FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Implementace Q* algoritmu v Julia
Student:	Jiří Klubal
Vedoucí:	Ing. Tomáš Kalvoda, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Umělá inteligence 2021
Katedra:	Katedra aplikované matematiky
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

1. Popište A* algoritmus a jeho využití při řešení problémů v oblasti umělé inteligence a využití strojového učení ke konstrukci heuristické funkce.
2. Seznamte se s jeho variantou, Q* algoritmem [1].
3. Vyberte vhodný problém řešitelný pomocí A* i Q* algoritmu.
4. Obě řešení implementujte v jazyce Julia. Porovnejte jejich výslednou efektivitu při řešení problému z bodu 3. a dále náročnost konstrukce heuristické funkce.

Literatura:

[1] Agostinelli et. al., A* Search Without Expansions: Learninig Heuristic Functions With Deep Q-Networks, 2023, https://arxiv.org/abs/2102.04518

Elektronicky schválil/a Ing. Magda Friedjungová, Ph.D. dne 17. ledna 2024 v Praze.

Bakalářská práce

IMPLEMENTACE Q* ALGORITMU V JULIA

Jiří Klubal

Fakulta informačních technologií
Katedra aplikované matematiky
Vedoucí: Ing. Tomáš Kalvoda, Ph.D.
16. května 2024

Odkaz na tuto práci: Klubal Jiří. *Implementace Q* algoritmu v Julia.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2024.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Seznam popisů algoritmů

Chtěl bych poděkovat svému vedoucímu Ing. Tomáši Kalvodovi, Ph.D. za vedení, rady a konzultace k tvorbě této bakalářské práce. Dále bych rád poděkoval své rodině a svým přátelům za neochvějnou podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2024

Abstrakt

V této bakalářské práci je představen algoritmus Q*, společně s algoritmem A*, na kterém je založen. Je vysvětlen systém automatického získávání heuristických funkcí pomocí metod posilovaného učení, konkrétně algoritmy Deep Q-learning, DAVI a navrhnutými variantami DQVI a naivní DQVI. Všechny algoritmy jsou implementovány v jazyce Julia. Za ukázkový problém je zvoleno řazení palačinek, pro který je navrhnuto virtuální rozhraní. Veškeré algoritmy i rozhraní jsou implementovány za účelem snadné upravitelnosti a znovupoužitelnosti i v jiných problémech či specifikacích, než je tato bakalářská práce.

Na problému o velikosti deseti palačinek jsou natrénované neuronové sítě k aproximování heuristické funkce pro algoritmus A* a Q*. Sítě jsou mezi sebou porovnané. Ukazuje se, že zatímco algoritmus DQVI se učí více než pětkrát rychleji než algoritmus DAVI nebo naivní DQVI, je po učení pro stejný počet iterací méně efektivní. Algoritmus Deep Q-learning dosahuje dobrých výsledků, často u něj ale dochází k divergenci. Algoritmy A* a Q* jsou následně pro vybrané heuristické funkce porovnány mezi sebou. V problému řešeném touto bakalářskou prací je algoritmus A* rychlejší a přesnější než algoritmus Q*, který ale využívá méně vyhodnocení neuronových sítí.

Klíčová slova Q*, A*, algoritmus hledání cesty, posilované učení, hluboké neuronové sítě, Julia

Abstract

In this bachelor thesis, the Q* algorithm is introduced, together with the A* algorithm on which it is based. A system for automatic creation of heuristic function using reinforcement learning methods is explained, specifically the Deep Q-learning, DAVI and its proposed variants DQVI and naive DQVI. All algorithms are implemented in the Julia language. The pancake sorting puzzle is chosen as a toy problem and a virtual interface is designed for it. All algorithms and interfaces are implemented for easy modifiability and reusability in problems and specifications outside of the scope of this thesis.

Neural networks are trained on a problem size of ten pancakes to approximate heuristic functions for the A* and Q* algorithms. The networks are compared to each other. It turns out that while the DQVI algorithm learns more than five times faster than the DAVI or naive DQVI algorithm, it is less efficient after learning for the same number of iterations. The Deep Q-learning algorithm achieves good results, but it often diverges. The A* and Q* algorithms are then compared with each other. In the problem researched in this thesis, the A* algorithm is faster and more accurate than the Q* algorithm, which however uses less neural network evaluation.

Keywords Q*, A*, pathfinding algorithm, reinforcement learning, deep neural networks, Julia

Seznam zkratek

BFS	Breafth First Search
DFS	Depth First Search
ReLU	Rectified Linear Unit
DAVI	Deep Approximate Value Iteration
PEA*	Partial Expansion A*
DHE	Deffered Heuristic Evaluation
BWAS	Batch Weighted A* Search
JIT	Just In Time
REPL	Read-Eval-Print Loop
DQVI	DAVI pro q-heuristiku

Úvod

Vývoj výpočetní techniky během dvacátého století přinesl možnost řešit dříve náročné problémy na počítačích. Jedním z častých požadavků bylo (a stále je) řešení problému hledání cesty stavovým prostorem, aneb plánování akcí, kterými se může systém dostat z počáteční konfigurace do cílového stavu.

Algoritmy prohledávání stavových prostorů lze rozřadit do dvou kategorií. První z nich se často označují za neinformované metody. Ty nemají žádné informace o stavovém prostoru, které si samy nezjistí. Mezi nejznámější neinformované algoritmy patří například BFS (*Breadth First Search* – prohledávání do šířky) nebo DFS (*Depth First Search* – prohledávání do hloubky).

Druhou kategorií algoritmů jsou informované metody. Ty využívají určité znalosti o stavovém prostoru – obvykle v podobě heuristické funkce. Ta nejčastěji představuje odhad vzdálenosti vybraného stavu od stavu koncového. Pravděpodobně nejrozšířenějším algoritmem z této skupiny je A* (poprvé prezentovaný již roku 1968), jehož variantou Q* (představený roku 2021) se bude zabývat i tato práce.

Pro aplikaci informovaných algoritmů v komplexních nebo abstraktních stavových prostorech není snadné vytvořit vhodnou heuristickou funkci. K jejímu automatickému získání pro libovolný problém lze využít technik strojového učení, v případě této práce se bude jednat o využití hlubokých neuronových sítí ve spojení s posilovaným učením.

Cíle

Cílem teoretické části této práce je popsat problematiku prohledávání stavového prostoru a to především pomocí algoritmu A*. Budou vysvětleny nutné základy v oboru strojového učení pro jeho zprovoznění v obecném případě. Dále se představí varianty algoritmu A* společně s jejich výhodami i nevýhodami. Důležitou částí bude popis algoritmu Q*, kterým se především zabývá tato práce. Pro tyto dva algoritmy bude představena problematika automatického získávání heuristických funkcí a ukázány konkrétní metody využívající technik

posilovaného strojového učení a hlubokých neuronových sítí. Dále bude představen programovací jazyk Julia, ve kterém bude implementována praktická část.

Praktická část má za cíl implementaci algoritmů A* a Q* společně s aparátem pro automatické získávání heuristických funkcí. Bude vytvořena univerzální reprezentace řešitelných problémů a na konkrétním případě budou provedeny experimenty. Ty porovnají časovou a paměťovou náročnost obou algoritmů a náročnost sestavování heuristické funkce.

Kapitola 1
Teoretický základ

1.1 Řazení palačinek

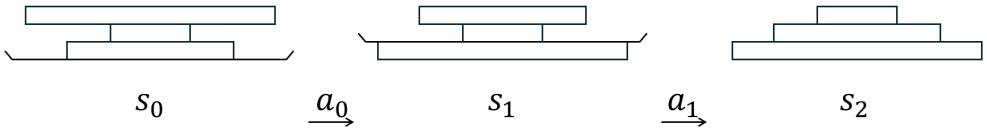
Pro účely této bakalářské práce byl vybrán jeden konkrétní ukázkový problém. Na něm bude znázorněna problematika, používaná terminologie, v rámci implementace bude implementován a také na něm budou provedeny všechny experimenty.

Zvoleným problémem v této práci je řazení palačinek. Tento problém byl poprvé publikovaný v článku [1] roku 1975. Byl definován takto (volný překlad): „*Kuchař v našem podniku je nedbalý, a když připravuje palačinky, má každá jinou velikost. Když je poté nesu zákazníkovi, srovnávám je tak, aby na spodku sloupce palačinek byla ta největší, každá nad ní o něco menší, až k té nejmenší na vrchu sloupce. Dělám to tak, že vždy vezmu několik horních palačinek, a otočím je spodní palačinkou nahoru. To opakuji s různými počty palačinek, dokud je nemám uspořádané. Pokud mám n palačinek, kolik nejvíce otočení budu muset provést?*“

Tímto problémem se zabývalo vícero vědeckých prací. Jednou z nich byla například [2], ve které autoři dokázali, že řešení této otázky je NP-těžký problém. V [3] pak byla roku 2009 představena (nedefinitivní) horní hranice počtu kroků k vyřešení problému o n palačinkách – $(18/11)n$ kroků[1].

Tato bakalářská práce se bude zaměřovat na implementaci algoritmu, který pro dané uspořádání palačinek najde způsob, jak je seřadit. Konkrétní uspořádání palačinek bude v textu nazýváno *stavem*. Proces otáčení několika horních palačinek bude označován *akce*, kde každá rozdílná akce ve stavu představuje jiný počet otočených palačinek. Uspořádání palačinek, které vznikne vykonáním akce a ve stavu s bude nazýváno stavem sousedním stavu s nebo jeho následovníkem.

[1] Touto hranicí se zabýval na začátku své kariéry i americký miliardář Bill Gates, společně s Christos H. Papadimitriou, v článku [4] vydaném již roku 1979. V něm stanovili horní hranici na $(5n + 5)/3$ otočení.



Obrázek 1.1 Ukázka řazení palačinek. Na obrázku jsou tři stavy s_0, s_1 a s_2. Horizontální čarou je naznačeno, ve kterém místě bylo provedeno otočení – příslušné akce a_0 a a_1. Poslední stav reprezentuje uspořádaný sloupec palačinek.

1.2 Strojové učení

Strojové učení je oblastí umělé inteligence, ve které se využívají informace, data a zkušenosti jako podklady pro tvoření správných predikcí či k optimalizaci procesů. Jednotlivé algoritmy přijímají data, obvykle v podobě strukturovaných záznamů, a na jejich základě přizpůsobují svůj výstup – samotný proces učení. [5]

Podle [6] existují tři hlavní kategorie strojového učení:

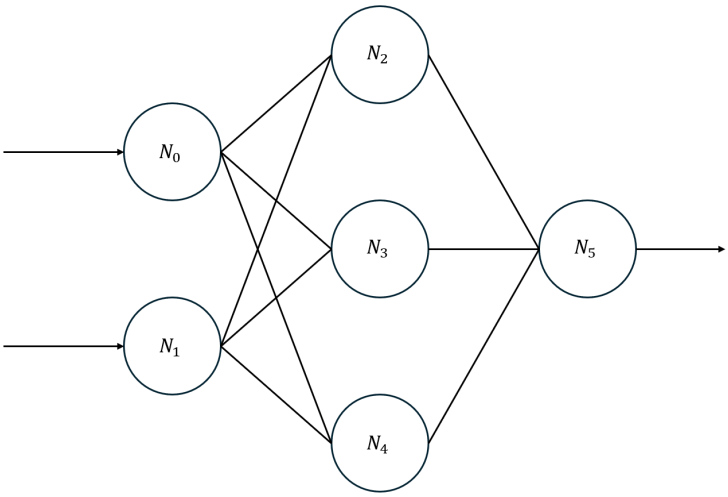
- Učení s učitelem (anglicky *supervised learning*): trénovací data obsahují i očekávané výsledné ohodnocení, které je cílem učení.
- Učení bez učitele (anglicky *unsupervised learning*): vstupní dataset nemá definovaný správný výstup. Cílem je odvodit vhodné spojitosti a vlastnosti dat.
- Posilované učení (anglicky *reinforcement learning*): cílem je naučit agenta provádět akce, které mu přinesou nejvyšší kumulativní odměnu. Tato práce se bude nejvíce zabývat právě posilovaným učením.

Užití nachází strojové učení v mnoha odvětvích. Používá se například pro klasifikaci dokumentů, překládání textů, v kamerách pro rozeznávání pozic osob nebo i v biologických odvětvích pro predikování funkce proteinů [5].

1.3 Neuronové sítě

Neuronové sítě jsou inspirovány neuronovými strukturami v lidském mozku. Skládají se z jednotlivých umělých neuronů, výpočetních jednotek, které provádí vážený součet vstupních hodnot a následně aplikují zvolenou nelineární transformaci. Tyto neurony jsou uspořádány do vrstev, viz schéma 1.2, kde výstup jedné vrstvy se stává vstupem vrstvy následující. První vrstva v neuronové síti se nazývá vstupní vrstva, poslední se označuje jako vrstva výstupní. Všechny zbylé vrstvy mezi nimi jsou nazývány skryté. Podle počtu vrstev se mluví buďto o mělkých, nebo hlubokých neuronových sítích[2]. [7]

[2]Neexistuje přesný konsensus, kde je mezi těmito skupinami hranice. Často je ale za hlubokou označována síť, která má tři a více vrstev – tedy alespoň jednu skrytou.



Obrázek 1.2 Schéma jednoduché neuronové sítě. Umělé neurony N_0 a N_1 jsou v první, vstupní vrstvě. Neurony N_2, N_3 a N_4 představují skrytou vrstvu a neuron N_5 je ve vrstvě výstupní.

V [8] bylo ukázáno, že neuronové sítě jsou univerzální aproximátory – tedy že při dostatečném počtu neuronů jsou schopny aproximovat téměř libovolnou funkci na vybranou přesnost. To je činí potenciálně velmi mocnými nástroji.

1.3.1 Aktivační funkce

Za nelineární transformaci – v rámci neuronových sítí nazývanou aktivační funkce – se dle [7] často volí následující:

- Sigmoida:
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
- Hyperbolický tangens:
$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$
- Rectified linear unit (*ReLU*):
$$\text{ReLU}(x) = \max(0, x)$$
- Leaky rectified linear unit (*Leaky ReLU*) [9]:
$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{pokud } x \geq 0 \\ 0{,}01x & \text{pokud } x < 0 \end{cases}$$

Tyto aktivační funkce slouží k napodobení funkce skutečného neuronu – ten se „spíná“, když jeho vstupy dosáhnou určité hodnoty. Umělý neuron tento jev simuluje pomocí váženého součtu vstupů (kombinace vstupů) a aktivační funkce (sepnutí). [7]

1.3.2 Učení neuronové sítě

Samotný model neuronové sítě se musí podrobit procesu učení – je nutné nastavit váhy vážených součtů v jednotlivých umělých neuronech tak, aby výsledek výpočtu neuronové sítě co nejvíce odpovídal cílenému výsledku. K detekci, jak dobrý je aktuální výpočet neuronové sítě, se využívá *ztrátové funkce* (anglicky *loss*), označovaná L. Ta udává, jak odlišné jsou aktuální výsledky od výsledků optimálních. Cílem učení je tedy minimalizace hodnoty této ztrátové funkce L. [7]

K minimalizaci ztrátové funkce jsou využívány optimalizační algoritmy. Ty využívají znalosti $\nabla L(\theta)$, tedy gradientu ztrátové funkce L pro parametry (váhy) θ_t dané neuronové sítě, spočítané nad trénovacími daty. S pomocí znalosti gradientu následně vytvoří nové parametry θ_{t+1} tak, aby se zmenšila hodnota ztrátové funkce: $L(\theta_{t+1}) < L(\theta_t)$ pro stejná trénovací data. Jednotlivé optimalizační algoritmy mívají vlastní hyperparametry – hodnoty, které určují přesné chování algoritmu, například velikost změny parametrů θ v každém kroku. Tyto hyperparametry ovlivňují kvalitu a rychlost procesu učení, ale jejich přesný vliv nebývá přesně vysvětlen. Pro proces učení tak často bývá vyzkoušeno vícero kombinací hyperparametrů. [10]

Dle [10] se mezi nejčastěji používané optimalizační algoritmy se řadí například:

- stochastický gradientní sestup,
- stochastický gradientní sestup s hybností,
- NESTEROV,
- RMSprop,
- ADAM.

Pro lepší průběh učení se používá tzv. *batching*. Ztrátová funkce (a s ní spojený gradient) se vždy vypočítá na skupině trénovacích dat a gradient pro celou skupinu je využit pro optimalizační algoritmus. Tímto principem je možné zvýšit rychlost zpracování velkého množství dat (nedochází k velkému množství aktualizací parametrů sítě) a zároveň se zlepšuje stabilita procesu učení. Nevýhodou je určitá prodleva mezi získáním dat a jejich vyhodnocením v síti, kvůli nutnosti vyhodnocení vícero záznamů najednou – tato prodleva může být nežádoucí, pokud je očekáváno zpracování dat v reálném čase. [7]

1.3.3 Problém Dying ReLU

Asi nejpopulárnější aktivační funkce ReLU dlouhodobě dosahuje dobrých výsledků. Trpí ale *Dying ReLU* problémem. Ten byl formálně definován v [11]. Problém je spojený s vlastností ReLU, která všem negativním výstupům umělého neuronu přiřadí hodnotu nula. Pokud váhy neuronu byly inicializovány, nebo kvůli nevhodnému gradientu byly upraveny na hodnoty, pro které pro všechny vstupy vrací zápornou hodnotu, po aplikaci ReLU bude výstup vždy roven hodnotě 0. Díky tomu bude i gradient vah daného neuronu vždy nulový a dané váhy tak nebudou v procesu učení dále upravovány – neuron tak efektivně *zemře*. V extrémních případech může dojít k deaktivaci všech neuronů sítě, kde celá neuronová síť zkolabuje na konstantní funkci.

Autoři článku [11] prezentovali více postupů řešení. Jedním z nich bylo používání tehdy již populární *batch normalization* vrstvy představené v [12], která pomáhá zachovat vhodný gradient – tuto metodu implementuje i tato bakalářská práce. Autoři daného článku navrhli také vlastní řešení *Dying ReLU* problému pomocí upravené inicializace vah sítě. Další variantou je využití Leaky ReLU [9], která má malé nenulové hodnoty i v záporných číslech.

1.4 Posilované učení

Posilované učení (anglicky *Reinforcement learning*) je podoblastí strojového učení. V typické úloze strojového učení se vyskytuje (robotický) agent, který interaguje s dynamickým prostředím. Jeho cílem je pak interagovat tak, aby získal co největší odměnu (anglicky *reinforcement*). [13]

Konkrétněji, agent v čase t obdrží stav s a na základě své *policy* (v překladu politika) π zvolí akci a, kterou provede. Následně dostane ohodnocení – odměnu za nový stav s', do kterého vykonáním akce a ve stavu s přešel. Tento proces se opakuje s novým stavem s'. Tento cyklus obvykle končí, když agent dosáhne koncového stavu. Cílem učení je najít optimální policy π^*. Pro tu platí, že pokud jí agent následuje, získá největší možnou odměnu. [14]

Velkou součástí posilovaného učení je problém tzv. *Exploration vs. Exploitation*. Jedná se o problém, kdy agent musí volit mezi otestováním nové či málo prozkoumané policy, která by mohla být potenciálně lepší, než ta stávající, a mezi využíváním své policy k volbě nejlepších rozhodnutí. Příkladem algoritmu, který toto dilema řeší, je tzv. *ε-greedy* algoritmus. Ten pro určité malé ε vybírá nejlepší možnou akci s pravděpodobností $1-\varepsilon$ a s pravděpodobností ε vybere náhodnou akci. [14]

1.4.1 Value Iteration

Value iteration je algoritmus dynamického programování. Má za cíl najít optimální *value function* (česky funkce hodnot) v^*, která každému stavu s přiřadí hodnotu – velikost očekávané kumulativní odměny, kterou agent ve stavu získá

za následování příslušející policy. Tato value function lze rozepsat do tzv. Bellmanovy rovnice [14]:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi\left(s'\right)\right]$$

kde $v_\pi(s)$ představuje hodnotu *value function* při následování policy π v bodě s. Dále $\pi(a \mid s)$ značí pravděpodobnost výběru akce a ve stavu s za dané policy a $p\left(s', r \mid s, a\right)$ pravděpodobnost přechodu do stavu s' a získání odměny r, pokud bude ve stavu s vykonána akce a. Poslední část rovnice $[r + \gamma v_\pi\left(s'\right)]$ pak představuje samotnou odměnu r sečtenou s hodnotou dané *value function* ve stavu s' vynásobenou parametrem γ – *discount factor*. Ten obvykle nabývá hodnot v intervalu $[0, 1]$ a umožňuje agentovi preferovat okamžitou (a jistější) odměnu před odměnami budoucími. [14]

Samotný algoritmus pak iterativně aplikuje následující rovnici [14]:

$$v_{k+1}(s) = \max_a \mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}\left(s' \mid s, a\right) v_k\left(s'\right)$$

Tedy v každém kroku $k+1$ pro všechny stavy s aktualizuje hodnotu $v_{k+1}(s)$ podle hodnot v_k stavů, do kterých může přejít vykonáním některé akcí. $\mathcal{R}(s, a)$ zde představuje odměnu, kterou agent dostane za vykonání akce a ve stavu s, $\mathcal{P}\left(s' \mid s, a\right)$ pravděpodobnost přechodu do stavu s' při vykonání akce a ve stavu s. Je dokázáno, že tento algoritmus konverguje k optimální value policy. [14]

1.4.2 Q-learning

Jiným přístupem je naučit se *action value function* (česky funkce hodnoty-akce) q^*, která přiřazuje hodnotu každé dvojici stav-akce. Stejně jako *value function* se q^* nechá rozepsat do podoby Bellmanovy rovnice [14]:

$$q_\pi(s, a) = \sum_{s',r} p\left(s', r \mid s, a\right) \left[r + \gamma \sum_{a'} \pi\left(a' \mid s'\right) q_\pi\left(s', a'\right)\right]$$

Ta je velice podobná Bellmanově rovnici pro *value function*, hlavním rozdílem je samotný vstup funkce – zde již zmíněná dvojice stav a akce v něm provedená.

V [15] je popsán agentův postup při využívání obecného algoritmu Q-learning, kde funkce $q(s, a)$ je reprezentována vyhledávací tabulkou, následujícím způsobem:

- proveď pozorování o aktuálním stavu s_k
- vyber a proveď akci a_k
- proveď pozorování o aktuálním stavu s_{k+1}

- získej odměnu r_k
- uprav hodnoty q_k za pomoci parametru α_k – faktoru učení následujícím způsobem:

$$q_{k+1}(s,a) = \begin{cases} (1-\alpha_k)\, q_k(s,a) + \alpha_k \left[r_k + \gamma V_k\left(y_n\right)\right] & \text{pokud } s = s_k \text{ a } a = a_k, \\ q_k(s,a) & \text{jinak} \end{cases}$$

kde

$$V_k(s_{k+1}) \equiv \max_b \{q_k(s_{k+1}, b)\}$$

představuje agentův aktuální odhad, kolik odměny může z nově dosaženého stavu nanejvýš ještě získat. Ten se bude během prvních několika iterací k určitě vymykat optimálním hodnotám. Za určitých podmínek, viz [15], ale tento algoritmus konverguje k optimálním hodnotám $q^*(x,a)$ pro všechny stavy a akce. Tento zápis také předpokládá reprezentaci funkce q pomocí vyhledávací tabulky – ostatní reprezentace nemusí umožňovat konvergenci.

1.4.3 Deep Q-learning

Ve skupině algoritmů nazývané hluboké posilované učení se využívá hlubokých neuronových sítí pro aproximaci *policy*, funkce hodnot jednotlivých stavů/dvojic akce-stav, nebo dalších funkcí využívaných agentem [14]. Jeden z nejznámějších algoritmů v této kategorii je *Deep Q-learning*, představený v [16] roku 2015. Představuje spojení klasického Q-learning algoritmu, kde je funkce q místo vyhledávací tabulky reprezentována neuronovou sítí Q. Konkrétněji, neuronová síť přijímá na vstupu reprezentaci stavu a výsledkem je vektor *q-faktorů* – tedy vektor, ve kterém jsou uloženy hodnoty pro jednotlivé možné akce. Snaha zprovoznit tento přístup byla již dříve – ukázalo se ale, že posilované učení bylo nestabilní nebo dokonce divergentní, pokud se aproximovalo nelineární funkcí, kterou jsou právě i zmíněné hluboké neuronové sítě. Autoři algoritmu Deep Q-learning proto přišli s několika úpravami, díky kterým se podařilo učení stabilizovat.

V algoritmu Deep Q-learning, tak jak byl popsán v [16], se vyskytuje učící se agent. Ten v každém časovém bodě t zaznamená údaje o aktuálním prostředí x_1 a dle své funkce Q vybere pomocí ϵ-*greedy* algoritmu nejvhodnější akci a_1 k provedení. Tu provede, zpozoruje nový stav x_2 a dostane okamžitou odměnu r_1. Tuto čtveřici údajů (x_1, a_1, x_2, r_1) uloží do své *replay memory*. Ta představuje určitý zásobník, který uchovává posledních N pozorování. Jeho cílem je snížit vzájemnou korelaci mezi stavy, která by vznikla při použití sekvenčních dat vzniklých pohybem agenta. Z tohoto zásobníku následně agent náhodně vybere M pozorování, které použije pro výpočet ztrátové funkce L, viz dále, jejího gradientu, a na jejichž základě upraví váhy θ své neuronové sítě. [16]

Ztrátová funkce L, v zjednodušeném zápisu, je rovna:

$$L(x_j, a_j) = \left(r_j + \gamma \max_{a'} Q\left(x_{j+1}, a'\right) - Q\left(x_j, a_j\right) \right)^2$$

Jedná se tedy o kvadrát rozdílu očekávané hodnoty výchozího stavu a odměny získané v novém stavu sečtené s odhadem nové nejlepší odměny z tohoto nového stavu. Kdyby ale učení probíhalo takto jednoduše, bylo by silně nestabilní – s úpravou hodnoty pro $Q(x_j, a)$ by se kvůli vlastnostem úpravy vah neuronových sítí upravila i hodnota $Q(x_{j+1}, a')$. To by mělo za následek neustálé měnění cíle učení, které by znemožňovalo konvergenci. Z tohoto důvodu byla zavedena druhá – cílová – neuronová síť Q^-. Ta je zpočátku inicializována na stejné hodnoty jako Q, v průběhu učení má ale vlastní hodnoty vah θ^- pevně nastavené – nemění je s každým časovým krokem. Ztrátová funkce je pak upravena na:

$$L(x_j, a_j) = \left(r_j + \gamma \max_{a'} Q^-\left(x_{j+1}, a'\right) - Q\left(x_j, a_j\right) \right)^2$$

Hodnota nového stavu je tak vypočítána pouze z cílové neuronové sítě. Není proto ovlivněna zpětnou vazbou při úpravách trénované sítě pro hodnotu výchozího stavu. To zajišťuje větší stabilitu algoritmu. Váhy sítě Q^- se pak periodicky synchronizují s váhami učené sítě Q, což zajistí propagaci nových znalostí. Jako další způsob stabilizování byl také omezen rozdíl $r_j + \gamma \max_{a'} Q^-\left(x_{j+1}, a'\right) - Q\left(x_j, a\right)$ ve ztrátové funkci L na hodnoty z intervalu $[-1, 1]$. [16]

1.5 Algoritmus A*

Algoritmus A* byl poprvé představený roku 1968 ve vědeckém článku [17]. Autoři popisují algoritmus, který v grafu – reprezentaci stavového prostoru – nalezne nejkratší cestu z počátečního stavu do nejbližšího stavu koncového. Využívá k tomu hladový princip – v každém kroku vybere dostupný stav n s nejmenší hodnotou *evaluační funkce* $f(n)$. Popis celého algoritmu je k nalezení v ukázce 1.1.

Hodnota funkce f v bodě n pak představuje cenu/délku nejkratší cesty z počátečního stavu k nejbližšímu z koncových stavů, která prochází stavem n. Jde tedy rozepsat na dvě části.

$$f(n) = g(n) + h(n)$$

$g(n)$ představuje délku cesty z počátečního stavu do stavu n a $h(n)$ značí délku cesty z bodu n do nejbližšího koncového stavu. Takováto funkce ale obvykle není k dispozici a musí být použit pouze její odhad $\hat{f}(n)$. Ten se bude skládat také za dvou složek – $\hat{g}(n)$ a $\hat{h}(n)$. $\hat{g}(n)$ je tedy odhadem hodnoty $g(n)$, a lze

1. Označ počáteční stav s jako otevřený, spočítej hodnotu $f(s)$.
2. Vyber otevřený stav n s nejmenší hodnotou $f(n)$, v případě shody hodnot preferuj stav, který je koncový, jinak vyber náhodně.
3. Pokud je stav n koncový, označ ho jako uzavřený a ukonči algoritmus.
4. Jinak označ n jako uzavřený. Pro všechny následovníky stavu n spočítej hodnotu funkce f. Otevři každého následovníka, který ještě nebyl otevřen. Znovu otevři každého uzavřeného následovníka, jehož nově vypočítaná hodnota f je nižší než jeho hodnota f v době, kdy byl uzavřen. Pokračuj opakováním kroku 2).

Popis algoritmu 1.1 Popis algoritmu A*, převzatý z [17].

za něj triviálně zvolit délku aktuálně nejkratší nalezené cesty k bodu n. Odhad funkce $h(n)$ (dále v textu nazývaná také *heuristická funkce*) je složitější a bude mu věnován větší prostor v následujících kapitolách. [17]

1.5.1 Vlastnosti algoritmu A*

Algoritmus A* má za cíl nalézt nejkratší cestu z počátečního bodu do stavu koncového. Autoři v [17] dokázali, že pokud používaná heuristika $\hat{h}(n)$ je přípustná (viz dále), bude výsledkem algoritmu A* zaručeně cesta nejkratší možné délky – algoritmus je optimální. O heuristice řekneme, že je přípustná, pokud platí

$$\hat{h}(n) \leq h(n)$$

pro všechny stavy n. Heuristika tedy musí být *optimistická* – jí určovaná hodnota musí být menší nebo rovna reálné vzdálenosti bodu n od koncového stavu. [17]

Pokud navíc pro heuristickou funkci $\hat{h}(n)$ platí

$$h(m,n) + \hat{h}(n) \geq \hat{h}(m)$$

tedy že hodnota heuristické funkce pro stav n sečtená s reálnou vzdáleností bodu m od n není menší než hodnota heuristické funkce pro stav m, pro libovolnou dvojici stavů (n, m), nazývá se tato heuristická funkce *konzistentní*. V rámci algoritmu A* konzistentnost heuristické funkce zaručí, že pokud stav byl jednou uzavřen, byla nalezena nejkratší cesta k němu – takový stav již nikdy nebude znovuotevřen ve druhém kroku algoritmu 1.1. Zároveň zajišťuje, že algoritmus je optimální – jakýkoli jiný algoritmus, který má stejné množství informací o stavovém prostoru jako A*, bude muset ve vyhledávání nejkratší cesty otevřít minimálně stejný počet stavů jako algoritmus A*. [17]

1.5.2 Heuristické funkce tvořené expertem

Jedním způsobem vytvoření heuristické funkce je její navrhnutí expertem – člověkem, který má určitou znalost o stavovém prostoru. Tento přístup je použitelný v běžných/snadno představitelných případech. Příkladem může být hledání nejkratší cesty mezi městy na mapě. Jedná se o dvourozměrný euklidovský prostor, kde pozici na mapě lze znázornit dvěma souřadnicemi: $m = (x_m, y_m)$. Díky této informovanosti o prostoru lze za heuristickou funkci pro algoritmus A* zvolit Euklidovskou vzdálenost.

$$\hat{h}(m, n) = \sqrt{(x_m - x_n)^2 + (y_m - y_n)^2}$$

Ta je určitě vždy menší nebo rovna realistické vzdálenosti mezi městy (silnice mohou zatáčet, jet oklikou) a proto splňuje podmínku přípustnosti popsanou v sekci 1.5.1. Euklidovská vzdálenost je navíc také konzistentní, a pro tuto míru informovanosti by algoritmus byl pro tento problém optimální. [17]

Metoda určování heuristiky expertem byla použita například i ve vědeckém článku [18], zabývajícím se protokolem pro komunikaci se satelity na oběžné dráze Země – zde byla za heuristickou funkci zvolena vzdálenost jednotlivých bodů, mezi kterými se komunikace směruje.

1.5.3 Heuristické funkce ze strojového učení – DAVI

Druhou variantou konstrukce heuristické funkce je využití metod strojového učení. Jeden takový přístup byl popsán v [19], kde autoři článku představili algoritmus *Deep approximate value iteration*, dále uváděno pouze jako *DAVI*. Jak už název napovídá, jedná se o propojení hlubokých neuronových sítí s algoritmem dynamického programování value iteration (popsán v sekci 1.4.1). DAVI využívá neuronovou síť k aproximaci funkce hodnot – nahrazuje tak obvykle používanou vyhledávací tabulku. Hlavní výhodou je umožnění pracovat s mnohem větším stavovým prostorem. Pro znázornění, problém Rubikovy kostky má celkem $4{,}3 \cdot 10^{19}$ různých stavů. Takto velká tabulka by byla nereprezentovatelná v paměti počítače – neuronová síť má ale fixní velikost a lze pomocí ní získat hodnotu pro jakýkoliv stav. Problémy řešené algoritmem A* jsou navíc ryze deterministické – vykonání libovolné akce v libovolném stavu má vždy přesně jeden možný výsledek – oproti obecné value iteration tedy není nutné počítat s pravděpodobnostmi přechodů.

Algoritmus DAVI se v průběhu učení učí hodnotu stavů, která představuje kolik kroků/akcí musí být provedeno, aby bylo dosaženo koncového stavu – ta se poté může použít jako heuristika algoritmu A*. Za svoji ztrátovou funkci volí kvadrát rozdílu odhadované hodnoty pro stav $J(s)$ a aktualizovaného odhadu $J'(s)$ pro stav s. Aktualizovaný odhad $J'(s)$ je určen jako :

$$J'(s) = \min_a \left(g^a(s, A(s, a)) + J(A(s, a))\right)$$

kde $A(s, a)$ představuje stav dosažený po vykonání akce a ve stavu s a $g^a(s, s')$ je cena přechodu ze stavu s do stavu s' při vykonání akce a. Tento přístup aktualizuje hodnoty stavů pouze na základě stavů sousedních – přesto se ukázalo, že J trénovaná pomocí DAVI je schopná aproximovat optimální funkci hodnot J^*. Za trénovací stavy s byly vybírány stavy vytvořené vykonáním jedné až K akcí ze stavu koncového – to zaručilo distribuci stavu, která umožňovala propagaci informace o vzdálenosti od stavu koncového ke stavům vzdálenějším. Za zmínku stojí také využití cílové neuronové sítě pro stabilizaci učení stejným způsobem, jakým se používá v Deep Q-learning algoritmu (sekce 1.4.3). [19]

Celý algoritmus DAVI funguje dle popisu algoritmu 1.2.

---

1. Inicializuj trénovanou neuronovou síť J a její kopii cílovou neuronovou síť J^-.
2. Pro každou z iterací proveď kroky 3-6.
3. Vyber trénovací stavy X.
4. Pomocí J^- spočítej aktualizované odhady hodnot stavů X.
5. Na základě ztrátové funkce mezi odhady trénované neuronové sítě J a aktualizovanými odhady cílové sítě J^- pro stavy X proveď aktualizaci vah sítě J.
6. Periodicky zkontroluj, zda je hodnota ztrátové funkce dostatečně malá. Pokud ano, přenes váhy trénované neuronové sítě do cílové sítě.

Popis algoritmu 1.2 Popis algoritmu DAVI, na základě [19].

1.5.3.1 Varianty A*

Za účelem zrychlení výpočtu, úspory paměti nebo vylepšení jiných charakteristik algoritmu A* bylo vytvořeno nepřeberné množství jeho variant. Jednou z nich, zaměřující se především na omezení paměťových nároků algoritmu, je *Partial Expansion A** (dále jen PEA*) představený roku 2000 v [20]. Ten se zaměňuje na stavové prostory s velkým množstvím akcí, tedy takové, kde každý stav má mnoho různých sousedů. Problém takovéhoto uspořádání je, že s každou expanzí/vyhodnocení stavu v algoritmu A* (odpovídá kroku 4) v popisu algoritmu 1.1) je do otevřených stavů nutné přidat velké množství nových záznamů. S každou iterací pak tento seznam neúnosně roste. Autoři PEA* proto navrhli zadefinovat konstantní hranici ořezu C s nezápornou hodnotou a k evaluační funkci f přidružili funkci F. Algoritmus postupuje podobně jako A*, z otevřených stavů ale vybírá na základě hodnoty funkce F. Ta je zpočátku inicializována na stejnou hodnotu jako f, v průběhu algoritmu se ale

může měnit, viz dále. Druhý rozdíl spočívá v expanzi stavu (odpovídá kroku 4) v popisu algoritmu 1.1). Poté, co jsou pro expandovaný stav s vypočítány hodnoty evaluační funkce jeho následovníků n_i, za otevřené označí pouze ty následovníky, pro které platí:

$$f(n_i) \leq F(s) + C$$

Stavy, u kterých byla hodnota $f(n_i)$ vyšší než požadováno, nejsou označeny jako otevřené. Místo toho je stav s znovuotevřen, jeho hodnota F se ale upraví na nejnižší hodnotu f jeho neotevřených následovníků. Díky tomuto principu je omezené generování stavů, které nejsou velmi slibné. Ty ale stále mohou být využity v budoucnosti, pokud stav s bude znovu expandován, nyní s (vyšší) hodnotou funkce F. Algoritmus PEA* zároveň zůstává přípustný (garantuje nalezení nejkratší cesty) za přípustnosti použité heuristické funkce. [20]

PEA* je vhodný k minimalizaci paměťových nároků úložiště v případě velmi větvících se stavových prostorů, výpočetní nároky spojené se získáváním hodnot heuristických funkcí ale zůstávají stejné (nebo dokonce vzrůstají z důvodu nutnosti znovuotevírání stavů). Tyto nároky jsou problematické především pokud je heuristická funkce složitá na výpočet – například reprezentace pomocí neuronové sítě. Metodou, která se zaměřuje na omezení počtu výpočtů heuristických funkcí, je například *Deffered Heuristic Evaluation* (dále jen DHE) představená v [21], kde byla použita ve spojitosti s metodou *preferovaných operátorů*. DHE použitá v algoritmu A* nepočítá hodnotu evaluační funkce f pro každého následovníka – místo toho označí za otevřené všechny následovníky a jejich hodnotu f nastaví na hodnotu f vypočítanou pro expandovaný stav. Tímto způsobem zajistí, že pro expandování stavu a otevření nových následovníků je potřeba pouze jeden výpočet heuristické funkce. V [22] bylo ale ukázáno, že takto triviální použití nedosahuje uspokojivých výsledků – algoritmus není schopný preferovat mezi jednotlivými následovníky (k čemuž sloužila v [21] technika preferovaných operátorů) a expanduje tak velké množství zbytečných stavů – v provedených pokusech několikrát nenašel výsledek z důvodu vyčerpání veškeré paměti.

Algoritmem, který obětuje přípustnost a optimalitu algoritmu A* za potenciálně vyšší rychlost a menší paměťovou náročnost, je vážený A*. Tento princip byl představen v [23] a spočívá v úpravě výpočtu evaluační funkce f – v daném článku byla představena jako:

$$f(n) = (1 - \omega) \cdot g(n) + \omega \cdot h(n)$$

kde ω je reálné číslo splňující $0 \leq \omega \leq 1$. Jde tedy o vážený součet složek evaluační funkce - délky cesty k danému bodu g a heuristické funkce h. Čím větší je ω, tím větší důraz je kladen na hodnotu heuristické funkce – takové prohledávání je typicky méně rozvětvené, otevře méně stavů, ale nalezená cesta může být delší, než cesta optimální. [23] Tato metoda byla použita i v [19], s úpravou evaluační funkce na pouhé $f(n) = \omega \cdot g(n) + h(n)$. V daném článku autoři

představili *Batch weighted A* search*, dále jen BWAS, která se zaměřovala na umožnění paralelizace algoritmu v případě reprezentace heuristické funkce neuronovými sítěmi. V takovém případě výpočet heuristické funkce trvá výrazně déle než generování sousedů nebo jiné části algoritmu A*. Autoři proto navrhli algoritmus, který pokaždé expanduje N nejvhodnějších otevřených stavů zároveň. To je umožněno díky velké schopnosti paralelizovat výpočty neuronových sítí na grafických kartách počítače. BWAS vytváří možnost kompromisu – čím menší ω, tím méně expandovaných stavů (za předpokladu dobré heuristiky), ale potenciálně méně optimální řešení. Čím větší počet expandovaných stavů N v jednom kroku, tím rychlejší procházení stavového prostoru, ale potenciálně vyšší paměťové nároky z důvodu expanze nevhodných stavů. [19]

1.6 Algoritmus Q*

Algoritmus Q* byl představen jako alternativa k algoritmu A* v [22]. Jeho hlavním cílem je omezit počet vyhodnocování heuristické funkce při běhu algoritmu. Místo typické neuronové sítě s jedním výstupem, jaká byla představena v DAVI, používá *Q-Network*, stejnou síť, která je použita v Deep Q-Learning algoritmu. Ta během jednoho vyhodnocení pro daný stav s vrátí vektor q-faktorů, hodnot pro dvojice stav-akce, pro všechny akce vykonatelné ze stavu s. Algoritmus Q* pak místo otevřených stavů používá tyto dvojice stav-akce.

1.6.1 Získání heuristické funkce pro Q*

Pro trénování neuronové sítě pro Q* bylo v [22] navrhnuto využít principu Deep Q-learning. Vzhledem k deterministickému stavovému prostoru problému řešených tímto algoritmem je možné zapsat q-faktor (hodnotu pro dvojici stav-akce) jako

$$Q(s,a) = g^a(s,s') + \gamma J(s')$$

kde s' je stav dosažený provedením akce a ve stavu s, $g^a(s,s')$ je cena/délka přechodu a $J(s')$ je odhad hodnoty pro nově dosažený stav s'. J lze dále rozepsat za použití Q jako

$$J(s') = \min_{a'} Q\left(s', a'\right)$$

tedy minimum z všech q-faktorů stavu s'. Funkce Q je reprezentovaná zmíněnou neuronovou sítí, a také využívá cílové neuronové sítě, zde označenou Q^-, pro stabilizaci výpočtů. Ztrátová funkce při trénování neuronové sítě odpovídá:

$$L(s,a) = \left(g^a\left(s,s'\right) + \gamma \min_{a'} Q^-\left(s',a'\right) - Q(s,a)\right)^2$$

tedy opět kvadrát rozdílu odhadované hodnoty q-faktoru $Q(s,a)$ a aktualizované hodnoty daného q-faktoru získané z jeho vypočítání pomocí možných přechodů, viz dříve. Hlavní výhodou použití Q-learningu oproti DAVI je možnost

spočítat minimum ve výpočtu aktualizované hodnoty díky jednomu výpočtu neuronové sítě. [22]

1.6.2 Popis algoritmu Q*

Algoritmus Q* funguje na podobném principu jako A*. Místo otevřených stavů využívá otevřených dvojic stav-akce, které také řadí podle evaluační funkce. Ta je stanovena na

$$f(s,a) = g(s) + g^a(s,s') + h(s')$$

Hodnota $g(s)$ opět odpovídá nejkratší nalezené cestě do stavu s. Vzhledem k tomu, že $g^a(s,s') + h(s')$ odpovídá q-faktoru $Q(s,a)$, a konstrukci neuronové sítě, která vrací vektor všech q-faktorů daného stavu, lze získat hodnotu evaluační funkce všech následovníků bez nutnosti počítat hodnotu heuristické funkce pro každého zvlášť. V popisu algoritmu 1.3 je vidět, že jediným bodem, jehož náročnost roste s rostoucím počtem možných akcí, je označování nových dvojic za otevřené v bodě 7) . [22]

---

1. Vytvoř počáteční dvojici stav-akce z dvojice počáteční stav-prázdná akce, označ za otevřený s hodnotou $f(s,a)$.
2. Vyber otevřenou dvojici stav-akce (s,a) s nejmenší hodnotou f a zjisti stav s', který vznikne provedením akce a ve stavu s.
3. Pokud je stav s' koncový, označ ho jako uzavřený a ukonči algoritmus.
4. Jinak, pokud stav s' nebyl uzavřený, nebo byl uzavřený s vyšší nejkratší nalezenou cestou k němu, uzavři ho s délkou nové nejkratší cesty k němu $g(s')$. V opačném případě pokračuj opakováním kroku 2).
5. Vypočítej vektor q-faktorů pro stav s'.
6. Pro každou akci a' proveditelnou ze stavu s' označ dvojici (s',a') za otevřenou s hodnotou evaluační funkce f rovnou součtu odpovídajícího q-faktoru a $g(s')$. Poté pokračuj opakováním kroku 2).

Popis algoritmu 1.3 Algoritmus Q*, přepsán na základě [22].

Kapitola 2
Implementace

2.1 Použité nástroje

V této sekci budou popsány hlavní použité nástroje pro tuto práci. V rámci zprovoznění byly použity i další balíčky jazyka Julia, například pro úpravu a formátování dat, ukládání nebo jejich vizualizace. Seznam všech použitých balíčků je k nalezení v souboru *Project.toml* v příloze.

2.1.1 Julia

Julia je programovací jazyk představený veřejnosti v [24] v roce 2012. Hlavní myšlenka byla vytvořit jazyk, který dokáže rychle a pro programátora jednoduše zpracovávat problémy v oblastech vědeckých výpočtů, strojového učení, vytěžování informací z dat, lineární algebry či paralelních distribuovaných výpočtů. Pro všechny tyto potřeby samostatně již existovaly vhodné jazyky, ale nebyly dostatečné ve zbylých oblastech.

Julia je vysokoúrovňový dynamický jazyk – umožňuje psát kód nezávisle na typech jednotlivých proměnných (i když umožňuje typy manuálně anotovat) a má relativně jednoduchou syntaxi. Vyznačuje se také plnou podporou maker a s tím možnostmi metaprogramování – schopnost programu generovat si svůj vlastní kód. Julia využívá JIT (*Just In Time*) kompilaci, tedy přístup, kdy se jednotlivé funkce kompilují až ve chvíli, kdy jsou poprvé volány. Typy proměnných jsou během kompilace rozpoznávány pomocí *Dataflow type inference algorithm*. To umožňuje kompilátoru specializovat funkce pro každý typ zvlášť a dosáhnout tak srovnatelně optimalizovaného kódu jako staticky typované jazyky. Hlavním principem jazyku Julia je *Multiple Dispatch*, který umožňuje dynamicky vybírat volanou metodu na základě vstupních parametrů. [25]

Uživatel může s Julia interagovat pomocí REPL (*Read-Eval-Print Loop*), kde uživatel zadává příkazy do terminálu a okamžitě dostává výstupní hodnoty, sepisováním příkazů do .jl souborů nebo s použitím *Jupyter notebook* –

interaktivního prostředí pro Julii, Python, R a jiné jazyky. [25]

2.1.2 Flux.jl

Flux.jl[1] je balíček pro jazyk Julia, představený v [26, 27]. Autoři zde uvádějí knihovnu pro strojové učení, která je psaná pouze v samotném jazyce Julia. Zabývá se zjednodušením a rozšířením přístupu ke grafickým kartám běžně dostupných v Julia a strojovou derivací (anglicky *Algorithmic Differentiation*) nutnou pro získávání gradientů funkcí. Té dosahuje pomocí Zygote – součást Flux.jl, která využívá znalosti syntaktického stromu zdrojového kódu (výhoda toho, že celý Flux.jl framework i většina standardní knihovny Julia je psaná samotnou Julií) k tomu, aby sestrojila derivační graf funkce, pomocí něhož získá složky gradientu odpovídající jednotlivým parametrům.

Dále se Flux.jl zabývá definováním základních modelů a vrstev neuronových sítí, způsoby získávání gradientu, algoritmů optimalizace vah v neuronových sítích a dalších. Za přední podmínku si ale klade *hackability* – umožnění uživateli definovat vlastní modely či algoritmy libovolné části procesu strojového učení a jejich jednořádkovou/jednoduchou integraci do aparátu Flux.jl. [27]

```
using Flux

network = Chain(
    Dense(2 => 3, relu),
    Dense(3 => 1)
)

network([1,0])
```

```
> Float32[-0.7986573]
```

Výpis kódu 2.1 Ukázka Flux.jl kódu - vytvoření jednoduchého modelu neuronové sítě, která do vstupní vrstvy přijímá vektor o dvou číslech, má jednu skrytou vrstvu se třemi neurony a aplikací funkce ReLU. Ve výstupní vrstvě vrací jednu hodnotu. Pomocí `network([1,0])` je pak neuronová síť vyhodnocená pro vstup [1,0]. Výstupem je vektor o jednom čísle – síť nebyla nijak učená, výsledek je tak závislý pouze na náhodné inicializaci.

[1] Stránky projektu: https://fluxml.ai

2.2 Přehled dalších zdrojů pro strojové učení v Julia

Nyní budou představeny některé další balíčky podporující strojové učení v jazyce Julia.

- **MLJ.jl:** Představen v [28]. Zaměřuje se především na jednoduchou integraci a spojitelnost jednotlivých modelů. Nabízí flexibilní a kompaktní rozhraní pro testování modelů, ladění jejich hyperparametrů a porovnávání jednotlivých modelů mezi sebou. Využívá k tomu vlastních typů v Julia, které udávají, jak se má s daty zacházet – například místo klasického Float (číslo s plovoucí čárkou) používá typ Continuous (spojitý datový typ).
- **KNet.jl:** Představen v [29]. Podobně jako Flux.jl implementuje mechanismy pro tvorbu modelů a strojovou derivaci. K té ale na rozdíl od Flux.jl používá balíček Autograd.jl.
- **Tensorflow.jl:** Představen v [30]. Jde o balíček, který vystavuje rozhraní Tensorflow (jeden z nejpopulárnějších balíčků strojového učení pro jazyk Python) do Julie. Pro svojí funkci využívá kombinaci nativního Julia kódu, volání TensorFlow C API i Python Tensorflow implementace.

2.3 Implementace prostředí

V rámci této bakalářské práce bylo navrhnuto minimalistické rozhraní stavového prostoru. Bylo implementováno jako abstraktní typ v jazyce Julia, v příloze se nachází v souboru *envs/env.jl*. Při definici konkrétních problémů je pak toto prostředí rozšířeno, a jsou mu definovány následující metody:

- **Initialize:** pro daný problém vrátí počáteční konfiguraci (v rámci prohledávacích algoritmů jde o koncový stav) stavu zadané velikosti.
- **IsFinal:** pro daný problém otestuje, zda zadaný stav je koncovým stavem – nutně tedy musí platit, že stav získaný z Initialize() splňuje podmínku IsFinal().
- **GetActions:** pro daný problém vrátí všechny proveditelné akce, které lze v zadaném stavu vykonat. Podoba výsledku musí odpovídat vektoru akcí reprezentovaných celými čísly.
- **NextState:** pro daný problém vrátí reprezentaci stavu, do něhož se přejde vykonáním zadané akce v zadaném stavu.

S takto definovaným prostředím pak pracují všechny implementované algoritmy. V průběhu návrhu algoritmů byly využívány také další metody. Jednou z nich byla NoOpAction(), která pro daný problém a stav vrátila označení akce,

která nezměnila stav při vykonání – tedy prázdná akce. Byla tvořena z důvodu implementace prvního kroku algoritmu [22] dle 1.3. Ukázalo se ale, že tento krok lze implementovat i bez ní a je nadbytečná. Druhým případem nevyužité funkce byla ApplyAction(), která přímo měnila stav aplikováním akce bez nutnosti kopie – ve finální verzi není využívaná, neboť pro některé reprezentace stavu tato úprava nemusela být možná (například konstantní/statický vektor hodnot) a výsledné zpomalení algoritmu bylo nepozorovatelné.

2.3.1 Reprezentace problému

Pro problém zpracovávaný touto bakalářskou prací byly utvořeny dvě reprezentace stavového prostoru. Prvním z nich, v příloze definovaný v souboru *envs/pancakes.jl*, byl vektor přirozených čísel. Koncový stav s^n pro n palačinek byl reprezentován jako vektor celých čísel od jedné do n:

$$s^n = \begin{pmatrix} 1 \\ 2 \\ \vdots \\ n \end{pmatrix}$$

Číslo 1 reprezentuje největší palačinku, číslo n palačinku nejmenší. První hodnota ve vektoru značí spodní pozici a poslední hodnota vektoru označuje palačinku na vrchu. V takovéto reprezentaci je velice jednoduché provádět akce – otočení horních k palačinek odpovídá obrácení pořadí posledních k hodnot ve vektoru.

Druhou reprezentací, která je v přiloženém zdrojovém kódu definovaná v souboru *envs/abstract_pancakes.jl*, je využití principu *OneHotEncoding*, kde pro každou pozici p ve sloupci palačinek bude vyhrazeno n míst ve vektoru hodnot. Všechny z těchto n míst příslušejících p pak budou obsazeny nulou, kromě jednoho – místo odpovídající velikosti palačinky zde se nacházející. Koncový stav s^3 pro 3 palačinky by vypadal (vedle přehlednější vizualizace zápisem do matice):

$$s^3 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \leftrightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Nevýhoda tohoto zápisu je jeho paměťová náročnost – pro n palačinek je potřeba vektor o velikosti n^2, pokud nejsou data reprezentována řídkou maticí. V předběžném testování se ukázalo, že nad takovouto reprezentací funguje strojové učení lépe a ve všech pokusech tak bylo pracováno s touto podobou.

2.4 Modely neuronových sítí

V rámci této bakalářské práce bylo třeba navrhnout dvě neuronové sítě – jednu pro algoritmy DAVI a A* a druhou pro Q-learning a Q*. V rámci zachování porovnatelnosti algoritmů byly navrženy tak, aby byly co nejpodobnější. Jediná jejich odlišnost byla v poslední (výstupní) vrstvě – v prvním případě končila jedním umělým neuronem, zatímco ve druhém se zde nacházel stejný počet umělých neuronů jako počet dostupných akcí.

Konkrétní tvar neuronových sítí v zápise balíčku Flux.jl je vidět v ukázce 2.2, případně v souboru *utils.jl* v přiloženém zdrojovém kódu. Jedná o dvě plně propojené vrstvy o stech umělých neuronech. Na ně jsou napojené dva bloky založené na myšlence ResNet [31]. Ty obsahují *batch normalizaci*, která zajišťuje regularizaci a stabilizaci modelu pomocí normalizování výstupů předchozí vrstvy pro danou skupinu (*batch*) dat, následovanou dvěma plně propojenými vrstvami o velikosti 100 neuronů. Po těchto blocích následuje dříve zmíněná poslední vrstva. Výsledné neuronové sítě pro A* a Q* mají 61 101 a 61 909 trénovaných parametrů respektive.

```
Q = Chain(
        Dense(100 => 100,relu),
        Dense(100 => 100,relu),
        SkipConnection(Chain(
            BatchNorm(100, relu),
            Dense(100 => 100,relu),
            Dense(100 => 100,relu),
        ),+),
        SkipConnection(Chain(
            BatchNorm(100, relu),
            Dense(100 => 100,relu),
            Dense(100 => 100,relu),
        ),+),
        Dense(100 => n-1)
    )
```

Výpis kódu 2.2 Struktura použité neuronové sítě. *Dense* představuje plně propojenou vrstvu, *BatchNorm* regularizační vrstvu, *Chain* pak samotný model neuronové sítě. *SkipConnenction* je speciální vrstva – síť uvnitř *SkipConnection* se provede jako kdyby byla součástí vnější sítě, k jejímu výstupu se ale přičte vstup vrstvy *SkipConnection*. Verze pro DAVI a A* se liší v poslední vrstvě, místo n-1 se vyskytuje 1.

2.5 Implementace DAVI

Pro generování heuristiky byl v jazyce Julia implementován algoritmus DAVI, k nalezení v souboru *DQVI.jl* v přiloženém zrdojovém kódu. Byl navrhnut,

aby byl co nejvíce obecný a znovupoužitelný, všechny parametry, které může uživatel nastavit, jsou k nalezení v tabulce 2.1.

model	model neuronové sítě
optimizer	optimalizační algoritmus
env	rozhraní problému
max_iter	počet iterací, po které má algoritmus běžet
train_dataset	trénovací dataset
bellman_func	funkce, která tvoří aktualizované odhady hodnot
name	název souboru pro tvoření záznamů
batch	počet záznamů v trénovací datasetu zpracovávaných najednou
eps	hranice, ke které se musí odhadovaná hodnota a aktualizovaná hodnota přiblížit, aby došlo k upravení vah cílové sítě
log_frequency	jak často vypisovat záznamy
checkpoint_frequency	jak často ukládat stavy neuronových sítí

Tabulka 2.1 Seznam upravitelných parametrů implementovaného algoritmu DAVI. V první části tabulky jsou povinné argumenty, ve druhé části argumenty volitelné.

Samotný algoritmus je implementován na základě popisu algoritmu 1.2. Velkou výhodou implementace je možnost dodat vlastní funkci, která získá aktualizované odhady hodnot. Byly tak implementovány tři různé funkce. První odpovídá algoritmu DAVI. Druhá, dále nazývaná DQVI, implementuje rozhraní podobné Q-learning algoritmu, včetně uzpůsobení pro heuristiku Q* algoritmu, která vyžaduje za výstup vektor q-faktorů. V tomto režimu Q-learning nevyužívá agenta a *replay-memory*, ale pouze předpřipravená trénovací data, viz sekce 2.7. Stále ale trénuje na dvojici stav-akce, kde akce je vybírána na základě Boltzmannova rozdělení, viz rovnice 2.1. Myšlenka takto vyřešit *exploration vs. exploitation* problém (představen v sekci 1.4) byla převzata z [22] společně s hodnotou T nastavenou na 1/3. Oproti DAVI potřebuje DQVI pro vyhodnocení stavu pouze dva běhy neuronové sítě (zatímco u DAVI roste počet vyhodnocení lineárně s počtem možných akcí).

$$p_{s,a} = \frac{e^{(Q(s,a)/T)}}{\sum_{a'=1}^{|\mathcal{A}|} e^{(Q(s,a')/T)}} \tag{2.1}$$

Třetí varianta, dále nazývaná naivní DQVI, je pak spojení těchto dvou funkcí – výsledkem je stále vektor hodnot vhodný pro Q*, aktualizované hodnoty jsou ale vypočítávány pro všechny akce z daného stavu. Tím ale ztrácí výpočetní výhodu oproti DAVI, neboť počet běhů neuronové sítě opět lineárně roste s počtem možných akcí – a výsledný výpočet je dokonce pomalejší než DAVI, a to

kvůli nutnosti počítat minimum q-faktorů pro získání nového aktualizovaného odhadu. Výhoda oproti DQVI spočívá v potenciálně stabilnějším učení.

2.6 Implementace Deep Q-Learning

Pro získání heuristiky algoritmu Q* byl také implementován Deep Q-learning algoritmus, k nalezení v souboru *QLearning.jl* v přiloženém zdrojovém kódu. Podobně jako DAVI byl navržen tak, aby byl co nejvíce univerzální. Oproti DAVI neumožňuje výběr vlastní funkce tvoření aktualizovaných hodnot – vždy se počítá pouze s dvojicí stav-akce a výsledkem je vektor q-faktorů.

Pro urychlení učení také v každém okamžiku existuje K agentů, kteří vykonávají nezávislé akce – to umožňuje rychlejší obnovování akcí a stavů v *replay memory*. Agenti vždy začínají v počáteční konfiguraci (která odpovídá koncovému stavu při prohledávání algoritmem Q*) a v každé iteraci vykonají akci na základě Boltzmannova rozdělení (viz rovnice 2.1). Agenti mají nastavený maximální počet akcí, které můžou vykonat. Po provedení všech svých pohybů jsou vráceni do počáteční konfigurace a opakují svoje prohledávání.

V každé iteraci, po provedení akcí všech agentů, je náhodně vybráno N záznamů z *replay memory* a z nich získány aktualizované hodnoty stejnou funkcí jako u DQVI. Ty jsou použité k aktualizaci hodnot způsobem popsaným v sekci 1.4.3.

2.7 Generování dat

Pro generování trénovacích a testovacích dat byl v jazyce Julia implementován skript *data_generator.jl*. Ten umožňuje pro uživatelem definované rozhraní problému a jeho velikost generovat tři datasety. První odpovídá náhodně generovaným trénovacím stavům, viz níže. Druhý je pak podmnožinou prvního datasetu, kde obsahuje každý stav pouze jednou. Tyto dva jsou tvořeny za účelem případného vzájemného porovnání efektivity učení. Třetím datasetem je testovací dataset. Je tvořen generováním náhodných stavů, ze kterých jsou odstraněny stavy již se vyskytující v trénovacích datasetech. To má za účel umožnit testování generalizace modelů.

Samotné stavy se generují pomocí postupu inspirovaným [22]. Pro každý stav je nejdříve vygenerována počáteční konfigurace. Z tohoto počátečního stavu se provede 0 až K náhodných akcí. Tím vznikne nový stav pro uložení do generovaného datasetu. Pro testování generalizace byla vždy hodnota K volena nižší pro trénovací datasety a vyšší pro datasety testovací.

Je nutné podotknout, že takto vytvořené datasety nebyly využity při učení Deep Q-learning algoritmu. Ten si stavy generuje sám na základě vlastního procházení. Je tedy možné, že se učil i na datech vyskytujících se v testovacím datasetu.

2.8 Implementace A*

Algoritmus A* byl implementován na základě popisu 1.1, v podobě BWAS (viz sekce 1.5.3.1) a je k nalezení v souboru *AStar.jl* v přiloženém zdrojovém kódu. Algoritmus v každé iteraci nejdříve vybere N otevřených stavů s nejmenší hodnotou evaluační funkce. Pokud je méně otevřených stavů než N, otevře všechny z nich. Pro všechny vybrané stavy pak zjistí všechny následovníky a uloží je do jedné matice. Z té jsou vypočítány heuristiky jednotlivých stavů, které jsou použité k vypočítávání evaluačních funkcí algoritmu A*. Výsledné heuristiky jsou před výpočtem upraveny – případné záporné hodnoty jsou upraveny na hodnotu 0. Získaná heuristická funkce by měla mít všechny hodnoty nezáporné, z důvodu nepřesnosti trénovaní neuronových sítí se ale záporné hodnoty můžou při výpočtu vyskytnout. Tímto způsobem jsou regulovány, v opačném případě by mohlo dojít k zacyklení algoritmu. Evaluační funkce je vypočítána jako $f(s) = \lambda g(s) + h(s)$, kde λ je volitelný parametr, $g(s)$ délka nejkratší nalezené cesty ke stavu s a $h(s)$ hodnota heuristické funkce pro stav s.

Uživatel při užívání této implementace algoritmu může kromě definice problému, modelu neuronové sítě a testovacích dat také libovolně určit hodnoty parametrů N a λ, tedy počet najednou prohledávaných stavů a váženou důležitost složek při výpočtu evaluační funkce. Důležitým volitelným parametrem funkce je časový limit – umožňuje ukončit algoritmus a zaznamenat chybnou hodnotu i v případě nenalezení koncového stavu.

2.9 Implementace Q*

Algoritmus Q* byl implementován na co nejpodobnější bázi jako algoritmus A*, aby bylo výsledné porovnání co nejobjektivnější. Implementace je k nalezení v souboru *Qstar.jl*. Také umožňuje definovat hodnoty parametrů λ, N i časový limit. Hlavní rozdíl je při generování následujících stavů – zatímco implementace algoritmu A* generuje všechny sousedy do matice a následně pro ně vypočítává hodnoty heuristických hodnot, algoritmus Q* generuje pro každou otevřenou dvojici stav-akce pouze jeden stav – následníka – pro kterého hodnota z Q-network odpovídá vektoru odpovídá všem hodnotám následujících dvojic stav-akce. Díky tomu se sníží počet stavů nutných k zpracování neuronovou sítí, aniž by se omezilo množství nových otevřených záznamů.

Kapitola 3
Experimenty

Veškeré testy byly prováděny pro problém řazení palačinek o deseti palačinkách. V tomto stavovém prostoru je 3 628 800 různých stavů, devět akcí, které dané stavy mění, a jedna akce která se nijak neprojeví – otočení pouze vrchní palačinky. Vygenerovaná trénovací data byla utvořena postupem popsaným v sekci 2.7 s maximálně osmi provedenými akcemi (parametr K) a testovací dataset s maximálně 15 provedenými akcemi.

3.1 Specifikace platformy

Veškeré experimenty s trénováním neuronových sítí byly provedeny na výpočetní platformě *ClusterFIT* provozované a zpřístupněné za účelem tvorby této bakalářské práce Fakultou Informačních Technologií ČVUT v Praze. Zdroje pro trénování byly přidělovány platformou automaticky, pro všechny testy tedy nemusely být vždy úplně stejné podmínky. Při testování na lokálním, méně výkonném stroji s 16 GB RAM paměti, procesorem *11th Gen Intel(R) Core(TM)i5-11400H @ 2.70GHz* a grafickou kartou *NVIDIA GeForce RTX 3060 Laptop GPU* se ale ukázalo, že v průběhu učení neuronových sítí byl výkon více než dostačující. Proto by ani výpočty na platformě ClusterFIT neměly být omezeny hardwarovou limitací. Konkrétní specifikace použitých zdrojů jsou k nalezení v tabulce 3.1. Experimenty s algoritmy A* a Q* byly provedeny na dříve zmíněném lokálním stroji.

3.2 Trénování neuronových sítí

Neuronové sítě byly trénovány čtyřmi způsoby. Pro algoritmus A* byla síť trénovaná pomocí algoritmu DAVI. Pro algoritmus Q* byly vyzkoušeny metody DQVI, naivní DQVI a Deep Q-learning.

První tři metody byly trénovány pro tři různé optimalizační algoritmy – RMSprop, ADAM a NESTEROV – pro několik hodnot hyperparametru *lear-*

Procesory	Intel® Xeon® Gold 6136 CPU @ 3.00GHz Intel® Xeon® Gold 6254 CPU @ 3.10GHz
Grafické karty	NVIDIA Tesla P100 16GB NVIDIA Tesla V100 32GB NVIDIA Tesla A100 40GB NVIDIA Tesla A100 80GB

Tabulka 3.1 Zdroje používané výpočetními uzly platformy ClusterFIT, které byly automaticky přidělovány experimentům. Každý výpočetní uzel také disponoval 64 GB RAM paměti.

ning rate (česky rychlost učení / učící parametr). Každá síť byla trénována po 5 000 iterací. Jedna iterace v tomto případě odpovídá vyhodnocení všech trénovacích dat, kde trénovací data představovaly 25 000 stavů (generování popsáno v sekci 2.7). Algoritmy vždy pracovaly s dávkou dat (batch) o velikosti 5 000 stavů.

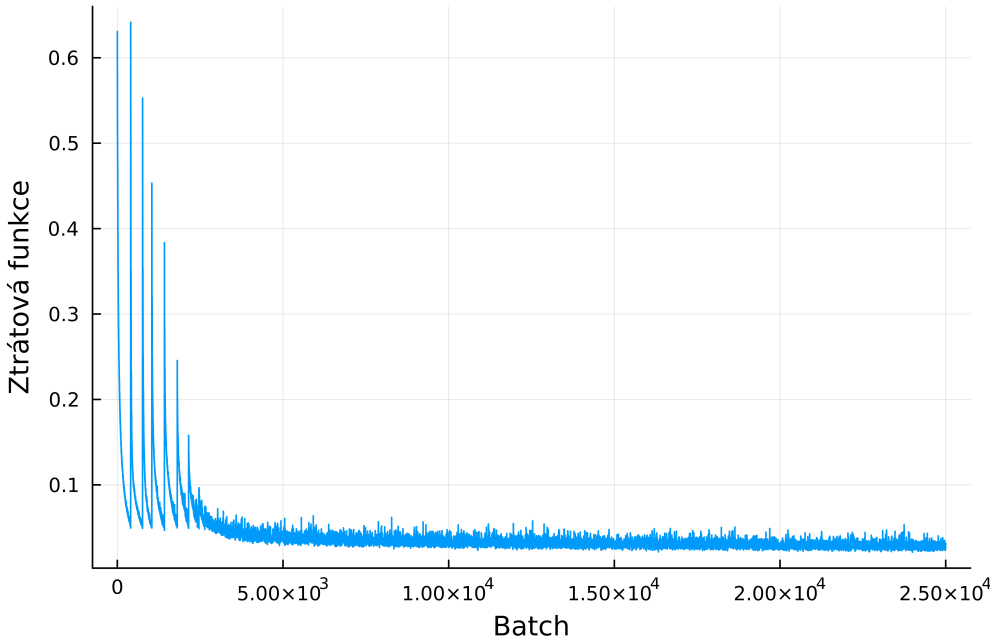
V těchto podmínkách byl snadno vidět rozdíl v rychlosti algoritmů – zatímco naučení DAVI a naivní DQVI obvykle trvalo 2,5 hodiny, DQVI provedlo 5 000 iterací již za 40 minut, tedy s více než pětinásobným zrychlením. To odpovídá očekávání. Při opakovaném běhu na jiném stroji se konkrétní hodnoty mohou lišit.

Algoritmus Deep Q-learning nevyužívá přegenerovaných stavů. Místo nich sám generuje nové stavy, na kterých trénuje – popsáno v sekci 2.6. Deep Q-learning byl trénován pouze s optimalizačním algoritmem ADAM, pro 1 000 000 iterací. Iterace v tomto případě odpovídá zpracování jedné skupiny (batch) dat o velikosti 5 000 stavů. Během učení bylo simulováno 100 agentů současně, bylo experimentováno s *learning rate* algoritmu ADAM, velikostí *replay memory* i hranicí, pro kterou se hodnoty cílové a trénované sítě musí přiblížit, aby se aktualizovaly parametry cílové sítě.

3.3 Vývoj ztrátové funkce

Při učení neuronových sítí byly zaznamenávány aktuální hodnoty ztrátové funkce. Ukázka vývoje ztrátové funkce může je znázorněna v grafu 3.1. V něm je zobrazen průběh při učení algoritmem DAVI. Graf zobrazuje, že v první části běhu se hodnota ztrátové funkce rychle snižuje. Ve chvíli, kdy je dosaženo požadované hodnoty, aktualizují se váhy cílové sítě. Tím, že se změní cílové hodnoty, skokově naroste hodnota ztrátové funkce – lze pozorovat v grafu. V průběhu učení se ale tyto výstřelky postupně snižují, až dojde ke konvergenci – hodnoty ztrátové funkce se ustálily pod/u hranice aktualizování cílové sítě. To může být způsobeno vícero příčinami – v ideálním případě se neuronová síť správně naučila heuristickou funkci. Mohlo se ale také stát, že hraniční hodnota byla nastavena příliš vysoko a nyní neumožňuje síti naučit se detailnější podobu problému. Třetí variantou je přeučení sítě na trénovacích datech. To

by znamenalo, že síti bylo poskytnuto příliš málo trénovacích dat nebo data příliš podobná. Neuronová síť se naučila heuristiku pro tato data, nebude ale schopná generalizovat pro stavy mimo trénovací sadu.

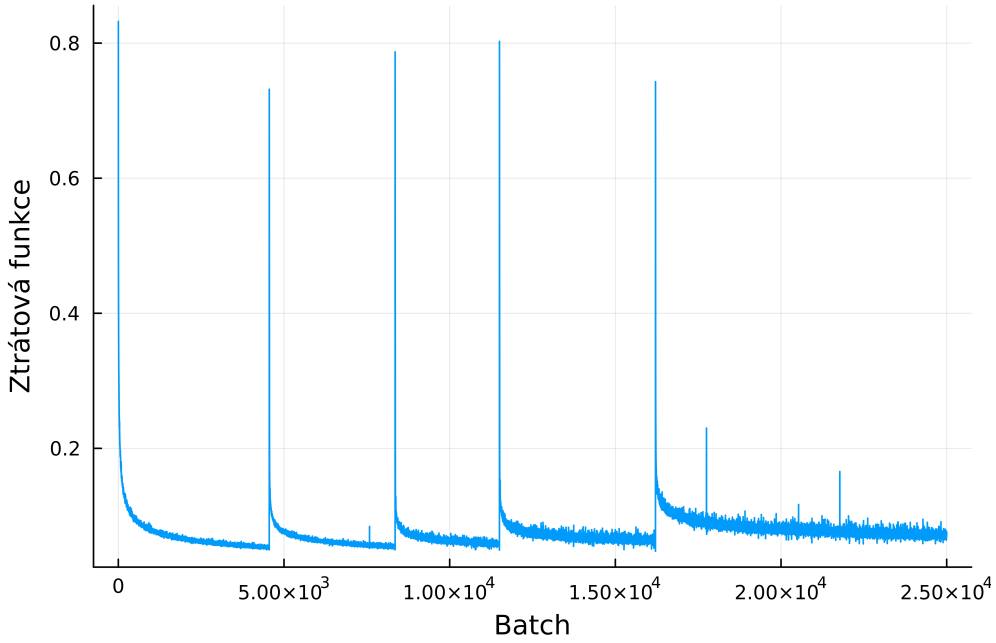


Obrázek 3.1 Vývoj ztrátové funkce. Výstup pro algoritmus DAVI s použitím optimalizačního algoritmu ADAM. Learning rate byl nastaven na hodnotu 0,0001.

Jiný průběh je vidět v grafu 3.2. Ztrátová funkce se v této konfiguraci snižovala k optimálním hodnotám pomaleji. Za celý běh byla cílová síť aktualizována pouze čtyřikrát, což lze opět snadno vidět na grafu. Na rozdíl od předchozí ukázky zde zřejmě nedošlo ke konvergenci – delší běh algoritmu by mohl pomoci.

3.4 Výběr vhodných heuristik

V rámci experimentů bylo vyzkoušeno vícero optimalizačních algoritmů a hodnot jejich parametru *learning rate*. V tabulce 3.2 lze vidět využití heuristické funkce získané z DAVI pro algoritmus A*, kde byl A* spouštěn s hodnotou parametru $\lambda = 1$ a *batch* byla o velikosti 100 stavů. Během experimentu byly vyzkoušeny tři různé optimalizační algoritmy pro dvě hodnoty parametru *leraning rate*. Jednotlivé běhy byly omezeny na 10 vteřin. Pokud algoritmus do té doby nenalezl správnou cestu, byl ukončen a zaznamenán neúspěch. Lze zpozorovat, že (libovolná) cesta byla nalezena v 5/6 případů. Jediná kombinace, která nedokázala najít cestu byl (NESTEROV-0,0001), zde byla i nejdelší doba běhu testu. Výsledek je logický, neboť zatímco ostatní algoritmy relativně



Obrázek 3.2 Vývoj ztrátové funkce. Výstup pro algoritmus naivní DQVI s použitím optimalizačního algoritmu RMSprop. Learning rate byl nastaven na hodnotu 0,0005.

rychle nalezly cestu, tato nevhodná kombinace v 33 případech dosáhla limitu 10 vteřin. Jako nejrychlejší kombinace se ukázal (ADAM-0,0005), a to i při opakovaných běžích/jiném výběru testovacích dat. Pro porovnání v následující kapitole tak byla vybrána právě tato neuronová síť.

Optimalizační algoritmy	**Learning rate**			
	0,0001		0,0005	
ADAM	100	18 s	100	12 s
NESTEROV	67	426 s	100	36 s
RMSprop	100	17 s	100	14 s

Tabulka 3.2 Počet nalezených cest a doba běhu na 100 stavech – trénováno DAVI, vyhodnocení A*

K diametrálně jinému výsledku dospělo učení neuronových sítí pomocí algoritmu DQVI pro algoritmus Q*. V tabulce 3.3 si lze povšimnout, že ve 4/6 kombinací byl Q* neúspěšný pro více než polovinu stavů. Nejlepší byl tentokrát optimalizační algoritmus RMSprop, který vyřešil 96/100 stavů.

Podobný experiment proběhl i s metodou naivní DQVI, k vidění v tabulce 3.4. V tomto testu byly experimentálně použity vyšší hodnoty parametru learning rate, stanoveny na 0,005 a 0,01. Algoritmus Q* s pomocí těchto sítí výrazně předčil klasické DQVI. Je ale vhodné zdůraznit, že naivní DQVI se učí všechny q-faktory stavů zároveň a doba trénování neuronové sítě od-

Optimalizační algoritmy	Learning rate			
	0,0001		0,0005	
ADAM	0	1 001 s	75	355 s
NESTEROV	1	990 s	41	619 s
RMSprop	96	101 s	38	735 s

Tabulka 3.3 Počet nalezených cest a doba běhu na 100 stavech – trénováno DQVI, vyhodnocení Q*

povídala algoritmu DAVI, zatímco DQVI byl více než pětinásobně rychlejší. Je tak možné, že s odpovídající dobou trénování by se výsledky DQVI a naivní DQVI přiblížily. Pro naivní DQVI nejlépe fungovala dvojice optimalizační algoritmus-learning rate (ADAM-0,005).

Optimalizační algoritmy	Learning rate			
	0,0001		0,0005	
ADAM	100	20 s	100	58 s
NESTEROV	23	780 s	96	111 s
RMSprop	93	149 s	67	429 s

Tabulka 3.4 Počet nalezených cest a doba běhu na 100 stavech – trénováno naivním DQVI, vyhodnocení Q*

Algoritmus Deep Q-learning byl trénován za použití optimalizačního algoritmu ADAM, pro který byly testovány různé hodnoty learning rate, společně s velikostí replay memory a hraniční hodnotou pro aktualizování vah cílové sítě. Ukázka výsledků je k nalezení v tabulce 3.5. Při trénování často docházelo k divergenci – pro potřeby heuristické sítě tak byl použit stav neuronové sítě v době posledního aktualizování hodnot cílové sítě. Jedinou výjimkou je první záznam tabulky, kdy aktualizování proběhlo více než dvacet tisíckrát, pravděpodobně z důvodu vysoké tolerance k ztrátové funkci – je tak použit snímek po tisícím aktualizování cílové sítě.

Learning rate	Replay memory	Hraniční hodnota	Vyřešeno	Čas
0,0005	50 000	0,01	100	23 s
0,001	50 000	0,01	96	150 s
0,00005	150 000	0,005	95	175 s

Tabulka 3.5 Počet nalezených cest a doba běhu na 100 stavech – trénováno Deep Q-learning, vyhodnocení Q*

3.5 Porovnání A* a Q*

Pro porovnání algoritmů A* a Q* byly vybrány dvě heuristické funkce. Pro algoritmus A* byla vybrána neuronová síť odpovídající parametrům (ADAM-0,0005) v tabulce 3.2. Pro algoritmus Q* byla vybrána síť získaná metodou Deep Q-Learning, odpovídající prvnímu řádku v tabulce 3.5.

Pro oba algoritmy byly vyzkoušeny různé parametry λ a velikosti *batch*. V tabulkách 3.6 a 3.7 je zaznačena úspěšnost a doba běhu jednotlivých algoritmů pro 100 testovacích stavů. Oba algoritmy nalezly nějaké řešení pro všechny stavy. Algoritmy se ale liší v závislosti doby výpočtu na velikosti zpracovávaných dat (batch). Zatímco pro A* byly malé velikosti batch vždy nejrychlejší a větší pomalé, u Q* probíhalo nejrychleji učení pro batch o velikosti 100 stavů. Pro batch o velikosti 1 000 stavů byl Q* dokonce rychlejší než A* pro všechny hodnoty λ.

	λ							
batch	0,2		0,5		0,8		1,0	
10	100	2 s	100	4 s	100	9 s	100	13 s
100	100	5 s	100	6 s	100	10 s	100	12 s
1 000	100	35 s	100	37 s	100	35 s	100	37 s

Tabulka 3.6 Počet nalezených cest a doba běhu na 100 stavech pro algoritmus A*

	λ							
batch	0,2		0,5		0,8		1,0	
10	100	13 s	100	20 s	100	35 s	100	42 s
100	100	7 s	100	12 s	100	18 s	100	23 s
1 000	100	22 s	100	23 s	100	27 s	100	31 s

Tabulka 3.7 Počet nalezených cest a doba běhu na 100 stavech pro algoritmus Q*

Zajímavé je porovnání počtu expandovaných stavů a průměrné délky nalezené cesty. Výsledky jsou k nahlédnutí v tabulkách 3.8 a 3.9. Ukazuje se, že zatímco A* nachází kratší cesty než Q*, využívá k tomu vícero výpočtu heuristické funkce. V krajním případě, kdy λ byla nastavena na hodnotu 1 a batch měla velikost 1 000, využil algoritmus Q* pouze pětinu výpočtů heuristické funkce oproti algoritmu A*.

Neuronové sítě, se kterými se pracuje v této bakalářské práci, jsou poměrně malé a jejich výpočet netrvá dlouho. Kdyby používané sítě byly násobně větší nebo složitější, s velkou pravděpodobností by se nutnost vyhodnocovat vícero heuristických funkcí projevila i v celkové době běhu algoritmu A*. Pro zvolený problém a strukturu neuronové sítě ale A* podává lepší výsledky než Q* jak v otázce rychlosti nalezení řešení, tak jeho korektnosti.

Byl také proveden experiment s limitací algoritmů na maximální délku

	λ							
batch	0,2		0,5		0,8		1,0	
10	842,32	7,12	1 991,80	6,82	3 924,01	6,72	5 641,27	6,68
100	3 678,57	6,85	4 574,76	6,73	6 299,11	6,69	7 659,49	6,66
1 000	24 446,76	6,70	24 292,69	6,67	25 126,43	6,66	25 767,36	6,66

Tabulka 3.8 Průměrný počet vypočtených heuristik a délka cesty na 100 stavech pro algoritmus A*

	λ							
batch	0,2		0,5		0,8		1,0	
10	842,25	12,33	1 256,18	8,86	2 090,03	7,93	2 618,92	7,63
100	889,54	9,55	1 474,49	8,09	2 112,93	7,54	2 749,42	7,39
1 000	3 951,08	7,44	4 068,33	7,3	4 693,53	7,16	5 139,29	7,05

Tabulka 3.9 Průměrný počet vypočtených heuristik a délka cesty na 100 stavech pro algoritmus Q*

cesty. Algoritmy při nalezení cesty zkontrolovaly její délku, a pokud byla delší než nastavená hranice, pokračovaly v hledání nového, lepšího řešení. Hranice byla nastavena na deseti krocích, což odpovídá nejdelší nalezené cestě na testovacích datech v předešlých výsledcích pro parametry algoritmu A* s nejkratší průměrnou délkou cesty. Výsledky o úspěšnosti a době běhu jsou k nalezení v tabulkách 3.10 a 3.11. Lze zpozorovat, že vliv limitu na A* byl minimální. Oproti výsledkům v tabulce 3.6 jsou doby běhu nepatrně vyšší, ale nelze zavrhnout nepřesnost měření kvůli jinému vytížení lokálního stroje. Jediným rozdílem je nenalezení jedné cesty a s ní spojený nárůst doby výpočtu pro nejmenší velikost batch a nejnižší learning rate parametr. Úspěšnost algoritmu Q* s limitací na délku cesty se zhoršila výrazněji než u algoritmu A*, zvláště pro malé hodnoty λ a malé batch velikosti. Vzrostla i doba běhu algoritmu.

	λ							
batch	0,2		0,5		0,8		1,0	
10	99	14 s	100	5 s	100	9 s	100	14 s
100	100	7 s	100	7 s	100	10 s	100	12 s
1 000	100	36 s	100	35 s	100	36 s	100	35 s

Tabulka 3.10 Počet nalezených cest a doba běhu na 100 stavech pro algoritmus A* s limitací na maximální délku cesty 10 kroků

V tabulkách 3.12 a 3.13 jsou zaznamenány průměrné počty vyhodnocených heuristických funkcí v bězích A* a Q* omezených maximální délkou cesty a průměrná délka cest. Data jsou počítána pouze z nalezených cest, pro některé kombinace hyperparametrů tak nelze utvořit relevantní porovnání s předchozími výsledky. Pro kombinace, kde relevantní porovnání udělat lze, tedy přede-

	λ							
batch	0,2		0,5		0,8		1,0	
10	87	233 s	95	87 s	99	60 s	100	56 s
100	97	86 s	98	40 s	100	31 s	100	29 s
1 000	98	49 s	99	39 s	100	32 s	100	34 s

Tabulka 3.11 Počet nalezených cest a doba běhu na 100 stavech pro algoritmus Q* s limitací na maximální délku cesty 10 kroků

vším vyšší hodnoty λ, platí, že omezení nemělo téměř žádný vliv na algoritmus A*. Ten už dříve nacházel většinu cest pod délku deseti kroků. Algoritmus Q* je proti svojí neomezené variantě v tabulce 3.9 lehce přesnější, ale také k tomu využívá vícero vyhodnocení heuristických funkcí. Při porovnání algoritmů A* a Q* mezi sebou se potvrzuje výsledek dřívějšího pozorování – A* je rychlejší a nachází kratší cesty, zatímco Q* vyhodnocuje méně heuristických funkcí.

	λ							
batch	0,2		0,5		0,8		1,0	
10	1 869,79	7,04	1 991,80	6,82	3 924,01	6,72	5 641,27	6,68
100	5 020,47	6,83	4 574,76	6,73	6 299,11	6,69	7 659,49	6,66
1 000	24 519,22	6,69	24 292,69	6,67	25 126,43	6,66	25 767,36	6,66

Tabulka 3.12 Průměrný počet vypočtených heuristik a délka cesty na 100 stavech pro algoritmus A* s limitací na maximální délku cesty 10 kroků

	λ							
batch	0,2		0,5		0,8		1,0	
10	6 023,03	8,45	2 191,56	8,08	2 931,70	7,71	3 348,00	7,63
100	5 740,94	7,76	2 353,50	7,61	3 311,93	7,42	3 312,29	7,39
1 000	4 636,13	7,12	4 751,44	7,11	5 259,36	7,07	5 589,53	7,02

Tabulka 3.13 Průměrný počet vypočtených heuristik a délka cesty na 100 stavech pro algoritmus Q* s limitací na maximální délku cesty 10 kroků

Kapitola 4
Závěr

Jedním z cílů této bakalářské práce bylo představit algoritmy hledání cesty ve stavovém prostoru A* a Q* společně s metodami strojového učení, které je podporují. Za tímto účelem byly popsány neuronové sítě a jejich učení, posilované učení a jeho podoba Deep Q-learning. Byl vysvětlen princip algoritmů A* a Q* a byly naznačeny typické situace jejich použití. Pro algoritmus A* byla popsána technika získávání heuristické funkce pomocí algoritmu DAVI. Z algoritmu DAVI byly také odvozeny dvě nové varianty DQVI a naivní DQVI, které byly použity pro konstrukci heuristické funkce pro algoritmus Q*, společně s dříve zmíněným algoritmem Deep Q-learning.

Dle cílů bakalářské práce byly algoritmy A* a Q* implementovány v jazyce Julia. Za využití balíčku Flux.jl byly navrhnuty neuronové sítě, které byly trénovány vlastní Julia implementací algoritmů DAVI, DQVI, naivní DQVI i Deep Q-learning. Všechny čtyři způsoby byly otestovány pro problém řazení palačinek. Ukázalo se, že především DQVI je nejrychlejší co se počtu iterací týče, DAVI ve spojení s A* algoritmem ale dávalo nejkonzistentnější výsledky.

Vybrané neuronové sítě natrénované pomocí algoritmů DAVI a Deep Q-learning byly použity pro porovnání algoritmů A* a Q*. Ukázalo se, že A* předčil v daných podmínkách algoritmus Q* v rychlosti i přesnosti. Q* byl naopak úspornější v používání heuristické funkce.

Výsledkem této bakalářské práce je kromě samotného textu také soubor zdrojových kódů. Algoritmy jsou psané genericky, aby uživatel mohl specifikovat nové stavové prostory a upravovat parametry učení neuronových sítí i algoritmů A* a Q*.

Příloha A

Manuál zprovoznění

Pro zprovoznění zdrojových kódů této bakalářské práce je nutné nainstalovat programovací jazyk Julia. Je doporučeno postupovat dle instrukcí na oficiálních webových stránkách projektu Julia[1].

Po instalaci Julia rozbalte přiložený adresář. V terminálovém okně spusťte v lokaci zdrojových kódů příkaz:

```
julia --project precompile.jl
```

Tím se nainstalují požadované balíčky jazyka Julia. Zdrojový kód byl vyvíjený v prostředí *Visual Studio Code* a je pro něj nejvíce uzpůsobený. Pokud je *Visual Studio Code* k dispozici, je pro funkci nutné stáhnutí rozšíření pro jazyk Julia. Poté lze interagovat přímo se zdrojovými kódy spuštěním bloků kódu v *.jl* souborech klávesovou zkratkou *Ctrl+Enter*. V *Examples.jl* jsou připraveny různé funkce a příkazy k spuštění trénování neuronových sítí či algoritmů A* a Q*.

Pokud není *Viusal Studio Code* dostupné, je možné *Examples.jl* editovat v libovolném textovém editoru. Pro jeho spuštění zadejte v lokaci zdrojových kódů příkaz:

```
julia --project Examples.jl
```

[1] V době tvorby této práce je přímý odkaz: https://julialang.org/

Bibliografie

1. KLEITMAN, D. J.; KRAMER, Edvard; CONWAY, J. H.; BELL, Stroughton; DWEIGHTER, Harry. Elementary Problems: E2564-E2569. *The American Mathematical Monthly* [online]. 1975, roč. 82, č. 10, s. 1009–1010 [cit. 2024-05-13]. ISSN 00029890, ISSN 19300972. Dostupné z: `http://www.jstor.org/stable/2318260`.
2. BULTEAU, Laurent; FERTIN, Guillaume; RUSU, Irena. Pancake Flipping is hard. *Journal of Computer and System Sciences*. 2015, roč. 81, č. 8, s. 1556–1574. ISSN 0022-0000. Dostupné z DOI: `https://doi.org/10.1016/j.jcss.2015.02.003`.
3. CHITTURI, B.; FAHLE, W.; MENG, Z.; MORALES, L.; SHIELDS, C.O.; SUDBOROUGH, I.H.; VOIT, W. An (18/11)n upper bound for sorting by prefix reversals. *Theoretical Computer Science*. 2009, roč. 410, č. 36, s. 3372–3390. ISSN 0304-3975. Dostupné z DOI: `https://doi.org/10.1016/j.tcs.2008.04.045`.
4. GATES, William H.; PAPADIMITRIOU, Christos H. Bounds for sorting by prefix reversal. *Discrete Mathematics*. 1979, roč. 27, č. 1, s. 47–57. ISSN 0012-365X. Dostupné z DOI: `https://doi.org/10.1016/0012-365X(79)90068-2`.
5. MOHRI, M.; ROSTAMIZADEH, A.; TALWALKAR, A. *Foundations of Machine Learning, second edition*. MIT Press, 2018. Adaptive Computation and Machine Learning series. ISBN 9780262351362. Dostupné také z: `https://books.google.cz/books?id=dWB9DwAAQBAJ`.
6. FRANÇOIS-LAVET, Vincent; HENDERSON, Peter; ISLAM, Riashat; BELLEMARE, Marc G.; PINEAU, Joelle. An Introduction to Deep Reinforcement Learning. *Foundations and Trends® in Machine Learning*. 2018, roč. 11, č. 3–4, s. 219–354. ISSN 1935-8245. Dostupné z DOI: `10.1561/2200000071`.

7. SZE, Vivienne; CHEN, Yu-Hsin; YANG, Tien-Ju; EMER, Joel S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*. 2017, roč. 105, č. 12, s. 2295–2329. Dostupné z DOI: 10.1109/JPROC.2017.2761740.
8. HORNIK, Kurt; STINCHCOMBE, Maxwell B.; WHITE, Halbert L. Multilayer feedforward networks are universal approximators. *Neural Networks*. 1989, roč. 2, s. 359–366. Dostupné z DOI: 10.1016/0893-6080(89)90020-8.
9. MAAS, Andrew L; HANNUN, Awni Y; NG, Andrew Y. Rectifier nonlinearities improve neural network acoustic models. In: 2013, sv. 30. Dostupné také z: http://robotics.stanford.edu/~amaas/papers/relu_hybrid_icml2013_final.pdf.
10. CHOI, Dami; SHALLUE, Christopher J.; NADO, Zachary; LEE, Jaehoon; MADDISON, Chris J.; DAHL, George E. *On Empirical Comparisons of Optimizers for Deep Learning*. 2020. Dostupné z arXiv: 1910.05446 [cs.LG].
11. LU, Lu; SHIN, Yeonjong; SU, Yanhui; KARNIADAKIS, George. Dying ReLU and Initialization: Theory and Numerical Examples. *Communications in Computational Physics*. 2020, roč. 28, s. 1671–1706. Dostupné z DOI: 10.4208/cicp.OA-2020-0165.
12. IOFFE, Sergey; SZEGEDY, Christian. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. Dostupné z arXiv: 1502.03167 [cs.LG].
13. KAELBLING, L. P.; LITTMAN, M. L.; MOORE, A. W. *Reinforcement Learning: A Survey*. 1996. Dostupné z arXiv: cs/9605103 [cs.AI].
14. LI, Yuxi. *Deep Reinforcement Learning*. 2018. Dostupné z arXiv: 1810.06339 [cs.LG].
15. WATKINS, Christopher J. C. H.; DAYAN, Peter. Q-learning. *Machine Learning*. 1992, roč. 8, č. 3–4, s. 279–292. ISSN 1573-0565. Dostupné z DOI: 10.1007/bf00992698.
16. MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; RUSU, Andrei A.; VENESS, Joel; BELLEMARE, Marc G.; GRAVES, Alex; RIEDMILLER, Martin; FIDJELAND, Andreas K.; OSTROVSKI, Georg; PETERSEN, Stig; BEATTIE, Charles; SADIK, Amir; ANTONOGLOU, Ioannis; KING, Helen; KUMARAN, Dharshan; WIERSTRA, Daan; LEGG, Shane; HASSABIS, Demis. Human-level control through deep reinforcement learning. *Nature*. 2015, roč. 518, č. 7540, s. 529–533. ISSN 1476-4687. Dostupné z DOI: 10.1038/nature14236.

17. HART, Peter E.; NILSSON, Nils J.; RAPHAEL, Bertram. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*. 1968, roč. 4, č. 2, s. 100–107. Dostupné z DOI: 10.1109/TSSC.1968.300136.

18. JI, Xuezhi; LIU, Lixiang; ZHAO, Pei; WANG, Dapeng. A-Star algorithm based on-demand routing protocol for hierarchical LEO/MEO satellite networks. In: *2015 IEEE International Conference on Big Data (Big Data)*. 2015, s. 1545–1549. Dostupné z DOI: 10.1109/BigData.2015.7363918.

19. AGOSTINELLI, Forest; MCALEER, Stephen; SHMAKOV, Alexander; BALDI, Pierre. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*. 2019, roč. 1, č. 8, s. 356–363. ISSN 2522-5839. Dostupné z DOI: 10.1038/s42256-019-0070-z.

20. YOSHIZUMI, Takayuki; MIURA, Teruhisa; ISHIDA, Toru. A* with Partial Expansion for Large Branching Factor Problems. In: KAUTZ, Henry A.; PORTER, Bruce W. (ed.). *AAAI/IAAI*. AAAI Press / The MIT Press, 2000, s. 923–929. ISBN 0-262-51112-6. Dostupné také z: http://dblp.uni-trier.de/db/conf/aaai/aaai2000.html#YoshizumiMI00.

21. HELMERT, M. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*. 2006, roč. 26, s. 191–246. ISSN 1076-9757. Dostupné z DOI: 10.1613/jair.1705.

22. AGOSTINELLI, Forest; SHMAKOV, Alexander; MCALEER, Stephen; FOX, Roy; BALDI, Pierre. *A* Search Without Expansions: Learning Heuristic Functions with Deep Q-Networks*. 2023. Dostupné z arXiv: 2102.04518 [cs.AI].

23. POHL, Ira. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*. 1970, roč. 1, č. 3, s. 193–204. ISSN 0004-3702. Dostupné z DOI: https://doi.org/10.1016/0004-3702(70)90007-X.

24. BEZANSON, Jeff; KARPINSKI, Stefan; SHAH, Viral B.; EDELMAN, Alan. *Julia: A Fast Dynamic Language for Technical Computing*. 2012. Dostupné z arXiv: 1209.5145 [cs.PL].

25. BEZANSON, Jeff; EDELMAN, Alan; KARPINSKI, Stefan; SHAH, Viral B. Julia: A fresh approach to numerical computing. *SIAM Review*. 2017, roč. 59, č. 1, s. 65–98. Dostupné z DOI: 10.1137/141000671.

26. INNES, Mike. Flux: Elegant Machine Learning with Julia. *Journal of Open Source Software*. 2018. Dostupné z DOI: 10.21105/joss.00602.

27. INNES, Michael; SABA, Elliot; FISCHER, Keno; GANDHI, Dhairya; RUDILOSSO, Marco Concetto; JOY, Neethu Mariya; KARMALI, Tejan; PAL, Avik; SHAH, Viral. Fashionable Modelling with Flux. *CoRR*. 2018, roč. abs/1811.01457. Dostupné z arXiv: 1811.01457.

28. BLAOM, Anthony D.; KIRALY, Franz; LIENART, Thibaut; SIMILLIDES, Yiannis; ARENAS, Diego; VOLLMER, Sebastian J. MLJ: A Julia package for composable machine learning. *Journal of Open Source Software*. 2020, roč. 5, č. 55, s. 2704. Dostupné z DOI: 10.21105/joss.02704.

29. YURET, Julia Deniz. Knet : beginning deep learning with 100 lines of Julia. In: *30th Conference on Neural Information Processing Systems (NIPS 2016)*. Barcelona, Spain, 2016. Dostupné také z: https://api.semanticscholar.org/CorpusID:38785151.

30. MALMAUD, Jonathan; WHITE, Lyndon. TensorFlow.jl: An Idiomatic Julia Front End for TensorFlow. *Journal of Open Source Software*. 2018, roč. 3, č. 31, s. 1002. Dostupné z DOI: 10.21105/joss.01002.

31. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. *Deep Residual Learning for Image Recognition*. 2015. Dostupné z arXiv: 1512.03385 [cs.CV].

Obsah příloh