РОССИЙСКИЙ УНИВЕРСИТЕТ ДРУЖБЫ НАРОДОВ

Факультет физико-математических и естественных наук

Кафедра________________________________

УТВЕРЖДАЮ

Заведующий кафедрой
прикладной информатики
и теории вероятностей
д.т.н., профессор
____________ К.Е. Самуйлов

«____»____________ 20__г.

КУРСОВАЯ РАБОТА БАКАЛАВРА

на тему

«Задача о Ханойской башне»

010200 — Математика и компьютерные науки

Разработчик

Студент группы НК-101

Студенческий билет №: _______________

______________________ Папикян.Г.Т

«__» _____________ 20 г.

Руководитель

Доцент кафедры прикладной информатики и теории вероятностей РУДН.

______________________Зарипова.Э.Р

Москва 20__

Оглавление

Введение

Данная головоломка была придумана в 1883 году французским математиком Эдуардом Люкой.[3]

Восемь дисков разных размеров нанизаны на один из трех колышков друг на друга в порядке уменьшения диаметра (получается пирамида).

Необходимо переместить данную "пирамиду" на один из других колышков, сохраняя два условия :

1) за один ход можно перемещать только один диск
2) больший диск никогда не должен находиться на меньшем

Изначально Люка связывал головоломку с легендой о башне Брамы , которая состоит из 64 дисков чистого золота , нанизанных на 3 алмазных шпиля. Согласно легенде , при сотворении мира Всевышний поместил диски на первый шпиль и повелел жрецам переместить их на третий (согласно описанным выше правилам).Когда же они сделают это , наступит конец света ! (Судя по всему , задание они все еще не выполнили)

В дальнейших разделах мы выясним , каково минимальное количество ходов необходимо для перекладывания n дисков (при условии , что колышков всего три),также рассмотрим несколько алгоритмов для решения задачи !

Все примеры приведены на языке C++

Цель работы

Рассмотрение способов решения задачи о Ханойской башне,
поиск оптимального количества шагов .

1.Поиск минимального количества ходов

Мы имеем три колышка и n дисков , изначально нанизанных на первый из колышков. Очевидно , что задача разрешима , в этом можно убедиться , представив случай $n=2$, который , вероятно , разрешится интуитивно.

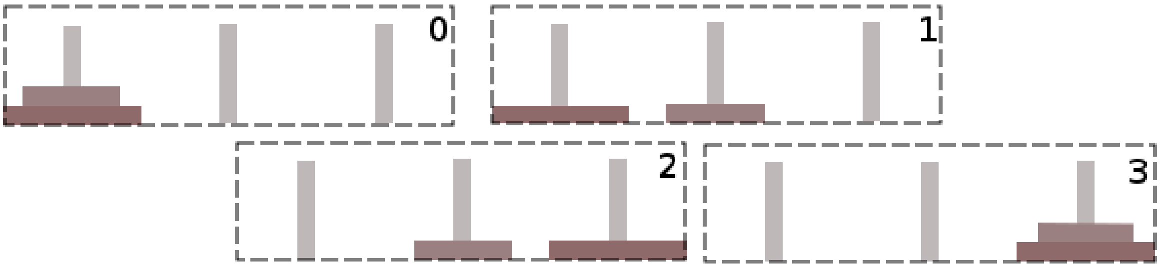

Рис.1.1
Решение задачи при n = 2.
(Перемещение с первого на третий шпиль)

После недолгих раздумий становится ясно , как поступать при $n=3$.

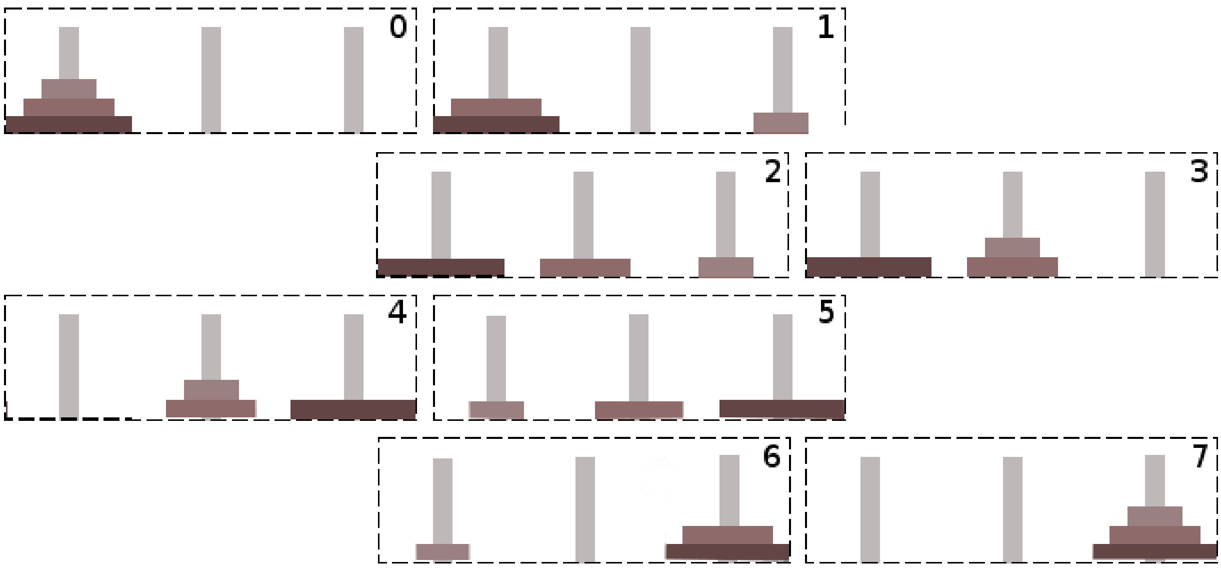

Рис.1.2
Решение задачи при n = 3.
(Перемещение с первого на третий шпиль)

Для удобства дальнейших рассуждений , минимальное число перекладываний для n дисков обозначим как T(n).

Таким образом , имеем : **T(0) = 0 , T(1) = 1, T(2) = 3, T(3) = 7**

Эксперименты показывают , для перемещения n дисков на третий шпиль необходимо переместить первые $n-1$ дисков на второй , переложить самый нижний диск на третий шпиль , и опять переместить $n-1$ дисков со второго на третий шпиль. Таким образом, при $n>0$ имеем: $T(n)\leq 2*T(n-1)+1$

Используется <= вместо = , так как мы пока не можем утверждать , что нет более "быстрого" способа для перемещения n дисков.

И все же , утверждается [3] , что $T(n)\leq 2*T(n-1)+1$ действительно является самым оптимальным способом .

Основанием служат следующие рассуждения:

При любом подходе необходимо переместить самый нижний диск , сделать это можно только в том случае , если оставшиеся n-1 дисков находятся на дугом (например, втором) колышке .После перемещения же бОльшего диска, необходимо переместить оставшиеся n-1 дисков на бОльший (все диски должны находиться на одном колышке).Таким образом , получаем описанную выше формулу.

$$T(0)=0 \quad , \quad T(n)=2*T(n-1)+1 \quad , \quad n>0$$

Мы получили рекурсивную формулу (рекуррентное соотношение) . Все работает прекрасно , количество минимальных достаточных перекладываний можно посчитать для любых n .Сложность заключается в том , что перед тем , как вычислить значение для n , прежде необходимо вычислить значение для $n-1$ и так далее , до $n=1$. В случае больших n в таком случае возникнут определенные трудности. Попробуем получить формулу для T(n) в замкнутом виде (формулу , не содержащую T(i)).Один из способов заключается в угадывании замкнутой формулы и последующем ее доказательстве при помощи математической индукции .

Посмотрим:

T(0) = 0

T(1) = 1

T(2) = 1 * 2 + 1 = 3

T(3) = 3 * 2 + 1 = 7

T(4) = 7 * 2 + 1 = 15

T(5) = 15 * 2 + 1 = 31

T(6) = 31 * 2 + 1 = 63

Складывается такое впечатление , что $T(n)=2^n-1$

Доказательство:

База индукции: T(0) = 0;

Индуктивный переход:

$T(n+1)=2^{n+1}+1=2*2^n+1$

сделаем замену p = n+1,

получаем исходное уравнение : $T(p)=2^p-1$;

Таким образом , T(8) = 2^8 - 1 = 255;

T(64) = 2^{64} - 1 = ... (теперь ясно , почему монахи все еще не справились с заданием)

2.Построение рекурсивного , итеративного алгоритмов

2.1.Рекурсивный алгоритм

Рекурсивный алгоритм предельно прост и строится , исходя из рассуждений выше :

Пояснение : Номер свободного шпиля вычисляется по формуле ***6-to-from***

Листинг 2.1:

```
void move(int from, int to,int n ){
    if(n == 1) {
        cout<<" move "<< n <<"th plate from "
              <<from<<"th to "<<to<<"th stick"<<endl;
        return ;
    }
    move(from,6-from-to,n-1);
    cout<<" move "<< n <<"th plate from "
              <<from<<"th to "<<to<<"th stick"<<endl;
    move(6-from-to,to,n-1);
}
```

2.2.Итеративный алгоритм

С итеративным алгоритмом дело обстоит иначе: для его построения необходимо найти определённые закономерности в перестановке дисков. Рекурсивный алгоритм можно разложить в дерево ходов, сделав тем самым дальнейшие рассуждения нагляднее.

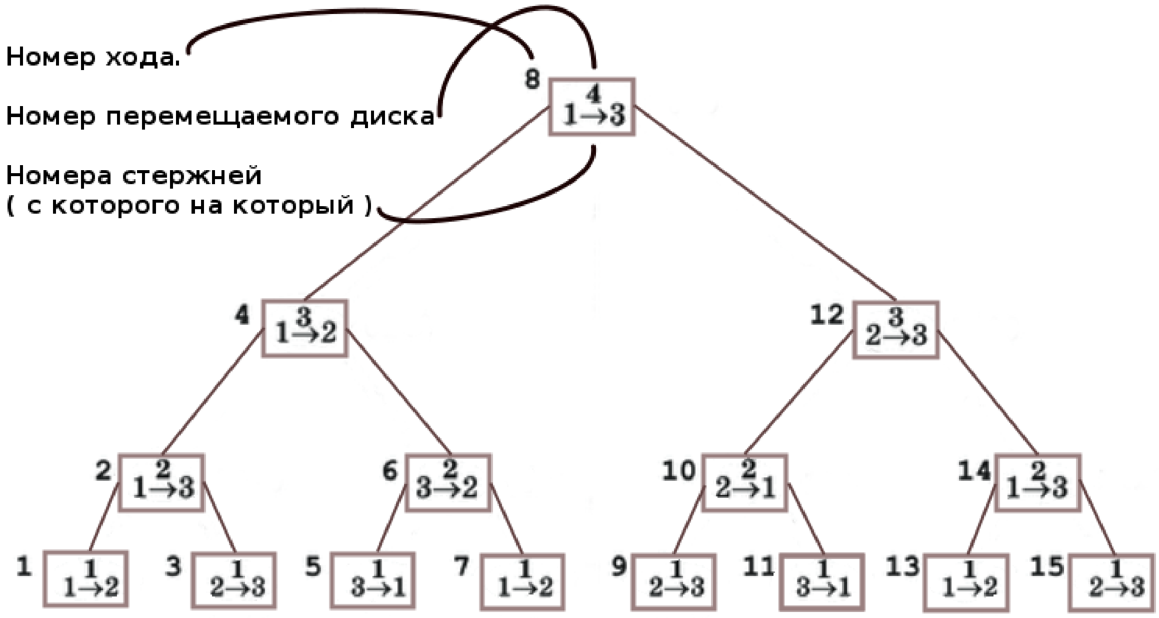

Рис.1.3
Дерево ходов при n = 4

Теперь попробуем порассуждать !

В книге [1] предлагается несколько вопросов относительно работы рекурсивного алгоритма , ответы на которые являются ключевыми в поиске итеративного . Рассмотрим их:

1) Сколько перемещений совершает каждый диск ?

при n = 4 имеем :

первый (самый маленький) — 8 раз

второй — 4 раза

третий — 2 раза

четвертый (самый большой)— 1 раз

Следовательно, для каждого диска имеем 2^{n-i} перемещений.

(i — номер диска, n — их количество)

Можно проверить , сложив количество перемещений для каждого диска в общем виде , тогда должна получиться формула $T(n) = 2^n - 1$ (количество перемещений для решения всей головоломки) : $2^0 + 2^0 + 2^2 + ... + 2^{n-1}$

Получили геометрическую прогрессию со знаменателем , равным двум , значит, $S=\frac{b_1*(1-q^n)}{1-q}=\frac{1*(1-2^n)}{1-2}=2^n-1$

2) Существует ли закономерность в очередности шпилей для каждого диска ?

В строчках дерева прослеживается закономерность,например , третий диск движется по траектории 1→ 2→ 3, второй — 1→ 3→ 2→ 1→ 3

Все станет намного интересней , если изобразить шпили виде вершин равностороннего треугольника , как предложено в [1]

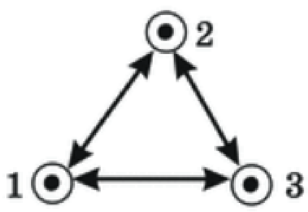

Рис.1.4

Альтернативный способ представления шпией

Получаем , что каждый диск на протяжении всех перестановок движется либо по часовой стрелке , либо против , более того , направления движения дисков чередуются на каждом уровне дерева , так диск под номером 4 движется против часовой стрелки , под номером 3 — по часовой , под номером 2 — опять против часовой.

В общем случае , диски, четность которых совпадает с четностью n — того , самого большого, диска, движутся в одном направлении , остальные — в противоположном

3) Как построить последовательность перемещаемых дисков?(без рекурсии)

при **n =** ***1*** имеем [***1***]

при **n =** ***2*** имеем [**1** ***2*** **1**]

при **n =** ***3*** имеем [**1 2 1** ***3*** **1 2 1**]

при **n =** ***4*** имеем [**1213121** ***4*** **1213121**]

Обозначим последовательность для n дисков , как D(n).

Таким образом, D(n) = D(n-1), n, D(n-1) при D(1) = 1

Несмотря на то , что формула является рекурсивной , данный алгоритм реализуется итеративно.

Например , сохраняем в переменную last_seq нашу первую последовательность (last_seq = "1") , далее , итерируем от 2 до n , изменяя last_seq подобным образом:

last_seq = last_seq + номер итерации + last_seq

4) Как часто перекладывается любой из дисков ?

На каком шаге начинается перемещение диска и через сколько шагов оно повторяется?Перемещение **диска 1** начинается **на шаге 1**

диска 2 - на шаге 2

диска 3 - на шаге 4

Вырисовывается такая картина : **диск A** впервые перемещается **на шаге** $\mathbf{2^{(A-1)}}$

Каждое следующее перемещение происходит через

$$2*(2^{A-1} - 1) + 1 = 2^{A} - 1 \text{ шагов.} \quad \text{(рис.1.3)}$$

Тогда найдем в общем виде формулу для шага , на котором перемещается диск A. Для этого посмотрим подробнее на последовательность перемещаемых дисков:

$$i = 2^{A-1} + k*2^{A} \quad , \ (\ k = 0,1,2, \ ... \ ,2^{n-A} - 1 \ )$$

правое слагаемое получается из формулы $2^{A}-1$

k — формально , номер перемещения диска , начинается нуля , так как первое перемещение левого диска всегда вычисляется с помощью левого слагаемого , таким образом , в действительности , номер перемещения диска равен k+1 .

После некоторых манипуляций получаем формулу $i = (2k + 1) * 2^{A-1}$, которая является более удобной для дальнейших рассуждений .

5) Как по номеру шага определить, какой диск перемещается на данном шаге , и в какой раз ?

Имеем уравнение: $i = (2k + 1) * 2^{A-1}$, где i - известный номер шага.

Если разложить его на произведение двоек и оного нечетного числа (2k+1) ,то количество двоек в этом разложении , плюс один , будет являться номером диска A.

Если же вычесть из нечетного множителя один и разделить на два , получим k , а как уже известно , номер перемещения диска равняется k+1

Пример ,104-й шаг :

$$104 = 13*2*2*2 = (2*6 + 1)*2^{3}$$

тогда , A = 3+1 = 4 ,номер перемещения диска A равняется 6+1 = 7

В действительности , всю эту процедуру намного удобнее проводить с двоичным представлением числа : $104_{10} = 1101000_{2}$

Как известно , при умножении двоичного числа но два , к нему справа "дописывается" нуль, следовательно , количество нулей слева в двоичном представлении - количество двоек в разложении. Так как к этому числу мы еще должны прибавить один , можем считать , что номер диска A - то же самое , что и позиция первой единицы , считая справа , а данном случае - 4.Для нахождения же k, отбрасываем все нули справа (чтобы получить 2k+1) и дополнительно отбрасываем следующую за ними единицу (т.к нужно вычесть один и разделить на два).

Теперь у нас есть достаточное понимание рекурсивного алгоритма.

Подводя итоги ,сформулируем алгоритм (пока что на естественном языке) :

Будем обходить дерево перемещений при помощи двоичного поиска .То есть , начиная с корня нашего дерева (шаг под номером 8) ,спускаясь вниз, ищем необходимые нам узлы. Никакая информация про сами узлы храниться не должна , вся необходимая информация будет сгенерирована на ходу , опираясь на описанные выше закономерности.

Для каждого узла нам нужен :

а) номер перемещаемого диска

б) номер шпиля , *с которого* его нужно перенести

в) номер шпиля , *на который* он будет перемещен

Номер перемещаемого диска находится из формулы $i = (2k + 1) * 2^{A-1}$ описанным выше образом .

Что же касается номеров шпилей , заметим такую закономерность (опираясь на дерево ходов) : на корне дерева (восьмой шаг), когда переносим самый большой диск, имеем такие номера шпилей : 1→ 3. Далее , если переходим к правому узлу , то меняется номер левого шпиля : 2→ 3 , а если переходим к левому , меняется номер правого : 1→ 2 .(Все это следует в том числе из рассмотренного вопроса номер 2)

Реализация:

Листинг 2.2:

```
int getPateNum(int move_num ){
    int plate_num = 1;
    while ( !( move_num % 2 ) ){
       move_num /= 2;
       plate_num++;
    }
    return plate_num;
}
void solve(int from,int to,int n){
    for(int i = 1; i <= pow(2,n)-1; i++ ){
        int f = from, t = to,
            plate_num = getPateNum(i),
            root_node = pow(2,n-1), needed_node = i ,
            current_node = root_node,
            step_to_next_node = pow(2,n-2);
        while (  current_node != needed_node ){
            if( needed_node > current_node ){// to right
                current_node += step_to_next_node;
                f = 6-t-f;
            }else
            if( needed_node < current_node ){
                current_node -= step_to_next_node;
```

```
            t = 6-t-f;
        }
        step_to_next_node /= 2;
    }
     cout<<" move "<< plate_num <<"th plate from "<<f<<"th to "<<t<<"th stick"<<endl;
  }
}
```

Пояснения :

Вспомогательная функция ***`int getPateNum(int move_num )`*** *возвращает номер перемещаемого диска во время хода* ***`move_num`***

На каждом наше , от ***`1`*** *до* ***`pow(2,n)-1`*** *в цикле* ***`while`*** *спускаемся по дереву , начиная от номера корня(номер хода)* ***`root_node = pow(2,n-1)`*** *с шагом*

`step_to_next_node = pow(2,n-2)`.

Каждый раз , спускаясь на уровень ниже ,уменьшаем шаг до следующего потомка в два раза (см.дерево перемещений) : ***`step_to_next_node /= 2;`***

3.Решение при помощи графов

Граф - фундаментальная структура данных ,представляющая из себя множество вершин , и множество связей между этими вершинами (ребра).

Самые часто используемые типы - (не)ориентированные (не)взвешенные графы.

Первая характеристика отвечает за направление отношений между ребрами.

В случае неориентированного графа , наличие ребра (a,b) не свидетельствует о наличии ребра (b,a).То есть , ребра (a,b) и (b,a) различны.

Вторая же характеристика относится к значениям , ассоциированным с каждым ребром :Если граф взвешенный , от с каждым ребром ассоциируется определённое значение .Это может быть расстояние от пункта **a** до пункта **b** , цена проезда от одного до другого или иная , возможно, более абстрактная информация. В невзвешенных же графах с ребрами никакая информация не ассоциируется.

Задачу о ханойской башне можно представить в виде графа [1] . Вершинами будут являться все возможные состояния шпилей , а ребра будут отображать возможность перехода из одного состояния в другое. Следовательно , это будет неориентированный невзвешенный граф. Выглядеть будет так :

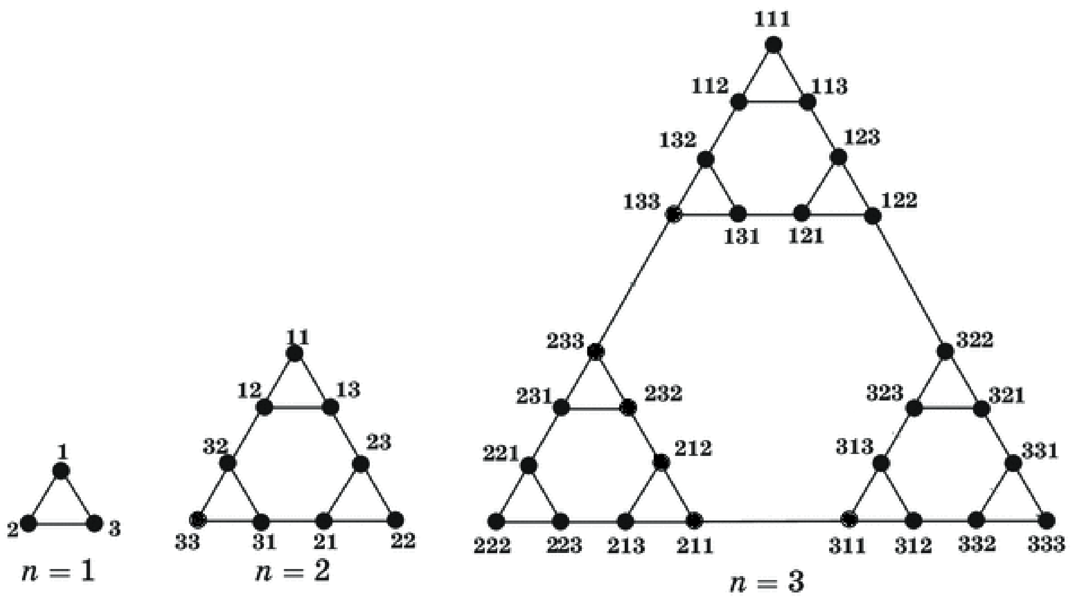

Рис.3.1
Изображение состояний "Ханойской башни" в виде графа

Вершины содержат числа длиной в n символов. Каждая цифра этого число отображает состояние одного из дисков.

Самая правая цифра (меньший разряд) описывает состояние меньшего диска , больший разряд - состояние бОльшего. Значение цифры есть номер шпиля , на котором находится диск , который данный разряд описывает.

(Стало быть , каждая цифра числа находится в диапазоне от 1 до 3)

Например , 321 описывает следующее состояние : первый (самый маленький) диск находится на первом шпиле , второй - на втором , третий диск (бОльший) - на третьем.

Заметим , что граф для n=2 состоит из трех графов для n=1 с незначительно изменёнными вершинами. И это отношение сохраняется с увеличением n. исходя из этого , можно получить формулу количества состояний для определенного n : 3^n. К подобному выводу также можно прийти , рассматривая предложенный способ представления состояния шпилей .Например , при n=3 число XYZ может кодировать 27 состояний , т.к каждая цифра может принимать только три состояния : 3*3*3 = 27.

Теперь , вернемся к нашей задаче. Угловые узлы графа отображают искомые или исходные состояния шпилей (когда все диски находятся на одном шпиле).На протяжении всего предыдущего раздела , пытаясь найти самый короткий способ решения , мы двигались от одной угловой вершины к другой , по ребрам треугольного графа.

Особенность решения , которое мы собираемся рассмотреть , заключается в том ,что можно с лёгкостью изменить первоначальную формулировку задачи, получив таким образом более общий случай!

Внимание , новая формулировка :

- Найти минимально возможную последовательность ходов для перехода из состояния A в состояние B
- A и B - произвольные разрешенные раскладки дисков на трех шпилях

- Разрешенной раскладкой считается та , при которой больший диск не лежит на меньшем.

(В частности , для решения задачи при обычной формулировке, можно рассматривать переход от раскладки 111 к раскладке 333)

Существует два основных алгоритма поиска по графу[2]: поиск в глубину , поиск в ширину.

Поиск в глубину предусматривает нахождение пути от одного узла до другого за минимально возможное время , но при этом не гарантируется , что найденный путь будет кратчайшим .

Поиск в ширину, напротив, ищет самый короткий путь от одной вершины до другой, потратив на это больше времени , по сравнению с поиском в глубину.

(Алгоритмы работают как с ориентированными , так и с неориентированными графами)

Мы , разумеется , будем использовать алгоритм поиска в ширину.

Для начала , попробуем понять суть алгоритма.(используя [2] в качестве справочника.)

Итак , поиск в ширину обходит все вершины , начиная с определённой стартовой вершины s , сохраняя при этом некоторую информацию в два массива , в книге они названы pred и dist (будем придерживаться традиций).

Массив pred[v] сохраняет предыдущий узел для узла v (с помощью данных этого массива впоследствии можно узнать путь от начальной вершины s до определенной вершины v) , а dist[v] - глубину узла v , начиная с самого первого узла s .(Это то же самое , что и дистанция между v и s).Стоит отметить также следующее : вершины , находящиеся на расстоянии d+1 от исходной вершины s , будут посещены после(!) посещения всех вершин , находящихся на расстоянии d. Во избежание циклического посещения вершин , каждая вершина окрашивается в один из трех цветов (белый , серый , черный).

Белые вершины - те , что еще не посещены , серые - те , которые посещены , но нет гарантии , что посещены все вершины , в которые можно попасть через данную (другими словами , серые вершины находятся в процессе обработки).Черные - вершины , которые полностью посещены ,и больше не нуждаются в обработке.

Рассмотрим пример реализации , приведенный в книге [2]

Листинг 3.1:

```
void bfsSearch(Graph const &graph, int s,vector<int>&dist,vector<int>&pred)
{
   // Инициализация dist и pred для пометки непосещенных вершин.
   // Начинаем с s и окрашиваем ее в серый цвет, так как ее
   // соседи еще не посещены.
   const int n = graph.numVertices();
   pred.assign(n, -1);
   dist.assign(n, numeric_limits<int>::max());
   vector<vertexColor> color (n, White);
   dist[s] = 0;
   color[s] = Gray;
   queue<int> q;
   q.push(s);
   while (!q.empty()) {
     int u = q.front();
     // Исследуем соседей u для расширения горизонта поиска
     for(VertexList::const_iterator ci = graph.begin (u);
                                     ci != graph.end (u); ++ci)
     {
          int v = ci->first;
          if (color[v] == White) {
```

```
                dist[v] = dist[u]+1;
                pred[v] = u;
                color[v] = Gray;
                q.push(v);
            }
        }
        q.pop();
        color[u] = Black;
    }
}
```

Функция поиска принимает граф , вершину , с которой следует начать поиск , и два массива , которые и будут являться результатом выполнения алгорима.

Можно обратить внимание на использование очереди queue. Благодаря ей реализуется условие посещения всех вершин на расстоянии d раньше посещения какой — либо вершины на расстоянии d+1.Для каждой вершины в очереди выполняются следующие действия:

"Вычисляются" все соседи - реализовано с помощью итерирования в цикле for.

Для каждого соседа , встречающегося в первый раз (белая вершина), сохраняются данные в массивы pred и dist , цвет вершины меняется на серый и он добавляется в очередь.

Наконец , вершина , соседи которой только - что обрабатывались , удаляется из очереди

(Подобное "размусоливание" кода может быть достаточно запутывающим)

Сохранение цвета вершины происходит при помощи массива (или вектора , в терминологии Standart Template Library [STL] C++) colors.

Следует также отметить , что не обязательно использовать именно два массива pred и dist, если задача позволяет.

Теперь мы готовы посмотреть на полное решение задачи !

Листинг 3.2:

```
#include <cmath>
#include <vector>
#include <string>
#include <queue>
#include <map>
#include "ngraph/ngraph.hpp"

using namespace NGraph;
using namespace std;

enum vertexColor { white, gray, black };
typedef Graph::vertex vertex;

vertex appendLeft(vertex v,char digit){
  return  ( 1 * pow(10, to_string(v).length()) )*digit + v;
}
void insertBiDirectedEdge (Graph& graph,Graph::vertex a,Graph::vertex b){
    graph.insert_edge(a,b);
    graph.insert_edge(b,a);
}
Graph* appendLeftOfAllNodes( const Graph& graph , char digit ){
    Graph* new_graph = new Graph(graph);
    for ( Graph::const_iterator p = graph.begin(); p != graph.end(); p++){
        long int old_node = Graph::node(p);
        long int new_node = appendLeft(old_node,digit);

        new_graph->insert_vertex(new_node);
        new_graph->absorb(new_node,old_node);
```

```
    }
    return new_graph;
}


Graph* generate( int n ){


    if( n == 1 ){
        Graph* smallestOne = new Graph;
        insertBiDirectedEdge(*smallestOne,1,2);
        insertBiDirectedEdge(*smallestOne,1,3);
        insertBiDirectedEdge(*smallestOne,2,3);
        return smallestOne;
    }
    Graph *base = generate(n-1),
          *first = appendLeftOfAllNodes(*base,1),
          *second = appendLeftOfAllNodes(*base,2),
          *third = appendLeftOfAllNodes(*base,3),
          *result = new Graph( *first + *second + *third );
    long int line_of_ones = 1;
    for(long int i = 0;i<n-2;i++) line_of_ones = line_of_ones*10 + 1;


                    insertBiDirectedEdge(*result,appendLeft(line_of_ones*1,2),
appendLeft(line_of_ones*1,3));
                    insertBiDirectedEdge(*result,appendLeft(line_of_ones*2,1),
appendLeft(line_of_ones*2,3));
                    insertBiDirectedEdge(*result,appendLeft(line_of_ones*3,2),
appendLeft(line_of_ones*3,1));


    delete first, delete second, delete third, delete base ;
    return result;
}
```

```
void bfsSearch(const Graph& graph, vertex s,
                            vertex destination,map<vertex,vertex>& pred)
{
     map<vertex,vertexColor> color;
     for ( Graph::const_iterator p = graph.begin(); p != graph.end(); p++)
        color[Graph::node(p)] = white;
     pred[s] = 0;
     color[s] = gray;
     queue<vertex> q;
     q.push(s);

     while (!q.empty()) {
        vertex u = q.front();
        Graph::vertex_set neighbors = graph.out_neighbors(u);
        for(Graph::vertex_set::iterator ci = neighbors.begin();
                                             ci != neighbors.end (); ++ci)
        {
           vertex v = Graph::node(ci);
           if (color[v] == white) {
             pred[v] = u;
             color[v] = gray;
             q.push(v);
           }
           if(v == destination) return;
        }
        q.pop() ;
        color[u] = black;
     }
}
```

```
void displayTheShortestWay( map<vertex,vertex>& pred,vertex a,vertex b){
    vertex current = b;
    int length = 0;
    while ( current != a ){
          length++;
          cout<<current<<endl;
          current = pred[current];
    }
    cout<<a<<endl;
    cout<<"Number of moves:"<<length<<endl;
}
```

Пояснения:

Функции

1. `vertex appendLeft(vertex v,char digit),`
2. `void insertBiDirectedEdge (Graph& graph,Graph::vertex a,Graph::vertex b),`
3. `Graph* appendLeftOfAllNodes( const Graph& graph , char digit ),`
4. `Graph* generate( int n ),`
5. `void displayTheShortestWay( map<vertex,vertex>& pred,vertex a,vertex b)`

Являются вспомогательными

Для реализации данного алгоритма использовалась библиотека **`ngraph`** ,

которая реализует ***ориентированный*** граф. По этой причине приходилось всегда добавлять два ребра вместо одной .Вторая функция в приведенном выше списке как раз реализует добавление вершины , как если бы это был неориентированный граф.

Функция 1 добавляет к вершине v цифру digit слева , она используется в

функции 3,которая получает граф , и возвращает другой ,ко всем вершинам которого добавлена цифра digit слева. Та , в свою очередь , используется в функции 4 для генерации графа , описывающего множество состояний ханойской башни при n дисках.

Основная функция - **`void bfsSearch(const Graph& graph, vertex s,vertex destination,map<vertex,vertex>& pred)`** немного видоизменена , по сравнению с листингом 3.1 . Во первых, здесь не используется массив dist за ненадобностью.Кроме того , появился новы аргумент — destination , он нужен , чтобы прервать выполнение функции , как только необходимая вершина достигнута.

Другая важная особенность — цикл инициализации массива (на самом деле , это не массив , а экземпляр класса map из стандартной библиотеки , но ради простоты будем использовать термин массив) colors .Т.к colors — теперь экземпляр map, а вершины , которые нужно инициализировать , расположены не по порядку , нельзя инициализировать при помощи конструктора , приходится использовать цикл. На самом деле , можно было обойтись и без цикла , но в таком случае проверка цвета вершины происходила бы более запутанным , для учебного примера , образом .

И , наконец , функция под номером 5 принимает массив pred , начальную и конечную вершины , и выводит на экран последовательность вершин (то есть состояний) для перехода из состояния **a** в состояние **b** . Состояния выводятся в обратном порядке .К сожалению , я не нашел более простого способа вывести их в прямом порядке , кроме как предварительно сохранив их в стек и еще раз пробежав по ним с помощью цикла. Этот подход не реализован , т.к код неоправданно станет еще более раздутым .

Следует также заметить , что по причине ограничений используемой библиотеки, количество дисков ограничено десятью (на моей машине) .Дело в том , что данная библиотека представляет каждую вершину как unsigned int .На одних машинах это 2 байта , на другх — 4 (как на моей, например). С помощью 4-х байтов можно кодировать 2^{32} = 4294967296 чисел .То есть, вершина графа не может содержать чисел , в которых больше 10-ти разрядов .А 10 разрядов — это 10 дисков .В случае же , когда имеем только 2 байта (2^{16} = 65536) , максимальное число дисков уменьшается до 5-ти .

Функция main может выглядеть , например , так:

```
int main()
{
    vertex a = 111, b = 333;
    map<vertex,vertex> pred;
    Graph *graph = generate(3);
    bfsSearch(*graph,a,b,pred);
    displayTheShortestWay(pred,a,b);
    return 0;
}
```

```
вывод будет таким:
333
331
321
322
122
123
113
111
Number of moves:7
```

Литература

1. С.М.Окулов, А.В.Лялин, «Ханойские башни» , Бином.Лаборатория знаний , 2008 , Москва

2. Джордж Хайнеман, Гэри Поллис, Стэнли Селков , «Алгоритмы.Справочник с примерами на C,C++,Java и Python 2-е издание» , Диалектика , 2017
3. Кнут Д., Грэхем Ф., Поташник О. Конкретная математика. Основание информатики: Пер. с англ. Изд.2, испр.2006. Твердый переплет. 704 с.