RĪGAS TEHNISKĀ UNIVERSITĀTE

Datorzinātnes un informācijas tehnoloģijas fakultāte

ATSKAITE

1. praktiskais darbs studiju kursā

“Mākslīgā intelekta pamati”

Saite uz koplietoto repozitoriju

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

27. komanda:

Artūrs Melmanis 221RDB183

Katrīna Apine 221RDB169

Aleksandrs Kapanskis 221RDB182

Māris Žeibe 221RDB138

2023./ 2024. m.g.

SATURS

SPĒLES APRAKSTS

Spēles sākumā ir dots cilvēka-spēlētāja izvēlētais skaitlis. Kopīgs punktu skaits ir vienāds ar 0 (punkti netiek skaitīti katram spēlētājam atsevišķi). Turklāt spēlē tiek izmantota spēles banka, kura sākotnēji ir vienāda ar 0. Spēlētāji veic gājienus pēc kārtas, reizinot pašreizējā brīdī esošu skaitli ar 3, 4 vai 5. Ja reizināšanas rezultātā tiek iegūts pāra skaitlis, tad kopīgajam punktu skaitam tiek pieskaitīts 1 punkts, bet ja nepāra skaitlis – tad 1 punkts tiek atņemts. Savukārt, ja tiek iegūts skaitlis, kas beidzas ar 0 vai 5, tad bankai tiek pieskaitīts 1 punkts. Spēle beidzas, kad ir iegūts skaitlis, kas ir lielāks par vai vienāds ar 3000. Ja kopīgais punktu skaits ir pāra skaitlis, tad no tā atņem bankā uzkrātos punktus. Ja tas ir nepāra skaitlis, tad tam pieskaita bankā uzkrātos punktus. Ja kopīgā punktu skaita gala vērtība ir pāra skaitlis, uzvar spēlētājs, kas uzsāka spēli. Ja nepāra skaitlis, tad otrais spēlētājs.

PAPILDUS PRASĪBAS PROGRAMMATŪRAI

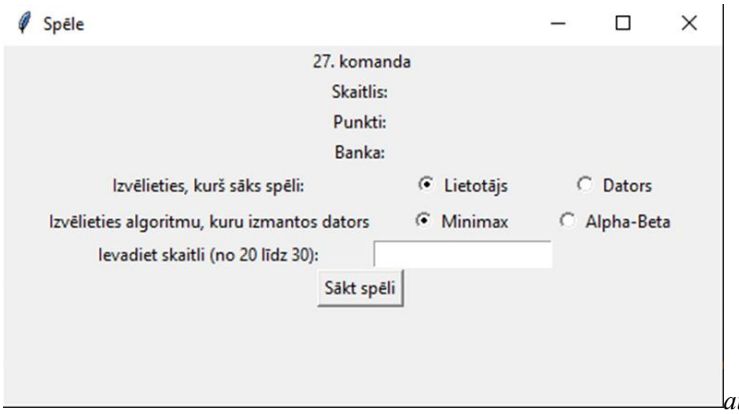
Spēles sākumā cilvēks-spēlētājs izvēlas, ar kuru skaitli diapazonā no 20 līdz 30 sākt spēli.

PROGRAMMATŪRAS DARBĪBAS DEMONSTRĀCIJAS PIEMĒRS VAI LIETOTĀJA CEĻVEDIS AR PASKAIDROJUMIEM

Šī programma ļauj lietotājam spēlēt spēli sākot ar datora gājienu vai lietotāja gājienu, kur tiek izmantots Minimaksa vai Alfa-Beta algoritms, lai pieņemtu lēmumu par gājienu veikšanu. Spēles mērķis ir sasniegt noteiktu skaitli (END_NUMBER) ar sākuma skaitli, punkti tiek piešķirti katram gājienam atkarībā no tā, vai rezultējošais skaitlis ir pāra vai nepāra, un vai tas dalās ar 5. Spēle turpinās, līdz tiek sasniegts vai pārsniegts noteiktais skaitlis.

Kad palaižam programmu bez grafiskā interfeisa (saskarnes), tad kods ļauj izvēlēties, vai spēli sāksim ar l (lietotāju) vai d (datoru). Apstiprinot izvēli tiek izvadīts nākamais paziņojums, kur ir jāizvēlas kādu algoritmu izmantos dators - m (Minimaksa) vai a (Alfa-beta) algoritmu. Kad tas tika izvēlēts, sekojošais paziņojums būs par sākuma skaitļa izvēli diapazonā no 20 līdz 30, kas būs spēles sākuma skaitlis. Kad tiek apstiprināta šī izvēle, tad sākas spēle, kur tiek skaitīti punkti, bankas skaitlis. Kad gājiens tiek veikts, tiek izvadīts paziņojums, kur jāizvēlas ar kādu skaitli tiks reizināts sākotnējais skaitlis (diapazonā no 3 līdz 5). Spēle turpinās līdz tiek iegūts skaitlis, kas ir lielāks par vai vienāds ar 3000. Pašās beigās tiek izvadīts gala skaitlis, iegūtais punktu skaits un bankas rezultāts, kā arī, tiek piedāvāts spēlēt spēli vēlreiz.

Grafiskās saskarnes realizācijas piemērs ar skaidrojumiem:

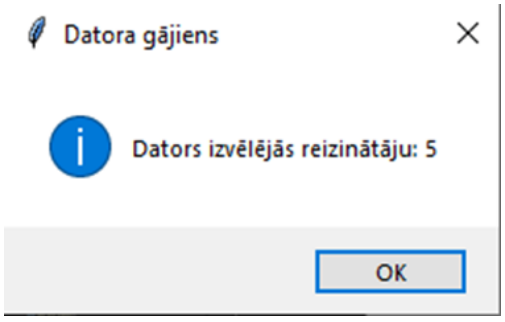


attēls1

Ja apskata grafisko saskarni (att. 1), tad varam redzēt, ka palaižot programmu tiek izvadīts logs, kurā redzēsim skaitli, punktus un bankas skaitli. Tāpat arī esam aicināti ar divām iespējām izvēlēties starp dalībnieku kurš sāks spēli (klikšķinot uz Lietotājs, ja vēlamies, lai spēli uzsāk lietotājs, vai Dators, ja vēlamies, lai spēli sāk dators. Nākamā izvēle, kuru esam aicināti veikt ir izvēlēties starp algoritmiem, kurus var izmantot dators – Minimax vai Alfa-Beta algoritms, un, visbeidzot, skaitli, ar kuru vēlamies uzsākt spēlēt, kas ir jāieraksta iekšā lodziņa un tad varam spiest pogu “Sākt spēli”.

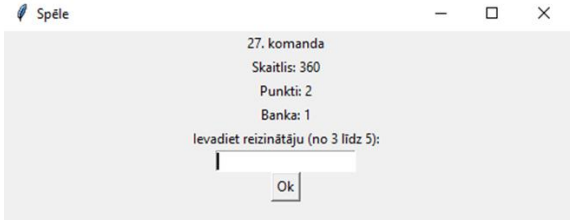
attēls2

Kad tas ir izdarīts mūs sagaida nākamais logs (att. 2), kur mūs aicina ievadīt reizinātāju starp skaitļiem 3, 4 vai 5, kas ir jāieraksta iekšā lodziņā (ja tika izvēlēts, ka pirmais spēli uzsāk Lietotājs).



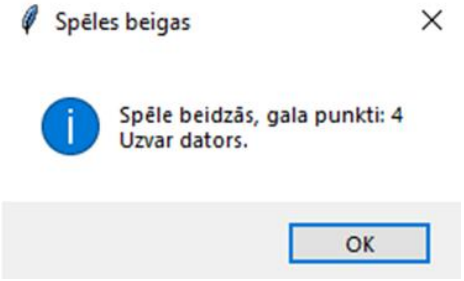
attēls3

Pēc "Ok" pogas nospiešanas, parādās jauns logs (att. 3) ar informāciju par datora veikto gājienu spēles biedra informēšanai.



attēls4

Augšējā loga pusē (att. 4) varam sekot līdzi rezultātam pēc katra spēlētāja gājiena.



attēls5

Visbeidzot, mūs sagaida "Spēles beigas" logs (att. 5), kur tiek paziņots par spēles beigām, kā arī paziņo spēles uzvarētāju un tā iegūtos punktus.

APRAKSTS PAR IZMANTOTAJĀM DATU STRUKTŪRĀM SPĒLES KOKA GLABĀŠANAI AR DETALIZĒTIEM KOMENTĀRIEM, KAS TIEŠI TIEK GLABĀTS KONKRĒTAJĀ DATU STRUKTŪRĀ

State (Stāvoklis):
Šī klase tiek izmantota, lai pārstāvētu spēles stāvokli koka struktūrā.

Izmantotie mainīgie:
number: pašreizējais skaitlis spēlē.
points: pašreizējie punkti spēlē.
bank: pašreizējā banka spēlē.
level: kāda ir pašreizējā stāvokļa līmenis koka struktūrā.
children: saraksts ar stāvokļu indeksiem, kas tiek sasniegti no šī stāvokļa.
value: vērtība, kas tiek izvērtēta algoritma darbības laikā, tiek saglabāta tikai izvades testa nolūkos.

Šī klase ir atbildīga par spēles stāvokļa pārstāvēšanu koka struktūrā, kur spēles koks veidojas sarakstā glabājot šīs klases objektus. Katrs objekts pārstāv vienu stāvokli spēles laikā ar tādiem atribūtiem kā *pašreizējais skaitlis, punktu skaits, bankas vērtējums un līmenis*.

Šīs klases metodes ļauj izveidot nākamo stāvokli, novērtēt pašreizējo stāvokli un pārvērst to par tekstuālu reprezentāciju.

Game (Spēle):
Šī klase pārvalda spēles gaitu, saņemot un apstrādājot gājienus un noteicot uzvarētāju.

Izmantotie mainīgie:
state: pašreizējais spēles stāvoklis.
algorithm: izvēlētais algoritms datora gājieniem.
players: spēlētāji (lietotājs un dators) un to secība.

Šī klase sākot spēli, apstrādā gājienus un nosaka uzvarētāju un tad saņem informāciju par spēles parametriem un izmanto algoritmus, lai veiktu gājienus datora pusei. Klase nodrošina interfeisu starp spēles loģiku un lietotāja saskarni.

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). Tas tiek aprēķināts nosakot, vai bankas un punktu skaits ir pāra skaitlis. Ja stāvoklis ir spēles beigu stāvoklis, banka netiek pieskaitīta, jo tas jau ir izdarīts.
x1 – vai spēles banka ir lielāka par 0. Ja banka ir lielāka par 1, tas nozīmē, ka katrā gājienā gan banka, gan punkti mainīsies par 1, kā rezultātā bankas un punktu summa vairs nemainīsies no pāra uz nepāra skaitli un otrādi. Pēc tā var noteikt gala rezultātu, tāpēc šis ir svarīgākais faktors.
x2 – ja ir spēles beigu stāvoklis, mazākam skaitlim ir lielāka vērtība. Tas ir tāpēc, lai uzvara tiktu sasniegta pa īsāku ceļu
x3 – ja nav spēles beigu stāvoklis, lielākam skaitlim ir lielāka vērtība. Šis faktors arī ļauj sasniegt īsāku uzvaras ceļu.

Kopējā heiristikās funkcijas vērtība tiek aprēķināta, saskaitot šos faktorus, kur k tiek izmantots, lai noskaidrotu, kurš spēlētājs ir labvēlīgā stāvoklī. Jo augstāka heiristikās funkcijas vērtība, jo labvēlīgāks ir stāvoklis pašreizējam spēlētājam. Šī vērtība tiek izmantota Minimax un Alfa-beta algoritmos, lai izvēlētos optimālo gājienu spēlē.
Šī funkcija tika izvēlēta, lai novērtētu pašreizējo spēles stāvokli, izvēloties labākos gājienus, pārbaudot un sekojot līdzi punktiem un gaitai, aprēķinot visu ātri un precīzi.

REALIZĒTO PAMATA ALGORITMU KODS AR STUDENTU SNIEGTAJIEM SKAIDROJUMIEM

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 spēles koka izveidošanu, 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 potenciālo bērnu. Funkcija atkārto šo procesu, līdz tiek sasniegts noteiktais meklēšanas dziļums (vai, ja kāds no stāvokļiem ir beigu stāvoklis). Šī funkcija tika izvēlēta, jo strādā efektīgi un optimizēti, atgriež nepieciešamos datus un sniedz iespēju ierobežot dziļumu izmantojot depth (dziļums) neizmantojot vairāk par pieejamajiem resursiem
`` print(f'value: {tree[index].evaluate_state():<9.4f}', end=' ') ```	**Heiristisko vērtējumu piešķiršana virsotnēm.** Šī rindiņa sniedz iespēju izdrukāt izvērtējumu katram spēles stāvoklim, lai varētu vieglāk sekot līdzi spēles algoritma darbībai un analizēt katru virsotni atbilstoši tās novērtētajai vērtībai. Šī rindiņa ir informatīva. Šī funkcija tika izvēlēta, jo izdara savu nepieciešamo darbu – izvada skaitli pēc nepieciešamā formatējuma, lai nodrošinātu vieglāku lasāmību un

	vienveidīgāku izvadi. Šajā vietā novērtējuma funkcija tiek izmantota tikai informatīviem nolūkiem
`` 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 ```	**Funkcija minimax_search ir Minimax algoritma realizācija.** Šī funkcija darbojas rekursīvi, meklējot labāko gājienu no pašreizējā stāvokļa. Šeit ir tās galvenās darbības: 1. Ja pašreizējam stāvoklim nav pēcteču (children) (t.i., beigu stāvokļa), 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 pēctecis (child), tiek atjaunināta pašreizējā stāvokļa vērtība ar labākā child vērtību, un šī vērtība tiek atgriezta. Šī funkcija tika izvēlēta, lai varētu strādāt ar Minimaksa algoritmu, kas prasa lēmumu pieņemšanu optimālā, gudrā veidā, kā arī tā bija prasība veicot projektu.
`` def alpha_beta_search(tree: list[State], index=0, alpha=float('-inf'), beta=float('inf')) -> dict[str, float]: ```	**Funkcija alpha_beta_search ir Alfa-beta algoritma realizācija**, kas tiek izmantota, lai uzlabotu Minimaksa algoritmu, samazinot vajadzīgo aprēķinu apjomu, nepārskatot liekus gājienus.

```
    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
```

Šī funkcija strādā līdzīgi kā Minimaksa algoritms, meklējot labāko gājienu no pašreizējā stāvokļa, tomēr, atšķirībā no Minimaksa, Alfa-beta algoritms veic savu darbību apgrieztā kārtībā un izlemj, vai apskatīt citus pēctečus vai pārtraukt meklēšanu, pamatojoties uz alfa un beta vērtībām.

Galvenās 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 *child* 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.

Šī funkcija tika izvēlēta, lai varētu strādāt ar Alfa-beta algoritmu kā tas tiek noteikts projekta nosacījumos, tas ir ātrs un nepēta liekus gājienus optimizējot mūsu spēli.

```python
def generate_tree(tree: list[State], index=0,
depth=SEARCH_DEPTH) -> list[State]: …

def minimax_search(tree: list[State], index=0) ->
dict[str, float]: …

def alpha_beta_search(tree: list[State], index=0,
alpha=float('-inf'), beta=float('inf')) -> dict[str,
float]: …
```

Uzvaru nesošo ceļu atrašanas funkcija.

Mūsu komandas spēles kodā uzvaru nesošais ceļš tiek noteikts funkcijās **genereate_tree, minimax_search** un **alpha_beta_search.**

generate_tree funkcija veido visus iespējamos gājienus, izveidojot koka struktūru, kur tiek ņemts vērā pašreizējais stāvoklis, iespējamie nākamie stāvokļi, noteiktais dziļums un beigu nosacījums (kur funkcija pārtrauc jaunu stāvokļu veidošanu.

minimax_search funkcija strādā ar izveidoto koku un izvēlas vislabāko gājienu katram spēles stāvoklim skatoties uz visiem iespējamajiem, izvēloties maksimālo vai minimālo vērtību atkarībā no gājiena veicēja.

alpha_beta_search funkcija veic Minimax algoritma optimizāciju efektīvāk pārbaudot iespējamos gājienus izmantojot alfa un beta vērtības likvidējot nevajadzīgus gājienus, lai programma rēķinātu ātrāk un produktīvāk.

Šādā veidā uzvaru nesošais ceļš tiek noteikts, izmantojot šīs funkcijas sistemātiskai izpētei un novērtējumam konkrētajā brīdī un nākotnē un rezultātā tiek izvēlēts ceļš, kas palielina spēlētāja izredzes uzvarēt.

ALGORITMU SALĪDZINĀJUMS UN KOMANDAS VEIKTIE SECINĀJUMI

Algoritmu salīdzinājums:

Tabulā apkopoti dati pēc principa: Gājiens- ja cilvēks, tad skaitlis- ja dators, tad skaitlis, virsotņu skaits, laiks (ms)

№	Algoritms	Pirmais	Skaitlis	Gājiens(1)	Gājiens(2)	Gājiens(3)	Gājiens(4)	Gājiens(5)	Uzvar
1	Alfa-Beta	Liet	20	3	3;9;0.105	3	5;8;0.06	3	Liet
2	Alfa-Beta	Liet	21	4	5;10;0.093	3	3;1;0.033	-	Dat
3	Alfa-Beta	Liet	22	5	3;9;0.046	3	3;2;0.038	3	Liet
4	Alfa-Beta	Liet	23	3	5;8;0.055	3	3;1;0.026	-	Dat
5	Alfa-Beta	Liet	24	3	5;9;0.054	4	3;1;0.027	-	Dat
6	Alfa-Beta	Dat	25	5;7;0.046	3	3;9;0.085	4	-	Dat
7	Alfa-Beta	Dat	26	5;9;0.058	5	5;8;0.048	-	-	Dat
8	Alfa-Beta	Dat	27	5;8;0.062	4	3;9;0.083	4	-	Dat
9	Alfa-Beta	Dat	28	5;9;0.056	3	3;9;0.049	5	-	Dat
10	Alfa-Beta	Dat	29	5;8;0.073	3	3;3;0.046	3	-	Dat
11	Minimax	Liet	20	3	3;12;0.06	3	5;12;0.048	3	Liet
12	Minimax	Liet	21	4	5;12;0.054	3	3;3;0.022	-	Dat
13	Minimax	Liet	22	5	3;12;0.055	3	3;6;0.033	3	Liet
14	Minimax	Liet	23	3	5;12;0.044	3	3;3;0.026	-	Dat
15	Minimax	Liet	24	3	5;12;0.076	4	3;3;0.033	-	Dat
16	Minimax	Dat	25	5;12;0.041	3	3;12;0.058	4	-	Dat
17	Minimax	Dat	26	5;12;0.031	5	5;9;0.064	-	-	Dat
18	Minimax	Dat	27	5;12;0.042	4	5;12;0.057	4	-	Dat
19	Minimax	Dat	28	5;12;0.068	3	3;12;0.056	5	-	Dat
20	Minimax	Dat	29	5;12;0.025	3	3;12;0.058	3	-	Dat

Algoritmu salīdzinājuma secinājumi:

Ar katru no algoritmiem tika veikti 10 eksperimenti, fiksējot datora un cilvēka uzvaru skaitu, datora apmeklēto virsotņu skaitu, datora vidējo laiku gājiena izpildei.

No iegūtajiem datiem tika sastādīta tabula. No tā izriet šādi secinājumi:

1. 4 lietotāja uzvaras un 16 datora uzvaras.

2. Ja dators startē, tad tā uzvara ir garantēta, tāpat arī lietotājam- ja viņš izvēlas pareizo stratēģiju.

3. Vairumā gadījumu Minimax algoritms patērē mazāk laika gājiena veikšanai

4. Pārsvarā abi algoritmi, kur spēli uzsāk dators, izvēlās vienādus vai ļoti līdzīgus reizinātājus.

5. Alfa beta algoritms apmeklē mazāk virsotņu nekā Minimax algoritms.

Biežā datora uzvara saistīta ar punktu skaitīšanu un uzvaras nosacījumiem šajā spēlē:

"Ja reizināšanas rezultātā tiek iegūts pāra skaitlis, tad kopīgajam punktu skaitam tiek pieskaitīts 1 punkts, bet ja nepāra skaitlis – tad 1 punkts tiek atņemts. Savukārt, ja tiek iegūts skaitlis, kas beidzas ar 0 vai 5, tad bankai tiek pieskaitīts 1 punkts."

"Ja kopīgais punktu skaits ir pāra skaitlis, tad no tā atņem bankā uzkrātos punktus. Ja tas ir nepāra skaitlis, tad tam pieskaita bankā uzkrātos punktus. Ja kopīgā punktu skaita gala vērtība ir pāra skaitlis, uzvar spēlētājs, kas uzsāka spēli. Ja nepāra skaitlis, tad otrais spēlētājs."

Pirmajā gājienā izdevīgi reizināt sākotnējo skaitli ar 5 (20 - 100; 21 -105; 22 - 110; 23 - 115; 24 - 120; 25 - 125; 26 - 130; 27 - 135; 28 - 140; 29 - 145; 30 - 150). Šāds pirmais gājiens dod kopīgajam punktu skaitam 1 vai -1 un 1 bankā. Turklāt jebkuri turpmāki reizinājumi dos skaitli, kas beigsies ar 0 vai 5. Tas novedīs pie datora uzvaras, jo kopīgā punktu skaita gala vērtība vienmēr būs pāra skaitlis.

SECINĀJUMI

Programma tika rakstīta programmēšanas valodā Python. Darbā bija dotas izvēles iespējas spēlētājam - kurš sāks, kāds būs algoritms un kāds būs sākotnējais skaitlis. Tika ģenerēts spēles koks un tā daļas. Tika izstrādāta heiristiskā novērtējuma funkcija un realizēts Minimax algoritms un Alfa-beta algoritms, veikti 10 eksperimenti ar katru no algoritmiem, un izdarīti atbilstoši secinājumi. Turklāt izmantojot Tkinter bibliotēku, izveidots lietotāja grafiskais interfeiss, ar kura palīdzību lietotājs var ērti izvēlēties spēles metodi, redzēt tās norisi un gala rezultātu.

Šis praktiskais uzdevums patiešām bija ļoti interesants un veiksmīgs. Mēs visi bijām ieinteresēti apgūt kursa tēmas, bet praktiskais darbs ļāva mums padziļināt mūsu izpratni un pielietot teorētiskās zināšanas reālajā darbā.

Mūsu komanda strādāja kopā lieliski. Katrs no mums bija atbildīgs par savu uzdevumu, bet vienlaikus mēs arī cieši sadarbojāmies, lai nodrošinātu vienmērīgu progresu un atbalstu vienam otram. Kad kāds no mums atklāja kļūdu vai ierosināja uzlabojumus, mēs ātri reaģējām un pielāgojāmies, lai sasniegtu labākos rezultātus.

Šī kopīgā pieredze ne tikai stiprināja mūsu zināšanas un prasmes kursa tēmu jomā, bet arī uzlaboja mūsu komandas darbību un saikni. Tas ir viens no veidiem, kā mēs augam un attīstāmies gan profesionāli, gan personiski.

VISS PROGRAMMATŪRAS KODS:

```
import tkinter as tk
from tkinter import simpledialog, messagebox
from enum import Enum
from time import perf_counter

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, offset=0) -> None:
    if algorithm == Algorithm.MINIMAX:
        minimax_search(tree)
    elif algorithm == Algorithm.ALPHA_BETA:
        alpha_beta_search(tree)
    print('\t' * offset, 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'level:', 'MAX' if tree[index].level % 2 == 0 else 'MIN', 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, offset + 1)


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:
        time1 = perf_counter()
        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:
            time2 = perf_counter()
            print_tree(tree, self.algorithm)
            print(f'Computer spent {(time2 - time1)*1000:.3f} miliseconds making a move')
        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:
  root = GUI()
  root.mainloop()
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
```