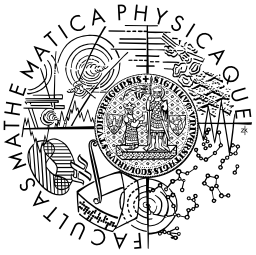


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Pavel Šindelář

Procedurální generování obsahu do tahové strategické hry

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.
Studijní program: Informatika

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Zde bych rád poděkoval RNDr. Martinu Pergelovi, Ph.D. za čas věnovaný konzultacím a za rady, které mi během nich předal.

Název práce: Procedurální generování obsahu do tahové strategické hry

Autor: Pavel Šindelář

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Cílem této práce je vytvoření postupu procedurálního tvoření map do počítačové hry Homeguard. Práce popisuje různé postupy tvoření vnoření pro mapy za použití jejich simulace a vzniklého stromu možných průběhů hry. Vnoření jsou využita k předpovědi uživatelových preferencí z uživatelova vstupu. Tato předpověď je použita k iterativnímu usměrnění generace map pomocí genetického algoritmu. Postup je poté testován na simulovaných uživatelských preferencích a simulovaném vstupu.

Klíčová slova: procedurální generování obsahu, strojové učení, neuronové sítě, genetické algoritmy

Title: Procedural content generation for a turn-based strategy game

Author: Pavel Šindelář

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D., Department of Software and Computer Science Education

Abstract: The goal of this thesis is to create a method for procedural content generation for the Homeguard computer game. The thesis describes different methods for the creation of embeddings for maps using their simulation and the resulting game playthrough tree. The embeddings are used to predict the user's preferences from user input. This prediction is used for iterative guiding of map generation using a genetic algorithm. The procedure is then tested on simulated user preferences and simulated input.

Keywords: procedural content generation, machine learning, neural networks, genetic algoritms

Obsah

1 Úvod

Procedurální tvorba obsahu [1], neboli PCG, je dnes nedílnou součástí vývoje videoher. Umožňuje jednoduše uspokojit moderní nároky na různorodé a detailní úrovně ve hrách. Existuje i mnoho herních žánrů postavených zcela na PCG.

PCG byla už u zrodu videoherního průmyslu. Rogue [2] a první rogue-like hry [3] přinesly ukázku toho, čeho lze dosáhnout touto metodou. V novém tisíciletí se PCG postupy ještě více rozvinuly. PCG se dnes používá v rolích, kde se i jen základní míra automatizace zdála před pár desetiletími nemožná. Například při návrhu herních mechanik [4].

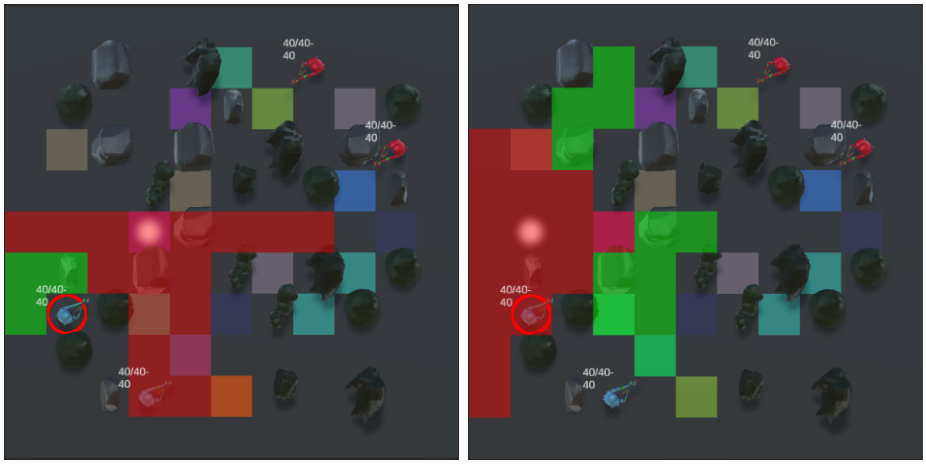
1.1 Cíl práce

Cílem práce je vytvořit postup PCG do hry Homeguard. Hlavními možnostmi pro to, který aspekt hry generovat, byly tvorba map a herní mechaniky. Tvorbou map zde myslíme tvorbu herní plochy, na které se pohybují jednotky. Jako jediný aspekt byla nakonec vybrána tvorba map. Hlavní požadavek na postup bylo, aby měl nízké požadavky na zkušenosti s návrhem map. Tím je myšleno, že jsme při návrhu omezení tím, aby postup vyžadoval pouze povrchní porozumění toho, jak gameplay závisí na pravidlech hry v kombinaci s výběrem mapy. Přes toto omezení bylo třeba stále zajistit, aby postup generoval mapy poskytující kvalitní gameplay a aby nevyžadoval příliš práce od vývojáře nebo od hráče. Požadavek na to, aby byly mapy vyvážené nakonec finální algoritmus přímo neuvažoval, jelikož bral v potaz hodnocení map. Pokud hráči nebo vývojáři vadí to, že má oponent výhodu, nebo to, že má výhodu hráč, může tento fakt vyjádřit ve svém hodnocení map. Práce se zabývá pouze vytvořením a otestováním všech nástrojů pro sestavení konečného postupu. Konečný postup je prezentovaný na ukázkové aplikaci Homeguard, která není součástí práce.

Slovo gameplay je poměrně volně definované. Používá se pro popis aspektů herního zážitku, které souvisí méně s tím, jak hra vypadá, a více s tím, jak se hraje. Kvalitní gameplay, neboli kvalitní zážitek ze hraní hry, je dle autora příliš subjektivní pojem na to, aby jeho používání bylo zcela precizní. V této práci je kvalitním zážitkem myšleno to, abychom uspokojili daného hráče. Problémem generování map s kvalitním gameplayem tedy myslíme to, že je generátor schopný reflektovat hráčovy preference. Autor často zmiňuje svůj názor na gameplay hry, to má ilustrativní charakter pro vysvětlení autorových rozhodnutí.

1.2 Hra Homeguard

Ústředním bodem, i když ne přímou součástí práce, je projekt Homeguard. Autor přemýšlel o začlenění projektu Homeguard do práce, ale přišlo mu to nadbytečné. Základní kámen této hry byl položen už jako finální úkol pro předmět Základy vývoje počítačových her. Druhá část práce na projektu byla vytvořena ve spolupráci s dalším kolegou v rámci zápočtového projektu z Pokročilé programování v jazyce C# (předmět NPRG038) a zároveň Ročníkového projektu (předmět NPRG045).



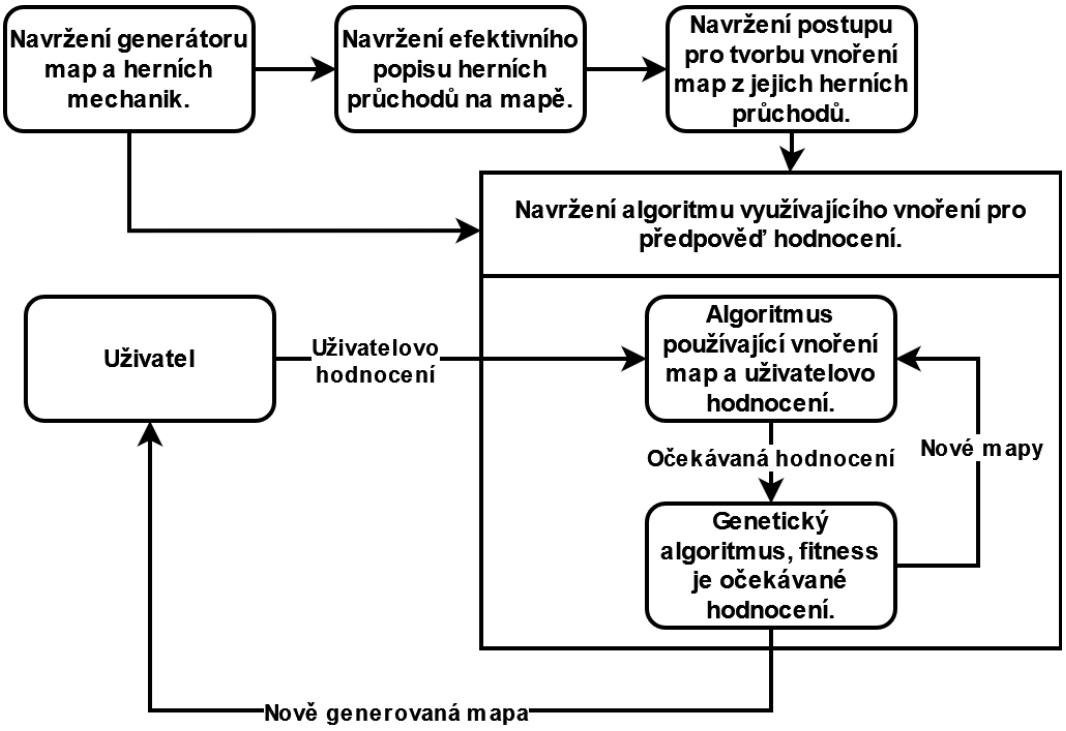
Obrázek 1.1 Ukázka šance vojáka na zásah, pokud by střílel z pozice bílé tečky do zbarvené pozice. Červené zbarvení značí zásah za deset poškození, zelené zásah za pět poškození. Nezbarvená políčka značí pozice, kam voják nemůže střílet.

Hra Homeguard je stále ve velmi raném stádiu vývoje. Úrovně použité pro demonstraci práce používají jen zlomek toho, co původní hra umožňovala, a pravidla hry byla upravena. Hráč hraje skrze tým vojáků. Vždy koná jen voják, co je na tahu. Používá akce, nejčastěji pohyb a nějaký typ střelby, aby zabil dostatečný počet členů nepřátelského týmu. Pravděpodobnost, že voják akcí střelba zasáhne se ve hře odvíjí od pozice nepřítele a krytu. Další akce v původní hře byly například akce léčení sama sebe a akce sprint zlevňující akci pohyb. Pro příklad dalších typů střelby zmíní autor spray, který vždy zasáhl, a headshot, který zasáhl s menší pravděpodobností, ale způsobil dvojnásobné poškození. Finální verze hry byla nakonec upravena pro jednodušší simulování a dle autorových preferencí záživnější gameplay. Každý voják si vybere jedno vedlejší políčko, do kterého se posunout. Poté všichni vojáci najednou vystřelí po vojákovi, kterému by dali největší poškození.

Hráč vybírá tahy svých vojáků, umělá inteligence vojáků nepřátelských. Vojáci se střídají na tahu vždy jednou za kolo v pořadí podle jejich iniciativy. Střelba ubírá body zdraví vojáků, kteří po ztrátě všech čtyřiceti svých bodů zdraví umírají a jsou obnoveni na pozicích obživení. Do hry bylo původně přidáno mnoho provizorních mechanik, ale pro účely prezentace byly odstraněny. Také původně úrovně hry obsahovaly mnoho vojáků na každé straně a vojáky s různorodějšími vlastnostmi, ale pro jednodušší simulování a jednodušší hraní od toho bylo upuštěno.

Voják dostřelí na vzdálenost čtyři a půl políčka od pozice, ze které střílí. Vzdálenost se počítá pomocí klasické euklidovské metriky. Stromy ve hře reprezentují dlouhé kryty, kameny reprezentují krátké kryty. Přes dlouhé kryty nelze střílet. Pokud v cestě nejsou žádné překážky, udělí voják cíli na pozici deset poškození. Pokud nějaký krátký kryt zakrývá cíl, snižuje to poškození, které utrží, na pět. Voják dokáže střílet přes krátké kryty, pokud jsou méně jak dvě políčka daleko od pozice, ze které střílí 1.1.

Pro zjednodušení generace map nakonec autor použil generátor, který umožnil pohyb na pozicích se sudým součtem souřadnic (například, když obě souřadnice byly sudé) a umožnil výskyt překážky na pozicích s lichým součtem souřadnic. Výsledné mapy byly vždy validní a stále poskytovaly velkou různorodost.



Obrázek 1.2 Diagram procesu.

1.3 Přehled

Zde se autor pokusí shrnout výsledný postup, pro vizualizaci lze nahlédnout zde 1.2. Pro dosažení cílů práce používá jednoduché deterministické generátory map, jejichž parametry byly vybírány genetickým algoritmem (GA) [5]. Fitness funkcí GA jsou předpokládaná hodnocení map. Ta je určována pomocí algoritmů založených na předpovídání očekávaného hodnocení nových map podle vnoření map, které uživatel už odehrál a ohodnotil. Vnoření map vývojář vytvoří pomocí algoritmů hlubokého učení na posloupnostech herních průchodů. Průchody jsou generovány na nahodilých mapách z deterministického generátoru. Postup vyžaduje návrh generátoru specifického pro daná pravidla jen tehdy, pokud chce vývojář omezit prostor možných generovaných map. Hlavní nevýhodou je, že postup vyžaduje nemalé znalosti práce s algoritmy hlubokého učení.

Poté, co je finální generátor vytvořen, je ale další práce vývojáře jednoduchá. Interakce vývojáře s finálním generátorem probíhá tak, že hodnotí předložené mapy. Interakce hráče s finálním generátorem probíhá na stejném postupu. Jediné, co musí vývojář udělat, je sdílet svůj soubor preferencí s hráčem. Hráč se poté může rozhodnout nechat si ke svým preferencím přidat preference vývojáře. Pokud chce, aby jeho hodnocení přebilo hodnocení vývojáře, stačí hodnotit mapy velkými hodnotami.

Celý postup pro tvorbu generátoru by šel shrnout následovně. Postup začíná vytvořením algoritmu pro simulování herních průchodů. Pro možnost ořezávání je možné průchody ukládat do stromu hry. Poté jsou průchody každé mapy převedeny na několik matic obsahujících v řádcích reprezentace průchodů, jedna z těchto dvou matic musí obsahovat reprezentace akcí. Tyto matice použijeme jako trénovací data neuronové sítě. Výslednou neuronovou síť poté použijeme pro tvorbu vnoření.

Výsledný generátor pracuje následovně. Spustíme GA, který pro každou vy-

generovanou mapu získá vnoření. Buď přímo průchodem neuronovou sítí, nebo pomocí předem připravených map s vnořeními. Předem připravené mapy jsou použity tak, že pro novou mapu je vybrána předem připravená mapa a použito vnoření předem připravené mapy. Pro novou mapu autor vybírá předem připravenou mapu jako nejbližší v euklidovské vzdálenosti přímo v prostoru reprezentace mapy. Autor se nakonec rozhodl pro vybírání vnoření z předem připravených, jako méně výpočetně náročné vnoření. GA vždy po nějaké době zastavíme, nějaký genom prezentujeme uživateli a vyžádáme od něj hodnocení. Jelikož náš algoritmus používá eliminační algoritmus, umožňuje pouze negativní, nebo neutrální hodnocení. Uživatelův vstup je spolu s dřívějším uživatelovým vstupem použit jako vstup eliminačního algoritmu pro seřazení všech map, co kdy GA vytvořil, vzestupně podle předpovězeného hodnocení. Pořadí v této posloupnosti je poté použito pro určení fitness funkce nových genomů. Pokud by postup fungoval, po nějakém počtu hodnocení by se algoritmus naučil hráčovy preference a hráč by všechny mapy hodnotil jako neutrální. Dále je samozřejmě možné po nějaké době vstup od hráče přestat vyžadovat.

2 Podobné algoritmy

Zde se pokusíme popsat obecné nástroje. Nebudeme se tedy zabývat algoritmy založenými na prohledávání. Ani těmi, založenými na pravidlech, které berou v potaz reálný význam generovaných objektů. Takové algoritmy jsou totiž často velmi specifické pro dané použití. Většina obecných algoritmů procedurální generace obsahu používá náhodu upravovanou jednoduchými postupy. Často se parametry těchto algoritmů nastavují nahodile. Většina práce vývojářů je poté strávena hledáním parametrů generujících převážně mapy s kýženými vlastnostmi.

Výčet všech postupů, které byly pro PCG navrženy, by byl nad rámec práce. Zde se tedy zabýváme metodami, se kterými se sám autor nejvíce setkal, nebo mu přišly, jako blízké jeho postupu. Mnoho postupů PCG používá generátor šumu se spojitými gradienty. Často jsou použity algoritmy simplex noise, perlin noise a algoritmus diamond-square. Dále lze generovat mapy aplikováním pravidel celulárních automatů, nebo replikovat lokální útvary ze zadaného obrázku pomocí Wave Function Collapse. Některé postupy pro generaci přímo využívají strojové učení na prostoru map. Takové postupy zpravidla vyžadují velké množství předem vytvořených kvalitních map. Náš postup není první, který funguje jako interaktivní generátor procedurálního obsahu, ale takové postupy jsou v menšině.

2.1 Wave Function Collapse

Zajímavý postup, jak generovat z vývojářova vstupu, je popsán ve [6], ukázku lze nalézt zde 2.1c. Máme danou množinu $N \times N$ útvarů a texturu s nezaplněnými pozicemi. Vezměme všechna správná nanesení útvaru na cílovou texturu. Správné v tom významu, že toto nanesení nekoliduje se zaplněnými pozicemi.

Počet různých útvarů použitelných na stejné místo nazvěme velikost superpozice. Aplikujeme na texturu vždy nějaké nanesení, které má nejmenší velikost superpozice, pokračujeme, dokud nedojdou možná nanesení útvarů. Nakonec dostaneme buď texturu, která nelze těmito útvary dokončit, nebo texturu, která obsahuje pouze dané útvary. Často se pro zjednodušení uživatelského rozhraní zadává textura a útvary se samplují z lokálních oken textury.

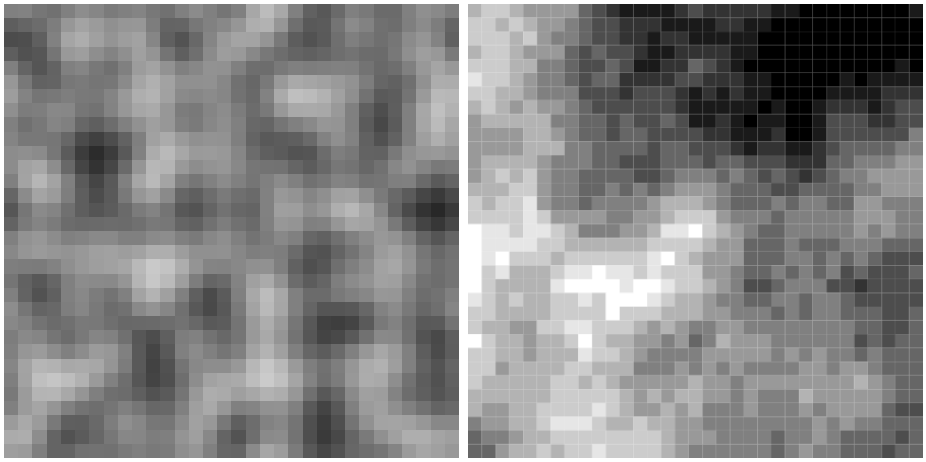
2.2 Algoritmy šumu

Perlin noise [7] je postup, jak generovat šum se spojitými derivacemi 2.1a. Článek definuje funkci noise $R^3 \to R$ ve dvou krocích.

1. Každé trojici (x,y,z) ze Z^3 přiřadíme funkcí $H : Z^3 \to R^4$ posloupnost čtyř na sobě nezávislých reálných hodnot $(a,b,c,d) = H((x,y,z))$. Čísla (a,b,e,d) definují lineární rovnici s gradientem (a,b,c) a hodnotou d v (x,y,z). H je nejlepší implementovat jako hešovací funkci.

2. Pokud (x,y,z) náleží Z^3, definujeme

$$\text{noise}((x,y,z)) = H((x,y,z))_4$$



(a) Ukázka výstupu algoritmu Perlin noise

(b) Ukázka výstupu algoritmu diamond square

(c) Ukázka výstupu algoritmu WFC

Obrázek 2.1 Ukázky výstupů několika zmíněných algoritmů.

Pokud (x,y,z) nenáleží Z^3 spočítáme hladkou interpolaci mezi koeficienty ze Z^3, například interpolaci pomocí kubického polynomu. Nejdříve aplikujeme interpolaci v x (na hranách), poté v y (na stěnách ve směru z), poté v z. Nakonec vyhodnotíme tuto interpolovanou lineární rovnici v bodě (x,y,z).

V moderních implementacích se často pokud (x,y,z) náleží Z^3, definuje

$$\text{noise}((x,y,z)) = 0,$$

zjednoduší se tak implementace.

Funkci jde rozšířit / redukovat na reálný prostor o libovolném počtu dimenzí. Nutná výpočetní síla ale roste exponenciálně s počtem dimenzí. Simplex noise má podobné vlastnosti jako Perlin noise, ale požaduje menší výpočetní sílu ve více dimenzích.

Podobné, ale hrubší výsledky má algoritmus Diamond-square. Diamond-square byl popsán v Computer rendering of stochastic models [8]. Ukázka výstupu z algoritmu v obrázku 2.1b.

2.3 Celulární automaty

Celulární automaty berou každou pozici v matici jako automat, který v každé iteraci podle hodnot sousedních pozic v minulé iteraci změní podle pravidel hodnotu dané pozice. Můžeme například pro každou pozici a její okolí vzít pravidla:

1. hodnota pozice, která má v okolí méně než T sousedů s hodnotou 1, je nahrazena 0,
2. hodnota pozice, která má v okolí alespoň T sousedů s hodnotou 1, je nahrazena 1.

Článek [9] ukázal, že takovými automaty (například $T = 5$ a okolí pozice je osm sousedních pozic) můžeme vytvářet přirozeně vypadající jeskynní prostory.

2.4 Přístupy používající postupy umělé inteligence na prostoru reprezentace map

N-gramy

Článek [10] používá metodu n-gramů, jednoduchý postup pracující s poslupností znaků. Pravděpodobnost dalšího znaku určíme z učící posloupnosti. Pravděpodobnost počítáme na principu, že vezmeme i-tice znaků, i od dvou do daného n, a spočítáme, kolikrát se i-tice, která by vznikla přidáním písmena vyskytuje v učící posloupnosti. Celý postup tedy spočívá v získání několika učících úrovní, převedení je na posloupnosti znaků a provedení předpovědi z nějaké náhodné nebo pevně dané počáteční posloupnosti symbolů pomocí n-gramů z učících úrovní. Tato metoda je schopná generovat stylově podobné mapy.

Markov chain Monte Carlo tree search

Článek [11] používá Markov chain Monte Carlo tree search pro nalezení úrovní. Výsledek akce v Monte Carlo tree search je stav odpovídající přidání vertikálního pruhu k současnému stavu. Skóre je poté hratelnost výsledné úrovně. Metoda je schopná generovat validní úrovně do *Super Mario Bros.* různých obtížností.

Bootstrapující generativní adversariální sítě

Článek [12] používá bootstrapující generativní adversariální síť. Bootstrapující je protože do trénovací množiny přidáváme vygenerované úrovně, které jsou hratelné. Pro usměrnění generace používá postup vektor počtů barev v generované mapě. Například barva odpovídající hráčově postavě se v hratelných úrovních vyskytuje jen jednou.

Porovnání s přístupy, které používají postupy umělé inteligence na prostoru reprezentace map

Náš přístup se nejvíce liší od těchto přístupů v tom, že není end to end. Většina z těchto přístupů od vývojáře bere jako vstup ukázkové mapy, nebo popis pravidel

a poté přímo generuje mapy. Náš přístup požaduje pouze ukázkové průběhy hrou pro danou mapu a umožňuje pouze předpovídat preference podle daných preferencí.

2.5 Přístupy používající uživatelovu interakci

Evoluce přizpůsobující se hráčově chování během hry

Hra Galactic Arms Race [13] generuje zbraně tak, že spustí na začátku hry GA pro procedurální generování zbraní. Iniciální populace je pevně daná. Selekce probíhá podle toho, jak jsou dané zbraně populární a potomci (výsledné zbraně) jsou vygenerovány do světa. Například zbraň, kterou nikdo nikdy nesebere, nikdy nebude mít žádné variace. Naproti tomu zbraň, pro kterou hráč sebere několik variací, bude mít mnoho variací. Pro zachování diverzity jsou generovány do světa i určité vývojářem vytvořené zbraně.

Evoluce v latentním prostoru generativní adversariální sítě

GA v [14] pracuje s latentním prostorem generativní adversariální sítě. To mu umožňuje jednoduše jedince převést do obrazové podoby a předložit hráči. Jelikož genetický algoritmus pracuje v latentním prostoru, umožňuje, aby se při evoluci projevila významová podobnost.

Porovnání s přístupy, které používají uživatelovu interakci

Náš přístup se nejvíce liší od přístupů, které přímo běží ve hře [13] v tom, že nevyžaduje pro generování velké množství odehraného času. Od přístupu v [14] se liší tím, že náš latentní prostor pracuje s herními průběhy, a tak přímo bere v potaz pravidla hry. Další rozdíl je, že autorův algoritmus integruje uživatelův vstup do GA pomocí fitness funkce a ne výběrové funkce. V budoucích pracích by se dalo výběrové křížení v autorem navrženém postupu také vyzkoušet.

2.6 Ludi

Ludi General Game Player, šlo by přeložit jako obecný hráč her Ludi, je systém pro popisování a zkoumání kombinatorických her. To jsou hry které jsou zároveň diskrétní, konečné, deterministické a s úplnou informací. Tento systém byl za pomoci GA použit v [4] pro generování her. GA pracoval s prostorem her, které mohou vzniknout kombinováním několika klasických deskových her (šachy, go, atd.). Jako fitness použil statistiky z průchodů hry vygenerovaných Ludi General Game Player. Z výsledku generace byly vybrány dvě hry Yavalath a Pentalath. Narozdíl od předchozích zmíněných algoritmů procedurální generace obsahu pracuje Ludi pouze s popisem pravidel a nevyžaduje další znalosti o hře. Strategie pro danou hru určuje přímo z pravidel pomocí GA. Absence podobně obecných obecně užívaných metod byla hlavní inspirace pro vytvoření práce.

2.7 Algoritmy pro hraní her

Mnohdy se snažíme hru a její strategie prozkoumat, jen protože ji chceme hrát úspěšněji. Algoritmů pro hraní her specifických pro dané hry nebo žánry her existuje nespočet. V posledních desetiletích bylo ale navrženo několik postupů, které vyžadují jen znalost pravidel. Například algoritmus AlphaZero [15] používá Monte Carlo tree search s temporal difference učením na herních deskách stavů hry. AlphaZero dosahuje nadlidských výsledků pro go, šachy a šógi. Tento postup tedy pravděpodobně nějakým způsobem dosahuje pochopení herní strategie.

3 Použitá technologie

K vytvoření postupu autor použil velké množství cizích algoritmů a nástrojů. V této kapitole autor zmíní ty, které se do nějaké míry vyskytují ve finální verzi aplikace a vysvětlí, proč je použil.

3.1 Proč GA

GA [5] se často používají pro hledání v prostoru parametrů namísto náhodného vzorkování. Obzvlášť, pokud je tento prostor příliš velký a vyhodnocování parametrů příliš pomalé. GA zohledňují ve výběru dalšího vektoru parametrů k prozkoumání i to, jaké hodnocení měly podobné vektory parametrů. Používají simulování přírodního procesu evoluce pro efektivní balancování různorodosti zkoumaných parametrů a očekávaného hodnocení parametrů. GA mají tu nevýhodu, že nelze pro jejich běh přímo použít předem připravené hodnoty. Také mají tendenci produkovat málo variabilní výsledky. Oba tyto problémy autor do nějaké míry ošetřil. První problém vyřešil použitím hodnot nejbližšího prvku z předem připravené množiny. Druhý problém z části vyřešil tím, že použil algoritmus CVT-MAPElites [16]. Nakonec se ale přesto rozhodl nabídnout i variantu postupu bez GA, která vybírá mapy pouze z předem připravených.

3.2 Proč strojové učení

Vytváření nástrojů s pomocí strojového učení [17] na dobré datové množině se ukázalo v mnoha disciplínách produktivnější než snaha vytvářet nástroje pomocí klasických algoritmů. Nejvíce se takto posunuly pochopení a generace řeči, obrazu a videa [18] [19]. Generování herního obsahu pro obecné hry, s ohledem na gameplay se zdálo jako dobrá disciplína, ve které strojové učení zkusit. Lze generovat značné množství dat a neexistují žádné dominantní postupy používající klasické algoritmy.

3.3 Minimax

Autor v částech práce používá algoritmus minimax a termín hodnota minimaxu. Článek [20], překlad článku [21], ukazuje, že lze pro hru dvou hráčů s nulovým součtem vždy určit strategii, která bude mít nejvyšší skóre v nejhorším případě. Výrazem skóre v nejhorším případě myslíme skóre minimální přes všechny možné strategie oponenta. Této hodnotě říkáme minimax. K nalezení této hodnoty použijeme skóre stavů hry uložené ve vrcholech stromu. Hrany vedou mezi vrcholy tak, aby větve stromu reprezentovaly všechny možné průběhy hry. Minimax lze nalézt tak, že vždy pro vrchol ve stromu hry nejdříve spočítáme minimax hry v každém podstromu. Poté vezmeme buď maximum, nebo minimum těchto hodnot v závislosti na tom, o jaký jde tah. Jelikož minimax listu je skóre, v listu lze takovým rekurzivním algoritmem minimax jednoduše spočítat.

3.4 Nástroje

Herní engine

Pro ukázkovou aplikaci bylo třeba vybrat jak zobrazit průběh hry. Projekt Homeguard lze jednoduše upravit pro to, aby byl takovou aplikací. Proto se autor rozhodl použít engine Unity [1] ve kterém byl projekt Homeguard napsán. Alternativně by byla možnost použít Unreal Engine [2] nebo Godot [3]. Godot by měl tu výhodu, že je open-source a má méně pro projekt nadbytečných funkcí.

Ukázková aplikace Homeguard je psaná v UnityEngine 2022.3.5f1 a jedné starší verzi. Verze byla upgradovaná hlavně z důvodu budoucí kompatibility a z pohledu projektu se neliší. Osobní Unity Engine licence kryje komerční použití až do $100,000 v zisku a financování (dosavadní financování a zisk projektu Homeguard je $0). Unity používá jako skriptovací jazyk vlastní verzi C# podobnou C# 9.0 a používá verzi kompilátoru Roslyn. Umožňuje použití knihoven .NET Standard 2.1.

Výkonný jazyk

Dále bylo třeba vybrat jazyk pro implementování algoritmů simulace a GA. Od takového jazyka vyžadujeme výkonnost, kvůli nutnosti procházet velké množství map. Jako výkonné jazyky se nabízí C a C++, méně výkonné ale stále dostatečné by mohly být například Java a C#. Jelikož autorovi přijde C# jako o mnoho pohodlnější na programování než C a C++ a má s ním nejvíce zkušeností, rozhodl se pro něj. Dále byla možnost zkombinovat nějaký vysokoúrovňový jazyk s C++ pomocí nativních knihoven, to ale komplikuje proces odstraňování bugů. Teoreticky by simulaci šlo vyhodnotit i pomocí technologie CUDA a využít tak GPU. To by mohlo běh mnohonásobně zrychlit. Nakonec se ale ukázalo použití nativních knihoven a CUDA jako nepotřebné. Použitím nějaké výkonnější alternativy než je C# by se pravděpodobně dalo algoritmus zrychlit. Jelikož jiné části postupu vyžadují poměrně velké množství výpočetní síly (v rámci jednotek hodin), malé změny v rychlosti běhu této části nejsou pro uživatele tak znatelné.

Jazyk pro strojové učení

Je mnoho jazyků, pro které jsou nabízeny knihovny strojového učení. Nejvíce takových knihoven existuje pro C, R, Python. Do menší míry existují pro mnoho dalších jazyků. Autor má nejvíce zkušeností s Pythonem a C# už používá ve dvou dalších součástech. Proto se autor rozhodl pro jednodušší algoritmy strojového učení použít jejich implementace v C#.

Pro složitější algoritmy strojového učení a obzvláště hluboké učení autor použil Python. Jeho knihovna Pytorch je velmi široce používaná a měl s ní ze studia nejvíce zkušeností. Jako alternativu by šlo použít například knihovnu Tensorflow. Autor se příliš nezabýval tím, které knihovny jsou v jakých aspektech rychlejší, jelikož Pytorch se ukázal jako dostatečně rychlý. Autor se rozhodl pro Python 3.8 z Anaconda [4] kvůli jednoduchosti instalace. Také používal verzi Pythonu ze

[1] `https://unity.com/`
[2] `https://www.unrealengine.com/`
[3] `https://godotengine.org/`
[4] `https://anaconda.org/`

služby [5]. Tato služba nabízí určité množství bezplatné výpočetní síly.

3.5 Knihovny

Knihovny pro C#

Pro strojové učení byly vyzkoušeny dvě knihovny.

Accord.NET [6] — knihovna poskytující základní implementace mnoha statistických a AI algoritmů.

ML.NET [7]— Microsoft knihovna poskytující implementace mnoha AI algoritmů.

Rozdíl v rychlosti většiny různě implementovaných algoritmů se nezdál významný.

Knihovny pro Python

Scikit-learn [22] je nejoblíbenější knihovna na strojové učení v Pythonu a je to knihovna se kterou má autor nejvíce zkušeností. PyTorch [23] použil autor ze stejných důvodů, šlo by místo něj použít například Tensorflow nebo mnoho jiných knihoven. Numpy [24] byl použit, protože je to velmi dobře optimalizovaná knihovna pro numerické algoritmy, ale také proto, že je propojený s API PyTorch a Scikit-learn.

PyTorch — Python knihovna poskytující středně, až vysoce abstrahované rozhraní ke tvorbě a učení neuronových sítí.

Numpy — Python knihovna umožňující jednoduše a efektivně provádět výpočty s poli.

Scikit-learn — Python knihovna nabízející jednotné vysoce abstraktní rozhraní k velkému množství algoritmů strojového učení.

Matplotlib [25] — Python knihovna pro vytváření grafů. Všechny spojnicové a bodové grafy v práci jsou vytvořené v této knihovně.

Scipy [26] — Python knihovna, kterou používáme pro intervalové odhady.

[5] `https://colab.research.google.com/`

[6] `http://accord-framework.net/`

[7] `https://dotnet.microsoft.com/en-us/apps/machinelearning-ai/ml-dotnet`

4 GA využívající vnoření

Jádro práce, které od zadání zůstalo nezměněné, je GA. Při výběru specifické implementace a fitness funkce ale proběhlo dost práce. Zde důkladněji popíšeme koncepty finálního algoritmu.

4.1 Generování map pomocí GA

Na GA [5] lze pohlížet jako na způsob prohledávání prostoru. Algoritmus hledá prvky maximalizující ziskovou funkci, takzvanou fitness. Při výběru dalšího zkoušeného prvku použije algoritmus znalosti o již vyzkoušených prvcích. Algoritmus využívá simulaci přírodního procesu pro to, aby dosáhl velké diversity a zároveň nekontroloval zbytečně mnoho prvků s malou fitness. Používá koncept generace, totiž prvků uložených během běhu. Při tvoření první generace bereme prvky z množiny náhodně. Vytvoření každé další generace provádíme tak, že vybereme prvky z předchozí generace. Z dvojic takto vybraných prvků vytváříme nové pomocí operace křížení. Nakonec každého jedince můžeme upravit operací mutace.

Pro generování map za pomoci GA se naskýtají dva způsoby:

1. Budeme jen upravovat parametry nebo vstup nedeterministického algoritmu a budeme ohodnocovat distribuci map, kterou vytvoří.

2. Vytvoříme deterministický algoritmus, kde vstupní vektor velmi úzce souvisí s výslednou mapou.

Autor se rozhodl použít druhou možnost.

4.2 Deterministické generátory map

LevelGeneratorCheckers

Generátor LevelGeneratorCheckers používá fakt, že lze i složité mapy aproximovat mapami, kde na pozicích s lichým součtem souřadnic jsou kryty a na pozicích se sudým součtem souřadnic se lze pohybovat. Tento algoritmus tedy vektor jednoduše převede na mapu tak, že vezme pozice, kde může být kryt a ty zaplní hodnotami z vektoru zleva doprava, odshora dolů.

LevelGeneratorLines

Generátor LevelGeneratorLines tvoří mapy tak, že vybere mříž pozic dělících plochu na stejně velké čtverce. Pozice na mříži, jsou jediné, které mohou obsahovat kryty. Vektor je nejdříve převeden na dvě mapy. Každá popisuje buď horizontální, nebo vertikální příčky mříže. Pro získání výsledné mapy stačí vzít mapy a otisknout je na mříž.

Portály

Oba tyto algoritmy ještě lze použít v kombinaci s portály. Ty pro pohyb propojují pozice, jako kdyby byly vedle sebe. Jsou generovány v daném počtu

```
procedure CVTMAPElites(k)
    C ← CVT(k)
    (χ, ρ) ← PrázdnýArchiv(k)
    for i ≤ I do
        x ← NáhodnýGenom(k)
        PřidejDoArchivu(x, χ, ρ, C)
    return (χ, ρ)
procedure PřidejDoArchivu(x, χ, ρ, C)
    (p, b) ← VyhodnotFitnessFunkci(x)
    c ← ČísloNejbližšíhoCentroidu(b, C)
    if ρ(c) = null or ρ(c) < p then
        ρ(c) ← p
        χ(c) ← x
```

Obrázek 4.1 Pseudokód CVTMAPElites

a odděleně od mapy. Pro jednoduchost ukládání je vždy jedna pozice propojena s nejvýše jednou další pozicí.

Ve finální verzi byl použit pouze generátor LevelGeneratorCheckers. Bylo by samozřejmě možné vymyslet další zjednodušené generátory, ale LevelGeneratorCheckers fungoval dle autora dostatečně dobře.

4.3 CVT-MAP-Elites

Autor definici algoritmu CVT-MAP-Elites přebírá ze článku [16] a sám algoritmus implementuje. Autor si vybral algoritmus CVT-MAP-Elites, jelikož díky svému konceptů evolučních specializací často oproti ostatním GA dosahuje více různorodých populací. CVT-MAP-Elites pracuje s prostorem možných genomů A. Každému genomu z A přiřazuje funkcí $V(x)$ prvek z množiny M. Množina M musí být konvexní podmnožinou R^n pro nějaké n. Množinu M dělí na disjunktní množinu specializací X pomocí centroidové Voroného tesselace (CVT). Centroidová Voroného tesselace dělí prostor na množiny pomocí posloupnosti centroidů C. Vždy vektor patří do množiny dělení, která náleží k prvnímu nejbližšímu vektoru z C. Posloupnost C je vytvořena clustrováním vzorků vybraných z prostoru. V definici je použita metoda k-průměrů [27], ale teoreticky lze použít libovolnou metodu clustrování.

Genom $x \in A$ spadá do specializace, jejíž je $V(x)$ prvkem. Porovnáme ho s dosavadním genomem s nejvyšší fitness z této specializace y a pokud je x lepší než y, nahradí y. Přesný postup popisuje pseudokód 4.1. V naší implementaci pro zadanou dimenzi n patřily centroidy množině R^n, ale množina $V(x)$ vracela jen hodnoty genomů a genomy mohly nabývat jen celočíselných hodnot.

4.4 Výběr nových genomů

Finální algoritmus jak je to u CVT-MAP-Elites zvykem používá jako selekci čistě náhodný výběr z populace. Genom uložený v každé specializaci už totiž má nejvyšší fitness ze všech genomů, které do dané specializace spadly, a další

selekce většinou zhoršuje diversitu. První algoritmus pro křížení nejdříve vytvořil náhodnou lineární kombinaci dvou genomů z populace a poté výsledek zaokrouhlil po složkách. Poté aplikoval mutaci, která každý gen mutovala s takovou pravděpodobností, aby v průměru mutoval daný počet genů. Nakonec se ukázalo, že lineární kombinace je u genomů, kde geny mohou nabývat jen malého počtu hodnot příliš slabá, a tak bylo místo toho použito překřížení v jednom bodě. Tato operace vybere jeden index a před něj zkopíruje geny z prvního genomu, za něj zkopíruje geny z druhého genomu.

V další zkoušené mutaci je každý gen, který mutujeme, vždy změněn na jiné číslo. Byla to varianta bit-flip mutace, ale pracovala s celým číslem a ne s jeho bity. U standardní uniformní mutace bereme novou hodnotu z celého nosného oboru a tedy můžeme hodnotu nahradit tou stejnou hodnotou, což znamená, že mutace vlastně neproběhla. Nakonec ale tato změna mutace ani na našem oboru hodnot velikosti tři neměla velký vliv.

Autor také přidal podmínku, že pokud se dva porovnávané genomy liší jen v nízkém počtu genů, tak se novější genom s danou pravděpodobností zahodí dříve, než se vypočítá jeho fitness. To dává smysl, jelikož, i když se odvozování map z podobných genomů chová z části chaoticky, stále mají podobné mapy často velmi podobné fitness a vnoření. Tato aproximace je hrubá, ale přinesla zvýšení výkonu.

4.5 Teorie kvality hry

Článek [4] shrnuje několik metrik pro měření různých aspektů gameplaye. Tyto metriky berou posloupnost herních průběhů, u stromu hry tedy jeho větve. Většinu těchto metrik lze omezit na určité důležité větve a nebo je použít jen na určité hloubky. Důležitými větvemi myslíme ty větve, které hráč s velkou pravděpodobností navštíví. Tuto podmínku aproximuje autor tak, že používá větve, které mají ve středu hry větší hodnotu minimaxu než je medián této veličiny.

Proměnné a funkce

- G množina odehraných her.
- $M : G \to N$ $M(g) =$ počet tahů hry g.
- $A : G \times N \to R$ $A(g, m) =$ výhoda hráče 1 nad hráčem 2 ve hře g po tahu číslo m před tahem $m+1$. Positivní A značí to, že hráč 1 vyhrává. Negativní A značí to, že hráč 1 prohrává.
- $E^w : G \times N \to R$ $E^w(g, m) =$ skóre hry g po tahu číslo m před tahem $m+1$ náležící hráči, který nakonec vyhrál hru.
- $E^l : G \times N \to R$ $E^l(g, m) =$ skóre hráče, který nakonec prohrál hru g po tahu číslo m před tahem $m + 1$.

Drama

Tato metrika měří jak jednoduché je se zotavit ze špatné herní situace. Vezměme

$$N = \{E^w(g,m) < E^l(g,m) | m \in [1,M(g)]\}.$$

Poté lze drama počítat jako

$$\sum_{g \in G} \frac{\sum_{m \in N} \left(\sqrt{E^l(g,m) - E^w(g,m)}\right)}{|N|}.$$

Uncertainty

Tato metrika měří jak nelineární je růst skóre výherce. Uncertainty počítáme jako

$$\sum_{g \in G} \frac{\sum_{m \in [1,M(g)]} \left((\frac{E^w(g,m)}{E^w(g,M(g))} - \frac{m}{M(g)})^2 \right)}{M(g)}.$$

Late uncertainty

Tato metrika se počítá stejně jako uncertainty, s tím rozdílem,

Lead change

Tato metrika měří kolikrát se změnil vedoucí.

$$\sum_{g \in G} |\{\text{sign}(A(g,m) \neq \text{sign}(A(g,m+1)) | m \in [1,M(g)-1]\}|$$

Lead possibility change

Tato metrika není použitá v žádném článku, který autor četl, ale dávala mu smysl vyzkoušet. Podobná jako lead change, ale místo počtu změn toho, jestli hráč 1 vyhrává, používá počty změn v tom, jestli hráč 1 může vyhrát. To je aproximováno pomocí minimaxu na větvích stromu hry.

4.6 Popisný vektor

Postup s popisným vektorem využíval toho, že procedurální generaci lze provádět iterativně a interaktivně. Idea je, že zakódujeme strom hry do vektoru a poté použijeme tento vektor k reprezentaci mapy. Hráči poté budou postupně předkládány mapy. Vždy mapu ohodnotí a pokračuje na další.

Hodnotící algoritmus poté bude používat tyto hodnocení pro výběr dalších map k předložení. Důležité je, aby místo průběhů hry, které jsou 2d struktura, hodnotící algoritmus porovnával jen 1d reprezentační vektory. Poté totiž bude o mnoho jednodušší ho navrhnout. Potřebujeme tedy vytvořit množství map, pro každou mapu vytvořit strom hry a pro každý strom vytvořit popisný vektor. Tato data je možné vygenerovat předem a pak uložit.

Další možnost je mapy generovat pomocí GA už podle preferencí hráče. Nevýhoda tohoto postupu je, že poté bude hráč muset čekat na vytvoření dat pokaždé, když změní preference. Podobně jako s mapami lze upravit algoritmus, aby pracoval s libovolnou jinou proměnnou. Například s umělou inteligencí, nebo s portály.

Problém s interaktivní generací je fakt, že hráč pravděpodobně nebude chtít hodnotit tisíce map. Tento postup tedy vyžaduje dobrý hodnotící algoritmus. Hlavní výhodou tohoto postupu oproti fitnes funkcím, které přímo hodnotí, je fakt, že takový algoritmus je poměrně jednoduše testovatelný. Lze ho testovat na různě definovaných množinách špatných map s různě velkými množinami známých hodnocení. Tím poté zjistíme kvalitu popisného vektoru.

4.7 Eliminační algoritmus

Definovat, jaké vlastnosti určují žádoucí mapu je často o dost složitější, než určit nežádoucí vlastnosti. Použijeme toho, že i pokud má mapa jen jednu z velmi nežádoucích vlastností, je třeba jí odstranit. Dále je pro testování dobré místo toho, které mapy odstranit, jen určit, s jakou prioritou je odstranit. Samotné odstranění by poté probíhalo tak, že vybereme dané procento dat, co ponechat. Dále je možné samplovat mapy z rozdělení definovaného podle toho, s jakou prioritou bychom je odstranili.

Bylo vyzkoušeno několik eliminačních algoritmů. Všechny jsou schválně jednoduché, jelikož se snažíme, aby vnoření byla maximálně reprezentativní. Všechny algoritmy pracují s nosnou množinou $S = \{1, 2, \ldots, s\}$ a berou jako vstup prvky označené hráčem jako špatné a matici vzdáleností. Matice vzdáleností M je matice $R^{s,s}$ kde M_{ij} je vzdálenost mezi prvky i a j. Všechny vrací vzestupné uspořádání map podle žádoucnosti (tedy od špatných po nejméně nežádoucí). V každém kroku musíme projít $O(n)$ prvků a kroků bude $O(n)$. Složitost tedy bude $O(n^2)$. Vzdálenosti by šlo ukládat i jinými způsoby. Například držet si jen vzdálenosti od známých špatných pro každý známý špatný zvlášť v haldě. Taková změna by snížila požadovaný čas na $O(n \log n)$. Pravděpodobně by ale měla poměrně velký overhead.

První algoritmus

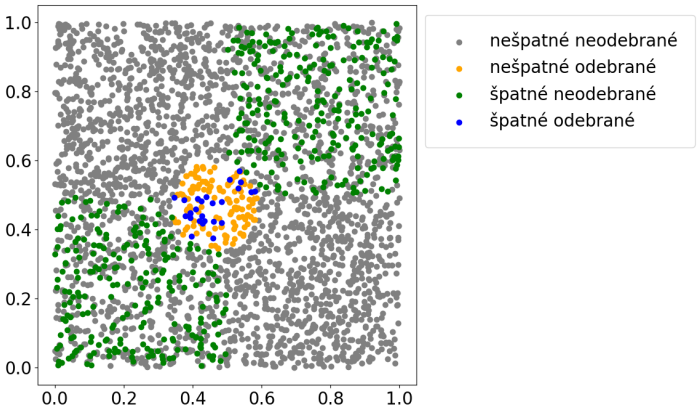
První algoritmus určuje pořadí žádoucnosti vzestupně podle průměrné vzdálenosti od daných nežádoucích prvků 4.2. Nejméně nežádoucí jsou tedy nejvzdálenější prvky od průměru vektorů nežádoucích prvků.

Druhý algoritmus

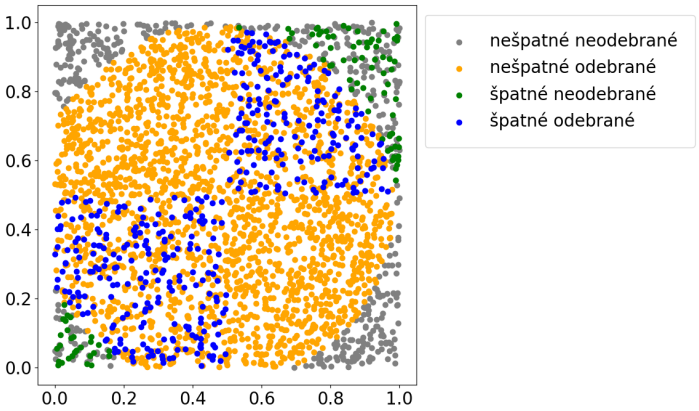
Druhý algoritmus pořadí žádoucnosti určuje pomocí postupného odebírání 4.3. Přeskakuje mezi danými nežádoucími prvky a vždy z neodebraných odebere prvek nejblíže aktuálnímu danému nežádoucímu prvku.

Označme dané nežádoucí jako

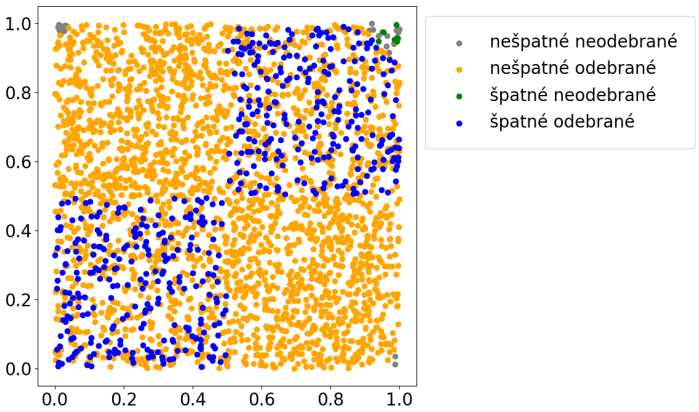
$$k_0, k_1, \ldots, k_p.$$



(a) Stav chvíli po inicializaci.



(b) Většina špatných byla odebraná.



(c) Téměř všechny špatné byly odebrány.

Obrázek 4.2 Eliminace prvním algoritmem. Vstup 20 známých z 200 špatných z 1000 bodů. Špatné pouze v levém dolním a pravém horním kvadrantu.

Množina neodebraných prvků v iteraci 0 je

$$S_0 = S.$$

Poté rekurzivně definujme pro $i \in [0, n]$. Prvek odstraněný v iteraci i bude

$$R_i = \underset{x \in S_i}{\operatorname{argmin}}(M_{x,k_j}).$$

Kde $j = i \mod p$. Množina neodebraných prvků v iteraci $(i + 1)$ bude

$$S_{i+1} = S_i - \{R_i\}.$$

Uspořádání podle předpověděné žádoucnosti poté bude $R_1, R_2, \ldots, R_n$

Třetí algoritmus

Třetí algoritmus upravoval druhý algoritmus. Využíval k tomu funkci softmax.

Definice 1. *Pro daný $v \in R^K$ a $i \in [1,K]$*

$$\operatorname{softmax}(v)_i = \frac{e^{v_i}}{\sum_{i \in [1,K]} e^{v_i}}.$$

Pro matici $M : K \times L$ a $i \in [1,K]$

$$\operatorname{softmax}(M)_{i*} = \operatorname{softmax}(M_{i*}).$$

Neboli je to softmax po řádcích.
Pro daný $v \in R^K$ a $i \in [1,K]$

$$\operatorname{softmin}(v)_i = \operatorname{softmax}(-v)_i$$

Často se také mluví o funkci softmax / softmin s teplotou. Toho se dosahuje vydělením vstupu touto teplotou, tedy softmax s teplotou t by se rovnal softmax $\left(\frac{v}{t}\right)$. Vyšší teplota činní výsledek více rovnoměrný.

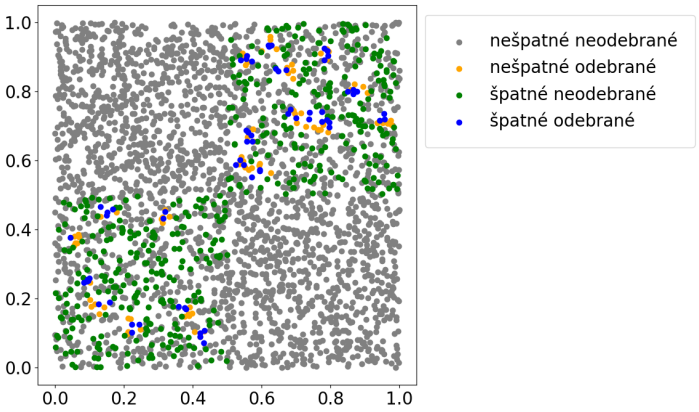
Druhý algoritmus byl upraven tak, aby umožňoval brát v potaz vzdálenost k nejbližšímu neodebranému prvku. V druhém algoritmu byly všechny známé špatné mapy

$$k_0, k_1, \ldots, k_p.$$

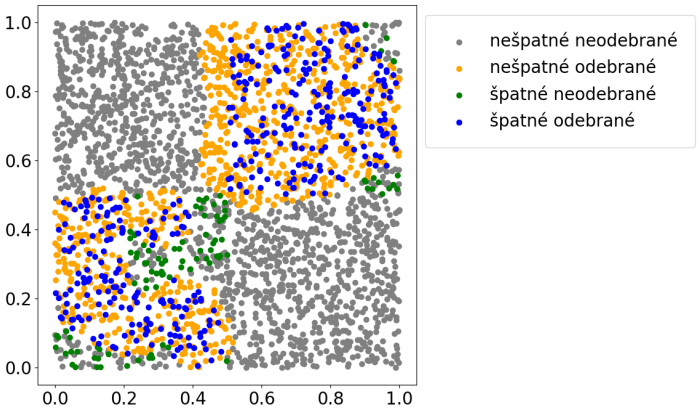
použity rovnoměrně. V iteraci i definujeme vektor $(f_i)_j = \min(\{M_{s,k_j} | s \in S_i\})$. V tomto algoritmu byla vždy v iteraci i vybírána jako instrument eliminace mapa k_j s pravděpodobností $\operatorname{softmin}(f_i/t)_j$. Argument teploty softminu poté určuje, jak moc se toto vybírání liší od rovnoměrného výběru. Jinak byl algoritmus stejný, takže pro vysoké teploty je často ekvivalentní s druhým algoritmem.

Čtvrtý algoritmus

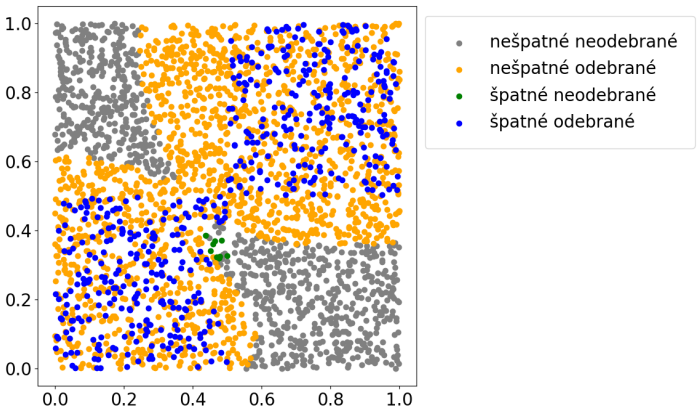
Druhý algoritmus lze brát jako simulaci, kde se v každé iteraci od známého špatného nakazí nejbližší nenakažený. Čtvrtý algoritmus provádí simulaci na podobném principu SIR [28]. Písmeno S v názvu značí množinu susceptible, neboli nakazitelní. Písmeno I v názvu značí množinu infected, neboli nakažení.



(a) Stav chvíli po inicializaci.



(b) Většina špatných byla odebraná.



(c) Téměř všechny špatné byly odebrány.

Obrázek 4.3 Eliminace druhým algoritmem. Vstup 20 známých z 200 špatných z 1000 bodů. Špatné pouze v levém dolním a pravém horním kvadrantu.

Písmeno R značí množinu recovered, neboli uzdravení. Množina infikovaných v první iteraci bude množina známých špatných. V naší simulaci se v iteraci vždy od každého nakaženého nakazí nejbližší nakazitelný. To, že se nakazí, znamená, že se přesune z množiny nakazitelných do množiny nakažených. Po čase potřebném pro zotavení t_r se všichni, kromě známých špatných, nakažení po alespoň tolik iterací, přesunou z množiny nakažených do množiny zotavených. Množina nakažených nebude tedy nikdy prázdná a každý prvek tedy bude alespoň jednou přidán do množiny nakažených. Iteraci, kde daný prvek přidáme do množiny nakažených, nazveme iterací nakažení. Algoritmus skončí, až bude mít každý prvek přiřazenou iteraci nakažení. Výsledné pořadí je určeno vzestupně, podle iterace nakažení.

Relativní hodnocení

Hráč pravděpodobně ocení, když bude mít i možnost nějaké své výběry označit za důležitější. K tomu lze použít relativní hodnocení. Nejjednodušší by bylo jednoduše hodnocení vložit do posloupnosti tolikrát, kolikrát více mají být použitá. Dále by šlo každému výběru přiřadit číslo. Poté v dané iteraci použijeme jen výběry, kde je číslo iterace dělitelné číslem přiřazeným výběru.

Shrnutí

Pro ideální vnoření by měly všechny eliminační algoritmy fungovat stejně dobře, ale ukázalo se, že na neideálních nebo méně dimenzionálních vnořeních je první algoritmus značně horší. I když je třetí algoritmus obecnější než druhý, funguje stejně dobře, jen když ho jeho parametr činí prakticky ekvivalentní. Tedy pouze pokud mu nastavíme velmi vysokou teplotu.

4.8 Propojení GA s eliminačním algoritmem

Eliminační algoritmus neposkytuje žádné přímé hodnocení, jen řadí mapy od nejhorší k nejlepší. Jako fitness vnoření bylo zvoleno procento map, které byly v uspořádání před mapou. Eliminační algoritmus potřebuje mapy označené jako špatné. Ty ze začátku bereme z validační množiny použité při trénování embedderu. Tedy těch špatných podle herních metrik 4.5. Další známé špatné mapy jsou poté přidávány podle volby hráče během iterativní generace.

4.9 Stromy hry

Pro pochopení hry lze použít množinu herních průběhů. Reprezentace herního průběhu obsahuje informace o posloupnosti akcí a posloupnosti herních stavů. Herní průběhy získává autor pomocí stromu hry. Každý vrchol stromu hry reprezentuje herní stav a akci, pomocí které jsme se do tohoto stavu dostali. Pro vytvoření synů vrcholu stromu vezmeme všechny možné hratelné akce a simulujeme jejich dopady. Odstraníme ty, které buď nejsou důležité, nebo jsou reprezentovatelné nějakými jinými akcemi. Zbylé stavy poté ukládáme do stromu a vždy po nějaké době odstraníme větve, které nejsou důležité, nebo je lze reprezentovat jinými větvemi. Takto opakujeme, dokud jsme neodsimulovali požadovanou hru. Často

dává smysl upravit pravidla hry tak, aby hra neměla žádný předčasný konec, protože je užitečné mít strom s listy ve stejné hloubce.

Byly tedy tři hlavní problémy.

- Jakou zvolit metodu výběru zajímavých akcí a kolik ponechat.
- Jak vybrat metodu výběru větví a jaké procento ponechat.
- Do jaké hloubky tvořit strom.

V základu počet různých průběhů hry závisí na hloubce exponenciálně. Často tedy nelze použít všechny průběhy. Specificky pro hru Homeguard, pokud necháme vojáka se pohnout o čtyři pole každé kolo a vystřelit na libovolného ze dvou nepřátel, má na výběr až z $(1 + 4 \cdot 2)^2 \cdot 2 = 162$ možných pohybů. Tento problém, jak bylo zmíněno dříve, byl z části vyřešen upravením pravidel hry. I pokud ale má simulovaný hráč v jednom tahu pro daného vojáka na výběr pouze ze čtyř možností, znamená to, že strom hloubky 20 bude mít 4^{10} vrcholů (polovina tahů patří umělé inteligenci). Tvorba jednoho vrcholu stojí pro jednoduchou hru jen několik stovek instrukcí a uložení vrcholu stojí jen několik bytů, takže takové číslo není samo o sobě problematické. Pokud ale chceme generovat a pracovat se stovkami, nebo tisíci map, tak to může být pro výpočetně slabší stroje problém. Pokud budeme chtít algoritmus použít na složitější hry, může exponenciální růst složitosti být problém i za použití velké výpočetní síly.

Bylo vytvořeno několik postupů, jak vybírat akce. Pro vytvoření reprezentujícího vektoru dané akce autor použil pozici, kam se voják pohne, a pozici, na kterou voják vystřelí. Dále ještě použil vektor čtyř hodnot, každá vždy reprezentovala, jakou velikost má vojákův kryt ze daného směru, buď nahoru, dolů, vlevo, vpravo. Tyto hodnoty byly vyskládány za sebe do vektoru. Byly ponechané jen akce, které reprezentovaly centroidy clustrování na těchto vnořeních. Nakonec byl proces výběru akcí vypuštěn a nahrazen zjednodušením pravidel hry.

Další možnost byla vybírat větve. To zahrnovalo ve stromu vzít listy, převést je do vektorů, provést clustrování a ponechat jen ty větve, jejichž listy zůstaly jako nejbližší nějakému centroidu. Tvorba vektorů vnoření zahrnovala dát životy a pozici každého vojáka za sebe do vektoru. Jako u akcí bylo vyzkoušeno přidat za každého vojáka čtveřici čísel popisujících, jak moc je voják z daných směrů krytý. Zařezávání větví lze provádět po každém tahu, to ale samozřejmě snižuje přesnost stromu více, než pokud zařezáváme méně často. Pro clustrování větví byly ze začátku použity stejné algoritmy jako pro clustrování akcí.

Nakonec pro zařezávání bylo použito náhodné zařezávání větví podle skóre listů. To se nedá interpretovat jako výběr reprezentativní množiny větví. Zařezávání podle skóre listů se ale dá interpretovat jako to, že hráč vybírá akce podle strategické hodnoty.

4.10 Manuálně tvořený popisný vektor

První idea byla využít takzvaný information print jako vnoření. Položka vektoru na pozici i je metrika kvality clustrování skóre na konci hry podle i prvních akcích. Dále bylo vyzkoušeno mnoho dalších méně zajímavých a více experimentálních hodnot.

Použití popisných vektorů, jak bylo popsáno dříve, vyžaduje nějaký algoritmus, který podle množiny ohodnocených vektorů předpoví ohodnocení každého z nehodnocené množiny vektorů. Manuálně tvořené vektory nebyly dostatečně reprezentativní pro použití eliminačního algoritmu, a tak byl místo něj použit algoritmus strojového učení s učitelem.

Jeden způsob byl použít rozhodovací stromy [17]. Zde je výhoda, že není selekce ani redukce potřeba a na rozdíl od support vector machines a neuronových sítí [17] jsou jednoduše vysvětlitelné. Dále byly vyzkoušeny i support vector machines a poměrně mělké neuronové sítě z knihovny sci-kit learn. Na vnořeních měly porovnatelné výsledky jako rozhodovací stromy. Rozhodovací stromy by navíc měly mít výhodu v tom, že mají málo hyperparametrů a nevadí jim, když používáme dlouhé vektory s redundantními dimenzemi, protože sami provádějí selekci dimenzí. To umožnilo vyzkoušení všech možných vlastností herních stromů při vytváření vnoření.

Nakonec se žádným z algoritmů používajících manuální vnoření nepodařilo dostat dostatečné výsledky na rozlišování různě parametrizovaných generátorů. Autor se tedy rozhodl pro generování vnoření přímo ze stromů pomocí strojového učení.

5 Hluboké učení pro tvorbu vnoření

Hluboké učení je jedna z nejméně vysvětlitelných disciplín strojového učení. Autor se použití hlubokého učení pro tvorbu vnoření [29] dlouho zdráhal, ale po neuspokojivých výsledcích ostatních postupů se rozhodl ho vyzkoušet. U hlubokého učení na rozdíl od ostatních algoritmů strojového učení je větší část úsilí soustředěná na hledání hyperparametrů algoritmu a architektury modelu. Úprava dat je samozřejmě stále důležitá, takže množství rozhodnutí, která musí autor udělat, je mnohem větší. Dříve zmíněná malá vysvětlitelnost dále dělá práci složitější.

V této kapitole budeme často zmiňovat pojem softmax 1. Další důležité pojmy jsou ReLU aktivace (ReLU activation), batchování (batching) a batch normalizace (batch normalization). ReLU [30] aktivace je funkce, která se pro kladná čísla rovná identitě a pro záporná rovná nule, lze jí zapsat jako $\mathrm{ReLU}(y) = \max(0, x)$. Je zvykem, že vždy síť provádí operace paralelně na b batchovaných vstupech. Tyto vstupy jsou uloženy do jednoho tensoru a všechny operace jsou prováděny tak, aby jejich efekty odpovídaly nezávislému aplikování operací na vstupy. Batchování pomáhá zlepšit aproximaci gradientu a veličin pro batch normalizaci. Také pomáhá lépe využít architekturu moderních procesorů a grafických karet. Batch normalizace je během trénování funkce, která spočítá průměr a variaci přes hodnoty všech vstupů v batchi, dané jí jako vstup. Poté každou hodnotu každého vstupu pomocí těchto veličin normalizuje, aby měly průměr 0 a standardní odchylku 1. Během předpovídání používáme k normalizaci exponenciální průměr veličin získaných během trénovaní. Batch normalizace pomáhá držet výstupy vrstev ve stejných škálách.

Ve strojovém učení se často dělí množina dat na trénovací množinu, validační množinu a testovací množinu [17]. Trénovací množina je jediná, podle které lze upravit učené parametry modelu. Testovací množina se liší od validační v tom, že výsledky modelu na ní nebereme v potaz při výběru architektury a hyperparametrů modelu. Tím se zachová nezávislost návrhu na konkrétní množině dat. Pro testování autor množinu dat rozdělil na 30 trénovacích a jednu testovací. Testovací množina používala stejný mechanismus jako předtím používala validační množina. Taková množina splňuje vlastnosti testovací množiny. Autor na této specifické množině totiž zkoušel jen jednu dvojici architektury a hyperparametrů.

5.1 Předzpracování dat

Vstup se skládá z posloupnosti n prvků, kde každý má k kanálů. Některé z těchto kanálů musí reprezentovat akce průběhu, aby bylo možné od sebe odlišit různé průběhy s jinak stejnými hodnotami. Autor zkoušel stavy hry reprezentovat na několika úrovních abstrakce. Nakonec se jako nejlepší ukázalo zůstat na nejvíce abstraktní úrovni a ze stavu použít jen index poslední akce a skóre.

Některé použité neuronové sítě nemají schopnost brát v potaz lokální vztahy. Nejjednodušší řešení, které se nabízelo, bylo místo skóre použít nejvyšší dosažitelné skóre z dané pozice přes všechny simulované průběhy. To na rozdíl od skóre přímo reprezentuje strukturu herního stromu. Autor nakonec použil skóre i maximální

skóre. Další řešení by mohlo být clustrovat. To by znamenalo reprezentovat větve rozdílné v několika posledních hloubkách vždy v prvku společného předka. To umožňuje použít kratší posloupnosti a do jisté míry pracovat i s velmi lokálními vztahy. To nakonec nebylo potřeba, jelikož posloupnosti byly dlouhé řádově jen několik tisíc herních průběhů. Nějaké hodnoty byly pro skoro všechny stromy stejné, proto, aby se na ně neplýtvala kapacita sítě, autor hodnoty s nulovou variací odebral při počítání ztráty. Počítat výsledky a nepoužít je ve ztrátě plýtvá výpočetní silou, ale umožňuje to jednodušší přenosnost na jiné množiny dat.

5.2 Architektura sítě

Největší rozhodnutí, které bylo třeba udělat, bylo, jakou páteřní architekturu zvolit. Síť se skládala encoderu, který pro každou posloupnost větví na vstupu vytvoří vektor vnoření, a decoderu, který poskytoval ztrátovou funkci. Architekturou neuronové sítě myslíme, v jakém pořadí provést operace. Těmto operacím se často říká vrstvy a jsou často parametrizovány učenými parametry. Učení sítě je proces upravování těchto učených parametrů vrstev pro minimalizaci ztrátové funkce. Tento proces je automatizovaný za využití gradient descentu, nebo podobných metod [17]. Proto pro možnost učení sítě musí všechny vrstvy být derivovatelné podle vstupu a podle jejich učených parametrů.

Definice 2. *Lineární vrstva z n kanálů do m kanálů pro posloupnost o délce l odpovídá funkci*

$$f : R^{l\times n} \times R^{n\times m} \to R^{l\times m}.$$

$$f(X, M) = MX.$$

Kde X je datová matice, kde sloupce jsou prvky posloupnosti a M váhová matice.

Lineární vrstva z n kanálů do m kanálů pro vektor odpovídá funkci

$$f : R^{n} \times R^{n\times m} \to R^{m}.$$

$$f(v, M) = Mv.$$

Kde v je datový vektor a M váhová matice.

Takovým vrstvám se také říká jednodimenzionální konvoluce s velikostí jádra jedna. Jelikož ale autor klasický princip konvoluce v práci nepoužívá, rozhodl se zavést tuto terminologii. Z kontextu je většinou jasné, jestli myslíme definici pro vektor, nebo pro posloupnost.

Lineární síť s průběžnými globálními vnořeními

První použitá neuronová síť se skládala z několika lineárních vrstev, byla inspirována architekturou ze článku [31]. Mezi lineárními vrstvami používala nelinearitu ReLU a batch normalizaci. Používala globální max pooling, neboli maximální hodnotu přes celý vstup v každém kanálu zvlášť. Takto vytvořenou globální informaci pak připojila ke každému vektoru v posloupnosti, aby mohla brát v potaz globální vlastnosti. Taková síť je invariantní ke všem operacím aplikovaným na posloupnost, ale je schopná brát v potaz pouze globální vlastnosti. Má časovou i paměťovou složitost $O(nk^2)$.

Transformer

Další sítě už používaly Scaled Dot-Product Attention [19], dále jen Attention. Zde pro vstup velikosti $n \times d_k$ lineární vrstvou vygenerujeme tři $n \times d_k$ matice Q,K,V. Výstup se poté spočítá jako

$$Attention(Q,K,V) = \text{softmax}(QK^T d_k^{-1/2})V.$$

Řádky Q reprezentují vyhledávající vektory a řádky K reprezentují klíčové vektory. Hodnota QK_{ij}^T reprezentuje tedy kosinovou podobnost $Q_{i*}aK_{j*}$. Hodnota $\text{softmax}(QK^T)_{ij}$ tedy bude pravděpodobnost, se kterou Q_{i*} náleží ke klíči K_{j*}, a tedy odpovídá pravděpodobnosti vrátit hodnotu V_{j*}. $\sum \text{softmax}(QK^T)_{ij}V_{j*}$ poté dává průměrný výsledek vyhledávání. Násobek matic dělíme $d_k^{1/2}$, protože pro velká d_k jsou hodnoty výsledku v regionu, kde má softmax malé gradienty.

Na rozdíl od lineárních vrstev jsme schopni porovnávat každé dva prvky posloupnosti. Navíc, pokud použijeme pro výrobu matic V a K jinou posloupnost vektorů, než pro výrobu vektorů Q, můžeme porovnávat každý prvek z jedné posloupnost s každým prvkem z druhé posloupnosti.

Asymptotická složitost výpočtu těchto matic je asymptotická složitost algoritmu pro vypočítání maticového násobení typu 1

$$R^{n\times k} \times R^{k\times n} \to R^{n\times n}$$

plus asymptotická složitost algoritmu pro vypočítání maticového násobení typu 2

$$R^{n\times k} \times R^{k\times k} \to R^{n\times k}.$$

Tato složitost záleží na implementaci. Nejčastější implementace pro maticové násobení ve strojovém učení je algoritmus, který pro každou pozici vynásobí sloupcový a řádkový vektor. Takový algoritmus má pro násobení typu 2 asymptotickou časovou složitost $O(nk^2)$ a má paměťovou složitost $O(nk)$. Násobení typu 1 nejčastěji trvá $O(n^2k)$ s paměťovou složitostí $O(n^2)$. Článek [19] doporučuje, aby se místo jedné Attention vrstvy použila Multi-Head Attention vrstva. Multi-Head Attention s d kanály a h hlavami, kde $(d = 0)\%h$, lze zapsat jako

$$MultiHead(Q, K, V) = Concat(head_1, head_2 \dots head_h)W^O,$$

kde

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V).$$

Zde W_i^Q, W_i^K, W_i^V jsou váhové matice $R^{d\times d/h}$ a W^O je váhová matice $R^{d\times d}$. V Multi Head Attention tedy vždy nakonec použijeme dohromady stejné množství vah jako v Attention. Na rozdíl od Attention ale vždy provádíme vyhledávání za použití několika podmnožin kanálů.

Můžeme použít úpravu dat, abychom se vypořádali s kvadratickým růstem složitosti v závislosti na délce posloupnosti. Pokud jeden prvek reprezentuje více větví stromu, lze použít poměrně krátkou posloupnost. Samozřejmě, pokud je poté prvek příliš dlouhý, projeví se druhá mocnina ve složitosti v závislosti na počtu kanálů. Pro dlouhé posloupnosti, neboli context length, je třeba použít řídkých matic a dalších postupů. Naše posloupnosti jsou dosti krátké, aby to nebylo třeba.

5.3 Regularizace a úpravy

Pokud používáme jako model větší síť, než problém vyžaduje, je pravděpodobné, že model bude overfitovaný. Model bude velmi dobrý na trénovacím úkolu na trénovací množině, ale na validační množině nebude tolik užitečný a výsledky jeho vrstev nebudou sloužit jako reprezentativní vnoření. Často se tedy hodí tuto tendenci overfitovat snížit, tomu se říká regularizování.

Nejjednodušší a nejrazantnější způsob regularizování je do ztrátové funkce přidat hodnotu, která závisí přímo na váhách modelu. Nejčastěji použijeme nějakou metriku mezi zploštělými váhovými maticemi a nulou. Pokud použijeme L2 metriku, zabráníme tomu, aby model příliš závisel na malém množství neuronů. Zároveň tím ale omezíme jeho možnost reprezentovat data. Další způsob je takzvaný dropout kanálů. V dané vrstvě modelu náhodně vybereme zvolené procento kanálů a nepoužijeme je pro výpočet.

Dále lze použít toho, že všechny vrstvy neuronových sítí je možné převést na takzvané depth seperable verze. Jednoduše rozdělíme kanály vstupu na bloky a na každý aplikujeme menší kopii vrstvy. Tím se pro $l$násobné rozdělení sníží složitost z $O(nk^2)$ na $O(ln(kl^{-1})^2) = O(nk^2l^{-1})$. Hodnoty v různých depth seperable blocích tedy v této vrstvě neinteragují. To má podobný výsledek, jako dropout. Možnou nevýhodou je, že tyto neinterakce jsou stejné napříč batchemi. Výhodou je, že na rozdíl od dropoutu neplýtváme kanály.

Pro zrychlení trénování je také možné změnit datový typ, na kterém jsou vrstvy založené. Základním datovým typem je `float`, neboli 32bitové číslo s plovoucím znaménkem. Velká část definičního oboru těchto čísel se ale nevyužije. Proto bylo navrženo několik 16bitových, 8bitových i 4bitových alternativ. Tyto alternativy ale často nejsou knihovnami pro strojové učení plně podporovány, nebo v nich způsobují chyby za běhu, proto se je autor rozhodl nepoužít.

Variační autoencoder

Další zkoušenou metodou byla metoda variačních autoencoderů [32]. Předchozí sítě vždy přímo vytvářely vnoření jako výstup neuronové vrstvy. Pokud ale místo toho pouze vygenerujeme vektor průměru μ a vektor odchylky σ, můžeme vektor vnoření táhnout z takto parametrizované normální distribuce. Jednoduše, pokud x je taženo ze standardní normální distribuce, pak náhodná veličina, určená funkcí

$$f(\mu, \sigma) = (x + \mu)\sigma,$$

má normální rozdělení se střední hodnotou μ a odchylku σ. Jelikož funkce f bere x jako konstantu, pro hodnotu taženou z f lze udělat klasickou backpropagaci. Během učení je poté třeba regularizovat μ a σ, aby byly distribuce přibližně standardně normálně rozdělené. Proto se ke ztrátě přičítá výraz

$$-0.5 \sum_{i \in [1,k]} (1 + log(\sigma_i^2) - \mu_i^2 - \sigma_i^2).$$

5.4 Self-suprevised učení

Princip self-suprevised učení je v dnešní době poměrně populární. Nejvíce v počítačovém vidění a zpracování přirozeného jazyka. V počítačovém vidění bylo

vyvinuto několik různých metod. Jednu definuje například [18]. Pro trénování sítě je potřeba ztrátová funkce. Jelikož ale naším cílem je vytvoření vnoření a ne klasifikace, nebo regrese, nemáme žádná označená data. Za tímto účelem je k encoderu přídána druhá část, decoder, která z výsledku a mezivýsledků encoderu a informací o vstupu takovou ztrátovou funkci vypočítá. Díky řetězovému pravidlu lze poté určit gradient pro encoder i pro přidanou část. Úkolům, které decoder řeší se říká pretext tasks. Takovýchto sítí se sdíleným encoderem lze vytvořit několik a můžeme různě volit načasování kvocientu, se kterými použijeme danou ztrátu.

Dokončování stromů

Jeden ze základních self-suprevised učení je doplňování. Vezmeme vstupní posloupnost a odebereme část, poté se tuto část pokusíme ze vstupu předpovědět. Na herních stromech můžeme tento cíl reprezentovat poměrně jednoduše. Jako vstup dodatečné sítě navíc k vnoření herního stromu vezmeme indexační posloupnost vytvořenou z odebraných větví. Každý prvek zde obsahuje pouze indexy tahů v dané větvi. Základní reprezentaci lze dostat tak, že použijeme zdegenerovanou konvoluci na indexační posloupnost. Indexační posloupnost upravíme tak, že ke každému prvku zřetězíme vektor vnoření. Z takové reprezentace jsme potom schopní pomocí dalších vrstev vytvořit předpověď pro odebrané hodnoty pro korespondující odebranou větev a vzít L2 ztrátu mezi předpověděnými hodnotami a hodnotami z větví.

Attention má zde tu výhodu, že umožňuje při dokončování brát v potaz i mezivýsledky. Na rozdíl od lineární vrstvy totiž násobí mezi sebou všechny dvojice ve vstupu a nevyžaduje, aby posloupnosti pozičně korespondovaly ani aby měly stejnou velikost.

V následujících podsekcích často mluví autor o jednoduché lineární síti. Tím je myšlena posloupnost lineárních vrstev s batch normalizací a ReLU mezi nevýstupními vrstvami.

Předpovídání map

Jednoduchá myšlenka, jak požadovat reprezentativnost vnoření stromu hry, je předpovídat z vnoření mapu, na které byl strom hry odehrán. Mapu zploštělou na vektor předpovídáme z vnoření pomocí jednoduché lineární sítě.

Problém můžeme pojmout jako regresi, ale, jelikož se často herní mapy skládají z dlaždic diskrétních kategorií, můžeme ho pojmou i jako klasifikaci. V tom případě je potřeba předpovědět pro každou pozici pravděpodobnost každé dlaždice. To se z výstupu jednoduché lineární sítě dosáhne pomocí funkce softmax 1. Znamená to ovšem klasifikovat do poměrně dlouhého vektoru, což komplikuje učení.

Předpovídání pozice

Stejně jako lze předpovídat pro pozice jejich hodnoty, lze pro hodnoty předpovídat jejich pozice. Na rozdíl od pozic se ale hodnoty mohou opakovat na různých pozicích. Pokud ale místo předpovídání pozice jedné hodnoty předpovídáme pozici nějaké sekce, pravděpodobnost opakovaného výskytu je minimální. V našem případě tedy z posloupnost hodnot sousedství větví předpovídáme vektor akcí, které větve sdílejí.

Ztráta na principu podobném DINO

Tato ztráta je založená na principu předloženém v [18]. Zde si držíme dvě verze sítě, jednu trénujeme normálně a druhou vždy po daném počtu batchů změníme metodou exponenciálního průběžného průměru. Bereme vnoření vytvořené pomocí průměru modelů (učitele) a trénovaného modelu (studenta). Výstupy učitele vycentrujeme pomocí exponenciálního průběžného průměru výstupu učitele v předchozích iterací. Jako vstupy bereme sekce vstupu různých velikostí. U učitele bereme pouze větší části vstupu, umožňujeme mu tedy globálnější pohled. U studenta bereme všechny vstupy. Vyžadujeme tedy po něm, aby také vytvářel reprezentativní vnoření za použití lokálních vlastností.

Vezměme každou dvojici výstupu učitele t_j a výstupu studenta s_i se stejným počtem kanálů K. Pro ty spočítáme křížovou entropii (ta totiž indikuje nepodobnost náhodných veličin) mezi jejich softmaxi 1 s tepotami T_t, T_s, tedy

$$\sum_{i \in [K]} - \operatorname{softmax}\left(\frac{t_j}{T_t}\right) \log\left(\operatorname{softmax}\left(\frac{s_i}{T_s}\right)\right).$$

Jako ztrátu vezmeme aritmetický průměr křížových entropií pro všechny kombinace i a j. Lze normalizovat velikostí poslední vrstvi.

Na tuto hodnotu lze také nahlížet jako na součet Kullbackovy–Leiblerovy divergence (KL) $KL(T||S)$ a entropie $H(T)$, kde T a S jsou rozdělení učitele a studenta po softmaxu. Ve ztrátě na principu DINO to znamená, že během trénování bereme v potaz entropii učitelova vnoření. Ta může být poměrně odlišná datový bod od bodu a iteraci od iterace. Autorova zkušenost ukázala, že je užitečné spočítat křížovou entropii a k ní přičíst stonásobek $KL(T||S)$, jelikož KL často bývá oproti křížové entropii velmi nízká.

5.5 Finální architektura

Nakonec většina předběžných testů ukázala, že použití jedné ztráty oproti použití několika ztrát nemá vliv na sílu předpovídání. To, jakou ztrátu autor použil, také nemělo vliv. Autor se tedy dále bude soustředit na ztrátu z doplňování, jelikož vede k nejrychleji se učícímu algoritmu. Je možné, že pro různé verze trénovacích dat by použité několika ztrát mohlo být užitečné, to by mohlo být testované v nějaké budoucí práci.

Attention blokem je myšlena attention vrstva následovaná batch normalizací a ReLU aktivací. Kde není zmíněno jinak uvažujeme jako vstup attention bloku jednu posloupnost, pomocí které vytvoříme Q, K i V matice. lineárním blokem je myšlena lineární vrstva následovaná batch normalizací a ReLU aktivací. Autor používá vždy attention bloky stejné velikosti. Šlo by velikosti vrstev v encoderu s hloubkou snižovat a v decoderu naopak s hloubkou zvyšovat. Síť je ale poměrně mělká, a tak autorovi nepřišlo toto jako důležité.

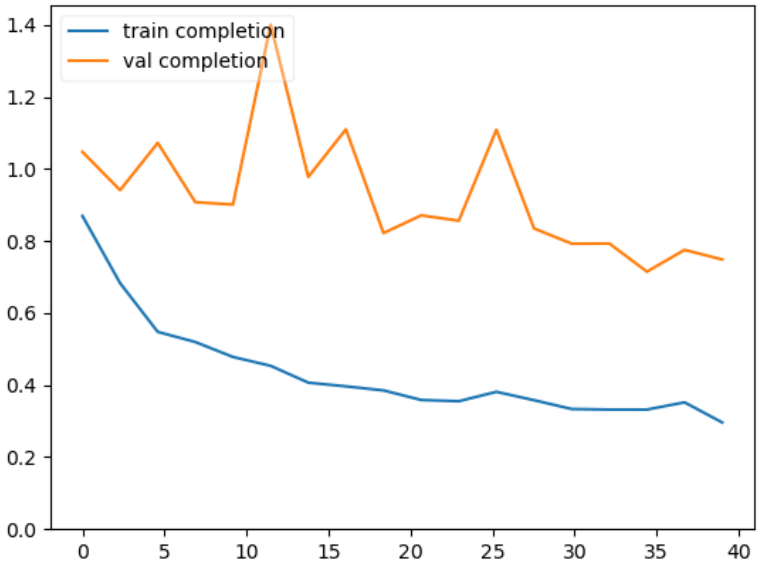
Autor vyzkoušel několik architektur encoderu a hlavy pro doplňovací ztrátu, vždy použil pro obě podobnou architekturu. Architektura založená pouze na lineárních blocích buď overfitovala, nebo underfitovala. Architektura, která byla založená pouze na tří attention blocích, se učila velmi pomalu. Architektura, která používala lineární blok stejné velikosti za každým ze tří attention bloků měla podobný problém. Pravděpodobně proto, že měla příliš mnoho vrstev.

Finální architektura využívá lineárního dvojbloku, dvou lineárních bloků o počtu kanálů (u, v), (v, k). Hodnota v je zde větší než k. S nimi lze dosáhnout toho, že v, sdílený počet kanálů, nezávisí na u a k. Zároveň k způsobuje bottleneck, tedy síť je nucena reprezentaci v kanály převést na reprezentaci méně kanály.

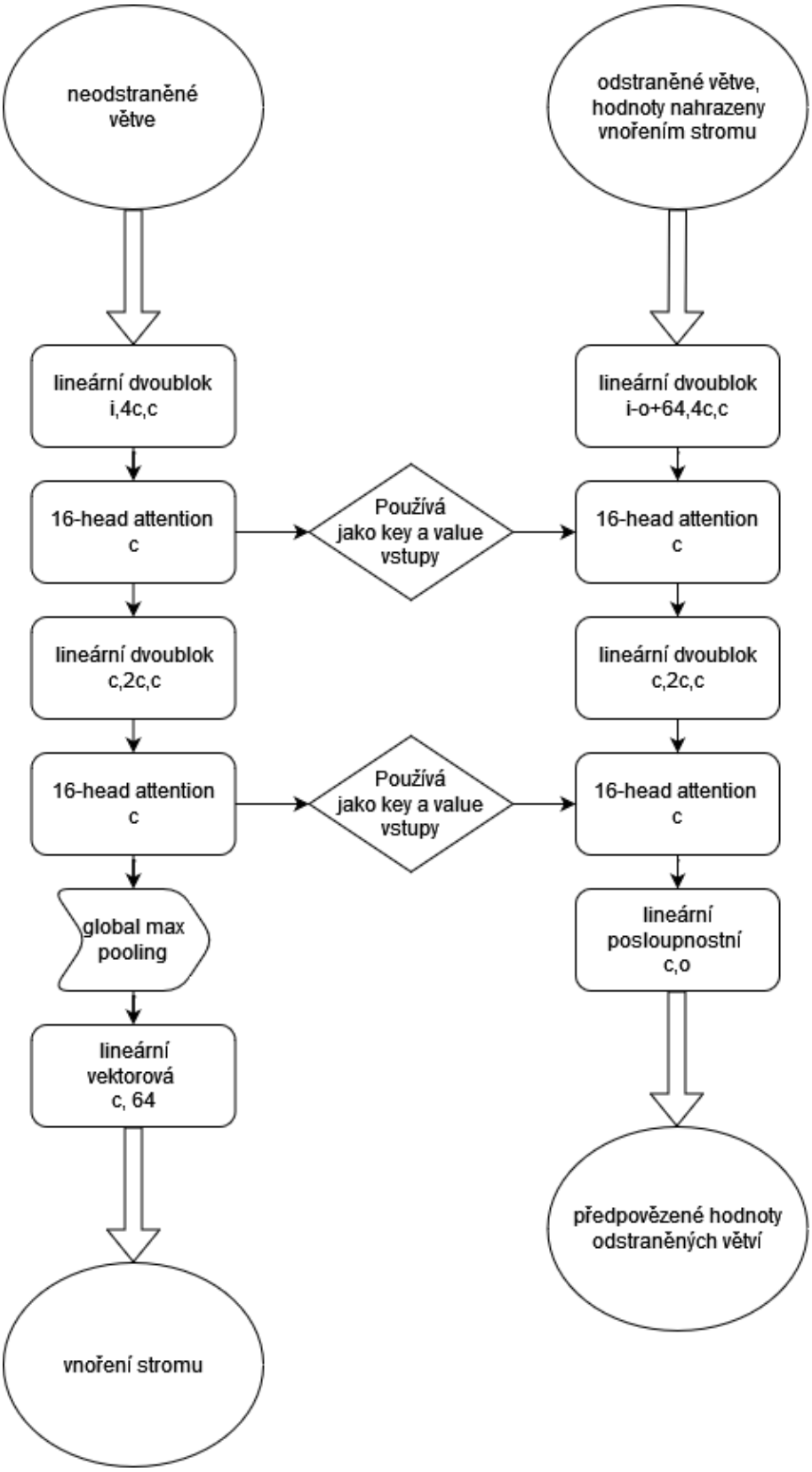
Finální architektura byla parametrizovaná počtem kanálů attention bloků encoderu $c = 128$. Finální verze sítě měla dropout nastavený na nula, místa, kde by byl případně aplikován se lze dočíst v 6.6. Encoder používá na začátku lineární dvojblok ($v = 4c$). Dále aplikuje attention vrstvu (c) a znovu lineární dvojblok ($v = 2c$). Poté aplikuje znovu attention blok (c). Nakonec použije globální max pooling a lineární vrstvu o výstupním počtu kanálů 64 pro vytvoření vektoru vnoření. Zde lineární vrstvou výjimečně myslíme lineární vrstvu pro vektor.

V decoderu používá autor podobnou architekturu jako v decoderu. Používá dvojblok lineárních vrstev na začátku ($v = 4c$). Dále aplikuje attention vrstvu (c). Jako key a value vstupy bere výstup poslední vrstvy encoderu. Následně aplikuje znovu dvojblok lineárních vrstev ($v = 2c$), attention vrstvu (c). Jako key a value vstupy bere výstup první vrstvy encoderu. Nakonec použije lineární vrstvu pro předpověď posloupnosti odstraněných hodnot.

Finální trénování probíhalo pomocí optimizéru Adam s learning rate 0.005 a learning rate decay 0.95. Velikost batchí byla 128. Trénovalo se po 20 epoch. Autor použil doplňovací ztrátu. Jako vstup encoderu autor použil indexy akcí, skóre a maximální dosažitelné skóre. Jako vstup decoderu použil indexy akcí. Jako předpovídaná data použil skóre a maximální dosažitelné skóre. Akce byly onehot encodované pro lepší zpracování, jelikož se s nimi nedá pracovat jako s ordinální veličinou. Průběh trénovaní lze nahlédnout zde 5.1. Z grafu lze vidět, že architektura stále není pro data dokonalá. Je důležité podotknout že, jelikož se jedná o self suprevised učení, jsou pro finální výsledek ztráty méně důležité, než u jiných typů trénování.



Obrázek 5.1 Křivka trénovací a validační chyby jednoho běhu algoritmu. Osa x udává epochu. První validace byla provedena až po několika trénovacích iteracích pro lepší škálování grafu.



Obrázek 5.2 Diagram finální neuronové sítě. Hodnota *i* značí počet kanálů vstupní posloupnosti, hodnota *o* značí počet odstraněných kanálů.

6 Uživatelská dokumentace

Zde bude zmíněna uživatelská dokumentace k python aplikaci, .NET 6.0 aplikaci MakeTrees a ukázkové aplikaci Homeguard.

6.1 Instalace C# aplikace

Soubory *Setup.exe* a *Setup.msi* ve složce IwonLevelGeneratorSetup spustí instalaci aplikace MakeTrees pro operační systém Windows. Instalace do zadané složky nainstaluje všechny tři knihovny a aplikaci MakeTrees. Nevytváří žádné zkratky. Autor se rozhodl uložit instalační soubory bez komprese, aby bylo možné aplikaci spustit, i pokud by instalátor z nějakého důvodu selhal. Aplikaci MakeTrees je totiž možné spustit přímo ze souboru *MakeTrees.exe* ve složce instalátoru. Pro vytvoření nového buildu aplikace MakeTrees lze použít aplikaci Visual Studio 2022 a nebo jinou aplikaci rozeznávající projekty uložené ve formátu sln a proj. Pro vytvoření nového buildu instalátoru lze použít aplikaci Visual Studio 2022 s nainstalovaným rozšířením Microsoft Visual Studio Installer Projects 2022.

6.2 Požadavky pro spuštění Python modulu

Pro spuštění Python modulu je třeba nainstalovat verzi Python 3.11.5 a potřebné balíčky dané souborem *requirements.txt*. Autor instaloval Python a balíčky skrze program Anaconda [1]. Trénovaní je poměrně rychlé i bez CUDA verze knihovny Pytorch. Pokud chce uživatel zkusit, o kolik se trénování zrychlí použitím CUDA, může následovat
`https://pytorch.org/get-started/locally/`. Samozřejmě je také možné využít cloudové služby, které mají Python a většinu knihoven předem připravené. Autor použil službu Google Colab
`https://colab.google/`. Ukázkový soubor `HomeguardLearning.ipynb` je sestaven tak, že ho lze ve službě Google Colab ihned spustit. Možná bude nutné nastavit `data_and_code_path` ve druhé buňce nebo `code_path` ve třetí buňce na hodnotu podle příslušného komentáře. Další dokumentace je přímo v souboru.

6.3 Stažení ukázkové aplikace Homeguard

Build projektu Homeguard pro 64bitové a 32bitové verze operačního systému Windows lze stáhnou na adrese
`https://gitlab.mff.cuni.cz/sindelp3/homeguardbase`. Složku
HomeguardBase obsahující Unity [2] projekt stejného jména lze také stáhnout na stejné adrese. Pro vytvoření
nového buildu projektu Homeguard je třeba otevřít složku *HomeguardBase* v editoru herního enginu Unity verze 2022.3.5f1, nebo jiné kompatibilní verze. Herní engine unity lze získat pomocí aplikace Unity Hub `https://unity.com/unity-hub`.

[1] `https://anaconda.org/`
[2] `https://unity.com/`

Poté je třeba následovat instrukce pro vytvoření buildu dle dokumentace dané verze herního enginu Unity.

6.4 Formáty a konvence

Binární formát iwonformat

Formát je uložen v little endian pořadí. Prvních 12 bytů obsahuje hlavičku. Hlavička obsahuje tři čísla ve formátu `uint32`, ty určují tvar tensoru. Zbylé byty jsou obsah tensoru libovolného typu. Všechny aplikace v projektu očekávají data typu `int16`. Dat by měl být samozřejmě takový počet, jako je produkt čísel v hlavičce.

Textový formát iwonmapformat

První část souboru je libovolný text hlavičky. Výraz `\n` značí znak pro nový řádek. Hlavičku od těla souboru odděluje výraz `----\n`. Informace o jednotlivých mapách jsou oddělené pomocí výrazu `\n\n\n`. Informace o mapě se skládají ze tří částí, ty jsou oddělené výrazem `\n\n`. První část obsahuje obsah mapy. Prázdná pole jsou reprezentována mezerníkem, krátké kryty výrazem 0, dlouhé kryty výrazem 1. Výraz 2 značí počáteční pozici hráčova vojáka, výraz 3 značí počáteční pozici oponentova vojáka. Velká písmena značí dvojice portálů. Mapa je vypsána po řádcích oddělených symbolem `\n`. Druhá část obsahuje několik vektorů vnoření oddělených symbolem `\n`. Čísla vektorů jsou oddělena středníkem. Každé číslo je ve formátu znaménko, čtyři číslice, čárka, čtyři číslice. Poslední část je číslo, které interpretujeme jako index třídy, do které příslušný strom hry patří.

6.5 Aplikace MakeTrees

MakeTrees je konzolová aplikace pro generování map a tvorbu stromů hry pomocí simulace. Je potřeba zajistit, že výstupní soubor je validní k přepsání a máme k němu přístup. Pokud tomu tak není, nebo přestane být, program tuto skutečnost oznámí a dál nepracuje. Někdy je pro získání oprávnění třeba spustit aplikaci v administrátorském režimu. Občas nastane při tvoření stromu hry málo častá chyba, potom program mapu přeskočí a pokračuje další mapou. Po dokončení simulace uloží program výsledky podle prvního argumentu *path*. Větve stromu hry uloží pomocí tří souborů formátu 6.4. Indexi poslední akce uloží jako *path.treeLAInds.iwonformat*. Skóre uloží jako *path.treeScores.iwonformat*. Nejvyšší dosažitelné skóre uloží jako *path.treeMinmaxes.iwonformat*. Mapy uloží ve formátu iwonmapformat do souboru *path*. Po doběhnutí program, pokud je spuštěn v interaktivním režimu, počká na stisknutí klávesy. Parametry příkazového řádku aplikace jsou:

1. Společný název souborů, kam uložit výsledek. Pokud je prázdný řetězec, je použita relativní cesta ve formátu `MM-dd_HH-mm`.

2. Počet map, který vygenerovat. Pro každou mapu je vytvořeno 10 stromů hry 4.9. Je lepší tuto hodnotu nastavit na vyšší hodnotu, než očekáváme, jelikož některé simulace mohou selhat. Musí být vyšší, než dvě.

3. Počet iterací, které proběhnou. Musí být větší, než nula. Hloubka, do které strom hry generovat, je čtyřnásobek tohoto čísla.

4. Počet větví, kolik vybrat pro další iteraci. Pokud je záporný nebo nula použijeme všechny větve.

5. Dropout, tedy zlomek akcí, které v průměru zahodíme. Akce zahazujeme jen v iteracích, ve kterých zahodíme nějaké větve. Například dropout hodnoty `0.99` znamená, že ve valné většině kroků zahodíme všechny akce kromě jedné.

Pokud nejsou podány žádné argumenty aplikace se spustí s těmito: `" " 20 1 500 0`. Pokud je podán jen jeden argument a to řetězec `test` je program otestován na předem připravených argumentech. Jakákoliv jiná kombinace argumentů je programem označena za nesprávné argumenty a algoritmus se nespustí. Pokud si uživatel chce pouze prohlédnout ukázkový výstup testů, poslouží mu k tomu soubor *test_results_level_generator.txt*.

6.6 Část pro replikaci

Tato aplikace je uložena jako adresář s Python skriptem. Stará se o spuštění algoritmu pro trénování a o definice neuronových sítí. Výstup ukládá do složky *out*, relativně k místu, kde byla spuštěna. Výčet výstupů programu vypadá takto.

- Soubor *out/results.json* slouží k ukládání výsledků experimentů mezi experimenty s různými trénovacími množinami.

- Soubor *out/maps.txt* obsahuje mapy z validační množiny s vnořeními. Mapy jsou uložené ve formátu iwonmapformat 6.4.

- Soubor *out/embeds_bad.txt* obsahuje mapy s vnořeními z validační množiny označené preferencemi jako špatné. Mapy jsou uložené ve formátu iwonmapformat 6.4. Jsou vždy přepsána na začátku experimentu s novými preferencemi a tak většinou skončí jako mapy označené za špatné preferencemi parametrizovanými poslední hodnotou v listu `perc_bad`.

- Složka *out/all* obsahuje výsledky experimentů algoritmu, kde jsou vnoření vytvářena trénovanou sítí.

- Složka *out/reference_average_minmax* obsahuje výsledky experimentů algoritmu s referenčními vnořeními aritmetického průměru maximálního dosaženého skóre z dané hloubky přes všechny průběhy.

- Složka *out/reference_global_minmax* obsahuje výsledky experimentů algoritmu s referenčními vnořeními maximálního dosaženého skóre.

Tím, že složka obsahuje výsledky experimentů myslíme, že obsahuje soubory jejichž jméno obsahuje na začátku číslo, podtržítko a další číslo. První číslo značí parametr simulovaných preferencí p, druhé číslo značí počet známých nežádoucích map f 7.5. Zbytek názvu značí význam vizualizace. Soubory končící na

perc_bad.png obsahují graf zlepšení oproti náhodným vnořením. Soubory končící na *perc_bad.txt* obsahují pro zlepšení oproti náhodným vnořením přesné hodnoty pro specifická procenta. Soubory končící na *perc_bad_value.png* obsahují graf hodnot experimentů. Soubory končící na *perc_bad_value.txt* obsahují přesné hodnoty experimentů pro specifická procenta.

Vstupní body

Soubor *main.py* je vstupní bod aplikace. Tento skript lze spustit pomocí interpreteru Python verze 3.11.5 s knihovnami dle 6.2. Příklad spuštění by mohl vypadat takto
`python main.py -m 07-06_14-20 -d`. Ukázková trénovací data lze stáhnout na adrese
`https://gitlab.mff.cuni.cz/sindelp3/homeguardbase`.

Soubor *test.py* spustí testování algoritmu s různými argumenty, podobně jako *main.py* vyžaduje interpreter Python verze 3.11.5 s knihovnami dle 6.2. Pokud existuje soubor *data/test* pokusí se ho a příslušné soubory dat (viz parametr příkazového řádku
`derive_filenames_from_map_file`) použít při testech. Testovací data *test_data.zip* lze stáhnout z adresy
`https://gitlab.mff.cuni.cz/sindelp3/homeguardbase`. Pokud není soubor *data/test* přítomný použije pro testy náhodně generovaná data. Soubor *zip.py* slouží pro jednoduché uložení zdrojového kódu modulu do zip archivu pro použití v Google Colab. Pokud si uživatel chce pouze prohlédnout ukázkový výstup testů části v Pythonu, poslouží mu k tomu soubor *test_results_learning.txt*.

Parametry příkazového řádku

Ke zprávě argumentů aplikace autor použil `argparse`, standardní knihovnu Pythonu. Argumenty, které mají v defaultní hodnotě `iwonformat` 6.4 jsou vždy vyžadovány v tomto formátu.

- `map_filename` je jméno souboru, kde jsou uloženy mapy. Defaultně (ve výchozím nastavení)
  `maps.txt`. U tohoto souboru je očekáváno, že bude ve formátu `UTF8` a obsahovat hodnoty dle formátu iwonmapformat 6.4. Vektory vnoření jsou v této aplikaci ignorovány. Uložené řetězce map také nejsou ve finální verzi algoritmu použity. Finální verze brala stromy hry hrané na mapě se stejnými kryty ale různými portály jako že patří do stejné třídy.

- `actions_filename` je jméno souboru, kde jsou uloženy akce stromu hry ve stejném pořadí jako mapy. Defaultně `actions.iwonformat`.

- `scores_filename` je jméno souboru, kde jsou uloženy skóre stavů stromu hry ve stejném pořadí jako mapy. Defaultně `scores.iwonformat`.

- `minmaxes_filename` je jméno souboru, kde jsou uloženy maximální dosažitelné skóre ze stavů stromu hry ve stejném pořadí jako mapy. Pokud bychom simulovali i protivníka, technicky by místo maximálního dosažitelného skóre ze stavu použít minimax ze stavu. Defaultně `minmaxes.iwonformat`.

- `derive_filenames_from_map_file` je vlajka určující, zda budou `actions_filename`, `scores_filename` a `scores_filename`, respektive odvozena jako
  `map_filename` + `.treeLAInds.iwonformat` ,
  `map_filename` + `.treeScores.iwonformat`
  a `map_filename` + `.treeMinmaxes.iwonformat`. Defaultně není nastavena.

- `config_filename` je jméno konfiguračního souboru. Defaultně `config.json`.

- `config_out_folder`: Pokud je hodnota nastavena na validní jméno neexistující složky, nespustí se algoritmus strojového učení. Místo toho jen program rozdělí konfigurační soubor do všech možných kombinací. Poté vytvoří složku a všechny nadsložky a výsledné soubory uloží do této složky. Defaultně prázdný řetězec.

- `experiments_also_before` Umožňuje vyhodnotit experimenty na předem naměřených datech před dalším vyhodnocením. Defaultně false. Defaultně není nastavena.

- `num_splits` je počet vytvořených trénovacích množin. Defaultně 20.

- `split` je trénovací množina indexovaná od nuly, která má být vyhodnocena. Defaultně 0.

- `max_val` je maximální počet prvků daných stranou pro validační množinu. Pokud je větší než polovina počtu dat, je použita polovina dat. Defaultně 2000.

- `seed` Trénovací množiny budou vytvořeny vždy stejně, dokud se nezmění. Defaultně 0.

- `do_splits_from_zero` Pokud je tato vlajka nastavena, místo toho, aby se vyhodnotila jen trénovací množina daná `split`, je vyhodnoceno `split` prvních trénovacích množin. Defaultně není nastavena.

Konfigurační soubor

Pro načtení konfiguračního souboru použil autor `json`, standardní knihovnu Pythonu. Soubor *config.json* používá formátu *json* pro uložení konfigurace učení a sítě. Hodnotu lze nahradit listem hodnot a poté budou spuštěny všechny kombinace hodnot všech parametrů. Některé hodnoty jsou označené jako listy, ty vždy musí být list, nebo list listů. Pokud je to list listů, počítá se, že každý vnitřní list bude obsahovat hodnoty pro jedno spuštění. Aplikace podporuje v konfiguračním souboru i další klíče, mnoho kombinací jejich hodnota ale finální verze aplikace nepodporuje, a tak není doporučené je používat. Každá nevyplněná, nebo nevalidní hodnota je nahrazena defaultní hodnotou. Ukázku konfiguračního souboru lze nalézt zde 6.1. Defaultní hodnoty pro argumenty lze nalézt zde 6.2. Hodnota `validate_every_n` je nastavena až jako dvojnásobek počtu batchí v trénovací množině.

- `learning_rate` je hodnota α v učícím algoritmu Adam [17]. Validní, pokud je reálné číslo větší než nula a menší nebo rovno než jedna. Některé hodnoty mohou způsobit nevalidní průběh učení.
- `learning_rate_decay` je parametr učícího algoritmu. Hodnota α učícího algoritmu Adam [17] je tímto parametrem vynásobena po každé polovině projetí trénovací množiny. Validní, pokud je reálné číslo větší než nula a menší nebo rovno než jedna.
- `embedding_dim` je dimenze vnoření. Validní, pokud je přirozené číslo.
- `experiment_name` udává jméno, pod kterým ukládat síť do složky models. Pokud je prázdným řetězcem, síť nebude ukládána. Validní, pokud je řetězec.
- `device` jmenuje zařízení, na kterém provádět operace neuronové sítě. Validní pokud je jedno z `cpu` nebo `cuda`. Pokud zařízení `cuda` není dostupné, je nastavena defaultní hodnota.
- `actions_range` udává horní neinclusivní limit hodnot indexů akcí. Čísla akcí budou onehot encodována do vektoru s délkou danou tímto číslem. Validní, pokud je přirozené číslo.
- `batch_size` je velikost batchí při trénování. Validní, pokud je přirozené číslo.
- `max_epochs` udává kolikrát projít trénovací množinu během trénování. Validní, pokud je přirozené číslo.
- `reference_embeddings`: Určuje jestli chceme ukázat efektivnost eliminačního algoritmu na náhodných vektorech, vektorech používajících pouze celkový minimax a vektorech používajících minimax po hloubkách. Validní, pokud je `true` nebo `false`.
- `print_every_n` udává kterou každou iteraci vypisovat trénovací chybu. Validní, pokud je přirozené číslo.
- `eliminator_known` je list určující kolik špatných známe při eliminaci, každá hodnota v listu je vyhodnocena v kombinaci s každou hodnotou v listu `perc_bad`. Je možné, že pro nějaké kombinace `perc_bad` a velikosti validační množiny nebude dostatečný počet špatných, program tento fakt vypíše a skončí. Validní, pokud je každá hodnota v listu přirozené číslo.
- `eliminator_iters` udává kolik různých výběrů špatných zkusit v eliminaci. Validní, pokud je přirozené číslo.
- `eliminator_combinations` je počet podmnožin typů špatných co zkusíme. Pokud chceme zjistit, jak na sobě různé typy map špatných podle 4.5 závisí, můžeme při vybírání známých špatných táhnou jen ze sjednocení dané podmnožiny typů špatných. Validní, pokud je přirozené číslo.
- `hidden_sizes` je list určující velikosti skrytých vrstev sítě. V aktuální verzi se pro všechny sítě používá první hodnota. Validní, pokud je každá hodnota v listu přirozené číslo.

- `dropouts_enc` je list určující pravděpodobnost, že v encoderu bude kanál vrstvy vyhozen. První hodnota ovlivňuje první multi-head attention vrstvu encoderu, druhá druhou. Validní, pokud je každá hodnota v listu reálné číslo větší nebo rovno než nula a menší než jedna.
- `dropouts_dec` je list určující pravděpodobnost, že v decoderu bude kanál vrstvy vyhozen. První hodnota určuje pravděpodobnost vyhození kanálů z vnoření stromu. Druhá hodnota ovlivňuje první multi-head attention vrstvu decoderu, třetí druhou. Validní, pokud je každá hodnota v listu reálné číslo větší nebo rovno než nula a menší než jedna.
- `perc_bad` každá hodnota z tohoto listu značí procento dat, která označí za špatná. Pro každé takové procento je eliminační algoritmus spuštěn zvlášť a výsledky jsou ukládány zvlášť. Validní, pokud je každá hodnota v listu reálné číslo větší nebo rovno než nula a menší než jedna.
- `validate_every_n` udává jak často se má provést validační krok algoritmu.

6.7 Ukázková aplikace Homeguard

Ukázková aplikace sice není součástí práce, ale autorovi přišlo užitečné podat dokumentaci k ukázkovému použití postupu.

Hlavní menu

Hlavní menu obsahuje několik položek uživatelského rozhraní.

- Tlačítko „Recalibrate on new sample“ spustí algoritmus interaktivní generace. První mapa je buď náhodná, nebo podle předem specifikovaných hodnocení.
- Tlačítko „End Game“ ukončí hru.
- Zaškrtávací políčko „Use Prespecified“ určuje, zda vždy navíc k hráčovým preferencím použít předem specifikované hodnocení.
- Zaškrtávací políčko „Use Evolution“ určuje, zda využít plná postup s evolučním algoritmem, nebo jen vybírat mapy z předem připravených pomocí eliminačního algoritmu.
- Zaškrtávací políčko „Show AI“ určuje, zda má být průběh hry demonstrován umělou inteligencí.

V hlavním menu aplikace dole napravo vypíše název složky. Tato složka slouží pro výpis map ohodnocených hráčem ve formátu iwonformat 6.4 do souboru *savedQueryResults.txt*. Pokud složka obsahuje před spuštěním aplikace soubor *maps.txt*, jsou jako předem vygenerované mapy s vnořeními využity tyto mapy, pokud ne, je tento soubor vytvořen podle defaultního. Pokud složka obsahuje před spuštěním aplikace soubor *embeds_bad.txt*, jsou jako předem specifikovaná hodnocení použity vnoření a hodnocení těchto map, hodnocení jsou čtena z položky indexu třídy 6.4. Pokud soubor neexistuje, je vytvořen podle defaultního. Očekáváme tedy, že vnoření *maps.txt* a *embeds_bad.txt* jsou spolu kompatibilní. Například byly vytvořeny neuronovou sítí se stejnými parametry i hyperparametry.

```
{
  "hidden_sizes": [
    128
  ],
  "dropouts_enc": [
    0,
    0
  ],
  "dropouts_dec": [
    0,
    0,
    0
  ],
  "validate_every_n": 10000,
  "actionsRange": 6,
  "learning_rate": 0.005,
  "learning_rate_decay": 0.95,
  "embedding_dim": 64,
  "perc_bad": [
    15,
    25,
    40
  ],
  "experiment_name": "1",
  "device": "cuda",
  "batch_size": 64,
  "batch_size_val": 4,
  "resume_ckpt": "",
  "max_epochs": 10,
  "reference_embeddings": false,
  "print_every_n": 1,
  "eliminator_known": [
    40,
    80
  ],
  "eliminator_iters": 100,
  "eliminator_combinations": 1
}
```


Obrázek 6.1 Ukázka konfiguračního souboru.

```
{
    "hidden_sizes": [128],
    "dropouts_enc": [0, 0],
    "dropouts_dec": [0, 0, 0],
    "learning_rate": 0.001,
    "learning_rate_decay": 0.25,
    "embedding_dim": 64,
    "experiment_name": "1",
    "device": "cpu",
    "batch_size": 64,
    "batch_size_val": 4,
    "max_epochs": 3,
    "reference_embeddings": false,
    "print_every_n": 10,
    "eliminator_iters": 1,
    "eliminator_combinations": 1,
    "perc_bad": [25, 40],
    "eliminator_known": [10],
    "actionsRange": 6
}
```

Obrázek 6.2 Ukázka defaultních hodnot.

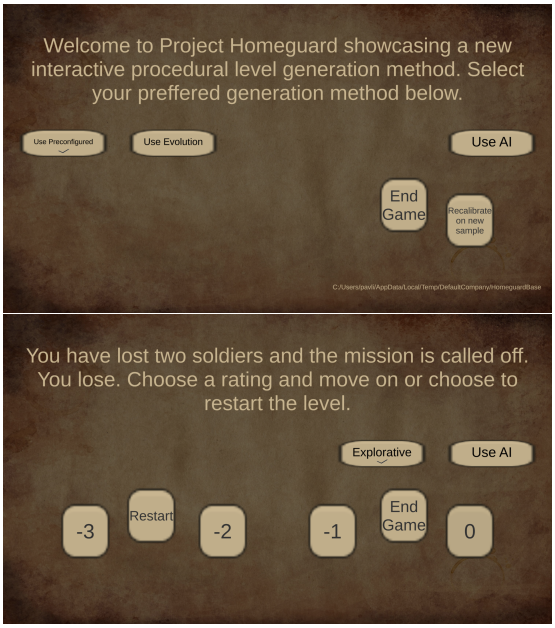
Algoritmus interaktivní generace

Uživatelská dokumentace k samotné ukázkové aplikaci bude strohá, jelikož aplikace sama obsahuje návod a mnoho nápovědných prvků. Pro vysvětlení, co značí části UI lze stisknout klávesu F2.

Poté, co se algoritmus spustí je hráči předložena mapa k odehrání, nebo je mu prezentován průběh hry na mapě umělou inteligencí. Hráč své vojáky ovládá tak, že vždy klikne na jednu ze zeleně vyznačených pozic vedle aktivního vojáka. Aktivní voják je označený červeným kruhem. Pokud nechce vojákem hýbat, může stisknout klávesu F1 pro přeskočení tahu. Pokud chce místo sebe nechat za vojáka v tomto tahu hrát umělou inteligenci, stačí stisknou klávesu F3.

Poté, co hráč buď vyhraje zabitím dvou oponentových vojáků, nebo prohraje tím, že dva jeho vojáci zemřou, může vybrat hodnocení mapy. Zkratka F11 umožňuje kdykoliv přeskočit přímo k hodnocení. Pokud nastane nějaká chyba za běhu, aplikace vypíše, že hráč prohrál, a přeskočí na výběr hodnocení.

Menu hodnocení umožní hráči přenastavit, jestli chce být prezentován průběhy za použití umělé inteligence. Je mu také předložena možnost vypnout průzkumný mód evolučního algoritmu, to způsobí, že mapy budou o něco méně různorodé. Dále je mu dána možnost pomocí tlačítka „Replay“ restartovat mapu, nebo jí ohodnotit a pokračovat některým z tlačítek s číselm hodnocení.



Obrázek 6.3 Ukázka hlavního menu a menu hodnocení.

7 Experimenty

To, jestli algoritmus opravdu bere v potaz uživatelovy preference, nebylo z manuálního testování finální aplikace jisté a vyžadovalo to mnoho úsilí. Bylo proto rozhodnuto nasimulovat uživatele za použití teorie hraní her a udělat statistické testy efektivity postupu. Ty potom byly použity pro usměrňování vývoje postupu.

Existuje mnoho způsobů jak heuristicky nebo statisticky testovat efektivitu algoritmů strojového učení. Statistické metody na rozdíl od těch heuristických lze použít v důkazu používajícím vědeckou metodu pro danou hladinu pravděpodobnosti. Jak bylo řečeno dříve, ve strojovém učení se často dělí množina dat na trénovací množinu, validační množinu a testovací množinu. Všechny měření se provádějí na testovací množině a jejich pravděpodobnost je podmíněná výběrem trénovací a validační množiny.

7.1 Heuristiky

V moderním strojovém učení se často pro porovnávání algoritmů publikují pouze heuristické veličiny jako accuracy, recall, precision, F1 skóre a podobné.

Accuracy je procento dat, co bylo správně klasifikováno. Precision je procento dat dané třídy, co bylo správně klasifikováno z těch, co byly klasifikovány do dané třídy. Recall je procento dat dané třídy, co bylo správně klasifikováno z těch, co patřily do dané třídy. F1 skóre je harmonický průměr precision a recall a je tedy vysoké, jen pokud jsou obě tyto statistiky vysoké.

Heuristické metody oproti mnoha statistickým nevyžadují zmenšení množiny trénovacích dat a pro ověření efektivity je nemusíme trénovat několikrát.

Heuristiky se používají namísto rigoróznějšího postupu obzvláště, pokud je algoritmus trénován dlouho a na velké trénovací množině. Potom jsou opakovaná trénování často restriktivně výpočetně náročná. To je ospravedlňováno tím, že se součástí výsledného postupu bere i trénovací množina a že pokud je trénovací množina dost veliká, je jeden průběh přes celou takovou množinu dostatečně reprezentativní. Často nás také zajímá jen hrubý odhad užitečnosti algoritmu, jelikož při použití v nekontrolovaném prostředí může být jeho užitečnost o dost nižší, než by indikovaly testy.

7.2 Studentův intervalový odhad

Studentův intervalový odhad ([33]) využívá toho, že pokud je náhodná veličina X normálně rozdělená s neznámým průměrem μ, pro výběry $X_1 \dots X_n$ má náhodná veličina

$$\frac{\sum_i^n X_i - \mu}{\sqrt{\frac{1}{1-n}\sum_i^n (X_i - \sum_i^n X_i)^2}}$$

Studentovo t rozdělení s $n-1$ stupni volnosti. Tuto vlastnost poté lze použít pro určení intervalového odhadu μ dané hladiny významnosti.

Studentův párovaný intervalový odhad využívá fakt, že, pokud je náhodný vektor normálně rozdělený, rozdíl jeho složek bude také normálně rozdělený. Naše

náhodná veličina je zhruba normálně rozdělená na intervalu $[0, 1] \times [0, 1]$. Některé intervalové odhady položily horní závoru rozdílu tak, že by musela hodnota náhodného vektoru být mimo tento rozsah. Jelikož se ale nad 1 vyskytuje jen nepatrná část pravděpodobnostní hustoty, přišlo autorovi vhodné toto zanedbat.

7.3 Testování eliminačního algoritmu

Pro hodnocení algoritmu pro tvorbu vnoření bylo nejdříve potřeba určit jak hodnotit algoritmus eliminace. Bylo rozhodnuto použít preference simulované mírami kvality gameplaye z 4.5. To bylo v naději, že budou reprezentovat užitečnost algoritmu, i když to samozřejmě není jisté. Tento postup testování je přípustný, protože tyto metriky nejsou nijak použity během trénování.

Eliminační algoritmus vrací pořadí. Je potřeba určit, jak je užitečné. Generování map trvá kratší čas, než trvá hráči mapu odehrát. Proto si můžeme dovolit odstranit velké množství map. Na počtu uspokojivých map, které byly odebrány, tedy až tolik nezáleží. Nejlepší se tedy zdálo pro určení míry užitečnosti použít posloupnost dvojice recall, procento odstraněných map.

Jako eliminační algoritmus se použil eliminační algoritmus číslo 2 z 4.7. Finální algoritmus pro tvoření vnoření se skládal z encoderu a decoderu pro doplňování, finální architekturu a parametry učícího algoritmu lze nahlédnout zde 5.2. Experimenty je možné replikovat pomocí postupu popsaném v 6.6 za použití ukázkového konfiguračního slovníku.

7.4 Experimenty s generováním vnoření

Datová množina měla celkem 20000 prvků. Algoritmus byl vyhodnocován vždy na stejné testovací množině o velikosti 2000 prvků. Zbylých 18000 prvků bylo rovnoměrně rozděleno mezi 20 trénovacích množin. Efektivnost algoritmu je náhodná veličinu závislá na výběru trénovací množiny a výběru f známých špatných prvků.

Recall algoritmu, který nedokáže rozlišit mezi špatnou a nešpatnou mapou z daných známých špatných map, bude zhruba lineárně růst s počtem odebraných map. Eliminační algoritmus 2 vždy nejdříve odebere známé špatné. Pokud nazveme velikost testovací množiny n, počet odebraných prvků p a počet špatných prvků a, poté pro $p \leq f$ platí

$$\text{RecallRandom}(p) = \frac{p}{a},$$

ale pro $p > f$, pokud $a >> f$, tak

$$\text{RecallRandom}(p) = \frac{p}{n}.$$

To nastane, protože jsou vnoření nezávislá na označení prvku a protože pořadí odebrání prvků s neznámým hodnocením závisí jen na vnořeních. Nakonec autor tento fakt nevyužil a rozhodl se pro lepší přesnost přímo simulovat algoritmus náhodného odebírání. Je ale užitečné konstatovat, že to znamená, že stačí porovnat náš algoritmus jen s tímto eliminačním algoritmem na náhodných vnořeních.

Náhodné seřazení, které nebere v potaz známé špatné, bude mít vždy v průměru horší nebo stejné výsledky.

$$r^j_{x,i} = \frac{i}{100}\text{recall}^j_{x,i},$$

kde $\text{recall}^j_{x,i}$ značí naměřený recall algoritmu x pro j. nezávislý náhodný výběr známých špatných pro i procent odebraných map. Parametr o značí počet nezávislých výběrů známých špatných.

Pro porovnání efektivnosti algoritmu a s náhodným odebíráním, neboli algoritmem n, autor nakonec použil náhodný vektor

$$b \in R^{99}.$$

Ten definoval jako

$$b_i = E\left(\frac{\sum_{j=1}^{o} r^j_{a,i} - r^j_{n,i}}{o}\right).$$

Vektor b tedy reprezentuje to, jaký zlomek nežádoucích odebere algoritmus s naučenými vnořeními herních stromů navíc, v porovnání s algoritmem s náhodnými vnořeními.

Existence střední hodnoty náhodné veličiny $r^j_{a,i} - r^j_{n,i}$ plyne z konečného počtu výběrů množiny známých špatných a možných pořadí odebírání. Z centrální limitní věty plyne, že jelikož pro dané $i \in [1, 99]$ jsou $r^1_{a,i} - r^1_{n,i}, r^2_{a,i} - r^2_{n,i}, \ldots, r^o_{a,i} - r^o_{n,i}$ i.i.d. s průměrem μ, bude b_i mít zhruba normální rozdělení s průměrem μ. Pro nezávislé popsání hodnot souřadnic tohoto vektoru vyhotovíme Studentův intervalový odhad s hladinou významnosti 95%.

7.5 Výsledky experimentů

Pro účely experimentů bylo pro každou mapu nezávisle vytvořeno deset stromů hry. Experimenty autor vyhotovil pro výběry několika různých hodnot parametrů simulovaných preferencí 4.5. Preference jsou simulovány pomocí vlastností herních průchodů průměrovaných přes všechny herní průchody. Vždy stačí, aby strom hry měl jednu z nežádoucích vlastností. Tyto vlastnosti jsou všechny parametrizovány stejným procentem dat p.

1. Není v p mapách s nejméně lead change.
2. Není v p mapách s nejméně lead possibility change.
3. Není v p mapách s nejméně dramatem.
4. Není v p mapách s nejméně uncertainty.
5. Není v p mapách s nejméně pozdní uncertainty.

Hodnocené mapy byly náhodně vybrány. Jejich počet nazvěme f. Pro každou kombinací p a f byly uskutečněny čtyři typy experimentů.

- Experiment *základní* nebral v potaz mapy a bral stromy jako nezávislé herní zážitky.

- Experiment *odlišování map* kontroluje, jestli je algoritmus schopný poznat, že strom byl hraný na nějaké z náhodně označených map. Zde bereme mapy se stejnými kryty ale různými portály jako stejné mapy.

- Experiment *0.1-konzistentní*. Experiment *k-konzistentní* přeznačí stromy. Pokud bylo k stromů hraných na stejné mapě označeno za špatné, jsou všechny stromy hrané na dané mapě označeny za špatné. Pokud nebyl dostatek stromů označený za špatné, jsou všechny stromy hrané na dané mapě označeny za nešpatné.

- Experiment *0.2-konzistentní*.

Jak bylo zmíněno dříve v 7.4, porovnáváme výsledky eliminačního algoritmu na vnořeních herních stromů s eliminačním algoritmem na náhodných vnořeních.

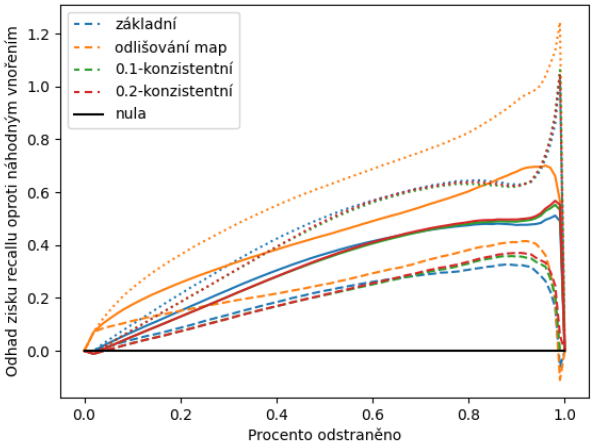
Pro vizualizaci experimentů se autor rozhodl do každého grafu dát čtyři typy experimentů. Křivkám výsledků experimentů je vždy přiřazena stejná barva. Plná čára ukazuje aritmetický průměr naměřených hodnot. Přerušovaná/tečkovaná čára ukazují dolní/horní interval 95% věrohodnosti odhadu průměru náhodného výběru. Hodnoty v 0 a 1 jsou nastaveny na 0, jelikož není možné dosáhnout zisku nad žádným algoritmem, když na pořadí eliminace nezáleží.

Výsledky je možné nahlédnout zde 7.1 7.2 7.3. Jelikož dolní odhad je pro valnou většinu testovaného intervalu nad hladinou náhody, lze prohlásit, že s 95% je eliminační algoritmus využívající vnoření herních stromů lepší než eliminační algoritmus využívající náhodná vnoření. Specificky toto platí při využití na simulovaných preferencích a pro zjišťování, které stromy hry byly odehrané na stejné mapě. Jak bylo řečeno dříve, je algoritmus využívající náhodná vnoření je vždy lepší, nebo stejně dobrý, než náhodné uspořádání. Z předchozích dvou vět plyne, že je eliminační algoritmus využívající vnoření herních stromů lepší než náhodné uspořádání.

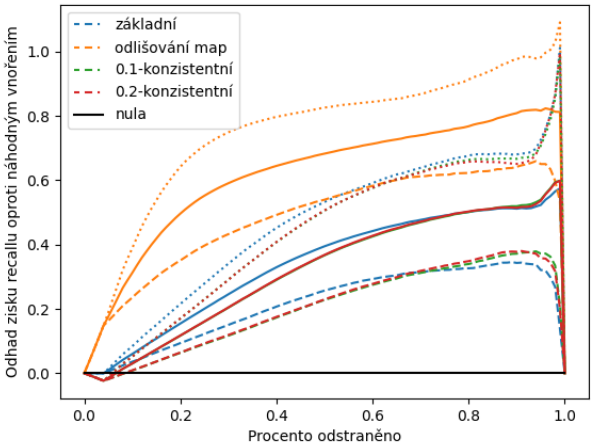
Pro dodatečné porovnání byl podobný experiment proveden s vnořeními vytvořenými jen z maximálního dosažitelného skóre 7.4. Takové experimenty nevyžadují učení neuronové sítě a jsou tedy zcela závislé na výběru validační množiny. Autor se rozhodl udělat jen jeden běh pro každý, jelikož změna validační množiny má velmi malou šanci na to znatelně ovlivnit výsledek. Tato vnoření mají dobrou schopnost předpovídat hráčovy preference, simulované podle teorie z 4.5. Nejsou ale schopná dobře rozlišovat, které stromy hry byly odehrány na které mapě. Nemají tedy charakter vnoření map, což je základní předpoklad pro to, aby byla v algoritmu vnoření užitečná. Z toho plyne, že to, jak simulujeme hráčovy preference, není pro určení kvality procesu užitečné. Jelikož autor referenční experimenty provedl už v konečném stádiu práce, budou muset lepší testovací preference být dodány v nějaké budoucí práci.

7.6 Další experimenty

Definitivně zjistit, které aspekty tvorby herního stromu a které aspekty návrhu postupu hlubokého učení byly nejdůležitější se autorovi nepodařilo. Malé změny v postupu tvorby herního stromu často vedly k tomu, že se změnila efektivita architektur. Dále by bylo na místě otestovat, jak se algoritmus chová pro různé simulované uživatele a různé generátory.

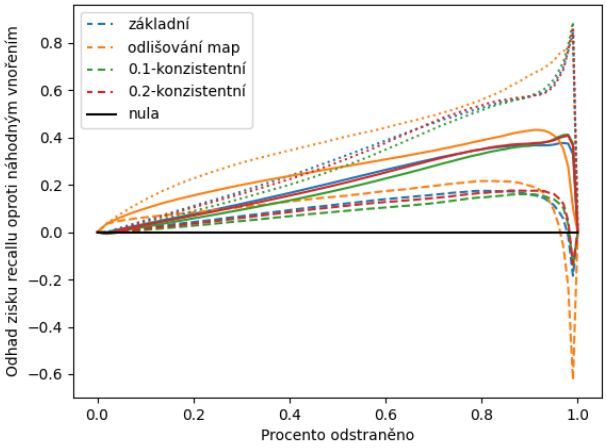


(a) Výsledky testů $f = 40$ $p = 15$.

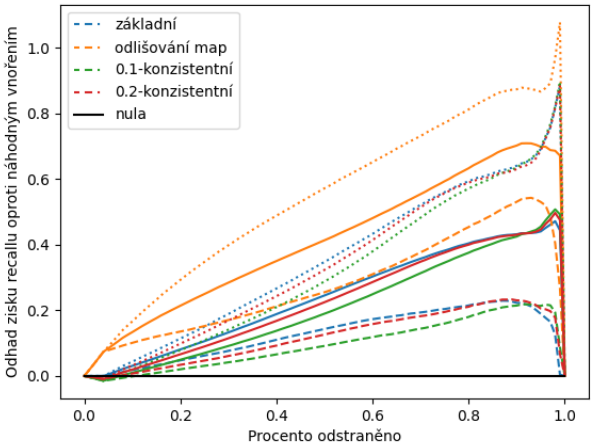


(b) Výsledky testů $f = 80$ $p = 15$.

Obrázek 7.1 Výsledky testů 7.5 $p = 15$.

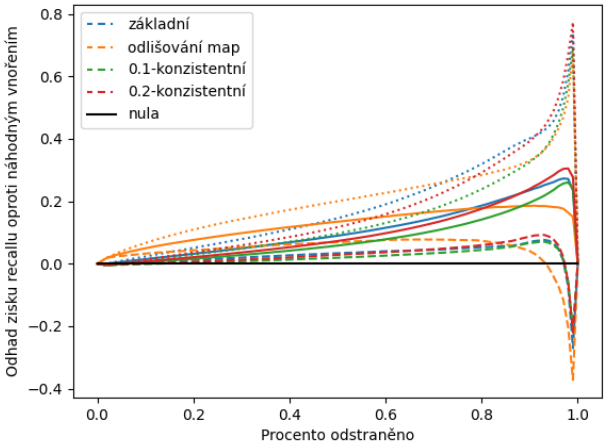


(a) Výsledky testů $f = 40$ $p = 25$.

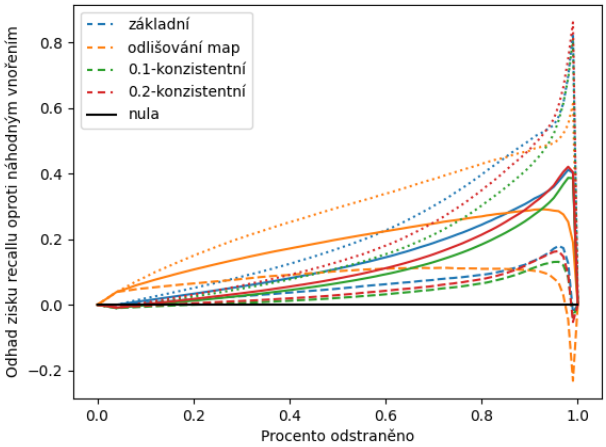


(b) Výsledky testů $f = 80$ $p = 25$.

Obrázek 7.2 Výsledky testů 7.5 $p = 25$.

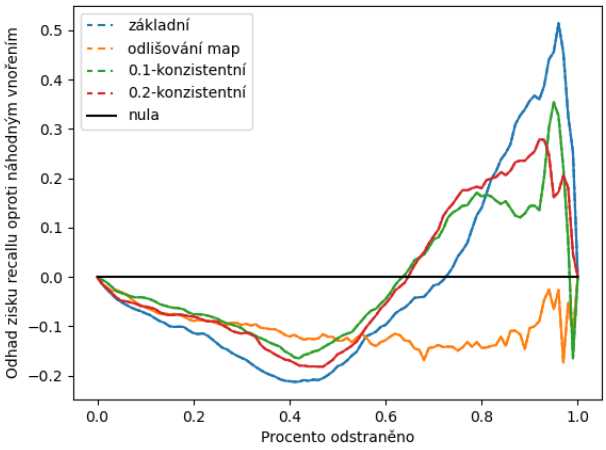


(a) Výsledky testů $f = 40$ $p = 40$.

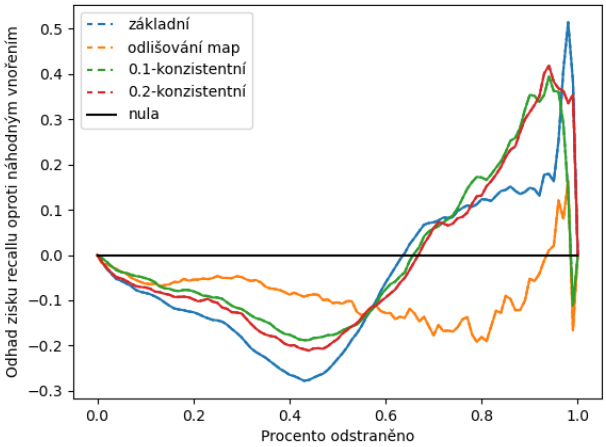


(b) Výsledky testů $f = 80$ $p = 40$.

Obrázek 7.3 Výsledky testů 7.5 $p = 40$.

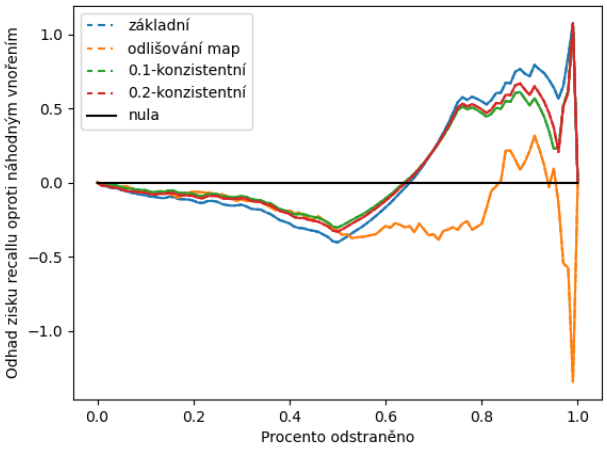


(a) Výsledky testů $f = 40$ $p = 25$.

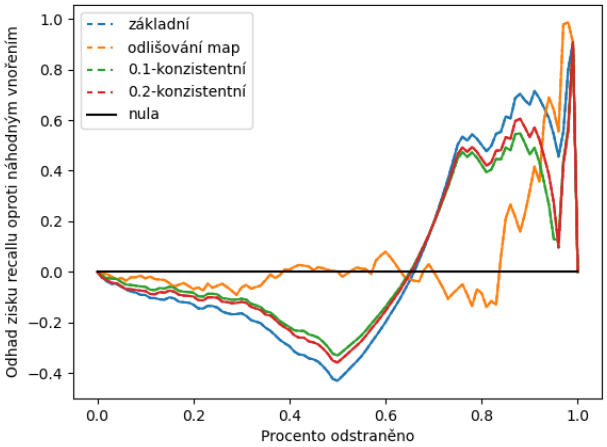


(b) Výsledky testů $f = 80$ $p = 25$.

Obrázek 7.4 Výsledky referenčních testů. Jako vnoření zde bereme jen maximální dosažitelné skóre přes všechny průběhy.



(a) Výsledky testů $f = 40$ $p = 25$.



(b) Výsledky testů $f = 80$ $p = 25$.

Obrázek 7.5 Výsledky referenčních testů. Jako vnoření zde bereme průměrné nejvyšší dosažitelné skóre v každé hloubce přes všechny průběhy.

Pro to, aby byl postup opravdu obecně použitelný, by bylo potřeba hlubšího pochopení, které architektury kdy fungují. Také by se hodilo použít nějakou z metod pro vysvětlení chování neuronových sítí, nebo nějaký lépe interpretovatelný model strojového učení.

U některých z těchto úkolů by se dalo brát, jako že jsou v rámci práce. Autor se ale snažil soustředit se na vytvoření co nejlepšího ukázkového postupu.

8 Programátorská dokumentace

Práce vyžadovala velké množství datových struktur a procedur. Jak bylo dříve zmíněno, autor tyto procedury a struktury implementoval v jazycích C# a Python. Autor se snažil psát kód tak, aby byl rozumně rozšiřitelný. Zvláště v částech specifických pro hru a hluboké učení ale často bylo třeba obětovat rozšiřitelnost ve prospěch efektivního využití výpočetní síly a možnosti rychle kód iterovat. V této kapitole autor pro každou kolekci procedur a datových struktur popisuje její účel, ať už je to tato kolekce modul, submodul, namespace nebo třída.

Část v C# se skládá ze tří knihoven typu dynamic link library napsaných v C# a jednoho projektu typu console application napsaného v C#. Část v Pythonu obsahuje dva skripty a několik modulů.

8.1 Knihovna Algorithms

Knihovna `Algorithms` obsahuje nejobecnější algoritmy a datové struktury.

Namespace Algorithms.DataStructures

Namespace `Algorithms.DataStructures` obsahuje obecně využitelné datové struktury

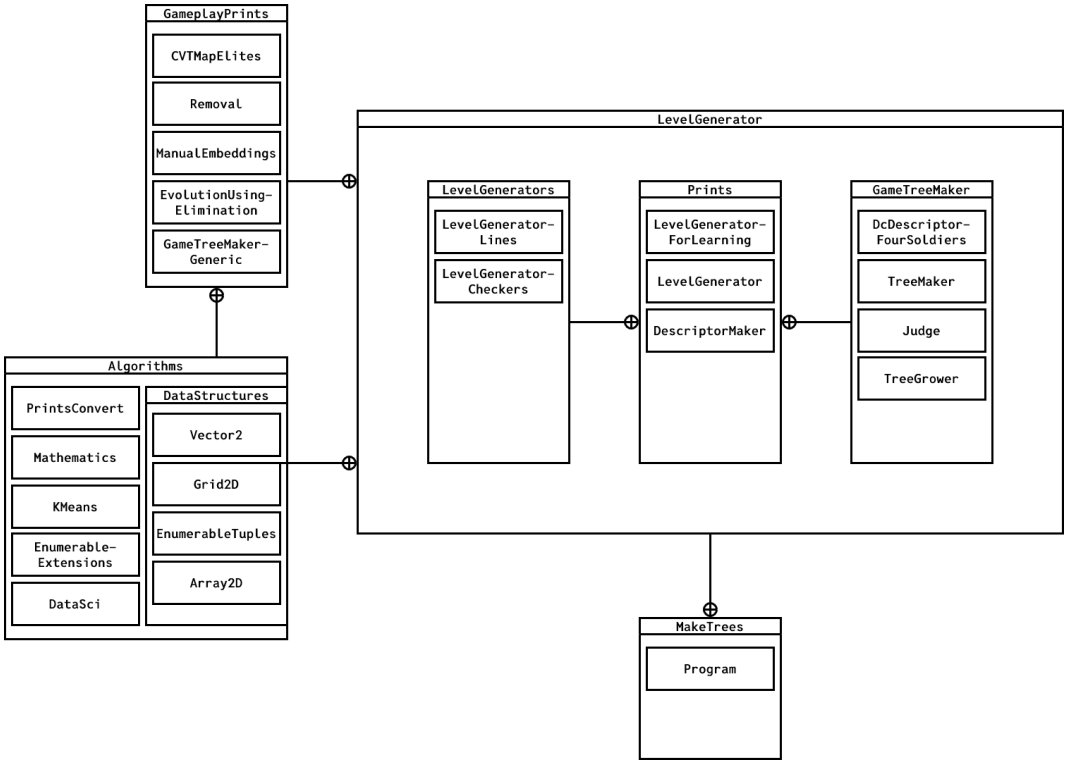
- Třída `Array2D` je dvou-dimenzionální pole uložené pomocí klasického .NET pole jako matice uložená po sloupcích.
- Třída `Grid2D` je 2d mříž s euklidovskou vzdáleností, kde jsou určité pozice zablokované dle dané funkce. Drží si pole sousedů, které upravujeme, aby implementovalo portály.
- Třída `Vector2` obsahuje dvojici čísel typu `int16`. Lze s ním provádět mnoho operací jako kdyby to byl reálný vektory. Nejdůležitěji sčítat, odčítat, násobit celým číslem a zjistit druhou mocninu délky.
- Statická třída `EnumerableTuples` zabaluje n-tice hodnot typu `T`. Implementuje `IEnumerable<T>` a indexovací metody. Hlavní idea byla oproti typu Array zvýšit lokalitu dat pro případy, kde je předem známá délka a stále mít přístup k Linq metodám a indexaci. Nakonec

Třída DataSci

Statická třída `DataSci` obsahuje funkce pro diskretizaci posloupnosti hodnot a výpočet vzájemné informace mezi dvěma posloupnostmi. Byla použita během procesu popsaném v 4.10.

Třída Mathematics

Statická třída `Mathematics` obsahuje jednoduché obecně užitečné matematické a pomocné funkce. Nejdůležitěji L2 metriku, komparátor listů v bibliografickém



Obrázek 8.1 Zjednodušený class diagram C# projektu

uspořádání a samplování čísel z distribuce popsané vektorem kumulativních pravděpodobností. Samplování provádí binárním vyhledáváním rovnoměrně náhodně taženého čísla z intervalu [0,1) ve vektoru kumulativních pravděpodobností.

Třída KMeans

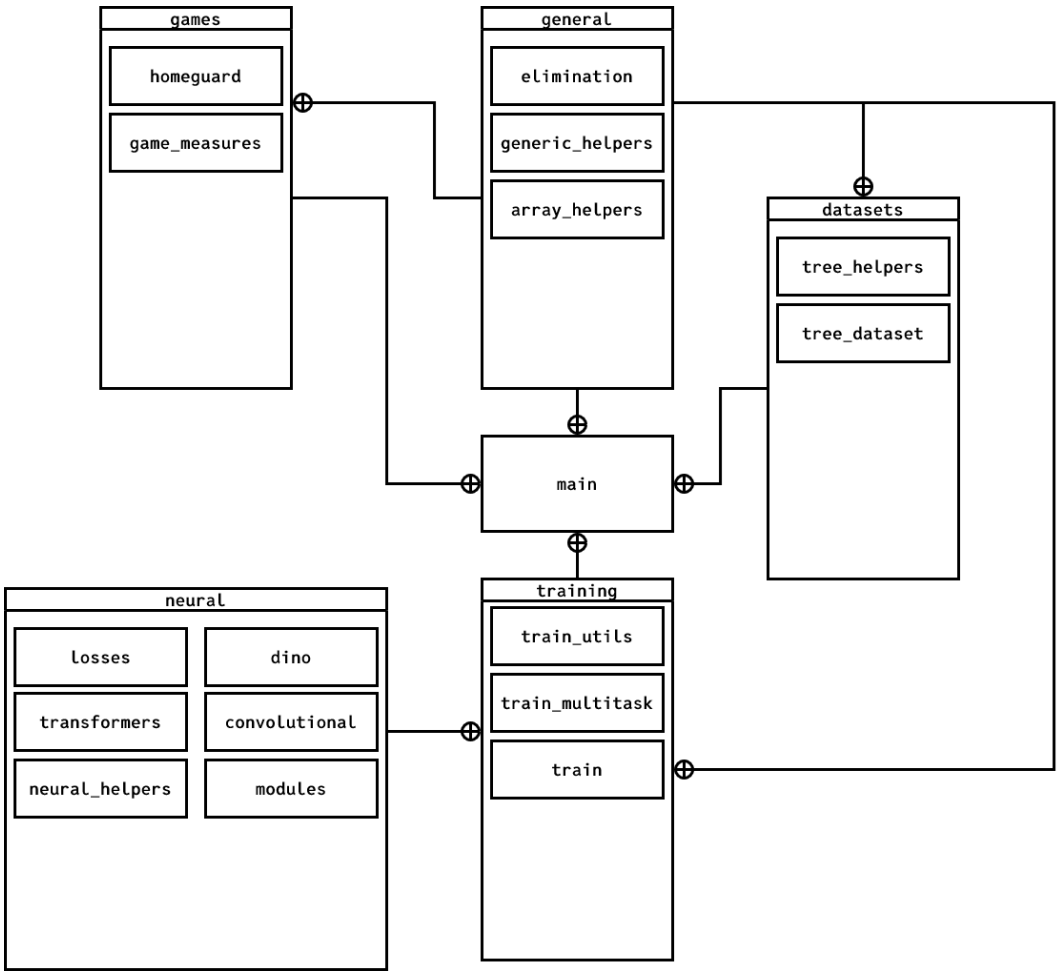
Statická třída `KMeans` obsahuje algoritmus CVT 4.3. Nejen pro implementaci CVT obsahuje funkce nabízející přístup pro algoritmus ze třídy `Accord.Kmeans`. Tato CVT implementace umožňuje pracovat s prostory, ve kterých mají různé hodnoty souřadnice různé váhy. To je užitečné například, pokud chceme generovat mapy, kde jsou různě dlouhé kryty různě pravděpodobné.

Třída EnumerableExtensions

Statická třída `EnumerableExtensions` obsahuje extension metody pro typy implementující generickou verzi interfacu `System.Linq.IEnumerable`. Velká část kódu je psána právě pomocí těchto metod. Často jsou podobné nějakým metodám ze standardního namespacu Linq. Některé jsou přímo vlastní implementace metod které jsou v Linq namespacu v .NET 6.0 ale nejsou v .NET Standart 2.1.

Třída PrintsConvert

Statická třída `PrintsConvert` zařizuje ukládání map, popisných vektorů do souborů čitelně pro člověka. Umožňuje také ukládání stromů hry do binárních souborů ve vlastním formátu .iwonformat. Původně autor pro ukládání používal formát .json, ale pro veliké stromy byl zápis do něj příliš pomalý a soubory, které generoval, příliš velké.



Obrázek 8.2 Zjednodušený class diagram Python projektu

8.2 Knihovna GamePlayPrints

Knihovna `GamePlayPrints` obsahuje metody nejvíce obecně uživatelné. V této sekci jsou popsány jednotlivé třídy z jediného namespacu projektu `GamePlayPrints`. Vyžaduje knihovnu `Algorithms`. Popisujeme ji nejhlouběji, protože je to nejdůležitější část pro další postupy. Metody jsou schválně psány tak, aby byly obecné a tedy užitečné i pro další podobné postupy.

Třída GameTreeMakerGeneric

Soubor GameTreeMakerGeneric obsahuje high level metody a rozhraní potřebná pro tvoření stromů hry 4.9.

Interface `IStateDescriptor<T, T2>` je většinou používaný v templatovaných funkcích. Typ `T` označuje nějaký typ, který dědí implementující třída. Jsou jím označováni další jedinci. Typ T2 označuje dodatečnou informaci pro `ChildrenF`.

- `Weight` je jedna plus množství zaříznutých větví, které stav reprezentuje.
- `ChildrenF` je funkce, která se zavolá pro každý stav se všemi přímo odvozenými stavy. Pro stavy v listech stromu hry se zavolá s prázdným listem.
- `ChildrenFScore` je funkce, která se zavolá pro každý stav v listu stromu hry na dodatečných dočasných přímo odvozených stavech. Ve finální verzi se nevolá nikdy, jelikož stromy byly dostatečně malé, aby nebyla potřeba.

`Node<T>` je invariantní struktura pro jednoduché ukládání stromu hry. Typ `T` značí, jak je reprezentován stav hry. Vlastnost `Desc` určuje stav hry uložený v daném vrcholu. Vlastnost `ParentIndex` určuje index rodiče ve struktuře, ve které je strom uložený. Vlastnost `Depth` je hloubka mínus jedna.

Funkce splňující delegát `GameTreeMakerGeneric.MakeInner<T, TShared>` mají za úkol tvořit podstromy průběhů hry, se kterými poté pracuje `GameTreeMakerGeneric`. Taková funkce nevrací žádnou hodnotu. Tvořené vrcholy ukládá do struktury `nodes`. Typ `T` určuje, čím reprezentujeme stavy hry. Typ TShared určuje, čím reprezentujeme pomocnou sdílenou hodnotu. Má k dispozici následující parametry.

- `desc` je stav určující kořen podstromu.
- `depth` je hloubka, ve které je kořen.
- `index` je index vrcholu (`Node<T>`) obsahujícího kořen podstromu.
- `maxDepth` je hloubka, do které tvořit.
- `nodes` je struktura, kam jsou vrcholy podstromu ukládané, umožňuje pouze ukládat vrcholy a vrací index ukládaného vrcholu.
- `seed` je konzistentní náhodná hodnota.
- `shared` je hodnota sdílená pro všechna zavolání v daném tvoření stromu.

- `dropout` je procento dalších akcí, které neprovést. První možná akce v daném stavu musí být vždy provedena a příslušný vrchol uložen do `nodes`. Vyžadujeme zde, aby hra měla v každém stavu nějakou proveditelnou akci.

Funkce `GameTreeMakerGeneric.Make<T,TShared,T2>` provádí výběr větví stromu a převádí strom na list větví. Pracuje s typem `T` implementujícím `IStateDescriptor<T, T2>`. Samotné tvoření podstromů definuje uživatel pomocí argumentu `makeInner`. Typem `TShared` určujeme pomocný argument funkce `makeInner`.

- `startDescriptor` je kořen stromu.
- `shared` je datová položka sdílená pro celou generaci stromu eg. mapa.
- `depth` je hloubka, do které strom tvořit.
- `seed` je konzistentně náhodné číslo.
- `layersPerIt` je počet tahů za iteraci, na konci každé iterace se vybírají větve.
- `clusteredBranches` je počet vybraných větví.
- `takeAllBottom` je počet prvních iterací, během nichž nemá zahazování proběhnout.
- `takeAllTop` je počet posledních iterací, během nichž nemá zahazování proběhnout.
- `makeInner` je funkce generující podstrom z kořene.
- `branchSelector` je funkce výběru větví, může udávat větvím váhu, ta se poté projevuje ve váze po váženém listu následujících vrcholů.
- `dropout` je pravděpodobnost, že v iteracích, po kterých budeme zahazovat větve zahodíme akci.
- `dropoutDontCluster` je pravděpodobnost, že v iteracích, po kterých nebudeme zahazovat větve zahodíme akci.
- `cleanUp` udává jestli uklízet zahozené větve. Zpomaluje běh algoritmu a uklízí jen zhruba polovinu větví, ale zmenšuje paměťovou náročnost.
- `extra` je dodatečný parametr funkce IStateDescription.ChildrenF() a IStateDescription.ChildrenFScore().

Statická třída ManualEmbeddings

Tato statická třída obsahuje funkci, která generuje manuálně vytvořená vnoření a umožňuje ukládat informace o tom, jak je interpretovat 4.10.

Funkce `ManualEmbeddings.MakePrint` bere jako vstup strom hry a indikátor, zda dané kolo je kolem hráče. Vrací popisný vektor herního stromu. Tento vektor je vrácen v dvojici s `PosInfo`, které obsahuje informace o tom, co je kde ve vektoru uloženo. To je výhodné v tom, že lze `PosInfo` uložit do hlavičky souboru.

- `bs` jsou větve stromu.
- `isPlayerTurn` je funkce, která pro každé číslo kola určí, jestli v něm hrál hráč.
- `dummy` je hodnota pro mockování. Pokud je nastavena na `true` je vektor nahrazen (0).

Třída CVTMapElites<TExtra>

Tato třída implementuje algoritmus CVTMapElites s celočíselnými genomy 4.3.

Konstruktor `CVTMapElites<TExtra>` vytváří tuto třídu a má následující argumenty.

- `TExtra` - libovolný typ dodatečného obsahu vraceného fitness funkcí.
- `fitness` funkce beroucí novou generaci genomů a vracející pro každý genom dvojici. Tato dvojice je číslo s plavoucí desetinnou čárkou a dodatečný obsah. Vracení dodatečného obsahu je užitečné především, pokud funkce provádí výpočetně složitý nebo randomizovaný výpočet, který je složité replikovat. Umožňuje uživateli jednoduše požít genomy ve výsledné generaci bez nutnosti dalších výpočtů.
- `length` je délka genomu.
- Geny nabývají hodnot od 0 do `maxExclusive`-1.
- `bucketCount` je počet specializací. Odpovídá velikosti populace nejvhodnějších.
- `seed` je konzistentně náhodné číslo.
- `mutationChance` je šance, že dojde k mutaci jedince.
- `crossoverChance` je šance, že dojde ke křížení jedinců.
- `weights` jsou váhy jednotlivých hodnot genu při náhodné generaci.
- `countOfNewGenomes` je počet genomů, kolik vytvořit během generace.
- `centroids` jsou centroidy centrální Voronoi tesselace.
- `eliteMapFitnesses` je list, kde každému nejvhodnějšímu genomu zapíšeme jeho fitness.
- `eliteMap` je list, kde každému centroidu přiřadíme nejvhodnější genom, který má tento centroid jako nejbližší.
- `history` je historie už vygenerovaných genomů, používá se ke zkontrolování, jestli už nebyl vytvořen genom, který by donutil nový genom zahodit pomocí preFitness.
- `centroidF` je funkce pro vytvoření centroidů.

  Předposlední čtyři parametry se především hodí pro reinicializaci GA podle jiného už někdy spuštěného.

Funkce `CVTMapElites<TExtra>.Init` umožňuje provést inicializační krok algoritmu, má následující argumenty.

- `initialGenSize` je velikost počáteční generace
- `shouldStop` je funkce beroucí index interace a indikující, jestli má algoritmus zastavit

Funkce `CVTMapElites<TExtra>.Run` spouští algoritmus, má následující argumenty.

- `shouldStop` je funkce beroucí index interace a indikující, jestli má algoritmus zastavit

EvolutionUsingElimination

Tato statická třída obsahuje funkci, která vytváří GA používající funkci pro vytvoření vnoření, hodnotící funkci a eliminační funkci. Hodnotící funkce zařizuje dané preference. Eliminační funkce poté předpovídá preference pro všechny genomy a vytváří fitness funkci algoritmu `CVTMapElites`. Je to část aplikace, která obsahuje nejkompletnější přístup k postupu 1.3. V práci není přímo využitý, ale používá ho ukázková aplikace Homeguard.

Funkce `EvolutionUsingElimination.EvolutionaryAlgElimination` má mnoho argumentů. Nepopsané argumenty značí argumenty vytvářené instance `CVTMapElites<(List<float>, T)>`. Typ `T` značí typ dodatečné informace, list floating point čísel odpovídá vnoření jedince.

- `embedderFunc` je funkce pro vytvoření vnoření.
- `ratingsFunc` je hodnotící funkce.
- `eliminationAlg` je eliminační funkce.
- `countOfNewGenomes`
- `length`
- `maxExclusive`
- `seed`
- `mutationChance`
- `crossoverChance`
- `bucketCount`
- `weights`
- `centroidF`

Removal

Tato statická třída obsahuje implementace jednoduchého algoritmu založeného na postupném odstraňování 4.7. Je velmi podobná implementaci pro Python a vychází z ní. Implementaci by šlo sjednotit použitím nativní knihovny, to ale autorovi přišlo zbytečné. Každému prvku v množině se známou důležitostí přiřadíme číslo podle jeho důležitosti. V každé iteraci jsou aktivní pouze ty prvky, jejichž číslo je dělitelem čísla iterace.

Funkce `Removal.OrderRemoving` má následující argumenty.

- `prints` je vnoření prvků.
- `knownIndsIwithRelativeImportance` je důležitost pro prvky, jejichž důležitost známe. Důležitost určuje, jak často jsou prvky využity v eliminačním algoritmu.
- `selectFunc` vybere nový prvek podle vzdáleností vnoření prvků od vnoření využitého prvku se známou důležitostí.

8.3 Knihovna IwonLevelGenerator

Tato knihovna obsahuje metody použivatelné v rámci generování map do projektu Homeguard. Vyžaduje knihovnu Algorithms a knihovnu GameplayPrints.

Namespace IwonLevelGenerator.GameTreeMakerCustom

Tento namespace obsahuje specifickou implementaci algoritmu simulace herního stromu hry Homeguard 1.2.

- Třída `DcDescriptorFourSoldier` obsahuje všechny informace o stavu hry a základní metody pro práci s tímto stavem, implementuje `IStateDescriptor<DcDescriptorFourSoldier, bool>` 8.2.
- Třída `TreeGrower` zařizuje nízkoúrovňovou simulaci hry. Je specifický pro zjednodušená pravidla hry. Hru simuluje kolo po kole a pro každé kolo vygeneruje dva popisy kola.
- Třída `GameTreeMakerCustom` zařizuje kompatibilitu `TreeGrower` s rozhraním obsaženým ve třídě `GameTreeMakerGeneric`. Specificky ho dělá kompatibilní s delegátem `GameTreeMakerGeneric.MakeInner`.
- Třída `Judge` ukládá pravidla, která nespecifikuje `TreeGrower`. Obsahuje metody pro zjištění pravděpodobnosti na zásah z pozice (a, b) do pozice (c, d). Pracuje s rovnoměrnou čtvercovou sítí a pozice (x, y) v matici délek krytů zde interpretujeme jako hodnotu pro čtverec jehož diagonála je úsečka mezi body $((x, y), (x + 1, y + 1))$. Pravděpodobnost na zásah počítá tak, že projde dvě úsečky kreslené mezi pozicemi. Jednu přímku popisuje pomocí $\frac{dx}{dy}$ druhou pomocí $\frac{dy}{dx}$. Prochází všechny čtverce, kterými tyto přímky prochází a bere maximum přes všechny jejich hodnoty.

Namespace IwonLevelGenerator.LevelGenerators

Tento namespace obsahuje generátory, které deterministicky převádí vektor čísel na výškovou mapu.

- Třída `LevelGeneratorCheckers` obsahuje generátor popsaný v 4.2.
- Třída `LevelGeneratorLines` obsahuje generátor popsaný v 4.2.

Namespace IwonLevelGenerator.Prints

Tento namespace zařizuje generování vnoření stromů hry.

- Třída `DescriptorMaker` ukládá argumenty pro generování stromů hry a map. Ukládá tedy portály a další parametry generátoru map. Poskytuje jednoduché vysokoúrovňové rozhraní pro generaci map a stromů hry pro daný vektor parametrů generátoru map.
- Třída `LevelGenerator` zařizuje rozhraní pro generování velkého množství map a stromů hry. Střídá mezi různými parametry tak, aby se každý použil zhruba ve stejném počtu. Vždy tvoří tolik stromů hry paralelně najednou, jako je počet vláken procesoru.
- Třída `LevelGeneratorForLearning` obaluje `LevelGenerator` a poskytuje rozhraní s vysokoúrovňovými parametry základních typů. Zařizuje, že lze uložit větve stromů ve formátu iwonformat. Buď stromy doplňuje maximální hodnotou typu `int16` a nebo odebírá větve tak, aby všechny stromy měly stejně větví jako je 80 percentil počtu větví.

8.4 Konzolová aplikace MakeTrees

MakeTrees umožňuje jednoduše generovat data použitá v sekci 7. Funkce `Program.Main` se zabývá zpracováním a kontrolou konzolových parametrů. Funkce `Program.DoTests` provádí end to end testy hlavní procedury `Program.Main` pro různé argumenty.

8.5 Část pro strojové učení

Kód části pro strojové učení se většinou odkazuje na knihovny a tedy je poměrně jednoduchý.

Modul general

- Submodul `elimination` obsahuje definice eliminačních algoritmů a nástroje na práci s nimi 4.7. Obsahuje i funkci, která pro eliminaci provádí různé experimenty.
- Submodul `generic_helpers` pomocné funkce pro práci s objekty z pythonu a ze standardních knihoven pythonu. Například převedení listu listů v Pythonu na list individuálních hodnot.
- Submodul `array_helpers` pomocné funkce pro typ `numpy.ndarray`.

Modul datasets

- Submodul `game_tree_helpers` pomáhá pracovat se stromem hry. Například umí rozdělovat úrovně podle společných předchozích akcí.
- Submodul `game_tree_dataset` umožňuje načítat data simulace podle autorova souborového formátu.

Modul games

- Submodul `homeguard` obsahuje simulátor preferencí pro hru homeguard.
- Submodul `game_measures` popisuje míry kvality hry 4.5.

Modul neural

- Submodul `losses` obsahuje definice ztrátových funkcí pro doplňovací ztrátu, ztráty na principu DINO [18] a ztráty založené na předpovídání sdíleného vektoru akcí pro hodnoty větví.
- Submodul `modules` obsahuje definice bloků z 5.5.
- Submodul `neural_helpers` obsahuje pomocné metody pro práci s knihovnou `torch`.
- Submodul `dino` obsahuje definici hlavy DINO a struktury ukládající potřebné modely (ie. učitele) [18].
- Submodul `convolutional` obsahuje jednoduché plně konvoluční sítě.
- Submodul `transformers` obsahuje sítě založené na attention.

Modul training

- Submodul `trian_utils` obsahuje vysokoúrovňové pomocné funkce k trénování neuronových sítí.
- Submodul `train` zařizuje potřebné argumenty pro specifický trénovací algoritmus.
- Submodul `train_multitask` obsahuje trénovací algoritmus umožňující použití doplňovací ztráty, ztráty na principu DINO a ztráty založené na předpovídání sdíleného vektoru akcí pro hodnoty větví.

main.py

Tento soubor obsahuje parser command-line argumentů. Také definuje funkce pro náhodné veličiny používané v 7. Je to vstupní skript do aplikace.

train.py

Tento soubor obsahuje skript, který vyzkouší mnoho různých parametrů příkazového řádku a konfiguračních slovníků.

handle_configs.py

Tento soubor obsahuje parser konfiguračního slovníku. Rozdělení jednoho slovníku na kombinace všech vícekrát určených hodnot, přiřazení defaultních hodnot a kontrolu validity.

9 Projektová dokumentace

Projekt prošel mnoha stádii vývoje, bylo vyzkoušeno nebo zváženo mnoho algoritmů a datových struktur. Zmiňovat všechny by bylo nad rámec této práce. Zde se tedy autor rozhodl popsat jen nejdůležitější rozhodnutí a stádia vývoje.

9.1 Migrace a zjednodušování

Prvním pokusem o generaci map byl pokus získat metriky, se kterými by bylo možné přímo evolučně vytvořit výškovou mapu reprezentující kvalitní mapu. Zprvu byly použity pouze jednoduché metriky přímo používající mapu. Autor se tak snažil zavrhnout mapy, které byly zcela nevhodné. Tyto metriky ale nebyly dostatečné ani pro tento jednoduchý úkol.

Další práce na projektu se týkala experimentování se získáváním informace ze simulací. Ze simulovaného boje jsme si pamatovali různé statistiky, například, kolikrát vyhrála každá ze stran, kolik byla průměrná šance na zásah a tak dále. Zprvu simulace pouze procházela náhodné průběhy hry. To se nakonec ukázalo, jako příliš výpočetně náročné. Dále byla snaha namísto náhody použít pro rozhodnutí tahů obou stran umělou inteligenci s mělkým minimaxem a heuristickou umělou inteligenci. Nepodařilo se použít statistiky z běhu nezávislých simulací pro vytvoření dobré metriky.

9.2 Experimentace s herními stromy

Metriky, které se nakonec prokázaly jako nejvhodnější, byly ty, které používaly aproximaci herního stromu. Při použití herního stromu nebyla potřeba si ukládat žádné statistiky, jelikož průběh byl přirozeně uložený v hranách (akcích) a vrcholech (herních stavech). Velikostí stromu bylo jednoduše možné vybrat přesnost, do jaké chceme strom popisovat. Vrcholy ukládaly o stavech jen základní informace. Tím se snížila potřeba určit, které pokročilé informace jsou důležité. Zároveň bylo jednoduše možné tyto různé průběhy porovnávat s různě příbuznými průběhy.

Aproximace herních stromů byla dosažena především nově možnou metodou clustrování na větvích. Zde se vzaly vrcholy ve stejné hloubce a byly clustrovány podle jim přiřazených vektorů. Každý vrchol vždy reprezentoval daný počet podobných vrcholů a tím se růst stromu zpomalil až exponenciálně (samozřejmě se ztrátou informace). Tuto ztrátu informace ale často vyvažovala hloubka stromu, které bylo takto možno dosáhnout. Dále bylo ještě vyzkoušeno clustrování akcí, které snižovalo rychlost větvení.

Zvýšením počtu akcí, které jsou vyzkoušeny a zvýšením počtu clusterů (nebo úplným vypnutím clustrování na větvích) se dal jednoduše vytvořit hustý strom hry, který byl více deterministický a tedy na něm šlo přesněji určit například, jak důležitý je pro danou hru výběr akce. Snížením počtu akcí a agresivnějším clustrováním zase bylo možné vyzkoušet více různých map a mít poté různorodější generaci.

9.3 Další zjednodušení

Kvůli podstatnému zjednodušení herních pravidel pro účely simulace se naskýtala otázka, jestli má smysl původní poměrně obecný algoritmus hry stále používat. Místo toho se autor rozhodl použít nějaký specifický, jednodušeji testovatelný a lépe optimalizovaný algoritmu. Optimalizovaný algoritmus byl o mnoho rychlejší a také jednodušší na upravení.

Clustrování akcí nepřinášelo kýžené výsledky a tak se autor rozhodl jako alternativu vyzkoušet upravení pravidel hry v simulátoru. Například omezil pohyb na jen o jedno pole za kolo a rozdělil pozice na dva typy podle příkladu šachovnice. Na jednom typu (např. lichém) se mohou pohybovat jednotky a na druhém typu (např. sudém) se mohou vyskytovat kryty. Omezil se tím tedy počet možných tahů vojáka, které vedly jen z místa, kde nemá krytí, do místa, kde také krytý není. Také se tím zajistilo to, zda se dá na danou pozici pohnout nezávislé na krytech.

9.4 Optimalizování skrze změnu pravidel hry

Při generování map většinou začneme s danou hrou a do ní generujeme mapy. Pokud ale hru simulujeme a poté vytváříme mapy na základě výsledků simulace, není potřeba simulovat danou hru zcela přesně. Navíc díky tomu, že je Homeguard ve velmi raném stádiu vývoje, bylo možno provést úpravy přímo v pravidlech hry. Tím autor přiblížil pravidla hry pravidlům simulace.

Pravidla simulace byla původně zcela stejná jako pravidla hry Homeguard 1.2. Rozmanité akce mohou přidat na hráčském požitku. Při simulování průběhu hry je ale nadbytečné mít příliš podobné akce. Z toho důvodu byl ponechán jen jeden typ střelby a léčení bylo zcela odstraněno. Sprint byl nakonec nahrazen portály. Ty, stejně jako sprint, umožňují větší volnost pohybu. Na rozdíl od sprintu nezpůsobují velký nárůst počtu možných akcí. Lze je zadat součástí definice herní plochy před začátkem simulace. Jde na ně tedy použít podobný postup jako při generování map.

Pro zjednodušení simulace vždy proti sobě bojují dva týmy o dvou vojácích. To bylo racionalizováno tím, že hlavní složkou gameplaye je manévrování. Chceme se například dostat nepříteli za záda s jednou skupinou, zatímco druhá skupina ho palbou drží na místě. Ve dvou vojácích už takové manévry mají smysl.

Největší zjednodušení oproti původní hře je spojení výběru cíle s výběrem pozice pohybu. V simulacích týmů po dvou to způsobilo snížení počtu akcí na polovinu. Simulovaný hráč vždy musí po pohybu zaútočit na nepřítele, na kterého má nejlepší šanci zásahu. Při remíze vždy vybere nepřítele číslo jedna pro zachování determinismu. Toto zjednodušení navíc umožňuje přehledný výpis herního stromu. Stačí, aby šlo z výpisu vyčíst pozici vojáka a nepřítele, na kterého měl voják nejlepší šanci zásahu. Další možné opatření, od kterého nakonec bylo odstoupeno, bylo omezení pohybu jen směrem k nepříteli. Takové zjednodušení ale příliš omezilo používání portálů a možnosti manévrování. Jak bylo řečeno v úvodu, byla hra nakonec upravena tak, aby se tahy konaly najednou. Jediná změna, kterou to způsobilo, byla to, že umělá inteligence musí vždy brát najednou v potaz dvojice tahů obou vojáků simulovaného hráče. Pro zjednodušení se v simulaci přeskakují tahy, kde by skončili dva vojáci na vedlejších pozicích. Tím se zajistí, že se nikdy dva vojáci neposunou na stejnou pozici.

Měnění pravidel hry by se technicky dalo nahradit tím, že by simulovaný hráč vybíral jen mezi několika strategiemi. Tak to často dělávají reální hráči. Zjišťování strategií by ale vyžadovalo poměrně velkou míru práce herního vývojáře, což by šlo proti cílům C1 a C3. Pokud by tyto strategie šlo generovat strojově, bylo by pomocí nich možné celý tento proces zjednodušování pravidel nahradit.

Jelikož obě strany začínaly v rozích na stejné diagonále, nevyužívaly celý rozsah mapy. Problém autor zmírnil změnou startovních pozic nepřátel na rohy druhé diagonály. Pozice pro oživení se zanechaly na první diagonále. To znamená, že simulovaný hráč se musí vypořádat s útoky z různých směrů.

Pro vyzkoušení toho, jak bude postup fungovat na jiných hrách a zlepšení herního zážitku se nakonec autor rozhodl pro jednu poměrně velkou změnu. Místo toho, aby hráli vojáci postupně, se vojáci vždy najednou pohnuli a poté najednou vystřelili. To zjednodušilo simulaci a dle autorova názoru to udělalo taktiku hry složitější.

9.5 Optimalizování pomocí změn umělé inteligence

Bylo třeba zvolit umělou inteligenci tak, aby svými tahy nekazila reprezentativnost vůči originální hře. Zároveň ale musí vést simulaci k tvorbě dosti odlišných a informativních stromů. Samozřejmě by bylo možné umělou inteligenci nahradit druhým simulovaným hráčem, to by ale zdvojnásobilo rychlost větvení. Nakonec nebylo možné větvení zmírnit do takové míry, kde by toto bylo přípustné.

Základní umělá inteligence pracovala na principech udržování ideální vzdálenosti, pohybu vedle krytů, minimalizování šance na utržení zásahu a maximalizování šance na udělení zásahu. Tyto aspekty byly vždy zváženy různými vahami a výsledkem byla umělá inteligence pro hru. Jelikož prostor možných vah nebyl velký, byl celý prozkoumán. Takové umělé inteligence ale nebyly dostatečně univerzální. Byla spousta map, kde se umělá inteligence nehýbala nebo se jen pohybovala směrem přímo na hráče.

Autor se rozhodl pro dvě jednoduché umělé inteligence. Jedna hladově vždy udělala tah s nejlepším skóre po jejím tahu. Druhá spočítala tahy s maximálním minimálním skóre v dalším tahu přes tahy oponenta.

9.6 Optimalizování simulace beze změn pravidel

Ve finální verzi projektu, když už bylo jasné, že se pravidla nebudou měnit, se autor pokusil znovu implementovat algoritmus simulace bez alokování paměti na haldě. To by teoreticky mělo zvýšit lokalitu dat tím, že bude stále použita stejná část stacku a snížit nátlak na garbage collector. Dále autor použil méně objemné datové typy. Autor se také rozhodl před-počítat udělené poškození při střelbě mezi každou dvojicí pozic. Autor také ukládal stav hry jenom po akcích hráče a pro akce umělé inteligence ukládal jen index akce. Tímto vším autor zrychlil výpočet alespoň trojnásobně.

9.7 Práce s vnořeními

Metriky přímo používající herní strom byly například maximální dosažitelné skóre, průměrné dosažitelné skóre, nebo průměrná vzájemná informace mezi vektory akcí a konečným skóre. Autor také zkoušel použít různé kombinace takových metrik a metrik přímo pracujících s mapou. Metriky přímo používající herní strom nepřinášely výsledky. Ve snaze najít nový postup bylo rozhodnuto, že místo generování hodnocení z herního stromu se přesune pozornost na generování vnoření z herních stromů. Tato vnoření by poté byla použita k interaktivní generaci obsahu. První takové vnoření vytvořil autor manuálně z několika potencionálně reprezentativních vlastností. První taková vnoření byla například vektor průměrného maximálního dosažitelného skóre, nebo průměrná vzájemná informace mezi vektory h prvních akcí a konečným skóre pro každou hloubku h.

Pro interaktivní generaci byl nejdříve použit jednoduchý postup s euklidovskou vzdáleností, kde byly mapy s příliš blízkým vnořením mapám manuálně označeným za nevhodné také označeny za nevhodné. Nakonec se ale autor rozhodl použít pro předpovídání nevhodných pomocí ručně tvořených vektorů strojové učení. Byly vyzkoušeny rozhodovací stromy a mělké plně propojené neuronové sítě. Nakonec vybrané vlastnosti nebyly dost silné. Strojové učení z nich nebylo schopné rozlišit ani mezi různě parametrizovanými generátory.

Tyto experimenty autor opakoval pro mnoho různých parametrů algoritmu tvoření stromu a několika způsobech tvoření vnoření. Jelikož ale takové experimentování bylo značně časově náročné a neneslo výsledky, rozhodl se autor vyzkoušet jiný přístup. Autor se původně zdráhal použít hlubokého učení. Když si ale uvědomil, že postupy strojového učení na ručně tvořených vnořeních budou pravděpodobně vyžadovat ještě více testování, než bude potřebovat hluboké učení, než budou nést výsledky, rozhodl se pro hlubokého učení.

9.8 Hluboké učení

Autor se rozhodl pro tvorbu lepších vnoření zkusit hluboké učení. Bylo vyvinuto mnoho postupů, které využívají schopnosti neuronových sítí vytvářet reprezentativní vnoření, například [29]. Kvůli složitosti výběru hyperparametrů a architektury hlubokého učení se autor nejdříve rozhodl nápady otestovat na jednoduché hře odebírání čísel z okrajů řady. Hra byla velmi jednoduchá, a tak jí spíše použil k zavržení nejhorších postupů. Nakonec se jako nejlepší vnoření ukázala ta, vzniklá po použití neuronové sítě na posloupnosti vektorů reprezentujících větvě stromu hry. Šlo by také reprezentovat strom hry jako graf a použít algoritmy učení na grafech. Původně měl autor v plánu vyzkoušet pro tvoření vnoření velké množství pretext tasks. Nakonec se ukázalo, že doplňování samotné je dostatečné a že architektura sítě je pro výkon důležitější.

9.9 Eliminační algoritmy

Použití euklidovské vzdálenosti od průměru skupiny je pro nějaké typy vnoření nejlepší způsob porovnání podobnosti se skupinou. V autorových experimentech ale tento přístup nefungoval. Rozhodl se tedy navrhnout jednoduchou úpravu

tohoto postupu. Pro porovnání na jednoduché množině se lze podívat na 4.2 a 4.3. Je možné, že by použití nějakého jednoduchého algoritmu strojového učení bylo lepší. Autorovi se ovšem líbilo to, že deterministické řazení prvků zjednodušovalo testování.

9.10 Propojení eliminačního algoritmu s neuronovou sítí

Jelikož byla struktura neuronové sítě definována v Python knihovně, bylo jí třeba exportovat. K tomu autor použil formát ONNX a funkci `torch.onnx.export`. Tato funkce sleduje všechny operace, kterými projde vstupní tensor a uloží je jako výpočetní graf. Pro inferenci autor použil třídu `OnnxTransformer` [1]. Jelikož je model poměrně malý a inference zabírá zanedbatelný čas oproti tvoření stromu, není třeba zde používat nástroj CUDA. Jelikož Unity nepodporuje `OnnxTransformer` a jiné způsoby, které podporuje, nepodporují autorovu neuronovou síť, rozhodl se autor přidat implementaci inference jako oddělenou aplikaci (server). Tento server by mohl komunikovat skrze nějaký síťový protokol, ale autor se rozhodl pro čistě souborovou implementaci.

Ve finální verzi pro určení vnoření nové mapy pouze použijeme vnoření nejbližší předem připravené mapy. To bylo potřeba hlavně, protože simulování her bylo příliš pomalé na to, aby byl prováděn dostatečný počet simulací za běhu. Také by šlo upravit GA, aby při vytváření nových jedinců přímo bral mapy z předem připravené množiny. To by podrývalo jeho evoluční podstatu, ale mohlo by to například zvýšit různorodost generovaných map.

9.11 Simulované preference

Dále bylo třeba zajistit hodnoty, na kterých efektivnost algoritmu simulovat. Autor se rozhodl pro jednoduché metriky z jednoho článku 4.5. Bylo by pravděpodobně lepší použít více různých metrik. Článek ovšem shrnoval mnoho metrik z různých dalších článků a autor článku ukázal, že se mnoho nehodí pro jeho postup. Autor se rozhodl použít jen ty, které měly ve článku nejlepší výsledky. Dále by šlo použít nějaká ručně sestavená testovací kritéria. Autor ale doufal, že pokud bude postup dostatečně dobrý na jeho simulovaných preferencích, půjde ho vyhodnotit přímo v ukázkové aplikaci. To se nakonec ukázalo jako mylný předpoklad. Přesto se simulované preference ukázaly jako velmi užitečné při vyhodnocování postupů, jelikož umožnily zavrhnout postupy, které ani na nich nefungovaly.

9.12 Práce s daty

Jako největší bariéra k dobrým výsledkům se autorovi zdálo to, že trénovací data neuronové sítě nenesly dost informací. Původně autor pracoval pouze s indexy akcí a se skóre hry. Přidání maximálního dosažitelného skóre zvýšilo rychlost trénování i konečné výsledky, jak přesnost v doplňování, tak efektivitu

[1] Microsoft.ML.Transforms.Onnx.OnnxTransformer

eliminačního algoritmu. Autorovi se nakonec nepodařilo zjistit, jestli velmi husté mělké stromy vedou k lepším výsledkům, než řídké a hluboké stromy. Je možné, že jen nezkoušel dostatečně široký interval hloubek.

9.13 GA

Autor s GA neexperimentoval, jelikož ani na předem vybraných mapách algoritmus předpovídání preferencí neměl viditelné výsledky. Autorovi tedy přišlo zbytečné pokoušet se upravovat hyperparametry GA, když šance, že uvidí výsledek, byla mizivá.

9.14 Ukázková aplikace

Dále bylo po míře testování jasné, že bude nutno se vrátit k vizuálnímu návrhu hry Homeguard. Je velmi těžké objektivně hodnotit kvalitu map, když vizuálně nevypadají příliš lákavě. Bylo jasné, že procedurální generování 2d textur je nad rámec práce. Autor se rozhodl použít předem vytvořené assety. Byly použity assety dedikované veřejné doméně a assety s creative commons licencí. Žádné assety nebyly nijak upravené. Všechny nezmíněné zdroje jsou ve veřejném vlastnictví, některé vyžadují zmínku.

- `https://quaternius.itch.io/150-lowpoly-nature-models`
  dáno do veřejného vlastnictví
- `https://jaks.itch.io/low-poly-rifle-pack`
  jméno: low-poly-rifle-pack
  autor: `https://jaks.itch.io`
  licence: Creative Commons Attribution v4.0 International
  `https://creativecommons.org/licenses/by/4.0/deed.en`
- `https://elijahcobden.itch.io/stylized-rock-pack`
  jméno: stylized-rock-pack
  autor: `https://elijahcobden.itch.io`
  licence: Creative Commons Attribution v4.0 International
  `https://creativecommons.org/licenses/by/4.0/deed.en`
- `https://dblob-ua.itch.io/low-poly-bot-02`
  dáno do veřejného vlastnictví
  `https://www.pexels.com/photo/empty-brown-canvas-235985/`
  dáno do veřejného vlastnictví

9.15 Shrnutí práce na projektu

Velká část úsilí práce na projektu se soustředila na práci s metrikami, které přímo hodnotily strom hry. Autor nebyl schopný zlepšit výsledky lepšími metrikami, a tak se soustředil na generování stále reprezentativnějších a reprezentativnějších stromů. Tato data se nakonec ukázala užitečná v postupech strojového učení. Mnoho vybraných součástí konečného postupu by nějak šlo nahradit. Vybrané součásti ale buď neměly mnoho alternativ, nebo jejich alternativy byly velmi podobné. Například pokročilé eliminační algoritmy měly skoro stejné výsledky jako jednodušší a mnoho různých architektur hlubokého učení dosahovalo velmi podobných výsledků.

10 Závěr

Algoritmus na simulovaných preferencích poskytuje značné zlepšení oproti uspořádání za použití náhodných vnoření. Samozřejmě není jasné, jestli se projeví podobně kladně na záživnosti gameplaye. Vnoření algoritmu mají na simulovaných preferencích podobné výsledky jako jednoduchá vnoření používající pouze maximální dosažitelné skóre. Předběžné testování také zatím naznačuje, že se spíše neprojevuje pozitivně. I pokud postup pro tuto specifickou hru není vhodný, pro nějakou jinou by mohl fungovat dobře. Je také možné, že postup je pro praktické použití na jakékoliv hře stále příliš nedokonalý. Bude pravděpodobně velmi pracné zjistit, jestli je problém někde v páteřních konceptech postupu, nebo v nějakém detailu použití. V nějaké budoucí práci bude tedy třeba vyzkoušet nahradit části postupu a změřit, jak změní výsledky. Dále také velké množství parametrů postupu není přístupné, protože jsou v nějakých kombinacích nestabilní. Pokud bude nějaká další práce chtít tyto parametry použít, bude třeba zjistit, z čeho plyne jejich nestabilita.

10.1 Možné směry pro vylepšení

Autora napadají dva hlavní směry, kterými by šlo postup vylepšit. Můžeme vyzkoušet další možné struktury neuronových sítí a pro každou více hledat v prostoru hyperparametrů. Další možnost je použít jako vstup jiná data. Například navrhnout pro každý žánr data specifická pro daný žánr. Dále by šlo se zamyslet nad dalšími pretext tasky, které bychom pomocí takových dat mohli zkonstruovat.

10.2 Potenciální použití

Prozatím se algoritmus hodí pro hry s velmi malou rychlostí větvení. Pokud by se proces simulování herních průchodů podařilo zrychlit, nebo vytvořit dobré aproximace posloupnosti průběhů hry, má postup nespočet možných použití. Lze ho použít pro porovnávání her za účelem hodnocení práce vývojáře. Bylo by možné jednoduše ohodnotit výsledky libovolného generátoru libovolného typu obsahu. Například ho použít k upravování pravidel a generování zcela nových pravidel, generování map, designu questů a levelů a návrhu nepřátel. Dokonce by potenciálně mohl být použit už v prvních fázích vývoje pro zavrhnutí her se špatně navrženými pravidly.

10.3 Další bádání

Použití herního stromu pro vytvoření vnoření je obecný postup, jak vytvořit reprezentaci spojující informaci o mapě a o herních pravidlech. V budoucnosti bude pravděpodobně dále snaha vytvořit postupy generace, které si vytyčí cíle podobné jako ty, o které se snažila tato práce. Použití strojového učení k takovému generování map bude potřebovat nějak vzít v potaz pravidla hry. Není pravděpodobné, že by šlo najít algoritmus strojového učení, který by pravidla odvodil jen z hodnocení map. To především proto, že na rozdíl od problémů, jako

jsou generace obrazu nebo textu, nemáme většinou pro danou hru velké databáze hodnocených map.

10.4 Potencionální dopady

Algoritmy pro předpovídání preferencí dnes najde uživatel prakticky v každé velké aplikaci, se kterou se setká. Tyto algoritmy z velké části vedly k dnešní podobě internetu. Budeme-li takové algoritmy aplikovat nadměrně, mohlo by to mít negativní dopady. Představme si, že by vznikl dostatečně dobrý algoritmus pro předpovídání preferencí her. Ve video-herním průmyslu by mohl být využíván pro tvoření velkého množství gameplayově kvalitních, ale uměle vytvořených her. To by mohlo snížit tržní sílu her, které jsou vytvořeny více jako umělecké dílo než jako tržní produkt. To jsou dle autorova názoru právě ty hry, které řemeslo tvorby videoher nejvíce posouvají. Autorovi přijde, že jeho postup je od takového obecného algoritmu dost daleko. Pravděpodobně tedy v blízké budoucnosti nebude mít průzkum podobných postupů takovéto negativní dopady. Malé týmy vývojářů často nemají jako velké firmy kapacitu na to platit velké množství testerů a dělat průzkumy trhu. Algoritmy, které jsou z několika mála hodnocení schopné vytvořit mnoho umělých, by mohly malým týmům pomoci vytvářet hry, o které je zájem.

Literatura

1. Shaker, Noor; Togelius, Julian; Nelson, Mark J. *Procedural content generation in games*. Springer, 2016. ISBN 9783319427140.
2. Toy, Michael C; Arnold, Kenneth CRC. A Guide to the Dungeons of Doom. *Computer Systems Research Group. University of California, Berkeley. Nd Web*. 2012, roč. 9.
3. Harris, John. *Exploring roguelike games*. Boca Raton, United States: CRC Press, 2020. ISBN 9781003053576.
4. Browne, Cameron. *Automatic generation and evaluation of recombination games*. Queensland, Australie, 2009. Disertace. Faculty of Information Technology, Queensland University of Technology.
5. Eiben, Agoston E; Smith, James E. *Introduction to evolutionary computing*. Springer, 2015. ISBN 9783662448731.
6. Kim, Hwanhee; Lee, Seongtaek; Lee, Hyundong; Hahn, Teasung; Kang, Shinjin. Automatic Generation of Game Content using and Graph-based Wave Function Collapse Algorithm. In: *2019 IEEE Conference on Games (CoG)*. Institute of Electrical a Electronics Engineers, 2019, s. 1–4. Dostupné z DOI: 10.1109/CIG.2019.8848019.
7. Perlin, Ken. An Image Synthesizer. In: *Computer Graphics: Proceedings of Siggraph '85, 12th Annual Conference, San Francisco, USA*. New York, USA: Association for Computing Machinery, 1985, s. 287–296. ISSN 0097-8930.
8. Fournier, Alain; Fussell, Don; Carpenter, Loren. Computer rendering of stochastic models. *Communications of the ACM*. 1982, roč. 25, č. 6, s. 371–384. ISSN 0001-0782.
9. Johnson, Lawrence; Yannakakis, Georgios N.; Togelius, Julian. Cellular Automata for Real-Time Generation of Infinite Cave Levels. In: *Proceedings of the 2010 Workshop on Procedural Content Generation in Games, Monterey, Californie*. New York, United States: Foundations of Digital Games, 2010, s. 7–10. ISBN 9781450300230.
10. Dahlskog, Steve; Togelius, Julian; Nelson, Mark J. Linear levels through n-grams. In: *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content & Services*. Tampere, Finland: Association for Computing Machinery, 2014, s. 200–206. AcademicMindTrek '14. ISBN 9781450330060. Dostupné z DOI: 10.1145/2676467.2676506.
11. Summerville, Adam; Philip, Shweta; Mateas, Michael. MCMCTS PCG 4 SMB: Monte Carlo Tree Search to Guide Platformer Level Generation. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. 2021, roč. 11, č. 3, s. 68–74. Dostupné z DOI: 10.1609/aiide.v11i3.12816.

12. Rodriguez Torrado, Ruben; Khalifa, Ahmed; Cerny Green, Michael; Justesen, Niels; Risi, Sebastian; Togelius, Julian. Bootstrapping Conditional GANs for Video Game Level Generation. In: *2020 IEEE Conference on Games (CoG)*. 2020, s. 41–48. Dostupné z doi: 10.1109/CoG47356.2020.9231576.

13. Hastings, Erin J.; Stanley, Kenneth O. Galactic Arms Race: an experiment in evolving video game content. *SIGEVOlution*. 2010, roč. 4, č. 4, s. 2–10. Dostupné z doi: 10.1145/1810136.1810137.

14. Schrum, Jacob; Gutierrez, Jake; Volz, Vanessa; Liu, Jialin; Lucas, Simon; Risi, Sebastian. Interactive evolution and exploration within latent level-design space of generative adversarial networks. In: *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*. Cancún, Mexico: Association for Computing Machinery, 2020, s. 148–156. GECCO '20. isbn 9781450371285. Dostupné z doi: 10.1145/3377930.3389821.

15. Silver, David; Hubert, Thomas; Schrittwieser, Julian; Antonoglou, Ioannis; Lai, Matthew; Guez, Arthur; Lanctot, Marc; Sifre, Laurent; Kumaran, Dharshan; Graepel, Thore; další. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*. 2018, roč. 362, č. 6419, s. 1140–1144.

16. Vassiliades, Vassilis; Chatzilygeroudis, Konstantinos; Mouret, Jean-Baptiste. Using Centroidal Voronoi Tessellations to Scale Up the Multidimensional Archive of Phenotypic Elites Algorithm. *IEEE Transactions on Evolutionary Computation*. 2018, roč. 22, č. 4, s. 623–630. issn 1089-778X.

17. Bishop, Christopher M. Pattern recognition and machine learning. *Springer google schola*. 2006, roč. 2, s. 1122–1128. isbn 9788132209065.

18. Caron, Mathilde; Touvron, Hugo; Misra, Ishan; Jegou, Hervé; Mairal, Julien; Bojanowski, Piotr; Joulin, Armand. Emerging Properties in Self-Supervised Vision Transformers. In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Montreal, Kanada: Institute of Electrical a Electronics Engineers, 2021, s. 9630–9640. isbn 9781665428125.

19. Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N; Kaiser, Łukasz; Polosukhin, Illia. Attention is all you need. *Advances in neural information processing systems*. 2017, roč. 30.

20. Neumann, John von. 1. On the Theory of Games of Strategy. In: *Contributions to the Theory of Games (AM-40), Volume IV*. Princeton, USA: Princeton University Press, 1959, s. 13–42. isbn 9781400882168.

21. Neumann, J v. Zur theorie der gesellschaftsspiele. *Mathematische annalen*. 1928, roč. 100, č. 1, s. 295–320.

22. Buitinck, Lars; Louppe, Gilles; Blondel, Mathieu; další. API design for machine learning software: experiences from the scikit-learn project. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning, Praha, Česká Republika*. Praha, Česká Republika, 2013, s. 108–122.

23. Paszke, Adam; Gross, Sam; Massa, Francisco; další. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, s. 8024–8035. Dostupné také z: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

24. Harris, Charles R.; Millman, K. Jarrod; Walt, Stéfan J. van der; další. Array programming with NumPy. *Nature*. 2020, roč. 585, č. 7825, s. 357–362. Dostupné z doi: 10.1038/s41586-020-2649-2.

25. Hunter, J. D. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*. 2007, roč. 9, č. 3, s. 90–95. Dostupné z doi: 10.1109/MCSE.2007.55.

26. Virtanen, Pauli; Gommers, Ralf; Oliphant, Travis E.; další. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*. 2020, roč. 17, s. 261–272. Dostupné z doi: 10.1038/s41592-019-0686-2.

27. MacQueen, James; další. Some methods for classification and analysis of multivariate observations. In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Berkeley, USA: University of California Press, 1967, sv. 1, s. 281–297. Č. 14.

28. Kermack, William Ogilvy; McKendrick, Anderson G. A contribution to the mathematical theory of epidemics. *Proceedings of the royal society of london. Series A, Containing papers of a mathematical and physical character*. 1927, roč. 115, č. 772, s. 700–721.

29. Zoupanos, Spyros; Kolovos, Stratis; Kanavos, Athanasios; Papadimitriou, Orestis; Maragoudakis, Manolis. Efficient comparison of sentence embeddings. In: *Proceedings of the 12th Hellenic Conference on Artificial Intelligence*. 2022, s. 1–6.

30. Agarap, Abien Fred. Deep learning using rectified linear units. *arXiv preprint arXiv:1803.08375*. 2018.

31. Qi, Charles R; Su, Hao; Mo, Kaichun; Guibas, Leonidas J. Pointnet: Deep learning on point sets for 3d classification and segmentation. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, s. 652–660.

32. Kingma, Diederik P.; Welling, Max. Auto-Encoding Variational Bayes. In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. 2014. Dostupné z arXiv: http://arxiv.org/abs/1312.6114v10 [stat.ML].

33. Student. On the probable error of the mean. *Biometrika*. 1908, roč. 6, s. 1–25.

Seznam obrázků

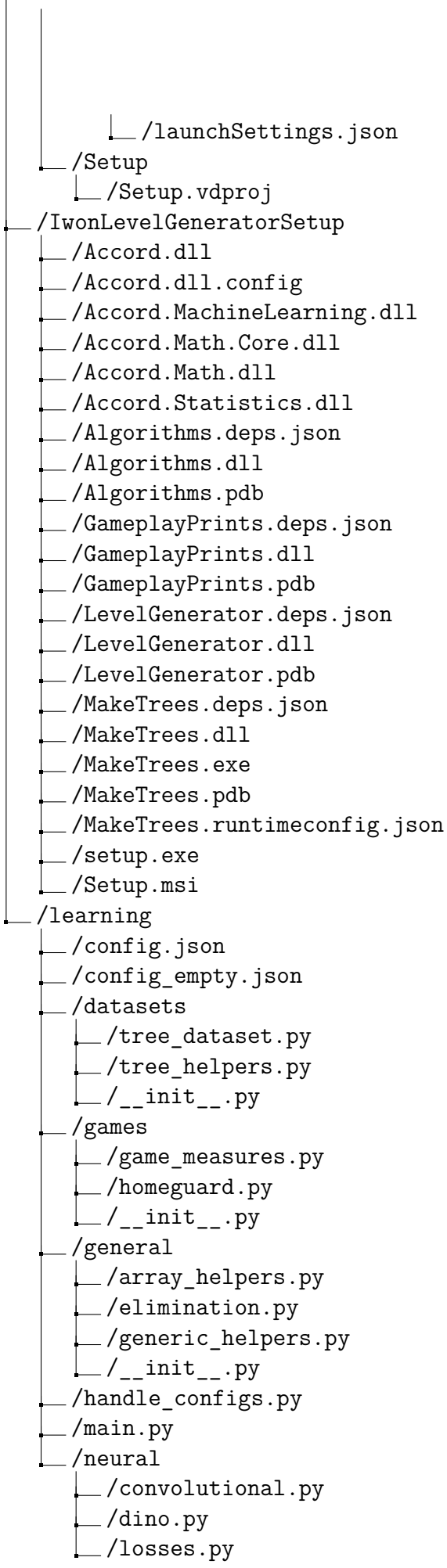
Seznam použitých zkratek

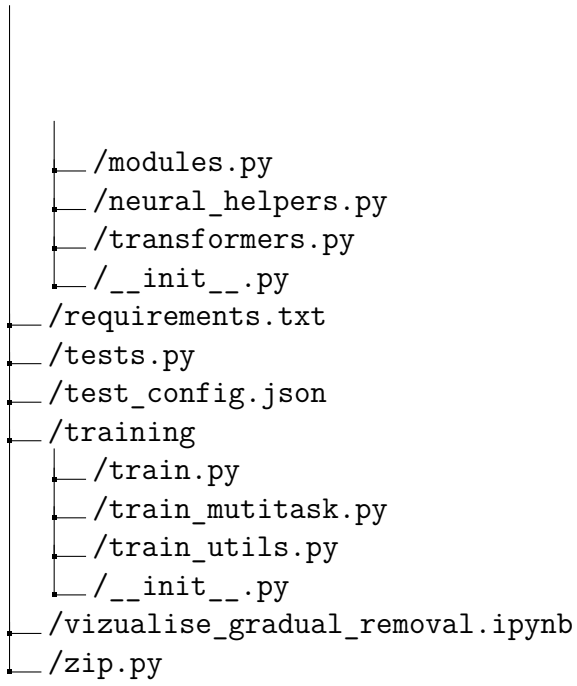
PCG — procedrální tvorba obsahu
GA — genetický algoritmus
CVT — centroidová Voroného teselace
MAP-Elites — algoritmus Multi-dimensional Archive of Phenotypic Elites
KL — Kullback-Leibnerova divergence
i.i.d. — Nezávislé stejně rozdělené náhodné veličiny
[n] — množina čísel {1, 2, …, n}
defaultně — ve výchozím nastavení

A Přílohy

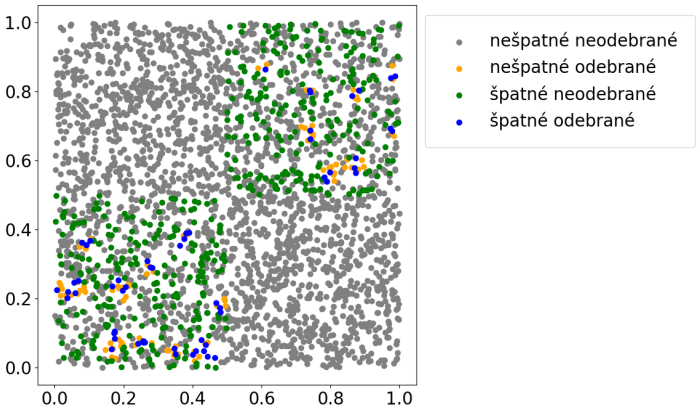
A.1 Obsah elektronické přílohy

- /HomeguardLearning.ipynb
- /main_results_learning.txt
- /test_results_learning.txt
- /test_results_level_generator.txt
- /IwonLevelGenerator
  - /Algorithms
    - /Algorithms.csproj
    - /DataSci.cs
    - /DataStructures
      - /Array2D.cs
      - /EnumerableTuples.cs
      - /Grid2D.cs
      - /Vector2.cs
    - /EnumerableExtensions.cs
    - /KMeans.cs
    - /Mathematics.cs
    - /PrintsConvert.cs
  - /GamePlayPrints
    - /CVTMapElites.cs
    - /EvolutionUsingElimination.cs
    - /GameplayPrints.csproj
    - /GameTreeMakerGeneric.cs
    - /ManualEmbeddings.cs
    - /Removal.cs
  - /IwonLevelGenerator
    - /GameTreeMaker
      - /DcDescriptorFourSoldiers.cs
      - /Judge.cs
      - /TreeGrower.cs
      - /TreeMaker.cs
    - /LevelGenerator.csproj
    - /LevelGenerators
      - /LevelGeneratorCheckers.cs
      - /LevelGeneratorLines.cs
    - /Prints
      - /DescriptorMaker.cs
      - /LevelGenerator.cs
      - /LevelGeneratorForLearning.cs
  - /IwonLevelGenerator.sln
  - /MakeTrees
    - /MakeTrees.csproj
    - /Program.cs
    - /Properties

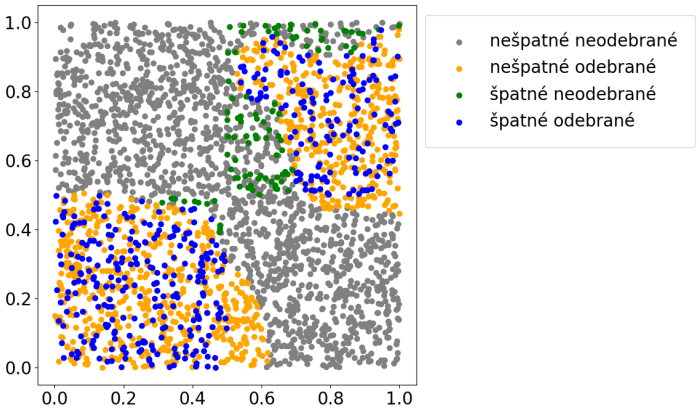

/launchSettings.json
/Setup
/Setup.vdproj
/IwonLevelGeneratorSetup
/Accord.dll
/Accord.dll.config
/Accord.MachineLearning.dll
/Accord.Math.Core.dll
/Accord.Math.dll
/Accord.Statistics.dll
/Algorithms.deps.json
/Algorithms.dll
/Algorithms.pdb
/GameplayPrints.deps.json
/GameplayPrints.dll
/GameplayPrints.pdb
/LevelGenerator.deps.json
/LevelGenerator.dll
/LevelGenerator.pdb
/MakeTrees.deps.json
/MakeTrees.dll
/MakeTrees.exe
/MakeTrees.pdb
/MakeTrees.runtimeconfig.json
/setup.exe
/Setup.msi
/learning
/config.json
/config_empty.json
/datasets
/tree_dataset.py
/tree_helpers.py
/__init__.py
/games
/game_measures.py
/homeguard.py
/__init__.py
/general
/array_helpers.py
/elimination.py
/generic_helpers.py
/__init__.py
/handle_configs.py
/main.py
/neural
/convolutional.py
/dino.py
/losses.py


/modules.py
/neural_helpers.py
/transformers.py
/__init__.py
/requirements.txt
/tests.py
/test_config.json
/training
/train.py
/train_mutitask.py
/train_utils.py
/__init__.py
/vizualise_gradual_removal.ipynb
/zip.py

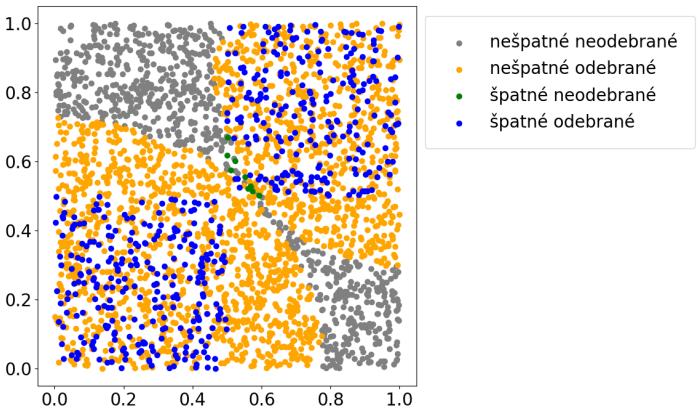
B Dodatečné grafy



(a) Stav chvíli po inicializaci.

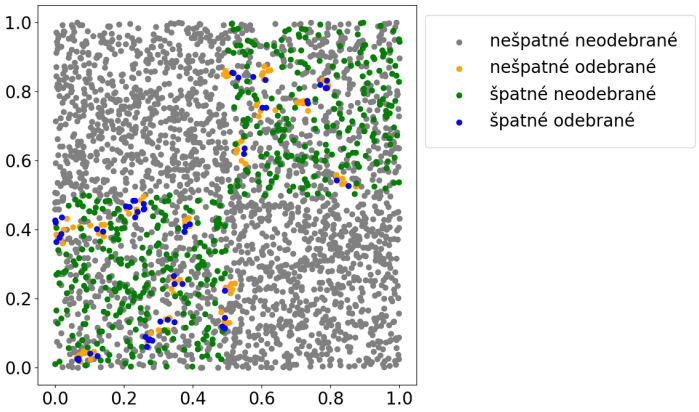


(b) Většina špatných byla odebraná.

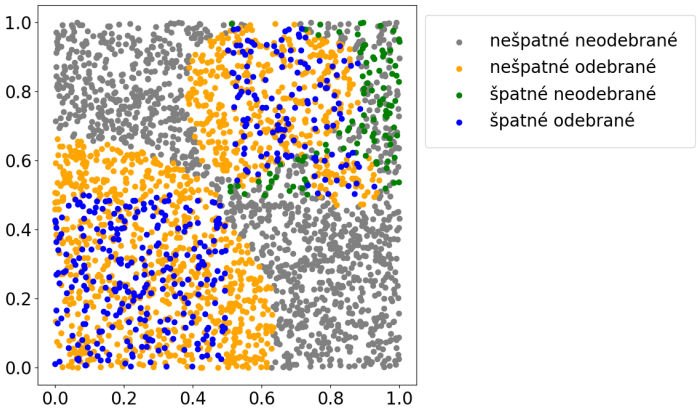


(c) Téměř všechny špatné byly odebrány.

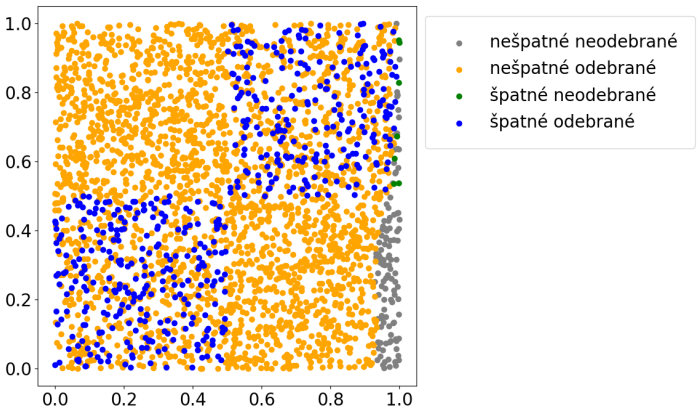
Obrázek B.1 Eliminace třetím algoritmem s teplotou 1. Vstup 20 známých z 200 špatných z 1000 bodů. Špatné pouze v levém dolním a pravém horním kvadrantu.



(a) Stav chvíli po inicializaci.

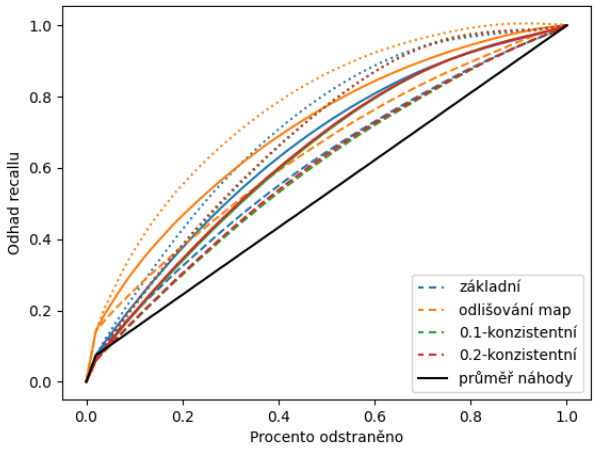


(b) Většina špatných byla odebraná.

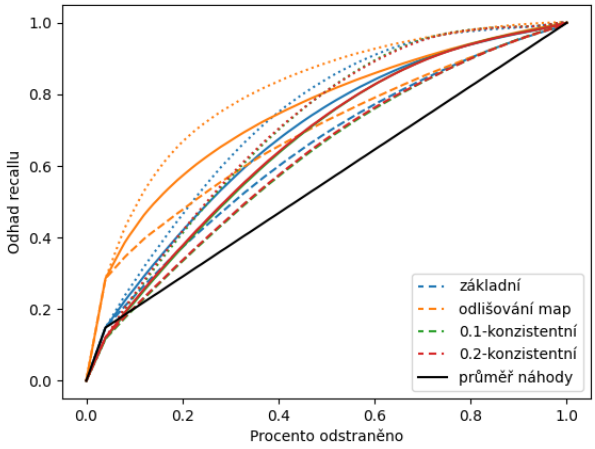


(c) Téměř všechny špatné byly odebrány.

Obrázek B.2 Eliminace čtvrtým algoritmem. Vstup 20 známých z 200 špatných z 1000 bodů. Špatné pouze v levém dolním a pravém horním kvadrantu.

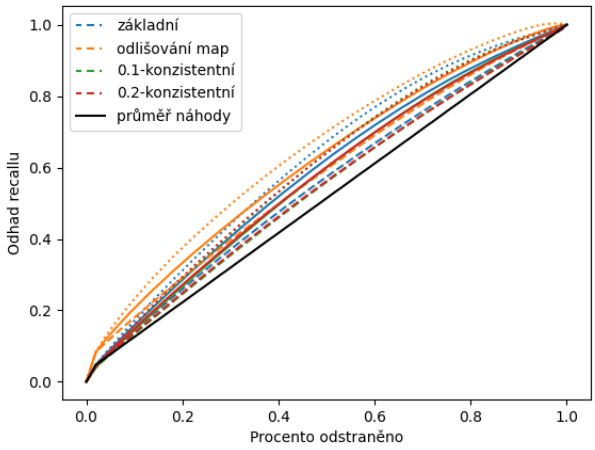


(a) Výsledky testů $f = 40$ $p = 15$.

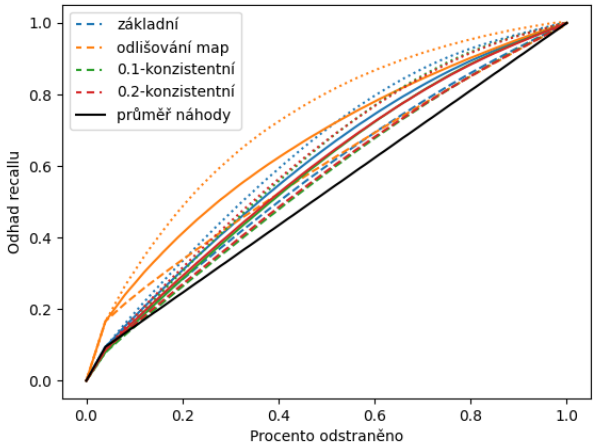


(b) Výsledky testů $f = 80$ $p = 15$.

Obrázek B.3 Výsledky testů 7.5 $p = 15$. Ukazuje přímo procento správně odebraných namísto zlomku zlepšení.

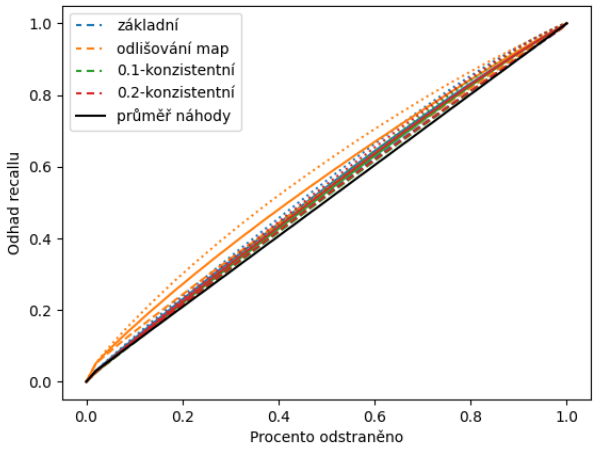


(a) Výsledky testů $f = 40$ $p = 25$.

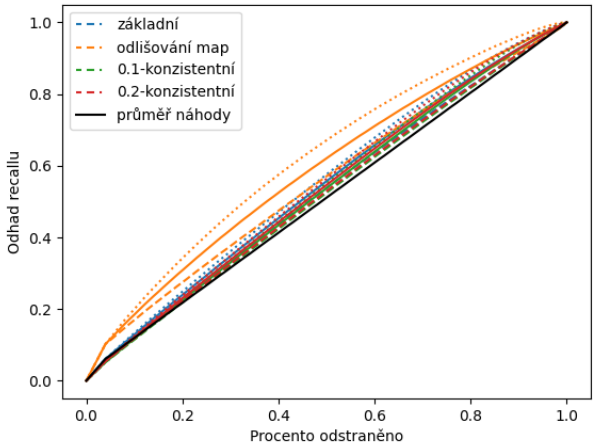


(b) Výsledky testů $f = 80$ $p = 25$

Obrázek B.4 Výsledky testů 7.5 $p = 25$. Ukazuje přímo procento správně odebraných namísto zlomku zlepšení.



(a) Výsledky testů $f = 40$ $p = 40$.



(b) Výsledky testů $f = 80$ $p = 40$.

Obrázek B.5 Výsledky testů 7.5 $p = 40$. Ukazuje přímo procento správně odebraných namísto zlomku zlepšení.

C Ukázky map

Zbarvená políčka značí dvojice portálů. Je trochu matoucí, že světle zeleně zbarvená políčka značí možné akce vojáka na tahu. Jelikož autor není schopen poznat rozdíl mezi mapami ani po velkém počtu hodnocení, rozhodl se použít mapy už po malém počtu hodnocení.


40/40-
40/40-
40
40/40-
40/40-
40


40/40-
40
40/40-
40