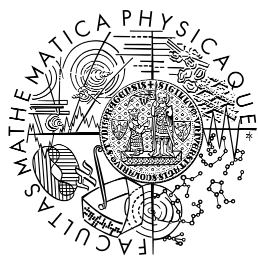


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

David Nguyen

Karetní hra Bang! online a umělá inteligence

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Tomáš Holan, Ph.D.
Studijní program: Informatika

Praha 2025

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

Rád bych poděkoval svému vedoucímu bakalářské práce RNDr. Tomáši Holanovi. Ph.D. za jeho cenné rady, užitečnou zpětnou vazbu, ochotu, motivaci, podporu a rychlou komunikaci. Také velice děkuji své rodině a své přítelkyni Sofii za jejich neustálou podporu. V neposlední řadě děkuji svým přátelům, zejména Davidovi a Tomášovi za podporu po celou dobu studia.

Název práce: Karetní hra Bang! online a umělá inteligence

Autor: David Nguyen

Department: Katedra softwaru a výuky informatiky

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

Abstrakt: Bang! je westernová karetní hra, kde hráči chtějí odhalit skryté role oponentů. Naším cílem bylo implementovat herní prostředí umožňující hru více hráčů po síti a následný výzkum a vývoj umělé inteligence. Hru jsme implementovali pomocí herního engine Unity. Poté jsme využili neuroevoluci a modely strojového učení k ohodnocování herních akcí v daném herním stavu. Trénovací data modelů jsme k zajištění různorodosti sbírali z náhodných her. Z testovaných modelů měly ensemble metody nejvyšší R^2 a nejnižší MSE. Dále jsme neuroevolučním algoritmem NEAT trénovali univerzální a specializované neuronové sítě. Univerzální sítě byly určeny pro všechny role, kdežto ty specializované pouze pro jednotlivé role. Výkonnost umělých inteligencí jsme měřili proti náhodným hráčům. Specializované sítě měly nejlepší výsledky, a proto jsme s nimi odehráli několik her. I přes potíže s odhadováním rolí a s některými jednoduchými herními situacemi vykazovaly, že rozumí hlavním konceptům hry, a byly schopny hrát v týmu i individuálně.

Klíčová slova: karetní hra, umělá inteligence, bang!

Title: Bang! the card game online and artificial intelligence

Author: David Nguyen

Department: Department of Software and Computer Science Education

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

Abstract: Bang! is a western-themed card game, where players aim to uncover hidden roles of opponents. Our goal was to implement a game environment enabling online multiplayer and to research and develop artificial intelligence (AI) for this game. We developed the game using the Unity game engine with a focus on extensibility. Then, we used neuroevolution and machine learning models to score game actions in a given game state. We collected the training data for the models from random games to ensure data diversity. Among the tested models, ensemble methods achieved the highest R^2 and the lowest MSE. Furthermore, we used the neuroevolutionary algorithm NEAT to train universal and specialized neural networks. Universal networks were for use with all roles, while specialized networks focused on individual roles. The performance of AIs was assessed against random players. Specialized NEAT-trained networks demonstrated the best results, and so we tried playing against them. Despite having difficulties with estimating roles and some simple game situations, they showed an understanding of the game's main concepts and were capable of playing both in teams and individually.

Keywords: card game, artificial intelligence, bang!

Obsah

1 Úvod **8**
1.1 Cíl práce . . . 8
1.2 Struktura práce . . . 8

2 Související práce **9**
2.1 Relevantní implementace hry . . . 9
2.1.1 BoardGameArena BANG! . . . 9
2.1.2 KBang . . . 9
2.1.3 Implementace M. Danilákové . . . 9
2.2 Existující řešení umělé inteligence . . . 10
2.2.1 If-else-then v kombinaci s pravděpodobnostním odhadem rolí 10
2.2.2 Monte Carlo tree search . . . 10
2.2.3 Evoluční algoritmus . . . 11
2.2.4 Kombinace Monte Carlo tree search a evolučního algoritmu 11

3 Základní pravidla a mechanismy hry **12**
3.1 Role a jejich cíle . . . 12
3.2 Postavy a jejich schopnosti . . . 12
3.3 Začátek hry . . . 12
3.4 Průběh tahu hráče . . . 13
3.4.1 První fáze . . . 13
3.4.2 Druhá fáze . . . 13
3.4.3 Třetí fáze . . . 13
3.5 Tresty a odměny za vyřazení hráče . . . 13
3.6 Vítězné podmínky . . . 13
3.7 Herní karty . . . 13
3.7.1 Hnědé karty . . . 14
3.7.2 Modré karty . . . 14
3.8 Vzdálenost . . . 14
3.9 Dostřel . . . 14
3.10 Sejmutí karty (odlišné od Dobírání/Líznutí) . . . 14
3.11 Vyřazení hráče . . . 14

4 Herní prostředí **15**
4.1 Analýza . . . 15
4.1.1 Výběr herního engine . . . 15
4.1.2 Síťování . . . 15
4.1.3 Hlavní komponenty . . . 16
4.1.4 Autorská práva . . . 18
4.2 Implementace . . . 18
4.2.1 Návrh herního pole a ovládání hry . . . 18
4.2.2 Komunikace a zpracovávání zpráv . . . 20
4.2.3 Fáze tahu . . . 22
4.2.4 Herní logika . . . 23
4.2.5 Zanořené efekty . . . 26

4.2.6 Oddělení herní logiky a síťování 26

5 Umělá inteligence 28
5.1 Strojové učení 28
5.1.1 Základní pojmy 28
5.1.2 Hladový výběr nejlepší herní akce pomocí regrese 29
5.1.3 Návrh a sběr trénovacích dat modelu 29
5.1.4 Implementace 32
5.1.5 Měření výkonnosti simulátoru her 34
5.1.6 Testování estimátorů různých druhů 35
5.1.7 Rozhodovací stromy a ensemble metody 37
5.1.8 Testování různých rozhodovacích stromů a ensemble metod 37
5.1.9 Trénování na větších datech 38
5.2 Neuroevoluce 40
5.2.1 Základní pojmy 40
5.2.2 NEAT 41
5.2.3 Trénování 45
5.3 Porovnávání umělých inteligencí 47
5.3.1 Náhodní hráči vs. Náhodní hráči 47
5.3.2 Strojové učení vs. Náhodní hráči 49
5.3.3 Univerzální NEAT vs. Náhodní hráči 52
5.3.4 Specializovaný NEAT vs. Náhodní hráči 55
5.3.5 Specializovaný NEAT vs. Člověk 58

6 Uživatelská dokumentace 61
6.1 Spuštění hry 61
6.2 Hlavní menu 61
6.3 Seznam uložených serverů 62
6.4 Seznam lobby 63
6.5 Vytvoření lobby 64
6.6 Lobby 65
6.7 Výběr postavy 66
6.8 Herní pole 67
6.8.1 Ovládání hry 67

7 Vývojová dokumentace 69
7.1 Celkový přehled 69
7.1.1 Organizace projektu 69
7.1.2 Assemblies 69
7.1.3 Hlavní komponenty 70
7.1.4 Kompilace 71
7.2 Uživatelské rozhraní 71
7.3 Síťování 72
7.3.1 WatsonTcp 72
7.3.2 BangTcpClient a BangTcpServer 72
7.3.3 NetworkMessage 72
7.3.4 Stavy klientů 73
7.4 Herní logika 75
7.4.1 Třída Game 75

7.4.2 Karty 75
7.4.3 Postavy 75
7.4.4 Herní ovladače a efekty 76

8 Závěr **80**
8.1 Navázání na práci 80

Literatura **81**

Seznam obrázků **83**

Seznam tabulek **84**

A Přílohy **86**
A.1 Sestavené herní prostředí 86
A.2 Konzolový klient pro umělé inteligence 86
A.3 Implementace experimentů 86
A.4 Projekt trénování modelů strojového učení 86
A.5 Zdrojový kód herního prostředí 87
A.6 Zdrojový kód uživatelské dokumentace 87
A.7 Natrénované umělé inteligence 87

1 Úvod

`Bang!` je karetní hra ve westernovém stylu, v níž se odehrává přestřelka mezi šerifem a skupinou banditů. Ve hře dále figurují pomocníci šerifa, kteří se ho snaží ochránit. V poslední řadě je zde odpadlík, jenž se chce stát novým šerifem.

Kromě výběru vhodného tahu je jedním z hlavních problémů hry i to, že role všech nevyřazených hráčů, kromě šerifa, jsou skryté. Hráči tak mohou k dosažení svých cílů využít nejen schopnosti postav či různé efekty karet, ale i lsti.

Výzkumem umělé inteligence pro hru `Bang!` se již zabývalo několik prací, které blíže popíšeme v další kapitole. My se pokusíme v tomto kontextu prozkoumat jiný přístup využívající metody strojového učení a následně i neuroevoluci.

1.1 Cíl práce

Cílem práce je implementace herního prostředí pro karetní hru `Bang!` umožňující hru více hráčů po síti. **Dalším cílem** je výzkum a vývoj umělé inteligence pro tuto hru.

1.2 Struktura práce

V práci nejdříve shrneme relevantní implementace hry spolu s existujícími řešeními umělé inteligence. Následně poskytneme rychlý úvod do základních pravidel a mechanismů hry potřebných k pochopení práce, načež se začneme věnovat hernímu prostředí. Shrneme hlavní rozhodnutí, která jsme museli učinit před začátkem implementace, popíšeme její hlavní problémy a znázorníme, jak jsme je řešili. Poté se budeme zabývat umělou inteligencí, kde jsme se pokusili prozkoumat využití metod strojového učení a neuroevoluce v rámci hry `Bang!`. Popíšeme, jak jsme naši umělou inteligenci trénovali a experimentálně otestujeme, jak dobře hraje. Na konec také poskytneme uživatelskou a vývojovou dokumentaci k našemu hernímu prostředí.

2 Související práce

Hledali jsme práce, které umožňovaly hru po síti, nebo se zabývaly umělou inteligencí hry. Dále nás také zajímalo, zda poskytují speciální prostředí pro vývoj umělé inteligence.

Celkem jsme nalezli pět takových prací, přičemž jsme zjistili, že speciální prostředí pro vývoj umělé inteligence poskytovala pouze jedna z nich. Nyní shrneme, co všechny nalezené práce přinesly.

2.1 Relevantní implementace hry

V době psaní textu byla nejrozšířenější implementací hry její webová verze `BoardGameArena Bang!` [1]. Kromě ní také i čtyři z nalezených prací zahrnovaly své vlastní herní prostředí. Mezi ně patří práce J. Schovánka [2] a M. Trejtnara [3], jejichž implementace však nebyla dostupná, a tak jsme je nemohli dále zkoumat. Mj. práce M. Trejtnara [3] měla poskytovat hru pouze po lokální síti.

Zabývali jsme se tedy třemi implementacemi, a to webovou hrou `BoardGameArena BANG!` [1], projektem `KBang` od M. Čevory [4] a implementací od M. Danilákové [5].

2.1.1 BoardGameArena BANG!

`BoardGameArena BANG!` [1] byl vydán v roce 2022. V době psaní práce byl nejrozšířenější implementací hry umožňující hru více hráčů po síti. Implementace má formu webové aplikace, díky čemuž je široce dostupná. Jedním z důvodů jejího úspěchu je to, že podporuje soutěživost – je v ní zabudován bodovací systém hráčů a organizuje online turnaje. Dokonce penalizuje hráče, kteří se odpojí uprostřed hry, či neprovedou v daném čase herní tah. V době psaní práce byla aplikace stále udržována.

2.1.2 KBang

Projekt `KBang` M. Čevory [4] je pravděpodobně první úspěšná implementace herního prostředí pro hru `Bang!` umožňující hru více hráčů. `KBang` [4] byl velice oblíbený v komunitě hráčů i několik let po jejím zveřejnění, a to i přes pozastavení vývoje.

V. Kolář [6] tento projekt použil k výzkumu umělé inteligence.

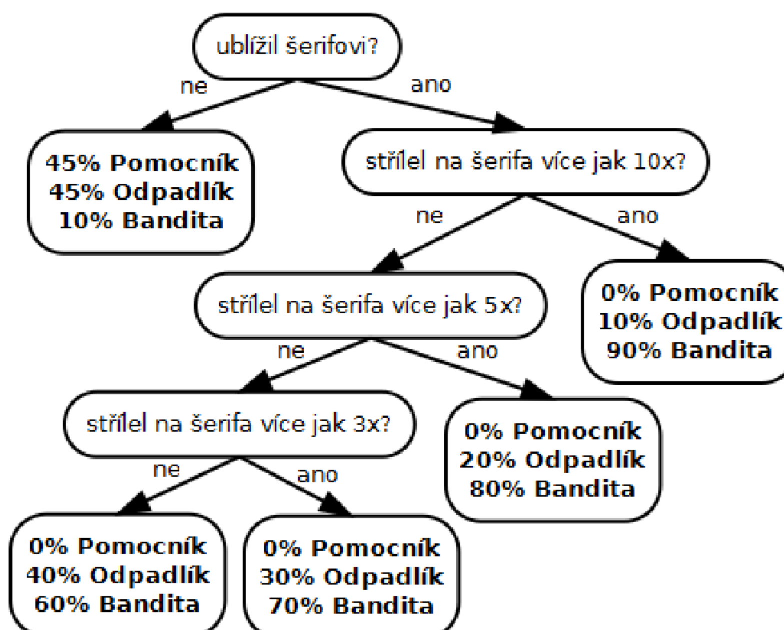
2.1.3 Implementace M. Danilákové

Implementace M. Danilákové [5] byla z nalezených jediná, která kromě herního prostředí poskytuje také prostředí speciálně určené pro měření výkonu umělých inteligencí.

2.2 Existující řešení umělé inteligence

Nalezli jsme několik různých řešení umělé inteligence pro hru `Bang!`. Práce M. Danilákové [5] byla jediná z nalezených, která k měření výkonu umělé inteligence nechala odehrát alespoň jednotky tisíců her, přičemž u ostatních prací proběhly řádově jednotky her. V následujících oddílech představíme jednotlivá existující řešení.

2.2.1 If-else-then v kombinaci s pravděpodobnostním odhadem rolí



Obrázek 2.1 Rozhodovací strom estimátoru rolí, V. Kolář [6]

`If-then-else` umělé inteligence jsou implementovány sériemi podmíněných příkazů, čímž se snaží napodobit myšlení lidského hráče.

V. Kolář [6] navrhl kombinaci `if-else-then` přístupu a pravděpodobnostního odhadování rolí hráčů. Distribuce pravděpodobností rolí daného hráče počítal pomocí rozhodovacích stromů. Jako nejúspěšnější se dle jeho práce ukázal rozhodovací strom, jehož Obrázek 2.1 citujeme.

M. Trejtnar [3] umělou inteligenci řešil podobným způsobem.

2.2.2 Monte Carlo tree search

Monte Carlo tree search (MCTS) je iterativní algoritmus, který prohledává stavový prostor a přitom zaznamenává, k jakým konečným stavům vedly akce provedené během prohledávání. Algoritmus následně vrací akci, která vedla k nejvýhodnějším konečným herním stavům. MCTS lze použít i v případech, kdy nemá čas k prohledání celého stavového prostoru, např. ve hře šachy, kde vyžadujeme časově omezenou reakci. V takových případech dokáže vrátit odhad nejvýhodnější akce na základě částečného prohledávání (Maciej Świechowski et al. [7]).

Tuto metodu aplikovala M. Daniláková [5], jejíž implementace umělé inteligence byla schopná plánovat dopředu. Avšak výsledkem její práce je, že prohledávání stavového prostoru nebylo pro `Bang!` příliš vhodné.

2.2.3 Evoluční algoritmus

Evoluční algoritmy simulují vývoj populace v generacích. Jedinec populace má nějakou DNA, která ovlivňuje jeho výkonnost (fitness). Ta je vyhodnocována fitness funkcí, přičemž chceme, aby křížení jedinců zlepšovalo jejich fitness.

Na začátku evolučních algoritmů nějakým způsobem (např. náhodně) inicializujeme DNA jedinců v populaci, a poté probíhá evoluce v iteracích. V každé iteraci nejprve spočítáme fitness všech jedinců v populaci. Poté na základě fitness vybíráme jedince, kteří poslouží jako rodiče pro další generaci. Jako poslední krok iterace vytvoříme novou populaci pomocí aplikování genetických operátorů na rodiče vybrané v předešlém kroku, načež začíná další iterace. Genetické operátory můžeme obecně dělit na mutace, při kterých vytváříme potomky náhodnou změnou části DNA rodiče, a křížení, kde vytváříme potomky kombinací DNA dvou rodičů.

Touto metodou se také zabývala M. Daniláková [5]. V jejím návrhu se DNA jedinců skládala z vah ohodnocovací funkce herního stavu. Jedinci simulovali všechny dostupné herní akce a následně prováděli takovou, která měla přinést nejlepší ohodnocení následujícího herního stavu.

Evoluční algoritmus se v práci autorky [5] ukázal jako nejefektivnější, přestože tato umělá inteligence nijak neplánovala dopředu a herní akce volila hladově.

2.2.4 Kombinace Monte Carlo tree search a evolučního algoritmu

Touto metodou se také zabývala M. Daniláková [5]. Jádrem metody je evoluční algoritmus, kde se jedinci rozhodovali na základě MCTS. Avšak při prohledávání se posloupnosti tahů hodnotily dle jejich vlivu na počty životů ostatních hráčů, nikoliv podle toho, zda vedly k výhře či prohře. Tím pádem se zkrátila výpočetní doba, jelikož nebylo potřeba simulovat hry až do konce. DNA jedinců tvořily váhy pro jednotku životů hráčů daných rolí.

Zde se ukázalo, že prohledávání stavového prostoru ani v kombinaci s evolučním algoritmem není pro tuto hru příliš vhodné.

3 Základní pravidla a mechanismy hry

V této kapitole vysvětlíme základní herní pravidla a mechanismy nutné k porozumění práce pro čtenáře, kteří ještě nikdy Bang! nehráli. Čerpat budeme z oficiálních pravidel hry [8], kde se mohou čtenáři seznámit s pravidly detailněji. Zkušení hráči mohou čtení pravidel přeskočit.

3.1 Role a jejich cíle

Role si hráči náhodně losují před začátkem každé hry. Ty pak určují jejich cíl ve hře. Počet hráčů určuje, z jaké kombinace rolí se losuje.

Šerif musí vyřadit všechny Bandity a Odpadlíka.

Bandité mají za cíl vyřadit Šerifa.

Pomocníci šerifa se snaží ochránit Šerifa za každou cenu. Vítězí tehdy, když zvítězí Šerif.

Odpadlík musí zůstat jako poslední naživu ve hře. K tomu musí nejprve vyřadit všechny hráče, kteří nejsou Šerifem. Poté musí vyřadit i Šerifa samotného.

Jediný hráč, jenž má roli odhalenou po celou dobu hry, je Šerif. Ostatní hráči musí ukázat svou roli ostatním pouze tehdy, když jsou vyřazeni.

3.2 Postavy a jejich schopnosti

Po losování rolí si hráči losují **postavy**. Každá z nich má speciální **schopnost** a také určuje maximální počet životů, které hráč může mít. Šerif má vždy ještě o jeden maximální život navíc. Schopnosti všech postav jsou stručně a jasně vysvětleny v pravidlech hry [8]. Obecně je lze rozdělit následovně:

- **Pasivní** – jejich efekt je platný po celou dobu hry. Např. `Paul Regret` má pasivní schopnost zvětšení vzdálenosti od ostatních hráčů o jedna.
- **Aktivované událostmi** – jejich efekt je automaticky aktivovaný při nějaké herní události. Např. schopnost `Suzy Lafayette` je aktivována pokaždé, když hráč nemá žádnou kartu v ruce.
- **Aktivované hráčem** – jejich efekt je aktivovaný na základě herní akce hráče, přičemž se musí zkontrolovat, zda je tato akce v souladu s pravidly hry. Např. schopnost postavy `Jourdonnais` lze použít, pokud je hráč cílem efektu `BANG!`.

3.3 Začátek hry

Každý hráč začíná s maximálním počtem životů dle své postavy. Každému hráči se náhodně rozdá jedna karta za každý život. První je na tahu Šerif a hráči se následně střídají ve směru hodinových ručiček.

3.4 Průběh tahu hráče

Tah hráče probíhá standardně ve třech fázích.

3.4.1 První fáze

V první fázi si hráč dobírá dvě vrchní karty z dobíracího balíčku, čímž se přesune do druhé fáze. Pokud karty v dobíracím balíčku dojdou, zamíchá se hromádka odhozených karet

3.4.2 Druhá fáze

Ve druhé fázi hráč smí zahrát libovolný počet karet. Jakmile už nechce hrát karty, ukončí své kolo. Jediná omezení jsou následující:

- Smí zahrát pouze jednu kartu `BANG!` během kola.
- Před každým hráčem smí být vyložena pouze jedna kopie od každé karty (vykládání karet před hráče vysvětlíme později).
- Smí mít ve hře pouze jednu zbraň.

3.4.3 Třetí fáze

Do třetí fáze se hráč dostane, pokud ve druhé fázi ukončí tah a má na ruce více karet než jeho počet životů. Ve třetí fázi hráč může pouze odhazovat přebytečné karty z ruky. Poté jeho tah končí tehdy, když se počet karet na ruce rovná počtu jeho životů.

3.5 Tresty a odměny za vyřazení hráče

Pokud Šerif vyřadí svého Pomocníka, musí odhodit všechny karty, které má v ruce a vyložené před sebou.

Pokud jakýkoliv hráč vyřadí Banditu, lízne si tři karty.

3.6 Vítězné podmínky

Hra končí vyřazením Šerifa. Pokud je poslední naživu Odpadlík, tak vítězí on. Jinak vítězí Bandité.

Další možností je, že jsou všichni Bandité a Odpadlík vyřazeni. Poté vítězí Šerif a jeho pomocníci.

3.7 Herní karty

V základním balíčku hry dělíme herní karty na dva druhy, které rozlišujeme dle barvy jejich okrajů na **Hnědé karty** a **Modré karty**. Výčet všech karet s vysvětlením jejich účinku lze nalézt v oficiálních pravidlech hry [8].

3.7.1 Hnědé karty

Hnědé karty mají okamžitý účinek. Zahrají se tak, že je umístíme lícem nahoru na hromádku odhozených karet. Mezi typy účinků jsou např. udělení poškození vybranému hráči, udělení poškození všem ostatním hráčům, či sebrání karty vybranému hráči. Základní balíček hry obsahuje 12 typů Hnědých karet s různými účinky.

3.7.2 Modré karty

Modré karty mají dlouhodobý účinek. Zahrají se tak, že je umístíme před sebe. V základním balíčku hry je jedinou výjimkou karta Vězení, která se vykládá před vybraného oponenta. Mezi typy účinků je např. povolení zahrání libovolného počtu karet `BANG!` během kola. Základní balíček hry obsahuje 10 typů Modrých karet s různými ůčinky.

3.8 Vzdálenost

Představme si, že hráči sedí v kruhu. Pak **vzdálenost** mezi dvěma hráči spočteme jako minimální počet míst mezi nimi (a to po směru nebo proti směru hodinových ručiček) zvětšený o jedna. Pokud jsou na cestě mezi dvěma hráči nějací mrtví hráči, pak místa mrtvých hráčů do vzdálenosti nepočítáme. Vzdálenost od hráčů a ke hráčům lze modifikovat různými herními efekty.

3.9 Dostřel

Dostřel je maximální vzdálenost, na kterou daný hráč může použít kartu `BANG!`. Základní dostřel je jedna a lze ho zvýšit vyloženými zbraněmi.

3.10 Sejmutí karty (odlišné od Dobírání/Líznutí)

Pro dokončení některých efektů je potřeba **sejmout kartu** z dobíracího balíčku. **Sejmutí karty** pro daný efekt se provádí tak, že si lízneme kartu z dobíracího balíčku, podíváme se na její karetní barvu a hodnotu a odhodíme ji do odhazovací hromádky. Na základě karetní barvy a hodnoty sejmuté karty se rozhoduje, co daný efekt provede. Např. pokud má hráč před sebou vyloženou kartu `Dynamit`, pak na začátku jeho kola musí sejmout kartu. Pokud je sejmutá karta piková dvojka až devítka, pak `Dynamit` vybuchne a hráč si musí ubrat tři životy. V opačném případě předává `Dynamit` dalšímu hráči na řadě.

3.11 Vyřazení hráče

Pokud hráčův počet životů klesne na nulu nebo níže, má možnost obnovit si okamžitě životy na kladný počet. Jestli si životy neobnoví, je vyřazen ze hry.

4 Herní prostředí

V této kapitole popíšeme, jak jsme řešili problémy v rámci vývoje našeho herního prostředí, přičemž se zaměříme na hlavní koncepty bez implementačních detailů.

4.1 Analýza

V této podkapitole se budeme zabývat rozhodnutími, která jsme provedli před řešením implementace herního prostředí. Zejména se zaměříme na výběr nástrojů, které jsme následně použili při implementaci, a řešením autorských práv hry.

4.1.1 Výběr herního engine

Herní engine je framework, který nabízí obecné funkcionality užitečné pro vývoj různých druhů her. Mezi běžnými funkcionalitami je např. grafický rendering, fyzika, zajištění multiplatformnosti, detekce vstupů od hráče či skriptování. Značnou výhodou herních engine je, že velmi urychlují a usnadňují vývoj her. Proto jsme se rozhodli k vývoji herního prostředí použít právě herní engine.

Jelikož jsme neměli žádné předchozí zkušenosti s vývojem her, zohledňovali jsme při výběru herního engine mj. přívětivost k úplným začátečníkům. Mezi nejznámějšími byly v době psaní textu Unreal Engine a Unity [9]. Z těchto dvou jsme zvolili Unity, jehož komunita je velice aktivní, a tak k němu existuje spousta online materiálů zdarma. Další výhodou je, že v Unity se ke skriptování používá jazyk C#, jenž preferujeme před jazykem C++, kterým se skriptuje v Unreal Engine.

4.1.2 Síťování

V tomto oddílu shrneme, jak jsme se rozhodli pro multiplayer architekturu autoritativního serveru, proč vyžadujeme spojovanou a spolehlivou síťovou komunikaci a proč jsme zvolili knihovnu WatsonTcp[10] jako řešení komunikace.

Multiplayer architektura

Při výběru multiplayer architektury jsme se řídili charakteristikami hry `Bang!`. Role daného hráče a jeho karty na ruce jsou informace, které by měly být skryté všem ostatním. Také by mělo být zajištěno, aby hráči nemohli provádět herní akce proti pravidlům.

Proto jsme se rozhodli využít architekturu autoritativního serveru. V té je veškerá herní logika soustředěna na straně serveru a klientům naopak není svěřena žádná část herní logiky. Pokud chce klient provést herní akci, tak požádá server, aby byla provedena. Server poté zkontroluje, zda je požadovaná herní akce v souladu s pravidly, a následně odešle všem klientům aktualizovaný herní stav bez skrytých informací.

Hosting

Rozhodovali jsme se mezi tím, zda realizovat multiplayer pomocí centrálního herního serveru, a nebo nechat komunitu hráčů, aby libovolně provozovala své vlastní servery.

Centrální server má výhodu v pohodlí pro hráče. Ti by totiž nemuseli specifikovat, na jaký server se chtějí připojit – zařídili bychom, aby se vždy připojili na náš centrální server. Zároveň by se díky koncentraci všech hráčů na jednom serveru rychleji organizovaly hry, jelikož by bylo k dispozici více potenciálních oponentů. Takové řešení je však potenciálně finančně nákladné.

Rozhodli jsme se tedy, že umožníme hostování serveru komukoliv z herní komunity. Takto může vzniknout síť komunitních serverů, o které se nemusíme starat. Nevýhodou je, že host serveru má přístup ke skrytým herním informacím, které může zneužívat jako hráč.

Síťové požadavky

`Bang!` je hra, ve které záleží na pořadí herních akcí, a zároveň je potřeba zajistit, aby herní akce klientů vždy dorazily na server. Potřebujeme tedy spojovanou a spolehlivou komunikaci, což zajišťuje TCP. Oproti UDP může být pomalejší, avšak pro účely karetní hry je toto zanedbatelné.

Jelikož jsme se rozhodli, že umožníme hostování komunitních serverů, bylo by uživatelsky příjemné, kdyby si uživatelé mohli prohlížet nějaký seznam serverů, na kterém bude vidět, jaká je jejich aktuální obsazenost. Tím pádem potřebujeme možnost komunikovat s více servery najednou, abychom mohli žádat o jejich úroveň obsazenosti.

Výběr řešení síťové komunikace

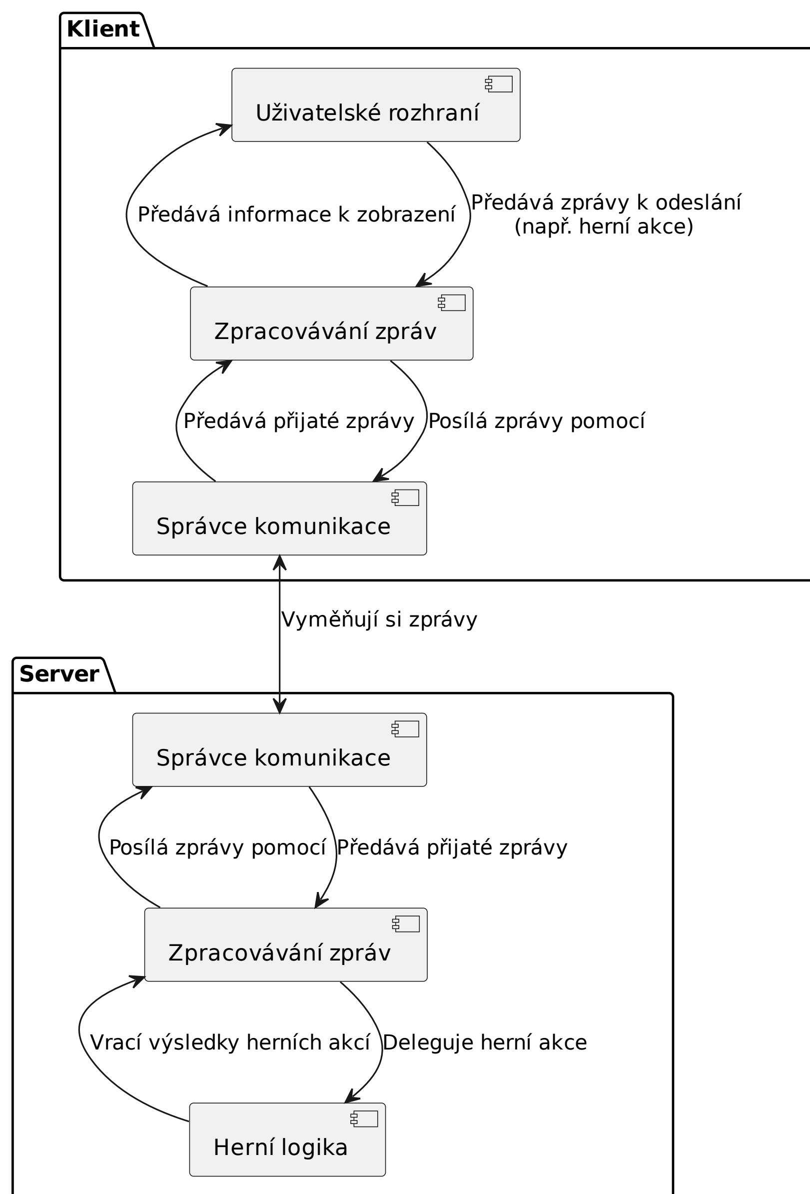
Při výběru vhodného řešení pro komunikaci jsme nejprve zkoumali high-level síťové knihovny. Zvažovali jsme Unity Networking for GameObjects [11], jelikož to je oficiální knihovna Unity, a Mirror Networking [12], protože tato knihovna byla velmi populární a praxí otestovaná na mnoha projektech. Obě knihovny jsou dostupné zdarma. Nakonec jsme je ale vyřadili z výběru, jelikož povolovaly připojování k maximálně jednomu serveru.

Začali jsme tedy hledat více low-level knihovny s větší flexibilitou, které však stále poskytují pohodlnou abstrakci. Rozhodovali jsme se mezi knihovnou Unity Transport [13], jež je specificky určená k vývoji her v Unity, a WatsonTcp[10], jejíž použití je vázáno pouze na platformu .NET.

Obě knihovny splňovaly naše síťové požadavky, avšak použití Unity Transport by znamenalo, že běh klienta i serveru by byl závislý na použití Unity. Rozhodli jsme se tedy nakonec využít knihovnu WatsonTcp, čímž jsme získali větší flexibilitu pro další vývoj (např. umělá inteligence pro naši hru).

4.1.3 Hlavní komponenty

Rozlišili jsme několik hlavních problémů, které bylo potřeba řešit – komunikace, zpracování zpráv, herní logika a uživatelské rozhraní. Kvůli tomu jsme se rozhodli aplikaci rozdělit na jednotlivé komponenty, které řeší jeden zadaný problém (viz



Obrázek 4.1 Hlavní komponenty.

Obrázek 4.1). Díky tomu v případě změny vnitřní implementace jedné komponenty (např. změna technologie přenosu, či úprava herní logiky) není potřeba jakkoliv měnit ostatní komponenty.

4.1.4 Autorská práva

V neposlední řadě jsme také mysleli na autorská práva hry `Bang!`, včetně originální herní grafiky. Kontaktovali jsme tedy firmu `daVinci Editrice S.r.l.`[14], která je držitelem těchto autorských práv, a obdrželi povolení k použití pro naše účely. Podmínkou bylo, že nesmíme naši implementaci hry nijak šířit mezi veřejný trh, přičemž se zveřejněním v univerzitním digitálním repozitáři firma souhlasila. Také jsme se museli řídit podmínkami pro fanouškovské příspěvky hry `Bang!`. Tímto bych chtěl firmě poděkovat za štědré povolení.

4.2 Implementace

V této podkapitole popíšeme hlavní problémy, které jsme řešili při implementaci herního prostředí.

4.2.1 Návrh herního pole a ovládání hry

Obrázek 4.2 Uspořádání herního pole, které je schopné zobrazit herní informace o maximálních velikostech.

V tomto oddílu popíšeme, jak jsme navrhovali herní pole a ovládání hry. Herní stav představuje relativně mnoho informací, a tak jsme museli dbát na to, abychom byli schopni tyto informace zobrazit v herním poli. Zároveň jsme se snažili přiblížit se co nejblíže prožitku z hraní fyzické verze hry.

Nejprve shrneme všechny informace z herního stavu, které musíme vždy zobrazovat:

- Oponenti:

  - Postava.
  - Počet životů.
  - Role – odhalená pouze, pokud je oponent Šerif nebo je vyřazený.
  - Vyložené modré karty.
  - Počet karet v ruce.
- Klient:
  - Postava.
  - Počet životů.
  - Role – pro klienta vždy viditelná.
  - Vyložené modré karty.
  - Karty v ruce.
- Obecné:
  - Poslední zahraná karta – hráči mohou pozorovat, jaké karty oponenti hrají, a kromě toho ji využívají některé schopnosti postav.

Můžeme si všimnout, že součástí herního stavu jsou informace, jejichž velikost je variabilní. Mezi takové patří karty v ruce klienta, karty vyložené před každým hráčem a počet oponentů. Pro ně platí následující limity:

- Počet karet v ruce – maximálně 18:
  - Jsme Šerif a máme postavu `Bart Cassidy`, díky jehož schopnosti si při ztrátě životu můžeme líznout jednu kartu.
  - Ukončíme tah s maximálním počtem životů (pět) a s *pěti* kartami v ruce.
  - Než začne náš další tah, oponenti nám uberou čtyři životy, díky čemuž si lízneme *čtyři* karty. Jeden z oponentů následně zahraje všechny (jeden) dostupné `Salony`, čímž nám obnoví jeden život, načež nám ho opět ubere, díky čemuž si lízneme *jednu* kartu. Nyní máme v ruce 10 karet.
  - Poté oponenti zahrají všechna (dvě) dostupná `Hokynářství`, čímž získáme další *dvě* karty a v ruce máme dohromady 12 karet.
  - Na začátku našeho tahu si lízneme *dvě* karty a díky tomu máme v ruce 14 karet.
  - Během našeho tahu zahrajeme všechny (dva) dostupné `Dostavníky` – tedy zahrajeme dvě karty, abychom získali čtyři karty. Tím počet karet v ruce stoupne o *dvě* na 16.
  - Jako poslední zahrajeme `Wells Fargo` – tedy zahrajeme jednu kartu, abychom získali tři další. Tím počet karet stoupne o *dvě* na 18.
  - Jelikož jsme vyčerpali všechny možnosti, jakými lze získat karty, jedná se o maximální počet karet v ruce.

- Počet vyložených karet před každým hráčem – maximálně šest: Zbraň, `Appaloosa`, `Barel`, `Dynamit`, `Mustang`, `Vězení`.
- Počet protivníků – maximálně šest.

Aby bylo pro hráče ovládání hry přirozené, snažili jsme ho co nejblíže napodobit hraní fyzické verze hry. Navržené funkcionality popisujeme pomocí `user stories`:

- Jako hráč chci být schopen zahrát kartu z ruky pomocí přetažení dané karty do středu herního pole, abych měl přirozený požitek z hraní hry.
- Jako hráč chci být schopen použít vyloženou modrou kartu kliknutím na ní, abych měl přirozený požitek z hraní hry.
- Jako hráč chci být schopen použít schopnost postavy kliknutím na ní, abych měl přirozený požitek z hraní hry.
- Jako hráč chci být schopen zvolit cílového hráče efektu kliknutím na něj, abych měl přirozený požitek z hraní hry.

Dále jsme pro zlepšení uživatelské přívětivosti přidali následující funkce, které opět popisujeme pomocí `user stories`:

- Jako hráč chci být schopen zvětšit kartu podržením kurzorem myši nad ní, abych si mohl lépe prohlédnout karty, které neznám.
- Jako hráč chci být schopen zvětšit postavu podržením kurzorem myši nad ní, abych si mohl lépe prohlédnout postavy, které neznám.
- Jako hráč chci být schopen přejetím myší nad oponentem zobrazit počet karet v jeho ruce, abych je nemusel sám počítat.
- Jako hráč chci vidět ikonku přesýpacích hodin u hráče, který je momentálně na tahu, abych viděl jasně, kdo je na tahu.
- Jako hráč chci vidět aktuální fázi tahu, abych se snadněji orientoval v dění hry.
- Jako hráč chci být schopen zobrazit si historii herních akcí, abych se mohl zpětně zorientovat v dění hry.

Výsledné uspořádání herního pole ukazuje Obrázek 4.2.

4.2.2 Komunikace a zpracovávání zpráv

Klient a server si mezi sebou posílají zprávy. Každá zpráva je objekt typu dědícího od třídy `NetworkMessage`.

Pro zpracovávání zpráv jsme se rozhodli použít návrhový vzor Visitor, jelikož v implementaci definujeme mnoho zpráv (např. pro každou herní akci), přičemž každá z nich vyžaduje své specifické zpracování. Díky použití tohoto návrhového vzoru není potřeba při přidání nového typu zpráv jakkoliv zasahovat do implementace zpracovávání zpráv (viz Program 1).

Program 1 Zpracování zpráv pomocí návrhového vzoru Visitor

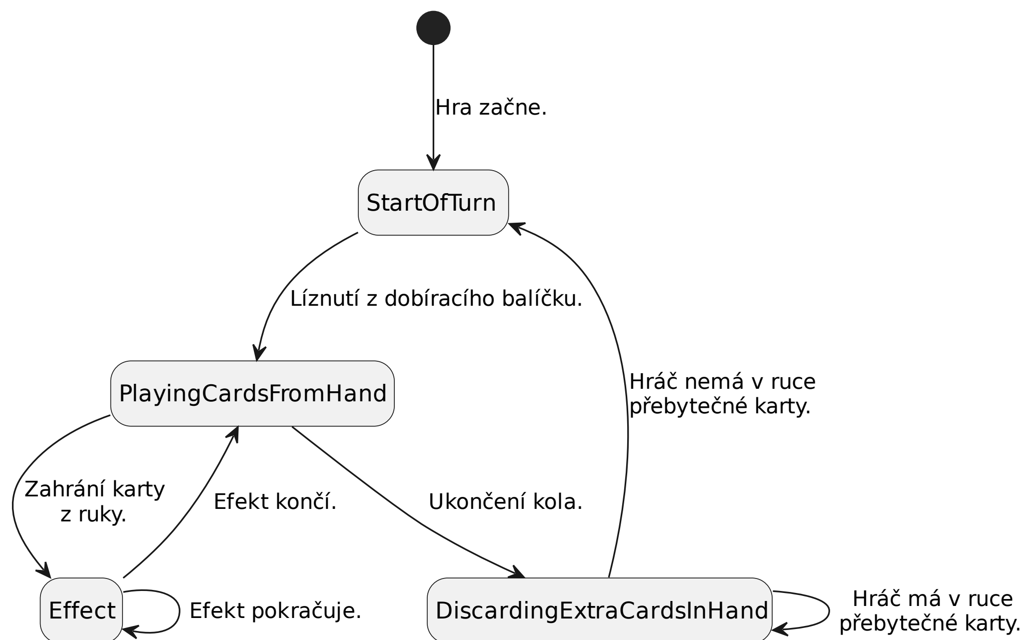
```
class PlayCardFromHandMessage : NetworkMessage {
    public override void ReceivedOnServer(Server s) {
        s.Receive(this);
    }
}

class UseCharacterAbilityNetworkMessage : NetworkMessage {
    public override void ReceivedOnServer(Server s) {
        s.Receive(this);
    }
}

class Server {
    public void ProcessMessage(NetworkMessage msg) {
        msg.ReceivedOnServer(this);
    }

    public void Receive(PlayCardFromHandMessage msg) {
        // Process the message.
    }

    public void Receive(UseCharacterAbilityNetworkMessage msg) {
        // Process the message.
    }
}
```

Program 2 Přímočaré zpracování zpráv pomocí kontroly typu

```
class Server {
    public void ProcessMessage(NetworkMessage msg) {
        if(msg is PlayCardFromHandNetworkMessage) {
            // Process the message
        }
        else if(msg is UseCharacterAbilityNetworkMessage) {
            // Process the message
        }
        // More type checks...
    }
}
```

Druhou cestou, kterou jsme se však nevydali, byl přímočarý přístup, při kterém se zpráva zpracovává na základě explicitní kontroly jejího typu a následnými `if-else` větvemi. Zde je značnou nevýhodou, že bychom v případě přidání nového typu zprávy museli měnit kód zpracovávání přidáním další `if-else` větve (viz v Program 2), což je z dlouhodobého hlediska špatně udržitelné.

4.2.3 Fáze tahu



Obrázek 4.3 Standardní průběh tahu.

Již jsme popsali, jak probíhá standardní tah hráče v Podkapitole 3.4. Fáze tahu jsme však v naší implementaci raději nazvali slovním popisem, namísto číselným, jak je tomu v pravidlech hry. Přechody mezi těmito fázemi pomocí herních akcí znázorňuje Obrázek 4.3.

Vedle fází tahu definovaných pravidly hry jsme přidali fázi `Effect` reprezentující probíhající efekt karty či schopnosti. Pro některé speciální herní efekty jsme uvnitř fáze `Effect` zavedli ještě navíc následující fáze:

- `ChoosingCard` – výběr karty, např. při efektech karet `Cat Balou` či `Hokynářství`.
- `ReactingToCard` – reakce na kartu, např. při kartě `BANG!`.
- `DrawingForEffect` – fáze sejmutí, např. při sejmutí karty pro efekt karty `Barel`,
- `RecoveringFromLethalDamage` – léčení smrtelného zásahu. Jakmile hráčův počet životů klesne pod jedna, má možnost se zachránit okamžitým zahráním karty `Pivo`.

4.2.4 Herní logika

V herní logice poskytujeme rozhraní pro příjem herních akcí, na základě kterých ovlivňujeme pozorovatelný herní stav. Předpokládáme, že každá přijatá herní akce se *vždy* váže ke hráči, který je momentálně na tahu.

Hra obsahuje mnoho karet a schopností, jejichž efekty specificky ovlivňují průběh tahu – např. omezením povolených herních akcí či omezením karet, které lze zahrát (např. karta `BANG!`). Ke správné implementaci hry dle pravidel je tedy potřeba vědět, jaký efekt karty či schopnosti postavy je momentálně aktivní.

Přímočarý přístup, kdy hra explicitně kontroluje, jaký efekt je aktivní, a podle toho zpracovává herní akce, jsme zavrhli, jelikož je nepřehledný a špatně se rozšiřuje (nastíněno v Programu 3).

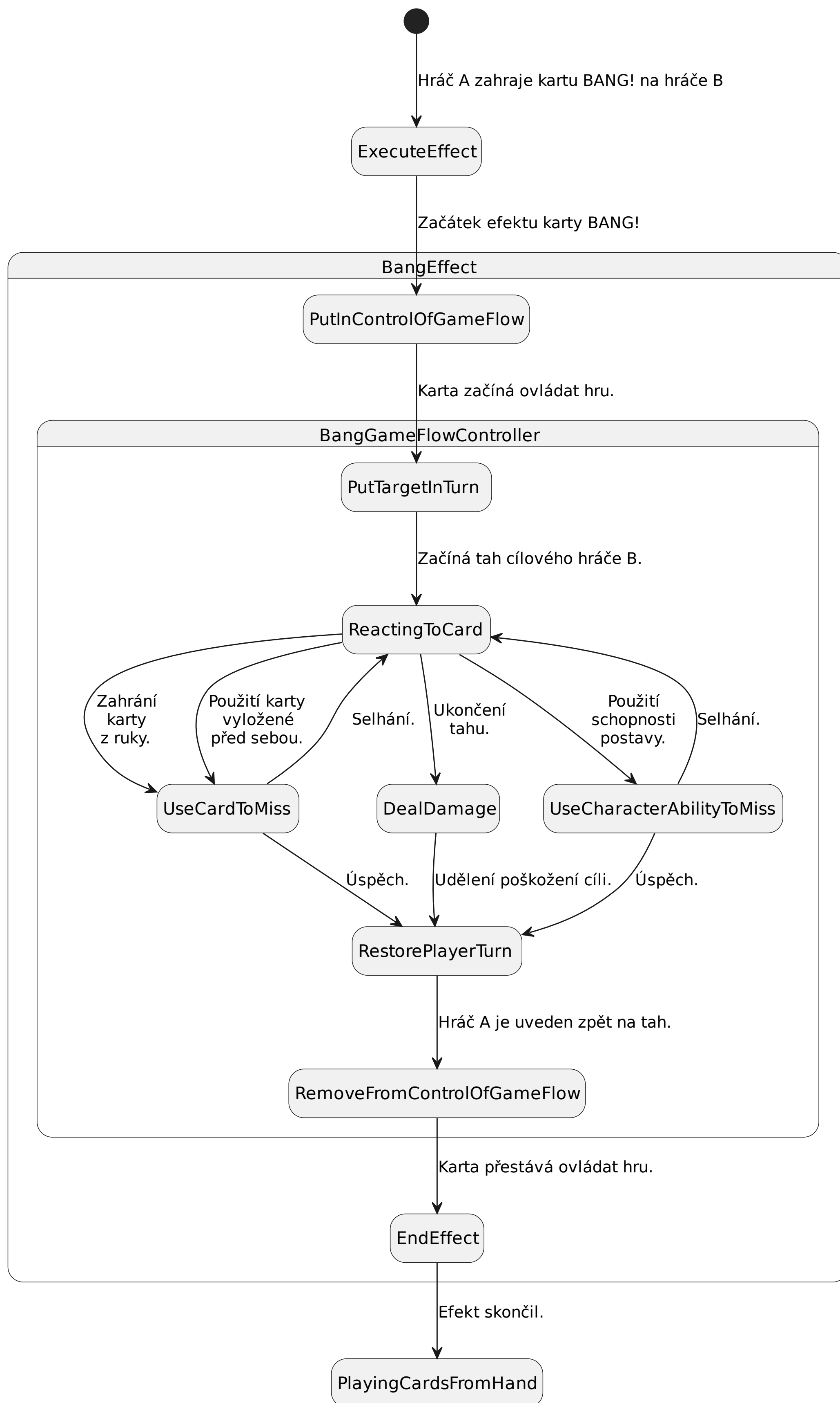
Program 3 Přímočará herní logika pomocí kontroly typu efektu

```
public void HandleEndTurnGameAction() {
    if(IsBangCardEffectActive()) {
        // Handle the game action correctly.
    }
    // More checks...
}
```

Rozhodli jsme se tedy, že nejen každý efekt, ale i standardní tah bude mít svůj specifický ovladač herního tahu, přičemž herní logika vždy deleguje herní akce aktuálnímu ovladači. Tím pádem efektu přenecháváme veškerou zodpovědnost za jeho správný průběh, včetně validace herních akcí (nastíněno v Programu 4). Obrázek 4.4 pak názorně ukazuje přibližnou implementaci karty `BANG!` dle tohoto řešení.

Program 4 Nástin herní logiky pomocí `interface IGameFlowController`

```
interface IGameFlowController {
    void EndTurn();
    // Other methods for handling game actions...
}

class GameField {
    public void PutInControl(IGameFlowController c){
        // Put the controller in control.
    }
    public void RemoveFromControl(IGameFlowController c) {
        // Make sure that the passed controller is currently
        // in control.
        // After that, remove the current controller from control.
    }
}

class BangCard : IGameFlowController {
    GameField _gf;
    public void ExecuteEffect() {
        _gf.PutInControl(this);
        // Set the target, etc.
    }

    public void EndTurn() {
        // Handle the game action correctly...
        // At the end, remove this from control.
        _gf.RemoveFromControl(this);
    }

    // Other methods for handling game actions...
}

public void HandleEndTurnGameAction() {
    // Redirect handling the action to the current controller.
    var controller = GetCurrentGameFlowController();
    controller.EndTurn();
}
```



Obrázek 4.4 Přibližný průběh efektu karty `BANG!`.

4.2.5 Zanořené efekty

Ve hře může dojít k zanořování efektů. Např. hráč `A` zahraje kartu `BANG!` na hráče `B`, který jako reakci použije vyloženou kartu `Barel`. Při efektu karty `Barel` si hráč lízne jednu kartu z dobíracího balíčku – pokud je srdcová, pak se hráč vyhnul efektu karty `BANG!`. V opačném případě má hráč ještě šanci vyhnout se kartě jinými způsoby. Zde si můžeme všimnout, že na základě výsledku efektu karty `Barel` se efekt karty `BANG!` ukončí, nebo se obnoví.

Jelikož jsou interakce mezi různými efekty obecně odlišné, rozhodli jsme se tyto interakce nechat na starosti právě efektům. Při zastavení kontroly aktuálního herního ovladače `X` vždy předáme kontrolu předchozímu ovladači `Y`. Případnou interakci mezi těmito ovladači následně necháváme na starost ovladači `X`. Toto řešení je nastíněno na Obrázku 4.5.

Největší hloubka zanoření efektů, kterého lze v základní verzi hry dosáhnout, jsou tři efekty. Toho lze dosáhnout tak, že při efektu karty `Barel` hráč použije navíc schopnost postavy `Lucky Duke` – díky ní hráč při mechanice `Sejmutí` (popsané v Podkapitole 3.10) může sejmout dvě karty a z nich si vybrat, kterou použije.

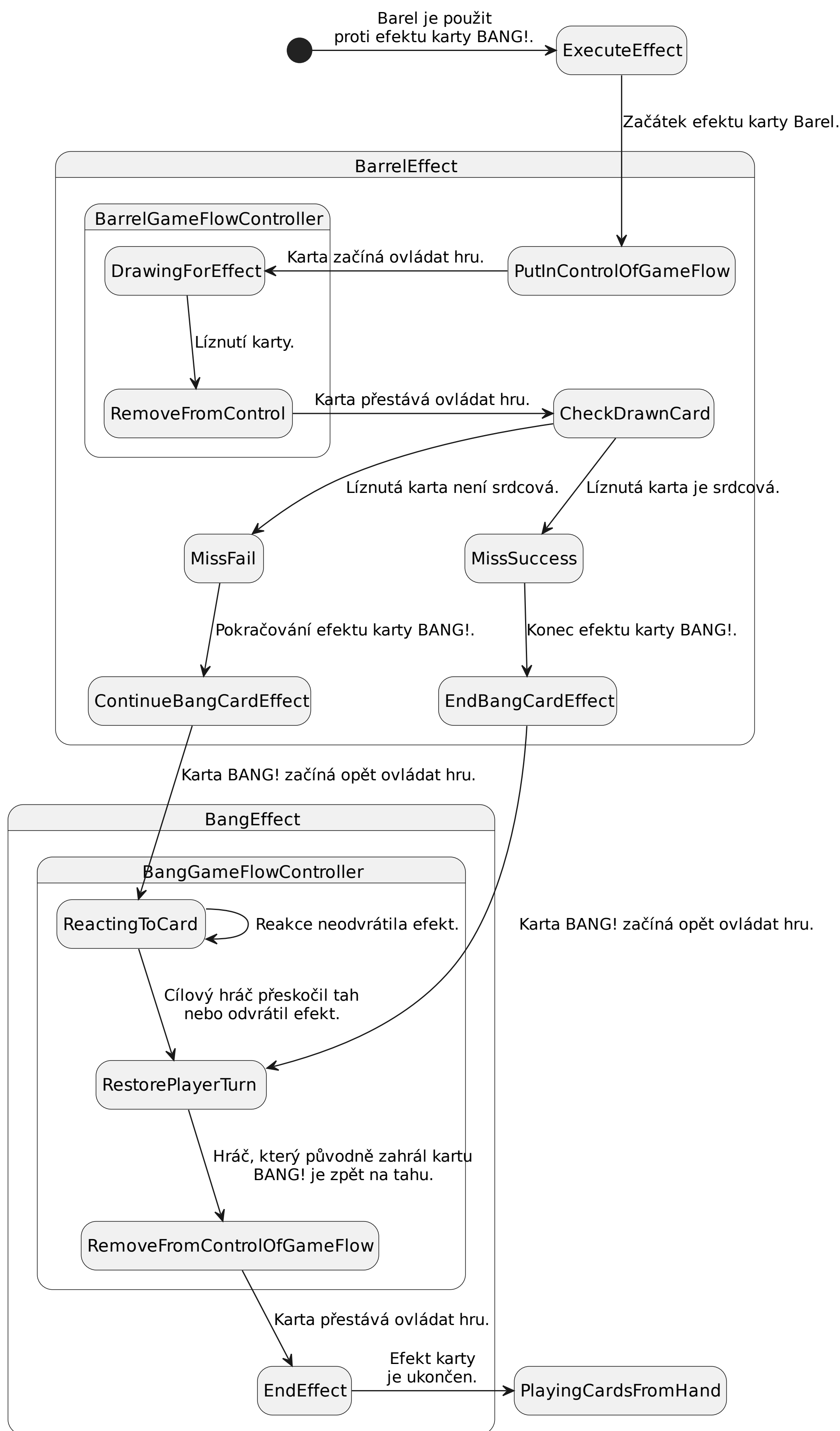
4.2.6 Oddělení herní logiky a síťování

Jelikož jsme nechtěli, aby byla herní logika spjatá se síťováním, rozhodli jsme se, že všechny efekty budou s hráči komunikovat pouze pomocí úprav herního stavu. Pokud např. nějaký efekt vyžaduje od určitého hráče herní akci, pak v herním stavu nastaví, že je tento hráč na tahu.

Můžeme si klást otázku, zda je možné vždy určit, jaký hráč je na tahu, jelikož kromě pravidelného střídání hráčů po ukončení tahu se ve hře se může stát, že někdo aktivuje efekt vyžadující reakci od oponentů. Takové reakce ale vždy probíhají v určitém pořadí, a tím pádem můžeme říct, že hráč, od kterého očekáváme reakci, je v rámci efektu na tahu.

Díky tomu můžeme na straně serveru vždy určit, od kterého hráče máme přijímat herní akce, a to tak, že zkontrolujeme, jaký hráč je momentálně na tahu. Když tedy přijmeme herní akci od nějakého klienta, který není na tahu, můžeme tuto herní akci jednoduše zamítnout.

V opačném případě delegujeme akci herní logice. Pokud je herní akce v souladu s pravidly, pak pošleme klientům aktualizovaný herní stav. Jinak pošleme danému klientu informaci o tom, že akce byla proti pravidlům, společně s vysvětlením proč.



Obrázek 4.5 Přibližné řešení zanořování efektů karet `BANG!` a `Barel`

5 Umělá inteligence

V této kapitole shrneme, jak jsme řešili výzkum a vývoj umělé inteligence pro hru `Bang!`.

Nejprve se budeme věnovat modelům strojového učení, které jsme trénovali k ohodnocování herních akcí v daném herním stavu. Trénovací data modelů jsme sbírali z her náhodných hráčů s myšlenkou, že poskytnou různorodá trénovací data. Poté popíšeme, jak jsme využili neuroevoluční algoritmus NEAT k trénování univerzálních a specializovaných neuronových sítí pro ohodnocování herních akcí.

Následně vyhodnotíme výkonnost natrénovaných umělých inteligencí ve hrách proti náhodným hráčům. Specializované neuronové sítě natrénované pomocí NEAT, které jevily nejlepší výsledky, pak otestujeme i proti člověku. Zjistíme, že přestože měly problém s odhadováním rolí oponentů a s výběrem správných rozhodnutí v jednoduchých herních situacích, tak vykazovaly, že rozumí hlavním konceptům hry a byly schopny hrát v týmu i samostatně.

5.1 Strojové učení

Strojové učení se již osvědčilo jako efektivní metoda v oblasti umělé inteligence, a proto jsme se rozhodli pokusit prozkoumat využití této metody i ve hře `Bang!`.

5.1.1 Základní pojmy

V tomto oddíle shrneme základní pojmy ze strojového učení potřebné k porozumění textu. U pojmů, pro které jsme nenalezli ustálený český název, jsme se rozhodli použít anglický název.

Agent je cokoliv, co vnímá své prostředí pomocí senzorů a reaguje na prostředí pomocí aktuátorů. Agent se **učí**, pokud zlepšuje svůj výkon v budoucích úlohách po pozorování prostředí. Rozdělujeme několik druhů učení. V **učení bez učitele** se agent učí bez jakékoliv zpětné vazby. Ve **zpětnovazebném učení** se agent učí na základě odměn, které značí, že provedl něco užitečného. V **učení s učitelem** agent pozoruje vzorové dvojice (vstup, výstup) a učí se funkci mapující daný vstup na výstup. (S. J. Russell a P. Norvig [15]).

V kontextu strojového učení našemu agentu budeme říkat **model**, který vnímá své prostředí příjmem dat na vstupu a reaguje na něj vrácením odpovídající **předpovědi** (**predikce**) na výstupu. Učení modelu pak budeme také říkat **trénování modelu**.

Vstupem modelu je typicky vektor reálných čísel předem známé velikosti, popř. reálná matice, jejíž řádky reprezentují vstupní vektory. Takové matici, kterou zároveň používáme k trénovaní modelu, také říkáme **trénovací množina** (**training set**) a jejím řádkům také říkáme **příklady** (**examples**) .

Výstup modelu závisí na tom, jakou úlohu řešíme. Mezi základní úlohy strojového učení patří regrese, klasifikace a clustering.

U **regrese** je cílem pro daný vstup předpovědět nějaké reálné číslo, např. předpověď zítřejší denní teploty, a výstupem je tedy reálné číslo.

U **klasifikace** je cílem pro daný vstup předpovědět jednu z konečné množiny tříd, např. zda zítra bude slunečno, zataženo, či jestli bude pršet. Zde může být

výstupem reálný vektor, jehož každá složka odpovídá jedné třídě a reprezentuje pravděpodobnost, s jakou je odpovídající třída ta správná.

V případě **clustering** úloh je cílem nalézt v datech nějaké shluky dat, které mají něco společného (S. J. Russell a P. Norvig [15]).

5.1.2 Hladový výběr nejlepší herní akce pomocí regrese

V Oddílu 2.2.3 popisujícím evoluční algoritmus M. Danilákové [5] jsme již zmínili, že její umělá inteligence vybírala hladově tu herní akci, která měla přinést nejlepší ohodnocení následujícího herního stavu. Tato strategie přinesla nejlepší výsledky, přičemž se ukázalo, že dopředné plánování pomocí MCTS se neosvědčilo.

Na základě toho jsme se tedy rozhodli, že naše umělá inteligence také nebude nijak plánovat dopředu, a bude hladově vybírat herní akci, která by měla přinést nejlepší ohodnocení následujícího herní stavu.

Problém výběru nejlepší herní akce je možné řešit jako klasifikaci nebo pomocí regrese. V případě klasifikace by model na výstupu vracel vektor reálných čísel reprezentujících pravděpodobnosti, s jakými jsou jednotlivé herní akce ty nejlepší. Jelikož se klasifikace řeší pomocí učení s učitelem, bylo by hlavním problémem to, že bychom nejprve museli sami umět určit tyto pravděpodobnosti.

Nakonec jsme se tedy rozhodli řešit náš problém pomocí regrese. V našem případě model na výstupu vrací ohodnocení herní akce, načež ze všech možných herních akcí provádíme tu, která je modelem hodnocena nejlépe.

5.1.3 Návrh a sběr trénovacích dat modelu

V tomto oddíle nejprve popíšeme, jak jsme navrhli vstupní data modelu, a poté popíšeme, jak jsme došli k rozhodnutí sbírat vzorové výstupy ze simulovaných her náhodných hráčů. Na závěr shrneme, jak počítáme ohodnocení herní akce, které slouží jako vzorový výstup.

Návrh vstupních dat modelu

V Oddílu 5.1.2 jsme již zmínili, že naším záměrem bylo natrénovat model ohodnocující herní akce. A jelikož vhodnost herních akcí závisí na aktuálním herním stavu, došli jsme přirozeně k tomu, že součástí vstupu modelu bude herní akce a aktuální herní stav.

Ve strojovém učení reprezentujeme vstupní data pomocí reálných vektorů fixní velikosti. Museli jsme tedy řešit, jak zakódovat herní stav a herní akce do vektoru fixní velikosti, přestože už jen např. počet hráčů ve hře je variabilní.

Seznam karet v ruce hráče reprezentujeme tak, že pro každý typ herní karty existuje ve vstupním vektoru složka, jejíž hodnota vyjadřuje, kolik karet daného typu má hráč v ruce (podobně pro seznam karet vyložených před hráčem).

Dále jsme museli řešit variabilní počet oponentů ve hře. Jelikož je jejich počet omezen na šest (hru lze maximálně hrát v sedmi hráčích), máme ve vstupním vektoru šest skupin složek, kde i-tá skupina představuje informace o i-tém oponentovi po směru hodinových ručiček z pohledu daného hráče. V případě, že zaznamenáváme data o méně než šesti oponentech, pak přebytečné složky nastavíme na speciální hodnoty značící, že se mají dané sloupce ignorovat.

V poslední řadě jsme řešili reprezentaci herních akcí, jež lze obecně dělit na osm typů:

- Líznutí karet.
- Zahrání karty – většina bez parametrů, některé vyžadují cílového hráče (např. Vězení) či herní karty (např. Cat Balou).
- Použití karty vyložené před sebou.
- Použití schopnosti postavy – opět většina bez parametrů, některé vyžadují cílového hráče efektu.
- Výběr karty (efekt karty Hokynářství).
- Výběr herní akce (schopnost postavy Jesse Jones).
- Odhození nadbytečné karty.
- Ukončení kola.

Můžeme si všimnout, že některé herní akce mají parametry (typ karty či cílového hráče) a některé ne. Rozhodli jsme se tedy reprezentovat herní akce pomocí tří složek – typ herní akce, typ karty a relativní pozice cílového hráče vůči klientovi ve směru hodinových ručiček. Typ karty a relativní pozice cílového hráče slouží jako volitelné parametry herní akce. Pokud herní akce dané parametry nepotřebuje, pak u nich nastavíme speciální hodnotu značící, že parametr není používaný.

Shrnutí informací o herním poli

Aby měl model co nejlepší přehled o hře, rozhodli jsme se ve vstupu zahrnout veškeré viditelné informace o herním stavu. Také jsme dále mj. ve vstupu zahrnuli informace, které by měly pomoct při rozpoznávání rolí, jako je např. počet útoků jednotlivých hráčů na Šerifa.

Ve shrnutí model přijímá na vstupu následující informace:

- Dostupné informace o všech hráčích (více dopodrobna níže).
- Karta navrchu odhazovacího balíčku (může být užitečná pro některé schopnosti postav).
- Fáze tahu.
- Karta, jejíž efekt je momentálně aktivní.
- Schopnost postavy, jejíž efekt je momentálně aktivní.

Detail informací hráčů

Pro každého hráče model přijímá na vstupu následující informace:

- Postava.
- Aktuální počet životů.
- Počet karet v ruce.
- Zda má či nemá před sebou vyložené jednotlivé modré herní karty (kromě karet, které pouze ovlivňují dosah zbraně a vzdálenost – tyto informace reprezentujeme zvlášť).
- Dosah zbraně.
- Vzdálenost od hráčů.
- Vzdálenost ke hráčům.

O hráči, z jehož pohledu hodnotíme herní stav, model přijímá navíc tyto informace:

- Role.
- Počet jednotlivých herních karet v ruce.

O ostatních hráčích model přijímá navíc tyto informace:

- Role – pokud je daný hráč mrtvý, pak musí odhalit svou roli. Také lze někdy vyřazovací metodou zjistit role ostatních hráčů. Jinak je role brána jako neznámá.
- Počet útoků na hráče, z jehož pohledu herní stav hodnotíme.
- Počet útoků na Šerifa.

Vzorové výstupy

Ve hře `Bang!` je hlavním problémem správně uhodnout role ostatních hráčů, což představuje překážku i při ohodnocování herních akcí. Např. pokud Šerif udělí poškození nějakému hráči, pak ohodnocení této akce závisí na roli vybraného hráče.

Naší hlavní myšlenkou bylo naučit model ohodnocovat herní akce na základě neúplných informací o herním stavu (např. neznámé role oponentů) tak, aby byl co nejblíže ohodnocení s úplnými informacemi (včetně rolí oponentů). Hledali jsme tedy způsob, jakým získat vzorové výstupy odpovídající ohodnocení herních akcí s úplnými informacemi o herním stavu.

V našem herním prostředí má přístup k úplným informacím o herním stavu pouze server. Např. pokud Šerif zaútočí na hráče, který je jeho Pomocník, pak je tato herní akce pro Šerifa nepřínosná – Šerif to samozřejmě nemůže s jistotou vědět, kdežto server ano, jelikož má přístup ke všem herním informacím.

Nabízí se tedy vzorové výstupy sbírat přímočaře právě na straně serveru, ale jelikož má síťování nezanedbatelnou režii, rozhodli jsme se sbírat data pomocí speciálního simulátoru her.

Učení se z her náhodných hráčů

Jelikož jsme nesměli herní prostředí veřejně distribuovat mimo univerzitní digitální repozitář (Oddíl 4.1.4), měli jsme omezené možnosti učení našeho modelu z her skutečných hráčů. Pokusili jsme se tedy náš model učit z dat her náhodných hráčů s myšlenkou, že náhodné hry prozkoumají různé výhodné i nevýhodné herní stavy a reakce na ně, ze kterých se model bude moct učit.

Ohodnocení herní akce a herního pole

Ohodnocení herní akce počítáme jako rozdíl ohodnocení herního stavu po provedení akce a ohodnocení herního stavu před provedením akce. Tím pádem kladné ohodnocení herní akce znamená, že přinesla zlepšení herního stavu, přičemž záporné znamená zhoršení. A zároveň vyšší ohodnocení znamená větší zlepšení.

U ohodnocení herního stavu jsme se inspirovali výsledky evolučního algoritmu M. Danilákové [5] (Oddíl 2.2.3) a počítáme ho z následujících informací:

- Počet životů hráče, z jehož pohledu hodnotíme.
- Počet životů oponentů podle vztahu jejich rolí vůči roli hráče, z jehož pohledu hodnotíme (např. pokud má nějaký oponent Bandita málo životů, tak herní stav bude hodnocen dobře z pohledu hráče, který je Šerif, ale nikoliv z pohledu hráče, který je také Bandita).
- Pro každého nepřítele a spojence:
  - Zda má před sebou vyloženou kartu `Barel`.
  - Zda má před sebou vyloženou kartu `Dynamit`.
  - Zda má před sebou vyloženou kartu `Vězení`.
  - Zda má před sebou vyloženou kartu `Volcanic`.
  - Dosah zbraně.
  - Vzdálenost od ostatních hráčů.
  - Vzdálenost k ostatním hráčům.
  - Počet karet v ruce.

5.1.4 Implementace

V tomto oddílu nejprve popíšeme, proč jsme k trénování modelů strojového učení zvolili knihovnu `scikit-learn`. Poté vysvětlíme, proč ukládáme natrénované modely ve formátu `ONNX`, a na závěr popíšeme transformace vstupních dat a metriky, pomocí kterých jsme měřili, jak dobře modely ohodnocují herní akce.

Knihovna scikit-learn

K trénování našeho modelu jsme se rozhodli využít knihovnu `scikit-learn` [16] pro jazyk Python, jelikož nabízí mnoho algoritmů (**estimátorů**) strojového učení včetně prostředků pro pohodlné zpracování dat a vyhodnocení modelů, je jednoduchá k použití a má výbornou dokumentaci. V době psaní práce byla knihovna stále udržovaná a populární, díky čemuž k ní existuje mnoho edukačních materiálů.

Formát ONNX a ONNX Runtime

Po natrénování modelu je vhodné ho perzistentně uložit, abychom ho při budoucím použití mohli pouze načíst bez nutnosti opětovného trénování. Jelikož je klient našeho herního prostředí vázán na platformu .NET, bylo potřeba ukládat model tak, abychom ho byli schopni v této platformě jednoduše načíst a používat.

Model natrénovaný pomocí knihovny `scikit-learn` potřebujeme pouze k predikci, a to bez jakýchkoliv informací o Python objektu, kterým byl reprezentován. V tomto případě dokumentace [17] knihovny `scikit-learn` doporučuje ukládat natrénovaný model ve formátu `ONNX` [18]. Jelikož lze takto uložené modely na platformě .NET přímočaře načíst a použít k predikci pomocí knihovny `ONNX Runtime` [19], rozhodli jsme se právě pro tento formát. Další výhodou je, že nevzniká závislost na použité technologii při trénování modelů. Stačí pouze, aby byl model uložen v tomto formátu.

Transformace vstupu

Ve vstupu se nachází několik sloupců, jejichž hodnoty reprezentují nějaké **kategorie** (náleží výčtu hodnot). Takové sloupce označujeme jako **kategorické**. V našich trénovacích datech máme např. **kategorický sloupec** reprezentující postavu hráče – hodnota `2` v tomto sloupci odpovídá postavě `Black Jack`, hodnota `3` odpovídá postavě `Calamity Janet`, atd.

Taková reprezentace však není vhodná pro všechny estimátory knihovny `scikit-learn`, jelikož by interpretovaly kategorie jako uspořádané (dokumentace knihovny `scikit-learn` [17]), což není žádané – v našem případě např. postavy a odpovídající hodnoty, kterými jsou reprezentovány, byly přiřazeny nahodile.

Rozhodli jsme se tedy, že kategorické sloupce budeme transformovat do `one-hot-encoding`. Jeho princip je takový, že kategorický sloupec nahradíme `n` novými sloupci, kde `n` je rovno počtu kategorií. Poté místo každé původní kategorické hodnoty `i` nastavíme ve stejném řádku hodnotu v `i`-tém novém sloupci na jedničku a ostatní na nulu. Knihovna `scikit-learn` pro tento typ transformace nabízí třídu `OneHotEncoder`, jejíž instanci s parametry `sparse_output=False, handle_unknown="ignore"` jsme využili k transformaci kategorických sloupců.

V dokumentaci knihovny `scikit-learn` [17] je dále zmíněno, že mnoho jejích estimátorů vyžaduje, aby vstupní data v jednotlivých sloupcích měla přibližně normální distribuci. Rozhodli jsme se tedy standardizovat data pomocí instance třídy `StandardScaler` (s výchozími parametry), čímž jsme tento požadavek splnili.

Metriky

Výběr vhodného algoritmu strojového učení je důležitou částí řešení našeho problému, jelikož různé algoritmy jsou lépe uzpůsobené pro různé typy dat a problémů. Vhodnost daného algoritmu lze měřit podle různých metrik, které obecně vyjadřují schopnost modelu správně předpovídat. My jsme se rozhodli ji měřit pomocí koeficientu determinace (R^2) a střední kvadratické chyby (MSE), jejichž definice nyní uvedeme.

Mějme n vstupních dat a nechť X_i značí i-tou predikovanou hodnotu pro i-tý vstup, Y_i značí vzorový výstup pro i-tý vstup a $\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i$ značí průměrnou

hodnotu všech vzorových výstupů. Pak definujeme **koeficient determinace** (**R^2**) následovně:

$$R^2 = 1 - \frac{\sum_{i=1}^{n} (X_i - Y_i)^2}{\sum_{i=1}^{n} (\bar{Y} - Y_i)^2},$$

a **střední kvadratickou chybu** (**MSE**) následovně:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (X_i - Y_i)^2,$$

přičemž R^2 lze interpretovat jako podíl variance závislé proměnné (výstup), který lze předpovídat z nezávislých proměnných (vstup) (D. Chicco, M. J. Warrens a G. Jurman [20]) – hodnota R^2 rovna jedné tedy znamená, že model umí pro daný vstup dokonale předpovědět správný výstup.

Simulátor her a sběr dat

Pro sběr dat jsme implementovali simulátor her umělých inteligencí, který lze také použít k měření jejich výkonnosti ve hře. Ten funguje v iteracích, přičemž v každé iteraci předá všem umělým inteligencím herní stav a navíc tu, která je na řadě, požádá o herní akci. Jakmile je současná hra u konce, automaticky začne v následující iteraci simulovat novou hru. Jedna instance simulátoru simuluje právě jednu hru, což umožňuje přímočarou paralelizaci.

Projekt programu určeného pro sběr dat náhodných hráčů můžeme nalézt v přílohách práce v adresáři `ExperimentsSolution/DataCollectionProject`. K zajištění determinismu v programu přijímáme mj. počet iterací, které máme provést, a seed, na základě kterého simulujeme náhodné jevy nejen v simulované hře, ale i při výběru herních akcí náhodnými hráči.

V programu simulujeme jednu hru pro každý možný počet hráčů ve hře a každá simulace zaznamenává data do vlastního textového souboru. Výsledná trénovací data jsme získali spojením nasbíraných dat za sebou, a to v pořadí od nejmenšího počtu hráčů po největší.

5.1.5 Měření výkonnosti simulátoru her

Abychom se ujistili, že výkonnost našeho simulátoru her nebude představovat omezení při sběru dat a měření výkonnosti umělých inteligencí, rozhodli jsme se změřit čas, za který zvládne simulovat daný počet herních tahů. Simulace probíhaly ve hrách náhodných hráčů. Předpokládáme, že netriviální umělé inteligence budou vybírat herní tahy déle než náhodní hráči, a tím pádem získáme horní odhad výkonnosti simulátoru her.

Projekt implementace experimentu můžeme nalézet v přílohách práce v adresáři `ExperimentsSolution/Experiment_AiMatchSimulatorPerformance`.

Metoda

- Počítač, na kterém proběhlo měření:
    - Značka a model: Lenovo Legion 5-15ACH6H laptop

- CPU: AMD Ryzen 5 5600H
- RAM: 16 GB

- Počet instancí simulací: 1
- Parametry programu experimentu:
  - Počet herních tahů: 1000000
  - Seed: 123
  - Počet náhodných hráčů: 4
  - Počet měření: 10

Výsledky

Číslo měření	Doba běhu v sekundách
1	2,56
2	1,98
3	1,95
4	2,25
5	2,08
6	2,11
7	2,07
8	2,07
9	2,04
10	2,07

Tabulka 5.1 Měření doby potřebné k simulaci 1000000 herních tahů

Při sériovém běhu simulace jsme dosáhli průměrného výkonu 1000000 simulovaných herních tahů za 2,12 sekund, z čehož jsme usoudili, že výkonnost našeho simulátor her je dostačující.

5.1.6 Testování estimátorů různých druhů

Knihovna `scikit-learn` nabízí mnoho algoritmů strojového učení (estimátory) určených pro řešení regrese. Jelikož jsme nebyli schopni předem jasně určit, jaký estimátor bude vhodný pro náš problém, rozhodli jsme se z každého druhu potenciálních estimátorů vybrat jeden a pomocí něj natrénovali model strojového učení. Následně jsme se na základě výsledků metrik rozhodli, zda se daným druhem algoritmů zabývat dále. Implementaci experimentu můžeme nalézt v přílohách práce, a to ve funkci `one_ml_alg_from_each_type` skriptu `MachineLearningProject/src/main.py`.

Metoda

- Data nasbíraná pomocí programu pro sběr dat (viz Oddíl 5.1.4):
  - Seed: 0

  - Počet iterací: 3000
- Počet odehraných her: 50
- Rozdělení výher dle rolí:
  - Bandité: 27
  - Šerif a Pomocníci šerifa: 21
  - Odpadlík: 2
- Velikost dat: 11998 řádků
- Rozdělení dat na trénovací a testovací pomocí funkce `sklearn.model_selection.train_test_split`:
  - `test_size`: 0,2
  - `random_state`: 0
- Transformace dat dle Oddílu 5.1.4
- Parametry estimátorů:
  - `random_state`: 0 (pokud bylo možné nastavit)
  - Ostatní: výchozí hodnoty

Výsledky

Estimátor knihovny scikit-learn	**Druh algoritmu**	**R^2**	**MSE**
`LinearRegression`	Lineární model	0,54	0,06
`KernelRidge`	Kernelová metoda	0,54	0,06
`SVR`	SVM[1]	0,42	0,07
`SGDRegressor`	SGD[2]	0,47	0,07
`KNeighborsRegressor`	k-NN[3]	−0,01	0,12
`GaussianProcessRegressor`	Gausovský proces	0,01	0,12
`DecisionTreeRegressor`	Rozhodovací strom	0,73	0,03
`HistGradientBoostingRegressor`	Ensemble metoda	0,82	0,02
`MLPRegressor`	Neuronová síť	0,57	0,05

Pozn.: [1] Support vector machines [2] Stochastic gradient descent [3] K-nearest neighbors

Tabulka 5.2 Výsledky modelů trénovaných různými metodami

Estimátory `DecisionTreeRegressor` a `HistGradientBoostingRegressor` vykazovaly jednoznačně nejlepší výsledky s relativně vysokou hodnotou R^2 a zároveň relativně malou hodnotou MSE (viz Tabulka 5.2). Rozhodli jsme se tedy dále zabývat rozhodovacími stromy a ensemble metodami.

5.1.7 Rozhodovací stromy a ensemble metody

V tomto oddílu shrneme, jak fungují rozhodovací stromy a ensemble metody, jelikož se jim budeme v práci dále věnovat.

Rozhodovací stromy provádějí předpovědi na základě posloupnosti testů podmínek na vstupních datech. Každý vnitřní vrchol stromu reprezentuje danou podmínku, kdežto listy stromu reprezentují hodnoty, které se mají vrátit na výstup. Rozhodování začíná v kořeni, kde otestujeme danou podmínku na vstupních datech, a podle hodnoty výsledku testu se následně přesouváme do příslušného potomka. Tento proces opakujeme, dokud se nepřesuneme do listu, odkud již vrátíme danou hodnotu, která je listem reprezentována (S. J. Russell a P. Norvig [15]).

Hlavní myšlenkou **ensemble metod** je využití kombinace několika estimátorů pro jednu předpověď, přičemž předpokládáme, že misklasifikace několika estimátorů je méně pravděpodobná, než misklasifikace jednoho estimátoru (S. J. Russell a P. Norvig [15]).

Jednou z nejrozšířenějších ensemble metod je **boosting**. U té pracujeme s **váženou trénovací množinou**, což je trénovací množina (viz Oddíl 5.1.1), jejíž každý příklad má nějakou nezápornou váhu, přičemž vyšší váha příkladu znamená, že je ve fázi trénování důležitější. Na začátku mají všechny příklady stejnou váhu a na této trénovací množině se natrénuje první model. Poté chceme, aby byl následující model lepší v předpovídání u příkladů, pro které předchozí model předpovídal špatně – těmto příkladům tedy nastavíme vyšší váhu a zároveň snížíme váhu příkladům, pro které byla předpověď správná. Na této nové vážené trénovací množině následně trénujeme nový model a proces opakujeme, dokud nemáme dostatečný počet modelů. Při predikci se poté rozhoduje dle předpovědí všech modelů, přičemž předpověď každého modelu má váhu podle toho, jak dobře si vedl při trénování(S. J. Russell a P. Norvig [15]).

5.1.8 Testování různých rozhodovacích stromů a ensemble metod

Na základě výsledků Oddílu 5.1.6 jsme se rozhodli, že otestujeme všechny implementace rozhodovacích stromů a ensemble metod, které knihovna `scikit-learn` nabízí. Zjistili jsme však, že knihovna `scikit-learn` nabízí pouze jednu další implementaci rozhodovacích stromů, která je určena k použití uvnitř ensemble metod, a tak jsme testovali pouze zbylé ensemble metody.

Implementaci experimentu můžeme nalézt v přílohách práce, a to ve funkci `dec_trees_and_ensemble_methods` skriptu `MachineLearningProject/src/main.py`.

Metoda

- Data nasbíraná pomocí programu pro sběr dat (viz Oddíl 5.1.4):
  - Seed: 0
  - Počet iterací: 3000
- Počet odehraných her: 50
- Rozdělení výher dle rolí:

    - Bandité: 27
    - Šerif a Pomocníci šerifa: 21
    - Odpadlík: 2
- Velikost dat: 11998 řádků
- Rozdělení dat na trénovací a testovací pomocí funkce `sklearn.model_selection.train_test_split`:
    - `test_size`: 0,2
    - `random_state`: 0
- Transformace dat dle Oddílu 5.1.4
- Parametry estimátorů:
    - `random_state`: 0
    - Ostatní: výchozí hodnoty

Výsledky

Estimátor knihovny sklearn	$\mathbf{R^2}$	**MSE**
`DecisionTreeRegressor`	0,73	0,033
`AdaBoostRegressor`	−0,29	0,159
`BaggingRegressor`	0,81	0,023
`ExtraTreesRegressor`	0,84	0,019
`HistGradientBoostingRegressor`	0,82	0,022
`IsolationForest`	−8,57	1,174
`RandomForestRegressor`	0,83	0,020

Tabulka 5.3 Výsledky různých ensemble metod a rozhodovacího stromu

Velmi podobné nejlepší výsledky vykazovaly estimátory `BaggingRegressor`, `ExtraTreesRegressor`, `HistGradientBoostingRegressor` a `RandomForestRegressor` (viz Tabulka 5.3), se kterými jsme experimentovali dále.

5.1.9 Trénování na větších datech

Pokusili jsme se zlepšit výkon slibných estimátorů z Oddílu 5.1.8 trénováním na velkých datech s očekáváním, že se na větších datech naučí lépe předpovídat.

Implementaci experimentu můžeme nalézt v přílohách práce, a to ve funkci `dec_tree_and_grad_boost_on_big_data` skriptu `MachineLearningProject/src/main.py`.

Metoda

- Data nasbíraná pomocí programu pro sběr dat (viz Oddíl 5.1.4):
  - Seed: 1
  - Počet iterací: 1000000
- Počet odehraných her: 18057
- Rozdělení výher dle rolí:
  - Bandité: 9493
  - Šerif a Pomocníci šerifa: 7414
  - Odpadlík: 1150
- Velikost dat: 3998903 řádků
- Rozdělení dat na trénovací a testovací pomocí funkce `sklearn.model_selection.train_test_split`:
  - `test_size`: 0,2
  - `random_state`: 1
- Transformace dat dle Oddílu 5.1.4
- Parametry estimátorů:
  - `random_state`: 1
  - `n_jobs` (pokud možno): -1
  - Ostatní: výchozí hodnoty

Výsledky

Estimátor knihovny scikit-learn	$\mathbf{R^2}$	**MSE**
`DecisionTreeRegressor`	0,90	0,011
`HistGradientBoostingRegressor`	0,93	0,008
`BaggingRegressor`	0,95	0,006
`ExtraTreesRegressor`	0,95	0,006
`RandomForestRegressor`	0,95	0,006

Tabulka 5.4 Výsledky vybraných modelů trénovaných na větších datech

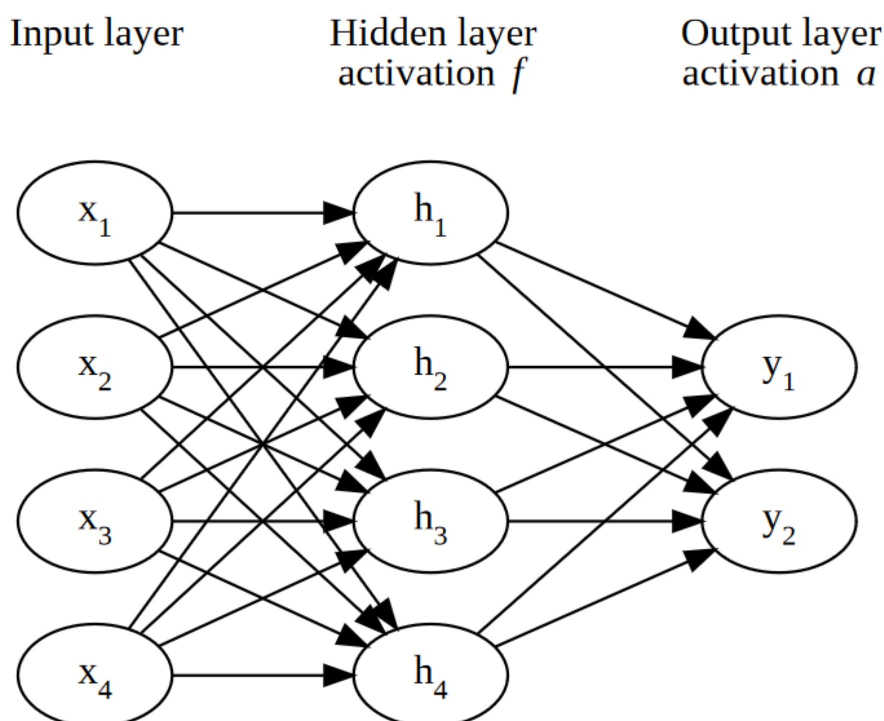
Znatelné zlepšení vykazovaly všechny testované estimátory (viz Tabulka 5.4). Jelikož dosáhly téměř maximálního R^2 skóre s téměř nulovým MSE, už jsme se dále nesnažili zlepšovat jejich trénování.

5.2 Neuroevoluce

Další metodou umělé inteligence, se kterou jsme experimentovali, byla evoluce neuronových sítí – **neuroevoluce**. V této podkapitole popíšeme, jak jsme zkoumali využití této metody pro umělou inteligenci hry `Bang!`.

Neuroevoluce nás zaujala, jelikož můžeme nechat umělé inteligence hrát proti sobě a tím je učit hrát `Bang!`. Na rozdíl od trénování našeho modelu strojového učení tedy nepotřebujeme žádné vzorové výstupy. Dále nás zaujalo, že můžeme trénovat neuronové sítě, aby ohodnocovovaly herní akce nikoliv takovým způsobem, aby přinesly nejlepší ohodnocení následujícího herního stavu, ale tak, aby vyhrávaly ve hře, což je primární cíl.

5.2.1 Základní pojmy



Obrázek 5.1 Dopředná neuronová síť, M. Straka [21]

V tomto oddíle shrneme základní pojmy z neuroevoluce potřebné k dalšímu porozumění textu.

Neuronové sítě jsou populární metodou strojového učení inspirovanou lidským mozkem. Skládají se z neuronů, které si předávají mezi sebou data. Tyto neurony lze dělit do tří vrstev – vstupní, skrytá a výstupní.

Existuje mnoho druhů neuronových sítí, my však pro představu uvedeme pouze jeden z nejznámějších druhů, a to **dopředné neuronové sítě**. Ty lze znázornit jako acyklický orientovaný graf (Obrázek 5.1), kde každá hrana má svoji **váhu** a každý vrchol reprezentuje **neuron**.

Neurony h_i ve skryté vrstvě počítáme následovně:

$$h_i = f(\sum_j x_j w_{j,i}^{(h)} + b_i^{(h)}),$$

přičemž:

- f je aktivační funkce skryté vrstvy (např. $\mathrm{ReLU}(x) = \max(0, x)$),
- x_j jsou vrcholy vstupní vrstvy,
- $w_{j,i}^{(h)}$ je váha hrany (x_j,h_i),
- $b_i^{(h)}$ je bias vrcholu h_i.

Neurony y_i ve výstupní vrstvě počítáme podobně:

$$y_i = a(\sum_j h_j w_{j,i}^{(y)} + b_i^{(y)}),$$

přičemž:

- a je aktivační funkce výstupní vrstvy,
- h_j jsou vrcholy vstupní vrstvy,
- $w_{j,i}^{(y)}$ je váha hrany (h_j,y_i),
- $b_i^{(y)}$ je bias vrcholu y_i.

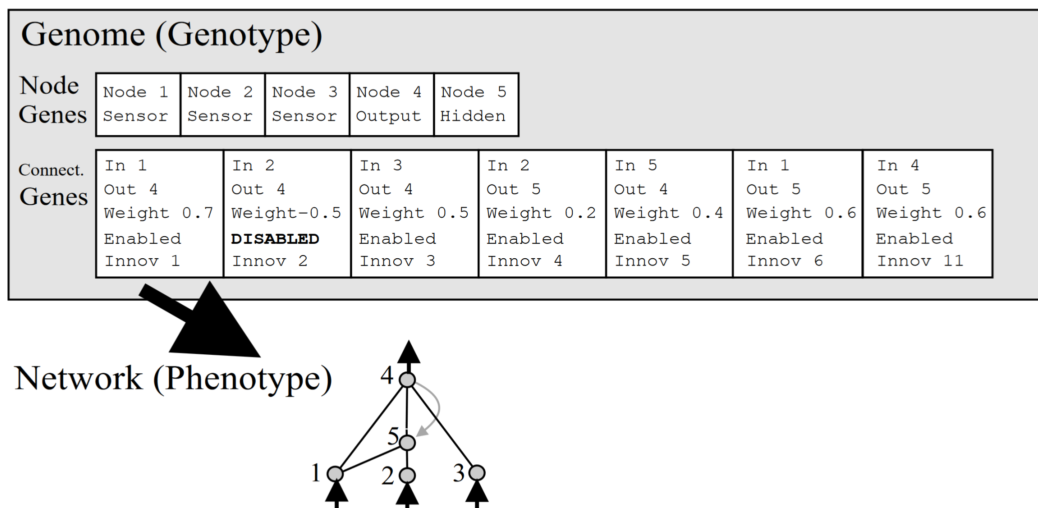
Mezi známé aktivační funkce výstupní vrstvy patří např.:

- identita (pro regresi)
- $\sigma(x) = \dfrac{1}{1 + e^{-x}}$ (modeluje Bernoulliho distribuci, pro binární klasifikaci)
- $\mathrm{softmax}(x_i) = \dfrac{e^{x_i}}{\sum_j e^{x_j}}$, kde x je vektor (pro obecnou klasifikaci)

Neuronové sítě typicky trénujeme gradientní metodou, avšak v **neuroevoluci** je trénujeme pomocí evolučních algoritmů. **Neuroevoluční algoritmy** se obecně zaměřují na evoluci vah neuronových sítí, nebo jejich topologie, či obojího najednou.

5.2.2 NEAT

NEAT, neboli NeuroEvolution of Augmenting Topologies (K. O. Stanley a R. Miikkulainen [22]), je neuroevoluční algoritmus zaměřující se jak na evoluci vah neuronových sítí, tak i jejich topologii. Byl úspěšně použit k natrénování umělé inteligence mnoha her, jako jsou např. Mario, Flappy Bird či Pac-Man. Pokusili jsme se tedy využít tohoto algoritmu při zkoumání umělé inteligence pro hru `Bang!`. V tomto oddíle shrneme jeho hlavní koncepty potřebné k dalšímu porozumění textu.



Obrázek 5.2 Mapování NEAT genomu na neuronovou síť, K. O. Stanley a R. Miikkulainen [22]

Jedinci

Jedinci v NEAT jsou neuronové sítě, které reprezentujeme pomocí genomů. **Genom** popisuje topologii i váhy dané neuronové sítě pomocí seznamu **neuronových genů** (node genes), reprezentujících neurony, a seznamu **spojových genů** (connection genes), reprezentujících hrany vedoucí mezi neurony (Obrázek 5.2). O každém neuronu zaznamenává, do jaké vrstvy neuronové sítě patří, a o každém spojení zaznamenává:

- Z jakého neuronu vede.
- Do jakého neuronu vede.
- Váhu.
- Zda je či není aktivní.
- Inovační číslo.

Inovační číslo slouží jako unikátní ID a reprezentuje chronologické pořadí, kdy byl daný gen vytvořen, čímž zaznamenává genetickou historii. Novým genům přiřazujeme zvyšující se inovační čísla.

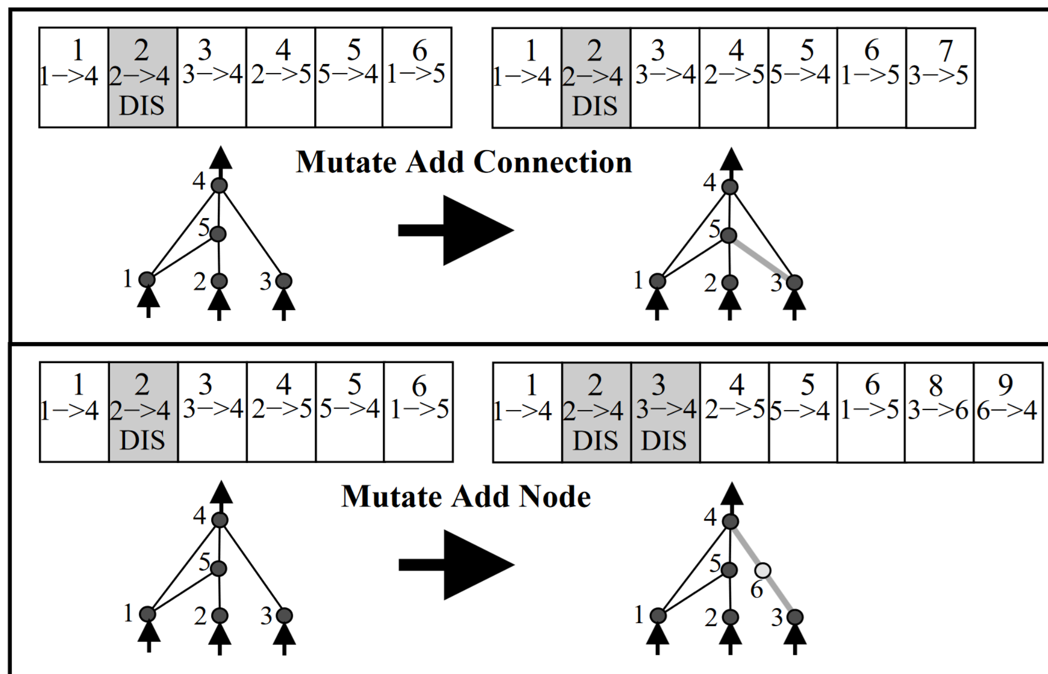
Inicializace algoritmu

Při inicializaci v NEAT začínáme s jednoduchými neuronovými sítěmi bez skrytých neuronů. Vstupní a výstupní neurony jsou tedy zpočátku spojeny přímo a algoritmus zvyšuje složitost sítě postupně.

Mutace a křížení

Po vyhodnocení fitness jedinců a selekci rodičů aplikujeme genetické operátory mutace a křížení na vybrané rodiče.

V NEAT využíváme dva druhy mutací (Obrázek 5.3):



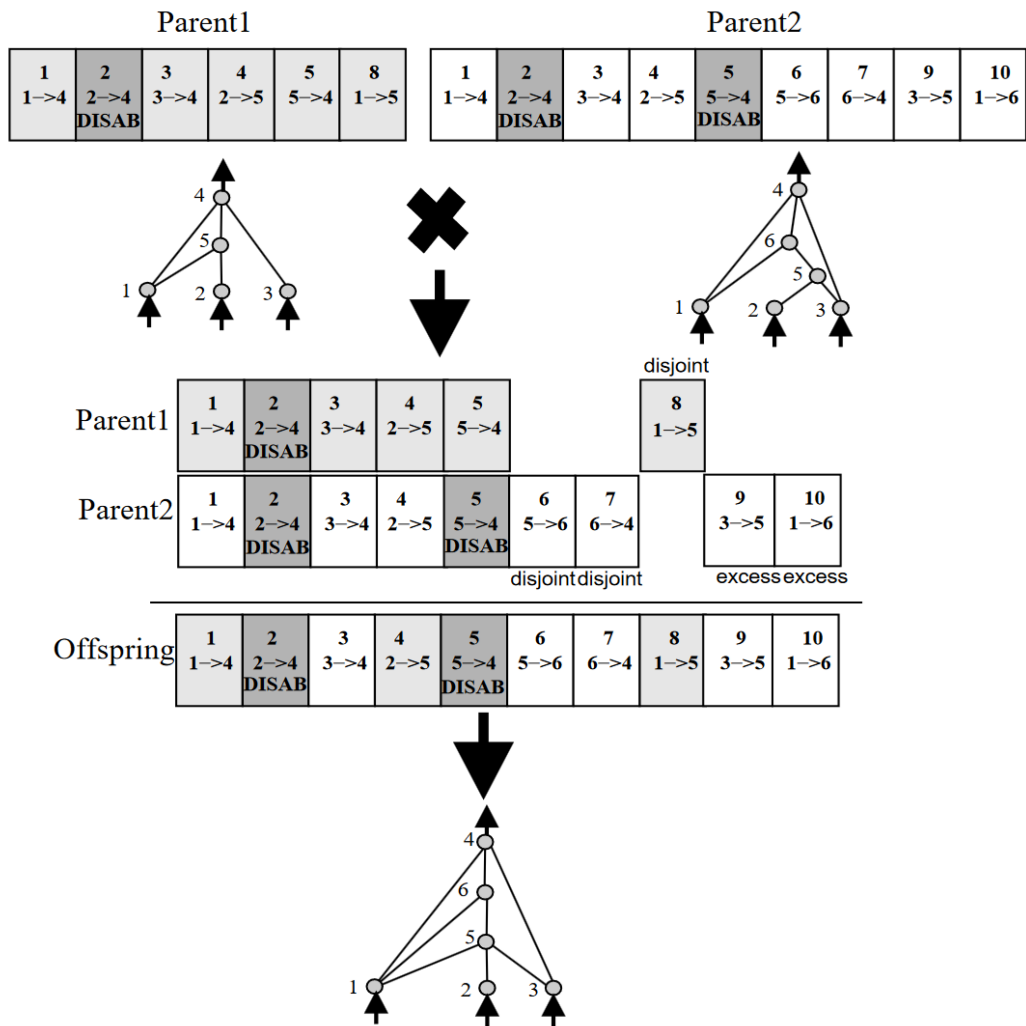
Obrázek 5.3 Dva typy mutací využívané v NEAT. Oba typy jsou ilustrovány se seznamem spojových genů genomu umístěným nad jeho odpovídající neuronovou sítí. Horní čísla v seznamu spojových genů představují inovační čísla jednotlivých spojení (K. O. Stanley a R. Miikkulainen [22]).

- Přidání spoje – mezi dvěma nespojenými neurony vytvoříme nový spojový gen s náhodnou váhou. Tento gen je přidán na konec seznamu spojových genů.
- Přidání neuronu – existující spojení rozdělíme na dvě přidáním nového neuronu. Původní spojení nastavíme jako neaktivní a přidáme do genomu dva nové spojové geny. Nové spojení vedoucí do nového neuronu dostane váhu jedna, přičemž nové spojení vedoucí z nového neuronu dostane váhu původního spojení.

Při křížení v NEAT se snažíme spojové geny rodičů spárovat pomocí inovačních čísel (Obrázek 5.4). Spojové geny, které se nachází v obou rodičích, nazýváme **společné** (matching) a potomci je dědí náhodně. Ostatní spojové geny, které se nachází pouze v jednom z rodičů, nazýváme **disjunktní** (disjoint) či **přebytečné** (excess) a dědí se právě tehdy, když má daný rodič větší fitness.

Dělení populace na druhy

V NEAT dále dělíme jednotlivce na **druhy** (species) dle jejich topologie, a to na základě počtu společných a nespolečných genů. Jednotlivci daného druhu následně **explicitně sdílí fitness** (explicit fitness sharing), tedy fitness daného jedince je vydělena počtem jedinců stejného druhu. Tím zachováváme genetickou rozmanitost, jelikož si daný druh jedinců nemůže dovolit být příliš početný, i přestože by vykazoval dobré výsledky. Také se tím nechává prostor k inovaci – nově vzniklé strategie mají čas k evoluci i v případě, že již existují nějací silní jedinci.



Obrázek 5.4 Párování genomů dvou různých síťových topologií pomocí inovačních čísel (K. O. Stanley a R. Miikkulainen [22]).

5.2.3 Trénování

V tomto oddíle se budeme věnovat tomu, jak jsme trénovali umělou inteligenci pro hru `Bang!` pomocí NEAT. Nejprve popíšeme, jak jsme navrhli jedince a jak jsme vyhodnocovali jejich fitness pomocí simulovaných her proti referenčním oponentům. Poté vysvětlíme, proč jsme zvolili knihovnu SharpNEAT jako implementaci NEAT, a jak jsme nastavili parametry algoritmu.

Jedinci

Jedinci v NEAT vždy představují neuronové sítě. Naším jediným úkolem z hlediska návrhu jedinců bylo tedy pouze navrhnout formát vstupních dat a výstup sítě.

Výstup sítě opět závisí na úloze, kterou bude řešit. První možností bylo jednoduše opět řešit výběr herní akce jako regresi, stejně jako u našeho modelu strojového učení (viz Oddíl 5.1.2). Druhou možností bylo řešit výběr herní akce jako klasifikaci, což by na rozdíl od trénovaní modelů strojového učení mělo jít přímočaře díky tomu, že nepotřebujeme v NEAT vzorové výstupy. Bylo by vhodné prozkoumat oba přístupy, avšak kvůli časovým omezením měli možnost prozkoumat pouze neuroevoluci k výběru nejlepší herní akce pomocí regrese. Vstupní data jsme tedy u NEAT jedinců přijímali ve stejném formátu, jako u modelů strojového učení (viz Oddíl 5.1.3).

Vyhodnocování fitness

Základní myšlenkou vyhodnocování fitness bylo nechat všechny jedince odehrát daný počet her proti nějakému referenčnímu oponentovi, a následně použít počet výher jako fitness. Otázkou však zůstává, jak zvolit referenčního oponenta.

První přímočarou možností bylo vyhodnocovat jedince proti náhodným hráčům. Zde je nevýhodou, že jelikož by se jedinci učili hrát proti oponentovi, který má neustále stejnou výkonnost, mohl by během evoluce nastat okamžik, kdy už se jedinci neučí nic nového, protože je oponent netlačí dostatečně ke zlepšování se. A navíc, jelikož náhodní hráči nehrají `Bang!` dobře, byl by strop dovedností nastaven nízko.

Nakonec jsme tedy zvolili možnost, kde je referenčním oponentem nejlepší jedinec z předešlé generace, čímž se jedinci vzájemně nutí neustále se zlepšovat (jako první referenční bod můžeme stále použít náhodné hráče). Jednou variantou je aktualizace referenčních oponentů např. každou generaci, či obecně po nějakém fixním počtu generací. Další variantou je aktualizovat referenční oponenty pouze tehdy, když jedinci při měření fitness překročí nějaký podíl výher z celkových odehraných her – tím bychom zajistili, že jedincům nebudeme zbytečně brzy zvyšovat laťku. Nakonec jsme otestovali obě zmíněné varianty.

Mohlo by se stát, že se jedinci naučí neustále přeskakovat své tahy, což by vedlo k nekončící hře, a vyhodnocování by se nikdy nezastavilo. Z toho důvodu jsme se rozhodli při vyhodnocování fitness omezit počet herních tahů pro jednu hru na 10000.

Dle vydavatelství hry `Bang!` totiž trvá jedna hra 20 až 90 minut – kdyby hráči prováděli jednu herní akci za sekundu, znamenalo by to až 5400 herních akcí za jednu hru. Usoudili jsme tedy, že omezení 10000 herních akcí za hru zcela jistě

nebude způsobovat příliš brzké zastavování běžných her a zároveň máme díky tomu jistotu, že se vyhodnocování vždy zastaví.

Univerzální a specializované neuronové sítě

Nejprve jsme zvažovali, zda využít výše popsaný návrh jedinců a vyhodnocování fitness k trénování univerzálních neuronových sítí, které by uměly ohodnocovat herní akce pro všechny role. Kvůli povaze hry, kde každá role vyžaduje odlišný styl hraní, nás však poté napadlo, že bychom mohli zkusit trénovat specializované neuronové sítě, které se zaměřují na ohodnocování herních akcí pro jednu konkrétní roli. Předpokládali jsme přitom, že specializace by mohla vést k lepším výsledkům, jelikož řeší menší oddělené problémy Nakonec jsme se rozhodli otestovat obě varianty.

SharpNEAT a jeho nastavení

Zkoumali jsme existující implementace algoritmu NEAT pro platformu .NET. Knihovna SharpNEAT [23] byla jediná z nalezených, která byla stále udržovaná a v době psaní práce prošla více než 20 lety aktivního vývoje. Dále jsme zvažovali, zda vytvořit vlastní implementaci algoritmu NEAT, ale jelikož by zcela jistě nedosahovala kvalit knihovny SharpNEAT, rozhodli jsme se použít právě tuto knihovnu.

SharpNEAT umožňuje přizpůsobení algoritmu pomocí různých parametrů. My jsme použili nastavevení z příkladu tvůrců knihovny pro řešení úlohy `PreyCapture`:

- `isAcyclic`: `false`
- `cyclesPerActivation`: 1
- `activationFnName`: `"LeakyReLU"`
- `evolutionAlgorithm`:
  - `speciesCount`: 15
  - `elitismProportion`: 0.66
  - `selectionProportion`: 0.66
  - `offspringAsexualProportion`: 0.5
  - `offspringSexualProportion`: 0.5
  - `interspeciesMatingProportion`: 0.01
- `reproductionAsexual`:
  - `connectionWeightMutationProbability`: 0.94
  - `addNodeMutationProbability`: 0.01
  - `addConnectionMutationProbability`: 0.025
  - `deleteConnectionMutationProbability`: 0.025
- `reproductionSexual`:

  - `secondaryParentGeneProbability`: 0.1

- `populationSize`: 225
- `initialInterconnectionsProportion`: 225
- `connectionWeightScale`: 225
- `complexityRegulationStrategy`:
  - `strategyName`: `"relative"`
  - `relativeComplexityCeiling`: 30
  - `minSimplifcationGenerations`: 10
- `degreeOfParallelism`: 3
- `enableHardwareAcceleratedNeuralNets`: `false`
- `enableHardwareAcceleratedActivationFunctions`: `false`

5.3 Porovnávání umělých inteligencí

V této podkapitole popíšeme experimenty s našimi umělými inteligencemi. Nechali jsme je hrát simulované hry s cílem pozorovat, jak dobře se naučily hrát `Bang!`. Implementaci všech porovnávacích experimemtů můžeme nalézt v přílohách práce v projektu `ExperimentsSolution/Experiment_AiVsAi`.

Ve všech měřeních výkonnosti využíváme záměrně stejný `seed`, aby měly testované umělé inteligence co nejpodobnější podmínky, a my tak mohli měřit jejich výkonnost férověji. Výsledky by tak měly lépe odrážet jejich schopnosti. Vstupní `seed` následně využíváme pro generování náhodných jevů každé hry, jako je generování dobíracího balíčku či vybírání herních akcí náhodných hráčů. Testované umělé inteligence vstupní seed tedy nijak neovlivňuje. Lepším řešením by bylo provést více měření s danou sadou počátečních `seeds`, avšak na tuto variantu nám již nezbyly prostředky.

5.3.1 Náhodní hráči vs. Náhodní hráči

Abychom zjistili, jak vypadá rozložení podílů výher různých rolí, když jsou všichni hráči na stejné úrovni, nechali jsme proti sobě hrát náhodné hráče.

Metoda

Parametry porovnávacího programu:

- Počet her pro každý počet hráčů: 250000
- Maximální počet herních akcí za hru: 10000
- Seed: 1234

Výsledky

Tým	Počet výher	Podíl výher
Bandité	109867	0,44
Šerif	118363	0,47
Odpadlík	21770	0,09

Tabulka 5.5 Výsledky z 250000 her čtyř náhodných hráčů.

Tým	Počet výher	Podíl výher
Bandité	126152	0,50
Šerif a Pomocník	107278	0,43
Odpadlík	16570	0,07

Tabulka 5.6 Výsledky z 250000 her pěti náhodných hráčů.

Tým	Počet výher	Podíl výher
Bandité	145663	0,58
Šerif a Pomocník	92667	0,37
Odpadlík	11670	0,05

Tabulka 5.7 Výsledky z 250000 her šesti náhodných hráčů.

Tým	Počet výher	Podíl výher
Bandité	154198	0,62
Šerif a Pomocníci	86693	0,35
Odpadlík	9109	0,04

Tabulka 5.8 Výsledky z 250000 her sedmi náhodných hráčů.

Tým	Počet výher	Podíl výher
Bandité	535880	0,54
Šerif a Pomocníci	405001	0,41
Odpadlík	59119	0,06

Tabulka 5.9 Celkové výsledky z 1000000 her čtyř až sedmi náhodných hráčů.

Diskuze

Celkové výsledky z Tabulky 5.9 ukazují, že role Banditů by měla být nejjednodušší, jelikož mají největší celkový podíl výher 0,54. Jako další s celkovým podílem výher 0,41 po nich následují Šerifové a Pomocníci, přičemž jako jednoznačně nejsložitější role se jeví Odpadlík s celkovým podílem výher 0,06.

5.3.2 Strojové učení vs. Náhodní hráči

Měřili jsme výkonnost modelů strojového učení trénovaných na velkých datech (viz Oddíl 5.1.9) ve hrách proti náhodným hráčům. Modely `ExtraTreesRegressor` a `RandomForestRegressor` jsme se rozhodli neotestovat, jelikož vykazovaly podobné výsledky v R^2 skóre i MSE jako `BaggingRegressor`, ale byly neprakticky komplexní.

Metoda

Sestavení her:

- 3 týmy: Šerif s Pomocníky, Bandité, Odpadlík
- Model strojového učení vždy hrál za jeden tým
- Ostatní týmy byly sestaveny z náhodných hráčů

Testované modely natrénované podle Oddílu 5.1.9:

- `HistGradientBoostingRegressor`
- `DecisionTreeRegressor`
- `BaggingRegressor`

Parametry programu měření výkonnosti:

- Seed: 500
- Počet her pro každý tým a počet hráčů: 100000
- Limit počet herních akcí za hru: 10000

Výsledky

Tým umělé inteligence	Počet výher	Podíl výher
`HistGradientBoostingRegressor`		
Bandité	52960	0,53
Šerif	44855	0,45
Odpadlík	9623	0,10
`DecisionTreeRegressor`		
Bandité	48726	0,49
Šerif	45970	0,46
Odpadlík	6109	0,06
`BaggingRegressor`		
Bandité	50928	0,51
Šerif	49749	0,50
Odpadlík	6097	0,06

Tabulka 5.10 Výsledky modelů trénovaných dle Oddílu 5.1.9 ve 100000 hrách čtyř hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`HistGradientBoostingRegressor`		
Bandité	62977	0,63
Šerif a Pomocník	59223	0,59
Odpadlík	8076	0,08
`DecisionTreeRegressor`		
Bandité	58791	0,59
Šerif a Pomocník	55391	0,55
Odpadlík	5018	0,05
`BaggingRegressor`		
Bandité	63666	0,64
Šerif a Pomocník	59695	0,60
Odpadlík	5133	0,05

Tabulka 5.11 Výsledky modelů trénovaných dle Oddílu 5.1.9 ve 100000 hrách pěti hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`HistGradientBoostingRegressor`		
Bandité	75093	0,75
Šerif a Pomocník	55888	0,56
Odpadlík	6661	0,07
`DecisionTreeRegressor`		
Bandité	67120	0,67
Šerif a Pomocník	52409	0,52
Odpadlík	4271	0,04
`BaggingRegressor`		
Bandité	70330	0,70
Šerif a Pomocník	56880	0,57
Odpadlík	4218	0,04

Tabulka 5.12 Výsledky modelů trénovaných dle Oddílu 5.1.9 ve 100000 hrách šesti hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`HistGradientBoostingRegressor`		
Bandité	79094	0,79
Šerif a Pomocníci	67329	0,67
Odpadlík	5771	0,06
`DecisionTreeRegressor`		
Bandité	73817	0,74
Šerif a Pomocníci	60561	0,61
Odpadlík	3684	0,04
`BaggingRegressor`		
Bandité	77874	0,78
Šerif a Pomocníci	62712	0,63
Odpadlík	3587	0,04

Tabulka 5.13 Výsledky modelů trénovaných dle Oddílu 5.1.9 ve 100000 hrách sedmi hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`HistGradientBoostingRegressor`		
Bandité	270124	0,68
Šerif a Pomocníci	227295	0,57
Odpadlík	30131	0,08
`DecisionTreeRegressor`		
Bandité	248454	0,62
Šerif a Pomocníci	214331	0,54
Odpadlík	19082	0,05
`BaggingRegressor`		
Bandité	262798	0,66
Šerif a Pomocníci	229036	0,57
Odpadlík	19035	0,05

Tabulka 5.14 Celkové výsledky modelů trénovaných dle Oddílu 5.1.9 ve 400000 hrách čtyř až sedmi hráčů za různé týmy.

Diskuze

V roli Banditů si modely vedly poměrně dobře – náhodné Bandity předčily ve všech případech a celkově měly podíl výher větší až o 0,14.

V týmu Šerifa a Pomocníků se modelům dařilo také relativně dobře. V celkovém podílu výher dosáhly zlepšení podílu výher až o 0,16 oproti náhodným hráčům. V případě her o čtyřech hráčích, kde měly podobnou úspěšnost jako náhodní hráči, se jim dařilo nejhůře, avšak ve hrách o pěti a více hráčů již vždy byly lepší, přičemž ve hrách o šesti a sedmi hráčích vykazovaly zlepšení podílu výher až o 0,32. To může značit, že modely neumí v tomto týmu hrát samy, ale dohromady jsou schopny spolupracovat a vyhrávat.

V roli Odpadlíka se modelům již nedařilo příliš dobře, což je nejspíše způsobeno tím, že role Odpadlíka je nejtěžší, jelikož nemá žádné spojence a musí ve hře přežít jako poslední. Jediný model, který dokázal v roli Odpadlíka předčit náhodné hráče, byl `HistGradientBoostingRegressor` s celkovým zlepšením podílu výher o 0,02. Ostatní modely v této roli vykazovaly naopak horší výkon než než náhodní hráči.

Celkově si nejlépe vedl model `HistGradientBoostingRegressor` (viz Tabulka 5.14), který také vykazoval výborné výsledky v našich zvolených metrikách (viz Tabulka 5.4), a jako jediný z našich modelů dokázal předčit náhodné hráče v roli Odpadlíka.

5.3.3 Univerzální NEAT vs. Náhodní hráči

Měřili jsme výkonnost dvou univerzálních neuronových sítí natrénovaných pomocí algoritmu NEAT (Oddíl 5.2.3) ve hrách proti náhodným hráčům. Jedna síť byla natrénovaná s pravidelnými aktualizacemi referenčních oponentů (`FixedStepBaselineUpdate`), druhá s aktualizacemi na základě podílu výher (`WinRateBaselineUpdate`). V obou případech jsme zvolili k testování nejlepšího jednotlivce z poslední generace.

Metoda

Nastavení trénování jedinců:

- Celkový počet generací: 1000
- Počet her na jedno vyhodnocení fitness: 50
- Limit počtu herních akcí za hru: 10000
- Počet generací pro aktualizaci referenčních oponentů: 20
- Podíl výher pro aktualizaci referenčních oponentů: 0,6
- Počet her na vyhodnocení fitness: 100

Sestavení her:

- 3 týmy: Šerif s Pomocníky, Bandité, Odpadlík
- Univerzální NEAT vždy hrál za jeden tým
- Ostatní týmy byly sestaveny z náhodných hráčů

Parametry programu měření výkonnosti:

- Seed: 500
- Počet her pro každý tým a počet hráčů: 100000
- Limit počtu herních akcí za hru: 10000

Výsledky

Tým umělé inteligence	Počet výher	Podíl výher
`FixedStepBaselineUpdate`		
Bandité	70972	0,71
Šerif	72862	0,73
Odpadlík	20344	0,20
`WinRateBaselineUpdate`		
Bandité	70972	0,71
Šerif	72862	0,73
Odpadlík	20344	0,20

Tabulka 5.15 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách čtyř hráčů za různé týmy.

Tým umělé inteligence	**Počet výher**	**Podíl výher**
`FixedStepBaselineUpdate`		
Bandité	74259	0,74
Šerif a Pomocník	58569	0,59
Odpadlík	15311	0,15
`WinRateBaselineUpdate`		
Bandité	74259	0,74
Šerif a Pomocník	58569	0,59
Odpadlík	15311	0,15

Tabulka 5.16 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách pěti hráčů za různé týmy.

Tým umělé inteligence	**Počet výher**	**Podíl výher**
`FixedStepBaselineUpdate`		
Bandité	81332	0,81
Šerif a Pomocník	54722	0,55
Odpadlík	12402	0,12
`WinRateBaselineUpdate`		
Bandité	81332	0,81
Šerif a Pomocník	54722	0,55
Odpadlík	12402	0,12

Tabulka 5.17 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách šesti hráčů za různé týmy.

Tým umělé inteligence	**Počet výher**	**Podíl výher**
`FixedStepBaselineUpdate`		
Bandité	83014	0,83
Šerif a Pomocníci	47416	0,47
Odpadlík	9892	0,10
`WinRateBaselineUpdate`		
Bandité	83014	0,83
Šerif a Pomocníci	47416	0,47
Odpadlík	9892	0,10

Tabulka 5.18 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách sedmi hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`FixedStepBaselineUpdate`		
Bandité	309577	0,77
Šerif a Pomocníci	233569	0,58
Odpadlík	57949	0,14
`WinRateBaselineUpdate`		
Bandité	309577	0,77
Šerif a Pomocníci	233569	0,58
Odpadlík	57949	0,14

Tabulka 5.19 Celkové výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 400000 hrách čtyř až sedmi hráčů za různé týmy.

Diskuze

Můžeme si na první pohled všimnout, že výsledky obou testovaných jedinců jsou naprosto totožné, což značí, že se naučili ohodnocovat herní akce stejným nebo podobným způsobem. Při dalším zkoumání těchto jedinců jsme zjistili, že mají pouze tři společné spojové geny a 22 nespolečných spojových genů. Je tedy možné, že právě tyto tři společné spojové geny vedly k podobnému ohodnocování herních akcí, a to i přesto, že měly zcela jiné váhy.

Jako Banditům se jedincům dařilo velmi dobře a v této roli získali podíl výher o 0,23 lepší než náhodní Bandité. Pozorovali jsme, že se zvyšuje podíl výher Banditů s počtem hráčů ve hře, přičemž nejlepšího podílu výher 0,83 dosahovali ve hrách o sedmi hráčích. To může značit, že jedinci umí v této roli spolupracovat, ale také to může být způsobeno tím, že Banditům obecně stoupá podíl výher s počtem hráčů (viz Oddíl 5.3.1).

V týmu Šerifa a Pomocníků jsme pozorovali zlepšení celkového podílu výher o 0,16, což je stejné jako u modelů strojového učení (viz Oddíl 5.3.2). Můžeme si ale všimnout, že podíl výher ve hrách čtyřech hráčů, což je jediná variace hry bez Pomocníků šerifa, činí 0,73, kdežto ve hrách sedmi hráčů pozorujeme pouze 0,47. To může být zapříčiněno tím, že hra v sedmi hráčích je více uzpůsobená Banditům (viz Tabulka 5.8), ale také to může značit, že se univerzální jedinci nenaučili v tomto týmu spolupracovat, a naopak preferují hrát sami.

V roli Odpadlíka jsme již pozorovali zlepšení v celkovém podílu výher, a to o 0,08. Jelikož náhodní hráči v této roli celkove vyhrávali pouze v 0,06 případů (viz Tabulka 5.9), jedná se o značné zlepšení.

5.3.4 Specializovaný NEAT vs. Náhodní hráči

Měřili jsme výkonnost specializovaných neuronových sítí natrénovaných pomocí algoritmu NEAT dle Oddílu 5.2.3 ve hrách proti náhodným hráčům. Podobně jako v Oddílu 5.3.3 jsme otestovali sítě natrénované s pravidelnými aktualizacemi referenčních oponentů (`FixedStepBaselineUpdate`) a s aktualizacemi na základě podílu výher (`WinRateBaselineUpdate`). V obou případech jsme rovněž zvolili k testování nejlepšího jednotlivce z poslední generace.

Metoda

Nastavení trénování jedinců:

- Celkový počet generací: 1000
- Počet her na jedno vyhodnocení fitness: 50
- Limit počtu herních akcí za hru: 10000
- Počet generací pro aktualizaci referenčních oponentů: 20
- Podíl výher pro aktualizaci referenčních oponentů (hodnoty jsme volili na základe výsledků náhodných her, viz Oddíl 5.9):
  - Bandité: 0,7
  - Šerif: 0,55
  - Pomocníci šerifa: 0,55
  - Odpadlík: 0,1

Sestavení her:

- 3 týmy: Šerif s Pomocníky, Bandité, Odpadlík
- Specializovaný NEAT vždy hrál za jeden tým
- Ostatní týmy byly sestaveny z náhodných hráčů

Parametry programu měření výkonnosti:

- Seed: 500
- Počet her pro každý tým a počet hráčů: 100000
- Limit počtu herních akcí za hru: 10000

Výsledky

Tým umělé inteligence	Počet výher	Podíl výher
`FixedStepBaselineUpdate`		
Bandité	70972	0,71
Šerif	77401	0,77
Odpadlík	20371	0,20
`WinRateBaselineUpdate`		
Bandité	70972	0,71
Šerif	72862	0,73
Odpadlík	22027	0,22

Tabulka 5.20 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách čtyř hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`FixedStepBaselineUpdate`		
Bandité	74259	0,74
Šerif a Pomocník	71419	0,71
Odpadlík	15522	0,16
`WinRateBaselineUpdate`		
Bandité	74259	0,74
Šerif a Pomocník	66856	0,67
Odpadlík	16672	0,17

Tabulka 5.21 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách pěti hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`FixedStepBaselineUpdate`		
Bandité	81332	0,81
Šerif a Pomocník	67591	0,68
Odpadlík	12282	0,12
`WinRateBaselineUpdate`		
Bandité	81332	0,81
Šerif a Pomocník	62556	0,63
Odpadlík	13559	0,14

Tabulka 5.22 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách šesti hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`FixedStepBaselineUpdate`		
Bandité	83014	0,83
Šerif a Pomocníci	63549	0,64
Odpadlík	9703	0,10
`WinRateBaselineUpdate`		
Bandité	83014	0,83
Šerif a Pomocníci	58377	0,58
Odpadlík	11159	0,11

Tabulka 5.23 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách sedmi hráčů za různé týmy.

Tým umělé inteligence	Počet výher	Podíl výher
`FixedStepBaselineUpdate`		
Bandité	309577	0,77
Šerif a Pomocníci	279960	0,70
Odpadlík	57878	0,14
`WinRateBaselineUpdate`		
Bandité	309577	0,77
Šerif a Pomocníci	260651	0,65
Odpadlík	63417	0,16

Tabulka 5.24 Celkové výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 400000 hrách čtyř až sedmi hráčů za různé týmy.

Diskuze

V roli Banditů jsou výsledky nejen identické pro oba testované jedince, ale také jsou naprosto stejné jako u univerzálních NEAT jedinců (viz Oddíl 5.3.3). Zkoumali jsme tedy jejich genomy pro Bandity a zjistili jsme, že mají dva společné spojové geny s různými váhami a 44 nespolečných spojových genů. Tento poměr společných a nespolečných genů může značit, že právě tyto dva společné spojové geny jsou pro strategii Banditů významné.

V týmu Šerifa a Pomocníků jedinci prokazovali oproti náhodným hráčům značné zlepšení celkového podílu výher až o 0,29. Jedinci v tomto týmu dokázali konzistentně vítězit ve většinách her, nehledě na počet hráčů. Dokázali tedy vítězit jak samostatně (hry čtyř hráčů), tak i dohromady (hry pěti a více hráčů), což může značit, že jedinci rozumí cíli této role a zároveň se naučili spolupracovat.

V roli Odpadlíka si jedinci vedli podobně jako univerzální NEAT jedinci (viz Oddíl 5.3.3).

Tento typ umělé inteligence je v naší práci jediný, který má dobré výsledky ve všech rolích hry a zároveň vykazuje schopnost hrát dobře samostatně i v týmu. Testované varianty aktualizování referenčních oponentů při trénování přitom neprojevují značné rozdíly mezi výkonnostmi jedinců ve hře.

5.3.5 Specializovaný NEAT vs. Člověk

Jelikož ze všech našich umělých inteligencí vykazovali specializovaní `FixedStepBaselineUpdate` NEAT jedinci (viz Oddíl 5.3.4) nejlepší výsledky při měření výkonnosti proti náhodným hráčům, rozhodli jsme se proti této umělé inteligence odehrát několik her.

Pro účely tohoto experimentu jsme v projektu `ConsoleAiClient` implementovali konzolovou aplikaci, která umožňuje uživateli připojit se na server našeho herního prostředí a následně zapojit umělou inteligenci do hry.

Metoda

Sestavení her:

- Jeden hráč: člověk

- Ostatní hráči: specializovaní `FixedStepBaselineUpdate` NEAT jedinci z Oddílu 5.3.4
- Role byly v každé hře přiřazeny náhodně.

Výsledky

Role člověka	**Počet her**	**Počet výher**	**Podíl výher**
Bandité	5	4	0,80
Šerif	1	1	1,00
Odpadlík	4	1	0,25

Tabulka 5.25 Výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách čtyř hráčů.

Role člověka	**Počet her**	**Počet výher**	**Podíl výher**
Bandité	6	3	0,50
Šerif	1	0	0
Pomocník šerifa	1	0	0
Odpadlík	2	0	0

Tabulka 5.26 Výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách pěti hráčů.

Role člověka	**Počet her**	**Počet výher**	**Podíl výher**
Bandité	4	2	0,5
Šerif	3	0	0
Pomocník šerifa	1	0	0
Odpadlík	2	0	0

Tabulka 5.27 Výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách šesti hráčů.

Role člověka	**Počet her**	**Počet výher**	**Podíl výher**
Bandité	6	4	0,67
Šerif	1	0	0
Pomocník šerifa	3	1	0,33
Odpadlík	1	0	0

Tabulka 5.28 Výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách sedmi hráčů.

Role člověka	Počet her	Počet výher	Podíl výher
Bandité	21	13	0,62
Šerif	6	1	0,17
Pomocník šerifa	5	1	0,20
Odpadlík	9	1	0,11

Tabulka 5.29 Celkové výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách čtyř až sedmi hráčů.

Diskuze

V roli Banditů jsme byli schopni poměrně konzistentně vyhrávat s celkovým podílem výher 0,62. V ostatních rolích jsme již naopak většinou prohrávali a ze všech her jsme byli schopni získat pouze jednu výhru za každou roli odlišnou od Banditů.

Jelikož jsme neměli příležitost odehrát s umělou inteligencí mnoho her, nejsou naše data příliš relevantní. Pozorovali jsme tedy způsob, jakým naše umělá inteligence hraje. Všimli jsme si, že vždy, když měla k dispozici nějaké modré karty, tak je použila. To by mohlo značit, že rozeznala, že bývá vhodné využívat modré karty, jelikož většinou s sebou nesou kladný účinek – z modrých karet v základním balíčku hry může mít negativní účinek pouze `Dynamit`. Na modré karty také většinou cílila v případech, kdy měla možnost je odebrat oponentovi pomocí karet `Cat Balou` či `Panika`, což může značit, že umělá inteligence rozeznává výhodu, jakou modré karty přináší oponentům. Dále většinou používala schopnost postavy, když to bylo možné, což bychom mohli interpretovat tak, že se naučila, že schopnosti postav přináší pouze kladné účinky.

Styl hraní naší umělé inteligence byl velmi agresivní – útočila při každé příležitosti a relativně často nás vyřazovala ze hry. V jednom případu se nám dokonce stalo, že jsme s nešťastnými kartami v ruce byli vyřazeni ještě před začátkem našeho prvního tahu, což je v lidských hrách nezvyklé. Tato agresivita umělé inteligence může být způsobená tím, že nijak neplánuje dopředu.

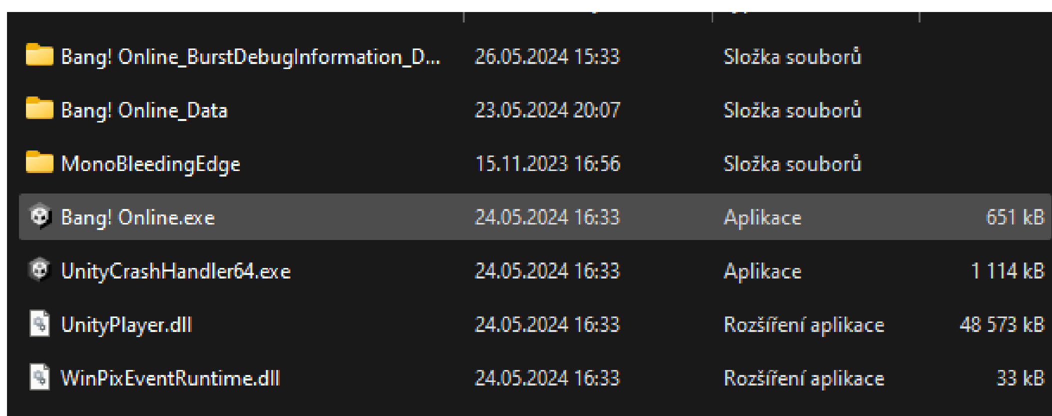
Ohledně odhadování rolí oponentů jsme pozorovali smíšené výsledky. Zaznamenali jsme kladné případy, kdy na nás Odpadlík či Pomocník šerifa cíleně útočil poté, co jsme v roli Banditů zaútočili na Šerifa. Dále jsme také nezaznamenali, že by Pomocník šerifa útočil na Šerifa. Na druhou stranu jsme pak mohli pozorovat, že spojenecký Bandita na nás útočil i přesto, že jsme ve hře zbývali již jen my s Šerifem, přičemž jsme na daného Banditu nijak v předchozích krocích neútočili. Z toho jsme usoudili, že naše umělá inteligence může mít obtíže s odhadováním rolí hráčů a volbou jednoduchých správných rozhodnutí.

Celkově naše umělá inteligence vykazovala nadějné výsledky ve hrách proti náhodným hráčům (viz Oddíl 5.3.4) a zároveň v pozorovaných hrách proti člověku ukázala, že se nejspíše zvládla naučit hlavní koncepty hry. Přesto však měla potíže nejen s odhadováním rolí hráčů, které je také obtížné pro lidské hráče, ale i s prováděním správných rozhodnutí v jednoduchých situacích.

6 Uživatelská dokumentace

Tato kapitola představuje uživatelskou dokumentaci k hernímu prostředí. Budeme ji představovat pomocí obrázků ilustrujících jednotlivé kroky hráče.

6.1 Spuštění hry



Obrázek 6.1 Spuštění hry

Hru zapneme spuštěním souboru `BANG! Online.exe`

6.2 Hlavní menu

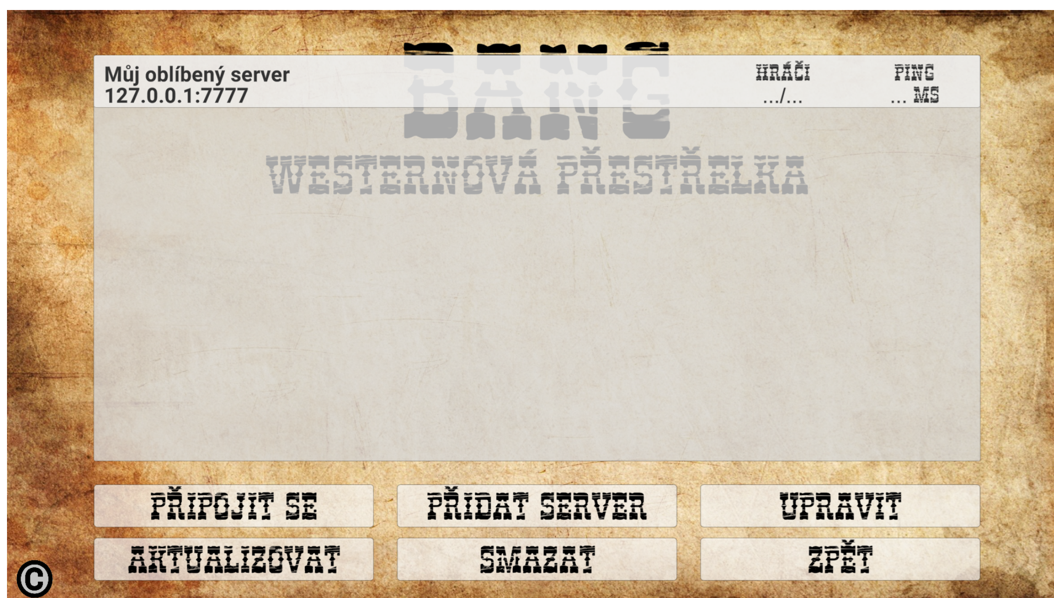
Obrázek 6.2 Hlavní menu

Jakmile se hra spustí, ocitneme se v hlavním menu hry.

- Pokud chceme začít hrát, stiskneme tlačítko `HRÁT`.

- Pokud chceme změnit svá nastavení, stiskneme tlačítko `NASTAVENÍ`.
- Pokud chceme hru vypnout, stiskneme tlačítko `EXIT`.
- Kliknutím na jednu z vlajek v pravém dolním rohu můžeme změnit jazyk hry.

6.3 Seznam uložených serverů



Obrázek 6.3 Seznam uložených serverů

Obrázek 6.4 Formulář uložení nového serveru

Když stiskneme v hlavním menu tlačítko `HRÁT`, budeme přesměrováni do seznamu serverů, kde jsou zobrazeny všechny naše uložené servery. Seznam serverů

je užitečný, jelikož jsme umožňujeme komukoliv založit si vlastní server. Tím pak může vzniknout celá komunitní síť serverů a hráči si mohou svobodně vybrat, na kterém hrát.

- Pokud chceme uložit nový server, stiskneme tlačítko `PŘIDAT SERVER`.
- Pokud chceme aktualizovat informace o stavu serverů, stiskneme tlačítko `PŘIDAT SERVER`.
- Pokud chceme upravit informace o uloženém serveru, nejprve vybereme server jedním kliknutím na něj, a poté stiskneme tlačítko `UPRAVIT`.
- Pokud se chceme připojit na server, můžeme to provést buďto dvojitým kliknutím na něj, anebo ho nejprve jedním kliknutím vybereme a poté stiskneme tlačítko `PŘIPOJIT SE`.

6.4 Seznam lobby

Obrázek 6.5 Seznam lobby

Jakmile se připojíme na server, ocitneme se v seznamu lobby. Zde jsou zobrazena všechna veřejná lobby na serveru. Máme možnost se připojit k již existujícímu lobby nebo vytvořit své vlastní.

Lobby slouží k organizaci her a plní funkci herní čekárny. Každé lobby má svého zakladatele, který může nejen měnit jeho nastavení, ale i spustit hru.

- Pokud si chceme vytvořit své vlastní lobby, stiskneme tlačítko `VYTVOŘIT LOBBY`
- Pokud se chceme připojit do jednoho ze zobrazených lobby, můžeme to provést buďto dvojitým kliknutím na něj, anebo nejprve jedním kliknutím vybereme a poté stiskneme tlačítko `PŘIPOJIT SE`.

- Pokud se chceme připojit k lobby pomocí jeho `ID`, stiskneme tlačítko `VYHLEDAT LOBBY`. Tímto se můžeme připojit i do soukromého lobby.

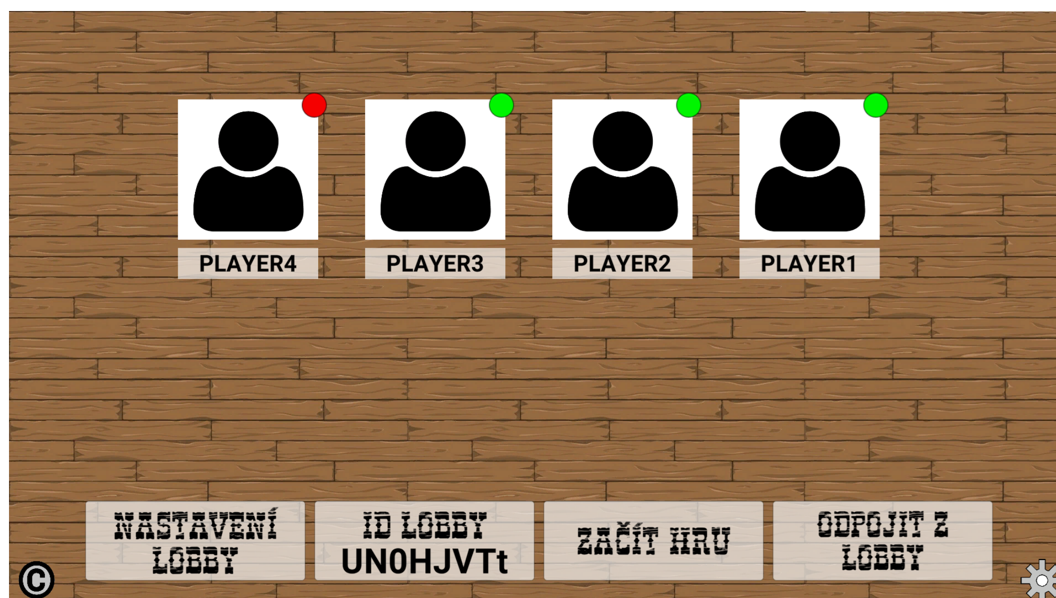
6.5 Vytvoření lobby

Obrázek 6.6 Formulář vytvoření lobby

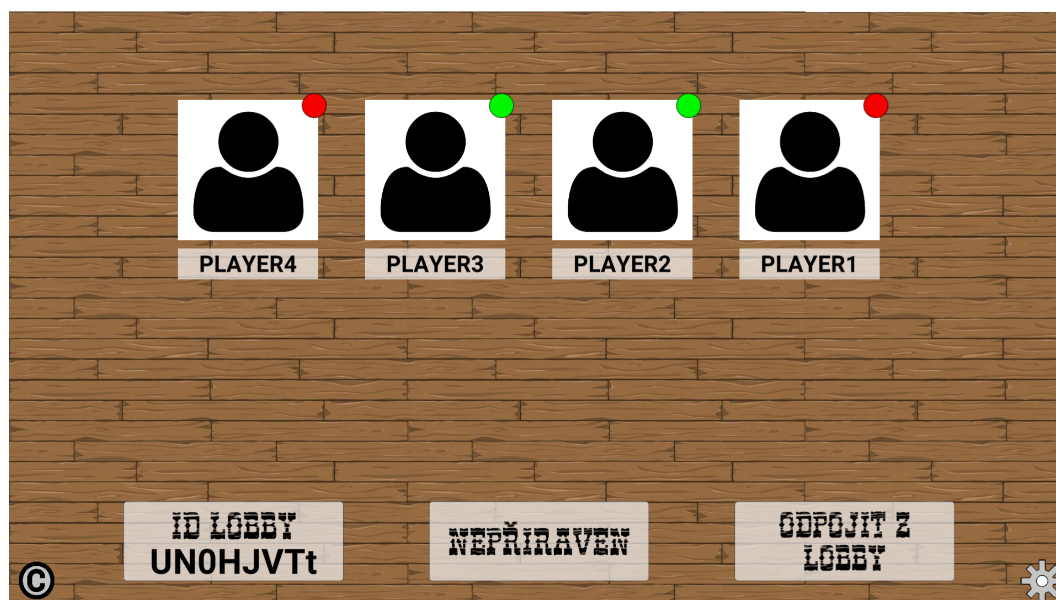
Při vytváření lobby můžeme nastavit jeho jméno, kapacitu a zda bude veřejné.

- Pokud chceme, aby naše lobby bylo veřejně viditelné v seznamu lobby, necháme zaškrtnuté pole `VEŘEJNÉ`.
- V opačném případě políčko nezaškrtáváme. Pak se do lobby mohou ostatní připojit pomocí unikátního ID lobby.

6.6 Lobby



Obrázek 6.7 Lobby z pohledu jeho tvůrce



Obrázek 6.8 Lobby z pohledu hráčů, kteří nejsou jeho tvůrci

Jakmile se připojíme do lobby, uvidíme všechny hráče, kteří se v něm nacházejí.

Ve spodní části obrazovky vidíme různá tlačítka. Tvůrce lobby bude mít k dispozici jiná tlačítka než ostatní hráči.

Možnosti všech hráčů

- Pokud se chceme odpojit z lobby, stiskneme tlačítko `ODPOJIT Z LOBBY`.
- Pokud chceme zkopírovat ID lobby, stiskneme tlačítko `ID LOBBY`.

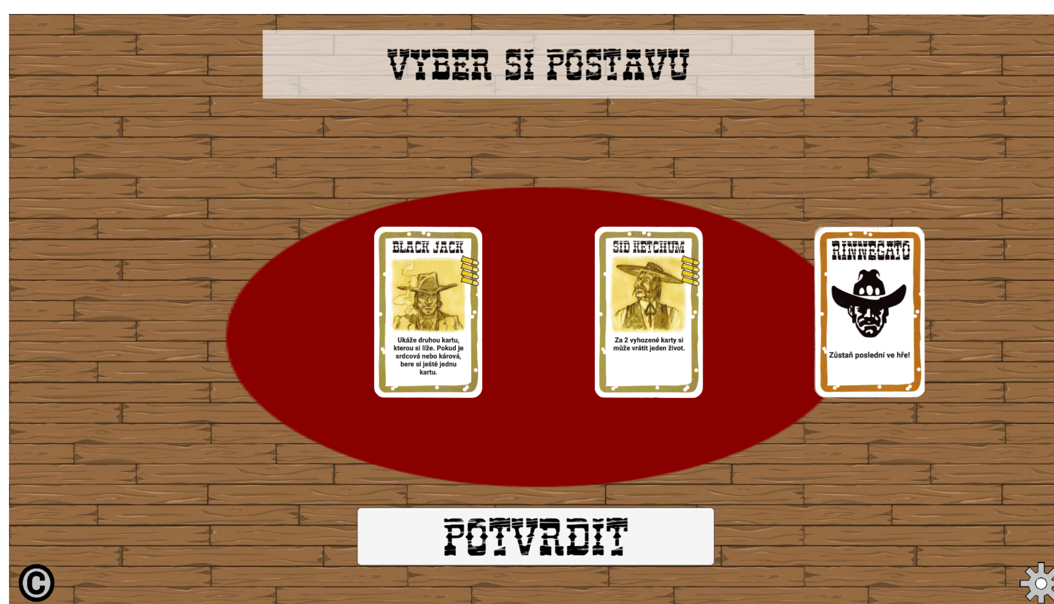
Možnosti tvůrce lobby

- Pokud chceme změnit nastavení lobby, stiskneme tlačítko `NASTAVENÍ LOBBY`.
- Pokud chceme začít hru, stiskneme tlačítko `ZAČÍT HRU`. Hra bude zahájena pouze tehdy, když jsou všichni ostatní hráči připraveni.

Možnosti hráčů, kteří nejsou tvůrci lobby

- Pokud se chceme označit jako připravení, resp. nepřipravení, stiskneme tlačítko `NEPŘIPRAVEN`, resp. `PŘIPRAVEN`.

6.7 Výběr postavy

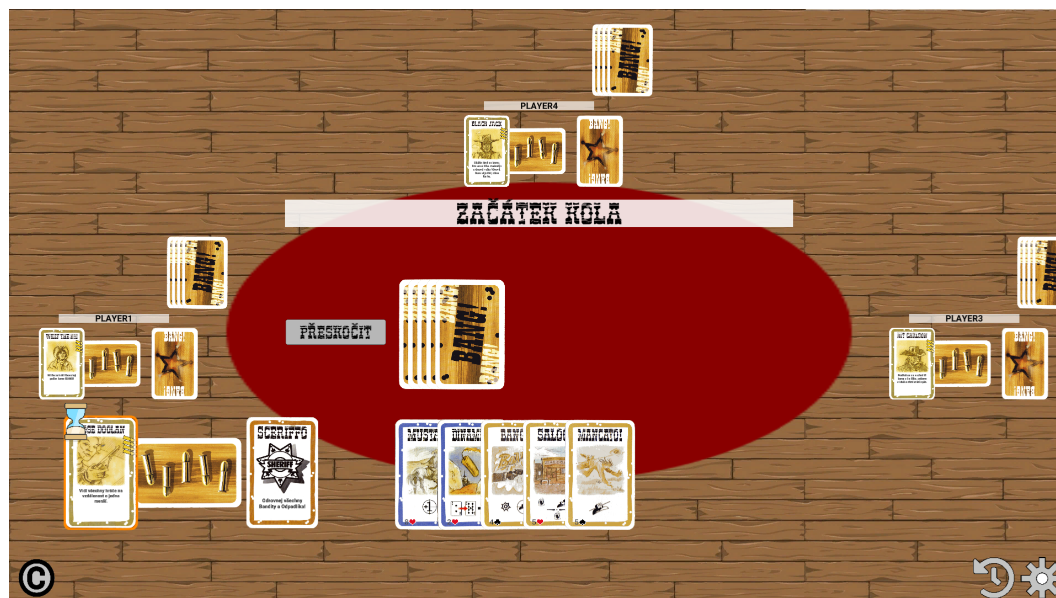


Obrázek 6.9 Výběr postavy

Když začne hra, zobrazí se nám role, která nám byla náhodně přidělena, a dostaneme na výběr ze dvou postav. Všichni hráči si musí jednu zvolit.

- Pokud chceme zvolit postavu, klikneme na ni a poté potvrdíme svou volbu stisknutím tlačítka `POTVRDIT`.

6.8 Herní pole



Obrázek 6.10 Herní pole

Jakmile si všichni hráči zvolí postavu, zobrazí se nám herní pole.

6.8.1 Ovládání hry

- Líznutí karty provedeme kliknutím klikneme na dobírací balíček ve středu herního pole.
- Zahrání karty z ruky provedeme, tak že ji stisknutím a podržením tlačítka myši přetáhneme do středu obrazovky a následně tlačítko myši pustíme.
- Použití karty vyložené před námi (např. `Barel`) provedeme kliknutím na danou kartu.
- Použití schopnosti postavy provedeme klikneme na naši postavu.
- Zvolení hráče jako cíl nějakého efektu karty či schopnosti postavy provedeme kliknutím na daného hráče.
- Ukončení, resp. přeskočení, tahu provedeme stisknutím tlačítka `UKONČIT TAH`, resp. `PŘESKOČIT`.
- Zobrazení historie herních informací provedeme stisknutím ikonky hodin v pravém dolním rohu obrazovky.
- Otevření menu pro nastavení hry provedeme stisknutím ikonky ozubeného kolečka v pravém dolním rohu obrazovky.
- Zvětšení nějaké karty, postavy, či role, provedeme podržením kurzoru myši nad daným objektem.

- Zobrazení počtu karet v ruce cizího hráče provedeme podržením kurzoru myši nad jeho kartami v ruce (jsou otočeny rubem nahoru).

7 Vývojová dokumentace

Tato kapitola představuje vývojovou dokumentaci k hernímu prostředí. Zaměříme se nejen na důležité implementační detaily potřebné k dalšímu rozšiřování, ale také na informace, které nelze snadno zjistit ze zdrojových kódů, jako je např. zdůvodnění použití určité architektury či knihovny. Záměrem kapitoly je pomoct případným novým vývojářům zorientovat se v projektu i bez přečtení celé naší práce. Z tohoto důvodu se mohou opakovat některé koncepty či obrázky z Kapitoly 4.

7.1 Celkový přehled

Herní prostředí pro hru `Bang!` umožňující hru více hráčů po síti jsme vyvíjeli pomocí herního engine Unity. V této podkapitole dále poskytneme celkový přehled tohoto projektu.

7.1.1 Organizace projektu

Jelikož jsme pro vývoj zvolili herní engine Unity, tak náš projekt organizujeme do adresářů dle zvyklostí a doporučení tohoto engine. Všechny adresáře však mají samodokumentující názvy, a tak vyjmenujeme pouze hlavní z nich:

- `Scripts` obsahuje veškerý zdrojový kód a dělíme ho do dalších adresářů dle `assemblies`, kterým se budeme věnovat později.
- `Scenes` obsahuje všechny scény projektu. Jedna scéna vždy reprezentuje jednu herní obrazovku a přechodům mezi nimi se budeme také věnovat později.

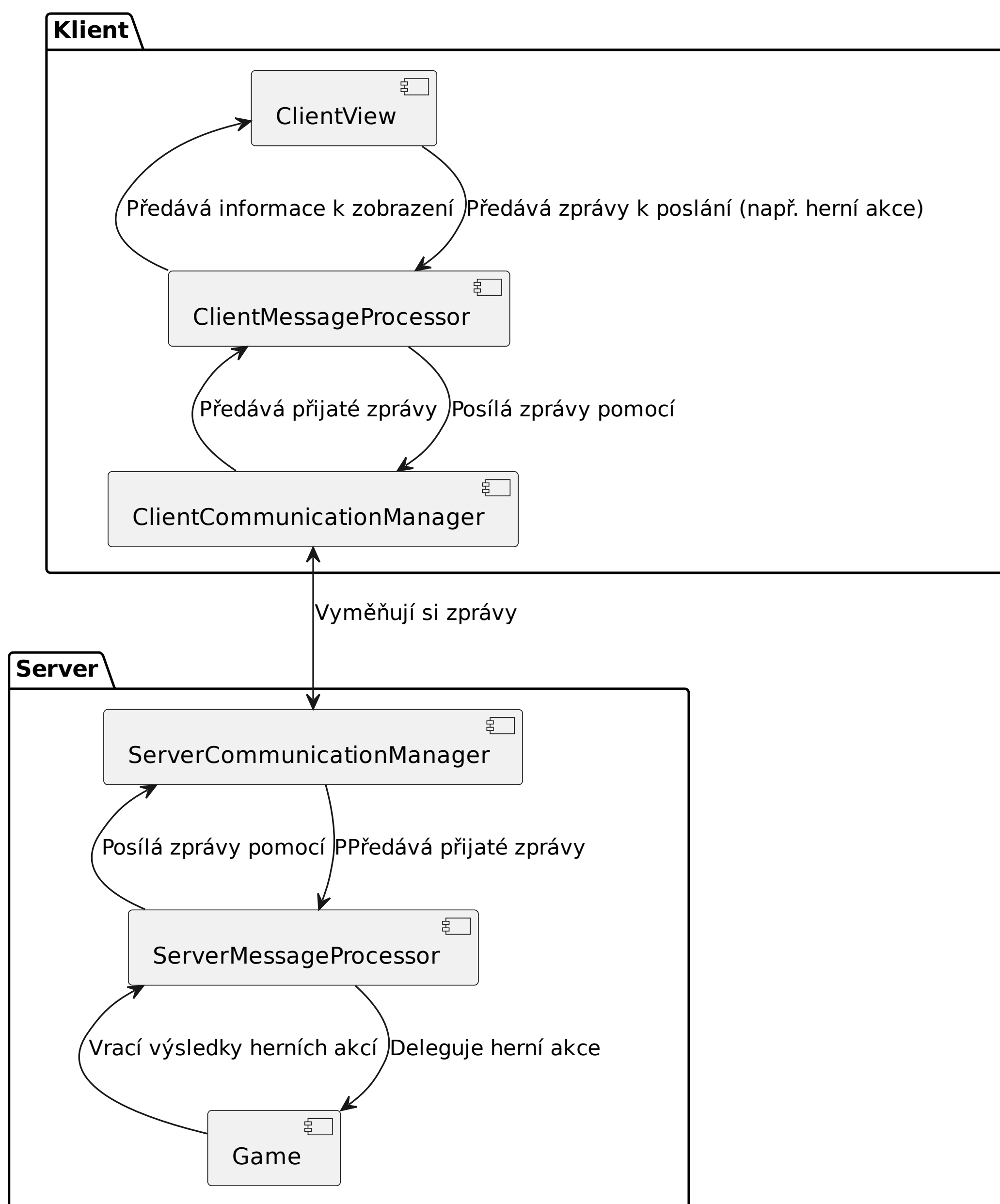
7.1.2 Assemblies

Náš projekt jsme rozdělili do třech `assemblies`, které obsahují dohromady přibližně 250 tříd. My pro každou `assembly` nyní pouze shrneme její hlavní obsah:

- `Bang.Common` obsahuje zdrojový kód, který je sdílený mezi klientem a serverem. Zejména pak definuje rozhraní, která musí klient či server splňovat, jako je např. `IClientMessageProcessor` či `IServerMessageProcessor`. Také definuje zprávy `NetworkMessage`, které si mezi sebou klient a server posílají. Dále obsahuje třídy herní logiky, jelikož k nim přistupuje jak server, tak klient – zde jsme si vědomi, že by bylo vhodné rozdělit herní logiku na datové třídy a na třídy, které implementují pravidla, avšak kvůli nedostatku času jsme zvolili tuto variantu.
- `Bang.Client` obsahuje převážně skripty uživatelského rozhraní (`namespace Bang.Client.Scenes`) a implementuje zpracování zpráv na straně klienta (třída `ClientMessageProcessor`).
- `Bang.Server` implementuje zpracování zpráv na straně serveru (třída `Server-MessageProcessor`).

- `Bang.Ai` ve třídě `BangAi` definuje společné rozhraní pro umělé inteligence hry `Bang!`. Také poskytuje užitečné funkcionality, jako je hledání možných herních tahů.

7.1.3 Hlavní komponenty



Obrázek 7.1 Hlavní komponenty herního prostředí

V této podkapitole popíšeme dekompozici našeho herního prostředí na hlavní kontejnery a komponenty, jejichž bližšímu popisu se budeme věnovat později.

V našem projektu můžeme server a klienta rozdělit na dva různé kontejnery, které se dále dělí do komponent dle Obrázku 7.1. Server a klient mezi sebou komunikují pomocí svých správců komunikace (`ServerCommunicationManager` a `ClientCommunicationManager`), kteří dále předávají veškeré přijaté zprávy

komponentám pro zpracování zpráv (`ServerMessageProcessor` a `ClientMessageProcessor`).

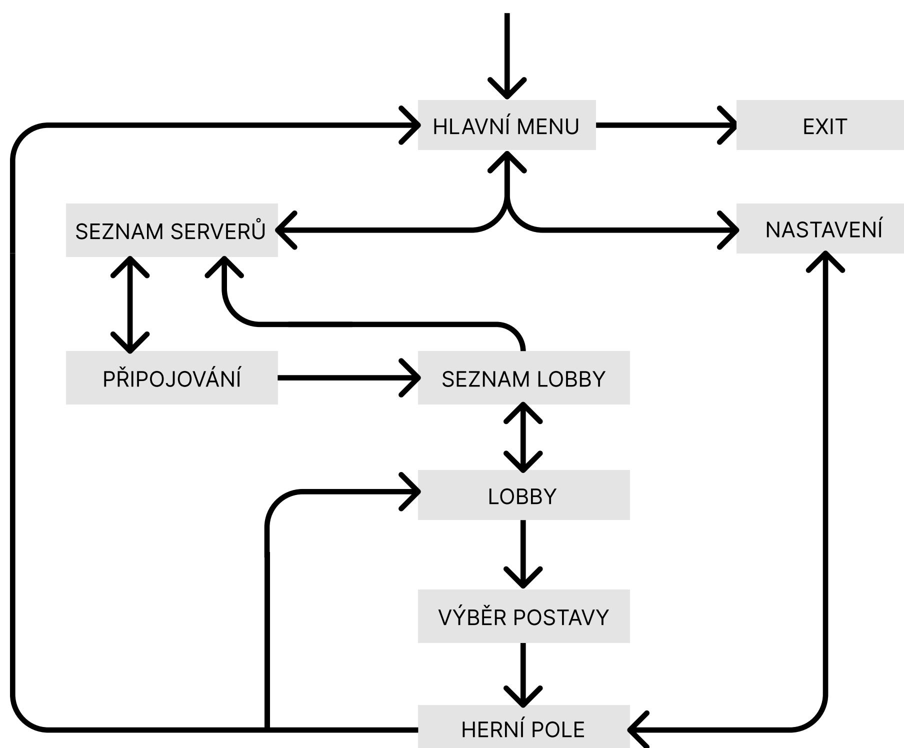
Na straně serveru jsou zprávy reprezentující herní akce ještě dále předávány herní logice (`Game`). Opačným směrem pak herní logika předává výsledky herních akcí zpět komponentě pro zpracování zpráv.

Na straně klienta jsou zprávy předávány komponentě pro zobrazování uživatelské rozhraní (`ClientView`). Opačným směrem může uživatelské rozhraní předávat zprávy reprezentující akce klienta (např. zahrání karty).

7.1.4 Kompilace

Unity poskytuje program Unity Hub [24], který slouží k pohodlné správě projektů vyvíjených právě v tomto herním engine. Náš projekt tedy můžeme načíst v Unity Hub, který automaticky detekuje verzi Unity, v níž byl projekt vyvíjen. Pokud daná verze Unity ještě není nainstalovaná, můžeme instalaci provést přímočaře v Unity Hub. Následně již můžeme otevřít náš projekt pomocí tohoto programu, který se postará o kompilaci.

7.2 Uživatelské rozhraní



Obrázek 7.2 Přechody mezi obrazovkami herního prostředí.

Uživatelské rozhraní jsme rozdělili do několika scén, přičemž každá reprezentuje jednu herní obrazovku, ve které se může hráč ocitnout (viz Obrázek 7.2). Dohromady máme tedy v herním prostředí devět scén, z nichž jedna scéna je implementační detail serveru běžícího v Unity.

7.3 Síťování

V této podkapitole zaměříme na to, jak jsme implementovali síťování hry. Zvolili jsme architekturu autoritativního serveru, jelikož jsme nechtěli, aby měli hráči přístup ke skrytým herním informacím, jako jsou např. role oponentů.

7.3.1 WatsonTcp

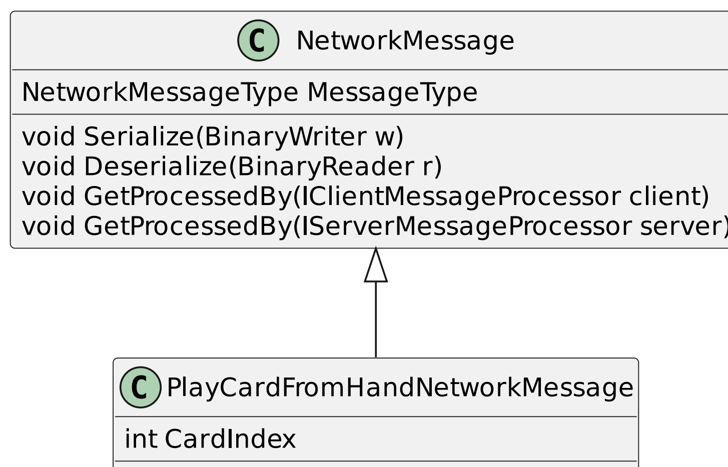
Ve hře `Bang!` záleží na pořadí herních akcí a zároveň potřebujeme garanci, že herní akce klientů vždy dorazí na server. To znamená, že potřebujeme spojovanou a spolehlivou komunikaci, což splňuje TCP.

Rozhodli jsme se pro síťování použít knihovnu WatsonTcp, jelikož vnitřně používá TCP a svým uživatelům poskytuje pohodlnou abstrakci při zachování flexibility.

7.3.2 BangTcpClient a BangTcpServer

Třídy `BangTcpClient` and `BangTcpServer` jsou určeny ke komunikaci mezi klientem a serverem naší hry a vnitřně používají již zmíněnou knihovnu WatsonTcp. Jejich rozhraní poskytují běžné funkcionality jako je připojování, odpojování, či odesílání a příjem zpráv. Je možné je využít i mimo prostředí Unity, čímž umožňujeme větší flexibilitu pro další vývoj (např. konzolová aplikace, která přesměrovává všechny požadavky o herní akce na umělou inteligenci).

7.3.3 NetworkMessage



Obrázek 7.3 Diagram třídy `NetworkMessage`.

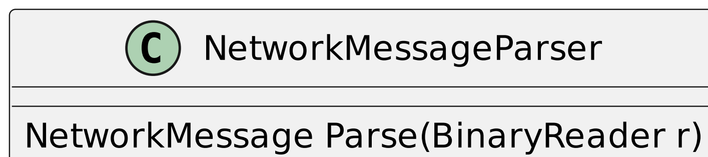
Zprávy, které si mezi sebou posílají klient a server, reprezentujeme jako objekty. `NetworkMessage` je abstraktní třída, od které dědí všechny typy našich zpráv (viz Obrázek 7.3). Poskytuje společné rozhraní pro serializaci a zpracovávání, přičemž virtuálními metodami přenechává dědícím třídám flexibilitu serializace vlastních dat. Metody `GetProcessedBy` slouží k implementaci návrhového vzoru Visitor (každá zpráva vyžaduje jiné zpracovávání).

Čtení zpráv pomocí reflexe

K pohodlnému čtení zpráv využíváme reflexi. Nejprve pomocí ní budujeme slovník, jehož klíče jsou typu `NetworkMessageType` a hodnoty jsou metody generující zprávu daného typu:

1. Najdeme všechny typy dědící od `NetworkMessage`, které nejsou `abstract` ani `interface`.
2. Z informací o daném typu najdeme jeho prázdný konstruktor.
3. Nalezený prázdný konstruktor využijeme k vytvoření instance daného typu.
4. Z vytvořené instance můžeme zjistit hodnotu jejího `MessageType`.
5. Zjištěnou hodnotu `MessageType` již můžeme použít jako klíč slovníku, a jako odpovídající hodnotu můžeme použít lambda metodu generující zprávu daného typu.

Při čtení zpráv následně stačí pouze přečíst typ zprávy (`NetworkMessageType`) a použít ho k nalezení metody generující zprávu správného typu.



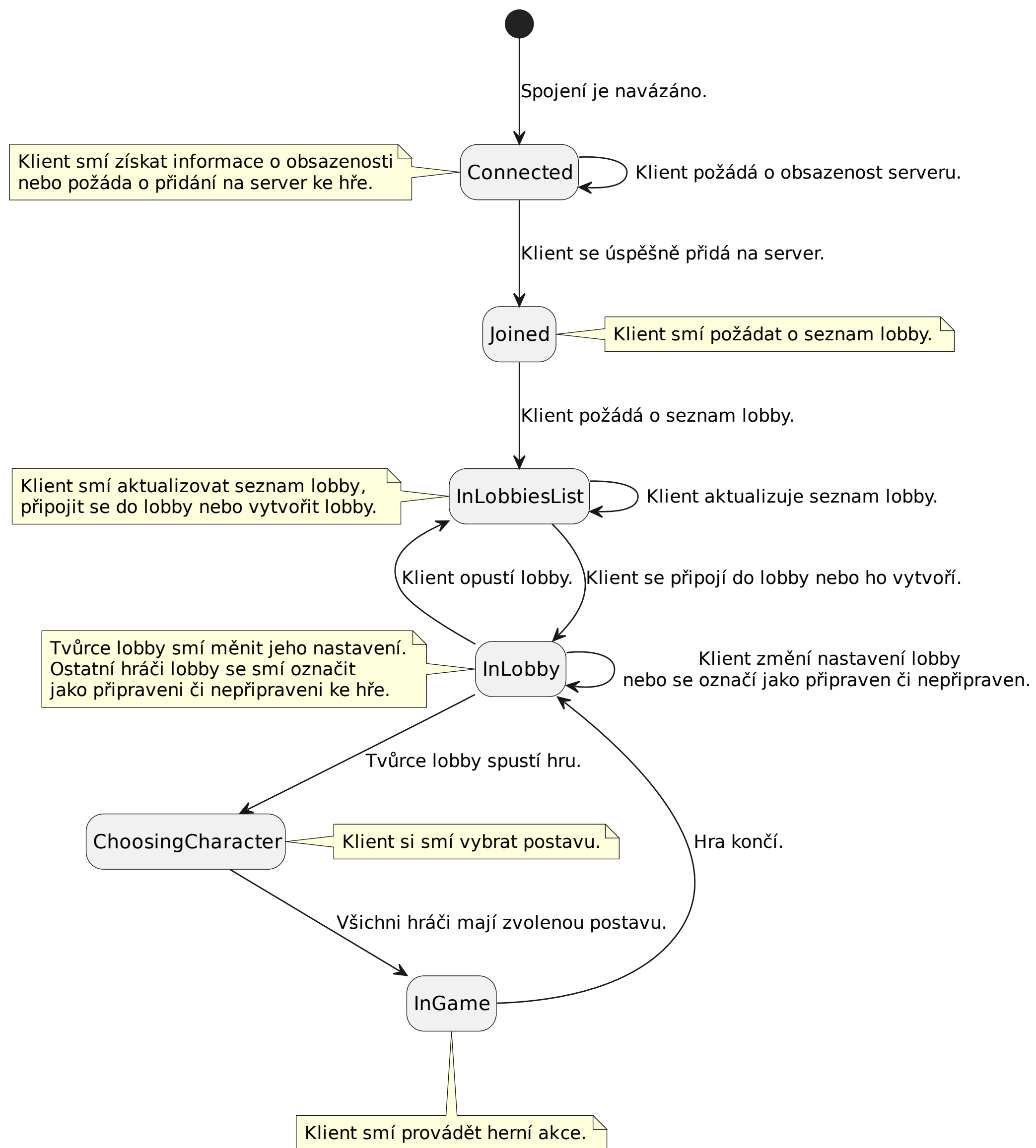
Obrázek 7.4 Diagram třídy `NetworkMessageParser`.

Třída `NetworkMessageParser` (Obrázek 7.4) poskytuje pohodlné rozhraní pro čtení zpráv, přičemž vnitřně využívá reflexi, jak jsme již popsali výše. Implementuje návrhový vzor Singleton a slovník s generujícími metodami inicializuje pouze jednou, při prvním přístupu k instanci.

Použití reflexe může mít negativní účinky na výkonnost, avšak pro naše účely karetní hry, kdy nemáme vysoké nároky na rychlost síťové komunikace, je toto zpomalení zanedbatelné.

7.3.4 Stavy klientů

Server o každém klientovi udržuje stav, ve kterém se nachází. Obrázek 7.5 znázorňuje přechody mezi těmito stavy a jaké akce jsou povoleny v každém stavu.



Obrázek 7.5 Přechody mezi stavy klienta.

7.4 Herní logika

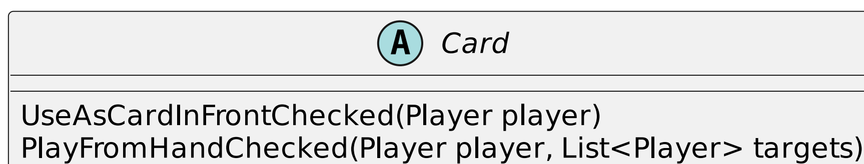
V této podkapitole popíšeme, jak jsme implementovali herní logiku. Nejprve se budeme věnovat reprezentaci herních informací, a následně shrneme, jak jsou implementovány speciální efekty karet a herních postav.

7.4.1 Třída Game

Třída `Game` představuje hlavní rozhraní k herní logice. Obsahuje herní informace a metody reprezentující herní akce, přičemž předpokládáme, že každá přijatá herní akce se vždy váže ke hráči, který je momentálně na tahu. Také se stará o šíření notifikací o různých herních událostech, což je potřebné k implementaci různých efektů.

Herní akce jsou ze třídy `Game` dále přesměrovávány herním ovladačům, které implementují specifické efekty karet a postav. Těm se ale budeme věnovat později.

7.4.2 Karty

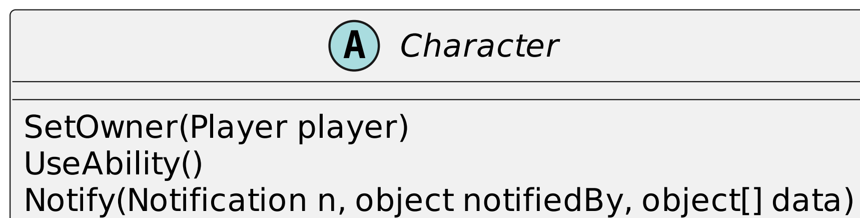


Obrázek 7.6 Diagram abstraktní třídy `Card`.

Všechny karty reprezentujeme jako objekty, jejichž typ dědí od abstraktní třídy `Card` (viz Obrázek 7.6). Tato třída poskytuje společné rozhraní pro zahrání karty z ruky nebo její použití jako aktivní vyložené karty.

Rozhodli jsme se rozhraní karet navrhnout tak, aby hráč, který danou kartu zahrál či použil, byl přijímán jako parametr v odpovídajících metodách. Druhou variantou bylo pro každou kartu zaznamenávat, kdo je její majitel, avšak to se nám zdálo jako zbytečně složité. Karty totiž potřebují znát své majitele pouze v moment, kdy je aktivován jejich efekt.

7.4.3 Postavy

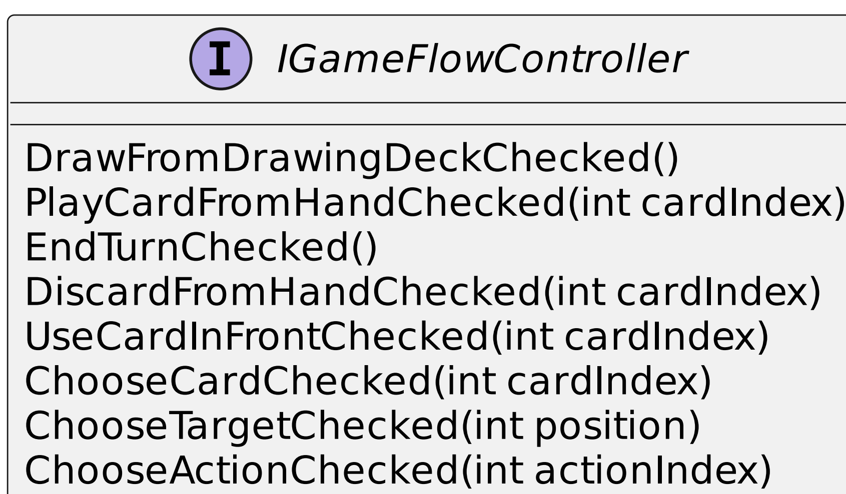


Obrázek 7.7 Diagram abstraktní třídy `Character`.

Všechny postavy reprezentujeme jako objekty, jejichž typ dědí od abstraktní třídy `Character` (viz Obrázek 7.7). Schopnosti postav představovaly hlavní problém, který jsme u jejich implementace museli řešit. Obecně můžeme schopnosti dělit následovně:

- Pasivní – jejich efekt působí celou hru, např. zvětšení vzdálenosti od ostatních hráčů od jedna. Aktivujeme je vždy před začátkem hry.
- Aktivované při událostech – jejich efekt je automaticky spuštěn při určité herní události. Pro tyto účely přijímají schopnosti notifikace.
- Aktivované hráčem – jejich efekt je spuštěn na základě herní akce použití schopnosti postavy.

7.4.4 Herní ovladače a efekty



Obrázek 7.8 Diagram rozhraní `IGameFlowController`.

Ve hře existuje mnoho efektů, které ovlivňují standardní průběh tahu hráče. Máme např. karty, které vyžadují reakce od ostatních hráčů, jako je `BANG!` či `Gatling`, či karty, které mohou přeskočit celý tah daného hráče, jako je `Vězení`. Různé efekty také mohou specificky ovlivňovat povolené herní akce, např. nemůžeme hrát karty, když je aktivní efekt `Vězení`. Jelikož by implementace veškerých těchto efektů v rámci třídy `Game` byla špatně rozšiřitelná, rozhodli jsme se, že každý efekt bude mít svůj ovladač hry, který zodpovídá za jeho správný průběh. Třída `Game` následně přijaté herní akce předává dále ke zpracováním aktuálnímu ovladači hry.

Rozhraní `IGameFlowController` (viz Obrázek 7.8) obsahuje metody pro každou herní akci, kterou může hráč provést. Každá taková metoda vrací výsledek kontroly pravidel příslušné akce. Kromě ukončení kola mají všechny metody reprezentující herní akce výchozí implementaci, která vrací výsledek pravidel reprezentující, že danou akci nelze v daný moment provést. Většina ovladačů definuje pouze malý počet herních akcí, a tak si výchozími implementacemi usnadníme jejich implementaci.

Ukončení kola je vždy definováno

Ukončení kola nemá výchozí implementaci, jelikož vyžadujeme po herních ovladačích, aby vždy tuto herní akci správně definovaly. Toho pak využíváme, když potřebujeme přeskočit kolo hráče, např. v případech, kdy hráč ztratí spojení.

Zanořování efektů

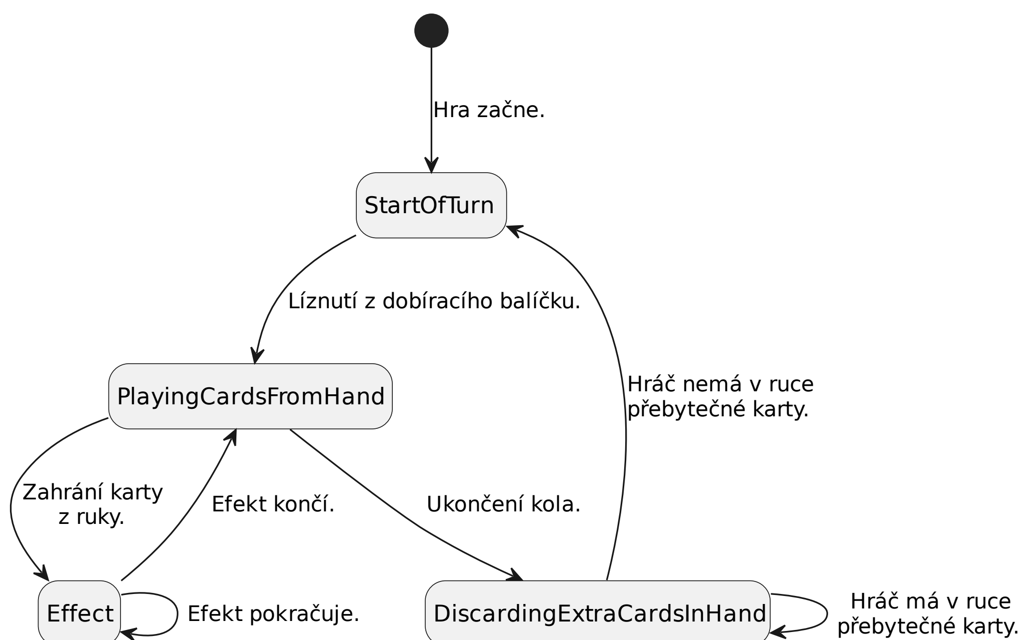
Ve hře se může stát, že se zanoří několik efektů za sebou, např.:

1. Hráč je cílem efektu karty `BANG!`.
2. Hráč použije kartu `Barel`.
3. Hráč použije schopnost postavy `Lucky Duke`.

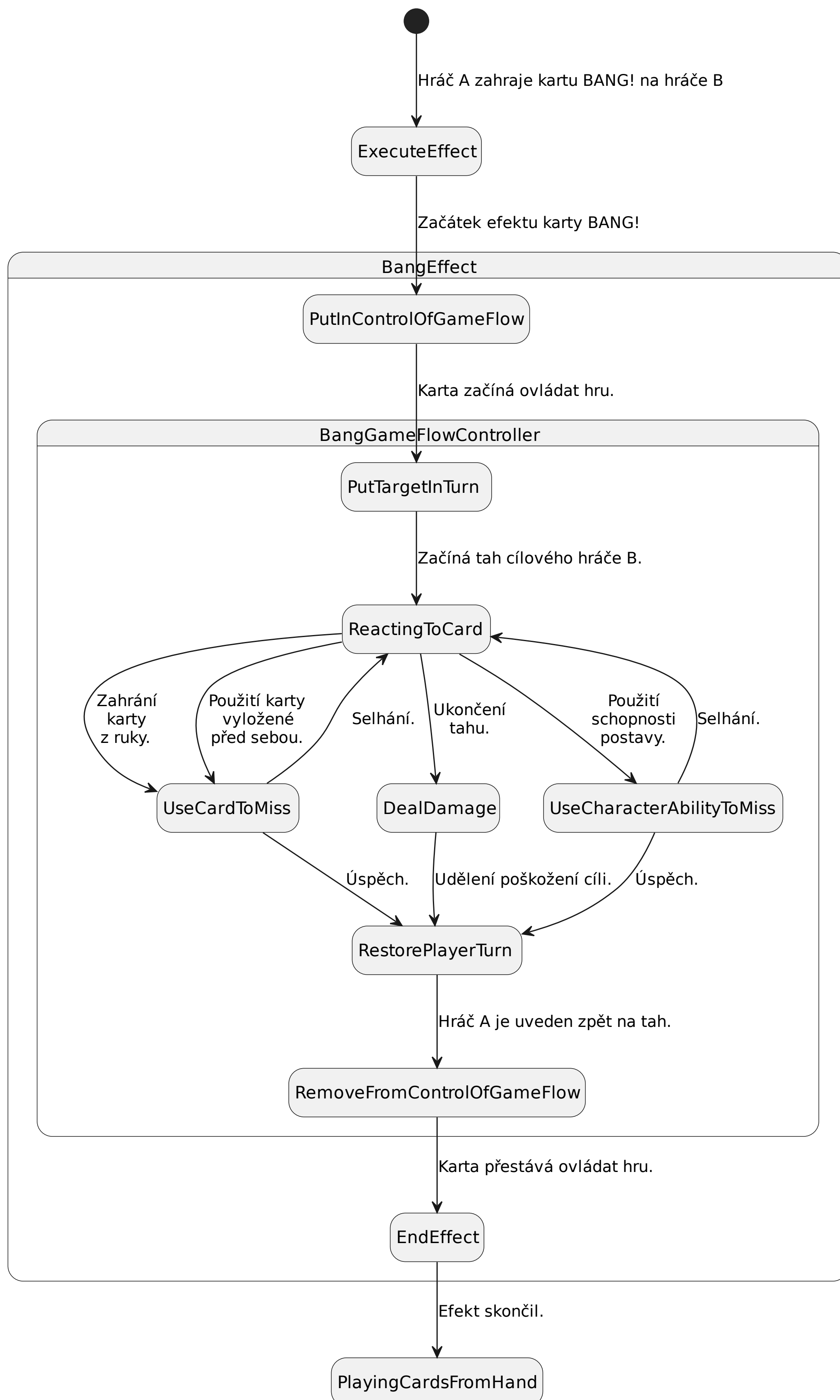
Můžeme si všimnout, že se efekty následně ruší v opačném pořadí. To nás přirozeně vedlo k použití zásobníku pro správu efektů. Jakmile aktuální herní ovladač přestane ovládat hru, předá kontrolu ovladači, který je v zásobníku před ním. Zanořené efekty mohou mít mezi sebou různé interakce. Jelikož tyto interakce nelze řešit obecným způsobem, rozhodli jsme se je přenechat na starost konkrétním efektům.

Příklady průběhů efektů

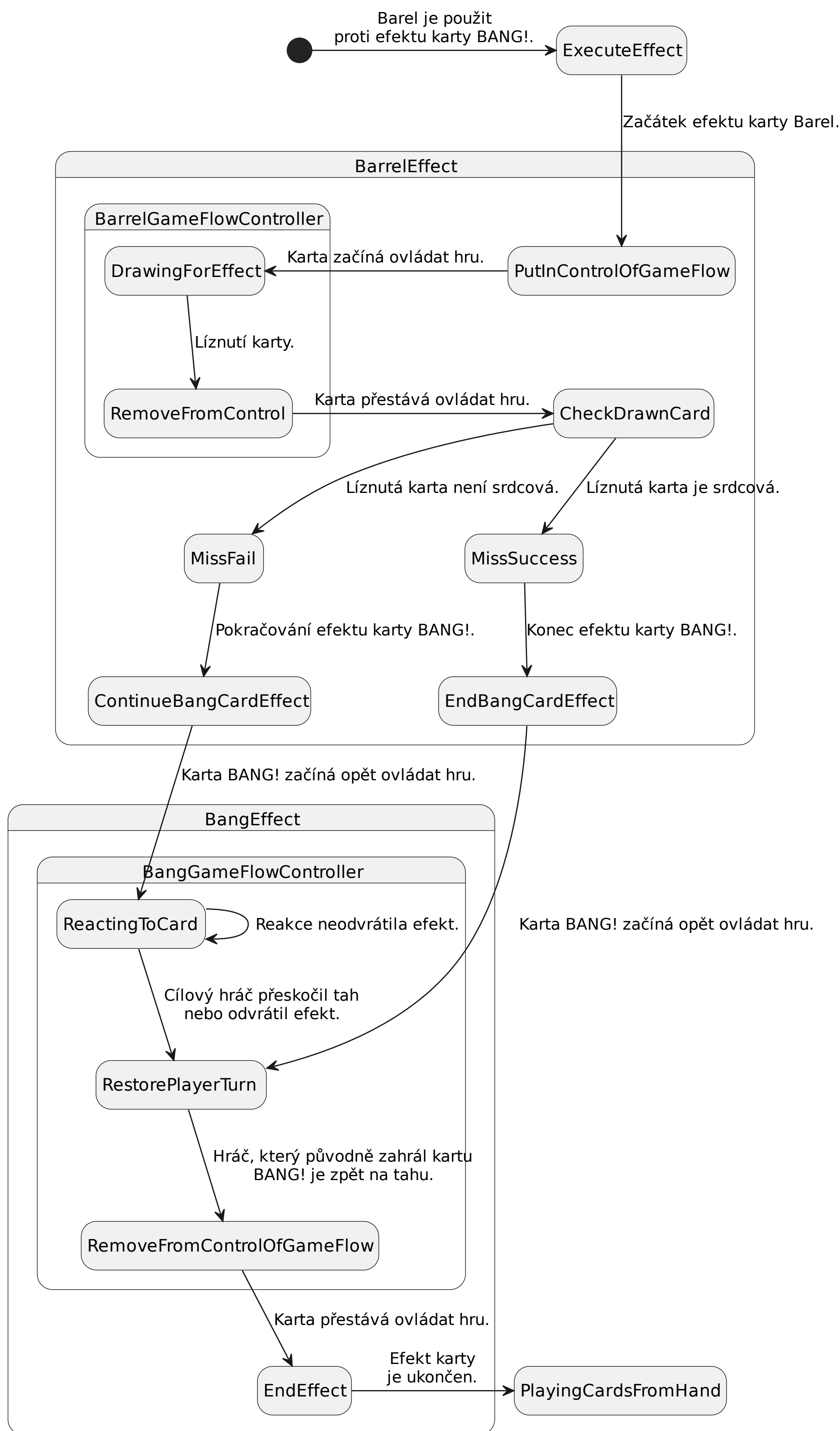
V této podkapitole ukážeme diagramy znázorňující, jak jsou implementovány efekty a různé průběhy tahů pomocí ovladačů hry.



Obrázek 7.9 Diagram standardního průběhu tahu. Ve fázi `Effect` se odehrávají speciální efekty karet a postav.



Obrázek 7.10 Diagram efektu karty `BANG!`.



Obrázek 7.11 Diagram interakce zanořených efektů karet `BANG!` a `Barel`.

8 Závěr

Práce měla dva hlavní cíle. Prvním byl vývoj herního prostředí karetní hry `Bang!` umožňujícího hru po síti a druhým cílem byl výzkum a vývoj umělé inteligence pro tuto hru.

Při vývoji herního prostředí jsme nejprve zvolili herní engine Unity jako hlavní nástroj pro vývoj hry. Následně jsme navrhli přívětivé uživatelské rozhraní, které mělo napodobovat hraní fyzické verze hry `Bang!`. Poté jsme rozdělili projekt na několik hlavních komponent, z nichž každá řeší jeden zadaný problém a poskytuje veřejné rozhraní pro ostatní komponenty. V rámci těchto komponent jsme se také snažili dbát na kvalitu zdrojového kódu a rozšiřitelnost komponent, a to zejména definováním vhodných rozhraní tříd a využitím návrhových vzorů. Výstupem tohoto projektu je herní prostředí umožňující hru více hráčů po síti, čímž jsme splnili náš první cíl.

V rámci výzkumu a vývoje umělé inteligence pro hru `Bang!` jsme se pokusili využít metod strojového učení a neuroevoluce. Naší hlavní myšlenkou bylo natrénovat umělou inteligenci, která by byla schopna ohodnocovat herní akce na základě dostupných informací o herním poli, přičemž by následně prováděla herní akci s nejlepším ohodnocením. Modely strojového učení jsme pro tuto úlohu trénovali pomocí dat nasbíraných z her náhodných hráčů. Zde bylo hlavní myšlenkou, že náhodní hráči prozkoumají různé herní situace, ze kterých se modely mohou učit. Z otestovaných modelů se jako nejpřesnější jevily ensemble metody, které dosahovaly vysokého R^2 skóre a zároveň nízké hodnoty MSE. Z neuroevolučních algoritmů jsme vybrali NEAT, pomocí kterého jsme trénovali univerzální neuronové sítě, které měly umět ohodnocovat herní akce pro všechny role, a speciální neuronové sítě, které se zaměřovaly na ohodnocování herních akcí pro jednu roli ve hře. U všech natrénovaných typů umělých inteligencí jsme následně měřili jejich výkonnost dle výsledků simulovaných her proti náhodným hráčům. Nejlepších výsledků v těchto experimentech dosahovala specializovaná neuronová síť natrénovaná pomocí algoritmu NEAT, proti které jsme následně hráli několik her a pozorovali přitom její způsob hraní. Avšak ve hrách proti člověku tato umělá inteligence měla potíže nejen s odhadováním rolí oponentů, které je obtížné i pro lidské hráče, ale i s volbou správných rozhodnutí v jednoduchých situacích hry. Přesto však nadějně vykazovala, že nejspíše rozumí hlavním konceptům hry, a výsledky experimentů naznačují, že je schopná hrát individuálně i v týmu, a tak považujeme náš druhý cíl za splněný.

8.1 Navázání na práci

Na práci lze navázat různými způsoby. Herní prostředí implementuje pouze základní verzi hry, a tak se nabízí přidat různá existující rozšíření hry. Dále lze zlepšit i uživatelské rozhraní přidáním herních animací a kvalitu zdrojového kódu lze jistě zlepšit přidáním unit testů. Také je možné prozkoumat další libovolné metody umělé inteligence, jako je např. hluboké zpětnovazební učení, které již umožnilo vytvoření agentů s nadliskou výkonností v různých hrách, jakou jsou šachy.

Literatura

1. Tisaac; KuWizard; Ginso. *BANG!* [online]. [cit. 2024-05-05]. Dostupné z: `https://en.boardgamearena.com/gamepanel?game=bang`.
2. Schovánek, Jan. *Online verze karetní hry BANG!* [online]. [cit. 2024-05-05]. Dostupné z: `http://hdl.handle.net/10084/87599`.
3. Trejtnar, Martin. *Multiplatformní karetní hra s umělou inteligencí* [online]. [cit. 2024-05-05]. Dostupné z: `http://hdl.handle.net/11012/199442`.
4. Čevora, Michal. *KBang* [online]. [cit. 2024-05-05]. Dostupné z: `http://hdl.handle.net/20.500.11956/26746`.
5. Daniláková, Monika. *Artificial Intelligence for the Bang! Game* [online]. [cit. 2024-05-05]. Dostupné z: `http://hdl.handle.net/20.500.11956/1994`.
6. Kolář, Vít. *Umělá inteligence ve hře Bang!* [online]. [cit. 2024-05-05]. Dostupné z: `http://hdl.handle.net/11012/52785`.
7. Świechowski, Maciej; Godlewski, Konrad; Sawicki, Bartosz; Mańdziuk, Jacek. *Monte Carlo Tree Search: a review of recent modifications and applications* [online]. [cit. 2024-05-05]. Dostupné z: `https://doi.org/10.1007/s10462-022-10228-y`.
8. Editrice S.r.l., daVinci. *BANG! rules* [online]. [cit. 2024-05-05]. Dostupné z: `https://www.dvgiochi.com/giochi/bang/download/Bang_rules_ENG.pdf`.
9. Inc., Unity Software. *Unity* [online]. [cit. 2025-01-09]. Dostupné z: `https://unity.com/`.
10. Christner, Joel. *WatsonTcp* [online]. [cit. 2024-12-21]. Dostupné z: `https://github.com/dotnet/WatsonTcp`.
11. Inc., Unity Software. *Unity Netcode for GameObjects* [online]. [cit. 2024-05-05]. Dostupné z: `https://docs-multiplayer.unity3d.com/netcode/current/about/`.
12. *Mirror Networking* [online]. [cit. 2024-05-05]. Dostupné z: `https://mirror-networking.com/`.
13. Inc., Unity Software. *Unity Transport* [online]. [cit. 2024-05-05]. Dostupné z: `https://docs-multiplayer.unity3d.com/transport/current/about/`.
14. Editrice S.r.l., daVinci. *daVinci Editrice S.r.l. website* [online]. [cit. 2024-05-05]. Dostupné z: `https://www.dvgiochi.com/`.
15. Russell, Stuart J.; Norvig, Peter. *Artificial Intelligence: A Modern Approach Third Edition*. New Jersey: Pearson Education, Inc., 2009. Third Edition. isbn 0-13-604259-7.
16. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; Duchesnay, E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, roč. 12, s. 2825–2830.

17. *scikit-learn User Guide* [online]. [cit. 2024-05-05]. Dostupné z: https://scikit-learn.org/stable/user_guide.html.
18. *ONNX* [online]. [cit. 2024-05-05]. Dostupné z: https://onnx.ai/.
19. DEVELOPERS, ONNX Runtime. *ONNX Runtime* [online]. [cit. 2024-05-05]. Dostupné z: https://onnxruntime.ai/.
20. CHICCO, Davide; WARRENS, Matthijs J.; JURMAN, Giuseppe. *The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation* [online]. [cit. 2024-07-07]. Dostupné z: https://doi.org/10.7717/peerj-cs.623.
21. STRAKA, Milan. *Machine Learning for Greenhorns* [online]. [cit. 2024-12-28]. Dostupné z: https://ufal.mff.cuni.cz/courses/npfl129/2223-winter.
22. STANLEY, Kenneth O.; MIIKKULAINEN, Risto. Evolving Neural Networks Through Augmenting Topologies. *Evolutionary Computation*. 2002, roč. 10, č. 2, s. 99–127. Dostupné také z: http://nn.cs.utexas.edu/?stanley:ec02.
23. GREEN, Colin. *SharpNEAT* [online]. [cit. 2025-01-01]. Dostupné z: https://github.com/colgreen/sharpneat.
24. INC., Unity Software. *Unity Hub* [online]. [cit. 2025-01-09]. Dostupné z: https://unity.com/unity-hub.

Seznam obrázků

2.1 Rozhodovací strom estimátoru rolí, V. Kolář [6] 10

4.1 Hlavní komponenty. 17
4.2 Uspořádání herního pole, které je schopné zobrazit herní informace o maximálních velikostech. 18
4.3 Standardní průběh tahu. 22
4.4 Přibližný průběh efektu karty `BANG!`. 25
4.5 Přibližné řešení zanořování efektů karet `BANG!` a `Barel` 27

5.1 Dopředná neuronová síť, M. Straka [21] 40
5.2 Mapování NEAT genomu na neuronovou síť, K. O. Stanley a R. Miikkulainen [22] 42
5.3 Dva typy mutací využívané v NEAT. Oba typy jsou ilustrovány se seznamem spojových genů genomu umístěným nad jeho odpovídající neuronovou sítí. Horní čísla v seznamu spojových genů představují inovační čísla jednotlivých spojení (K. O. Stanley a R. Miikkulainen [22]). 43
5.4 Párování genomů dvou různých síťových topologií pomocí inovačních čísel (K. O. Stanley a R. Miikkulainen [22]). 44

6.1 Spuštění hry 61
6.2 Hlavní menu 61
6.3 Seznam uložených serverů 62
6.4 Formulář uložení nového serveru 62
6.5 Seznam lobby 63
6.6 Formulář vytvoření lobby 64
6.7 Lobby z pohledu jeho tvůrce 65
6.8 Lobby z pohledu hráčů, kteří nejsou jeho tvůrci 65
6.9 Výběr postavy 66
6.10 Herní pole 67

7.1 Hlavní komponenty herního prostředí 70
7.2 Přechody mezi obrazovkami herního prostředí. 71
7.3 Diagram třídy `NetworkMessage`. 72
7.4 Diagram třídy `NetworkMessageParser`. 73
7.5 Přechody mezi stavy klienta. 74
7.6 Diagram abstraktní třídy `Card`. 75
7.7 Diagram abstraktní třídy `Character`. 75
7.8 Diagram rozhraní `IGameFlowController`. 76
7.9 Diagram standardního průběhu tahu. Ve fázi `Effect` se odehrávají speciální efekty karet a postav. 77
7.10 Diagram efektu karty `BANG!`. 78
7.11 Diagram interakce zanořených efektů karet `BANG!` a `Barel`. 79

Seznam tabulek

5.1 Měření doby potřebné k simulaci 1000000 herních tahů 35
5.2 Výsledky modelů trénovaných různými metodami 36
5.3 Výsledky různých ensemble metod a rozhodovacího stromu 38
5.4 Výsledky vybraných modelů trénovaných na větších datech 39
5.5 Výsledky z 250000 her čtyř náhodných hráčů. 48
5.6 Výsledky z 250000 her pěti náhodných hráčů. 48
5.7 Výsledky z 250000 her šesti náhodných hráčů. 48
5.8 Výsledky z 250000 her sedmi náhodných hráčů. 48
5.9 Celkové výsledky z 1000000 her čtyř až sedmi náhodných hráčů. . 48
5.10 Výsledky modelů trénovaných dle Oddílu 5.1.9 ve 100000 hrách čtyř hráčů za různé týmy. 50
5.11 Výsledky modelů trénovaných dle Oddílu 5.1.9 ve 100000 hrách pěti hráčů za různé týmy. 50
5.12 Výsledky modelů trénovaných dle Oddílu 5.1.9 ve 100000 hrách šesti hráčů za různé týmy. 51
5.13 Výsledky modelů trénovaných dle Oddílu 5.1.9 ve 100000 hrách sedmi hráčů za různé týmy. 51
5.14 Celkové výsledky modelů trénovaných dle Oddílu 5.1.9 ve 400000 hrách čtyř až sedmi hráčů za různé týmy. 52
5.15 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách čtyř hráčů za různé týmy. 53
5.16 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách pěti hráčů za různé týmy. 54
5.17 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách šesti hráčů za různé týmy. 54
5.18 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách sedmi hráčů za různé týmy. 54
5.19 Celkové výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 400000 hrách čtyř až sedmi hráčů za různé týmy. 55
5.20 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách čtyř hráčů za různé týmy. 56
5.21 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách pěti hráčů za různé týmy. 57
5.22 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách šesti hráčů za různé týmy. 57
5.23 Výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 100000 hrách sedmi hráčů za různé týmy. 57
5.24 Celkové výsledky nejlepšího jedince z 1000. generace trénovaného dle Oddílu 5.2.3 ve 400000 hrách čtyř až sedmi hráčů za různé týmy. 58
5.25 Výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách čtyř hráčů. 59
5.26 Výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách pěti hráčů. 59
5.27 Výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách šesti hráčů. 59

5.28 Výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách sedmi hráčů. . . . 59
5.29 Celkové výsledky člověka proti nejlepšímu specializovanému jedinci z 1000. generace trénovaného dle Oddílu 5.2.3 ve hrách čtyř až sedmi hráčů. 60

A Přílohy

Přílohou této práce je `.zip` archiv, který dále dělíme do šest adresářů obsahujících zejména zdrojový kód, natrénované umělé inteligence a doplňující dokumentace.

A.1 Sestavené herní prostředí

V adresáři `Builds` poskytujeme sestavenou klientskou i serverovou aplikaci našeho herního prostředí:

- Klient: Adresář `Builds/Client`
- Server: Adresář `Builds/DedicatedServer`

A.2 Konzolový klient pro umělé inteligence

V adresáři `ConsoleAiClient` poskytujeme projekt integrující umělou inteligenci do herního prostředí. Návod k použití můžeme nalézt v tomto adresáři v souboru `README.md`.

A.3 Implementace experimentů

Adresář `ExperimentsSolution` představuje `solution`, které dále dělíme na projekty:

- `DataCollectionProject` – program pro sběr trénovacích dat z náhodných her ke trénování modelů strojového učení.
- `Experiment_AiMatchSimulatorPerformance` – experiment měření výkonnosti simulátoru her umělých inteligencí.
- `Experiment_AiVsAi` – experimenty měření výkonnosti umělých inteligencí v simulovaných hrách.
- `NeatTraining` – program pro trénování umělé inteligence pomocí NEAT.

Každý projekt obsahuje soubor `README.md`, který představuje stručný návod k použití.

A.4 Projekt trénování modelů strojového učení

Adresář `MachineLearningProject` obsahuje projekt pro trénování modelů strojového učení. Soubor `README.md` v tomto adresáři představuje stručný návod k použití.

A.5 Zdrojový kód herního prostředí

Adresář `UnityProject` představuje projekt, ve kterém bylo vyvinuto naše herní prostředí. Soubor `README.md` v tomto adresáři slouží jako směrnice do jednotlivých části projektu.

A.6 Zdrojový kód uživatelské dokumentace

Adresář `UserDocsSource` obsahuje zdrojové kódy uživatelských dokumentací.

A.7 Natrénované umělé inteligence

Natrénované ONNX modely a jedince NEAT můžeme nalézt v adresáři `TrainedAi`. Tento adresář rovněž obsahuje soubor `README.md`, ve kterém popisujeme, v jakých souborech lze nalézt jednotlivé umělé inteligence.