|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  |  |  | | 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 № 6** | |
| **по дисциплине** | |
| **«**Структуры и алгоритмы обработки данных**»**  **Тема: «Основные алгоритмы работы с графами»** | |
|  | |
| Выполнил студент группы ИКБО-10-21 | Черномуров С.А. |
| Принял преподаватель | Филатов А.С. |

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

Москва 2022

# **Цель работы**

Получение практических навыков по выполнению операций над структурой данных граф.

# **Постановка задачи**

**Задание.**

1. Разработать класс «Граф», обеспечивающий хранение и работу со структурой данных «граф», в соответствии с вариантом индивидуального задания. Реализовать метод ввода графа с клавиатуры, наполнение графа осуществлять с помощью метода добавления одного ребра. Реализовать метод вывода графа и методы, выполняющие задачи, определенные вариантом индивидуального задания.
2. Разработать программу, демонстрирующую работу всех методов класса.
3. Произвести тестирование программы на графе, предложенном в таблице
4. Составить отчет, отобразив в нем описание выполнения всех этапов разработки, тестирования и код всей программы со скриншотами результатов тестирования.

Вариант №7. Условие задания:

|  |  |
| --- | --- |
| Упражнение 1 | **Представление графа в памяти:**  Матрица смежности  **Задачи:**  Определить, является ли граф связным  Составить программу нахождения кратчайшего пути в графе от заданной вершины к другой заданной вершине методом «Дейкстры» и вывести этот путь. |

# **Решение**

**Объяснение алгоритмов применяемых функций**

**Задание 1**

В конструкторе объекта класса Graph задается число узлов графа, на основании этого числа инициализируются матрицы смежности, достижимости, а также массив для хранения размера минимального пути от одного узла к другому и массив, содержащий сам минимальный путь.

|  |
| --- |
| //конструктор графа  Graph::Graph(int amount) {  this->amount = amount;  matrixSmezh = new int\* [amount];  for (int i = 0; i < amount; i++)  matrixSmezh[i] = new int[amount];  matrixReach = new int\* [amount];  for (int i = 0; i < amount; i++)  matrixReach[i] = new int[amount];  this->path = new int[amount];  memset(path, 0, amount\*sizeof(int));  this->minPath = new int[amount]; //минимальное расстояние  } |

В методе initialize матрица смежности графа вводится с клавиатуры.

|  |
| --- |
| //метод ввода графа с клавиатуры  void Graph::initialize() {  for (int i = 0; i < amount; i++) {  for (int j = 0; j < amount; j++) {  cin >> matrixSmezh[i][j];  }  }  } |

В методе print сначала выводятся номера вершин по порядку, далее выводится сама матрица смежности графа.

|  |
| --- |
| //метод вывода графа на экран  void Graph::print() {  cout << " ";  for (int i = 0; i < amount; i++)  cout << setw(3) << i + 1;  cout << "\n";  cout << "\_\_\_";  for (int i = 0; i < amount; i++) {  cout << "\_\_\_";  }  cout << "\n";  for (int i = 0; i < amount; i++) {  cout << setw(2) << right << i + 1;  cout << "|";  for (int j = 0; j < amount; j++) {  cout << setw(3) << right << matrixSmezh[i][j];  }  cout << "\n";  }  } |

В методе printMatrixOfReach сначала выводятся номера вершин по порядку, далее выводится сама матрица достижимости графа.

|  |
| --- |
| //метод вывода матрицы достижимости на экран (вспомогательный)  void Graph::printMatrixOfReach() {  cout << " ";  for (int i = 0; i < amount; i++)  cout << setw(3) << i + 1;  cout << "\n";  cout << "\_\_\_";  for (int i = 0; i < amount; i++) {  cout << "\_\_\_";  }  cout << "\n";  for (int i = 0; i < amount; i++) {  cout << setw(2) << right << i + 1;  cout << "|";  for (int j = 0; j < amount; j++) {  cout << setw(3) << right << matrixReach[i][j];  }  cout << "\n";  }  } |

В методе createMatrixOfReach создается модифицированная копия матрицы смежности (вместо весов ребер каждое ребро обозначается единицей), далее в матрицу достижимости копируется модифицированная матрица смежности, после чего матрица достижимости дополнительно форматируется для создания полной матрицы достижимости.

|  |
| --- |
| //метод создания матрицы достижимости  void Graph::createMatrixOfReach() {  int\*\* matrixSmezh2 = new int\* [amount];  for (int i = 0; i < amount; i++)  matrixSmezh2[i] = new int[amount];    for (int i = 0; i < amount; i++)  for (int j = 0; j < amount; j++)  if (matrixSmezh[i][j] <=0) matrixSmezh2[i][j] = 0;  else matrixSmezh2[i][j] = 1;  for (int i = 0; i < amount; i++)  for (int j = 0; j < amount; j++)  matrixReach[i][j] = matrixSmezh2[i][j];    for (int k = 0; k < amount; k++)  for (int i = 0; i < amount; i++)  for (int j = 0; j < amount; j++)  matrixReach[i][j] = (matrixReach[i][j] | (matrixReach[k][j] & matrixReach[i][k]));  } |
|  |

В методе checkConnectivity проверяется матрица достижимости графа, если в ней встречается хоть один ноль (что означает, что вершины нельзя достигнуть), то возвращается значение «Ложь», говорящее о несвязности графа. Если при обходе матрицы достижимости не был найден ни один ноль, то граф считается связным и метод возвращает значение «Истина».

|  |
| --- |
| //метод проверки связности графа  bool Graph::checkConnectivity() {  for (int i = 0; i < amount; i++) {  for (int j = 0; j < amount; j++) {  if (matrixReach[i][j] == 0) return false;  }  }  return true;  } |

В методе findMinimalPath передаются значения вершины-начала пути и вершины-конца пути. Далее инициализируется массив, отражающий посещение вершин, а массив минимальных расстояний до каждой вершины заполняется значениями 10000 (считается, что значения веса пути 10000 достичь невозможно). Далее происходит обход графа и составление минимальных путей до всех вершин методом Дейкстры, после чего по матрице смежности восстанавливается минимальный путь от вершины beginIndex к вершине endIndex.

|  |
| --- |
| //метод нахождения минимального пути методом Дейкстры  void Graph::findMinimalPath(int beginIndex, int endIndex) {  beginIndex--;  endIndex--;    int\* visited = new int[amount]; //посещенные вершины  //Инициализация вершин и расстояний  for (int i = 0; i < amount; i++)  {  minPath[i] = 10000;  visited[i] = 1; // 1 - не посещена  }  int minIndex, minWeight;  minPath[beginIndex] = 0;  // Шаг алгоритма  do {  minIndex = 10000;  minWeight = 10000;  for (int i = 0; i < amount; i++)  { // Если вершину ещё не обошли и вес меньше минимального  if (visited[i] == 1 && minPath[i] < minWeight)  {  minWeight = minPath[i];  minIndex = i;  }  }  // Добавляем найденный минимальный вес  // к текущему весу вершины  // и сравниваем с текущим минимальным весом вершины  if (minIndex != 10000)  {  for (int i = 0; i < amount; i++)  {  if (matrixSmezh[minIndex][i] > 0)  {  int temp = minWeight + matrixSmezh[minIndex][i];  if (temp < minPath[i])  {  minPath[i] = temp;  }  }  }  visited[minIndex] = 0; // вершина посещена  }  } while (minIndex < 10000);  // Восстановление пути    path[0] = endIndex + 1; // первый элемент - конечная вершина (все перевернуто)  int k = 1; // индекс предыдущей вершины (0+1)  int weight = minPath[endIndex]; // вес конечной вершины  while (endIndex != beginIndex) // пока не дошли до начальной вершины  {  for (int i = 0; i < amount; i++) // просматриваем все вершины  if (matrixSmezh[i][endIndex] != 0) // если связь есть  {  int temp = weight - matrixSmezh[i][endIndex]; // определяем вес пути из предыдущей вершины  if (temp == minPath[i]) // если вес совпал с рассчитанным  { // значит из этой вершины и был переход  weight = temp; // сохраняем новый вес  endIndex = i; // сохраняем предыдущую вершину  path[k] = i + 1; // и записываем ее в массив  k++;  }  }  }  this->k = k - 1;    } |

Метод printPath выводит на экран минимальный путь между двумя заданными ранее вершинами.

|  |
| --- |
| //метод вывода минимального пути на экран  void Graph::printPath() {  for (int i = k; i >= 0; i--) {  cout << path[i] << " ";  }  } |

Метод printMinimalWeight выводит вес соответствующего минимального пути на экран.

|  |
| --- |
| //метод вывода веса минимального пути на экран  void Graph::printMinimalWeight(int n) {  cout << minPath[n-1];  } |

**Описание работы пользовательского интерфейса**

Сначала выводится выбор задания, пользователь вводом с клавиатуры выбирает задание. После чего консоль очищается и отображается ввод в зависимости от выбранного задания. Введенные данные обрабатываются, результат обработки выводится на экран, программа автоматически перезапускается.

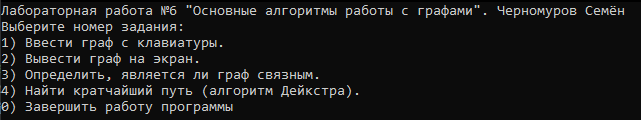


Рисунок 1. Интерфейс программы

# **Тестирование**

Для тестирования работы программы был использован граф:

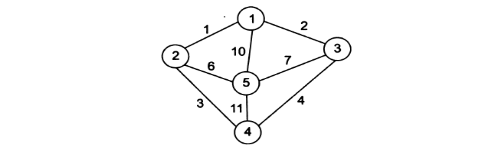


Рисунок 2. Граф для тестирования

Матрицу смежности этого графа в программе можно представить следующей комбинацией чисел: 0 1 2 0 0 1 0 0 3 6 2 0 0 4 7 0 3 4 0 11 0 6 7 11 0

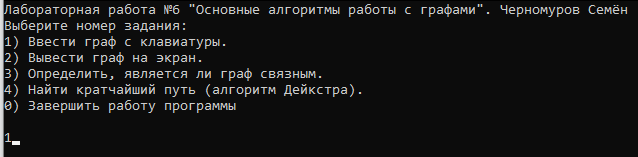


Рисунок 3. Выбор задания 1

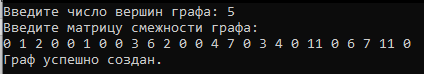


Рисунок 4. Результат работы программы (задание 1)

Программе вводится число вершин графа и матрица смежности, соответствующая этому числу вершин, далее программа инициализурет матрицу смежности графа.

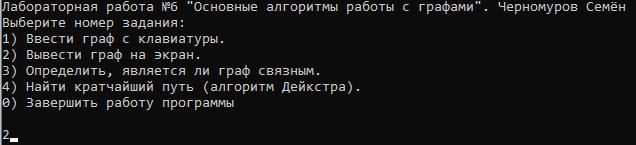


Рисунок 5. Выбор задания 2

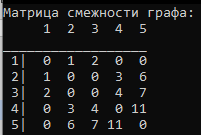


Рисунок 6. Результат работы программы (задание 2)

Программа выводит на экран матрицу смежности графа, которая хранится в качестве свойства объекта класса Graph.

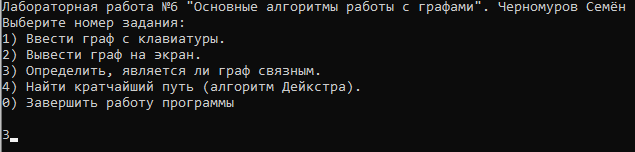


Рисунок 7. Выбор задания 3

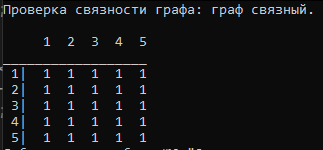


Рисунок 8. Результат работы программы (задание 3)

Программа проверяет связность графа по матрице достижимости, после чего выводит результат проверки (тестируемый граф – связный) и саму матрицу достижимости этого графа.

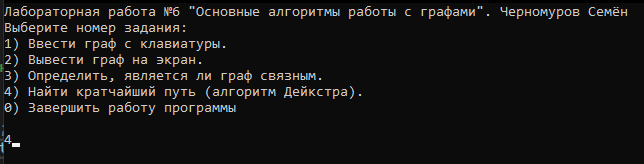


Рисунок 9. Выбор задания 4



Рисунок 10. Результат работы программы (задание 4)

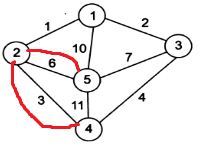


Рисунок 11. Граф с визуализированным на нем кратчайшем путём

Программа получает на вход вершину-начало пути (4) и вершину-конец пути (5), после чего методом Дейкстры находит кратчайший путь между этими вершинами и выводит вес этого пути (9) и сам путь (4 2 5) на экран.

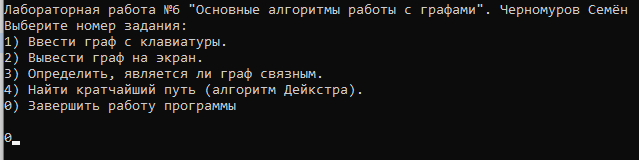


Рисунок 12. Выбор завершения работы программы



Рисунок 13. Завершение работы программы

При выборе функции завершения работы программы программа завершает свою работу.

Из результатов выполнения программы видно, что программа работает корректно.

# **Вывод**

В результате выполнения работы я:

1. Получил практические навыки по выполнению операций над структурой данных граф.
2. Закрепил знания по тестированию корректности работы программы.
3. Закрепил навыки создания пользовательских интерфейсов.

# **Исходный код программы**

**Файл main.cpp (основной алгоритм программы)**

|  |
| --- |
| #include "graph.h"  #include <iostream>  #include <chrono>  using namespace std;  Graph\* graph;  int main() {  setlocale(LC\_ALL, "");  cout << "Лабораторная работа №6 \"Основные алгоритмы работы с графами\". Черномуров Семён\n";  cout << "Выберите номер задания:\n1) Ввести граф с клавиатуры.\n" <<  "2) Вывести граф на экран.\n" <<  "3) Определить, является ли граф связным.\n" <<  "4) Найти кратчайший путь (алгоритм Дейкстра).\n" <<  "0) Завершить работу программы\n\n";  int choice1;  do {  cin >> choice1;  if (choice1 != 1 && choice1 != 2 && choice1 != 3 && choice1 != 4 && choice1 != 0) cout << "Введено неверное значение, попробуйте снова.\n";  } while (choice1 != 1 && choice1 != 2 && choice1 != 3 && choice1 != 4 && choice1 != 0);  system("cls");    switch (choice1)  {  case 1: {  cout << "Введите число вершин графа: ";  int n;  cin >> n;  graph = new Graph(n);  cout << "Введите матрицу смежности графа:\n";  graph->initialize();  cout << "Граф успешно создан.\n\n";  break;  }  case 2: {  cout << "Матрица смежности графа:\n";  graph->print();  break;  }  case 3: {  cout << "Проверка связности графа: граф ";  if (graph->checkConnectivity()) cout << "связный.\n\n";  else cout << "несвязный.\n\n";  graph->createMatrixOfReach();  graph->printMatrixOfReach();  break;  }  case 4: {  cout << "Введите номер вершин начала и конца: ";  int begin, end;  cin >> begin >> end;  graph->findMinimalPath(begin, end);  cout << "Минимальный вес пути между вершинами: ";  graph->printMinimalWeight(end);  cout << "\nЭтот путь: ";  graph->printPath();  cout << "\n\n";  break;  }    case 0:  return 0;  }    main();  } |

**Файл graph.h (заголовочный файл класса Graph для работы с графом)**

|  |
| --- |
| #pragma once  #ifndef GRAPH\_H  #define GRAPH\_H  class Graph  {  public:  Graph(int amount);  void initialize();  void print();  void createMatrixOfReach();  bool checkConnectivity();;  void printMatrixOfReach();  void findMinimalPath(int beginIndex, int endIndex);  void printPath();  void printMinimalWeight(int n); // endIndex  private:  int amount = 0;  int\*\* matrixSmezh;  int\*\* matrixReach;  int\* path;  int\* minPath;  int k;  };  #endif |

**Файл graph.cpp (реализация методов класса graph)**

|  |
| --- |
| #include "graph.h"  #include <iostream>  #include <iomanip>  using namespace std;  //конструктор графа  Graph::Graph(int amount) {  this->amount = amount;  matrixSmezh = new int\* [amount];  for (int i = 0; i < amount; i++)  matrixSmezh[i] = new int[amount];  matrixReach = new int\* [amount];  for (int i = 0; i < amount; i++)  matrixReach[i] = new int[amount];  this->path = new int[amount];  memset(path, 0, amount\*sizeof(int));  this->minPath = new int[amount]; //минимальное расстояние  }  //метод ввода графа с клавиатуры  void Graph::initialize() {  for (int i = 0; i < amount; i++) {  for (int j = 0; j < amount; j++) {  cin >> matrixSmezh[i][j];  }  }  }  //метод вывода графа на экран  void Graph::print() {  cout << " ";  for (int i = 0; i < amount; i++)  cout << setw(3) << i + 1;  cout << "\n";  cout << "\_\_\_";  for (int i = 0; i < amount; i++) {  cout << "\_\_\_";  }  cout << "\n";  for (int i = 0; i < amount; i++) {  cout << setw(2) << right << i + 1;  cout << "|";  for (int j = 0; j < amount; j++) {  cout << setw(3) << right << matrixSmezh[i][j];  }  cout << "\n";  }  }  //метод вывода матрицы достижимости на экран (вспомогательный)  void Graph::printMatrixOfReach() {  cout << " ";  for (int i = 0; i < amount; i++)  cout << setw(3) << i + 1;  cout << "\n";  cout << "\_\_\_";  for (int i = 0; i < amount; i++) {  cout << "\_\_\_";  }  cout << "\n";  for (int i = 0; i < amount; i++) {  cout << setw(2) << right << i + 1;  cout << "|";  for (int j = 0; j < amount; j++) {  cout << setw(3) << right << matrixReach[i][j];  }  cout << "\n";  }  }  //метод создания матрицы достижимости  void Graph::createMatrixOfReach() {  int\*\* matrixSmezh2 = new int\* [amount];  for (int i = 0; i < amount; i++)  matrixSmezh2[i] = new int[amount];    for (int i = 0; i < amount; i++)  for (int j = 0; j < amount; j++)  if (matrixSmezh[i][j] <=0) matrixSmezh2[i][j] = 0;  else matrixSmezh2[i][j] = 1;  for (int i = 0; i < amount; i++)  for (int j = 0; j < amount; j++)  matrixReach[i][j] = matrixSmezh2[i][j];    for (int k = 0; k < amount; k++)  for (int i = 0; i < amount; i++)  for (int j = 0; j < amount; j++)  matrixReach[i][j] = (matrixReach[i][j] | (matrixReach[k][j] & matrixReach[i][k]));  }  //метод проверки связности графа  bool Graph::checkConnectivity() {  for (int i = 0; i < amount; i++) {  for (int j = 0; j < amount; j++) {  if (matrixReach[i][j] == 0) return false;  }  }  return true;  }  //метод нахождения минимального пути методом Дейкстры  void Graph::findMinimalPath(int beginIndex, int endIndex) {  beginIndex--;  endIndex--;    int\* visited = new int[amount]; //посещенные вершины  //Инициализация вершин и расстояний  for (int i = 0; i < amount; i++)  {  minPath[i] = 10000;  visited[i] = 1; // 1 - не посещена  }  int minIndex, minWeight;  minPath[beginIndex] = 0;  // Шаг алгоритма  do {  minIndex = 10000;  minWeight = 10000;  for (int i = 0; i < amount; i++)  { // Если вершину ещё не обошли и вес меньше минимального  if (visited[i] == 1 && minPath[i] < minWeight)  {  minWeight = minPath[i];  minIndex = i;  }  }  // Добавляем найденный минимальный вес  // к текущему весу вершины  // и сравниваем с текущим минимальным весом вершины  if (minIndex != 10000)  {  for (int i = 0; i < amount; i++)  {  if (matrixSmezh[minIndex][i] > 0)  {  int temp = minWeight + matrixSmezh[minIndex][i];  if (temp < minPath[i])  {  minPath[i] = temp;  }  }  }  visited[minIndex] = 0; // вершина посещена  }  } while (minIndex < 10000);  // Восстановление пути    path[0] = endIndex + 1; // первый элемент - конечная вершина (все перевернуто)  int k = 1; // индекс предыдущей вершины (0+1)  int weight = minPath[endIndex]; // вес конечной вершины  while (endIndex != beginIndex) // пока не дошли до начальной вершины  {  for (int i = 0; i < amount; i++) // просматриваем все вершины  if (matrixSmezh[i][endIndex] != 0) // если связь есть  {  int temp = weight - matrixSmezh[i][endIndex]; // определяем вес пути из предыдущей вершины  if (temp == minPath[i]) // если вес совпал с рассчитанным  { // значит из этой вершины и был переход  weight = temp; // сохраняем новый вес  endIndex = i; // сохраняем предыдущую вершину  path[k] = i + 1; // и записываем ее в массив  k++;  }  }  }  this->k = k - 1;    }  //метод вывода минимального пути на экран  void Graph::printPath() {  for (int i = k; i >= 0; i--) {  cout << path[i] << " ";  }  }  //метод вывода веса минимального пути на экран  void Graph::printMinimalWeight(int n) {  cout << minPath[n-1];  } |