
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**
**BRNO UNIVERSITY OF TECHNOLOGY**

**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**
**FACULTY OF INFORMATION TECHNOLOGY**

**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**
**DEPARTMENT OF INTELLIGENT SYSTEMS**


# METODY PRO HRANÍ HRY 'LIAR'S DICE' S VYUŽÍTÍM DYNAMICKÉHO PROGRAMOVÁNÍ

**THESIS TITLE**

**BAKALÁŘSKÁ PRÁCE**
**BACHELOR'S THESIS**


**AUTOR PRÁCE** **MAREK LOHN**
**AUTHOR**

**VEDOUCÍ PRÁCE** **doc. Ing. FRANTIŠEK ZBOŘIL, Ph.D.**
**SUPERVISOR**



**BRNO 2024**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

# Zadání bakalářské práce

156218

Ústav: Ústav inteligentních systémů (UITS)
Student: **Lohn Marek**
Program: Informační technologie
Název: **Metody pro hraní hry Liar's Dice s využitím dynamického programování**
Kategorie: Umělá inteligence
Akademický rok: 2023/24

Zadání:

1. Seznamte se s pravidly hry Liar's Dice a s metodami, které byly publikovány pro hraní této hry počítačem. Zaměřte se na takové, které využívají prostředky dynamického programování.
2. Zvolte vlastní přístup k vytvoření takové metody, který by překonával nebo rozšiřoval publikované metody, které jste nastudoval.
3. Implementujte tyto metody v aplikaci, která umožní hrát jednoho nebo více hráčů strojem.
4. Otestujte kvalitu vašeho řešení s existujícími implementacemi, pokud naleznete vhodné, a také s vhodnou skupinou hráčů, kteří budou ochotni tuto aplikaci řádně otestovat.
5. Diskutujte dosažené výsledky, zlepšení či nižší úspěšnost vašeho algoritmu vůči existujícím realizacím a navrhněte další možné vylepšení metody.

Literatura:

- Trevor Johnson, An evaluation of how Dynamic Programming and Game Theory are applied to Liar's Dice, BP, 2007

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Zbořil František, doc. Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 6.11.2023

# Abstrakt

Tato práce řeší metody hraní hry Liar's Dice s využitím dynamického programování. Pro přístup k této práci byl zvolen algoritmus posilovaného učení SARSA, který je upravenou verzí algoritmu Q-Learning. Tento algoritmus byl následně porovnáván s již existujícími přístupy takovým způsobem, že byl ponechán hrát proti nim za pomocí aplikace, která byla vytvořena v herním engine Unity. Porovnávání proběhlo konkrétně nad algoritmy Q-Learning a Counterfactual Regret Minimization. Ve výsledku bylo dosaženo úspěšnosti 69,147 % ve hře proti Q-Learning a úspěšnosti pouze 25 % proti algoritmu Counterfactual Regret Minimization. Tato práce poskytuje hlavní přehled o tom, jak upravená verze algoritmu SARSA je velmi efektivní ve hře proti algoritmu Q-Learning. Při hraní proti algoritmu Counterfactual Regret Minimization je algoritmus SARSA ve značné nevýhodě.



# Abstract

This project is about Methods of playing game Liar's Dice using dynamic programming. The algorithm that was chosen for my study is SARSA, short for State Action Reward State Action algorithm. It is a modified version of algorithm named Q-Learning. It comparing algorithm SARSA with other algorithms by letting them play against each other in application, that was made in Unity Engine. Algorithms that were compared to SARSA are Q-Learning and Counterfactual Regret Minimization. SARSA achieved a 69,147 % win ratio in a game against Q-Learning. In games against Counterfactual Regret Minimization it was only 25 % win ratio. The main outcome of this study is that modified SARSA is effective against Q-Learning algorithm in a game of Liar's Dice. On the other hand the SARSA algorithm was very ineffective against the Counterfactual Regret Minimization algorithm.



# Klíčová slova

hraní, hry, her, Liars dice, dynamické programování, metody, programovaní, posilované učení, umělá inteligence, AI, učení umělé inteligence

# Keywords

game, playing, dynamic programming, programming, methods, Liars Dice, reinforcement learning, AI learning, AI


# Citace

LOHN, Marek. *Metody pro hraní hry 'Liar's Dice' s využítím dynamického programování.* Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. František Zbořil, Ph.D.

# Metody pro hraní hry 'Liar's Dice' s využitím dynamického programování

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Zbořila Františka Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Marek Lohn
8. května 2024


## Poděkování

Děkuji hlavně svému vedoucím práce, panu doc. Ing. Zbořilovi Františkovi Ph.D. za povzbuzující slova a za užitečné rady, které mi poskytnul.

# Obsah

# Seznam obrázků

# Kapitola 1

# Úvod

Tato práce se věnuje metodám hraní hry 'Liar's Dice' za pomocí dynamického programování. Liar's Dice je hra s neurčitostí pro dva a více hráčů. V této hře se hráči snaží odhadnout počet kostek stejné hodnoty na hrací ploše, každý hráč však zná pouze hodnoty svých vlastních kostek. Při této hře je důležité správně lhát pro oklamání protihráčů a zajištění svého vítězství. Každé kolo končí v okamžiku, kdy hráč, který je na řadě, nazve toho předchozího lhářem. V tomto momentu se odkryjí všechny kostky a zjistí se, který hráč měl pravdu, a tudíž vyhrál kolo.

Účelem bakalářské práce je nalézt metody hraní hry Liar's Dice, v oblasti dynamického programování. Liar's Dice je hra s neurčitostí, ve které jsou hráči nuceni lhát a blafovat. Prozkoumáváním praktik hraní této hry se budou řešit schopnosti umělé inteligence, která byla vytvořena pro tuto práci s využitím již existujících algoritmů a bude se pokoušet napodobit důvěryhodně lidské lhaní.

Ve světě počítačových her patří mezi jedny z primárních pravidel přesvědčit hráče autentičností a flexibilitou umělé inteligence do takového měřítka, kdy bude zcela ponořený do aktuálně probíhajícího děje. Pro zajištění věrohodného průběhu hry musí umělá inteligence působit důvěryhodně ve svém chování. Pokud budou nastavena jednodušší pravidla pro umělou inteligenci, kdy lhát a kdy ne, tak hráči můžou této skutečnosti využít pro svůj vlastní prospěch, a tudíž hra bude pro ně působit zcela bez výzvy a nebudou mít důvod pokračovat.

Metody dynamického programování, které se v minulosti používaly pro hraní hry Liar's Dice, jsou Nash Equilibrium (dále jako Nashova rovnováha) a Minimax. Tyto postupy se zabývají pravděpodobností na vítězství v daném tahu. V dnešní digitální době se k těmto praktikám přidalo posilované učení a neuronové sítě.

Účelem bylo nalezení věrohodného a inteligentního chování umělé inteligence při hraní hry, která je založena na neurčitosti a na lhaní. Metoda, která by toto chování splňovala, by následně mohla být aplikována v herním průmyslu.

Finální výsledek by měl určit vhodnou taktiku pro hraní hry Liar's Dice. Tyto postupy by měly být inteligentní a přiblížit se k napodobení lidskému chování.

Práce je rozdělena na Shrnutí dosavadního stavu, Návrh implementace, Implementace, Testování a Závěr.

# Kapitola 2

# Shrnutí dosavadního stavu

Tento segment popisuje existující metody, algoritmy a vysvětlení základních odborných termínů, které je nutné znát pro porozumění tohoto odborného díla v celém kontextu.

## 2.1 Dynamické programování

Dynamické programování je způsob pro optimalizaci, která řeší komplexní problém rozdělením na podmnožinu jednodušších úkolů a nabízí tak obecný a přívětivý způsob, jak analyzovat obsáhlejší množství různorodých úkonů. Tento postup analýzy umožňuje odlišné techniky optimalizace, které řeší určité aspekty obecných problémů. Pro diverzitu, zda-li se může daný problém vyřešit dynamickým programem, je ve většině případů nezbytné využít kreativního myšlení. Částečná znalost struktury problému je důležitá pro schopnost řešit efektivně pomocí rozdělení na menší podproblémy[5].

Druh tohoto programování navrhl a vytvořil americký matematik Richard Bellman. Ve svém díle se zaměřil na proces rozhodování v kontextu, kde jsou jednotlivá rozhodnutí závislá na stavech, které se mohou měnit v čase. Richard Bellman formuloval princip optimality (z jazyka anglického "Principle of Optimality"), jenž je základní koncept dynamického programování. Ten říká, že zbývající rozhodnutí optimální strategie musí vytvářet optimální techniku s ohledem na stav vyplynulý z prvního rozhodnutí, bez ohledu na počáteční stav a rozhodnutí. Zmíněný princip umožňuje rozložení komplexní rozhodovací procesy na jednodušší podproblémy, které lze řešit postupně.

Řešení optimalizačních problémů ve výrobních a logistických procesech podpořilo vývoj dynamického programování, které později našlo široké uplatnění i v různých oblastech průmyslu od ekonomiky přes inženýrství, až po bioinformatiku. Metoda se stala základním kamenem pro výpočty spojené s umělou inteligencí, strojovým učením a pro vývoj mnoha algoritmů v informatice.

## 2.2 Nashova rovnováha

Nashova rovnováha je stav, kdy ve hře ani jeden z hráčů nemůže změnou své strategie získat výhodu. Každá konečná hra má alespoň jednu takovou rovnováhu. Dosažením této rovnováhy jsou šance na vítězství pro všechny hráče stejné.
Formální definice z knihy Game Theory[4] zní následovně:
V každé strategické hře pro dva hráče, strategický profil $s^* = (s_1^*, s_2^*) \in S_1 \times S_2$ je Nashova

rovnováha právě tehdy, když splňuje následující dvě podmínky:

$$\text{Pro každé } s_1 \in S_1, \pi_1(s_1^*, s_2^*) \geq \pi_1(s_1, s_2^*)$$

$$\text{Pro každé } s_2 \in S_2, \pi_2(s_1^*, s_2^*) \geq \pi_1(s_1^*, s_2)$$

Jedna z možných interpretací z knihy Game Theory[4] je popsaná jako:
Představme si, že hráči mají možnost před začátkem hry komunikovat a domluví se na nezávazné dohodě vyjádřené strategickým profilem $s^*$, tímto způsobem ani jeden hráč nebude mít důvod porušit jejich dohodu (pokud věří jeden druhému, že oba dodrží tuto dohodu) jenom v případě, že strategický profil $s^*$ odpovídá Nashově Rovnováze.
Nalezení Nashovy rovnováhy je časově náročné ve hrách s velkým stavovým prostorem.

## 2.3 Markovův řetězec

Markovův řetězec popisuje možné události, ve kterých pravděpodobnost každé této události závisí pouze na stavu, ve kterém se právě nachází. Hra Liar's Dice byla převedena na Markovův řetězec, kde události které mohly nastat v každém stavu vedly na pokračování hry a nebo na vítězství jednoho hráče.
Definice Markovových Řetězců z online materiálu Markov Chain[8] je vyjádřena následovně: Řekněme že $(X_n) \geq 0$ je Markovův řetězec s počáteční distribucí $\lambda$ a přechodovou maticí P, pokud pro všechna $n \geq 0$ a $i_0, ..., i_{n+1} \in I$, platí že:

$$\text{(i) } P(X_0 = i_0) = \lambda_{i_0}$$

$$\text{(ii) } P(X_{n+1} = i_{n+1} | X_0 = i_0, ..., X_n = i_n) = P(X_{n+1} = i_{n+1} | X_n = i_n) = p_{i_n i_{n+1}}$$

Zjednodušeně řečeno můžeme říci, že $(X_n)_{n\geq 0}$ je $Markov(\lambda, P)$. Kontrola podmínek (i) a (ii) je ve většině případů ta nejlepší cesta, jak určit, zda je náhodný proces $(X_n)_{n\geq 0}$ Markovův řetězec.

## 2.4 Umělá inteligence

Umělá inteligence (AI = z jazyka anglického Artificial Intelligence) je obor informatiky, který se zabývá vytvářením autonomních systémů a algoritmů, které dokáží provádět úkoly, které by jinak vyžadovaly lidskou inteligenci. Tyto úkoly mohou být například rozpoznání obrázků, zpracování jazyka, rozhodování, plánování, učení, generování textů, audiovizuálních souborů a další. AI využívá různé techniky, jako jsou strojové učení, neuronové sítě a další, aby se adaptovala a zlepšovala svoje výsledky v závislosti na poskytnutých datech a zkušenostech. Umělá inteligence má široké uplatnění v různých odvětvích, včetně průmyslu, zdravotnictví, financí, autonomních vozidel, hrách a mnoha dalších. Její vývoj a aplikace jsou doprovázeny otázkami ohledně etiky, bezpečnosti a sociálních dopadů.

### 2.4.1 Historické základy

Alan Mathison Turing, John McCarthy a Marvin Minsky jsou tři důležité postavy ve vývoji umělé inteligence. Každý z nich přispěl zásadním způsobem k teoretickým a praktickým základům tohoto oboru.

Největší příspěvek Alan Mathison Turinga k umělé inteligenci je v návrhu Turingova testu nebo-li takzvané imitační hry. Jedná se o test pro hodnocení schopnosti stroje vykazovat inteligentní chování srovnatelné s člověkem. Imitační hra zahrnuje komunikaci člověka s počítačem, kde pokud člověk nemůže rozpoznat, zda ten s kým komunikuje je stroj nebo člověk, stroj se považuje za "inteligentní"[14]. Tento test položil základ pro filozofické a etické debaty o povaze myšlení a inteligence u strojů a byl podstatný pro další výzkum v oblasti umělé inteligence.

John McCarthy poprvé použil termín umělá inteligence v roce 1956 během Dartmouthské konference, kterou sám organizoval. Tato konference se stala zlomovým bodem, protože zde poprvé zazněla myšlenka, že "Každý aspekt učení nebo jiného znaku inteligence může v principu být tak přesně popsaný, že jej lze nasimulovat strojem". John McCarthy také vyvinul programovací jazyk Lisp, který se stal jedním z nejvíce používaných jazyků ve výzkumu umělé inteligence, díky své schopnosti efektivního zpracování rekurzivních algoritmů. Jeho práce na vývoji metod pro řešení problémů za pomocí počítačů bez explicitních pokynů měla dalekosáhlý vliv.

Marvin Minsky je známý svou prací v oblasti kognitivní psychologie a umělé inteligence, kde se zaměřoval především o imitaci lidského myšlení a chápání. Navrhoval a stavěl mechanické ruce a další roboty, které simulovaly lidské senzorické a motorické funkce. Marvin Minsky je také spolutvůrcem teorie "Society of Mind", která popisuje lidskou mysl jako soubor vzájemně propojených a vzájemně ovlivňujících se součástí, což byl revoluční pohled na kognitivní a výpočetní procesy.

### 2.4.2 Rozvoj umělé inteligence

Díky rozvoji počítačové technologie a zvyšování výpočetního výkonu, se začalo experimentovat s prvními programy umělé inteligence, jako byl například program Eliza nebo šachový program Deep Blue.

Eliza byla jedním z pokusů o simulaci lidské konverzace pomocí počítače. Program, který vytvořil Joseph Weizenbaum v roce 1967, simuloval psychologickou podporu. Eliza byla schopna rozpoznávat klíčová slova ve vstupu od uživatele a reagovat podle předem definovaných scénářů. Ačkoliv byla Eliza poměrně primitivní, ukázala potenciál počítačů pro zpracování přirozeného jazyka a inspirovala další výzkum v oblasti komunikace umělé inteligence.

V 90. letech 20. století vyvinula společnost International Business Machines šachový počítač Deep Blue, který hrál šachy na úrovni tehdejšího mistra světa. Superpočítač Deep Blue kombinoval pokročilé algoritmy pro hodnocení šachové pozice s výkonným hardwarem, který mohl analyzovat miliony možných tahů za sekundu. Rozvoj mikroprocesorové technologie a paralelního programování umožnil Deep Blue vykonávat rozsáhlé výpočty potřebné pro analýzu šachové hry na nejvyšší úrovni. V roce 1997 porazil superpočítač Deep Blue šachového mistra světa Garry Kasparov. Toto byla významná událost demonstrující schopnosti umělé inteligence v komplexních úlohách.

Nástupem pokročilých neuronových sítí, metodou zpětného šíření chyb a konvolučními neuronovými sítěmi se značně vylepšily schopnosti umělé inteligence. Umožnila se aplikace jejich použití v řadě průmyslových a výzkumných oblastí.

Neuronové sítě, inspirované strukturou a chováním lidského mozku, se skládají z vrstev neuronů, které zpracovávají vstupy a předávají signály dále. Tato struktura umožňuje umělé inteligenci učit se a provádět složité úkoly podobným způsobem, jakým mozek zpracovává informace. Neuronové sítě se staly základem pro mnoho systémů umělé inteligence, protože

jejich schopnost učit se z velkého množství dat překonává tradiční přístupy s použitím algoritmů.

Konvoluční neuronové sítě jsou speciálním typem neuronových sítí, které byly navrženy především pro zpracování obrazu. Díky své schopnosti efektivně rozpoznávat vzory a struktury ve vizuálních datech, se konvoluční neuronové sítě staly základem pro analýzu video záznamů. Tyto sítě využívají konvoluční vrstvy, které efektivně identifikují a učí se důležité rysy z obrazů. To umožňuje umělé inteligenci rozpoznat objekty, tváře, scény a další vizuální elementy s vysokou přesností.

Zpětné šíření chyb je strategie používaná pro učení neuronových sítí. Tato metoda umožňuje sítím učit se z chyb tím, že upravuje váhy mezi neurony na základě rozdílu mezi očekávaným a skutečným výstupem. Zpětné šíření chyb poskytlo systematický způsob, jak optimalizovat neuronové sítě. To bylo důležité pro další vývoj složitějších modelů schopných vykonávat náročnější úkoly.

### 2.4.3 Současný stav umělé inteligence

Umělá inteligence dosáhla bodu, kde jsme schopni vytvářet systémy, které mohou autonomně řídit vozidla, rozpoznávat a zpracovávat přirozený jazyk, a provádět složité rozhodovací úkoly. Autonomní vozidla využívají kombinaci konvoluční neuronové sítě a dalších neuronových sítí k analýze vizuálních dat z kamer a senzorů pro navigaci v reálném čase. Systémy pro zpracování přirozeného jazyka nejčastěji vídáme jako virtuální asistenty, například Alexa (od společnosti Amazon), Siri (kterou vlastní firma Apple) a další. Tito asistenti využívají pokročilé neuronové sítě k porozumění a generování lidské řeči, která umožňuje přirozenou a smysluplnou interakci mezi uživatelem a umělou inteligencí.

V dnešní době je téměř přirozené, že umělá inteligence je součástí spotřební elektroniky, od inteligentních asistentů v telefonech a domácích zařízeních až po webové služby nabízející různé asistenty pro programování, tvorbu digitálních obrazů a dalších. Tyto systémy zlepšují uživatelskou přívětivost tím, že se učí z předchozích interakcí a přizpůsobují své reakce potřebám a preferencím uživatele.

### 2.4.4 Etické problémy umělé inteligence

Rozvoj umělé inteligence vyvolává otázky ohledně spravedlnosti a transparentnosti rozhodování. Algoritmy mohou reprodukovat nebo poukázat na stávající sociální a ekonomické nerovnosti. Toto chování se často projeví až při používání nástroje s umělou inteligencí a je potřeba jej regulovat a správně ošetřit. Světová populace z tohoto důvodu hledá možnosti, jak formulovat etické směrnice pro vývoj a používání umělých inteligencí, aby technologie sloužila společnosti bezpečně a spravedlivě. Vývoj umělé inteligence je tak rapidní, že momentálně málokterý státní útvar je schopen urychleně reagovat na danou situaci a dle toho upravovat nebo zavádět regulace[11].

### 2.4.5 Umělá inteligence - ochrana soukromí

S rostoucími schopnostmi umělé inteligence shromažďovat, analyzovat a pochopit velké objemy dat se zvyšují i obavy o soukromí a bezpečnost. Nedostatečně zabezpečené databáze a protokoly pro komunikaci s těmito nástroji mohou vést k zneužití dat a vážnému narušení soukromí uživatelů. Firmy si začínají vyrábět vlastní asistenty s umělou inteligencí, aby omezili možnost úniku dat. Přesto je velmi důležité nedávat soukromé nebo firemní infor-

mace volně dostupné umělé inteligenci, protože nemůže být zaručena bezpečnost uchování těchto dat.

### 2.4.6 Pohled společnosti na umělou inteligenci

Automatizace, kterou umožňuje umělá inteligence, má potenciál nahradit určité druhy zaměstnání a vytvořit nové. To vyvolává u některých lidí obavy, že jejich práci bude odvádět automatický stroj s umělou inteligencí. Lidé také začínají vytvářet umělecká díla a odborné práce za pomocí asistentů s umělou inteligencí a vydávají je za svá vlastní. To vede k větší míře nedůvěry ve společnosti a začínají vznikat pravidla, která se snaží toto chování omezit. Některé nástroje jsou již natolik vyspělé, že vzniká obtíž rozpoznat rozdíl mezi vygenerovaným obsahem a lidským dílem.

### 2.4.7 Shrnutí

Umělá inteligence se rychle stává jedním z nejužitečnějších nástrojů naší doby. Její schopnosti a aplikace neustále rostou a to nám umožňuje řešit problémy, které byly dříve považovány za nedosažitelné. Ačkoliv potenciál umělé inteligence je obrovský, je důležité se naučit, jak ji používat. To zahrnuje pochopení rizik spojených s používáním a prevenci úniku citlivých dat.

## 2.5 Posilované učení

Posilované učení patří pod metody strojového učení a funguje na principu osvojování zkušeností z provedených akcí. Algoritmus vybere některou z dostupných akcí a přesune se do určitého stavu. Podle toho do jakého stavu se dostal, bude akce kterou algoritmus zvolil odměněna. V příštích tazích algoritmus vybírá spíše ty akce, za které dostal největší odměny. Podle online knihy Reinforcement Learning: An Introduction [12] je posilované učení výpočetní metoda pro pochopení a automatizaci učení pro dosažení cíle a učení rozhodovacího procesu. Na rozdíl od ostatních výpočetních postupů, se tento postup soustředí na učení agenta pomocí přímé interakce s prostředím ve kterém se nachází. Posilované učení je nezávislé na neustálém dozoru a na kompletních modelech svého prostředí. Je jedno z prvních oborů, které se do hloubky zabývá výpočetními problémy při učení se z interakcí s prostředím za účelem dosažení dlouhodobých cílů.
Posilované učení používá formální rámec, který definuje interakci mezi učícím se agentem a jeho prostředím pomocí stavů, akcí a odměn. Tento rámec by měl být jednoduchou cestou, jak reprezentovat důležité vlastnosti umělé inteligence. Mezi tyto vlastnosti patří smysl pro příčinu a následek, smysl pro neurčitost a nedeterminismus, a existence explicitních cílů.
Mezi nejdůležitější vlastnosti většiny metod posilovaného učení, patří konceptuální definice hodnoty a funkce počítající tyto hodnoty. Funkce počítající hodnoty jsou důležité pro efektivní vyhledávání v prostoru strategií. Posilované učení se od evolučních metod, které hledají strategii přímo ve stavovém prostoru pomocí skalárního výpočtu všech strategií, liší užitím funkcí pro výpočet hodnoty.

### 2.5.1 Q-Learning

Q-Learning je posilované učení, které získává hodnotu dané akce v příslušném stavu. Q-Learning hledá optimální řešení tím, že zvyšuje hodnotu akcí, které vedly k lepšímu stavu. Tímto způsobem získají největší hodnoty ty akce, které postupně vedly až k vyžadovanému

cíli. Ve hře Liar's Dice jsou tyto lepší stavy, když nepřítel přijde o kostku a když se nikomu nic nestalo.
Podle online knihy Reinforcement Learning: An Introduction [12] je funkce pro výpočet hodnoty definována takto:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma maxQ(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.1}$$

V tomto případě, naučená funkce pro výpočet hodnoty akce, Q, upřesňuje optimální funkci $q^*$ pro výpočet hodnoty, nezávisle na strategii. Tento způsob dramaticky zjednodušuje analýzu tohoto algoritmu. Použitá strategie má efekt na to, které páry stavu a akce jsou použity a aktualizovány. Jediná podmínka pro správnou konvergenci je ta, že všechny páry stavu a akce, budou nadále aktualizovány.
Pro rovnici 2.1 platí, že $Q_{new}(s_t, a_t)$ je hodnota Q ve stavu $s_t$ a s vybranou akcí $a_t$. Symbol $\alpha$ je rychlost učení v rozmezí od nuly do jedné. Rychlost učení mění jak rychle se algoritmus bude učit nové hodnoty Q. Symbol $\gamma$ znázorňuje jak moc má algoritmus přihlížet k hodnotě Q v následujícím stavu $s_{t+1}$ s vybranou akcí $a_{t+1}$v následujícím stavu. Symbol $\gamma$ má rozmezí od nuly do jedné. Proměnná $r_t$ je odměna za akci $a_t$.

### 2.5.2 SARSA

SARSA je modifikovaná verze Q-Learning, kde pro výpočet hodnoty dané akce v příslušném stavu nezáleží pouze na stavu do kterého se dostane, ale i na předpokládané hodnotě následující akce. Tímto způsobem algoritmus SARSA vytváří posloupnost akcí, které vedou k vyžadovanému cíli.
SARSA byla navržena v technické práci G. A. Rummery a M. Niranjan [9] pod jménem Modified Connectionist Q-Learning.
Druhé jméno SARSA pro tento algoritmus, které navrhl Rich Sutton, odráží fakt, že hlavní funkce pro aktualizaci hodnoty Q, závisí na současném stavu $s_1$ agenta, na akci $a_1$, kterou agent provede. Dále závisí na odměně $r$, kterou agent dostane za volbu akce $a_1$, na stavu $s_2$, do kterého se agent dostane zvolením akce $a_1$ a na následující akci $a_2$, kterou si agent vybere ve svém novém stavu $s_2$. Definice aktualizační funkce, která je převzatá z článku State–action–reward–state–action[1] zní následovně:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{2.2}$$

### 2.5.3 Counterfactual Regret Minimization

Counterfactual Regret Minimization je algoritmus posilového učení, který na rozdíl od Q-Learning a SARSA, hledá optimální způsob dosažení cíle (v tomto případě poražení nepřítele) skrze vyhýbání se akcím, které vedou k porážce. Tento algoritmus přidává hodnoty "lítosti"(z jazyka anglického "regret") akcím, které vedly ke špatnému výsledku. Podle vypracovaného textu od Todd W. Neller a Marc Lanctot[7], je Counterfactual Regret Minimization vysvětlen takto:
V každé informační sadě, která byla rekurzivně navštívena při iteraci během učení, je vypočítána komplexní strategie, za pomocí rovnice pro výpočet hodnot lítosti. Tato rovnice je definována následovně:
Nechť $A$ je sada všech možných akcí ve hře. Nechť $I$ je informační sada a $A(I)$, znázorňuje sadu povolených akcí pro informační sadu $I$. Nechť $t$ a $T$ značí časové kroky. $t$ se zvyšuje při každém přístupu k informační sadě $I$. Strategie $\sigma_i^t$ pro hráče $i$ mapuje každou informační

sadu tohoto hráče a povolenou akci $a \in A(I_i)$ na pravděpodobnost, že si hráč vybere v časovém kroku $t$, akci $a$ v informační sadě hráče $I_i$. Všechny hráčovy strategie v čase $t$ vytváří strategický profil $\sigma^t$. Strategický profil, který neobsahuje strategii hráče $i$, je označován jako $\sigma_{-i}$. Nechť $\sigma_{I \to a}$ značí profil, který je ekvivalentní s profilem $\sigma$, až na to, že akce $a$ je vždy vybrána z informační sady $I$.
Historie $h$ je postupnost akcí (včetně náhodných výsledků), začínající od počátku hry. Nechť $\pi^\sigma(h)$ je pravděpodobnost dosažení hry s historií $h$ a strategickým profilem $\sigma$. Dále nechť $\pi^\sigma(I)$ je pravděpodobnost dosažení informační sady $I$, přes jakoukoliv možnou historii $h$ v informační sadě $I$. Z toho vyplývá, že $\pi^\sigma(I) = \sum_{h \in I} \pi^\sigma(h)$. Kontra-faktuální pravděpodobnost dosažení informační sady $I$, značena $\pi^\sigma_{-i}(I)$, je pravděpodobnost dosažení informační sady I se strategickým profilem $\sigma$, s předpokladem, že všechny akce hráče $i$, které vedly do tohoto stavu, mají pravděpodobnost 1. Ke všem situacím nazývaným jako kontra-faktuální se chováme tak, jako kdyby výpočet strategie hráče $i$, byl modifikován aby úmyslně hrál informační sadu $I_i$. Řečeno jinak, vynecháme pravděpodobnosti, které se konkrétně měly hráčem $i$ zahrát, z výpočtu.
Nechť $Z$ značí sadu všech konečných historií (postupnost od kořene až po list). Poté vlastní předpona $h@z$, kde $z \in Z$, je historie hry, která není konečná. Nechť označení $u_i(z)$ je pro hráče $i$, užitečnost konečné historie $z$. Definice kontra-faktuální hodnoty v historii $h$, která není konečná, je následovná:

$$\upsilon_i(\sigma, h) = \sum_{z \in Z, h@z} \pi^\sigma_{-i}(h) \pi^\sigma(h, z) u_i(z)$$

Kontra-faktuální hodnota lítosti nevyužité akce $a$ v historii $h$, je definována jako:

$$r(h, a) = \upsilon_i(\sigma_{I \to a}, h) - \upsilon(\sigma, h)$$

Následně můžeme určit kontra-faktuální hodnotu lítosti nevyužité akce $a$ v informační sadě $I$ jako:

$$r(I, a) = \sum_{h \in I} r(h, a)$$

Nechť $r_i^t(I, a)$ označuje hodnotu lítosti při použité strategii $\sigma^t$, když hráč nepoužije akci $a$ v informační sadě $I$, patřící hráči $i$. Kumulativní kontra-faktuální hodnota lítosti je definována jako:

$$R_i^T(I, a) = \sum_{t=1}^{T} r_i^t(I, a)$$

Hodnota lítosti akce je rozdíl mezi hodnotou, kdy se vždy vybere akce $a$ a předpokládanou hodnotou při použití strategie $\sigma$. Tato hodnota je vydělena pravděpodobností, že ostatní hráči (včetně náhody) budou hrát, aby dosáhli tohoto stavu. Pokud definujeme nezápornou hodnotu kontra-faktuální hodnoty lítosti jako $R_i^{T,+}(I, a) = max(R_i^T(I, a), 0)$, tak dostaneme novou strategii:

$$\sigma_{\mathrm{i}}^{\mathrm{T+1}}(\mathrm{I}, \mathrm{a}) = \begin{cases} \frac{R_i^{T,+}(I,a)}{\Sigma_{a \in A(I)} R_i^{T,+}(I,a)} & \text{pro } \Sigma_{a \in A(I)} R_i^{T,+}(I, a) > 0 \\ \frac{1}{A(I)} & \text{jinak} \end{cases}$$

Tato rovnice je použita pro výpočet pravděpodobnosti akcí v poměru k nezáporné kumulativní hodnotě lítosti, pro každou informační sadu. Counterfactual Regret Minimization vygeneruje pro každou akci, další stav ve hře a vypočítá užitečnost každé akce rekurzivně.

Hodnoty lítosti jsou vypočteny pomocí užitečnosti akce a hodnota dosažení současného stavu je tak závěrečně vypočtena a navrácena.
Průměrný strategický profil v informační sadě $I$, $\sigma^{-T}(I)$, se přibližuje k $T \to \infty$. Průměrná strategie v informační sadě I, $\sigma^{-T}(I)$ se získá normalizací $s_I$ přes všechny akce $a \in A(I)$. Tato průměrná strategie následně konverguje k Nashově rovnováze.

## 2.6 Posilované učení - Strategie

Strategie, také nazýváno politika (z jazyka anglického 'Policy'), určuje způsob chování umělého hráče v daném čase. Jednodušeji řečeno, strategie je způsob, jak určit akci, která se vykoná v daném stavu z množiny dostupných akcí pro tento stav. Tato strategie se v psychologii podobá sadě pravidel, které reagují na podnět nebo asociaci. Pro některé případy může být strategie jednoduchá funkce nebo tabulka hodnot, kde každý podnět má přiřazenou hodnotu. Strategie ale může být i výpočetně náročná operace, například prohledávání. Strategie je klíčovým prvkem posilovaného učení, protože sama o sobě dokáže určit chování. Obecně řečeno, strategie posilovaného učení mohou být stochastické[12].

### 2.6.1 Epsilon-greedy strategie

Chamtivá (z jazyka anglického "greedy") strategie, vždy zvolí tu akci, která má největší odměnu. Tato strategie neztrácí čas na kontrole, zda-li akce s menší odměnou jsou potencionálně lepší, než akce s největší odměnou.

Alternativou chamtivé strategie je strategie epsilon-greedy, která vybírá většinu času chamtivě, ale s malou pravděpodobností $\epsilon$ vybere z ostatních povolených akcí se stejnou pravděpodobností jednu náhodně[12]. Stategie výběru se dá popsat jako:

$$P(a_g) = 1 - \epsilon$$

kde $a_g$ je akce $a$ s největší možnou odměnou v daném stavu. $P(a_g)$ je pravděpodobnost výběru akce $a_g$ a $\epsilon$ je většinou malé číslo, které značí pravděpodobnost výběru náhodné nechamtivé akce.

### 2.6.2 Adaptivní epsilon-greedy strategie založena na rozdílu hodnot

Adaptivní epsilon-greedy strategie založena na rozdílu hodnot mění hodnotu $\epsilon$ podle rozdílu ve změně potencionální odměny. Velká změna potencionální odměny zvětší hodnotu $\epsilon$ a vede k větší exploraci. Malá změna potencionální odměny zmenší hodnotu $\epsilon$ a tím povede k větší chamtivosti a menší exploraci.

Nevýhoda je, že průzkumné akce jsou vybírány s pravděpodobností rovnoměrně rozdělenou mezi všechny možné akce za současného stavu. Takové průzkumné chování může vést ke neoptimalizovanému výkonu, kde mnoho akcí současného stavu vede k relativně vysoké negativní odměně. I když je tato znalost přítomna prostřednictvím v již naučené hodnotě Q[13].

### 2.6.3 Probability matching strategie

Probability Matching je strategie, kterou používá umělá inteligence v rámci posilovaného učení k rozhodování o svých akcích na základě pravděpodobnosti úspěchu každé akce. Tato strategie je často používána v situacích, kde jsou nejisté informace o odměnách za každou akci.

Podstata strategie Probability Matching spočívá v tom, že umělá inteligence volí své akce v souladu s pravděpodobnostmi jejich úspěchu. Nevolí automaticky akci s nejvyšší pravděpodobností úspěchu, ale volí svůj tah náhodně, kde každá akce má pravděpodobnost výběru stejnou jako pravděpodobnost, že bude odměněna.

V uměle navozeném případě máme umělou inteligenci, která se snaží naučit se, jakým způsobem dojít k východu v bludišti. V každém kroku má na výběr mezi čtyřmi možnými směry: nahoru, dolů, doprava a doleva. Každý směr má různou pravděpodobnost, že vede k východu. Namísto toho, aby umělá inteligence jednoduše zvolila směr s nejvyšší pravděpodobností úspěchu, rozhodne se podle pravděpodobností úspěchu všech směrů. Pokud je například pravděpodobnost úspěchu směru nahoru 0.4, dolů 0.3, doprava 0.2 a doleva 0.1, agent zvolí směr nahoru s pravděpodobností 40%, směr dolů 30%, směr doprava 20% a směr doleva s pravděpodobností 10%. Umělá inteligence tímto způsobem stále upřednostňuje směry s vyšší pravděpodobností úspěchu, ale zároveň je schopna reagovat na nejistotu odměn spojených s jednotlivými směry.

Je důležité poznamenat, že strategie Probability Matching může být vhodná v určitých situacích, ale není univerzálně nejlepší strategií pro všechny problémy v rámci posilovaného učení.

## 2.7 Teoretické porovnání Q-Learning a SARSA

Algoritmus SARSA bere v úvahu nejen okamžitou odměnu za akci, ale i hodnotu následujícího stavu a akce, které jsou výsledkem této akce. Na druhé straně Q-Learning při počítání potencionálních odměn počítá s maximální potenciální odměnou pro následující stav bez ohledu na to, jaká akce bude skutečně provedena. Q-Learning tímto způsobem hodnotí potencionální odměny za akce, které mohou působit zdánlivě výhodně, ale ve skutečnosti vedou k většímu riziku.

SARSA se může uchylovat k pomalejšímu učení, neboť tento algoritmus se často zaměřuje více na exploraci. To znamená, že během učení pravidelně testuje nové akce, aby získal lepší porozumění o celkovém prostředí, což může výrazně prodloužit proces nalezení optimální politiky. Algoritmus SARSA váží výběr akcí založený na aktuálně sledované strategii, což mu pomáhá získat realističtější odhad toho, jak dobré dané akce ve skutečnosti jsou.

Na druhé straně, Q-Learning může být méně stabilní, protože se zaměřuje na nalezení akcí, které nabízejí nejvyšší okamžitou odměnu bez ohledu na to, zda je daná cesta opravdu optimální ve větších časových rámcích. Tento přístup může vést k situacím, kde algoritmus "uvázne"u akcí, které se zdají být přívětivé na první pohled, ale ve skutečnosti nevedou k dlouhodobě udržitelnému úspěchu. Například, pokud Q-Learning narazí na zvýhodněnou cestu, která nabízí vysoké odměny ve krátkodobém horizontu, může ignorovat alternativní cesty, které by ve skutečnosti mohly vést k vyšším celkovým výnosům.

## 2.8 Teoretické porovnání SARSA a Counterfactual Regret Minimization

SARSA a Counterfactual Regret Minimization jsou oba algoritmy posilovaného učení, ale jsou zásadně odlišné v jejich přístupech a aplikacích. SARSA je metoda založená na strategii, která aktualizuje hodnoty své politiky na základě odměn získaných během skutečně provedených akcí a následných stavů, tedy učí se na základě zkušeností získaných přímo ze hry.

Na druhou stranu, Counterfactual Regret Minimization je sofistikovanější technika, která se zaměřuje na minimalizaci "lítosti", což je rozdíl mezi odměnou, kterou by umělá inteligence získala, kdyby se rozhodla jinak, a odměnou, kterou skutečně získala. Tento algoritmus je široce používán v hrách s více hráči, zejména v těch, které zahrnují nepřímou komunikaci a záměrné rozhodování, jako jsou poker a jiné karetní hry, kde strategie vyžaduje předvídání a reakci na chování ostatních hráčů. Counterfactual Regret Minimization systematicky prochází možné rozhodovací body ve hře a upravuje strategie tak, aby minimalizoval průměrnou lítost po dobu celé hry.

Zatímco SARSA může být efektivně implementována ve většině standardních prostředích posilovaného učení a je relativně snadná na pochopení a aplikaci, Counterfactual Regret Minimization vyžaduje složitější výpočty a je specificky přizpůsoben pro složitější situace, kde se rozhodnutí jednotlivých hráčů navzájem ovlivňují. SARSA obvykle hodnotí jednotlivé akce na základě očekávaného přímého výsledku a Counterfactual Regret Minimization analyzuje celkový výsledek strategií přes možné řetězce rozhodnutí a jejich dlouhodobé dopady.

SARSA je vhodnější pro situace, kde prostředí umělé inteligence je relativně předvídatelné a kde jsou rozhodnutí nezávislá, zatímco Counterfactual Regret Minimization vyniká v komplexnějších hrách. Například strategické hry pro více hráčů, kde je zapotřebí hluboké pochopení a předvídání rozhodnutí ostatních hráčů.

## 2.9 Unity

Unity je herní engine s podporou pro mnoho platform, vyvinutý společností Unity Technologies[3], který se využívá pro vývoj her.
Unity nabízí svým uživatelů možnost vytvářet digitální hry a zážitky ve druhé i třetí dimenzi. Samotný engine dává uživatelům přístup k aplikačnímu programovacímu rozhraní v jazyce C# pro Unity editor v podobě nasazovacích modulů (z jazyka anglického "plugin"), a pro hry samotné, s možností přemísťování prvků (z jazyka anglického Drag and Drop). Před používáním jazyku C# jako hlavní programovacím jazykem, byl podporován jazyk Boo. Podpora programovacího jazyka Boo skončila při vydání Unity 5. Následně skončila podpora implementace programovacího jazyka UnityScript, který byl implementací programovacího jazyka JavaScript na základě jazyka Boo.
Pro dvou dimenzionální hry Unity nabízí možnost importování takzvaných Sprites, což jsou dvou dimenzionální grafické objekty, obvykle používané pro postavy, objekty a projektily. Dále nabízí pokročilý nástroj pro vykreslování dvou dimenzionálního světa.
Pro tří dimenzionální hry, nabízí Unity specifikaci komprese textur, mipmapy (sada textur různého rozlišení pro vykreslování v závislosti na vzdálenosti od kamery) a nastavení rozlišení pro každou platformu, kterou Unity podporuje. Dále Unity podporuje mapování odrazů, hloubky, SSAO, dynamické stíny pomocí stínových map, vykreslování na textury a efekty po zpracování (z jazyka anglického post-processing effects). Unity má také dostupné dva vykreslovací způsoby(z jazyka anglického render pipelines), High Definition Render Pipeline (HDRP) a Universal Render Pipeline. K těmto dvěma je navíc dostupná legacy built-in pipeline. Všechny vykreslovací způsoby jsou navzájem nekompatibilní.

## 2.10 Unreal Engine

Unreal Engine je sbírka 3D grafických herních nástrojů, vytvořena pro vývoj počítačových her společností Epic Games. Poprvé se objevil ve hře Unreal roku 1998. Původně byl zamýšlen pouze pro počítačové akční hry, ale od té doby se dostal i do většiny herních žánrů a dokonce i do filmové scény. Unreal Engine je napsán v jazyce C++ a je dostupný pro širokou škálu platform, mezi které patří počítač, mobilní telefon, konzole a virtuální realita. Nejnovější generace, Unreal Engine 5, vyšla roku 2022 v Dubnu. Zdrojový kód je dostupný pro registrované uživatele na stránkách GitHub[2].

Vývoj Unreal Engine pokračoval s vydáním Unreal Engine 2 v roce 2002. Tato verze přinesla vylepšenou grafiku a podporu pro konzole PlayStation 2 a Xbox. V roce 2006 byl vydán Unreal Engine 3. Tato verze přinesla pokročilé osvětlení, fyziku a animaci, což umožnilo vytvářet ještě realističtější a zajímavější hry. Unreal Engine 4 byl oficiálně oznámen v roce 2012 a byl vydán v roce 2014. Tato verze přinesla ještě větší grafické a vývojářské možnosti, včetně podpory pro virtuální realitu.

Unreal Engine 5 přinesl další novinky, které vylepšily možnosti vývoje her a vizuálních projektů. Jednou z hlavních inovací je technologie nazvaná Nanite, která umožňuje vykreslovat scény s obrovským množstvím detailů a polygonů přímo v samotném vývojovém prostředí. To zjednodušuje proces tvorby a zvyšuje úroveň detailů výsledného produktu. Další novinkou je technologie Lumen, dynamický globální osvětlovací systém, který umožňuje realistické osvětlení scén bez manuálního nastavování světel či předvypočítaných osvětlovacích map. Unreal Engine 5 také přidal MetaSounds, novou zvukovou technologii, umožňující vývojářům vytvářet dynamické zvukové efekty a prostředí s minimálním znalostí programování. Mezi další novinky patří podpora pro specifické funkce herního ovladače DualSense pro konzoli PlayStation 5 a rozšířená podpora technologie Ray-tracing.

## 2.11 Godot

Godot Engine je pro všechny uživatele volně dostupný program, který z počátku začal jako osobní projekt Juana Linietskyho a Ariela Manzura. Ze začátku byl vyvíjen jako nástroj pro osobní použití v jejich herním studiu, ale později se stal veřejným i pro širokou veřejnost. První začátky Godot Enginu se objevují již v roce 2007, kdy Juan Linietsky začal vyvíjet herní engine pod názvem "D.". Tento engine byl určen pro vývoj her pro platformu PlayStation Portable (PSP). Po několika letech vývoje se Linietsky spojil s Arielem Manzurou a společně přejmenovali projekt na již známý Godot. V roce 2014 se Godot Engine stal veřejným programem pod licencí MIT. To znamená, že kdokoli může přispívat k jeho vývoji a používat ho zcela zdarma pro jakýkoli účel, včetně komerčního využití. Tento krok byl klíčovým momentem v historii Godot Engine, protože přilákal mnoho nezávislých vývojářů, kteří chtěli přispět k jeho rozvoji.

Postupně se Godot Engine stal populárním nástrojem pro vývoj her díky své flexibilitě a výkonu. Jeho komunita rostla a přispívala k jeho zdokonalování. Engine nabízí širokou škálu funkcí, včetně podpory pro dvou i třírozměrnou grafiku, fyziku, zvuk, animace, design uživatelského rozhraní, a mnoho dalšího. Godot Engine je stále aktivně vyvíjen a zdokonalován komunitou vývojářů po celém světě. Mnoho her vzniká pomocí tohoto enginu a Godot získal uznání jako silný a dostupný nástroj pro tvorbu počítačových her a interaktivních aplikací.

## 2.12 Hry s neurčitostí

Hry s neurčitostí jsou takové hry, které obsahují prvek náhodnosti při jejich hraní. Příkladem jsou hry s kostkou, kde to jaká čísla padnou na kostkách, je náhodný jev. Neurčitost v těchto hrách komplikuje jejich řešení, a proto vznikla snaha nalézt strategii hraní pomocí algoritmů.
Neurčitost je primární charakteristika mnoha způsobů hraní[6], ne jenom her. Řečeno programátorsky můžeme nazvat hru podtřídou hraní a charakteristika neurčitosti se tak dědí z třídy hraní na podtřídu hra.
Každá hra zkušeností obsahuje už z definice riziko, že hráč může svůj tah pokazit a hra může skončit jeho porážkou. Bez tohoto rizika, přestává být hra zábavná. Na druhou stranu, pokud bude hráč až příliš vytrénovaný v dané hře a bude vyhrávat bez námahy, hra pro něj také přestává být zajímavá.
Pokud hra začne být předvídatelná, ztrácí charakteristiku neurčitosti.
Ve většině případů se budou muset agenti umělé inteligence vypořádat s neurčitostí, ať už kvůli částečného pozorování, ne-determinismu, nebo kombinací obou případů[10]. Může se stát, že agent nebude vědět v jakém stavu se nachází nebo do jakého stavu se dostane po postupnosti akcí. Existují způsoby, které řeší neurčitost tím, že si uchovávají informace o všech možných stavech, ve kterých se mohou nacházet, a generují si plán postupu, který řeší jakýkoliv možný děj, který je agent schopen zaznamenat. Navzdory mnoha výhodám, má tento postup vážné následky pokud se bere jako přesný návod k vytváření agentů. Mezi nevýhody tohoto postupu patří příliš velký stavový prostor, plán postupu může být také příliš velký, protože je potřeba promyslet všechny možné události, nehledě na to, jak moc nepravděpodobné jsou. Může nastat i situace, kdy neexistuje plán, který by zaručil, že splní daný cíl, ale přesto se agent musí rozhodnout jakou akci vykoná.
Můžeme říct, že agent daný problém vyřeší, když nenastane žádná neurčitá událost. Jelikož ale není jisté, zda nastane některá z neurčitých událostí, vzniká kvalifikační problém. Pravděpodobnost se vypořádává s neurčitostí zajišťuje způsob, jak vyřešit kvalifikační problém. Agent si nemusí být jistý, zda daný postup vede určitě k vyžadovanému cíli, stačí mu vědět, že má dostatečně dobrou pravděpodobnost se k vyžadovanému cíli dostat. Prohlášení, které bylo provedeno na základě pravděpodobnosti, se určuje na základě vědomostí o daném stavu, ne na základě reálného stavu, který nám je téměř vždy pouze částečně odhalen.
Stále může existovat vícero možných řešení a rozhodování pouze na základě nejvyšší pravděpodobnosti nemusí být, to nejlepší řešení. Například agent, který se potřebuje dostat na letiště. Agent pojede autem, bezpečnou rychlostí. Pokud nenastane žádná nečekaná událost, agent je schopen se na letiště dostat za 30 minut. Pokud agent vyrazí o 90 minut dříve, než mu odletí letadlo, bude mít pravděpodobnost 95 %, že dorazí tak aby letadlo zastihl. Agent může vyrazit na letiště i o 24 hodin dříve a pravděpodobnost, že dorazí včas bude 99,9 %.Ze zkušeností však víme, že takový plán není optimální. Aby se agent uměl správně rozhodnout, musí mít preference, na jejichž základě si následně může vybírat mezi výsledky možných plánů. Výsledek je kompletně specifikovaný stav, včetně vlastností jako je například, zda agent dorazil včas a jak dlouho musel čekat na letadlo. Pro reprezentaci preferencí se používá teorie použitelnosti. Teorie použitelnosti udává každému stavu míru užitečnosti, nebo použitelnosti, na základě které, si agent vybírá stavy s větší mírou použitelnosti.
Pravděpodobnost kombinovaná s použitelností spojí dohromady teorii rozhodování pro agenta hrajícího hru s neurčitostí. Základní myšlenka teorie rozhodování je, že agent je racionální tehdy a pouze tehdy pokud zvolí akci, která má tu největší míru použitelnosti, zprůměrněná všemi možnými výsledky vybrané akce.

## 2.13 Liar's Dice

Liar's Dice je hra s kostkami pro dva a více hráčů. Každý hráč má k dispozici pět kostek. Na počátku hry všichni hráči hodí svými kostkami a zakryjí je tak, aby na ně nemohli protihráči vidět. Pro první kolo se vybere náhodný hráč, který bude začínat. Následuje ho hráč po směru hodinových ručiček. Hráči během kola postupně navyšují sázku, to znamená kolik kostek stejného čísla na celé hrací ploše padlo (počítají se vlastní kostky i skryté kostky protihráčů). Kolo končí, když hráč nezvýší sázku ale nazve předchozího hráče lhářem. V ten moment se odkryjí všechny kostky a zjistí se, který hráč měl pravdu. Pokud byl stejný nebo větší počet kostek totožného čísla než bylo řečeno v poslední sázce, vyhrává hráč, který tuto sázku zahrál. Protihráč, který ho nazval neoprávněně lhářem ztrácí jednu kostku. V opačném případě ztrácí kostku hráč, jehož sázka byla nesprávná a vyhrává protihráč, který nazval tohoto hráče lhářem. Po konci kola je na řadě hráč, který prohrál. Hráč přestává hrát, když přijde o všechny své kostky. Hráč, který zůstává ve hře jako poslední vyhrává.
Na konci každého kola, kdy se odkrývají všechny kostky, se dá jednoduše zjistit, zda protihráči lhali při svých sázkách. Pro nejlepší výsledky v této hře, musí být hráči nepředvídatelní. Předvídatelného hráče, který nelže o počtu kostek stejného čísla na hrací ploše, lze snadno obehrát s využitím jeho pravdivých sázek. V opačném případě, předvídatelného hráče, který neustále lže, lze obehrát tak, že ho neustále nazýváme lhářem.

### 2.13.1 Pravidla hry

#### Zvyšování sázky

Varianty této hry se rozlišují podle pravidel pro zvyšování sázky.
Varianta číslo 1: Hráč může zvýšit počet kostek kteréhokoliv čísla kostky, nebo nechat stejný počet kostek, ale číslo kostky musí být větší než v předchozí sázce. Například, když byla předchozí sázka "2 * 3"(Padly dvě trojky), hráč může zvýšit sázku na "3 * 1"(Padly tři jedničky) a podobně.
Varianta číslo 2: Hráč může zvýšit počet kostek stejného čísla kostky, nebo zvolit jakýkoliv počet kostek pro vyšší čísla kostky. Například, pokud byla předchozí sázka "2 * 3", hráč může zvýšit sázku na "1 * 4"(Padla jedna čtyřka) a podobně.
Varianta číslo 3: Hráč může zvýšit alespoň počet kostek stejného čísla kostky, nebo číslo kostky stejného počtu kostek. Například, když předchozí sázka byla "2 * 3", hráč může zvýšit počet na "3 * 3"(Padly tři trojky) a podobně, nebo zvýšit číslo kostky na "2 * 4"(Padly dvě čtyřky) a podobně. V této variantě není povoleno snížit počet ani číslo (hodnotu) kostky.

#### Zvolená varianta

Pro tuto práci byla zvolena varianta číslo 3. Jednotlivá kola hry Liar's Dice probíhají rychleji s přísnějšími pravidly, kdy jsou hráči omezeni počtem i hodnotou kostek z poslední sázky. V této variantě je výpočetní náročnost výběru tahu zmenšena závisle na počtu povolených akcí. Rychlejší hry umožní náhled na účinnost algoritmu v širším kontextu.

# Kapitola 3

# Návrh Implementace

Cílem práce je navrhnout a implementovat metodu hraní hry Liar's Dice s využitím Dynamického programování, která by se vyrovnala současným algoritmům pro hraní této hry. Nově navržená metoda by měla být schopna hrát proti lidskému hráči.

## 3.1 Postup

V implementační části této práce se postupovalo systematicky, začínaje jednoduchou verzí uživatelského rozhraní v prostředí Unity, které umožnilo sledovat skóre jednotlivých hráčů. Hráči z počátku vybírali svůj tah náhodně z povolených akcí. Tento první prototyp byl důležitý pro ověření správného fungování základní herní mechaniky.

Následně byl kladen důraz na integraci algoritmů pro umělou inteligenci. Prvně se implementoval algoritmus SARSA pro jednoho z hráčů, zatímco druhý hráč stále volil své tahy náhodně. Toto umožnilo ověřit správnost implementace algoritmu a jeho schopnost řídit chování hráče v rámci hry.

Dalším krokem bylo rozšíření hry o algoritmus Q-learning, jenž umožnil hraní mezi dvěma umělými hráči. Zde se naskytla příležitost pozorovat interakci mezi dvěma různými algoritmy a zhodnotit jejich chování a výsledky.

Po několika experimentech mezi dvěma umělými hráči se zoptimalizovalo chování hráče používajícího algoritmus SARSA. Byl implementován mechanismus pro sběr dat o četnosti lhaní protivníka a využit k adaptaci strategie hráče. Tento proces umožnil vylepšit výkon hráče na základě reálných herních situací.

V poslední fázi byl algoritmus SARSA konfrontován s pokročilejším přístupem reprezentovaným algoritmem Counterfactual Regret Minimization. Algoritmus Counterfactual Regret Minimization pro hraní hry Liar's Dice byl však napsán v programovacím jazyce Python, takže bylo potřeba navrhnout a implementovat komunikaci mezi Python skriptem a herním prostředím Unity.

Celkově tato implementační fáze poskytla důkladný pohled na proces vývoje a testování herního prostředí s různými algoritmy umělé inteligence, a to jak v samotném Unity, tak i v jeho interakci s externími nástroji a algoritmy implementovanými v jiných programovacích jazycích.

## 3.2 Návrh aplikace

Aplikace bude až pro tři hráče. První hráč bude člověk, druhý hráč bude algoritmus SARSA a třetí hráč algoritmus Q-Learning později nahrazen pro testování za Counterfactual Regret minimization. Aplikace bude mít možnost odstranit ze hry lidského hráče pro testování algoritmů proti sobě. Hráči budou mít pouze dvě možnosti, zvýšit sázku (zvýšit kolik kostek jaké hodnoty padlo) a nebo nazvat předchozího hráče lhářem. Zvýšení sázky bude povoleno pouze na hodnoty, které povolují pravidla hry, v tomto případě jsou povoleny pouze sázky, které navyšují počet kostek nebo hodnotu kostky. Pokud bude navýšena hodnota padlých kostek, počet musí zůstat stejný nebo být navýšen. Uživatel bude moct zvolit svou akci za použití tlačítek v uživatelském rozhraní pro odpovídající akci.

## 3.3 Herní engine

### 3.3.1 Unity

#### Programovací jazyk

Unity využívá programovací jazyk C# jako hlavní jazyk pro vývoj her, což poskytuje jednoduché a intuitivní prostředí pro programátory. Daný engine má samozřejmě svoje knihovny, ke kterým poskytuje obsáhlou online dokumentaci. Znalost jazyka C# vzdaluje daného programátora pomocí abstrakce od složitých operacích, které se v C++ musí řešit. Například C# řeší za uživatele alokování paměti automaticky. Toto ovšem může vést k horšímu výkonu.

#### Grafický výstup

Unity může nabídnout kvalitní grafiku, ale pro rychlé prototypování lze využít méně náročných grafických nastavení, což usnadňuje běh aplikací na široké škále hardwaru. Toto je výhodné při testování konceptů na slabších zařízeních.

### 3.3.2 Unreal Engine

#### Programovací jazyk

Unreal Engine používá programovací jazyk C++ a Blueprinty, vizuální skriptovací nástroj. C++ může být složitější na učení, ale nabízí lepší kontrolu, výkon a také přístup přímo k paměti přes ukazatele. Blueprinty v Unreal Engine umožňují rychlý vývoj bez hlubokých znalostí programování, ale můžou vést k horšímu výkonu. Obvykle se používá kombinace C++ a Blueprintů, kde v kódu se implementují základní mechaniky a funkce a následně v Blueprintech se z nich staví složitější a obsáhlejší struktury.

#### Grafický výstup

Unreal Engine je známý svými vysokými standardy pro vizuální kvalitu, což může být problém pro prototypování. Vytváření vizuálně ohromujících prototypů je možné, ale vývojář musí mít zkušenosti v práci s grafickými prvky, které Unreal nabízí. Vizuální efekty a fotorealistická grafika většinou vyžaduje více zdrojů a je těžkopádnější na slabším hardwaru. Nicméně, vysoká kvalita vizuálních šablon a materiálů může pomoct prezentovat prototyp s lepším grafickým vzhledem.

### 3.3.3 Godot

#### Programovací jazyk

Herní engine Godot podporuje různé programovací jazyky pro vývoj her, především svůj vlastní jazyk GDScript a jazyky C++ a C#. Přídavnou funkcionalitou je GDNative, nástroj pro komunikaci se sdílenými knihovnami během běhu aplikace, bez potřeby kompilace kódu.

#### Grafický výstup

Godot zvládá běžné grafické úkony jako je například mapování normál, dynamické stíny při používání stínových map a dynamické globální osvětlení. Celkově je dobrou alternativou herního engine Unity.

### 3.3.4 Výběr Unity

Prototypování a testování herní implementace v Unity bylo přívětivější volbou pro toto téma. Unity nabízí nejen menší náročnost na zdroje, ale i propracovanou dokumentaci k třídám, které nabízí. Prostředí je uživatelsky přívětivé a snadno pochopitelné. Unity nabízí volně dostupné herní komponenty, které usnadňují práci pro uživatele. Široká podpora ze strany samotných vývojářů i komunity obohacuje daný ekosystém svými herními komponenty, nasazujícími moduly a jinými přídavnými balíčky.

## 3.4 Návrh stavového prostoru

Stavový prostor pro algoritmus SARSA bude pro jednoduchost a pro vytvoření Markovova řetězce složen pouze z poslední akce, která proběhla ve hře a z poslední sázky, která byla odehrána. To znamená, že stavový prostor bude rozdělen podle předchozí akce.

## 3.5 Návrh odměn a rychlosti učení

Pro posilované učení algoritmu musí být určeny odměny za akce, které provede podle toho do jakého stavu ho dostanou. Odměna za vítězství kola ,když nepřítel ztratí kostku, bude 1. Odměna za přežití, když se algoritmus dostane k další volbě akce bez ztráty kostky u obou hráčů, bude 0,5 a odměna za prohru kola, když algoritmus ztratí kostku, bude 0. Rychlost učení bude zpočátku nastavena na 0,5.

## 3.6 Návrh herní logiky

V této části práce byl kladen důraz na návrh herní logiky pro hru Liar's Dice, která bude umožňovat účast několika hráčů. Hráči budou hrát postupně po sobě, přičemž první hráč každého kola bude mít specifickou úlohu v určení počáteční sázky pro dané kolo.

Hráč, který kolo začne, má dostupnou pouze jedinou akci a to zvolit počáteční sázku. Toto rozhodnutí bude klíčové pro nastavení strategie pro dané kolo.

Následující hráči, kteří budou hrát po prvním hráči v daném kole, budou mít na výběr dvě akce: buď mohou zvýšit aktuální sázku, nebo mohou nazvat předchozího hráče lhářem. Tato dvojice akcí poskytuje hráčům flexibilitu v jejich strategiích a umožňuje jim reagovat na situaci na herním stole.

Pro účely implementace algoritmů umělé inteligence, které budou hrát proti lidským hráčům, bude nezbytné zajistit, aby měly přístup ke všem povoleným možnostem pro zvýšení sázky v daném kole. To znamená, že algoritmy musí být schopny analyzovat současný stav hry a možné kombinace kostek, aby se rozhodly jak budou reagovat.

Pro výpočet dostupných možností pro zvýšení nebo nastavení počáteční sázky bude dostačující znát počet kostek, které nebyly vyřazeny z hry. Tato informace poskytne algoritmům dostatečný kontext pro strategické rozhodování v rámci hry.

V této části návrhu herní logiky je důležité definovat pravidla pro určení pořadí hráčů a správnou logiku pro vyřazení hráče ze hry. Dále je také nutné specifikovat, jak budou algoritmy umělé inteligence používat dostupné informace k tomu, aby vybraly nejlepší možnou akci v daném kontextu.

Každý hráč bude mít přiřazeno číslo, které určuje jejich pořadí v prvním kole. Po každém kole bude hráč, který prohrál, začínat další kolo. Je důležité zajistit, aby v případě vyřazení hráče ze hry byl také vyloučen z pořadí ve hře, aby neovlivňoval budoucí průběh a pořadí hráčů.

Algoritmus, který je v daném kole na řadě, obdrží jako kontext všechny jeho povolené akce, včetně všech kombinací čísel, která může použít pro zvýšení sázky. Tento kontext je klíčový pro rozhodování algoritmu, protože mu poskytuje kompletní informace o současném stavu hry a možnostech, které má k dispozici. Algoritmy umělé inteligence se budou zaměřovat na to, aby na základě poskytnutého kontextu vybraly akci s největší pravděpodobností úspěchu. Tento přístup umožní algoritmům reagovat na dynamiku hry a optimalizovat své rozhodování v každém kole.

Pro úspěšnou implementaci algoritmů umělé inteligence je nezbytné zajistit, že budou mít přístup k paměti pro ukládání hodnot Q, které vyjadřují potenciální odměny za výběr akcí z daných stavů. Pro tento účel bude nutné provést serializaci dat, aby bylo možné je efektivně ukládat a načítat.

Před zahájením hry budou algoritmy muset načíst tato data, aby měly přístup k uloženým hodnotám Q a mohly je využít při strategickém rozhodování.

V průběhu hry bude důležité kontinuálně aktualizovat a ukládat serializovaná data, aby se algoritmy mohly přizpůsobovat měnícím se podmínkám a optimalizovat své rozhodování na základě aktuálního stavu hry. To zahrnuje aktualizaci hodnot Q na základě nových zkušeností získaných během hraní a optimalizaci strategie v souladu s těmito informacemi.

Celkově bude návrh herní logiky pro hru Liar's Dice představovat klíčový rámec pro implementaci a testování algoritmů umělé inteligence, stejně jako pro interakci s lidskými hráči v prostředí Unity.

## 3.7 Návrh reprezentace dat

Pro efektivní ukládání a reprezentaci dat Q je důležité zvolit si strukturu, která umožní snadný přístup k hodnotám Q na základě předchozího stavu a provedené akce. Pro tento účel bude výhodné vytvořit pole třídy, která definuje prvek složený ze stavu hry, z akce která byla zvolena a nakonec z hodnoty Q.

Tímto způsobem se mohou snadno ukládat a získávat hodnoty Q pro libovolnou kombinaci předchozího stavu a provedené akce, což umožní algoritmům umělé inteligence efektivně vyhodnocovat a aktualizovat své strategie v průběhu hry. Tento návrh reprezentace dat zajistí, že algoritmy budou schopny efektivně pracovat s uloženými hodnotami Q a optimalizovat své rozhodování na základě aktuálního stavu hry.

## 3.8 Návrh Serializace dat

Pro efektivní serializaci dat se bude muset rozdělit třída, která definuje předchozí stav hry, akci a hodnotu Q, na jednotlivé části. To umožní jednoduché a systematické ukládání a načítání těchto dat. Každá část bude reprezentována jako pole pro dané informace.

Konkrétně bude potřeba vytvořit tři pole:

- Pole hodnot Q: Toto pole bude obsahovat hodnoty Q.
- Pole stavů hry: Toto pole bude obsahovat informace o předchozím stavu hry pro každou hodnotu Q.
- Pole akcí: Toto pole bude obsahovat informace o provedené akci pro každou hodnotu Q.

Všechna tato pole budou mít stejnou délku a budou indexována stejným způsobem, aby bylo možné snadno provádět operace jako načítání a ukládání dat.

Tento přístup k serializaci dat umožní efektivní ukládání a načítání hodnot Q a souvisejících informací a zajistí zachování integrity dat během procesu serializace a deserializace.

# Kapitola 4

# Implementace

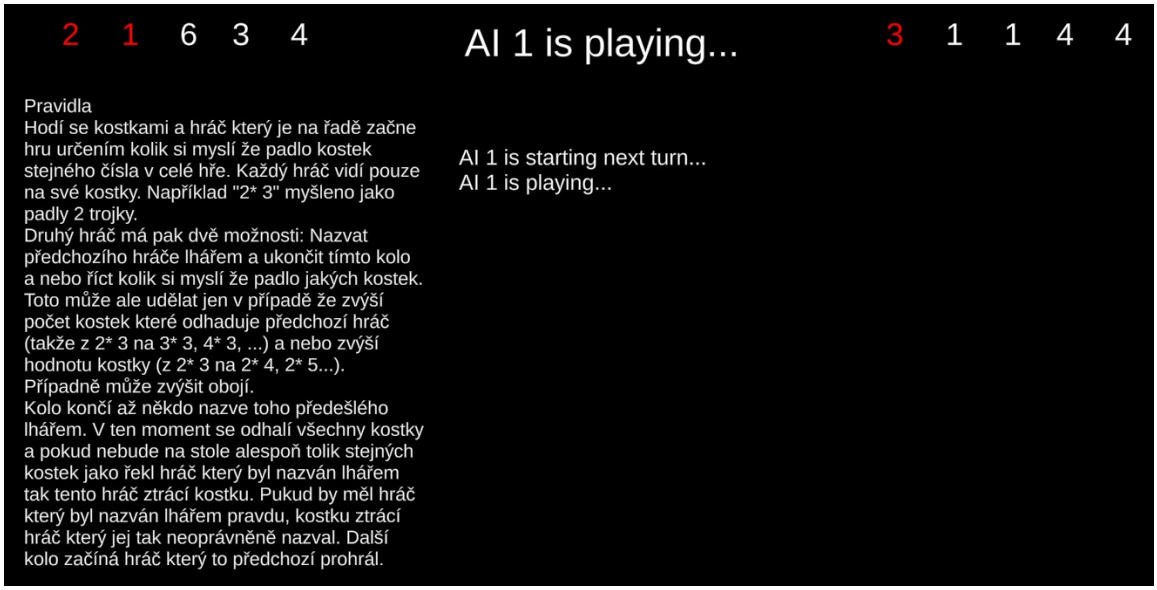
Tato kapitola řeší implementaci hry Liar's Dice v Unity engine a implementace umělé inteligence do této hry za pomocí algoritmu SARSA.

## 4.1 Implementace Liar's Dice v Unity

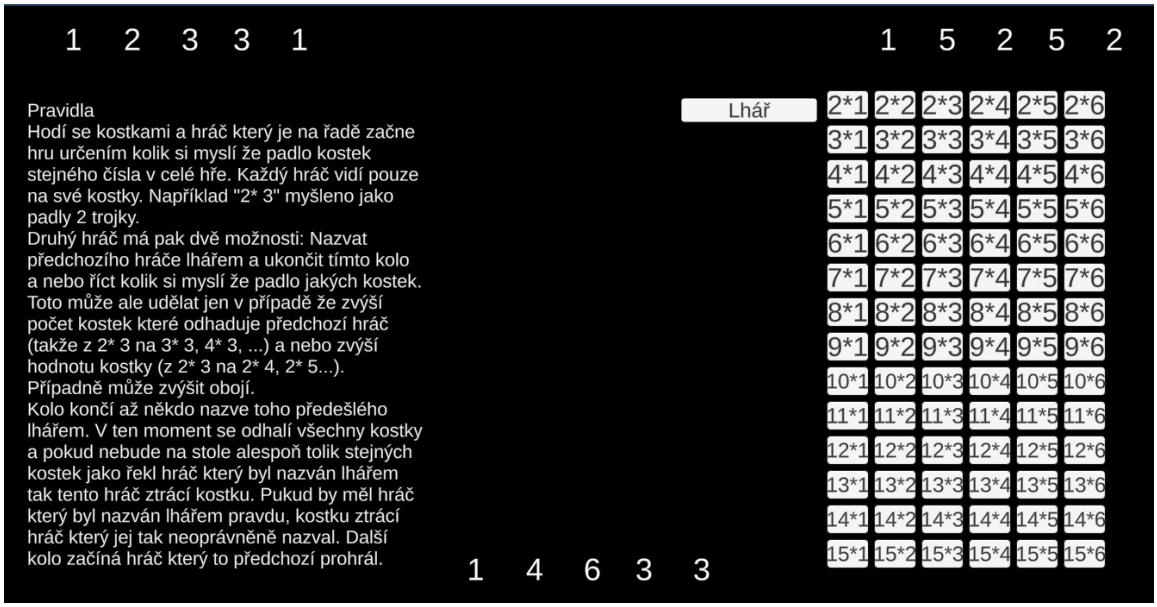
Výsledek implementace hry Liar's Dice v Unity engine je rozdělen na Uživatelské rozhraní, Pravidla, Hráče, Kostky a Umělá Inteligence.

### 4.1.1 Uživatelské rozhraní

Uživatelské rozhraní se skládá ze tří skupin po pěti čísel, zobrazujících hodnoty na padlých kostkách pro každého hráče, text s pravidly hry na levé straně, text nahoře uprostřed, zobrazující informace o posledním tahu a o tom, kdo má právě hrát. Pro uživatele se zobrazí tlačítka uprostřed obrazovky vždy, když má odehrát další tah. Čísla, která jsou zobrazena červenou barvou jsou neplatná a znázorňují ztracené kostky.




Obrázek 4.1: Ukázka uživatelského rozhraní




Obrázek 4.2: Ukázka uživatelského rozhraní hlavního hráče

### 4.1.2 Pravidla

Na začátku každého kola se zavolá funkce, která náhodně vygeneruje čísla od 1 do 6 pro všechny kostky všech hráčů. Hráči se střídají v tom, kdo je na tahu, po směru hodinových ručiček. Hra zvolí hráče, který je na řadě a ten se chová podle toho zda-li hraje uživatel nebo umělá inteligence. Uživatel má na výběr jednotlivé tlačítka, která odpovídají zvýšení sázky na odpovídající hodnoty a tlačítko pro nazvání předchozího hráče lhářem. Pokud je uživatel ve hře, hra počká na výběr vstupu uživatele. Umělá inteligence volá své výpočetní funkce a algoritmy pro rozhodnutí svého tahu. Hra předá umělé inteligenci pouze hodnoty kostek, které má umělý hráč k dispozici a všechny dostupné tahy, které může umělá inteligence vykonat. Hráč, který začíná kolo nemůže nikoho nazvat lhářem. Po zvýšení sázky se omezí počet akcí na další zvýšení sázky podle pravidel hry. Zvýšení sázky musí zvýšit počet kostek na větší než ten, který odhaduje předchozí hráč nebo zvýšit hodnotu kostky. Kolo končí, až někdo nazve toho předešlého hráč lhářem. V ten moment se odhalí všechny kostky a pokud nebude na stole alespoň tolik stejných kostek jako řekl hráč, který byl nazván lhářem, tak tento hráč ztrácí kostku. Pokud by měl hráč, který byl nazván lhářem pravdu, kostku ztrácí hráč, který jej tak neoprávněně nazval. Další kolo začíná hráč, který to předchozí prohrál. Pseudokód: [fragile]

```
pripravAlgoritmy()
hraci = vsichniHraci()
kdoHraje = 0
liarsDice = zacatekHry()

void OnUpdate()
{
    hraje = hraci[kdohraje]
    if(hraje != Uzivatel)
        hraje.hraj()
}
Class hrac{
    string typHrace
```

```
14      AI ai
15      void hraj()
16      {
17          if(typHrace == "AI")
18          {
19              akce = null
20              if(ai.typ == "SARSA")
21              {
22                  akce = ai.vyberAkci(liarsDice.dostupneakce)
23                  ai.aktualizujQ(ai.posledniAkce, akce.hodnotaQ, ai.odmenaZaPosledniAkci)
24              }
25              else if(ai.typ == "Qlearning")
26              {
27                  akce = ai.vyberAkci(liarsDice.dostupneAkce)
28                  nejvetsiQ = ai.nejvetsiQ(liarsDice.dostupneAkce)
29                  ai.aktualizujQ(ai.posledniAkce, nejvetsiQ, ai.odmenaZaPosledniAkci)
30              }
31
32              if(akce != null)
33              {
34                  if(akce.typ == "Lhar")
35                  {
36                      lhar(hrac)
37                      kdohraje = kdoProhral()
38                  }
39                  else
40                  {
41                      zvysitSazku(sazka, hrac)
42                      kdoHraje = dalsiHrac()
43                  }
44              }
45          }
46      }
47  }
48
```

### 4.1.3 Hráči

Každý hráč má jméno, pět kostek a vlastnost, která značí zda-li přišel o všechny kostky. Hráči s umělou inteligencí, obsahují navíc skript umělé inteligence, který se stará o tahy daného hráče. V Unity engine byla naprogramována umělá inteligence s algoritmem SARSA a s algoritmem Q-Learning. Pro algoritmus Counterfactual Regret Minimization byl navázán existující program, napsaný v Python, na rozhraní v Unity.

### 4.1.4 Kostky

Kostky obsahují text, který je zobrazen v uživatelském rozhraní a referenci na instanci kostky. Instance kostky obsahuje hodnotu kostky a vlastnost určující zda-li je kostka aktivní (zda byla ztracena ve hře). Instance kostky má pouze jednu funkci, která vygeneruje náhodné číslo od 1 do 6 a přiřadí jej do hodnoty kostky.

## 4.2 Implementace algoritmu SARSA do hry Liar's Dice

### 4.2.1 SARSA

SARSA je implementována jako třída, která obsahuje všechny potřebné vlastnosti a funkce pro rozhodovací proces. Tato třída obsahuje nastavitelné hodnoty $\alpha$, $\gamma$, odměna za vítězství, odměna za přežití, odměna za prohru a vlajku zda-li se má použít při výpočtu Q hodnoty rovnice ze SARSA nebo z Q-Learning.
Aby vytvořená umělá inteligence nezapomínala hodnoty Q po dokončení hry, zapisuje všechny hodnoty Q do souboru, kde je v následujícím kole načte zase zpátky. Algoritmy SARSA a Q-Learning mají separátní soubor pro své hodnoty Q
Pro výběr následujícího tahu bylo navrženo, že z možných následujících tahů se náhodně vybere jeden na základě procentuální šance. Procentuální šance na výběr daného tahu je velikost hodnoty Q vydělena celkovým součtem hodnot Q z možných tahů.

### 4.2.2 Qhodnota - SARSA

Pro výpočet hodnoty Q je potřeba se dostat alespoň do druhého tahu dané umělé inteligence. Pro algoritmus SARSA je také potřeba vědět hodnotu Q následující akce. Výpočet hodnoty Q v algoritmu SARSA má tuto rovnici:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{4.1}$$

kde $Q_{new}(s_t, a_t)$ je hodnota Q v předchozím stavu $s_t$ a s vybranou akcí v předchozím stavu $a_t$. Symbol $\alpha$ je rychlost učení v rozmezí od nuly do jedné. Rychlost učení mění, jak rychle se algoritmus bude učit nové hodnoty Q. Symbol $\gamma$ znázorňuje, jak moc má algoritmus přihlížet k hodnotě Q v současném stavu $s_{t+1}$ s vybranou akcí $a_{t+1}$v současném stavu. Proměnná $\gamma$ má rozmezí od nuly do jedné. Proměnná $r_t$ je odměna za akci $a_t$ z předchozího stavu.

### 4.2.3 Qhodnota - Q-Learning

Pro výpočet hodnoty Q je potřeba se dostat alespoň do druhého tahu dané umělé inteligence. Výpočet hodnoty Q v algoritmu Q-Learning značí tato rovnice:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma maxQ(s_{t+1}, a) - Q(s_t, a_t)] \tag{4.2}$$

Na rozdíl od algoritmu SARSA je potřeba znát maximální hodnotu Q z možných akcí v současném stavu, v rovnici označeno jako $maxQ(s_{t+1}, a)$.

### 4.2.4 Stavový prostor

Každý stav ve stavovém prostoru pro algoritmy SARSA a Q-Learning je určen akcí a předchozím stavem. To znamená, že stav je definován podle toho, která akce a v jakém stavu vedla do stavu současného. Q hodnoty jsou za běhu programu uloženy v poli. Pro přístup k dané hodnotě Q je potřeba předat právě předchozí stav a akci, která způsobila přesun ze stavu předchozího do toho současného.

### 4.2.5 Výběr tahu - SARSA

Pro výběr tahu v algoritmu SARSA se z množiny povolených následujících tahů náhodně vybere jeden na základě hodnoty Q. Každá hodnota Q z povolených akcí je převedena

na procentuální šanci, která určuje pravděpodobnost výběru daného tahu a je založena na podílu hodnoty Q daného tahu a celkového součtu hodnot Q z povolených tahů. Pro pravděpodobnost výběru tahu platí:

$$P(a) = \frac{Q^{SARSA}(s_t, a)}{\sum_{a_p \in A} Q^{SARSA}(s_t, a_p)}$$

kde $P(a)$ je pravděpodobnost výběru akce $a$, $Q^{SARSA}(s_t, a)$ je hodnota Q ve stavu $s_t$ pro akci $a$.$A$ je množina všech povolených akcí ve stavu $s_t$, $a_p$ je označení pro jednotlivé akce z množiny $A$ a výraz $Q^{SARSA}(s_t, a_p)$ vyjadřuje hodnotu Q pro stav $s_t$ s povolenou akcí $a_p$.

Při každém výběru tahu v algoritmu SARSA se používá tento mechanismus pro určení nejvhodnějšího tahu, který bude prováděn v dané situaci. Tímto způsobem postupně zlepšuje své rozhodování a adaptuje svou strategii na základě získaných zkušeností a hodnot Q, které jsou aktualizovány v průběhu hry.

Celkově tento mechanismus výběru tahu umožňuje algoritmu SARSA efektivně zkoumat a objevovat strategické možnosti v rámci hry a postupně optimalizovat své chování na základě získaných informací a zkušeností.

### 4.2.6 Výběr tahu - Q-Learning

Výběr tahu pro algoritmus Q-Learning byl implementován velmi podobně jako u algoritmu SARSA. Jediná změna je, že se používají hodnoty Q algoritmu Q-Learning. Pro pravděpodobnost výběru akce platí tato rovnice:

$$P(a) = \frac{Q^{Q-Learning}(s_t, a)}{\sum_{a_p \in A} Q^{Q-Learning}(s_t, a_p)}$$

kde $P(a)$ je pravděpodobnost výběru akce $a$, $Q^{Q-Learning}(s_t, a)$ je hodnota Q algoritmu Q-Learning ve stavu $s_t$ pro akci $a$.$A$ je množina všech povolených akcí ve stavu $s_t$, $a_p$ je označení pro jednotlivé akce z množiny $A$ a výraz $Q^{Q-Learning}(s_t, a_p)$ vyjadřuje hodnotu Q pro stav $s_t$ s povolenou akcí $a_p$.

### 4.2.7 Výběr tahu - Strategie

Pro výběr tahů u obou algoritmů bylo použito pravděpodobnostní rozložení na základě hodnoty Q daného algoritmu. To zajistí, že umělá inteligence obou hráčů bude spíše explorativní než chamtivá. V momentě, kdy bude některá akce výrazně lepší než ostatní, je velká pravděpodobnost, že tato akce bude vybrána. Stále je ale zde ponechán prostor pro exploraci, který vede k větší nepředvídatelnosti algoritmu. Strategie bude vždy spíše chamtivá u algoritmu Q-Learning, protože hodnoty Q jsou aktualizovány s potencionálně nejvyšší možnou odměnou, kterou mohl hráč vykonat. U algoritmu SARSA se tyto hodnoty aktualizují podle reálné odměny, která byla získána za předešlou akci a bude tedy směřovat spíše ke strategii explorativní.

### 4.2.8 Serializace a ukládání dat

Důležitá data pro funkční algoritmus SARSA musí být uložena, aby algoritmus neztrácel kontext a získané zkušenosti mezi jednotlivými hry. Nejdůležitější data pro rozhodování jsou hodnoty Q. Pro jejich uložení na paměť je potřeba je nejdříve serializovat. Serializace rozdělí pole třídy Q a rozdělí jej na pole stavů, pole akcí a samotné pole hodnot Q. Tyto

hodnoty jsou pak ve formátu JSON uloženy do souboru. Pro překlad serializovaných dat do formátu JSON byla použita funkce ToJson z knihovny JsonUtility.

## 4.3 Modifikace algoritmu SARSA

Po několika pokusech hraní SARSA proti Q-Learning bylo zjištěno, že je potřeba algoritmus SARSA ještě modifikovat. Bylo adekvátní soustředit se na tahy, ve kterých protihráč lže. Sázky byly rozdělené na malé, střední a velké podle hodnoty kostky a počtu kostek. Na konci kola, kdy se odhalí všechny kostky je program schopen spočítat kolikrát nepřítel zalhal v jaké sázce. Tyto procentuální hodnoty jsou následně využity při vybírání následujícího tahu.

### 4.3.1 Implementace modifikace

Po odehrání každého tahu nepřítele je uložena jeho akce do pole. Na konci kola, až budou všechny kostky odhaleny, se porovnají jednotlivé akce protihráče se skutečnými hodnotami kostek.
Vysoké sázky jsou takové, kdy sázka obsahuje počet kostek větší než 2/3 celkového počtu aktivních kostek na stole, nebo sázka obsahovala hodnoty kostek 5 a 6.
Střední sázky jsou takové, kdy sázka obsahuje počet kostek větší než 1/3 a menší než 2/3 z celkového počtu aktivních kostek na stole, nebo sázka obsahovala hodnoty kostek 3 a 4.
Malé sázky jsou takové, kdy sázka obsahovala počet kostek menší než 1/3 celkového počtu aktivních kostek na stole, nebo sázka obsahovala hodnoty kostek 1 a 2.
Pokud sázka splňuje podmínky dvou kategorií (například počet kostek sázky je větší než 2/3 celkového počtu a zároveň hodnoty kostek jsou 2), platí pouze nejvyšší kategorie.

### 4.3.2 Výběr tahu s modifikací

Při výběru tahu se chová modifikovaný algoritmus SARSA velmi podobně jako původní verze. Následující tah se náhodně vybere na základě procentuální šance. Procentuální šance na výběr daného tahu je velikost hodnoty Q vydělena celkovým součtem hodnot Q z možných tahů výjimkou pro akci: Nazvat nepřítele lhářem. Pro výpočet procentuální šance pro tuto akci platí, že její Q hodnota je pouze pro výpočet modifikována tako:

$$Q^{tmp}(s_t, a_{liar}) \leftarrow Q(s_t, a_{liar}) + L(c)max(Q(s_t, a)) \times 2 \tag{4.3}$$

kde $Q^{tmp}(s_t, a_{liar})$ je dočasná hodnota Q pro výpočet procentuální šance. Proměnná $s_t$ je současný stav a $a_{liar}$ je akce: Nazvat protihráče lhářem. Symbol $L(c)$ znázorňuje procentuální hodnotu zda bude protihráč v kategorii "c"lhát. Maximální hodnota Q v současném stavu $s_t$ z povolených akcí "a"je značena jako $max(Q(s_t, a))$ v rovnici. Pro pravděpodobnost výběru akce poté platí tato rovnice:

$$P(a) = \frac{Q^{tmp}(s_t, a)}{\sum_{a_p \in A} Q^{tmp}(s_t, a_p)}$$

### 4.3.3 Obnova profilu lhaní

Aby profil lhaní protihráče nezastaral, modifikovaný algoritmus SARSA obnoví počáteční hodnoty profilu každých 100 odehraných kol.

### 4.3.4 Serializace a ukládání dat s modifikací

Upravený algoritmus musí nově serializovat a ukládat profil lhaní pro zachování kontextu. Profil lhaní se rozloží na vysoké, střední a nízké sázky. Pro každou sázku se zvlášť uloží procentuální šance na to, že nepřítel bude lhát pro danou sázku a počet kolikrát již nepřítel lhal.

## 4.4 Komunikace s Counterfactual Regret Minimization

Pro implementaci dalšího protivníka do projektu byl zvolen program napsaný v Pythonu, který implementuje algoritmus Counterfactual Regret Minimization speciálně pro hru Liar's Dice. Tento algoritmus se využívá pro vývoj strategií ve hrách s neúplnými informacemi, jako je poker nebo právě Liar's Dice. Po důkladném prozkoumání dostupného aplikačního rozhraní programu bylo však zjištěno, že toto rozhraní neposkytuje některé podstatné funkce, které jsou nezbytné pro komunikaci s programem napsaným v Unity.

Z tohoto důvodu byla provedena důkladná analýza zdrojového kódu programu pro Counterfactual Regret Minimization. Cílem bylo identifikovat, jak jsou jednotlivé funkce implementovány a jak mohu systém rozšířit o dodatečné funkcionality. Během této analýzy bylo zaznamenáno několik klíčových oblastí, které vyžadovaly rozšíření nebo úpravu, aby bylo možné efektivně provést integraci s vlastním programem.

Jedním z hlavních úkolů bylo vytvoření nové funkce, která umožňuje programu Counterfactual Regret Minimization komunikovat s algoritmem SARSA. Tato funkce zajistila správný přenos stavů a rozhodnutí mezi oběma algoritmy, aby mohla být strategie postupně upravována a optimalizována. Toto zahrnovalo vytvoření nových metod pro přenos výběru akce v reálném čase.

Výsledkem těchto úprav je robustnější systém, který nejen zvládá základní funkce původního programu, ale také efektivně komunikuje s upraveným programem pro algoritmus SARSA.

# Kapitola 5

# Testování

Testování probíhalo přes vlastně napsanou aplikaci v Unity engine. Pro testování algoritmů SARSA a Q-Learning byly oba algoritmy v této aplikaci implementovány a následně ponechány hrát proti sobě. Program byl nastaven tak, aby po dokončení hry, kdy jeden hráč přijde o všechny kostky, se ve formě návratové hodnoty programu vracelo vyhodnocení, kdo vyhrál a kolik kostek mu zbylo.

Pro testování algoritmů SARSA a Counterfactual Regret Minimization byla propojena vlastní aplikace napsaná v Unity engine se skriptem existujícího řešení Counterfactual Regret Minimization a opět ponechány hrát proti sobě. Výsledky byly uloženy do textových souborů a porovnávany s napsanými prototypy upraveného algoritmu SARSA.

Pro sběr výsledků experimentů, byl napsán jednoduchý kód v programovacím jazyce Python. Tento program se staral o spouštění aplikace v Unity a interpretoval výstupní hodnoty pro výpočet a záznam výsledků z odehraných her. Stejný software byl použit pro první dvě části experimentu, jak pro hru SARSA proti Q-Learning, tak i pro modifikovaný algoritmus SARSA proti Q-Learning.

## 5.1 SARSA proti Q-Learning

V prvním prototypu nemodifikované verze algoritmu SARSA proti algoritmu Q-Learning bylo dosáhnuto úspěšnosti přibližně 50 %
Získaná data byla tato při první iteraci:
Počet her: 11
z toho vyhrál SARSA her: 4
z toho prohrál SARSA kol: 43
z toho vyhrál Q-Learning her: 7
z toho prohrál Q-Learning kol: 34

Získaná data při druhé iteraci:
Počet her: 84
z toho vyhrál SARSA her: 46
z toho prohrál SARSA kol: 290
z toho vyhrál Q-Learning her: 38
z toho prohrál Q-Learning kol: 317
Z výsledných dat lze vyčíst, že algoritmus SARSA nejdříve vyrovnal a následně získal mírný

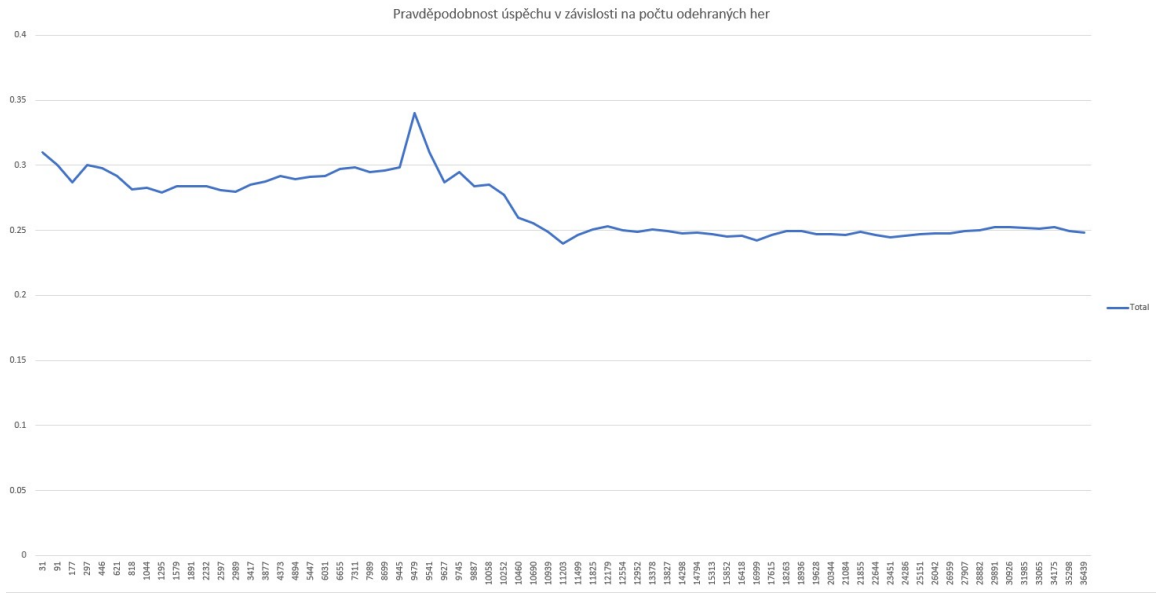
náskok v počtu vyhraných her. Algoritmy získávaly podobné zkušenosti a byly postaveny na stejném počátečním modelu hodnot Q. To byl zřejmě zdroj takto vyrovnané hry.

## 5.2 SARSA s modifikací proti Q-Learning

Ve hře proti algoritmu Q-Learning bylo dosáhnuto úspěšnosti 69,147 %.
Získaná data při první iteraci:
Počet her: 2310
z toho vyhrál SARSA her: 1610
z toho prohrál SARSA kol: 6461
z toho vyhrál Q-Learning her: 700
z toho prohrál Q-Learning kol: 9824
V první iteraci, která trvala přibližně deset hodin, byl upravený algoritmus SARSA schopen získat významnou převahu s procentuální šancí 69,69 % na výhru. Profilování lhaní se prokázalo jako úspěšná modifikace algoritmu SARSA pro hraní hry Liar's Dice.
Získaná data při druhé iteraci:
Počet her: 2198
z toho vyhrál SARSA her: 1447
z toho prohrál SARSA kol: 6282
z toho vyhrál Q-Learning her: 751
z toho prohrál Q-Learning kol: 9127
Během druhé iterace šance na vítězství implementovaného algoritmu klesla na 65,83 %. Upravená SARSA volila spíše explorativní akce za účelem získání potřebných informací o stavovém prostoru. To vedlo k poklesu procentuální šance na výhru. Iterace opět trvala přibližně deset hodin.
Získaná data při třetí iteraci:
Počet her: 1008
z toho vyhrál SARSA her: 697
z toho prohrál SARSA kol: 2624
z toho vyhrál Q-Learning her: 311
z toho prohrál Q-Learning kol: 4245
Po třetí iteraci se algoritmus SARSA stabilizoval přibližně na hodnotě 69,14 % pravděpodobnosti vítězství. Z těchto dat bylo usouzeno, že algoritmus SARSA s profilováním lhaní nepřítele, je v dostatečné výhodě proti algoritmu Q-Learning.

## 5.3 SARSA s modifikací proti Counterfactual Regret Minimization

Pro experiment, kde byl porovnáván upravený algoritmus SARSA proti Counterfactual Regret Minimization, byl upraven program pro sbírání výsledků hry, aby monitoroval a průběžně ukládal výsledná data, každých deset her. Ve hře proti Counterfactual Regret Minimization bylo dosáhnuto úspěšnosti pouze 25 % Graf 5.1 znázorňuje data sbíraná každou desátou hru. Na počátku lze vidět měnící se pravděpodobnost v době, kdy se modifikovaný algoritmus SARSA pokoušel získávat zkušenosti ve hře proti novému oponentu. Je zde zaznamenán i krátkodobý výkyv velkého zlepšení, avšak postupem času se pravděpodobnost ustálila přibližně na hodnotě 25 % pro výhru upraven0ho algoritmu SARSA.




Obrázek 5.1: Graf závislosti pravděpodobnosti úspěchu SARSA proti Counterfactual Regret Minimization

# Kapitola 6

# Závěr

Cílem této práce bylo navrhnout a implementovat metodu hraní hry Liar's Dice s využitím dynamického programování, která by se vyrovnala současným metodám hraní této hry. Tento cíl se podařilo částečně naplnit realizací algoritmu SARSA s modifikací pro sledování, kdy protihráč lže. Současné metody hraní hry Liar's Dice, jsou algoritmy Q-Learning a Counterfactual Regret Minimization. S počátečním prototypem algoritmu SARSA bylo při testování zjištěno, že samotný algoritmus nebude dostačující, a proto vznikla potřeba tento algoritmus modifikovat.
Pro zhotovení této práce bylo nutno nastudovat si pečlivě pravidla hry Liar's Dice a seznámit se možnými přístupy pro hraní her s neurčitostí. Následně byla naprogramována počítačová verze této hry v programu Unity a navrhnuta implementace algoritmu SARSA. Rozvržení stavového prostoru bylo inspirováno Markovovými řetězci.
Realizace modifikovaného algoritmu SARSA, využívá algoritmus SARSA pro výpočet hodnot Q (potencionální odměna za přesun do daného stavu). Pro výběr následující akce se náhodně vybere jeden tah, podle procentuální šance. Procentuální šance každého možného tahu je dána podílem hodnoty Q daného tahu se součtem všech hodnot Q z možných tahů. Pro akci nazvání protihráče lhářem se navíc tato hodnota Q upravuje pomocí zmíněné rovnice 4.3. Tímto způsobem bude modifikovaný algoritmus SARSA, více trestat protihráče za lhaní.
Touto realizací modifikovaného algoritmu SARSA, bylo dosáhnuto úspěšnosti 69,147 % ve hře proti algoritmu Q-Learning a úspěšnosti pouze 25 % proti algoritmu Counterfactual Regret Minimization. Implementované řešení je mnohem lepší metoda, než existující algoritmus Q-Learning, ale není nejlepší současnou metodou hraní hry Liar's Dice.
Tato práce byla přínosem v oblasti řešení her s neurčitostí. Popisuje získané zkušenosti a pohled na různé metody pro hraní her s neurčitostí. Rozšiřuje znalosti o posilovaném učení a umělé inteligenci.

Pro budoucí postup se navrhuje prozkoumat další algoritmy posilovaného učení s modifikací, která byla použita a testována na algoritmu SARSA. Profilování lhaní by mohlo přinést zajímavé výsledky při použití se zmíněným algoritmem Counterfactual Regret Minimization. Naskytuje se zde i příležitost prozkoumat další neuronové sítě pro hraní hry Liar's Dice.

# Literatura


[1] *State–action–reward–state–action — Wikipedia, The Free Encyclopedia* [online]. 2022 [cit. 2023-9-05]. Dostupné z: https://en.wikipedia.org/wiki/StateâĂŞactionâĂŞrewardâĂŞstateâĂŞaction.

[2] *Unreal Engine — Wikipedia, The Free Encyclopedia* [online]. 2024 [cit. 2024-7-05]. Dostupné z: https://en.wikipedia.org/wiki/Unreal_Engine.

[3] *Unity (game engine) — Wikipedia, The Free Encyclopedia* [online]. 2023 [cit. 2023-9-05]. Dostupné z: https://en.wikipedia.org/wiki/Unity_(game_engine).

[4] Bonanno, G. *Game Theory*. 2. vyd. Kindle Direct Publishing, 2018. ISBN ISBN-13: 978-1983604638, ISBN-13: 978-1985862517. Dostupné z: https://faculty.econ.ucdavis.edu/faculty/bonanno/PDF/GT_book.pdf.

[5] Bradley, S. P., Hax, A. C. a Magnanti, T. L. *Applied Mathematical Programming*. Addison-Wesley, 1977. ISBN 978-0201004649. Dostupné z: https://web.mit.edu/15.053/www/AMP-Chapter-11.pdf.

[6] Costikyan, G. *Uncertainty In Games*. The MIT Press, 2013. ISBN 978-0-262-01896-8.

[7] Neller, T. W. a Lanctot, M. *An Introduction to Counterfactual Regret Minimization* [online]. 2013 [cit. 2023-9-05]. Dostupné z: http://modelai.gettysburg.edu/2013/cfr/cfr.pdf.

[8] Norris, J. *Markov Chains* [online]. [cit. 2023-9-05]. Dostupné z: http://www.statslab.cam.ac.uk/~rrw1/markov/M.pdf.

[9] Rummery, G. A. a Niranjan, M. *ON-LINE Q-LEARNING USING CONNECTIONIST SYSTEMS* [online]. 1994 [cit. 2023-9-05]. Dostupné z: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.17.2539&rep=rep1&type=pdf.

[10] Russell, S. a Norvig, P. *Artificial Intelligence: A Modern Approach*. Third. Prentice Hall, 2010. ISBN ISBN-13: 978-0-13-604259-4, ISBN-10: 0-13-604259-7. Dostupné z: https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf.

[11] Satariano, A. a Kang, C. THE A.I. RACE. *The New York Times*. Prosinec 2023, č. 3. Dostupné z: https://www.nytimes.com/2023/12/06/technology/ai-regulation-policies.html.

[12] Sutton, R. S. a Barto, A. G. *Reinforcement Learning:: An Introduction* [online]. Second. The MIT Press, 2015. Dostupné z: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf.

[13] Tokic, M. a Palm, G. *Value-Difference based Exploration: Adaptive Control between epsilon-Greedy and Softmax*. 2011. Dostupné z: https://www.tokic.com/www/tokicm/publikationen/papers/KI2011.pdf.

[14] TURING, A. M. I.—COMPUTING MACHINERY AND INTELLIGENCE. *Mind*. Říjen 1950, LIX, č. 236, s. 433–460. DOI: 10.1093/mind/LIX.236.433. ISSN 0026-4423. Dostupné z: https://doi.org/10.1093/mind/LIX.236.433.

# Příloha A

# Zdrojový kód v python pro výpočet statistik

[fragile]

```
import subprocess

gamesCount = 0
numberOfRounds = 0
sarsaWon = 0
sarsaLostRound = 0
qLostRounds = 0
qWon = 0

sarsaWonLast100 = 0
numberOfRoundsLast100 = 0
sarsaLostRoundLast100 = 0
CRMLostRoundsLast100 = 0
CRMWonLast100 = 0

def saveMePlease():
  f = open("stats.txt", "w")
  f.write("gamesCount: " + str(gamesCount) + "\n")
  f.write("sarsaWon: " + str(sarsaWon) + "; percent: " + str(sarsaWon/gamesCount*100) + "%\
    n")
  f.write("sarsaLostRounds: " + str(sarsaLostRound) + "; percent: " + str(sarsaLostRound/
    numberOfRounds*100) + "%\n")
  f.write("qWon: " + str(qWon) + "; percent: " + str(qWon/gamesCount*100) + "%\n")
  f.write("qLostRounds: " + str(qLostRounds) + "; percent: " + str(qLostRounds/
    numberOfRounds*100) + "%\n")
  f.close()

def saveLearningRate():
  f2 = open("learningStats.txt", "a")
  f2.write("----------------------------"
        + str(gamesCount) + "----------------------------------------\n")
  f2.write("sarsaAllWon: " + str(sarsaWon) +
        "; percent: " + str(sarsaWon/gamesCount*100) + "%\n")
  f2.write("CRMAllWon: " + str(qWon) +
        "; percent: " + str(qWon/gamesCount*100) + "%\n")
  f2.write("Last 100 Games:\n")
  f2.write("sarsaWon: " + str(sarsaWonLast100) +
        "; percent: " + str(sarsaWonLast100) + "%\n")
```

```
  f2.write("sarsaLostRounds: " + str(sarsaLostRoundLast100) +
        "; percent: " + str(sarsaLostRoundLast100/numberOfRoundsLast100*100) + "%\n")
  f2.write("CRMWon: " + str(CRMWonLast100) +
        "; percent: " + str(CRMWonLast100) + "%\n")
  f2.write("CRMLostRounds: " +
        str(CRMLostRoundsLast100) + "; percent: " +
        str(CRMLostRoundsLast100/numberOfRounds*100) + "%\n")
  f2.close()

try:
  while 1:
    returnValue = subprocess.run(['..\\build\\LiarAI.exe']).returncode
    print(returnValue)
    if (returnValue < 20) and (returnValue >= 10):
      sarsaWon += 1
      sarsaLostRound += 5 - returnValue % 10
      qLostRounds += 5
      # CRM
      sarsaWonLast100 += 1
      sarsaLostRoundLast100 += 5 - returnValue % 10
      CRMLostRoundsLast100 += 5
    elif (returnValue >= 20) and (returnValue < 30):
      qWon += 1
      qLostRounds += 5 - returnValue % 10
      sarsaLostRound += 5
      # CRM
      CRMWonLast100 += 1
      CRMLostRoundsLast100 += 5 - returnValue % 10
      sarsaLostRoundLast100 += 5
    else:
      continue

    numberOfRounds += 5 + (5 - returnValue % 10)
    numberOfRoundsLast100 += 5 + (5 - returnValue % 10)
    gamesCount += 1

    if gamesCount % 10 == 0:
      saveMePlease()
    if gamesCount % 100 == 0:
      saveLearningRate()
      sarsaWonLast100 = 0
      numberOfRoundsLast100 = 0
      sarsaLostRoundLast100 = 0
      CRMLostRoundsLast100 = 0
      CRMWonLast100 = 0
except:
  saveMePlease()

```

# Příloha B

# Zdrojový kód umělé inteligence v jazyce C#

[fagile]

```
1  using System;
2  using System.Collections.Generic;
3  using System.IO;
4  using UnityEngine;
5  using Random = UnityEngine.Random;
6
7  public class AISARSA : MonoBehaviour
8  {
9      public List<LDStateAction> Qsarsa;
10     public float alpha = 0.5f;
11     public float gamma = 0.5f;
12     public float rewardWin = 1.0f;
13     public float rewardAnotherCall = 0.5f;
14     public float rewardLose = 0;
15     public LDStateAction lastAction;
16     public float lastReward;
17     public bool useSARSA;
18     public EnemyProfiling enemyProfile = new EnemyProfiling();
19
20     private int maxQuantity;
21     public class LDState
22     {
23         public int quantity;
24         public int face;
25
26         public LDState(int setQuantity, int setFace)
27         {
28             quantity = setQuantity;
29             face = setFace;
30         }
31     }
32
33     public enum LDAction
34     {
35         Call,
36         Liar
37     }
38
```

```
39      public class LDStateAction
40      {
41          public LDAction action;
42          public LDState state;
43          public double qValue;
44
45          public LDStateAction(LDAction setAction, LDState setLDState, int maxQuantity)
46          {
47              action = setAction;
48              state = setLDState;
49              if (action == LDAction.Call)
50              {
51                  qValue = 1.0d/(2.0d*state.quantity);
52              }
53              else
54              {
55                  double editedQuantity = (maxQuantity-state.quantity+1);
56                  qValue = 1.0d/(2.0d*editedQuantity);
57              }
58          }
59          public LDStateAction(LDAction setAction, LDState setLDState, double setQ)
60          {
61              action = setAction;
62              state = setLDState;
63              qValue = setQ;
64          }
65      }
66
67      public class EnemyMove
68      {
69          public LiarsDice.Choice choice;
70          public int allDiceCount = 0;
71      }
72      public class EnemyProfiling
73      {
74          public float highGameLying = 0.0f;
75          public float midGameLying = 0.0f;
76          public float lowGameLying = 0.0f;
77          public int highGameCount = 0;
78          public int midGameCount = 0;
79          public int lowGameCount = 0;
80          public int highGameLieCount = 0;
81          public int midGameLieCount = 0;
82          public int lowGameLieCount = 0;
83          public int gamesPlayed = 0;
84          public List<EnemyMove> enemyMoves = new List<EnemyMove>();
85
86          public void addEnemyMove(LiarsDice.Choice enemyChoice, int allDiceCount)
87          {
88              EnemyMove enemyMove = new EnemyMove();
89              enemyMove.choice = enemyChoice;
90              enemyMove.allDiceCount = allDiceCount;
91              enemyMoves.Add(enemyMove);
92          }
93
94          public void clearEnemyMoves()
95          {
96              enemyMoves.Clear();
97          }
```

```
98
99          public bool didHeLie(LiarsDice.Choice enemyChoice, List<DiceInterface>
    listOfAllDice)
100         {
101             int countOfDiceTrue = 0;
102             foreach (var diceInterface in listOfAllDice)
103             {
104                 if (diceInterface.dice.isActive && diceInterface.dice.value == enemyChoice.
    diceNumber)
105                 {
106                     countOfDiceTrue++;
107                 }
108             }
109
110             if (countOfDiceTrue >= enemyChoice.multpiplier)
111             {
112                 return false;
113             }
114             else return true;
115         }
116
117         public void computeProfile(List<DiceInterface> listOfAllDice)
118         {
119             //updating counters
120             foreach (var enemyMove in enemyMoves)
121             {
122                 if ((enemyMove.choice.multpiplier > Math.Ceiling(2.0f * enemyMove.
    allDiceCount / 3.0f)) ||
123                     (enemyMove.choice.diceNumber > 4))
124                 {
125                     highGameCount++;
126                     //compute lie
127                     if (didHeLie(enemyMove.choice, listOfAllDice))
128                     {
129                         highGameLieCount++;
130                     }
131                 }
132                 else if ((enemyMove.choice.multpiplier > Math.Ceiling(enemyMove.
    allDiceCount / 3.0f)) ||
133                          (enemyMove.choice.diceNumber > 2))
134                 {
135                     midGameCount++;
136                     //compute lie
137                     if (didHeLie(enemyMove.choice, listOfAllDice))
138                     {
139                         midGameLieCount++;
140                     }
141                 }
142                 else
143                 {
144                     lowGameCount++;
145                     //compute lie
146                     if (didHeLie(enemyMove.choice, listOfAllDice))
147                     {
148                         lowGameLieCount++;
149                     }
150                 }
151             }
152
```

```
153             if(highGameCount != 0)
154                 highGameLying = ((float) highGameLieCount / highGameCount);
155             if(midGameCount != 0)
156                 midGameLying = ((float) midGameLieCount /  midGameCount);
157             if(lowGameCount != 0)
158                 lowGameLying = ((float) lowGameLieCount / lowGameCount);
159 
160             gamesPlayed++;
161             if (gamesPlayed > 100)
162             {
163                 gamesPlayed = 0;
164                 highGameCount = 0;
165                 midGameCount = 0;
166                 lowGameCount = 0;
167                 highGameLieCount = 0;
168                 midGameLieCount = 0;
169                 lowGameLieCount = 0;
170             }
171 
172             Debug.Log("Profiling - High: " + highGameLying + "%, Mid: " + midGameLying +
173                       "%, Low: " + lowGameLying + "%");
174         }
175     }
176 
177     public List<LDState> getAllStates(int playerCount, int maxCountPlayersDice)
178     {
179         int allDice = playerCount * maxCountPlayersDice;
180         List<LDState> listOfLDStates = new List<LDState>();
181 
182         for (int i = 1; i <= allDice; i++)
183         {
184             for (int j = 1; j <= 6; j++)
185             {
186                 listOfLDStates.Add(new LDState(i,j));
187             }
188         }
189 
190         return listOfLDStates;
191     }
192 
193     public List<LDStateAction> getAllStateActions(List<LDState> states)
194     {
195         List<LDStateAction> listOfStateActions = new List<LDStateAction>();
196 
197         foreach (var state in states)
198         {
199             listOfStateActions.Add(new LDStateAction(LDAction.Liar, state, maxQuantity));
200             listOfStateActions.Add(new LDStateAction(LDAction.Call, state, maxQuantity));
201         }
202 
203         return listOfStateActions;
204     }
205 
206     public void setupSARSA(int playerCount, int maxCountPlayersDice)
207     {
208         List<LDState> states = getAllStates(playerCount, maxCountPlayersDice);
209         maxQuantity = playerCount * maxCountPlayersDice;
210         if ((useSARSA && File.Exists(AIVarsAndStuff.aiSarsaPathToFile)) ||
211             (!useSARSA && File.Exists(AIVarsAndStuff.aiPathToFile)))
```

```
212         {
213             //I am dumb and this is how this works..... Please write this later
214             LoadQValues();
215         }
216         else
217         {
218             Qsarsa = getAllStateActions(states);
219         }
220 
221         lastAction = null;
222     }
223 
224     public double getQvalue(int quantity, int face, LDAction forAction)
225     {
226         foreach (var stateAction in Qsarsa)
227         {
228             if (stateAction.action == forAction && stateAction.state.quantity == quantity
    &&
229                 stateAction.state.face == face)
230             {
231                 return stateAction.qValue;
232             }
233         }
234 
235         //Should not happen
236         return Mathf.Infinity;
237     }
238 
239     public double getMaxPossibleQValue(List<LiarsDice.Choice> choices, LiarsDice.Choice
    liarChoice)
240     {
241         double maxQ = Double.NegativeInfinity;
242         double qValue;
243         foreach (var choice in choices)
244         {
245             qValue = getQvalue(choice.multpiplier, choice.diceNumber, LDAction.Call);
246             if (qValue > maxQ)
247                 maxQ = qValue;
248         }
249 
250         if (liarChoice != null)
251         {
252             double qValueLiar = getQvalue(liarChoice.multpiplier, liarChoice.diceNumber,
    LDAction.Liar);
253             if (qValueLiar > maxQ)
254                 maxQ = qValueLiar;
255         }
256 
257         if (double.IsNegativeInfinity(maxQ))
258             throw new Exception("Why am I so bad at this? Got negative Infinity in
    getMaxPossibleQValue");
259 
260         return maxQ;
261     }
262 
263     public void updateQ(LDStateAction QpreviousRound, double QValueNow, float reward)
264     {
265         if(QpreviousRound == null) return;
266         int index=-1;
```

```
267         foreach (var stateAction in Qsarsa)
268         {
269             if (stateAction.action == QpreviousRound.action &&
270                 stateAction.state.quantity == QpreviousRound.state.quantity &&
271                 stateAction.state.face == QpreviousRound.state.face)
272             {
273                 stateAction.qValue += alpha*(reward + gamma*QValueNow - QpreviousRound.
    qValue);
274                 index = Qsarsa.IndexOf(stateAction);
275                 break;
276             }
277         }
278 
279         if (index >= 0)
280         {
281             //debugPrint(Qsarsa[index].action, Qsarsa[index].state);
282         }
283     }
284 
285     public double getValueBasedOnProfiling(LiarsDice.Choice liarsChoice, List<LiarsDice.
    Choice> listOfChoices)
286     {
287         //Deducing all available dices from listOfChoices
288         int allAvailableDiceCount = 0;
289         foreach (var choice in listOfChoices)
290         {
291             if (choice.multpiplier > allAvailableDiceCount)
292                 allAvailableDiceCount = choice.multpiplier;
293         }
294 
295         //Getting the highest Qvalue
296         double highestQ = 0.0f;
297         foreach (var ldStateAction in Qsarsa)
298         {
299             if (ldStateAction.qValue > highestQ)
300                 highestQ = ldStateAction.qValue;
301         }
302 
303         if ((liarsChoice.multpiplier > Math.Ceiling(2.0f * allAvailableDiceCount / 3.0f))
    ||
304             (liarsChoice.diceNumber > 4))
305         {
306             return enemyProfile.highGameLying * highestQ;
307         }
308         else if ((liarsChoice.multpiplier > Math.Ceiling(allAvailableDiceCount / 3.0f)) ||
309                  (liarsChoice.diceNumber > 2))
310         {
311             return enemyProfile.midGameLying * highestQ;
312         }
313         else
314         {
315             return enemyProfile.lowGameLying * highestQ;
316         }
317     }
318     public LDStateAction roullete(List<LiarsDice.Choice> listOfAvaibleChoices, LiarsDice.
    Choice liarState)
319     {
320         //Getting Qvalue of Liar State
321         double callLiar;
```

```
322         if (liarState != null)
323         {
324             callLiar = getQvalue(liarState.multpiplier, liarState.diceNumber, LDAction.Liar
    );
325             if (useSARSA)
326             {
327                 //adding value to the LiarsCall based on profiling
328                 callLiar += (getValueBasedOnProfiling(liarState, listOfAvaibleChoices) *
    2.0d);
329             }
330         }
331         else
332             callLiar = 0.0f;
333 
334         //Setting up threshold field, each item is threshold it needs surpass in order to
    take its value
335         List<float> thresholds = new List<float>();
336         float allOfQs = 0.0f;
337         for(int i = 0; i < listOfAvaibleChoices.Count; i++)
338         {
339             var choice = listOfAvaibleChoices[i];
340             float q;
341             q = Mathf.Abs((float)getQvalue(choice.multpiplier, choice.diceNumber, LDAction.
    Call));
342             allOfQs += q;
343             thresholds.Add(allOfQs);
344         }
345 
346         //Adding liar call to the threshold
347         allOfQs += Mathf.Abs((float)callLiar);
348         thresholds.Add(allOfQs);
349 
350         //Random number generator
351         float random = Random.Range(0.0f, allOfQs);
352 
353         //Getting the index of the threshold that won the roulette
354         int indexOfThreshold = -1;
355         for (int i = 0; i < thresholds.Count; i++)
356         {
357             if (random <= thresholds[i])
358             {
359                 indexOfThreshold = i;
360                 break;
361             }
362         }
363 
364         //This should never happen please
365         if (indexOfThreshold < 0)
366         {
367             throw new Exception("Threshold not found, this is bad: " + random);
368         }
369 
370         //Using that index to get corresponding action from listofAvailableChoices or from
    the liar call
371         LDStateAction chosenAction;
372         if (indexOfThreshold == listOfAvaibleChoices.Count)
373         {
374             chosenAction = new LDStateAction(LDAction.Liar,
375                 new LDState(liarState.multpiplier, liarState.diceNumber),
```

```
376                 maxQuantity);
377         }
378         else
379         {
380             chosenAction = new LDStateAction(LDAction.Call,
381                 new LDState(listOfAvaibleChoices[indexOfThreshold].multpiplier,
382                     listOfAvaibleChoices[indexOfThreshold].diceNumber), maxQuantity);
383         }
384 
385         //because chosenAction is newly created and it needs the Qvalue for further
    computing, I add it through my func
386         chosenAction.qValue = getQvalue(chosenAction.state.quantity, chosenAction.state.
    face, chosenAction.action);
387 
388         lastAction = chosenAction;
389         return chosenAction;
390     }
391 
392     public void SaveAIQvalues()
393     {
394         if(Qsarsa.Count < 1) return;
395         AISaveSystem.SaveAIData(this);
396     }
397 
398     public void LoadQValues()
399     {
400         AIData data = AISaveSystem.LoadAIData(useSARSA);
401         Qsarsa = new List<LDStateAction>();
402         for (int i = 0; i < data.actions.Length; i++)
403         {
404             LDState state = new LDState(data.quantityField[i],data.faceField[i]);
405             LDStateAction stateAction = new LDStateAction(data.actions[i],state,data.
    qValues[i]);
406             Qsarsa.Add(stateAction);
407         }
408         //debugPrint();
409         //Profiling
410         EnemyProfiling profile = new EnemyProfiling();
411         profile.highGameCount = data.highGameCount;
412         profile.midGameCount = data.midGameCount;
413         profile.lowGameCount = data.lowGameCount;
414         profile.highGameLieCount = data.highGameLieCount;
415         profile.midGameLieCount = data.midGameLieCount;
416         profile.lowGameLieCount = data.lowGameLieCount;
417         profile.highGameLying = data.highGameLying;
418         profile.midGameLying = data.midGameLying;
419         profile.lowGameLying = data.lowGameLying;
420         profile.gamesPlayed = data.gamesPlayed;
421         enemyProfile = profile;
422     }
423 
424     void OnDisable()
425     {
426         SaveAIQvalues();
427     }
428 
429     public void debugPrint()
430     {
431         foreach (var q in Qsarsa)
```

```
        {
            string action = "";
            if (q.action == LDAction.Call)
                action = "Call";
            else action = "Liar";

            Debug.Log("Act: "+action+";Quant: "+q.state.quantity+";Face: "+q.state.face+";Q
    : "+q.qValue);
        }
    }

    public void debugPrint(LDAction action, LDState state)
    {
        foreach (var stateAction in Qsarsa)
        {
            if (stateAction.action == action && stateAction.state == state)
            {
                string act = "";
                if (action == LDAction.Call)
                    act = "Call";
                else act = "Liar";
                Debug.Log("Act: "+act+
                          ";Quant: "+stateAction.state.quantity+
                          ";Face: "+stateAction.state.face+
                          ";Q: "+stateAction.qValue);
            }
        }
    }
}
```