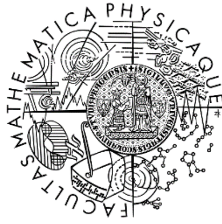


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Matyáš Halíř

Využití strojového učení pro tvorbu umělé inteligence v turn-based hře

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.

Studijní program: Informatika (B0613A140006)

Studijní obor: Informatika se specializací
Počítačová grafika, vidění a vývoj her

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval 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 Praze dne .... podpis

Rád bych zde poděkoval svému vedoucímu práce RNDr. Martinu Pergelovi, Ph.D. za jeho trpělivost s dokončením práce, cennými radami jak pro programovací, tak psanou část a skvělý přístup. Nadále bych chtěl poděkovat kolegům Pavlu Šindelářovi, Michalu Popkovi a Silvii Paprskářové, jejichž rady k formátování, programu či oběma zároveň značně pomohly s postupem práce. Na závěr bych chtěl také poděkovat svému tatínkovi a babičce, kteří mne v průběhu celé práce podporovali.

Název práce: Využití strojového učení pro tvorbu umělé inteligence v turn-based hře

Autor: Matyáš Halíř

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.

Abstrakt: Programování umělé inteligence je důležitou a často poměrně dlouhou součástí vývoje her, která se velkým dílem podílí na koncové kvalitě samotné hry. Se vzestupem technologií strojového učení se naskýtá otázka, zda tyto technologie nemají své místo i při vývoji her. V této práci se soustředíme na genetické algoritmy a jejich potenciál pro využití oproti pevně zakódované umělé inteligenci pro tahové strategie. Navrhneme hru, na které se umělá inteligence bude učit, následně pro tuto hru implementujeme genetický algoritmus optimalizující umělé inteligence pomocí matice a na závěr provedeme analýzu výsledků na dvou různých případech průběhu algoritmu a k těmto výsledkům podáme možné hypotézy a prozkoumáme možné navazující práce.

Klíčová slova: genetický algoritmus, strojové učení, vývoj her

Title: Machine-learning methods in artificial-intelligence design for a turn-based game

Author: Matyáš Halíř

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D.

Abstract: The programming of artificial intelligences is an important and often quite a lengthy part of game development, which greatly contributes to the overall end quality of the game. With the rise of machine learning technologies comes the question of if these technologies have their place in game development. In this thesis we focus on genetic algorithms and their use potential over hard-coded artificial intelligences for turn-based strategies. We'll design a game where the artificial intelligence will learn, then we will implement a genetic algorithm which optimizes the artificial intelligences based on a matrix and we'll analyse the results of two separate runs of this algorithm and pose hypotheses based on the results while examining potential future work.

Keywords: genetic algorithm, machine learning, game development

Obsah

1. ÚVOD 1
1.1. Umělá inteligence 3
1.2. Soudobé metody 4
1.3. Související práce 4
1.4. Genetický algoritmus 5

2. IMPLEMENTACE HRY 6
2.1. Pravidla hry 6
2.2. Gameboard 6
2.3. Generace map 7
2.4. Využité optimalizace 7
2.5. Statické AI 8

3. IMPLEMENTACE GENETICKÉHO ALGORITMU 10
3.1. Ohodnocení generace 10
3.2. Křížení a mutace 12

4. UŽIVATELSKÁ DOKUMENTACE 13
4.1. Úprava konfigurace souboru 13
4.2. Formát vstupu/výstupu 14

5. PROGRAMÁTORSKÁ DOKUMENTACE 16
5.1. Math.NET Numerics 16
5.2. Actions.cs 16
5.3. Config.cs, GAConfig.cs 16
5.4. GAInterpreter.cs 17
5.5. Gameboard.cs 17
5.6. GeneticAlgo.cs 19
5.7. IAI.cs 19
5.8. SoldierInstance.cs 20
5.9. PrimitiveTests.cs 20
5.10. SoldierClass.cs 20

5.11. Stats.cs ______ **20**
5.12. WeaponClass.cs ______ **21**

6. **PROJEKTOVÁ DOKUMENTACE** ______ **22**

7. **EXPERIMENTY A ANALÝZA VÝSLEDKŮ** ______ **23**
7.1. První experiment ______ **23**
7.2. Druhý experiment ______ **24**
7.3. Analýza výsledků ______ **26**

8. **ZÁVĚR** ______ **28**
8.1. Budoucí práce ______ **28**

9. **SEZNAM POUŽITÉ LITERATURY** ______ **29**

1. Úvod

Pro hry s aspekty hraní proti počítači (*PvE*, *player versus environment*) je umělá inteligence jedním z nejdůležitějších faktorů kvality hry. O to více toto platí ve strategických hrách, kde hráč hraje proti počítači. I přesto, že se obtížnost hry dá ovlivnit i jinými faktory, například změnou vlastností nepřátel, je kvalita umělé inteligence protivníka důležitá: Nekvalitní umělá inteligence může hráče vytrhnout ze hry a tím z ní zkazit celkový pocit.

1) Tahová strategie jako počítačová hra: XCOM 2

Zdroj: snímek obrazovky

2) Tahová strategie: Go
Zdroj: Wikipedia.org

Tato práce se zabývá využitím metod strojového učení (machine learning), specificky genetických algoritmů a jejich možného využití pro nahrazení natvrdo zakódované umělé inteligence v tahové strategii. Jinými slovy, provádíme experiment, zda se umělá inteligence sama dokáže naučit hrát lépe, než naše zakódovaná umělá inteligence hraním proti ostatním verzím sebe sama a zda konverguje k výsledkům, které jsou lepší oproti naší umělé inteligenci.

Možnost naučení umělé inteligence dostatečně dobře hrát tahové strategie by umožnila jejich využití ve hrách namísto natvrdo kódovaných inteligencí, potenciálně snížila počet vývojářů pro tyto hry a také zkrátila dobu vývoje umělých inteligencí. Mimo tahové strategie by se podobný princip dal potenciálně využít i u her v reálném čase, obzvlášť ve hrách, kdy umělá inteligence ovládá například jednotlivé postavy

v jednotkách (pro situace, kdy chceme, aby se postavy nepohybovaly ve formaci), nebo pro analýzu stolních her.

1.1. Umělá inteligence

Obecně lze umělou inteligenci definovat jako „studium systémů, které se chovají jakýmkoliv způsobem tak, aby jakémukoliv pozorovateli připadaly inteligentní“. (Ben Coppin, [1]). Podobnou definici můžeme najít i v dnešní době: “Artificial intelligence is about making computers able to perform the thinking tasks that humans and animals are capable of.” [2]. V kontextu vývoje her se lze odkázat na definici z knihy AI for game developers [3]: “Anything that gives the illusion of intelligence to an appropriate level, thus making the game more immersive, challenging, and, most importantly, fun, can be considered game AI.“

Mezi dalšími definicemi lze najít i takové, které zahrnují hledání nejkratších cest či detekci kolizí [3], jakožto i mnohem užší definice (Silné AI, [1]). Pro kvalifikaci našeho případu použijeme definice prostředí z knihy Artificial Intelligence: A modern approach [4], které jsou přiloženy v tabulce níže.

Plně pozorovatelné	Částečně pozorovatelné
Deterministické	Stochastické
Diskrétní	Spojité
Známé	Neznámé
Nekonkurenční	Konkurenční

3) Tabulka – Definice prostředí hry dle [4]

Zdroj tabulky: Hyrkäs Jarno [5], přeloženo

Mluvíme o tedy o umělé inteligenci v poměrně širokém kontextu. Pro tuto práci se zaměříme na umělou inteligenci ovládající jednotlivé postavy (Agenty) v kontextu stochastické tahové strategie. Jelikož se jedná o tahovou strategii, dá se říci, že se jedná o diskrétní, sekvenční prostředí. Jak bude rozvinuto v dalších kapitolách, v našem případě bude prostředí plně pozorovatelné.

1.2. Soudobé metody

Využití metod strojového učení je v praxi stále omezené. V deterministických hrách (např. šachy) jsou velmi populární algoritmy založené na prohledávání stavových stromů. Nejsilnějšími takovými algoritmy jsou Stockfish [6] a Komodo [7], u kterých se musí podotknout, že Komodo prochází stavový strom s pomocí Monte Carlo heuristiky. Tyto umělé inteligence dokáží porazit, nebo remizovat s nejlepšími hráči i na domácích počítačích. V posledních letech se do obrazu dostávají i neuronové sítě, mezi kterými je nutno zmínit AlphaZero [8], která byla schopna enginy porazit jen na základě samoučení. V roce 2020 začaly jak Stockfish, tak Komodo neuronové sítě také využívat.

V počítačových hrách se stále nejčastěji vyskytují variace na stavové automaty. Příkladem tohoto přístupu mohou být hry jako například XCOM 2, kde hráči ovšem přišli na to, jak v některých stavech AI reaguje s velmi vysokou přesností. Tato možnost AI navést na některé akce se stává značnou nevýhodou ve chvíli, kdy je možné nepřítele navést, aby podnikl nějakou akci, o které hráč ví, že vždy nastane. Toto je jedna z největších limitací stavových automatů u komplexnějších sad možných akcí; tím hráč může ztratit zájem o hru. V praxi však zůstávají velmi populární vzhledem k jejich jednoduché implementaci, testování a stabilitě.

Jedním z takovýchto případů právě ze hry XCOM 2 může být AI pro nepřítele typu „Archon“. Pokud nevidí nepřítele, který by pro něj byl odkrytý, pak vždy použije svůj speciální útok, jenž ale nemá efekt v daném kole. Jiným podobným nepřítelem je „Sectoid“, kde lze zamezit využití jeho speciálního útoku nebo střelby, pokud může oživit mrtvého vojáka a nemohl by střílet na vojáka z pozice, kdy by voják byl odkrytý.

1.3. Související práce

Jednoduché metody strojového učení včetně genetických algoritmů mají své kořeny ve vývoji her před více jak dvaceti lety, například v již zmíněné knize AI for Game Developers [3]. Jejich využití v praxi však stále není časté vzhledem k předem zmíněným faktorům. V dnešní době se rozmáhají metody hlubokého učení [9], reinforcement learningu [10] a neuronových sítí [11], avšak stále s malým využitím v praxi.

Pokud se ovšem neomezíme jen na vývoj umělé inteligence do tahových strategií, můžeme najít práce, které umělou inteligenci využívají jak pro vylepšení rekordů v závodních hrách [12] anebo pro testování [13], nebo pro generaci map do her [14], čímž se zabývá i práce jednoho z mých kolegů.

1.4. Genetický algoritmus

Naše práce se zaměřuje na genetické algoritmy ze dvou důvodů: Jedním je nápad, že není nutné používat stromy chování. Stačí nám využití matic pro seřazení vhodnosti jednotlivých akcí. Druhým je, že se jedná o poměrně jednoduchou metodu, kterou lze lehce naprogramovat a následně využívat bez vysokých nákladů na počítač koncového uživatele.

Nejprve se zaměříme na samotný engine hry a jeho implementaci, následně na genetický algoritmus a jeho vlastní implementaci, spolu s problémy implementace pro samostatné učení. Pokračovat budeme nástinem samotného programu a jeho dokumentací. Nakonec se zaměříme na výsledky trénování a zakončíme porovnáním s našimi cíli a analýzou výsledků.

2. Implementace hry

Původní implementaci programu bylo nutno přepsat vzhledem k problémům způsobeným rozpadem týmu, který hru psal, a tím dané degradaci kódu, který byl ponechán v neuspokojivém stavu pro tuto práci.

Engine samotný byl sepsán v čistě kódové podobě pro zrychlení simulace hry a možnost nasadit jej bez kompletního přepsání do jakéhokoliv herního enginu podporujícího C# (nebo naprogramovat vlastní) beze změn v logice programu, s většinou změn týkajících se pouze interakcí s vizuální stránkou a vlastním enginem. Čistě kódové řešení napomáhá zrychlit simulaci, jelikož některé enginy (např. Unity) nejsou schopny dobře podporovat masivní paralelizaci kvůli své architektuře a ztráceli bychom výkon na samotném enginu.

2.1. Pravidla hry

Hra je specifikována jako odehrávající se na čtvercové mapě předem dané velikosti, kde každý čtverec mapy má danou výšku 0 až 7. Vojáci se mohou pohybovat maximálně o jeden výškový bod nahoru a neomezeně dolů.

Možnost střelby se řídí simulovaným raycastem z pozice vojáka nebo vyhovujících (viz pozdější sekce) sousedních pozic do pozic sousedních druhému vojákovi. Pokud jakákoliv kombinace pozic zakončí úspěšným raycastem, voják může na druhého střílet. Raycast samotný využívá definice mapy dle políček, díky čemuž můžeme provádět výpočty průniků jen na hranách jednotlivých čtverců, což umožňuje využít adaptované standardní algoritmy na kreslení čar.

2.2. Gameboard

Nejdůležitější součástí řešení je třída Gameboard. Ta funguje jako řídící komponenta zodpovědná za vytvoření a průběh hry, předávání možnosti vykonat jednotlivé akce a jejich vykonávání. Designová filozofie této třídy je její funkčnost jako „gamemaster“- tedy tato třída jako jediná modifikuje a vykonává akce ve hře, zatímco ostatní třídy mají velmi omezené kompetence. Také se v ní nachází pro danou hru jediný používaný generátor náhodných čísel.

2.3. Generace map

Automatická generace map byla navržena tak, aby generovala zcela symetrické mapy, které jsou před přidáním krytů pro vojáky zcela přístupné. Tímto zajištujeme, že ať začíná jakákoliv strana, tak se nám nestane, že by jedna strana měla poziční výhodu. Pole se generují po vlnách dle jejich Manhattanské vzdálenosti od rohu na pozici (0,0) a jsou následně zrcadleny na druhou stranu mapy. Přístupnost zde definujeme jako podmínku, že nově generované pole musí být výškově maximálně o jedna vzdálené od všech svých sousedů, kde lze dokázat, že tuto mapu můžeme vygenerovat (důkaz v dokumentaci). Pro náš případ jsou mapy generované velikosti 24x24.

Následně na tuto vygenerovanou mapu přidáme předem definované úkryty, které tuto souvislost mohou lokálně narušit. Vzhledem k hlídání sousedních polí, zda se na nich již kryt nenachází, stále neporušíme možnost přesouvat se po celé mapě. Díky této podmínce mohou být vojáci rozmístěni v opačných rozích a nestane se nám, že by se někteří nemohli pohybovat.

Generace mapy by šla provést jinými způsoby včetně využití strojového učení (jeden z mých kolegů má práci na toto téma, ale vzhledem k tématu práce jsem vybral rychlou a efektivní metodu).

2.4. Využité optimalizace

Vzhledem k architektuře hry je Gameboard také jedna z nejnáročnějších tříd na CPU, a proto využívá caching ve dvou často volaných metodách. První z těchto metod je caching map vzdáleností, které se používají k získání přístupných polí a jejich vzdáleností pro pohyb vojáků. Samotná generace vzdálenostních map je provedena s pomocí využití Djikstrova algoritmu na grafu možných přesunů, kde vzdálenost každého pohybu je 1.

Druhou metodou, která využívá caching, je samotný raycasting mezi pozicemi. Vzhledem k překrývání těchto raycastů pro hledání pozic pro střelbu je nejlepší metodou caching výsledku samotného jednoho raycastu místo cachingu možností střelby.

Alternativní přístup předgenerováním v obou případech byl zamítnut vzhledem k dlouhému času inicializace a možnému nepřistoupení ke všem cachovaným metodám v průběhu jedné hry. Nutno podotknout, že větší zrychlení bylo dosáhnuto cachingem vzdálenostních map, což samo bylo schopné změnit rychlost algoritmu z hodin na generaci na generace za hodinu.

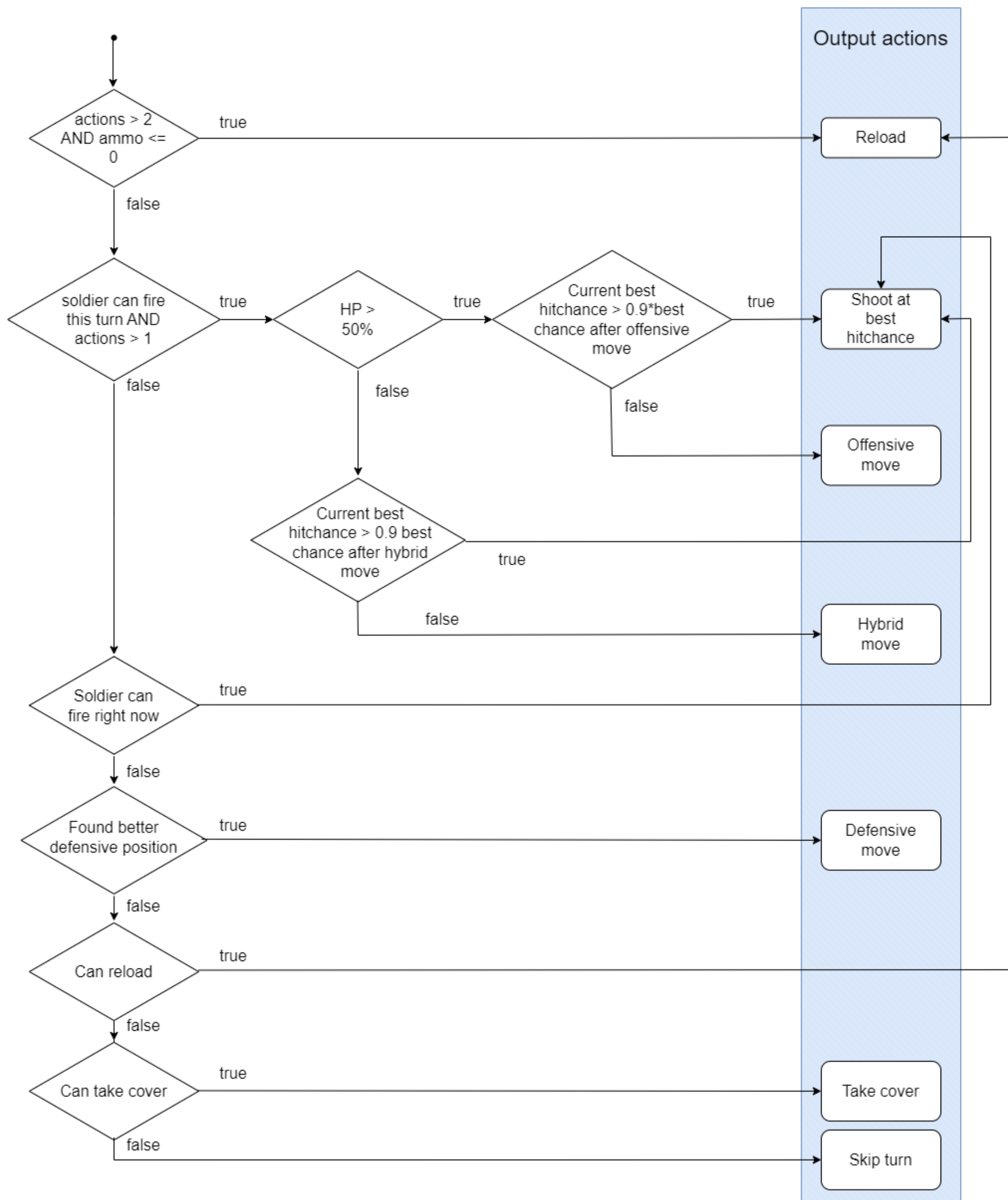
Nevýhodou těchto metod je jejich nutné přepsání, pokud by byla přidána možnost měnit terén. V té situaci by tyto metody byly méně efektivní, ale zachovaly by si lepší vlastnosti než předpočítávání při každé změně mapy.

2.5. Statické AI

Samotné AI, proti kterému měříme námi vygenerovanou inteligenci, je variantou na stavový automat, který vykonává předem vybrané akce, u nichž zná některé parametry. Tato architektura byla vybrána vzhledem k její popularitě v praxi.

Akce, ze kterých může vybírat, jsou: přebití zbraně, střelba na cíl s nejlepší šancí zásahu, přesun do pozice maximalizující šanci střelit nějaký cíl (přesun ponechává alespoň jednu akci na střelbu), přesun na pozici s vysokou průměrnou ochranou proti střelbě, přesun na pozici, která se snaží maximalizovat oba předchozí aspekty, zvýšení obrany vojáka, pokud je v krytu. V případě, že jinou akci nelze nalézt, voják přeskočí svůj tah.

Umělou inteligenci (AI) nazýváme statickou, jelikož se v našem kontextu na rozdíl od AI generovaného genetickým algoritmem nemění. Statické AI v této práci definujeme jako umělé inteligence, které nemají možnosti učení.



4) Stavový automat statického AI

3. Implementace genetického algoritmu

Principem celého genetického algoritmu je násobení vektoru vstupních dat s maticí, kterou generuje genetický algoritmus. Tyto hodnoty jsou využity jako hodnoty vhodnosti dané akce, které jsou stejné jako pro statické AI. Tato metoda je odlišná od více zkoumaných metod strojového učení využívajících genetické algoritmy díky tomuto využití matic.

Podobné práce využívají jiné metody, například stromy chování [15], nebo jednodušší optimalizace vah jednotlivých akcí pro hraní real-time strategií jako tahových, kde jsou časové intervaly využity pro abstrakci hry na tahovou strategii [16]. Tyto metody by se pro náš případ daly také použít, ale vzhledem k možné výhodě, kterou by skýtala možnost porovnání vhodnosti akcí byla vybrána metoda optimalizace matice. Tím se přibližujeme architektuře zmiňované v práci Maurice Bergsma a Pietera Sproncka [17]. Je nutno zmínit, že by zde šlo optimalizovat i pohyb hráče samostatně místo vybírání akcí, čímž se jiné práce v kontextu tahových strategií také zabývají [18].

Při programování byly mimo již dříve zmíněné zdroje využity sekce o genetických algoritmech z knih od Laurence Moroneyho [19], Francese Buontempa [20] a M. Tima Jonese [21].

Samotný genetický algoritmus byl naprogramován na principu samoučení, tedy bez možnosti porovnávání externího fitness value. Tímto zaručujeme, že pokud je tato metoda vhodná pro vytvoření vylepšeného AI oproti automatu popsanému v minulé sekci, nebude nutné programovat tento automat v ostatních hrách, pokud bude použita tato metoda učení.

3.1. Ohodnocení generace

Tento přístup má ale zásadní problém. Oproti standardnímu využití genetických algoritmů nemůžeme využít globálního ohodnocení fitness value, jelikož není vůči čemu porovnávat. Pro správné relativní porovnání hodnot tedy musíme ohodnotit každého jedince s každým jiným, jinak bychom nemohli spolehlivě zjistit, které matice jsou oproti ostatním lepší.

Samotné ohodnocení je prováděno sadami her mezi každým párem odlišných jedinců, kde se při každé hře mění, kdo začíná jako první. Za výhru jednotlivé hry dostává AI 2 body, nevyhraje-li ani jedna strana do časového limitu, dostávají obě strany 1 bod a za prohru nedostávají žádný. V algoritmu je následně možnost tuto hodnotu umocnit na druhou mocninu, pokud chceme, aby algoritmus byl více nakloněn lepším řešením. Tato hodnota je nastavitelná v konfiguraci.

Je nutné zde podotknout, že toto ohodnocení se liší od standardního 3-1-0 ve sportech. Důvod je jednoduchý: pokud bychom měli dvě umělé inteligence, které jsou stejně dobré (50% prohra, 50% výhra) a dvě, které skoro vždy remizují, pak bychom vzhledem k celkovému přidání bodů nadhodnocovali první pár. Možnost jiné konfigurace je však přidána do konfigurace souboru.

V tomto kroku je důležitá paralelizace. Na moderním procesoru s velkým množstvím logických jader jsme schopni zrychlit výpočet o více jak jeden řád, což spolu s optimalizací samotné hry umožňuje sledovat postup samotného algoritmu v reálném čase i na domácích počítačích. Samotné výpočty her neparalelizujeme vzhledem k limitacím context switchingu. Na našem domácím počítači toto zhoršilo čas na generaci i přes lepší teoretické využití procesoru.

Nejlepší matice z generace je následně ohodnocena oproti statickému AI ve velkém počtu her. Zde lze také využít paralelizaci, nyní na jednotlivé hry. Tato matice je i se svým počtem výher následně uložena do souboru formátu .txt, ze kterého je možné matici rekonstruovat a zjistit, kolikaprocentní byla její výhernost. Hry, které skončí remízou, z výsledku vynecháme, jelikož z ohodnocení nevíme, jakým způsobem by hra skončila. Více ohodnocení by bylo možných, ale tím se dostáváme do optimalizace více cílů, které nelze jednoznačně seřadit.

Následuje vytvoření další generace, kde podle poměru rodičů a nových matic vybereme nejlepší matice této generace, které zůstanou a doplníme novými maticemi vytvořenými křížením starých matic. Iterace skončí ve chvíli, kdy dosáhneme buď daných výsledků (zde vybráno jako výhra 70 % her proti našemu statickému AI), nebo daného počtu generací. Důvod vybrání takto vysoké výhernosti je, že tím snižujeme šanci, že algoritmus dosáhne takového výsledku pouhou šancí.

3.2. Křížení a mutace

Selekce pro křížení probíhá selekcí dvou různých rodičů z vážené rulety. Zde existuje více možností, ale pokud již máme relativní fitness value, selekce z vážené rulety není o nic horší než jakýkoliv jiný druh selekce, jelikož pro výpočet relativního fitness value využíváme porovnání všech metod proti sobě. Pro zamezení nekonečného opakování je v selekci úniková varianta, kdy při opakovaném vybírání té samé hodnoty vybereme nejlepšího souseda podle fitness value. To není nutné, ale pro situace, kdy jedna hodnota značně převyšuje ostatní, bychom tím mohli značně snížit různorodost populace. Turnajová selekce by zde byla možná, ale nezbavíme se tím nutnosti vypočítat všechny hry. Proto tuto metodu nepoužíváme i přes její ekvivalentní výsledky.

Samotné křížení probíhá selekcí po jednotlivých hodnotách v matici, kde každý rodič má stejnou šanci zanechat svůj gen. Postupem, který zde byl vyzkoušen a zamítnut, bylo křížení po sloupcích matice, jelikož každá hodnota může podstatně ovlivnit výpočet (například pokud v jedné matici máme lepší váhu pro šanci zásahu, ale druhá má zbylé lepší, pak bychom zbytečně přidávali restrikci na výběr). Místo toho máme nyní šanci vybrat všechny ‚lepší' hodnoty.

Šance mutace je také nastavitelná pro oba typy v práci. Prvním z těchto typů je malá mutace, která má šanci nastat jednou pro každé pole. Dodanou součástí tohoto algoritmu je annealing: jinak řečeno, v konfiguraci je možné zapnout, zda se s pokračujícím procentem z celkového maximálního počtu generací bude snižovat šance této mutace až na limitní hodnotu pro poslední generaci.

Druhým typem mutace je ‚velká' mutace, která zamění celé sloupce matice. Tato mutace je mnohem vzácnější (jen jednou pro každou matici) a reprezentuje záměnu výpočtů vhodností jednotlivých akcí. Pro tuto mutaci annealing neprovádíme.

4. Uživatelská dokumentace

Jelikož je náš program navržen pro využití vývojářem hry, můžeme usoudit, že jsou schopni zkompilovat program a případně jej editovat před samotným použitím.

Samotný program je navržen jako konzolová aplikace, kterou lze spustit z jediného bodu. Tím myslíme zkompilovaný program z přiloženého souboru, který je námi cílený uživatel schopen zkompilovat a následně spustit. Tím se automaticky spustí algoritmus, který se bude učit na instanci hry z námi definovaného souboru. Po zakončení algoritmu bude tento konec vypsán na konec konzolového výstupu. V tuto chvíli lze program zakončit dle instrukcí v konzoli.

Dokumentace pro hru bude přiložena spolu s odkazem na hru, jelikož se nejedná pouze o mé vlastní dílo a z toho důvodu ji nemohu přiložit jako soubor k tomuto dílu.

4.1. Úprava konfigurace souboru

Pro úpravu konfigurace souboru nám slouží dvě statické třídy uvnitř programu: Třída *Config* v souboru *Config.cs* a *GAConfig* v souboru *GAConfig.cs*. Tyto třídy v sobě mají konstanty, na něž se celá hra odkazuje. Komentáře v kódu jsou psány anglicky.

Editací konstant samotné hry jsme schopni vytvořit různé verze hry běžící na stejném enginu, které mohou mít různá nastavení. Samotné konstanty jsou popsané komentáři v kódu, kde jsou i doporučené a testované limity pro správný chod funkcí (mimo ně nemůžeme garantovat chod programu). Zde se tedy zastavíme nad možnostmi, které nám tato nastavení nabízí.

Změnou konstant v souboru *Config.cs* před kompilací můžeme nastavit:

- Konstanty pro samotný gameboard, zde obranné bonusy vojáků
- Statistiky vojáků využitých pro simulaci a jejich zbraní
- Pozměnit výpočet přesnosti zbraně vojáků
- Změnit maximální a minimální výšku mapy a výskyt krytů na mapě
- Změnit velikost mapy.

Změnou konstant genetického algoritmu jsme schopni změnit chod samotného učení a priorit pro program. Tyto možnosti jsou:

- Počet her pro ohodnocení jak relativního fitness value, tak proti statickému AI
- Poměr rodičů a dětí v nové generaci
- Maximální počet generací a procentuální výhernost proti statickému AI pro předčasné ukončení
- Počet vojáků pro obě ohodnocení zvlášť
- Velikost generace
- Úprava fitness value umocněním
- Možnost annealingu
- Nastavení ohodnocení pro generaci.

4.2. Formát vstupu/výstupu

Formát souboru s maticí pro vstup/výstup je soubor formátu .txt s jednotlivými položkami oddělenými novým řádkem. Ty jsou v tomto pořadí:

- Počet řádek matice (1 řádek)
- Počet sloupců matice (1 řádek)
- Jednotlivé položky v matici (desetinná čísla) s tečkou mezi celou a desetinnou částí, seřazené po řádcích (počet řádků závisí na množství položek v matici, i těch nulových)
- Pokud ukládáme výhernost, následuje řádek „Winrate“ (1 řádek)
- Výhernost ve formě desetinného čísla (1 řádek).

3	*Počet řádků matice*
2	*Počet sloupců matice*
-9.531748644163184	*Položka matice na pozici [1,1]*
8.94721474113365	*Položka matice na pozici [1,2]*
0.47671614981244126	*Položka matice na pozici [2,1]*
0.6748752049692825	*Položka matice na pozici [2,2]*
-0.7731763909044211	*Položka matice na pozici [3,1]*
-5.875104761084344	*Položka matice na pozici [3,2]*
Winrate	*Textový řetězec „Winrate“*
0.47	*Výhernost této matice proti statickému AI v daném kole*

5) Příklad datového formátu pro uloženou matici velikosti 3x2. Hodnoty tučně bezpatkově, popisky kurzívou.

Pro správnou funkčnost formátu by na žádném řádku neměly být mezery, tabulátory ani jiné znaky až na nové řádky. Jakékoliv znaky za takto definovaným souborem budou ignorovány a soubor bude přijat.

5. Programátorská dokumentace

V této části se zaměříme na dokumentaci důležitých částí programu pro údržbu, praktické využití a úpravu pro jiné účely. Navíc se zaměříme na jedinou vybranou knihovnu a důvod jejího výběru a následně na jednotlivé součásti programu po souborech.

5.1. Math.NET Numerics

Tato knihovna byla vybrána vzhledem k podpoře libovolně velkých matic a jednoduché konverze mezi polem a maticí. Toto je rozdílné oproti podobným knihovnám včetně systémových, které matice libovolných velikostí nepodporují. Samotná systémová knihovna navíc nepodporuje ani matice datového typu double, čímž dále omezuje své využití.

5.2. Actions.cs

Akce jsou psány přes společný interface IAction, kde pro každou akci je v enum ActionType zavedena samostatná položka. Každá funkce má vlastnosti cooldown a remainingCooldown popsané v komentářích samotného interface, zatímco metody IsPossible, TryExecute a Execute jsou zodpovědné za samotné fungování akce.

Tělo metody Execute a metoda, která se věnuje možnosti akci vykonat, by měly být napsány do třídy Gameboard, čímž splníme design hry, který jsme zmínili v jedné z předchozích kapitol. Pro přidání této akce do schopností vojáka musíme samostatnou instanci přiřadit do akcí daného vojáka. Pokud bychom sdíleli instance, můžeme docílit globální, nebo týmové limitace dané akce.

5.3. Config.cs, GAConfig.cs

Tyto třídy obsahují statické proměnné pro konfiguraci samotné hry, respektive genetického algoritmu. Pro bližší popis jednotlivých proměnných a možných změn v konfiguraci (spolu s limity hodnot) odkazuji na uživatelskou dokumentaci a komentáře v kódu.

5.4. GAInterpreter.cs

Soubor GAInterpreter obsahuje dvě třídy: GeneticAlgoInterpreter a GAUtils. Třída GAInterpreter slouží pro interpretování AI matice jako umělé inteligence, skládající se z funkcí definovaných v interface IAI, funkce pro vykonání jedné akce a vykonání správné akce dle indexu.

Pro přidání dalších akcí by se musela rozšířit jak matice, tak funkce execActionByIndex. Zde je důležité dodat, že vzhledem k využití knihovny Math.NET Numerics lze zadat libovolně velkou matici. Je na programátorovi, aby zajistil správnou velikost.

Druhou třídou v tomto souboru je statická třída GAUtils, která umožňuje načítání a ukládání matic. Pro specifikaci formátu souboru se odkazuji na uživatelskou dokumentaci.

5.5. Gameboard.cs

Třída Gameboard slouží jako hlavní centrum důležitých funkcí celé simulace a engine celé hry. Je zcela zodpovědná za vytváření map, vykonávání akcí a zodpovídání otázek jako jsou vzdálenostní mapy, nebo možnosti střelby.

Funkce CanMove, CanShootEnemy, CanReload a CanTakeCover jsou všechno funkce, které odpovídají, zda je daná akce ve jméně funkce možná.

Funkce ExecuteReload, ExecuteMove, ExecuteTakeCover, ExecuteSkip a ExecuteShoot tyto akce vykonají, pokud jsou splněny podmínky jejich vykonání, jinak vyhodí výjimku kvůli jejich nemožnosti.

Střílení je provedeno pomocí raycastů z pozice vojáka a sousedních pozic vzdálenosti 1 oběma směry do pozice nepřítele a jeho sousedních pozic vzdálenosti 1 jen směrem z pozice nepřítele.

Raycasting samotný využívá faktu, že se nacházíme v prostředí sestaveném pouze z krychlí: Tedy, nemusíme se starat o souvislý raycasting, stačí nám projít všechny hrany, kde bychom mohli --- a vypočítat, zda jsou tyto průsečíky správně. Nadále můžeme tyto raycasty zjednodušit odebráním výškových raycastů, jelikož ty budou pokryty jednou ze sousedních hran.

Pro získání šance se trefit z dané pozice na danou pozici existuje funkce GetHitModifiers, která pro integraci do hry vydává kromě listu všech statistik měnících šanci zásahu i jejich jména. Tato funkce také má overload pro střelbu z dané pozice.

Pro získání vzdálenostních mapy z dané pozice lze zavolat funkci GetDistanceMap, která je implementací Djikstrova algoritmu v grafu sousedností s délkami hran 1.

Tato funkce je rozšířena funkcí HasRouteBetween, která odpovídá, zda existuje cesta mezi dvěma pozicemi a zpětně posílá i danou cestu. Pro tuto funkci existuje volitelný parametr maximální délky cesty.

Pro správné vytvoření nového gameboardu bez manuálního zásahu je nutno zavolat funkci SetUpGame, která seřadí dané vojáky pro hru, přiřadí jim zadanou umělou inteligenci a následně vytvoří novou mapu pomocí funkce SetMap. První se vygeneruje spojitá mapa pomocí podmínky, že každá výška může být maximálně o 1 vzdálena sousedním vygenerovaným výškám.

Pro ověření, že tato mapa bude spojitá, nám stačí uvažovat generaci z jednoho rohu do středu, jelikož mapu zrcadlíme. Protože generujeme dle manhattanské vzdálenosti z rohu, víme, že máme maximálně dvě pole, která byla vygenerována, a tím ovlivňují nové pole. Jelikož ale tato pole mají jednoho společného rodiče, jejich výška se nemůže lišit o více než dva, a tedy podle pravidel existuje hodnota taková, která je od obou maximálně o jedna vzdálena. Pro zajímavost: důkaz se dá rozšířit i na středovou symetrii místo osové, i když tím se zde nebudeme zabývat.

Následně jsou na tuto mapu přidány kryty tak, aby zachovaly prostupnost pomocí pravidla, že při generaci krytů se žádné dva nemohou dotýkat v maximové metrice. Proto vždy existuje cesta mezi každými dvěma kryty. Jelikož mapa není nutně v rozích spojitá (a zde by mohla nastat chyba v generaci pro rohy), zakazujeme generaci krytů na okrajích mapy a v manhattanské vzdálenosti 2 od rohu mapy vzhledem k možným tvarům krytů. Následně provádíme korekci po diagonále, kde mohou vzniknout neopustitelné pozice.

Na závěr zde zmíníme, jak je vyřešena sekce posloupnosti tahů vojáků. Tato sekce je řešena tak, že na začátku funkce SetUpGame rovnoměrně rozmístí vojáky obou stran s tím, že první vždy bude voják v listu greenTeam. Při smrti jakéhokoliv vojáka

je daný index odebrán z obou listů a případně je provedena korekce momentálního indexu. Toto je zajištěno ve funkci ProcessDeath.

5.6. GeneticAlgo.cs

Většina této sekce byla prodiskutována v sekci implementace, zde jen rozšíříme některé důležité body.

Vyhodnocení matice probíhá hrou každého AI s každým jiným. Funkce TrainNewAI a EvalWithBaseAI využívají funkce Parallel.For pro paralelní zpracování a tím výsledné velké zrychlení výpočtů. Jelikož ukládáme výsledky samostatně pro každé párování, funkce nemusí používat zámky mimo těch na samotnou konzoli, kde se vypisuje status programu. Funkce EvaluatePairing provádí samotný výpočet a prohledávání.

Jelikož byla tato část poměrně detailně popsána ve zbytku práce, zbylou dokumentaci lze najít v sekci o genetickém algoritmu.

5.7. IAI.cs

Soubor IAI.cs obsahuje primitivní interface pro možnost změn umělé inteligence ve hře, statické třídy AIUtils a ExecutionUtils a nakonec napevno zakódované AI ve třídě AIAutomaton.

Interface samotný obsahuje metody executeTurn, executeMove a copyThisAI. První dvě metody reprezentují potenciálně odlišné metody využití interface pro hru, které by vzhledem k designu metody měly z většiny běžet uvnitř třídy Gameboard.

Třída AIUtils umožňuje hledání samotných jednoduchých pohybů pro AI (přesun na nejlepší útočnou/obrannou pozici mezi jinými – jména funkcí popisují jejich funkci), zatímco ExecutionUtils pomáhá překlenout mezeru mezi akcemi a AI jednotlivých vojáků vyhledáním a vykonáním dané akce.

AI a schéma jeho automatu lze najít v sekci Implementace hry.

5.8. SoldierInstance.cs

Tato třída nám slouží jako jednotlivá instance vojáka, která dohromady má všechny jeho důležité akce. Jedná se o lehce rozšířený container pro vojáka, který v sobě má několik metod, které se používají jako utilities. Lehkou vadou na této třídě a potenciální budoucí změnou je přepsání pozice z dvou jednotlivých proměnných na tuple, který je standardem ve zbytku programu.

5.9. PrimitiveTests.cs

Pro otestování funkce enginu slouží obsah souboru PrimitiveTests.cs. Všechny testy spouštíme funkcí RunTests, která spustí kategorie testů, jež pokrývají většinu enginu (ne samotný genetický algoritmus). Tyto testy jsou rozděleny do čtyř kategorií: line-of-sight testy, possibility testy, execution testy a nakonec herní testy. Každá kategorie provádí více testů najednou a vydá výsledky za celou sekci, případně i s částí, kde se stala první chyba (po chybě testy nepokračují pro zbytek kategorie).

Spolu s těmito metodami se v souboru nachází statická třída TestUtils, ve které se nacházejí pomocné funkce pro testy. Testy nebyly navrženy s pomocí testovacích frameworků kvůli brzké interpretaci a pro zlehčení implementace do herního enginu i přes výhody, které by unit testing přinesl včetně možnosti lépe uzavřít většinu metod.

5.10.SoldierClass.cs

Třída SoldierClass byla navržena jako možné rozšíření případné hry do strategické části, kde SoldierInstance přebírá hlavní roli pro jednotlivou hru, SoldierClass funguje jako kontejner pro neměnná data samotného vojáka: nezměněné statistiky, jakou zbraň má a jeho akce.

5.11.Stats.cs

Třída Stats je data carrier pro důležité statistiky jednotlivých vojáků a při případném rozšíření také jejich vybavení. Samotná třída nabízí možnost sčítaní dvou instancí a negace jedné, díky čemuž lze provádět základní aritmetiku.

5.12.WeaponClass.cs

WeaponClass je celkově koncipována jako třída, ve které můžeme definovat zbraně vojáka. V této třídě jsou uloženy statistiky dané zbraně, výpočet přesnosti a jak moc uškodí protivníkovi, pokud se trefí. Nejdůležitějším bodem, kde mohou nastat problémy, je výpočet přesnosti, který se řídí vzdáleností od optima a přidává/odebírá přesnost od vojáka (pro snížení je nutné mít nastavené negativní hodnoty v daných proměnných).

6. Projektová dokumentace

V této části se budeme více neformálně věnovat některým rozhodnutím, výběru algoritmů a obecně součástem programu, o jejichž vývoji by mělo být řečeno více.

První z těchto sekcí je výběr primitivního testování oproti jiným metodám. Tato metoda byla využita ze dvou hlavních důvodů: jednoduchost integrace do herních enginů, jelikož informace o testech pro herní enginy nejsou dobře dostupné a také relativně malé zkušenosti s jejich využitím. Vzhledem k problémům, které ale tímto rozhodnutím nastaly (projekt je mnohem více otevřený, než by bylo vhodné) nemůžeme sekci považovat za optimální řešení. Pokud bych měl více času, zaměřil bych se na tuto součást mnohem více.

Dalším důležitým rozhodnutím je caching. Jak již bylo řečeno dříve, caching dvou specifických funkcí násobně zrychlil funkčnost programu. Ten byl implementován po zkušebním spuštění genetického algoritmu, kdy performance profiler ukazoval, že v určitých funkcích program stráví dohromady přes 90 % času. Vzhledem k těmto výsledkům by to znamenalo, že i první experiment, diskutovaný v další sekci, by běžel zhruba měsíc.

Dalším problémem, který nastal již před minulým zmíněným problémem, je nevyužití interfaces pro některé třídy: U zbraní a vojáků se tímto ztrácí určitá variabilita, která snižuje možnosti pro další vývoj. Pro zbraně toto není velká nevýhoda, ovšem vzhledem k architektuře programu, převzaté ze stolních her, by se pro přidání jiných typů jednotek (třeba dronů) musela změnit koncepce třídy soldierInstance.

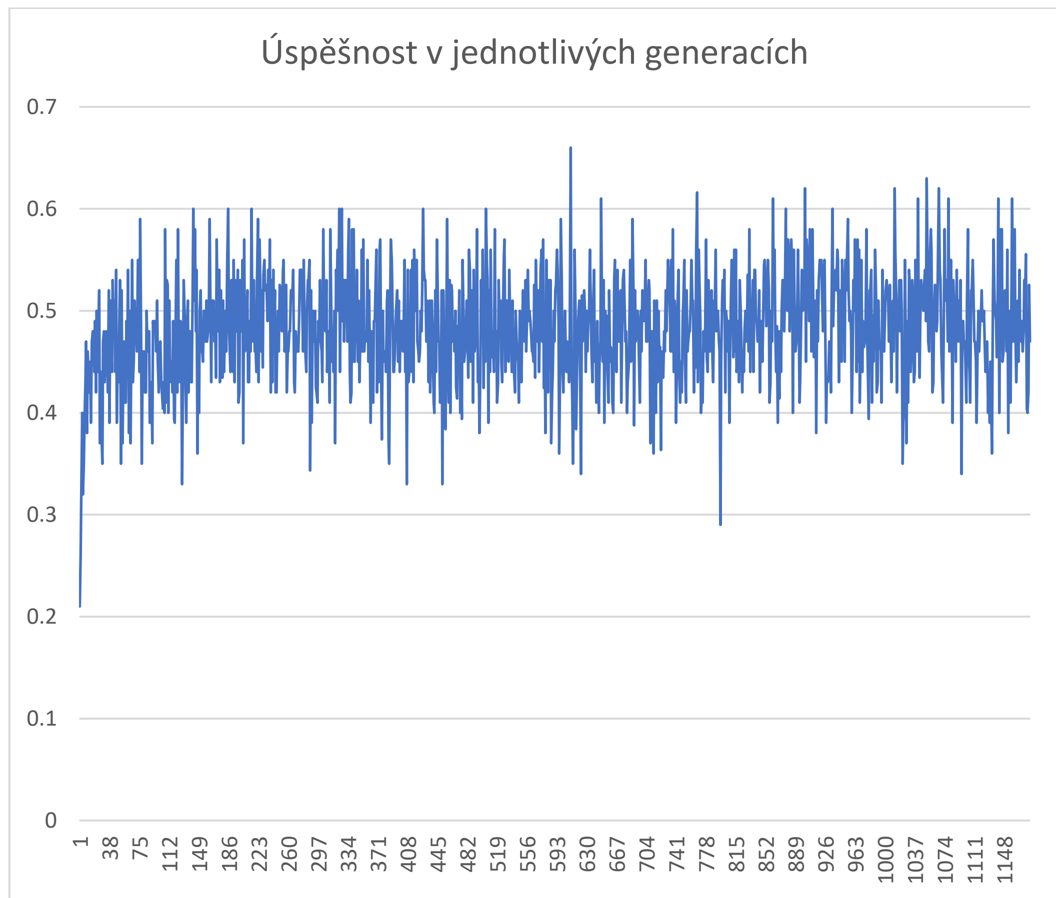
Nadále by měla být zmíněna poněkud nepřehledná část využívání akcí umělou inteligencí. Tuto součást by šlo nejspíše naimplementovat lépe, vzhledem k nutnosti překladu přes třídu ExecutionUtils.

Posledním bodem, kdy tento program narazil na poměrně velké nevýhody, o kterých nebylo moc řečeno, je samotné využití knihoven: Knihovny nehrají nejlépe s herními enginy, v některých případech se musí kvůli nim instalovat extenze editorů.

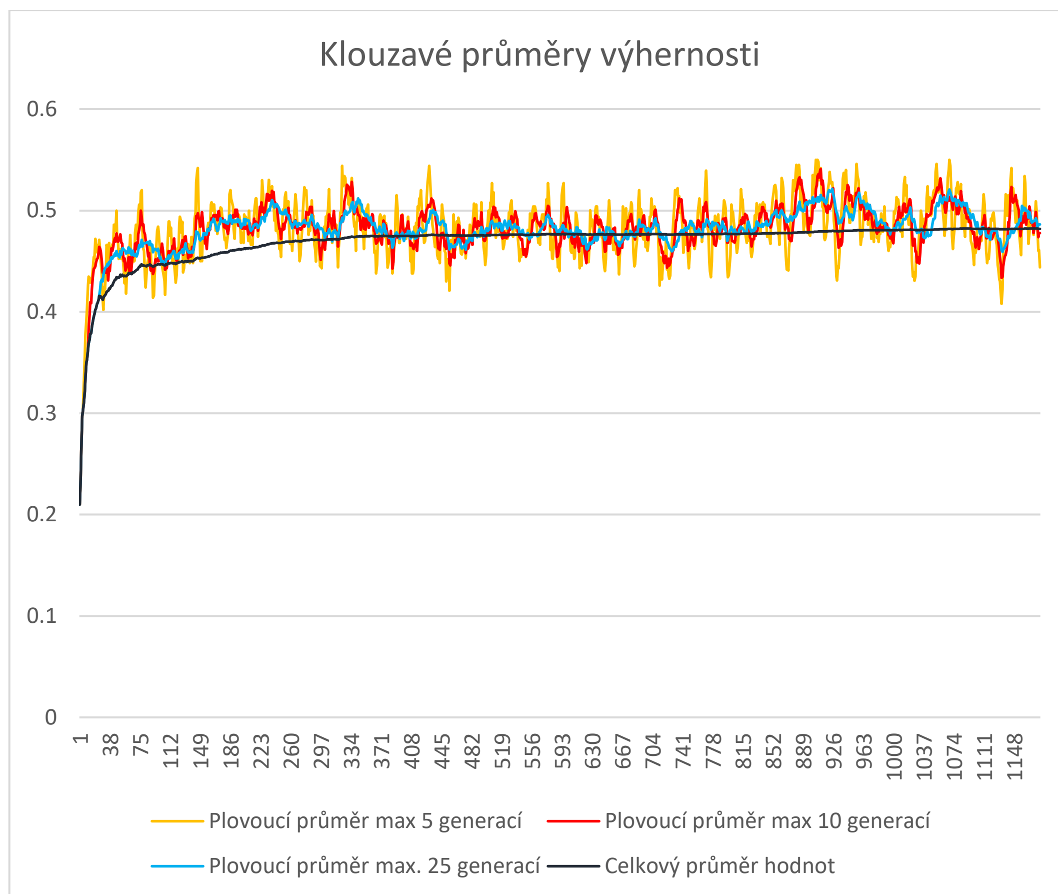
7. Experimenty a analýza výsledků

7.1. První experiment

Při prvním běhu algoritmu nebyl annealing využit, fitness value bylo umocněno a algoritmus běžel na generacích velikosti 25 jedinců po dobu 1180 generací. Během této doby nebyla překonána daná výhernost po dobu 45 hodin a 38 minut na moderním, 8jádrovém procesoru (16 logických jader) s taktovací frekvencí přes 4.5 GHz. Z toho získáváme průměrný čas cca 2 minut 19 s na generaci na tomto procesoru. Následující grafy ukazují výhernost nejlepšího jedince v dané generaci a následně průměrné hodnoty výsledků v algoritmu.



6) Spojnicový graf vývoje úspěšnosti po generacích

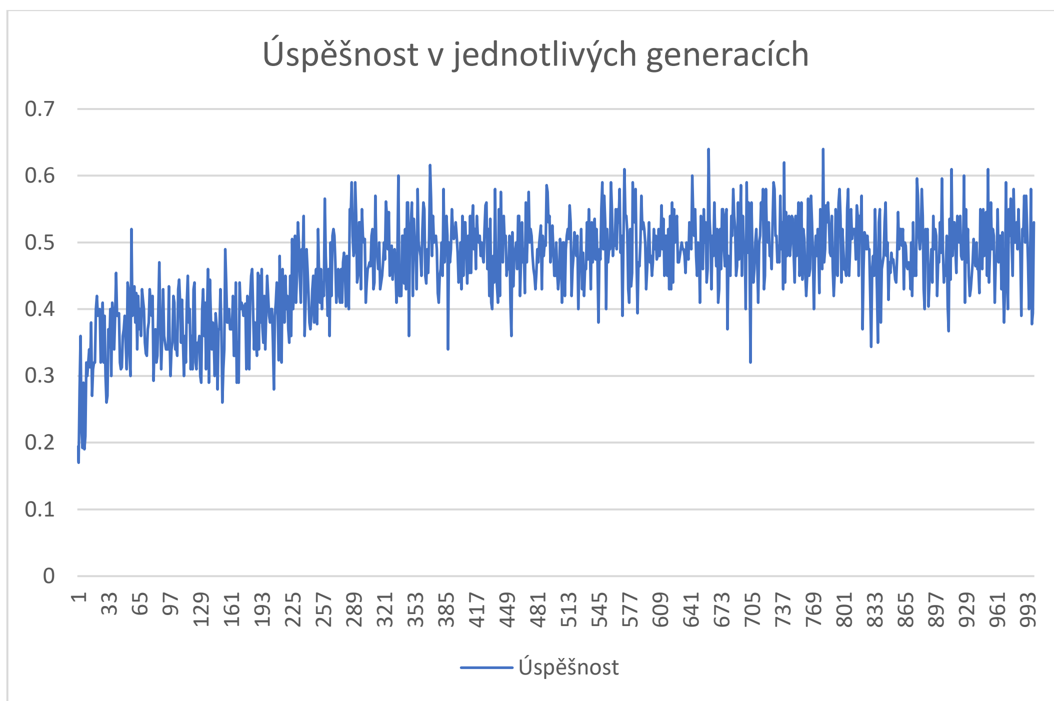


7) Graf klouzavých průměrů výher

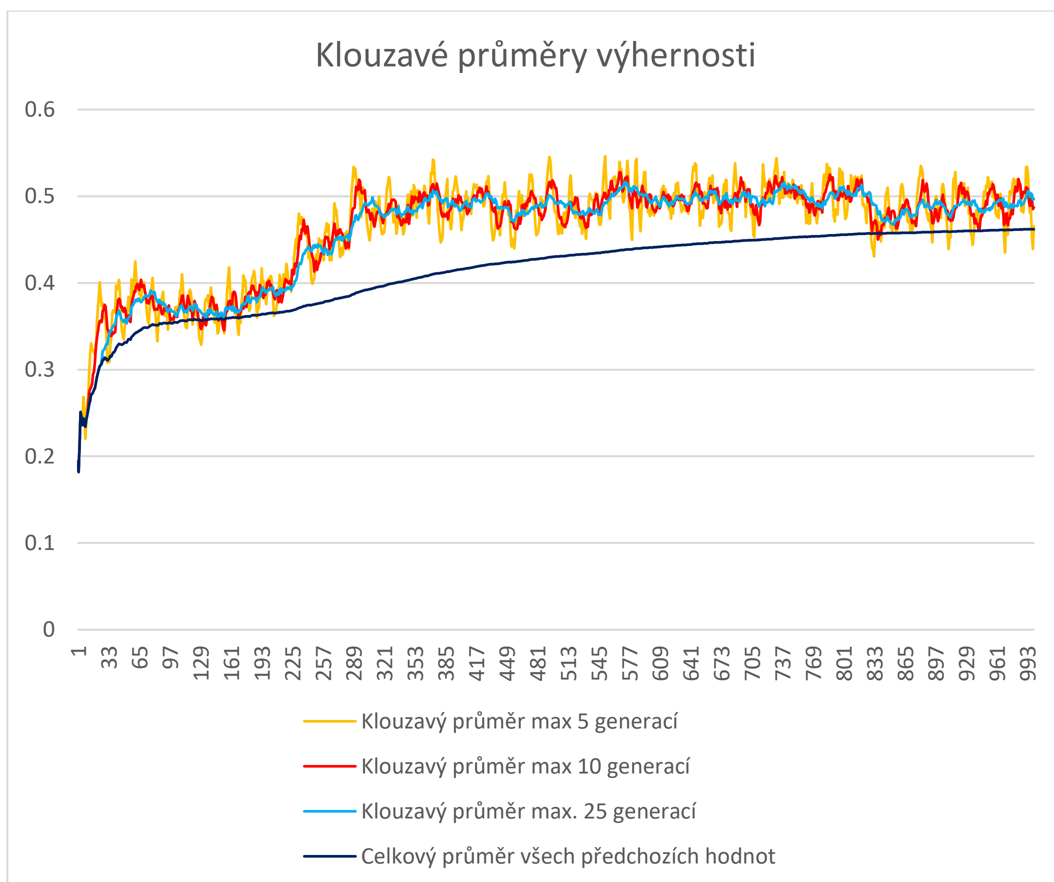
Z výsledků je vidět určité zlepšování včetně několika bodů s velmi dobrými procenty výher. Ovšem i v pozdějších generacích vidíme časté výsledky s výherností značně pod polovinu. Z výsledků je dále možné nahlédnout postupné zpomalení až zastavení růstu průměrné výhernosti.

7.2. Druhý experiment

V druhém experimentu byl využit annealing a upustili jsme od mocnění výsledků. Navíc jsme zvětšili populaci na 35 jedinců, což zvýšilo nutný počet porovnávání z 300 pro 25 jedinců na celkových 595. Tím se průměrný čas na generaci více než zdvojnásobil. Algoritmus běžel po dobu tří dnů, 16 hodin a 9 minut, což nám dává čas na generaci v průměru 5 min 17 s. Níže jsou prezentovány výsledky pro tento průběh algoritmu.



8) Spojnicový graf vývoje úspěšnosti po generacích



9) Graf klouzavých průměrů výher

V tomto experimentu algoritmus začínal s mnohem horší počáteční maticí a algoritmu tedy trvalo dosáhnout podobných výsledků mnohem déle. Je zde také vidět sekce v prvních 250 generacích, kdy klouzavé průměry naopak ukazují na zhoršení výhernosti.

7.3. Analýza výsledků

Jak lze již z výše uvedených grafů nahlédnout, při samoučení našeho algoritmu průměrná výhernost stagnuje blízko 50 %. Nemůžeme tedy říci, že bychom naším algoritmem určitě nalezli umělou inteligenci, která statické AI předčí.

Hypotéz pro tento výsledek je mnoho, nyní tedy prezentujeme některé z nich. První z těchto hypotéz je, že počítač, využitý k práci na tomto algoritmu, nebyl dostatečně výkonný pro toto využití. Vzhledem k simulaci na CPU místo GPU a celkové výpočetní síle je možné, že jsme nemohli získat dostatečně přesné relativní ohodnocení, nebo jsme nemohli využít dostatečně velké generace či dostatečný počet vstupních dat, a tím jsme se zastavili v lokálním optimu.

Vzhledem k výsledkům druhého experimentu se jeví nemožnost využívat annealing jako nepravděpodobný scénář, jelikož po určitém čase se průběh obou experimentů začne chovat podobným způsobem.

Další hypotéza, která se jeví jako možné vysvětlení zastavení konvergence, se týká počtu akcí, které má umělá inteligence k dispozici. Tato hypotéza přináší prostor pro další výzkum. Jako u první hypotézy však narážíme na limity v podobě výpočetního času pro ověření této hypotézy.

Mezi hypotézy lze také zařadit problém klasifikace fitness value: je možné, že výsledky jsou ovlivněny využitím jediného fitness value pro ohodnocení jednotlivých vygenerovaných umělých inteligencí. Naší hypotézou je, že výhernost lze rozdělit do více hodnot, pro které optimalizujeme a získáme tím vícedimenzionální prostor řešení, který je schopen lépe zachovávat lepší výsledky. Pro více dimenzí je však problémem klasifikace těchto hodnot a následná propagace do dalších generací, která je netriviální.

Poslední zde zmíněnou hypotézou bude typický problém optimalizace samoučících se systémů: optimalizace programu dále pokračuje, ale jednotlivé umělé inteligence se začnou specializovat proti sobě samým a ztratí tím možnost lépe abstrahovat problémy

proti našemu výsledku. Toto je jedno z možných vysvětlení pro pozorované zhoršení průměrné výhernosti v druhém experimentu před zlepšením celkové výhernosti.

Každá z těchto hypotéz nebo jakákoliv jejich kombinace je možným vysvětlením, proč algoritmus stabilně nedosahoval vyšší výhernosti. Nemůžeme však ani zavrhnout možnost, že jsme se dostali k hranici možného zlepšování proti našemu statickému AI. Vzhledem k výsledkům však lze říci, že tato metoda je schopna určité úrovně samoučení pro tuto situaci, a že další výzkum by byl pro tento problém přínosný.

I přes tyto nejasné výsledky jsme následně tuto hru implementovali do počítačové hry v enginu Unity, kde je možné načíst umělé inteligence správného formátu a hrát proti nim. Bohužel se vyskytly potíže vzhledem k zapojení kolegů do implementace a hra tedy nebyla dokončena včas pro provedení průzkumu hráčů. Odkaz na hru je však přiložen jako příloha pro možnost hry proti těmto umělým inteligencím.

8. Závěr

V této práci jsme navrhli a následně implementovali novou metodu generování umělé inteligence genetickým algoritmem pro tahové strategie. Zmínili jsme její implementaci, potíže s využíváním jen relativního fitness value a možné vylepšení implementace programu. Následně jsme analyzovali dva průběhy takového generování umělé inteligence a podali hypotézy k jejich průběhu.

8.1. Budoucí práce

Východiskem pro další práce by v tomto případě bylo využití většího výpočetního výkonu a lepší optimalizace pro tuto metodu, čímž by mohly být překonány limity genetických algoritmů pro malé generace.

Další možnou navazující prací je porovnání s ostatními možnými metodami generace umělých inteligencí a z toho navazující otázka, zda tato metoda má možnost využití v tomto problému, nebo zda existují jiné, lepší metody, které by šly využít lépe.

Jinou možnou cestou je rozšíření samotného problému buď na jiná využití, nebo rozšíření možností v tahových strategiích a následná analýza chování za takto nalezených podmínek. Pro tuto metodu by také bylo nutné zvýšit výpočetní výkon, což by pomohlo s rychlostí chodu programu.

Posledním zde zmíněným pokračováním je analýza chování takto generované umělé inteligence pro hry s dalšími hráči. Jak bylo zmíněno v úvodu, lepší chování oproti dnešním metodám by bylo dostatečnou výhodou pro využití této metody.

Pokud však tyto metody selžou, je z našeho hlediska jednodušší a časově efektivnější využití dnešních metod. Přesto však zůstává vývoj umělých inteligencí do her jednou z nejdůležitějších součástí vývoje her a tento obor má tedy velký potenciál pro budoucí výzkum, ať již pro genetické algoritmy či jiné, nové metody.

9. Seznam použité literatury

[1] Ben Coppin. Artificial Intelligence Illuminated, Jones & Barlett Learning, 2004. ISBN 9780763732301

[2] Ian Millington, AI for Games, Third Edition, CRC Press, 2019 ISBN 978-1138483972

[3] D. M. Bourg, G. Seeman. AI for Game Developers, O'Reilly, 2004. ISBN 0-596-00555-5

[4] Stuart Russel, Peter Norvig. Artificial Intelligence: A Modern Approach, 4th edition, Pearson 2021 ISBN 9780137505135

[5] Jarno Hyrkäs. Reinforcement learning in a turn-based strategy game, Kajaanin ammattikorkeakoulu, 2015

[6] Chess.com chess terms: Stockfish [online]. [cit. K 15.7.2024] Dostupné z: https://www.chess.com/terms/stockfish-chess-engine

[7] Chess.com chess terms: Komodo [online]. [cit. K 15.7.2024] Dostupné z: https://www.chess.com/terms/komodo-chess-engine

[8] Chess.com chess terms: AlphaZero [online]. [cit. K 15.7.2024] Dostupné z: https://www.chess.com/terms/alphazero-chess-engine

[9] Sanz Sanz, Pablo V. Quirós, Juan Carlos. Deep learning applied to turn-based board games, Universidad Complutense Madrid, 2021

[10] Gabriel Jonathan, Nur Ulfa Maulidevi. Reinforcement Learning for Turn-Based Strategy Game, 2023 10th International Conference on Advanced Informatics: Concept, Theory and Application (ICAICTA), 2023. ISBN 979-8-3503-2991-9

[11] Like Zhang, Hui Pan, Qi Fan, Changqing Ai, Yanqing Jing. 1GBDT, LR & Deep Learning for Turn-based Strategy Game AI, 2019. ISBN 978-1-7281-1884-0

[12] Yosh. Training an unbeatable AI in Trackmania, 2023 [online] [cit. K 15.7.2024] dostupné z: https://www.youtube.com/watch?v=Dw3BZ6O_8LY

[13] Joakim Nilsson, Andreas Jonasson. Using Artificial Intelligence for Gameplay Testing On Turn-Based Games, Blekinge Institute of Technology, 2018

[14] William L. Raffe; Fabio Zambetta; Xiaodong Li; Kenneth O. Stanley. Integrated Approach to Personalized Procedural Map Generation Using Evolutionary Algorithms, IEEE, 2015. ISSN: 1943-0698

[15] Bentley James Oakes, Practical and theoretical issues of evolving behaviour trees for a turn-based game, McGill University, 2013

[16] A. Fernández-Ares; A. M. Mora; J J. Merelo; P. García-Sánchez; C. Fernandes. Optimizing player behavior in a real-time strategy game using evolutionary algorithms, 2011 IEEE Congress of Evolutionary Computation, 2011. ISBN 978-1-4244-7835-4

[17] Maurice Bergsma, Pieter Spronck: Adaptive Spatial Reasoning for Turn-based Strategy Games, Tilburg University, 2008

[18] Kristo Radion Purba, Liliana Liliana, Johan Pranata. Optimization of Units Movement in Turn-Based Strategy Game, Petra Christian University, 2016

[19] L. Moroney: AI and Machine Learning for Coders: A Programmer's Guide to Artificial Intelligence, O'Reilly, 2020. ISBN 978-1492078197

[20] F. Buontempo: Genetic Algorithms and Machine Learning for Programmers: Create AI Models and Evolve Solutions, Pragmatic Programmers, 2019. ISBN 978-1680506204

[21] M. Tim Jones. Artificial Intelligence: A Systems Approach, Jones & Bartlett Learning, 2009. ISBN 9780763773373