**РУСЕНСКИ УНИВЕРСИТЕТ “АНГЕЛ КЪНЧЕВ”**

**КУРСОВА РАБОТА**

**№1**

**по Изкуствен интелект**

Студент: **Бейтула Хамди Реджеб**

Факултетен номер: **103217**

Група: **26Б**

Специалност: **КСТ**

Курс: **3**

**Изготвил:**

**Дата: Проверил:**

# Задание.

“Мисионери и канибали” (Missionary and cannibal problem) - I-ви вариант:

N мисионери (N>=1) и N канибали се намират на левия бряг на една река. Целта на задачата е всички мисионери и канибали да бъдат прехвърлени на десния бряг на реката като се използва лодка. Ограниченията са следните:

1. Лодката е най-много за двама човека (единият, от които гребе).
2. На кой да е бряг броят на мисионерите трябва да е по-голям или равен на броя на канибалите.

Нека разходите за превозването на канибал са 1 единица, а на мисионер - 2 единици.

*Забележки:*

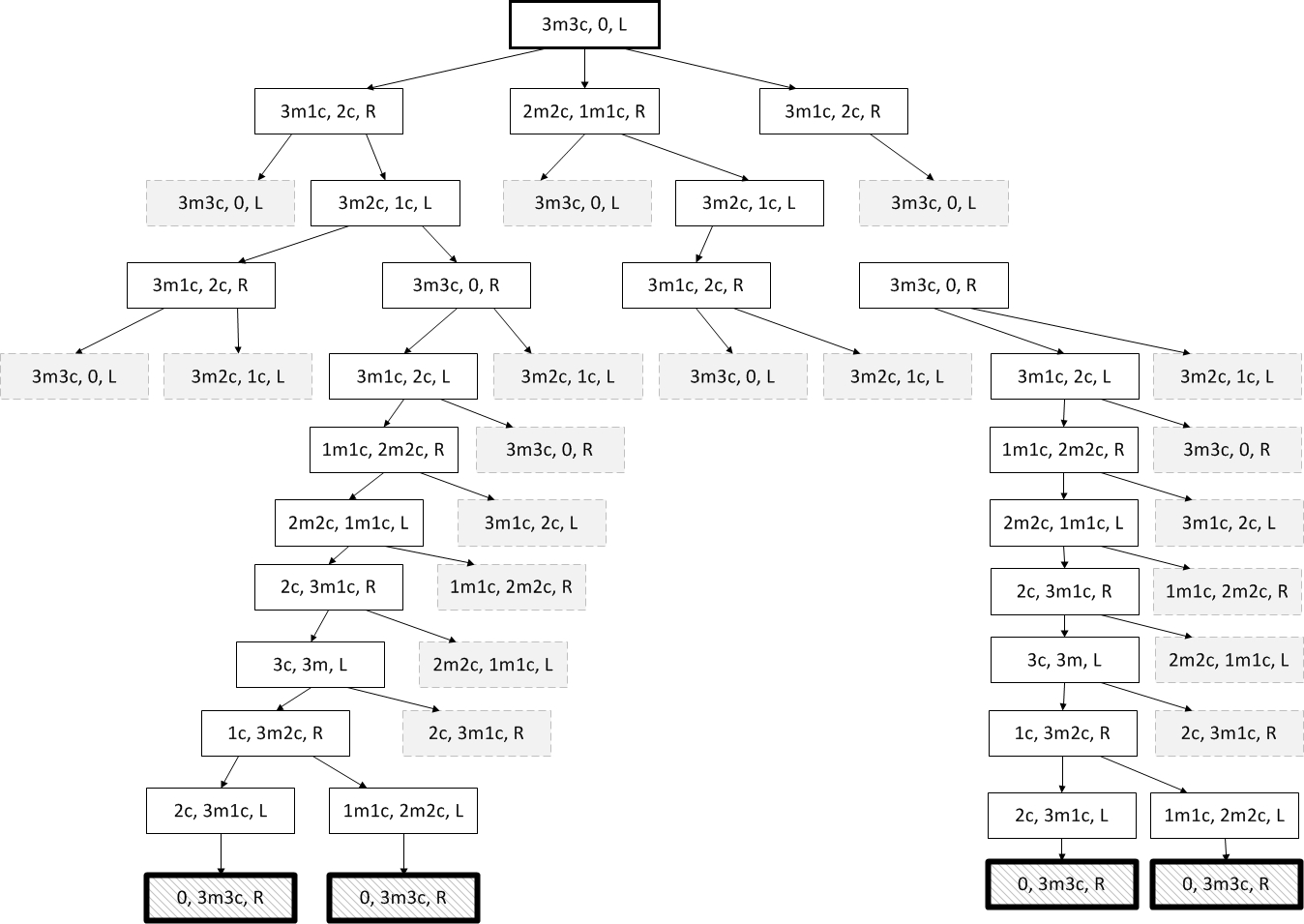
1. Предложете решение на задачата за N=3.
2. Предложеното решение за избор на състояние да е различно от това в ръководството!

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

# 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 и α2 описват кои хора са съответно на левия и десния бряг, а α3 показва къде се намира лодката. Ако със символа m се означат мисионерите, с c – канибалите , а с L и R – съответно ляв и десен бряг.
* **Начално състояние** – (3m3c, 0, L).
* **Финално състояние** – (0, 3m3c, R).
* **Оператори и разходи на стрелките** –
* **1m** – прехвърляне на един мисионер (2 ед.);
* **2m** – прехвърляне на два мисионера (4 ед.);
* **1m1c** – прехвърляне на 1 мисионер и 1 канибал (3 ед.);
* **1c** – прехвърляне на един канибал (1 ед.);
* **2c** – прехвърляне на два канибала (2 ед.).

# Граф на ПС.



# Програмна реализация.

## Псевдо-код на алгоритъма.

procedure breadth\_first\_search;

begin

OPEN:=[START];

CLOSED:=[ ];

while OPEN<>[ ] do begin

Изтрий най-лявото състояние на OPEN и го запиши в X;

if X=GOAL then return(success)

else begin

Намери наследниците на X;

Запиши X в CLOSED;

Изтрий тези наследници на X, които са вече в OPEN или CLOSED;

Запиши отдясно на OPEN “оцелелите” наследници;

end; {else}

end; {while}

return(failure);

end;

## Описание на разработениете модули.

* **Клас State**
  + **Член-данни:**

**int m** –брой мисионери;

**int c** –брой канибали;

**int b** –състояние на лодката;

**static int level** –брояч на нивото;

**static int price** –цена/разходи на пътя;

**State pred** –„родител“ на състоянието.

* + **Член-функции:**

**State(State s)** –копиращ конструктор;

**State(int m, int c, int b)** –инициализиращ конструктор;

**boolean isValid()** –ф-я за валидация на дадено състояние;

**State move(int m, int c)** –ф-я за генериране на състояние;

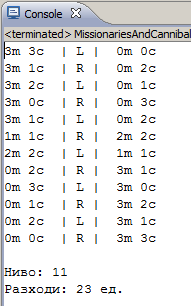
**void display()** –функция за печатане на резултата (пътя).

* **Клас MissionariesAndCannibalsBFS**
  + **Член-функции:**

**private void breadthFirstSearch()** – функция за търсене в широчина;

**public static void main(String[] args)** – главна функция.

## Тестови примери.



## Код на програмата.

**import** java.util.LinkedList;

**import** java.util.Queue;

**class** State {

**int** m; // брой мисионери

**int** c; // брой канибали

**int** b; // 0 – лодката е на левия бряг, 1 - лодката е на десния бряг

**static** **int** *level*; // брояч на нивото

**static** **int** *price*; // цена/разходи на пътя

State pred; // "родител" на състоянието

// копиращ конструктор

State(State s) {

m = s.m; c = s.c; b = s.b;

}

// инициализиращ конструктор

State(**int** m, **int** c, **int** b) {

**this**.m = m; **this**.c = c; **this**.b = b;

}

// член-функция за проверка на състояние за валидност

**boolean** isValid() {

**if** (m < 0 || m > 3 || c < 0 || c > 3) **return** **false**;

**if** (3 - m < 0 || 3 - m > 3 || 3 - c < 0 || 3 - c > 3) **return** **false**;

**if** (m > 0 && c > m) **return** **false**;

**if** (3 - m > 0 && 3 - c > 3 - m) **return** **false**;

**return** **true**;

}

// член-функция за генериране на състояние

State move(**int** m, **int** c) {

State ans = **new** State(**this**);

**if** (b == 0) {

ans.m = **this**.m - m;

ans.c = **this**.c - c;

ans.b = 1;

}

**else** {

ans.m = **this**.m + m;

ans.c = **this**.c + c;

ans.b = 0;

}

ans.pred = **this**;

**return** ans;

}

// член-функция за печатане на резултата (пътя)

**void** display() {

**if** (pred != **null**) {

pred.display();

**if** (**this**.m >= **this**.pred.m)

State.*price* += (**this**.m - **this**.pred.m) \* 2;

**else**

State.*price* += (**this**.pred.m - **this**.m) \* 2;

**if** (**this**.c >= **this**.pred.c)

State.*price* += **this**.c - **this**.pred.c;

**else**

State.*price* += **this**.pred.c - **this**.c;

State.*level*++; // трябва да се премахне, за да се намeрят други решения

}

System.*out*.print(m + "m " + c + "c " + (b == 0 ? " | L | " : " | R | "));

System.*out*.println((3 - m) + "m " + (3 - c) + "c ");

}

}

**public** **class** MissionariesAndCannibalsBFS {

// търсене в широчина / breadth-first search (BFS)

**private** **void** breadthFirstSearch() {

// тримерен масив, който съхранява информация затова

// дали дадено състояние вече е изследвано или не

**boolean**[][][] visited = **new** **boolean**[4][4][2];

**int** i, j;

Queue<State> q = **new** LinkedList<State>();

State start = **new** State(3, 3, 0);

State.*level* = 0;

State.*price* = 0;

q.add(start);

**while** (!q.isEmpty()) {

State next = q.poll(); // q.remove();

**if** (next.m == 0 && next.c == 0 && next.b == 1) {

next.display();

System.*out*.printf("\nНиво: %d", State.*level*);

System.*out*.printf("\nРазходи: %d ед.", State.*price*);

**break**;

}

**else** {

**for** (i = 0; i <= 2; i++)

**for** (j = 0; i + j <= 2; j++) {

**if** (i == 0 && j == 0)

**continue**;

State p = next.move(i, j);

**if** (!p.isValid() || visited[p.m][p.c][p.b])

**continue**;

visited[p.m][p.c][p.b] = **true**;

q.add(p);

}

}

}

}

**public** **static** **void** main(String[] args) {

**new** MissionariesAndCannibalsBFS().breadthFirstSearch();

}

}

# Творческа задача.

Достатъчно е да се премахне оператора **break** за преждевременно излизане от цикъла, да не се помни информация за възлите, които вече са посетени, т.е да се премахне **visited[ ][ ][ ]** (***Closed[ ]***), след като се намери първото решение да се запази в някаква променлива текущото ниво (оптимално ниво) и всеки път при достигане на целта да се провери нивото дали е по-малко или равно на оптималното. Ако е така да се разпечатва резултата, иначе – излизане от цикъла. По този начин ще се намерят всички решения до оптималното ниво, но при тази задача възможните решения са четири и са на 11. ниво (оптималното ниво).

**private** **void** breadthFirstSearch() {

**int** i, j, count = 0;

**boolean** isFirstFound = **false**; // първи намерен път ли е?

**boolean** isAllFound = **false**; // последно намерен път ли е?

**int** optimalLevel = 0; // оптимално ниво

Queue<State> q = **new** LinkedList<State>();

State start = **new** State(3, 3, 0);

State.*level* = 0;

State.*price* = 0;

q.add(start);

**while** (!q.isEmpty() && !isAllFound) {

State next = q.poll();

State.*level*++;

**if** (next.m == 0 && next.c == 0 && next.b == 1) {

**if** (isFirstFound) {

**if** (State.*level* <= optimalLevel) {

count++;

System.*out*.println(count + ". решение:");

next.display();

System.*out*.printf("\nРазходи: %d ед.", State.*price*);

State.*price* = 0;

System.*out*.println();

}

**else** {

isAllFound = **true**;

}

}

**else** {

isFirstFound = **true**;

count++;

System.*out*.println(count + ". решение:");

next.display();

System.*out*.printf("\nРазходи: %d ед.", State.*price*);

State.*price* = 0;

optimalLevel = State.*level*;

State.*level* = 0;

System.*out*.println();

}

}

**else** {

**for** (i = 0; i <= 2; i++)

**for** (j = 0; i + j <= 2; j++) {

**if** (i == 0 && j == 0)

**continue**;

State p = next.move(i, j);

**if** (!p.isValid())

**continue**;

q.add(p);

}

}

}

}