**РУСЕНСКИ УНИВЕРСИТЕТ “АНГЕЛ КЪНЧЕВ”**

**КУРСОВА РАБОТА**

**по Изкуствен интелект**

Студент:

Факултетен номер:

Група:

Специалност:

Курс:

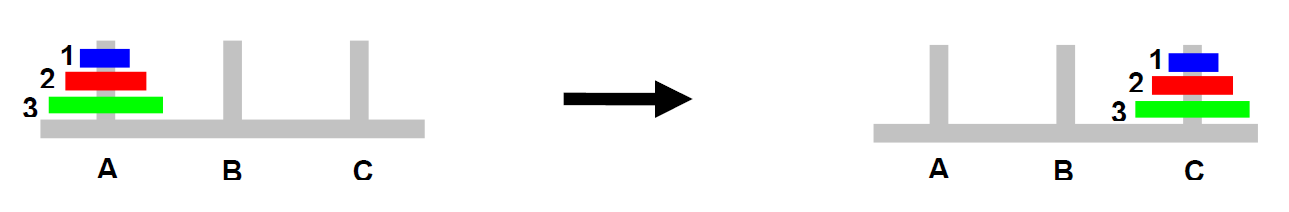
**Изготвил:**

**Дата: Проверил:**

# Задание.

**“Ханойски кули” (Towers of Hanoi problem):**

Три диска (1, 2, 3) с различна големина са разположени върху пилон А (фиг.1). Целта е те да бъдат преместени върху пилон С, като за помощен се използва пилон В. Може да се мести само най-горният диск върху по-голям диск или върху основата на пилона.



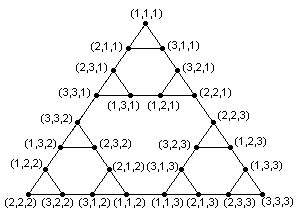
Фигура 1 Ханойски кули

Алгоритъм: търсене в широчина [*Breadth-first search (BFS)*].

1. **Представяне на задачата като задача за търсене в ПС.**

* **Състояние**: Нека състоянието на тази задача да се представи чрез наредена тройка (α1, α2, α3), където α1, α2 и α3 са номерата на пилоните на които се намират съответно първият, вторият и третият диск.
* **Начално състояние**: (1, 1, 1).
* **Финално състояние**: (3, 3, 3).

# Граф на ПС.



# Програмна реализация.

## Псевдо-код на алгоритъма.

BFS ( start, end, visited)

{

Създаваме празна опашка queue;

Добавяме към опашката върха start;

while (Опашката не е празна)

{

current = вземаме поредния елемент от опашката;

Анализираме върха p;

If (current == end) break; // върхът е намерен

Намираме всички съседи n на current

for (от j = 0 до j < n)

{

if (!visited[j])

{

добавяме „оцелелият“ наследник в опашката

„Маркираме“ j като посетен

}

}

}

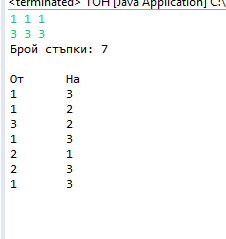
If (current == end) извеждане на резултата;

}

## Описание на разработениете модули.

* **Вид на функцията**: private static void minMovesToTarget(Node source, Node target, Set<Node> visited)
* **Действие**: Функцията служи за намиране на път между два върха в един граф чрез *Breadth-first search* алгоритъма;
* **Параметри**: source, target, visited – съответно началният, крайния връх и множеството на посетените върхове
* **Използвани функции**: printOutput()
* **Вид на функцията**: static void printOutput(Node target)
* **Действие**: Функцията служи за отпечатване на резултата;
* **Параметри**: target – крайният връх: съдържа пътя от началния връх до него;
* **Използвани функции**: няма
* **Вид на функцията**: private static Node readPegsConfiguration(int k, int n, Scanner sc)
* **Действие**: Функцията служи за четене на входа от конзолата
* **Параметри**: k, n, sc – съответно: брой кули, брой дискове, помощен обект за четене на входа от конзолата;
* **Използвани функции**: няма.
* **Вид на функцията**: public static void main(String args[])
* **Действие**: Главна функция
* **Параметри**: String args[];
* **Използвани функции**: minMovesToTarget(), readPegsConfiguration()

## Тестови примери.



Фигура 2 Тестов пример

## Код на програмата.

* TOH.java

import java.util.Collections;

import java.util.HashSet;

import java.util.LinkedList;

import java.util.List;

import java.util.Map;

import java.util.Queue;

import java.util.Scanner;

import java.util.Set;

import java.util.Stack;

import java.util.TreeMap;

public class TOH {

/\*

\* 1 1 1 - и трите диска се намират на 1-ви пилон

\* 3 3 3 - и трите диска трябва да се преместят на 3-ти пилон

\* \*/

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

int n = 3; //sc.nextInt(); // брой дискове

int k = 3; //sc.nextInt(); // брой кули

// прочитане на началното състояние

Node source = readPegsConfiguration(k, n, sc);

// прочитане на крайното състояние

Node target = readPegsConfiguration(k, n, sc);

// за следене на посетените върхове и избягване на цикли

Set<Node> visited = new HashSet<>();

try {

minMovesToTarget(source, target, visited);

} catch (Exception ex) {

System.out.println("Exception = " + ex);

}

}

private static void minMovesToTarget(Node source, Node target, Set<Node> visited) throws CloneNotSupportedException {

// извършване на търсене в широчина - BFS

Queue<Node> q = new LinkedList<>();

// добавяне на началния връх в опашката

q.add(source);

Node current = source;

while (!q.isEmpty()) {

current = (Node) q.poll();

if (current.equals(target)) { // върхът е намерен

break;

}

List<Node> neighbors = current.neighbors();

if (neighbors.size() > 0) {

for (Node n : neighbors) {

if (!visited.contains(n)) {// ако върхът не е посетен

// добавя се в опашката

q.offer(n);

visited.add(n);

}

}

}

}

// отпечатване на резултата ако върхът е намерен

if (current.equals(target)) {

printOutput(current);

}

}

private static Node readPegsConfiguration(int k, int n, Scanner sc) {

Stack<Integer>[] initialState = new Stack[k];

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

initialState[i] = new Stack<Integer>();

}

// четене и обръщане т.к е нужно да поставиме елементите

// в намаляващ ред

// дискът е ключът, а кулата е стойността

TreeMap<Integer, Integer> map = new TreeMap<Integer, Integer>(Collections.reverseOrder());

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

map.put(i, sc.nextInt());

}

// prepare towers

for (Map.Entry<Integer, Integer> entry : map.entrySet()) {

initialState[entry.getValue() - 1].push(entry.getKey());

}

return new Node(initialState);

}

static void printOutput(Node target) {

Stack<Move> stack = new Stack<>(); // използваме стек, тъй като трябва да отпечатаме

// пътя от началния до крайния връх

while (target.parent != null) {

stack.push(target.move);

target = target.parent;

}

System.out.println("Брой стъпки: " + stack.size());

System.out.println("\nOт\tНа");

while (!stack.isEmpty()) {

System.out.println(stack.pop().toString());

}

}

}

* Node.java

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import java.util.Stack;

class Node implements Cloneable {

// кули

Stack<Integer>[] state = null;

Node parent = null;

Move move = null; // при преминаване на следващия връх

public Node(Stack<Integer>[] st) {

state = st;

}

@Override

protected Node clone() throws CloneNotSupportedException {

Stack<Integer>[] cloneStacks = new Stack[state.length];

for (int i = 0; i < state.length; i++) {

cloneStacks[i] = (Stack<Integer>) state[i].clone();

}

Node clone = new Node(cloneStacks);

return clone;

}

// връща на съседните върхове

public List<Node> neighbors() throws CloneNotSupportedException {

List<Node> neighbors = new ArrayList<>();

int k = state.length;

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

for (int j = 0; j < k; j++) {

if (i != j && !state[i].isEmpty()) {

// нужно е да клонираме за да избегнем "родителя"

// Hint - в java обектите не са mutable и са референтни типове

Node child = this.clone();

// преместване

if (canWeMove(child.state[i], child.state[j])) {

child.state[j].push(child.state[i].pop());

child.parent = this;

child.move = new Move(i, j);

neighbors.add(child);

}

}

}

}

return neighbors;

}

public boolean canWeMove(Stack<Integer> fromTower, Stack<Integer> toTower) {

boolean answer = false;

if (toTower.isEmpty()) {// ако целевата кула е "празна" -

// може да преместим диска

return true;

}

int toDisc = (int) toTower.peek();

int fromDisc = (int) fromTower.peek();

if (fromDisc < toDisc) { // може да поставяме само по-малък диск

answer = true;

}

return answer;

}

@Override

public int hashCode() {

int hash = 7;

return hash;

}

@Override

public boolean equals(Object obj) {

if (obj == null) {

return false;

}

if (getClass() != obj.getClass()) {

return false;

}

final Node other = (Node) obj;

if (!Arrays.deepEquals(this.state, other.state)) {

return false;

}

return true;

}

}

* Move.java

class Move {

int towerFrom, towerTo;

public Move(int f, int t) {

towerFrom = f + 1;

towerTo = t + 1;

}

@Override

public String toString() {

return towerFrom + "\t" + towerTo;

}

}

# Творческа задача.

Необходимо е да се премахне логиката за прекратяване на търсенето след като е намерен крайният връх и да не се маркират вече посетените върхове. Също така трябва да се има предвид и зациклянето.