**RĪGAS TEHNISKĀ UNIVERSITĀTE**

**Datorzinātnes un informācijas tehnoloģijas fakultāte**

**1.Praktiskais darbs studiju kursā**

**“Mākslīgā intelekta pamati”**

**Saite: https://github.com/MarisZeibe/AI-pr-d-1-27.git**

**Atskaite**

**27. komanda:**

**Artūrs Melmanis 221RDB183**

**Katrīna Apine 221RDB169**

**Aleksandrs Kapanskis 221RDB182**

**Māris Žeibe 221RDB138**

**2023./ 2024. m.g.**

Studentu komandai ir jāiesniedz atskaite par izstrādāto programmatūru. Atskaitei ir jāsatur:

* programmatūras darbības demonstrācijas piemērs vai lietotāja ceļvedis ar paskaidrojumiem;
* apraksts par izmantotajām datu struktūrām spēles koka glabāšanai ar detalizētiem komentāriem, kas tieši tiek glabāts konkrētajā datu struktūrā;
* heiristiskā novērtējuma funkcijas apraksts un pamatojums izvēlētajai funkcijai;

|  |  |
| --- | --- |
| def evaluate\_state(state):      k = 1 if ((state.bank + state.points) % 2 == 0) else -1      x1 = min(state.bank, 1)      x2 = (1 if state.number >= 3000 else 0) \* (3000 \* 5 - state.number) / (3000 \* 5 - 20)      x3 = (0 if state.number >= 3000 else 1) \* state.number / (3000 \* 5 - 20)      return k \* (100\*x1 + 10\*x2 + 1\*x3) | k – Vai stāvoklis ir labvēlīgs pirmajam spēlētājam (maksimizētājam) vai otrajam spēlētājam (minimizētājam)  x1 – Vai spēles banka ir lielāka par 0  x2 – Ja ir spēles beigu stāvoklis, mazākam skaitlim ir lielāka vērtība  x3 – Ja nav spēles beigu stāvoklis, lielākam skaitlim ir lielāka vērtība  Jo tuvāk esošais skaitlis ir 3000, jo labāk spēlētājam. Tāpēc funkcijai šis faktors ir jāņem vērā. Funkcijai arī jāņem vērā spēlētāja pašreizējais punktu un bankas skaits, jo tie ietekmē viņa pozīciju spēlē.  Izvēlētā novērtēšanas funkcija ļauj algoritmam pieņemt lēmumus, pamatojoties uz pašreizējo spēles stāvokli un spēlētāju mērķiem. |

* realizēto pamatalgoritmu (spēles koka ģenerēšana, heiristisko vērtējumu piešķiršana virsotnēm, spēles algoritma pielietojums, uzvaru nesošo ceļu atrašana) kods ar studentu sniegtajiem skaidrojumiem. Kods atskaitei ir jāpievieno tikai teksta veidā. Nav atļauts to pievienot attēlu veidā;

|  |  |
| --- | --- |
| def generate\_tree(index=0):      global gameTree      if gameTree[index].number > 3000: # 1          return      for n in range(3, 6):          new\_state = State( # 2              gameTree[index].number \* n,              gameTree[index].points + (1 if (gameTree[index].number\*n) % 2 == 0 else -1),              gameTree[index].bank + (1 if (gameTree[index].number\*n) % 5 == 0 else 0),              gameTree[index].level + 1          )          if new\_state.number >= 3000:              new\_state.points += bank \* (-1 if new\_state.points % 2 == 0 else 1)          if new\_state not in gameTree: # 3              gameTree.append(new\_state)              gameTree[index].children.append(len(gameTree) - 1)              generate\_tree(len(gameTree) - 1)          else:              gameTree[index].children.append(gameTree.index(new\_state)) | 1. Sākumā mēs pārbaudām, vai pašreizējais mezgls pārsniedz 3000. Ja jā, tas nozīmē, ka spēle ir beigusies, un mēs vienkārši atgriežamies no funkcijas.  2. Katram reizinātāja vērtībai mēs izveidojam jaunu stāvokli (new\_state), reizinot pašreizējā mezgla skaitli ar reizinātāja vērtību. Tāpat mēs aprēķinām jaunas punktu un banku bonusu vērtības atkarībā no spēles nosacījumiem.  3. Mēs pārbaudām, vai šāds stāvoklis jau eksistē spēles koka struktūrā. Ja nav, tad mēs pievienojam šo jauno stāvokli koka struktūrai (gameTree) un pievienojam tās saiti kā pašreizējā mezgla bērnu sarakstā. Pēc tam izsaucam funkciju “generate\_tree” jaunizveidotajam mezglam, lai turpinātu jauno bērnu ģenerēšanu. Ja jaunizveidotais stāvoklis jau eksistē koka struktūrā, tad vienkārši pievienojam saiti uz to pašreizējā mezgla bērnu sarakstā. |
| print(f'value: {evaluate\_state(gameTree[index]):<9.4f}', end=' ') | (heiristisko vērtējumu piešķiršana virsotnēm)  Šī rindiņa koda ļauj izdrukāt izvērtējumu katram spēles stāvoklim, lai varētu vieglāk sekot līdzi spēles algoritma darbībai un analizēt katru virsotni atbilstoši tās novērtētajai vērtībai. |
| Funkcija:  def minimax\_search(state, player):      if state.number >= 3000:          return evaluate\_state(state)      if player == Player.Lietotajs:          best\_value = float('-inf')          for child\_index in state.children:              child\_value = minimax\_search(gameTree[child\_index], Player.Dators)              best\_value = max(best\_value, child\_value)          return best\_value      else:          best\_value = float('inf')          for child\_index in state.children:              child\_value = minimax\_search(gameTree[child\_index], Player.Lietotajs)              best\_value = min(best\_value, child\_value)          return best\_value  Pielietojums:  if algorithm == Algorithm.MINIMAX:                  available\_moves = [3, 4, 5]                  best\_value = float('-inf')                  best\_move = None                  for move in available\_moves:                      new\_number = number \* move                      new\_points = points + (1 if new\_number % 2 == 0 else -1)                      new\_bank = bank + (1 if new\_number % 5 == 0 else 0)                      new\_level = gameTree[0].level + 1                      child\_state = State(new\_number, new\_points, new\_bank, new\_level)                      value = minimax\_search(child\_state, player)                      if value > best\_value:                          best\_value = value                          best\_move = move | spēles Minimaksa algoritma pielietojums |
| Funkcija:  def alpha\_beta\_search(state, alpha, beta, player):      if state.number >= 3000:          return evaluate\_state(state)      if player == Player.Lietotajs:          value = float('-inf')          for child\_index in state.children:              child\_value = alpha\_beta\_search(gameTree[child\_index], alpha, beta, Player.Dators)              value = max(value, child\_value)              alpha = max(alpha, value)              if alpha >= beta:                  break          return value      else:          value = float('inf')          for child\_index in state.children:              child\_value = alpha\_beta\_search(gameTree[child\_index], alpha, beta, Player.Lietotajs)              value = min(value, child\_value)              beta = min(beta, value)              if alpha >= beta:                  break          return value  Pielietojums:  elif algorithm == Algorithm.ALPHA\_BETA:                  available\_moves = [3, 4, 5]                  best\_value = float('-inf')                  best\_move = None                  for move in available\_moves:                      new\_number = number \* move                      new\_points = points + (1 if new\_number % 2 == 0 else -1)                      new\_bank = bank + (1 if new\_number % 5 == 0 else 0)                      new\_level = gameTree[0].level + 1                      child\_state = State(new\_number, new\_points, new\_bank, new\_level)                      value = alpha\_beta\_search(child\_state, float('-inf'), float('inf'), player)                      if value > best\_value:                          best\_value = value                          best\_move = move                  number \*= best\_move                  print('Datora gājiens. Izvēlēts skaitlis:', best\_move) | spēles Alfa-beta algoritma pielietojums |

* algoritmu salīdzinājums un komandas veiktos secinājumus.

Viss programmatūras kods:

import random

from enum import Enum

class Algorithm(Enum):

    MINIMAX = 1

    ALPHA\_BETA = 2

class Player(Enum):

    Lietotajs = 1

    Dators = 2

def choose\_starting\_player() -> Player:

    while True:

        print("Izvēlieties, kurš sāks spēli:")

        print("Lietotājs vai dators")

        choice = input("Ievadiet izvēlēto spēlētāju (l vai d): ")

        if choice == "l":

            return Player.Lietotajs

        elif choice == "d":

            return Player.Dators

        else:

            print("Nepareiza izvēle. Mēģiniet vēlreiz.")

def choose\_algorithm() -> Algorithm:

    while True:

        print("Izvēlieties algoritmu, kuru izmantos dators:")

        print("Minimaksa algoritms vai Alfa-beta algoritms")

        choice = input("Ievadiet izvēlēto algoritmu (m vai a): ")

        if choice == "m":

            return Algorithm.MINIMAX

        elif choice == "a":

            return Algorithm.ALPHA\_BETA

        else:

            print("Nepareiza izvēle. Mēģiniet vēlreiz.")

def evaluate\_state(state):

    # Vai stāvoklis ir labvēlīgs pirmajam spēlētājam (maksimizētājam) vai otrajam spēlētājam (minimizētājam)

    k = 1 if ((state.bank + state.points) % 2 == 0) else -1

    # Vai spēles banka ir lielāka par 0

    x1 = min(state.bank, 1)

    # Ja ir spēles beigu stāvoklis, mazākam skaitlim ir lielāka vērtība

    x2 = (1 if state.number >= 3000 else 0) \* (3000 \* 5 - state.number) / (3000 \* 5 - 20)

    # Ja nav spēles beigu stāvoklis, lielākam skaitlim ir lielāka vērtība

    x3 = (0 if state.number >= 3000 else 1) \* state.number / (3000 \* 5 - 20)

    return k \* (100\*x1 + 10\*x2 + 1\*x3)

def minimax\_search(state, player):

    if state.number >= 3000:

        return evaluate\_state(state)

    if player == Player.Lietotajs:

        best\_value = float('-inf')

        for child\_index in state.children:

            child\_value = minimax\_search(gameTree[child\_index], Player.Dators)

            best\_value = max(best\_value, child\_value)

        return best\_value

    else:

        best\_value = float('inf')

        for child\_index in state.children:

            child\_value = minimax\_search(gameTree[child\_index], Player.Lietotajs)

            best\_value = min(best\_value, child\_value)

        return best\_value

def alpha\_beta\_search(state, alpha, beta, player):

    if state.number >= 3000:

        return evaluate\_state(state)

    if player == Player.Lietotajs:

        value = float('-inf')

        for child\_index in state.children:

            child\_value = alpha\_beta\_search(gameTree[child\_index], alpha, beta, Player.Dators)

            value = max(value, child\_value)

            alpha = max(alpha, value)

            if alpha >= beta:

                break

        return value

    else:

        value = float('inf')

        for child\_index in state.children:

            child\_value = alpha\_beta\_search(gameTree[child\_index], alpha, beta, Player.Lietotajs)

            value = min(value, child\_value)

            beta = min(beta, value)

            if alpha >= beta:

                break

        return value

class State:

    number: int

    points: int

    bank: int

    level: int

    children: list

    def \_\_init\_\_(self, number, points, bank, level):

        self.number = number

        self.points = points

        self.bank = bank

        self.level = level

        self.children = []

    def \_\_eq\_\_(self, other):

        if other.\_\_class\_\_ is not self.\_\_class\_\_:

            return NotImplemented

        return (self.number, self.points, self.bank, self.level) == (other.number, other.points, other.bank, other.level)

def int\_input(message: str, number\_range: range) -> int:

    print(message, end='')

    while True:

        try:

            input\_number = int(input())

            if input\_number in number\_range:

                return input\_number

            else:

                print('Nepareiz skaitlis, mēģiniet vēlreiz: ', end='')

        except ValueError:

            print('Kļūda, mēģiniet vēlreiz: ', end='')

def generate\_tree(index=0):

    global gameTree

    if gameTree[index].number > 3000:

        return

    for n in range(3, 6):

        new\_state = State(

            gameTree[index].number \* n,

            gameTree[index].points + (1 if (gameTree[index].number\*n) % 2 == 0 else -1),

            gameTree[index].bank + (1 if (gameTree[index].number\*n) % 5 == 0 else 0),

            gameTree[index].level + 1

        )

        if new\_state.number >= 3000:

            new\_state.points += bank \* (-1 if new\_state.points % 2 == 0 else 1)

        if new\_state not in gameTree:

            gameTree.append(new\_state)

            gameTree[index].children.append(len(gameTree) - 1)

            generate\_tree(len(gameTree) - 1)

        else:

            gameTree[index].children.append(gameTree.index(new\_state))

def print\_tree(generate\_values=False, index=0):

    print('\t' \* gameTree[index].level, end='')

    print(f'number: {gameTree[index].number:<5}', end=' ')

    print(f'points: {gameTree[index].points:<2}', end=' ')

    print(f'bank: {gameTree[index].bank}', end=' ')

    if generate\_values:

        print(f'value: {evaluate\_state(gameTree[index]):<9.4f}', end=' ')

        current\_player = player.Lietotajs if gameTree[index].level % 2 == 0 else player.Dators

        print(f'minimax: {minimax\_search(gameTree[index], current\_player):.4f}', end='')

    print()

    for state\_index in gameTree[index].children:

        print\_tree(generate\_values, state\_index)

while True:

    points = 0

    bank = 0

    player = choose\_starting\_player()

    algorithm = choose\_algorithm()

    number = int\_input('Ievadiet skaitli no 20 līdz 30: ', range(20, 31))

    gameTree = [State(number, points, bank, 0)]

    generate\_tree()

    print\_tree(True)

    while number <= 3000:

        print(player, '. spēlētāja gājiens', sep='')

        if player == Player.Lietotajs:

            number \*= int\_input('Ievadiet reizinātāju (3, 4 vai 5): ', range(3, 6))

        else:

            if algorithm == Algorithm.MINIMAX:

                available\_moves = [3, 4, 5]

                best\_value = float('-inf')

                best\_move = None

                for move in available\_moves:

                    new\_number = number \* move

                    new\_points = points + (1 if new\_number % 2 == 0 else -1)

                    new\_bank = bank + (1 if new\_number % 5 == 0 else 0)

                    new\_level = gameTree[0].level + 1

                    child\_state = State(new\_number, new\_points, new\_bank, new\_level)

                    value = minimax\_search(child\_state, player)

                    if value > best\_value:

                        best\_value = value

                        best\_move = move

                number \*= best\_move

                print('Datora gājiens. Izvēlēts skaitlis:', best\_move)

            elif algorithm == Algorithm.ALPHA\_BETA:

                available\_moves = [3, 4, 5]

                best\_value = float('-inf')

                best\_move = None

                for move in available\_moves:

                    new\_number = number \* move

                    new\_points = points + (1 if new\_number % 2 == 0 else -1)

                    new\_bank = bank + (1 if new\_number % 5 == 0 else 0)

                    new\_level = gameTree[0].level + 1

                    child\_state = State(new\_number, new\_points, new\_bank, new\_level)

                    value = alpha\_beta\_search(child\_state, float('-inf'), float('inf'), player)

                    if value > best\_value:

                        best\_value = value

                        best\_move = move

                number \*= best\_move

                print('Datora gājiens. Izvēlēts skaitlis:', best\_move)

        points += 1 if number % 2 == 0 else -1

        bank += 1 if number % 5 == 0 else 0

        # player = (player % 2) + 1

        player = Player.Lietotajs if player == Player.Dators else Player.Dators

        print('skailtis:', number, '| punkti:', points, '| banka:', bank)

    points += bank \* (-1 if points % 2 == 0 else 1)

    print('Spēle beidzās, gala punkti:', points)

    print('Uzvar ', (points % 2) + 1, '. spēlētājs', sep='')

    play\_again = input("Vai vēlaties spēlēt vēlreiz? (j/n): ")

    if play\_again.lower() != "j":

        break