Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
Ilkovičova 2, 842 16 Bratislava 4

Predmet / Subject

– Umelá inteligencia / Artificial intelligence –

- Dokumentácia / Documentation -

Zadanie č.2

Ak. Rok / Academic term: 2022/2023, zimný semester

Cvičiaci / Instructors:
Ing. Martin Komák, PhD.

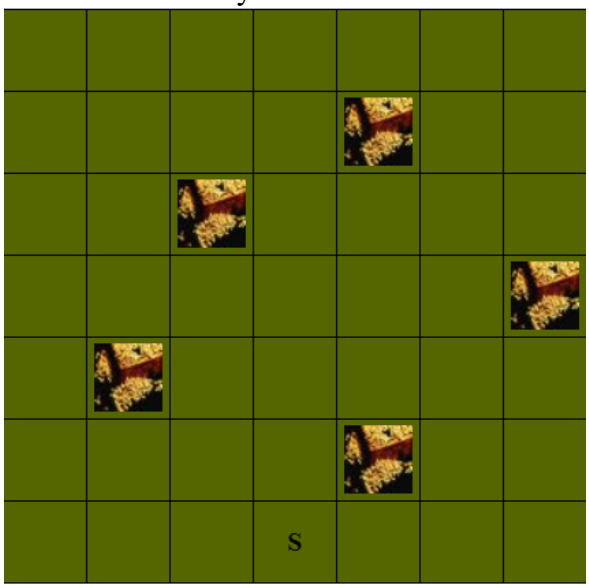
Študent / Student:
Adam Grík

Bratislava, 2022.

Obsah / Content:

1 Úvod do problematiky

Mojou úlohou v tomto zadaní bolo vyriešiť problém hľadača pokladov. Máme hľadača pokladov, ktorý sa pohybuje vo svete definovanom dvojrozmernou mriežkou (viď. obrázok) a zbiera poklady, ktoré nájde po ceste. Začína na políčku označenom písmenom S a môže sa pohybovať štyrmi rôznymi smermi: hore H, dole D, doprava P a doľava L. K dispozícii má konečný počet krokov. Jeho úlohou je nazbierať čo najviac pokladov. Za nájdenie pokladu sa považuje len pozícia, pri ktorej je hľadač aj poklad na tom istom políčku. Susedné políčka sa neberú do úvahy.



Danú úlohu riešim pomocou **evolučného programovania nad virtuálnym strojom**.

2 Evolučné programovanie nad virtuálnym strojom

Tento spôsob programovania využíva spoločnú pamäť pre údaje a inštrukcie. Vždy na začiatku programu je pamäť vynulovaná a naplnená náhodnými inštrukciami. Následne náš virtuálny stroj číta postupne všetky bunky od prvej pamäťovej bunky. Inštrukcie modifikujú pamäťové bunky. Modifikiácie sú napr. inkrementácia alebo dekrementácia, prípadne skok alebo výpis ale tieto dve inštrukcie nemodifikujú pamäťové bunky. Vykonávanie virtuálneho stroja s pamäťovými bunkami jedného jedinca sa končí, ak virtuálny stroj vykonal 500 inštrukcií, ak jedinec vyšiel z hracieho poľa, alebo ak jedinec našiel všetkých 5 pokladov.

3 Virtuálny stroj

Stroj má 64 pamäťových buniek o veľkosti 1 byte.
Bude poznať štyri inštrukcie: inkrementáciu hodnoty pamäťovej bunky, dekrementáciu hodnoty pamäťovej bunky, skok na adresu a výpis (H, D, P alebo L) podľa hodnoty pamäťovej bunky.

inštrukcia	tvar
inkrementácia	00XXXXXX
dekrementácia	01XXXXXX
skok	10XXXXXX
výpis	11XXXXXX

Hodnota XXXXXX predstavuje 6-bitovú adresu pamäťovej bunky s ktorou inštrukcia pracuje. Posledná inštrukcia výpis zabezpečuje pohyb jedinca po poli. Pri inštrukcii výpis sa program pozrie na posedné dve hodnoty v danej pamäťovej bunke, ak sú hodnoty 11 tak hráč vykoná pohyb smerom hore, ak 00 tak vykoná pohyb smerom dole, ak 10 tak vykoná pohyb doľava, a ak 01 tak hráč vykoná pohyb doprava.

4 Popis implementovaného evolučného algoritmu

Riešenie tohto problému implementujem v programovacom jazyku Java.

4.1 Opis populácie, jedinca a génu

4.1.1 Populácia

Každá populácia sa nachádza v samostatnej triede, v ktorej sa nachádza pole jedincov (trieda Player)

```
public class Population {
    Player[] players;
    Player bestPlayer;

    Population(Player[] players) {
        this.players = players;
    }
}
```

4.1.2 Jedinec

Každý jedinec je reprezentovaný triedou Player. Svoje gény ma uložené v premenej originalGenes, ktorú reprezentuje pole triedy DataCell. Následne trieda Player obsahuje ďalšie premenné ako napr. hodnotu fitness, počet vykonaných krokov, String hodnotu vykonaných pohybov, a boolean hodnotu či je jedinec v danej populácií najlepší alebo nie.

```
public class Player {
    DataCell[] originalGenes;
    int fitness;
    int numberOfMoves;
    String moves = "";
    boolean isBest = false;

    Player(DataCell[] originalGenes) {
        this.originalGenes = originalGenes;
        this.moves = "";
    }
```

4.1.3 Gény

Jeden gén každého jedinca je reprezentovaný triedou DataCell, v ktorej sa nachádzajú dve polia bajtov. Prvé pole obsahuje adresu danej dátovej bunky, a druhé pole obsahuje hondotu danej pamäťovej bunky.

```
public class DataCell {
    byte[] address;
    byte[] value;

    DataCell(byte[] address, byte[] value) {
        this.address = address;
        this.value = value;
    }
}
```

4.2 Inicializácia prvej generácie

Inicializáciu prvej generácie vykonávam pomocou cyklu for, kde pre každého jedinca v generácií generujem náhodné hodnoty v pamäťových bunkách pomocou metódy createGenesForPlayer().

```
Player[] players = new Player[101];
for (int i = 0; i < players.length; i++) {
    DataCell[] originalGenes = createGenesForPlayer();
    players[i] = new Player(originalGenes);
    num = num + 1;
}
```

V metóde createGenesForPlayer() si ako prvé povytváram všetky potrebné premenné, následne si ako prvé nainicializujem 64 hodnôt v binárnej sústave.

```
public static DataCell[] createGenesForPlayer() {
    // initialize addresses for virtual machine from 000000(0) to 111111(63)
    byte[] address = {0, 0, 0, 0, 0, 0};
    byte[][] addressesArray = new byte[64][];   // array of 64 addresses
    DataCell[] virtualAddresses = new DataCell[64];     // array of 64 data cells
    byte[][] firstTwoBits = {{0, 0}, {0, 1}, {1, 0}, {1, 1}};   // first two bits
    int min = 0;
    int max = 3;
    int max2 = 63;    // minimum and maximum for random numbers

    // initialize 64 addresses for virtual machine
    for (int i = 0; i < 64; i++) {
        if (i == 0) {
            addressesArray[i] = address;
            virtualAddresses[i] = new DataCell(addressesArray[i], value: null);
        } else {
            addressesArray[i] = increment(addressesArray[i - 1]);
            virtualAddresses[i] = new DataCell(addressesArray[i], value: null);
        }
    }
```

Hodnoty dátových buniek následne inicializujem v ďalšom cykle for, kde do každej bunky najprv naincializujem random prvé dva bajty, a následne random vkladám adresy od 0 po 63 z daného pola všetkých adries.

```
// initialize 64 values for addresses in virtual machine
for (int i = 0; i < 64; i++) {
    int randomForFirstTwoBits = (int) (Math.random() * (max - min + 1) + min);
    int randomForNextBits = (int) (Math.random() * (max2 - min + 1) + min);
    byte[] valueOfDataCell = new byte[8];
    int index = 0;
    for (int j = 0; j < 2; j++) {
        valueOfDataCell[index] = firstTwoBits[randomForFirstTwoBits][j];
        index++;
    }
    for (int j = 0; j < 6; j++) {
        valueOfDataCell[index] = addressesArray[randomForNextBits][j];
        index++;
    }
    virtualAddresses[i].value = valueOfDataCell;
}
return virtualAddresses;
```

4.3 Virtuálny stroj

Celý priebeh programu následne prebieha v rekurzívnej funkcií doGame().

Všetko prebieha v hlavnom for cykle, kde prechádzam cez každého jedinca v danej generácií. V prvej podmienke zisťujem, či daný jedinec je doposiaľ najlepší nájdený, ak je tak prechádzam na ďalšieho jedinca. Ako ďalšie si prekopírujem gény daného jedinca do premennej temp, aby som si neprepisoval jeho pôvodné gény.

```
mainLoop:
for (int x = 0; x < population.players.length; x++) {

    if(population.players[x].isBest == true) {
        continue  mainLoop;
    }

    String[][] array = createArray();
    int numberOfInstructions = 0;
    int numberOfMoves = 0;
    int numberOfFoundTreasures = 0;
    int index = 0;

    DataCell[] temp = new DataCell[64];
    for(int i = 0; i < temp.length; i++) {
        temp[i] = new DataCell(population.players[x].originalGenes[i].address, population.players[x].originalGenes[i].value);
    }
```

Následne ešte stále vo funkcii doGame(), prechádzam do cyklu for, ktorý prebehne 500 krát, alebo pokiaľ jedinec nevyjde z daného hracieho poľa. V tomto cykle sa vykonávajú všetky dané inštrukcie, inkrementácia, dekrementácia, skok a vypís.
Všetky tieto inštrukcie vykonávam podľa toho, čo sa nachádza na prvých dvoch bitoch hodnoty dáneho génu. Ak sa na prvých dvoch bitoch nachádza 00, tak následne si do premennej desiredAdress vložím adresu, pre ktorú sa má daná inštrukcia vykonať, potom prechádzam cez pole všetkých génov daného jedinca a ak sa adresa zhoduje s adresou v premennej desiredAdress, tak zavolám metódu increment(), ktorá inkrementuje hodnotu v danom géne.
Takýmto istým princípom fungujú aj inštrukcia dekrementovania, akurát že volám metódu decrement(), a takto isto funguje aj inštrukcia skok, s tým že v tom prípade skáčem na danú adresu.

```
// loop for 500 instructions, do virtual machine, find treasures, do moves
for (int k = 0; k < 500; k++) {


    if (index > 63) {
        index = 0;
    }

    // incrementing case
    if (temp[index].value[0] == 0 && temp[index].value[1] == 0) {
        numberOfInstructions++;
        // create array for which address looking for
        byte[] desiredAddress = {temp[index].value[2], temp[index].value[3], temp[index].value[4],
                temp[index].value[5], temp[index].value[6], temp[index].value[7]};
        // find address which is equal to desired address
        for (int j = 0; j < temp.length; j++) {
            if (Arrays.equals(temp[j].address, desiredAddress)) {
                temp[j].value = increment(temp[j].value);
                break;
            }
        }
        index++;
        continue;
    }
```

Pri inštrukcii pohybu volám metódu doMove(), ktorá zabezpečuje pohyb hráča po poli a vracia nové pole, ktoré vznikne po vykonaní pohybu, ak metóda nevráti žiadne pole znamená že jedinec vyšiel z daného hracieho poľa, to znamená že cyklus for pre tohto jedinca končí, následne ešte zistíme jeho fitness, počet vykonaných pohybov a prechádzame na ďalšieho jedinca v generácií.

```
// move case
if (temp[index].value[0] == 1 && temp[index].value[1] == 1) {
    numberOfInstructions++;
    byte[] desiredAddress = {temp[index].value[2], temp[index].value[3], temp[index].value[4],
            temp[index].value[5], temp[index].value[6], temp[index].value[7]};
    for (int j = 0; j < temp.length; j++) {
        if (Arrays.equals(temp[j].address, desiredAddress)) {
            String[][] pomArray = copy(array);
            array = doMove(array, temp[j].value, population.players[x]);
            numberOfMoves++;
            if (array == null) {
                numberOfFoundTreasures = 5 - checkFoundTreasures(pomArray);
                population.players[x].fitness = numberOfFoundTreasures;
                population.players[x].numberOfMoves = numberOfMoves;
                continue mainLoop;
            }
            break;
        }
    }
    index++;
```

Ako ďalšie v metóde doMove() nájdem najlepšieho jedinca v danej populácií, ak najlepší jedinec našiel všetky poklady tak vychádzame z metódy a tým končí aj celý program.

```
    Player bestPlayer = new Player(findTheBestPlayer(population));
    System.out.println("Najlepsi jedinec " + generationNumber + ". generacie: " +
            "pocet najedenych pokladov: " + bestPlayer.fitness + ", " +
            "pocet krokov hraca: " + bestPlayer.numberOfMoves + ", " +
            "Postupnost krokov: " + bestPlayer.moves);

    if(bestPlayer.fitness == 5) {
        return;
    }
```

Za ďalšie zisťujeme v podmienke či sa ešte nenachádzame na poslednej generácií, ak nie tak ideme vytvárať novú populáciu v metóde doCrossAndMutation, ak už je vytvorená nová generácia vďaka kríženiu a mutovaniu, tak rekurzívne voláme opäť funkciu doGame().

```
if(numberOfGeneration != 0) {

    generationNumber++;
    Player[] newPlayers = doCrossAndMutation(population, bestPlayer);
    Population newPopulation = new Population(newPlayers);
    doGame(newPopulation, numberOfGeneration, worksheet, workbook, generationNumber);
}
```

Ako posledné, ak sa nachádzame už na poslednej generácií, tak sa pýtame používateľa či chce vytvoriť ešte jednu generáciu, alebo chce ukončiť program.

```
if(numberOfGeneration == 0) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Prajes si vytvorit dalsiu generaciu (y-ano,n-nie): ");
    String string = myObj.nextLine();
    if(string.equals("y")) {
        generationNumber++;
        Player[] newPlayers = doCrossAndMutation(population, bestPlayer);
        Population newPopulation = new Population(newPlayers);
        doGame(newPopulation, numberOfGeneration, worksheet, workbook, generationNumber);
    }
    else{
        return;
    }
}
```

4.4 Kríženie

Kríženie nových jedincov prebieha v metóde doCrossAndMutation(). Celý proces kríženia prebieha v cykle for, ktorý prebehne 50 krát, a v každom cykle vytvorím vždy dvoch nových jedincov, to znamená že vytvorím dokopy 100 nových jedincov do novej populácie. Ako prvé pomocou metódy getFatherAndMother() získam metódou selekcie turnaj dvoch jedincov (otec a mama) z predošlej generácie. Potom idem vytvárať prvého potomka týchto dvoch rodičov, a to tak že si zvolím náhodný bod zlomu (napr. 30 pre vysvetlenie), to znamená, že nový jedinec zdedí gény od 1 po 30 od otca a od 30 po 64 zdedí po mame. Druhý potomok v tomto cykle bude vytvorený podobne ale opačne, to znamená, že gény od 1 po 30 zdedí od mamy a gény od 30 po 64 zdedí po otcovi.

```
// method for make new population
public static Player[] doCrossAndMutation(Population population, Player bestPlayer) {
    Player[] newPlayers = new Player[101];

    int index = 0;
    int num = 0;
    for (int i = 0; i < 50; i++) {
        Player[] fatherAndMother = getFatherAndMother(population);


        int min = 0;
        int max = 63;
        int random = (int) (Math.random() * (max - min + 1) + min);
        DataCell[] temp1 = new DataCell[64];
        for (int j = 0; j < random; j++) {
            temp1[j] = fatherAndMother[0].originalGenes[j];
        }
        for (int j = random; j < 64; j++) {
            temp1[j] = fatherAndMother[1].originalGenes[j];
        }
        DataCell[] temp1Mutated = doMutation(temp1);
        newPlayers[index] = new Player(temp1Mutated);

        num = num + 1;
```

Na záver tejto metódy, ešte do novej populácie pridám najlepšieho jedinca z predošlej generácie.

```
DataCell[] temp = new DataCell[64];
for(int i = 0; i < temp.length; i++) {
    temp[i] = new DataCell(bestPlayer.originalGenes[i].address, bestPlayer.originalGenes[i].value);
}
newPlayers[100] = new Player(temp);
newPlayers[100].moves = bestPlayer.getMoves();
newPlayers[100].fitness = bestPlayer.getFitness();
newPlayers[100].numberOfMoves = bestPlayer.getNumberOfMoves();
newPlayers[100].isBest = true;


return newPlayers;
```

4.5 Selekcia

Selekciu rodičov určujem **turnajovým spôsobom**.
V mojom programe má na starosti selekciu rodičov metóda getFatherAndMother(). Selekcia prebieha tak, že ako prvé randomne zamiešam populáciu jedincov, následne vyberiem prvý pár jedincov, a potom druhý pár jedincov. Nakoniec z každého páru vyberiem toho jedinca, ktorý má lepšie fitness, a tým pádom sa z týchto dvoch jedincov stávajú otec a mama, ktorých posielam do funkcie doCrossAndMutation(), a vznikajú z nich nový jedinci.

```
public static Player[] getFatherAndMother(Population population) {

    List<Player> list = new ArrayList<~>(population.players.length);
    Collections.addAll(list, population.players);

    // get first couple for father
    Collections.shuffle(list);

    Player[] firstCouple = new Player[2];
    for (int i = 0; i < 2; i++) {
        firstCouple[i] = list.get(i);
        list.remove(i);
    }

    // get second couple for mother
    Collections.shuffle(list);
    Player[] secondCouple = new Player[2];
    for (int i = 0; i < 2; i++) {
        secondCouple[i] = list.get(i);
        list.remove(i);
    }


    Player[] twoBest = new Player[2];

    twoBest[0] = new Player(firstCouple[0].originalGenes);
    if (firstCouple[1].fitness > firstCouple[0].fitness) {
        twoBest[0] = firstCouple[1];
    }

    twoBest[1] = new Player(secondCouple[0].originalGenes);
    if (secondCouple[1].fitness > secondCouple[0].fitness) {
        twoBest[1] = secondCouple[1];
    }

    return twoBest;
}
```

4.6 Mutácia

Mutácia prebieha v metóde doMutation(), a to tak, že ako prvé si randomne zvolím 8 čísel. Následne v cykloch for prechádzam najprv poľom všetkých génov, a potom poľom random čísel, a mutujem gény, ktoré sa nachádzajú na 8mich randomne zvolených pozíciách. Mutujem ich tak, že 1ky zamieňam za 0, a 0 zamieňam za 1ky.

```
// method for make mutation
public static DataCell[] doMutation(DataCell[] genes) {
    int min = 0; int max = 63;
    int[] arrayOfRandomNumbers = new int[8];
    for(int i = 0; i < arrayOfRandomNumbers.length; i++) {
        int random = (int) (Math.random() * (max - min + 1) + min);
        arrayOfRandomNumbers[i] = random;
    }

    DataCell[] temp = new DataCell[64];
    for(int i = 0; i < temp.length; i++) {
        temp[i] = new DataCell(genes[i].address, genes[i].value);
    }

    for(int i = 0; i < temp.length; i++) {
        for(int j = 0; j < arrayOfRandomNumbers.length; j++) {
            if(i == arrayOfRandomNumbers[j]) {
                temp[i].value = changeZerosAndOnes(temp[i].value);
            }
        }
    }

    return temp;

}
```

4.7 Nová generácia s použitím elitárstva

Ako druhý spôsob tvorby novej generácie som pre porovnanie použil kombináciu 20% elitárstva a selekcie turnaj, to znamená že do ďalšej generácie išlo automaticky 20 najlepších jedincov, bez akéhokoľvek kríženia, títo jedinci iba zmutovali.

Princíp je v podstate rovnaký ako pri predošlej tvorbe, akurát s tým že na začiatku som pridal do novej generácie 20 najlepších jedincov, ktorí iba zmutovali.

```
public static Player[] doCrossAndMutationWithElitism(Population population, Player bestPlayer) {
    Player[] newPlayers = new Player[101];

    ArrayList<Player> list = new ArrayList<~>(Arrays.asList(population.players));
    sort(list);

    Collections.reverse(list);

    int index = 0;
    for(int i = 0; i < 20; i++) {
        DataCell[] temp = new DataCell[64];
        for(int j = 0; j < 64; j++) {
            temp[j] = list.get(i).originalGenes[j];
        }
        DataCell[] tempMutated = doMutation(temp);
        newPlayers[index] = new Player(tempMutated);
        index++;
        list.remove(i);
    }

    Player[] players = list.toArray(new Player[list.size()]);
    Population newPopulation = new Population(players);
```

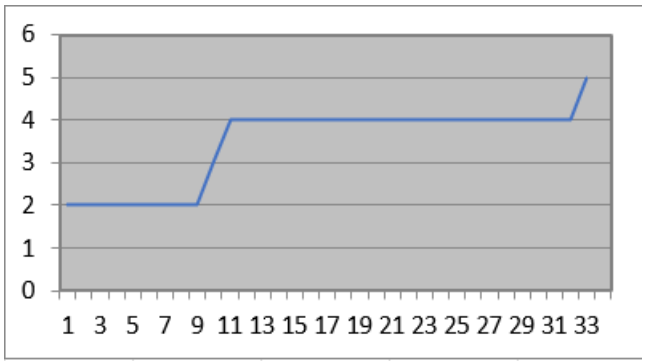
5 Testovanie

Testovanie som vykonával na vzorke 50 populácií, kde v každej populácií bolo 101 jedincov. Pomocou externej knižnice Aspose Cells mi program automaticky vytvorí graf v programe Excel z každého spusteného riešenia.
Program dosahuje pomerne dobré výsledky, a to aj napriek tomu že riešenie je do určitej miery ovplyvnené faktorom náhody.

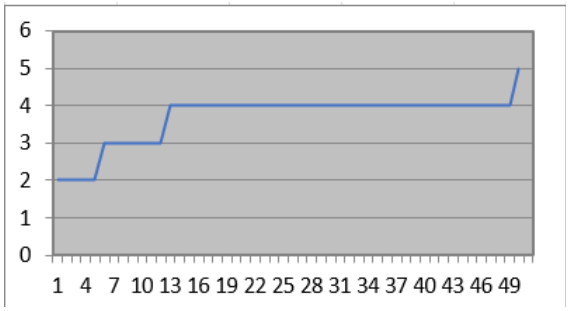
Či už program riešenie nájde alebo nie, v každom prípade sa postupne blíži k hodnote 5.

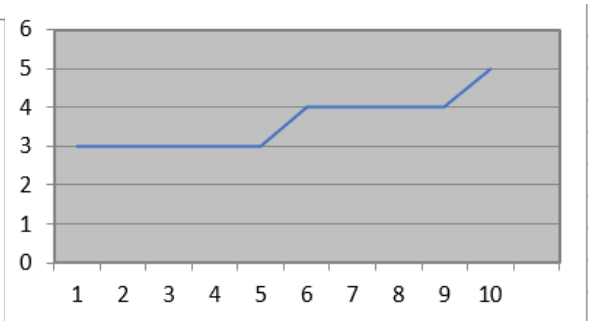
5.1 Selekcia typu turnaj

V grafe nižšie môžeme vidieť že program našiel riešnie už v 33. generácií, a stúpal od hodnoty 2 postupne až k hodnote 5.

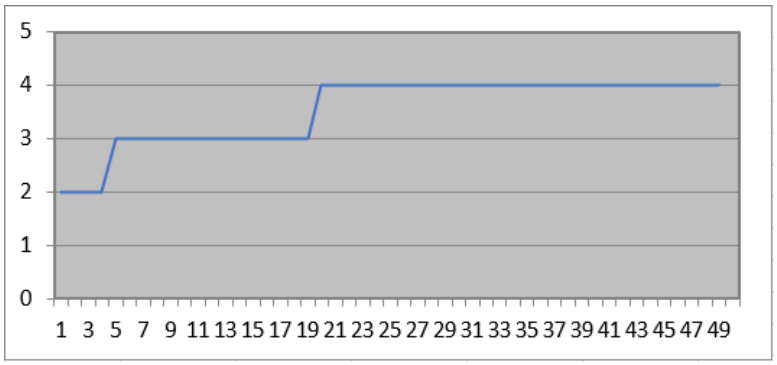


Tak isto veľmi podobné výsledky môžeme vidieť aj v týchto dvoch grafoch nižšie.



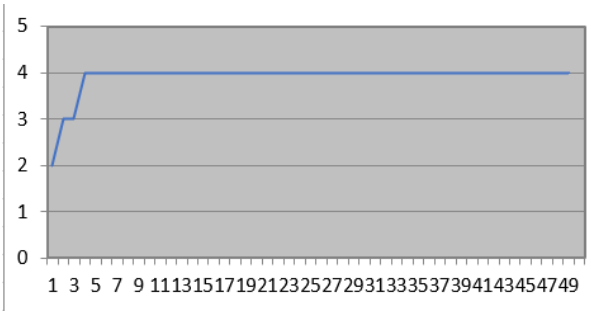


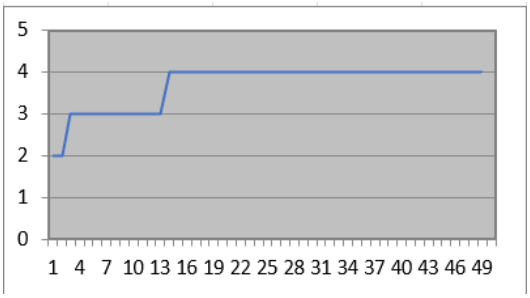
Samozrejme boli aj prípady kedy program riešenie nenašiel, ale tiež môžeme na grafe vidieť že hodnoty sa nám postupne blížia k hodnote 5.



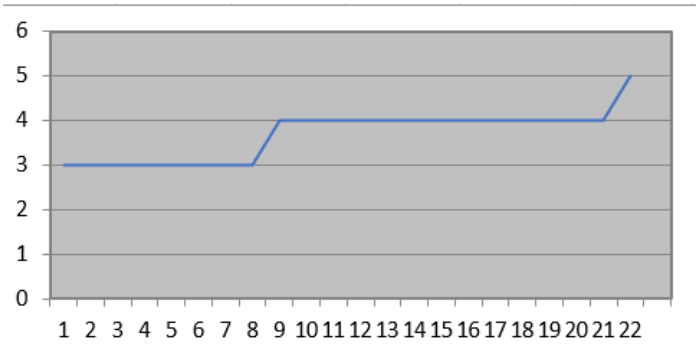
5.2 Kombinácia selekcie typu turnaj a elitárstva (20%)

Ako môžeme vidieť v grafoch nižšie, tak aj pri tomto type tvorby novej generácie sa nám hodnoty postupne blížia k hodnote 5.





Tak isto sa objavili aj prípady kedy program našiel riešenie, tým že jedinec našiel všetky poklady.



5.3 Porovnanie testovania

Z môjho pohľadu ak si porovnáme grafy jednotlivých testovaní, tak môžeme vidieť, že s použitím elitárstva krivka vystúpa o niečo rýchlejšie k hodnote 4, ako pri prípadoch kde sme elitárstvo nepoužili. Na druhú stranu, pri viacerých testovaniach som zistil že prípady bez použitia elitárstva našli riešenie častejšie ako prípady s použitím elitárstva.

Podľa môjho názoru to môže byť spôsobené tým, že pri použití elitárstva sme stratili určitú časť rozmanitosti pri tvorbe novej generácie, ktoré nám selekcia typu turnaj zapezpečuje.

6 Záver

Výsledky dosiahnuté evolučným programovaním, sú do istej miery ťažko hodnotiteľné pretože všetko závisí od náhodne vygenerovaných počiatočných génov, a tak isto aj od náhodne vybratých rodičov zo selekcie typu turnaj.

Aj napriek tomu môžeme vidieť že výsledky v testovaní sú pomerne úspešné a v každom prípade sa blížia k hodnote 5. Je to možno spôsobené aj tým, že selekcia typu turnaj síce nezaručuje vybratie vždy najlepších jedincov, ale zaisťuje hlavne rozmanitosť a selekčný tlak. Naopak pri kombinácií elitárstva a selekcie typu turnaj, strácame určitú časť rozmanitosti, čo má za príčinu to, že výsledky testovania sa nám mierne líšia.

Tak isto je to ovplyvnené aj tým, aký úspešný bude najlepší jedinec hneď v prvej generácií, pretože ak hneď v prvej generácií nájde nejaký jedinec 4 poklady, tak už sa nám krivka v grafe ťažko alebo veľmi pomaly bude blížiť k hodnote 5.

Program by sa dal samozrejme ešte doplniť o ďalšie typy selekcie, ako napríklad ruleta alebo výber podľa poradia, a následne porovnať dosiahnuté výsledky viacerých selekcií.