**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;

END\_NUMBER = 3000; MAX\_MULTIPLIER = 5

|  |  |
| --- | --- |
| def evaluate\_state(self) -> float:          max\_number = END\_NUMBER \* MAX\_MULTIPLIER          k = 1 if (((self.bank if self.number < END\_NUMBER else 0) + self.points) % 2 == 0) else -1          x1 = min(self.bank, 1)          x2 = (1 if self.number >= END\_NUMBER else 0) \* (max\_number - self.number) / (max\_number - 20)          x3 = (0 if self.number >= END\_NUMBER else 1) \* self.number / (max\_number - 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  Kopējā heuristikas vērtība tiek aprēķināta, saskaitot šos faktorus, kur k tiek izmantots, lai noskaidrotu, kurš spēlētājs ir labvēlīgā stāvoklī. Jo augstāka heuristikas vērtība, jo labvēlīgāks ir stāvoklis pašreizējam spēlētājam. Šī vērtība tiek izmantota minimax un alfa-beta algoritmos, lai izvēlētos optimālo gājienu spēlē. |

* 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;

SEARCH\_DEPTH = 2; END\_NUMBER = 3000; MIN\_START\_NUMBER = 20; MAX\_START\_NUMBER = 30; MIN\_MULTIPLIER = 3; MAX\_MULTIPLIER = 5

|  |  |
| --- | --- |
| def generate\_tree(tree: list[State], index=0, depth=SEARCH\_DEPTH) -> list[State]:      tree[index].children = []      if tree[index].number >= END\_NUMBER:          return tree      for multiplier in range(MIN\_MULTIPLIER, MAX\_MULTIPLIER + 1):          new\_state = tree[index].next\_state(multiplier)          if new\_state not in tree:              tree.append(new\_state)              tree[index].children.append(len(tree) - 1)              if depth > 1:                  tree = generate\_tree(tree, len(tree) - 1, depth - 1)          else:              tree[index].children.append(tree.index(new\_state))      return tree | Spēles koka ģenerēšanas funkcija generate\_tree ir atbildīga par izveidot spēles koku, kur katrs mezgls (vai stāvoklis) pārstāv iespējamo spēles stāvokli. Katrs mezgls satur informāciju par pašreizējo skaitli, punktiem, banku, līmeni un sarakstu ar tā potenciālajiem bērniem (vai turpmākajiem stāvokļiem).  Šī funkcija sāk ar sākotnējo stāvokli un pēc tam, izmantojot spēles noteikumus, izveido visus iespējamos nākamā gājiena stāvokļus.  Ja jaunizveidotais stāvoklis vēl nav sastopams koka struktūrā, tas tiek pievienots koka beigās un norādīts kā potenciālais bērns pašreizējam mezglam. Ja stāvoklis jau eksistē koka struktūrā, tad tikai tiek norādīts uz jau esošo mezglu kā potenciālais bērns.  Funkcija atkārto šo procesu, līdz tiek sasniegts noteiktais meklēšanas dziļums (vai, ja kāds no stāvokļiem ir beigu stāvoklis). |
| print(f'value: {tree[index].evaluate\_state():<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. |
| def minimax\_search(tree: list[State], index=0) -> dict[str, float]:      if len(tree[index].children) == 0:          return {'value': tree[index].evaluate\_state()}      if tree[index].level % 2 == 0:          best = {'value': float('-inf')}          for child\_index in tree[index].children:              child = {'value': minimax\_search(tree, child\_index)['value'],                       'index': tree[index].children.index(child\_index)}              if child['value'] > best['value']:                  best = child                  tree[child\_index].value = best['value']      else:          best = {'value': float('inf')}          for child\_index in tree[index].children:              child = {'value': minimax\_search(tree, child\_index)['value'],                       'index': tree[index].children.index(child\_index)}              if child['value'] < best['value']:                  best = child                  tree[child\_index].value = best['value']      tree[index].value = best['value']      return best | Šī funkcija darbojas rekursīvi, meklējot labāko gājienu no pašreizējā stāvokļa. Šeit ir tās galvenās darbības:  1. Ja pašreizējam stāvoklim nav children (t.i., tas ir beigu stāvoklis), tiek izvērsta bāzes gadījuma apstrāde, un tiek atgriezta šī virsotnes vērtība, kas iegūta, izmantojot funkciju evaluate\_state.  2. Ja pašreizējais stāvoklis ir maksimizējošais spēlētājs (piemēram, cilvēks), tad tiek sākta meklēšana pēc maksimālās vērtības starp children stāvokļiem. Šajā gadījumā algoritms meklē children stāvokli ar vislielāko vērtību, jo tas simbolizē labāko iespējamo gājienu pašreizējam spēlētājam.  3. Ja pašreizējais stāvoklis ir minimizējošais spēlētājs (piemēram, dators), tad tiek sākta meklēšana pēc minimālās vērtības starp children stāvokļiem. Šajā gadījumā algoritms meklē children stāvokli ar vismazāko vērtību, kas simbolizē labāko iespējamo gājienu pretējam spēlētājam.  4. Pēc tam, kad ir atrasts labākais child, tiek atjaunināta pašreizējā stāvokļa vērtība ar labākā child vērtību, un šī vērtība tiek atgriezta. |
| def alpha\_beta\_search(tree: list[State], index=0, alpha=float('-inf'), beta=float('inf')) -> dict[str, float]:      if len(tree[index].children) == 0:          return {'value': tree[index].evaluate\_state()}      if tree[index].level % 2 == 0:          best = {'value': float('-inf')}          for child\_index in tree[index].children:              child = {'value': alpha\_beta\_search(tree, child\_index, alpha, beta)['value'],                       'index': tree[index].children.index(child\_index)}              if child['value'] > best['value']:                  best = child                  tree[child\_index].value = best['value']              alpha = max(alpha, best['value'])              if alpha >= beta:                  break      else:          best = {'value': float('inf')}          for child\_index in tree[index].children:              child = {'value': alpha\_beta\_search(tree, child\_index, alpha, beta)['value'],                       'index': tree[index].children.index(child\_index)}              if child['value'] < best['value']:                  best = child                  tree[child\_index].value = best['value']              beta = min(beta, best['value'])              if alpha >= beta:                  break      tree[index].value = best['value']      return best | Funkcija alpha\_beta\_search ir alfa-beta algoritma realizācija, kas tiek izmantota, lai uzlabotu minimaksa algoritmu, samazinot vajadzīgo aprēķinu apjomu, pārskatot nevajadzīgus gājienus.  Šī funkcija strādā līdzīgi kā minimaksa algoritms, meklējot labāko gājienu no pašreizējā stāvokļa. Tomēr, atšķirībā no minimaksa, alfa-beta algoritms veic apgrieztā kārta un izlemj, vai apskatīt citus bērnus vai pārtraukt meklēšanu, pamatojoties uz alfa un beta vērtībām.  Galvēnas atšķirības: 1. Ja pašreizējais stāvoklis ir maksimizējošais spēlētājs, tad ar maksimālo vērtību tiek vienlaikus saglabāta alfa vērtība. Ja kāds child vērtībā pārsniedz alfa vērtību, alfa tiek atjaunināta.  2. Ja pašreizējais stāvoklis ir minimizējošais spēlētājs, ar minimālo vērtību tiek saglabāta beta vērtība. Ja kāds bērns vērtībā ir mazāks par beta vērtību, beta tiek atjaunināta.  3. Ja alfa vērtība kļūst lielāka vai vienāda ar beta vērtību, tas nozīmē, ka mēs esam atraduši gājienu, kas nav labāks par citiem jau apskatītajiem gājieniem, un tāpēc mums nav jēgas turpināt meklēt. Tāpēc algoritms tiek pārtraukts. |
|  | uzvaru nesošo ceļu atrašana |

* algoritmu salīdzinājums un komandas veiktos secinājumus.

Algoritmu salīdzinājums:

Secinājumi:

Viss programmatūras kods:

from enum import Enum

GUI\_MODE = False

DEBUG = True

SEARCH\_DEPTH = 2

END\_NUMBER = 3000

MIN\_START\_NUMBER = 20

MAX\_START\_NUMBER = 30

MIN\_MULTIPLIER = 3

MAX\_MULTIPLIER = 5

class Algorithm(Enum):

    MINIMAX = 1

    ALPHA\_BETA = 2

class Player(Enum):

    USER = 1

    COMPUTER = 2

class State:

    number: int

    points: int

    bank: int

    level: int

    children: list

    value: float | None

    def \_\_init\_\_(self, number, points, bank, level) -> None:

        self.number = number

        self.points = points

        self.bank = bank

        self.level = level

        self.children = []

        self.value = None

    def \_\_eq\_\_(self, other) -> bool:

        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 next\_state(self, multiplier: int) -> 'State':

        new\_state = State(

            self.number \* multiplier,

            self.points + (1 if (self.number \* multiplier) % 2 == 0 else -1),

            self.bank + (1 if (self.number \* multiplier) % 5 == 0 else 0),

            self.level + 1

        )

        if new\_state.number >= END\_NUMBER:

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

        return new\_state

    def evaluate\_state(self) -> float:

        max\_number = END\_NUMBER \* MAX\_MULTIPLIER

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

        k = 1 if (((self.bank if self.number < END\_NUMBER else 0) + self.points) % 2 == 0) else -1

        # Vai spēles banka ir lielāka par 0

        x1 = min(self.bank, 1)

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

        x2 = (1 if self.number >= END\_NUMBER else 0) \* (max\_number - self.number) / (max\_number - 20)

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

        x3 = (0 if self.number >= END\_NUMBER else 1) \* self.number / (max\_number - 20)

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

def generate\_tree(tree: list[State], index=0, depth=SEARCH\_DEPTH) -> list[State]:

    tree[index].children = []

    if tree[index].number >= END\_NUMBER:

        return tree

    for multiplier in range(MIN\_MULTIPLIER, MAX\_MULTIPLIER + 1):

        new\_state = tree[index].next\_state(multiplier)

        if new\_state not in tree:

            tree.append(new\_state)

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

            if depth > 1:

                tree = generate\_tree(tree, len(tree) - 1, depth - 1)

        else:

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

    return tree

def print\_tree(tree: list[State], algorithm: Algorithm | None = None, index=0) -> None:

    if algorithm == Algorithm.MINIMAX:

        minimax\_search(tree)

    elif algorithm == Algorithm.ALPHA\_BETA:

        alpha\_beta\_search(tree)

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

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

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

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

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

    if tree[index].value is not None:

        print(f'algorithm: {tree[index].value:.4f}', end='')

    print()

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

        print\_tree(tree, None, state\_index)

def minimax\_search(tree: list[State], index=0) -> dict[str, float]:

    if len(tree[index].children) == 0:

        return {'value': tree[index].evaluate\_state()}

    if tree[index].level % 2 == 0:

        best = {'value': float('-inf')}

        for child\_index in tree[index].children:

            child = {'value': minimax\_search(tree, child\_index)['value'],

                     'index': tree[index].children.index(child\_index)}

            if child['value'] > best['value']:

                best = child

                tree[child\_index].value = best['value']

    else:

        best = {'value': float('inf')}

        for child\_index in tree[index].children:

            child = {'value': minimax\_search(tree, child\_index)['value'],

                     'index': tree[index].children.index(child\_index)}

            if child['value'] < best['value']:

                best = child

                tree[child\_index].value = best['value']

    tree[index].value = best['value']

    return best

def alpha\_beta\_search(tree: list[State], index=0, alpha=float('-inf'), beta=float('inf')) -> dict[str, float]:

    if len(tree[index].children) == 0:

        return {'value': tree[index].evaluate\_state()}

    if tree[index].level % 2 == 0:

        best = {'value': float('-inf')}

        for child\_index in tree[index].children:

            child = {'value': alpha\_beta\_search(tree, child\_index, alpha, beta)['value'],

                     'index': tree[index].children.index(child\_index)}

            if child['value'] > best['value']:

                best = child

                tree[child\_index].value = best['value']

            alpha = max(alpha, best['value'])

            if alpha >= beta:

                break

    else:

        best = {'value': float('inf')}

        for child\_index in tree[index].children:

            child = {'value': alpha\_beta\_search(tree, child\_index, alpha, beta)['value'],

                     'index': tree[index].children.index(child\_index)}

            if child['value'] < best['value']:

                best = child

                tree[child\_index].value = best['value']

            beta = min(beta, best['value'])

            if alpha >= beta:

                break

    tree[index].value = best['value']

    return best

class Game:

    state: State

    algorithm: Algorithm

    players: list[Player]

    def \_\_init\_\_(self, starting\_player: Player, algorithm: Algorithm, starting\_number: int) -> None:

        self.state = State(starting\_number, 0, 0, 0)

        self.algorithm = algorithm

        if starting\_player == Player.USER:

            self.players = [Player.USER, Player.COMPUTER]

        else:

            self.players = [Player.COMPUTER, Player.USER]

        if DEBUG:

            print\_tree(generate\_tree([self.state], 0, 10 \*\* 5), algorithm)

    def user\_move(self, multiplier: int) -> None:

        self.state = self.state.next\_state(multiplier)

    def computer\_move(self) -> int:

        tree = generate\_tree([self.state])

        if self.algorithm == Algorithm.MINIMAX:

            multiplier = int(minimax\_search(tree)['index']) + MIN\_MULTIPLIER

        else:

            multiplier = int(alpha\_beta\_search(tree)['index']) + MIN\_MULTIPLIER

        self.user\_move(multiplier)

        if DEBUG:

            print\_tree(tree, self.algorithm)

        return multiplier

    def get\_current\_player(self) -> Player:

        return self.players[self.state.level % 2]

    def is\_game\_finished(self) -> bool:

        return self.state.number >= END\_NUMBER

    def get\_winner(self) -> Player:

        return self.players[self.state.points % 2]

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 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.USER

        elif choice == "d":

            return Player.COMPUTER

        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 choose\_starting\_number() -> int:

    return int\_input(f'Ievadiet skaitli no {MIN\_START\_NUMBER} līdz {MAX\_START\_NUMBER}: ',

                     range(MIN\_START\_NUMBER, MAX\_START\_NUMBER + 1))

if GUI\_MODE:

    pass

else:

    while True:

        game = Game(choose\_starting\_player(), choose\_algorithm(), choose\_starting\_number())

        while not game.is\_game\_finished():

            if game.get\_current\_player() == Player.USER:

                print('Lietotāja gājiens:')

                input\_multiplier = int\_input(f'Ievadiet reizinātāju no {MIN\_MULTIPLIER} līdz {MAX\_MULTIPLIER}: ',

                                             range(MIN\_MULTIPLIER, MAX\_MULTIPLIER + 1))

                game.user\_move(input\_multiplier)

            else:

                print('Datora gājiens:')

                print('Dators izvēlējās skaitli:', game.computer\_move())

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

        print('Spēle beidzās, gala punkti:', game.state.points)

        print('Uzvarēja', 'lietotājs' if game.get\_winner() == Player.USER else 'dators')

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

        if play\_again.lower() != "j":

            break