МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

"Нижегородский государственный университет им. Н.И. Лобачевского"

Институт информационных технологий, математики и механики

Кафедра информатики и автоматизации научных исследований

Направление подготовки:

«Прикладная информатика»

ОТЧЕТ ПО ТИПОВОМУ КУРСОВОМУ ПРОЕКТУ

ТЕМА

**РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМА**

**ВЕТВЕЙ И ГРАНИЦ ДЛЯ ТИПОВОЙ ЗАДАЧИ №2**

Выполнили:

студенты 3 курса, гр. 381507-2

Суворов Кирилл \_\_\_\_\_\_\_\_\_\_\_

Мячев Анатолий \_\_\_\_\_\_\_\_\_\_

Научный руководитель:

Старостин Н.В.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(подпись)

Нижний Новгород

2018

**Введение**

На одном предприятии произошла модернизация рабочего процесса: закупили и подключили новые рабочие станции между собой. Кроме того, на работу претендует некоторое количество специалистов, причем каждый из них имеет определенный навык по работе на данной аппаратуре. Однако, для успешного взаимодействия важен не столько опыт конкретного оператора, сколько разница с его соседями, поскольку между станциями есть возможность предложить свою помощь или запросить ее у другого. Перед работодателем встал вопрос: Кого из претендентов взять на работу и как рассадить их по рабочим местам, что бы эффективность работы была максимальная.

Рассмотрим задачу об оптимальной рассадке работников по местам. Известно, что на предприятии есть определённое количество рабочих станций, а также сеть их взаимодействия между собой. Некоторые машины имеют достаточный приоритет для начала коммуникации с другой станицей. Кроме того, известно множество потенциальных работников, обладающих определёнными навыками, которая выражается численной характеристикой. Необходимо таким образом назначить людей на рабочие места, чтобы взаимодействие между всеми работниками было максимально. Причем, взаимодействие между двумя работникам находится как разница в их навыках, а общее взаимодействие находится как сумма всех возможных взаимосвязей между работниками.

**Цель курсового проекта**

Разработка алгоритма, основанного на классическом методе ветвей и границ, обеспечивающий точное решение задачи дискретной оптимизации.

**Этапы работ:**

- Построить формальную постановку задачи

- Продумать алгоритм решения задачи

- Выполнить программную реализацию алгоритма

- Провести вычислительный эксперимент

- Проанализировать полученный результат

1. **Формальная постановка задачи**
   1. Исходные данные задачи

на вход алгоритм получает следующие входные данные:  
 – граф рабочих мест, где:

*–* множество рабочих мест

- множество связей между рабочими местами

- численная характеристика навыков i-го сотрудника

* 1. Математическая модель

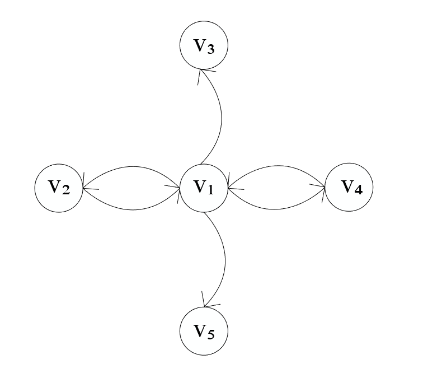
Варьируемый параметры:  
 – это характеристика сотрудника на i-ом месте

Ограничения модели:

* 1. Критерий задачи
  2. Комбинаторная и математическая сложность

Допустимым решением является выборка без повторений из k характеристик (), размещенных на n вершин (n! вариантов). Таким образом комбинаторная сложность равна . Алгоритм, отличный от полного перебора, не был найден, поэтому задача относится к классу NP.

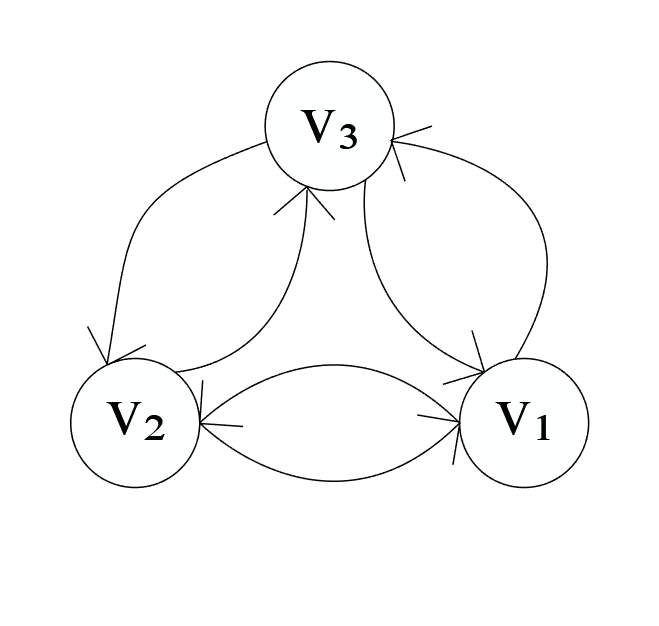
* 1. Индивидуальные постановки задач

1. Дан граф расположения рабочих мест

И множество работников

Недопустимое решение:

Допустимое решение (так как элементы не повторяются):

1. Дан граф расположения рабочих мест

И множество работников

Недопустимое решение:

Допустимое решение (так как элементы не повторяются):

1. **Алгоритм решения задачи**
   1. Общая идея:

Для решения поставленной задачи мы взяли за основу идею метода ветвей и границ: последовательное использование конечности множества вариантов решений задачи и замена полного перебора сокращенным, направленным перебором. Полного перебора удается избежать за счет отбрасывания “неперспективных” множеств вариантов, т.е. таких, которые заведомо не могут содержать решения “лучшего”, чем решения, оставшиеся в не отброшенном множестве.[[1]](#footnote-1)

Данный метод состоит из нескольких основных процедур: ветвление, оценка, отсев и останов. Далее рассмотрим каждую процедуру более детально.

* 1. Процедура Ветвления

Процедура ветвления заключается в разбиении множества решений на более мелкие подмножества, которые составляют дерево поиска оптимального решения. В данном случае мы используем ветвление по численным характеристикам навыков сотрудников. Таким образом на i-ом уровне дерева выбирается работник, который будет размещаться на i-ом рабочем месте.

* 1. Процедура оценки

Для каждого множества решений, находящегося в дереве поиска мы находим верхнюю(V) и нижнюю(H) оценку возможных решений в данном множестве. Так как наша задача поставлена с критерием на максимум, то нижняя оценка будет достижимой на некотором допустимом решении, а верхняя, в большинстве случаях, не достижимая.

* + 1. Нижняя оценка

Нижнюю оценку находим как допустимое решение на множестве решений из дерева поиска. Возьмем не занятое рабочее место наименьшего номера из множества V и проведем соответствие с произвольной, не использованной ранее (например, первой по порядку) численной характеристикой из множества W. Таким образом мы занимаем все рабочие места и получаем допустимое решение (X’). Критерий этого допустимого решения и будет нижней оценкой рассматриваемого множества решений из дерева поиска, то есть V=F(X’).

* + 1. Верхняя оценка

Q – Множество “назначенных вершин”

– минимальная характеристика среди назначенных

– максимальная характеристика среди назначенных

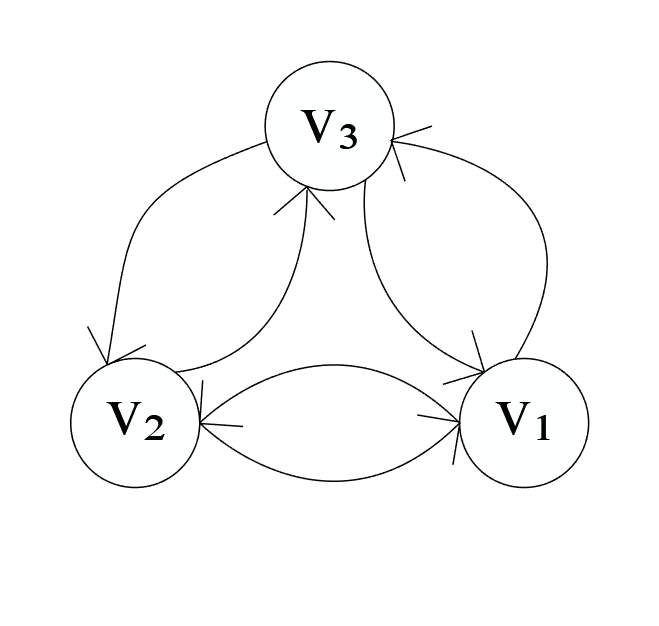
По формуле:

* 1. Процедура отсева.

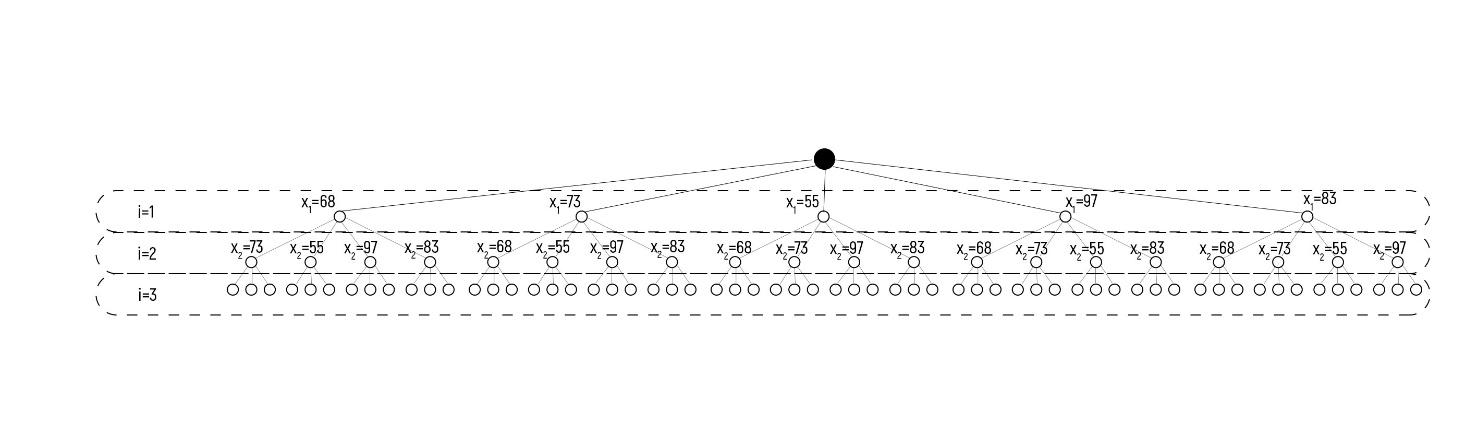
Данная процедура является универсальной для метода ветвей и границ. Пусть у нас есть 2 множества решений, для которого подсчитаны верхние и нижние оценки: . В таком случае, если верхняя оценка одного множества меньше или равна нижней оценке другого множества (, то оно является неперспективным. Это означает, что во втором множестве можно гарантированно найти решение лучше, чем из первого, поэтому неперспективные множества будем отбрасывать.

* 1. Процедура останова.

Процедура останова так же является универсальной процедурой метода ветвей и границ. Эта процедура позволяет алгоритму закончить работу. Если в дереве поиска осталось одно не отброшенное множество решений, в котором верхняя и нижняя оценка совпадают (V=H), то задача решена и максимальное значение критерия, будет определяться достижимой оценкой, в нашем случае – нижней.

* 1.  Пример работы алгоритма

Рассмотрим граф расположения рабочих мест

И множество работников

Строим дерево поиска:

Обходим дерево поиска решений методом поиска в ширину:

* Находим верхнюю и нижнюю оценку для выбранного множества решений
* Оцениваем перспективность дальнейшего разбиения множества

Если не перспективно – отбрасываем

* Переходим к следующему элементу

Таким образом, получается дерево решений, которое представлено в приложении №2.

1. **Программная реализация и эксперимент**

3.1 Функциональные возможности программы

Программа запрашивает у пользователя название файла, в котором находится описание входных данных в следующем виде:

* Количество рёбер
* Список рёбер
* Количество характеристик
* Список характеристик

Проверяет входные данные на корректность (на уникальность характеристик, на повторение рёбер).

Решает задачу методом ветвей и границ с отсечением неперспективных вариантов и выводит найденное решение со значением критерия на нём.

После этого программа спрашивает пользователя: продолжать работу или нет?

3.2 Основные функции

* Проверка входных данных на корректность

|  |
| --- |
| bool verify\_input\_data(const graph& grh, const characteristics& v\_chr) |

Этот метод принимает граф и список характеристик, проверяет их на корректность, возвращает true в случае корректных данных и false иначе.

* Подсчет нижний оценки

|  |
| --- |
| int method\_branches\_borders::lower\_bound(const vector<int16\_t>& \_v\_app) |

Этот метод принимает список проведённых назначений, считает нижнюю оценку и возвращает её.

* Подсчет верхней оценки

|  |
| --- |
| int method\_branches\_borders::upper\_bound(const vector<int16\_t>& v\_app) |

Этот метод принимает список проведённых назначений, считает верхнюю оценку и возвращает её.

* Метод ветвей и границ

|  |
| --- |
| pair<node\_decisions\_tree, int>\* method\_branches\_borders::process() |

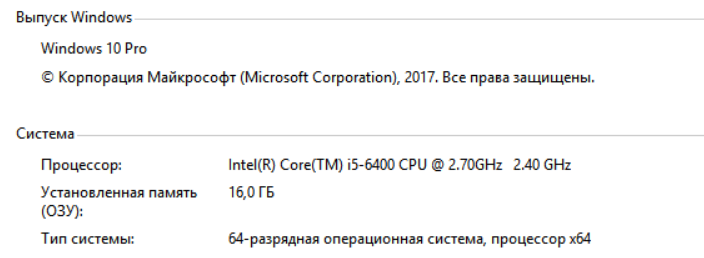
Это метод строит дерево поиска решения метода ветвей и границ и возвращает найденное решение и значение критерия на нём.

3.3. Методика проведения эксперимента

Было сконструировано 6 моделей, для проверки и анализа работы данного алгоритма.

В 1-ом тесте мы взяли граф из 3 вершин и 6 ребер, а также множество из 5 характеристик. Критерий и оптимальное решение было известно заранее. Поэтому мы можем убедиться, что ответ, полученный алгоритмом, является верным.

Далее, для оценки отличия данного алгоритма от полного перебора, были произведены ряд тестовых запусков с разными входными параметрами. Для этого было посчитано количество просматриваемых элементов, элементов в полном дереве поиска оптимального решения и измерено время работы программы.

Все запуски программ были произведены на данной машине: 

3.4. Результаты экспериментов

Обработав все 6 моделей были получены следующие результаты:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| № | Кол-во вершин | Кол-во ребер | Коли-во характеристик | Коли-во элементов в дереве решений | Кол-во элементов в полном дереве поиска | Время работы | % выигрыша по памяти |
| 1 | 3 | 6 | 5 | 44 | 86 | 5мс | 48,8372093 |
| 2 | 6 | 11 | 9 | 8759 | 79210 | 19мс | 88,94205277 |
| 3 | 8 | 16 | 9 | 50150 | 623530 | 61мс | 91,95708306 |
| 4 | 8 | 16 | 11 | 239455 | 8713112 | 239мс | 97,25178558 |
| 5 | 10 | 20 | 13 | 22809479 | 1359245486 | 22сек | 98,32190143 |
| 6 | 10 | 20 | 15 | 96058626 | 13009392076 | 110сек | 99,26162095 |

По данным результатам мы можем видеть, что эффективность выбранного алгоритма увеличивается пропорционально размерности задачи, несмотря на внушительное время работы программы. Кроме того, можно заметить, что количество пройденных элементов, а, следовательно, время работы программы, зависит в большей степени от количества характеристик, нежели от количества ребер в исходном графе. Кроме того, количество ребер практически не заметно влияет на скорость работы. Это происходит потому, что от них не зависит ветвление дерева поиска оптимального решения, а лишь изменяет скорость нахождения оценок.

Проанализировав полученные данные, была получена примерная оценка времени работы алгоритма: О(k!/(n - k)!) \* O(m + nk)

**Заключение**

В рамках данной работы была формализована задача об оптимальной рассадке работников по местам: поставлен критерий и сформулированы ограничения. Кроме того, алгоритм решения был сконструирован и реализован на языке С++.

Полученный алгоритм, позволил сократить полный перебор, что привело к сокращению времени работы. Однако, при больших размерностях нахождение ответа довольно затруднительно, ведь полиномиального алгоритма для этой задачи на данный момент не известно. Поэтому работы в этом направлении могут продолжаться. Например, для данной интерпретации метода ветвей и границ, можно найти более точную процедуру нахождения верхней оценки, что позволит отсечь большее количество неперспективных направлений.

**Список используемой литературы:**

* http://ru.cppreference.com – справочник по языку программирования
* http://edu.nstu.ru/courses/saod/bfs.htm - Метод обхода дерева в ширину
* Корбут А.А., Финкельштейн Ю.Ю. Дискретное программирование
* Методическое пособие по курсу "Математические основы информатики" для студентов очно-заочного отделения факультета ВМК специальности "Прикладная информатика". Часть 3. / Нижег. гос. ун-т, 2000, с.118. М.Х. Прилуцкий
* «О сложности вычисления» А.А. Разборов, Математическое просвещение. — МЦНМО, 1999. — № 3. — С. 127—141.

**Приложение №1.**

В данном приложении представлен код программы. Он состоит из 5 частей:

“graph.h”

#pragma once

#include <iterator>

#include <iostream>

#include <stdint.h>

#include <vector>

#include <deque>

using std::istream\_iterator;

using std::pair;

using std::cout;

using std::endl;

using std::vector;

using std::deque;

typedef pair<int16\_t, int16\_t> edge; //первый элемент пары - начало ребра; второй - конец ребра

/\*Вершина ~ рабочее место; ребро ~ связь между местами; edges[i] - список ребёр идущих из вершины номера i + 1\*/

class graph {

int16\_t count\_vertex;

int16\_t copy\_counter; //кол-во копий графа

vector<vector<edge>>\* edges;

public:

graph(vector<vector<edge>>\* \_edges) :edges(\_edges), count\_vertex(\_edges->size()),

copy\_counter(1) {};

graph(const graph& grh);

int16\_t get\_count\_vertex() const { return count\_vertex; };

const vector<vector<edge>>\* get\_edges() const { return edges; };

void print();

~graph();

};

/\*Вектор характеристик рабочих\*/

class characteristics : public vector<int16\_t> {

public:

characteristics(istream\_iterator<int16\_t>& iter) {

int count\_chr = \*(iter++);

for (int i = 0; i < count\_chr; ++i)

this->push\_back(\*(iter++));

}

characteristics() : vector<int16\_t>() {};

void print();

void bubble\_sort();

};

/\*v\_app - вектор назначений; элемент v\_app[i] - характеристика назначенная на i + 1 - ое место

v\_app может содержать назначения не для всех вершин\*/

struct node\_decisions\_tree {

vector<int16\_t> v\_app;

node\_decisions\_tree(): v\_app(vector<int16\_t>()) {}

node\_decisions\_tree(vector<int16\_t> \_v\_app) : v\_app(\_v\_app.begin(), \_v\_app.end()) {};

};

/\*Контейнер позволяющий эмулировать обход в ширину\*/

class deque\_pnode : public deque<node\_decisions\_tree\*> {

public:

~deque\_pnode() {

for (deque<node\_decisions\_tree\*>::iterator iter = this->begin(); iter != this->end(); ++iter)

delete \*iter;

}

};

class method\_branches\_borders {

graph\* grh;

characteristics\* v\_chr;

size\_t size\_chr;

size\_t size\_edges;

size\_t\* hash\_sizes; //в (\* hash\_sizes + i) хранится кол-во рёбер для i + 1 вершины

public:

method\_branches\_borders(graph\* \_grh, characteristics\* \_v\_chr) : grh(\_grh), v\_chr(\_v\_chr),

size\_chr(v\_chr->size()), size\_edges(\_grh->get\_count\_vertex()) {

size\_t count = \_grh->get\_count\_vertex();

hash\_sizes = new size\_t[count];

const vector<vector<edge>>\* edges = grh->get\_edges();

for (register int i = 0; i < count; ++i)

\*(hash\_sizes + i) = (\*edges)[i].size();

};

pair<node\_decisions\_tree, int>\* process();

void complete\_best\_solution(vector<int16\_t>& v\_app);

int lower\_bound(const vector<int16\_t>& \_v\_app);

int upper\_bound(const vector<int16\_t>& \_v\_app);

bool find(const vector<int16\_t>& v\_app, int16\_t check\_chr);

~method\_branches\_borders() {

delete[] hash\_sizes;

}

};

bool verify\_input\_data(const graph& grh, const characteristics& v\_chr);

vector<vector<edge>>\* read\_edges(istream\_iterator<int16\_t>& iter);

void format\_input\_data();

“graph.cpp”

#include "graph.h"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*graph\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

graph::graph(const graph & grh)

{

this->copy\_counter = grh.copy\_counter;

this->count\_vertex = grh.count\_vertex;

this->edges = grh.edges;

++copy\_counter;

}

void graph::print() {

cout << "---------------graph------------------" << endl;

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

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

cout << (\*edges)[i][j].first << " " << (\*edges)[i][j].second << endl;

}

}

cout << "---------------graph------------------" << endl;

}

graph::~graph() {

--copy\_counter;

if (copy\_counter == 0) delete edges;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*characteristics\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void characteristics::print() {

cout << endl << "characteristics: ";

for (register int i = 0; i < this->size(); ++i)

cout << (\*this)[i] << " ";

cout << endl;

}

void characteristics::bubble\_sort() {

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

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

if ((\*this)[j] > (\*this)[j + 1]) {

(\*this)[j] ^= ((\*this)[j + 1] ^= (\*this)[j]);

(\*this)[j + 1] ^= (\*this)[j];

}

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*auxiliary\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

bool verify\_input\_data(const graph& grh, const characteristics& v\_chr) {

if (grh.get\_count\_vertex() > v\_chr.size()) return false;

for (int i = 0; i < v\_chr.size(); ++i) // Проверка элементов на уникальность

for (register int j = 0; j < v\_chr.size(); ++j)

if (i != j)

if (v\_chr[i] == v\_chr[j]) return false;

const vector<vector<edge>>\* edges = grh.get\_edges();

for (int i = 0; i < edges->size(); ++i) // Проверяем рёбра на уникальность

for (int j = 0; j < (\*edges)[i].size(); ++j)

for (register int k = 0; j != k, k < (\*edges)[i].size(); ++k)

if (j != k)

if (((\*edges)[i][j].first == (\*edges)[i][k].first) && ((\*edges)[i][j].second == (\*edges)[i][k].second))

return false;

return true;

}

vector<vector<edge>>\* read\_edges(istream\_iterator<int16\_t>& iter) {

edge \_edge;

size\_t size;

int16\_t count\_new\_vertex;

vector<vector<edge>>\* edges = new vector<vector<edge>>();

int count\_edges = \*(iter++);

for (int i = 0; i < count\_edges; ++i) {

\_edge.first = \*(iter++);

\_edge.second = \*(iter++);

size = edges->size();

if ((size < \_edge.first) || (size < \_edge.second)) { //добавляем для новых вершин списки рёбер

count\_new\_vertex = (\_edge.first > \_edge.second) ? (\_edge.first - size) :

(\_edge.second - size);

for (int i = 0; i < count\_new\_vertex; ++i) {

edges->push\_back(std::vector<edge>());

}

}

(\*edges)[\_edge.first - 1].push\_back(\_edge);

}

return edges;

}

vector<int16\_t>\* read\_characteristics(istream\_iterator<int16\_t>& iter) {

vector<int16\_t>\* vect = new vector<int16\_t>();

int count\_chr = \*(iter++);

for (int i = 0; i < count\_chr; ++i)

vect->push\_back(\*(iter++));

return vect;

}

/\*Ищет элемент check\_chr в v\_app\*/

bool method\_branches\_borders::find(const vector<int16\_t>& v\_app, int16\_t check\_chr)

{

size\_t size = v\_app.size();

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

if (v\_app[i] == check\_chr) return true;

return false;

}

void format\_input\_data() {

cout << "--------------input data format-----------------" << endl;

cout << "Count edges:" << endl

<< "list of edges:" << endl

<< "Count characteristics:" << endl

<< "list of characteristics" << endl;

cout << "--------------input data format-----------------" << endl;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*method\_branches\_borders\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*Метод ветвей и ганиц\*/

pair<node\_decisions\_tree, int>\* method\_branches\_borders::process()

{

node\_decisions\_tree\* temp\_node = nullptr;

deque\_pnode algorithm\_path;

node\_decisions\_tree\* best\_node = new node\_decisions\_tree();

int best\_lower\_bound = lower\_bound(best\_node->v\_app);

int current\_upper\_bound = 0;

int current\_lower\_bound = 0;

algorithm\_path.push\_back(best\_node);

for (int i = 0; i < algorithm\_path.size(); ++i) {

current\_upper\_bound = upper\_bound(algorithm\_path[i]->v\_app);

current\_lower\_bound = lower\_bound(algorithm\_path[i]->v\_app);

if (current\_upper\_bound <= best\_lower\_bound) continue;

if (current\_lower\_bound > best\_lower\_bound) {

best\_lower\_bound = current\_lower\_bound;

best\_node = algorithm\_path[i];

}

if (current\_lower\_bound == current\_upper\_bound) continue;

//if (algorithm\_path[i]->v\_app.size() == size\_edges) continue;

for (int j = 0; j < size\_chr; ++j) {

if (find(algorithm\_path[i]->v\_app, (\*v\_chr)[j]) == false) {

temp\_node = new node\_decisions\_tree(algorithm\_path[i]->v\_app);

temp\_node->v\_app.push\_back((\*v\_chr)[j]);

algorithm\_path.push\_back(temp\_node);

}

}

}

cout << "The number of elements in the tree: " << algorithm\_path.size() << endl;

return new pair<node\_decisions\_tree, int>(\*best\_node, best\_lower\_bound);

}

/\*Чтобы посчитать нижнюю оценку нужно достроить решение\*/

int method\_branches\_borders::lower\_bound(const vector<int16\_t>& \_v\_app) {

vector<int16\_t> v\_app(\_v\_app.begin(), \_v\_app.end());

for (int i = 0; v\_app.size() != size\_edges; ++i) {

if (find(v\_app, (\*v\_chr)[i]) == false) v\_app.push\_back((\*v\_chr)[i]);

}

const vector<vector<edge>>\* edges = grh->get\_edges();

edge temp\_edge;

int sum = 0;

int temp = 0;

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

for (int j = 0; j < hash\_sizes[i]; ++j) {

temp\_edge = (\*edges)[i][j];

temp = v\_app[temp\_edge.first - 1] - v\_app[temp\_edge.second - 1];

sum += abs(temp);

}

}

return sum;

}

/\*w\_max, w\_min - максимальная и минимальная характеристики во множестве неназначенных характеристик

v\_chr - упорядоченное(от меньшего к большему) множество характеристик

check1, check2 - 1(2) показывает, что вершина среди начначенных; 0 - не назначена\*/

int method\_branches\_borders::upper\_bound(const vector<int16\_t>& v\_app) {

int16\_t w\_max = 0;

int16\_t w\_min = 0;

for (int i = size\_chr - 1; i >= 0; --i)

if (find(v\_app, (\*v\_chr)[i]) == false) {

w\_max = (\*v\_chr)[i];

break;

}

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

if (find(v\_app, (\*v\_chr)[i]) == false) {

w\_min = (\*v\_chr)[i];

break;

}

edge temp\_edge;

int sum = 0;

int temp;

int check1;

int check2;

size\_t size\_v\_app = v\_app.size();

const vector<vector<edge>>\* edges = grh->get\_edges();

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

for (int j = 0; j < hash\_sizes[i]; ++j) {

temp\_edge = (\*edges)[i][j];

check1 = ((temp\_edge.first - 1) < size\_v\_app) ? 1 : 0;

check2 = ((temp\_edge.second - 1) < size\_v\_app) ? 2 : 0;

switch (check1 + check2) {

case 0:

temp = w\_max - w\_min;

break;

case 1:

temp = v\_app[temp\_edge.first - 1] -

((abs(v\_app[temp\_edge.first - 1] - w\_max) > abs(v\_app[temp\_edge.first - 1] - w\_min)) ?

w\_max : w\_min);

break;

case 2:

temp = v\_app[temp\_edge.second - 1] -

((abs(v\_app[temp\_edge.second - 1] - w\_max) > abs(v\_app[temp\_edge.second - 1] - w\_min)) ?

w\_max : w\_min);

break;

case 3:

temp = v\_app[temp\_edge.first - 1] - v\_app[temp\_edge.second - 1];

break;

}

sum += abs(temp);

}

}

return sum;

}

/\*Метод ветвей и границ может закончиться и вернуть решение в недостроенном виде;

достроим его тем же способом, который использутся при построении допустимого решение в lower\_bound\*/

void method\_branches\_borders::complete\_best\_solution(vector<int16\_t>& v\_app)

{

size\_t count\_vertex = grh->get\_count\_vertex();

for (int i = 0; v\_app.size() != count\_vertex; ++i) {

if (find(v\_app, (\*v\_chr)[i]) == false) v\_app.push\_back((\*v\_chr)[i]);

}

}

“main.cpp”

#include "graph.h"

#include "graph\_test.h"

#include <chrono>

#include <fstream>

#include <string>

using std::ifstream;

using std::string;

using std::cin;

int main() {

format\_input\_data();

string continue\_work;

string name\_file;

do {

cout << endl << "enter the full file name" << endl;

cin >> name\_file;

auto start\_time = std::chrono::steady\_clock::now();

ifstream fin(name\_file);

if (fin.is\_open()) {

istream\_iterator<int16\_t> iter(fin);

graph grh(read\_edges(iter));

characteristics v\_chr(iter);

if (verify\_input\_data(grh, v\_chr) == false) {

cout << "input data do not satisfy the restriction of injection" << endl;

system("pause");

continue;

}

v\_chr.bubble\_sort();

grh.print();

v\_chr.print();

method\_branches\_borders alg(&grh, &v\_chr);

pair<node\_decisions\_tree, int>\* result = alg.process();

alg.complete\_best\_solution(result->first.v\_app);

cout << "criterion value: " << result->second << endl;

cout << "--------------decision----------------" << endl;

for (int i = 0; i < result->first.v\_app.size(); ++i) {

cout << (i + 1) << " " << result->first.v\_app[i] << endl;

}

cout << "--------------decision----------------" << endl;

auto end\_time = std::chrono::steady\_clock::now();

auto elapsed\_ns = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_time - start\_time);

cout << "time working: " << elapsed\_ns.count() << " ms\n";

delete result;

}

else {

cout << "file with this name does not exist" << endl;

system("pause");

}

cout << "continue work? (y or n): " << endl;

cin >> continue\_work;

} while (continue\_work == "y");

return 0;

}

“graph\_test.h”

#pragma once

#include "graph.h"

bool test\_verify\_input\_data();

bool test\_lower\_bound();

bool test\_upper\_bound();

bool test\_process();

“graph\_test.cpp”

#include "graph\_test.h"

bool test\_verify\_input\_data() {

bool check\_test = true;

vector<vector<edge>>\* edges = new vector<vector<edge>>();

edges->push\_back(vector<edge>());

edges->push\_back(vector<edge>());

(\*edges)[0].push\_back(edge(1, 2));

(\*edges)[1].push\_back(edge(2, 1));

graph grh(edges);

characteristics chr;

chr.push\_back(1); chr.push\_back(2); chr.push\_back(4); chr.push\_back(7);

if (verify\_input\_data(grh, chr) != true) check\_test = false;

(\*edges)[0].push\_back(edge(1, 2));

if (verify\_input\_data(grh, chr) != false) check\_test = false;

(\*edges)[0].pop\_back();

chr.push\_back(4);

if (verify\_input\_data(grh, chr) != false) check\_test = false;

chr.pop\_back();

vector<vector<edge>>\* edges1 = new vector<vector<edge>>();

edges1->push\_back(vector<edge>());

edges1->push\_back(vector<edge>());

edges1->push\_back(vector<edge>());

edges1->push\_back(vector<edge>());

edges1->push\_back(vector<edge>());

(\*edges1)[1].push\_back(edge(2, 1));

(\*edges1)[2].push\_back(edge(3, 1));

(\*edges1)[3].push\_back(edge(4, 1));

(\*edges1)[4].push\_back(edge(5, 1));

graph grh1(edges1);

if (verify\_input\_data(grh1, chr) != false) check\_test = false;

return check\_test;

}

bool test\_lower\_bound() {

bool check\_test = true;

vector<vector<edge>>\* edges = new vector<vector<edge>>();

edges->push\_back(vector<edge>()); edges->push\_back(vector<edge>());

edges->push\_back(vector<edge>()); edges->push\_back(vector<edge>());

(\*edges)[0].push\_back(edge(1, 2));

(\*edges)[1].push\_back(edge(2, 3));

(\*edges)[2].push\_back(edge(3, 4));

(\*edges)[3].push\_back(edge(4, 2));

graph grh(edges);

characteristics chr;

chr.push\_back(1); chr.push\_back(2); chr.push\_back(4); chr.push\_back(7);

method\_branches\_borders mbb(&grh, &chr);

vector<int16\_t> v\_app;

v\_app.push\_back(1); v\_app.push\_back(2); v\_app.push\_back(4); v\_app.push\_back(7);

if (mbb.lower\_bound(v\_app) != 11) check\_test = false;

vector<int16\_t> v\_app1;

v\_app1.push\_back(4); v\_app1.push\_back(7); v\_app1.push\_back(1); v\_app1.push\_back(2);

if (mbb.lower\_bound(v\_app1) != 15) check\_test = false;

vector<int16\_t> v\_app2;

v\_app2.push\_back(4); v\_app2.push\_back(2);

if (mbb.lower\_bound(v\_app2) != 14) check\_test = false;

return check\_test;

}

bool test\_upper\_bound() {

bool check\_test = true;

vector<vector<edge>>\* edges = new vector<vector<edge>>();

edges->push\_back(vector<edge>()); edges->push\_back(vector<edge>());

edges->push\_back(vector<edge>()); edges->push\_back(vector<edge>());

(\*edges)[0].push\_back(edge(1, 2));

(\*edges)[1].push\_back(edge(2, 3));

(\*edges)[2].push\_back(edge(3, 4));

(\*edges)[3].push\_back(edge(4, 2));

graph grh(edges);

characteristics chr;

chr.push\_back(1); chr.push\_back(2); chr.push\_back(4); chr.push\_back(7);

method\_branches\_borders mbb(&grh, &chr);

vector<int16\_t> v\_app;

v\_app.push\_back(1);

if (mbb.upper\_bound(v\_app) != 21) check\_test = false;

v\_app.push\_back(4);

if (mbb.upper\_bound(v\_app) != 14) check\_test = false;

return check\_test;

}

bool test\_process() {

bool check\_test = true;

vector<vector<edge>>\* edges = new vector<vector<edge>>();

edges->push\_back(vector<edge>()); edges->push\_back(vector<edge>());

edges->push\_back(vector<edge>()); edges->push\_back(vector<edge>());

(\*edges)[0].push\_back(edge(1, 2));

(\*edges)[1].push\_back(edge(2, 3));

(\*edges)[2].push\_back(edge(3, 4));

(\*edges)[3].push\_back(edge(4, 2));

graph grh(edges);

characteristics chr;

chr.push\_back(1); chr.push\_back(2); chr.push\_back(4); chr.push\_back(7);

method\_branches\_borders mbb(&grh, &chr);

pair<node\_decisions\_tree, int>\* decision = mbb.process();

if (decision->second != 17) check\_test = false;

if (decision->first.v\_app[0] != 2) check\_test = false;

if (decision->first.v\_app[1] != 7) check\_test = false;

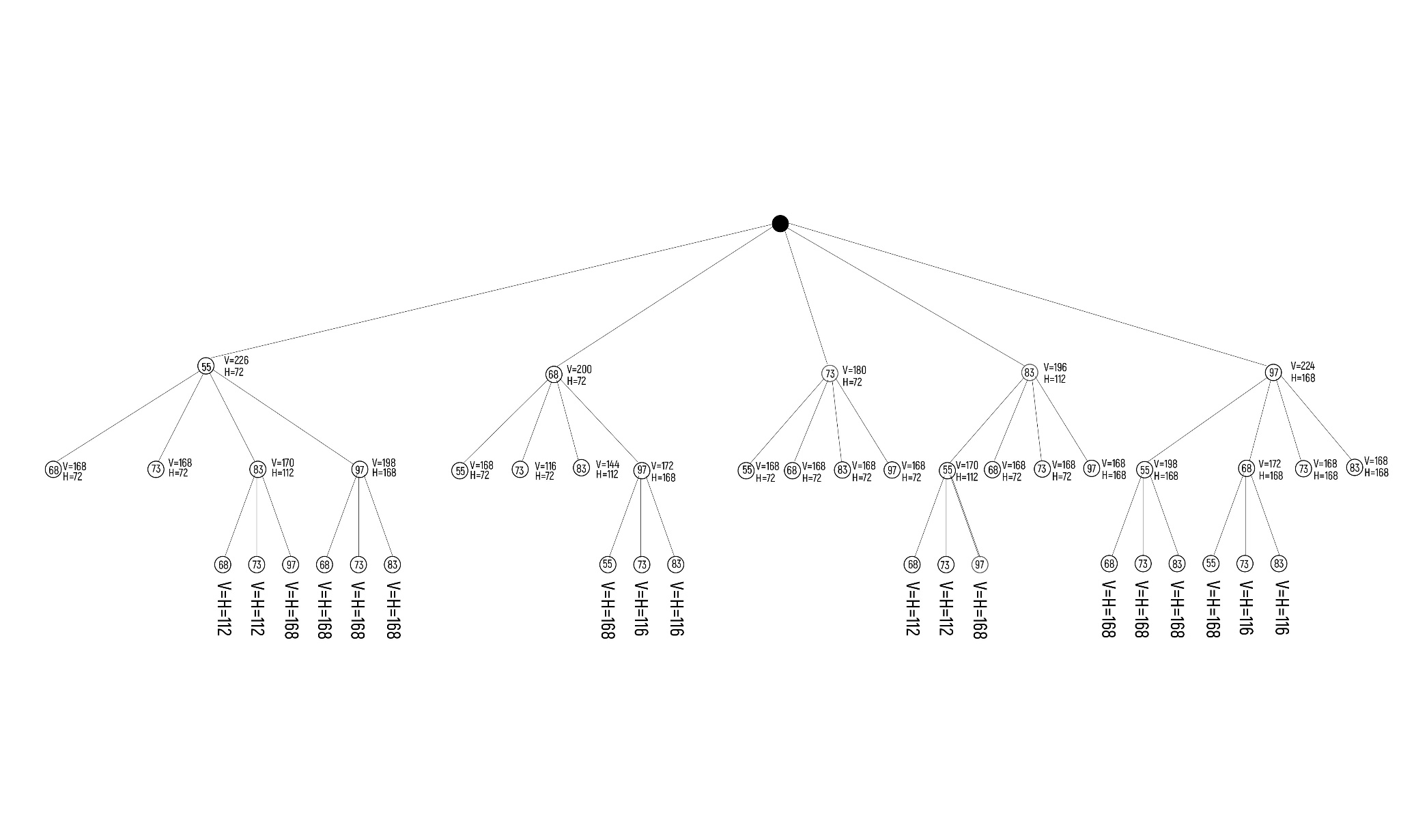
if (decision->first.v\_app[2] != 4) check\_test = false;

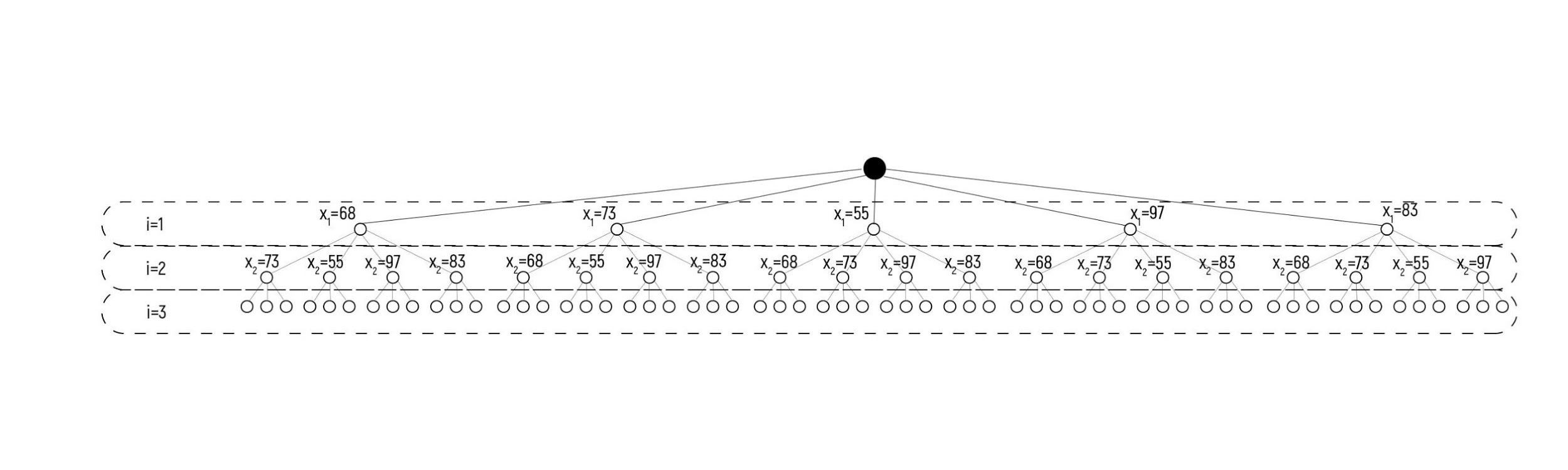
if (decision->first.v\_app[3] != 1) check\_test = false;

return true;

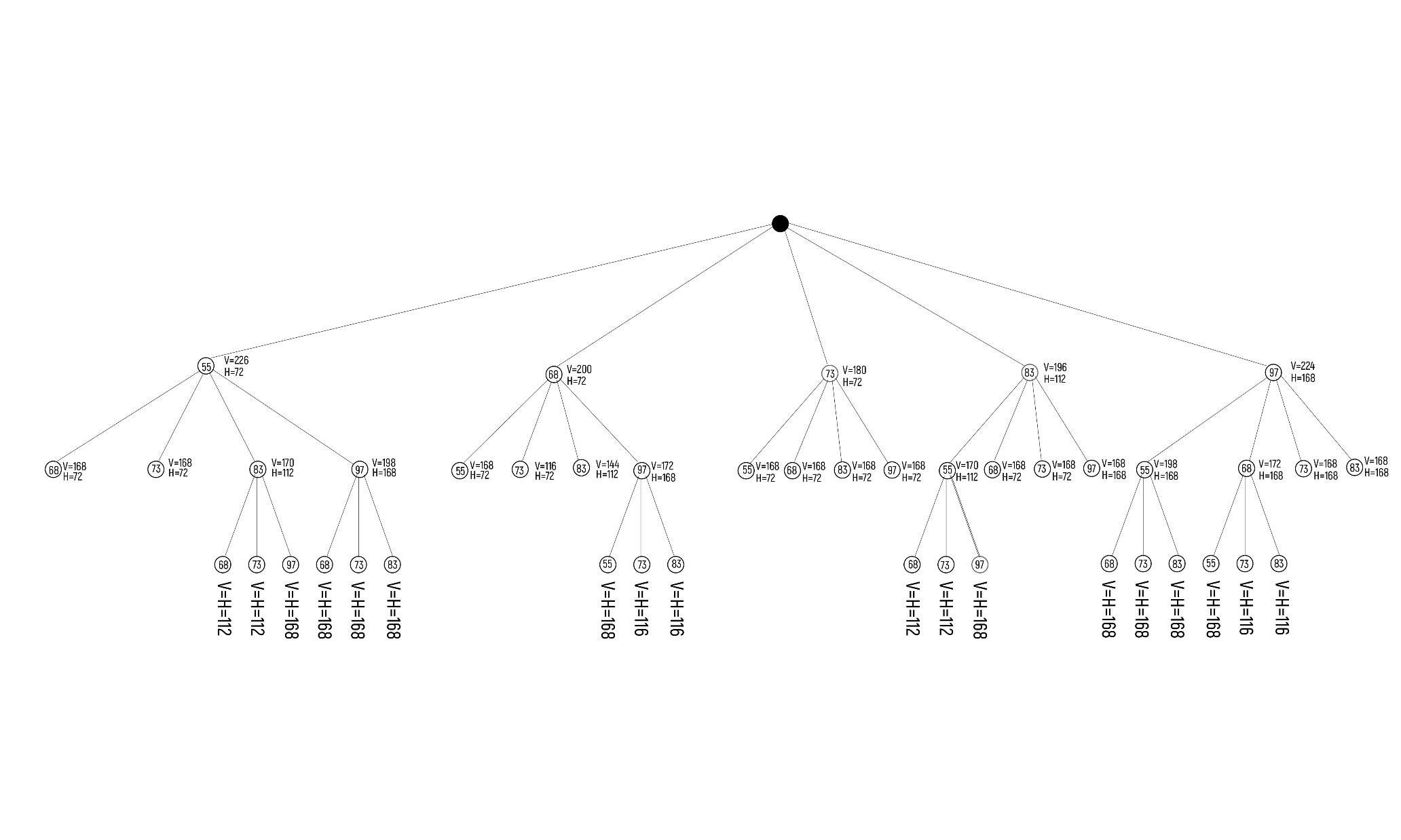
}

**Приложение №2.**

Дерево решений для примера работы алгоритма (2.6.) :

Приложение №2

Изображение дерева решений, для примера работы алгоритма:



1. Методическое пособие по курсу "Математические основы информатики" для студентов очно-заочного отделения факультета ВМК специальности "Прикладная информатика". Часть 3. / Нижег. гос. ун-т, 2000, с.118. М.Х. Прилуцкий [↑](#footnote-ref-1)