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

Алексеева Александра Александровича

|  |  |  |
| --- | --- | --- |
| Преподаватель  профессор, д.ф.-м.н. | \_\_\_\_\_\_\_\_\_\_ | В. А. Молчанов |
|  | подпись, дата |  |

Саратов 2022

1 Цель работы и порядок её выполнения­

Цель работы – изучение основных свойств бинарных отношений и операций замыкания бинарных отношений.

Порядок выполненных работы:

1. Разобрать определения отношения эквивалентности, фактор-множества. Разобрать 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 диаграмм Хассе.

3. Разработать определения контекста и концепта. Разработать алгоритм вычисления решётки концептов.

2 Теория

2.1 Отношение эквивалентности и фактор-множество

*Определение*. Бинарное отношение *ε* на множестве *А* называют отношением эквивалентности, если оно рефлексивно, симметрично и транзитивно.

Для обозначения эквивалентности ε используется инфиксная запись с помощью символа «≡»: вместо (*a*, *b*) *ε* пишут *a* ≡ *b*(*ε*) или просто *a* ≡ *b*.

Срезы *ε*(*а*) называются классами эквивалентности по отношению *ε* и сокращённо обозначаются символом [*а*].

*Определение.* Множество всех таких классов эквивалентности {[*a*]:*a* **} называется фактор-множеством множества *А* по эквивалентности *ε* и обозначается символом *A/ε*.

Характерное свойство: для любых элементов *a*, *b* **условие [*a*]∩[*b*]≠Ø

Подмножество *T* ** называется полной системой представителей классов эквивалентности ε на множестве *А*, если:

1) *ε*(*T*) = *А*,

2) из условия *t*1=*t*2(*ε*) следует *t*1=*t*2.

Классы эквивалентности [*t*] *ε* могут быть отождествлены со своими представителями *t* и фактор-множество *A/ε* может быть отождествлено с множеством *T*.

2.2 Алгоритм построения эквивалентного замыкания бин. отношения

Вход. Матрица бинарного отношения mas и размер матрицы N.

Выход. Матрица бинарного отношения с замыканием эквивалентности.

Шаг 1. Для матрицы бин. отношения применить алгоритм построения замыкания рефлексивности.

Шаг 2. Для матрицы бин. отношения с замыканием рефлексивности применить алгоритм построения замыкания симметричности.

Шаг 3. Для матрицы бин. отношения с замыканиями рефлексивности и симметричности применить алгоритм построения замыкания транзитивности.

Шаг 4. Вывести матрицу бин. отношения с замыканием эквивалентности.

2.3 Алгоритм построения фактор-множества

*Вход*. Матрица бинарного отношения *matrix* и размер матрицы *N*.

*Выход*. Фактор-множество *classes*.

Шаг 1. Создаём список *used* размера *N*, в котором будем храниться информация о посещённых элементах, изначально все его элементы равны 0.

Шаг 2. Цикл по *i* от 1 до *N*. Для каждого *i* создаём список *clas* и выполняем цикл по *j* от *1* до *N*. Если *matrix*[*i*][*j*] = 1 и *used*[*j*] = 0, в список *clas* кладём элемент *j* и *used*[*j*] = 1.

Шаг 3. По окончанию цикла по *j* проверяем список *clas* на пустоту. Если список *clas* не пустой, кладём его в список списков *classes*.

Шаг 4. Выводим фактор-множество *classes*.

2.4 Алгоритм построения системы представителей фактор-множества

*Вход*. Фактор-множество classes.

*Выход*. Система представителей на заданном множестве.

Шаг 1. Создаём переменную *size*, в которой хранится количество классов фактор-множества.

Шаг 2. Цикл по *i* от 1 до *size*. Для каждого *i* сортируем список classes[*i*] по возрастанию и выводим первый элемент classes[*i*][1].

2.5 Отношения порядка и диаграммы Хассе

*Определение.* Бинарное отношение *ε* на множестве A называется отношением порядка, если оно рефлексивно, антисимметрично и транзитивно.

Поскольку отношение порядка интуитивно отражает свойство «больше-меньше», то для обозначения порядка *ε* используется инфиксная запись с помощью символа ≤: вместо (*a*, *b*) *ε* принято писать *a* ≤ *b*.

Запись *a* < *b* означает, что *a* ≤ *b* и *a* ≠ *b*.

Запись *a* <∙ *b* означает, что *a* ≤ *b* и нет элементов *x*, удовлетворяющих условию *a* < *x* < *b*. В этом случае говорят, что элемент *b* покрывает элемент *a* или что элемент *a* покрывается элементом *b*.

Элемент а упорядоченного множества (А, ≤) называется:

– минимальным, если (∀*x* **) *x* ≤ *a* ⇒ *x* = *a*,

– максимальным, если (∀*x* **) *a* ≤ *x* ⇒ *x* = *a*,

– наименьшим, если (∀*x* **) *a* ≤ *x,*

– наибольшим, если (∀*x* **) *x* ≤ *a.*

Наименьший и наибольший элементы упорядоченного множества (А, ≤) принято обозначать 0 и 1.

Очевидно, наименьший (соответственно, наибольший) элемент является минимальным (соответственно, максимальным), но в общем случае обратное неверно.

*Лемма*. Для любого конечного упорядоченного множества *A* = (*А*, ≤) справедливы следующие утверждения:

1) любой элемент множества *A* содержится в некотором максимальном элементе и содержит некоторый минимальный элемент;

2) если упорядоченное множество *A* имеет один максимальный (соответственно, минимальный) элемент, то он является наибольшим (соответственно, наименьшим) элементом этого множества.

Упорядоченное множество *A* = (*А*, ≤) наглядно представляется диаграммой Хассе, которая представляет элементы множества *А* точками плоскости и пары *a* <∙ *b* представляет линиями, идущими вверх от элемента *a* к элементу *b*.

*Алгоритм построения диаграммы Хассе* конечного упорядоченного множества *A* = (*А*, ≤):

1. В упорядоченном множестве *A* = (*А*, ≤) найти множество *А1*, всех минимальных элементов и расположить их в один горизонтальный ряд (это первый уровень диаграммы).

2. В упорядоченном множестве *А*\*А1*, найти множество *А2* 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 *А* \ (*А1* ∪ *А2*) найти множество *А3* 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.*

Подмножество X упорядоченного множества (*А*, ≤) называется:

− ограниченным сверху*,* если найдется такой элемент *a* *A,* что *x* ≤ *a* для всех *x*  *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*  *X* значение sup*X* является наибольшим элементом множества и обозначается max*X* ;

− ограниченным снизу*,* если найдется такой элемент *a* *A,* что *a* ≤ *x* для всех *x* *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*  *X* значение inf*X* является наименьшим элементом множества и обозначается min*X* .

Если само упорядоченное множество (*А*, ≤) ограничено сверху (соответственно, снизу), то его верхняя (соответственно, нижняя) грань является наибольшим (соответственно, наименьшим) элементом множества *A* и обозначается символом 1 (соответственно, 0).

Порядок ≤ на множестве *А* называется:

− линейным, если любые два элемента этого множества сравнимы, т.е. выполняется (∀*a*, *b* ∈ *A*) (*a* ≤ *b* ∨ *b* ≤ *a*);

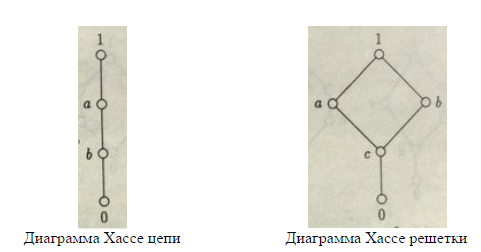
− полным, если его любое непустое подмножество имеет точную верхнюю и точную нижнюю грани;

− решеточным*,* если для любых *a,b* ∈ *A* существуют sup{*a,b*} и inf{*a,b*}, которые обозначаются соответственно *a* ∨ *b*, *a* ∧ *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.

Очевидно, что любая цепь является решеткой, но обратное в общем случае не выполняется.



2.6 Алгоритм разбиения элементов отношения порядка на уровни

*Вход*. Отношение порядка div в виде списка.

*Выход*. Список списков *lvls* элементов отношения порядка, разбитых на уровни.

Шаг 1. Создаём список *lvl*, который будет содержать минимальные элементы отношения порядка. Добавляем в него первый элемент отношения *div*[1] и удаляем этот элемент из списка *div*. Вычисляем размер списка *div* и сохраняем в *size*.

Шаг 2. Цикл по *i* от 1 до *size*. Для каждого *i* создаём флаг *flag* со значением true и запускаем цикл по *j* от 1 до *lvl*.size. Для каждого *j* если остаток от деления *div*[*i*] на *lvl*[*j*] = 0, то *flag* = false и завершить цикл по *j*.

Шаг 3. Для каждого *i* если *flag* = true, то добавляем в список *lvl* элемент *div*[*i*], удаляем этот элемент из списка *div* из уменьшаем *i* на единицу.

Шаг 4. После прохождения цикла по *i* добавляем список *lvl* в список списков *lvls*.

Шаг 5. Если список *div* не пустой, повторяем алгоритм.

2.7 Алгоритм вычисления минимальных и наименьшего элементов множества

*Вход*. Отношение порядка *div*.

*Выход*. Минимальные и наименьший элементы.

Шаг 1. Применяем алгоритм разбиения элементов отношения порядка на уровни для отношения *div* и сохраняем результат в список списков *lvls*.

Шаг 2. Вычисляем размер 1-го списка *lvls* и сохраняем результат в *size*.

Шаг 3. Если *size* > 1, то наименьший элемент отсутствует. Циклом по *i* от 1 до *size* выводим минимальные элементы, т.е. *lvls*[1][*i*].

Шаг 4. Иначе выводим наименьший и минимальный элемент, т.е. *lvls*[1][1].

2.8 Алгоритм вычисления максимальных и наибольшего элементов множества

*Вход*. Список списков *lvls* элементов отношения порядка, разбитых на уровни, матрица отношения порядка matrix.

*Выход*. Максимальные и наибольший элементы.

Шаг 1. Создаём результирующую список *res*. Вычисляем количество списков в списке *lvls* и сохраняем в *iSize*. Цикл по *i* от 1 до *size* – 1.

Шаг 2. Для каждого *i* вычисляем количество элементов списка *lvls*[*i*] и сохраняем в *jSize*. Для *i* цикл по *j* от 1 до *jSize*. Для каждого *j* создаём флаг *flag* с верным значением true, вычисляем количество элементов списка *lvls*[*i* + 1] и сохраняем в *kSize*.

Шаг 3. Для *j* цикл по *k* от 1 до *kSize*. Если элемент матрицы *matrix*[*lvls*[*i*][*j*]][*lvls*[*i* + 1][*k*]] = 1, то *flag* = false и выход из цикла по *k*.

*Шаг 4*. Если *flag* = true, добавляем элемент *lvls*[*i*][*j*] в список *res*.

*Шаг 5*. После прохода всех циклов вычисляем размер списка *res* и сохраняем в *size*.

*Шаг 6*. Если *size* > 1, наибольший элемент отсутствует.

*Шаг 7*. Иначе выводим наибольший элемент *res*[1].

*Шаг 8*. Цикл по *i* от 1 до *size*. Выводим максимальные элементы *res*[*i*].

2.9 Алгоритм Построения диаграммы Хассе

*Вход*. Список списков *lvls* элементов отношения порядка, разбитых на уровни, матрица отношения порядка matrix.

*Выход*. Диаграмма Хассе.

Шаг 1. Создаём список *lvl*, в котором будут храниться элементы минимального уровня. Вычисляем количество списков списка *lvls* и сохраняем в *nSize*.

Шаг 2. Цикл по *n* от *nSize* до 1. Для каждого *n* вычисляем количество элементов списка *lvls*[*n*] и сохраняем в *iSize*.

Шаг 3. Для *n* цикл по *i* от 1 до *iSize*. Для каждого *i* выводим элемент *lvls*[*n*][*i*], вычисляем количество элементов списка *lvls*[*n*-1] и сохраняем в *jSize*.

Шаг 4. Цикл по *j* от 1 до *jSize*. Если *matrix*[*lvls*[*n*-1][*j*]][*lvls*[*n*][*i*]] = 1, сохраняем элемент *lvls*[*n*-1][*j*]] в список *lvl*.

Шаг 5. После выхода из цикла по *j* выводим элементы списка *lvl*, которые связаны с элементом *lvls*[*n*][*i*].

2.10 Контексты и концепты

*Теорема Галуа*. Пусть *ρ* ⊂ А×В - произвольное бинарное отношение между элементами множеств А и В. Для любого подмножества Х ⊂ А определим подмножество X' ⊂ В по формуле

Х' = {b ∈ B | (х, b) ∈ ρ для всех х ∈ Х}

и для любого подмножества Y ⊂ В определим подмножество Y' ⊂ А по формуле

Y’ = {a ∈ А| (а, y) ∈ ρ для всех у ∈ Y}.

Тогда отображения Х → Х' и Y → Y' множеств Р(А) и Р(В) друг в друга обладают следующими свойствами:

1) Х1 ⊂ X2 ⇒ X2' ⊂ X1';

2) Y1 ⊂ Y2 ⇒ Y2' ⊂ Y1';

3) X ⊂ X'' , Y ⊂ Y'';

4) Х"' = Х' , Y"' = Y'.

Пара таких отображений называется соответствием Галуа, которое определяет два оператора замыкания fA и fB на множествах А и В по формулам:

fA(X) = Х'' для любых X ⊂ А и fB(Y) = Y'' для любых Y ⊂ B.

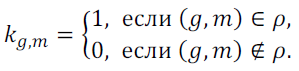
При этом отображения Х → Х' и Y → Y' определяют антиизоморфизмы между системами замыканий и на множествах А и В, т.е. это биекции, которые удовлетворяют условиям:

X ⊂ Y ⇔ X' ⊃ Y' для любых Х,Y ∈ ,

X ⊂ Y ⇔ X' ⊃ Y' для любых Х,Y ∈ .

*Определение*. Контекстом называется алгебраическая система *K* = (*G*, *M*, *ρ*), состоящая из множества объектов *G*, множества атрибутов *M* и бинарного отношения *ρ* *G* × *M*, показывающего (*g*, *m*) ∈ *ρ*, что объект *g* имеет атрибут *m*.

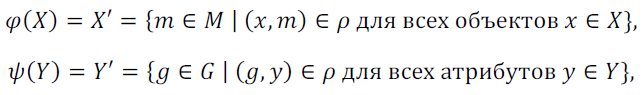
Контекст *K* = (*G*, *M*, *ρ*) наглядно изображается таблицей, в которой строки помечены элементами множества *G*, столбцы помечены элементами множества *M* и на пересечении строки с метой *g* ∈ *G* и столбца с меткой *m* ∈ *M* стоит элемент



По теореме Галуа для контекста *K* = (*G*, *M*, *ρ*) отображения



которые для *X* *G*, *Y* *M* определяются по формулам:

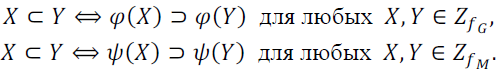


образуют соответствие Галуа между подмножествами множеств *G* и *M*, которое определяет на этих множествах операторы замыкания по формулам:



где *X* *G*, *Y* *M*.

При этом отображения *φ* и *ψ* определяют антиизоморфизмы между системами замыканий и на множествах *G* и *M*, т.е. это биекции, которые удовлетворяют условиям:



*Определение*. Упорядоченная пара (*X*, *Y*) замкнутых множеств *X* ∈ , *Y* ∈ , удовлетворяющих условиям *φ*(*X*) = *Y*, *ψ*(*Y*) = *X*, называется концептом контекста *K* = (*G*, *M*, *ρ*). При этом компонента *X* называется объёмом и компонента *Y* – содержанием концепта (*X*, *Y*).

Множество всех концептов *C*(*K*) так упорядочивается отношением



что (*C*(*K*), ≤) является полной решёткой, которая изоморфна решётке замкнутых подмножеств множества *G*.

В частности, по определению для g ∈ G, m ∈ M получаем

φ({g}) = {m ∈ M | (x, m) ∈ ρ для всех объектов x ∈ {g}} = ρ(g),

ψ({m}) = {g ∈ G | (g, y) ∈ ρ для всех атрибутов y ∈ {m}} = ρ-1(m).

Тогда для X  G, Y  M получаем

*φ*(*X*) = = , *ψ*(*Y*) = =

*fG*(*X*) = *X''* = = = ,

() = = = = .

*Алгоритм вычисления системы замыканий* на множестве G:

1. Рассматриваем множество G .

2. Последовательно перебираем все элементы *m* *M* и вычисляем для них *ψ*({*m*}) = -1(*m*).

3. Вычисляем все новые пересечения множества *ψ*({*m*}) с ранее полученными множествами и добавляем новые множества к .

Аналогично вычисляется система замыканий на множестве *M*.

2.11 Алгоритм построения системы замыкания

*Вход*. Матрица бинарного отношения matrix, размер матрицы N, список объектов object.

*Выход*. Система замыканий

Шаг 1. Создаём список списков *closureSystem*, добавляем в него с список *object*.

Шаг 2. Цикл по *i* от 1 до *N*. Для каждого *i* делаем срез по столбцу *i* бинарного отношения *matrix* и результат сохраняем в список *srez*. Список *srez* пересекаем с каждым списком из списка *closureSystem*. Если после пересечения список *srez* является уникальным в списке *closureSystem*, то добавляем его в список *closureSystem*.

Шаг 3. Выводим список списков *closureSystem*, т.е. систему замыканий.

2.12 Алгоритм построения решётки концептов

*Вход*. Матрица бинарного отношения *matrix*, размер матрицы *N*, список атрибутов *attr*, система замыканий *closureSystem*.

*Выход*. Решётка концептов.

Шаг 1. Вычисляем размер *closureSystem* и сохраняем в *size*.

Шаг 2. Создаём список списков *sets*. Цикл по *i* от 1 до *size*. Вычисляем срезы по каждому элементу элемента списка *closureSystem*[*i*] и добавляем результат в *sets*.

Шаг 3. Пока размер списка *sets* > 1. Пересекаем первые 2 элемента списка *sets*, результат добавляем в конец списка *sets*. Удаляем первые 2 элемента списка *sets*.

Шаг 4. Для каждого *i* выводим *closureSystem*[*i*] и список *sets*. После вывода обнуляем список *sets*.

3 Результаты работы

3.1 Оценка временной сложности алгоритмов

Построение эквивалентного замыкания бин. отношения – O(N4).

Построение фактор-множества – O(N2).

Построение системы представителей фактор-множества– O(N+NlogN) ~ O(NlogN).

Разбиение элементов отношения порядка на уровни – O(N2).

Вычисление минимальных и наименьшего элементов множества– O(N).

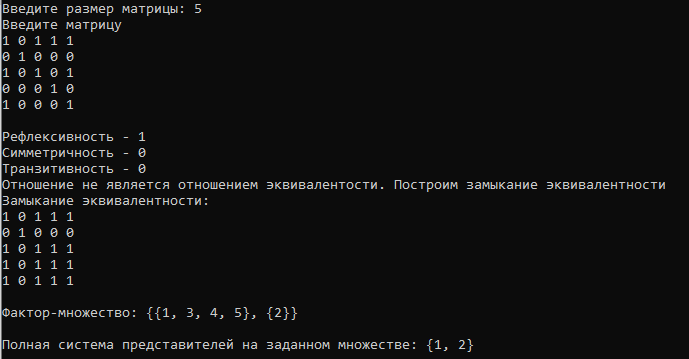
Вычисление макс. и наибольшего элементов множества – O(N\* M \* L), где N – количество уровней, на которые разбиты элементы отношения порядка; M – количество элементов наибольшего уровня; L – количество элементов на уровне, который находится выше наибольшего уровня.

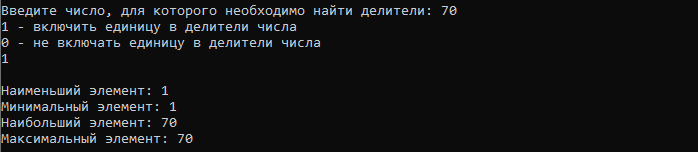
Построение диаграммы Хассе– O(N\* M), где N – количество уровней, на которые разбиты элементы отношения порядка, M – количество элементов на уровне, на котором находится наибольшее количество элементов.

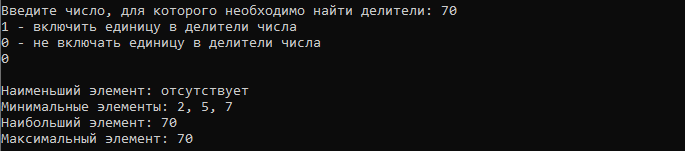
Построение системы замыкания – O(N2 \* M \* L), где N – количество объектов в контексте; M – количество элементов в системе замыканий, L – размер наибольшего элемента системы замыканий(не считая множество объектов G).

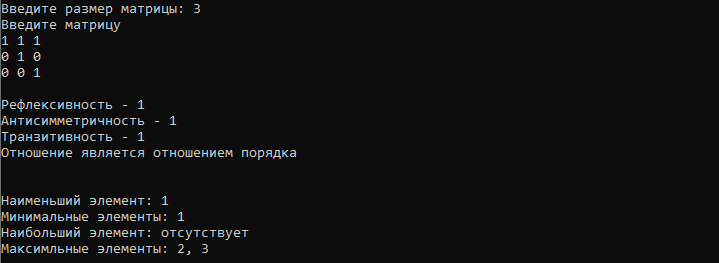
Построение решётки концептов – O(N2 \* M), где N – количество элементов в системе замыканий; M – количество элементов в наибольшем элементе системы замыканий.

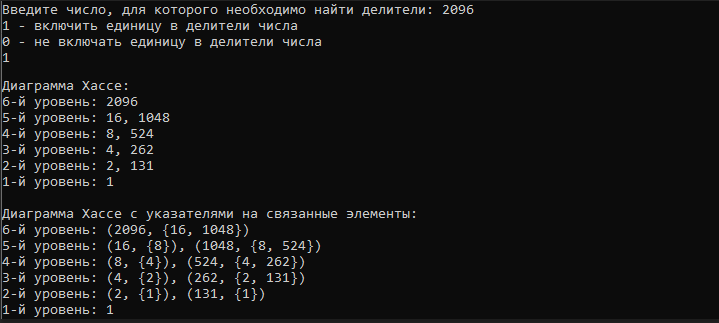
3.2 Результаты тестирования программ

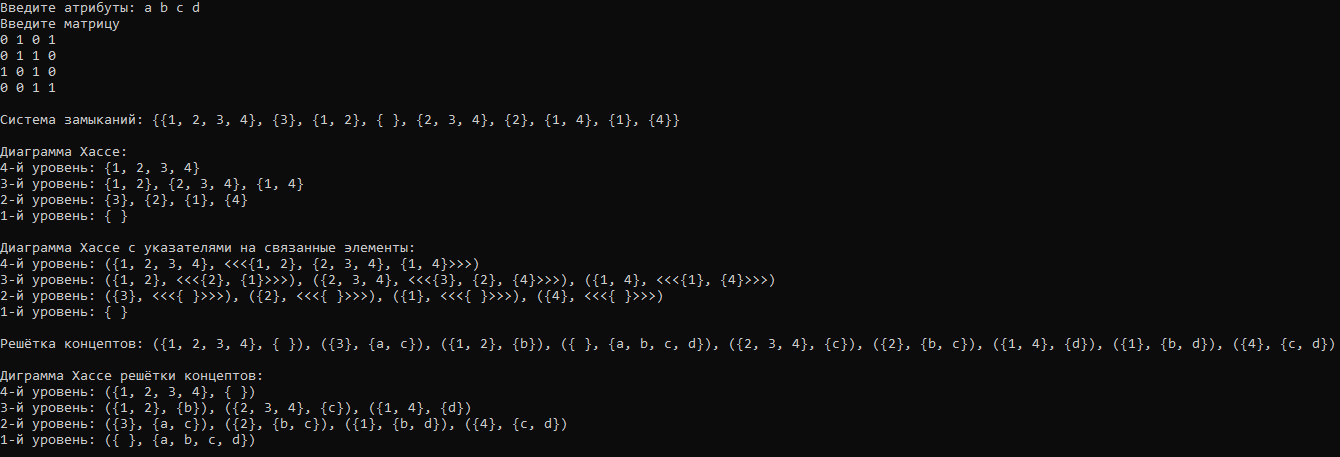












3.3 Код программы

#include "iostream"

#include "vector"

#include "iomanip"

using namespace std;

bool reflexivity(int\*\* mas, int N) {

for (int i = 0; i < N; i++)

if (mas[i][i] == 0)

return false;

return true;

}

bool symmetry(int\*\* mas, int N) {

for (int i = 0; i < N; i++)

for (int j = i; j < N; j++)

if (mas[i][j] != mas[j][i])

return false;

return true;

}

bool antisymmetry(int\*\* mas, int N) {

for (int i = 0; i < N; i++)

for (int j = i; j < N; j++)

if (i != j && mas[i][j] == 1 && mas[j][i] == 1)

return false;

return true;

}

bool transitivity(int\*\* mas, int N) {

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

for (int k = 0; k < N; k++)

if (mas[i][j] == 1 && mas[j][k] == 1 && mas[i][k] != 1)

return false;

return true;

}

void make\_reflexivity(int\*\* mas, int N, bool flag) {

for (int i = 0; i < N; i++)

mas[i][i] = 1;

if (flag) {

cout << "Замыкание рефлексивности:\n";

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

for (int j = 0; j < N; j++)

cout << mas[i][j] << " ";

cout << endl;

}

}

}

void make\_symmetry(int\*\* mas, int N, bool flag) {

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

if (mas[i][j] == 1)

mas[j][i] = 1;

if (flag) {

cout << "Замыкание симметричности:\n";

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

for (int j = 0; j < N; j++)

cout << mas[i][j] << " ";

cout << endl;

}

}

}

void make\_transitivity(int\*\* mas, int N, bool flag) {

for (int n = 0; n < N; n++)

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

for (int k = 0; k < N; k++)

if (mas[i][j] == 1 && mas[j][k] == 1)

mas[i][k] = 1;

if (flag) {

cout << "Замыкание транзитивности:\n";

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

for (int j = 0; j < N; j++)

cout << mas[i][j] << " ";

cout << endl;

}

}

}

void make\_equivalence(int\*\* mas, int N) {

make\_reflexivity(mas, N, false);

make\_symmetry(mas, N, false);

make\_transitivity(mas, N, false);

cout << "Замыкание эквивалентности:\n";

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

for (int j = 0; j < N; j++)

cout << mas[i][j] << " ";

cout << endl;

}

cout << endl;

}

void hasse(vector <vector <int> >lvls) {

int size = lvls.size();

cout << endl << "Диаграмма Хассе: \n";

for (int n = size - 1; n > 0; n--) {

cout << n + 1 << "-й уровень: ";

int iSize = lvls[n].size();

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

if (i == iSize - 1)

cout << lvls[n][i] << endl;

else

cout << lvls[n][i] << ", ";

}

}

cout << "1-й уровень: ";

size = lvls[0].size();

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

if (i == size - 1)

cout << lvls[0][i] << endl;

else

cout << lvls[0][i] << ", ";

}

}

void work\_equivalence() { //Работаем с отношением эквивалентности

int N;

cout << "Введите размер матрицы: ";

cin >> N;

cout << "Введите матрицу \n";

int\*\* matrix = new int\* [N];

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

matrix[i] = new int[N];

for (int j = 0; j < N; j++)

cin >> matrix[i][j];

}

cout << endl;

int ref = reflexivity(matrix, N), sym = symmetry(matrix, N), tran = transitivity(matrix, N);;

cout << "Рефлексивность - " << ref << endl << "Симметричность - " << sym << endl << "Транзитивность - " << tran << endl;

if (ref == 1 && sym == 1 && tran == 1)

cout << "Отношение является отношением эквивалентости\n\n";

else {

cout << "Отношение не является отношением эквивалентости. Построим замыкание эквивалентности\n";

make\_equivalence(matrix, N);

}

//Разбиваем элементы на классы эквивалентности

vector <int> used(N, 0);

vector <vector <int> > classes;

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

vector <int> clas;

for (int j = 0; j < N; j++) {

if (matrix[i][j] && !used[j]) {

clas.push\_back(j + 1);

used[j] = 1;

}

}

if (!clas.empty())

classes.push\_back(clas);

}

cout << "Фактор-множество: {";

int size = classes.size();

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

cout << "{";

int iSize = classes[i].size();

for (int j = 0; j < iSize; j++) {

if (j == iSize - 1)

cout << classes[i][j] << "}";

else

cout << classes[i][j] << ", ";

}

if (i == size - 1)

cout << "} \n\n";

else

cout << ", ";

}

cout << "Полная система представителей на заданном множестве: {";

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

if (i == size - 1)

cout << classes[i][0] << "}\n";

else

cout << classes[i][0] << ", ";

}

}

void findMinDiv(vector <vector <int> >& lvls, vector <int>& div) {

vector <int> lvl;

lvl.push\_back(div[0]);

div.erase(div.begin());

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

bool flag = true;

for (int j = 0; j < lvl.size(); j++) {

if (div[i] % lvl[j] == 0) {

flag = false;

break;

}

}

if (flag) {

lvl.push\_back(div[i]);

div.erase(div.begin() + i);

i--;

}

}

lvls.push\_back(lvl);

}

void work\_order\_number() { //Работаем с отношением порядка, ввод числом

int N;

cout << "Введите число, для которого необходимо найти делители: ";

cin >> N;

int one, i;

cout << "1 - включить единицу в делители числа \n0 - не включать единицу в делители числа \n";

cin >> one;

if (one)

i = 1;

else

i = 2;

//Ищем делители числа

vector <int> div;

for (i; i <= N / 2; i++)

if (N % i == 0)

div.push\_back(i);

div.push\_back(N);

//Распологаем делители по уровням

vector <vector <int> > lvls;

while (!div.empty())

findMinDiv(lvls, div);

//Находим мин(макс), наим(наиб) элементы

cout << endl;

int size = lvls[0].size();

if (size > 1) {

cout << "Наименьший элемент: отсутствует \nМинимальные элементы: ";

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

if (i == size - 1)

cout << lvls[0][i] << endl;

else

cout << lvls[0][i] << ", ";

}

}

else

cout << "Наименьший элемент: " << lvls[0][0] << "\nМинимальный элемент: " << lvls[0][0] << endl;

cout << "Наибольший элемент: " << lvls.back()[0] << "\nМаксимальный элемент: " << lvls.back()[0] << endl;

//Строим диаграмму Хассе

hasse(lvls);

//Строим диаграмму Хассе с указателями на связанные элементы

vector <int> lvl;

size = lvls.size();

cout << endl << "Диаграмма Хассе с указателями на связанные элементы: \n";

for (int n = size - 1; n > 0; n--) {

cout << n + 1 << "-й уровень: ";

for (int i = 0; i < lvls[n].size(); i++) {

cout << "(" << lvls[n][i] << ", {";

lvl.resize(0);

for (int j = 0; j < lvls[n - 1].size(); j++) {

int mod = lvls[n][i] % lvls[n - 1][j];

if (mod == 0)

lvl.push\_back(lvls[n - 1][j]);

}

int jSize = lvl.size();

for (int j = 0; j < jSize; j++) {

if (j == jSize - 1 && i == lvls[n].size() - 1)

cout << lvl[j] << "}) \n";

else if (j == jSize - 1)

cout << lvl[j] << "}), ";

else

cout << lvl[j] << ", ";

}

}

}

cout << "1-й уровень: ";

size = lvls[0].size();

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

if (i == size - 1)

cout << lvls[0][i] << endl;

else

cout << lvls[0][i] << ", ";

}

}

void findMinEl(int\*\* matrix, int N, vector <vector <int> >& lvls) { //Разбиваем на уровни

int\*\* newMatrix = new int\* [N];

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

newMatrix[i] = new int[N];

for (int j = 0; j < N; j++)

newMatrix[i][j] = matrix[i][j];

}

vector <int> lvl;

int flag = true;

int min;

for (int exit = 0; exit < N;) {

min = N + 1;

lvl.resize(0);

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

int count = 0;

bool flag = true;

for (int j = 0; j < N; j++) {

if (newMatrix[j][i] == -1) {

flag = false;

break;

}

if (newMatrix[j][i] == 1)

count++;

}

if (!flag)

continue;

if (count < min) {

min = count;

lvl.resize(0);

lvl.push\_back(i + 1);

}

else if (count == min)

lvl.push\_back(i + 1);

}

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

for (int j = 0; j < N; j++)

newMatrix[j][lvl[i] - 1] = -1;

exit++;

}

lvls.push\_back(lvl);

}

}

pair <vector <int>, vector<int> > min\_max(int\*\* matrix, vector <vector <int> > lvls) { //Находим минимальные и максимальные элементы

pair <vector <int>, vector<int> > res;

for (int i = 0; i < lvls[0].size(); i++)

res.first.push\_back(lvls[0][i]);

for (int i = 0; i < lvls.size() - 1; i++)

for (int j = 0; j < lvls[i].size(); j++) {

bool flag = true;

for (int k = 0; k < lvls[i + 1].size(); k++)

if (matrix[lvls[i][j] - 1][lvls[i + 1][k] - 1]) {

flag = false;

break;

}

if (flag)

res.second.push\_back(lvls[i][j]);

}

for (int i = 0; i < lvls.back().size(); i++)

res.second.push\_back(lvls.back()[i]);

return res;

}

void work\_order\_matrix() { //Работаем с отношением порядка, ввод матрицей

int N;

cout << "Введите размер матрицы: ";

cin >> N;

cout << "Введите матрицу \n";

int\*\* matrix = new int\* [N];

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

matrix[i] = new int[N];

for (int j = 0; j < N; j++)

cin >> matrix[i][j];

}

cout << endl;

int ref = reflexivity(matrix, N), antisym = antisymmetry(matrix, N), tran = transitivity(matrix, N);

cout << "Рефлексивность - " << ref << endl << "Антисимметричность - " << antisym << endl << "Транзитивность - " << tran << endl;

if (ref == 1 && antisym == 1 && tran == 1)

cout << "Отношение является отношением порядка \n\n";

else {

cout << "Отношение не является отношением порядка \n";

make\_equivalence(matrix, N);

return;

}

//Распологаем элементы по уровням

vector <vector <int> > lvls;

findMinEl(matrix, N, lvls);

//Находим мин(макс), наим(наиб) элементы

cout << endl;

pair <vector <int>, vector <int>> minmax = min\_max(matrix, lvls);

int size = minmax.first.size();

if (size > 1)

cout << "Наименьший элемент: отсутствует \n";

else

cout << "Наименьший элемент: " << minmax.first[0] << "\n";

cout << "Минимальные элементы: ";

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

if (i == size - 1)

cout << minmax.first[i] << "\n";

else

cout << minmax.first[i] << ", ";

}

size = minmax.second.size();

if (size > 1)

cout << "Наибольший элемент: отсутствует \n";

else

cout << "Наибольший элемент: " << minmax.second[0] << "\n";

cout << "Максимльные элементы: ";

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

if (i == size - 1)

cout << minmax.second[i] << "\n";

else

cout << minmax.second[i] << ", ";

}

//Строим диаграмму Хассе

hasse(lvls);

//Строим диаграмму Хассе с указателями на связанные элементы

vector <int> lvl;

size = lvls.size();

cout << endl << "Диаграмма Хассе с указателями на связанные элементы: \n";

for (int n = size - 1; n > 0; n--) {

cout << n + 1 << "-й уровень: ";

for (int i = 0; i < lvls[n].size(); i++) {

cout << "(" << lvls[n][i] << ", {";

lvl.resize(0);

for (int j = 0; j < lvls[n - 1].size(); j++) {

if (matrix[lvls[n - 1][j] - 1][lvls[n][i] - 1])

lvl.push\_back(lvls[n - 1][j]);

}

int jSize = lvl.size();

for (int j = 0; j < jSize; j++) {

if (j == jSize - 1 && i == lvls[n].size() - 1)

cout << lvl[j] << "}) \n";

else if (j == jSize - 1)

cout << lvl[j] << "}), ";

else

cout << lvl[j] << ", ";

}

}

}

cout << "1-й уровень: ";

size = lvls[0].size();

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

if (i == size - 1)

cout << lvls[0][i] << endl;

else

cout << lvls[0][i] << ", ";

}

}

vector <int> generalEl(vector <int> x, vector <int> y) { //Ищем пересечение 2х множеств

vector <int> res;

for (int i = 0; i < x.size(); i++)

for (int j = 0; j < y.size(); j++)

if (y[j] == x[i]) {

res.push\_back(y[j]);

y.erase(y.begin() + j);

j--;

}

return res;

}

vector <int> cut(int\*\* matrix, int N, int num, string str) { //Срез строки или колонны

vector <int> res;

if (str == "row") {

for (int i = 0; i < N; i++)

if (matrix[num][i])

res.push\_back(i + 1);

}

else

for (int i = 0; i < N; i++)

if (matrix[i][num])

res.push\_back(i + 1);

return res;

}

bool unique(vector <vector <int> > lvls, vector <int> lvl) { //Смотрит, является ли наше множество уникальным

int size = lvl.size();

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

int iSize = lvls[i].size();

if (size != iSize)

continue;

bool flag = true;

for (int j = 0; j < size; j++)

if (lvl[j] != lvls[i][j]) {

flag = false;

break;

}

if (flag)

return false;

}

return true;

}

void findMaxSet(vector<vector <vector <int> > >& lvls, vector<vector <int> >& sets) { //Распологаем множества по уровням

vector <vector <int> > lvl;

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

bool flag = true;

for (int j = 0; j < sets.size(); j++) {

if (i != j && generalEl(sets[i], sets[j]) == sets[i]) {

flag = false;

break;

}

}

if (flag)

lvl.push\_back(sets[i]);

}

int size = lvl.size();

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

for (int j = 0; j < sets.size(); j++)

if (lvl[i] == sets[j]) {

sets.erase(sets.begin() + j);

break;

}

lvls.push\_back(lvl);

}

void printVec(vector <int> vec) {

cout << "{";

int size = vec.size();

if (size == 0) {

cout << " }";

return;

}

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

if (i == size - 1)

cout << vec[i] << "}";

else

cout << vec[i] << ", ";

}

}

void printConc(int\*\* matrix, int N, char\* attr, vector <int> vec) {

cout << "{";

if (vec.empty()) {

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

if (i == N - 1)

cout << attr[i] << "}";

else

cout << attr[i] << ", ";

}

return;

}

vector <vector <int> > sets;

for (int i = 0; i < vec.size(); i++)

sets.push\_back(cut(matrix, N, vec[i] - 1, "row"));

while (sets.size() > 1) {

sets.push\_back(generalEl(sets[0], sets[1]));

sets.erase(sets.begin() + 1);

sets.erase(sets.begin());

}

int size = sets[0].size();

if (size == 0) {

cout << " }";

return;

}

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

if (i == size - 1)

cout << attr[sets[0][i] - 1] << "}";

else

cout << attr[sets[0][i] - 1] << ", ";

}

}

void work\_concept() { //Работаем с контекстом

int N;

cout << "Введите количество объектов и атрибутов: ";

cin >> N;

vector <int> objects(N);

char\* attr = new char[N];

cout << "Введите объекты: ";

for (int i = 0; i < N; i++)

cin >> objects[i];

cout << "Введите атрибуты: ";

for (int i = 0; i < N; i++)

cin >> attr[i];

cout << "Введите матрицу \n";

int\*\* matrix = new int\* [N];

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

matrix[i] = new int[N];

for (int j = 0; j < N; j++)

cin >> matrix[i][j];

}

cout << endl;

//Строим систему замыканий

vector <vector <int> > closureSystem;

closureSystem.push\_back(objects);

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

vector <int> srez = cut(matrix, N, i, "column");

for (int j = 0; j < closureSystem.size(); j++) {

vector <int> set = generalEl(closureSystem[j], srez);

if (unique(closureSystem, set))

closureSystem.push\_back(set);

}

}

cout << "Система замыканий: {";

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

if (i == closureSystem.size() - 1) {

printVec(closureSystem[i]);

cout << "} \n\n";

}

else {

printVec(closureSystem[i]);

cout << ", ";

}

}

//Распологаем элементы по уровням

vector <vector <int> > copy = closureSystem;

vector <vector <vector <int> > > lvls;

while (!copy.empty())

findMaxSet(lvls, copy);

//Строим диаграмму Хассе

cout << "Диаграмма Хассе: \n";

int size = lvls.size();

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

cout << size - i << "-й уровень: ";

int iSize = lvls[i].size();

for (int j = 0; j < iSize; j++) {

if (j == iSize - 1)

printVec(lvls[i][j]);

else {

printVec(lvls[i][j]);

cout << ", ";

}

}

cout << endl;

}

//Строим диаграмму Хассе с указателями на связанные элементы

cout << "\nДиаграмма Хассе с указателями на связанные элементы: \n";

size = lvls.size();

for (int i = 0; i < size - 1; i++) {

cout << size - i << "-й уровень: ";

int iSize = lvls[i].size();

for (int j = 0; j < iSize; j++) {

copy.resize(0);

cout << "(";

printVec(lvls[i][j]);

cout << ", <<<";

for (int k = 0; k < lvls[i + 1].size(); k++)

if (generalEl(lvls[i][j], lvls[i + 1][k]) == lvls[i + 1][k] || lvls[i + 1][k].empty())

copy.push\_back(lvls[i + 1][k]);

int copySize = copy.size();

for (int k = 0; k < copySize; k++) {

if (k == copySize - 1) {

printVec(copy[k]);

cout << ">>>)";

}

else {

printVec(copy[k]);

cout << ", ";

}

}

if (j != iSize - 1)

cout << ", ";

}

cout << endl;

}

cout << "1-й уровень: ";

for (int i = 0; i < lvls.back().size(); i++)

printVec(lvls.back()[i]);

//Решётка концептов

cout << "\n\nРешётка концептов: ";

size = closureSystem.size();

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

if (i == size - 1) {

cout << "(";

printVec(closureSystem[i]);

cout << ", ";

printConc(matrix, N, attr, closureSystem[i]);

cout << ") \n";

}

else {

cout << "(";

printVec(closureSystem[i]);

cout << ", ";

printConc(matrix, N, attr, closureSystem[i]);

cout << "), ";

}

}

//Диграмма Хассе решётки концептов

cout << "\nДиграмма Хассе решётки концептов: \n";

size = lvls.size();

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

cout << size - i << "-й уровень: ";

int iSize = lvls[i].size();

for (int j = 0; j < iSize; j++) {

copy.resize(0);

cout << "(";

printVec(lvls[i][j]);

cout << ", ";

printConc(matrix, N, attr, lvls[i][j]);

cout << ")";

if (j < iSize - 1)

cout << ", ";

}

cout << endl;

}

}

int main() {

setlocale(LC\_ALL, "ru");

for (;;) {

cout << "С каким отношением работаем? \n1 - Отношение эквивалентности \n2 - Отношение порядка, ввод числом \n3 - Отношение порядка, ввод матрицей \n";

cout << "4 - Решётка концептов \n5 - Выход \n";

int x;

cin >> x;

switch (x) {

case 1:

work\_equivalence();

cout << endl;

break;

case 2:

work\_order\_number();

cout << endl;

break;

case 3:

work\_order\_matrix();

cout << endl;

break;

case 4:

work\_concept();

cout << endl;

break;

case 5:

return 0;

}

}

}

ЗАКЛЮЧЕНИЕ

В ходе данной работы были изучены и реализованы алгоритмы построения эквивалентного замыкания бинарного отношения, фактор-множества, системы представителей фактор-множества, вычисление минимальных(максимальных) и наименьших(наибольших) элементов множества, а также построение диаграммы Хассе, системы замыкания и решётки концептов.