Лабораторная работа №3

Артамоновой Анастасии ПИН-24

**Конспект**

Алгоритмы нахождения кратчайших путей на графах

**Метод динамического программирования** рассматривает многостадийные процессы принятия решения. При постановке задачи динамического программирования формируется некоторый критерий. Процесс разбивается на стадии (шаги),в которых принимаются решения, приводящие к достижению общей цели. Таким образом, метод динамического программирования - метод пошаговой оптимизации.

Введем функцию fi, определяющую минимальную длину пути из начальной вершины в вершину i. Обозначим через Sij длину пути между вершинами i и j, а fj - наименьшую длину пути между вершиной j и начальной вершиной. Выбирая в качестве i такую вершину, которая минимизирует сумму (Sij + fj), получаем уравнение: fi = min {Sij + fj}.

Трудность решения данного уравнения заключается в том, что неизвестная функция входит в обе части равенства. В такой ситуации приходится прибегать к классическому методу последовательных приближений (итераций), используя рекуррентную формулу: fi^(k+1) = min {Sij + fj^(k)}, где fj^(k) - k-е приближение функции.

Возможен другой подход к решению поставленной задачи с помощью метода стратегий. При движении из начальной точки i в конечную k получается приближение fi^(0) = Sik, где Sik - длина пути между точками i и k. Следующее приближение - поиск решения в классе двухзвенных ломаных.

Дальнейшие приближения ищутся в классе трехзвенных, четырехзвенных и других ломаных

**Метод Дейкстры** предназначен для нахождения кратчайшего пути между вершинами в неориентированном графе.

Сначала выбираем путь до начальной вершины равным нулю, и заносим эту вершину во множество уже выбранных, расстояние от которых до оставшихся невыбранных вершин определено. На каждом следующем этапе находим невыбранную вершину, расстояние до которой наименьшее, соединенную ребром с какой-нибудь вершиной из множества выбранных (это расстояние будет равно расстоянию до уже выбранной вершины плюс длина ребра).

Алгоритм Флойда

Пусть в матрице A[i, j] записаны длины ребер графа (элемент A[i, j] равен весу ребра, соединяющего вершины с номерами i и j, если же такого ребра нет, то в соответствующем элементе записано некоторое очень большое число). Построим новые матрицы Ck[i, j] (k = 0,…,N). Элемент матрицы Ck[i, j] будет равен минимально возможной длине такого пути из i в j, в котором в качестве промежуточных вершин используются вершины с номерами от 1 до k. В этом случае рассматриваются пути, которые могут проходить через вершины с номерами от 1 до k, но заведомо не проходят через вершины с номерами от (k + 1) до N. В матрицу записывается длина кратчайшего из таких путей. Если таких путей не существует, записывается то же большое число, которым обозначается отсутствие ребра.

Если вычислена матрица Ck–1[i, j], то элементы матрицы Ck[i, j] можно определить по следующей формуле: Ck[i, j] = min(Ck–1[i, j], Ck–1[i, k] + Ck–1[k, j]). Рассмотрим кратчайший путь из вершины i в вершину j, который в качестве промежуточных вершин использует только вершины с номерами от 1 до k. Тогда возможны два случая:

1. путь не проходит через вершину с номером k, значит его промежуточные вершины - это вершины с номерами от 1 до (k – 1). Но длина этого пути уже вычислена в элементе Ck–1[i, j];
2. путь проходит через вершину с номером k. Но его можно разбить на две части: сначала из вершины i доходим оптимальным образом до вершины k, используя в качестве промежуточных вершины с номерами от 1 до (k – 1) (длина такого оптимального пути вычислена в Ck–1[i, k]), а потом от вершины k идем в вершину j оптимальным способом, вновь используя в качестве промежуточных вершин только вершины с номерами от 1 до k (Ck–1[k, j]).

Выбирая из этих двух вариантов кратчайший путь, получаем Ck[i, j]. Последовательно вычисляя матрицы C0, C1, C2 и т.д., получим искомую матрицу CN кратчайших расстояний между всеми парами вершин в графе.

Алгоритм Йена

Пусть граф задан матрицей A[i, j]. Вершина s выбрана как начальная. Найдем длины k наименьших путей до каждой вершины от вершины s (ребра в одном пути могут повторяться неоднократно) при условии, что эти пути существуют. Результат будет храниться в матрице B размером N × k, где N - количество вершин. Элемент массива B[i, j] равен j-му по длине пути до вершины i.

Для каждой вершины в массиве C будем хранить количество уже найденных до нее наименьших путей. Изначально все элементы массива C равны нулю. Все элементы матрицы B делаем равными какой-нибудь большой константе, заведомо большей всех возможных путей. Во время исполнения алгоритма в матрице B будем хранить лучшие k путей до каждой вершины, найденные во время исполнения, при этом первые C[i] путей для вершины i найдены уже окончательно (элементы матрицы B[i, 1], B[i, 2], …, B[i, k] для всех i упорядочены в возрастающем порядке). Таким образом, можно действовать на последующих шагах.

Пусть уже найдены какие-то кратчайшие пути. Тогда, чтобы найти следующий по длине путь, удлиним каждый из уже полученных на одно ребро. Найдем кратчайший из них, причем оканчивающийся на вершину, до которой найдено менее k путей и занесем его в таблицу результата

Алгоритм Беллмана – Форда

Пусть в матрице A[i, j] записаны длины ребер графа. Найдем кратчайшие расстояния от заданной вершины s до всех остальных вершин графа. Алгоритм Беллмана - Форда решает эту задачу при наличии ребер отрицательного веса. Обозначим через МинСт(s, v, k) наименьшую стоимость проезда из s в v менее чем с k пересадками.

МинСт(s, v, k + 1) = min(МинСт(s, v, k), МинСт(s, i, k) + A[i][v]), (i = 1,..., n). Искомым ответом является МинСт(s, i, n) для всех i = 1,..., n.

Эвристические алгоритмы

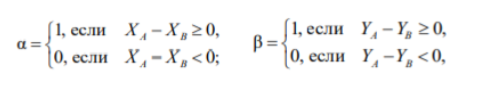
Волновой алгоритм

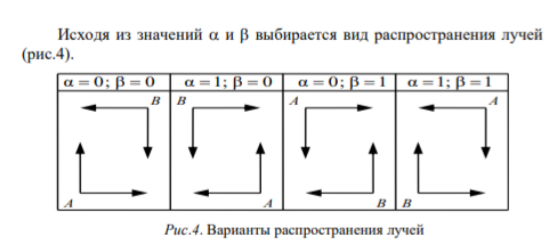
Волновой алгоритм, или алгоритм Ли, первоначально использовался для поиска пути в лабиринте или в игровых задачах. В настоящее время алгоритм Ли (волновой) является основным в микроэлектронике для трассировки, или соединения элементов интегральных схем (ИС)

В лабиринте (на подложке ИС) выбираются две точки: А (начальная) и В (конечная). Из начальной точки в четырех направлениях выходит волна Цифрами обозначается номер фронта волны или ее путевые координаты. Путевые координаты определяют шаг распространения волны. Каждый элемент первого фронта волны является источником вторичной волны Элементы второго фронта генерируют третий фронт и т.д. От запрещенных элементов волна не распространяется. Процесс продолжается до тех пор, пока не будет достигнут конечный элемент. Траектория пути (трасса) определяется обратным просмотром, от конечного элемента к начальному. При этом разработчик задает приоритеты движения: вверх, вниз, влево, вправо.

Двухлучевой алгоритм

В двухлучевом алгоритме из начального и конечного элементов одновременно выходят по два луча. Трасса считается проведенной, если пересекаются два разноименных луча (от разных источников). Если на пути луча встречается запрещенный элемент, то его обход осуществляется по второму приоритетному направлению, характерному для лучей, выходящих из одной точки. Если же оба направления оказываются заблокированными запрещенными элементами либо достигнут край координатной сетки, то движение луча прекращается. Существуют четыре варианта распространения лучей (рис.4). Коэффициенты a и b вычисляются следующим образом



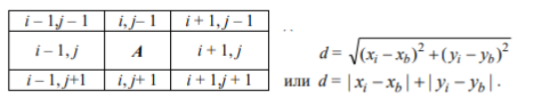


Четырехлучевой алгоритм

Из начальной и конечной точек одновременно выходят по четыре луча. Лучи движутся строго по заданным направлениям и «затухают» (прекращают движение), если достигли края координатной сетки либо встретили запрещенный элемент.

Маршрутный алгоритм

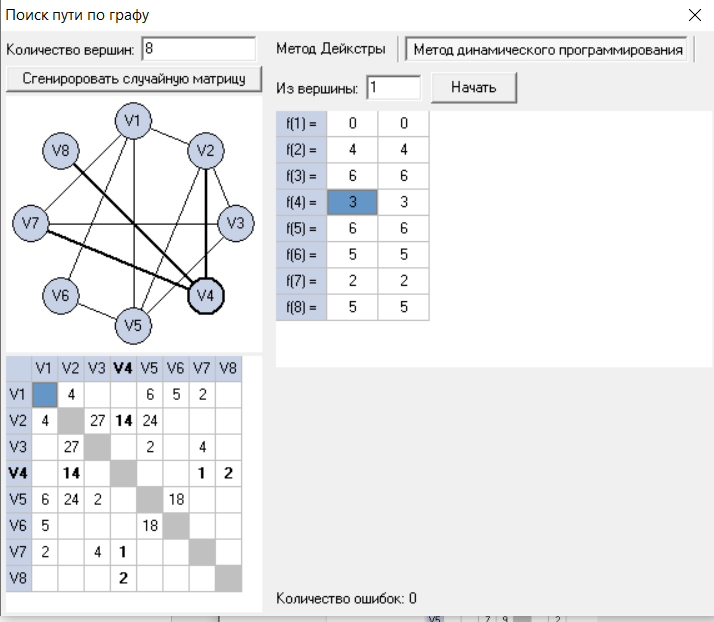
Маршрутный алгоритм получил свое название потому, что осуществляет одновременно и формирование фронта и прокладывание трассы. Источником анализируемых элементов на каждом шаге является конечный элемент участка трассы, проложенной на предыдущем шаге. В маршрутном алгоритме рассматривается восьмиэлементная окрестность исходного элемента.



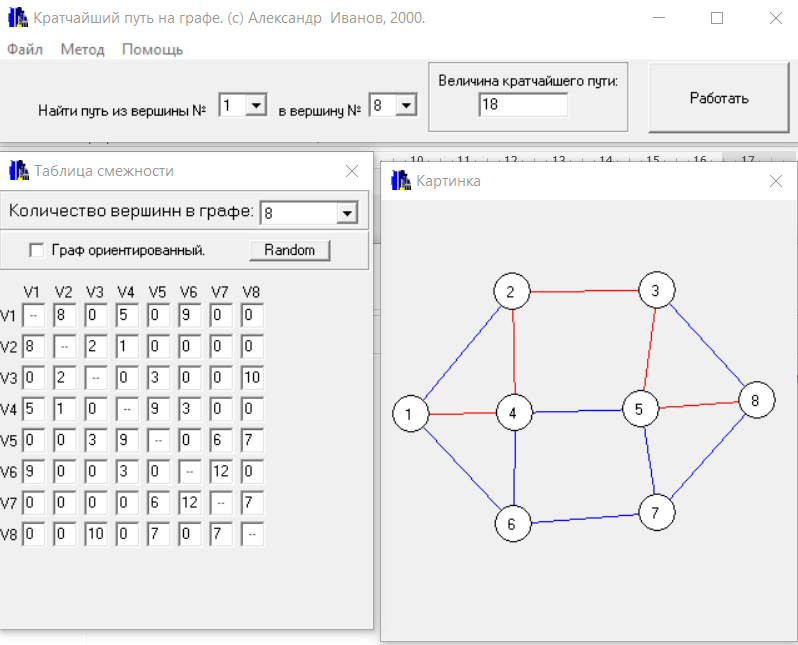
От каждого элемента окружения оценивается расстояние d до конечного элемента B.

Таким образом определяется восемь значений расстояний, из которых выбирается минимальное.

**Задание 5.1**

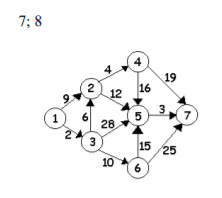


**Задание 5.2**



**Задание 5.3**

Метод динамического программирования



|  |  |
| --- | --- |
| F1 | 0 |

|  |  |
| --- | --- |
| F2 | F1+9 = 9 |
| F3+6 = 8 |

|  |  |
| --- | --- |
| F3 | F1+2 = 2 |

|  |  |
| --- | --- |
| F4 | F2+4 = 12 |

|  |  |
| --- | --- |
| F5 | F4+16 = 28 |
| F2+12 = 20 |
| F6+15 = 27 |
| F3+28 = 30 |

|  |  |
| --- | --- |
| F6 | F3+10 =12 |

|  |  |
| --- | --- |
| F7 | F4+19 = 31 |
| F5+3 = 23 |
| F6+25 = 37 |

#define \_CRT\_SECURE\_NO\_WARNINGS

#include<iostream>

#include<string.h>

#include<stdio.h>

#include<stdlib.h>

#include<conio.h>

#include<clocale>

#include <time.h>

#define word unsigned int

using namespace std;

int i, j, n, p, xn, xk;

int flag[11];

word c[11][11], l[11];

char s[80], path[80][11];

int min(int n)

{

int i, result;

for (i = 0; i < n; i++)

if (!(flag[i])) result = i;

for (i = 0; i < n; i++)

if ((l[result] > l[i]) && (!flag[i])) result = i;

return result;

}

word minim(word x, word y)

{

if (x < y) return x;

return y;

}

void main()

{

setlocale(LC\_ALL, "Russian");

clock\_t start = clock();

cout << "Напишите число вершин:";

cin >> n;

for (i = 0; i < n; i++)

for (j = 0; j < n; j++) c[i][j] = 0;

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

{

cout << " Задайте длину ребер: " << i + 1 << " до " << j + 1 << ": ";

cin >> c[i][j];

}

cout << " ";

for (i = 0; i < n; i++) cout << " " << i + 1;

cout << endl << endl;

for (i = 0; i < n; i++)

{

printf("%d", i + 1);

for (j = 0; j < n; j++)

{

printf("%6d", c[i][j]);

//c[j][i]=c[i][j];

}

printf("\n\n");

}

for (i = 0; i < n; i++)

for (j = 0; j < n; j++)

if (c[i][j] == 0) c[i][j] = 65535;

cout << " Задайте начальную вершину: ";

cin >> xn;

cout << " Задайте конечную вершнину: ";

cin >> xk;

xk--;

xn--;

if (xn == xk)

{

cout << "Начальная и коненая вершина совпадает" << endl;

\_getch();

return;

}

for (i = 0; i < n; i++)

{

flag[i] = 0;

l[i] = 65535;

}

l[xn] = 0;

flag[xn] = 1;

p = xn;

\_itoa(xn + 1, s, 10);

for (i = 1; i <= n; i++)

{

strcat(path[i], s);

}

do

{

for (i = 0; i < n; i++)

if ((c[p][i] != 65535) && (!flag[i]) && (i != p))

{

if (l[i] > l[p] + c[p][i])

{

\_itoa(i + 1, s, 10);

strcpy(path[i + 1], path[p + 1]);

strcat(path[i + 1], s);

}

l[i] = minim(l[i], l[p] + c[p][i]);

}

p = min(n);

flag[p] = 1;

}

while (p != xk);

if (l[p] != 65535)

{

cout << "Путь: " << path[p + 1] << " " << endl;

cout << "Длина пути: " << l[p] << endl;

}

else

cout << "Пути нет!" << endl;

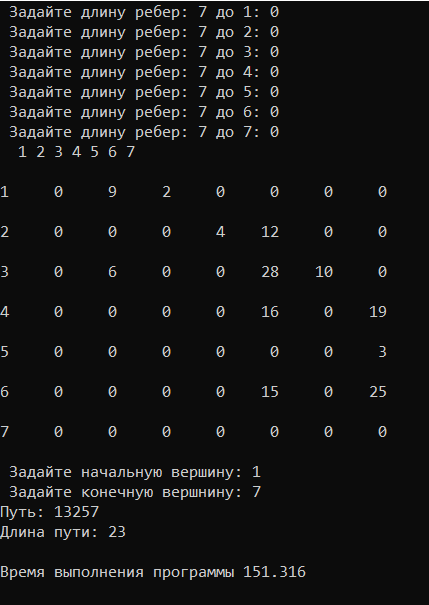
clock\_t end = clock(); //Фиксируем время конца работы

double seconds = (double)(end - start) / CLOCKS\_PER\_SEC; //Находим время работы

cout << endl << "Время выполнения программы " << seconds << endl; //Выводим это время на экран

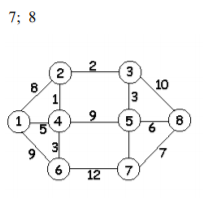
\_getch();

}



**Задание 5.4**

Метод Дейкстры



57 -6

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Выбранная вершина | Кратчайший путь | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 0 | 8 | - | 5 | - | 9 | - | - |
| 4 | 5 | 6 | - |  | 14 | 8 | - | - |
| 2 | 6 |  | 8 |  | 14 | 8 | - | - |
| 3 | 8 |  |  |  | 11 | 8 | - | 18 |
| 6 | 8 |  |  |  | 11 |  | 20 | 18 |
| 5 | 11 |  |  |  |  |  | 17 | 17 |
| 7 | 17 |  |  |  |  |  |  | 17 |

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <iostream>

#define SIZE 8

int main()

{

int a[SIZE][SIZE]; // матрица связей

int d[SIZE]; // минимальное расстояние

int v[SIZE]; // посещенные вершины

int temp, minindex, min;

int begin\_index = 0;

system("chcp 1251");

system("cls");

clock\_t start = clock(); //Фиксируем время начала работы

// Инициализация матрицы связей

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

{

a[i][i] = 0;

for (int j = i + 1; j < SIZE; j++) {

printf("Введите расстояние %d - %d: ", i + 1, j + 1);

scanf("%d", &temp);

a[i][j] = temp;

a[j][i] = temp;

}

}

// Вывод матрицы связей

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

{

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

printf("%5d ", a[i][j]);

printf("\n");

}

//Инициализация вершин и расстояний

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

{

d[i] = 10000;

v[i] = 1;

}

d[begin\_index] = 0;

// Шаг алгоритма

do {

minindex = 10000;

min = 10000;

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

{ // Если вершину ещё не обошли и вес меньше min

if ((v[i] == 1) && (d[i] < min))

{ // Переприсваиваем значения

min = d[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 < SIZE; i++)

{

if (a[minindex][i] > 0)

{

temp = min + a[minindex][i];

if (temp < d[i])

{

d[i] = temp;

}

}

}

v[minindex] = 0;

}

} while (minindex < 10000);

// Вывод кратчайших расстояний до вершин

printf("\nКратчайшие расстояния до вершин: \n");

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

printf("%5d ", d[i]);

// Восстановление пути

int ver[SIZE]; // массив посещенных вершин

int end = 4; // индекс конечной вершины = 5 - 1

ver[0] = end + 1; // начальный элемент - конечная вершина

int k = 1; // индекс предыдущей вершины

int weight = d[end]; // вес конечной вершины

while (end != begin\_index) // пока не дошли до начальной вершины

{

for (int i = 0; i < SIZE; i++) // просматриваем все вершины

if (a[i][end] != 0) // если связь есть

{

int temp = weight - a[i][end]; // определяем вес пути из предыдущей вершины

if (temp == d[i]) // если вес совпал с рассчитанным

{ // значит из этой вершины и был переход

weight = temp; // сохраняем новый вес

end = i; // сохраняем предыдущую вершину

ver[k] = i + 1; // и записываем ее в массив

k++;

}

}

}

// Вывод пути (начальная вершина оказалась в конце массива из k элементов)

printf("\nВывод кратчайшего пути от вершины (1) до (8) \n");

for (int i = k - 1; i >= 0; i--)

printf("%3d ", ver[i]);

clock\_t end\_ = clock(); //Фиксируем время конца работы

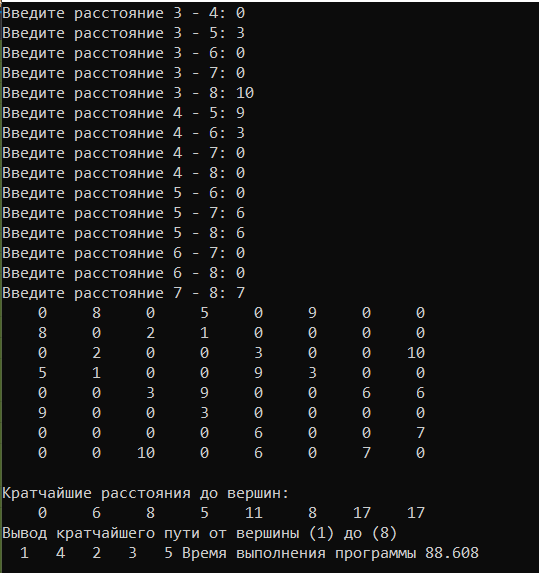
double seconds = (double)(end\_ - start) / CLOCKS\_PER\_SEC; //Находим время работы

std::cout << "Время выполнения программы " << seconds; //Выводим это время на экран

getchar(); getchar();

return 0;

}



**Задание 7.4**

Составить оптимальное расписание работы четырех процессоров, для которых известно t1, ..., t11.

#include <iostream>

#include <vector>

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <string>

#include <iomanip>

#include <cstdlib>

#include <sstream>

using namespace std;

int number\_int() {

int a;

bool status = false;

do {

cin >> a;

if ((char)cin.get() != '\n' || a < 0)

{

cout << "Ошибка! Повторите ввод" << endl;

cin.clear();

cin.ignore(cin.rdbuf()->in\_avail());

status = true;

}

else status = false;

} while (status);

return a;

}

struct task

{

string name;//название задачи

int time;//время выполнения задачи

};

class CPU

{

vector <task> list;

int time\_work;

public:

CPU()

{

vector <task> list(0);

time\_work = 0;

}

CPU(vector<task> \_list)

{

list = \_list;

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

time\_work += list[i].time;

}

~CPU()

{

list.clear();

}

vector<task> GetList()

{

return this->list;

}

int GetTimeWork()

{

return this->time\_work;

}

void AddList(task \_list)

{

list.push\_back(\_list);

time\_work += \_list.time;

}

void SetTime(int \_time)

{

time\_work = \_time;

}

int MaxTime(CPU\* cpu, int size)

{

int m\_time = cpu[0].GetTimeWork();

for (int i = 1; i < size; i++)

if (cpu[i].GetTimeWork() > m\_time)

m\_time = cpu[i].GetTimeWork();

return m\_time;

}

void PrintCpu(CPU\* cpu, int size)

{

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

{

cout << "Процессор P" << i << " : ";

for (int j = 0; j < cpu[i].list.size(); j++)

cout << cpu[i].list[j].name << " ";

cout << endl;

}

}

};

void Sort(task\* list, int lenD)

{

int j = 0;

task tmp;

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

j = i;

for (int k = i; k < lenD; k++) {

if (list[j].time < list[k].time) {

j = k;

}

}

tmp = list[i];

list[i] = list[j];

list[j] = tmp;

}

}

int min(CPU\* list, int lenD)//происк процессора с наименьшим временем работы

{

int min = 0;

for (int i = 1; i < lenD; i++)

{

if (list[i].GetTimeWork() < list[min].GetTimeWork())

min = i;

}

return min;

}

int main()

{

stringstream ss;

setlocale(LC\_ALL, "rus");

task\* catalog = new task[11];//список задач

string str; //переменная для названия задачи

int time;

cout << "Введите время выполнения каждой задачи:" << endl;

for (int i = 0; i < 11; i++)

{

ss << i;

ss >> str;

cout << "J" + str + ":";

str = " T" + str;

cout << str << "=";

catalog[i].time = number\_int();

catalog[i].name = str;

ss.clear();

}

cout << endl;

Sort(catalog, 11);

CPU\* cpu = new CPU[4];

for (int i = 0; i < 11; i++)

{

cpu[min(cpu, 4)].AddList(catalog[i]);

}

cpu[0].PrintCpu(cpu, 4);

cout << endl << "4 процессора выполнят все задачи за оптимальное время равное " << cpu[0].MaxTime(cpu, 4) << endl;

for (int i = 0; i < 4; i++)

{

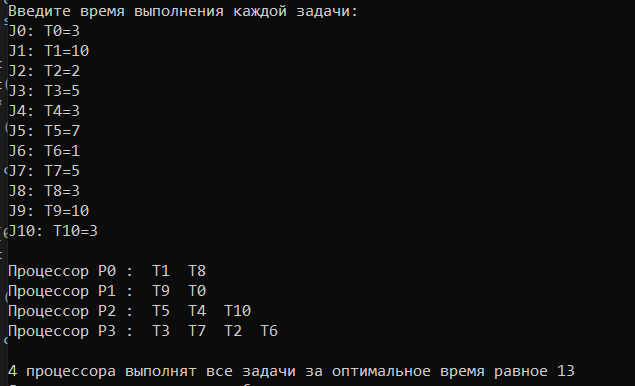
cpu[i].~CPU();

}

system("pause");

return 0;

}



**Задание 7.5**

Составить алгоритм оптимальной упаковки 12 предметов размером от 1 до 4 в ящики размером 6.

#include <iostream>

using namespace std;

void reset\_boxes(int\* arr, int n)

{

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

arr[i] = 0;

}

void Sort(int\* arr, int n)

{

int temp = 0;

bool ex = false;

while (!ex)

{

ex = true;

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

if (arr[i] < arr[i + 1])

{

temp = arr[i];

arr[i] = arr[i + 1];

arr[i + 1] = temp;

ex = false;

}

}

}

int main()

{

setlocale(LC\_ALL, "rus");

cout << "Введите кол-во ящиков: ";

int nbox;

cin >> nbox;

int\* box = new int[nbox];

reset\_boxes(box, nbox);

cout << "Введите объем ящиков: ";

int size\_box;

cin >> size\_box;

cout << "Введите кол-во элементов: ";

int nel;

cin >> nel;

int\* el = new int[nel];

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

{

cout << "Введите элемент " << i + 1 << " : ";

cin >> el[i];

}

Sort(el, nel);

int sum\_el = 0;

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

{

sum\_el = sum\_el + el[i];

}

cout << endl;

bool flag = true;

if (el[0] > size\_box)

{

cout << "Ошибка. Некоторые элементы по одиночке не могут влезть в коробку!"

<< endl;

flag = false;

}

if (sum\_el > nbox \* size\_box)

{

cout << "Ошибка. Общая сумма элементов больше свободного места в коробках!"

<< endl;

flag = false;

}

if (flag)

{

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

{

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

{

if ((size\_box - box[j]) >= el[i])

{

box[j] = box[j] + el[i];

cout << "Добавили " << el[i] << " в ящик " << j + 1 << endl;

break;

}

}

}

}

if (flag)

{

cout << "\n\nПеред Вами " << nbox << " ящиков, где в каждом соответсвенно написан заполненный объём\n";

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

cout << box[j] << " ";

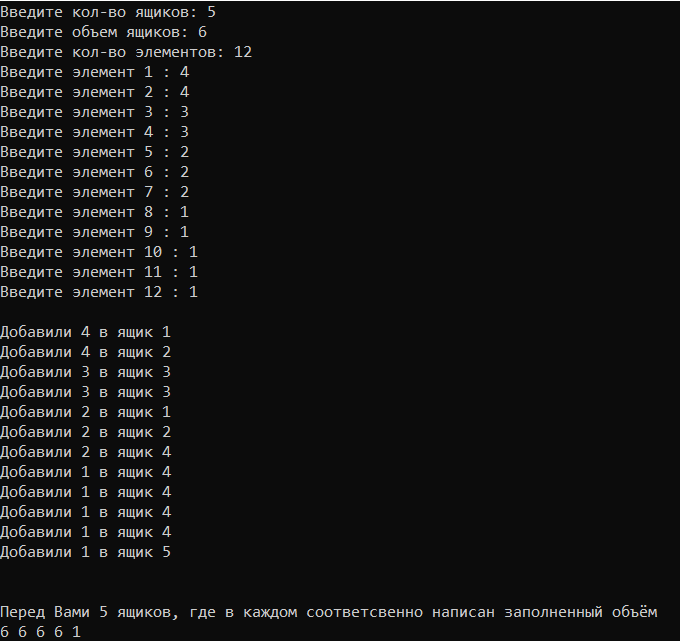
cout << "\n";

}

system("pause");

return 0;

}



**Задание 7.6**

***Задача о минимально доминирующем множестве.***

Дан граф *G=(V,E)*, где *V={v1, v2,…, vN}* – вершины графа, *E={e1, e2,…, eM}* – его ребра. Требуется найти минимальное (по мощности) множество вершин *U* такое, что любая из остальных вершин смежна с какой-либо вершиной из *U*.

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <iostream>

#define SIZE 5

using namespace std;

struct variety

{

int arr[SIZE]; //Вершины входящие в подмножество

int quantity; //Их количество

};

bool check(variety line, int graf[SIZE][SIZE])

{

int count = 0, k = 0;

for (int i = 0; i < line.quantity - 1; i++)

for (int c = i + 1; c < line.quantity; c++)

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

{

if ((((graf[line.arr[i]][j] == 0) && (graf[line.arr[c]][j] = 1)) || ((graf[line.arr[i]][j] == 1) && (graf[line.arr[c]][j] = 0)) || ((graf[line.arr[i]][j] == 1) && (graf[line.arr[c]][j] = 1)))

|| (((graf[line.arr[i]][j] == 0) && (graf[line.arr[c]][j] == 0)) && ((graf[j][line.arr[i]] == 0) && (graf[j][line.arr[c]] == 0))))

count++;

}

if (count == SIZE) return true;

else return false;

}

int Max(int graf[SIZE][SIZE], variety line)

{

int max = 0, count = 0, top = 0, help = 0;

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

{

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

{

count = count + graf[i][j];

}

if ((count >= max) && (line.quantity == 0))

{

max = count;

top = j;

}

if ((count >= max) && (line.quantity != 0))

{

help = 0;

for (int c = 0; c < line.quantity; c++)

if (line.arr[c] == j) help++;

if (help > 0)

{

max = max;

top = top;

}

else if (help == 0)

{

max = count;

top = j;

}

}

count = 0;

}

return top;

}

void main()

{

system("chcp 1251");

system("cls");

variety line;

line.quantity = 0;

int graf[SIZE][SIZE];

int temp, top, count = 0;

bool flag = true;

cout << "Введите значение ребра (0 - ребро есть, 1 - ребра нет) " << endl;

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

{

graf[i][i] = 0;

for (int j = i + 1; j < SIZE; j++) {

cout << i + 1 << " - " << j + 1 << " :";

cin >> temp;

graf[i][j] = temp;

graf[j][i] = temp;

}

}

// Вывод матрицы связей

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

{

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

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

cout << endl;

}

top = Max(graf, line);

line.arr[line.quantity] = top;

line.quantity++;

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

{

if ((graf[i][top] == 1) || (i == top))

count++;

}

if (count != SIZE)

{

while (flag == true)

{

top = Max(graf, line);

line.arr[line.quantity] = top;

line.quantity++;

if (check(line, graf)) flag = false;

}

}

cout << "Доминирующее множество содержит вершины: " << endl;

for (int i = 0; i < line.quantity; i++)

cout << line.arr[i] + 1 << " ";

cout << endl;

system("pause");

}

