1. Типы данных, структуры данных, абстрактные типы данных.

Схема процесса создания программы для решения некоторой прикладной задачи разделяется на 3 этапа:

1. Задача описания с помощью некоторого математического аппарата, т.е. создаётся математическая модель.
2. На основе математической модели определяются абстрактные типы данных. АТД – математическая модель с совокупностью операторов, определённых в рамках этой модели. Далее создаётся программа на псевдоязыке.
3. АТД преобразуется в операторы языка программирования. Реализация зависит от структур данных, реализующих АТД.

Раскраска географической карты.

1. Математическая модель – граф. Неформальный алгоритм – жадный.
2. АТД – граф, с определёнными на нём вершинами. Нахождение вершин, смежных с вершиной i, определяет цвета вершины, раскраска вершины цветом color и т.д.

цикл i=1 to кол-во вершин {

обнулить цвета флагов;

цикл j=1 to кол-во вершин {

если (вершина j смежная с i и вершина j выкрашена цветом colorк)

установить флаг цвета colorк;

найти 1-ый свободный цвет colors;

раскрасить вершину i цветом colors;

}}

1. Структуры. Двумерный и одномерный массивы. Программа на языке Java.
2. Линейные списки. Стеки. Представление стека массивом.

LIFO: Last In – First Out

Стек – абстрактный тип данных в виде линейного списка, действующий по принципу последним зашёл – первым вышел.

Операции:

nullStack – обнуление стека

empty – проверка стека на пустоту

push – добавление элемента в стек

pop – удаление элемента из стека

АТД стек обычно реализуются двумя способами: с помощью одномерного массива и с помощью динамического списка.

Реализация стека массивом:

struct Stack1 {

int top;

int \*data;};

void InitStack(Stack1 &st1, int capacity) {

st1.data = new int[capacity];

st1.top = -1;}

void push(Stack1 &st1, int value) {

st1.data[++st1.top] = value;}

int pop(Stack1 &st1) {

return st1.data[st1.top--];}

bool empty(Stack1 &st1) {

return st1.top == -1;}

void print(Stack1 &st1) {

int tmp = st1.top;

while (!empty(st1))

cout << pop(st1) << " ";

st1.top = tmp;}

void nullStack(Stack1 &st1) {

st1.top = -1;}

1. Линейные списки. Стеки. Представление стека динамическим линейным списком.

LIFO: Last In – First Out

Стек – абстрактный тип данных в виде линейного списка, действующий по принципу последним зашёл – первым вышел.

Операции:

nullStack – обнуление стека

empty – проверка стека на пустоту

push – добавление элемента в стек

pop – удаление элемента из стека

АТД стек обычно реализуются двумя способами: с помощью одномерного массива и с помощью динамического списка.

Каждый элемент списка обычно называется узлом node – это структура в динамической памяти, состоящая из двух полей data (данные) и next (указатель на следующий элемент списка).

Реализация стека динамическим линейным списком:

struct Node1 {

int data; //информационный элемент

Node1 \*next; //Голова стека};

void InitStack(Node1 \*&top) { top = NULL; }

void push(Node1 \*&top, int value){

Node1 \*tmp = new Node1;

tmp->next = top;

top = tmp;

top->data = value;}

int pop(Node1 \*&top) {

Node1 \*tmp = top;

int d = top->data;

top = top->next;

delete (tmp);

return d;}

bool empty(Node1 \*&top) {

return top == NULL;}

void print(Node1 \*top) {

Node1 \*tmp = top; //Объявляем указатель и Указываем ему, что его позиция в голове стека

//с помощью цикла проходим по всему стеку

while (tmp != NULL) //выходим при встрече с пустым полем

{

cout << tmp->data << " "; //Выводим на экран элемент стека

tmp = tmp->next; // Переходим к следующему элементу

}}

void nullStack(Node1 \*&top) {

Node1 \*tmp;

while (!empty(top)) {

tmp = top;

top = top->next;

delete(tmp);

}}

1. Линейные списки. Очереди. Представление очереди массивом.

FIFO: First In – First Out.

Очередь АТД в виде линейного списка, действующий по принципу первый вошёл - первый вышел.

Операции:

nullQueue – обнуление очереди

empty – проверка очереди на пустоту

add (enqueue) – добавление элемента в очередь

del (dequeue) – удаление элемента из очереди

В отличие от стека, который имеет одну точку входа – выхода, вход в очередь производится в хвосте (tail) , выход с головы (head).

Реализация массивом вызывает несколько сложностей. Логично считать очередь полной, когда хвост совпадает с последним элементом массива, однако, к тому времени, когда хвост достигнет конца массива из головы, могут быть удалены несколько элементов и начало массива будет неиспользованным.

Чтобы избежать сдвига элементов при каждом удалении из головы очереди массив представляют закольцованным, где за последним элементом следует первый. В этом случае номер элемента следующий за n-ым будет вычисляться по формуле (n+1)%size, где size – количество элементов массива. Кроме того, при обычной реализации очереди в виде закольцованного массива невозможно отличить переполненную очередь от пустой. Принято считать, что если за хвостом очереди сразу следует голова, то очередь пуста, а если между хвостом и головой имеется ещё один элемент, то очередь переполнена, а значит в массиве из n элементов поместится n-1.

struct Queue1 {

int head, tail, size;

int \*data;};

void nullQueue(Queue1 &q) {

q.head = 0; q.tail = q.size - 1;}

void InitQueue(Queue1 &q, int capacity) {

q.size = capacity + 1;

q.data = new int[q.size];

nullQueue(q);}

int next\_(Queue1 &q, int n) {

return (n + 1) % q.size;}

bool empty(Queue1 &q) {

return next\_(q, q.tail) == q.head;}

void add(Queue1 &q, int value) {

if (next\_(q, next\_(q, q.tail)) == q.head)

cout << "Queue overflow" << endl;

else

{

q.tail = next\_(q, q.tail);

q.data[q.tail] = value;}}

int del(Queue1 &q) {

if (empty(q)) {

cout << "Queue is empty" << endl;

return 0;}

else

{

int d = q.data[q.head];

q.head = next\_(q, q.head);

return d;}}

void print(Queue1 &q) {

if (empty(q)) {

cout << "Queue is empty" << endl;

return;}

else

{

int tmp = q.head;

while (!empty(q)) {

int d = q.data[q.head];

q.head = next\_(q, q.head);

cout << d << " ";

}

q.head = tmp;}}

1. Линейные списки. Очереди. Представление очереди динамическим линейным списком.

FIFO: First In – First Out.

Очередь АТД в виде линейного списка, действующий по принципу первый вошёл - первый вышел.

Операции:

nullQueue – обнуление очереди

empty – проверка очереди на пустоту

add (enqueue) – добавление элемента в очередь

del (dequeue) – удаление элемента из очереди

В отличие от стека, который имеет одну точку входа – выхода, вход в очередь производится в хвосте (tail) , выход с головы (head).

С точки зрения внутренней структуры динамическая очередь выглядит также как динамический стек. Отличие заключается в том, что работа со стеком производится с одной стороны списка, а работа с очередью с обеих сторон.

class Queue {

private:

struct Node {

int data;

Node \*next;};

Node \*head, \*tail;

public:

Queue() {

head = NULL;

tail = NULL;}

bool empty() {

return head == NULL;}

void add(int value) {

if (empty()) {

head = new Node;

head->data = value;

head->next = NULL;

tail = head;}

else {

tail->next = new Node;

tail = tail->next;

tail->data = value;

tail->next = NULL;}}

int del() {

if (empty()) { cout << "Queue is empty" << endl; return 0; }

else {

int d = head->data;

Node \*tmp = head;

head = head->next;

delete(tmp);

return d;}}

void nullQueue() {

Node \*tmp;

while (!empty()) {

tmp = head;

head = head->next;

delete(tmp);}}

void print() {

Node \*tmp = head; //Временный указатель на начало списка

while (tmp != NULL) //Пока в списке что-то встречается

{

cout << tmp->data << " "; //Выводим значения из списка на экран

tmp = tmp->next; //Сдвигаем указатель на начало на адрес следующего элемента

}

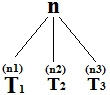
cout << endl;}};

1. Деревья. Основные понятия. Создание, обход, представление деревьев.

Дерево – совокупность элементов, которые называются узлами, один из которых определяется как корень родительских отношений, образующих иерархическую структуру узлов.

Рекурентное определение дерева:

1. Один узел является деревом, если этот же узел также является его корнем.
2. Пусть n – узел, а T1,T2, …Tk – деревья с корнями n1, n2, … nk, тогда можно создать новое дерево, в котором n будет корень, а T1,T2, …Tk – поддеревья этого корня. Узлы n1, n2, … nk прямые потомки или сыновья узла n.



Путём из узла n1 в узел nk называется последовательностью узлов n1, n2, … nk, где для всех 1<=i<k ni->ni+1.

Длиной пути называется число на единицу меньшее количества узлов, составляющих этот путь.

Если существует путь из узла a в узел b, то a – предок узла b, а b – потомок узла a.

Считается, что любой узел является одновременно предком и потомком самому себе. Таким образом, корень дерева – узел, не имеющий других предков, кроме самого себя. Узел, не имеющий потомков, кроме самого себя, называется листом или терминальной вершиной дерева.

Высотой узла дерева называется длина самого длинного пути от этого узла до какого-либо листа дерева.

Высота дерева совпадает с высотой корня.

Глубина узла определяется как длина пути от корня до этого узла.

Если в дереве не ни одной вершины, то оно называется нулевым деревом.

Способы обхода деревьев.

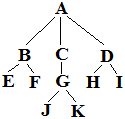
Существует несколько способов обхода (упорядоченных по какому-либо правилу) всех узлов дерева. Наиболее часто встречается три способа обхода: прямой, обратный, симметричный.

Если дерево T является нулевым, то в список обхода заносится пустая запись. Если дерево T состоит из одного узла, то в список обхода заносится этот узел. Пусть T дерево с корнем n и поддеревьями T1,T2,T3, …Tk, тогда для различных способов обхода имеем:

а) при прямом обходе сначала посещаем корень и записи поддеревья T1,T2, …Tk также в прямом порядке ABEFCGJKDHI;

б) при обратном обходе сначала посещаем в обратном порядке все узлы поддерева T1, далее T2,T3, …Tk. Последним посещаем корень n. EFBJKGCHIDA;

в) при симметричном обходе сначала посещаем в симметричном порядке все узлы поддерева T1, далее корень n, затем посещаем также в симметричном порядке все узлы поддеревьев T2,T3, …Tk. EBFACJGKHDI.



Отдельный интерес вызывают деревья, каждый узел которого может иметь не более двух сыновей (прямых потомков). Такие деревья называются двоичными (бинарными). Бинарное дерево называется сбалансированным, если для любого его узла высота левого и правого поддеревьев отличаются не более чем на единицу.

Идеально сбалансированным бинарным деревом называют такое дерево, у которого высоты левого и правого поддеревьев любого узла всегда равны.

Способы представления бинарных деревьев:

1. Одномерный массив – корень дерева помещают в первый элемент массива. Для всех остальных узлов выполняется правило: прямые потомки i-го элемента массива располагаются в ячейках i->2\*i, 2\*i+1.

 ABCDEF

1. Динамические структуры – каждый узел дерева представляет собой структуру, которая состоит как минимум из трёх полей: поля данных и указателей на левое и правое поддеревья.

struct Node{

int data;

Node \*left, right};

void add(int a, Node \*&pnode){

if (pnode==0){pnode =new Node;

pnode->data=a;

pnode->left=0;

pnode->right=0;}}

void printTree(Node \*&pnode){

cout<<pnode->data<<” ”; // прямой обход

printTree(pnode->left); // если вывод после этой строки, то симм. обход

printTree(pnode->right); // если вывод после этой строки, то обр. обход

}}

int main(){

Node \*t=0;

add(7,t);

add(6,t->left);

add(3,t->right);

add(1,t->left->right);

add(0,t->left->right);

printTree(t);

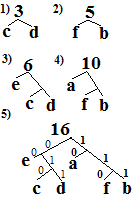
cout<<endl;

system(“pause”);

return 0;}



Частоты вхождения: a-5; b-3; c-1; d-2; e-3; f-2.



c - 010; d – 011; e – 00

В качества узла дерева будем использовать структуру, которая состоит из пяти полей:

1-е Частота int

2-е Char – символ

1. и 4 \*left, right
2. \*parent – указатель на родителя. Нужен для того, чтобы быстро найти код любого символа.

Кроме того, заведём два массива указателей на узлы. Первый массив trees будет хранить все создаваемые деревья, до тех пор, пока в нём не останется одно единственное дерево. Второй массив symbols постоянно хранит адреса всех символов с тем, чтобы, когда дерево построено, можно было по полю parent найти код нужного символа.

1. Пример использования деревьев: код Хаффмана.

Метод Хаффмана - метод оптимального кодирования, который позволяет минимизировать избыточность кода.

Идея алгоритма состоит в следующем: зная вероятности символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды. Коды Хаффмана обладают свойством префиксности (то есть ни одно кодовое слово не является префиксом другого), что позволяет однозначно их декодировать.

Классический алгоритм Хаффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хаффмана (Н-дерево).

Алгоритм:

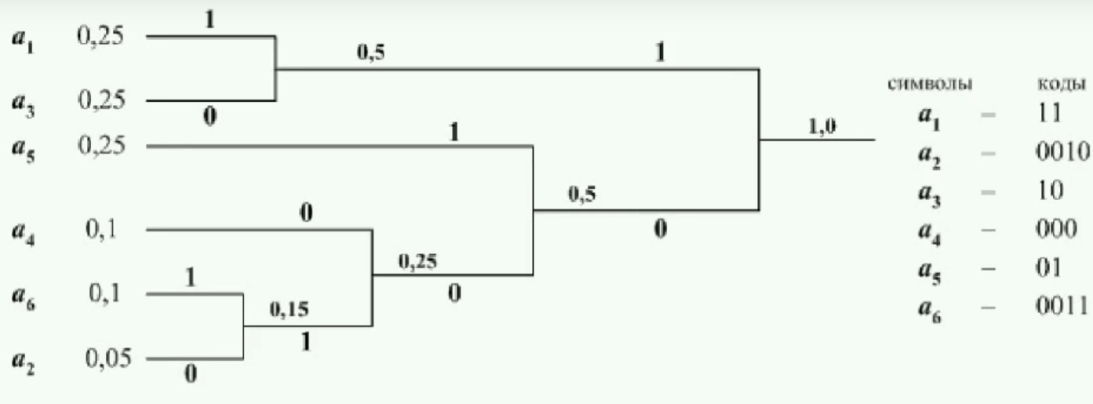
1. Упорядочиваем символы исходного алфавита в порядке невозрастания их вероятностей.
2. Объединяем два символа с наименьшими вероятностями. Символу с большей вероятностью приписываем “1”, символу с меньшей – “0” в качестве элементов их кодов.
3. Считаем объединение символов за один символ с вероятностью, равной сумме вероятностей объединённых символов.
4. Возвращаемсяна Шаг 2 до тех пор, пока все символы не будут объединены в один с вероятностью равной единице.

Пример:

Исходный алфавит A={a1,a2,a3,a4,a5,a6},

его символы встречаются с вероятностями (0,25; 0,05, 0,25; 0,1;0,25;0,1),

кодовый алфавит: B={0,1}.



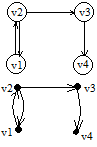
Кодирование Хаффмана широко применяется при сжатии данных, в том числе при сжатии фото- и видеоизображений (JPEG, MPEG), в популярных архиваторах (PKZIP, LZH и др.), в протоколах передачи данных HTTP (Deflate), MNP5 и MNP7 и других.

1. Графы. Основные понятия. Способы представления графов.

Любое конечное множество точек (вершин), некоторые из которых соединены попарно стрелками (дугами) можно рассматривать как граф.

V={v1,v2,v3,v4}

E={(v1,v2),(v2,v3),(v2,v3),(v3,v4)}



Не любой граф можно изобразить на плоскости так, чтобы дуги пересекались только в вершинах. Две вершины могут быть соединены несколькими дугами, идущими в одном направлении. Такие дуги называются кратными, а граф, содержащий кратные дуги – мультиграфом. В дальнейшем будем оперировать с простыми графами, т.е. графами без кратных дуг.

Основные определения.

Простым ориентированным графом называется пара объектов G=(V,E), где V – некоторое конечное множество вершин или узлов, а E – конечное множество пар (в случае неориентированного графа — неупорядоченных) вершин, называемых рёбрами.

Дуга представима в виде упорядоченной пары вершин (v1,v2), где v1 – начало дуги, а v2 – конец. При этом говорят, что дуга ведёт от вершины v1 к v2, а вершина v2 называется смежной с v1.

Граф G = (V, E) называется симметрическим, если в множестве дуг E для любой дуги (vi, vj) существует также противоположно ориентированная дуга (vj, vi).

Часто в симметрических графах пары дуг (vi,vj) и (vj,vi) заменяют рёбрами, тем самым получая неориентированный граф.

Инцидентность — понятие, используемое только в отношении ребра и вершины: если v1,v2 — вершины, а e = (v1,v2) — соединяющее их ребро, тогда вершина v1 и ребро e инцидентны, вершина v2 и ребро e тоже инцидентны. Две вершины (или два ребра) инцидентными быть не могут.

Степень вершины V называют количество инцидентных с ней рёбер.

Путём в графе G=(V,E) называется последовательность вершин v1,v2,v3,…,vn-1,vn для которых существуют дуги (v1,v2),(v2,v3),…(vn-1,vn). Длина пути – количество дуг, составляющих этот путь. Иногда каждой дуге графа присваивается некоторая стоимость (вес), такой граф называется взвешенным. В этом случае под длиной пути понимают сумму стоимости всех дуг, составляющих этот путь. Путь называется простым, если все вершины на этом пути, за исключением быть может первой и последней различны.

Цикл – простой путь длиной не менее единицы, в котором первая и последняя вершины совпадают.

Граф G=(V,E) называется сильносвязным, если для любых двух вершин (vi,vj) таких, что vi≠vj существует путь из vi в vj.

Подграф G’ называется компонентой связности графа G, если: G’ – связен и G’ обладает свойством максимальности, т.е. не существует ещё одной компоненты сильной связности G’’ включающую в себя G’.



Способы представления графа:

1. Матрица смежности для графа G=(V,E), где V={1,2,3,…,n} это матрица A размером nxn, где A[i,j]=1 тогда и только тогда, когда существует дуга из (i,j) все остальные элементы матрицы равны нулю.

0 1 0 0

1 0 1 0

0 0 0 1

0 0 0 0

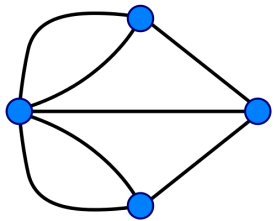
Плюс её заключается в том, что это структура простого доступа, а следовательно за 1 шаг можно получить доступ к любому элементу матрицы и узнать существует дуга между i и j вершинами графа или не существует. Недостаток – для хранения матрицы требуется объём памяти пропорциональный n2, где n – количество вершин графа и для получения сведений о всём графе также требуется количество операция n2.

1. Список смежности – представляет собой массив указателей из n элементов, в котором i-ый элемент содержит адрес линейного списка j, смежных с i-ой вершиной графа.

Список смежности удобно использовать в том случае, когда граф имеет большое количество вершин и малое количество дуг, в противном случае он никакого преимущества не даёт, кроме того невозможен прямой доступ к элементам списка.

1. Эйлеров цикл в графах. Алгоритм нахождения Эйлерова цикла.

Исторически первой задачей в теории графов была задача о кёнигсбергских мостах. Город Кёнигсберг располагался на двух берегах и двух островах р. Преголь. Острова и берега соединены между собой семью мостами. Можно ли выйдя из дома пройти по каждому мосту ровно один раз, после чего вернуться домой? Ответ на этот вопрос дал русский математик Леонард Эйлер. Он представил план Кёнигсберга в виде графа, в котором вершины – части суши, а рёбра – мосты.



Тогда задача сводится к следующей: существует ли в графе простой цикл, содержащий все рёбра? Такой цикл называется эйлеровым.

Торема. Эйлеров цикл в неориентированном связном графе существует тогда и только тогда, когда степени всех его вершин – чётные числа.

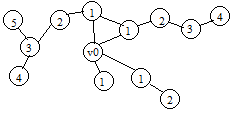
Алгоритм построения эйлерова цикла.

1. Произвольно выбирается некоторая вершина v0.
2. Произвольно выбирается некоторое ребро и инцидентное вершине v0, ему присваивается первый номер и ребро считается пройденным.
3. Каждое пройденное ребро вычёркивается из графа и ему присваивается номер на единицу больший предыдущего ребра.
4. Находясь в вершине vi не выбираем рёбра, соединённые vi с v0, если есть другая возможность.
5. Находясь в вершине vi не выбираем рёбра, которые являются “перешейком” (при удалении которого граф, образованный незачеркнутыми ребрами, распадается на две компоненты связности, имеющие хотя бы по одному ребру).
6. После того, как в графе будут занумерованы все рёбра – цикл v1,v2,…vn, образованный рёбрами 1,2,3,…n, где n – число рёбер в графе – есть эйлеров цикл.
7. Путь с наименьшим количеством дуг в графе. Волновой обход графов.

Пусть задан произвольный граф G=(V,E). Требуется найти такой путь, соединяющий две вершины vi и vj, который содержит наименьшее количество дуг. Для нахождения такого пути можно воспользоваться обходом графа в ширину или волновым обходом.

Алгоритм:

1. В графе выбирается начальная вершина v0 и её присваивается метка 0, которая делается текущей.
2. Все вершины, смежные с вершинами, имеющими текущую метку и ещё не помеченные, получают значения метки на единицу большей текущей.
3. Если не существует вершины, помещенной в п.2. – процесс заканчивается. Если конечная вершина vj получила метку, то её значение и есть длина минимального пути. В противном случае – текущая метка увеличивается на единицу и идёт к п.2.



Значение метки в соответствующей вершине даёт нам длину пути. Сам путь можно найти двигаясь от конечной вершины к начальной, переходя каждый раз к вершине, смежной и имеющей метку на единицу меньше.

int makeWave(int a, int b)

{

int path[] = new int[size + 1]; // динамический массив длины кратчаших путей

for (int i = 1; i <= size; i++) path[i] = -1;

path[a] = 0;

int num = 1, i = 1;

bool f = false, f1 = true;

do {

while (i <= size&&path[i]!num - 1)i++;

if (i <= size) {

f = true;

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

if (gr[i][j] == 1 && path[j] == 1) { path[j] = num; }

i++;

}

else { if (f) { i = 1; f = false; num++; } else f1 = false; }

} while (f1);

return path[h];

}

Волновой обход можно также использовать для нахождения областей связности в неориентированных графах.

11. Путь кратчайшей длины в графе. Алгоритм Дейкстры.

Пусть каждой дуге графа поставлена в соответствии некоторая стоимость. Требуется найти путь из вершины Vi в Vj минимальной стоимости. Ответ на этот вопрос даёт алгоритм Дейкстры.

Алгоритм:

1) Начальной вершине присваивается метка 0, которая считается постоянной.

2) Остальным вершинам графа присваивается метка +∞ считающаяся временной.

3) Берется последняя вершина Vк получившая постоянную метку. Для каждой из смежных с ней вершин Vt пересчитывается временная метка по правилу metkat = min { metkat , metkak­ + skt }, где skt стоимость ребра между k и t вершин.

4) Из всех временных меток выбирается наименьшая и делается постоянной.

5) Если конечная вершина приобрела постоянную метку, алгоритм закончен, значение постоянной метки конечное – это длина минимального пути. Если конечная вершина имеет временную метку, то идем в П.3.

Поиск самого пути как при волновом обходе от конца к началу возможен, но не рационален, поэтому обычно заводится массив, в котором для каждой вершины при пересчёте метки, заносится, из какой вершины произошёл пересчёт.

12. Пути кратчайшей длины в графе. Алгоритм Флойда. Транзитивное замыкание.

**Флойд.** Данный алгоритм использует матрицу А размером n x n в которой вычисляются длины кратчайших путей между всеми вершинами графа.

В начале А совпадает с матрицей стоимости, однако если дуга i -> j отсутствует, то значение матрицы A[ i ][ j ] = ∞; A[ i ][ j ] = 0, ∀ 1 ≤ j ≤ n

Над матрицей А выполняется n итераций. После k-ой итерации A[ i ][ j ] содержит значение наименьшей длины путей из вершины i в вершину j которые не проходят через вершину с номером больше k, т.е. на k шаге для вычисления матрицы А используется следующая формула

Ak[ i ][ j ] = min( Ak-1[ i ][ j ], Ak-1[ i ][ k ] + Ak-1[ k ][ j ]

*for (int k=1; k<=size; k++)*

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

*for (int j=1; j<=size; j++)*

*if ( a[ i ][ k ] + a[ k ][ j ] < a[ i ][ j ] )*

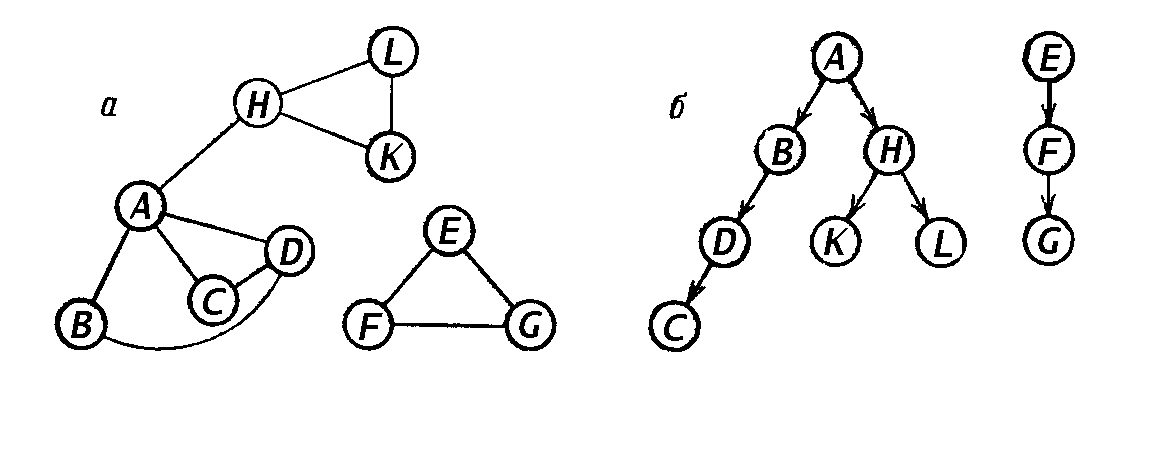
*a[ i ][ j ] = a[ i ][ k ] + a[ k ][ j ];*

**Транз. Замыкание**. Пусть имеется матрица смежности А некоторого графа G. Требуется преобразовать ее к такому виду, что А[ i ][ j ] = 1, тогда и только тогда, когда существует путь из вершины i в вершину j длиной не менее 1 и А[ i ][ j ] = 0 в противном случае. Транзитивное замыкание можно вычислить по формуле

Ak[ i ][ j ] = Ak-1[ i ][ j ] or ( Ak-1[ i ][ k ] and Ak-1[ k ][ j ] )

13. Обход графов. Метод поиска в глубину. Глубинный остовный лес графов.

Пусть имеется граф G в котором первоначально все вершины помечены меткой «Не посещалась». Обход в глубину начинается с выбора начальной вершины V, которая помечается «Посещалась». Далее для каждой вершины смежной с V, которая еще не посещалась, рекурсивно применяется алгоритм поиска в глубину. Когда все вершины, которые можно достать из вершины V, будут посещены, обход заканчивается. Если при этом некоторые вершины остались не посещенными, выбирается 1 из них и обход повторяется. Процесс продолжается, пока посещенными не будут помечены все вершины графа.



*const int n = 10;*

*int gr[ n ][ n ], metka[ n ];*

*int count = 1;*

*void mark (int V) {*

*metka[ V ] = count++;*

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

*if (gr[ V ][ i ] = 1 && metka[ i ] = 0)*

*mark( i ); }*

В процессе обхода в глубину к вершинам, которые еще не посещались ведут только определенные дуги, причем какие именно во многом зависит от выбора начальной вершины.

Дуги, которые проходятся при поиске в глубину, образуют так называемый **глубинный остовный лес** графа.

14. Остовное дерево минимальной стоимости в графах. Алгоритм Прима.

Остовное дерево минимальной стоимости – это остовное дерево с минимальной суммой весов его ребер.

**Алг. Прима.** Пусть V – множество всех вершин графа G, а U –Ø;

1. Добавим в U произвольную вершину из множества V;
2. Выберем такое ребро, соединяющее вершины множества U с вершинами множества V – U (без U), чтобы оно имело минимальную стоимость и вершину к которой ведет это ребро занесем в U;
3. Процесс заканчивается, когда множество U совпадает с множеством V.

15. Циклы в графах. Нахождение компонент сильной связности в графе.

Поиск циклов в графе.

Если при обходе в глубину встречается обратная дуга (дуга, идущая к уже посещенной вершине), то очевидно, что граф имеет цикл. Верно и обратное, если в графе существует цикл, то при обходе в глубину в нем встретится обратная дуга.

Сильная связность.

Сильно связной компонентой орграфа называют множество вершин в которых существует путь из любой вершины в любую другую.

1. Выполняется обход в глубину по графу G при этом вершины нумеруются в порядке завершения рекурсивного обхода (последняя посещенная вершина нумеруется 1);
2. Конструируется новый граф G’ путем обращения всех дуг графа G;
3. Выполняется обход в глубину графа G’ начиная с вершины с наибольшим номером, полученным в пункте 1. Если при этом не все вершины обходятся, среди оставшихся вновь выбирается вершина с наибольшим номером и т.д.
4. Каждое дерево полученное при обходе графа G’ и есть сильно связная компонента графа G.

16. Метод полного перебора. Перебор циклами. рекурсивный перебор.

Многие задачи на графах и не только заключаются в выборе одного или нескольких вариантов, отвечающих определенным критериям, из общего количества всех возможных вариантов.

Существует универсальный способ решения всех подобных задач, заключающийся в том, что перебирается весь возможный набор вариантов из которого выбирается искомое.

Главный недостаток полного перебора заключается в том, что всех вариантов может оказаться столько, что их будет не в состоянии перебрать самая современная вычислительная техника.

Перебор циклами.

Когда заранее известно общее количество всех вариантов, задачу выбора можно решить одним или несколькими вложенными циклами.

*«счастливый автобусный билет»*

*void main() {*

*int col=0;*

*for (int a=0; a<=9; a++)*

*for (int b=0; b<=9; b++)*

*for (int c=0; c<=9; c++)*

*for (int d=0; d<=9; d++)*

*for (int e=0; e<=9; e++)*

*for (int f=0; f<=9; f++)*

*if (a+b+c == d+e+f)*

*col++;*

Рекурсивный перебор.

Обычно используется, когда требуется перебирать достаточно сложные траектории, в этом случае на каждом шаге рекурсивно рассматривается движение в каждом из возможных направлений траекторий.

*void FindPath (int x, int y, int sum)*

*{sum += m[ x ][ y ];*

*If (x != n-1 || y != n-1) {*

*If (x < n-1) FindPath (x+1, y, sum);*

*If (y < n-1) FindPath (x, y+1, sum);*

*}*

*else if (sum > max) max=sum;*

17. Метод полного перебора. Полный p-ичный перебор.

Многие задачи на графах и не только заключаются в выборе одного или нескольких вариантов, отвечающих определенным критериям, из общего количества всех возможных вариантов.

Существует универсальный способ решения всех подобных задач, заключающийся в том, что перебирается весь возможный набор вариантов из которого выбирается искомое.

Главный недостаток полного перебора заключается в том, что всех вариантов может оказаться столько, что их будет не в состоянии перебрать самая современная вычислительная техника.

Полный р-ичный перебор.

Дано n предметов массами M1, M2,…, Mn. Разделить предметы на 2 кучи так, чтобы разность масс первой и второй кучи была минимальной.

Для каждого предмета имеется 2 варианта. Он попадет либо в первую, либо во вторую кучу. Закрепим за каждым предметом двоичную цифру: 0 – первая куча, 1 – вторая. Тогда некоторый расклад представляет собой последовательность 0 и 1 (двоичное число) длиной n, а всего таких раскладов будет от n нулей (000…0) }n до n единиц (111…1) }n. Таким образом задача сводится к перебору всех двоичных чисел разряда n. Двоичное число удобно держать в массиве и перебор всех чисел в массиве можно производить прибавляя каждый раз к предыдущему числу 1.

*int main() {*

*int m[ ] = {0, 10, 10, 5, 4, 5, 4};*

*int bin[ ] = {0, 0, 0, 0, 0, 0, 0};*

*int n=6;*

*int r[ 7 ];*

*int min = 100000;*

*while (bin[ 0 ] == 0) {*

*int S1 = 0, S2 = 0;*

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

*if (bin[ i ] == 0) S1 += m[ i ];*

*else S2 += m[ i ];*

*if (min > abs(S1 – S2)) { min = abs(S1 – S2);*

*for (int j=1; j<=n; j++) r[ j ] = bin[ j ]; }*

*int j = n;*

*while (bin[ j ] == 1) {bin[ j ] = 0; j - - ; }*

*bin[ j ] = 1; }*

*cout << min << endl;*

*cout << “1: “;*

*for (int j=1; j<=n; j++)*

*if (r[ j ] == 0) cout << m[ j ] << “ “;*

*cout << endl;*

*cout << “2: “;*

*for (int j=1; j<=n; j++)*

*if (r[ j ] == 1) cout << m[ j ] << “ “;*

*cout << endl;*

18. Метод полного перебора. Рекурсивный перебор.

Многие задачи на графах и не только заключаются в выборе одного или нескольких вариантов, отвечающих определенным критериям, из общего количества всех возможных вариантов.

Существует универсальный способ решения всех подобных задач, заключающийся в том, что перебирается весь возможный набор вариантов из которого выбирается искомое.

Главный недостаток полного перебора заключается в том, что всех вариантов может оказаться столько, что их будет не в состоянии перебрать самая современная вычислительная техника.

Рекурсивный перебор.

Обычно используется, когда требуется перебирать достаточно сложные траектории, в этом случае на каждом шаге рекурсивно рассматривается движение в каждом из возможных направлений траекторий.

*void FindPath (int x, int y, int sum)*

*{sum += m[ x ][ y ];*

*If (x != n-1 || y != n-1) {*

*If (x < n-1) FindPath (x+1, y, sum);*

*If (y < n-1) FindPath (x, y+1, sum);*

*}*

*else if (sum > max) max=sum;*

19. Динамическое программирование. Принцип оптимальности Беллмана.

Многие задачи поиска максимального или минимального значений можно решить быстрее чем полным перебором. Суть метода динамического программирования заключается в разбиении большой общей задачи на более мелкие подзадачи и нахождения решения для всех задач размерности 1 на их основе 2 далее 3 … n. При использовании этого метода должно выполнятся условие «то, что хорошо локально будет хорошо и глобально». Строится еще один двумерный массив размером n x n, который заполняется по следующим правилам:

1. Когда мы попадаем в «последний» элемент массива a[ n ][ n ], значение суммы изменяется только лишь на значение этого элемента. Занесем его в b[ n ][ n ].
2. В a[ n ][ n ] мы можем попасть либо из a[ n-1 ][ n ] либо из a[ n ][ n-1 ]. Их общая сумма изменится на значение этих элементов.
3. Для всех остальных элементов будем вычислять

b[ i , j ] = max { a[ i , j ] + b[ i , j+1 ]; a[ i , j ] + b[ i+1 , j ] }

1. Полученное значение b[ 1 ][ 1 ] и есть искомая максимальная сумма. Сам путь находится движением по массиву b жадным алгоритмом.

На каждом шаге приведенного примера мы вычисляли значение элемента b[ i , j ] как максимум из двух сумм. Нахождение подобного рода значений в динамическом программировании называется **принципом оптимальности**.

Будем просчитывать каждый следующий «этаж» на основе предыдущего по следующему правилу

b[ i , j ] = min { a[ i , j ] + b[ i-1 , j ]; a[ i , j ] + b[ i , j-1 ]; a[ i , j ] + b[ i , j+1 ] }

В получившемся массиве b на последнем этаже находится минимальный элемент это и будет минимальная сумма «взятки».

Для того, чтобы найти путь, заводится еще один двумерный массив в котором отмечается для каждого «кабинета» откуда мы в него пришли

0 – пришли снизу

1 – пришли справа

-1 – пришли слева