаётся минимальным в текущем множестве делителей. Поскольку поиск такого делителя $d_j$ для $d_i$ осуществляется проходом по текущему множеству делителей, то $T(get\_mins, n) = O(|D|^2)$, где $|D|$ – мощность множества делителей заданного числа $n$.

Следующая операция – удаление найденных на текущей итерации минимальных элементов из множества делителей. Достигается путём поиска соответствующих элементов в множестве делителей и их удалении оттуда. Сложность операции: $O(|D|^2)$.

В худшем случае цикл *while* будет выполняться $|D|$ раз – в случае, если диаграмма Хассе представляет собой цепь. Тогда временная сложность содержимого цикла составляет: $T(while(), n) = O(|D|(|D|^2 + |D|^2)) = O(|D|^3)$, где $|D|$ – мощность множества делителей заданного числа $n$.

Последняя операция (функция *makeDiagramConnections*) отвечает за процесс формирования связей между уровнями диаграммы Хассе. Операция выполняется следующим образом: между каждыми двумя смежными уровнями диаграммы Хассе формируются связи, соответствующие отношению делимости между элементами этих уровней. Для каждого элемента нижнего уровня просматриваются все элементы верхнего уровня и выполняется проверка на то, делит ли элемент нижнего уровня элемент верхнего уровня. Если да, то факт наличия связи между двумя рассматриваемыми элементами сохраняется в контейнере связей.

Получим оценку временной сложности этой операции. Будем считать, что 1 и $n$ учитывается в множестве делителей. Предположим, что ранее была сформирована диаграмма Хассе с $p$ уровнями. Через $n_i$ $(i = 1, \ldots, p)$ обозначим число элементов на каждом уровне диаграммы Хассе. Очевидно, что $\sum_{i=1}^{p} n_i = |D|$, где $|D|$ – мощность множества делителей заданного числа $n$. Тогда общее число операций, совершаемое функцией *makeDiagramConnections* составляет $\sum_{i=1}^{p-1} n_i n_{i+1}$. Определим $N = \max n_i$, $i = 1, \ldots, p$. Заметим, что $n_i \leq N \leq |D|$ и $p \leq |D|$. Оценим число операций функции *makeDiagramConnections*

сверху:

$$\sum_{i=1}^{p-1} n_i n_{i+1} \leq \sum_{i=1}^{p-1} N^2 = N^2 \sum_{i=1}^{p-1} 1 \leq N^2(p-1) \leq |D|^3.$$

Тогда, общая временная оценка алгоритма построения диаграммы Хассе по заданному числу: $T(Hasse\_diagram, n) = O(n+|D|^3+|D|^3) = O(n+|D|^3)$, где $|D|$ – мощность множества числа делителей заданного числа n.

### 3.4 Алгоритм построения диаграммы Хассе по заданной матрице порядка

**Описание алгоритма построения диаграммы Хассе по заданной матрице порядка.**

**Вход:** матрица порядка бинарного отношения $\rho$.

**Выход:** диаграмма Хассе заданного бинарного отношения порядка $\rho$.

**Метод:** сначала формируется контейнер, содержащий элементы заданного отношения порядка. Затем, пока этот контейнер не пуст, осуществляется поиск минимальных элементов отношения порядка по заданной матрице. В матрице минимальному элементу отношения порядка соответствует такой элемент, что его столбец не содержит единиц, кроме как на позиции, где $i$ равен $j$ (номер строки совпадает с номером столбца – в силу рефлексивности отношения порядка). Найденные на первой итерации минимальные элементы составляют первый уровень диаграммы Хассе. После, эти минимальные элементы должны быть убраны из контейнера и поиск минимальных элементов теперь осуществляется в обновлённом контейнере. Найденные на второй итерации минимальные элементы будут составлять второй уровень диаграммы Хассе и т.д.. После составления диаграммы Хассе необходимо сформировать связи между элементами уровней диаграммы. Для этого рассматриваются каждые два смежных уровня получившейся диаграммы Хассе. Для каждого элемента нижнего уровня в матрице порядка ищутся элементы верхнего уровня, с которыми он состоит в отношении порядка. Соответствующие связи добавляются в контейнер связей.

**Псевдокод алгоритма построения диаграммы Хассе по заданной матрице порядка.**

```
1  Hasse_diagram_matrix(<matrix> orderBinaryRelation)
2  {
3    eltContainer[] = get_elements(orderBinaryRelation);
4    HasseDiagram[[]], mins[];
5
6    while (!eltContainer.empty())
7    {
8      mins = get_min_elts(matrix, eltContainer);
9      HasseDiagram.push_back(mins);
10     remove_mins (orderBinaryRelation, eltContainer, mins);
11   }
```

```
  HasseDiagram = getHasseConnectionsMatrix(
                    HasseDiagram, orderBinaryRelation);
}
```

Листинг 7: Псевдокод алгоритма.

В качестве вспомогательных используются функции *getMinsMatrix*, извлекающая на текущей итерации минимальные элементы отношения порядка, *removeMinElts*, удаляющая из контейнера *elements* минимальные элементы, найденные на текущей итерации, *cleanRows*, выполняющая аналогичную функцию, но для матрицы порядка, и *getHasseConnectionsMatrix*, выполняющая построение связей между уровнями диаграммы Хассе.

**Код программы построения диаграммы Хассе по заданной матрице порядка.**

```
void cleanRows (vector<vector<int>>& matrix, vector<int> mins)
{
  int i;

  for (int min : mins)
    for (i = 0; i < matrixDimension; ++i)
      if (matrix[min][i] == 1)
        matrix[min][i] = 0;
}

void removeMinElts (vector<int>& elts, vector<int> mins)
{
  int i;

  for (int min : mins)
  {
    auto it = elts.cbegin ();
    for (i = 0; i < elts.size (); ++i)
      if (elts[i] == min)
        elts.erase (it + i);
  }
}

bool inElements (vector<int> elements, int i)
{
```

```
26    int j;
27
28    for (j = 0; j < elements.size (); ++j)
29      if (elements[j] == i)
30        return (true);
31
32    return (false);
33  }
34
35  vector<int> getMinsMatrix (vector<vector<int>> matrix,
36                             vector<int> elements)
37  {
38    int i, j;
39    vector<int> mins;
40
41    for (i = 0; i < matrixDimension; ++i)
42    {
43      bool isMin = true;
44
45      for (j = 0; j < matrixDimension; ++j)
46        if (matrix[j][i] == 1 && i != j)
47        {
48          isMin = false;
49          break;
50        }
51      if (isMin && inElements (elements, i))
52        mins.push_back (i);
53    }
54    return (mins);
55  }
56
57  void getHasseConnectionsMatrix (
58    vector<vector<int>> matrix,
59    vector<int> pLevel, vector<int> nLevel,
60    vector<vector<pair<int, int>>>& levelsConnections)
61  {
62    int i, j;
63    vector<pair<int, int>> levelConnections;
64
65    for (int pLvlElt : pLevel)
66      for (int nLvlElt : nLevel)
```

```
67        if (matrix[pLvlElt][nLvlElt] == 1)
68          levelConnections.push_back (
69            make_pair (pLvlElt + 1, nLvlElt + 1));
70
71    levelsConnections.push_back (levelConnections);
72  }
73
74  void HasseDiagramMatrix ()
75  {
76    int i;
77    vector<vector<int>> matrix, matrixCopy, HasseLevels;
78    vector<vector<pair<int, int>>> HasseLevelConnections;
79    vector<int> elements, mins;
80    cout << "INPUT MATRIX DIMENSION:\n";
81    cin >> matrixDimension;
82
83    getMatrix (matrix);
84
85    matrixCopy = matrix;
86
87    for (i = 0; i < matrix.size (); ++i)
88      elements.push_back (i);
89
90    while (!elements.empty ())
91    {
92      mins = getMinsMatrix (matrix, elements);
93      HasseLevels.push_back (mins);
94      removeMinElts (elements, mins);
95      cleanRows (matrix, mins);
96    }
97
98    for (i = 0; i < HasseLevels.size () - 1; ++i)
99      getHasseConnectionsMatrix (
100       matrixCopy, HasseLevels[i],
101       HasseLevels[i + 1], HasseLevelConnections);
102
103   displayLevelsAndConnections (HasseLevels,
104                                HasseLevelConnections);
105 }
```

Листинг 8: Код программы.

**Результат тестирования программы построения диаграммы Хассе по заданной матрице порядка.**

Рассмотрим отношение порядка $\delta$, заданное матрицей. Построим диаграмму Хассе для рассматриваемого отношения порядка:

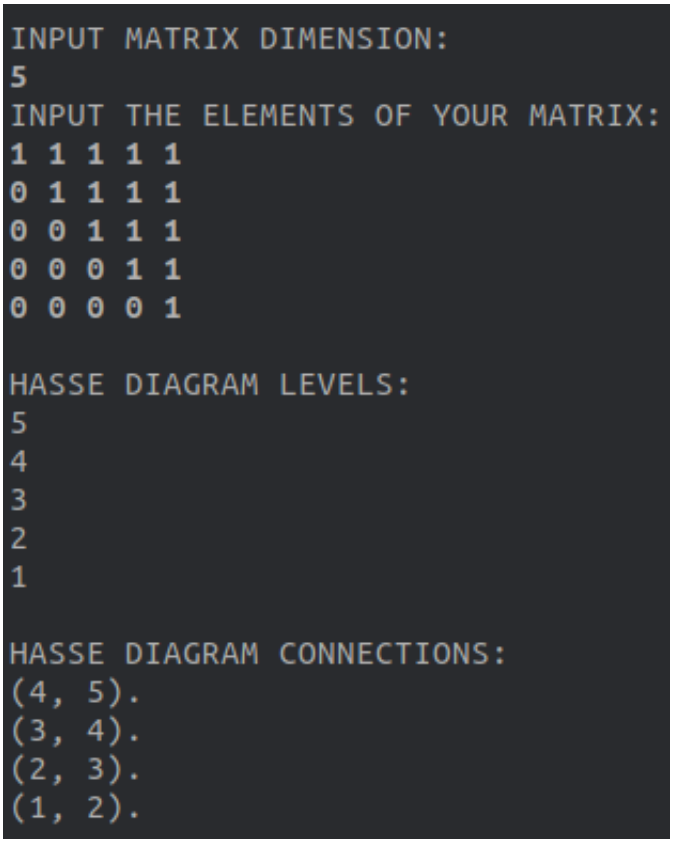


Рисунок 4 – Диаграмма Хассе заданного отношения порядка $\delta$.

Как видно, построенная диаграмма Хассе для заданного отношения порядка $\delta$ является *цепью*.

**Оценка временной сложности алгоритма построения диаграммы Хассе по заданной матрице порядка.**

В соответствии с псевдокодом будем последовательно рассматривать сложность каждой из выполняемых операций.

Первая операция – формирование контейнера элементов. Сложность операции: $O(n)$, т.к. должен быть учтён каждый уникальный элемент, участвующий в отношении порядка, где $n$ – размерность входной квадратной матрицы порядка.

Далее будем рассматривать операции, выполняемые внутри цикла *while*. Первая операция внутри цикла – поиск минимальных элементов отношения

порядка на текущей итерации. В качестве примера рассмотрим число совершаемых операций на первой итерации, т.е. во время поиска первых минимальных элементов отношения порядка. Т.к. для определения того, является ли элемент минимальным, необходимо полностью просмотреть его столбец в матрице порядка, то для нахождения всех минимальных элементов на первой итерации необходимо осуществить $n^2$ операций. На следующих итерациях, когда минимальные элементы предыдущих итераций рассматривать уже не надо, число операций будет уменьшаться, однако верхняя граница в $n^2$ операций (где $n^2$ – число элементов в исходной матрице порядка) по-прежнему будет верна. Таким образом, $T(getMinsMatrix, n) = O(n^2)$.

Вторая операция внутри цикла – удаление минимальных элементов из контейнера и обновление значений в матрице. Для того, чтобы удалить минимальное значение из контейнера, его сначала необходимо найти. После нахождения нужного значения оно удаляется. Верхняя граница сложности такой операции: $O(n^2)$, поскольку на первой итерации все элементы могут оказаться минимальными, и придётся осуществлять поиск каждого элемента в контейнере. В случае с матрицей строка элемента, являющегося минимальным на текущей итерации заполняется нулями. Таким образом, верхняя граница сложности такой операции: $O(n^2)$.

Поскольку на каждом шаге выполнения алгоритма находится как минимум один минимальный элемент (при корректных входных данных), то условие *while (!eltContainer.empty())* может быть выполнено максимум $n$ раз, где $n$ – число строк или столбцов в исходной квадратной матрице порядка. Таким образом верхняя граница сложности рассматриваемого цикла: $T(while(), n) = O(n^3)$.

После выполнения цикла необходимо сформировать связи между уровнями диаграммы Хассе. Для этого рассматриваются каждые два смежных уровня в диаграмме Хассе. Для каждого элемента нижнего уровня рассматривается каждый элемент верхнего уровня и ищется значение в матрице (в копии матрицы, т.к. после построения диаграммы Хассе исходная матрица порядка – нулевая), стоящее на пересечении соответствующих строки и столбца. Если значение – единица, то в контейнер связей добавляется отношение между соответствующими элементами (пара этих элементов).

Допустим, что после построения диаграмма Хассе содержит $p$ уровней. Через $n_i$ $(i = 1, \ldots, p)$ обозначим число элементов на каждом уровне. Очевид-

но, что $\sum_{i=1}^{p} n_i = n$, где $n$ – число строк или столбцов в исходной квадратной матрице порядка. Тогда число операций, которое необходимо выполнить для формирования связей между уровнями диаграммы Хассе в соответствии с описанием процедуры: $\sum_{i=1}^{p-1} n_i n_{i+1}$. Обозначим $N = \max n_i \, (i = 1, \ldots, p)$. Также видно, что $n_i \leq N \leq n$ и $p \leq n$. Тогда:

$$\sum_{i=1}^{p-1} n_i n_{i+1} \leq N^2 \sum_{i=1}^{p-1} 1 = N^2(p-1) \leq n^3$$

Таким образом, временная сложность алгоритма построения диаграммы Хассе по заданной матрице порядка составляет: $T(Hasse\_diagram\_matrix, n) = O(n + n(n^2 + n^2) + n^3) = O(n^3)$.

### 3.5 Алгоритм построения решётки концептов

**Описание алгоритма построения решётки концептов.**

**Вход:** контекст $K$ (множество объектов $G$, множество атрибутов $M$, бинарное отношение $\rho \subset G \times M$).

**Выход:** решётка концептов $C(K)$ и её диаграмма Хассе.

**Метод:** сначала вычисляется система замыканий $Z_{f_G}$ на множестве $G$ и строится её диаграмма Хассе. Затем составляется решётка концептов $C(K)$, формируются связи между уровнями диаграммы Хассе. Полученная диаграмма Хассе дополняется элементами из решётки концептов. Система замыканий $Z_{f_G}$ строится путём вычисления $\rho^{-1}(a_i)$, $a_i \in M$ $(i = 1, \ldots, |M|)$, добавления его в $Z_{f_G}$ и попарного пересечения получившегося множества $\rho^{-1}(a_i)$ с уже имеющимися в $Z_{f_G}$ множествами и их последующего добавления в $Z_{f_G}$. Диаграмма Хассе системы замыканий $Z_{f_G}$ строится на основе отношения теоретико-множественного включения. Для поиска минимальных элементов в $Z_{f_G}$ для каждого элемента (множества) из $Z_{f_G}$ на каждой итерации алгоритма построения диаграммы Хассе будем хранить набор множеств из $Z_{f_G}$, содержащихся в рассматриваемом множестве. Таким образом, если рассматриваемое множество не содержит в себе ни одного элемента (множества) из $Z_{f_G}$, кроме самого себя, то оно является минимальным элементом на текущей итерации построения диаграммы Хассе. Минимальные элементы, полученные на первой итерации алгоритма построения диаграммы Хассе представляют собой первый уровень диаграммы. Для построения второго уровня диаграммы из системы замыканий $Z_{f_G}$ необходимо удалить минимальные элементы, найденные на первой итерации алгоритма построения диаграммы Хассе и повторить процедуру построения содержащихся наборов множеств. Алгоритм будет выполняться и формировать новые уровни диаграммы Хассе, пока $Z_{f_G}$ не станет пустым. После построения диаграммы связи между уровнями формируются следующим образом: для каждых двух смежных уровней диаграммы для каждого элемента (множества) нижнего уровня проверяется факт его включения в элементы (множества) верхнего уровня. Пара элементов (множеств), для которых это условие выполняется, заносится в контейнер связей. После построения диаграммы Хассе и формирования её связей, в дополнение к каждому элементу (множеству объектов) каждого уровня диаграммы ставится в соответствие элемент (множество атрибутов) решётки концептов – набор общих атрибутов для соответствующего множества объектов.

**Псевдокод алгоритма построения решётки концептов.**

```
get_concept_grid(objectSet[], attrSet[], <matrix> binaryRelation)
{
  mins[[]], HasseDiagram[[[]]], Z_fG[[]];
  Z_fG = get_closure_system(objectSet, attrSet, binaryRelation);

  while (!Z_fG.empty())
  {
    mins = get_mins(construct_subsets(Z_fG));
    HasseDiagram.push_back(mins);
    remove_mins(Z_fG, mins);
  }
  HasseDiagram = get_Hasse_Connections(HasseDiagram);
  HasseDiagram = complete_with_attributes(HasseDiagram);
}
```

Листинг 9: Псевдокод алгоритма.

В качестве вспомогательных используются функции *getInverse* и *getIntersects*, осуществляющие вычисление системы замыканий $Z_{f_G}$, *getHasseDiagramConceptGrid*, осуществляющая построение диаграммы Хассе по полученной системе замыканий $Z_{f_G}$, *getSubsetRelation*, которая осуществляет идею построения набора множеств из $Z_{f_G}$, содержащихся в некотором множестве из $Z_{f_G}$ для каждого множества из $Z_{f_G}$, *removeMinsConcept*, осуществляющая удаление минимальных элементов, найденных на текущей итерации, из $Z_{f_G}$, *getHasseConnectionsConcept*, формирующая связи между элементами смежных уровней диаграммы Хассе, *HasseAttributeSet*, ставящая в дополнение к каждому элементу (множеству объектов) каждого уровня диаграммы Хассе соответствующий элемент (множество атрибутов) решётки концептов.

**Код программы построения решётки концептов.**

```
void getIntersects (vector<vector<int>>& setGrid)
{
  int i, j;

  for (i = 1; i < setGrid.size (); ++i)
  {
    for (j = 1; j < setGrid.size (); ++j)
```

```
8       {
9         vector<int> intersection = intersect (setGrid[i],
10                                              setGrid[j]);
11
12        if (i != j && !inGrid (setGrid, intersection))
13          setGrid.push_back (intersection);
14      }
15    }
16  }
17
18  void getSubsetRelation (
19    map<vector<int>, vector<vector<int>>>& subsetRelation,
20    vector<vector<int>> setGrid)
21  {
22    int i, j;
23
24    for (auto setAndSubsets : subsetRelation)
25    {
26      vector<vector<int>> subsets;
27
28      for (j = 0; j < setGrid.size (); ++j)
29        if (isSubset (setAndSubsets.first, setGrid[j]) &&
30            setAndSubsets.first != setGrid[j])
31          subsets.push_back (setGrid[j]);
32      subsetRelation[setAndSubsets.first] = subsets;
33    }
34  }
35
36  vector<vector<int>> getMinsConcept (
37    map<vector<int>, vector<vector<int>>> subsetRelation)
38  {
39    int i;
40    vector<vector<int>> mins;
41
42    for (auto setAndSubsets : subsetRelation)
43    {
44      vector<int> set = setAndSubsets.first;
45
46      if (subsetRelation[set].empty ())
47        mins.push_back (set);
48    }
```

```
49
50    return (mins);
51  }
52
53  void removeMinsConcept (
54    map<vector<int>, vector<vector<int>>>& subsetRelation,
55    vector<vector<int>>& setGrid, vector<vector<int>> mins)
56  {
57    int i;
58
59    for (vector<int> min : mins)
60    {
61      auto it = setGrid.cbegin ();
62
63      for (i = 0; i < setGrid.size (); ++i)
64        if (setGrid[i] == min)
65          setGrid.erase (it + i);
66    }
67    for (i = 0; i < mins.size (); ++i)
68      subsetRelation.erase (subsetRelation.find (mins[i]));
69  }
70
71  vector<char> getAttributeSetConceptMachinerie (
72    int objectNum, vector<vector<int>> matrix)
73  {
74    int i;
75    vector<char> objectAttributes;
76
77    for (i = 0; i < matrix.size (); ++i)
78      if (matrix[objectNum][i] == 1)
79        objectAttributes.push_back (attributeSet[i]);
80
81    if (objectAttributes.empty ())
82      objectAttributes.push_back ('!');
83
84    return (objectAttributes);
85  }
86
87  vector<char> getAttributeSetConcept (
88    vector<int> HasseLevelSet, vector<vector<int>> matrix)
89  {
```

```
90    int i;
91    vector<char> objectAttributeSet;
92
93    if (HasseLevelSet[0] == -1)
94      return (attributeSet);
95
96    for (i = 0; i < matrix.size (); ++i)
97      if (matrix[HasseLevelSet[0]][i] == 1)
98        objectAttributeSet.push_back (attributeSet[i]);
99
100   if (objectAttributeSet.empty ())
101   {
102     objectAttributeSet.push_back ('!');
103     return (objectAttributeSet);
104   }
105
106   for (i = 1; i < HasseLevelSet.size (); ++i)
107     objectAttributeSet = charIntersect (
108       objectAttributeSet,
109       getAttributeSetConceptMachinerie (HasseLevelSet[i],
110                                         matrix));
111
112   if (objectAttributeSet.empty ())
113   {
114     objectAttributeSet.push_back ('!');
115     return (objectAttributeSet);
116   }
117
118   return (objectAttributeSet);
119 }
120
121 void HasseAttributeSet (
122   vector<vector<vector<int>>> HasseLevels,
123   vector<vector<pair<vector<int>,
124   vector<int>>>> HasseLevelsConnections,
125   vector<vector<int>> matrix)
126 {
127   int i, j;
128   vector<vector<pair<vector<int>,
129                      vector<char>>>> alignedAttributeSets;
130
```

```
131   for (auto HasseLevel : HasseLevels)
132   {
133     vector<pair<vector<int>,
134            vector<char>>> levelAlignedAttributeSets;
135
136     for (vector<int> set : HasseLevel)
137       levelAlignedAttributeSets.push_back (
138         make_pair (set, getAttributeSetConcept (set, matrix)));
139
140     alignedAttributeSets.push_back (levelAlignedAttributeSets);
141   }
142   displayConceptGrid (HasseLevelsConnections,
143                       alignedAttributeSets);
144 }
145
146 vector<pair<vector<int>,
147        vector<int>>> getHasseConnectionsConceptMachinerie (
148   vector<vector<int>> pLevel, vector<vector<int>> nLevel)
149 {
150   int i, j;
151   vector<pair<vector<int>, vector<int>>> levelConnections;
152
153   for (i = 0; i < pLevel.size (); ++i)
154   {
155     for (j = 0; j < nLevel.size (); ++j)
156       if (isSubset (nLevel[j], pLevel[i]))
157         levelConnections.push_back (make_pair (pLevel[i],
158                                                nLevel[j]));
159   }
160   return (levelConnections);
161 }
162
163 void getHasseConnectionsConcept (
164   vector<vector<vector<int>>> HasseLevels,
165   vector<vector<pair<vector<int>,
166                 vector<int>>>>& HasseLevelsConnections)
167 {
168   int i;
169
170   for (i = 0; i < HasseLevels.size () - 1; ++i)
171     HasseLevelsConnections.push_back (
```

```
172       getHasseConnectionsConceptMachinerie (
173         HasseLevels[i], HasseLevels[i + 1]));
174 }
175
176 void getHasseDiagramConceptGrid (
177   vector<vector<int>>& closuredSetGrid,
178   vector<vector<int>> matrix)
179 {
180   int i;
181   vector<vector<int>> mins;
182   vector<vector<vector<int>>> HasseLevels;
183   vector<vector<pair<vector<int>,
184                      vector<int>>>> HasseLevelsConnections;
185   map<vector<int>, vector<vector<int>>> subsetRelation;
186
187   for (i = 0; i < closuredSetGrid.size (); ++i)
188   {
189     vector<vector<int>> subsets;
190     subsetRelation[closuredSetGrid[i]] = subsets;
191   }
192
193   while (!closuredSetGrid.empty ())
194   {
195     getSubsetRelation (subsetRelation, closuredSetGrid);
196     mins = getMinsConcept (subsetRelation);
197     HasseLevels.push_back (mins);
198     removeMinsConcept (subsetRelation, closuredSetGrid, mins);
199   }
200   getHasseConnectionsConcept (HasseLevels,
201                               HasseLevelsConnections);
202   HasseAttributeSet (HasseLevels,
203                      HasseLevelsConnections,
204                      matrix);
205 }
206
207 void getInverse (vector<vector<int>> matrix,
208                  vector<vector<int>>& closuredSetGrid)
209 {
210   int i, j;
211   vector<int> inverseRow;
212
```

```
213   for (i = 0; i < matrixDimension; ++i)
214     inverseRow.push_back (i);
215
216   closuredSetGrid.push_back (inverseRow);
217   inverseRow.clear ();
218
219   for (i = 0; i < matrixDimension; ++i)
220   {
221     for (j = 0; j < matrixDimension; ++j)
222       if (matrix[j][i] == 1)
223         inverseRow.push_back (j);
224
225     if (closuredSetGrid[0] != inverseRow)
226       closuredSetGrid.push_back (inverseRow);
227
228     inverseRow.clear ();
229   }
230
231   getIntersects (closuredSetGrid);
232 }
233
234 void conceptGrid ()
235 {
236   vector<int> objectSet;
237   vector<vector<int>> matrix;
238   vector<vector<int>> closuredSetGrid;
239
240   cout << "INPUT MATRIX DIMENSION\n";
241   cin >> matrixDimension;
242
243   cout << "INPUT YOUR OBJECTS:\n";
244   getObjectSet (objectSet);
245
246   cout << "INPUT YOUR ATTRIBUTES:\n";
247   getAttributeSet ();
248   getMatrix (matrix);
249
250   getInverse (matrix, closuredSetGrid);
251   getHasseDiagramConceptGrid (closuredSetGrid, matrix);
252 }
```

Листинг 10: Псевдокод алгоритма.

**Результат тестирования программы построения решётки концептов.**

Рассмотрим контекст $K$ с множеством объектов $G = \{1, 2, 3, 4\}$, множеством атрибутов $M = \{a, b, c, d\}$ и отношением $\rho$, определяемым матрицей.

```
INPUT MATRIX DIMENSION
4
INPUT YOUR OBJECTS:
1 2 3 4
INPUT YOUR ATTRIBUTES:
a b c d
INPUT THE ELEMENTS OF YOUR MATRIX:
1 0 1 0
1 1 0 0
0 1 0 1
0 1 0 1

HASSE DIAGRAM:
<1, 2, 3, 4> -> (EMPTY SET).
<1, 2> -> (a), <2, 3, 4> -> (b).
<1> -> (a, c), <2> -> (a, b), <3, 4> -> (b, d).
<EMPTY SET> -> (a, b, c, d).

HASSE DIAGRAM CONNECTIONS:
<1, 2> -> <1, 2, 3, 4>, <2, 3, 4> -> <1, 2, 3, 4>.
<1> -> <1, 2>, <2> -> <1, 2>, <2> -> <2, 3, 4>, <3, 4> -> <2, 3, 4>.
<EMPTY SET> -> <1>, <EMPTY SET> -> <2>, <EMPTY SET> -> <3, 4>.
```

Рисунок 5 – Диаграмма Хассе решётки концептов $C(K)$ заданного контекста K, изоморфная диаграмме системы замыканий $Z_{f_G}$.

Результатом работы программы является вывод построенной для решётки концептов $C(K)$ диаграммы Хассе по уровням, а также вывод сформированных между элементами уровней связей.

**Оценка временной сложности алгоритма построения решётки концептов.**

Для оценки временной сложности алгоритма будем рассматривать сложность каждой реализуемой им операции в соответствии с псевдокодом.

Первая операция – построение системы замыканий $Z_{f_G}$ на множестве $G$. Изначально в $Z_{f_G}$ кладётся само множество $G$, затем вычисляются значения $\rho^{-1}(a_i)$, $i = 1, \ldots, |M|$, $\rho \subset G \times M$ – заданное бинарное отношение. Для вычисления значения $\rho^{-1}(a_i)$ необходимо осуществить проход по столбцу соответствующего атрибута $a_i$. Тогда число операций, необходимое для вычисления промежуточного значения системы замыканий $Z_{f_G}$ составляет $n^2$, где $n$

– число строк или столбцов в квадратной матрице бинарного отношения $\rho$. Для получения полной системы замыканий $Z_{f_G}$ каждый элемент промежуточного значения системы $Z_{f_G}$ пересекается со всеми остальными элементами в $Z_{f_G}$, и полученные новые множества добавляются в $Z_{f_G}$. Заметим, что после построения $Z_{f_G}$ верно, что $|Z_{f_G}| \leq |2^G|$ – мощность булеана множества объектов $G$. Грубо оценим сверху число необходимых операций для построения системы замыканий $Z_{f_G}$ как $|2^G| \cdot |2^G| \cdot n^2$, где $n^2$ – верхняя граница числа операций, необходимых для выполнения пересечения двух множеств (мощность каждого множества не превосходит $n$). $|2^G|$ может быть заменено на $2^n$, тогда: $T(get\_closure\_system, n) = O(2^n \cdot 2^n \cdot n^2 + n^2) = O(n^2 \cdot 4^n)$, где $n$ – число строк или столбцов в квадратной матрице заданного бинарного отношения $\rho$.

Перейдём к рассмотрению операций, выполняемых внутри цикла $while$. Первая операция – поиск минимальных элементов на текущей итерации в системе замыканий $Z_{f_G}$. Для этого для каждого элемента (множества) из $Z_{f_G}$ хранится набор множеств из $Z_{f_G}$, содержащихся в рассматриваемом множестве. Для построения такого набора необходимо зафиксировать некоторый элемент (множество) из $Z_{f_G}$ и для оставшихся в $Z_{f_G}$ множеств проверить, являются ли они подмножествами зафиксированного элемента. Поскольку мощность каждого элемента $Z_{f_G}$ не превосходит $n$ ($n$ – мощность множества объектов или множества атрибутов), то верхней границей числа операций, необходимых для определения того, является ли рассматриваемое множество из $Z_{f_G}$ подмножеством фиксированного элемента, будет $n^2$. Также, поскольку $|Z_{f_g}| \leq |2^G| = 2^n$, временная оценка сложности операции поиска минимальных элементов на текущей итерации принимает вид: $T(get\_mins, n) = O(n^2 \cdot 4^n)$.

Следующая операция – удаление найденных минимальных элементов из системы замыканий $Z_{f_G}$. Здесь будут действовать те же самые оценки временной сложности, полученные для предыдущих операций, за исключением того, что не будет производиться дополнительных процедур с временной сложность $O(n^2)$, поэтому: $T(remove\_mins, n) = O(4^n)$.

Поскольку две рассмотренные выше операции выполняются внутри цикла, который может выполняться не более $2^n$ раз, то общая временная сложность операции с циклом составляет: $T(while(), n) = O(2^n \cdot (n^2 \cdot 4^n + 4^n)) = O(n^2 \cdot 8^n)$.

Следующая операция – построение связей между элементами разных уровней диаграммы Хассе. Будем считать, что была сформирована диаграм-

ма Хассе с $p$ уровнями. Через $n_i$ обозначим число элементов на каждом уровне диаграммы Хассе, $i = 1, \ldots, p$. Очевидно, что $\sum_{i=1}^{p} n_i = |Z_{f_G}|$. Обозначим через $n_{ij}$ $j$-ый элемент $i$-го уровня ($j = 1, \ldots, n_i,\ i = 1, \ldots, p$). Будем считать, что между элементами должна быть сформирована связь, если $n_{ij} \subset n_{i+1k}\,(i = 1, \ldots, p-1,\, j = 1, \ldots, n_i,\, k = 1, \ldots, n_{i+1})$. Тогда число операций, необходимое для формирования связей между элементами уровней диаграммы Хассе не превосходит $\sum_{i=1}^{p-1} n_i n_{i+1} \cdot \max\limits_{j=1,\ldots,n_i} |n_{ij}| \cdot \max\limits_{k=1,\ldots,n_{i+1}} |n_{i+1k}|$. Т.к. $n_i \leq \max\limits_{i=1,\ldots,p} n_i \leq |Z_{f_G}| \leq |2^G| = 2^n$ и $\max\limits_{j=1,\ldots,n_i} |n_{ij}| \leq n$ и $p \leq |Z_{f_G}| \leq |2^G| = 2^n$, то $\sum_{i=1}^{p-1} n_i n_{i+1} \cdot \max\limits_{j=1,\ldots,n_i} |n_{ij}| \cdot \max\limits_{k=1,\ldots,n_{i+1}} |n_{i+1k}| \leq n^2 \cdot 4^n \sum_{i=1}^{p-1} 1 \leq n^2 \cdot 8^n$. Таким образом, оценка временной сложности операции формирования связей между элементами уровней диаграммы Хассе: $T(get\_Hasse\_connections, n) = O(n^2 \cdot 8^n)$.

Последняя операция – дополнение каждого элемента диаграммы Хассе соответствующим элементом решётки концептов $C(K)$ заданного контекста $K$. Т.к. необходимо дополнить каждый элемент системы замыканий $Z_{f_G}$, которых не более, чем $2^n$, множеством из решётки концептов, которое представляет собой пересечение не более, чем $n$ множеств (с операцией пересечения, требующей не более $n^2$ шагов), то оценка временной сложности операции дополнения каждого элемента диаграммы Хассе принимает вид: $T(complete\_with\_attributes, n) = O(n^3 \cdot 2^n)$.

Тогда оценка временной сложности алгоритма построения решётки концептов: $T(get\_concept\_grid, n) = O(n^2 \cdot 4^n + n^2 \cdot 8^n + n^3 \cdot 2^n) = O(n^2 \cdot 8^n)$, где $n$ – мощность множества объектов или множества атрибутов.

# ЗАКЛЮЧЕНИЕ

В ходе лабораторной работы мною были изучены основные свойства бинарных отношений и операций их замыкания. В качестве практического задания мною были описаны и реализованы алгоритмы построения эквивалентного замыкания бинарного отношения, алгоритм построения фактор-множества, нахождения минимальных, наименьшего и максимальных, наибольшего элементов в заданном отношении порядка, алгоритм построения диаграммы Хассе для отношения делимости по заданному натуральному числу, алгоритм построения диаграммы Хассе по заданной матрице порядка, а также алгоритм построения диаграммы Хассе решётки концептов по заданному контексту. Для этих алгоритмов были представлены псевдокоды, осуществлена их программная реализация, а также проведена оценка временной сложности.