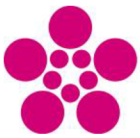


Ekonomická fakulta
Faculty of Economics

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích
Ekonomická fakulta
Katedra aplikované matematiky a informatiky

Diplomová práce

Posilované učení pro volbu trasy v rámci scénáře abstraktního provozu

Vypracoval: Bc. Leoš Glaser
Vedoucí práce: doc. Ing. Ladislav Beránek, CSc., MBA

České Budějovice 2023

JIHOČESKÁ UNIVERZITA V ČESKÝCH BUDĚJOVICÍCH

Ekonomická fakulta

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Leoš GLASER**
Osobní číslo:	**E21033**
Studijní program:	**N0613A140025 Aplikovaná informatika**
Specializace:	**Podniková informatika**
Téma práce:	**Posilované učení pro volbu trasy v rámci scénáře abstraktního provozu**
Zadávající katedra:	**Katedra aplikované matematiky a informatiky**

Zásady pro vypracování

Při cestách (např. dojíždění do města) je cílem každého řidiče dosáhnout přiměřené doby jízdy ze svého výchozího bodu do cíle. Z globálního hlediska je žádoucí, aby byla dopravní zátěž rozložena úměrně kapacitě silnic v dopravní síti.
Cílem práce bude využít přístup posilovaného učení pro návrh volbu trasy, který by se opíral pouze o zkušenosti řidičů. Půjde tedy o příklad využití konceptu nezávislých učících se agentů.

Metodický postup:

1. Studium odborné literatury.
2. Úvod do problematiky reinforcement learning a obecný popis řešení.
3. Teoretický popis možného řešení a jednotlivých komponent.
4. Popis implementace řešení.
5. Zhodnocení a možnosti alternativ.

Rozsah pracovní zprávy:	**50 – 60 stran**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování diplomové práce:	**tištěná**

Seznam doporučené literatury:

1. PRETORIUS, A., & at al. *Mava: a research framework for distributed multi-agent reinforcement learning.* [online]. [cit. 2021-11-01]. Dostupné z: <https://arxiv.org/abs/2107.01460>
2. TAVARES, A. R., & BAZZAN, A.L.C. (2012). Reinforcement learning for route choice in an abstract traffic scenario. In *VI Workshop-Escola de Sistemas de Agentes, seus Ambientes e aplicaçoes (WESAAC).* s. 141-153.
3. WINDER, P. (2020). *Reinforcement Learning: Industrial Applications of Intelligent Agents.* Newton, MA: O'Reilly Media.

Vedoucí diplomové práce: **doc. Ing. Ladislav Beránek, CSc., MBA**
Katedra aplikované matematiky a informatiky

Datum zadání diplomové práce: **11. ledna 2022**
Termín odevzdání diplomové práce: **14. dubna 2023**

doc. Dr. Ing. Dagmar Škodová Parmová
děkanka

JIHOČESKÁ UNIVERZITA
V ČESKÝCH BUDĚJOVICÍCH
EKONOMICKÁ FAKULTA
Studentská 13 (26)
370 05 České Budějovice

doc. RNDr. Tomáš Mrkvička, Ph.D.
vedoucí katedry

V Českých Budějovicích dne 28. února 2022

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své diplomové práce, a to - v nezkrácené podobě - elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne

_____

Leoš Glaser

Poděkování

Děkuji vedoucímu této práce doc. Ing. Ladislavovi Beránkovi, CSc., MBA za jeho cenné rady a konzultace. Děkuji též své rodině a přátelům za jejich podporu během studia.

Obsah

Úvod

Návrh tras je problematika, která se týká mnoha oblastí, jako je například doprava, logistika, turistika nebo robotika. Vhodně naplánované trasy mohou přispět ke snížení času na přepravu, k minimalizaci nákladů anebo ke zvýšení plynulosti dopravy. Kromě ekonomických efektů je plánovaní tras důležité mimo jiného i z hlediska životního prostředí.

Existují různé metody pro navrhování tras, od klasických algoritmů teorie grafů, přes genetické algoritmy, heuristické metody až po metody umělé inteligence. Tato diplomová práce se zabývá návrhem trasy pro jednoho agenta pomocí posilovaného učení.

Tato práce obsahuje šest kapitol. V první kapitole jsou představeny teoretické základy umělé inteligence a agentů, druhá kapitola je zaměřena na teorii posilovaného učení a na popis několika vybraných metod. Třetí kapitola se stručně zabývá problematikou simulace dopravy a dává stručný přehled několika souvisejících problémů. Ve čtvrté kapitole je popsán návrh softwarové aplikace řešící cíl práce a detailní popis jejího vývoje je popsán v páté kapitole. Šestá kapitola je zaměřena na návrh trasy v případě konkrétní lokality v Českých Budějovicích. V závěru práce jsou stručně shrnuty dosažené výsledky a diskutována možná zlepšení a doporučení navazující na tuto práci.

Cíl práce

Cílem praktické části této práce je vyvinutí aplikace na bázi posilovaného učení simulující pohyb jednoho agenta v silniční síti po nejkratší okružní trase z počátečního bodu tak, aby v rámci trasy navštívil (obsloužil) předem definované body sítě. Agent na začátku nemá žádné předdefinované trasy a jednotlivé trasy volí (učí se) postupně na základě získaných zkušeností (odměn) v rámci cestování v síti. Aplikace bude otestována na úloze svozu odpadu ve vybrané lokalitě.

V kontextu reálného provozu se lze kromě svozu odpadu s výše uvedeným problémem setkat i v jiných aplikačních oblastech, například distribuce zásilek nebo zásobování obchodních jednotek.

1 Umělá inteligence a strojové učení

1.1 Umělá inteligence

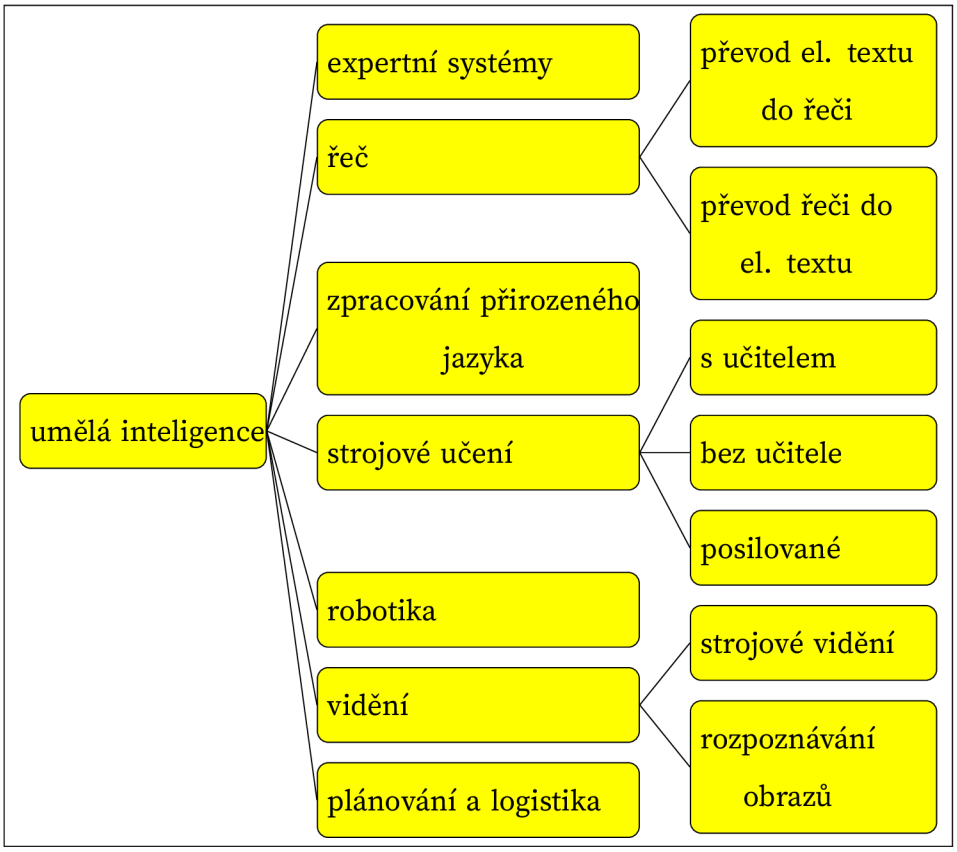
Umělá inteligence (Artificial Intelligence, AI) má mnoho definicí, některé kladou důraz na podobnost s tím, jak by jednal *člověk*, jiné naopak vycházejí z konceptu *racionality* - zjednoušeně řečeno „dělání správných věcí". Z jiné úrovně pohledu je u umělé inteligence kladen zájem na inteligentní *uvažování* (angl. reasoning) nebo na inteligentní *chování*. (Norvig & Russel, 2020)

K umělé inteligenci lze tedy přistupovat vícero způsoby, přičemž tyto se od sebe liší jak svými teoretickými východisky, tak i zkoumanými problémy a metodami k jejich řešení. Dané přístupy nejsou disjunktní.

- *Lidské chování*: s tímto přístupem souvisí Turingův test, který ve své základní podobě probíhá formou výměny textových zpráv mezi člověkem a AI systémem. AI systém úspěšně splňuje Turingův test, pokud člověk nedokáže rozlišit, zda komunikuje s AI systémem nebo s člověkem.
  Souvisejícími disciplínami jsou zpracování přirozeného jazyka, reprezentace znalostí, automatické uvažování či strojové učení.
- *Lidské uvažování*: tento přístup se zaměřuje na porovnávání a vzájemné doplňování se toho, jak o problémech přemýšlí a jak je řeší AI, a jak člověk. S tím souvisí interdisciplinární obor *kognitivních věd*, kam se řadí disciplíny a metody z oblasti psychologie, strojového učení, neurozobrazování, lingvistiky a další.
- *Racionální uvažování*: hlavním bodem zájmu jsou logika a pravděpodobnost a jejich pravidla. Pomocí nich jsou vytvářeny modely racionálního uvažování.
- *Racionální chování*: v tomto přístupu hrají významnou roli *racionální agenti*. Racionální agent jedná tak, aby dosáhl nejlepšího výsledku nebo, v případě nejistoty, nejlepšího očekávaného výsledku.

Stručný přehled oblastí umělé inteligence je na obrázku 1.

Obrázek 1: Oblasti umělé inteligence



Zdroj: zpracováno autorem podle Hendl (2021)

1.2 Strojové učení

Strojové učení znamená schopnost systému (programu, algoritmu) učit se z dat. Strojové učení postupuje dle Hendl (2021) v těchto krocích:

1. Získání dat
2. Úprava dat (angl. data wrangling): čištění dat a konverze do podoby požadované zvolenými procedurami strojového učení
3. Explorace dat: zkoumání a analýza dat, jelikož ne všechna data jsou nutně ve zvolené proceduře strojového učení zapotřebí
4. Učení algoritmu: algoritmus se „cvičí“ pomocí dat, aby „rozuměl“ konfiguracím a vztahům v datech
5. Testování algoritmu: algoritmus se testuje pomocí nových dat a určuje se jeho přesnost a rychlost
6. Aplikace: model je využíván, lze jej dále vylepšovat a trénovat

Metody strojového učení se obvykle dělí do tří skupin:

- učení s učitelem
- učení bez učitele
- posilované učení

Při **učení s učitelem** uživatel poskytne algoritmu dvojice vstupů a požadovaných výstupů a algoritmus najde způsob, jak při daném vstupu vytvořit požadovaný výstup. Algoritmus je zejména schopen vytvořit výstup pro vstup, který nikdy předtím neviděl, bez pomoci člověka. (Müller & Guido, 2016)

Techniky učení s učitelem se rozdělují do dvou oblastí: *regresní techniky* a *klasifikační techniky*. Cílem regresních technik je predikce výstupu, reprezentovaného vektorem Y, na základě vektoru vstupů X. Formálně lze toto zapsat vztahem $Y = f(X)$ kde f je algoritmus, jehož podoba se hledá. Příkladem využití je odhad tržní ceny domu na základě různých parametrů (lokalita, stáří, počet pokojů, aj.)

Cílem klasifikačních technik je rozdělení vstupních dat do tříd. Na základě vstupů X mají být přiřazeny výstupy Y, kde Y označuje nějaké klasifikační třídy. Příkladem využití je určení, zda email je SPAM nebo určení na základě rentgenového snímku nádoru zda nádor je zhoubný nebo není. (Hendl, 2021)

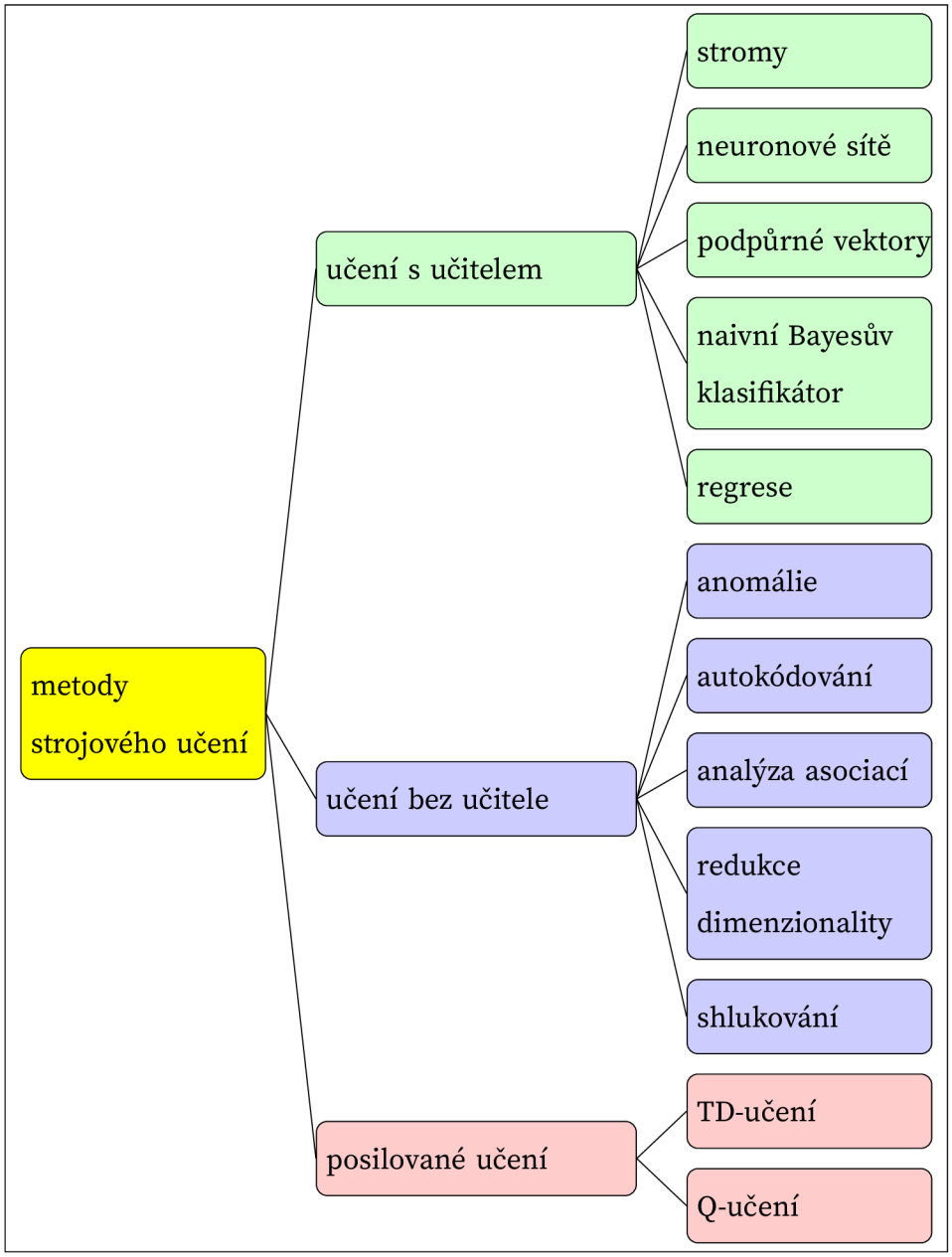
Při **učení bez učitele** jsou známa pouze vstupní data X a algoritmus nemá k dispozici žádná výstupní data Y. Algoritmy se snaží naučit podle vstupních dat, rozpoznat v datech vlastnosti, které zvýší přesnost algoritmu pro danou úlohu. Příkladem je zjištění parametrů pro co nejlepší členění zákazníků do skupin.

Další věcí, kterou algoritmy mohou činit, je vytváření *generativních modelů*, typicky ve formě pravděpodobnostního rozdělení, z něhož lze generovat nové vzorky. Příkladem je generování obrázků lidských obličejů. (Müller & Guido, 2016)

Posilované učení bude popsáno v další části tohoto textu.

Existuje mnoho metod strojového učení, pro ilustraci jsou na obrázku 2 uvedeny hlavní z nich.

Obrázek 2: Zástupci hlavních kategorií strojového učení



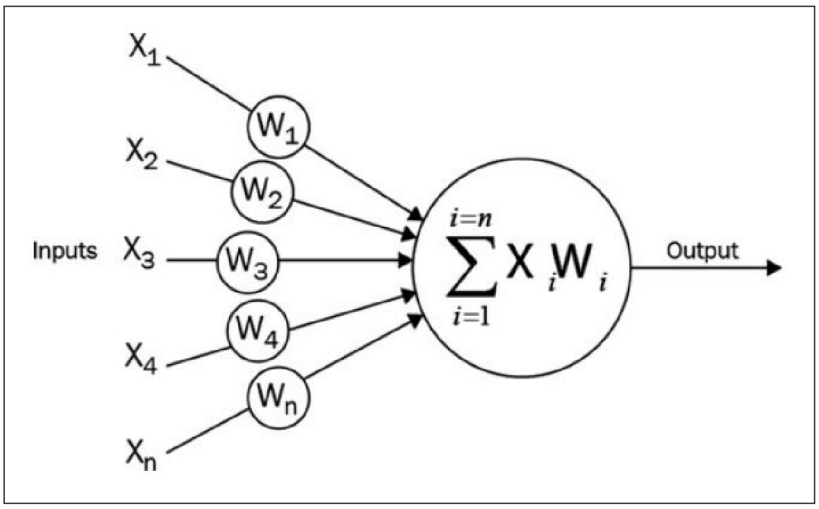
Zdroj: zpracováno autorem podle Hendl (2021)

1.3 Základy neuronových sítí

Umělé neuronové sítě se inspirují činností mozku. Neuronová síť se skládá ze vzájemně propojených neuronů, které zpracovávají vážené vstupy a poskytují výstupy. Síť se učí pomocí zvoleného učícího algoritmu a postupně se upravují váhy sítě pomocí vzorků z trénovacího souboru dat.

V případě biologického neuronu platí, že k přenosu signálu dojde, pokud celková síla signálu z jednotlivých vstupů překročí určitou mez a na tomto principu je založeno i fungování umělého neuronu. *Umělý neuron* přijímá vstup ve formě vektoru $X = \{x_1, x_2, ..., x_n\}$ vážený vektorem $W = \{w_1, w_2, ..., w_n\}$. V rámci trénování sítě se vektor W upravuje ve snaze najít optimální hodnoty jednotlivých vah tak, aby byl získán nejlepší výstup. Nejlepším výstupem je zde myšlen takový výstup, při kterém je nejmenší chyba, respektive nejmenší rozdíl mezi skutečnými a předpovídanými hodnotami napříč různými vzorky v množině trénovacích dat (Sewak, 2019). Zjednodušené schéma umělého neuronu je na obrázku 3.

Obrázek 3: Schéma umělého neuronu



Zdroj: Sewak (2019)

V případě učení sítě s učitelem, kdy jsou u učících příkladů k dispozici správné výstupy, se modifikace vah zjednodušeně děje takto:

$$w_i^{'} = w_i + \eta(\widehat{y} - y)x_i \tag{1}$$

kde $w_i^{'}$ je nová váha, w_i je stará váha, $\widehat{y}$ je výstup neuronu, y je správný výstup a η je tzv. rychlost učení, přičemž η je mezi hodnotami 0 a 1.

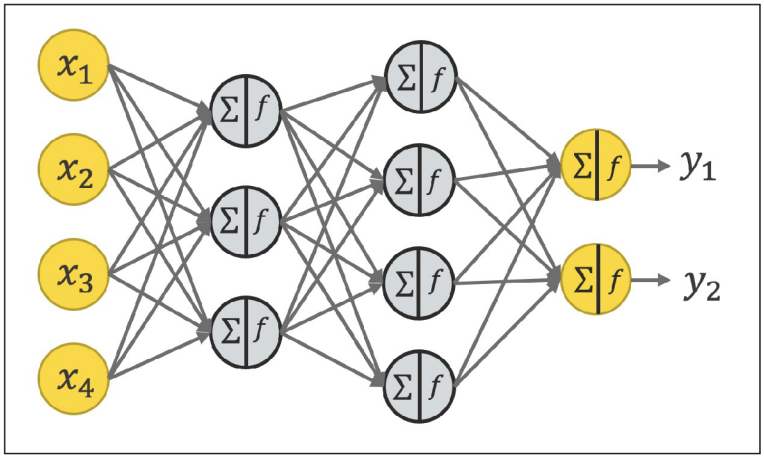
Způsob, jak je neuron aktivován určuje *aktivační funkce* f. Tato funkce může mít obecně různé tvary, například:

- skoková aktivační funkce neuron aktivuje, pokud součet součinů vah a vstupních signálů překročí stanovenou mez b, tj. neuron je aktivován, pokud $\sum_i^n w_i x_i > b$
- u semilineární aktivační funkce stoupá míra aktivace ve vztahu ke vstupu lineárně od 0 k 1.
- u sigmoidní aktivační funkce stoupá míra aktivace též od 0 do 1, ale ne lineárně

V závislosti na zvolené aktivační funkci tedy může, ale též nemusí existovat „tvrdá“ mez pro aktivaci neuronu a díky tomu lze určit jak „jemně“ neuron reaguje na vstupy (Hendl, 2021).

Neurony bývají uspořádány do vrstev. Typická neuronová síť má vstupní vrstvu, jednu nebo vícero skrytých vrstev a výstupní vrstvu. Je-li skrytých vrstev více, hovoří se o *hluboké neuronové síti*. Na obrázku 4 je schéma jednoduché hluboké neuronové sítě se 4 vstupy, 2 výstupy a 2 skrytými vrstvami o 3 a 4 neuronech.

Obrázek 4: Hluboká neuronová sít se 2 skrytými vrstvami



Zdroj: Melcher (2021)

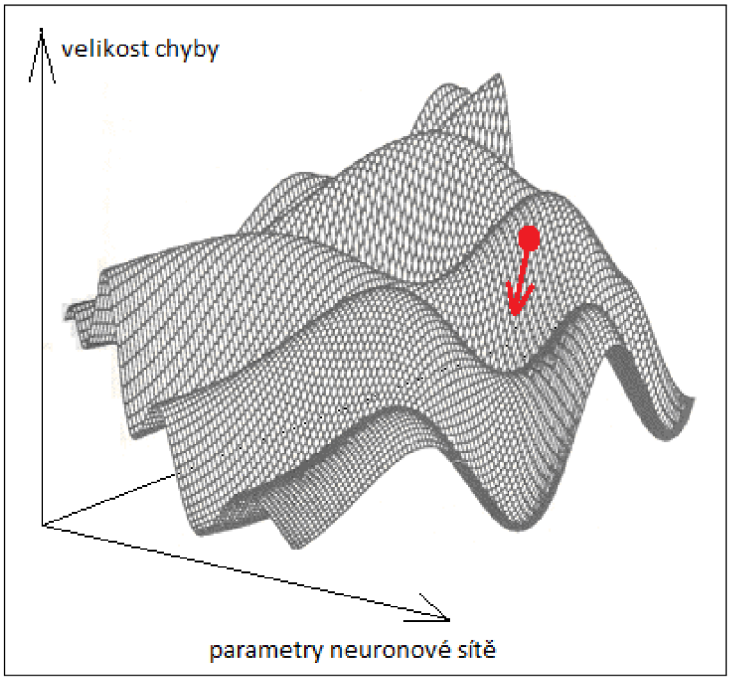
Při učení hluboké neuronové sítě se váhy sítě upravují iterativně pomocí algoritmu zpětné propagace. Stručně řečeno, algoritmus zpětné propagace iteruje v mnoha cyklech, ve kterých vždy probíhají dva procesy:

1. *fáze dopředného šíření,* v níž jsou neurony aktivovány vstupem z trénovací množiny případů postupně od vstupní vrstvy, přes skryté vrstvy až po výstupní vrstvu, která produkuje výstup $\widehat{y}$.

2. *fáze zpětného šíření*, ve které se nejdřív porovná výstup $\widehat{y}$ se správnou hodnotou výstupu y pro daný případ. Rozdíl se pak šíří od výstupní vrstvy zpět až k vstupní vrstvě a modifikují se váhy spojů mezi neurony (Hendl, 2021).

K modifikaci vah se používá gradientní metoda. Derivací aktivační funkce se získá gradient, což je vektor pomocí nějž lze zjistit, ve kterém směru ze současného bodu (tj. současného nastavení vah sítě) klesá chyba nejrychleji. Chyba je minimalizována postupnými sestupy (tj. úpravami vah sítě). Ilustrativně je princip metody naznačen na obrázku 5.

Obrázek 5: Gradientní sestup



Zdroj: Chalupník (2012)

1.4 Inteligentní agenti

1.4.1 Agent a prostředí

Agentem se rozumí cokoliv, co vnímá své *prostředí* prostřednictvím *senzorů* a působí na něj prostřednictvím *aktuátorů* (též akčních členů).

Posloupnost vjemů agenta je kompletní historie všeho, co kdy agent vnímal. Obecně platí, že volba akce agenta v daném okamžiku může záviset na jeho vestavěných znalostech a na celé dosud pozorované posloupnosti vjemů, ale ne na něčem, co dosud nevnímal. (Norvig & Russel, 2020)

Formálně lze agenta popsat tzv. *agentovou funkcí*:

$$V \rightarrow A \tag{2}$$

kde V je množina vjemů a A je množina akcí. Agentová funkce je obecný matematický popis agenta, zatímco její konkrétní implementací je agentův program. Je požadováno, aby se agent choval co „nejsprávněji“. S tímto souvisí koncept racionality agentů a hodnocení výkonnosti agentů.

Racionální agent (též inteligentní agent) je takový agent, který vždy volí akci, při jejímž provedením se očekává, že příslušná míra výkonu bude optimalizovaná. **Míra výkonu** je obvykle určena uživatelem (návrhářem agenta) a odráží to, co uživatel od agenta očekává v rámci daného úkolu. Například agent v elektronické aukci se musí snažit minimalizovat výdaje. (Vlassis, 2007)

Racionální agent by měl být *autonomní*, tj. měl by se učit, co může, aby kompenzoval své částečné nebo nesprávné předchozí znalosti. V praxi se málokdy vyžaduje úplná autonomie od samého počátku: když má agent málo zkušeností nebo nemá žádné, musel by jednat náhodně. Po dostatečném nabytí zkušeností s prostředím se chování racionálního agenta může stát nezávislým na jeho předchozích znalostech.

Prvním krokem při vývoji agenta je co nejúplnější specifikace čtveřice

(*míra výkonu, prostředí, aktuátory, senzory*)

Norvig and Russel (2020) toto přibližují na na příkladu vývoje agenta pro řízení taxi:

- Míra výkonu: čím bude výkon agenta posuzována, čeho by agent měl dosahovat? Mezi žádoucí vlastnosti patří dojet do správného cíle, minimalizace spotřeby paliva, nákladů nebo doby jízdy, minimalizace porušování dopravních předpisů, maximalizace zisku nebo bezpečnosti cestujících. Je zřejmé, že některé z těchto cílů jsou v rozporu, takže bude nutné najít kompromisy.
- Prostředí: v jakém prostředí se agent bude pohybovat, co má být do prostředí zahrnuto a od čeho naopak bude abstrahováno? Je tedy potřeba specifikovat například jaké typy cest bude prostředí obsahovat (venkovské cesty, silnice I. třídy, dálnice, aj.) Na silnicích se vyskytují další účastníci provozu, dále též chodci, zvířata, práce na silnici, atd. Čím omezenější je prostředí, tím je návrh a vývoj agenta jednodušší.
- Aktuátory: agent bude přinejmenším muset umět ovládat pohyb vozidla (plyn, brždění, zatáčení), dále i nějaký způsob komunikace s pasažéry.
- Senzory: mezi základní senzory taxíku bude patřit jedna nebo více videokamer, senzory pro zjišťování vzdálenosti od ostatních vozů a překážek. Měl by mít rychloměr, akcelerometr, senzory pro sledování motoru, paliva a elektrické soustavy. Dále bude možná potřebovat senzory pro GPS, aby mohl používat navigaci. V neposlední řadě bude muset být vybaven i senzory na zpracování vstupu od pasažérů.

Významnou roli pro navrhování a vývoj agenta hraje prostředí, ve kterém agent působí. Prostředí lze charakterizovat dle různých vlastností (Norvig & Russel, 2020):

- *Plně (resp. částečně) pozorovatelné*: Pokud senzory agenta umožňují přístup ke kompletnímu stavu prostředí v každém časovém okamžiku, prostředí je plně pozorovatelné. Senzory detekují všechny aspekty relevantní pro volbu akce.

  Prostředí může být částečně pozorovatelné kvůli šumu a nepřesnosti senzorů nebo proto, že části stavu v datech senzorů chybí. Taxi agent například nezjistí, co si myslí ostatní řidiči. Pokud agent nemá vůbec žádné senzory, pak je prostředí nepozorovatelné.
- *Epizodické (resp. sekvenční)*: V epizodickém prostředí lze chování agenta rozdělit do atomických epizod, kde každé epizodě agent obdrží vjem a poté

provede jednu akci. Zásadní je, že následující epizoda nezávisí na akcích provedených v předchozích epizodách. Příkladem epizodického prostředí je agent rozpoznávající na montážní lince vadné díly - každé rozhodnutí je učiněno dle aktuálního dílu bez ohledu na předchozí rozhodnutí (díly) a aktuální rozhodnutí navíc neovlivňuje, zda je vadný další díl. Epizodická prostředí jsou výrazně jednodušší než prostředí sekvenční, protože agent nemusí přemýšlet dopředu. V sekvenčních prostředích může aktuální rozhodnutí ovlivnit všechna budoucí rozhodnutí. Příkladem takových prostředí je výše uvedené řízení taxíku nebo hra šachy.

- *Proměnlivé prostředí*: Pokud se prostředí může během rozhodování agenta měnit, pak prostředí je pro daného agenta dynamické, v opačném případě je statické. Se statickým prostředím se pracuje snadno, protože agent nemusí neustále pozorovat svět, zatímco se rozhoduje o akci. Pokud se prostředí samo o sobě nemění, ale skóre výkonnosti agenta ano, pak se prostředí nazývá semidynamické. Jízda taxíkem je dynamická, šachy hrané s časomírou jsou semidynamické. Hra křížovky je statická.
- *Diskrétní (resp. spojité):* Diskrétní prostředí má konečný počet různých stavů, zatímco spojité prostředí jich má nekonečně mnoho. Například šachy mají konečný počet různých stavů (nepočítaje stav hodin) i končený počet vjemů a akcí. Stavy při řízení taxi jsou spojité, např. rychlost a poloha taxíku se mění plynule v čase. Akce při řízení taxíku jsou rovněž spojité (úhly řízení atd.)
- *Deterministické (resp. nedeterministické):* Pokud je příští stav prostředí zcela určen aktuálním stavem a akcí provedenou agentem, prostředí je deterministické, v opačném případě je nedeterministické.
- *Jednoagentní vs. multiagentní:* Pokud v prostředí působí pouze jeden agent, který pracuje sám, jde o jednoagentní prostředí. Působí-li v prostředí více agentů, jde o multiagentní prostředí.

V tabulce 1 jsou uvedeny další příklady prostředí a jejich charakteristik.

Tabulka 1: Příklady prostředí a jejich vlastností

	Poker	Obrázky (analýza)	Medicínská diagnostika
Pozorovatelnost	částečně	plně	částečně
Počet agentů	multi	jeden	jeden
Determinističnost	nedeterministické	deterministické	nedeterministické
Epizodičnost	sekvenční	epizodické	sekvenční
Proměnlivost	statické	semidynamické	dynamické
Spojitost	diskrétní	spojité	spojité

Zdroj: převzato z Norvig and Russel (2020)

1.4.2 Typy agentů

Zásadním problémem při návrhu a vývoji inteligentních agentů je volba agentova programu, podle kterého se agent bude chovat. Existuje několik základních typů programů, resp. agentů (Norvig & Russel, 2020).

Jednoduchý reaktivní agent volí akce na základě aktuálního vjemu a ignoruje zbytek vjemů z minulosti. Program agenta je implementován pomocí pravidel typu

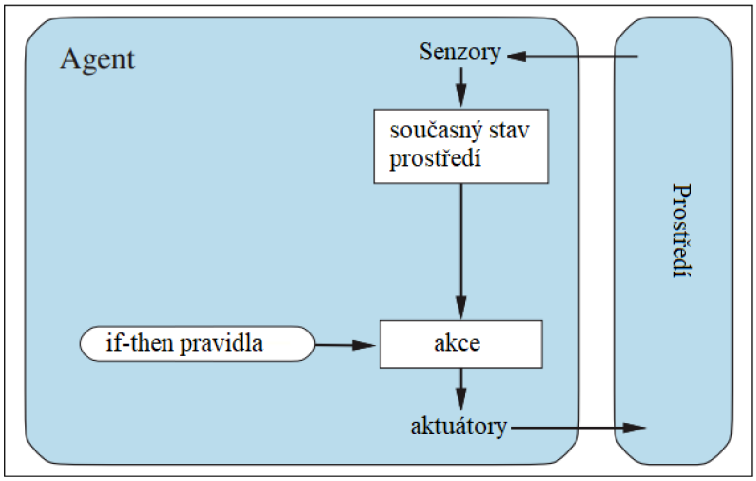
podmínka - akce

Výhodou agenta je jeho jednoduchost, ale nevýhodou je jeho omezená inteligence. Agent bude správně fungovat pouze tehdy, pokud lze na základě aktuálního vjemu učinit správné rozhodnutí, tedy pouze pokud je prostředí plně pozorovatelné.

Jednoduší reaktivní agenti se v prostředích, která nejsou plně pozorovatelná, často zacyklí. Zacyklení lze předejít pomocí randomizace výběru akcí agenta, nicméně ve většině případů je vhodnější použít jiný typ agenta.

Schéma jednoduchého reaktivního agenta je na obrázku 6.

Obrázek 6: Schéma jednoduchého reaktivního agenta



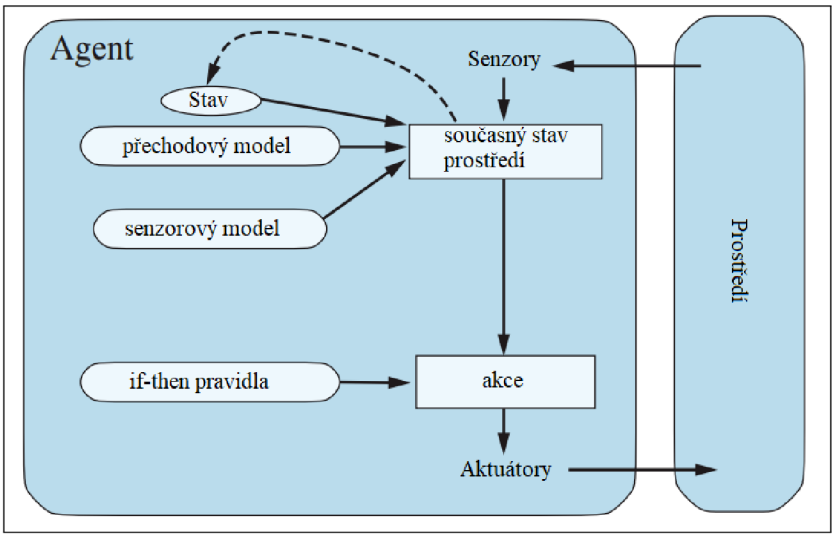
Zdroj: Norvig and Russel (2020)

Reaktivní agent s vnitřním stavem si udržuje vnitřní stav, který závisí na historii vjemů a reprezentuje stav světa. Agent si v paměti udržuje dva modely:

- Přechodový model zahrnuje informace o tom, jak je prostředí ovlivněno akcemi učiněných agentem a též jak se prostředí vyvíjí samo o sobě, nezávisle na agentovi.
- Senzorový model toho, jak je prostředí reprezentováno ve vnímání agenta.

Modely a reprezentace stavů se značně liší v závislosti na typu prostředí, řešeném problému a konkrétní technologii použité při návrhu a vývoji agenta. Schéma reaktivního agenta s vnitřním stavem je na obrázku 7.

Obrázek 7: Schéma reaktivního agenta s vnitřním stavem



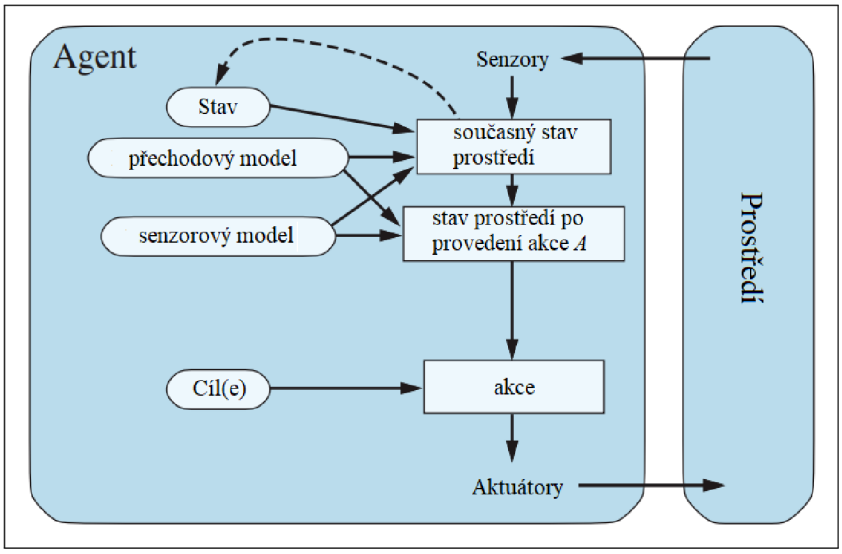
Zdroj: Norvig and Russel (2020)

Dosud byli představeni agenti, kteří se rozhodovali na základě současného stavu, přičemž rozhodnutí mohlo být ovlivněno minulými vjemy. Ke správnějšímu rozhodnutí ale toto ne vždy stačí. Uvažuje-li se například problém křižovatky při hledání optimální trasy, tak na rozhodnutí jakým směrem se má agent vydat potřebuje agent znát další informaci - v tomto případě cíl trasy.

Cílově orientovaný agent tedy potřebuje znát cíl, který popisuje žádoucí stavy. Někdy je volba akce na základě cíle jednoduchá, jindy ke splnění cíle vede dlouhá posloupnost akcí a rozhodnutí. Agent do rozhodování o volbě akce zahrnuje uvažování o budoucnosti, zatímco reaktivní agenti volí akci přímo dle vjemů a případně dle vnitřního stavu.

Schéma cílově orientovaného agenta s vnitřním stavem je na obrázku 8.

Obrázek 8: Schéma cílově orientovaného agenta s vnitřním stavem



Zdroj: Norvig and Russel (2020)

Cíl sám o sobě však pro co nejlepší chování agenta nestačí. Například taxi agent se do své cílové destinace může dostat vícero cestami, tzn. existuje více posloupností akcí, pomocí kterých se agent dostane do cílového stavu. Nicméně některé cesty, respektive posloupnosti stavů agenta vedoucí do cíle, jsou vhodnější než jiné - jsou rychlejší, kratší anebo méně nákladné. Pro rozlišení jak jsou jednotlivé stavy žádoucí se proto zavádí užitek, který je formálně popsán užitkovou funkcí U

$$U : s \rightarrow u \tag{3}$$

kde s je stav a u je užitek stavu s.

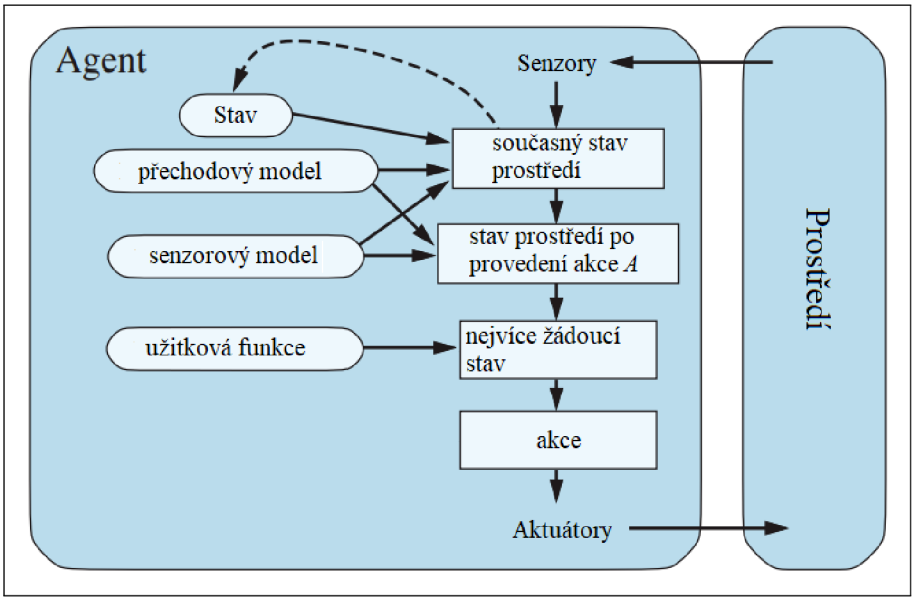
Racionální **agent s užitkovou funkcí** volí takovou akci, která maximalizuje očekávaný užitek. Očekávaný užitek akce se počítá váženým průměrem užitku každého možného výsledku (stavu), kterého agent volbou dané akce dosáhne. Váhou je pravděpodobnost daného výsledku (stavu) vzhledem k prováděné akci a^*. Agent tedy volí akci

$$a^* = arg \max_{a_i \in A} \sum_{s_j \in S} p(s_j|a_i) \cdot U(s_j) \tag{4}$$

kde A je množina všech proveditelných akcí, S je množina všech stavů po provedení dané akce, $p(s_j|a_i)$ je pravděpodobnost přechodu ze současného stavu do stavu s_j při volbě dané akce a $U(s_j)$ je užitek stavu s_j (Li & Soh, 2004).

Schéma agenta s užitkovou funkcí je na obrázku 9.

Obrázek 9: Schéma agenta s užitkovou funkcí



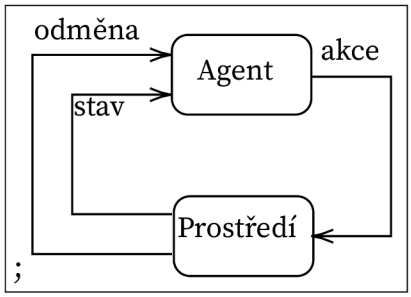
Zdroj: Norvig and Russel (2020)

2 Posilované učení

2.1 Základní prvky posilovaného učení

Při posilovaném učení (též zpětnovazebním učení) agent vnímá prostředí, provádí akce a přijímá zpětnou vazbu ve formě odměn. Cílem agenta je naučit se chovat tak, aby v dlouhodobém časovém horizontu maximalizoval očekávané odměny (Géron, 2017).

Obrázek 10: Základní princip posilovaného učení



Zdroj: vlastní zpracování

Posilované učení zahrnuje 6 dílčích prvků (Sutton & Barto, 2020):

- prostředí,
- agenta,
- agentovu strategii,
- odměnový signál,
- hodnotovou funkci,
- a případně model prostředí

Prostředí a agenti byli popsáni v předchozí části. Agentova *strategie* (anglicky *policy*) definuje způsob chování agenta v daném čase. Mapuje stavy prostředí na akce, které mají být provedeny. Strategie může být realizována jako jednoduchá funkce nebo vyhledávací tabulka, v jiných případech však může zahrnovat rozsáhlé výpočty a podúlohy.

Agent získává v každém časovém okamžiku od prostředí prostřednictvím *odměnového signálu* (anglicky *reward signal*) jedno číslo - odměnu. Odměnový signál je hlavním prvkem ovlivňujícím strategii agenta: pokud po zvolené akci následuje nízká odměna, agent může svou strategii upravit, aby v budoucnu v dané situaci zvolil lepší akci.

Hodnotová funkce určuje, co je pro agenta žádoucí z dlouhodobého hlediska. Hodnota stavu je celková očekávána výše odměny, kterou agent začínající v daném stavu

v budoucnu získá. Zatímco odměny určují jak jsou stavy žádoucí v aktuálním čase, hodnoty udávají jak jsou stavy žádoucí z dlouhodobého časového hlediska. Hodnoty zohledňují, respektive predikují očekávané odměny v budoucích stavech.

Hodnoty se určují obtížněji než odměny. Odměny jsou získány přímo z prostředí, ale hodnoty se určují na základě historie vjemů agenta.

Model prostředí imituje chování prostředí, ve kterém agent působí. Model umožňuje odhadovat vývoj prostředí a jeho budoucí stavy.

2.2 Druhy metod posilovaného učení

Všechny metody posilovaného učení lze klasifikovat dle různých charakteristik. Lapan (2018) uvádí dělení na:

- model-free nebo model-based
- value-based nebo policy-based
- on-policy nebo off-policy

Model-free metody nevytváří model prostředí ani odměn, pouze přímo propojuje pozorování stavů prostředí na akce. Model-based metody se naopak snaží predikovat budoucí stavy a případně i odměny. Model-based metody se obvykle používají v deterministických prostředích s pevně danými a známými pravidly, např. deskové hry. Model-free metody jsou naopak vhodné u úloh se složitým, nedeterministickým prostředím a jelikož tedy tyto metody nezahrnují složitý model prostředí, je jejich trénování snadnější.

Policy-based metody přímo aproximují strategii agenta, tzn. kterou akci má agent v každém kroku v prostředí učinit. Strategie je obvykle reprezentována pravděpodobnostním rozdělením jednotlivých akcí. Value-based metody ohodnocují akce hodnotou a agent volí dle určitých kritérií akce.

On-policy metody se snaží vyhodnocovat a zlepšovat strategii, dle které agent volí akce. Při off-policy metodách se agent průzkumem prostředí přibližuje k optimální strategii.

2.3 Explorace a exploatace prostředí

Agent používá hodnotové funkce k tomu, aby vylepšil svoji strategii. Výběr vhodné strategie pro učení je složitý problém, při kterém se musí vyvažovat prohledávání prostředí (explorace) a využívání známých informací (exploatace) (Pilát, n.d.).

Agent předem hodnoty stavů nezná a učí se je za běhu. Bude-li mít špatný odhad určitého stavu, je možné, že ten stav nikdy nenavštíví. Přitom by však ten stav při použití odlišné strategie učení byl pro agenta velmi žádoucí.

Metod výběru akce je vícero. Častou metodou je ϵ-hladová strategie, dle které agent ve stavu s zvolí s pravděpodobností ϵ k provedení akci náhodně a s pravděpodobností $(1-\epsilon)$ zvolí k provedení nejlepší akci (např. s nejvyšší Q-hodnotou). Obvykle se začíná s $\epsilon = 1$ a postupně se pomalu snižuje na malou hodnotu, např. $\epsilon = 0.05$.

2.4 Markovův rozhodovací proces

Markovův rozhodovací proces (MRP) je formální matematický rámec obecně popisující chování agenta v úlohách řešených pomocí posilovaného učení (Vlassis, 2007).

Konečný MRP jednoho agenta se skládá z následujících částí:

S	diskrétní množina stavů
A	diskrétní množina akcí
t	diskrétní čas
$p(s_i \to s_{i+1} \mid a_i)$	pravděpodobnost přechodu ze stavu s_i v čase t do stavu s_{i+1} v čase $t+1$, zvolí-li agent v s_i akci a_i
R	odměnová funkce $R : S \times A \to \mathbb{R}$. Agent obdrží odměnu $R(s_i, a_i)$ zvolí-li ve stavu s_i akci a_i
H	plánovací horizont (může být nekonečný)

Úkolem agenta je najít optimální strategii π^*, která v plánovacím horizontu maximalizuje kumulativní odměnu. K nalezení takové strategie potřebuje znát optimální hodnoty jednotlivých stavů. Ty lze určit pomocí tzv. Bellmanových rovnic:

$$V^*(s_i) = \max_{a_i} \sum_{s_{i+1}} p(s_i \to s_{i+1}|a_i)[R(s_i, a_i) + \gamma V^*(s_{i+1})] \qquad \forall s_i \tag{5}$$

kde $V^*(s_i)$ je optimální hodnota stavu s_i, též *V-hodnota*, $\gamma \in [0, 1)$ je diskontní faktor.

Neboli pokud se agent chová optimálně, pak optimální hodnota je rovna odměně, kterou v průměru získá po provedení jedné optimální akce, plus očekávané optimální hodnoty všech možných dalších stavů, ke kterým může volba této akce vést.

Optimální strategie $\pi^*(s_i)$ je pak

$$\pi^*(s_i) = arg \max_{a_i} \sum_{s_{i+1}} p(s_i \to s_{i+1}|a_i) V^*(s_{i+1}) \tag{6}$$

2.5 Vybrané metody posilovaného učení

V této podkapitole jsou představeny některé z metod posilovaného učení. Jelikož metod a jejich variant existuje mnoho, je tato podkapitola zaměřena spíše na představení základních metod a principů, na kterých tyto metody stojí, nežli na jejich detailní matematicko-formalistický popis.

2.5.1 Q-učení

Q-učení je založeno na hodnotách nikoliv stavů, nýbrž hodnotách akcí. Pro každý stav s_i a každou proveditelnou akci a_i se definuje hodnota akce neboli Q-hodnota:

$$Q(s_i, a_i) = \sum_{s_{i+1}} p(s_i \to s_{i+1}|a_i)[R(s_i, a_i) + \gamma \cdot \max_{a_{i+1}} Q(s_{i+1}, a_{i+1})] \qquad \forall (s_i, a_i) \tag{7}$$

Q-hodnota $Q(s_i, a_i)$ vyjadřuje očekávanou celkovou diskontovanou odměnu, pokud agent provede ve stavu s_i akci a_i a poté bude jednat optimálně.

Q-hodnoty jsou zaneseny v Q-matici, která má rozměry dle celkového počtu stavů a akcí. Obvykle jsou na začátku všechny prvky Q-matice inicializovány na nulu. Pokud agent provede ve stavu s_i akci a_i, upraví se příslušná Q-hodnota v matici dle vzorce

$$Q(s_i, a_i) = Q(s_i, a_i) + \alpha \cdot [R(s_i, a_i) + \gamma \max_{a_{i+1}} Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i)] \tag{8}$$

kde α ($0 \leq \alpha \leq 1$) je učící člen, který udává do jaké míry jsou staré informace přepsány novými. V případě $\alpha = 0$ se agent nebude vůbec učit nové informace, v případě $\alpha = 1$ naopak agent při učení plně uvažuje nově získané informace.

Algoritmus Q-učení (pro jednu epizodu učení) probíhá následovně (Sutton & Barto, 2020):

1. inicializace $Q(s, a)$ pro všechny (s, a) na libovolnou hodnotu, $Q(s_{koncovy}, -)$ na hodnotu 0
2. inicializace stavu s
3. zvolit ve stavu s akci a podle strategie
4. pozorovat odměnu R a nový stav s_n
5. upravit hodnotu $Q(s, a)$ podle vztahu v rovnici (8).
6. $s = s_n$
7. opakovat kroky 2-6 dokud není dosažen koncový stav

Q-učení je jedna z nejpoužívanějších metod posilovaného učení, zejména protože bylo dokázáno, že v Q-učení dvojice (s, a) konvergují k optimu, značeno Q^* (Watkins & Dayan, 1992). Optimální strategie je potom

$$\pi^*(s) = arg\ \max_a\ Q^*(s, a) \tag{9}$$

2.5.2 SARSA

Akronym SARSA značí State-Akce-Reward-State-Akce, zapsáno formálněji: $[(s_i, a_i), r, (s_{i+1}, a_{i+1})]$, respektive s notací Q hodnot takto

$$Q(s_i, a_i) = (1 - \alpha)Q(s_i, a_i) + \alpha(r + \gamma Q(s_{i+1}, a_{i+1})) \tag{10}$$

a tuto rovnici lze přepsat do tvaru

$$Q(s_i, a_i) = Q(s_i, a_i) + \alpha \cdot [R(s_i, a_i) + \gamma\, Q(s_{i+1}, a_{i+1}) - Q(s_i, a_i)] \tag{11}$$

která je téměř identická rovnici 8 pro aktualizaci Q-hodnoty v Q-učení popsanému v předcházející sekci. Algoritmus SARSA je tedy velmi podobný Q-učení, ale s tím rozdílem, že SARSA aplikuje stejnou strategii pro výběr akce (tj. chování agenta) i pro aktualizaci Q-hodnot, zatímco při Q-učení se pro chování agenta i pro aktualizaci Q-hodnot uplatňují dvě různé strategie. SARSA volí akci a_i pomocí jedné strategie, například ϵ-hladové. Touto strategií zároveň volí akci a_{i+1} a následně ve výpočtu aktualizace Q-hodnoty použije γ-diskontovaný odhad Q-hodnoty v příští iteraci, tj. $Q(s_{i+1}, a_{i+1})$. Q-učení volí akci a_i ϵ-hladovou strategií, ale v rámci aktualizace hodnoty používá takovou akci a_{i+1}, při které je γ-diskontovaný odhad Q-hodnoty $Q(s_{i+1}, a_{i+1})$ největší.

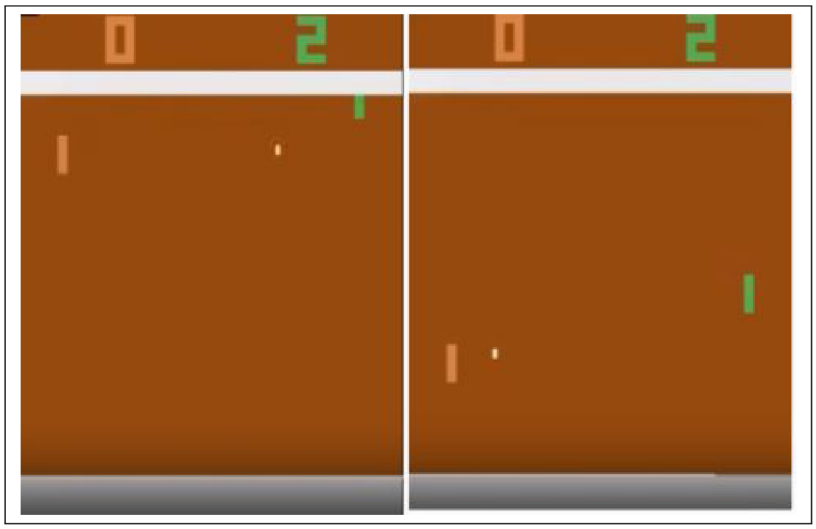
2.5.3 Hluboké Q-učení

Hlavní nevýhodou standardního Q-učení je jeho škálovatelnost. MRP s mnoha stavy a akcemi vedou na velké Q-tabulky a agent se kvůli tomu učí velmi pomalu. Lapan (2018) jako jeden z příkladů prostředí s velkým počtem stavů uvádí hru Pong. Počet stavů se odvíjí od počtu pixelů, pozice pálek jednotlivých hráčů a pozice míčku. Reprezentovat prostředí pomocí Q-tabulky zachycující stav jednotlivých pixelů (nebo jejich skupin, jelikož rozdíl v pozici míčku o pár pixelů je zanedbatelný) je z výpočetního hlediska náročné. Na obrázku 11 jsou dva odlišné stavy prostředí během hry.

Hluboké Q-učení aproximuje optimální Q-funkci $Q^*(s, a)$ pomocí hluboké neuronové sítě (též nazývaná „Q-síť“) s parametry θ, která pro každý stav vrací ohodnocení všech akcí. Čili se hledá Q-funkce taková, že $Q_\theta(s, a) \approx Q^*(s, a)$. Síť se trénuje tak, aby při každé iteraci i minimalizovala ztrátovou funkci $L_i(\theta_i)$

$$L_i(\theta_i) = (R(s_i, a_i) + \gamma \max_{a_{i+1}} Q_{\theta_i}(s_{i+1}, a_{i+1}) - Q_{\theta_i}(s_i, a_i))^2 \tag{12}$$

Obrázek 11: Dva různé stavy prostředí ve hře Pong



Zdroj: Lapan (2018)

K optimalizaci ztrátové funkce lze použít algoritmy gradientního sestupu (Mnih et al., 2013).

Hluboké učení má však za určitých podmínek i nezanedbatelné nevýhody. Je-li vstupem Q sítě obraz, například snímek obrazovky (viz obrázek 11), získá při hře síť

v krátkém čase mnoho snímků za sebou. Jelikož tyto snímky jdou po sobě, mají takovéto vstupy mezi sebou vysokou korelaci. V základním algoritmu Q-učení jsou Q-hodnoty aktualizovány v každém kroku. Následkem tohoto je, že váhy Q sítě jsou aktualizovány v určitém směru. Při minimalizací ztrátové funkce aktualizací vah neuronové sítě tedy může dojít k uváznutí v lokálních minimech. Optimalizace ztrátové funkce může být při vyvstání těchto jevů obtížná a trénování Q sítě může vyžadovat použití velmi složitých optimalizátorů (Sewak, 2019).

Byly navrhnuty různé techniky jak výše uvedené problémy učení Q-sítě odstranit nebo zmenšit. Dvěmi z nich jsou paměť zkušeností (experience replay) a využití cílové sítě (target network). *Paměť zkušeností* je technika, kdy si agent ukládá podmnožinu svých zkušeností ve tvaru $(s_i, a_i, R(s_i, a_i), s_{i+1})$ do vyrovnávací paměti o pevně dané velikosti a tudíž přidání nejnovější zkušenosti do paměti znamená odstranění nejstarší zkušenosti uložené v paměti. Při trénování formou Q-učení se pak vybírá náhodná zkušenost z paměti. Paměť zkušeností umožňuje trénovat síť na zkušenostech které jsou mezi sebou víceméně nezávislé, ale zároveň jsou dostatečně aktuální (Sewak, 2019).

Princip *cílové sítě* spočívá ve dvou odlišných neuronových sítích: jedna (základní) síť s parametry θ se používá pro výběr akce a druhá (cílová) síť s parametry θ^- pro odhad Q-hodnot. Myšlenkou je zlepšit stabilitu učení tím, že Q-hodnoty cílové sítě se užívají jako vstupy k učení základní sítě. Ztrátová funkce je až na drobnou změnu stejná jako v rovnici 12:

$$L_i(\theta_i) = (R(s_i, a_i) + \gamma \max_{a_{i+1}} Q_{\theta^-}(s_{i+1}, a_{i+1}) - Q_{\theta_i}(s_i, a_i))^2 \tag{13}$$

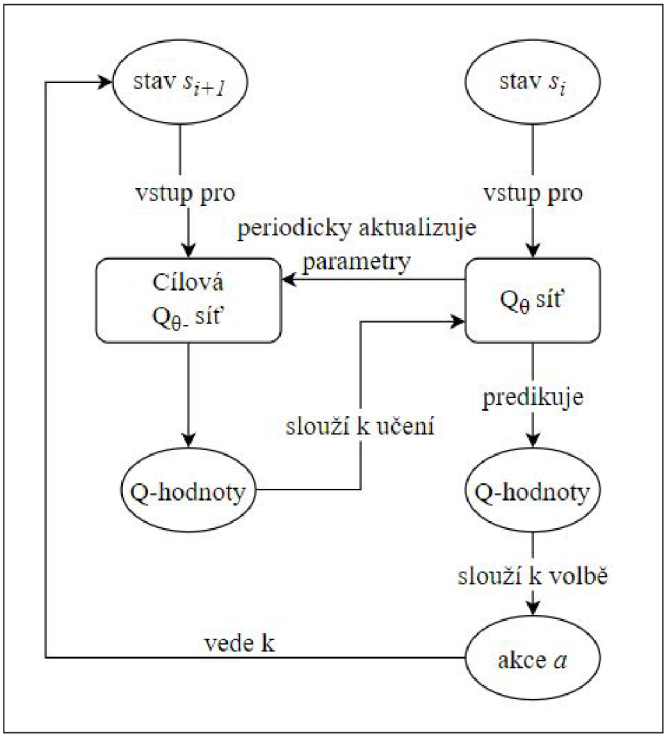
Algoritmus s využitím cílové sítě je následující (Brown & Zai, 2020):

1. inicializace základní Q-sítě s parametry θ
2. inicializace cílové sítě jako kopie základní Q-sítě, ale s vlastními parametry θ^-, nastavit $\theta^- = \theta$
3. ϵ-hladovou strategií základní Q-sítě vybrat akci a_i
4. pozorovat odměnu $r = R(s_i, a_i)$ a stav s_{i+1}
5. nastavit Q-hodnotu cílové sítě na r pokud epizoda hry skončila, jinak nastavit na $r + \gamma \max_{a_{i+1}} Q_{\theta^-}(s_{i+1}, a_{i+1})$

6. zpětně propagovat Q-hodnotu cílové sítě do základní sítě, čímž dojde k úpravě parametrů θ
7. periodicky po C krocích nastavit $\theta^{-} = \theta$

Obecný průběh algoritmu je na obrázku 12.

Obrázek 12: Hluboké Q-učení s cílovou sítí



Zdroj: Upraveno podle Brown and Zai (2020)

2.5.4 Metody aproximující strategii

Tyto aproximační metody k dosažení optimální agentovy strategie π^* nevyužívají hodnotovou funkci, nýbrž upravují přímo agentovu strategii π. Strategie π je parametrizována skupinou parametrů θ a hledají se jejich optimální hodnoty. To lze činit použitím různých aproximačních technik, například genetických algoritmů, ale nejčastěji používanou technikou je *gradient strategie* (anglicky *policy gradient*). (Heeswijk, W., 2022a)

V případě Q-učení byla optimální strategie určena deterministicky - těmi největšími - hodnotami z Q-tabulky (viz rovnice 9). Strategie π_θ u metody gradientu strategie je naopak stochaistická, parametry θ ovlivňují podobu pravděpodobnostního rozdělení akcí.

Pohyb agenta prostředím lze popsat trajektorií stavů a akcí τ:

$$\tau = (s_1, a_1, s_2, a_2, ..., s_n, a_n)$$

kde pro $t, 1 \leq i \leq n$, je s_t stav, ve kterém se agent v čase t nacházel a a_t je akce, kterou v čase t zvolil.

Cílem agenta je maximalizovat očekávanou kumulativní odměnu, čili

$$\max_{\theta} J(\theta) = \max_{\theta} \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} R(\tau) \tag{14}$$

kde $J(\theta)$ se nazývá objektivní funkce, $\tau \sim \pi_\theta$ je distribuce trajektorií při strategii π_θ a $R(\tau)$ je kumulativní odměna trajektorie τ.

Pro gradient funkce $J(\theta)$ lze odvodit tvar

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \nabla_\theta log P(\tau, \theta) R(\tau) \tag{15}$$

který lze pro n vzorků trajektorií aproximovat jako

$$\nabla_\theta J(\theta) \approx \frac{1}{n} \sum_{i=1}^{n} log \nabla_\theta P(\tau^{(i)}, \theta) R(\tau^{(i)}) \tag{16}$$

kde $P(\tau, \theta)$ je pravděpodobnost trajektorie τ při strategii π_θ. (Heeswijk, W., 2022a)

Parametry θ se pak postupně aktualizují dle vztahu

$$\Delta\theta = \alpha \nabla_\theta J(\theta) \tag{17}$$

ke $\alpha \in (0, 1]$ je míra učení.

Nevýhodou výše uvedeného způsobu jsou velké změny parametrů θ, což konvergenci k parametrům θ^* zpomaluje, nebo dokonce může způsobit uváznutí v oblasti, ze které již pomocí dalších aktualizací nebude možno dokonvergovat k θ^*.

S tímto problémem se lze vypořádat různě, jedním z novějších algoritmů je PPO (Proximal Policy Optimization). Stručně řečeno, PPO pro aktualizaci parametrů strategie používá funkci s členem $r = \pi_{\theta\,\text{nová}} / \pi_{\theta\,\text{stará}}$. Poměr r je ořezán tak, aby $r \in [1 - \epsilon, 1 + \epsilon]$, kde ϵ je hyperparametr (v původním článku o PPO byl $\epsilon = 0.2$). Ořezáním je zajištěno, že aktualizace parametrů nejsou příliš velké (tj. že se stará a nová strategie od sebe příliš neliší), což vede k tomu, že agentovo učení se strategie je více stabilní.(Simonini, 2022b)

Detailnější popis PPO je k přečetení v (Schulman et al., 2017), (Bick, 2021) nebo (Heeswijk, W., 2022b).

2.5.5 Actor-Critic

Actor-Critic metody propojují value-based a policy-based metody ve snaze posílit jejich klady a utlumit jejich zápory. Principem je existence dvou sítí, resp. funkcí:

- *Actor*, π_θ je síť (funkce) strategie
- *Critic*, q_w je síť (funkce) hodnot

kdy Actor volí akce a Critic ohodnocuje, jak dobře Actor volí. Actor i Critic si postupně upravují své parametry. Proces učení probíhá dle Simonini (2022a) následovně:

1. v čase t Actor i Critic přijmou stav s_t
2. Actor zvolí akci a_t
3. Critic spočítá hodnotu provedení akce a_t ve stavu s_t: $q_w(s_t, a_t)$
4. provedením akce a_t se dosáhne odměny $R(s_t, a_t)$ a nového stavu s_{t+1}
5. Actor aktualizuje své parametry dle vztahu

$$\Delta\theta = \alpha\nabla_\theta(log\pi_\theta(s_t, a_t)) \cdot q_w(s_t, a_t) \tag{18}$$

   kde α je míra učení sítě Actor, $\pi_\theta(s_t, a_t)$ je pravděpodobnost zvolení akce a_t ve stavu s_t při strategii π_θ.
6. Critic aktualizuje své váhy dle vztahu

$$\Delta w = \beta\left[R(s_t, a_t) + \gamma q_w(s_{t+1}, a_{t+1}) - q_w(s_t, a_t)\right] \cdot \nabla_w q_w(s_t, a_t) \tag{19}$$

   kde β je míra učení sítě Critic a γ je diskontní míra.

Ze základních Actor-Critic principů popsaných v odstavcích vyšel algoritmus *Advantage Actor-Critic*, česky lze přeložit jako *Actor-Critic s výhodou*. Myšlenka algoritmu spočívá v tom, že Critic nepočítá hodnoty párů $(stav, akce)$, ale tzv. výhody:

$$A(s, a) = Q(s, a) - V(s) \tag{20}$$

kde $A(s, a)$ je výhoda, $Q(s, a)$ je hodnota provedení akce a_t ve stavu s_t a $V(s)$ je průměrná hodnota stavu s. Výhoda udává, o kolik lepší je provést danou akci v daném stavu ve srovnání s průměrnou hodnotou stavu.

2.6 Nástroje pro implementaci posilovaného učení

V této části jsou představeny některé z populárních platforem a programátorských nástrojů pro vývoj agentů anebo prostředí pro účely posilovaného učení.

Nejpopulárnější platformou pro vývoj v oblasti posilovaného učení je **Open AI Gym**. Je to knihovna pro programovací jazyk Python, která poskytuje jednotné rozhraní pro mnoho prostředí s různými typy řešených problémů a pro různé typy aplikačních domén (Sewak, 2019).

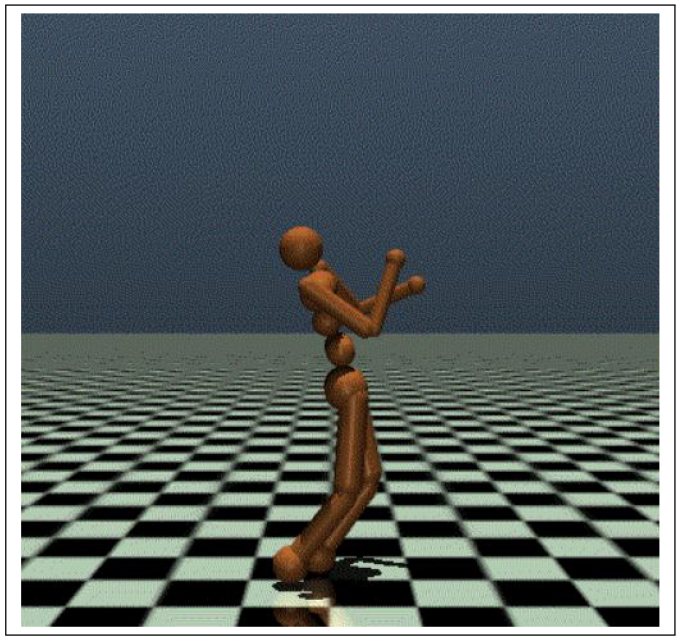
Celkem Gym nabízí více než 700 prostředí, byť mnohá jsou jen modifikované verze jednoho základního prostředí (Bongsang, 2018). Dokumentace Gym uvádí následující skupiny prostředí ("Gym Documentation", 2023) :

- Atari - sada prostředí simulující různé hry pro starou videoherní konzoli Atari 2600. Prostor akcí má velikost 18 (akce „doleva", „doprava", „vystřelit", a další), prostor pozorování může být RGB obrázek, čili obrazovka hry, nebo 128 MB paměť konzole. Na obrázku 13 si lze prohlédnout prostředí hry Assault.
- MuJoCo (Multi-Joint dynamics with Contact) - sada prostředí pro simulaci pohybu tvorů. Je využito open-source knihovny MuJoCo pro pokročilé fyzikální simulace. Ukázka prostředí s humanoidní figurkou je na obrázku 14.
- Toy Text - čtyři prostředí pro jednoduché textově-založené hry.
- Classic Control - pět prostředí s klasickými fyzikálními problémy. Například v prostředí Mountain Car je agentem auto na úpatí kopce, jedinou akcí je zrychlení (s diskrétní nebo spojitou velikostí) doleva nebo doprava a cílem je, aby auto vyjelo na kopec.
- Box2D - tři prostředí pro hry založené na kontrolování fyzikálních parametrů. Například prostředí Lunar Lander simulující přistání na Měsíci, cílem hry je korigovat trajektorii tak, aby modul přistál bezpečně.

Obrázek 13: Gym prostředí pro Atari hru Assault

Zdroj: "Gym Documentation - Atari" (2023)

Obrázek 14: Gym prostředí pro simulaci pohybu humanoidní postavy



Zdroj: "Gym Documentation - MuJoCo" (2023)

Stable-Baselines3 (vycházející z Open AI Baselines) je platforma obsahující implementaci různých algoritmů a agentů posilovaného učení. Prostředí jsou kompatibilní s Open AI Gym. Dále Stable-Baselines3 nabízí i integraci s měřícím nástrojem TensorBoard a před-trénované agenty pro některá prostředí (Raffin et al., 2021a).

Platforma je modulární a díky tomu umožňuje rychle a programátorsky přívětivé psát aplikace posilovaného učení s různými typy agentů a prostředí.

DeepMind Lab je framework pro Python od Google poskytující pokročilá 3D prostředí, která umožňují vykreslovat scény s pokročilými vizuálními prvky. Agenti ve 3D prostředích plní navigačních úlohy nebo různé hádanky. DeepMind Lab se zaměřuje zejména na hluboké posilované učení (Alphabet Inc., 2018).

Project Malmo je AI experimentální platforma od společnosti Microsoft, založená na populární hře Minecraft. Platforma, stejně jako Minecraft, je napsaná v jazyce Java, ale agenti mohou být naprogramování v různých populárních programovacích jazycích (Microsoft Corporation, n.d.).

RLlib je open-source Python knihovna, která nabízí podporu pro vysoce distribuované úlohy posilovaného učení a zároveň zachovává jednotné a jednoduché rozhraní pro nejrůznější průmyslové aplikace. RLib je široce používaná v praxi v mnoha oblastech (výroba, logistika, videohry, robotika, a další). (The Ray Team, n.d.)

Tensorforce je open-source knihovna pro Python kladoucí důraz na flexibilní a modulární tvorbu aplikací posilovaného učení a snadnou použitelnost v oblasti výzkumu i průmyslu. Tensorforce je postaven na frameworku TensorFlow od Google (Tensorforce Team, n.d.).

MATLAB® obsahuje Reinforcement Learning Toolbox, který nabízí moduly pro trénování strategií pomocí několika algoritmů posilovaného učení. (The MathWorks, Inc., n.d.)

3 Simulace dopravy

Dopravu a s ní související problémy lze simulovat různě. V této kapitole jsou stručně uvedeny základní typy simulačních modelů dopravy a některé významné dopravní problémy.

3.1 Simulační modely

Způsobů simulace dopravy a jejích jednotlivých prvků a jevů existuje mnoho. Dle toho, na jaké úrovni detailu je doprava simulována, se obecně rozdělují tři typy simulací, resp. modelů dopravy:

- Mikroskopické modely slouží k detailní simulaci chování dopravního systému a jeho jednotlivých prvků. S využitím matematických a fyzikálních rovnic simulují pohyb každého jednotlivého vozidla s ohledem na jeho konkrétní vlastnosti, jako je hmotnost, rychlost a zrychlení v závislosti na okolních vozidlech a prostředí na silnici.
- Makroskopické modely nerozlišují jednotlivé prvky a zaměřují se na vlastnosti dopravního toku na vyšší agregované úrovni. Dopravní tok je v makroskopických modelech simulován podobně jako tok plynu nebo tekutiny.
- Mezoskopické modely jsou kombinací mikroskopických a makroskopických modelů. Existují dva hlavní přístupy k mezoskopické simulaci dopravy: v prvním se jednotlivá vozidla neberou v úvahu a vozidla jsou sdružena do skupin, jejichž pohyb a interakce jsou popsány příslušnými rovnicemi. Ve druhém přístupu je dopravní tok reprezentován jednotlivými vozidly, avšak jejich pohyb je popsán jednoduššími rovnicemi než v případě mikroskopických simulací.

Modely lze dělit i na základě jiných hledisek, například na statické a dynamické. U statických modelů zůstávají vstupní parametry modelu, například frekventovanost silnice, v čase neměnné, zatímco u dynamických modelů se parametry s časem mění v závislosti na různých proměnných (Elefteriadou, 2014).

Silniční sítě jsou obvykle reprezentovány pomocí grafů. Grafem se rozumí uspořádaná dvojice (V, E), kde V je množina vrcholů (uzlů) a E je množina hran (spojů. linek). Hrany mohou být orientované a mít přiřazené váhy. (Barceló, 2010) Dle povahy

zkoumané úlohy lze k základním prvkům grafu přidávat další parametry, například parametr hrany reprezentující maximální kapacitu silnice a parametr využití kapacity v konkrétním časovém okamžiku.

3.2 Vybrané problémy

Existuje vícero různých problémů, které se v praxi vyskytují v oblasti dopravy, ale též i jinde. Níže jsou uvedeny některé z nich:

- *problém nejkratší cesty*: cílem je najít ve váženém spojitém grafu takovou cestu mezi výchozím vrcholem a cílovým vrcholem, která má minimální váhu. V kontextu dopravy se jedná o nalezení nejrychlejší trasy mezi dvěma místy v silniční síti, kde je známa doba jízdy pro každý úsek sítě. Anebo pokud je známá délka každého úseku, lze najít nejkratší trasu mezi dvěma místy, kdy délka je minimální. (Zverovich, 2021)
- *problém rovnováhy*: Dojíždění v silniční síti lze formalizovat jako hru s nenulovým součtem n hráčů (řidičů), kteří spolu soupeří a jejichž cílem je dojet do cílového vrcholu s co nejmenšími náklady (časovými, finančními, aj.) Hráči si vybírají spoje v síti až dokud není dosaženo rovnováhy. V rovnováze nemůže žádný hráč snížit svoje náklady na cestu zvolením jiné cesty. S tím souvisí Braessův paradox, dle kterého přidání nového spoje do sítě může vést k rovnovážnému stavu s vyššími náklady hráčů, než jaké byly před přidáním spoje.(Zverovich, 2021)
- *problém obchodního cestujícího*: hledá se nejkratší (nejméně nákladná) cesta v grafu, přičemž každý vrchol grafu je navštíven právě jednou s výjimkou výchozího uzlu, který je zároveň posledním navštíveným uzlem. (Zverovich, 2021)
- *problém okružních jízd (anglicky Vehicle Routing Problem)*: zobecnění problému obchodního cestujícího. Cílem je navrhnout takovou množinu cest s minimálními náklady, které začínají a končí v jednom uzlu, pro skupinu aktérů obsluhujících množinu uzlů. Každý uzel je obsloužen právě jednou. Jde o NP-těžký problém, který se velmi často vyskytuje v praxi. (Toth & Vigo, 2014)
- *problém čínského listonoše*: cílem je najít nejkratší (nejméně nákladnou) cestu v grafu, která zahrnuje všechny hrany grafu a která končí ve výchozím bodě.

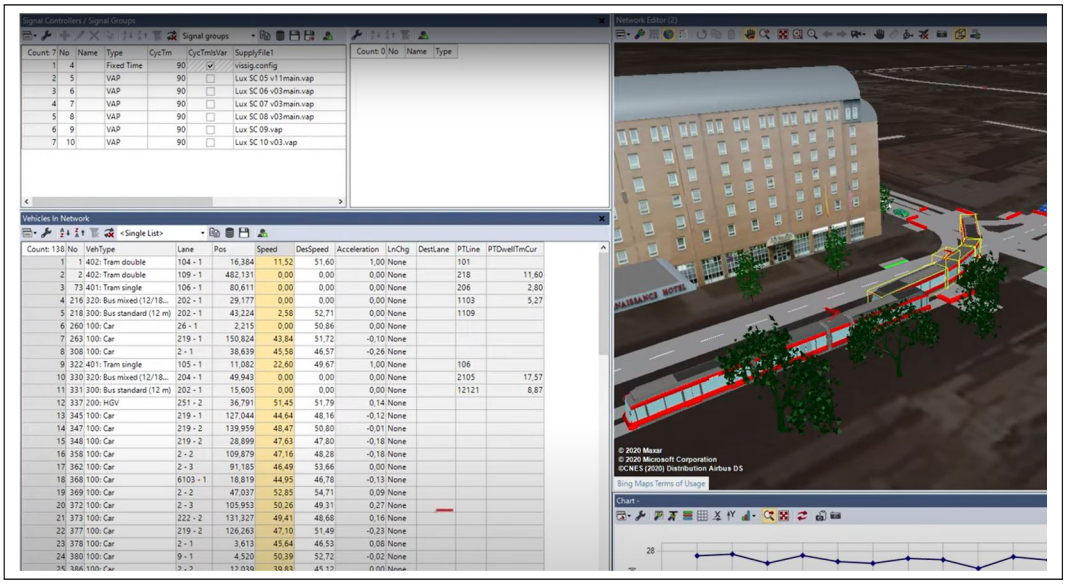
Každý z výše uvedených problémů existuje ve vícero variantách.

3.3 Nástroje pro simulaci dopravy

Vzhledem k různorodosti simulačních modelů existují různé pokročilé simulátory a nástroje pro simulaci dopravy. Níže jsou uvedeny jedny z v praxi nejvíce rozšířených.

VISSIM je rodina mikroskopických i mezoskopických víceúčelových simulátorů různých aspektů dopravy. Nabízené aplikace pokrývají širokou škálu oblastí od veřejné městské dopravy, dálniční sítě, navigace, logistiku poslední míle a jiné. (PTV Group, n.d.)

Obrázek 15: Ukázka práce v nástroji VISSIM



Zdroj: PTV Group (n.d.)

SUMO (**S**imulation of **U**rban **MO**bility) je open-source mikroskopický simulátor dopravy. Lze jím modelovat pohyb silničních vozidel, chování chodců či optimalizovat světelné signály. SUMO poskytuje aplikační rozhraní pro vytváření a ovládání vlastních modelů.(Eclipse Foundation, n.d.)

AnyLogic je víceúčelový simulační software. Lze jím simulovat dopravu od mikroskopické úrovně (např. pohyb vozidel křižovatkou) až po úroveň globálních dodavatelských řetězců a mezikontinentální dopravy. Nabízí i moduly a nástroje pro modelování a simulaci v dalších odvětvích jako je výroba produktů, skladování, podnikové procesy a další.(The AnyLogic Company, n.d.)

4 Posilované učení pro návrh trasy

V této kapitole je popsán návrh aplikace vyvinuté v rámci praktické části, včetně formalizace zkoumané problematiky dle teorie popsané v předcházející teoretické části této práce.

4.1 Formulace jako Markovův rozhodovací problém

Teoreticky je Markovův rozhodovací problém (MDP) popsán v sekci 2.4. Pro problém nejkratší trasy řešený aplikací je MDP následující:

- **množina stavů** S rovna množině všech uzlů silniční sítě. Stav agenta v daném čase je uzel, ve kterém se agent v daném čase nachází.
- **množina akcí** A je množina všech hran mezi uzly silniční sítě.
- **čas** t je diskrétní, přechod agenta z jednoho stavu (uzlu) do následujícího trvá jednu jednotku času, tzn. aplikace nesimuluje reálný čas dojezdu.
- **pravděpodobnosti přechodů** mezi stavy (uzly) není nutno uvažovat, jelikož je předpokládáno, že přechod mezi jednotlivými stavy se v každém časovém okamžiku úspěšně provede.
- **odměnová funkce** R je blíže popsána v sekci 4.1.1.
- **plánovací horizont** H je konečný.

Silniční síť je reprezentována jako neorientovaný graf s nezápornými vahami, které odpovídají délce jednotlivých silnic. Uvažované prostředí je

- plně pozorovatelné - agent má v každém časovém okamžiku přístup ke všem uzlům sítě a může si o nich zjistit informace
- sekvenční - volba jedné akce (silnice) v čase t ovlivní, které akce (silnice) agent volí v čase $t + 1$, případně i $t + 2$, $t + 3$ a tak dále.
- dynamické - silniční síť se může měnit (uzavření silnic)
- diskrétní - silniční síť má konečný počet silnic a uzlů.
- deterministické - následující stav (uzel) je v uvažovaném problému vždy jednoznačně určen současným stavem a akcí, tj. neuvažuje se, že by agent po silnici nemusel uspět v dojetí do dalšího uzlu.
- jednoagentní - uvažuje se návrh trasy pro jedno vozidlo

4.1.1 Odměnová funkce

Návrh odměnové funkce je významný pro účinnost algoritmu. Vyvíjená aplikace má odměny definovány takto:

$$R(s,a) = \begin{cases} -w_n & \text{pokud } s \notin P \text{ nebo pokud } s \in P, \text{ ale již byl navštíven} \\ +5 & \text{pokud } s \in P \text{ a je navštíven poprvé} \\ -5 & \text{pokud akci } a \text{ nelze ve stavu } s \text{ provést} \end{cases} \tag{21}$$

kde s je aktuální stav, ve kterém se agent nachází, P je množina uzlů, které mají být obslouženy, a je zvolená akce a $w_n \in [0,1]$ je normalizovaná váha w hrany dle vztahu

$$w_n = \frac{w - w_{min}}{w_{max} - w_{min}} \tag{22}$$

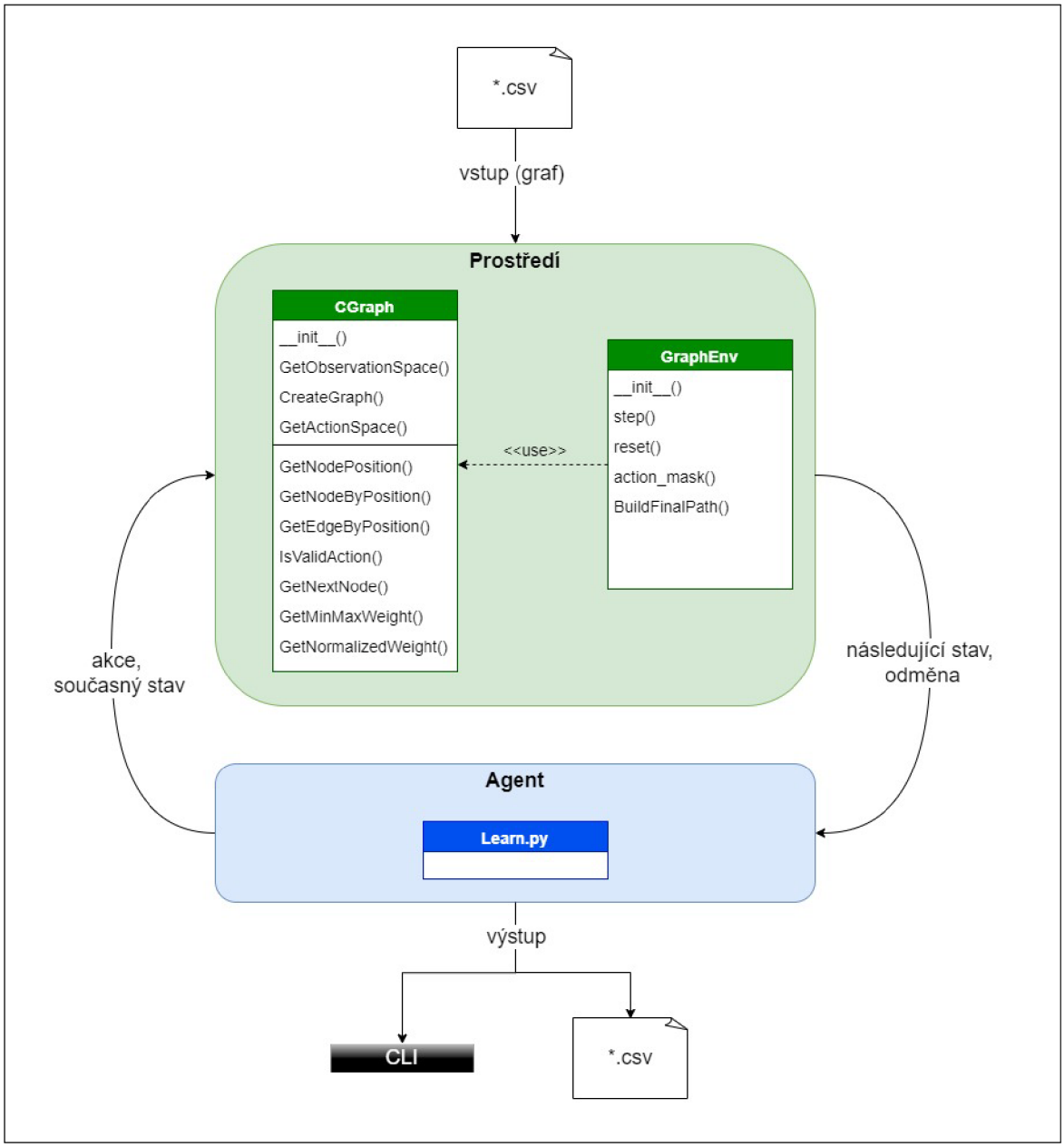
kde w_{min}, resp. w_{max} je nejmenší, resp. největší váha z množiny vah všech hran v síti.

Výše uvedená odměnová funkce je tedy navržena tak, aby agent byl motivován navštívit všechny uzly, jež má obsloužit (obdrží v nich vysokou odměnu), zároveň aby se okolo těchto uzlů necyklil (vysokou odměnu obdrží jen při prvním navštívení) a aby tak učinil s co nejmenší ujetou vzdáleností (záporná odměna za každou další jízdu).

4.2 Návrh aplikace

Aplikace je konzolová a je nasazená v lokálním prostředí. Její algoritmus v principu probíhá dle cyklu posilovaného učení popsaného v části 2.1. Vstupem je `csv` soubor se strukturou dané silniční sítě, jež je v aplikaci reprezentována třídou `CGraph`. Třída `GraphEnv` vytváří prostředí, ve kterém se agent pohybuje, je zde definováno přidělování odměn agentovi a návrh trasy. Agent je definován ve skriptu `Learn.py`. Schéma aplikace si lze prohlédnout na obrázku 16.

Obrázek 16: Rozvržení aplikace



Zdroj: autor

4.3 Algoritmus

4.3.1 Obecný popis algoritmu

Algoritmus učení agenta probíhá inkrementálně v krocích (viz obrázek 17), přičemž počet kroků je explicitně stanoven uživatelem. Obecně u posilovaného učení platí, že pro co nejlepší výsledky by počet kroků měl být co největší.(Raffin et al., 2021a)

V každém kroku (viz obrázek 18) agent vybere akci a z množiny všech akcí A a na základě současného stavu s a vybrané akce obdrží odměnu dle odměnové funkce popsané v části 4.1.1. Zároveň se určí následující stav, do kterého se agent volbou akce dostane a tento stav se přidá do trasy.

V případě, že trasa obsahuje všechny uzly, které měly být obslouženy, je epizoda ukončena a spočítá se výsledná nejkratší trasa epizody. Agent totiž v rámci učení - zejména zpočátku, když ještě nemá mnoho naučených zkušeností - prochází síť po zbytečných redundantních smyčkách nebo zbytečně „pendluje“ mezi uzly. Buď například neorientovaný graf

$$G = (\{A, B, C, D\}, \{(A, B), (A, C), (B, C), (C, D)\})$$

kde všechny hrany mají stejnou váhu, A je výchozí uzel a D je uzel, jež má být obsloužen. Potom trasa $A-B-A-C-B-C-D$ je co do dosažení bodu D de facto identická trase $A - C - D$, s tím rozdílem, že na první trase agent několikrát zbytečně „přeskakoval“ mezi body A, B a C než se dostal do D. Aplikace tedy trasu, kterou agent v rámci učení prošel, očistí o redundantní kroky.

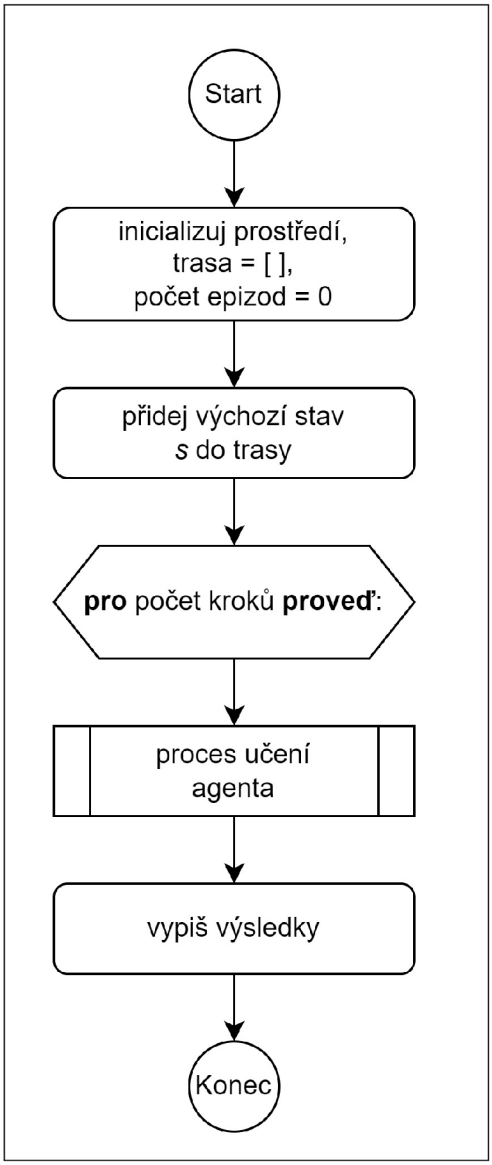
4.3.2 Použité metody posilovaného učení

Aplikace používá knihovnu Stable Baselines3, která poskytuje vzor pro tvorbu prostředí kompatibilních s Open AI Gym a zároveň obsahuje i několik implementovaných algoritmů posilovaného učení. Na návrh trasy v prostředí výše popsané silniční sítě, lze aplikovat následující algoritmy z knihovny:

- **A2C** - synchronní varianta Advantage Actor Critic algoritmu.
- **DQN** - hluboké Q-učení s pamětí zkušeností, cílovou Q sítí a ořezáváním gradientu (anglicky *gradient clipping*) pro zlepšení procesu zpětné propagace a nastavování vah sítě.

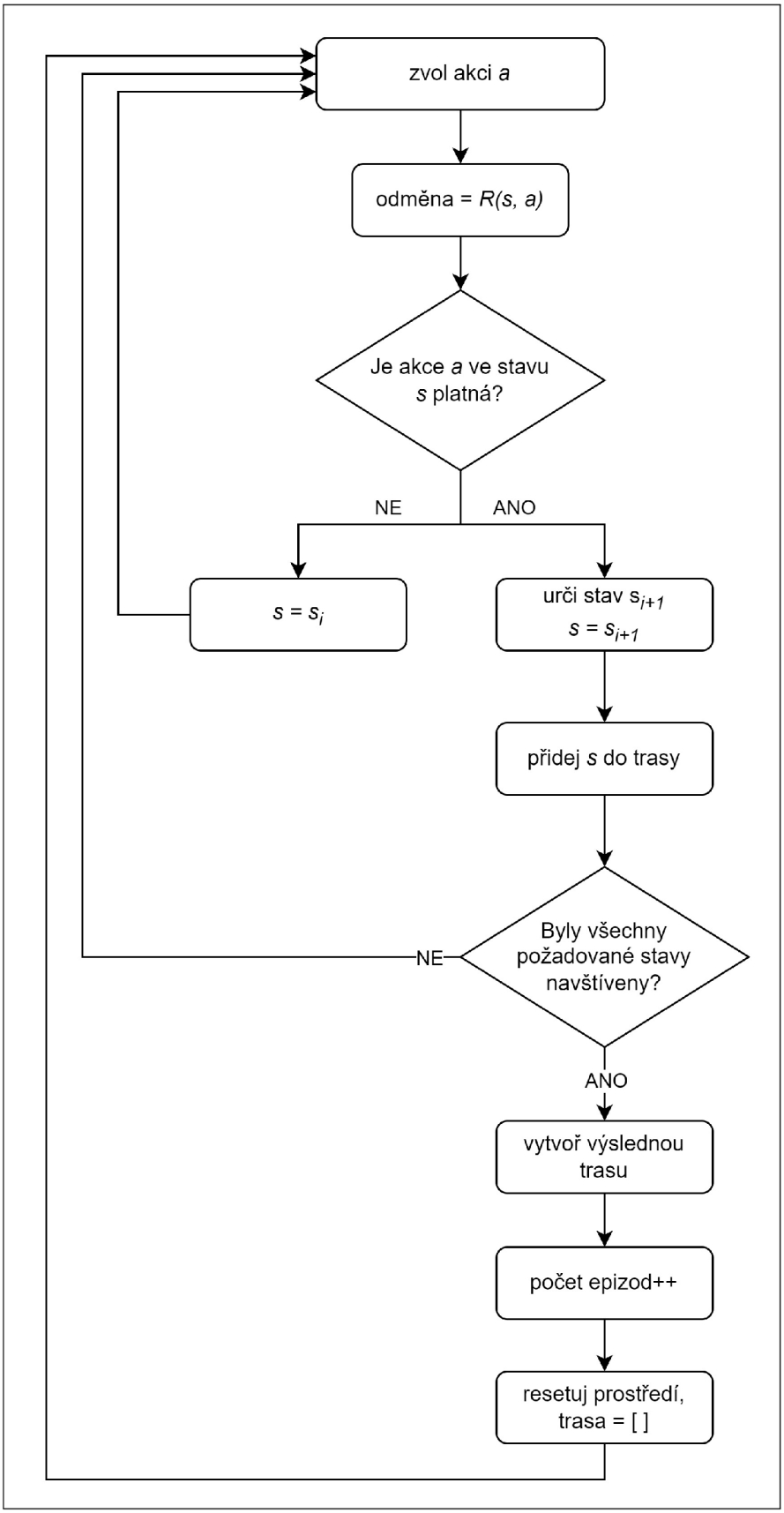
- **PPO** - algoritmus popsaný v teoretické části.
- **Maskable PPO** - stejný PPO algoritmus jako v předchozím bodě, avšak při učení jsou „odfiltrovány“ akce, které v daném stavu nelze provést. To je realizováno maskou, tj. k vektoru $(a_1, a_2, ..., a_n)$ obsahujícímu všechny akce je v každém stavu s přiřazena maska $(v_1, v_2, ..., v_n)$ kde $v_i \in \{0, 1\}$ vyjadřuje, zda i-tou akci lze ve stavu s provést.

Obrázek 17: Zjednodušený algoritmus inkrementálního učení



Zdroj: autor

Obrázek 18: Proces učení agenta



Zdroj: autor

5 Implementace v Pythonu

V této kapitole je detailně popsána implementace aplikace v programovacím jazyce Python. Pro vývoj aplikace byly použity následující knihovny:

- **Stable Baselines3** - prostředí a algoritmy pro posilované učení
- **networkx** - práce s grafy
- **numpy** - práce s poli a maticemi
- **pandas** - práce s daty a datovými soubory

5.1 Reprezentace silniční sítě

Vlastní rešerší bylo zjištěno, že Stable Baselines3 neobsahuje žádné prostředí, kterým by se dala grafově reprezentovat silniční síť. Proto byla v prvním kroce vytvořena pomocná třída `CGraph`, která slouží jako jakýsi „grafový backend“ pro prostředí `GraphEnv`.

Jako parametry pro konstruktor se zadává cesta k `csv` souboru obsahujícímu popis grafu a seznam uzlů, které má agent na trase navštívit. Funkcí `__CreateGraph()` se vytvoří `networkx` graf a dle něj prostor akcí a prostor stavů. Pro účely normalizace vah je ještě potřeba získat maximální a minimální váhy hran z grafu.

Kód 1: Konstruktor CGraph

```
1 class CGraph:
2     def __init__(self, graphCsv : str, nodesOnPath : list):
3         self.graph = self.__CreateGraph(graphCsv)
4 
5         self.__actionSpace = list(self.graph.edges)
6         self.__observationSpace = list(self.graph.nodes)
7         self.__minWeight,self.__maxWeight=self.GetMinMaxWeight()
```

Zdroj: autor

Vstupní `csv` soubor obsahuje všechny hrany grafu ve formátu

`startovní uzel hrany, koncový uzel hrany, váha hrany`

V rámci funkce `__CreateGraph()` se nejdříve vytvoří prázdný graf a poté se pomocí příslušné metody načtou data z csv do datového rámce. Všechny hrany jsou pak jedna po jedné přidány do grafu. Uzly není nutno explicitně vytvářet, jelikož jsou vytvořeny automaticky spolu s hranami.

Kód 2: Vytvoření grafu

```
8     def __CreateGraph(self, graphCsv : str):
9         graph = nx.Graph()
10
11        rawData = pd.read_csv(graphCsv)
12        edges = [tuple(edge) for edge in rawData.to_numpy()]
13
14        for edgeData in edges:
15            nodeStart = edgeData[0]
16            nodeEnd = edgeData[1]
17            weight = edgeData[2]
18
19            graph.add_edge(nodeStart, nodeEnd, weight=weight)
20        return graph
```

Zdroj: autor

Prostor akcí a prostor stavů byly popsány v části 4.1. Jelikož prostředí `GraphEnv` je nutno vytvořit dle vzoru prostředí Open AI Gym, prostor akcí a prostor stavů budou v `GraphEnv` uloženy jako pole. Třída `CGraph` tedy buď musí vracet velikosti prostoru akcí a prostoru stavů, nebo vrátí přímo tyto prostory. Pro účely kopírování prostorů akcí a prostoru stavů v případě, že by `CGraph` byl použit i pro jiné účely, než jen jako podklad pro Open AI Gym prostředí, vrací třída přímo tyto dva prostory.

Kód 3: Získání prostoru akcí a stavů

```
21    def GetActionSpace(self):
22        return self.__actionSpace
23
24    def GetObservationSpace(self):^^I
25        return self.__observationSpace
```

Zdroj: autor

K hranám a uzlům bude často přistupováno, tudíž jsou nutné metody pro získání uzlu z prostoru stavů a pro získání pozice uzlu v tomto prostoru. Zdaleka nejčastěji bude přistupováno k hranám, jelikož ty jsou volené agentem. Funkce `GetEdgeByPosition()` vrací uzly, kterými je hrana definována v `csv` souboru, a data hrany.

Kód 4: Získání uzlů a hran

```
26      def GetNodePosition(self, node) -> int:
27          return self.__observationSpace.index(node)
28
29      def GetNodeByPosition(self, nodePos : int):
30          return self.__observationSpace[nodePos]
31
32      def GetEdgeByPosition(self, edgePos : int):
33          edge  = self.__actionSpace[edgePos]
34          return edge[0], edge[1], self.graph[edge[0]][edge[1]]
```

Zdroj: autor

Velmi důležitou funkcí je detekování validity akce (hrany) v daném stavu (uzlu). Hrana je určena dvěma uzly. Pokud se ani jeden z těchto uzlů nerovná uzlu, ve kterém se agent v současnosti nachází, akce není platná.

Kód 5: Ověření validity akce

```
35      def IsValidAction(self, node, startNode, endNode):
36          if not self.graph.has_edge(startNode,endNode):
37              isValid = False
38          else:
39              isValid = (node == startNode or node == endNode)
40          return isValid
```

Zdroj: autor

Další významnou funkcí je určení dalšího uzlu podle současného uzlu, v němž se agent nachází a podle agentem zvolené hrany. Tato funkce neošetřuje situaci, že by hrana ze současného uzlu nevedla. Předpokládá, že vstupem funkce jsou jen validní kombinace uzlů a jejich hran. Detekce validity je popsána výše.

Kód 6: Získání dalšího uzlu

```
41      def GetNextNode(self, node, startNode, endNode):
42          if node == startNode:
43              nextNode = endNode
44          else:
45              nextNode = startNode
46          return nextNode
```

Zdroj: autor

Posledními potřebnými funkcemi jsou funkce pro normalizaci vah. Nejmenší a největší váhy jsou uloženy do proměnných třídy a výpočet normalizované váhy pak již probíhá dle vzorce 22.

Kód 7: Funkce pro normalizaci vah hran

```
47      def GetMinMaxWeight(self):
48          weights = [
49              self.graph[e[0]][e[1]]["weight"] for e
50              in self.graph.edges
51              ]
52          return min(weights), max(weights)
53
54      def GetNormalizedWeight(self, weight):
55          return (weight - self.__minWeight)
56          /(self.__maxWeight - self.__minWeight)
```

Zdroj: autor

5.2 Prostředí agenta

Stable Baselines3 prostředí je nutné vytvářet v souladu s unifikovanou šablonou, viz kód 8. Metody, které je nutno definovat, aby prostředí fungovalo, jsou tyto:

`__init__()`	konstruktor, vytváří celé prostředí.
`step()`	logika agentova kroku v prostředí. V případě aplikace vyvíjené v této diplomové práci tedy `step()` odpovídá tomu, co se stane, když agent zvolí silnici mezi dvěma uzly, po které chce dále jet. Metoda vrací stav prostředí (`observation`) po vykonání kroku, odměnu agenta za vykonání kroku (`reward`), indikaci, zda agent úspěšně dosáhl cíle (`done`) a případné další informace pro agenta (`info`).
`reset()`	resetuje prostředí

Kód 8: Šablona pro Stable Baselines3-kompatibilní prostředí

```
1  import gym
2  import numpy as np
3  from gym import spaces
4
5  class CustomEnv(gym.Env):
6      metadata = {"render.modes": ["human"]}
7      def __init__(self, arg1, arg2, ...):
8          super().__init__()
9          self.action_space = ...
10         self.observation_space = ...
11
12     def step(self, action):
13         ...
14         return observation, reward, done, info
15
16     def reset(self):
17         ...
18         return observation
19
20     def render(self, mode="human"):
21         ...
22
23     def close(self):
24         ...
```

Zdroj: (Raffin et al., 2021b)

5.2.1 Konstruktor

Při vytvoření prostředí se nastaví jeho parametry, čili zejména prostor akcí, prostor stavů, obsluhované uzly, kapacity a výchozí uzel. Dále jsou definovány výchozí hodnoty pomocných proměnných.

Kód 9: Konstruktor prostředí

```
7  def __init__(self, graphCsv, nodesOnPath, nodesCapacities,
8               startCapacity = 0, maxCapacity = np.inf):
9      super().__init__()
10     self.graph = CGraph(graphCsv,nodesOnPath)
11
12     self.action_space_size = len(self.graph.GetActionSpace())
13     self.observation_space_size = len(self.graph.GetObservationSpace())
14     self.action_space = spaces.Discrete(self.action_space_size)
15     self.observation_space=spaces.Discrete(self.observation_space_size)
16
17     self.nodeOrigin = nodesOnPath[0]
18     self.nodeCurrent = self.nodeOrigin
19     self.completePath = nodesOnPath
20
21     self.nodesCapacities = nodesCapacities
22     self.startCapacity = startCapacity
23     self.currentCapacity = self.startCapacity
24     self.maxCapacity = maxCapacity
25
26     self.invalidActionReward = -5
27     self.pathNodeReward = 5
28
29     self.finalPath = None
30     self.finalPathLength = 0
31     self.finalPathLengthMin = np.inf
32     self.finalPathLengthMax = -np.inf
33
34     self.logText = ""
35     self.stepCountAll = 0
36     self.rewardTotal = 0
37     self.episodes = 0
38     self.distanceEpisode = 0
39     self.distanceTotal = 0
40     self.distanceMinimal = np.inf
41     self.stepCountPerEpisode = 0
42     self.rewardPerEpisode = 0
43     self.episodePath = list()
```

Zdroj: autor

5.2.2 Krok učení

Metoda `step()` je zásadní pro učení agenta. Metoda má na vstupu akci z prostoru akcí, kterou agent zvolil strategií volby akce dle použité metody posilovaného učení. Prvním krokem je zjištění, zda zvolená akce je validní nebo nikoliv. Pokud není validní, zůstává agent ve stejném uzlu a obdrží odměnu dle odměnové funkce (viz 4.1.1).

Kód 10: Krok učení - neplatná akce

```
44 def step(self, action):
45      observation = None
46      reward = None
47      done = None
48      info = {}
49
50      self.stepCountAll += 1
51      self.stepCountPerEpisode += 1
52
53      nodeStart, nodeEnd, edgeData = self.graph.GetEdgeByPosition(action)
54      isValidAction = self.graph.IsValidAction(self.nodeCurrent,
55             nodeStart, nodeEnd)
56
57      if not isValidAction:
58           observation = self.graph.GetNodePosition(self.nodeCurrent)
59           reward = self.invalidActionReward
```

Zdroj: autor

Pokud je akce validní, určí se další uzel, který je dosažený volbou akce a získá se odměna dle odměnové funkce. Pokud má být další uzel obsloužen, je zvýšeno zaplnění agenta o kapacitu toho uzlu.

Kód 11: Krok učení - přiřazení odměny

```
60 else:
61      nextNode = self.graph.GetNextNode(self.nodeCurren,nodeStart,nodeEnd)
62
63      if (nextNode in self.episodePath):
64          reward = -self.graph.GetNormalizedWeight(edgeData["weight"])
65          observation = self.graph.GetNodePosition(nextNode)
66
67          self.nodeCurrent = nextNode
68          self.episodePath.append(nextNode)
69
70          self.distanceEpisode += edgeData["weight"]
71          self.distanceTotal += self.distanceEpisode
72      else:
```

```
73          if  (nextNode in self.completePath):
74              reward = self.pathNodeReward
75              self.currentCapacity += self.nodesCapacities[
76              self.completePath.index(nextNode)]
77          else:
78              reward = -self.graph.GetNormalizedWeight(edgeData["weight"])
```

Zdroj: autor

Po přiřazení odměny následuje ošetření zaplněnosti agenta. Pokud agent obsloužil maximální počet uzlů, vrací se zpět do výchozího uzlu, kde je jeho zaplnění vynulováno. Pro návrat do výchozího uzlu je použit Dijkstrův algoritmus z knihovny `networkx`. Odměna je snížena o délku cesty do výchozího bodu, což by agenta mělo motivovat nalézat takovou posloupnost prvních obsloužených uzlů, aby za návrat do výchozího uzlu byl co nejméně penalizován. Jinými slovy, aby nedosáhl maximální kapacity na vzdáleném okraji prostředí a neprodlužoval si zbytečně návratem do výchozího bodu celkovou trasu pro obsloužení všech uzlů.

Kód 12: Krok učení - dosažení maximální kapacity

```
79      if self.currentCapacity >= self.maxCapacity:
80
81           self.currentCapacity = 0
82           self.episodePath.append(nextNode)
83
84           returnPath = nx.shortest_path(self.graph.graph, nextNode,
85                                         self.nodeOrigin,
86                                         weight="weight")[1:]
87           nextNode = self.nodeOrigin
88           reward -= nx.path_weight(self.graph.graph,returnPath,
89                                    weight="weight")
90           self.distanceEpisode += reward
```

Zdroj: autor

Nyní je metoda `step()` téměř u konce, ale ještě je nutné ověřit zda úloha byla vyřešena a pokud ano, sestavit finální trasu a vygenerovat výstup.

Kód 13: Krok učení - ověření splnění úlohy, konec metody

```
91      done = True
92      for node in self.completePath:
93           if node not in self.episodePath:
94                done = False
95
96      self.nodeCurrent = nextNode
```

```
97        self.rewardTotal += reward
98        self.rewardPerEpisode += reward
99
100       self.distanceEpisode += edgeData["weight"]
101       self.distanceTotal += self.distanceEpisode
102
103       if done:
104            self.episodes += 1
105
106            self.BuildFinalPath()
107            if self.finalPathLength < self.finalPathLengthMin:
108                 self.finalPathLengthMin = self.finalPathLength
109             if (self.finalPathLength > self.finalPathLengthMax):
110                 self.finalPathLengthMax = self.finalPathLength
111
112            output = f"{self.episodes};
113                       {self.stepCountPerEpisode};
114                       {self.stepCountAll};
115                       {self.rewardPerEpisode};
116                       {self.rewardTotal};
117                       {self.distanceEpisode};
118                       {self.distanceTotal};
119                       {len(self.finalPath)-1};
120                       {self.finalPathLength};
121                       {self.finalPath};
122                       {len(self.episodePath)-1};
123                       {self.episodePath}\n"
124            self.logText += output
125
126       observation = self.graph.GetNodePosition(nextNode)
127
128  return observation, reward, done, info
```

Zdroj: autor

5.2.3 Sestavení finální trasy

V rámci každé epizody agent postupně pohybem prostředím navštíví všechny obsluhované uzly. Tato epizodní trasa obsahuje posloupnost všech obsluhovaných uzlů. Jak bylo naznačeno v podkapitole 4.3.1, celá trasa epizody není optimální vzhledem k posloupnosti obsluhovaných uzlů. Finální navržená trasa je proto určena nejkratšími trasami mezi obsluhovanými uzly a výchozím uzlem, ve kterém agent začíná a do kterého se po obsloužení všech patřičných uzlů vrací.

Na začátku obsahuje výsledná trasa pouze tzv. pivotní uzel, který je roven výchozímu uzlu. Algoritmus probíhá tak, že se prochází trasa epizody od pivotu postupně uzel po uzlu, dokud se nenarazí na první obsluhovaný uzel. Dijkstrovým algoritmem se určí

nejkratší trasa mezi pivotem a obsluhovaným uzlem, tato trasa se přidá k výsledné trase, obsluhovaný uzel je označen za běžný uzel (aby se do něj agent vícekrát nevracel) pivot je nastaven na obsluhovaný uzel a algoritmus pokračuje dál. Je-li dosaženo maximální kapacity, k výsledné trase se přidá nejkratší cesta do výchozího uzlu, kapacita agenta je vynulována a algoritmus pokračuje dál po jednotlivých uzlech trasy epizody.

Po navštívení všech obsloužených uzlů se k výsledné trase přidá nejkratší trasa návratu do výchozího uzlu. Tím je výsledná trasa hotova a už se pouze spočítá její celková délka (váha) algoritmem knihovny `networkx`.

Kód 14: Sestavení výsledné trasy

```
129     def BuildFinalPath(self):
130         path = []
131         targetNodes = self.completePath.copy()
132         pivot = self.completePath[0]
133         targetNodes.remove(pivot)
134         self.finalPath = [pivot]
135         capacity = 0
136 
137         for n in self.episodePath:
138             if (capacity >= self.maxCapacity):
139                 capacity = 0
140                 targetNodes.append(self.nodeOrigin)
141                 n = self.nodeOrigin
142             if n in targetNodes:
143                 targetNodes.remove(n)
144 
145                 path = nx.shortest_path(self.graph.graph, pivot,n,
146                                         weight="weight")
147                 self.finalPath += path[1:]
148 
149                 pivot = n
150                 nodeCapacity = self.nodesCapacities[
151                                     self.completePath.index(n) ]
152                 capacity += nodeCapacity
153 
154         # return to origin node & calculate final length
155         self.finalPath += nx.shortest_path(self.graph.graph,
156                               n, self.completePath[0], weight="weight")[1:]
157         self.finalPathLength = nx.path_weight(self.graph.graph,
158                                     self.finalPath, weight="weight")
```

Zdroj: autor

5.2.4 Reset prostředí

Metoda `reset()` je prostředím automaticky zavolána vždy po dokončení epizody, tj. když `done = True`. V této metodě se pouze nastaví příslušné parametry prostředí na výchozí hodnoty.

Kód 15: Resetování prostředí

```
159     def reset(self):
160         self.nodeCurrent = self.nodeOrigin
161         self.stepCountPerEpisode = 0
162         self.rewardPerEpisode = 0
163         self.episodePath = list()
164         self.distanceEpisode = 0
165         self.finalPathLength = 0
166         self.finalPath = []
167         self.currentCapacity = self.startCapacity
168 
169         self.episodePath.append(self.nodeCurrent)
```

Zdroj: autor

5.2.5 Validace akcí

Metoda PPO s maskováním akcí byla popsána v 4.3.2. Samotné vytvoření masky akcí je triviální - projdou se všechny akce z prostoru akcí, pro každou akci se zjistí její validita pomocí metody `IsValidAction` (viz vysvětlení u kódu 5) a takto je iterativně vyplněna celá maska akcí.

Kód 16: Vytvoření masky validit akcí

```
170     def action_mask(self) -> np.ndarray:
171         a_mask = list()
172         for action in range(self.action_space_size):
173             nodeStart,nodeEnd,edgeData = self.graph.GetEdgeByPosition(
174                                                 action )
175             isValidAction = self.graph.IsValidAction(self.nodeCurrent,
176                                 nodeStart, nodeEnd)
177             a_mask.append(isValidAction)
178 
179         return np.array(a_mask)
```

Zdroj: autor

5.3 Učící skript

Poslední částí je učící skript, ve kterém se definují konkrétní parametry agenta a prostředí. V případě, že bude používán algoritmus PPO s detekcí validity akcí, je potřeba nejdříve definovat funkci vracející masku akcí. V tomto případě funkce jednoduše volá metodu `action_mask()` prostředí `GraphEnv`, která byla popsána v předcházející podkapitole.

Následně je vytvořeno prostředí a agent. Agentovi je nutno určit prostředí, ve kterém se bude učit, a strategii pro volbu akcí. Tyto strategie závisí na použité metodě posilovaného učení, nicméně pro snažší zápis knihovna Stable Baselines3 poskytuje alias `MlpPolicy`, a není tedy nutné pro každou metodu uvádět plně kvalifikované jméno strategie. V kódu 17 jsou pro úplnost uvedeny příkazy pro vytvoření agenta s jednotlivými metodami uvedenými v 4.3.2, tj. hluboké učení, Actor Critic a (bez)masková Proximal Policy Optimization.

Kód 17: Vytvoření prostředí a agenta

```
import gym
import numpy as np
import networkx as nx
from stable_baselines3 import DQN
from stable_baselines3 import PPO
from stable_baselines3 import A2C
from sb3_contrib.common.maskable.policies import (
     MaskableActorCriticPolicy )
from sb3_contrib.common.wrappers import ActionMasker
from sb3_contrib.ppo_mask import MaskablePPO

# action masking function
def mask_fn(env: gym.Env) -> np.ndarray:
    return env.action_mask()

# create environment
env = envs.GraphEnv(
    "graph.csv", nodesOnPath = [1, 2, 3, 4, 5],
    nodesCapacities = [0, 1, 2, 0, 1], maxCapacity     = 3 )

# create agent (various methods shown)
agent = DQN( "MlpPolicy",  env) # Deep Q-learning agent
agent = A2C( "MlpPolicy",  env) # Actor Critic agent
agent = PPO( "MlpPolicy",  env) # Proximal Policy Optimization agent
agent = MaskablePPO("MlpPolicy",ActionMasker(env,mask_fn)) # masked PPO
```

Zdroj: autor

Učení agenta probíhá v rámci funkce `learn()` s parametrem `total_timesteps` určujícím, jak dlouho se agent bude učit, respektive kolik volání metody `step()` prostředí proběhne. Dále lze definovat další nepovinné argumenty, například `log_interval` určující interval logování proměnných agenta (př. `log_interval = 100` znamená, že proměnné budou logovány po každých 100 epizodách), nicméně lze použít i vlastní způsob logování.

Kód 18: Učení agenta

```
26 agent.learn(total_timesteps=1000, log_interval=100)
```

Zdroj: autor

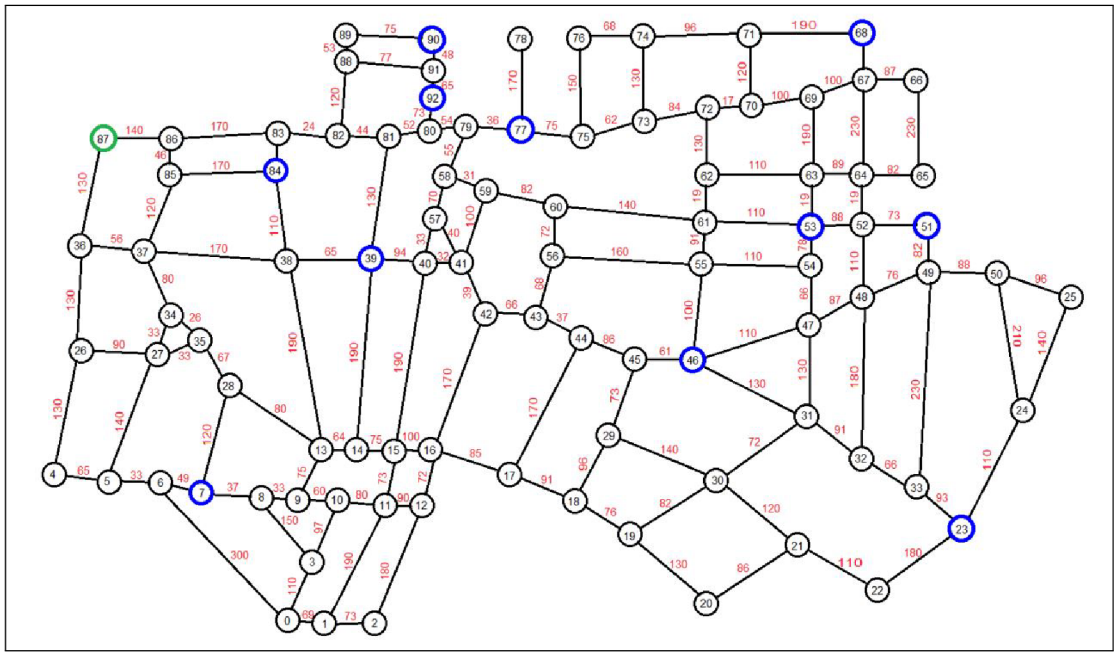
6 Příklad návrhu trasy

Aplikace bude použita k návrhu trasy pro svozové vozidlo svážející tříděný odpad ve čtvrti Suché Vrbné v Českých Budějovicích. Tento příklad je zvolen z následujících důvodů:

- jedná se o problém z praxe,
- dopravní síť čtvrti je přiměřeně rozsáhlá,
- problém svozu odpadu ve čtvrti již dříve řešil Vácha (2016) metodou rojové inteligence a lze tedy do jisté míry porovnat výsledky dosažené touto metodou a metodou z této diplomové práce

Na obrázku 19 je graf čtvrti Suché Vrbné, zelený uzel č. 87 je výchozí uzel, modré uzly jsou kontejnery, které musí svozové vozidlo všechny vyzvednout. Červeně jsou označeny délky jednotlivých silnic v metrech.

Obrázek 19: Grafová reprezentace dopravní sítě čtvrti Suché Vrbné



Zdroj: převzato z Vácha (2016)

6.1 Návrh trasy bez omezení kapacity vozidla

V tomto experimentu jsou použity všechny metody popsané v 4.3.2, tj. hluboké učení (DQN), Actor Critic (A2C), Proximal Policy optimalizace bez maskování i s maskováním (PPO, MPPO). Vozidlo mělo neomezenou kapacitu. Každá metoda byla spuštěna pro počet kroků (v tisících) 1, 10, 50, 100, 500 a 1000. Jelikož použité metody posilovaného učení nevrací vždy stejné řešení, byla aplikace pro každý počet kroků spuštěna 10krát. Celkem tedy bylo provedeno 240 běhů aplikace.

Není zajištěno, že použité metody vrátí optimální řešení, tj. v tomto příkladě tu nejkratší trasu. Lze předpokládat, že čím vyšší počet kroků bude, tím více se budou nalezená řešení přibližovat optimálnímu. Pro ověření jak blízko či daleko jsou nalezená řešení od optimálního, bylo optimální řešení pro uvažovanou dopravní síť vygenerováno hrubou silou. Optimální řešení je trasa

[87, 86, 83, 84, 83, 82, 19, 20, 21, 22, 23, 33, 32, 31, 46, 55, 54, 53, 52, 51, 52, 64, 67, 68, 71, 74, 76, 75, 77, 79, 80, 92, 91, 90, 91, 92, 80, 81, 39, 14, 13, 28, 7, 6, 5, 4, 26, 36, 87]

s délkou 4 355 metrů.

Výsledky jsou v tabulce 2. Je patrné, že pro nižší počty kroků vrací nějaká řešení pouze metoda MPPO. Celkově lze usoudit, že MPPO ve všech veličinách, s výjimkou směrodatné odchylky délek nalezených tras, vykazuje nejlepší výsledky a ostatní metody zcela dominuje co do počtu nalezených řešení - MPPO nalezl 99.5 % ze všech nalezených řešení celého experimentu. Tato dominance zjevně tkví v tom, že MPPO se poměrně dobře naučí nevolit nevalidní akce, zatímco ostatní tři metody jsou v tomto ohledu mnohem omezenější. V jednom případě dokonce MPPO nalezl optimální řešení a už pro 50 tisíc kroků se nejlepší nalezené řešení odlišovalo od optimálního pouze o zhruba 2 %, zatímco ostatní metody se ve většině případů takto nízkému rozdílu nepřiblížily.

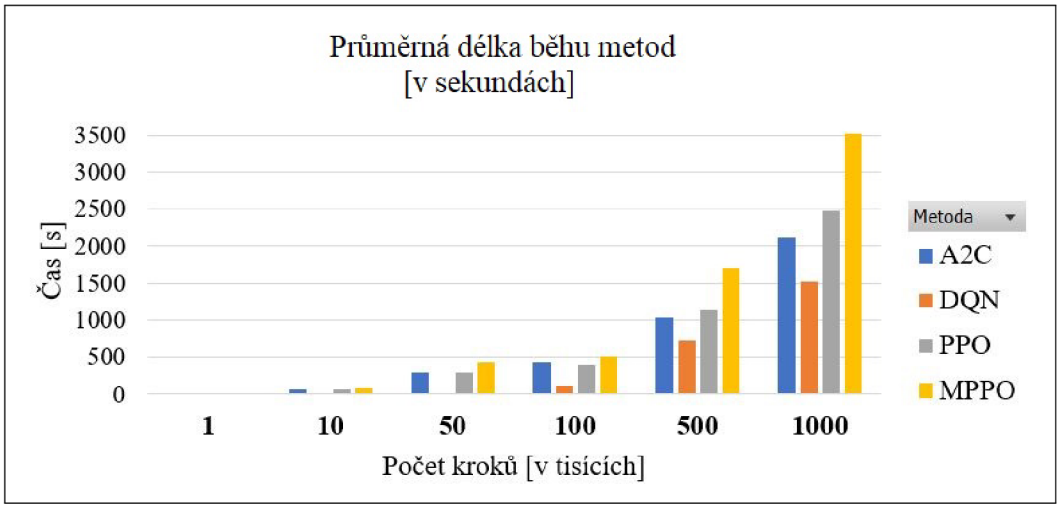
Avšak z hlediska času je MPPO vždy nejpomalejší, nicméně narozdíl od ostatních metod dává dobré výsledky ve vyšším počtu již při nižších počtech kroků. Srovnání časů metod je na obrázku 20.

Tabulka 2: Výsledky jednotlivých metod v případě neomezené kapacity

Metoda	Počet kroků [tis.]	Počet nalezených řešení	Min. trasa [m]	Max. trasa [m]	Průměr. trasa [m]	Směrodat. odchylka [m]	Min. trasa – optimum [%]
A2C	1	0	-	-	-	-	-
	10	0	-	-	-	-	-
	50	21	5009	6279	5744	586	15.02
	100	37	4941	6088	5877	593	13.46
	500	126	4510	5437	5968	616	3.56
	1000	175	4709	5403	6065	662	8.13
DQN	1	0	-	-	-	-	-
	10	0	-	-	-	-	-
	50	19	5134	6718	5991	491	17.89
	100	21	4921	6251	5517	429	13.00
	500	23	4626	6895	5871	502	6.22
	1000	34	4792	5742	5805	703	10.03
PPO	1	0	-	-	-	-	-
	10	0	-	-	-	-	-
	50	6	5872	6525	6214	267	34.83
	100	25	4792	6613	5771	781	10.03
	500	21	4980	6915	6152	590	14.35
	1000	20	5211	6939	6107	577	19.66
MPPO	1	27	5082	7367	5956	699	16.69
	10	104	4903	5570	6063	652	12.58
	50	620	4449	4967	5843	589	2.16
	100	1791	4449	4825	5889	636	2.16
	500	34586	4437	4744	5507	678	1.88
	1000	81605	4355	4722	5357	618	0.00

Zdroj: autor

Obrázek 20: Srovnání časové náročnosti metod



Zdroj: autor

Poznámka: Aplikace byla spouštěna v konfiguraci pro CPU. Krátce byl otestován i běh na GPU s podporou CUDA, ale to se ukázalo být pomalejší než běh na CPU. Aplikace byla spouštěna v prostředí s touto konfigurací: CPU Intel Core i5-10300H, 2.5 GHz, 4 jádra / 8 vláken; NVIDIA GeForce GTX 1660 Ti, 6 GB; CUDA 12.1; Python 3.9.9; Windows 10 Home.

6.2 Návrh trasy s omezením kapacity vozidla

V následujících experimentech je uvažována maximální kapacita vozidla rovna 7 jednotkám a každý kontejner má kapacitu 1. Vzhledem k výsledkům z předcházejícího experimentu a k poměru řešení MPPO vůči ostatním metodám je v následujících experimentech použita pouze metoda MPPO.

6.2.1 Bez návratu na původní trasu

V tomto experimentu se vozidlo po zaplnění kapacity vrátí do výchozího uzlu, vyloží náklad (kapacita je resetována na 0) a z výchozího uzlu se následně vydává vyzvednou zbývající kontejnery. Vozidlo se tedy nevrací do uzlu, kde došlo k zaplnění.

Výsledky jsou v tabulce 3. Rozdíly oproti případu bez omezení kapacity vozidla jsou markantní, zejména počet nalezených řešení je řádově nižší a průměrná trasa je zhruba o 2-3 km delší.

Tabulka 3: Výsledky v případě omezené kapacity bez návratu na původní trasu

Metoda	Počet kroků [tis.]	Počet nalezených řešení	Min. trasa [m]	Max. trasa [m]	Průměr. trasa [m]	Směrodat. odchylka [m]
MPPO	1	5	6 985	8409	7662	651
	10	19	6892	10270	8195	931
	50	15	7458	8525	8008	456
	100	48	6507	8457	8119	671
	500	54	7023	8668	8228	694
	1000	55	6310	8309	8118	855

Zdroj: autor

6.2.2 S návratem na původní trasu

V tomto experimentu se vozidlo po zaplnění kapacity vrátí do výchozího uzlu, vyloží náklad (kapacita je resetována na 0), vrátí se nejkratší trasou k poslednímu vyzvednutému kontejneru a pokračuje ve sběru zbývajících kontejnerů. Výsledky jsou v tabulce 4. Je vidět, že počty nalezených řešení jsou blíže variantě bez omezení

kapacity. Průměrné délky nalezených tras se od délek průměrných tras varianty bez návratu (viz tabulka 3) odlišují o 1 - 1.5 km, což je rozdíl zhruba o 50 % menší než u varianty bez návratu.

Tabulka 4: Výsledky v případě omezené kapacity s návratem na původní trasu

Metoda	Počet kroků [tis.]	Počet nalezených řešení	Min. trasa [m]	Max. trasa [m]	Průměr. trasa [m]	Směrodat. odchylka [m]
MPPO	1	18	6029	7733	7094	591
	10	110	5554	6056	7017	714
	50	720	5538	6088	6958	663
	100	1590	5366	5715	6956	711
	500	29135	5361	5667	6521	779
	1000	56316	5342	5608	6776	593

Zdroj: autor

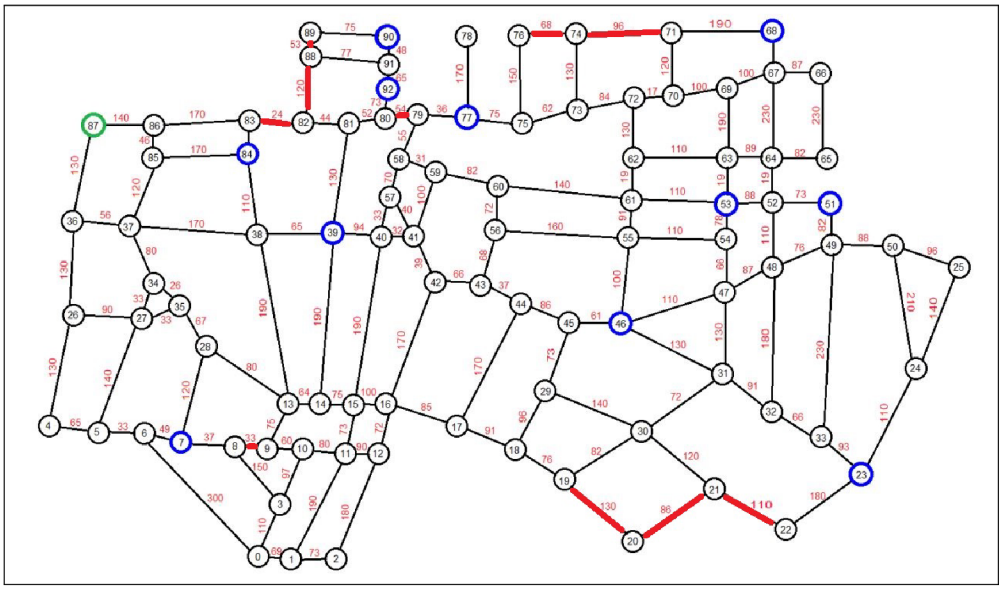
6.3 Návrh trasy při výpadku spoje

V tomto experimentu se bude zkoumat vliv výpadku vybraného spoje na agentovo učení. Analýzou epizodních tras získaných při navrhování trasy bez omezení kapacity bylo zjištěno, že z množiny silnic neústících přímo do uzlů s kontejnery agent nejčastěji volil hrany $82-88$, $88-89$, $79-80$, $82-83$, $71-74$, $21-22$, $19-20$, $20-21$, $74-76$ a $8-9$. Tyto hrany jsou vyznačeny červeně na obrázku 21.

Experiment proběhne tak, že po polovině proběhlých kroků učení bude vybraná hrana označena za nevalidní. K odstranění byla zvolena hrana $79-80$, protože představuje krátký spoj mezi levou a pravou částí sítě. Navíc je tato hrana blízko třem uzlům s kontejnery a agent po odstranění hrany bude muset využít delší objížďku. To by se mělo projevit zvýšením průměrné délky výsledné navržené trasy.

V praktickém kontextu lze experiment připodobnit k řidiči, který musí svůj „zažitý“ způsob svozu odpadu upravit v případě uzavření silnice.

Obrázek 21: Nejčastěji volené hrany



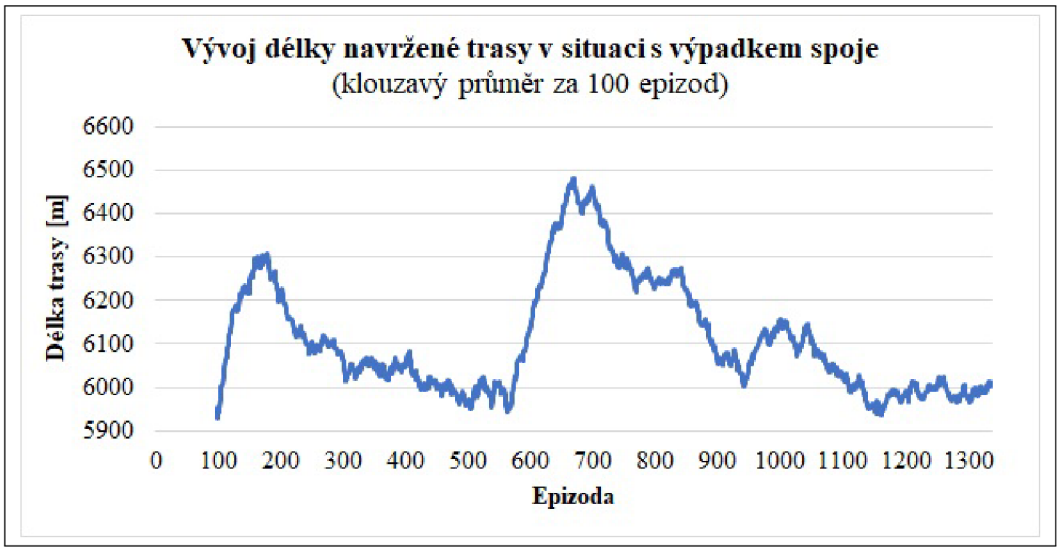
Zdroj: autor

Aplikace byla spuštěna s počtem kroků 500 tisíc. Před výpadkem hrany se uskutečnilo 567 epizod s průměrnou délkou navrhnuté trasy 6 036 metrů, po výpadku 771 epizod s průměrnou délkou 6135 metrů. Na obrázku 22 je po 567. epizodě vidět významný nárůst délky trasy, přičemž předvýpadkových délek se opět začne dosahovat až za zhruba 500 epizod.

Pokus byl opakován s počtem kroků 1 milion. Před výpadkem se uskutečnilo 3148 epizod s průměrnou délkou navrhované trasy 5378 metrů, po výpadku 1805 epizod s průměrnou délkou trasy 6037 metrů.

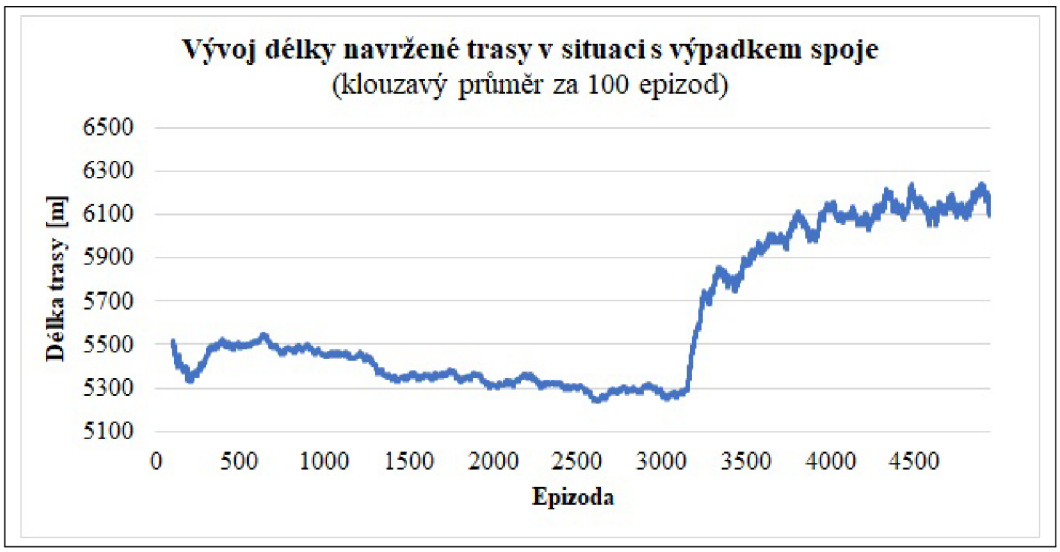
Na obrázku 23 lze opět pozorovat strmý nárůst délky trasy po výpadku, avšak narozdíl od předchozího běhu tentokrát během zbývajících epizod zůstaly délky tras oproti předvýpadkovým hodnotám výrazně výše. Možným vysvětlením pro tuto skutečnost je, že jelikož mnohem více epizod proběhlo před výpadkem, agent se velmi dobře naučil volit trasu v původní dopravní síti a trvá mu proto déle adaptovat se na změněnou síť po výpadku. Nicméně v posledních zhruba tisíci epizodách se tempo zvyšování navrhovaných délek tras snižuje a zřejmě by později začaly délky opět klesat.

Obrázek 22: Vývoj délky navržené trasy před a po výpadku spoje (500 tisíc kroků)



Zdroj: autor

Obrázek 23: Vývoj délky navržené trasy před a po výpadku spoje (1 milion kroků)



Zdroj: autor

6.4 Srovnání s metodou rojové inteligence

Srovnání průměrných výsledků jednotlivých metod posilovaného učení (PU) a výsledků metody rojové inteligence (RI) aplikované v (Vácha, 2016) je v tabulce 5. Jsou srovnávány případ bez omezení kapacity. Je nutné poznamenat, že výsledky uvedené u metody RI byly získány pouze ze dvou běhů algoritmu, zatímco výsledky uvedené u PU jsou pro každou jednu metodu zprůměrované z celkem 60 běhů pro 6 různých počtů kroků.

RI najde větší počet řešení než DQN a PPO, avšak je dominována metodou MPPO.
S výjimkou PPO jsou metody PU lepší v nalézání nejkratší trasy, avšak nejdelší nalezené trasy jsou více rozptýlené (A2C a MPPO vs. DQN a PPO). Srovnatelných výsledků obě metody dosahují v průměrné délce navrhované trasy, avšak RI má menší variabilitu výsledků než PU.

Jelikož není veřejně dostupný kód metody RI, nelze metody porovnat z hlediska časové náročnosti.

Tabulka 5: Srovnání výsledků posilovaného učení a rojové inteligence

Metoda	Varianta	Počet řešení	Min. trasa [m]	Max. trasa [m]	Průměrná trasa [m]	Směrodat. odchylka [m]
Rojová inteligence	10 tis. iterací	20	5437	6806	6065	384
	20 tis. iterací	10	5437	6893	5976	490
	Nejlepší	20	5437	6806	5976	384
Posilované učení	A2C	13	5102	7080	6002	642
	DQN	2	5471	6114	5798	598
	PPO	1.8	5708	6311	6010	668
	MPPO	7898	4530	7782	5412	643

Zdroj: vlastní výsledky a výsledky převzaté z (Vácha, 2016)

Závěr

Tato diplomová práce se zabývala přístupem posilovaného učení pro návrh trasy agentovi ve zjednodušeném scénáři pohybu v dopravní síti. Navrhování tras je podstatné téma z mnoha důvodů a vzhledem k technologickým trendům je zřejmé, že nejen v oblasti dopravy či logistiky budou mít přístupy založené na umělé inteligenci čím dál významnější roli.

V teoretické části byly představeny základy umělé inteligence, posilovaného učení a blíže popsány vybrané klasické i moderní metody posilovaného učení. Dále byla velmi stručně uvedena základní teorie týkající se simulování dopravy, včetně vybraných v praxi používaných nástrojů.

V rámci praktické části byla navrhnuta a vyvinuta konzolová aplikace umožňující jednoduché použití několika state-of-the-art metod posilovaného učení. Metody byly aplikovány na úlohu svozu odpadu v konkrétní čtvrti Českých Budějovic a porovnány s metodou řešící tuto úlohu pomocí rojové inteligence. Použité metody posilovaného učení poskytly výsledky srovnatelné s rojovou inteligencí a v jednom případu dokonce bylo dosaženo optimálního řešení. Celkově nejlepší výsledky poskytovala metoda Proximal Policy Optimization s detekcí validity akcí. Tato metoda však vyžadovala nejvíce času.

Na tuto práci by mohlo navazovat porovnání vybraných metod v různorodých scénářích provozu, dále například úprava aplikace tak, aby umožňovala i multiagentní přístup, přidání GUI anebo modifikace metod tak, aby byly škálovatelné na větší prostory stavů a akcí. Zajímavé by bylo i srovnání návrhů tras učiněných lidmi s návrhy učiněnými umělou inteligencí.

Summary and keywords

This thesis deals with reinforcement learning approach for designing a route for an agent in a simplified traffic network scenario.

In the theoretical part, the fundamentals of artificial intelligence, reinforcement learning and some methods of reinforcement learning are introduced. Basic theory related to traffic simulation is also briefly mentioned.

In the pracital part of the thesis, a console application utilizing several state-of-the-art reinforcement learning methods is designed and developed. The methods were applied to the task of waste collection in a selected district of České Budějovice and compared to a method solving this task using swarm intelligence. The results achieved by the reinforcement learning are similar to the results obtained by swarm intelligence, with Proximal Policy Optimization with action masking being the most successful method overall. In one case, an optimal solution was found.

Keywords: reinforcement learning, machine learning, artificial intelligence, traffic, route selection

Seznam použité literatury

Alphabet Inc. (2018). Open-sourcing DeepMind Lab [Citováno 10.02.2023]. Dostupné z https://www.deepmind.com/blog/open-sourcing-deepmind-lab.

Barceló, J. (2010). *Fundamentals of Traffic Simulation*. Springer.

Bick, D. (2021). *Towards delivering a coherent self-contained explanation of proximal policy optimization* (Master's thesis). University of Groningen.

Bongsang, K. (2018). OpenAI Gym Environment Full List [Citováno 10.02.2023]. Dostupné z https://medium.com/@researchplex/openai-gym-environment-full-list-8b2e8ac4c1f7.

Brown, B., & Zai, A. (2020). *Deep Reinforcement Learning in Action*. Manning Publications Co.

Chalupník, V. (2012). Biologické algoritmy (5) - Neuronové sítě [Citováno 28.01.2023]. Dostupné z https://www.root.cz/clanky/biologicke-algoritmy-5-neuronove-site/.

Eclipse Foundation. (n.d.). Simulation of Urban MObility [Citováno 10.02.2023]. Dostupné z https://www.eclipse.org/sumo/.

Elefteriadou, L. (2014). *An Introduction to Traffic Flow Theory*. Springer.

Géron, A. (2017). *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media.

Gym Documentation [Citováno 10.02.2023]. (2023). Dostupné z https://www.gymlibrary.dev/.

Gym Documentation - Atari [Citováno 10.02.2023]. (2023). Dostupné z https://www.gymlibrary.dev/environments/atari/.

Gym Documentation - MuJoCo [Citováno 10.02.2023]. (2023). Dostupné z https://www.gymlibrary.dev/environments/mujoco/.

Heeswijk, W. (2022a). Policy Gradients In Reinforcement Learning Explained [Citováno 7.03.2023]. Dostupné z https://towardsdatascience.com/policy-gradients-in-reinforcement-learning-explained-ecec7df94245.

Heeswijk, W. (2022b). Proximal Policy Optimization (PPO) Explained [Citováno 7.03.2023]. Dostupné z https://towardsdatascience.com/proximal-policy-optimization-ppo-explained-abed1952457b.

Hendl, J. (2021). *Big Data: Věda o datech - základy a aplikace*. Grada.

Lapan, M. (2018). *Deep Reinforcement Learning Hands-On*. Packt Publishing.

Li, X., & Soh, L. (2004). *Applications of Decision and Utility Theory in Multi-Agent Systems* (CSE Technical reports No. 85). Dostupné z https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1092&context=csetechreports.

Melcher, K. (2021). A Friendly Introduction to [Deep] Neural Networks [Citováno 28.01.2023]. Dostupné z https://www.knime.com/blog/a-friendly-introduction-to-deep-neural-networks.

Microsoft Corporation. (n.d.). Project Malmo [Citováno 10.02.2023]. Dostupné z https://www.microsoft.com/en-us/research/project/project-malmo/.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning [Citováno 28.01.2023]. Dostupné z https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

Müller, A. C., & Guido, S. (2016). *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media.

Norvig, P., & Russel, S. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

Pilát, M. (n.d.). Zpětnovazební učení [Citováno 14.01.2023]. Dostupné z https://martinpilat.com/cs/prirodou-inspirovane-algoritmy/zpetnovazebni-uceni.

PTV Group. (n.d.). PTV Vissim [Citováno 10.02.2023]. Dostupné z https://www.ptvgroup.com/en/solutionsproducts/ptv-vissim/.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021a). Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, *22*(268), 1–8. http://jmlr.org/papers/v22/20-1364.html

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., & Dormann, N. (2021b). Using Custom Environments [Citováno 06.03.2023]. Dostupné z https://stable-baselines3.readthedocs.io/en/master/guide/custom_env.html.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. https://doi.org/10.48550/ARXIV.1707.06347

Sewak, M. (2019). *Deep Reinforcement Learning*. Springer.

Simonini, T. (2022a). Advantage Actor Critic (A2C) [Citováno 08.03.2023]. Dostupné z https://huggingface.co/blog/deep-rl-a2c.

Simonini, T. (2022b). Proximal Policy Optimization (PPO) [Citováno 07.03.2023]. Dostupné z https://huggingface.co/blog/deep-rl-ppo.

Sutton, R., & Barto, A. (2020). *Reinforcement Learning: An Introduction* (2nd ed.). The MIT Press.

Tensorforce Team. (n.d.). Tensorforce: a TensorFlow library for applied reinforcement learning [Citováno 10.02.2023]. Dostupné z https://tensorforce.readthedocs.io/en/latest/.

The AnyLogic Company. (n.d.). AnyLogic [Citováno 10.02.2023]. Dostupné z https://www.anylogic.com/.

The MathWorks, Inc. (n.d.). Reinforcement Learning Toolbox [Citováno 10.02.2023]. Dostupné z https://www.mathworks.com/products/reinforcement-learning.html.

The Ray Team. (n.d.). RLlib: Industry-Grade Reinforcement Learning [Citováno 10.02.2023]. Dostupné z https://docs.ray.io/en/latest/rllib/index.html.

Toth, P., & Vigo, D. (Eds.). (2014). *Vehicle Routing Problems, Methods, and Applications*. Society for Industrial; Applied Mathematics.

Vácha, L. (2016). *Řešení optimální cesty svozu odpadů pomocí rojové inteligence* (Master's thesis). Jihočeská univerzita v Českých Budějovicích.

Vlassis, N. (2007). *Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence*. Morgan & Claypool Publishers. Dostupné z https://jmvidal.cse.sc.edu/library/vlassis07a.pdf.

Watkins, C., & Dayan, P. (1992). Q-learning. *Machine Learning*, *8*, 279–292.

Zverovich, V. (2021). *Modern Applications of Graph Theory*. Oxford University Press.

Seznam obrázků

Seznam tabulek

Seznam ukázek zdrojových kódů