VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS
PROGRAMŲ SISTEMŲ BAKALAURO STUDIJŲ PROGRAMA

Stalo žaidimo dirbtinio intelekto kūrimas naudojant alfa-beta paiešką ir mašininį mokymąsi

Board Game AI Development using Alpha-Beta Pruning and Machine Learning

Bakalauro baigiamasis darbas

Atliko:	Tomas Mikna	(parašas)
Darbo vadovas:	lekt. Rokas Astrauskas	(parašas)
Darbo recenzentas:	dr. Vytautas Valaitis	(parašas)

Vilnius – 2021

Santrauka

Amazonių žaidimas - tai didelio sudėtingumo lentos dominavimo stalo žaidimas, apimantis šachmatų bei go elementus. Dėl itin didelio galimų skirtingų ėjimo kiekio, sukurti šį žaidimą žaidžiantį dirbtinį intelektą yra sudėtinga. Šiame darbe buvo sukurtas toks dirbtinis intelektas panaudojant alfa-beta paieškos algoritmą bei 2 tipų įvertinimo funkcijas - heuristines ir naudojančias konvoliucinį neuroninį tinklą.

Klasikinė alfa-beta paieška buvo pakankamai lėta dėl itin didelio žaidimo medžio išsišakojimo, todėl buvo sukurtos 2 būtent amazonių žaidimui pritaikytos versijos. Abiejų šių versijų žaidimo pajėgumas naudojat tą pačią įvertinimo funkciją yra šiek tiek mažesnis nei klasikinės alfa-beta paieškos, tačiau kelis kartus didesnis greitis leidžia išnagrinėti gilesnį žaidimo medį

Itin paprastos heuristinės įvertinimo funkcijos parodė neblogą rezultatą bei didelę greitaveiką, tuo tarpu konvoliucinio neuroninio tinklo įvertinimo funkcija veikė kur kas lėčiau bei parodė prastesnį rezultatą, pralaimint visas partijas prieš DI su heuristine funkcija.

Raktiniai žodžiai: Amazonių žaidimas, Alfa-beta paieška, Konvoliucinis neuroninis tinklas, Dirbtinis intelektas

Summary

Game of Amazons - high complexity board domination table board game, having Chess and Go elements. It is challenging to create artificial intelligence playing this game because of it's very big possible moves variety. In this paper such AI was created using Alpha-Beta Pruning and 2 types of evaluation functions: heuristic and based on convolutional neural networks.

Very high Game of Amazons game tree branching factor caused classic Alpha-Beta Pruning to be slow. To counter this issue two special algorithm modifications were created for this game specifically. Both showed to be slightly weaker then classical version when in the same search depth, however they both were several times faster allowing to look deeper into the search tree.

Extremely simple heuristic evaluation functions showed satisfactory playing strength and great speed. Convolutional neural network evaluation function was significantly slower and showed worse playing strength as well, losing all games against AIs with heuristic evaluation functions.

Keywords: Game of Amazons, Alpha-Beta Pruning, Convolutional Neural Network, Artificial Inteligence

TURINYS

ĮVADAS 4

1. AMAZONIŲ ŽAIDIMAS 6
   1.1. Kilmė ir taisyklės 6

2. ALGORITMAI IR TECHNOLOGIJOS 7
   2.1. Paieškos algoritmai 7
      2.1.1. Minimax paieška 7
      2.1.2. Alfa-beta paieška 8
   2.2. Įvertinimo funkcija 8
   2.3. Konvoliucinis neuroninis tinklas 9
      2.3.1. Pagrindinės sąvokos 9
      2.3.2. Panaudojimas įvertinimui 10
   2.4. Panašių programų analizė 10
      2.4.1. AlphaGo 10
      2.4.2. Michel Fugers programa 11

3. ĮGYVENDINIMAS 12
   3.1. Žaidimo mechanika 12
   3.2. Alfa-beta paieškos algoritmai 13
      3.2.1. Klasikinis 13
      3.2.2. Ėjimas tada strėlė 13
      3.2.3. Dviejų iteracijų 14
   3.3. Heuristinis būsenos įvertinimas 14
      3.3.1. Pradinė įvertinimo funkcija 14
      3.3.2. Patobulinta įvertinimo funkcija 15
   3.4. Konvoliucinis neuroninis tinklas 15
      3.4.1. Apmokymo duomenys 15
      3.4.2. Įgyvendinimas ir konfigūracija 16
      3.4.3. Skirtingų konfigūracijų tyrimas 18
      3.4.4. Integracija su žaidimu 19

4. SUKURTO DI VERTINIMAS 20
   4.1. Įvertinimo metodikos 20
   4.2. Pajėgumas 20
      4.2.1. Pagerintos heuristinės funkcijos geriausio koeficiento nustatymas 20
      4.2.2. Skirtingų įvertinimo funkcijų palyginimas 22
      4.2.3. Skirtingų alfa-beta algoritmo versijų palyginimas 22
      4.2.4. Žaidimas prieš tikrą žaidėją 23
   4.3. Veikimo greitis 23

REZULTATAI IR IŠVADOS 24

LITERATŪRA 25

PRIEDAI 26
   1 priedas. Amazonė tada strėlė algoritmo pseudokodas 27
   2 priedas. Dviejų iteracijų paieškos algoritmo pseudokodas 29

Įvadas

Stalo žaidimai yra daugelio mėgstama pramoga. Nors ir reikalaujantys pastabumo, šiek tiek loginio mastymo, daugeliui jie yra lengvai suprantami. Žmogus, atlikdamas ėjimą, mato visus galimus ėjimus vienu metu, supranta, kaip kiekvienas ėjimas pakeis lentoje esamą situaciją, ir gali nesunkiai nuspręsti norimą ėjimą, o aiškiai netinkamus ėjimus atmeta be jokių pastangų. Tuo tarpu kompiuteris yra sukurtas tiksliai ir nuosekliai atlikti iš anksto apibrėžtus algoritmus. Nors įvairūs algoritmai ir gali elgtis skirtingai priklausimai nuo išorinių sąlygų, kiekviena elgesio variacija turi būti iš anksto apibrėžta. Stalo žaidimo atveju tai reikštų apibrėžimą, ką daryti esant kiekvienai įmanomai stalo žaidimo (lentos) būsenai. Deja dauguma stalo žaidimų turi itin didelį skaičių galimų būsenų. Pavyzdžiui, Stefan Steinerberger teigimu, šachmatuose yra apie $1.52 * 10^{40}$ skirtingų pozicijų [Ste15]. Aprašyti kaip algoritmas turėtų elgtis kiekvienoje iš jų nėra realu.

Stalo žaidimai yra viena pirmųjų dirbtinio intelekto panaudojimo sričių. Elektroniniams kompiuteriams atsiradus 1940 metais, jau 1954 metais, dar tik formuojantis dirbtinio intelekto sąvokai, buvo sukurta dirbtinio intelekto programa, žaidžianti šaškėmis [Sch09]. Andreas Kaplan ir Michael Haenlein apibrėžia dirbtinį intelektą kaip sistemos sugebėjimą teisingai interpretuoti išorinius duomenis, mokytis iš šių duomenų ir lanksčiai pritaikant išmoktas žinias jas panaudoti, kad būtų įgyvendinti konkretūs tikslai ir uždaviniai [KH19]. Būtent teisingas lentos interpretavimas yra reikalingas siekiant konkretaus tikslo - pergalės. Pagrindinės stalo žaidimams pritaikomos dirbtinio intelekto sritys yra paieškos algoritmai ir mašininis mokymasis, dažnai taikomi kartu. Paprasčiausi ir labiausiai išnagrinėti yra paieškos algoritmai. Vienas populiariausių paieškos algoritmų stalo žaidimams yra Minimax algoritmas, kuris nagrinėja ėjimus iki nustatyto gylio žaidimų medyje ir suranda potencialiai geriausią. Tobulesnė šio algoritmo versija yra alfa-beta paieškos algoritmas, kuris išvengia kai kurių medžio šakų nagrinėjimo, jei jose negali būti geresnio ėjimo nei jau surastas.

Nagrinėjant žaidimo medį pasiekus nustatytą gylį, euristiniais algoritmais yra nustatoma, kiek žaidėjui palanki yra lentos būsena, jei būtų atlikti toje iteracijoje nagrinėjami veiksmai, bei nustatoma ar padėtis geresnė nei geriausia padėtis, gauta iš ankstesnėse iteracijose išnagrinėtų ėjimų kombinacijų. Kai kurių žaidimų, pavyzdžiui, šachmatų, būseną yra pakankamai lengva apibrėžti - galime skaičiuoti esamų figūrų kiekį bei jų vertę ir padėtį priešininko figūrų atžvilgiu. O daliai žaidimų tai apibrėžti yra pakankamai sunku. Pavyzdžiui, GO objektyviai įvertinti taip sunku, kad geras rezultatas gali būti pasiektas tik panaudojus mašininį mokymąsi [SHM+16].

Amazonių žaidimas (Game of the Amazons) yra žaidimas, apimantis tiek šachmatų, tiek GO elementus. Žaidžiama ant šachmatų lentos, tačiau 10x10 dydžio, taip pat figūros po lentą juda taip, kaip karalienė šachmatuose. Tačiau, kaip ir GO, tai yra lentos dominavimo žaidimas, kuriam sukurti optimaliai veikiančią įvertinimo funkciją nėra lengva. Taip pat šiame žaidime, ypač jo pradžioje, yra labai daug galimų ėjimų, kas reikalauja, kad įvertinimo funkcija būtų pakankamai greita, o tai dar labiau apsunkina jos kūrimą. Tinkamai sukurta mašininio mokymosi sistema gali veikti greičiau ir efektyviau nei euristinės įvertinimo funkcijos, tačiau jas sukurti taip pat nėra lengva. Michel Fugers savo magistro darbui sėkmingai sukūrė šį žaidimą žaidžiantį dirbtinį intelektą, įvertinimo funkcijai panaudodamas 1 paslėpto sluoksnio tiesioginio sklidimo neuroninį tinklą. Tačiau nors su-

kurtasis neuroninis tinklas veikė greitai, kaip pats autorius išvadose teigia, jo pajėgumas nusileido netgi tradiciniais būdais kurtoms įvertinimo funkcijoms [M F17].

Tačiau negilūs tiesioginio sklidimo neuroniniai tinklai dažnai yra pranokstami gilių neuroninių tinklų, ypač sudėtingesnėse užduotyse [HS20]. Vienas iš giliųjų neuroninių tinklų tipų, konvoliuciniai neuroniniai tinklai, yra sėkmingai naudojami atpažinti objektams nuotraukose, skaičiams, rašto simboliams bei kitoms struktūroms, kurios yra sudarytos iš erdvėje išsidėsčiusių mažesnių elementų (pvz.: pikselių ar jų grupių nuotraukoje) [RN20]. Stalo žaidimų būsenų vertinimas yra panaši užduotis: Tam reikia atpažinti ir įvertinti galimas sąveikas tarp plokštumoje (žaidimo lentoje) išsidėsčiusių figūrų. Kiekviena situacija yra sudaryta iš mažesnių detalių - kur yra kiekviena figūra, kokia kiekvienos figūros artima aplinka bei galimybės judėti ar paveikti kitas figūras. Taip pat, konvoliuciniai neuroniniai tinklai kartu su kitomis dirbtinio intelekto rūšimis buvo panaudoti kuriant AlphaGo sistemą, Go žaidime nugalėjusią Europos Go čempioną. Todėl šio darbo metu bandoma parodyti, jog amazonių žaidimą žaidžiančio dirbtinio intelekto įvertinimo funkcija gali būti konvoliuciniai neuroniniai tinklai.

Šio darbo uždaviniai:

1. Sukurti dirbtinį intelektą, žaidžiantį Amazonių žaidimą panaudojat alfa-beta paieškos algoritmą.
2. Sukurti heuristinę įvertinimo funkciją žaidimui.
3. Sukurti įvertinimo funkciją panaudojant konvoliucinį neuroninį tinklą.
4. Įvertinti sukurto dirbtinio intelekto pajėgumą.

1. Amazonių žaidimas

1.1. Kilmė ir taisyklės

Amazonių žaidimas (The Game of the Amazons) yra 2 žaidėjų pilnos informacijos strateginis stalo žaidimas, 1988 metais sukurtas Argentiniečio Walter Zamkauskas [Peg].

- Žaidžiama 10x10 langelių lentoje.
- Abu žaidėjai turi po 4 amazones.
- Pradinėje būsenoje baltosios amazonės yra išdėstytos A4/D1/G1/J4 langeliuose, juodosios amazonės yra išdėstytos A7/D10/G10/J7 langeliuose.
- Baltieji pradeda.
- Ėjimas sudarytas iš 2 dalių - Amazonės ėjimo ir strėlės šūvio. Abi dalys yra privalomos:
  - Amazonė juda taip, kaip karalienė šachmatuose - tiesiai ar įstrižai bet kuria kryptimi per 1 ar daugiau langelių.
  - Lygiai 1 strėlę iššauna paėjusi amazonė iš naujosios pozicijos pagal tas pačias taisykles - tiesiai ar įstrižai bet kuria kryptimi per 1 ar daugiau langelių.
  - Amazonė negali nei judėti, nei šauti strėlės į langelį ar už jo, jei jame yra kita amazonė ar strėlė.
- Lentoje gali būti daugiausiai 92 strėlės.
- Ant lentos negali padaugėti ar sumažėti amazonių.
- Visos iššautos strėlės lieka savo pozicijoje iki pat žaidimo pabaigos.
- Pralaimi žaidėjas, kuris pirmas nebeturi nei vieno galimo ėjimo.
- Lygiosios yra neįmanomos.

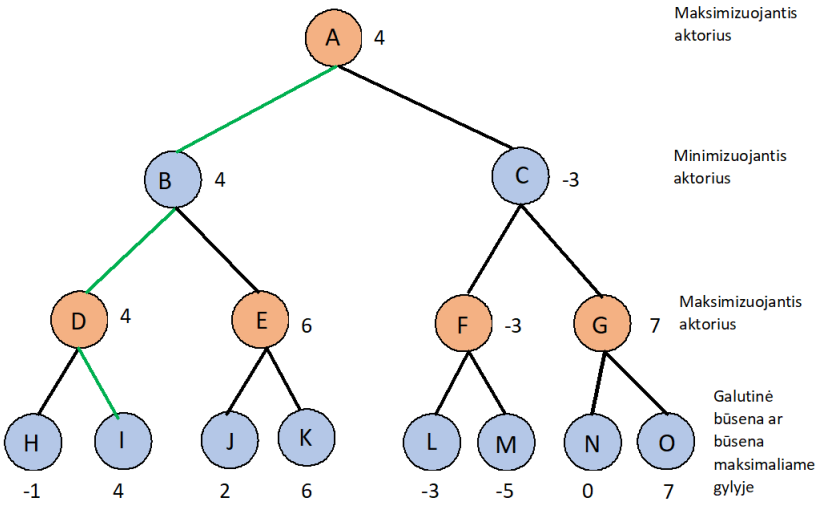
2. Algoritmai ir technologijos

2.1. Paieškos algoritmai

2.1.1. Minimax paieška

Minimax algoritmas, dabar naudojamas keliose srityse, buvo sukurtas nulinės sumos žaidimų teorijai (angl. zero-sum game theory) [Kje01]. Nulinės sumos žaidimai - tai tokie žaidimai, kur vieno žaidėjo praradimas lygus kito žaidėjo įgytam pranašumui. Amazonių žaidimas taip pat yra šio tipo žaidimas - vienam žaidėjui įgijus n galimų ėjimų pranašumą, priešininkui tai yra praradimas, atitinkamai sumažinantis jo galimybes laimėti. Minimax algoritmo principas yra sumažinti maksimalų praradimą. Nulinės sumos žaidimams tai galima perfrazuoti: sumažinti maksimalų priešininko įgytą pranašumą. Tai yra, algoritmas parenka tokį ėjimą, kad priešininkas galėtų įgyti kiek įmanoma mažiau pranašumo (arba patirtų kuo didesnį praradimą) bet kokiu galimu ėjimu.

Algoritmas sudarytas iš dviejų etapų arba aktorių - maksimizuojačio aktoriaus ir minimizuojančio aktoriaus. Maksimizuojantis aktorius parenka tokį ėjimą, kuris suteiktų žaidėjui didžiausią pranašumą, tuomet minimizuojantis aktorius parenka tokį ėjimą, kuris suteikia mažiausią pranašumą (tad suteikia didžiausią pranašumą priešininkui). Minimizuojantis aktorius tarsi atlieka priešininko ėjimą. Tuomet vėl maksimizuojantis aktorius renka geriausią ėjimą ir t.t.



1 pav. Minimax algoritmo iliustracija (nubraižyta remiantis pavyzdžiu: https://www.javatpoint.com/mini-max-algorithm-in-ai)

1 paveikslėlis vaizduoja Minimax algoritmo veikimą: A viršūnė yra pradinė būsena, iš kurios reikia daryti ėjimą. A būsenoje galimi du ėjimai: įvykdžius pirmąjį būsena pasikeistų į B, padarius antrąjį - į C. B ir C būsenose taip pat yra galimi 2 ėjimai, pakeičiantys būseną į būseną, esančią tolimesnėse medžio viršūnėse. Lapai žymi galutines žaidimo būsenas, t.y. pergalę arba pralaimėjimą, arba būsenas nurodytame ieškojimo gylyje. Tuomet maksimizuojantis aktorius D viršūnėje

parenka didžiausią galima vertę max(-1,4) = 4. Atitinkamai, E viršūnei parenkama 6, F - 3, G - 7. Tada minimizuojantis aktorius renka mažiausias galimas reikšmes - B viršūnei tai yra min(4,6) = 4, C viršūnei - -3. Tada vėl maksimizuojantis aktorius renka didžiausią vertę, šiuo atveju tai yra 4. Pradinėje būsenoje A gauname įvertinimą 4, kuris gautas medyje einant iliustracijoje žaliai pažymėtu keliu. Tai reiškia, kad per 3 ėjimus iš būsenos A abiems žaidėjams darant geriausius ėjimus (kurie atitinka žaliai pažymėtą kelią), žaidėjas, kurio eilė dabar daryti ėjimą, gali daugiausia įgyti pranašumą 4. Tam, kad tai pasiektų, turi būti pasirinktas ėjimas į būseną B.

Kuo gilesnis žaidimo medis, tuo didesnė tikimybė išvengti lokaliai optimalių pozicijų, ilgesnėje perspektyvoje vedančių į didesnį praradimą (pavyzdžiui, šachmatuose arklio nukirtimas su pėstininku atrodo labai daug pranašumo suteikiančiu ėjimu, tačiau jei tai atidengia karalienę priešininkui, dar po vieno ėjimo rezultatas jau bus priešininko naudai). Taip pat gilesnis medis didina tikimybę pasiekti galutinę žaidimo būseną, kas leistų įvertinti tiksliai, t.y. ar ėjimų seka baigiasi pergale ar pralaimėjimu. Net ir nepasiekus galutinės būsenos, gilesnis medis pasiekia būsenas, esančias arčiau galutinės, taigi pasiektą būseną leidžia įvertinti tiksliau.

Skirtingai nei 1 paveikslėlyje, realybėje daugumoje žaidimų kiekvienoje būsenoje galima rinktis iš daug galimų ėjimų, tad žaidimo medyje kiekviena viršūnė turi labai daug vaikų. Todėl gilėjant žaidimo medžiui viršūnių skaičius didėja eksponentiškai, kas lemia ir algoritmo veikimui reikalingo laiko eksponentinį didėjimą. Taigi, Minimax algoritmas pakankamai greitai gali žaisti tik ieškodamas geriausio ėjimo mažame gylyje, kas lemia, jog galutinis rezultatas dažnai nebūna labai geras.

2.1.2. Alfa-beta paieška

Alfa-beta paieškos algoritmas (angl. alfa-beta pruning) yra Minimax algoritmo optimizavimas. Tam yra įvedami du papildomi kintamieji - alfa ir beta.

Alfa yra didžiausias jau žinomas maksimizuojačio aktoriaus turimas įvertis, beta - minimizuojačio aktoriaus. Alfa ir beta kintamieji perduodami tik iš viršaus žemyn, todėl tai yra aukščiau medyje esančių maksimizuojačio ir minimizuojančio aktorių įverčiai. Maksimizuojantis aktorius, ieškodamas geriausios viršūnės, taip pat ieško ir viršūnės, geresnės nei alfa, ir radęs atnaujina alfa. Tuomet jei alfa >= beta, tai medyje virš jo esantis minimizuojantis aktorius net ir suradus dar geresnį įvertį, pasirinks beta vertę turinčią viršūnę, todėl tolesnės paieškos tęsti nebeverta, grąžinama alfa reikšmė. Taip pačiai minimizuojantis aktorius ieško mažiausios galimos reikšmės, tačiau, suradęs mažesnę nei alfa, paiešką gali nutraukti, nes net radus viršūnę su mažesniu įverčiu ir šį įvertį grąžinus, virš jo esantis maksimizuojantis aktorius pasirinks alfa viršūnę, kuri yra didesnė.

2.2. Įvertinimo funkcija

Įvertinimo funkcija (angl. Evaluation Function) - tai funkcija ar metodas, apskaičiuojantis žaidimo lentos būsenos palankumą žaidėjui [RN20]. Pilnos informacijos nulinės sumos žaidimams abiejų žaidėjų įvertinimo funkcijos suma paprastai yra 0, tai yra, vieno žaidėjo teigimas pranašumas atitinka kito žaidėjo neigiamą pranašumą ir atvirkščiai. Skaičiais tiksliai įvertinti stalo žaidimo

lentos būseną yra labai sudėtinga, kadangi tik labai nedideli figūrų išsidėstymo skirtumai gali lemti labai skirtingą situaciją lentoje. Šachmatuose, pavyzdžiui, galima skaičiuoti visas įmanomas figūrų sąveikas - jei žaidėjo figūra gali kirsti geresnę žaidėjo figūrą, įvertinimo funkcijos reikšmė stipriai išauga, jei gali kirsti silpnesnę, išauga nestipriai, bet jei tuo pat metu toji silpnesnė figūra gali kirsti žaidėjo geresnę - įvertinimo funkcijos reikšmė mažėja. Tuomet galima vertinti abiejų žaidėjų figūrų kiekį ant lentos bei jų tipus, žaidėjas turintis daugiau geresnių figūrų turi pranašumą. Dar galima vertinti sąveiką tarp savų figūrų ir taip toliau. Tuomet visi šie veiksniai yra sudedami padauginus juos iš tam tikro koeficiento. Surasti skirtingą koeficientą taip pat nėra paprasta, kadangi net ta pati sąveika gali turėti skirtingą įtaką žaidimo būsenai esant skirtingam figūrų išsidėstymui.

Kuomet dirbtiniam intelektui sukurti naudojami paieškos algoritmai, įvertinimo funkcija turi veikti pakankamai greitai, todėl negali būti labai sudėtinga, kas dar apsunkina jos kūrimą, bei verčia rinktis tarp gilesnio paieškos medžio ir geresnio būsenos įvertinimo.

Amazonių žaidimo tikslas yra priversti priešininką nebeturėti jokių galimų ėjimų pirmiau nei pačiam likti be galimų ėjimų, tad žaidėjo amazonių mobilumas yra itin svarbus vertinant žaidimo lentos būseną. Todėl šiame darbe buvo naudotos 2 įvertinimo funkcijos, vertinančios ėjimo žaidėjo mobilumą. Pirmoji funkcija vertina galimų ėjimo pasirinkimų kiekį kitam ėjimui, antroji dar atsižvelgia ir į galimų ėjimo krypčių skaičių kiekvienai amazonei. Detalesnis įvertinimo funkcijų įgyvendinimas aprašytas 3.3 poskyryje.

2.3. Konvoliucinis neuroninis tinklas

2.3.1. Pagrindinės sąvokos

Konvoliucinis neuroninis tinklas - giliojo neuroninio tinklo tipas, galintis išmokti atpažinti erdvines struktūras.

Konvoliucija - tai matricų daugyba tarp filtro (dar vadinamo filtro branduoliu) ir vaizdo ar kitos erdvine matrica išreikštos struktūros.

Filtras (angl. filter) ar filtro branduolys (angl. kernel) - tai svorių matrica, naudojama vykdyti konvoliucijoms. Tas pats filtras naudojamas vykdyti konvoliucijas visai įvesties matricai. Kaskart įvykdant konvoliuciją su įvesties submatrica ir paslenkant per nustatytą žingsnio dydį kol pereinama per visą įvesties matricą.

Nuostolių funkcija (angl. loss function) - tai funkcija, apskaičiuojanti neuroninio tinklo spėjimo nukrypimą nuo teisingo atsakymo. Kuo nuostolių funkcija mažesnė, tuo tikslesnis yra neuroninis tinklas

- Vidutinės kvadratinės paklaidos (angl. Mean squared error - MSE) nuostolių funkcija - regresijos spėjimams įvertinti naudojama nuostolių funcija. Nuostolio verė apskaičiuojama pagal formulę $loss = (y - \widehat{y})^2$, kur y yra tikroji reikšmė, o $\widehat{y}$ - neuroninio modelio spėjimas. Tai yra, skirtumo tarp tikros ir spėtosios reikšmės kvadratas.

- Dvejetainės kryžminės entropijos (angl. Binary Cross-entropy) nuostolių funkcija - logaritminė nuostolių funkcija, paprastai naudojama klasifikacijos uždaviniuose. Ji gali būti užra-

šyta formule $loss = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(\widehat{y_i}) + (1-y_i)log(1-y_i)]$, kur N yra galimų klasių skaičius.

Aktyvacijos funkcija (angl. activation function) - tai funkcija, kuri perskaičiuoja neurono vertę po to, kai iš ankstesnio sluoksnio atėję parametrai yra padauginti iš atitinkamų svorių ir sudėti. Ji nusprendžia kokią reikšmę neuronas perduos į sekantį sluoksnį.

1. Dalimis tiesinio vieneto - (angl. Rectified Linear Unit - ReLU) aktyvacijos funkcija - tiesinė aktyvacijos funkcija, kai įvestis daugiau nei 0: $relu(x) = max(0,x)$. Tai intin paprasta ir greita įvertinimo funcija, taip pat jos paparastai nepaveikia išnykstančio gradiento problema.

2. Sigmoido aktyvacijos funkcija - s formos grafiko funkcija, gražinanti skaičių tarp 0 ir 1, kuo įvesties skaičius didesnis, tuo gražinama vertė arčiau 1 ir kuo mažesnis - tuo arčiau 0. Ji gali būti užrašyta formule: $sigmoid(x) = 1/(1+exp(-x))$

Atsitiktinio praretinimo transformacija (Dropout) - tai atsitiktinis dalies neuronų išjungimas tam, kad išvengti tinklo persimokymo. Kaskart atsitiktinai išjungus dalį neuronų, tinklas turi išmokti atpažinti sąveikas naudodamas skirtingas neuronų kombinacijas, taip priverčiant daugiau neuronų dirbti bei abstrakčiau vertinti įvesties duomenis.

2.3.2. Panaudojimas įvertinimui

Stalo žaidimo lentą ir ant jos išdėstytas figūras galima laikytu 2 dimensijų struktūra (stebint iš viršaus). Žaidėjui palankų ar nepalankų figūrų išsidėstymą taip pat galima vertinti kaip 2 dimensijų ant lentos esančią struktūrą ar objektą. O būtent struktūroms atpažinti konvoliuciniai neuroniniai tinklai yra naudojami itin sėkmingai [RW17]. Išmokius šį neuroninį tinklą atpažinti bei įvertinti ant lentos išsidėsčiusių figūrų visumą gali būti panaudota įvertinti lentos būsenos palankumui žaidėjui. Giliuosius konvoliucinius neuroninius tinklus šachmatuose sėkmingai panaudojo J.Czech su kolegomis [CWB+20]

2.4. Panašių programų analizė

2.4.1. AlphaGo

AlphaGo - dirbtinio intelekto sistema žaidžianti stalo žaidimą GO [SHM+16], kuris laikomas vienu sudėtingiausių žaisti kompiuterinėms sistemoms. AlphaGo 2015 metų spalio mėnesį nugalėjo Europos čempioną Fan Hui rezultatu 5-0. Vėlesnės AlphaGo versijos iki šiol yra nenugalimos šiame žaidime.

AlphaGo veikia panaudojant Monte Karlo paieškos algoritmą. Pozicijos pirmiausia įvertinamos naudojant strategijos tinklą (policy network), tuomet vertės tinklą (value network) ir Monte Karlo paiešką iki maksimalaus gylio be išsišakojimų (Monte Karlo rollouts). Strategijos tinklui sukurti buvo panaudoti konvoliuciniai neuroniniai tinkai, treniruoti su duomenimis iš 160 tūkstančių žaidimo partijų, sužaistų tarp profesionalių žaidėjų, kurie dar buvo papildyti duomenų augmentacijos būdu. Tuomet tinklas buvo toliau tobulinamas naudojant skatinamąjį mokymąsi.

Vertės tinklas buvo treniruotas regresijos būdu treniravimui naudojant ėjimus iš strategijos tinklo žaistų žaidimo partijų ir bet kokius galimus atsitiktinius ėjimus.

Visas treniravimas užtruko apie 4 savaites, buvo naudojama 50 vaizdo plokščių [SHM+16].

2.4.2. Michel Fugers programa

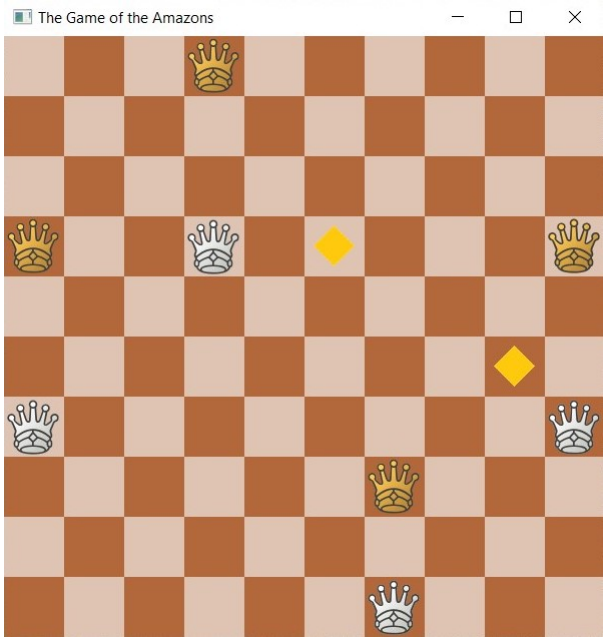
Michel Fugers savo magistriniam darbui sukūrė amazonių žaidimą, žaidžiantį DI. M. Fugers tyrinėjo alfa-beta bei Monte Karlo paieškos medžius su heuristinėm bei neuroninio tinklo aktyvacijos funkcijomis. Neuroniniui tinklui M. Fugers panaudojo paprastos struktūros, vos vieno paslėpto sluoksnio gylio laikinųjų skirtumų mokymo algoritmą (angl. Temporal Difference Learning). Neuroninio tinklo apmokymui autorius naudojo tikrų žaidėjų duomenis 3 dimensijų matricos formoje. Jis išbandė 3 duomenų formatus: kuomet atskirai užkoduojama blokados (bet kuri amazonė ar strėlė) ir strėlės (žaidėjo, priešininko ar nėra), atskirai užkoduojama blokados ir tame langelyje esančios amazonės mobilumas, atskirai užkoduojama žaidėjo amazonės, priešininko amazonės, strėlės, tušti langeliai. Autorius dar naudojo 3 papildomas savybes: atliktų ėjimų skaičių, juodų bei baltų amazonių mobilumus. Elioto simetrinė aktyvacijos funkcija buvo panaudota paslėptame sluoksnyje bet tiesinė aktyvacija išvesties sluoksnyje. Deja, kaip matyti pateiktuose rezultatuose, naudojant aktyvacijos funkciją, sukurtą naudojant neuroninį tinklą, rezultatai buvo prasti, DI pasirodė tik vos geriau nei DI, naudojantis atsitiktinę reikšmę grąžinančią įvertinimo funkciją [M F17].

3. Įgyvendinimas

3.1. Žaidimo mechanika

Žaidimas sukurtas C++ kalba. C++ pasirinkta dėl greito veikimo ir efektyvių duomenų saugojimų struktūrų, kas leidžia paieškos algoritmams veikti griečiau ir palengvina kūrimo procesą, sumažina testavimo laiką.

Grafinei vartotojo sąsajai sukurti buvo panaudota SFML (Simple and Fast Multimedia Library, https://github.com/SFML/SFML#sfml–simple-and-fast-multimedia-library) - tai atviro kodo C++ biblioteka, skirta žaidimų ir multimedijos programų grafinei sąsajai, garso ir tinklo ryšio funkcionalumams kurti. Šiame darbe SFML garso ir tinklo ryšio moduliai nėra naudojami. Kadangi grafinė sąsaja nėra itin svarbi algoritmo analizavimui, ji yra minimalistinė (2 pav.), tačiau pakankama matyti lentos būsenai bei žaisti žaidimui.



2 pav. Grafinė vartotojo sąsaja, atlikti 2 ėjimai

Lentos būsena saugoma dvimačiame masyve, kuriame skirtingais sveikaisiais skaičiais žymima langelio būsena: tuščias langelis, užimtas baltos amazonės, užimtas juodos amazonės ar užimtas strėlės. Kadangi nėra skirtumo tarp žaidėjo ir priešininko padėtos strėlės, jos žymimos tuo pačiu skaičiumi. Atskiruose masyvuose papildomai saugoma baltųjų ir juodųjų amazonių padėtis, kad jas būtų galima greičiau surasti žaidimo lentos masyve. Šis saugojimo metodas tinkamas paieškos algoritmams bei heuristinėms įvertinimo funkcijoms, tačiau pradėjus planuoti konvoliucinio neuroninio tinklo struktūrą buvo suprasta, jog skirtingų būsenų žymėjimas skirtingais skaičiais būsenas tam tikra prasme išrikuoja pagal joms žymėti naudojamą skaičių. Todėl lentos būsenos saugojimui buvo pridėtas trijų dimensijų dvejetainio tipo masyvas, kurio pirmos 2 dimensijos yra lentos aukštis ir plotis, o 3 dimensija išskaido lentą į atskirus sluoksnius: baltas amazones, juodas amazones, strėles, bei užimtus langelius bendrai, kiekvieno iš šių elementų buvimas yra žymimas kaip ’tiesa’ (dvejetainis 1), o nebuvimas - netiesa (dvejetainis 0).

3.2. Alfa-beta paieškos algoritmai

3.2.1. Klasikinis

Tai yra klasikinis alfa-beta paieškos algoritmo įgyvendinimas. Kadangi vienas amazonių žaidimo ėjimas susideda iš 2 veiksmų, randamos ir nagrinėjamos visos galimos jų kombinacijos (išskyrus tas, kurios yra atmetamos alfa-beta algoritmo). Šį algoritmą pseudokodu galima užrašyti taip:

Algorithm 1 Klasikinė alfa-beta Paieška

```
function AlphaBeta(boardState, depth, alpha, beta, isMaximising)
    if isMaximising then
        if depth = 0 or terminalState then return Evaluate(boardState)
        end if
        maxEvaluation = −∞
        for all legal moves do
            for all legal arrows from the move do
                evaluation = AlphaBeta(boardState, depth − 1, alpha, beta, false)
                maxEvaluation = max(maxEvaluation, evaluation)
                alpha = max(alpha, maxEvaluation)
                if alpha ≥ beta then break
                end if
            end for
        end for
        return maxEvaluation
    else
        if depth = 0 or terminalState then return Evaluate(boardState)
        end if
        minEvaluation = +∞
        for all legal moves do
            for all legal arrows from the move do
                evaluation = AlphaBeta(boardState, depth − 1, alpha, beta, true)
                minEvaluation = min(minEvaluation, evaluation)
                beta = min(beta, minEvaluation)
                if alpha ≥ beta then break
                end if
            end for
            if alpha ≥ beta then break
            end if
        end for
        return minEvaluation
    end if
end function
```

3.2.2. Ėjimas tada strėlė

Pirma randamas geriausias amazonės ėjimas, tada geriausias strėlės šūvis.

Kadangi amazonių žaidime kiekvienas ėjimas sudarytas iš dviejų veiksmų, jų kombinacija

sudaro labai didelį kiekį galimų ėjimų. Pavyzdžiui, pirmajam ėjimui yra virš 2000 galimų variantų (palyginimui - šachmatuose 50, šaškėse - vos 7). Tam, kad sumažintų kombinacijų skaičių, šis algoritmas pirmiausia suranda geriausią amazonės ėjimą ir tada geriausią strėlės poziciją. Šis algoritmas galimai praleidžia ir dalį potencialiai gerų ėjimų, tačiau veikia patenkinamai (rezultatai žaidžiant prieš atsitintintį DI bei kitus algoritmus yra 5 lentelėje). Šio algoritmo pseudokodas pateiktas priede nr. 1.

3.2.3. Dviejų iteracijų

Amazonės ėjimas ir strėlės šūvis atskirose iteracijose.

Šis algoritmas yra tarpinis variantas tarp dviejų pirmųjų algoritmų. Ėjimas išskaidomas į dvi dalis, kurios atliekamos skirtingose iteracijose. Taip gauname, jog žaidimo medyje kiekviena viršūnė yra išskaidoma į dvi - amazonės ėjimo ir strėlės šūvio. Tuomet medyje gaunasi 2 lygiai maksimizuojančio agento (geriausia strėlė ir ėjimas) bei 2 minimizuojančio. Taigi yra patikrinami visi galimi amazonės ėjimai ir kiekvienam ėjimui patikrinamos visos strėlės pozicijos, tačiau tai atliekama per 2 viršūnes (t.y. per 2 algoritmo iteracijas), todėl atsiranda daugiau vietų, kur algoritmas gali vykdyti medžio genėjimą (angl. pruning), taip ženkliai pagreitindamas algoritmo veikimą. Šio algoritmo pseudokodas pateiktas priede nr. 2.

Verta pastebėti, jog nors ėjimas ir išskaidytas į 2 iteracijas, jos abi laikomos to pačio gylio, todėl gylis mažinamas tik antrojoje iteracijoje. Taip pat, būsenos įvertinimas skaičiuojamas tik ėjimo pradžioje (ėjimo viduryje galutinės būsenos neįmanoma pasiekti, o gylis nepakinta).

3.3. Heuristinis būsenos įvertinimas

3.3.1. Pradinė įvertinimo funkcija

Ši įvertinimo funkcija skaičiuoja santykį tarp žaidėjui ir jo priešininkui galimų ėjimų skaičiaus. Jei a - žaidėjui galimų ėjimų skaičius, b - priešininkui galimų ėjimų skaičius, tai įvertinimo funkciją galima užrašyti taip: $(a - b + bias)/(a + b + bias)$. Bias skirtas išvengti dalybos iš 0, jei abu žaidėjai neturi galimo ėjimo. Taip pat, kadangi esant lygiam galimų ėjimų skaičiui tas žaidėjas, kurio dabar ėjimas, yra pralaimėnčiojoje pozicijoje, todėl bias leidžia įvertinti tokią poziciją, kaip jam mažiau palankesnę nei priešininkui. $-1 < bias < 1, bias \neq 0$. Bias neigiamas, kai dabartinis ėjimas yra žaidėjo ėjimas, ir teigiamas - kai priešininko. Skirtingų algoritmų lyginimo bei laiko testavimo metu buvo nustatyta bias = 0,4. Taip pat bias įtraukia labai mažą atsitiktinį skaičių, kad išvengtume deterministio algoritmo veikimo ir būtų galima tiksliau palyginti žaidžiant keletą ar keliasdešimt partijų.

Įvertinimo funkcijos vertė yra tarp 1 ir -1. 1 reiškia žaidėjas laimėjo, -1 - pralaimėjo. Jei vertė > 0, tai yra žaidėjui palanki pozicija, jei < 0 - nepalanki, jei 0 - būsena yra neutrali, t.y. nesuteikianti jokio pranašumo nei vienam žaidėjui.

3.3.2. Patobulinta įvertinimo funkcija

Tai yra patobulinta pradinė įvertinimo funkcija. Ši funkcija taip pačiai skaičiuoja ėjimų santykį, tačiau papildomai dar skaičiuoja galimų amazonių ėjimo krypčių santykį, skaičiuojama tuo pačiu principu: $(a - b + bias)/(a + b + bias)$, kur a - visoms žaidėjo amazonėms galimų judėjimo krypčių suma, b - visoms priešininko amazonėms galimų judėjimo krypčių suma, bias - tas pats kaip ir pagrindinėje įvertinimo funkcijoje. Tuomet ėjimų santykis padauginamas iš koeficiento cof1, galimų krypčių santykis padauginamas iš koeficiento cof2 ir sudedama. $0 \leq cof1 \leq 1$, $cof2 = 1 - cof1$. Abiejų koeficientų suma yra 1, todėl galutinė šios įvertinimo funkcijos reikšmė išlieka tarp -1 ir 1 ir gali būti interpretuojama taip pačiai, kaip ir pagrindinė įvertinimo funkcija.

3.4. Konvoliucinis neuroninis tinklas

3.4.1. Apmokymo duomenys

Duomenų iš profesionalių žaidimų partijų (pvz. oficialių turnyrų) rasti nepavyko. Sugeneruoti duomenis neturint jokio labai gerai žaidžiančio agento taip pat nebuvo įmanoma. Todėl tinklo treniravimui buvo naudojami duomenys iš https://www.littlegolem.net/ svetainės. „Little Golem“ - tai internetinė stalo žaidimų platforma, kurioje žaidėjai gali tarpusavyje žaisti įvairius žaidimus, organizuojami internetiniai turnyrai. Amazonių žaidimas yra vienas iš svetainėje siūlomų žaidimų. Duomenys apie visų žaidimų eigą yra viešai prieinami tekstiniu formatu. Taip pat, žaidėjai yra reitinguojami pagal jų rezultatus. Neuroninio tinklo apmokymui panaudota 100 geriausią reitingą turinčių žaidėjų (apytiksliai viena trečioji geriausiai įvertintų žaidėjų) sužaistos partijos, kartu sudėjus - virš 19000 žaidimo partijų.

Dalis partijų yra pradėtos, tačiau nepabaigtos, todėl visos partijos, kuriose atlikta mažiau nei 10 ėjimų, buvo iš karto pašalintos iš apmokymo duomenų. Taip pat, dauguma iš svetainės paimtų partijų nėra iki galo užbaigtos, nes Amazonių žaidime laimėtojas dažniausiai būna aiškus dar prieš žaidimo pabaigą, kuomet abu žaidėjai dar turi ėjimų, bet vienas iš žaidėjų jų turi aiškiai daugiau ir, jam nepadarius labai akivaizdžios klaidos, priešininkas nebeturi šansų laimėti ir pasiduoda. Todėl galutinė lentos būsena buvo įvertinta paprasta heuristine įvertinimo funkcija, vertinančia galimų ėjimų ir galimų krypčių kiekį kiekvienai amazonei. Funcija plačiau aprašyta 3.3.2 skirsnyje. Visos partijos, kurių galutinės būsenos įvertinimas buvo arti 0 (t.y. nei vienas žaidėjas neturi ryškaus pranašumo) buvo pašalintos iš apmokymo duomenų. Toliau visi žaidimai buvo sužaisti iki galo DI žaidėjų. Kiek stipresnis DI žaidėjas žaisdavo už prastesnėje pozicijoje esančia spalvą ir silpnesnis - už geresnėje. Jei silpnesnis DI laimėdavo, laikyta, jog žaidimas buvo pabaigtas, jis įtraukiamas į galutinį apmokymo duomenų rinkinį. Tuo tarpu jei stipresnis DI žaidėjas, būdamas prastesnėje pradinėje pozicijoje, laimėdavo, partija laikyta neturinti aiškaus laimėtojo ir buvo neįtraukiama į galutinį apmokymo duomenų rinkinį. Panašiu duomenų atrinkimo principu rėmėsi Michel Fugers [M F17].

Atrinkus duomenis liko 8680 treniravimui tinkamų partijų, kurias sudarė 304757 žaidimo lentos būsenos. Taip pat dalyje eksperimentų ieškant geriausios konvoliucinio tinklo konfigūracijos, tam kad sutrumpinti treniravimo laiką, naudota 30% turimų duomenų - atsitiktinai atrinktos

88682 žaidimo lentos būsenos. Galiausiai, pastebėta, jog naudojant tik geriausių žaidėjų duomenis pasiekiamas šiek tiek geresnis rezultatas, todėl panaudota duomenys iš 88820 lentos būsenų iš 20 geriausių žaidėjų žaistų partijų.

3.4.2. Įgyvendinimas ir konfigūracija

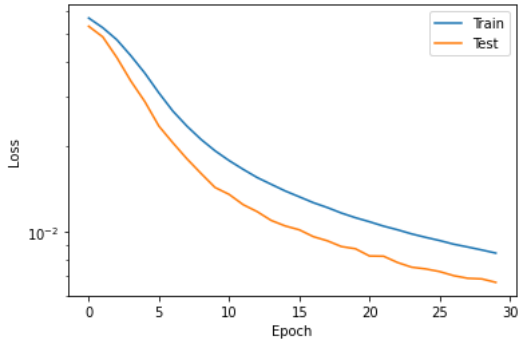
Konvoliucinio neuroninio tinklo kūrimui naudota Python programavimo kalba, bei Tensorflow ir Keras platformos. Neuroninis tinklas kurtas bei treniruotas Anaconda3 Jupyter Notebook aplinkoje. Galutinio neuroninio tinklo struktūra pateikta 1 lentelėje.

1 lentelė. Konvoliucinio neuroninio tinklo struktūra

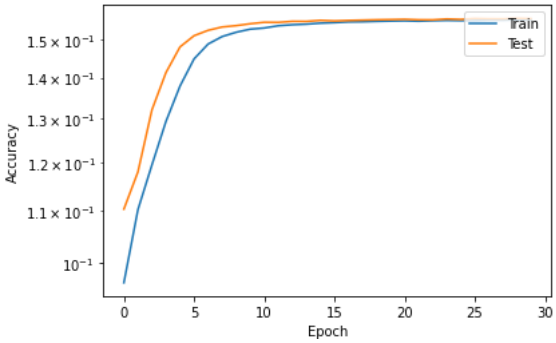
Eil. nr.	Sluoksnio tipas	Išvesties forma	Prametrų sk.
1	Conv2D	(None, 10, 10, 32)	1184
2	Dropout	(None, 10, 10, 32)	0
3	Conv2D	(None, 10, 10, 32)	9248
4	Conv2D	(None, 10, 10, 32)	9248
5	Dropout	(None, 10, 10, 32)	0
6	Flatten	(None, 3200)	0
7	Dense	(None, 256)	819456
8	Dense	(None, 256)	65792
9	Dense	(None, 1)	257

Kiti pagrindiniai parametrai:

- Dalimis tiesinio vieneto (ReLU) nuostolių funkcija paslėptuosiuose sluoksniuose. Taip pat buvo pabandyta panaudoti sigmoido bei keletą kitų keras bibliotekos siūlomų nuostolių funkcijų, tačiau rezultatai buvo prastesni nei naudojant ReLU
- Sigmoido nuostolių funkcija paskutiniame sluoksnyje.
- Vidutinės kvadratinės paklaidos nuostolių funkcija. Taip pat buvo pabandyta dvejetainės kryžminės entropijos nuostolių funkcija, tačiau tai nedavė norimo rezultato
- 30 treniravimo epochų. Taip pasirinkta dėl to, nes nuostolio grafike (3 paveikslėlis), nuostolis ties 30 epocha jau mažėja lėtai, o tikslumas (4 paveikslėlis) nustoja didėti dar prieš 15 epochą. Taip pat buvo pabandyta treniruoti 50 epochų, bet rezultatas pagerėjo labai nežymiai. (2 lentelės 4 ir 5 eilutės)
- Įvestis - žaidimo lentos būsena.
- Išvestis - įvertinimo funkcijos reikšmės spėjimas žaidimo pabaigoje



3 pav. Nuostolio (angl. loss) priklausomybė nuo treniravimo epochų skaičiaus epochų skaičius



4 pav. Tikslumo (angl. accuracy) priklausomybė nuo treniravimo epochų skaičiaus epochų skaičius

3.4.3. Skirtingų konfigūracijų tyrimas

2 lentelė. Geriausius rezultatus pasiekusių konfigūracijų palyginimas. MN - minimalus nuostolis, MT - maksimalus tikslumas, MG - Mokymosi greitis, Tn - n-tojo Tiesinio neuroninio sluoksnio filtrų skaičius, EP - epochų skaičius, DRD - Duomenų rinkinio dydis (angl. batch size), ADK - apmokymo duomenų (lentos būsenų) kiekis, APTn - n-tos Atsitiktinio praretinimo transformacijos koeficientas, Kn - n-tojo konvolioucinio sluoksnio filtrų skaičius. * - įvertinimo funkcijai panaudoto konvoliucinio tinklo konfigūracijos

	MN	MT	MG	T1	T2	T3	EP	DRD	ADK	APT1	APT2	K1	K2	K3
1*	0.007051	0.172888	0.02	256	256	256	30	64	88820	0.2	0.2	32	32	32
2*	0.006558	0.156179	0.02	256	256	256	30	64	304749	0.2	0.2	32	32	32
3	0.007442	0.155930	0.02	128	128	128	30	64	304749	0	0.2	32	32	32
4	0.005461	0.156258	0.02	256	256	256	30	64	304749	0.2	0.2	64	64	64
5	0.017463	0.152722	0.08	256	256	256	30	64	88682	0.2	0.2	32	32	32
6	0.012662	0.154210	0.02	256	256	256	50	64	88682	0.2	0.2	32	32	32
7	0.012805	0.153669	0.005	256	256	256	50	64	88682	0.2	0.2	32	32	32
8	0.018156	0.152541	0.04	256	256	256	30	64	88682	0.2	0.2	32	32	32
9	0.008649	0.154887	0.08	256	256	256	30	64	88682	0.2	0.2	32	32	32
10	0.016752	0.153263	0.01	256	256	256	30	64	88682	0.2	0.2	32	32	32
11	0.012016	0.154210	0.02	256	128	64	30	64	88682	0.2	0.2	32	32	32
12	0.015084	0.153669	0.02	64	128	256	30	64	88682	0.2	0.2	32	32	32
13	0.011507	0.154210	0.02	256	256	256	30	64	88682	0.2	0.2	32	32	32
14	0.012566	0.154300	0.02	128	128	128	30	64	88682	0.2	0.2	32	32	32
15	0.010718	0.154345	0.02	512	128	128	30	64	88682	0.2	0.2	32	32	32
16	0.013064	0.153849	0.02	128	128	512	30	64	88682	0.2	0.2	32	32	32
17	0.026750	0.146136	0.02	32	32	32	25	64	88682	0.2	0.5	32	32	32
18	0.021205	0.150557	0.02	32	32	32	25	64	88682	0.2	0.3	32	32	32
19	0.016188	0.152992	0.02	32	32	32	30	64	88682	0.2	0.1	32	32	32
20	0.016948	0.151819	0.02	32	32	32	30	64	88682	0.2	0.2	32	32	32
21	0.023226	0.149790	0.02	32	32	32	30	128	88682	0.2	0.2	32	32	32
22	0.046042	0.124937	0.02	32	32	32	30	512	88682	0.2	0.2	32	32	32
23	0.014126	0.152676	0.02	32	32	32	30	16	88682	0.2	0.2	32	32	32
24	0.014419	0.152857	0.02	32	32	32	30	32	88682	0.2	0.2	32	32	32
25	0.028822	0.146091	0.02	16	16	16	30	64	88682	0.2	0.2	16	16	16

2 lentelė įtraukia tik geriausius rezultatus pasiekusias konvoliucinio tinklo konfigūracijas. geriausias rezultatas buvo pasiektas naudojat 20 geriausių žaidėjų duomenis (1 eilutė), su ta pačia konfigūracija taip pat pasiektas antras geriausias rezultatas. Buvo vienu metu nagrinėjamas tam tikro parametro poveikis, pavyzdžiui 20-24 eilutėse ieškoma optimalaus duomenų rinkinio dydžio, nustatyta, kad 64 lentos būsenų dydis yra optimalus - jį didinat mažėja tikslumas, o toliau mažinant tikslumas nekinta, tačiau mažesnis duomenų rinkinio dydis prailgina treniravimo laiką tad taupant laiką nebeverta toliau mažinti. 11-16 eilutėse tiriama skirtingas neuronų kiekio išsidėstymas, 2-4 geriausios konfigūracijos išbandomos su pilnu duomenų rinkiniu, bandoma sumažinti atsitiktinio praretinimo transformaciją.

3.4.4. Integracija su žaidimu

Kadangi žaidimas ir neuroninio tinklo modelis buvo sukurti skirtingomis programavimo kalbomis, frugally-deep biblioteka buvo panaudota juos integruoti. Frugally-deep yra C++ biblioteka, skirta integruoti Keras modelius C++ aplinkoje. Keras modelis, išsaugotas .h5 formatu, yra konvertuojamas į json failą, iš kurio biblioteka sukuria modelį C++ kalboje. Šios kalbos privalumai - itin lengva integracija, išlaikomas didelis modelio veikimo greitis. Pagrindinis trūkumas - biblioteka nepalaiko grafikos plokščių, taip pat ir modelių, treniruotų naudojant grafikos plokštę. Todėl tyrinėjant skirtingų modelio konfigūracijų rezultatus, jie buvo treniruojami naudojant vaizdo plokštę, tačiau modeliai, atrinkti testuoti žaidime, turėjo būti treniruojami iš naujo, naudojant tik procesorių. Kadangi, išskyrus pavienius atvejus, treniravimas naudojant vien procesorių užtrukdavo bent 5 kartus ilgiau, papildomas reikalingas laikas stipriai apribojo žaidime išbandytų modelių kiekį.

4. Sukurto DI vertinimas

4.1. Įvertinimo metodikos

Bandant sukurti objektyvią įvertinimo metodiką paaiškėjo, jog tai yra itin sudėtinga. Visų pirma, šis žaidimas nėra labai populiarus, todėl sunku rasti gerų žaidėjų, prieš kuriuos žaidžiant būtų galimą sukurtą DI įvertinti ar bent su kurių žaidimu būtų galima palyginti. Kita problema - tranzityvumo nebuvimas. Jei žaidėjas A laimi prieš žaidėją B, o žaidėjas B laimi prieš žaidėją C negarantuoja, jog žaidėjas A laimės ir prieš žaidėją C. Vertinti per kiek ėjimų DI laimi ar pralaimi taip pat nėra tikslinga, kadangi tai didele dalimi priklauso nuo žaidėjo strategijos.

Bandyti vertinti atskirus ėjimus ar situacijas neverta dėl to, nes žaidimo lygis nėra tolygus [M F17] ir net neuroninių tinklų pagalba kurtos įvertinimo funkcijos skirtingas situacijas gali įvertinti geriau ar prasčiau. Galimos to priežastys - panašios situacijos visiškas nebuvimas tarp neuroninio tinklo apmokymo duomenų (tai ypač aktualu amazonių žaidime, kadangi galimų situacijų kiekis yra itin didelis, o apmokymui tinkamų duomenų kiekis ribotas, ypač žaidimo pabaigoje), situacijos, kuomet ėjimas atrodo geras, tačiau atneša žalos po daugiau ėjimų nei paieškos medžio gylis ir pan.

Todėl DI žaidėjai buvo vertinami tik žaidžiant visiems prieš visus ir lyginant bendrą pergalių skaičių. Taip pat 2 DI žaidėjai gali būti palyginti tarpusavyje, tačiau tokį rezultatą reikia interpretuoti kaip ”žaidžiant tarpusavyje A yra geresnis nei B, tačiau nebūtinai geresnis globaliai”.

Kitoks būdas įvertinti DI, tai vertinti veikimo greitį. Tą galima daryti dviem būdais: vertinti tik DI sugaištą laiką nuspręsti jo ėjimams arba vertinti visą partijos trukmę. Pirmuoju atveju objektyviau įvertinamas DI veikimo laikas. Antru atveju įvertinama ir kaip labai žaidimas yra apsunkinamas priešininkui. Tačiau šiuo atveju laikas labai priklauso nuo priešininko greitaveikos, todėl jį lyginti su kito DI laiku žaidžiant prieš kitą priešininką būtų netikslinga. Laiko vertinimas tiesiogiai neparodo, kuris DI žaidėjas yra pajėgesnis, tačiau kartu su žaidimo rezultatais padeda vertinti panašaus pajėgumo DI žaidėjus: greitesnis DI žaidėjas įgyja pranašumą esant ribotam partijos laikui, jis turi daugiau potencialo tobulinimui išlaikant pakankamą greitaveiką - pridėjus papildomos logikos, pagilinus paieškos medį ar kitaip sustiprinus DI jis potencialiai taptų geresnis ir susilygintų greitaveika su lėtesniu, taip tapdamas už jį geresniu. Be to, didesnė greitaveika reiškia jog ėjimas atliekamas su mažiau skaičiavimų, taigi skaičiavimai yra tikslesni (t,y, atliekama mažiau nereikalingo tikrinimo) nei panašaus lygio lėčiau žaidžiančio DI.

4.2. Pajėgumas

4.2.1. Pagerintos heuristinės funkcijos geriausio koeficiento nustatymas

Buvo palygintos pagrindinė ir patobulinta įvertinimo funkcijos. Buvo lyginama dviejų iteracijų algoritmas su pradine įvertinimo funkcija žaidžiantis prieš tą patį algoritmą su patobulinta įvertinimo funkcija. Šis algoritmas pasirinktas dėl to, nes jis veikia daug greičiau nei klasikinis, kas palengvina testavimą, taip pat jis yra pranašesnis nei ėjimas tada strėlė algoritmas. Rezultatai matomi 3 lentelėje. Kadangi $cof2 = 1 - cof1$, lentelėje parodomas tik $cof1$. Žaidžiama 100 partijų po to apsikeičiama spalvomis. Skaičius spalvos langelyje nurodo pergalių bei pralaimėjimų

skirtumą pagerintai įvertinimo funkcijai žaidžiant už tą spalvą.

3 lentelė. Skirtingų įvertinimo funkcijų palyginimas

Cof1	Balti,	Juodi	Rezultatas
0,0	-13	-16	-29
0,1	-74	30	-44
0,2	-42	40	-2
0,3	-16	52	46
0,35	30	28	58
0,4	35	14	49
0,43	54	6	60
0,45	62	34	96
0,47	72	16	88
0,5	68	24	92
0,55	100	-44	56
0,6	-26	-4	-30
0,7	-22	-2	-24
0,8	14	-65	-51
0,9	-99	100	1
1,0	-8	4	-4

Lentelėje nr. 3 matyti, jog pagrindinė įvertinimo funkcija pranoksta pradinę įvertinimo funkciją, kai $cof1$ yra intervale tarp 0,3 ir 0,55. Kadangi $cof1$ nurodo kokią galį galutinės įvertinimo funkcijos sudarys galimų ėjimų santykį, geresni rezultatai šiame intervalo rodo, jog galimų ėjimų santykis yra kiek mažiau svarbus nei galimų krypčių santykis vertinat žaidimo lentos būseną. O kuomet galimų ėjimų santykis sudaro daugiau nei pusę galutinio įvertinimo, rezultatai tampa prastesni nei naudojant vien tik šį santykį. Iš to išplaukia išvada, jog galimų krypčių santykis gali pagerinti įvertinimo funkcijos veikimą, tačiau kuomet jo dalis galutinėje funkcijos gražinamoje reikšmėje yra per mažas, jis tik iškreipia įvertinimą, tačiau neįtakoja pakankamai kad teigiamai paveikti galutinį būsenos įvertinimą. Taip pat lentelėje matyti, jog geriausias rezultatas pasiektas, kai $cof1 = 0{,}45$, todėl ši reikšmė ir buvo naudojama tolesniuose DI tyrimuose.

4.2.2. Skirtingų įvertinimo funkcijų palyginimas

4 lentelė. Skirtingų įvertinimo funkcijų palyginimas. Žaidžiama po 20 partijų už kiekvieną spalvą, su 2 iteracijų algoritmu. Langelio reikšmė - DI su eilutės įvertinimo funkcija pergalių/pralaimėjimų skaičius prieš DI su stulpelio įvertinimo funkcija. DI nugalėta - laikoma kad DI nugalėjo kitą DI, jei prieš jį laimėjo daugiau kartų nei pralaimėjo. Pradinė ir pagerinta - heuristinės, KNT1 ir KNT2 - konvoliucinio neuroninio tinklo įvertinimo funkcijos. KNT1 treniruota su pilnais duomenimis, KNT2 - su 20 geriausių žaidėjų duomenimis.

Algoritmas	Pradinė	Pagerinta	KNT1	KNT2	Bendras rez.	DI nugalėta
Pradinė	14-6	17/23	40/0	40/0	97/23	2
Pagerinta	23/17	7-13	40/0	40/0	103/17	3
KNT1	0/40	0/40	X	20/20	20/100	0
KNT2	0/40	0/40	20/20	X	20/100	0
Bendras rezultatas	97/23	103/17	20/100	20-100	-	-
DI pralaimėta	1	0	2	2	-	-

4 lentelėje apžvelgta įvertinimo funkcijų tarpusavio palyginimas. DI su pagerinta heuristine įvertinimo funkcija laimėjo šiek tiek daugiau partijų nei pradinė, tačiau jos abi užtikrintai laimėjo prieš DI su konvoliucinio neuroninio tinklo įvertinimo funkcija, kurios pasirodė vienodai prastai. Taip pat buvo pastebėta įdomi tendencija - tarpusavyje žaidžiant DI su konvoliucinio neuroninio tinklo įvertinimo funkcijomis dažniau laimėdavo už Juodą komandą žaidžiantis DI. To priežastis galėtų būti netolygus treniravimo duomenų pasiskirstymas pagal laiminčią komandą.

4.2.3. Skirtingų alfa-beta algoritmo versijų palyginimas

5 lentelė. Skirtingų versijų palyginimas. Žaidžiama po 20 partijų už kiekvieną spalvą. Langelio reikšmė - eilutės algoritmo pergalių/pralaimėjimų skaičius prieš stulpelio algoritmą. Žaidžant prieš save pirmas skaičius yra baltųjų pergalės - antras juodųjų.

Algoritmas	Atsitiktinis	Klasikinis	Ėjimas tada strėlė	Dviejų iteracijų	Suma
Atsitiktinis	17/23	0/40	0/40	0/40	0/120
Klasikinis	40/0	19-1	40/0	36/4	116/4
Ėjimas tada strėlė	40/0	0/40	9-11	12/28	52/68
Dviejų Iteracijų	40/0	4/36	28/12	14-6	82/48
Suma	17/143	116/4	52/68	82/48	-

Klasikinė alfa-beta algoritmo versija esant vienodam paieškos gyliui žaidžia geriausiai, pralaimėdama vos keletą partijų dievų iteracijų algoritmui. Kartu tai parodo, jog ir dviejų iteracijų algoritmas, nors ir žaidžia prastai, nėra visiškai beviltiškas ir gali būti panaudotas, ypač kuomet žaidimo laikas yra svarbus. Ėjimas tada strėlė algoritmas pasirodė beviltiškai prieš klasikinį alfa-beta paieškos algoritmą, tačiau laimėjo beveik trečdalį partijų prieš 2 iteracijų algoritmą. Tai parodo jė esant prasčiausią iš paieškos algoritmų, tačiau kartu su faktu, kad jis užtikrintai laimėjo prieš atsitiktinį žaidėją, parodo, jog is taip pat yra veikiantis.

4.2.4. Žaidimas prieš tikrą žaidėją

DI su konvoliucinio neuroninio tinklo įvertinimo funkcija pajėgumas yra itin menkas, autorius (nepatyręs žaidėjas) gali užtikrintai nugalėti šį DI be didesnių pastangų. Tuo tarpu žaisti prieš DI su bet kuria iš heuristinių įvertinimo funkcijų yra sunkiau. Norint laimėti prieš šį DI autoriui reikėjo apgalvoti ėjimus, pasistengi, autorius sugebėjo laimėti ne kiekvieną partiją, ypač jei naudojamas didesnis medžio gylis. (Dviejų iteracijų algoritmas su bet kuria iš heuristinių funkcijų pakankamai greitai veikia 5 lygių gylio paieškos medyje).

4.3. Veikimo greitis

6 lentelė. Vidutinių veikimo greičių palyginimas žaidžiant tarpusavyje. Langelio reikšmės pirma vertė: vid. laikas (sek.) žaidžiant už baltus, antra: vid. laikas (sek.) žaidžiant už juodus. Žaidžiama 20 partijų ir apskaičiuojamas vidutinis partijos laikas, visų algoritmų paieškos medžio gylis = 3. Vidurkis neįtraukia žaidimo prieš save

Algoritmas	Klasikinis	Ėjimas tada strėlė	Dviejų Iteracijų	Vidurkis
Klasikinis	413,30	154,22; 152,63	129,20; 95,49	132,88
Ėjimas tada strėlė	152,63; 154,22	0,57	0,55; 0,53	76,98
Dviejų Iteracijų	95,49; 129,20	0,53; 0.55	0,47	56,30

6 lentelėje pavaizduoti Skirtingų alfa-beta algoritmo versijų greičiai. Ėjimas tada strėlė ir dviejų iteracijų algoritmai veikai labai panašiai, tuo tarpu klasikinis alfa-beta paieškos algoritmas yra keliasdešimt kartų lėtesnis, kas apsunkina gilesnio paieškos medžio nagrinėjimą.

Konvoliucinio neuroninio tinklo bei heuristinių evaluacijų greičių skirtumas taip pat yra labai didelis. Su 2 iteracijų algoritmu, ieškant 3 paieškos medžio gylyje DI su konvoliucinio neuroninio tinklo įvertinimo funkcija užtrunka iki 800 sekundžių žaidžiant pačiam su savimi, tuo tarpu heuristinės funkcijos - apie 0,5 sekundės. Konvoliucinio neuroninio tinklo įvertinimo funkcijos greitaveiką galima padidinti keletu būdų: Galima patį tinklą kurti naudojant C++ kalbą. Perkėlimui panaudota frugalfy-deep biblioteka nors ir veikia greitai, tačiau vis tiek apytiksliai 2 kartus lėčiau nei Python aplinkoje. Taip pat ši biblioteka nepalaiko vaizdo plokštės naudojimo skaičiavimams atlikti, kas dar sumažina greitaveiką. Taip pat sumažinus sluoksnių ir/ar neuronų skaičių neuroniniame tinkle jis veiktų greičiau ir galimai išlaikytų panašų rezultatą. Šio darbo metu nebuvo atlikta daug tyrimų su mažu neuroniniu tinklu, kadangi buvo siekiama kuo geresnės žaidimo kokybės, tačiau keletas simuliacijų parodė, kad tikslumas ir nuostolis mažinant tinklo dydį mažėja nestipriai.

Rezultatai ir išvados

Rezultatų ir išvadų dalyje išdėstomi pagrindiniai darbo rezultatai (kažkas išanalizuota, kažkas sukurta, kažkas įdiegta), toliau pateikiamos išvados (daromi nagrinėtų problemų sprendimo metodų palyginimai, siūlomos rekomendacijos, akcentuojamos naujovės). Rezultatai ir išvados pateikiami sunumeruotų (gali būti hierarchiniai) sąrašų pavidalu. Darbo rezultatai turi atitikti darbo tikslą.

Rezultatai

1. Sukurta kompiuterinė programa, skirta žaisti Amazonių žaidimą.
2. Sukurtas alfa-beta paieškos algoritmas, žaidžiantis Amazonių žaidimą ir sukurti 2 šio algoritmo patobulinimai bei jie tarpusavyje palyginti tiek greitaveikos tiek pajėgumo atžvilgiu.
3. Visi sukurti alfa-beta paieškos algoritmai palyginti tarpusavyje ir su atsitiktinai žaidžiančiu dirbtiniu intelektu.
4. Sukurtos 2 heuristinės įvertinimo funkcijos - pagrindinė (naudota daugiausia) ir patobulinta, jos palygintos tarpusavyje, nustatyta kad geresnis rezultatas pasiekiamas, kuomet yra vertinama galimų ėjimų kiekis kartu su galimų krypčių kiekių, krypčių kiekiui suteikiant šiek tiek didesnį svorį.
5. Sukurta įvertinimo funkcija panaudojant konvoliucinį neuroninį tinklą.
6. Patyrinėta įvairios konvoliucinio neuroninio tinklo struktūros. Nustatyta, kad yra konvoliuciniam neuroniniam tinklui yra sudėtinga įvertinti žaidimo lentos būseną net ir turint nemažą kiekį treniravimo duomenų, o mažesnis kiekis duomenų iš geresnių žaidimų netgi šiek tiek pagerina rezultatą
7. Konvoliucinio neuroninio tinklo įvertinimo funkcija palyginta su heuristinėmis įvertinimo funkcijomis bei nustatyta, kad heuristinės įvertinimo funkcijos veikia geriau ir greičiau

Išvados

Sukurti gerai veikianti dirbtinį intelektą žaisti amazonių žaidimui yra itin sudėtinga. Naujoko lygio DI žaidėją galima sukurti panaudojant optimizuotą alfa-beta paiešką bei itin paprastą heuristinę funkciją, vertinančią galimų ėjimų ir galimų krypčių skaičių. Tokia įvertinimo funkcija yra labai greita, todėl leidžia nagrinėti gilesnį žaidimo medį, kuris amazonių žaidime yra itin platus.

Tuo tarpu panaudoti konvoliucinį neuroninį tinklą yra kur kas sunkiau. Net ir su dideliu duomenų kiekiu treniruotas neuroninis tinklas žaidžia vos geriau už atsitiktinį žaidėją. Viena iš to priežasčių gali būti žaidimo mechanika - nors strėlės nejuda, amazonės gali labai greitai atsidurti visai kitoje lentos pusėje ir nuspėti kokios situacijos gali būti vėliau iš dabartinės pozicijos yra labai sudėtinga.

Literatūra

[CWB+20] Johannes Czech, Moritz Willig, Alena Beyer, Kristian Kersting ir Johannes Fürnkranz. Learning to play the chess variant crazyhouse above world champion level with deep neural networks and human data. *Frontiers in Artificial Intelligence*, 2020. ISSN: 2624-8212. DOI: 10.3389/frai.2020.00024. URL: https://www.frontiersin.org/article/10.3389/frai.2020.00024.

[HS20] Satoshi Hayakawa ir Taiji Suzuki. On the minimax optimality and superiority of deep neural network learning over sparse parameter spaces. *Neural Networks*, 123:343–361, 2020. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2019.12.014. URL: https://www.sciencedirect.com/science/article/pii/S089360801930406X.

[KH19] Andreas Kaplan ir Michael Haenlein. Siri, siri, in my hand: who's the fairest in the land? on the interpretations, illustrations, and implications of artificial intelligence. *Business Horizons*, 62(1):15–25, 2019. ISSN: 0007-6813. DOI: https://doi.org/10.1016/j.bushor.2018.08.004. URL: http://www.sciencedirect.com/science/article/pii/S0007681318301393.

[Kje01] Tinne Hoff Kjeldsen. John von neumann's conception of the minimax theorem: a journey through different mathematical contexts. *Archive for history of exact sciences*, 56(1):42, 2001.

[M F17] CM M. Fugers. *A data-driven approach for making a quick evaluation function for Amazons*. Magistrinis darbas, Utrecht University, 2017.

[Peg] Ed Pegg. Amazons by argentinian walter zamkauskas. URL: https://www.chessvariants.com/other.dir/amazons.html.

[RN20] S. J. Russell and P. Norvig. *Artificial Intelligence A Modern Approach*. Pearson, 2020. 5.3.1 sk.

[RW17] Waseem Rawat ir Zenghui Wang. Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review. *Neural Computation*, 29(9):2352–2449, 2017-09. ISSN: 0899-7667. DOI: 10.1162/neco_a_00990. eprint: https://direct.mit.edu/neco/article-pdf/29/9/2352/1017965/neco_a_00990.pdf. URL: https://doi.org/10.1162/neco%5C_a%5C_00990.

[Sch09] Jonathan Schaeffer. Didn't samuel solve that game? *One Jump Ahead*, p. 1–11. Springer, 2009.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez ir k.t. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016-01. ISSN: 1476-4687. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[Ste15] Stefan Steinerberger. On the number of positions in chess without promotion. *International Journal of Game Theory*, 44(3):761–767, 2015-08. ISSN: 1432-1270. DOI: 10.1007/s00182-014-0453-7. URL: https://doi.org/10.1007/s00182-014-0453-7.

Priedas nr. 1

Amazonė tada strėlė algoritmo pseudokodas

Algorithm 2 Alfa-beta paieška atskirai ieškant geriausios pozicijos amazonei ir strėlei

```
function AlphaBeta(boardState, depth, alpha, beta, isMaximising)
    if isMaximising then
        if depth = 0 or terminalState then return Evaluate(boardState)
        end if
        maxEvaluation = −∞
        for all legal moves do
            evaluation = AlphaBeta(boardState, depth − 1, alpha, beta, false)
            maxEvaluation = max(maxEvaluation, evaluation)
            alpha = max(alpha, maxEvaluation)
            if alpha ≥ beta then break
            end if
        end for
        for all legal arrows from the move do
            arrowEvaluation = AlphaBeta(boardState, depth − 1, alpha, beta, false)
            maxArrowEvaluation = max(maxEvaluation, evaluation)
            alpha = max(alpha, maxArrowEvaluation)
        end for
        return maxEvaluation
    else
        if depth = 0 or terminalState then return Evaluate(boardState)
        end if
        minEvaluation = +∞
        for all legal moves do
            evaluation = AlphaBeta(boardState, depth − 1, alpha, beta, true)
            minEvaluation = min(minEvaluation, evaluation)
            beta = min(beta, minEvaluation)
            if alpha ≥ beta then break
            end if
        end for
        for all legal arrows from the move do
            arrowWvaluation = AlphaBeta(boardState, depth − 1, alpha, beta, true)
            minArrowEvaluation = min(minEvaluation, evaluation)
            beta = min(beta, minArrowEvaluation)
            if alpha ≥ beta then break
            end if
            if alpha ≥ beta then break
            end if
        end for
        return minEvaluation
    end if
end function
```

Priedas nr. 2

Dviejų iteracijų paieškos algoritmo pseudokodas

Algorithm 3 Į 2 iteracijas išskaidyta klasikinė alfa-beta paieška

```
function AlphaBeta(boardState, depth, alpha, beta, isMaximising)
    if isMaximising then
        if depth = 0 or terminalState then return Evaluate(boardState)
        end if
        maxEvaluation = −∞
        for all legal amazon moves do
            evaluation = AlphaBetaArrow(boardState, depth, alpha, beta, true, move)
            find maxEvaluation, alpha, do pruning
        end for
        return maxEvaluation
    else
        if depth = 0 or terminalState then return Evaluate(boardState)
        end if
        minEvaluation = +∞
        for all legal amazon moves do
            evaluation = AlphaBetaArrow(boardState, depth, alpha, beta, false, move)
            find minEvaluation, beta, do pruning
        end for
        return minEvaluation
    end if
end function
function AlphaBetaArrow(boardState, depth, alpha, beta, isMaximising, move)
    if isMaximising then
        maxArrowEvaluation = −∞
        for all legal arrows from the move do
            evaluation = AlphaBeta(boardState, depth − 1, alpha, beta, false)
            find maxArrowEvaluation, alpha, do pruning
        end for
        return maxArrowEvaluation
    else
        minArrowEvaluation = +∞
        for all legal arrows from the move do
            evaluation = AlphaBeta(boardState, depth − 1, alpha, beta, true)
            find minArrowEvaluation, beta, do pruning
        end for
        return minArrowEvaluation
    end if
end function
```