|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | МИНОБРНАУКИ РОССИИ | | | | Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА – Российский технологический университет»**  **РТУ МИРЭА** | | |   Институт Информационных технологий | |
|  | |
| Кафедра Математического обеспечения и стандартизации информационных технологий | |
|  | |
|  | |

|  |  |
| --- | --- |
| **ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 7** | |
| **по дисциплине** | |
| **«**Кодирование и сжатие данных методами без потерь**»**  **Тема: «Основные алгоритмы работы с графами»** | |
|  | |
| Выполнил студент группы ИКБО-10-21 | Смольников А.Б. |
| Принял преподаватель | Филатов А.С. |

|  |  |  |
| --- | --- | --- |
| Лабораторная работа выполнена | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись студента)* |
|  |  |  |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_202\_\_ г. | *(подпись руководителя)* |

Москва 2022

# **Цель работы**

Получение практических навыков и знаний по выполнению сжатия данных рассматриваемыми методами.

# **Постановка задачи**

Вариант 22. Вид дерева: АВЛ

1. Составить программу сжатия текста, используя метод RLE (run length encoding/кодирование длин серий/групповое кодирование). Рассчитать коэффициент сжатия в тестах.
   1. Провести тестирование на длинной серии повторяющихся символов.
   2. Провести тестирование на длинной серии неповторяющихся символов.
2. Составить программу сжатия текста алгоритмом Лемпеля-Зива (LZ77), LZ78. Рассчитать коэффициент сжатия в тестах.
   1. Провести тестирование реализации LZ77 в соответствии с индивидуальным вариантом задания, используя двухсимвольный алфавит (0, 1). Описать процесс восстановления сжатого текста.
   2. Провести тестирование реализации LZ78 в соответствии с индивидуальным вариантом задания. Описать процесс восстановления сжатого текста.
3. Составить программу сжатия и восстановления текста алгоритмами Шеннона-Фано и Хаффмана.
   1. Провести тестирование метода Шеннона-Фано в соответствии с индивидуальным вариантом задания. Представить в отчете таблицу формирования кода, изобразить префиксное дерево, рассчитать коэффициент сжатия.
   2. Провести тестирование метода Хаффмана на строке, содержащей ваше ФИО. Построить таблицу частот встречаемости символов, сформировать алфавит исходной строки и посчитать частоту вхождений и вероятность появления символов. Изобразить префиксное дерево Хаффмана. Показать кодирование каждого символа в строке. Рассчитать коэффициент сжатия, среднюю длину кода и дисперсию.
   3. Применить алгоритм к большому текстовому файлу. Архивировать тот же файл любым архиватором. Выполнить сравнительный анализ сжатия этими способами.
4. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Таблица 1. Задания варианта №22

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Сжатие данных по методу Лемпеля-Зива LZ77 | Закодировать следующую фразу, используя код LZ78 | Закодировать фразу методами Шеннона-Фано |
| 2 | 0100100010010000101 | упупапекапекаупуп | One, two, Freddy's coming for you  Three, four, better lock  your door Five, six, grab a crucifix  Seven, eight, gonna stay up late. |

# **Решение**

* 1. **Теоретическое введение**

Кодирование длин серий (RLE) — алгоритм сжатия данных, заменяющий повторяющиеся символы (серии) на один символ и число его повторов. Серией называется последовательность, состоящая из нескольких одинаковых символов. При кодировании строка одинаковых символов, составляющих серию, заменяется строкой, содержащей сам повторяющийся символ и количество его повторов.

Алгоритмы LZ77 обеспечивают сжатие путем замены повторяющихся вхождений данных ссылками на единственную копию этих данных, существовавшую ранее в несжатом потоке данных. Совпадение кодируется парой чисел, называемых парой длина-расстояние, что эквивалентно утверждению "каждый из следующих символов длины равен символам, расположенным точно на расстоянии символов за ним в несжатом потоке".

Чтобы определить совпадения, кодировщик должен отслеживать некоторый объем самых последних данных, таких как последние 2 КБ, 4 КБ или 32 КБ. Структура, в которой хранятся эти данные, называется скользящим окном, поэтому LZ77 иногда называют сжатием со скользящим окном. Кодеру необходимо сохранить эти данные для поиска совпадений, а декодеру необходимо сохранить эти данные для интерпретации совпадений, на которые ссылается кодер. Чем больше скользящее окно, тем дольше кодировщик может выполнять поиск ссылок на создание.

Алгоритмы LZ78 сжимают последовательные данные путем построения словаря последовательностей токенов на основе входных данных, а затем заменяют второе и последующее вхождение последовательности в потоке данных ссылкой на словарную запись. Наблюдение состоит в том, что количество повторяющихся последовательностей является хорошей мерой неслучайного характера последовательности. Алгоритмы представляют словарь в виде n-арного дерева, где n - количество токенов, используемых для формирования последовательностей токенов. Каждая словарная запись имеет вид dictionary[...] = {index, token}, где index - это индекс словарной записи, представляющей ранее просмотренную последовательность, а token - это следующий токен из входных данных, который делает эту запись уникальной в словаре.

В области сжатия данных кодирование Шеннона–Фано, названное в честь Клода Шеннона и Роберта Фано - это название, данное двум различным, но родственным методам построения префиксного кода на основе набора символов и их вероятностей (оцененных или измеренных).

Метод Шеннона выбирает префиксный код, в котором исходному символу присваивается длина кодового. Один из распространенных способов выбора кодовых слов использует двоичное разложение кумулятивных вероятностей. Этот метод был предложен в "Математической теории коммуникации" Шеннона (1948), его статье, вводящей в область теории информации.

Метод Фано делит исходные символы на два набора ("0" и "1") с вероятностями, максимально близкими к 1/2. Затем эти наборы сами делятся на два, и так далее, пока каждый набор не будет содержать только один символ. Кодовым словом для этого символа является строка из "0" и "1", которая записывает, на какую половину делений он пришелся. Этот метод был предложен в более позднем техническом отчете Фано (1949).

Коды Шеннона–Фано неоптимальны в том смысле, что они не всегда достигают минимально возможной ожидаемой длины кодового слова, как это делает кодирование Хаффмана. Однако коды Шеннона–Фано имеют ожидаемую длину кодового слова в пределах 1 бита от оптимальной. Метод Фано обычно производит кодирование с более короткими ожидаемыми длинами, чем метод Шеннона. Однако метод Шеннона легче поддается теоретическому анализу.

Код Хаффмана — это особый тип оптимального префиксного кода, который обычно используется для сжатия данных без потерь. Процесс поиска или использования такого кода осуществляется с помощью кодирования Хаффмана.

Выходные данные алгоритма Хаффмана можно рассматривать как кодовую таблицу переменной длины для кодирования исходного символа (например, символа в файле). Алгоритм выводит эту таблицу из расчетной вероятности или частоты появления (веса) для каждого возможного значения исходного символа. Как и в других методах энтропийного кодирования, более распространенные символы обычно представляются с использованием меньшего количества битов, чем менее распространенные символы. Метод Хаффмана может быть эффективно реализован, находя код за время, линейное количеству входных весов, если эти веса отсортированы.

* 1. **Функции задания №1**

Для кодирования методом RLE используется функция encodeRLE, принимающая на вход строку и итеративно проходя по каждому символу определяет серии и формирует закодированную строку.

|  |
| --- |
| string encodeRLE(string data){  string result;  const unsigned long long len = data.length();  if(len==0) return "";  unsigned long long cnt=1;  for(unsigned long long i=0;i<len;i++){  if(i==len-1 || data[i]!=data[i+1]){  result+=to\_string(cnt);  result+=data[i];  cnt=1;  } else cnt++;  }  return result;  } |
|  |

* 1. **Функции задания №2**

Для реализации алгоритма сжатия LZ77 используем класс с идентичным названием, а также структуру единичной записи. В классе созданы все необходимые методы и поля. Метод fillNode является «конструктором» записи, исходный код опущен в связи с простотой.

|  |
| --- |
| struct Node{  unsigned char offset;  unsigned char length;  char next;  };  class LZ77{  int windowLength=8;  Node fillNode(int off,int l, char nex);  public:  vector<Node> encoded;  vector<Node> encode(string text);  string decode(vector<Node> encoded);  int getEncodedLength();  void printEncoded();  }; |

Метод encode() принимает на вход сжимаемую строку, и итеративно рассматривая символы, заполняет вектор с результатом. В ходе сжатия выполняется поиск повторяющихся вхождений в окне и запись их в соответствующую структуру данных.

|  |
| --- |
| vector<Node> LZ77::encode(std::string text) {  vector<Node> result;  windowLength=text.length();  result.push\_back(fillNode(0,0,text[0]));  for(int i=1;i<text.length();i++) {  string window;  if(true){  window = text.substr(0,i);  }else{  window= text.substr(i-windowLength,windowLength);  }  //finding match in buffer  int matchLength=0;  int matchOffset=0;  for(int j=0;j<window.length();j++){  int k=0;  int pos=j+k;  int startPos=j+k;  while(window[pos]==text[i+k]){  k++;  pos++;  if(pos>=window.length()){  pos-=(int)window.length();  }  }  if(k>matchLength){  matchLength=k;  matchOffset=i-startPos;  }  }  if(matchLength==0){  result.push\_back(fillNode(0,0,text[i]));  }else{  result.push\_back(fillNode(matchOffset,matchLength,text[i+matchLength]));  i+=matchLength;  }  }  encoded=result;  return result;  } |

Метод decode() принимает на вход вектор структур записей и итеративно декодирует строку, используя сдвиги в соответствии с записями.

|  |
| --- |
| string LZ77::decode(vector<Node> encoded){  string result;  for(int i=0;i<encoded.size();i++){  if(encoded[i].offset==0){  result+=encoded[i].next;  }else{  int pos=result.length()-encoded[i].offset;  int resultLen=result.length();  for(int j=0;j<encoded[i].length;j++){  result+=result[pos];  pos++;  if(pos>=resultLen){  pos-=resultLen;  }  }  result+=encoded[i].next;  }  // cout<<result<<endl;  }  return result;  } |

Методы printEncoded() и getEncodedLength() являются служебными, первый выводит архивированную строку в консоль, второй считает объем памяти, занимаемый закодированной информацией.

|  |
| --- |
| void LZ77::printEncoded() {  for(int i=0;i<encoded.size();i++){  if(encoded[i].next=='\0'){  cout<<"("<<(int)encoded[i].offset<<","<<(int)encoded[i].length<<", NTERM)"<<endl;  }else{  cout<<"("<<(int)encoded[i].offset<<","<<(int)encoded[i].length<<", "<<encoded[i].next<<")"<<endl;  }  }  }  int LZ77::getEncodedLength() {  return encoded.size()\*(1+1+1);  } |

Для реализации LZ78 аналогично существует соответствующий класс, а также структура записи. В классе реализованы необходимые методы и поля.

|  |
| --- |
| struct LZ78node{  unsigned char index;  char next;  };  class LZ78{  vector<LZ78node> encoded;  map<string,int> dictionary;  public:  int getEncodedLength();  void printEncoded();  vector<LZ78node> encode(string text);  string decode();  }; |

Метод encode() принимает на вход исходную строку и возвращает вектор записей, закодированную строку. В процессе итеративно рассматривает текст и выполняет поиск по словарю максимального совпадения, также заполняет сам словарь, рассматривая случай, когда после рассмотрения строки в буфере остались символы.

|  |
| --- |
| vector<LZ78node> LZ78::encode(string text) {  int index=0;  string buffer="";  vector<LZ78node> answer;  for(int i=0;i<text.length();i++){  if(dictionary.find(buffer+text[i]) != dictionary.end()){  buffer+=text[i];  }else{  answer.push\_back(LZ78node{static\_cast<unsigned char>(dictionary[buffer]), text[i]});  dictionary[buffer+text[i]]=dictionary.size()+1;  buffer="";  }  }  if(!buffer.empty()){  char last\_ch = buffer[buffer.length()-1];  buffer.pop\_back();  answer.push\_back(LZ78node{static\_cast<unsigned char>(dictionary[buffer]),last\_ch});  }  encoded=answer;  return answer;  } |

Методы getEncodedLength() и printEncoded() являются служебными и отвечают за возврат занимаемой памяти и читаемый вывод закодированной последовательности соответсвенно.

|  |
| --- |
| int LZ78::getEncodedLength() {  return 2\*encoded.size();  }  void LZ78::printEncoded() {  for(int i=0;i<encoded.size();i++){  cout<<(int)encoded[i].index<<" "<<encoded[i].next<<endl;  }  } |

* 1. **Функции задания №3**

Для архивации текста методом Шеннона-Фано используется специальный класс ShennonFano, который включает в себя поля:

* Целочисленный тип - длина строки;
* Вектор логических значений – ответ;
* Ассоциативный массив char, int – счетчик;
* Вектор кортежей char, double – вероятность;
* Ассоциативный массив кодов символов;
* Дерево закодированных символов.

В реализации задачи будем считать, что логическое значение в памяти занимает 1 бит, а не 1 байт. Такой подход связан с невозможностью динамического использования bitset в STL (возможно только в библиотеке boost). Для простоты и читаемости используем логические значения, а не контейнер char.

|  |
| --- |
| class ShennonFano{  long long s\_length=0;  vector<bool> answer;  map<char,int>count;  vector<tuple<char,double>>dataProbability;  map<char,string>codeTable;  int tempCnt=0;  struct treeNode{  char c;  string code;  treeNode \*left;  treeNode \*right;  };  treeNode \*rootTree = new treeNode;  public:  void scanText(string text);  vector<bool> encode(string text);  void buildCodeTree(vector<tuple<char,double>>data, string curValue, treeNode \*root);  string decode();  int getByteSize();  ~ShennonFano();  }; |

Метод scanText() принимает на вход изначальную строку и выполняет первоначальное сканирование строки, заполняет ассоциативный массив количеством встречаемых символов, инициализирует поле s\_length, а также заполняет вектор вероятностей и сортирует его по второму полю корежа.

|  |
| --- |
| void ShennonFano::scanText(string text) {  for(auto c: text){  count.find(c) == count.end() ? count[c]=1 : count[c]++;  }  s\_length=text.length();  for(auto it : count){  dataProbability.emplace\_back(it.first,(double)it.second/ (double)s\_length);  }  sort(dataProbability.begin(),dataProbability.end(),[](tuple<char,double> const &a, tuple<char,double> const &b){  return get<1>(a) >= get<1>(b);  });  } |
|  |
|  |

Метод buildCodeTree() принимает на вход контейнер вероятностей, текущий код, а также родительский элемент и рекурсивно строит дерево получившихся кодов. В процессе выполнения вектор разделяется на две части так, чтобы различие в сумме вероятностей было минимально. Далее создаются два дополнительных вектора, хранящие в себе левую и правую часть, метод вызывается снова с обновленными параметрами. Условием выхода из рекурсии является единичная длина вектора, тогда создается лист и выполнение заканчивается.

|  |
| --- |
|  |
| VoidShennonFano::buildCodeTree(vector<tuple<char,double>>data,string curValue, treeNode \*root) {  if(data.size()==1){  root->c = get<0>(data[0]);  root->code = curValue;  codeTable[get<0>(data[0])] = curValue;  tempCnt++;  return;  }else if(data.size()==0){  return;  }  int minPos=1;  double minDelta=1.0;  for(int i=1;i<data.size();i++){  double accumulateLeft=0.0;  double accumulateRight=0.0;  for(int j=0;j<data.size();j++){  if(j<i){  accumulateLeft+=get<1>(data[j]);  }else{  accumulateRight+=get<1>(data[j]);  }  }  double delta = abs(accumulateLeft-accumulateRight);  if(delta<minDelta){  minDelta=delta;  minPos=i;  }  }  vector<tuple<char,double>>leftV;  vector<tuple<char,double>>rightV;  leftV = vector<tuple<char,double>>(data.begin(),data.begin()+minPos);  rightV = vector<tuple<char,double>>(data.begin()+minPos,data.end());  treeNode \*leftNode = new treeNode;  treeNode \*rightNode = new treeNode;  root->left = leftNode;  root->right = rightNode;  buildCodeTree(leftV, curValue+"0",root->left);  buildCodeTree(rightV, curValue+"1",root->right);  } |

Метод encode() принимает на вход исходный текст, вызывает два ранее описанных подготовительных метода. В процессе выполнения по построенному дереву кодов при помощи ассоциативного массива итеративно кодирует строку.

|  |
| --- |
|  |
| vector<bool> ShennonFano::encode(string text) {  scanText(text);  buildCodeTree(dataProbability,"",rootTree);  // for(auto c: codeTable){  // cout << c.first << " " << c.second << endl;  // }  for(auto c: text){  for(auto code: codeTable[c]){  if(code=='0') answer.push\_back(false);  else answer.push\_back(true);  }  }  return {};  } |

Метод decode() выполняет декодирование строки по данным из памяти. В процессе выполнения итеративно рассматривает двоичный код и восстанавливает информацию по словарю кодов, возвращает строку.

|  |
| --- |
| string ShennonFano::decode() {  string result="";  string temp\_code;  while(!answer.empty()){  temp\_code+=answer.front() ? "1" : "0";  answer.erase(answer.begin());  for(auto it: codeTable){  if(it.second==temp\_code){  result+=it.first;  temp\_code.clear();  break;  }  }  }  return result;  } |
|  |

Метод getByteSize() рассчитывает размер закодированной строки и возвращает целое число.

|  |
| --- |
| int ShennonFano::getByteSize() {  if(answer.size()%8==0){  return answer.size()/8;  }else{  return answer.size()/8+1;  }  } |
|  |

* 1. **Функции задания №4**

Для реализации сжатия по Хаффману реализуем соответствующий класс, который включает в себя поля:

* Структура записи, состоящая из строки и веса;
* Структура узла дерева, хранящая запись и указатели;
* Вектор записей data;
* Вектор логических значений (результат);
* Ассоциативный массив кодов символов;
* Ассоциативный массив для декодирования;
* Указатель на корень дерева.

|  |
| --- |
| class Huffman{  struct Node{  string s="";  int weight=0;  };  struct treeNode{  Node data;  treeNode \*left;  treeNode \*right;  treeNode \*parent;  };  vector<Node> data;  vector<bool> answer;  map<string,string>codeTable;  map<string,string>decodeTable;  treeNode \*rootTree;  public:  void scanText(string dataText);  int find\_el(char c);  void createCodes(treeNode \*pNode, string code);  void printEncoded();  int getEncodedLength();  string decode();  vector<bool> encode(string dataText, string fileName="", bool isFile=false);  long long s\_length=0;  }; |
|  |

Метод scanText() принимает на вход исходную строку и выполняет первоначальное сканирование текста, заполняя поле длины текста и вектор записей символом и количеством его вхождений в текст.

|  |
| --- |
| void Huffman::scanText(string dataText) {  for(char c:dataText){  s\_length++;  int index = find\_el(c);  if(index!=-1){  this->data[index].weight++;  }else{  this->data.push\_back(Node{string(1,c),1});  }  }  } |
|  |

Метод encode() принимает на вход строку – исходный текст, имя файла и логическую переменную isFille. Если пользователь хочет обработать строку, вызывается метод сканирования и вектор записей сортируется, иначе сканируются все строки файла по имени. Далее заполняется вектор узлов дерева (для удобного пользования). Алгоритм построения использует приоритетную очередь, в которой узлу с наименьшей вероятностью присваивается наивысший приоритет. Пока в очереди находится более одного узла, удалить два узла с наивысшим приоритетом (наименьшей вероятностью) из очереди, создать новый внутренний узел с этими двумя узлами в качестве дочерних и с вероятностью, равной сумме вероятностей двух узлов. Добавить новый узел в очередь. Оставшийся узел является корневым, дерево завершено. Далее создаются и выводятся коды для каждого узла. В самом конце происходит непосредственно кодировка текста по полученному ассоциативному массиву. Если кодируется файл, все строки читаются итеративно.

|  |
| --- |
| vector<bool> Huffman::encode(string dataText, string fileName, bool isFile) {  if(!isFile) {  scanText(dataText);  sort(this->data.begin(), this->data.end(), [&](Node a, Node b) { return a.weight > b.weight; });  }else{  ifstream file(fileName);  string str;  while (getline(file, str)) {  scanText(str+"\n");  }  sort(this->data.begin(), this->data.end(), [&](Node a, Node b) { return a.weight > b.weight; });  }  vector<treeNode\*> nodes;  for(int i=0;i<this->data.size();i++){  nodes.push\_back(new treeNode{this->data[i], nullptr, nullptr, nullptr});  }  while(nodes.size()>1){  treeNode \*left = nodes[nodes.size()-1];  treeNode \*right = nodes[nodes.size()-2];  treeNode \*parent = new treeNode{Node{"",left->data.weight+right->data.weight},left,right,nullptr};  left->parent=parent;  right->parent=parent;  nodes.pop\_back();  nodes.pop\_back();  nodes.push\_back(parent);  sort(nodes.begin(), nodes.end(), [&](treeNode \*a, treeNode \*b) { return a->data.weight > b->data.weight; });  }  this->rootTree = nodes[0];  //create codes for all tree  createCodes(rootTree,"");  //cout map  for(auto i:codeTable){  cout<<i.first<<" "<<i.second<<endl;  }  //encode data  if(!isFile){  for(char c : dataText){  for(char code : codeTable[string(1,c)]){  if(code=='0') answer.push\_back(false);  else answer.push\_back(true);  }  }  }else{  ifstream file(fileName);  string str;  while (getline(file, str)) {  for(char c : str){  for(char code : codeTable[string(1,c)]){  if(code=='0') answer.push\_back(false);  else answer.push\_back(true);  }  }  for(char code : codeTable[string(1,'\n')]){  if(code=='0') answer.push\_back(false);  else answer.push\_back(true);  }  }  }  return answer;  } |
|  |

Метод createCodes() принимает на вход узел и код, рекурсивно обходит дерево в глубину и инициализирует коды символов в соответствии с частотой их встречаемости.

|  |
| --- |
| void Huffman::createCodes(Huffman::treeNode \*pNode, string code) {  if(pNode->left== nullptr && pNode->right== nullptr){  codeTable[pNode->data.s]=code;  decodeTable[code]=pNode->data.s;  return;  }  createCodes(pNode->left,code+"0");  createCodes(pNode->right,code+"1");  } |
|  |

Метод decode() выполняет расшифровку строки по известному набору префиксных кодов алфавита, итеративно рассматривая двоичный код и декодируя подстроки.

|  |
| --- |
| string Huffman::decode(){  string decoded;  string code;  for(bool i:answer){  if(i) code+="1";  else code+="0";  if(decodeTable.find(code)!=decodeTable.end()){  decoded+=decodeTable[code];  code="";  }  }  return decoded;  } |
|  |

Методы printEncoded() и getEncodedLength() являются вспомогательными, выводят закодированную строку и считают занимаемую память соответственно.

|  |
| --- |
| void Huffman::printEncoded(){  for(bool i: answer){  cout<<i;  }  cout<<endl;  }  int Huffman::getEncodedLength() {  return answer.size()/8;  } |
|  |

* 1. **Интерфейс**

При запуске программы пользователю предоставляется текствое меню

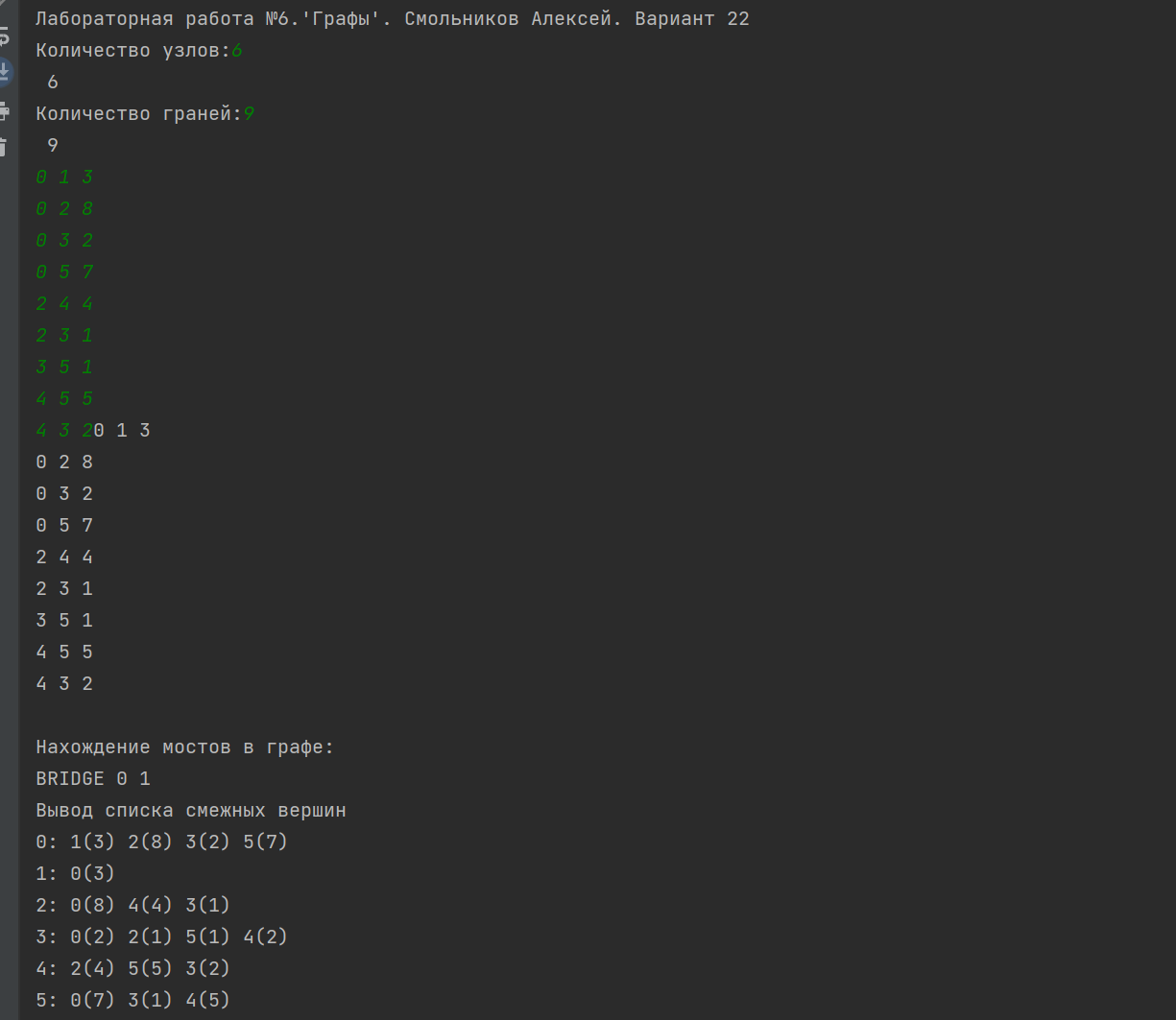


Рисунок 1. Интерфейс программы

# **Тестирование**

Тестирование проведем на графе 1 (риснок 2). Введем данные в нужном виде, рассмотрим список смежных вершин. Видно, что граф создается корректно, список соответствует действительности (рисунок 3).

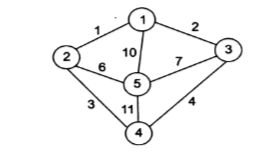


Рисунок 2. Исходный граф

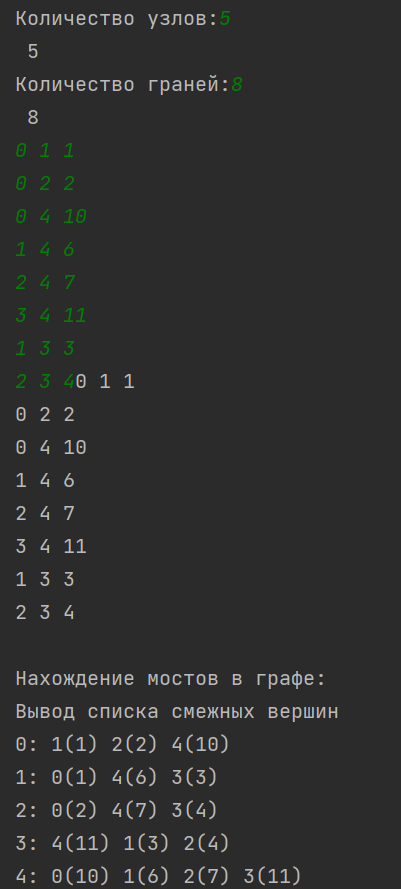


Рисунок 3. Результат заполнения графа

Нахождение мостов проверим на графе 12 (рисунок 4). Из рисунка 5 видно, что алгоритм работает корректно.

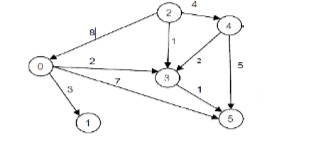


Рисунок 4. Линейный поиск

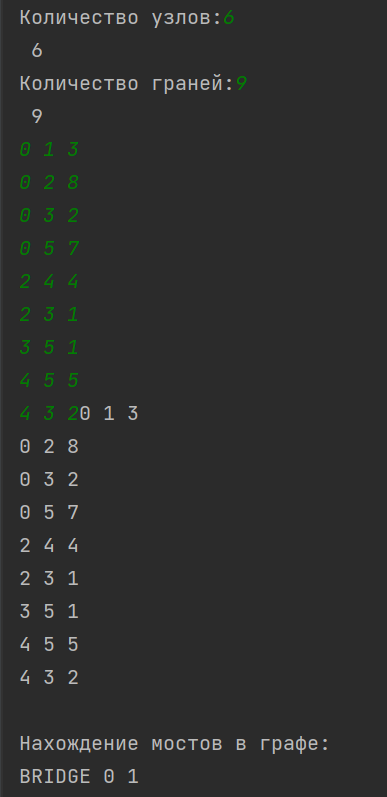


Рисунок 5. Поиск моста в графе

Для этого же графа (рисунок 4) выведем отсортированные ребра и циклы. Из рисунка 6 видно, что алгоритм работает корректно. Также из рисунка 7 заметно, что алгоритм Крускала работает корректно.

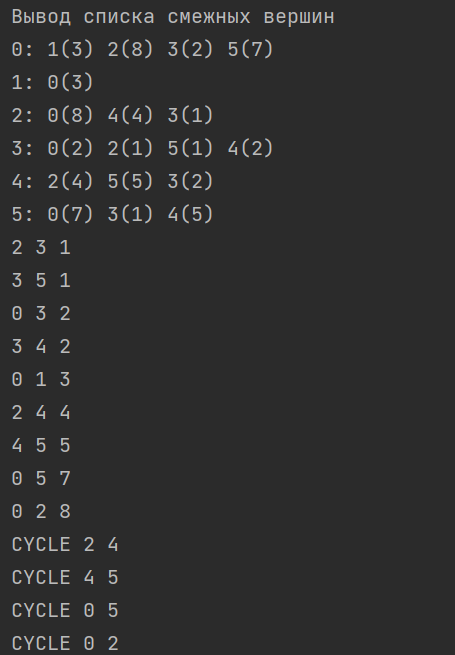


Рисунок 6. Добавление записи

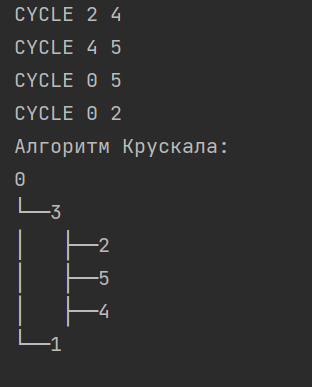


Рисунок 7. Результат работы алгоритма Крускала

При помощи средств языка Graphviz реализован графический вывод графов. Сравним граф 12 (рисунок 4) до и после применения алгоритма Крускала (рисунки 8 и 9), программа работает корректно:

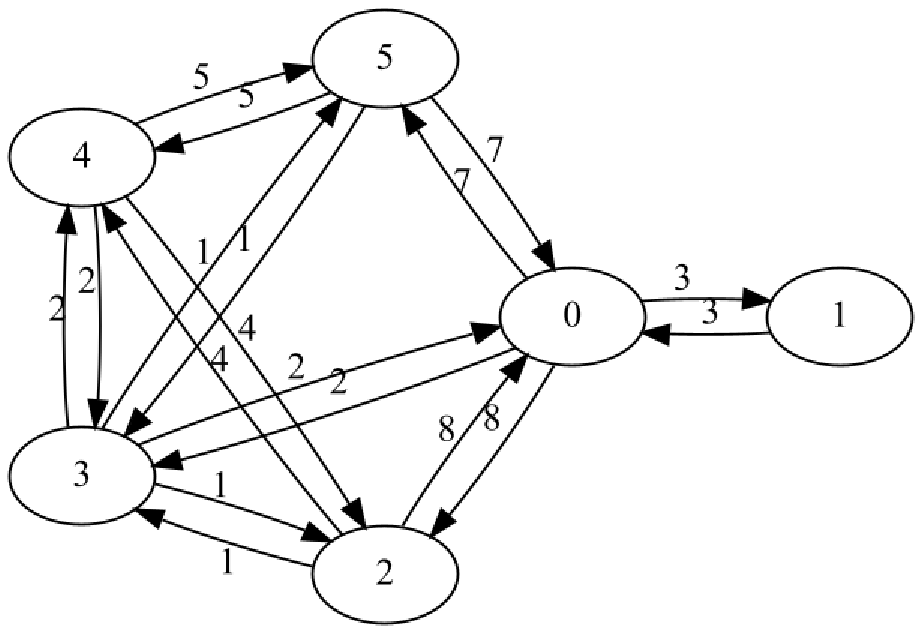


Рисунок 8. Введенное дерево

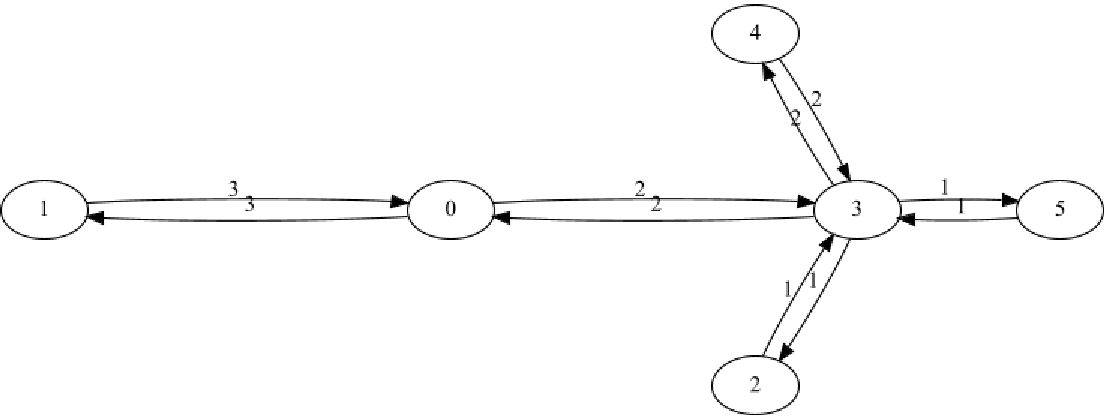


Рисунок 9. Полученное минимальное остовное дерево

# **Выводы**

В результате выполнения работы были:

* получены навыки работы со структурой данных граф;
* были реализованы алгоритмы, связанные с этой структурой на языке С++.

# **Исходный код программы**

main.cpp

|  |
| --- |
| #include <iostream>  #include <string>  #include "Graph.h"  using namespace std;  int main()  {  system("chcp 65001");  int n,a,b,w;  cout<<"Лабораторная работа №6.'Графы'. Смольников Алексей. Вариант 22"<<endl;  cout<<"Количество узлов: ";  cin>>n;  cout<<"Количество граней: ";  int e;  cin>>e;  Graph graph(n);  for (int i=0;i<e;i++)  {  cin>>a;  cin>>b;  cin>>w;  graph.addEdge(a,b,w);  }  cout<<"Нахождение мостов в графе: "<<endl;  graph.findBridges();  cout<<"Вывод списка смежных вершин"<<endl;  graph.printGraphList();  graph.toGraphviz("graph.gv");  Graph graph2 = graph.Kruksal();  graph2.toGraphviz("graphKruksal.gv");  cout<<"Алгоритм Крускала: "<<endl;  graph2.printTree();  return 0;  } |

Graph.h

|  |
| --- |
| //  // Created by lexa2k on 15.11.2022.  //  #ifndef SIAOD2SEM\_GRAPH\_H  #define SIAOD2SEM\_GRAPH\_H  #include <iostream>  #include <vector>  #include <queue>  #include <string>  #include <fstream>  #include <algorithm>  using namespace std;  // По заданию гарантируется, что вершины нащзываются от 0 до n-1,  // а при создании графа мы создаем n вершин, так что оператор индексирования работает корректно  class Graph {  struct edge{  int vertexEdge;  int weight;  };  struct adjacent{  int vertex;  vector<edge> edges;  };  struct edge2way{  int vertex1;  int vertex2;  int weight;  };  int size=0;  vector<adjacent> graph = vector<adjacent>(0);  vector<bool> PrintVisited;  public:  Graph(int size);  void addEdge(int vertex1, int vertex2, int weight, bool doRepeat = true);  void removeEdge(int vertex1, int vertex2);  void printGraphList();  void findBridges();  void dfs(int v, vector<bool> &visited, vector<int> &timeIn, vector<int> &fup, vector<int> &parent, int &timer);  Graph Kruksal();  bool isCycled(int v, vector<bool> &visited, int parent=-1);  void toGraphviz(std::string filename);  void printTree(std::string prefix="", bool root=true, int vertex=0);  };  #endif //SIAOD2SEM\_GRAPH\_H |

Graph.cpp

|  |
| --- |
| //  // Created by lexa2k on 15.11.2022.  //  #include "Graph.h"  Graph::Graph(int size) {  this->size = size;  for (int i = 0; i < size; ++i) {  adjacent adjacent1;  adjacent1.vertex = i;  graph.push\_back(adjacent1);  }  }  void Graph::addEdge(int vertex1, int vertex2, int weight, bool doRepeat) {  if(vertex1>=size || vertex2>=size){  cerr<<"Error: vertex out of range"<<endl;  return;  }  edge newEdge;  newEdge.vertexEdge = vertex2;  newEdge.weight = weight;  if(vertex1!=vertex2 and doRepeat){  edge newEdge2;  newEdge2.vertexEdge = vertex1;  newEdge2.weight = weight;  graph[vertex2].edges.push\_back(newEdge2);  }  graph[vertex1].edges.push\_back(newEdge);  }  void Graph::printGraphList() {  for (int i = 0; i < graph.size(); ++i) {  cout << graph[i].vertex << ": ";  for (int j = 0; j < graph[i].edges.size(); ++j) {  cout << graph[i].edges[j].vertexEdge << "(" << graph[i].edges[j].weight << ") ";  }  cout << endl;  }  }  //https://e-maxx.ru/algo/bridge\_searching  //https://e-maxx.ru/algo/dfs  void Graph::findBridges() {  vector<int> timeIn(size, -1);  vector<int> fup(size, -1);  vector<bool> visited(size, false);  vector<int> parent(size, -1);  int timer = 0;  for (int i = 0; i < size; ++i) {  if (!visited[i]) {  dfs(i, visited, timeIn, fup, parent, timer);  }  }  }  void Graph::dfs(int v, vector<bool> &visited, vector<int> &timeIn, vector<int> &fup, vector<int> &parent, int &timer){  visited[v] = true;  timeIn[v] = fup[v] = timer++;  for (int i = 0; i < graph[v].edges.size(); ++i) {  int to = graph[v].edges[i].vertexEdge;  if (to == parent[v]) {  continue;  }  if (visited[to]) {  fup[v] = min(fup[v], timeIn[to]);  } else {  parent[to] = v;  dfs(to, visited, timeIn, fup, parent, timer);  fup[v] = min(fup[v], fup[to]);  if (fup[to] > timeIn[v]) {  cout<<"BRIDGE "<< v << " " << to << endl;  }  }  }  }  Graph Graph::Kruksal() {  vector<edge2way> edgesKruskal;  for (int i = 0; i < graph.size(); ++i) {  for (int j = 0; j < graph[i].edges.size(); ++j) {  edge2way edge;  edge.vertex1 = graph[i].vertex;  edge.vertex2 = graph[i].edges[j].vertexEdge;  edge.weight = graph[i].edges[j].weight;  edgesKruskal.push\_back(edge);  }  }  //delete duplicate edges  for (int i = 0; i < edgesKruskal.size(); ++i) {  for (int j = i+1; j < edgesKruskal.size(); ++j) {  if(edgesKruskal[i].vertex1 == edgesKruskal[j].vertex2 && edgesKruskal[i].vertex2 == edgesKruskal[j].vertex1){  edgesKruskal.erase(edgesKruskal.begin()+j);  j--;  }  }  }  //sort edges by weight  sort(edgesKruskal.begin(), edgesKruskal.end(), [](edge2way a, edge2way b){  return a.weight < b.weight;  });  for(auto e: edgesKruskal){  cout<<e.vertex1<<" "<<e.vertex2<<" "<<e.weight<<endl;  }  Graph graphKruskal(size);  //add edges to graph if they don't create cycle  for (int i = 0; i < edgesKruskal.size(); ++i) {  vector<bool> visited(size, false);  graphKruskal.addEdge(edgesKruskal[i].vertex1, edgesKruskal[i].vertex2, edgesKruskal[i].weight, true);  // cout<<"Added edge "<<edgesKruskal[i].vertex1<<" "<<edgesKruskal[i].vertex2<<endl;  if(graphKruskal.isCycled(edgesKruskal[i].vertex1, visited)){  cout<<"CYCLE "<<edgesKruskal[i].vertex1<<" "<<edgesKruskal[i].vertex2<<endl;  graphKruskal.removeEdge(edgesKruskal[i].vertex1, edgesKruskal[i].vertex2);  }  }  return graphKruskal;  }  void Graph::removeEdge(int vertex1, int vertex2) {  for (int i = 0; i < graph[vertex1].edges.size(); ++i) {  if(graph[vertex1].edges[i].vertexEdge == vertex2){  graph[vertex1].edges.erase(graph[vertex1].edges.begin()+i);  break;  }  }  for (int i = 0; i < graph[vertex2].edges.size(); ++i) {  if(graph[vertex2].edges[i].vertexEdge == vertex1){  graph[vertex2].edges.erase(graph[vertex2].edges.begin()+i);  break;  }  }  }  //check acyclic using dfs  bool Graph::isCycled(int v, vector<bool> &visited, int parent) {  visited[v] = true;  for (int i = 0; i < graph[v].edges.size(); ++i) {  int to = graph[v].edges[i].vertexEdge;  if (to == parent) {  continue;  }  if (visited[to]) {  return true;  } else {  if(isCycled(to, visited, v)){  return true;  }  }  }  return false;  }  void Graph::toGraphviz(std::string filename) {  ofstream file;  file.open(filename);  file << "digraph G {" << endl;  file << "layout=circo;"<<endl;  for (int i = 0; i < size; ++i) {  for (int j = 0; j < graph[i].edges.size(); ++j) {  file << graph[i].vertex << " -> " << graph[i].edges[j].vertexEdge << " [label=" << graph[i].edges[j].weight << "]" << endl;  }  }  file << "}";  file.close();  }  //pretty print tree in console  void Graph::printTree(std::string prefix, bool root, int vertex) {  if(root){  PrintVisited = vector<bool>(size, false);  cout<<vertex<<endl;  }  PrintVisited[vertex] = true;  adjacent element = graph[vertex];  int adj\_start = element.vertex;  for(edge ed : element.edges){  int adj\_end = ed.vertexEdge;  if(!PrintVisited[adj\_end]){  cout << prefix;  if(root){  cout << "└──";  }else{  cout << "├──";  }  cout << adj\_end << endl;  printTree(prefix + (root ? "│ " : "│ "), false, adj\_end);  }  // PrintVisited[adj\_end] = true;  }  } |