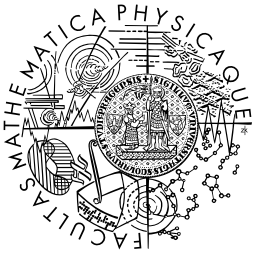


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Ondřej Roztočil

Evoluce umělé inteligence pro tahovou strategickou hru

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D.

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2020

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

Děkuji RNDr. Tomáši Holanovi, Ph.D. za cenné rady a podněty při tvorbě práce.
Děkuji Tereze Jindrové a své rodině za trpělivost.

Název práce: Evoluce umělé inteligence pro tahovou strategickou hru

Autor: Ondřej Roztočil

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Práce se věnuje vývoji umělé inteligence pro zjednodušenou variantu tahové strategické hry Advance Wars. Zaměřujeme se přitom na srovnání možností tradičních „ručních“ metod herní AI a metod minimalizujících závislost na doménových znalostech o hře. První část práce obsahuje analýzu hry, přehled vybraných technik herní AI a popis implementace agentů na bázi rozhodovacích stromů, užitkových přístupů a neuronových sítí. Dále prezentujeme výsledky pokusů s učením parametrů agentů a vah neuronových sítí pomocí genetických algoritmů a evolučních strategií. Experimentovali jsme s trénováním agentů proti fixním oponentům a s mechanismem kompetitivní koevoluce.

Ve druhé části práce detailně prezentujeme vytvořený software. Pro potřeby vývoje AI byl implementován efektivní simulátor hry vhodný pro rychlou hru agentů a sada pomocných nástrojů. Vytvořen byl také plnohodnotný grafický klient umožňující hru lidských i počítačových hráčů.

Klíčová slova: tahová strategická hra, umělá inteligence, evoluční algoritmus

Title: Evolution of artificial intelligence for turn-based strategy game

Author: Ondřej Roztočil

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Tomáš Holan, Ph.D., Department of Software and Computer Science Education

Abstract: This thesis deals with the development of artificial intelligence for a simplified version of turn-based strategy game Advance Wars. Our focus is on comparing traditional "ad-hoc" game AI methods with methods that minimize the dependance on domain knowledge of the game. First part of the thesis contains analysis of the game, survey of selected game AI techniques and an overview of the implemented agents using decision trees, utility-based methods and neural networks. We present the results of our experiments with learning agent parameters and neural network weights using genetic algorithms and evolutionary strategies. We experimented with training agents against fixed opponents and with competitive coevolution.

In the second part we present the created software in detail. An efficient game simulator and a set of utility tools were implemented to support the AI development. A full-fledged graphical client that allows for both human and AI gameplay was also created.

Keywords: turn-based strategy game, artificial intelligence, evolutionary algorithm

Obsah

Úvod

Hraní her, konkrétně hraní kompetitivních her s cílem zvítězit, bylo předmětem výzkumu od samotného počátku odvětví počítačové umělé inteligence (AI). Od prvních úvah zakladatelů oboru (Shannon 1950; Turing 1953) po medializovaná vítězství nad nejlepšími lidskými experty za pomocí heuristik a hrubé výpočetní síly (šachový počítač Deep Blue, Hsu a kol. 1995) nebo strojového učení (program AlphaGo, Silver a kol. 2016), představovaly hry populární oblast pro rozvíjení či demonstraci pokroku metod umělé inteligence.

Mezi hlavní příčiny této popularity patří ty, které vyplývají ze zvláštního charakteru her. Na rozdíl od problémů reálného světa, hry mají jasné cíle, neměnná, formálně popsatelná pravidla a omezené možnosti interakce mezi aktéry (Schaeffer 2001). Navíc pro klasické stolní hry, na které se po většinu své historie výzkum zaměřoval, existuje velké množství expertního lidského vědění a kompetenci vytvořených počítačových hráčů lze testovat hrou s kvalifikovanými lidskými oponenty.

Nezávisle na akademickém prostředí vyrostl od 70. let minulého století rozsáhlý průmysl počítačových her. V rámci něj vzniklo množství nových typů her, jejichž návrh oproti klasickým stolním hrám využíval technických možností moderních počítačů reprezentovat řádově větší herní stavy a simulovat komplexnější herní prostředí. Ačkoli implementace nějaké, byť primitivní, formy umělé inteligence je nutnou součástí tvorby většiny počítačových her a kvalita tohoto řešení může mít významný vliv na celkový úspěch produktu, sféry akademického výzkumu a herního průmyslu zůstávaly dlouhou dobu do značné míry oddělené (Yannakakis a Togelius 2018).

Kromě historických faktorů lze tento jev vysvětlit odlišnými zájmy a požadavky, které obě skupiny vzhledem k umělé inteligenci ve hrách mají. Při vývoji her je hlavním cílem poskytnout uživatelům zajímavý zážitek s přiměřenou mírou obtížnosti (pokud je AI použita v roli oponenta). Velmi důležitá jsou také praktická kritéria jako časová nákladnost implementace, spolehlivost výsledku a běhová náročnost. Z tohoto důvodu byly v praxi jen zřídka nebo pomalu přejímány výsledky akademického výzkumu, v rámci kterého je kladen důraz na experimentaci s novými technikami umělé inteligence a na kvantifikovatelnou sílu výsledných agentů, byť za cenu „nelidskosti" jejich hry či výpočetní náročnosti použitých algoritmů.

V opačném směru bývá použití počítačových her jako pole pro výzkum limitováno jejich typicky proprietárním charakterem a nedostupností zdrojových kódů nebo veřejných programátorských rozhraní umožňujících efektivní přístup k hernímu stavu a simulaci herních pravidel. Z tohoto důvodu pracovala dlouhou dobu většina výzkumu herní AI – zaměříme-li se na strategické hry – s open-source projekty jako ORTS, Wargus nebo Freeciv (Ponsen a kol. 2005). Nejlepší ilustrací užitečnosti veřejných programátorských rozhraní je pak masivní rozvoj výzkumu pro hru Starcraft, který nastal po vzniku projektu BWAPI.

Uvedené skutečnosti motivovaly i naši vlastní práci, jejíž příspěvek by měl být zčásti implementační, zčásti experimentální. Na rovině programátorské jsme vytvořili rozsáhlý projekt skládající se z herního enginu pro tahovou strategickou hru s rozhraním pro AI agenty a ze sady nástrojů pro učení, testování a prezentaci funkcionality agentů. Součástí projektu je rychlý negrafický simulátor hry a plnohodnotný grafický klient, který umožňuje hru lidských i počítačových hráčů. Jako vzor implementované hry jsme zvolili japonskou sérii tahových strategických her Advance Wars, a to zejména kvůli její vhodné míře komplexity, která hru zasazuje do pozice mezi klasickými stolními hrami a rozsáhlými počítačovými hrami jako je Starcraft.

Na této platformě jsme následně experimentovali s tvorbou herních agentů. Během studia metod herní AI v odborné literatuře i během vlastní implementace agentů nám šlo o porovnání vlastností technik pocházejících z prostředí praktického vývoje počítačových her a z prostředí AI výzkumu. Použili jsme při tom nástroje jako jsou neuronové sítě a evoluční algoritmy, s cílem vyzkoušet možnost jejich začlenění do herního vývoje. Zajímal nás zejména potenciál ušetřit část lidské práce, která je součástí klasického „ručního" vývoje herní AI.

Cíle práce

Záměrem práce bylo vytvoření kompetentního počítačového agenta pro zjednodušenou verzi tahové strategické hry Advance Wars. Za tímto účelem byly plněny tři cíle. Za prvé, byla implementována samotná hra a vytvořeno softwarové prostředí pro vývoj herních agentů. Za druhé, byla provedena rešerše technik herní umělé inteligence v odborné literatuře. Za třetí, bylo vytvořeno několik herních agentů používajících vybrané techniky, jejichž parametry byly optimalizovány pomocí evolučních algoritmů a hry agentů proti sobě samým.

Struktura práce

Kapitola 1 představuje implementovanou hru a podává analýzu jejích herněteoretických vlastností. Kapitola 2 prezentuje výsledky rešerše technik herní umělé inteligence v odborné literatuře. Podává stručný popis jednotlivých metod a příklady jejich využití v publikovaném výzkumu.

Kapitola 3 popisuje návrh AI agentů implementovaných v rámci práce. Kapitola 4 popisuje provedené pokusy s evolučními algoritmy pro optimalizaci parametrů agentů a prezentuje dosažené výsledky.

Kapitola 5 obsahuje uživatelskou dokumentaci k vytvořeným programům. Kapitola 6 se věnuje technické implementaci projektu. Diskutuje volbu technologií, popisuje celkovou architekturu a dokumentuje objektový návrh jednotlivých programů.

1. Implementovaná hra

1.1 Popis hry

OpenAW je taktická tahová hra s vojenskou tématikou, v níž 2 až 4 hráči soupeří prostřednictvím jednotek umístěných na herní mapě. V následujícím textu stručně představíme hlavní koncepty hry. Podrobná pravidla jsou dostupná v elektronické příloze práce v souboru `/Docs/pravidla.pdf`.

Hlavním prostředkem hráčovy interakce v rámci hry jsou jednotky. Každá jednotka má přiřazený typ a patří některému z hráčů. Typ jednotky určuje např. její možnosti pohybu, bojové parametry či speciální akce, které může vykonávat. Kromě statických vlastností daných typem má jednotka svůj proměnlivý stav. Jeho součástí jsou pozice na mapě a číselné údaje o zdraví jednotky, množství munice a paliva, a také údaj, zda již jednotka provedla v aktuálním kole akci.

Obrázek 1.1: Ilustrace hry probíhající v implementovaném grafickém klientovi

Herní mapa je konečná obdélníková mřížka stejně velkých herních polí. Hra se může odehrávat na různých mapách lišících se svojí velikostí i složením polí. Každé pole má pevně přiřazený typ („terén“) a může se na něm nacházet nejvýše jedna jednotka. Typ pole modifikuje bojové parametry jednotky, která se na něm nachází, a pohybové možnosti jednotky, která chce přes pole přejít. Pole některých speciálních typů mohou hráči obsazovat pomocí jednotek. Obsazená speciální pole přinášejí svým vlastníkům průběžné benefity a umožňují kupovat nové jednotky.

V průběhu hry se střídají kola jednotlivých hráčů (v pořadí daném mapou). Během svého kola může hráč provést s každou svojí jednotkou a některými obsazenými poli až jednu herní akci. Akce jsou prováděny sekvenčně v pořadí určeném hráčem. V závislosti na svém typu a herní pozici mohou jednotky provádět akce přesunu, útoku, obsazení speciálního pole, nebo přesunu následovaného jednou další akcí na nové pozici. Zatímco přesun a útok jsou instantní akce, obsazení pole trvá v závislosti na síle obsazující jednotky několik kol (minimálně dvě). Jednotky mají omezené množství paliva, které spotřebovávají při přesunu. Některé jednotky navíc mají omezené množství munice, kterou spotřebovávají při boji. Palivo i munici lze doplňovat umístěním jednotek na odpovídající speciální pole nebo pomocí podpůrných jednotek. Transportní jednotky mohou po jedné převážet spřátelené jednotky vybraných typů.

Hráč má také určité množství financí. Na začátku hráčova kola se jeho finance zvýší o obnos odpovídající počtu jím obsazených speciálních polí. Finance lze utrácet za nové jednotky kupované na vlastněných speciálních polích odpovídajících typů (továrny, letiště, přístavy). Na stejných polích lze také za finance opravovat poškozené jednotky.

Každý hráč začíná hru s obsazeným speciálním polem – velitelstvím. Cílem hry je ubránit svoje a obsadit oponentovo velitelství. Hráč, který přijde o velitelství, prohrává a je ze hry odstraněn. Vítězem hry se stává hráč, který zůstane jako poslední.

Zjednodušená varianta pravidel

Vedle výše popsané plné varianty pravidel jsme implementovali podporu pro selektivní vypínání či omezovaní určitých aspektů hry. V rámci vývoje herní AI jsme se omezili (u netriviálních agentů) na podporu pouze zjednodušené varianty pravidel. Ta se liší v následujících ohledech:

- Je možná pouze hra právě dvou hráčů.
- Jednotky mají k dispozici neomezené množství paliva a munice.
- Ve hře je k dispozici pouze část typů jednotek a speciálních polí – odstraněny jsou námořní a letecké jednotky a také transportní jednotky.
- Je přidán konfigurovatelný limit počtu herních kol. Po jeho dosažení skončí partie remízou.

1.2 Analýza hry

Následuje přehled vlastností hry OpenAW relevantní pro realizaci herní AI. Jeho součástí je charakterizace hry z hlediska teorie her a návrhu strategických počítačových her.

Distribuce odměny

OpenAW je kompetitivní hra, na jejímž konci je jeden hráč vítězem a ostatní hráči prohrávají. Formálněji řečeno, jde o hru s nulovým součtem, v níž vítěz na konci obdrží odměnu $+1$ a ostatní hráči odměnu $-1/(n-1)$, kde n je počet hráčů. Ve zjednodušené variantě pravidel je možná také remíza odpovídající odměně 0 pro všechny hráče.

Dostupnost informací

OpenAW je hra s perfektní informací. To znamená, že hráči mají všechny informace o aktuálním stavu hry a jeho historii. OpenAW je také hrou s úplnou informací. To znamená, že každý hráč zná distribuci odměn všech hráčů a ví, jaké akce jsou dostupné všem hráčům (Tadelis 2013).

Determinismus

OpenAW neobsahuje prvky náhody. To znamená, že pro každý herní stav a akci dostupnou v tomto stavu existuje právě jeden výsledný herní stav. Jinými slovy, výsledek hry závisí pouze na počátečním stavu a posloupnosti akcí zvolených hráči. OpenAW neobsahuje ani externí náhodné elementy, jako je náhodné určení pořadí hráčů nebo generování herní mapy.

Konečnost

OpenAW v plné variantě pravidel je nekonečná hra. V každém stavu je dostupné pouze konečné množství akcí (omezené počtem jednotek a speciálních polí na konečné mapě), existují ale nekonečné posloupnosti akcí, které nedosáhnou koncového stavu. Zjednodušená varianta hry je konečná, protože přidává pravidlo o limitu počtu kol.

Sekvenčnost a časová granularita

OpenAW je tahová hra, v níž se sekvenčně střídají tahy hráčů. Během tahu může hráč provést více akcí v libovolném pořadí. Alternativním typem jsou tzv. real-time strategické (RTS) hry, v nichž hráči vykonávají akce současně a herní simulace běží nepřetržitě.

„Žánrové" vlastnosti

OpenAW se může hrát na symetrických i asymetrických mapách. Během hry mohou přibývat herní jednotky (tudíž i počet dostupných akcí), v počtu teoreticky limitovaném jen rozměry mapy. Zdroj, za který jsou jednotky pořizovány (finance), není vyčerpatelný.

2. AI techniky a související práce

Z předchozí analýzy vyplývá, že hra OpenAW má stejné teoretické vlastnosti jako šachy a další klasické stolní hry, tj. jde o deterministickou hru s perfektní informací a nulovým součtem. Pro hry tohoto typu byli vypracováni silní počítačoví hráči na bázi heuristického prohledávání stromu hry. Aplikovatelnost tohoto přístupu na naši hru je silně limitována velkým faktorem větvení jejího stromu. Hráč totiž během svého kola neprovádí jeden tah, ale může táhnout s každou svojí jednotkou. Jednotlivé akce na sobě navíc nemusí být nezávislé, tudíž může být potřeba zohlednit jejich různé permutace.

Atraktivním polem výzkumu byly v posledních letech přístupy snažící se řešit problém prohledávání v komplexních hrách pomocí stochastické aproximace, např. techniky z rodiny Monte Carlo Tree Search (Branavan a kol. 2011) nebo online evoluční plánování (Justesen a kol. 2017). Při praktickém vývoji počítačových her bývají problémy vyplývající z neúnosného faktoru větvení stromu hry řešeny dvěma způsoby. Jedním způsobem je rovinu plánování zcela ignorovat a chování herních agentů řídit reaktivně, nebo dokonce čistě staticky. Druhý typ řešení představují hierarchické modely, v nichž se plánování či prohledávání odehrává v řádově menším prostoru předdefinovaných strategií nebo tzv. makro akcí.

V naší práci se omezujeme na úroveň reaktivního rozhodování. Naším cílem ale je dosáhnout natolik adaptivního rozhodovacího procesu, aby i bez explicitního plánování nebo ručně definovaných strategií, byla jeho výsledkem kompetentní a smysluplná hra. K tomuto účelu chceme vyzkoušet zejména použití neuronových sítí a učení pomocí evolučních algoritmů. Protože nemáme možnost výsledek srovnat s existujícími počítačovými agenty, implementujeme několik vlastních variant agentů a tyto dále porovnáme.

Ve zbytku této kapitoly stručně představíme metody pro realizaci herní umělé inteligence, jež byly v nějaké formě využity v rámci práce. U jednotlivých metod popíšeme hlavní principy a možnosti použití. Uvedeme také některé příklady použití v kontextu strategických her popsané v odborné literatuře

2.1 Ad hoc metody

Termín ad hoc metody označuje neostře vymezenou kategorii technik, které jsou různými způsoby založené na programátorem ručně definovaném chování herních agentů. Negativně je lze definovat jako metody, které neobsahují významné aspekty vyhledávání, plánování nebo učení. Z tohoto důvodu často nejsou považovány akademickým prostředím za součást umělé inteligence (Millington 2019). V rámci komerčního vývoje her naopak tyto metody historicky představují hlavní nástroje pro tvorbu herní AI. Mezi jejich výhody patří snadnost implementace (byť špatně škálující s požadovanou komplexitou chování), dobrá testovatelnost a zpravidla nízké běhové nároky.

Nejběžnějšími příklady ad hoc metod jsou přímé skriptování herních akcí, rozhodovací stromy, konečně stavové automaty, stromy chování (*behavior trees*), systémy pravidel nebo užitkové přístupy. Techniky je možné vzájemně kombinovat, např. určovat pomocí stavového automatu, který skript nebo sada pravidel se má aktuálně používat. Také je lze doplnit o pravděpodobnostní elementy snižující monotónnost výsledného chování.

Jednotlivé techniky jsou popsány například v učebnicích Bourg a Seemann (2004) nebo Millington (2019). Vzhledem k významné roli ad hoc technik v rámci herní produkce a jejich relativní nezajímavosti pro akademický výzkum herní AI, se těmto metodám věnují zejména prakticky orientované texty psané vývojáři z herního průmyslu (Rabin 2002, 2013).

Jistá pozornost je věnována možnosti konstruovat struktury použité v ad hoc metodách pomocí strojového učení. Baumgarten a kol. (2008) aplikovali algoritmus ID3, vytvořený pro řešení úlohy klasifikace dat, na tvorbu rozhodovacích stromů pro agenty v RTS hře Defcon. Oakes (2013) použil metodu genetického programování pro konstrukci stromů chování reprezentujících strategie v tahové hře Battle for Wesnoth.

V rámci naší práce ze jmenovaných metod používáme ručně vytvořené rozhodovací stromy a užitkové přístupy. Dále používáme jako nástroj pro analýzu herního stavu mapy vlivu. Tyto metody zde stručně popíšeme.

2.1.1 Rozhodovací stromy

Herní rozhodovací strom představuje strukturální reprezentaci jednoduché formy rozhodování – posloupnosti testů přiřazujících vstupním stavům výstupní akce. V zobecněné variantě jde o n-ární strom, jehož každý vnitřní (tzv. rozhodovací) vrchol obsahuje podmínku testující nějakou vlastnost vstupního stavu, a jehož každý list obsahuje akci. Rozhodovací vrcholy mohou představovat například porovnání s boolovskou nebo číselnou hodnotou nebo selekci přes hodnoty výčtového typu.

Použití rozhodovacího stromu pak spočívá v průchodu od kořene k listu, při němž je v každém rozhodovacím vrcholu zvolena hrana odpovídající výsledku daného testu. Výstupem mohou být nejen elementární herní akce, ale také například makro akce v rámci hierarchického návrhu nebo změny stavu nadřazeného stavového algoritmu.

2.1.2 Užitkové přístupy

Další základní způsob reaktivního rozhodování spočívá v heuristickém ohodnocení dostupných akcí a volbě akce podle obdrženého skóre. Tzv. užitkové (*utility-based*) přístupy představují modulární systematizaci tohoto způsobu, inspirovanou ekonomickou teorií užitku. Úvod do užitkových přístupů v kontextu herní umělé inteligence podává Mark (2009).

Užitkový přístup k rozhodování spočívá ve výběru akce dle číselně vyjádřeného očekávaného užitku, který agentovi v daném stavu přinese. Při jeho použití je každé dostupné akci její ohodnocovací funkcí přiřazeno skóre ze stejného intervalu. Následně je zvolena nejlepší akce nebo je skóre použito pro určení pravděpodobností výběru jednotlivých akcí.

Hlavní část práce se odehrává v ohodnocovacích funkcích pro jednotlivé druhy akcí. Užitková funkce skládá dohromady určité množství faktorů. Faktor zde znamená nějaké kvantitativní pozorování o vstupním stavu. Zpravidla chceme tuto hodnotu vyjádřit v relativním tvaru (např. jako podíl maxima, kterého může daná veličina nabývat), aby výsledek nezávisel na měřítku vstupních hodnot. Faktor se na výpočtu užitku nemusí podílet přímou úměrou, složitější vztahy je možné modelovat aplikací vhodné funkce.

Pro ilustraci si představme strategickou hru s vojenským tématem, kde chceme pro hráčovu jednotku ohodnotit v intervalu $[0, 1]$ akci útoku na nepřátelskou jednotku. Řekněme, že máme pro jednotku i k dispozici skalární hodnoty popisující jejich sílu S_i (celé číslo) a důležitost D_i (na škále $1 - 100$). Jednoduchá ohodnocovací funkce se dvěma faktory pak může mít předpis:

$$U = \left(\frac{S_1}{S_1 + S_2}\right)^{\alpha} \frac{D_2}{100}$$

Prvním faktorem je zde poměr sil mezi jednotkami, přičemž závislost na této hodnotě je vyjádřena nelineární funkcí (pro $\alpha \neq 1$). Druhým faktorem je důležitost cíle útoku normalizovaná jako podíl na maximální hodnotě. Obdobně bychom navrhli ohodnocovací funkce pro další druhy akcí s jinými faktory a vztahy.

Tato jednoduchá architektura má několik potenciálně silných stránek. Ve srovnání například s rozhodovacími stromy nebo stavovými automaty má umožnit snáze dosáhnout smysluplného jednání v široké škále situací, bez nutnosti je explicitně podchytit v rozhodovacím algoritmu. Při výpočtu očekávaného užitku lze přirozeně pracovat s neúplnou informací a náhodou. Z hlediska implementace je výhodou její modularita a snadnost přidávání nových akcí a kritérií.

Hlavní nevýhodou užitkového přístupu je potřeba dobrého porozumění doméně, s níž autor pracuje. Musí určit, které veličiny je třeba pro dané rozhodnutí zohlednit, jakým způsobem a s jakou vahou vzhledem k ostatním faktorům. Nabízí se možnost tento problém obejít pomocí algoritmické optimalizace. Hledat můžeme parametry ručně vytvořené funkce, případně lze neznámou funkci aproximovat pomocí neuronové sítě nebo klasického zpětnovazebního učení.

2.1.3 Mapy vlivu

Mapy vlivu (*influence maps*) jsou pomocnou technikou pro analýzu stavu her, v nichž je důležitý prostor a pozice herních objektů. Jsou motivovány úvahami, které lidský hráč provádí intuitivně. Za prvé, pro adekvátní zohlednění pozic objektů (např. jednotek) nestačí brát v potaz jen jejich aktuální pozici, ale i vliv nebo potenciál působení, který mají vzhledem ke svému okolí. Za druhé, vlivy

více objektů je potřeba určitým způsobem složit dohromady. Mapy vlivu umožňují AI zjednodušeným způsobem zohledňovat vztahy objektů v herním světě a činit prediktivní rozhodnutí, bez nutnosti provádět plnohodnotné plánování. V následujícím textu vycházíme z prací Tozour (2001) a Millington (2019).

Mapa vlivu je typicky reprezentována dvourozměrnou maticí reálných čísel. V základním případě odpovídá jedno herní pole jednomu prvku matice, mohou být ale použita i složitější zobrazení. Při inicializaci je k polím mapy vlivu (prvkům matice) přičten vliv každého mapovaného objektu. Počáteční hodnota vlivu I_0 je přičtena k poli odpovídajícímu herní pozici objektu. Pro další pole ve vzdálenosti d je přičítaná hodnota I_d vypočtena podle zvoleného vzorce šíření vlivu, například:

$$I_d = max\,(0, I_0 - s\ \cdot d) \tag{2.1}$$

$$I_d = I_0 \cdot q^d \tag{2.2}$$

$$I_d = \frac{I_0}{(1+d)^p} \tag{2.3}$$

kde $q \in (0, 1)$, p, s jsou kladná reálná čísla. Obrázek 2.1 ukazuje příklad mapy vlivu se dvěma objekty, vypočtené podle vzorce 2.2 s parametry $I_0 = 1, q = 0.5$. Pro objekty různých typů může být použita různá hodnota I_0. Vzdálenost d může představovat manhattanskou vzdálenost pole od zdroje vlivu, nebo například počet tahů za který se jednotka může na pole přesunout. Pro rychlé šíření vlivu se někdy používají techniky jako *map flooding* nebo konvoluční filtry pocházející z oblasti počítačové grafiky.

0.14	0.28	0.19	0.09	0.05	0.02
0.28	0.56	0.38	0.19	0.09	0.05
0.56	1.12	0.75	0.38	0.19	0.09
0.38	0.75	0.75	0.38	0.19	0.09
0.38	0.75	1.12	0.56	0.28	0.14
0.19	0.38	0.56	0.28	0.14	0.07

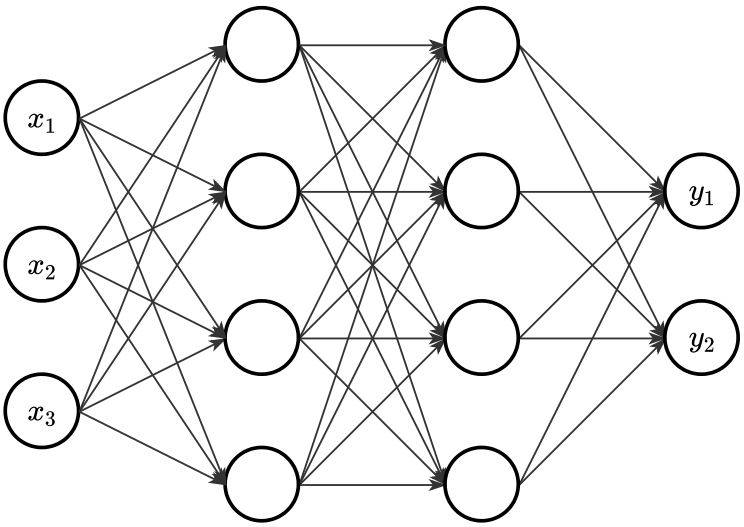
Obrázek 2.1: Příklad mapy vlivu

Mapy vlivu jsou typicky používány jako zdroj dat pro ohodnocovací funkce nebo pro heuristické algoritmy na hledání cest. Ve strategických hrách pomáhají např. analyzovat poměr sil hráčů v různých částech mapy, vybírat atraktivní cíle pro útok nebo bezpečně navigovat jednotky mezi rizikovými oblastmi. V kontextu RTS her je používána také příbuzná technika potenciálových polí (Hagelbäck a Johansson 2009).

2.2 Neuronové sítě

Umělé neuronové sítě jsou široce použitelným výpočetním modelem inspirovaným biologickými nervovými systémy. Různé varianty neuronových sítí jsou součástí nejlepších dostupných řešení pro klasifikační a regresní problémy z mnoha oblastí. My budeme pracovat se základními dopřednými neuronovými sítěmi, tzv. vícevrstvými perceptrony (*multilayer perceptron*, dále MLP).

Síť typu MLP má topologii orientovaného acyklického grafu. Tvoří ji výpočetní jednotky nazývané neurony, které jsou uspořádané do vrstev a propojené hranami vedoucími z předchozí do následující vrstvy. Každá hrana má přiřazenou váhu reprezentující sílu synaptického spojení mezi propojenými neurony. Síť se skládá ze vstupní vrstvy, výstupní vrstvy a jedné, nebo více skrytých vrstev nacházejících se mezi nimi.



Obrázek 2.2: Příklad topologie sítě typu MLP

Během výpočtu je vstupem sítě vektor reálných čísel, jehož složky tvoří výstupy neuronů ve vstupní vrstvě. Neurony v dalších vrstvách svoji výstupní hodnotu y spočítají podle vzorce

$$y \ = f\left(\mathbf{w}^T\mathbf{x} + b\right) = f\left(\sum_{i=1}^{n} w_i x_i + b\right),$$

kde $\mathbf{x}$ je vektor vstupů (tj. výstupů předchozí vrstvy), $\mathbf{w}$ vektor odpovídajících vah, b je přidaný práh (*bias*) a f je tzv. aktivační funkce. Existuje velké množství aktivačních funkcí, klasickými příklady jsou:

$$f(x) = \frac{1}{1+e^{-x}} \qquad \text{(sigmoida)}$$

$$f(x) = \max(0, x) \qquad \text{(ReLU)}$$

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad \text{(hyperbolický tangens)}$$

Součástí nasazení neuronové sítě je vždy algoritmus pro učení vah. V kontextu hraní her se používají tři paradigmata. Učení s učitelem (*supervised learning*) upravuje váhy sítě tak, aby minimalizovalo chybu na trénovací sadě označkovaných dat (dvojic vstup - správný výstup). Zpětnovazební učení (*reinforcement learning*) upravuje síť tak, aby aproximovala optimální strategii (výběr akcí) agenta v prostředí s odměnami. Tyto metody mají společné to, že modifikují váhy trénované sítě o gradient určité chybové funkce. Alternativním přístupem je použití evolučních algoritmů, kterým se věnuje sekce 2.3.

V rámci tvorby herní umělé inteligence jsou neuronové sítě využívány mimo jiné pro přímý výběr akcí, ohodnocování stavů, výběr strategií a modelování oponenta (Risi a Togelius 2014). Bergsma a Spronck (2008) použili MLP síť a mapy vlivu pro selekci akcí jednotek ve zjednodušené verzi hry Advance Wars. Stanescu a kol. (2016) pomocí konvoluční sítě ohodnocovali stavy v rámci prohledávání stromu hry Starcraft. Program AlphaStar (Silver a kol. 2019), představující *state-of-the-art* řešení pro hraní RTS her, kombinuje MLP, konvoluční, rekurentní a ukazatelové sítě. V první fázi používá učení s učitelem pro nápodobu hry lidských expertů, vzniklé agenty pak dále zlepšuje hrou proti sobě a zpětnovazebním učením parametrů sítí.

2.3 Evoluční algoritmy

Evoluční algoritmy tvoří rodinu optimalizačních metaheuristik inspirovaných přírodní evolucí. Představují silný nástroj u problémů, u nichž nemáme efektivní přímou metodu pro nalezení optimálního řešení, ale pro libovolné konkrétní řešení jsme schopni určit jeho kvalitu. Optimalizace parametrů herních agentů, nebo jimi používaných dílčích algoritmů, je právě takovým typem problému.

Evoluční algoritmy pracují iterativně s množinou kandidátů na řešení daného problému, tzv. populací jedinců nebo chromozomů. V každé iteraci je pro jedince v aktuální populaci spočtena hodnota *fitness*, vyjadřující jejich kvalitu jakožto řešení. Následně je aplikována určitá sada operací, pomocí níž je vytvořena populace pro další krok (nová generace). Alespoň některé z použitých operací obsahují prvky náhody. Algoritmus tak stochasticky prohledává prostor řešení daného problému a používá hodnotu fitness jako heuristiku pro určení perspektivních oblastí. Reprezentace řešení v jedinci a skladba operací prováděných v každé iteraci závisí na konkrétním problému a zvoleném algoritmu.

2.3.1 Genetické algoritmy a evoluční strategie

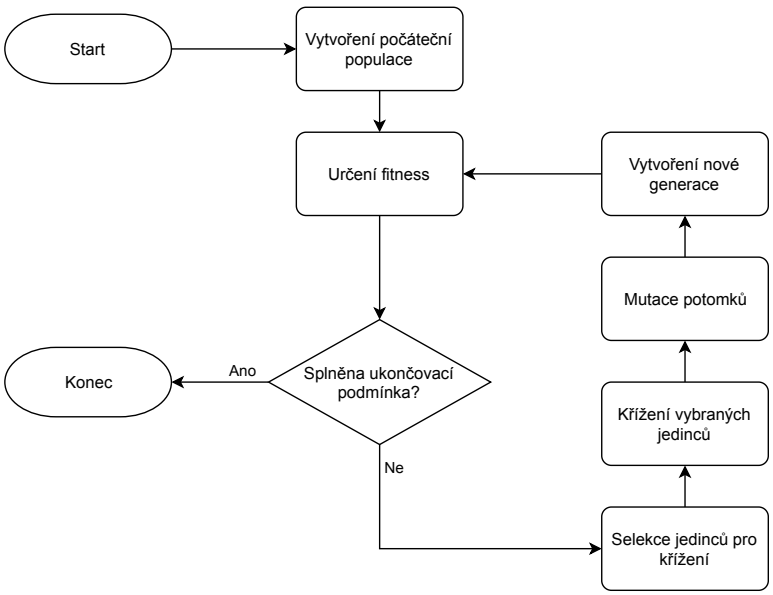
Genetické algoritmy jsou nejznámější kategorií evolučních algoritmů založenou na opakované aplikaci operací selekce, křížení a mutace. Obecně řečeno, selekce zajišťuje statistickou preferenci jedinců s vysokou fitness. Typicky n-ární operace křížení slouží pro výměnu informací mezi vybranými jedinci. Unární operace mutace pak typicky představuje nástroj diverzifikace populace a pojistku před ustrnutím v lokálních optimech. Systematické představení genetických algoritmů,

včetně diskuze způsobů, jak realizovat jednotlivé genetické operace, podávají například Mitchell (1998) nebo Eiben a Smith (2015). Zde stručně představíme několik klasických variant, s nimiž jsme experimentovali během práce.

Obrázek 2.3 znázorňuje průběh základního genetického algoritmu. Algoritmus začne zpravidla náhodnou inicializací první generace jedinců. Po vyhodnocení jejich fitness jsou pomocí operátoru selekce vybráni jedinci pro roli rodičů další generace. Při ruletové selekci je jedinec náhodně zvolen s pravděpodobností úměrnou jeho podílu na celkové fitness. Selekce dle pořadí místo podílu na populační fitness určuje šanci jedince na výběr pomocí jeho pořadí v sestupném uspořádání podle fitness. Turnajová selekce vybírá z náhodné n-tice s vysokou pravděpodobností jedince s nejvyšší fitness.

Následně je aplikován operátor křížení. Ten nějakým způsobem zkombinuje data rodičů a vytvoří požadovaný počet potomků. Pro jedince reprezentované polem prvků (bitů, čísel, znaků) lze použít například uniformní křížení, kdy je pro každou složku náhodně určeno, ze kterého rodiče se má převzít. Při n-bodovém křížení se pole rozdělí na $n + 1$ úseků a do jednoho potomka se zkopírují střídavě úseky z prvního a druhého rodiče, do druhého potomka naopak. Operace výběru rodičů (s vracením) a křížení jsou prováděny, dokud není vytvořeno požadované množství potomků. Na nové jedince je s určitou pravděpodobností aplikován operátor mutace. Jeho podoba opět závisí na reprezentaci jedince. Typickými možnostmi jsou nahrazení některých složek jedince novou náhodnou hodnotou z odpovídající domény nebo, v případě skalárních složek, přičtení náhodné hodnoty z vhodného rozdělení.

Tento cyklus se opakuje, dokud není splněna ukončovací podmínka. Tou může být například dosažení dané úrovně fitness, počet zpracovaných generací, nebo běhový čas.



Obrázek 2.3: Schéma základního genetického algoritmu

Dalším použitým typem evolučních algoritmů jsou evoluční strategie.[1] Jde o kategorii algoritmů určených primárně pro spojitou optimalizaci. Jejich charakteristickými rysy jsou použití mutace jako jediného nebo hlavního prostředku genetické změny a adaptace parametrů vlastního algoritmu v jeho průběhu (globálně, nebo na úrovní jedinců). Podrobný teoretický úvod do evolučních strategií podávají Beyer a Schwefel (2002). Zde popíšeme základní variantu evoluční strategie.

V té se pracuje s populací velikosti μ a v každé iteraci je vytvořeno λ potomků pomocí mutace náhodně vybraných rodičů. Mutace spočívá v přičtení vektoru náhodných čísel z normálního rozdělení se střední hodnotou nula a standardní odchylkou σ. V závislosti na úspěšnosti mutace lze parametr σ během algoritmu upravovat (princip adaptace), čímž je určováno, jak velké „kroky" může mutace v prostoru řešení dělat. Sofistikovanější evoluční strategie udržují parametr mutace pro každého jedince, nebo každou jeho složku zvlášť, jako součást jeho reprezentace (princip sebe-adaptace). Vedle mutace lze provádět také křížení, například váženým zprůměrováním určitého množství rodičů. Nová generace je zpravidla vytvořena deterministickou selekcí μ jedinců s nejvyšší fitness, buď z množiny potomků (pak $\lambda > \mu$), nebo z potomků i rodičů.

2.3.2 Výpočet fitness a kompetitivní koevoluce

Při implementaci libovolného evolučního algoritmu je potřeba určit způsob, jakým bude určována fitness jedinců. V kontextu optimalizace agentů pro hraní her lze často použít existující interní ohodnocení hráčů, jako dosažené skóre, počet zvládnutých úrovní nebo čas dokončení. V případě her více hráčů s nulovým součtem je nejrelevantnějším měřítkem počet vyhraných partií.

Pokud má být fitness určována hrou s oponenty, je třeba poskytnout agenty, kteří budou plnit tuto roli. Jednou možností je nechat trénované jedince hrát proti předem vytvořenému agentovi nebo sadě agentů. Algoritmus pak řeší přímočarý optimalizační problém a pracuje s absolutní fitness, umožňující lineárně uspořádat kandidáty na řešení dle kvality a zajistit (s pomocí technik jako elitismus) neklesající vývoj fitness v průběhu algoritmu.

Základní nevýhodou tohoto přístupu je samotná nutnost mít k dispozici existujícího kompetentního agenta. U komplexních her může být zapotřebí poskytnout rozmanitou sadu oponentských strategií, jinak hrozí, že výsledkem evoluce budou jedinci, kteří nereprezentují skutečnou přenositelnou kompetenci, ale pouze exploataci slabých stránek konkrétních soupeřů. Trénovací oponenti také musí mít adekvátní sílu, aby byla fitness schopná rozlišit různě kvalitní jedince. Pokud je výzva příliš vysoká, žádný z náhodně inicializovaných jedinců nemusí být schopen vyhrát a všichni obdrží nulovou fitness. Pokud je naopak příliš nízká, populace dosáhne maximální fitness po neuspokojivém objektivním zlepšení. Tento problém

[1] Vzhledem k přejímání technik a hybridizaci algoritmů lze říci, že důvody pro oddělování genetických algoritmů a evolučních strategií jako kategorií jsou spíše historické. Přesto jde o užitečnou distinkci vzhledem k odlišným typickým implementacím.

lze řešit pomocí tzv. inkrementální evoluce, kde jsou jedinci v první fázi trénováni proti jednoduchým oponentům nebo na zjednodušených scénářích hry. Do dalších obtížnějších fází pak algoritmus přechází teprve po dosažení požadované úrovně populační fitness (Risi a Togelius 2014).

Možnost obejít problémy vývoje proti dodaným externím oponentům slibuje mechanismus kompetitivní koevoluce. V rámci něj je fitness určována pomocí hry přímo mezi vyvíjenými jedinci – v základní variantě mezi členy jedné populace, nebo členy dvou souběžně vyvíjených konkurenčních populací.[2] Díky tomu není potřeba mít existující implementaci soupeřů, náročnost evaluace přirozeně roste se zlepšováním populace, a potenciál vyvinutých jedinců není nutně limitován kvalitou trénovacích oponentů nebo směrován jejich specifiky. Ideálem koevoluce je dosažení otevřeného evolučního procesu, který v duchu „závodu ve zbrojení" zabraňuje převládnutí řešení specializovaných proti jednomu druhu oponenta a vede k dlouhodobému růstu objektivní kvality řešení (Reisinger a kol. 2007).

V praxi není nasazení koevolučního algoritmu tak přímočaré. Absolutní fitness poskytovaná externím arbitrem je nahrazena relativní fitness, která závisí na ostatních jedincích v aktuální populaci, a nelze ji použít pro porovnání jedinců mezi generacemi. Důsledkem toho může být stagnace nebo klesání kvality řešení v průběhu algoritmu (Livingstone 2005). Nejlepší jedinec se totiž může kontinuálně zhoršovat, aniž by se to projevilo na jeho reprodukčním úspěchu, dokud zůstává nejsilnějším ve své populaci. Určení objektivní kvality může také komplikovat netranzitivita převahy mezi jedinci – A poráží B, B poráží C, ale zároveň C poráží A. Tato kruhovost se může projevit i na úrovni celého algoritmu, který pak sleduje omezený cyklický optimalizační cíl (Watson a Pollack 2001).

Tyto problémy bývají řešeny doplněním algoritmu o nějakou formu paměti. Jednoduchou možností je implementovat archiv nejlepších jedinců z jednotlivých generací, jehož náhodný vzorek je používán při vyhodnocování fitness za účelem zvýšení tlaku na robustnost nových řešení. U dvoupopulačního modelu lze ve fázích střídat role učitele a žáka, například podle toho, která populace aktuálně obsahuje množinu jedinců schopných porazit určité množství členů druhé populace (Rosin a Belew 1997). Existují také algoritmy postavené na technikách vytvořených pro vícekriteriální optimalizaci. Ty vycházejí z perspektivy, v níž každý koevoluční oponent představuje vlastní funkci, kterou je třeba optimalizovat. Například de Jong (2004) popsal algoritmus na bázi hierarchického archivu garantující monotonní postup směrem k nalezení množiny paretovsky optimálních řešení.

2.3.3 Neuroevoluce

Jedním z našich cílů je použít evoluční algoritmy pro trénování neuronových sítí (tzv. neuroevoluce). Jednodušší variantou, na kterou se v naší práci ome-

[2]Někdy bývá dvoupopulační model (pravděpodobně kvůli motivaci mezidruhovými metaforami predátor-kořist či parazit-hostitel) považován za součást definice kompetitivní koevoluce. My pracujeme s obecnější definicí, založenou pouze na způsobu výpočtu fitness.

zujeme, je hledání vah pro síťe s fixní strukturou. V tomto případě lze váhy reprezentovat jako vektor reálných čísel a použít běžné algoritmy pro spojitou optimalizaci. Kromě vah sítě lze také vyvíjet její topologii nebo oba aspekty zároveň. Pro tyto účely byla vypracována řada specializovaných algoritmů (např. NEAT a jeho varianty), lišících se způsobem přímého či nepřímého (generativního) zakódování sítě v jedinci a použitými evolučními technikami.

Při učení vah neuronových sítí pro hraní her mají evoluční algoritmy ve srovnání s gradientními metodami několik výhod. Oproti učení s učitelem jsou použitelné bez předem připravených dat – například záznamů partií lidských hráčů. V porovnání se zpětnovazebním učením mají jednodušší implementaci, nevadí jim řídké odměny (učící signály) a vyhodnocení fitness je přirozeně paralelizovatelné (Salimans a kol. 2017).

Obecný přehled tématu neuroevoluce v kontextu počítačových her podávají Risi a Togelius (2014). Salimans a kol. (2017) úspěšně použili evoluční strategii při učení rozsáhlé sítě, určené pro hraní sady her na konzoli Atari bez pomoci doménově specifických vědomostí. Mezi příklady použití neuroevoluce při tvorbě umělé inteligence pro strategické hry patří práce Zhen a Watson (2013) a Iuhasz a kol. (2014), kteří použili varianty algoritmu NEAT pro vývoj sítí ovládajících jednotky v omezených scénářích hry Starcraft. Bryant a Miikkulainen (2003) optimalizovali váhy sítě ovládající jednotky ve vlastní tahové hře, přičemž váhy pro každý neuron vyvíjeli v samostatné populaci.

3. AI agenti

V této kapitole popisujeme architekturu herních agentů vytvořených v rámci práce.

3.1 Testovací agenti

Nejjednodušším implementovaným agentem je *Pasivní agent*, který nevykonává žádné herní akce, pouze okamžitě ukončí své kolo. Tento agent může být užitečný pro rychlé testování základní funkcionality dalších AI.

Dále byl vytvořen *Náhodný agent*. V každém kole pro každou jednotku náhodně (s rovnoměrným rozdělením) zvolí jednu z akcí, které má k dispozici, nebo neprovede žádnou akci. Obdobně v každé továrně náhodně vyrobí jednu z dostupných jednotek, nebo nevyrobí žádnou.

3.2 Agent s rozhodovacími stromy

První AI, schopnou smysluplně hrát a vyhrát hru, je *Agresivní agent*. Ten implementuje přímočarou ofenzivní strategii pomocí dvou ručně definovaných deterministických rozhodovacích procedur, logicky reprezentujících průchod rozhodovacím stromem.

Jedna z procedur všem hráčovým jednotkám postupně přiřadí akci pro dané kolo. Upřednostňuje obsazování polí, následované útokem na výhodné cíle, následované posunem směrem k nepřátelskému velitelství nebo stažením se do obrané pozice, pokud je jednotka příliš oslabená. Druhá procedura rozhodne pro každou hráčovu továrnu, jakou má vytvořit jednotku. Usiluje přitom o dosažení fixní kompozice armády, skládající se z určitého množství jednotek každého typu.

3.3 Agent s užitkovými funkcemi

Dále byl implementován *Užitkový agent*. Ten vybírá nejlepší akce pomocí sady ručně definovaných ohodnocovacích funkcí. Agent pracuje tak, že vygeneruje všechny dostupné akce hráče a spočítá pomocí odpovídajících funkcí jejich užitkovou hodnotu v intervalu (0, 1). Následně deterministicky vybere a naplánuje akci s nejvyšší hodnotou. Toto provádí iterativně tak dlouho, dokud mu v aktuálním kole zbývají nějaké akce. Součástí výpočtů jsou také externí parametry umožňující konfigurovat a optimalizovat chování agenta.

Agent má ohodnocovací funkci pro každý podporovaný typ herní akce, tj. pro akce jednotek přesun, útok a obsazení, a pro akci zakoupení nové jednotky na hráčem obsazené továrně. Pro ilustraci zde popíšeme návrh některých funkcí. Přesný tvar všech funkcí lze prostudovat ve zdrojovém kódu v elektronické příloze.

Protože jednotka může v rámci jednoho kola provést přesun následovaný akcí útok, nebo obsazení (má-li správný typ a dostupné cíle), spočítá agent pro jednotku nejprve pomocné skóre všech polí, na něž se může přesunout. To se pak dále podílí na ohodnocení akcí prováděných z těchto polí. Skóre herního pole se skládá ze dvou hodnot odhadujících „riziko" a „příležitost", které pole nabízí. Při jejich výpočtu se používají hodnoty z map vlivu, pomocí nichž agent analyzuje relativní lokální síly svých a nepřátelských jednotek a rozmístění obsaditelných polí na mapě.

Jako příklad konkrétní funkce uveďme výpočet rizika r pole p pro jednotku j. Výpočet používá součet místních vlivů jednotlivých typů nepřátelských jednotek rs, vážených relativní efektivitou typu nepřítele proti typu agentovy jednotky. Výsledek je dále vážen obraným bonusem pole a zbývajícími životy jednotky. Funkce má tvar

$$risk(j,p) = \left(q - \frac{bonus(p)}{40}\right)\left(\sum_i rs(i)\right)\left(2 - \frac{hp(j)}{100}\right),$$

kde q je externí parametr. Podobným způsobem je vypočtena hodnota příležitosti pole. Ta se skládá z útočné a obranné příležitosti, vypočtené z hodnot místního vlivu nepřátelských, resp. spojeneckých obsazených polí. Podíl, jaký mají obě dílčí hodnoty na celkové příležitosti, závisí na globálním poměru sil hráčů na mapě. Agent je tak motivován hrát v případě převahy agresivně, a naopak hrát defenzivně, pokud je ve slabší pozici.

Ohodnocující funkce konkrétních akcí zahrnují hodnoty rizika a příležitosti jako faktory do svých výpočtů. Při ohodnocení akcí útoku se dále zohledňují poměry ztrát v životech jednotek a v jejich finanční hodnotě. Pro akci obsazení pole se zohledňuje obecná užitečnost daného typu pole nebo rychlost, jakou je jednotka schopná pole obsadit. Při ohodnocování akcí pro nákup jednotek se berou v potaz složení armád obou hráčů a relativní zastoupených typů jednotek.

Celkové fungování agenta je významně ovlivněno sadou (v dodané implementaci dvaceti) externích parametrů. Tyto parametry jsme dále optimalizovali pomocí evolučních algoritmů.[1]

3.4 Agent s neuronovými sítěmi

Poslední implementovanou AI je *Neuronový agent*. Ten má stejnou architekturu jako Užitkový agent, ale ručně definované ohodnocovací funkce nahrazuje sadou neuronových sítí typu MLP. Celkem agent používá pět sítí. Přehled rozměrů a rolí jednotlivých sítí podává tabulka 3.1. Velikosti vrstev jsou udávány v pořadí pro vstupní, skrytou a výstupní vrstvu. Všechny sítě mají jednu skrytou vrstvu. Poznamenejme, že velikosti skrytých vrstev a velikost výstupu pomocné

[1] Původně jsme nalezli slušně fungující sadu parametrů i pomocí ručního ladění. Poté, co byla v pozdní fázi vývoje v kódu hry nalezena a opravena nekritická, ale vlivná logická chyba, začal agent se stejnými parametry hrát zásadně hůře. Tato anekdota ilustruje zrádnost „ad hoc" metod herní AI.

sítě č. 1, byly určeny arbitrárně. Existuje proto potenciál pro zlepšením výkonu agenta pomocí prostého nastavení lepších hodnot těchto hyperparametrů.

Sítě pracují s implicitním prahem, který spočívá v rozšíření každé vrstvy (kromě poslední) o neuron s výstupní hodnotou 1. Velikost prahu je tak určena odpovídajícími vahami.

Všechny sítě používají aktivační funkci SoftSign s předpisem

$$f(x) = \frac{x}{1 + |x|}.$$

Tuto méně běžnou funkci jsme původně zvolili jako výpočetně rychlejší alternativu funkce tanh, se kterou sdílí pro potřeby ohodnocovacích funkcí vhodný obor hodnot $(-1, 1)$. Vzhledem k dobrým výsledkům v předběžném testování jsme u funkce SoftSign zůstali. Při dalších pokusech by bylo vhodné vyzkoušet kombinaci některé z populárních neomezených funkcí (ReLU a její varianty) s omezenou aktivační funkcí v poslední vrstvě sítě.

č.	**Role**	**Rozměry vrstev**	**Počet vah**
1	Pomocné ohodnocení herních polí	28, 14, 3	451
2	Ohodnocení akcí útok	9, 4, 1	45
3	Ohodnocení akcí obsazení	7, 3, 1	28
4	Ohodnocení akcí přesun	3, 3, 1	16
5	Ohodnocení typů jednotek pro nákup	17, 12, 8	320

Tabulka 3.1: Přehled sítí používaných Neuronovým agentem

Sítě se používají podobným způsobem, jako ohodnocovací funkce v Užitkovém agentovi. Pro každou hráčovu jednotku jsou nejprve zpracována všechna dosažitelná pole, následně jsou ohodnoceny všechny akce dostupné z těchto polí. Místo explicitně zformulovaných faktorů jako „riziko" a „příležitost" jsou výstupy pomocné sítě pro ohodnocení polí tři anonymní čísla, jejichž „role" se určuje nepřímo v interakci s ostatními sítěmi během procesu učení vah. Dále sítě používají jako vstupy kombinaci lokálních a globálních statistik, které nám přišly relevantní pro daný typ akce. Přehled vstupů jednotlivých sítí podávají tabulky 3.2 až 3.6.

Celkově byla z hlediska porozumění hře tvorba Neuronového agenta jednodušší, než obou předchozích. Agent sice stále stojí na doménových znalostech, implementovaných v podobě volby vstupů sítí, ovšem žádné jiné volby jsme při návrhu dělat nemuseli. Druhou stranou této skutečnosti je to, že bez procesu učení je agent nepoužitelný, protože náhodně inicializované váhy vedou k prakticky náhodné hře. Zároveň po skončení procesu učení představuje agent „černou skříňku", ze které nelze jednoduše přenést získanou kompetenci.

Pozice	Hodnota
0	Počet životů jednotky / 100
1	Pokud jednotka může obsadit pole, tak 1, jinak 0
2	Obranný bonus pole / 40
3-9	Vlivy hráčových jednotek jednotlivých typů
10-16	Vlivy nepřátelských jednotek jednotlivých typů
17	Vliv hráčova velitelství
18	Vliv hráčových továren
19	Vliv hráčových měst
20	Vliv nepřátelského velitelství
21	Vliv nepřátelských továren
22	Vliv nepřátelských a neutrálních měst
23	Poměr počtů polí obsazených hráči
24	Poměr financí hráčů
25	Ohrožení hráčova velitelství
26	Ohrožení oponentova velitelství
27	Souhrnná relativní síla blízkých nepřátelských jednotek

Tabulka 3.2: Vstupy sítě č. 1 (ohodnocení polí)

Pozice	Hodnota
0-2	Výstupy sítě č. 1
3	Počet životů jednotky po útoku / 100
4	Pokud po útoku jednotka přežije, tak 1, jinak 0
5	Počet životů nepřítele po útoku / 100
6	Pokud po útoku nepřítel přežije, tak 1, jinak 0
7	Poměr finančních ztrát způsobených útokem
8	Poměr financí hráčů

Tabulka 3.3: Vstupy sítě č. 2 (ohodnocení akcí útok)

Pozice	Hodnota
0-2	Výstupy sítě č. 1
3	Pokud je na poli město, tak 1, jinak 0
4	Pokud je na poli továrna, tak 1, jinak 0
5	Pokud je na poli velitelství, tak 1, jinak 0
6	Poměr financí hráčů

Tabulka 3.4: Vstupy sítě č. 3 (ohodnocení akcí obsazení)

Pozice	Hodnota
0-2	Výstupy sítě č. 1

Tabulka 3.5: Vstupy sítě č. 4 (ohodnocení akcí přesun)

Pozice	Hodnota
0	Poměr financí hráčů
1	Poměr financí na továrnu
2-8	Poměrné zastoupení typů jednotek v hráčově armádě
9-15	Poměrné zastoupení typů jednotek v oponentově armádě
16	Poměr počtu vlastněných budov mezi hráči

Tabulka 3.6: Vstupy sítě č. 5 (ohodnocení kupovaných jednotek)

4. Optimalizace parametrů

Užitkový a Neuronový agent používají při výpočtech jisté množství konfigurovatelných parametrů. Neuronový agent pracuje se stovkami vah neuronových sítí, jejichž hodnoty je potřeba nalézt vhodným algoritmem. My jsme za tímto účelem použili evoluční algoritmy. Samotné algoritmy byly realizovány v rámci pomocné konzolové aplikace, dále nazývané *evoluční trenér*. Ta pro vyhodnocování fitness agentů používá námi implementovaný simulátor hry. Instrukce k ovládání těchto programů podává kapitola 5.

4.1 Varianty evolučních algoritmů

Během experimentů jsme pracovali se dvěma základními variantami evolučních algoritmů popsanými v sekci 2.3. První z nich je genetický algoritmus používající turnajovou selekci, jednobodové křížení a gaussovskou mutaci. Druhou je evoluční strategie, která používá náhodnou selekci rodičů bez křížení, gaussovskou mutaci pro tvorbu potomků a deterministickou selekci potomků do další generace. Tabulky 4.1 a 4.2 ukazují parametry operátorů obou algoritmů, které byly použity ve všech provedených bězích.

Parametr	**Hodnota**
Velikost selekčních turnajů	2
Pravděpodobnost selekce lepšího jedince	0,9
Pravděpodobnost křížení	0,9
Pravděpodobnost aplikace mutace	0,1
Pravděpodobnost mutace jednoho genu	0,2
Střední hodnota mutace	0
Rozptyl mutace	0,5

Tabulka 4.1: Parametry operací genetického algoritmu

Parametr	**Hodnota**
Pravděpodobnost aplikace mutace	1,0
Pravděpodobnost mutace jednoho genu	0,95
Střední hodnota mutace	0
Počáteční rozptyl mutace	0,5

Tabulka 4.2: Parametry operací evoluční strategie

Použitá evoluční strategie je typu (μ, λ). To znamená, že jedinci do další generace jsou vybíráni pouze z λ mutovaných potomků. Přidali jsme ale princip elitismu, kdy je alespoň jeden nejlepší rodič beze změny převzat do další generace. Evoluční strategie také provádí jednoduchou adaptaci rozptylu mutace v průběhu algoritmu. Použili jsme klasické „pravidlo 1/5” (Beyer a Schwefel 2002), v rámci

něhož je po určité množství generací (v našich pokusech 10) počítána statistika úspěšnosti mutace. Mutace je považována za úspěšnou, pokud vzniklý jedinec dosáhl vyšší fitness, než jeho rodič. Následně je parametr mutace σ nahrazen hodnotou σ/a, pokud byla úspěšnost mutace vyšší než než 1/5, nebo hodnotou $\sigma \cdot a$, pokud byla úspěšnost nižší. V provedených pokusech byla použita hodnota parametru $a = 0{,}9$.

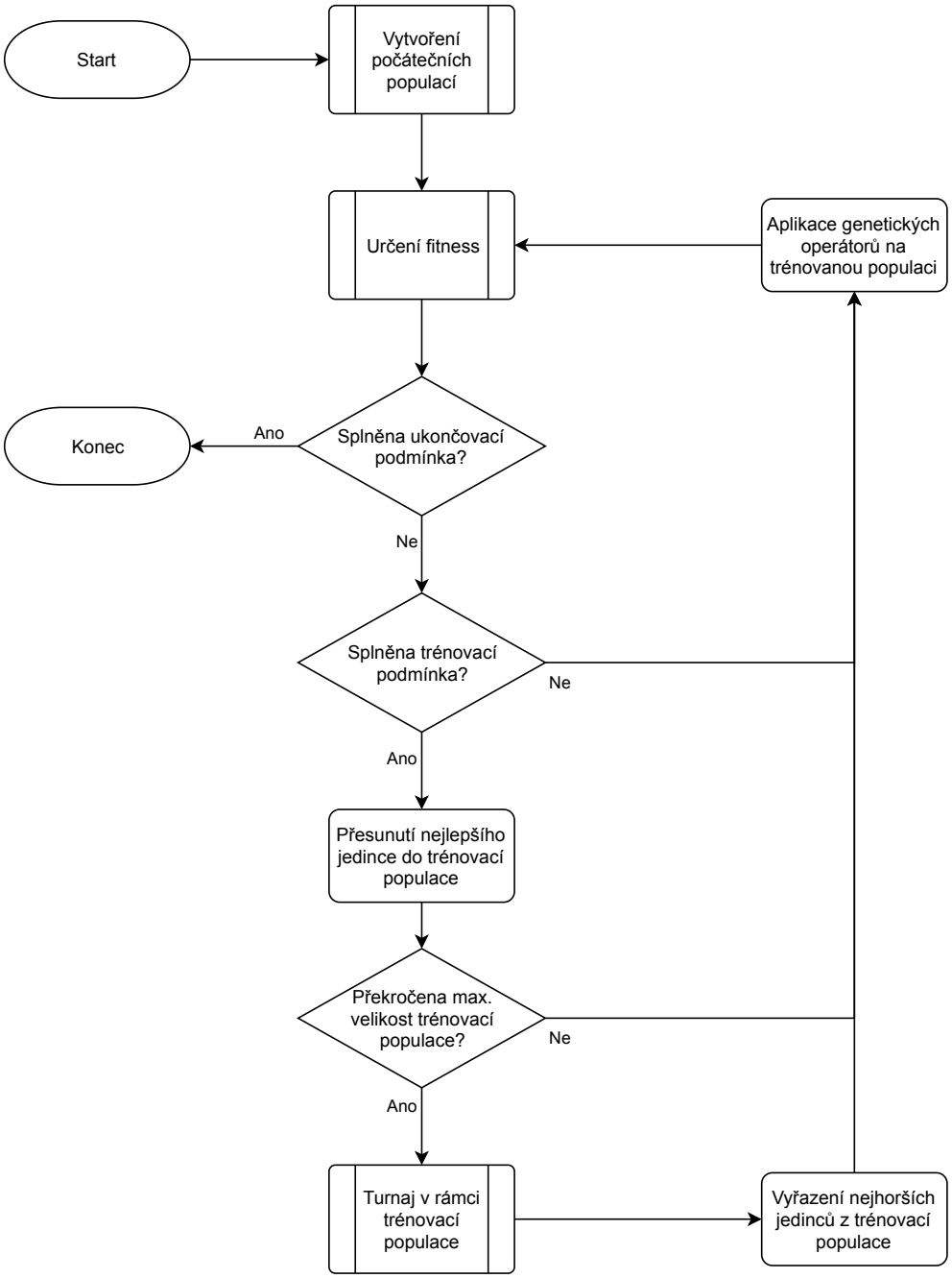
Jedince reprezentujeme jako vektory reálných čísel, která jsou před zahájením hry namapována na parametry trénovaného agenta. Fitness jedince je určována z výsledků instance agenta s odpovídající konfigurací parametrů v určité sadě zápasů provedených v simulátoru hry. Protože prostý počet výher nedokáže příliš dobře rozlišit výkon jednotlivých agentů (zejména při dosažení lokálního optima), použili jsme pro porovnání fitness jedinců místo jedné skalární hodnoty vícerozměrnou strukturu obsahující údaje o počtu výher, remíz a proher, a také o délce vyhraných a prohraných partií v počtu herních kol. Při porovnání dvou jedinců je pak za lepšího považován jedinec (v tomto pořadí důležitosti):

1. s vyšším počtem výher,
2. s nižším počtem proher (tj. ten, který v nevyhraných hrách více remizoval),
3. s nižší celkovou délkou vyhraných her,
4. s vyšší celkovou délkou prohraných her.

Lze se domnívat, že tato metrika do algoritmu přidává tlak na agresivnější hru agentů, která nemusí být objektivně silnější. V předběžném testování jsme si ale ověřili její pozitivní dopad na schopnost konvergence k jedincům dosahujícím vyššího počtu výher, který zůstává hlavní optimalizovanou hodnotou.

4.2 Trénovací oponenti

Jednotlivé experimenty se kromě použitého základního evolučního algoritmu liší také druhem oponentů, proti nimž je agent trénován. První kategorií je optimalizace hry proti fixní množině agentů. Dále byl vyzkoušen koevoluční vývoj v rámci jedné populace. Nedostatky tohoto přístupu, diskutované v sekci 2.3.2, jsme se pokusili řešit vytvořením inkrementálního koevolučního algoritmu (obrázek 4.1). Ten vedle trénované populace udržuje malou trénovací populaci a také dlouhodobý archiv předchozích úspěšných řešení. Během vývoje trénovaní jedinci získávají fitness pomocí hry se členy trénovací populace, úspěšní jedinci jsou pak do trénovací populace přesouváni. Při překročení maximální velikosti trénovací populace je sehrán turnaj mezi stávajícími členy, novými kandidáty a náhodným vzorkem archivu. Vzhledem k časové náročnosti tréninku proti většímu množství oponentů bylo v praxi potřeba držet velikost trénovací populace nízko. Vyzkoušena byla také kombinace inkrementální koevoluce s hrou proti fixním oponentům.



Obrázek 4.1: Schéma inkrementálního koevolučního algoritmu

4.3 Experimenty

Celkem jsme provedli 10 běhů evolučních algoritmů. Jejich konfigurace představuje tabulka 4.3. Zkratky „GA" a „ES" v prvním sloupci indikují, že byl v dané konfiguraci použit genetický algoritmus, respektive evoluční strategie.

Během všech pokusů byla fitness jedinců vyhodnocována hrou na stejné trénovací sadě herních map. Vedle ní jsme připravili také testovací sadu map, kterou byla použita při dalším porovnání výsledných agentů. Přehled základních parametrů map podává tabulka 4.4. Pro každé párování hráčů byly odehrány vždy dvě hry, mezi nimiž si agenti vystřídali pozice na mapě. Pro každou hru byl nastaven limit délky 200 kol (tedy 100 na hráče), po jehož překročení skončila remízou.

ID	**Agent**	**Oponenti**	**Populace**	**Potomci**	**Generace**
U1-GA	Užitkový	Agresivní agent	100	100	200
U1-ES	Užitkový	Agresivní agent	11	100	200
N1-GA	Neuronový	Agresivní agent	100	100	200
N1-ES	Neuronový	Agresivní agent	11	100	200
U2-GA	Užitkový	Nejlepší agent z U1	100	100	100
N2-ES	Neuronový	Nejlepší agent z U1	11	100	100
U3-GA	Užitkový	Přímá koevoluce	20	20	50
U4-ES	Užitkový	Inkrementální koevoluce	11, 3-5	50	100
N4-ES	Neuronový	Inkrementální koevoluce	11, 2-5	40	100
N5-ES	Neuronový	Inkrementální koevoluce + Agresivní + nejlepší z U1	10, 1-2	50	100

Tabulka 4.3: Konfigurace experimentů

Název ve hře	**Rozměry**	**Počet měst**	**Počet továren**
Trénovací sada			
Close Combat	10x10	5	4
Mountain City	16x16	18	6
Sabre Range	15x11	10	4
Zero Wood	15x11	7	4
Asphalt Maze	15x11	4	8
Spann Island	16x12	10	8
Bean Island	16x15	18	8
Testovací sada			
Ridge Island	19x16	13	9
Volcano Island	16x15	16	8
Green Valley	21x17	24	8
Normal Map	20x20	14	6
Squash Island	48x15	25	13
Liasion Wood	19x14	4	0

Tabulka 4.4: Přehled herních map použitých v rámci experimentů

Konfigurace	Trénovací mapy			Testovací mapy			% výher
	Výhry	*Remízy*	*Prohry*	*Výhry*	*Remízy*	*Prohry*	
U1-GA	13	0	1	6	1	5	73,08%
U1-ES	13	0	1	5	4	3	69,23%
N1-GA	8	3	3	4	4	4	46,15%
N1-ES	14	0	0	6	3	3	**76,92%**
U2-GA	8	1	5	6	2	4	53,85%
N2-ES	5	2	7	6	5	1	42,31%
U3-GA	10	1	3	6	2	4	61,54%
U4-ES	10	0	4	6	3	3	61,54%
N4-ES	8	1	5	6	2	4	53,85%
N5-ES	11	0	3	9	2	1	**76,92%**

Tabulka 4.5: Výsledky ve hře proti Agresivnímu agentovi

Konfigurace	Trénovací mapy			Testovací mapy			% výher
	Výhry	*Remízy*	*Prohry*	*Výhry*	*Remízy*	*Prohry*	
U1-GA	3	9	2	2	9	1	19,23%
N1-GA	4	2	8	0	8	4	15,38%
N1-ES	4	1	9	3	7	2	26,92%
U2-GA	7	4	3	2	8	2	34,62%
N2-ES	12	0	2	5	7	0	65,38%
U3-GA	5	6	3	1	7	4	23,08%
U4-ES	4	5	5	2	7	3	23,08%
N4-ES	9	0	5	2	7	3	42,31%
N5-ES	12	0	2	9	3	0	**80,77%**

Tabulka 4.6: Výsledky ve hře proti nejlepšímu agentovi z konfigurace U1-ES

4.3.1 Výsledky

Grafy vývoje fitness pro jednotlivé konfigurace lze nalézt v příloze A. Zde prezentujeme výsledky hry nejlepších agentů vytvořených v rámci každého běhu. U konfigurací s fixním oponentem je nejlepší jedinec určen přímo dle dosažené fitness. V případě koevolučních konfigurací byl nejlepší jedinec určen na konci turnajem úspěšných jedinců z celého běhu.

Tabulky 4.5 a 4.6 obsahují výsledky her proti Agresivnímu agentovi a proti nejlepší variantě Užitkového agenta, která sama vznikla při optimalizaci proti Agresivnímu agentovi (konfigurace U1-ES). Tabulka 4.7 pak obsahuje výsledky závěrečného turnaje všech vytvořených agentů proti všem ostatním. V tabulkách uvádíme zvlášť výsledky z her na trénovacích a testovacích mapách. Poslední sloupec obsahuje souhrné procento vyhraných her na obou sadách herních map.

Získaná data ukazují několik zajímavých skutečností. Za nejdůležitější výsledek považujeme úspěch konfigurace N5-ES, tedy Neuronového agenta trénovaného pomocí kombinace hry proti fixním agentům a inkrementální koevoluce. Tato konfigurace jako jediná dosáhla dobrých výsledků na trénovací i testovací sadě dat, a to jak proti dvěma fixním agentům, kteří byli součástí její trénovací množiny,

Konfigurace	Trénovací mapy			Testovací mapy			% výher
	Výhry	*Remízy*	*Prohry*	*Výhry*	*Remízy*	*Prohry*	
Agresivní AI	32	8	100	32	28	60	24,62%
U1-GA	61	30	49	25	58	37	33,08%
U1-ES	52	27	61	24	67	29	29,23%
N1-GA	37	20	83	14	54	52	19,62%
N1-ES	55	19	66	38	45	37	35,77%
U2-GA	53	24	63	28	54	38	31,15%
N2-ES	67	11	62	34	59	27	38,85%
U3-GA	55	29	56	24	50	46	30,38%
U4-ES	64	26	80	34	51	35	37,69%
N4-ES	88	8	44	56	44	20	55,38%
N5-ES	102	6	32	81	30	9	**70,38%**

Tabulka 4.7: Výsledky turnaje všech agentů

tak proti variantám agentů, s nimiž nikdy nehrála. Oproti tomu konfigurace N1-ES, která dosáhla perfektní specializace ve hře proti Agresivnímu agentovi na trénovacích mapách (kde neprohraje jedinou hru), dosahuje výrazně horšího výsledku při hře proti stejnému agentovi na neznámých mapách. Podobně vychází srovnání s konfigurací N2-ES. Zdá se tedy, že přidání koevolučního elementu vedlo k vytvoření robustnějšího agenta.

Prostá koevoluce a inkrementální koevoluce bez přidaných fixních oponentů si vedly hůře. Z porovnání konfigurací U3-GA a U4-ES se zdá, že samotný mechanismus trénovací populace není dostatečným vylepšením.

Celkově se potvrdila očekávaná převaha Neuronového agenta nad Užitkovým. Oba jsou pak schopní se naučit silné hře proti statickému Agresivnímu agentovi. Kromě diskutované konfigurace N5-ES, se ale žádné jiné nepodařilo tuto znalost přesvědčivě přenést do nových scénářů. Zajímavou skutečností je, že zatímco Užitkový agent se učil přibližně stejně dobře s genetickým algoritmem i evoluční strategií, v případě učení vah neuronových sítí podávala znatelně lepší výsledky evoluční strategie.

Na základě krátkého pozorování charakteru hry trénovaných agentů lze říci, že nejpřirozeněji hraje Neuronový agent. Agresivní agent hraje smysluplným, ale monotónním a omezeným způsobem. Užitkový agent je lépe schopen využívat vlastností herní mapy a různých typů jednotek, častěji ale vykonává „podivné" a necílevědomé akce. Hra Neuronového agenta více připomíná hru člověka, který ví, co je cílem hry. Za potěšující považujeme například skutečnost, že se agent sám naučil uhýbat jednotkou, která stojí na obsaditelném poli, ale sama není schopná jej obsadit. U ostatních agentů jsme za tímto účelem museli přidat zvláštní pravidlo, protože se stávalo, že si agent zastoupil oponentovo velitelství nevhodnou jednotkou a nebyl schopný hru dokončit.

5. Uživatelská dokumentace

Tato kapitola obsahuje instrukce k použití programů vytvořených v rámci práce. Všechny souborové cesty jsou uváděné relativně vůči kořeni elektronické přílohy. Přehled jejího obsahu je uveden v příloze B. V rámci této kapitoly je také nastíněna konfigurace některých aspektů běhu programů pomocí vstupních konfiguračních souborů. Úplný popis formátů těchto souborů a všech parametrů lze nalézt v elektronické příloze v dokumentu `Docs/formaty.pdf`.

5.1 Instalace a spuštění

Software není nutné instalovat, stačí zkopírovat obsah elektronické přílohy do lokality s povolením k zápisu. Spustitelné soubory jednotlivých programů se nachází v odpovídajících podadresářích `GameClient`, `AgentTrainer` a `MapEditor` adresáře `Binaries`.

Software je distribuován ve verzi pro operační systémy Microsoft Windows (podporovány jsou Windows 7 64-bit a novější) a Linux (testováno v distribuci Linux Mint 19). Herní klient může ke spuštění na Windows vyžadovat instalaci Visual C++ Redistributable 2017. Evoluční trenér vyžaduje instalaci .NET Core verze 2.0 nebo novější. Editor map je dostupný pouze pro Windows a vyžaduje instalaci .NET Framework 4.0 nebo novější.

5.2 Instrukce pro sestavení

Pro sestavení na platformě Windows jsou k dispozici projekty vývojového prostředí Microsoft Visual Studio spravované řešením `Source/OpenAW.sln`. Při překladu herního klienta (projekty v adresáři `Source/GameClient`) je kvůli podpoře standardu C++17 vyžadována verze prostředí Visual Studio 2019 nebo Visual Studio 2017 15.7 a novější.

Pro sestavení na platformě Linux je připraven makefile v adresáři `Source`. Součástí distribuce jsou knihovny linkované herním klientem, které byly přeloženy pomocí překladače GCC 9.2. Při použití jiného překladače může být nutné nejprve přeložit knihovny SFML 2.5.1 a TGUI 0.8.6 a umístit vzniklé .so soubory do podadresářů `SFML/libs` a `TGUI/libs` adresáře `Source/GameClient/Dependencies`.

Program AgentTrainer vyžaduje na obou platformách pro sestavení instalaci .NET Core SDK verze 2.0 nebo novější.

5.3 Grafický klient

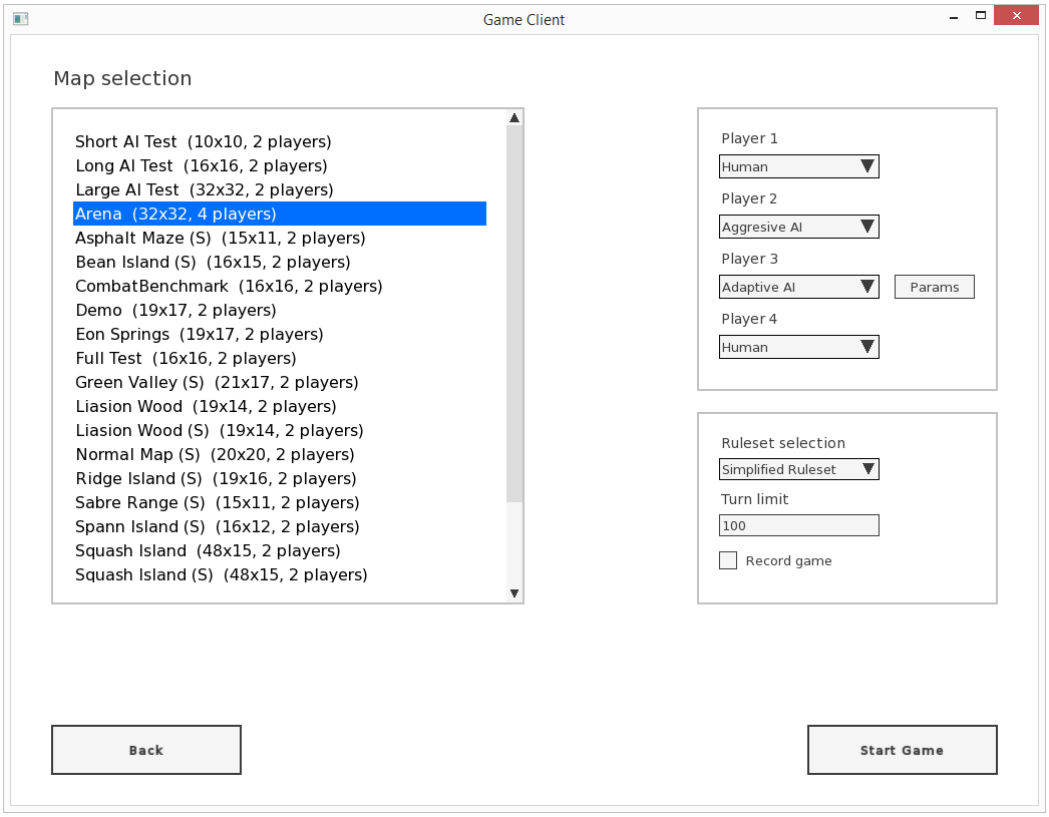
Herní klient má dvě varianty – grafickou a konzolovou. Nejprve popíšeme ovládání grafického klienta.

5.3.1 Menu hry

Úvodní obrazovka klienta obsahuje tlačítkové menu sloužící pro přechod k nastavení nové hry (položka *New game*), načtení uložené hry (*Load game*), přehrání záznamu hry (*Recorded games*) a ukončení programu (*Exit*).

Dialog pro nastavení nové herní relace ukazuje obrázek 5.1. Box v levé části obrazovky slouží k výběru mapy, na níž se má hrát. V pravé horní části je sekce pro nastavení typů hráčů. Zde jde u každého hráče (počet je dán zvolenou mapou) určit, zda půjde o lidského hráče, nebo vybrat typ počítačového agenta. U parametrizovatelných agentů také lze pomocí tlačítka *Params* vybrat některou z dostupných sad parametrů. Níže se nachází sekce obsahující další nastavení herní relace. Zde lze zvolit sadu pravidel (k dispozici je plná a zjednodušená varianta hry), nastavit limit počtu kol (0 znamená žádný limit) a pomocí přepínače určit, zda se má ze hry uložit záznam.

Načtení uložené hry nebo záznamu hry zprostředkovávají jednoduché dialogy pro výběr souboru odpovídajícího typu.



Obrázek 5.1: Obrazovka nastavení nové hry

5.3.2 Herní relace

Hlavní částí klienta je obrazovka samotné herní relace. Obrazovka se skládá ze dvou částí – herní mapy a spodní lišty. Spodní lišta obsahuje tlačítko *Menu* otevírající menu s možnostmi pro uložení hry, rychlé načtení poslední uložené hry

a ukončení herní relace. Uprostřed lišty se nachází informační panel zobrazující jméno hráče, který je právě na tahu a množství jeho financí. Aktuální hráč je indikován také barvou lišty. V pravém rohu lišty se nachází tlačítko pro ukončení kola. Tlačítko je aktivní pouze když je na tahu lidský hráč.

Zbytek obrazovky zabírá dvourozměrná reprezentace herní mapy, obsahující mřížku herních polí a na nich umístěných jednotek. Každé pole a jednotka má přiřazený obrázek podle svého typu. Jednotky a pole jednotlivých hráčů jsou odlišeny pomocí barevného schématu odpovídajícímu barvě hráče (šedivé „budovy" jsou neobsazená pole nepatřící žádnému hráči). Čísla a ikony u obrázků jednotek reprezentují počet jejich životů (v pravém dolním rohu jednotky) nebo informace o dalším stavu jednotky, jako je postup obsazování pole nebo transport jiné jednotky (značeno nad jednotkou).

Obrázek 5.2: Obrazovka herní relace

Ovládání obrazovky herní relace

Ovládání mapy: Pokud se herní mapa nevejde do aktuálního okna programu, lze posouvat viditelným výřezem pomocí šipek na klávesnici nebo najetím kurzorem myši k odpovídajícímu okraji.

Výběr objektů na mapě: Hráč může kliknutím levým tlačítkem myši označovat své objekty na mapě, tj. jednotky a obsazená pole. Označen může být vždy pouze jeden objekt. Kliknutím na část mapy neobsahující označitelný objekt se hráč vrátí do stavu, kdy není označen žádný objekt.

Po označení objektu se objeví plovoucí box, v němž se zobrazují informace o objektu, případně jsou zde interaktivní prvky pro provedení relevantní herní akce s objektem. Je-li označena jednotka, zobrazuje box název a ikonu jednotky, množství munice a paliva. Pokud má jednotka navíc k dispozici na svém poli speciální akci (obsazení pole nebo doplnění zásob), objeví se v boxu tlačítko pro provedení této akce. Je-li označeno speciální pole, na němž lze vytvářet jednotky, objeví se v boxu dialog pro zvolení typu vytvářené jednotky. Informační plovoucí box lze také zobrazit pro libovolný objekt či pole kliknutím pravým tlačítkem myši.

Akce jednotek: Označená jednotka může provést několik typů akcí. Může se zaprvé přesunout na některé z pro ni dostupných polí, zaútočit na pro ni dostupnou nepřátelskou jednotku nebo nastoupit do pro ni dostupné spřátelené „transportní“ jednotky. Relevantní cílová pole pro tyto akce jsou ve chvíli graficky označena – pomocí poloprůhledných značek na polích (šedá pro pohyb, červená pro útok, šedá s okrajem pro transport). Přepínat mezi označením transportní jednotky a jednotky jí transportované lze opakovaným klikáním myši na dané pole.

Ovládání rychlosti animací: Pomocí kláves + a – na klávesnici, lze zvyšovat, resp. snižovat rychlost všech herních animací, a to v rozsahu 0.5 až 4 násobku standardní rychlosti.

Režim přehrávání záznamu

Obrazovka herní relace je využívána ve dvou režimech: normální hra a přehrávání záznamu hry. Při něm je zablokována veškerá interaktivní funkcionalita (kromě hlavního menu) a klient pouze zobrazuje simulaci načteného záznamu. Stejně jako v běžném režimu lze pomocí kláves + a – měnit rychlost animací.

5.3.3 Spuštění s automatickou konfigurací

Grafického klienta lze pro testovací účely spustit rovnou do přednastavené herní relace. Toho lze dosáhnout předáním konfigurace ve formátu JSON jako parametru příkazové řádky. Jednou možností je zapsat konfiguraci do textového souboru a uvést relativní cestu k tomuto souboru za přepínačem `-f`. Alternativně lze JSON objekt s nastavením předat přímo jako řetězec (s adekvátně ošetřenými uvozovkami) za přepínačem `-j`:

```
> GUIClient.exe -j "{ \"key\": \"value\", ... }"
```

V rámci konfigurace je třeba určit mapu (parametr `map`), sadu pravidel (parametr `ruleset`) a nastavení hráčů (pole `players`). Kód 5.1 obsahuje ukázku konfiguračního souboru.

```
{
  "map": "../../Data/Maps/test.awmap",
  "ruleset": "simple",
  "players": [
    { "type": "human" },
    { "type": "ai", "aiConfig": { "providerId": "native",
      "agentId": "dtAgent" } }
  ]
}
```

Kód 5.1: Příklad konfigurace spuštění grafického klienta

5.4 Konzolový klient

Konzolový klient je určen pro hru počítačových agentů. Spuštění konzolového klienta se konfiguruje obdobně jako spuštění grafického klienta popsané v sekci 5.3.3. V případě konzolového klienta je ale konfigurace povinná, protože klient pracuje automaticky a není uživatelem ovládán při běhu. Na základě konfigurace klient naplánuje seznam herních relací (konkrétních her na daných mapách s danými hráči), které následně odehraje a vytvoří záznamy o jejich výsledcích.

Klient podporuje tři režimy plánování herních relací, které se liší způsobem párování zadaných hráčů. Režim se volí pomocí parametru `matching`, který může mít jednu z následujících hodnot.

V režimu `all` spolu hraje každá dvojice agentů zadaných v parametru `players`. Pro každou dvojici se naplánují dvě hry na každé z map zadaných v parametru `maps`. V režimu `vs` hraje každý z agentů zadaných v parametru `players` proti každému z agentů zadaných v parametru `vsOpponents`. Plánování her jinak probíhá stejně jako v předchozím případě. Konečně v režimu `manual` klient sehraje pouze v konfiguraci přímo zadaný seznam herních relací.

```
{
  "resultsDir": "../../Experiments/test/",
  "gameConfig": {
    "matching": "vs",
    "ruleset": "simple",
    "turnLimit": 150,
    "maps": [ "../../Data/Maps/testA.awmap", "../../Data/Maps/testB.awmap" ],
    "players": [
      { "providerId": "native", "agentId": "neuralAgent", "params": [ ... ] },
      ...
    ],
    "vsOpponents": [ { "providerId": "native", "agentId": "dtAgent" } ]
  }
}
```

Kód 5.2: Příklad konfigurace spuštění konzolového klienta

Dále je nutné specifikovat sadu pravidel (parametr `ruleset`) a limit počtu kol (`turnLimit`), které mají být použity v herních relacích. Lze také pomocí

parametru `repeats` nastavit počet opakování všech naplánovaných her. To se může hodit při použití nedeterministických agentů.

Po spuštění klient postupně sehraje všechny naplánované hry a do adresáře určeného parametrem `resultsDir` zapíše informace o výsledcích jednotlivých her a celkové statistiky agentů. Volitelně zde také uloží záznamy sehraných her (parametr `shouldRecord`).

5.5 Evoluční trenér

Evoluční trenér je pomocný program určený pro optimalizaci parametrů herních agentů. Jde o konzolovou aplikaci implementující algoritmy diskutované v kapitole 4. Stejně jako konzolový klient, jehož služeb využívá pro vyhodnocení fitness agentů, vyžaduje evoluční trenér při spuštění poskytnutí konfigurace ve formátů JSON. K dispozici je opět předání pomocí souboru s přepínačem `-f` nebo na příkazové řádce s přepínačem `-j`. Protože jde o systémově přenositelnou aplikaci běžící v prostředí .NET Core, spouští se navíc nepřímo pomocí příkazu `dotnet`. Spuštění může tedy vypadat například takto:

```
> dotnet AgentTrainer.dll -f "config.json"
```

V adresáři `Experiments/Scripts` jsou také k dispozici skripty `start_training.bat` (pro Windows) a `start_training.sh` (pro Linux) automatizující spuštění trenéra s konfigurací uloženou v souboru `default_training_config.json`.

Konfigurace programu zahrnuje množství povinných a nepovinných parametrů, jejichž kompletní specifikace je součástí elektronické přílohy. Vedle technických nastavení (souborové cesty, počet vláken pro zpracování) a herních parametrů (volba map a pravidel) jsou to zejména nastavení herního agenta, který má být optimalizován. Ty jsou shromážděné v parametru konfigurace `agentTemplate`, který obsahuje identifikátory zvoleného agenta používané herním klientem a seznamy agentových „fixních" parametrů (`fixedParams`) a parametrů určených pro optimalizaci trenérem (`trainedParams`). Fixní parametry jsou specifikovány s konkrétními hodnotami libovolného JSON typu. Optimalizované parametry mohou mít za hodnoty pouze vektory reálných čísel a jsou popsány pomocí horních a dolních mezí, mezi nimiž se tato čísla mohou v průběhu algoritmu pohybovat. Kód 5.3 obsahuje částečnou ukázku popsané konfigurace agenta.

Dále je v konfiguraci nutné zvolit pomocí parametru `trainingStrategy` variantu evolučního algoritmu („trénovací strategii"), která se má použít. V objektu `trainingStrategyParams` jsou shromážděny parametry specifické pro jednotlivé strategie. V dodaném programu jsou k dispozici následující tři možnosti.

Strategie `vsTraining` optimalizuje trénovaného agenta ve hře proti zadané množině oponentů. Oponenti jsou popsáni v alespoň jednoprvkovém poli `vsOpponents`. Velikost trénované populace nastavuje parametr `popSize`. Délka běhu algoritmu je určena počtem generací v parametru `generations`.

V rámci strategie `simpleCoevolution` hrají jedinci z trénované populace sami

```
"agentTemplate": {
  "providerId": "native",
  "agentId": "adaptiveAgent",
  "fixedParams": [
    { "id": "recruitNetLayers", "value": [27, 13, 8] },
    { "id": "moveNetLayers", "value": [15, 15, 1] },
    ...
  ],
  "trainedParams": [
    { "id": "recruitNetWeights", "count": 462, "min": -10.0, "max": 10.0 },
    { "id": "moveNetWeights", "count": 256, "min": -10.0, "max": 10.0 },
    ...
  ]
}
```

Kód 5.3: Příklad nastavení parametrů herního agenta určených pro optimalizaci

proti sobě. Jako v předchozím případě lze nastavit velikost populace a počet generací.

Strategie `incrementalCoevolution` používá algoritmus kompetitivní koevoluce s pamětí (viz obrázek 4.1). V něm jedinci z trénované populace hrají proti jedincům ze separátní trénovací populace tvořené předchozími úspěšnými jedinci. Minimální a maximální velikost trénovací populace určují parametry `minTrainingPopSize` a `maxTrainingPopSize`.

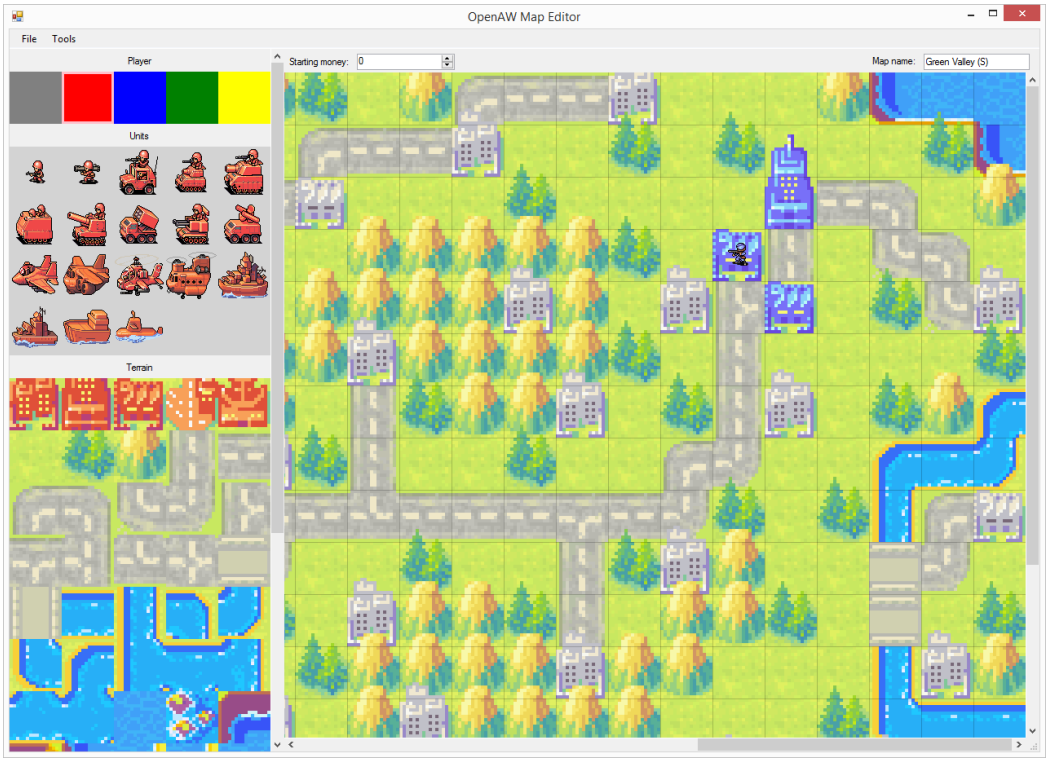
5.6 Editor map

Editor map je grafická aplikace umožňující vytvářet nové nebo upravovat existující mapy. Lze pomocí něj nastavovat parametry mapy a „kreslit“ herní pole a jednotky. Editor lze ovládat kompletně myší. Některé funkce mají navíc klávesové zkratky, jejichž přehled podává tabulka 5.1. Program tvoří jediná obrazovka znázorněná na obrázku 5.3. Následuje popis jednotlivých sekcí editoru.

Klávesy	**Funkce**
Ctrl + N	Otevře dialog pro vytvoření nové mapy
Ctrl + O	Otevře dialog pro načtení existující mapy
Ctrl + S	Uloží mapu, otevře dialog pokud se ukládá poprvé
G	Přepíná zobrazení mřížky na herní mapě
Šipky, W/A/S/D	Posouvá výběr typu objektu pro přidání do mapy

Tabulka 5.1: Seznam klávesových zkratek editoru map

V horní částí obrazovky se nachází lišta se dvěma rozbalovacími menu. Pod položkou *File* se nachází možnosti pro vytvoření nové mapy, uložení rozpracované nebo načtení existující mapy, či ukončení programu. Dialog pro vytvoření nové mapy umožňuje nastavit rozměry a název mapy. Při ukládání editor zkontroluje platnost mapy a případně upozorní na nalezené chyby (minimální počet hráčů, počet HQ). Položka *Tools* horní lišty obsahuje přepínače nastavující velikost zob-



Obrázek 5.3: Obrazovka editoru map

razení herních polí (k dispozici jsou režimy 32x32 a 64x64 pixelů) a zobrazení mřížky na herní mapě. Dále menu obsahuje tlačítko pro uložení aktuální mapy jako obrázku ve formátu PNG.

V levé části editoru se nachází panel pro výběr herních objektů vkládaných do mapy. V pořadí od shora dolů obsahuje subpanely pro výběr hráče, pro výběr přidávané jednotky a pro výběr přidávaného terénu. Výběr hráče modifikuje kterému hráči budou patřit dále přidávané jednotky a budovy. Jednotky musí vždy patřit nějakému hráči, budovy mohou být umístěné jako neutrální (neobsazené).

Centrální část editoru zabírá prostor pro grafickou reprezentaci herní mapy. Do ní lze klikáním či tažením levým tlačítkem myši kreslit zvolený terén či jednotku (dle volby v levém panelu). Terén i jednotky lze nahrazovat prostým překreslením. Jednotky lze navíc odstraňovat klikáním či tažením pravým tlačítkem myši.

Nad panelem s mapou se nachází lišta obsahující boxy pro nastavení počátečního množství financí pro aktuálně zvoleného hráče a pro nastavení jména mapy.

6. Implementace

Jedním z hlavních cílů práce bylo vytvoření herního klienta schopného provozovat hru jednak v grafickém, uživatelsky orientovaném režimu, jednak v konzolovém režimu určeném pro rychlou hru počítačových agentů. Dále bylo nutné implementovat evoluční algoritmus pro optimalizaci parametrů AI agentů. Aby bylo možné rozumně vytvářet herní mapy, bylo také potřeba implementovat grafický editor map.

V této kapitole představíme architekturu a technické parametry vytvořených programů. U každého programu uvedeme hlavní požadavky na řešení, použité technologie a důvody jejich volby. Dále popíšeme jeho logickou strukturu a objektový návrh. K podrobnějšímu studiu implementace lze využít i programátorskou dokumentaci (v angličtině) vygenerovanou z komentářů ve zdrojovém kódu pomocí programu Doxygen. Dokumentace se nachází v elektronické příloze práce v adresáři `/Docs/Doxygen`. Formáty externích datových a konfiguračních souborů jsou zdokumentovány v souboru `/Docs/formaty.pdf` elektronické přílohy.

6.1 Herní klient

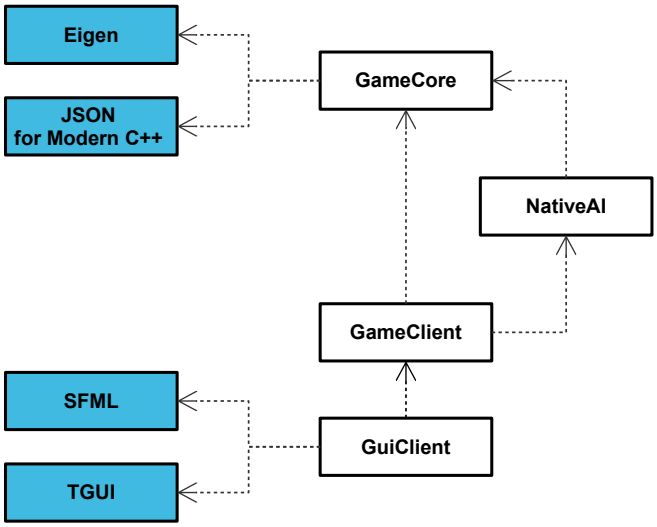
Herní klient je software vyžadující implementaci celé řady systémů. Ty lze rozdělit do několika základních kategorií:

1. Herní logika a správa herního stavu
2. Obsluha uživatelských vstupů a produkce výstupů (včetně grafické reprezentace)
3. Načítání a správa zdrojů

V rámci produkce počítačových her je dnes asi nejčastějším řešením použít některý z hotových herních enginů (např. Unity, Unreal Engine), které nabízejí rozsáhlou sadu knihoven, nástrojů a abstrakcí pokrývající většinu aspektů vývoje hry. V rámci naší práce jsme se rozhodli touto cestou nejít. Hlavními důvody tohoto rozhodnutí byla, za prvé, snaha vyhnout se běhové režii komplexního enginu, která by byla neopodstatněná vzhledem k našim nízkým nárokům na grafickou reprezentaci a naopak vysokým požadavkům na efektivitu herního simulátoru při mnohonásobném vyhodnocování fitness v rámci evolučního algoritmu. Za druhé nás motivoval prostý zájem vyzkoušet si prakticky větší část vývoje her.

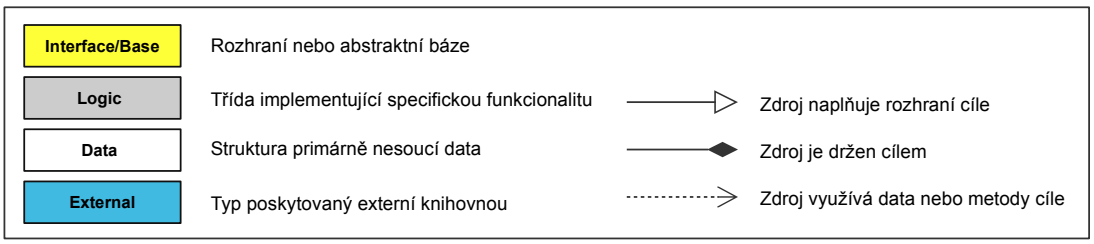
Zvolili jsme proto více nízkoúrovňové řešení napsané v jazyce C++ (standardu C++17) a složené z efektivního, na uživatelské úrovni nezávislého logického jádra doplněného grafickým klientem postaveným na jednoduché mediální knihovně SFML. Tato knihovna poskytuje systémově nezávislou obsluhu okna a vstupů klávesnice a myši a abstrakci nad OpenGL zaměřenou na tvorbu 2D her. Dále klient používá knihovnu TGUI pro realizaci grafického uživatelského rozhraní a knihovnu JSON for Modern C++ pro serializaci a deserializaci dat.

Vlastní řešení je rozdělené do několika částí (odpovídajících jednotlivým Visual Studio projektům). Jejich vztahy, včetně závislostí na externích knihovnách, znázorňuje obrázek 6.1. Knihovna `GameCore` obsahuje logický engine hry, definice a nástroje používané dalšími částmi řešení, včetně rozhraní pro tvorbu AI agentů. Toto rozhraní využívá knihovna `NativeAI` obsahující nativní C++ implementaci tzv. poskytovatele AI. Její součástí jsou také všichni v této práci popsaní AI agenti. Rozhraní pro implementaci konkrétních herních klientů a implementace konzolového klienta se nachází v projektu `GameClient`. Grafický klient je pak na tomto základě realizován v projektu `GuiClient`.

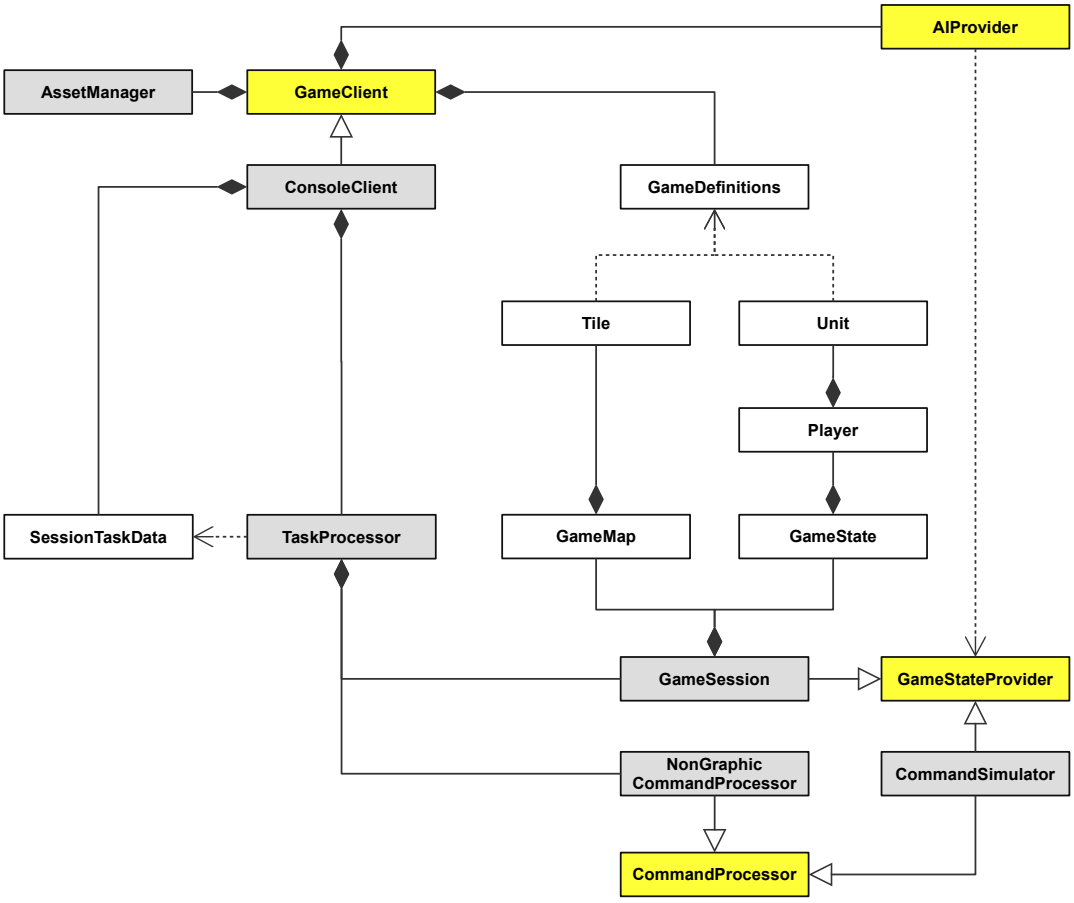


Obrázek 6.1: Schéma projektů a závislostí herního klienta

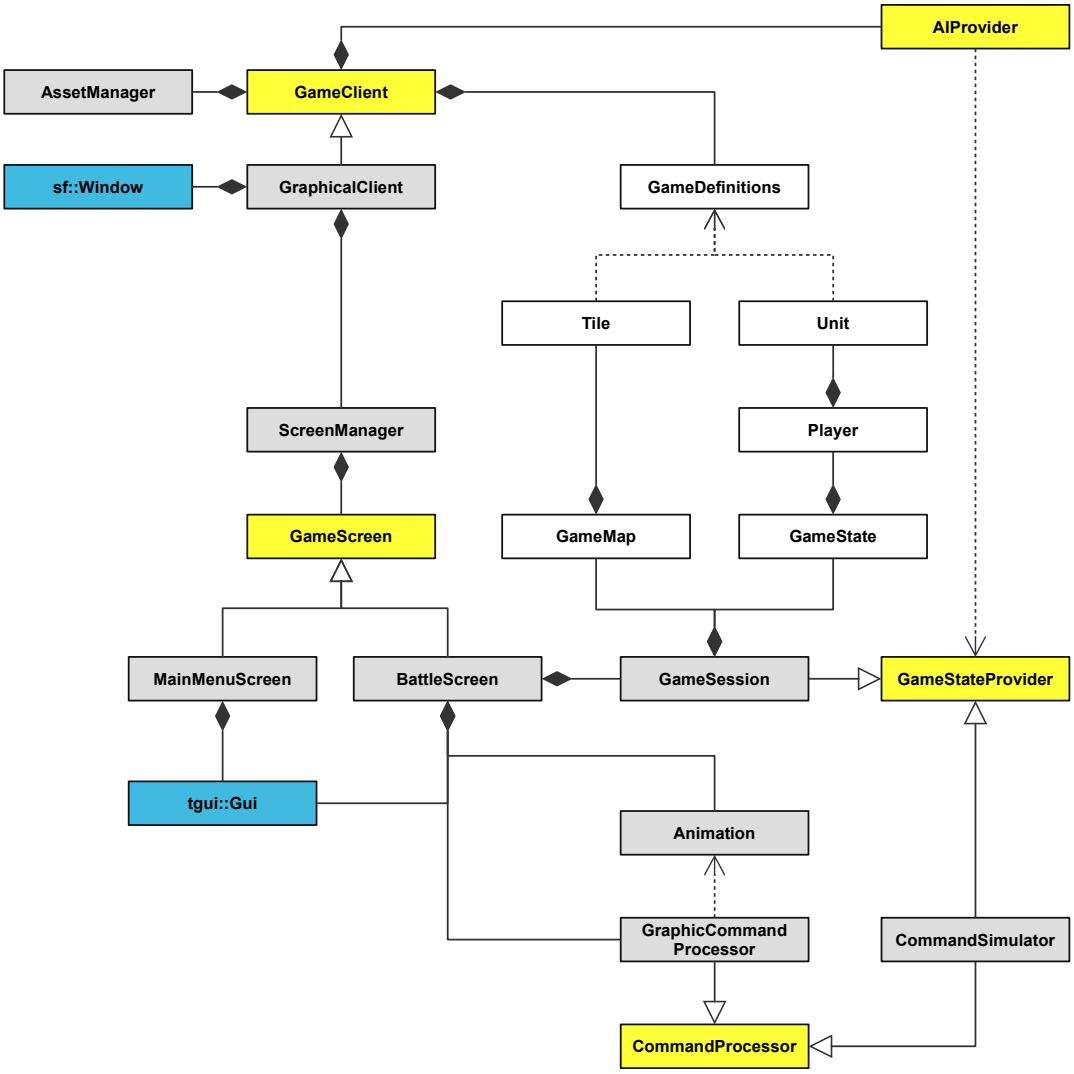
Následující text představuje hlavní aspekty implementace pomocí popisu zodpovědností a fungování klíčových tříd. Objektový návrh a vztahy mezi třídami znázorňují diagramy na obrázcích 6.3 a 6.4. První obsahuje třídy relevantní pro běh klienta v negrafickém, konzolovém režimu. Druhý obsahuje třídy relevantní pro běh v grafickém režimu. Tyto diagramy (a další diagramy objektového návrhu ve zbytku kapitoly) prezentujeme ve formátu inspirovaném jazykem UML. Obrázek 6.2 obsahuje legendu k tomuto formátu.



Obrázek 6.2: Legenda ke schématům objektového návrhu



Obrázek 6.3: Schéma tříd konzolového klienta



Obrázek 6.4: Schéma tříd grafického klienta

6.1.1 Třída GameClient a správa sdílených zdrojů

V srdci programu je instance některé z implementací abstraktní třídy `GameClient` (grafický `GraphicalClient` nebo konzolový `ConsoleClient`), která vlastní sdílené zdroje potřebné v průběhu hry a realizuje základní herní smyčku, tj. zpracování vstupů, update stavu a vykreslení. Pro načítání a správu externích zdrojů (textury, fonty, mapy) je určena instance třídy `AssetManager` zajišťující správnou inicializaci dat a přístup k načteným zdrojům skrze unikátní ID. Třída `GameClient` také drží ve struktuře `GameDefinitions` informace o vlastnostech jednotek a typech herních polí načtené z textových konfiguračních souborů. Grafický klient používá pro správu svých uživatelských stavů instanci třídy `ScreenManager` držící zásobník objektů typu `GameScreen`.

6.1.2 Třída GameScreen a fáze programu

Objekty dědící od abstraktní třídy `GameScreen` („obrazovky“) reprezentují globální stavy programu a určují jeho vzhled a funkcionalitu v daný moment. Každá třída odvozená od `GameScreen` implementuje virtuální metody jako `HandleInput`, `Update` a `Draw`. V rámci hlavní herní smyčky jsou pak volány varianty té implementace, jejíž instance leží na vrcholu zásobníku `ScreenManageru`. Přechody mezi obrazovkami jsou tedy řešeny jednoduchými zásobníkovými operacemi mezi cykly herní smyčky. Kromě toho mohou mít jednotlivé obrazovky svůj interní stav uživatelského rozhraní, spravovaný jejich instancí třídy `tgui::Gui`.

Obrazovky odpovídají jednotlivým fázím použití programu. V rámci `LoadingScreen` jsou načteny všechny sdílené externí zdroje. `MainMenuScreen` poskytuje zobrazení a obsluhu grafického menu pro spuštění nové hry a načtení uložené hry nebo záznamu. Po zvolení režimu a mapy probíhá hlavní část hry v rámci obrazovky `BattleScreen`. Zprávu o výsledku herní relace zobrazuje obrazovka `VictoryScreen`.

6.1.3 Třída BattleScreen a grafický engine hry

Nejdůležitější ze tříd obsluhujících herní smyčku je třída `BattleScreen`. Ta zodpovídá za ovládání, logiku a vykreslování samotné hry. Obrazovka se skládá ze dvou skupin interaktivních prvků: 1) herní mapy a jejích objektů, 2) GUI elementů (tlačítek, ikon, apod.) spravovaných instancí třídy `tgui::Gui`.

V rámci obsluhy vstupů jsou události dodávané knihovnou SFML rozděleny: stisky kláves jsou vyhodnoceny přímo, události myši jsou nejprve předány do fronty správce GUI pro otestování stisků tlačítek či výběrových boxů, nezachycené stisky jsou testovány oproti herní mapě. Validní kliknutí do mapy jsou vyhodnoceny podle typu zasaženého objektu a pozice jednoduchého stavového automatu. Tak je zajištěn výběr jednotek a budov, udílení příkazů k pohybu, atd.

Vykreslování hry probíhá dvěma způsoby. Menší jednotlivé objekty (jak herní jednotky, tak různé animační efekty, texty, tlačítka, atd.) jsou vykreslovány pomocí separátních volání `draw` na jejich grafické reprezentace. Protože počet těchto

volání je optimálně třeba držet co nejnižší, není statická herní mapa vykreslována stejným způsobem po jednotlivých polích, ale pomocí předpočítaného bufferu typu `sf::VertexArray`, který umožňuje mapu odeslat na grafickou kartu v rámci jednoho volání.

Animace (pohyb jednotek, „létající texty“ indikující hodnoty útoků, apod.) jsou řešeny pomocí fronty objektů odvozených od abstraktní třídy `Animation`. V každém cyklu herní smyčky si animační objekty přepočítají svoje parametry a vykreslí se. Pro dosažení plynulého průběhu animací a pohybu kamery je relevantním metodám předávána časová delta od posledního snímku, kterou jsou škálovány všechny změny.

6.1.4 Třída GameState a reprezentace herní pozice

Aktuální logický stav hry (herní pozice) je oddělen od ostatních subsystémů souvisejících se vstupy či výstupy. Je realizován strukturou `GameState` obsahující následující data:

- Informace o průběhu hry – který hráč je na tahu, seznam aktivních hráčů, pořadí střídání hráčů.
- Informace o hráčích – seznam jednotek na mapě, seznam obsazených polí, množství financí.
- Informace o jednotkách – typ (konstantní), pozice na mapě, HP, munice, palivo, postup obsazení a další údaje.

Každý hráč je reprezentován objektem typu `Player`, který drží seznam vlastněných jednotek a obsazených polí. Jednotky jsou reprezentovány instancemi typu `Unit`, grafický klient používá odvozený typ `UnitWithGraphics` s přidanými grafickými daty. Pro účely vyšší efektivity algoritmů (zejména hledání cest a dostupných cílů) jsou některá data herního stavu zdvojena v objektu typu `GameMap`. Zde jsou u jednotlivých políček kromě údaje o typu pole vedeny informace o přítomných jednotkách (pomocí nevlastnících ukazatelů) či o obsazení hráči. Všechna logicky orientovaná data jedné herní relace (tedy stav, mapa, odkazy na použité definice a sadu pravidel) zapouzdřuje struktura `GameSession`.

6.1.5 Třídy Command a CommandProcessor

Z hlediska modifikace herního stavu je oddělení logické a uživatelské úrovně realizováno pomocí mechanismu příkazů. Uživatelská úroveň nemodifikuje herní stav přímo, ale skrze frontu příkazů zpracovávaných tzv. zpracovateli. Příkazy jsou instance typů odvozených od třídy `Command` nesoucí minimální data nutná k provedení příkazu z hlediska hry (např. pro pohyb jednotkou struktura `MoveCommand` obsahuje pouze souřadnice jednotky a souřadnice cíle). Zpracovatelé jsou instance typů odvozených od třídy `CommandProcessor`, které s příkazy interagují ve stylu návrhového vzoru *visitor*. Ten využívá mechanismů virtuálního volání

a přetěžování metod k určení, který kód se má provést, na základě běhových typů dvou objektů (tzv. *double dispatch*) – v našem případě herního příkazu a zpracovatele příkazu. Tento návrh byl zvolen jako jeden ze způsobů oddělení datové reprezentace a různých druhů funkcionality, které byly nad stejnými daty postupně přidávány. Grafická odezva pro modifikaci herního stavu je tak například dodána pomocí grafického zpracovatele `GraphicCommandProcessor`. Zpracovatel `CommandRecorder` vytváří a serializuje záznam průběhu hry. Třída `MapAnalytics` poskytuje AI systému pomocná data a statistiky o herním stavu průběžně přepočítávané na základě zpracovávaných příkazů.

6.1.6 Třída ConsoleClient a negrafický režim

Negrafický konzolový režim klienta poskytuje třída `ConsoleClient`. Ta využívá stejnou implementaci herní logiky a dat jako grafický klient, nepoužívá ale jeho uživatelsky orientované systémy. Konzolový režim je určen pro efektivní zpracování hry počítačových agentů. Při spuštění načte uživatelem dodaný seznam parametrů a úkolů, tj. herních relací se stanovenými agenty na stanovených herních mapách, které má klient odehrát. Zpracování herních relací probíhá (volitelně) vícevláknově za pomoci třídy `TaskProcessor`. Každá probíhající relace je reprezentována strukturou `GameSession` a má k dispozici svoji instanci třídy `NonGraphicCommandProcessor` pro modifikaci herního stavu na základě příkazů dodávaných AI systémem.

6.1.7 Rozhraní pro AI agenty

Hlavními cíli návrhu AI rozhraní byla logická a technologická separace hry a herních agentů. Za tímto účelem byla vytvořena mezivrstva v podobě tzv. poskytovatelů AI. Jde o třídy naplňující rozhraní čistě abstraktní třídy `AIProvider`, jejichž rolí je zajistit obousměrnou komunikaci mezi herním klientem a agenty. Poskytovatel hernímu klientu inzeruje dostupné agenty a na vyžádání vrací příkazy vypočítané zvoleným agentem. V opačném směru poskytovatel agentům zprostředkovává data o herním stavu získaná od klienta. Technologie agentů přitom závisí na konkrétním poskytovateli. V rámci práce je používán jen poskytovatel pro nativní C++ agenty `NativeAIProvider`. Toto řešení bylo zvoleno ve snaze o maximalizaci běhové efektivity agentů, užitečné zejména v kontextu strojového učení. Stejné rozhraní je ale možné využít pro přidání podpory pro agenty na bázi vlastního skriptovacího jazyka nebo pro vytvoření poskytovatele, který pomocí *embedded* interpretu provozuje agenty napsané např. v jazyce Python nebo Lua. Takové řešení by bylo praktičtější pro ruční tvorbu herních strategií.

Druhou stranu rozhraní představuje čistě abstraktní třída `GameStateProvider`. Při každé výzvě k poskytnutí herních příkazů dostává poskytovatel AI referenci na objekt odvozený od této třídy (typicky strukturu `GameSession`), který zprostředkovává kontrolovaný přístup k aktuálnímu hernímu stavu, mapě, sadě pravidel a definicím herních typů.

6.2 AI agenti

V této sekci se věnujeme implementačním aspektům herních agentů popsaných v kapitole 3. Všichni dodaní agenti jsou součástí statické knihovny `NativeAI` napsané v jazyce C++. Vztahy hlavních tříd souvisejících s herními agenty znázorňuje obrázek 6.5.

6.2.1 Registrace a použití agentů

Nativní herní agenti jsou implementováni třídami odvozenými od abstraktní třídy `NativeAIAgent` a jsou spravováni instancí třídy poskytovatele `NativeAIProvider`. Přidání nového typu agenta spočívá v zavolání generické metody `RegisterAgentType` v konstruktoru poskytovatele. Parametry této metody se skládají z typové informace o třídě agenta, struktury `AgentDescriptor`, která obsahuje identifikátor agenta a deklaraci jeho vlastností, a z tovární metody schopné vytvářet instance agenta. Poskytovatel následně klientovi předává informace o dostupných agentech a vytváří instance požadovaných agentů. V případě parametrizovatelných agentů poskytovatel udržuje zvlášť varianty agenta lišící se sadou parametrů načtených z externích konfiguračních souborů.

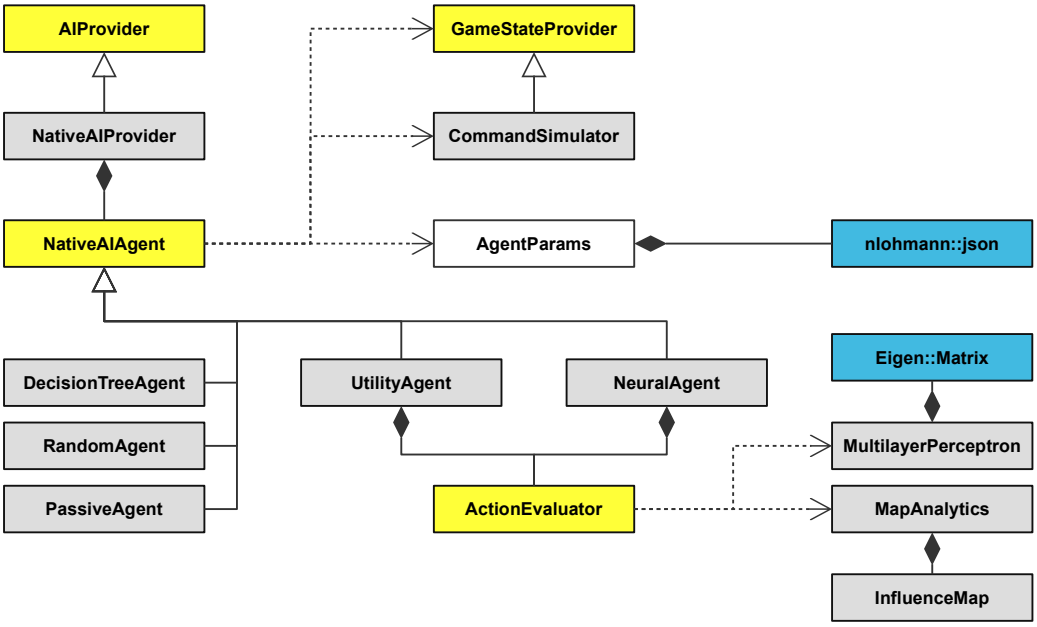
Během hry klient používá AI systém tak, že ve chvíli, kdy je na tahu počítačový hráč, vyžádá si od poskytovatele herní příkazy vypočítané instancí agenta asociovanou s aktuálním hráčem. Tato žádost je uskutečněna voláním metody `GetCommands`, jehož součástí je předání reference na objekt naplňující rozhraní `GameStateProvider`, který agentovi zprostředkovává kontrolovaný přístup k hernímu stavu.

Při implementaci agentů je třeba brát v potaz, že poskytovatel používá právě jednu instanci pro každou variantu agenta. To je důležité v kontextu konzolového klienta, který může hrát ve více vláknech několik herních relací najednou. Pokud si agent chce udržovat nějaký vnitřní stav, je jeho zodpovědností, aby zabránil datovým kolizím. Všichni implementovaní agenti jsou bezstavoví a pomocné datové struktury, které při výpočtu používají, jsou udržovány pouze lokálně pro konkrétní volání metody `GetCommands`.

6.2.2 Nástroje pro tvorbu agentů

Algoritmus, podle kterého agent vygeneruje příkazy pro řízeného hráče, je zcela na něm. Součástí implementace je však několik pomocných tříd, které mohou být užitečné pro širokou škálu agentů.

Nejdůležitější z těchto nástrojů je třída `CommandSimulator`, která kombinuje funkcionalitu rozhraní `CommandProcessor` a `GameStateProvider`. Ať už agent provádí plánování v prostoru stavů, nebo po jedné heuristicky vybírá akce, je nutné, aby výstupem jeho práce byla platná posloupnost akcí. Akce na sobě přitom často nejsou nezávislé. `CommandSimulator` proto poskytuje dopřednou simulaci změn herního stavu podle stejných pravidel, jaká interně používá klient. Simulátor používají všichni netriviální implementovaní agenti.



Obrázek 6.5: Schéma tříd nativní implementace herní AI

Dále je k dispozici třída `MapAnalytics`. Její hlavní funkcí je tvorba a aktualizace map vlivu, popisujících prostorové zastoupení různých typů jednotek a budov jednotlivých hráčů na mapě. Poskytuje také některé globální číselné statistiky herního stavu, jako jsou počty hráčových polí a součty síly jednotlivých typů jednotek. Tento nástroj používá Užitkový a Neuronový agent. První z nich implementuje část svého algoritmu v rozšířené variantě `UtilityMapAnalytics`.

Implementaci map vlivu poskytují třídy `UnitInfluenceMap` a `BuildingInfluenceMap`, specializované pro mapování vlivu jednotek a obsazených budov (speciálních polí). Ty zajišťují adekvátní metriku při šíření vlivu daného typu objektů (pro jednotky se vzdálenost počítá v počtu kol) a poskytují metody pro jeho typické modifikace. Kvůli rychlejšímu provádění změn tyto třídy vedle hlavní souhrnné mapy udržují také dílčí mapy pro jednotlivé objekty, jejichž přičítáním a odčítáním inkrementálně upravují hlavní mapu.

6.2.3 Implementace neuronových sítí

Neuronový agent používá při ohodnocování herních akcí neuronové sítě typu MLP. Výpočet této sítě lze efektivně realizovat pomocí maticového násobení. Každá vrstva sítě (kromě první) je reprezentována maticí vah, v níž každý řádek nese váhy vstupních hran pro jeden neuron vrstvy. Při zpracování vrstvy je pak vektor hodnot vynásoben maticí vah, na prvky vzniklého vektoru je aplikována zvolená aktivační funkce a výsledek je předán následující vrstvě.

Funkcionalitu sítě zapouzdřuje třída `MultilayerPerceptron`. Ta při konstrukci vytvoří matice s rozměry odpovídajícími zadané posloupnosti velikostí vrstev (zvětšených o jedna kvůli implicitním prahům) a načte do nich váhy. Pro

reprezentaci matic a provádění výpočtů používáme vysoce optimalizovanou knihovnu Eigen.

6.3 Evoluční trenér

Dalším úkolem bylo vytvořit nástroj pro optimalizaci parametrů herních agentů pomocí evolučních algoritmů. Jednou možností bylo realizovat tuto funkcionalitu přímo v herním klientovi. Za účelem lepšího oddělení rolí jsme se rozhodli vytvořit pro ni samostatný program. Výsledkem je konzolová aplikace napsaná v jazyce C# verze 6 na platformě .NET Core 2.0. Jazyk C# byl vybrán kvůli vysoké produktivitě při tvorbě datově orientovaných aplikací, platforma .NET Core kvůli systémové nezávislosti.

Při návrhu řešení bylo třeba zvolit způsob komunikace mezi trenérem a herním klientem. Zvažovali jsme komunikaci pomocí souborů, pomocí socketů a komunikačního protokolu TCP/IP, skrze systémové vstupy a výstupy aplikací (*pipes*) nebo pomocí některé z paměťových metod meziprocesové komunikace. Rozhodli jsme se pro přístup na bázi výměny textových souborů, kvůli jeho přímočaré implementaci, dobré odladitelnosti (každý krok mechanismu je sám sobě záznamem) a snadnému potenciálnímu propojení s jinými technologiemi. Režie tohoto řešení, spočívající v opakované serializaci a deserializaci dat do souborů na pevném disku, se potvrdila jako zanedbatelná v kontextu hromadné simulace stovek nebo tisíců her během vyhodnocení fitness jedné evoluční generace.

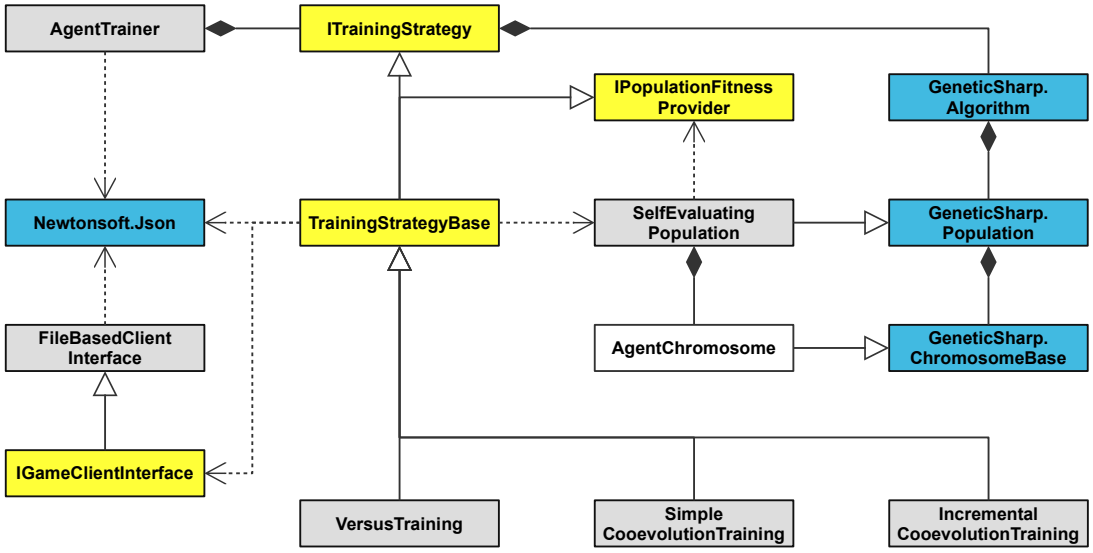
Všechny vstupní a výstupní soubory programu jsou ve formátu JSON. Pro práci s tímto formátem používáme knihovnu Json.NET. Samotné evoluční algoritmy byly implementovány na základě poskytnutém knihovnou GeneticSharp. Knihovnu bylo třeba rozšířit pro podporu externího vyhodnocování fitness, evolučních strategií a koevolučních algoritmů.

Jednotlivé složky implementace představíme v rámci popisu rolí hlavních tříd. Jejich vztahy znázorňuje obrázek 6.6.

6.3.1 Trénovací strategie

Centrálním objektem programu je tzv. trénovací strategie, která odpovídá za inicializaci a řízení běhu evolučního algoritmu. Všechny implementované strategie jsou třídy dědící z abstraktní třídy `TrainingStrategyBase`. Ta zapouzdřuje instanci objektu `GeneticAlgorithm` knihovny GeneticSharp, který realizuje samotnou smyčku evolučního algoritmu. Strategie načte uživatelskou konfiguraci, inicializuje instanci algoritmu a dodá mu odpovídající sadu genetických operátorů. Během práce algoritmu pak poskytuje ohodnocení populační fitness skrze rozhraní `IPopulationFitnessProvider` a zpracovává výstupní statistiky.

Navenek každá trénovací strategie naplňuje rozhraní `ITrainingStrategy`. Skrze něj poskytuje přístup k nakonfigurované instanci algoritmu a ke statistikám a nejlepším jedincům vzniklým při běhu algoritmu. Toto rozhraní používá



Obrázek 6.6: Schéma tříd evolučního trenéra

vstupní třída programu `AgentTrainer`, která po načtení konfigurace předává řízení instanci zvoleného typu trénovací strategie, a po doběhnutí algoritmu zapíše výsledky.

V rámci práce byly implementovány tři trénovací strategie, odpovídající optimalizačním cílům popsaným v sekci 4.2. Strategie `VersusTraining` optimalizuje agenty proti na vstupu zadané množině oponentů. Strategie `SimpleCoevolutionTraining` používá kompetitivní koevoluci v rámci jedné vyvíjené populace. Strategie `IncrementalCoevolutionTraining` kombinuje obě předchozí a optimalizuje trénované agenty proti skupině trénovacích oponentů, která je tvořena úspěšnými agenty z předchozích generací. U každé trénovací strategie lze zvolit, jestli má být jako základní evoluční algoritmus použit genetický algoritmus, nebo evoluční strategie.

6.3.2 Implementace evolučních algoritmů

Protože knihovna GeneticSharp poskytuje jen omezené možnosti modifikace hlavní smyčky algoritmu a pořadí aplikace operací, museli jsme, za účelem dosažení námi požadované podoby algoritmů, rozšířit či reimplementovat některé knihovní typy a operátory.

Reprezentaci evolučního jedince poskytuje třída `AgentChromosome`. Jejím základem je pole reálných čísel představujících parametry herního agenta. Jednotlivá čísla jsou mapována na parametry agenta pomocí šablony zadané ve vstupní konfiguraci trenéra (viz kód 5.3 a specifikace formátů v elektronické příloze). Šablona také určuje přípustné rozsahy hodnot pro jednotlivé parametry, které jedinec dodržuje při všech modifikacích v průběhu algoritmu. Jedinec si také nese informaci o herním skóre (počtu výher, remíz, atd.) obdrženém při aktuální evaluaci, která je používána místo skalární fitness pro porovnávání kvality jedinců.

Ohodnocení jedinců probíhá najednou pro celou populaci, reprezentovanou třídou `SelfEvaluationPopulation`. V genetickém algoritmu se tak děje po vytvoření nové generace, u evoluční strategie naopak na konci cyklu, před environmentální selekcí jedinců do nové generace. Hromadné ohodnocení je použito jednak kvůli vyšší efektivitě zpracování naplánovaných her v herním klientovi, jednak je nezbytné pro koevoluční strategie. Ohodnocení populace získává pomocí rozhraní trénovací strategie `IPopulationFitnessProvider`. Strategie, v závislosti na svém algoritmu, naplánuje určitou kombinaci her mezi agenty a skrze rozhraní `IGameClientInterface` si vyžádá od herního klienta jejich výsledky. Součástí práce je jediná implementace tohoto rozhraní, třída `FileBasedClientInterface`. Ta realizuje komunikaci s herním klientem pomocí výše zmíněné výměny textových souborů. Zadání pro klienta je serializované do souboru ve formátu JSON, který je předán jako parametr příkazové řádky nové instanci klienta. Po doběhnutí klientského procesu si trenér přečte výsledky ze souboru na dohodnutém místě a předá je zpět trénovací strategii ke zpracování.

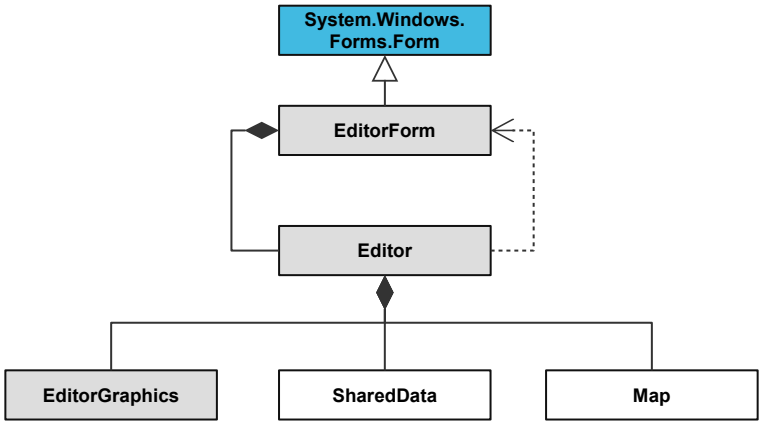
Vytvořeno bylo také několik nových genetických operátorů. Genetický algoritmus používá stochastickou variantu turnajové selekce, podporující porovnání jedinců podle vícerozměrného skóre. Implementovali jsme operátor pro gaussovskou mutaci, modifikující jedince o vektor náhodných čísel z parametrizovatelného normálního rozdělení. Na této mutaci stojí naše implementace evoluční strategie, která volitelně může v průběhu algoritmu parametry mutace adaptovat. Evoluční strategie dále oproti genetickému algoritmu používá operátory pro neváženou náhodnou selekci rodičů a naopak deterministickou selekci potomků do další generace.

6.4 Editor map

Editor map je grafická aplikace vytvořená pro usnadnění tvorby herních map použitých v rámci herního klienta. Při návrhu aplikace nám šlo primárně o co nejrychlejší vytvoření praktické pomůcky řešící specializovanou úlohu. Pro tvorbu editoru jsme proto zvolili jazyk C# a GUI knihovnu Windows Forms, která je součástí platformy .NET Framework. Serializace a deserializace dat používá knihovnu Json.NET.

Hlavní výhodou Windows Forms je právě rychlost návrhu a implementace funkcionality pro jednoduché grafické aplikace. Nevýhodami jsou vykreslování pomocí staršího neakcelerovaného rozhraní GDI+ a architektura knihovny postavená na principu „přímé" obsluhy událostí jednotlivých ovládacích prvků (místo např. *data binding* mechanismu), která potenciálně vede k obtížnější udržitelnosti rozsáhlých programů. Vzhledem k omezeným požadavkům na výkon a rozšiřitelnost aplikace pro nás tyto faktory nebyly důležité.

Funkcionalita aplikace je rozdělena mezi tři třídy. Třída `EditorForm` poskytuje obsluhu událostí knihovních ovládacích prvků. Třída `EditorGraphics` zajišťuje vykreslování herní mapy a grafického menu pro výběr přidávaných herních polí



Obrázek 6.7: Schéma tříd editoru map

a objektů. Pro vyšší efektivitu při překreslování obrazovky rozhraním GDI+, používá třída vlastní bitmapové buffery. Třída `Editor` spravuje logický stav editoru, data rozpracované mapy a načtená externí data shromážděná v instanci třídy `SharedData` (definice herních objektů, textury).

Závěr

V naší práci jsme se věnovali tvorbě umělé inteligence pro tahovou strategickou hru. Na základě analýzy vlastností hry a rešerše technik používaných v herním průmyslu a vědecké literatuře jsme vytvořili tři varianty herních agentů, včetně agenta na bázi neuronových sítí. Parametry agentů jsme optimalizovali v několika scénářích za pomoci evolučních algoritmů. Nejdůležitějším dosaženým výsledkem, byl úspěch Neuronového agenta trénovaného pomocí kombinace fixních oponentů a inkrementální koevoluce.

Tyto experimenty jsme mohli provést díky implementaci efektivního simulátoru hry. Ten umožňuje hromadné plánování zápasů a rychlou vícevláknovou simulaci hry bez grafické reprezentace. Dále byl vytvořen grafický klient hry použitelný pro hru lidských i počítačových hráčů a pro přehrávání záznamů her odehraných v konzolovém simulátoru. Tyto nástroje, spolu s editorem herních map a aplikací pro trénování agentů pomocí evolučních algoritmů, tvoří dohromady platformu použitelnou pro další potenciální výzkum.

Možnosti pokračování

Na výsledky práce lze navázat ve více směrech:

1. *Experimenty s evolučními algoritmy* – Během práce se nám z časových důvodů nepodařilo provést tolik pokusů s evoluční optimalizací, kolik jsme zamýšleli. Pro získání statisticky spolehlivějších dat by bylo potřeba všechny provedené pokusy vícekrát zopakovat. Dále existuje velký prostor pro zlepšení, a to na úrovni hledání vhodnějších parametrů stávajících algoritmů, implementace nových operátorů i celých algoritmů. Zejména bychom se pak chtěli věnovat konceptu koevoluce.
2. *Vylepšení stávajících agentů* – V rámci práce jsme implementovali agenty podporující zjednodušenou podmnožinu herních pravidel. Jednou z výzev je proto dosažení podpory plné varianty hry.
3. *Vyzkoušení dalších AI technik* – Od začátku práce jsme se omezili na reaktivní AI. Další výzvou (související s předchozím bodem) je proto implementace agenta používajícího nějakou formu plánování. Perspektivní v tomto směru jsou např. techniky na principu MCTS a online evoluce.
4. *Rozvoj softwarové platformy* – Vytvořený software by mohl být rozšířen pro podporu výše uvedených technik. Rozhraní pro tvorbu agentů bychom mohli rozšířit o podporu dalších, zejména interpretovaných jazyků, které jsou užitečné pro interaktivní vývoj ručních AI metod. V opačném směru stále existuje velký potenciál pro zrychlení simulátoru hry, jehož efektivita má zásadní vliv na schopnost provádět rozsáhlé experimenty se strojovým učením.

Seznam použité literatury

Baumgarten, R., Colton, S. a Morris, M. (2008). Combining AI Methods for Learning Bots in a Real-Time Strategy Game. *International Journal of Computer Games Technology*, **2009**. doi: 10.1155/2009/129075.

Bergsma, M. a Spronck, P. (2008). Adaptive Spatial Reasoning for Turn-based Strategy Games. In *AIIDE*.

Beyer, H.-G. a Schwefel, H.-P. (2002). Evolution strategies. A comprehensive introduction. *Natural Computing*, **1**(1), 3–52. ISSN 1572-9796. doi: 10.1023/A:1015059928466. Dostupné z: `https://doi.org/10.1023/A:1015059928466`.

Bourg, D. M. a Seemann, G. (2004). *AI for Game Developers*. O'Reilly Media.

Branavan, S., Silver, D. a Barzilay, R. (2011). Non-Linear Monte-Carlo Search in Civilization II. In *IJCAI International Joint Conference on Artificial Intelligence*.

Bryant, B. D. a Miikkulainen, R. (2003). Neuroevolution for adaptive teams. In *Proceedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, pages 2194–2201. IEEE.

de Jong, E. D. (2004). The Incremental Pareto-Coevolution Archive. In *Genetic and Evolutionary Computation – GECCO 2004*, pages 525–536. Springer Berlin Heidelberg.

Eiben, A. E. a Smith, J. E. (2015). *Introduction to Evolutionary Computing*. Springer Publishing Company, Incorporated, 2nd edition. ISBN 3662448734, 9783662448731.

Hagelbäck, J. a Johansson, S. J. (2009). A Multiagent Potential Field-Based Bot for Real-Time Strategy Games. *International Journal of Computer Games Technology*, **2009**. doi: 10.1155/2009/910819.

Hsu, F.-h., Campbell, M. S. a Hoane, Jr., A. J. (1995). Deep blue system overview. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, pages 240–244. ACM. ISBN 0-89791-728-6.

Iuhasz, G., Negru, V. a Zaharie, D. (2014). Neuroevolution Based Multi-Agent System with Ontology Based Template Creation for Micromanagement in Real-Time Strategy Games. *Information technology and control*, **43**, 98–109.

Justesen, N., Mahlmann, T., Risi, S. a Togelius, J. (2017). Playing multi-action adversarial games: Online evolutionary planning versus tree search. *IEEE Transactions on Computational Intelligence and AI in Games*.

Livingstone, D. (2005). Coevolution in hierarchical ai for strategy games. In *CIG*.

Mark, D. (2009). *Behavioral Mathematics for Game AI*. COURSE TECHNOLOGY.

Millington, I. (2019). *AI for Games*. CRC Press, third edition.

Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA. ISBN 0262631857.

Oakes, B. (2013). Practical and Theoretical Issues of Evolving Behaviour Trees for a Turn-based Game. Master's thesis, McGill University.

Ponsen, M. J. V., Lee-urban, S., Muñoz-avila, H., Aha, D. W. a Molineaux, M. (2005). Stratagus: An open-source game engine for research in real-time strategy games. Technical report, Naval Research Laboratory.

Rabin, S. (2002). *AI Game Programming Wisdom*. Charles River Media.

Rabin, S., editor (2013). *Game AI Pro*. Taylor & Francis Ltd.

Reisinger, J., Bahceci, E., Karpov, I. a Miikkulainen, R. (2007). Coevolving Strategies for General Game Playing. In *2007 IEEE Symposium on Computational Intelligence and Games*, pages 320–327. doi: 10.1109/CIG.2007.368115.

Risi, S. a Togelius, J. (2014). Neuroevolution in games: State of the art and open challenges. *CoRR*.

Rosin, C. D. a Belew, R. K. (1997). *Coevolutionary Search Among Adversaries*. PhD thesis, University of California, San Diego, La Jolla, CA, USA. UMI Order No. GAX97-32685.

Salimans, T., Ho, J., Chen, X., Sidor, S. a Sutskever, I. (2017). Evolution Strategies as a Scalable Alternative to Reinforcement Learning.

Schaeffer, J. (2001). A gamut of games. *AI Magazine*, **22**(3), 29–46.

Shannon, C. E. (1950). Programming a computer playing chess. *Philosophical Magazine*, **Ser. 7, 41**(312).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach,

M., Kavukcuoglu, K., Graepel, T. a Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, **529**, 484–503.

Silver, D., Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D. a Apps, C. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, **575**(7782), 350–354. ISSN 1476-4687. Dostupné z: `https://doi.org/10.1038/s41586-019-1724-z`.

Stanescu, M., Barriga, N. A., Hess, A. a Buro, M. (2016). Evaluating real-time strategy game states using convolutional neural networks. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–7. doi: 10.1109/CIG.2016.7860439.

Tadelis, S. (2013). *Game Theory: An Introduction*. Princeton University Press.

Tozour, P. (2001). Influence Mapping. In DeLoura, M., editor, *Game Programming Gems 2*, pages 287–297. Charles River Media.

Turing, A. (1953). Digital computers applied to games. In Bowden, B. V., editor, *Faster Than Thought*, pages 286–310. Pitman Publishing.

Watson, R. A. a Pollack, J. B. (2001). Coevolutionary Dynamics in a Minimal Substrate. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, pages 702–709, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1-55860-774-9. Dostupné z: `http://dl.acm.org/citation.cfm?id=2955239.2955343`.

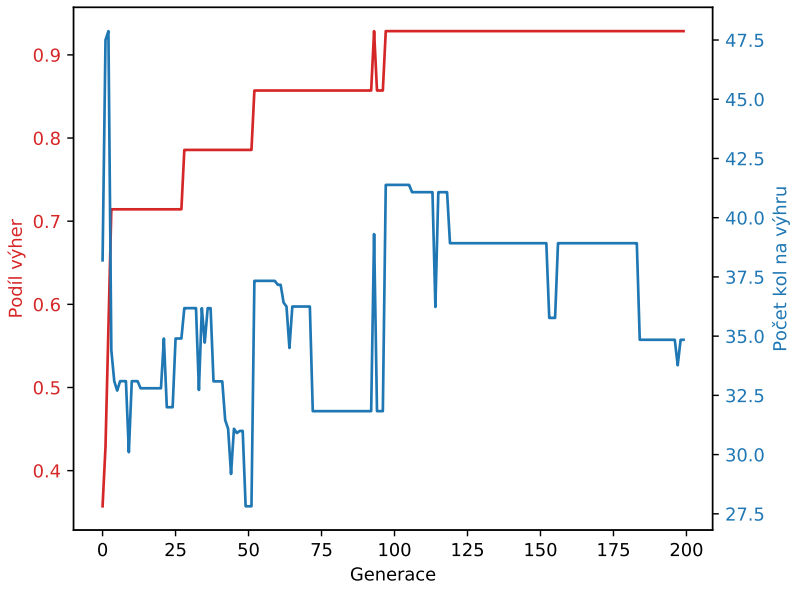
Yannakakis, G. N. a Togelius, J. (2018). *Artificial Intelligence and Games*. Springer.

Zhen, J. S. a Watson, I. (2013). Neuroevolution for Micromanagement in the Real-Time Strategy Game Starcraft: Brood War. In Cranefield, S. a Nayak, A., editors, *AI 2013: Advances in Artificial Intelligence*, pages 259–270. Springer International Publishing.

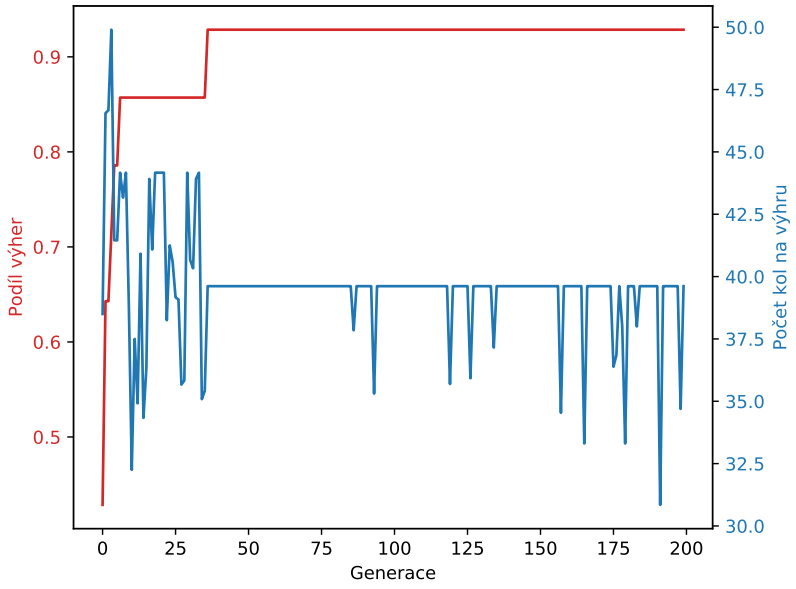
Seznam obrázků

Seznam tabulek

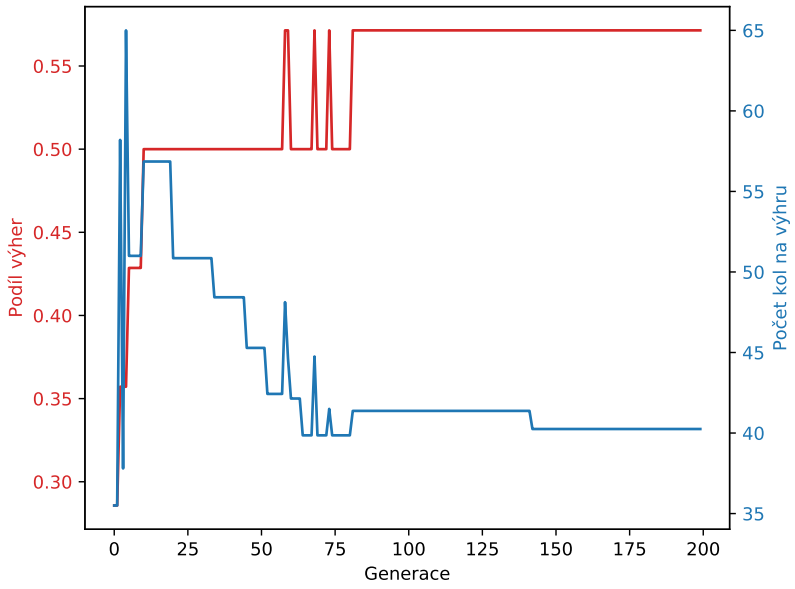
A. Grafy průběhů evolučních algoritmů



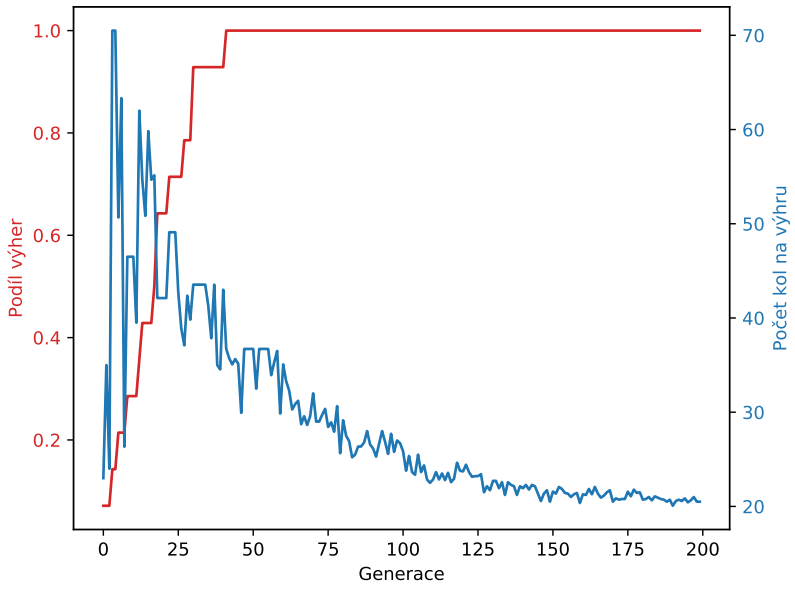
Obrázek A.1: Průběh algoritmu s konfigurací U1-GA



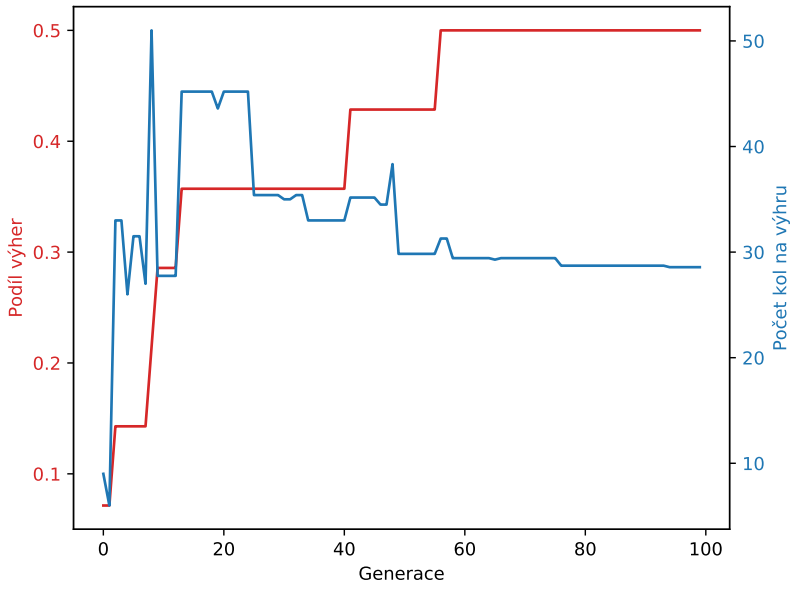
Obrázek A.2: Průběh algoritmu s konfigurací U1-ES



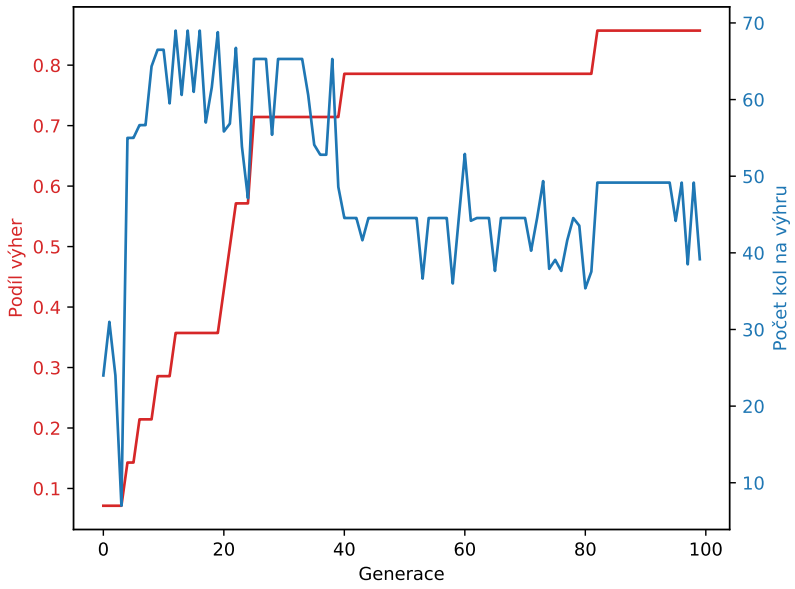
Obrázek A.3: Průběh algoritmu s konfigurací N1-GA



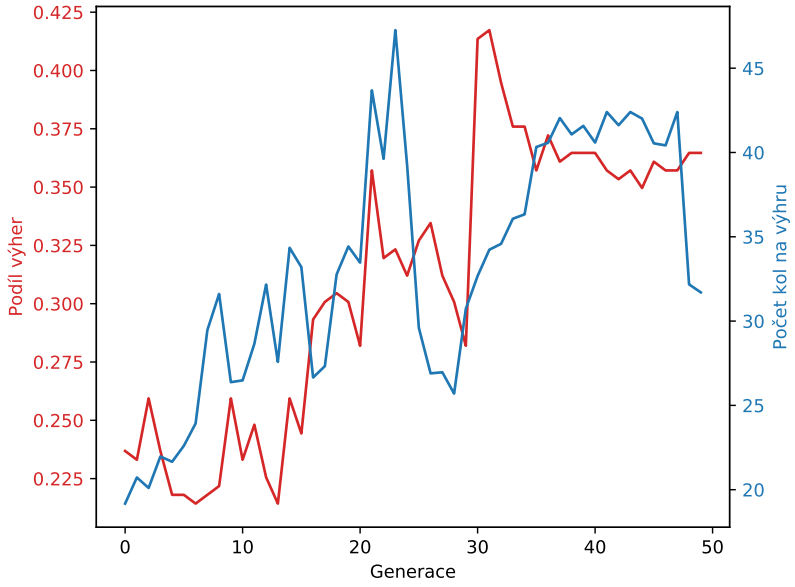
Obrázek A.4: Průběh algoritmu s konfigurací N1-ES



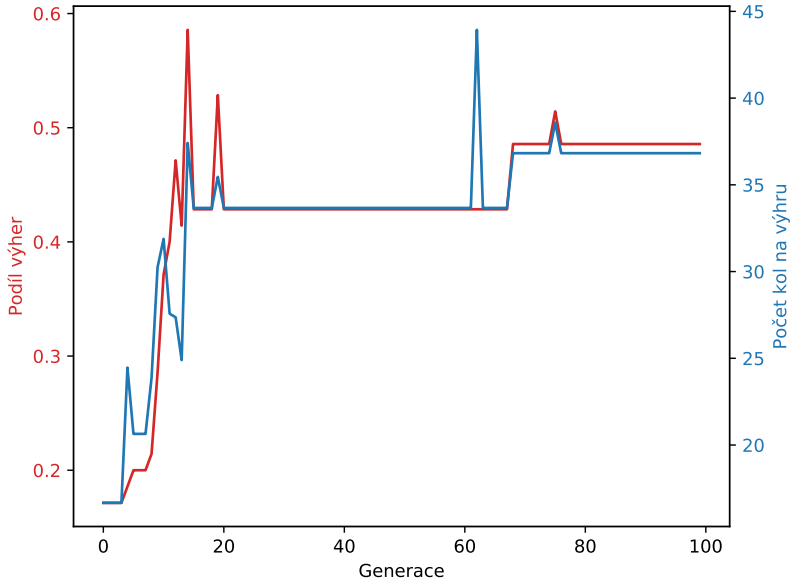
Obrázek A.5: Průběh algoritmu s konfigurací U2-GA



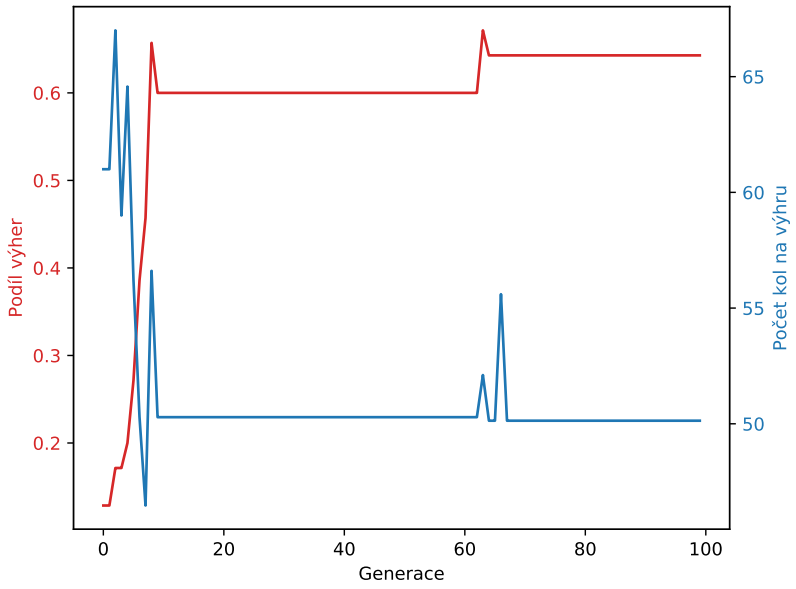
Obrázek A.6: Průběh algoritmu s konfigurací N2-ES



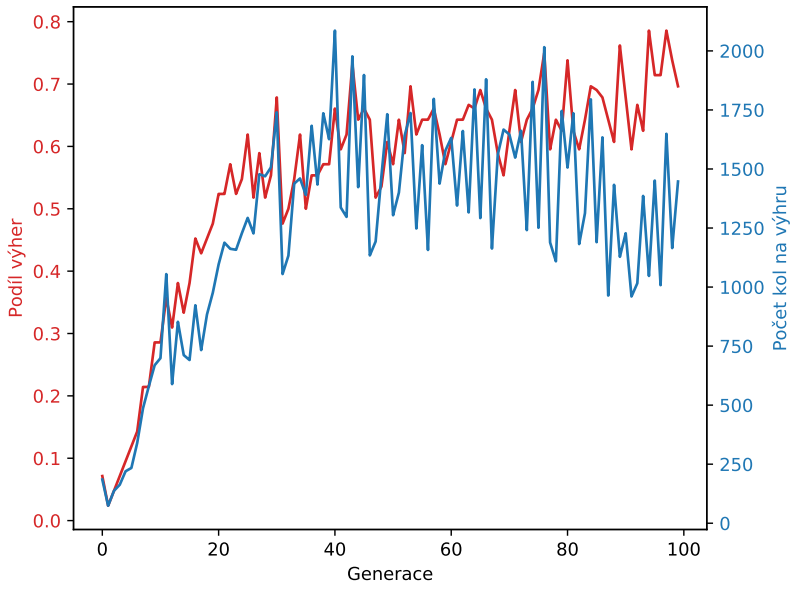
Obrázek A.7: Průběh algoritmu s konfigurací U3-GA



Obrázek A.8: Průběh algoritmu s konfigurací U4-ES



Obrázek A.9: Průběh algoritmu s konfigurací N4-ES



Obrázek A.10: Průběh algoritmu s konfigurací N5-ES

B. Obsah elektronické přílohy

Elektronická příloha je rozdělena do několika hlavních adresářů. Těmi jsou:

- `Binaries` – Obsahuje přeložené spustitelné soubory a knihovny.
- `Data` – Obsahuje načítaná herní data (definice typů, grafické soubory, herní mapy, definice pravidel, definice AI agentů) a výstupní soubory (uložené hry, záznamy her).
- `Docs` – Obsahuje dokumentaci práce. Soubor `formaty.pdf` představuje specifikaci formátů datových a konfiguračních souborů použitých v práci. Soubor `pravidla.pdf` obsahuje podrobná pravidla hry. Adresář `Doxygen` obsahuje vygenerovanou programátorskou dokumentaci.
- `Experiments` – Obsahuje soubory spojené s evolučními experimenty.
- `Source` – Obsahuje zdrojové kódy veškerého vytvořeného softwaru.