F3

Fakulta elektrotechnická

Katedra Počítačů

Diplomová práce

Reprezentace problémů klasického plánování pro učení heuristické informace grafovými neuronovými sítěmi

Tomáš Grim

Otevřená informatika

Květen 2023

Vedoucí práce: Ing. Michaela Urbanovská

Poděkování / Prohlášení

Tímto bych chtěl poděkovat své vedoucí Ing. Michaele Urbanovské, jejíž vedení a rady byly velkým přínosem. Také bych chtěl poděkovat své rodině, která mi nejen při psaní práce, ale i po celou dobu studia byla nesmírnou podporou.

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.

V Praze dne 26. 5. 2023

.....

Abstrakt / Abstract

V oblasti umělé inteligence je klasické plánování považováno za důležitou součást výzkumu, která se zaměřuje na vývoj algoritmů a technik pro řešení problémů v deterministickém a plně pozorovatelném prostředí. Práce se věnuje propojení klasické plánování se strojovým učením s pomocí 3 vytvořených grafových reprezentací pro stavy PDDL, Všechny reprezentace jsou použity při učení modelů heuristické funkce pomocí grafových neuronových sítí, jejichž výsledky jsou experimentálně otestovány a porovnány s existujícími řešeními.

Klíčová slova: PDDL, grafové neuronové sítě, klasické plánování

In the field of artificial intelligence, classical planning is considered an important part of research that focuses on the development of algorithms and techniques for solving problems in a deterministic and fully observable environment. The work is dedicated to connecting classical planning with machine learning with the help of 3 created graph representations for PDDL states. All representations are used in learning heuristic function models using graph neural networks, the results of which are experimentally tested and compared with existing solutions.

Keywords: PDDL, graph neural networks, classical planning

Title translation: Representation of classical planning problems for learning heuristic information by graph neural networks

Obsah /

Tabulky / Obrázky

Kapitola 1
Úvod

V oblasti umělé inteligence je klasické plánování považováno za důležitou součást výzkumu, která se zaměřuje na vývoj algoritmů a technik pro řešení problémů v deterministickém a plně pozorovatelném prostředí. Řešením problému klasického plánování je posloupnost akcí, které dokáží přetransformovat počáteční stav na požadovaný cílový stav, za dodržení sady předem definovaných pravidel a omezení.

Pro formální popis problémů se v klasickém plánování využívá standardizovaný jazyk PDDL[1] [1]. Jazyk poskytuje formální syntaxi pro specifikaci akcí, počátečního stavu, cílového stavu a omezení plánovacího problému. Poskytuje tím tak unifikovanou reprezentaci použitelnou nezávisle na problému či doméně pro libovolnou plánovací techniku. Jazyk PDDL byl vytvořen pro první ročník soutěže IPC[2] [2], která se koná od roku 1998. Plánovače jsou v soutěži hodnoceny na obtížných doménách, které často vycházejí z reálných problémů a jsou navrhovány tak, aby odhalovaly problémy a limitace v metodách klasického plánování.

Jednou z klíčových výzev klasického plánování je vývoj efektivní heuristické funkce. Heuristika usměrňuje prohledávací algoritmy tím, že ohodnocuje každý prohledávaný stav. Schopnosti prohledávacích algoritmů jsou tedy silně závislé na zvolené heuristické funkci. Tradičně byly efektivní heuristické funkce vytvářeny pro konkrétní domény, nicméně existuje také několik relativně dobrých heuristik, které jsou doménově nezávislé. Avšak tvorba těchto univerzálních heuristik je velice obtížná a časově náročná.

V posledních letech se objevují aplikace strojového učení v metodách klasického plánování. Jednou z často využívaných technik strojového učení jsou neuronové sítě, které prokázaly své schopnosti v různých oblastech umělé inteligence, jako je zpracování přirozeného jazyka [3] nebo klasifikace obrázků [4]. V kontextu klasického plánování mohou být neuronové sítě využity jako heuristické funkce pro plánovací algoritmy. Jedním z vhodných typů pro práci se strukturovanými daty klasického plánování jsou grafové neuronové sítě.

Cílem této práce je propojit klasické plánování se strojovým učením za pomoci grafové reprezentace stavů nezávislé na doméně. Všechny tři vytvořené reprezentace jsou použity při učení modelů heuristické funkce pomocí grafových neuronových sítí. Natrénované heuristické funkce jsou experimentálně otestovány a porovnány s existujícími řešeními.

[1] Planning Domain Definition Language
[2] International Planning Competition

1.1 Předchozí/Předcházející/Předešlý výzkum

V oblasti klasického plánování a strojového učení proběhlo několik pokusů o jejich propojení. Jednou z prací, propojující tyto dvě oblasti je *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary* [5], ve která je navrhnuta metoda LatPlan. Tato metoda vytváří grafickou podobu plánu nalezeného pomocí klasického plánování. Opačným směrem se zabývala práce *Human-level control through deep reinforcement learning* [6], která navrhuje síť DQN schopnou tvorby strategie pouze na základě informací z herního displeje některé ze 2600 Atari her. Další z možných způsobů, jak propojit dané oblasti je využití neuronové sítě jako heuristiky k vytvoření portfolia [7]. Oblastí klasického plánování, která se přímo vybízí k použití neuronových sítí, je plánování na mřížce. Práce *Learning Generalized Reactive Policies Using Deep Neural Networks* [8] řeší tento problém využitím konvolučního modelu neuronové sítě GRP, který jako vstup používá PDDL reprezentaci a vrací zobecněnou strategii rozhodování.

Jedním z prvních pokusů o vytvoření doménově nezávislého modelu neuronové sítě je práce *ASNets: Deep Learning for Generalised Planning*[9]. Princip ASNets je inspirován konvolučními sítěmi. Síť je tvořena dvěma typy vrstev, které se několikrát navzájem střídají. Prvním typem je akční vrstva, vytvořená z akcí, druhým typem je pak propoziční vrstva, která obsahuje predikáty, akce i predikáty jsou v definici domény. Vrstvy jsou mezi sebou propojeny na základě sdílení stejných objektů problému. Bloky spadající do jedné vrstvy mají stejnou sadu trénovatelných vah, což umožňuje vytvoření jedné sítě pro celou doménu. Výstupem sítě je strategie, jak se rozhodovat v konkrétním stavu. Tento přístup umožňuje vynechat z klasického postupu prohledávací algoritmus. To má za následek ztrátu záruk, které poskytují některé kombinace heuristik a prohledávacích algoritmů. Další nevýhodou tohoto řešení je, že vede k vytvoření velké neuronové sítě i pro relativně malé problémy.

Další prací v této oblasti je *Learning Heuristics for Planning with Hypergraph Networks* [10]. Architektura STRIPS-HGN je kombinací rekurentní a grafové sítě o třech blocích. První blok se stará o zakódování příznaků vstupního hyper-grafu do požadované dimenze. Prostřední a zároveň hlavní blok využívá rekurence pro zvýšení informační hodnoty příznaků. Poslední blok, pak dekóduje natrénované příznaky do výsledného skaláru, který je použit jako heuristická hodnota pro plánovací algoritmus. Stejně jako u ASNets [9] je výsledná neuronová síť relativně mohutná, kvůli vysoké kombinaci akcí a predikátů.

Další pohled, na problematiku využití neuronových sítí v klasickém plánování přinesli autoři práce *Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits* [11]. Ti vycházeli z předpokladu, že většina variant neuronových sítí má stejnou expresivitu, jako formule v predikátové logice prvního řádu o dvou proměnných. Zajímavostí jejich postupu je vytvoření grafu z formulí predikátové logiky. Tyto formule a rozhodující strategie, kterou se má síť naučit, však byly vytvořeny ručně pro testované domény a jejich postup, tak nelze přímo aplikovat na domény zadané jenom v jazyce PDDL.

Kapitola 2
Analýza problému

V práci jsou využívány metody a pojmy ze 4 oblastí. V první sekci jsou zadefinovány pojmy z oblasti grafů, které jsou ve druhé sekci v kombinaci s definicí plánovacího problému použity pro popis heuristických funkcí a plánovacích algoritmů. Následuje sekce s popisem jednotlivých částí jazyka PDDL a Kapitola je zakončena popisem nejdůležitějších částí neuronových sítí.

2.1 Grafy

V informatice jsou grafy velmi často využívány v mnoha různých oblastech. Uplatňují se například při ukládání dat [12], kde díky některým vlastnostem umožňují efektivnější kódování informací. Další oblastí kde se grafy často využívají je plánování. Zde se grafy zase prosadili díky svým univerzálním vlastnostem a schopnosti zaznamenat vztahy mezi objekty. Následující definice budou použity při převodu plánovacího problému do grafové reprezentace.

Definice 2.1 (Neorientovaný graf). *Neorientovaný graf je trojice $G = (V, E, \epsilon)$, kde V je neprázdná konečná množina vrcholů (též zvaných uzlů), E je konečná množina jmen hran a ϵ je přiřazení, které každé hraně $e \in E$ přiřazuje množinu $\{u, v\}$ (kde $u, v \in V$ jsou vrcholy) a nazývá se vztah incidence.*

Definice 2.2 (Paralelní hrana). *Jestliže neorientovaném grafu existují dvě různé hrany e_1, e_2, pro které platí, že $\epsilon(e_1) = \epsilon(e_2)$, říkáme že hrany e_1, e_2 jsou paralelní.*

Definice 2.3 (Smyčka). *Jestliže $\epsilon(e) = \{u, v\}$ pro $u, v \in V$. Je-li $u = v$, říkáme že e je smyčka.*

Definice 2.4 (Paralelní hrany). *Jestliže v orientovaném nebo neorientovaném grafu existují dvě různé hrany e_1, e_2, pro které platí, že $\epsilon(e1) = \epsilon(e_2)$, říkáme, že hrany e_1, $e2$ jsou paralelní.*

Definice 2.5 (Prostý graf). *Graf se nazývá prostý graf, nemá-li paralelní hrany.*

Definice 2.6 (Neorientovaný sled). *Je dán neorientovaný graf. Pak neorientovaný sled je posloupnost vrcholů hran*

$$v_1, e_1, v_2, e_2, \ldots, v_{k-1}, e_{k-1}, v_k$$

taková že hrana e_i je incidentní s vrcholy v_i a v_{i+1} pro všechny $i = 1, 2, \ldots, k-1$.

Definice 2.7 (Tah). *Neorientovaní sled nazýváme neorientovaným tahem, jestliže se v něm neopakují hrany.*

Definice 2.8 (Cesta). *Neorientovaný tah je cestou, jestliže se v něm neopakují vrcholy s tou výjimkou, že může být uzavřený, tj. může být $v_1 = v_k$.*

Definice 2.9 (Úplný graf). *Jedná se o graf, který má nejvíce hran a je prostý a bez smyček.*

Definice 2.10 (Souvislý graf). *Řekneme, že neorientovaný graf je souvislý, jestliže pro každé dva vrcholy u, v grafu existuje neorientovaná cesta z u do v.*

2.2 Klasické plánování

Klasické plánování je důležitá oblast výzkumu spadající pod umělou inteligenci, zabývající se vývojem algoritmů a technik pro řešení problémů v deterministických a plně pozorovatelných doménách. Úlohu klasického plánování lze zadefinovat několika způsoby, nejčastěji jsou používány definice STRIPS a FDR.

Definice 2.11 (Plánovací úloha STRIPS). *Plánovací úloha STRIPS* Π *je 5-tice*

$$\Pi = \langle F, O, s_i, s_g, c \rangle$$

- $F = f_1, f_2, \ldots, f_n$ *je konečná množina faktů*
- $O = o_1, o_2, \ldots, o_m$ *je konečná množina všech operátorů*
- $s_i \subseteq F$ *je počáteční stav obsahující fakty, které platí v počátečním stavu*
- $s_g \subseteq F$ *je cílová podmínka, obsahující fakta, která musejí platit pro každý cílový stav*
- $c(o)\colon o \to \mathbb{R}^+)$ *je ohodnocovací funkce, přiřazují kladné hodnoty operátorům* o

Definice 2.12 (Operátor). *Operátor* o *je 3-jice*

$$o = \langle pre(o), eff(o), add(o) \rangle$$

- $pre(o) \subseteq F$ *je množina podmínek obsahující fakta, která musejí platit, aby bylo možné operátor* o *použít*
- $add(o) \subseteq F$ *je množina faktů, které budou přidány do stavu, po použití operátoru* o
- $del(o) \subseteq F$ *je množina faktů, které budou odebrány ze stavu, po použití operátoru* o

Definice 2.13 (Stav). *Stav* $s = \{f_1, \ldots, f_n\}$ *je konečná množina všech faktů pravdivých ve stavu* s

Definice 2.14 (Stavový prostor). *Stavový prostor* $S = \{s_1, \ldots, s_n\}$ *je konečná množina obsahující všechny stavy.*

Stav je společně se stavovým prostorem velice důležitým konceptem, jelikož dohromady poskytují plánovacím algoritmům důležité informace o problému. Další důležitou částí pro plánovací algoritmy .

Věta 2.1. *Množinu všech aplikovatelných operátorů* $o \in O$ *ve stavu* $s \subseteq S$ *značíme* $O(s) \in O$

Definice 2.15 (Plán). *Plán* π *je konečná sekvence operátorů* $\pi = o_0, \ldots, o_n$*, která generuje sekvenci stavů* $s_0, \ldots, s_{n+1}$*, takovou že* $pre(o_i) \subseteq s_i$ *a* $s_{i+1} = (s_i \setminus del(o_i)) \cup add(o_i)$ *a* $s_{n+1} \in s_g$*.*

Věta 2.2 (Délka plánu). *Délka plánu $\pi = o_0, \ldots, o_n$ je rovna počtu provedených akcí, tedy $|\pi| = n$. O plánu π řekneme že je optimální, pokud neexistuje žádný jiný plán π', který je kratší, tedy $|\pi| < |\pi'|$.*

Věta 2.3 (Cena Plánu). *Cena plánu $\pi = o_0, \ldots, o_n$ je rovna součtu cen provedených akcí*

$$c(\pi) = \sum_{i=0}^{n} c(o_i).$$

2.2.1 Heuristické funkce

Smyslem heuristických funkcí je pomoci plánovacímu algoritmu, efektivněji prohledávat velké stavové prostory. Heuristická funkce přiřazuje hodnoty stavům, které umožňují algoritmům využívat různé strategie prohledávání.

Definice 2.16 (Heuristická funkce). *Heuristická funkce $h(s)$: $s \to \mathbb{R}$ mapuje stav s na reálnou hodnotu.*

Definice 2.17 (Optimální heuristika). *O heuristice řekneme, že je optimální, pokud $h(s) = |\pi|$ pro $\forall s \in S$ a π je optimální plán z s. Značí se $h^*(s)$.*

Definice 2.18 (Přípustná heuristika). *Heuristická funkce h je přípustná, pokud $0 < h(s) < h^*(s)$, pro $\forall s \in S$.*

Výpočet heuristiky není jednoduchá záležitost a v některých případech může být hlavním důvodem pomalého prohledávání stavového prostoru. Například optimální heuristika (2.17) se v praxi vůbec nevyužívá, jelikož pro její výpočet je potřeba znát řešení problému. Při tvorbě heuristické funkce je tak potřeba dbát na rovnováhu mezi časem stráveným při jejím výpočtu a informační hodnotou kterou může poskytnout.

Většina moderních heuristik funguje na relaxačním principu. Tento princip zjednodušuje STRIPS plánovací úlohu 2.11 tím, že místo operátorů 2.12 využívá jejich relaxovanou variantu. Akce tedy neobsahují množinu faktů $del(a)$, jejichž hodnota by normálně byla po použití akce změněna na nepravdu.

HMax

h_{max} heuristika je konzistentní heuristikou, což z ní dělá vhodného kandidáta pro spoustu plánovacích algoritmů. Heuristika řeší relaxovanou úlohu a jejím výsledkem je cena nejdražšího faktu nutného pro splnění cíle. Pro zjištění cen faktů v cílovém stavu je možné použít libovolného algoritmu pro zjištění nejkratší cesty.

Algorithm $h_{max}(start, goal)$:
    $costsList$ = dictionary with $cost$ of all possible $predicates$ set to ∞
    $state$ = empty set \\ Set will continualy grow, allowing the use of new actions
    for each $fact$ **in** $start$:
        set $cost$ of $fact$ to 0
        add $fact$ to $state$
    while $goal$ **not in** $state$:
        for each new applicable $action$ from $state$: \\ Only unused actions
            $actionCost$ = cost of $action$
            $maxCost$ = highest cost of fact from $pre(action)$

```
        for each fact in add(action):
            add fact to state
            if actionCost + maxCost < costList[fact]:
              costList[fact] = actionCost + maxCost  \\ Cheaper way to get the fact
    return value of highest costing goal fact from costList
```

FF heuristika

Fast-Forward heuristika je relaxovanou heuristikou, pocházející ze stejnojmenného plánovače. Narozdíl od h_{max} , h_{ff} není přípustnou heuristikou, proto se nepoužívá pro optimální plánování. V případě, že je cílem nalezení libovolného řešení, poskytuje h_{ff} velmi kvalitní informační hodnotu prohledávaném stavu. Pro výpočet heuristiky je potřeba vytvořit graf dosažitelnosti, ve kterém se od konce označují akce a fakta podle sady pravidel. Výsledkem h_{ff} heuristiky je počet označených akcí.

```
Algorithm Reachability graph(start, goal):
    G_R = empty graph                                          \\ Reachability graph
    state = empty set          \\ Set will continualy grow, allowing the use of new actions
    currList = empty list                                      \\ List storing current level verticies
    for each fact in start:
        add fact to state
        create vertex v form fact and add it to G_R
        add v to currList
    while goal not in state:          \\ Creates graph with changing fact and action layers
        \\ Fact layer
        nextList = emtpy list                                  \\ List storing next level verticies
        for each fact in state:
            create vertex u and add it to G_R
            add u to nextList
            create edge e between u and vertex corresponding to fact from currList
            add e to G_R
        \\ Action layer
        for each applicable action from state:
            create vertex v from action and add it to G
            add v to actionList
            for each fact in pre(action):
                create edge e between v and vertex corresponding to fact from currList
                add e to G
            for each fact in add(action):
                if fact not in state:
                    create vertex w and add it to G
                    add w to nextActionList
                    add fact to state
                create edge between v and vertex corresponding to fact from nextList
    return G_R
```

Algorithm $h_{ff}(G_R, goal)$:
\\ Action vertex is justified when all precondition fact varticies are marked
\\ Fact vertex is justified when at least one immediate predecessor vertex is marked
mark all verticies corresponding to facts from *goal*
until all marked verticies are justified \\ Marking procedure is applied layer by layer
1)mark all immediate predecessor of marked unjustified *action* vertex
2)mark all immediate predecessor of marked unjustified $fact$ vertex
with only one immedeate predecessor
3)mark all immediate predecessor of marked unjustified $fact$ vertex
connected via and idle arc
4)mark any immediate predecessor of marked unjustified $fact$ vertex
return number of marked action verticies

2.2.2 Plánovací algoritmy

Algoritmy používané pro nalezení řešení v klasickém plánování většinou pocházejí z teorie grafů a jsou nejčastěji založeny na prohledávání stavového prostoru. Díky zvolené definici STRIPS plánovací úlohy 2.11 není přechod mezi grafem a plánovacím problémem příliš složitý a odpovídá intuici. Jednotlivé stavy se stanou vrcholy grafu a přechodová funkce určuje jakým směrem mají být z akcí vytvořené orientované hrany.

Plánovací algoritmy můžeme rozdělit podle způsobu prohledávání. Zde je výčet pouze několika nejznámějších algoritmů [13].

- Neinformované algoritmy
  - Prohledávání do šířky (BFS)
  - Prohledávání do hloubky (DFS)
  - Iterativní prohledávání do hloubky (IDS)
- Informované algoritmy
  - Greedy best first search (GBFS)
  - A*

I přes zajímavé záruky, které poskytují některé neinformované algoritmy, jako je například optimalita plánu při prohledávání pomocí BFS algoritmu za předpokladu existence cíle a jednotkové ceny akce [13], je jejich využití v praxi dosti omezené. Hlavní nevýhodou je, že velmi často potřebují pro nalezení řešení prozkoumat velkou část stavového prostoru. Na reálných problémech bývají stavové prostory tak velké, že se jejich prozkoumávání s pomocí neinformovaných algoritmů přestává být výpočetně proveditelné.

Naštěstí informované algoritmy jsou při prohledávání stavového prostoru výrazně efektivnější. Jejich síla spočívá ve využití dodatečné informace poskytnuté heuristickou funkcí. Pokud je heuristika přípustná 2.18, pak například A-star algoritmus vrací optimální plán [13].

Greedy best first search

Greedy Best First Search (GBFS) je jedním z nejrozšířenějších informovaných algoritmů, využívajících se ke hledání plánů, které nemusí být optimální[13]. Jedna z jeho výhod je, že nepotřebuje dopředu znát jak vypadá celý stavový prostor a graf který jej reprezentuje vytváří dynamicky. Algoritmus pracuje se dvěma seznamy, do prvního tzv. otevřeného seznamu ukládá nenavštívené stavy a do druhého tzv. uzavřeného seznamu ukládá již prozkoumané stavy. Algoritmus postupně vybírá stavy z otevřeného seznamu s nejnižší heuristickou hodnotou. Pro každý vybraný stav zjistí sousední stavy za pomoci přechodové funkce a uloží je do otevřeného seznamu. Následně vybraný stav přesune do uzavřeného seznamu. Algoritmus skončí ve chvíli, kdy vybere cílový stav z otevřeného seznamu nebo mu dojdou stavy k prohledávání. Pro nalezení výsledného plánu je nutné, aby si každý stav pamatoval svého předchůdce. Algoritmus GBFS využívá pro zvýšení efektivity, namísto seznamů otevřeného seznamu prioritní frontu a namísto uzavřeného setu využívá množinu.

```
Algorithm GBFS(start, goal):
    openList = empty priority queue // Nodes are stored based on their heuristic values
    closedList = empty set                                  // The set stores visited nodes
    add start to OpenList
    while openList not empty:
        current = remove the highest priority node from openList
        if current == goal:
            return path from start to current
        if current in closedList:
            continue                                  // The current node was already visited
        add current to closedList                       // Marks the current node as visited
        for each neighbor of current:
            if neighbor not in closedList:
                calculate the heuristic value h for neighbor
                add neighbor to openList with priority h
    return failure                                                  // In case goal isn't found
```

2.3 PDDL

PDDL je zkratka pro Planning Domain Definition Language, což je formální jazyk používaný k popisu a specifikaci plánovacích problémů v oblasti umělé inteligence a automatizovaného plánování. Slouží jako standardizovaný způsob reprezentace plánovací domény a souvisejících plánovacích problémů.

PDDL byl poprvé představen v roce 1998 [1] pro první ročník International Planning Competition (IPC) [2] a od té doby se stal široce používaným jazykem pro popis plánovacích problémů. PDDL sdílí mnoho podobností s několika předešlými dalšími jazyky z klasického plánování, jako je ADL [14] a UCPOP [15].

Samotný jazyk je navržen tak, aby byl čitelný pro člověka, ale aby byl srozumitelný pro počítače. PDDL poskytuje jasný a strukturovaný způsob, jak reprezentovat problémy v klasickém plánování, což usnadňuje vývoj a sdílení plánovacích modelů a algoritmů.

2.3.1 Doména

Doména je „univerzální“ částí plánovacího problému, která definuje svět, jeho pravidla a akce společné pro sadu problémů. Každý prvek domény má své klíčové slovo, následované argumenty uzavřenými do závorek.

Název domény

Každý soubor s definicí domény vždy začíná názvem a je nutné zkontrolovat zdali se shoduje s názvem domény v definicí problému. Bohužel mnoho plánovačů tento krok přeskočí a může se stát, že při nešikovném nastavení, plánovače nakonec řeší jinou úlohu.

Rozšíření

Klíčové slovo `:extends` umožňuje doméně zdědit definice a deklarace z rodičovské domény. Využití je v však v praxi velice nízké a proto mnoho plánovačů nepodporuje tuto funkci.

Požadavky

`:requirements` je svým chováním podobné příkazům „import/include“ známých z programovacích jazyků, protože je ale PDDL deklarativním typem jazyka, tak klíčové slovo `:requirements` „požaduje“ od plánovače podporu nějakého rozšiřujícího aspektu jazyka. Lze využít hned několika různých rozšíření najednou, jména těchto rozšíření pak stačí napsat za `:requirements` oddělené mezerou.

Typy objektů

Ačkoliv se jedná o rozšíření, jeho podpora je u plánovacích algoritmů tak široká, že by se dalo považovat za součást jazyka PDDL. Klíčové slovo `:types` umožňuje vytvoření základních typů, podtypů a jejich hierarchii. Tyto struktury se pak dají využít pro další obecné nebo specifické definice.

Konstanty

Klíčové slovo `:constants` je samo o sobě dost vypovídající a umožňuje vytvoření konstant které jsou přítomné ve všech problémů z dané domény.

Predikáty

Jedna z nejdůležitějších částí domény je označené klíčovým slovem `:predicates`. V této části jsou deklarace všech obecných i konkrétních predikátů. Predikáty mohou být aplikovány na objekty konkrétního typu nebo na všechny objekty, záleží zdali bylo využito rozšíření `:types`. Stejně jako v predikátové logice prvního řádu predikáty mohou být pravda nebo nepravda a jejich hodnota může být v průběhu řešení problému změněna. Většina predikátů v jazyce PDDL obsahuje pouze jeden nebo dva argumenty. Není to však podmínkou a predikáty tak mohou teoreticky mít spočetně mnoho argumentů. Predikát popisuje vztahy nebo vlastnosti jeho argumentů.

Akce

Druhá nejdůležitější část domény je označená klíčovým slovem `:action`. Akce definují možné přechody mezi jednotlivými stavy problému. Každá akce má tři rozlišné části:

- **Parametry** `(:parameters)` - deklarují objekty (mohou být typované) na kterých je akce prováděna.
- **Předpoklady** `(:preconditions)` - obvykle obsahují sérii konjunkcí a disjunkcí predikátů, která musí být splněná, aby akce mohla být aplikována. Splnění předpokladů však ještě neznamená že musí být využita.
- **Efekty** `(:effect)` - obsahují logický výraz složený z konjunkcí predikátů, které určují jaké hodnoty predikátů mají být změněny, v případě provedení akce.

2.3.2 Problém

Pod označením problém se v jazyce PDDL myslí konkrétní úloha s přesně definovanými objekty, cíli a iniciálním stavem. V kombinaci s doménou 2.3.1 tak tvoří instanci plánovacího problému.

Název problému

Stejně jako doména i problém začíná svým názvem, jenž slouží hlavně jako unikátní identifikátor.

Název domény

Hned za názvem problému, je název domény, se kterou by měl být daný problém spárován pro správný průběh plánovacího algoritmu.

Situace

Klíčové slovo `:situation` funguje podobně jako `:exends` (2.3.1) a to ve smyslu, že zdědí definice rodičovského problému. V některých případech mohou mít všechny problémy velké množství společných prvků, které mohou být zadefinovány v samostatném souboru a snížit tak paměť zabíranou problémem. Využití tohoto klíčového slova není mezi dnešními plánovači příliš časté, jelikož skok v technologických možnostech je od představení PDDL obrovský.

Objekty

Blok označený klíčovým slovem `:objects` je místem, kde jsou definovány konkrétní objekty, na kterých je problém řešen. Každý objekt musí mít unikátní jméno a může být typovaný. Objekty se používají jako argumenty v predikátech.

Počáteční stav

Klíčové slovo `:init` označuje část problému, kde je zadefinován jeho počáteční stav. Obsahuje seznam predikátů aplikovaných na objekty, která v počátečním stavu platí. Hodnoty ostatních predikátů jsou považovány za nepravdu.

Cíl

Cíl plánovacího problému je zadefinován pomocí klíčového slova `:goal` a logického výrazu, obsahujícího predikáty, které musí být splněny, aby daný stav mohl být považován za cíl. To znamená, že ve výraz musí obsahovat všechny predikáty jejich hodnota musí být pravda, ale také i všechny predikáty jejich hodnota musí být nepravda.

2.3.3 Verze PDDL

V sekci 2.3.1 a 2.3.2 je popsaná verze PDDL 1.2, která byla použita v prvním roce IPC [2]. Přes veškerou univerzálnost a expresivitu této verze byl popis některých úkolů nepraktický a bylo potřeba provést určité úpravy jazyka.

Verze 2.1

Tato verze [16] byla použita ve třetím běhu IPC v roce 2002. Oproti verzi 1.2 má lepší podporu pro časové a numerické problémy. Oba typy těchto problémů bylo možné zapsat ve verzí 1.2, ale pouze za pomocí výpisu všech možných čísel, jako predikátů a operací nad nimi jako akcí. Nepraktičnost takového zápisu byla zřejmá a nikdy nebyla použita v plánovacích algoritmech v reálném světě. Z tohoto důvodu přidává verze 2.1 do `:requirements` tři důležité rozšiřující části:

- `:fluents`
- `:durative-actions`
- `:continuous-effects`

Všechny tyto požadavky pomáhají zjednodušit zápis v úkolech časového plánování.

Verze 2.2

Tato verze [17] nepřinesla mnoho změn, jen u některých částí o něco lepší syntaxi a jeden nový požadavek:

- `:timed-initial-literals`

Tento požadavek umožňuje aby byl predikát platný pouze na určitém časovém intervalu.

Verze 3.0

S tím, jak se klasické plánování rozšiřovalo na problémy s většími a složitějšími úkoly, tak způsob jakým byly zadávány cíle přestal být postačující. Změna přišla rok před 5. IPC [18]. Tato verze představila dva důležité koncepty:

- `:constraints`
- `:preferences`

`:constraints` jsou sadou omezujících podmínek, jež je složená z konjunkcí predikátů, které musí být splněné po celou dobu plánu. To napomáhá plánovacím algoritmům snížit počet prohledávaných stavů. `:preferences` jsou na druhé straně logické výrazy, které umožňují „měkká omezení“ ve smyslu, že jejich splnění není nutné, ale může být penalizováno v metrikách plánu.

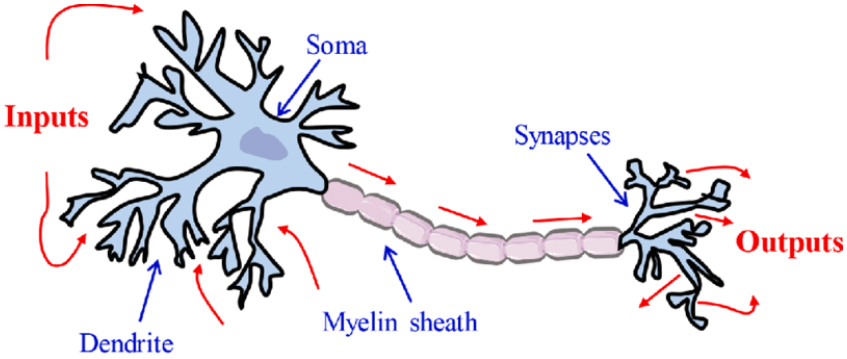
Verze 3.1

Nejnovější verze PDDL byla představena při 6. ročníku IPC [19] v roce 2008 a stále je využívána. Jediné změny, které tato verze přinesla, byly malé syntaktické změny v požadavku `:fluents`.

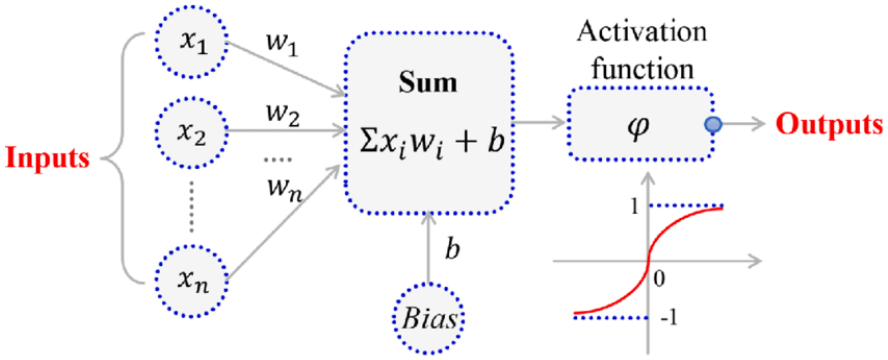
2.4 Neuronové sítě

Neuronové sítě jsou výpočetní modely strojového učení, které se dnes považují nedílnou součást umělé inteligence. S pokrokem ve výpočetní technice získali neuronové sítě na popularitě díky skvělým výsledkům v oblastech zpracování obrazu[4], zpracování přirozeného jazyka [3] a datové analýzy.

Jejich návrh [20] byl v polovině minulého století inspirován tehdejšími znalostmi fungování lidského mozku. Podobnost mezi lidským neuronem a umělým neuronem je vidět na 2.1. První fungující neuronovou sítí byl Perceptron v roce 1958[20], obsahující pouze jednu vrstvu o jednom neuronu. Nedostatky které měli jednovrstvé sítě byly známy již při jejich vzniku, ale trénování vícevrstvých sítí bylo mimo výpočetní schopnosti dobových počítačů.



a) lidský neuron



b) umělý neuron

Obrázek 2.1. Porovnání lidského a umělého neuronu převzatý z [21]

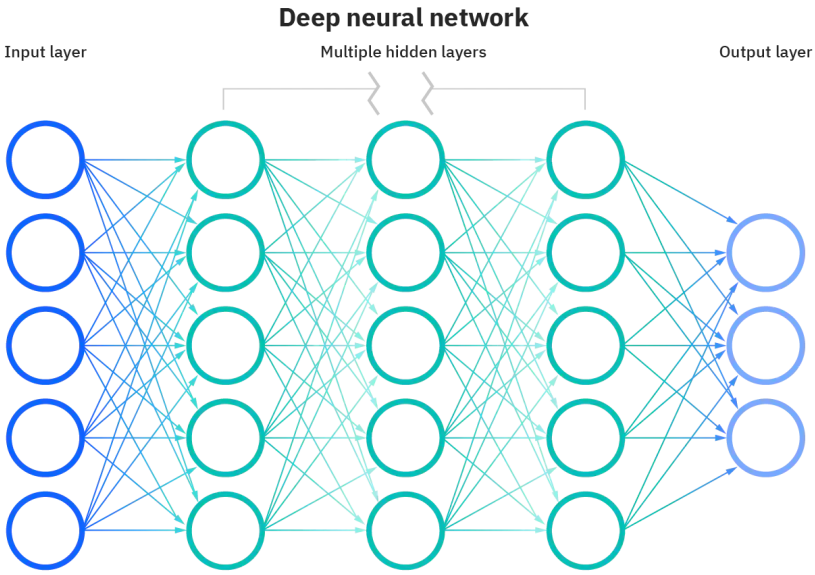
Změna začala v roce 1986 s popularizací backpropagace [22], která se v kombinací s metodou stochastického gradientního sestupu stala nejčastějším způsobem trénování neuronových sítí. I přes možnost trénování vícevrstvých sítí však panovalo přesvědčení,

že hlubší neuronové sítě přinášejí pouze zmenšující se zlepšení. Tento pohled na věc se výrazně změnil v roce 2012 kdy síť nazvaná AlexNet [4] vyhrála soutěž ImageNet.

2.4.1 Průchod sítí

Neuronové sítě jsou vetšinou tvořeny velkým počtem propojených neuronů, ale na rozdíl od lidského mozku jsou neurony uvnitř neuronových síti uspořádány do jednotlivých vrstev. To umožňuje efektivní výpočet výstupu několika neuronů najednou. Každá neuronová síť se skládá ze tří částí:

- **Vstupní vrstva** - první vrstva neuronů, jednotlivé neurony jsou závislé na velikosti vstupních dat
- **Skryté vrstvy** - obsahuje většinu vrstev neuronové sítě, jejich počet ani velikost není závislá na vstupu ani výstupu [1]
- **Výstupní vrstva** - poslední vrstva neuronů, počet neuronů bývá určen problémem který síť řeší



Obrázek 2.2. Ukázka neuronové sítě s lineárními vrstvami.

Každý neuron obsahuje sadu unikátní vah (parametrů), které je jsou při trénování upravovány, tak aby bylo dosaženo požadovaného výsledku. Při výpočtu výstupu jednotlivého neuronu je nejdříve spočítán vážený součet vstupů x, ke kterému je přičten „bias“[2] b, následně je použita aktivační funkce φ,

$$y = \varphi\left(\sum_{i=1}^{n}(x_i \cdot w_i) + b\right).$$

Aktivační funkce φ musí být nelineární funkcí, jinak by šlo výpočet výstupu zřetězených neuronů nahradit jednou sadou vah.

[1] Skryté se jim říká kvůli tomu, že pokud se budeme koukat na neuronovou síť jako na celek a budeme sledovat její vstup a výstup, tak neexistuje způsob jakým určit jaký typ ani kolik těchto vrstev síť obsahuje.
[2] Jedná se další trénovatelný parametr

Lineární vrstva

V případě že je každý neuron ve vrstvě k spojený[3]Spojení se nazývá bipartitní] se všemi neurony z předešlé vrstvy $k-1$, tak se vrstvě k říká lineární vrstva. Výpočet výstupu lineární vrstvy o n neuronech je možné vektorizovat.

$$\vec{y} = \phi(\vec{x}W + \vec{b})$$

- $\vec{y} = [y_1, y_2, \ldots, y_n]$ je vektor výstupů vrstvy k
- $\vec{x} = [x_1, x_2, \ldots, x_m]$ je vektor vstupů vrstvy k/výstupů vrstvy $k-1$
- $W = [\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_n]$ je matice vytvořená spojením vektorů vah, $W \in \mathbb{R}^{n \times m}$
- $\vec{b} = [b_1, b_2, \ldots, b_n]$ je vektor obsahující „bias“ jednotlivých neuronů
- ϕ je vektorová varianta aktivační funkce

2.4.2 Backpropagace

Při trénování neuronové sítě je po dopředném výpočtu její výstup porovnán se správnými hodnotami pomocí ztrátové funkce. Výsledek ztrátové funkce je použit jako odrazový bod pro zahájení backpropagace, jež postupně počítá parciální derivace vzhledem ke každému parametru. Hodnota parciálních gradientů ukazuje, jakým dílem přispěl daný parametr k výstupu sítě. Efektivita počítání parciálních derivací je dosažena za využití řetízkového pravidla, které je aplikováno v každé vrstvě. Řetízkové pravidlo umožňuje využití parciálních derivací vypočtených pro předešlé vrstvy. [22]

2.4.3 Stochastický gradientní sestup

Cílem tréninku neuronových sítí je dosažení co nejlepších výsledků. Jedním ze způsobů jak toho lze dosáhnout je minimalizace hodnoty ztrátové funkce. Ztrátová funkce je v podstatě metrika určující jak dobré nebo špatné řešení síť vrátila. Nejčastěji používaným způsobem minimalizace jsou metody založené na gradientním sestupu. Tyto metody iterativně upravují váhy neuronů pomocí vypočteného gradientu pronásobeného velikostí učícího kroku. Stochastický gradientní sestup se od normálního gradientního sestupu liší pouze tím, že místo výpočtu gradientu na celém trénovacím datasetu, spočítá gradient pouze na jeho části.

Algorithm Stochastic Gradient Descend($model$,α,$dataset$,$\mathcal{L}$):
Initialize $model$ weights w_0, set $t = 0$
until stopping criterion is met
1)Sample a mini-batch of data samples and their labels from $dataset$
2)Compute forward pass with loss $\mathcal{L}$
3)Estimate gradient $\hat{g}_t$ on mini-batch using backpropagation
4)Update weights $w_{t+1} = w_t - \alpha \cdot \hat{g}_t$; set $t = t + 1$ \\ α is learning step

[3] [

Adam

Adam[23] je SOTA gradientní metodou, která zrychluje konvergenci k minimální hodnotě ztrátové funkce. Algoritmus vznikl se značnou inspirací ve dvou dalších populárních gradientních metodách, AdaGrad [24] a RMSProp [25]. První výhodou kterou Adam má oproti SGD , je využití mementu podobně jako je tomu RMSProp. Druhou výhodu získává z algoritmu AdaGrad, což je schopnost upravovat učící krok pro jednotlivé váhy. Tyto výhody z něj dělají jeden z nejčastěji využívaných algoritmů pro optimalizaci ztrátové funkce.

Algorithm Adam($model$,α,β,γ,$dataset$,$\mathcal{L}$):
Initialize $model$ weights w_0, set $t = 0$, $m_0 = 0$, $v_0 = 0$ \\ m and v are moments
until stopping criterion is met
1)Sample a mini-batch of data samples and their labels from $dataset$
2)Compute forward pass with loss $\mathcal{L}$
3)Estimate gradient $\hat{g}_t$ on mini-batch using backpropagation
4)Compute 1st and 2nd momentum
$m_t = \beta \cdot m_{t-1} + (1 - \beta) \cdot \hat{g}_t$
$v_t = \gamma \cdot v_{t-1} + (1 - \gamma) \cdot \hat{g}_t^2$
5)Update weights $w_{t+1} = w_t - \alpha/(\sqrt{v}_t + \epsilon) \cdot \hat{g}_t$, \\ α is learning step
set $t = t + 1$

2.4.4 Grafové neuronové sítě

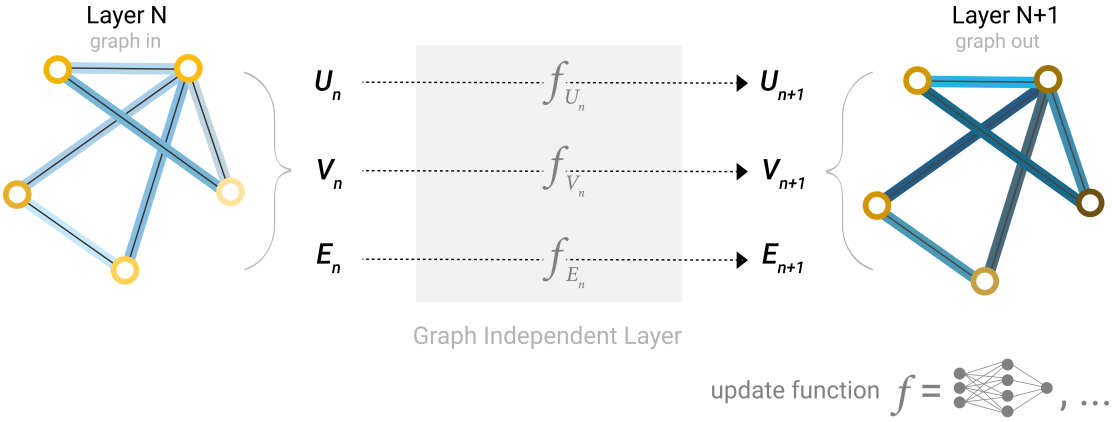
Grafové neuronové sítě (GNN) jsou typem neuronových sítí speciálně navržených pro práci s grafově strukturovanými daty. Zatímco u tradičních architektur neuronových sítích je vstupem vektor nebo matice, tak u GNN je vstupem graf skládající se z vrcholů a hran. Každý vrchol obvykle představuje nějaký konkrétní prvek se sadou vlastností a hrana zachycuje vztah mezi dvěma prvky[4]. Problémem klasický neuronových sítí je se zachycení grafové struktury pomocí vektorů a matic. Hlavní překážkou je isomorfismus grafů, jelikož isomorfní grafy reprezentují stejná data a tedy i výstup neuronové sítě byl měl být stejný.

Grafové neuronové sítě pracují s informacemi zakódovanými na třech úrovních (grafové, hranové, vrcholové). GNN představují transformaci na všech třech úrovních, bez změny struktury grafu. Výsledkem sítě je tak opět graf s nově naučenými se vlastnostmi kterékoliv úrovně. Princip na kterém jsou grafové sítě založeny, je postupná transformace vstupních informací v uvnitř jednotlivých vrstev. Důvodem proč jsou GNN grafově invariantní je sdílení aktualizační funkce[5]. Uvnitř jedné vrstvy aktualizační funkce využívá jedné sady vah pro všechny prvky dané úrovně. Na obrázku 2.3 je vidět návrh jednoduché vrstvy GNN.

Grafové konvoluční sítě

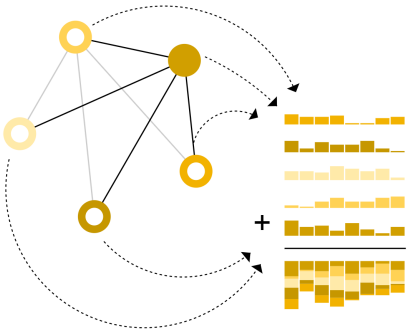
Grafové konvoluční sítě (GCN) jsou velice podobné klasickým konvolučním sítím (CNN). Zatímco CNN pracují s mřížkově reprezentovanými daty, kde má každý bod stejný počet sousedů a je možné využít konvoluční jádra s fixní velikostí, tak v grafech

[4] Hrany mohou také mít sadu vlastností
[5] Může se jednat o jakoukoliv diferencovatelnou funkci, která obsahuje naučitelné parametry, nejčastěji je pak používán MLP.

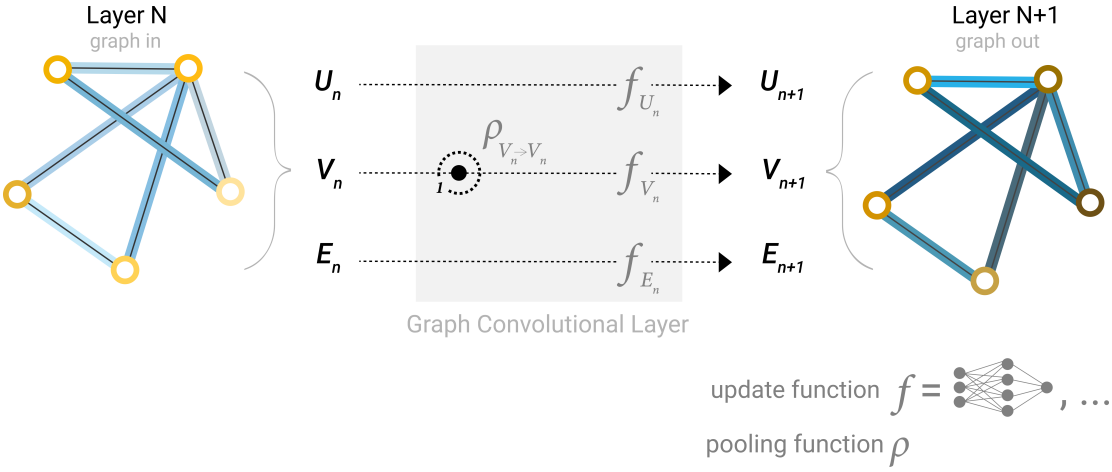


Obrázek 2.3. Návrh jednoduché GNN vrstvy [26]

předpoklad o fixním počtu sousedů neplatí. Proto se v GCN používá sdružování vlastností sousedních prvků před aplikováním aktualizační funkce (viz obr. 2.4). Nejčastěji používanými funkcemi pro agregaci je průměrování, výběr maxima nebo součet. Návrhy složitějších konvolučních vrstev mohou využívat princip sdružování informací sousedních prvků i mezi jednotlivými úrovněmi. V takovém případě je potřeba nezapomenout, že vlastnosti prvků na různých úrovních mohou mít různé velikosti.



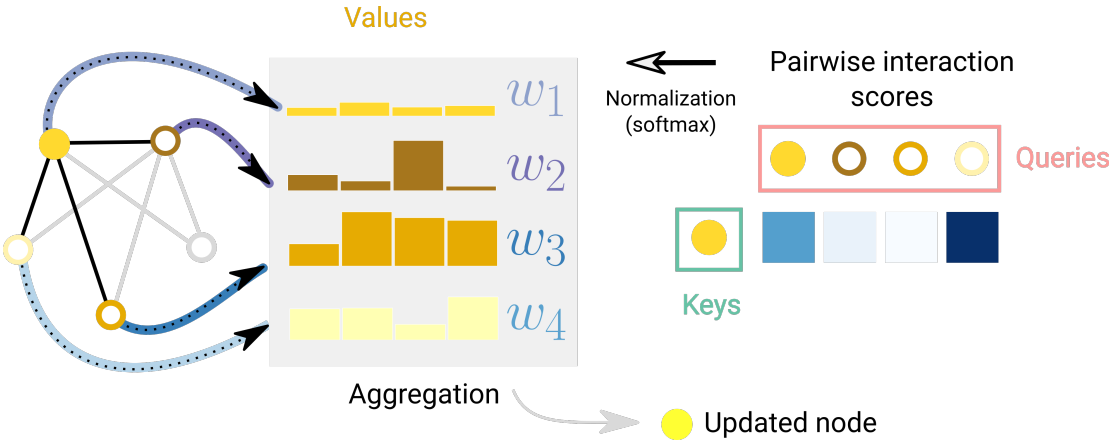
Obrázek 2.4. Ilustrace sdružení vlastností sousedních vrcholů.[26]



Obrázek 2.5. Návrh jednoduché GCN vrstvy využívající konvoluci pouze na úrovni vrcholů.[26]

Grafová attention síť

Graph attention networks (GAT) [27] jsou dalším populárním typem grafových neuronových sítí, které rozšiřují myšlenku konvoluce o vážení sdružovaných sousedních informací. Pro realizaci této myšlenky je potřeba využít sofistikovanějšího mechanismu než je vážený součet, jelikož ten nezachovává grafovou invarianci. V GAT vrstvě je to vyřešeno pomocí tzv. „attention“ mechanismu viz.(2.6), který využívá bodovací funkci. Hodnotu funkce lze interpretovat jako míru relevance dvou prvků. Často se za bodovací funkci volí skalární součin. Vypočtené váhy „attention“ mechanismu je ještě dobré normalizovat například softmax funkcí, čímž se posílí váha relevantnějších prvků.



Obrázek 2.6. Ilustrace attention mechanismu.[26]

Kapitola 3
Navrnuté řešení

Tato kapitola obsahuje rozbor dílčích řešení, použitých pro splnění zadání diplomové práce. Na začátku kapitoly je v sekci 3.1 popsán výběr domén klasického plánování a generování vhodných trénovacích dat. Následuje sekce 3.2 s popisem grafové reprezentace stavu PDDL problému. V další sekci 3.3 se pak nachází popis modelu vytvořené grafové neuronové sítě a kapitola je zakončena sekcí 3.3.1 zabývající se výběrem ztrátové funkce.

3.1 Vybrané domény

Omezení při výběru vhodných domén bylo hned několik. Prvním bylo využití pouze domén reprezentovaných jazykem PDDL, jak je popsáno v sekci 2.3.1. Díky jeho vysoké popularitě má většina domén svou vlastní PDDL reprezentaci. Dále vybrané domény nesměly obsahovat predikáty třetího a vyššího řádu. Toto omezení umožnilo vytvoření grafové reprezentace popsané v sekci 3.2 a lze jej odůvodnit tím, že člověk při popisu problému intuitivně přiřazuje objektům vlastnosti[1] a vytváří vztahy[2] pouze mezi dvojicemi objektů.

Složitost domény byla jedním z hlavních faktorů, který ovlivnil finální výběr. Doména musela poskytovat dostatečný počet jednoduchých problémů pro vygenerování trénovací množiny a zároveň musela být dostatečně obtížná, aby na ní mohlo být provedeno porovnání mezi použitými přístupy. Kvůli možnosti srovnání byly využity pouze domény, které se alespoň jednou objevily na soutěži IPC.

Tvorba trénovacího datasetu začínala vygenerováním dostatečného počtu jednoduchých problému v dané doméně. Na každém vygenerovaném problému bylo spuštěno prvotní prohledávání do šířky, při kterém byl ukládán prohledávací strom. Využití BFS algoritmu bylo kompromisem mezi snahou o využití celého stavového prostoru a nižším počtem prohledaných stavů vytvořený informovanými algoritmy. Stavy z prohledávacího stromu byly následně použity ke spuštění sekundárního prohledávání za pomocí A* algoritmu a h_{max} heuristiky. Cílem sekundárního prohledávání bylo vytvořit množinu dvojic $(stav, hodnota)$, kde hodnota odpovídala délce optimálního plánu mezi stavem a cílem problému.

3.1.1 Gripper

První vybranou doménou je Gripper, která obsahuje 7 predikátů a 3 akce pomocí kterých popisuje problém přemístění míčků za pomoci robotické ruky z jedné místností do jiné. Doména byla představena již při prvním ročníku IPC [2], i přes možnost vytvořit problémy s více roboty a místnostmi je v problémech použitých při IPC využito pouze

[1] predikát 1. řádu
[2] predikát 2. řádu

Algorithm Dataset Generator(*domain*):
  generate sufficient number of *problems*
  dataset = empty set
  for each *problem*:
    get *goal* from *problem*
    use BFS to solve *problem* and save *searchTree* \\ Initial search
    for each *state* **in** *searchTree*:
      use A-star to find *solution* of *problem* from *state* to *goal* \\ Secondary search
      add (*state*, *solution*) to *dataset*
  return *dataset*

```
(define (domain gripper)
    (:predicates (room ?r)
                 (ball ?b)
                 (gripper ?g)
                 (at-robby ?r)
                 (at ?b ?r)
                 (free ?g)
                 (carry ?o ?g))

    (:action move
        :parameters (?from ?to)
        :precondition (and (room ?from) (room ?to) (at-robby ?from))
        :effect (and (at-robby ?to) (not (at-robby ?from))))

    (:action pick
        :parameters (?obj ?room ?gripper)
        :precondition (and (ball ?obj) (room ?room) (gripper ?gripper)
                           (at ?obj ?room) (at-robby ?room) (free ?gripper))
        :effect (and (carry ?obj ?gripper) (not (at ?obj ?room))
                     (not (free ?gripper))))

   (:action drop
       :parameters (?obj ?room ?gripper)
       :precondition (and (ball ?obj) (room ?room) (gripper ?gripper)
                          (carry ?obj ?gripper) (at-robby ?room))
       :effect (and (at ?obj ?room) (free ?gripper)
                    (not (carry ?obj ?gripper)))))
```

2 robotů a 2 místností. Vygenerované problémy tyto umělá omezení dodržují a liší se tedy pouze v počtu míčků. Pro vytvoření trénovacího datasetu byly použity problémy s 1 až 9 míčky, dohromady tyto problémy vygenerovaly 26548 unikátních datových bodů.

3.1.2 Blocksworld

Druhou zvolenou doménou je Blocksworld, která popisuje problém poskládání kostek do věže v určitém pořadí. Doména existuje ve dvou variantách, první z nich obsahuje 3

akce a 3 predikáty, v tomto podání se jedná o velmi jednoduchou úlohu. Složitější verze představená na druhém ročníku IPC, obsahuje 5 predikátů a 4 akce. Pro tuto práci byla zvolena složitější verze z důvodu většího stavového prostoru. Obtížnost jednotlivých problémů je dána především počtem kostek. Pro trénování je využito problémů se 3 až 7 kostkami. Dohromady bylo vygenerováno 25 problémů, 5 pro každou velikost, které vytvořili dataset o 40163 trénovacích bodech.

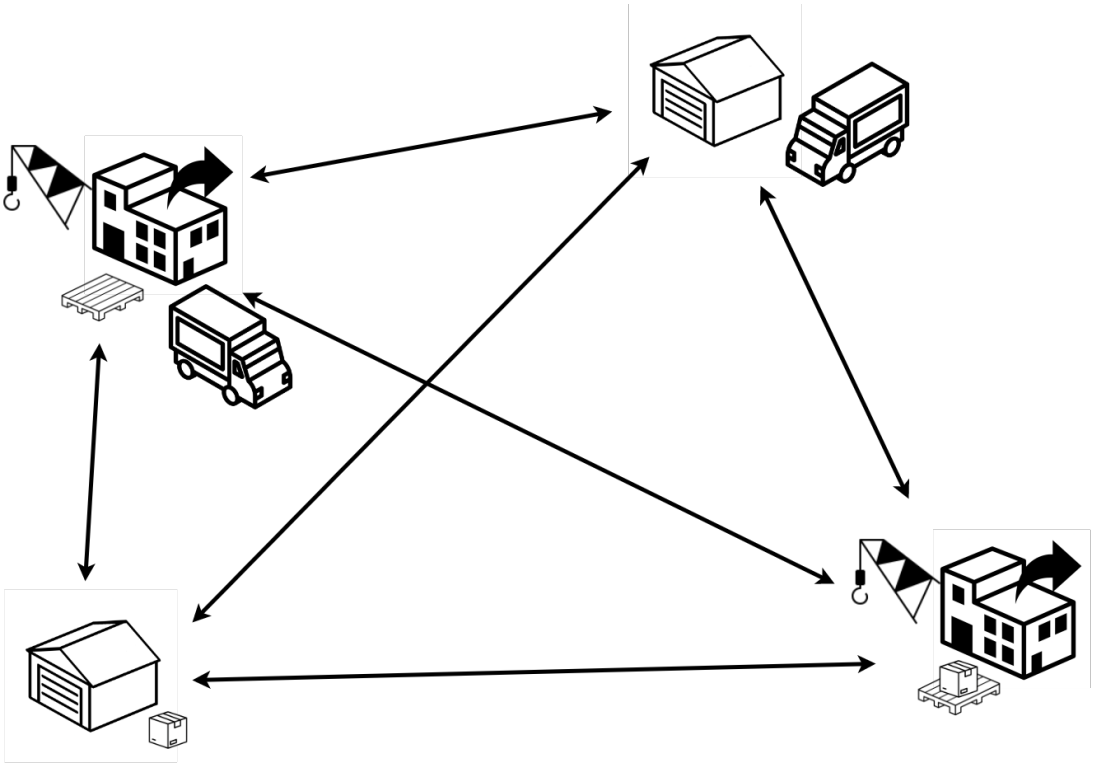
```
(define (domain blocksworld)
    (:requirements :strips)
    (:predicates (clear ?x)
                 (on-table ?x)
                 (arm-empty)
                 (holding ?x)
                 (on ?x ?y))

    (:action pickup
        :parameters (?ob)
        :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
        :effect (and (holding ?ob) (not (clear ?ob))
                (not (on-table ?ob)) (not (arm-empty))))

    (:action putdown
        :parameters  (?ob)
        :precondition (holding ?ob)
        :effect (and (clear ?ob) (on-table ?ob)
                (arm-empty) (not (holding ?ob))))

    (:action stack
        :parameters (?ob ?underob)
        :precondition (and (clear ?underob) (holding ?ob))
        :effect (and (arm-empty) (clear ?ob) (on ?ob ?underob)
                (not (clear ?underob)) (not (holding ?ob))))

    (:action unstack
        :parameters (?ob ?underob)
        :precondition (and (on ?ob ?underob) (clear ?ob) (arm-empty))
        :effect (and (holding ?ob) (clear ?underob) (not (arm-empty))
                (not (on ?ob ?underob)) (not (clear ?ob)))))
```

3.1.3 Depot

Třetí vybranou doménou je Depot, která modeluje převoz beden na paletách mezi mezi distributory, za pomoci 6 predikátů a 5 akcí. Doména vznikla kombinací dvou jiných domén, Logistic a Blocksworld. Depot byla zařazena mezi domény třetího ročníku IPC, poté co se doména Logistic, ve druhém ročníku, ukázala jako příliš jednoduchá. Velikost problému lze regulovat mnoha parametry, například počtem distributorů nebo počtem beden. Cílem problémů je rozvézt palety s bednami mezi distributory a sklady, tak aby

název	#skladů	#distributorů	#nákladáků	#palet	#beden	#zvedáků
prob01.pddl	2	2	2	4	4	3
prob02.pddl	2	2	2	4	4	3
prob03.pddl	2	2	2	4	4	3
prob04.pddl	2	3	2	5	5	3
prob05.pddl	2	3	2	5	5	3
prob06.pddl	2	3	2	5	5	3
prob07.pddl	3	3	2	6	6	3
prob08.pddl	3	3	2	6	6	3

Tabulka 3.1. Počet jednotlivých objektů při generování problémů v doméně Depot



Obrázek 3.1. Ukázka problému z domény Depot.

každá bedna seděla na správné paletě. Pro tuto doménu bylo vygenerováno 8 problémů, dataset vzniklý po prohledání problémů obsahuje 46015 trénovacích dat.

3.1.4 Visitall

Poslední vybranou doménou je Visitall, která zachycuje pohyb robota na grafu. Definice domény je velice jednoduchá a jsou k ní potřeba pouze 3 predikáty a 1 akce. Cílem úlohy je aby robot navštívil požadované vrcholy. V soutěži IPC se objevila hned dvakrát a to v roce 2011 a 2014, v obou případech byly problémy zadefinované na grafu s mřížkovou strukturou o velikosti $n \times n$. Polovina problémů měla za cíl navštívení všech políček, pro splnění cílů té druhé bylo zapotřebí navštívit pouze polovinu polí. Při generování trénovacích problémů se zvolila stejná mřížková struktura jako při soutěži, přičemž velikost mřížky byla 2×2, 3×3 a 4×4. Celkem se tak jednalo o 6 trénovacích problémů, které vytvořili dataset o velikosti 39636.

```
(define (domain depots)
    (:requirements :strips :typing)
    (:types place locatable - object
       depot distributor - place
            truck hoist surface - locatable
            pallet crate - surface)

    (:predicates (at ?x - locatable ?y - place)
                 (on ?x - crate ?y - surface)
                 (in ?x - crate ?y - truck)
                 (lifting ?x - hoist ?y - crate)
                 (available ?x - hoist)
                 (clear ?x - surface))

    (:action Drive
        :parameters (?x - truck ?y - place ?z - place)
        :precondition (and (at ?x ?y))
        :effect (and (at ?x ?z) (not (at ?x ?y))))

    (:action Lift
        :parameters (?x - hoist ?y - crate ?z - surface ?p - place)
        :precondition (and (at ?x ?p) (available ?x)
                     (at ?y ?p) (on ?y ?z) (clear ?y))
        :effect (and (clear ?z) (lifting ?x ?y) (not (available ?x))
                (not (clear ?y)) (not (at ?y ?p)) (not (on ?y ?z))))

    (:action Drop
        :parameters (?x - hoist ?y - crate ?z - surface ?p - place)
        :precondition (and (at ?x ?p) (at ?z ?p)
                     (clear ?z) (lifting ?x ?y))
        :effect (and (available ?x) (at ?y ?p) (clear ?y) (on ?y ?z)
                (not (clear ?z)) (not (lifting ?x ?y)) ))

    (:action Load
        :parameters (?x - hoist ?y - crate ?z - truck ?p - place)
        :precondition (and (at ?x ?p) (at ?z ?p) (lifting ?x ?y))
        :effect (and (in ?y ?z) (available ?x) (not (lifting ?x ?y))))

    (:action Unload
        :parameters (?x - hoist ?y - crate ?z - truck ?p - place)
        :precondition (and (at ?x ?p) (at ?z ?p)
                      (available ?x) (in ?y ?z))
        :effect (and (lifting ?x ?y) (not (in ?y ?z))
                (not (available ?x)))))
```

```
(define (domain visitall)
    (:requirements :typing)
    (:types place - object)
    (:predicates (connected ?x ?y - place)
                 (at-robot ?x - place)
                 (visited ?x - place))

    (:action move
        :parameters (?curpos ?nextpos - place)
        :precondition (and (connected ?curpos ?nextpos)
                           (at-robot ?curpos))
        :effect (and (at-robot ?nextpos) (visited ?nextpos)
                (not (at-robot ?curpos)))))
```

3.2 Grafová reprezentace

Navrhnutí a následné vytvoření reprezentace PDDL stavu bylo hlavním přínosem této práce. Proto byla snaha o vytvoření univerzálního řešení, které bude možné aplikovat nezávisle na doméně. Další podmínkou pro reprezentaci bylo aby natrénované modely mohly být využity i pro různě velké problémy.

Nejjednodušším způsobem jak vytvořit reprezentaci pro neuronové sítě, by bylo vzít všechna fakta daného stavu a pomocí hashovací funkce je převést na vektory, které se následně spojí do jednoho většího. Na tomto jednoduchém řešení by však mohly fungovat pouze rekurentní sítě, jelikož velikost vstupního vektoru se mění podle velikosti stavu. Problémem by však zůstávalo pořadí při spojování vektorů. Pro jeden stav tak existuje velké množství různých sekvencí, které na výstupu rekurentní sítě dávají odlišné výsledky. Tato skutečnost znemožňuje učení modelu neuronové sítě a tak je tato reprezentace považována za nevyhovující.

Způsobem jak získat vstup, který řeší problém s pořadím faktů, je vytvoření one-hot kódování ze všech možných instancí predikátů v daném problému. Tato reprezentace by umožňovala využití většího množství typů neuronových sítí, bohužel je však závislá na velikosti problému. To omezuje využití natrénovaných modelů pouze na problémy se stejnými objekty. Dalším nepříjemností této reprezentace je její vysoká paměťová náročnost, způsobená predikáty vyšších řádů.

Problémy předchozích návrhů je možné vyřešit s využitím grafů, což je společným rysem všech následně popsaných řešení. Jak bylo již dříve zmíněno v sekci 3.1, vytvořené reprezentace nejsou úplně doménově nezávislé, jediná podmínka, která musí být splněna je využití predikátů maximálně druhého řádu.

3.2.1 Minimální grafová reprezentace

Hlavní myšlenkou této reprezentace bylo co nejefektivněji zakódovat PDDL stav pomocí grafu. Když se podíváme na stavový prostor klasické plánovací úlohy (Definice 2.11), vidíme že jediné čím jsou jednotlivé stavy definovány jsou fakta. Fakta jsou tvořeny kombinací predikátů a objektů . Pokud tedy ze všech objektů vytvoříme vrcholy, můžeme pomocí binárních[3] predikátů vytvořit hrany, případně již existující hraně při-

[3] Označení pro predikát druhého řádu.

dat predikát jako další vlastnost. Unární[4] predikáty použijeme pro vytvoření vlastností vrcholů. Ve výsledném grafu je počet vrcholů rovný počtu objektů v problému a počet hran je shora omezen počtem faktů s binárním predikátovým symbolem. Vektory s vlastnostmi vrcholů a hran jsou one-hot kódováním predikátů. Aby reprezentace byla invariantní vzhledem k pořadí zpracování faktů je potřeba znát predikáty a všechny objekty před tvorbou grafu. Algoritmus MGR zobrazuje tvorbu grafu.

Algorithm Minimal Graph Representation(*domain*,*problem*,*state*):
  G_M = empty graph
  for each *object* **in** *problem*:
    create vertex o and add it to G_M
  for each *fact* **in** *state*:
    get *predicate* form *fact*
    get *objects* from *fact*
    if length(*objects*) == 1:
      add *predicate* as feature to corresponding vertex o
    else if length(*objects*) == 2:
      get verticies o_1, o_2 from G_M corresponding to *objects*
      create edge e from o_1 and o_2
      if e **not in** G_M:
        add e to G_M
      add *predicate* as feature to corresponding edge e
  return G_M

3.2.2 Úplná grafová reprezentace

Myšlenka úplné grafové reprezentace zůstává stejná jako u 3.2.1. Graf je tvořen stejným způsobem jenom jsou přidány i hrany které nejsou součástí stavu, čímž vzniká tak úplný graf. Pro tyto hran se však ponechává vektor s vlastnostmi prázdný. Důvodem proč reprezentace využívá úplný graf, je snaha zlepšit šíření informace mezi jednotlivými částmi grafu. Tato změna je důležitá pro grafové neuronové sítě využívající konvoluční principy.

Algorithm Full Graph Representation(*domain*,*problem*,*state*):
  G_F = full graph with *object* verticies from *problem*
  for each *fact* **in** *state*:
    get *predicate* form *fact*
    get *objects* from *fact*
    if length(*objects*) == 1:
      add *predicate* as feature to corresponding vertex o
    else if length(*objects*) == 2:
      get verticies o_1, o_2 from G_F corresponding to *objects*
      create edge e from o_1 and o_2
      add *predicate* as feature to corresponding edge e
  return G_F

3.2.3 Úplná grafová reprezentace s cílem

Reprezentace 3.2.1 i 3.2.2 poskytují grafové neuronové síti úplné informace o konkrétním stavu, nikoli však úplné informace o řešeném problému. Hlavní informací která oběma

[4] Označení pro predikát prvního řádu.

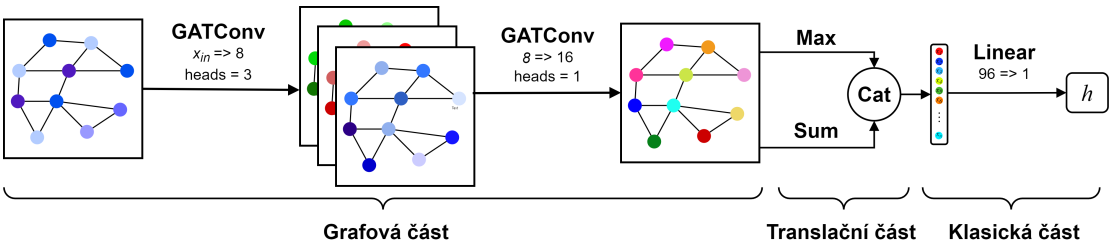
předchozím reprezentacím schází je informace o cíli. Ke grafu vytvořeném algoritmem FGR je přidán další velice podobný graf vytvořený z faktů vyskytujících se v cíli. Druhý vytvořený graf není úplně přesnou reprezentací cíle, protože v PDDL je cíl reprezentován jako konjunkce logických výrazů. Drobnou nevýhodou této reprezentace je, že druhý graf poskytuje pouze informaci, která fakta jsou pro cíl důležitá.

Algorithm Full Graph Representation with Goal(*domain*,*problem*,*state*):
  G_F = graph created using FGR algorithm
  G_G = full graph with *object* verticies from *problem*
  get *goal* from *problem*
  for each *fact* **in** *goal*:
    get *predicate* form *fact*
    get *objects* from *fact*
    if length(*objects*) == 1:
      add *predicate* as feature to corresponding vertex o
    else if length(*objects*) == 2:
      get verticies o_1, o_2 from G_G corresponding to *objects*
      create edge e from o_1 and o_2
      add *predicate* as feature to corresponding edge e
  G_{FG} = addition of G_G to G_F
  return G_{FG}

3.3 Architektura neuronové sítě

Návrh architektury neuronové sítě je velice často ovlivňován jejím použitím. Například ASNets [9] se snaží architekturou sítě[5] modelovat přechody mezi jednotlivými stavy a naučit se rozhodovací strategii[6]. Další architekturou je STRIPS-HGN [10], která svým modelem připomíná rekurentní sítě a snaží se naučit přípustnou heuristiku. Obě řešení se na problém dívají spíše ze strany klasického plánování a svým principem připomínají průchod stavovým prostorem.

Oproti zadání se navržená architektura dívá na problém čistě ze strany neuronových sítí a snaží se tak být obecnějším konceptem, který může být použit pro řešení různých problémů. Kvůli tomu je v architektuře využito modulárního přístupu, díky kterému lze jednotlivé části nezávisle upravovat. Konkrétní model použitý pro výpočet heuristiky je vidět na obrázku 3.2



Obrázek 3.2. Model použité neuronové sítě.

[5] Informace o architektuře sítě jsou v sekci 1.1
[6] Anglicky „policy“.

Grafová část

Grafová část architektury může teoreticky obsahovat jakýkoliv model grafové neuronové sítě, který pracuje s vlastnostmi na všech úrovních. Smyslem této části je naučit se, s pomocí grafové struktury, lepší reprezentaci vlastností jednotlivých prvků vstupního grafu.

Translační část

Translační část převádí graf s nově naučenými vlastnostmi na vektor. Pro převod je použito sdružení vlastností všech prvků stejného typu[7], pomocí dvou agregačních funkcí. První zvolenou agregační funkcí je maximum, které do klasické části přinese informaci nejdůležitějším prvku. Jako protiváha k maximu je za druhou agregační funkcí zvolen průměr, který lépe zachytí význam konkrétní vlastnosti. Výstupy obou agregačních funkci, aplikovaných na všech úrovních, jsou spojeny za sebe do jediného vektoru.

Klasická část

Klasická část architektury se stará o finální transformaci na požadovaný výstup. V případě že má architektura modelovat heuristickou funkci je výstupem skalár. Stejně jako grafové části i zde může být použit libovolný model klasické neuronové sítě.

3.3.1 Ztrátová funkce

S architekturou neuronové sítě úzce souvisí i volba ztrátové funkce, která je metrikou popisující kvalitu neuronové sítě. Pro její výpočet je potřeba znát výstup generovaný neuronovou sítí a výstup který měla sít vygenerovat. Hodnota ztrátové funkce je také počátečním bodem backpropagace (sekce ??), která je využívána gradientními metodami (sekce 2.4.3). Ztrátová funkce je tak nedílnou součástí trénovací smyčky neuronových sítí.

Výběr vhodné ztrátové funkce je komplikovaný, protože je závislý na řešeném problému a odvíjí se od něj celý průběh tréninku neuronové sítě. Návrh speciální ztrátové funkce pro konkrétní problém, může v některých případech vylepšit vlastnosti neuronové sítě.

Mean square error

MSE[8] je asi nejčastěji využívanou ztrátovou funkcí, pro problémy kdy je výstupem neuronové sítě reálné číslo. MSE penalizuje přísněji případy, ve kterých jsou rozdíly mezi správnou a predikovanou hodnotou vysoké.

$$\mathcal{L}(\mathbf{x}, \mathbf{y}) = \frac{1}{n} \cdot \sum_{i=1}^{n} \Big(y_i - x_i \Big)^2$$

- $\mathcal{L}$ je ztrátová funkce MSE
- $\mathbf{x}$ je vektor obsahující výstupy generované modelem na mini-sadě
- $\mathbf{y}$ je vektor obsahující korespondující správné výstupy
- n je počet trénovacích instancí v mini-sadě

[7] Tedy pro vlastnosti vrcholů, hran a grafu, zvlášť
[8] Někdy je chybně označována jako L_2

Při učení neuronové sítě modelující optimální heuristiku nastává problém co s hodnotami ∞, které jsou přiřazené stavům z nichž nelze dosáhnout cíle. Použitým řešením je náhrada ∞ za konstantu M, která je odvozená od délky nejdelšího plánu $|\pi|$ z trénovací množiny jako $M = |\pi| + 100$. Hodnota 100 je zvolena jako bezpečná rezerva pro případy, kdy je model testován na větších problémech u kterých lze očekávat delší plány.

Rankovací ztrátová funkce

Myšlenka rankovací funkce [28] je neučit se přesné hodnoty optimální heuristiky, ale pouze monotónnost heuristické funkce h^*. Toho je dosaženo vytvořením matice s relativními vzdálenostmi mezi všemi stavy z momentální mini-sady trénovacího datasetu.

$$d_{i,j} = \max\left(0, \tau - \text{sign}(y_i - y_j) \cdot (x_i - x_j)\right)$$

$$\mathbf{M}_d = \begin{bmatrix} d_{1,1} & \cdots & d_{1,n} \\ \vdots & \ddots & \vdots \\ d_{n,1} & \cdots & d_{n,n} \end{bmatrix}$$

$$\mathcal{L}(\mathbf{x}, \mathbf{y}) = \frac{1}{2n} \cdot \sum_{i=1}^{n} \sum_{j=1}^{n} (\mathbf{M}_d - \mathbf{I}\tau)$$

- $d_{i,j}$ je relativní vzdálenost špatně seřazených stavů i a j
- $\mathbf{M}_d$ je symetrickou maticí relativních vzdáleností $d_{i,j}$, $\mathbf{M}_d \in \mathbb{R}^{n \times n}$
- τ je minimální vzdálenost aby byly stavy považovány za správně seřazené
- $\mathbf{I}$ je jednotková matice
- $\mathcal{L}$ je rankovací ztrátová funkce
- $\mathbf{x}$ je vektor obsahující výstupy generované modelem na mini-sadě
- $\mathbf{y}$ je vektor obsahující korespondující správné výstupy
- n je počet trénovacích instancí v mini-sadě

Výsledná hodnota rankovací ztrátové funkce je na rozdíl od MSE vážena parametrem $\frac{1}{2n}$ a to z důvodu, že matice $\mathbf{M}_d$ je symetrická. Nejlépe funguje rankovací funkce, když je počítána na celém trénovacím datasetu. Vzhledem k velikosti počítané matice $\mathbf{M}_d \in \mathbb{R}_+^{n \times n}$ je tak nutné využít rozdělení trénovací množiny do mini-sad. V případě že natrénovaný model bude dosahovat hodnoty $\mathcal{L} = 0$ ve všech stavech, pak plánovací algoritmus GBFS bude mít stejné vlastnosti, jako při využití optimální heuristiky h^*.

Kapitola 4
Experimenty

Experimentální evaluace navržených řešení se skládá ze 2 části. V První části 4.1 se nachází analýza trénování modelů neuronových sítí. Druhá část 4.2 poté přináší popis a výsledky integrace natrénovaných modelů do paradigmatu klasického plánování.

4.1 Trénink modelů GNN

Trénink modelů byl prováděn na čtyřech doménách popsaných v sekci 3.1. Na každé doméně byly vyzkoušeny modely, využívající tři navržené grafové reprezentace popsané v sekci 3.2. Všechny modely byly trénovány v kombinaci s oběma ztrátovými funkcemi vysvětlenými v sekci 3.3.1. Celkově tak bylo skrze 4 domény natrénováno 24 modelů. Různé pohledy, včetně rozboru, trénovacích a validačních křivek modelů jsou poskytnuty v sekci 4.1.2. Tabulka 4.1 zaznamenává výsledky učení modelů, grafy s oddělenými průběhy učení se nacházejí v příloze A.

4.1.1 Technické parametry a nastavení

Modely byly učeny na serveru se 4 grafickými kartami nVidia RTX 2080Ti. Trénink modelu probíhal využitím gradientní metody Adam s parametry nastavenými na $\alpha = 0.01$, $\beta = 0.9$, $\gamma = 0.999$ a maximálním počtem iterací omezeným na $t_{max} = 1000$. Datasety vzniklé z trénovacích problémů[1] byly nadále rozděleny na trénovací a validační množiny poměrem $(0.8, 0.2)$. Rozdělení bylo náhodné a pro zachování validity výsledků, byl náhodný generátor fixován parametrem $seed = 42$. Kvůli efektivnějšímu využití paměti grafických karet byla trénovací množina ještě rozdělena do 8 mini-sad.

4.1.2 Porovnání konvergenčních grafů

Na konvergenční grafy, vzniklé trénováním modelů neuronových sítí s různými meta-parametry, se lze dívat 3 pohledy. Nejprve na ně lze nahlížet ze směru, jež porovnává rychlost učení modelu na jednotlivých doménách (viz Obrázek 4.1 a 4.2). Druhý pohled nabízí porovnání schopnosti, jednotlivých modelů, naučit se odhad optimální heuristické funkce na dané doméně (viz Obrázek 4.4 a 4.3). Porovnání za využití posledního možného řezu daty, nedává smysl, kvůli tomu že hodnota použité rankovací ztrátové funkce není správně normalizována vzhledem k parametru τ a je závislá na velikosti mini-sady. Toto je také důvodem, proč jsou u grafů s rankovací ztrátovou funkcí dvě různá měřítka vertikálních os.

Obrázek 4.1 a 4.2 ukazuje, že nejlépe si vedly modely na doméně visitall. Dále je z nich možné vypozorovat, že hlavní pokrok při učení je dosažen hned během prvních iterací a od zhruba 200 iterace už nedochází k výraznějším zlepšením. Obrázek 4.1 také

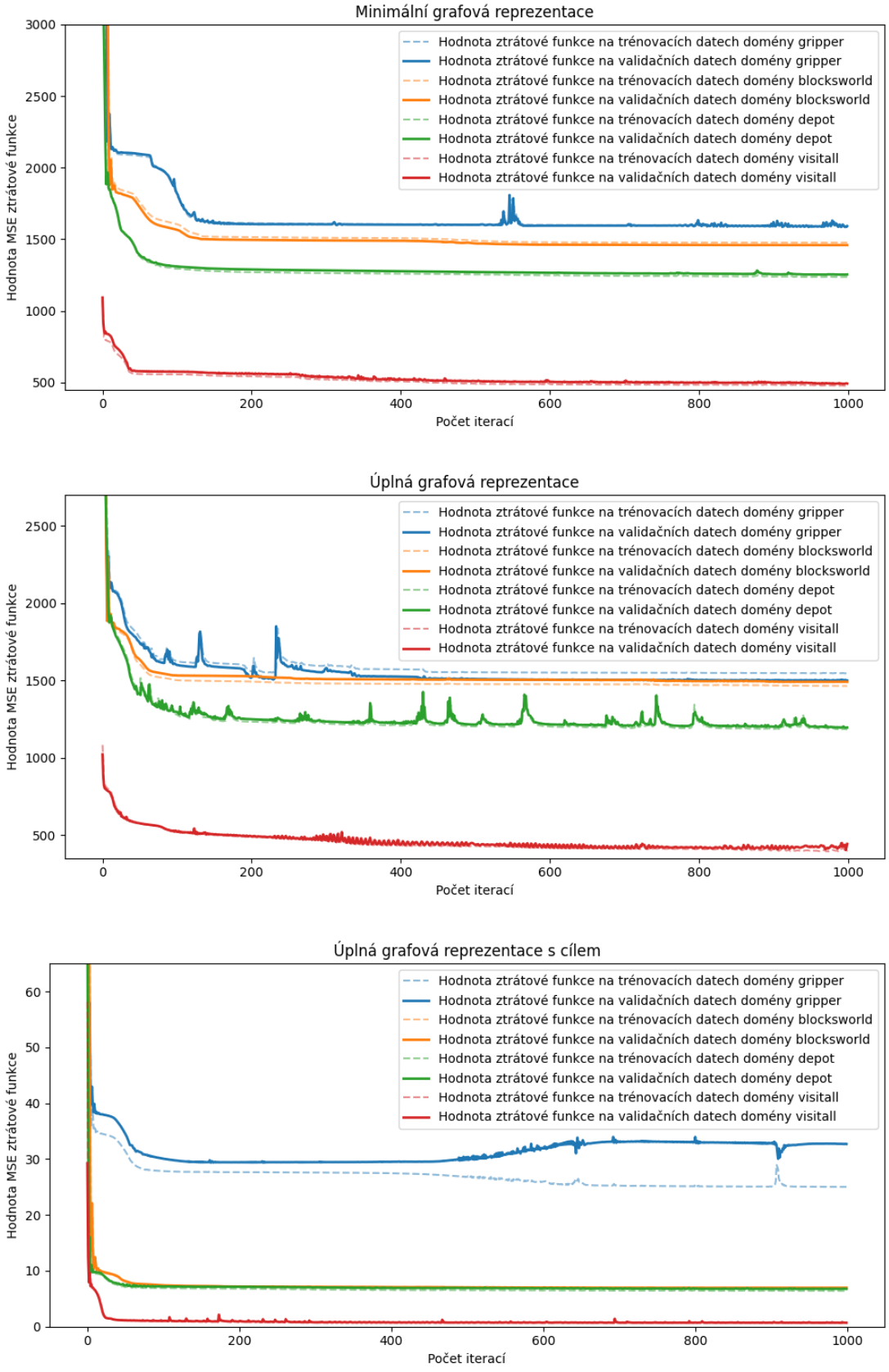
[1] jejich bližší popis se nachází v sekci s doménami 2.3.1

Typ ztrátové funkce	Typ domény	Typ Grafové reprezentace	Hodnota ztrátové funkce na trénovacích datech	Hodnota ztrátové funkce na validačních datech
MSE	Gripper	MGR	1588.294	1591.768
		FGR	1547.866	1500.296
		FGG	25.033	32.707
	Blocksword	MGR	1476.779	1459.348
		FGR	1464.409	1491.280
		FGG	6.864	6.974
	Depot	MGR	1236.796	1254.035
		FGR	1183.944	1195.636
		FGG	6.401	6.769
	Visitall	MGR	478.937	492.402
		FGR	401.986	442.147
		FGG	0.730	0.695
Rankovací funkce	Gripper	MGR	3767.038	1879.711
		FGR	3760.519	1887.703
		FGG	4317.307	2136.825
	Blocksword	MGR	6519.855	3259.539
		FGR	6489.451	3277.583
		FGG	5609.220	2794.829
	Depot	MGR	6158.277	3115.533
		FGR	5972.715	3008.910
		FGG	3239.838	2895.214
	Visitall	MGR	1588.542	1407.480
		FGR	1641.332	1447.530
		FGG	1052.829	918.804

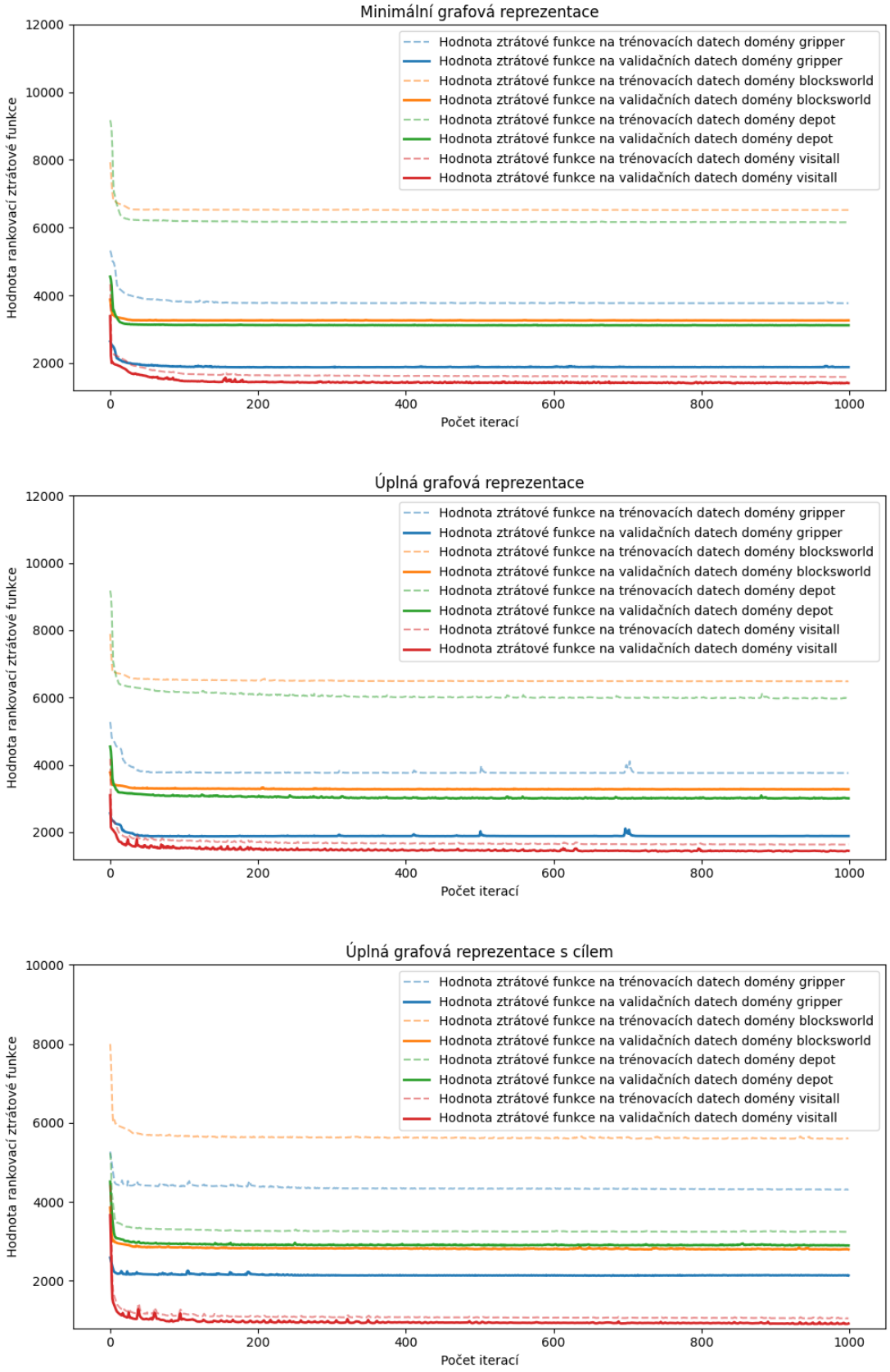
Tabulka 4.1. Hodnoty ztrátových funkcí naměřené na konci tréninku vytvořených modelů.

ještě zachycuje přeučení modelu používajícího úplné grafové reprezentace s cílem, které nastává okolo 500 iterace.

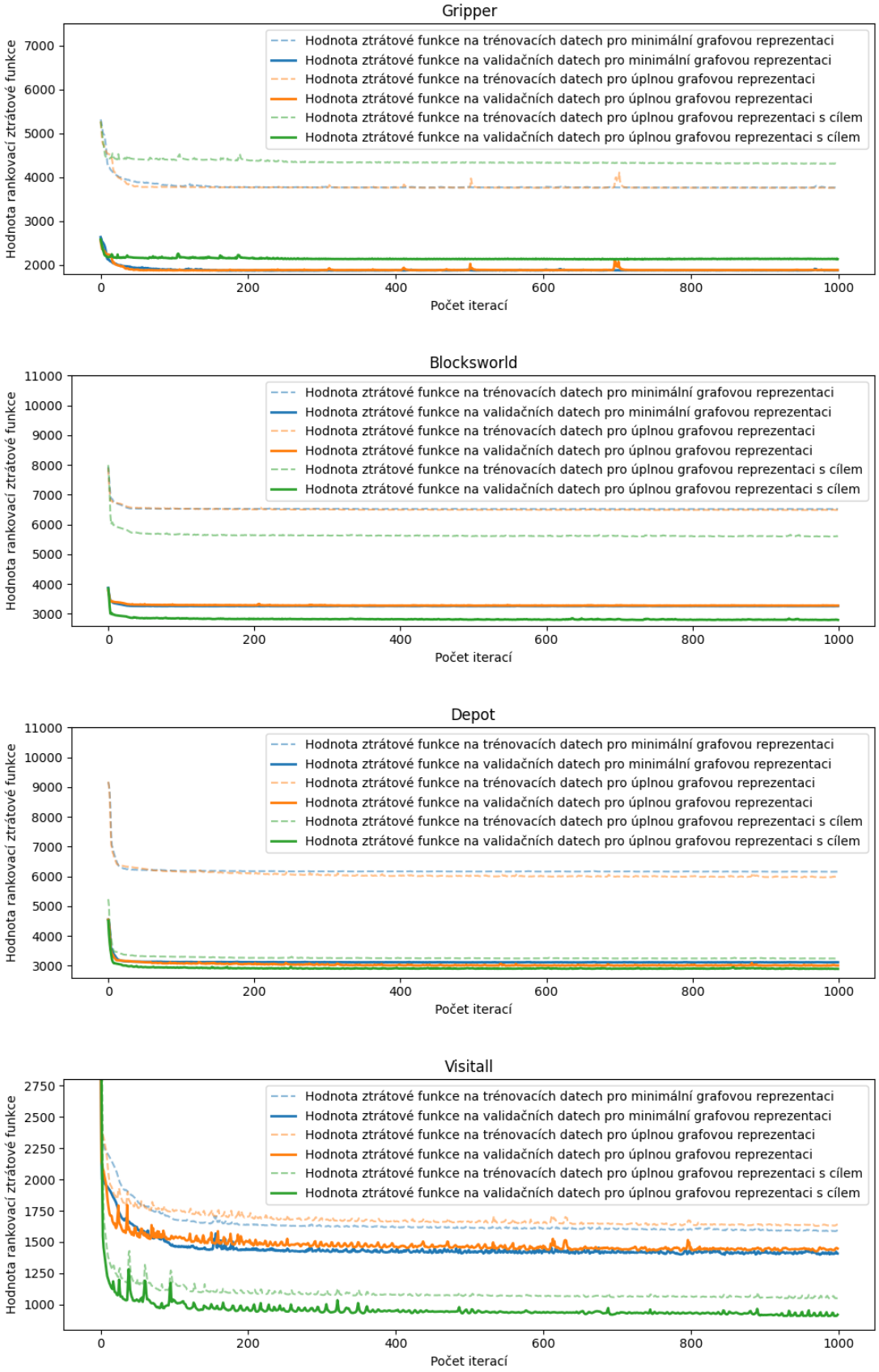
Obrázek 4.4 a 4.3 ukazuje, že ve všech případech, až na jednu výjimku, si nejlépe vede model využívající úplné grafové reprezentace s cílem. Dále je z obou obrázků viditelná podoba chování modelů používající minimální a grafové reprezentace.



Obrázek 4.1. Porovnání domén při tréninku modelů pomocí MSE.



Obrázek 4.2. Porovnání domén při tréninku modelů pomocí rankovací ztrátové funkce.



Obrázek 4.3. Porovnání grafových reprezentací při tréninku pomocí rankovací ztrátové funkce.

4.2 Plánovací experimenty

Plánovací experimenty porovnávají všechny modely natrénované v sekci 4.1 se standardními heuristikami na doménách popsaných v sekci 2.3.1. Experimenty využívaly vlastní sady vygenerovaných problémů[2] a sady problémů ze soutěže IPC. Pro porovnání natrénovaných modelů s heuristikami h_{max} a h_{ff} bylo využito plánovacího algoritmu GBFS . Rámcové tabulky s výsledky jsou prezentovány v sekci 4.2.2.

4.2.1 Nastavení experimentů a použité metriky

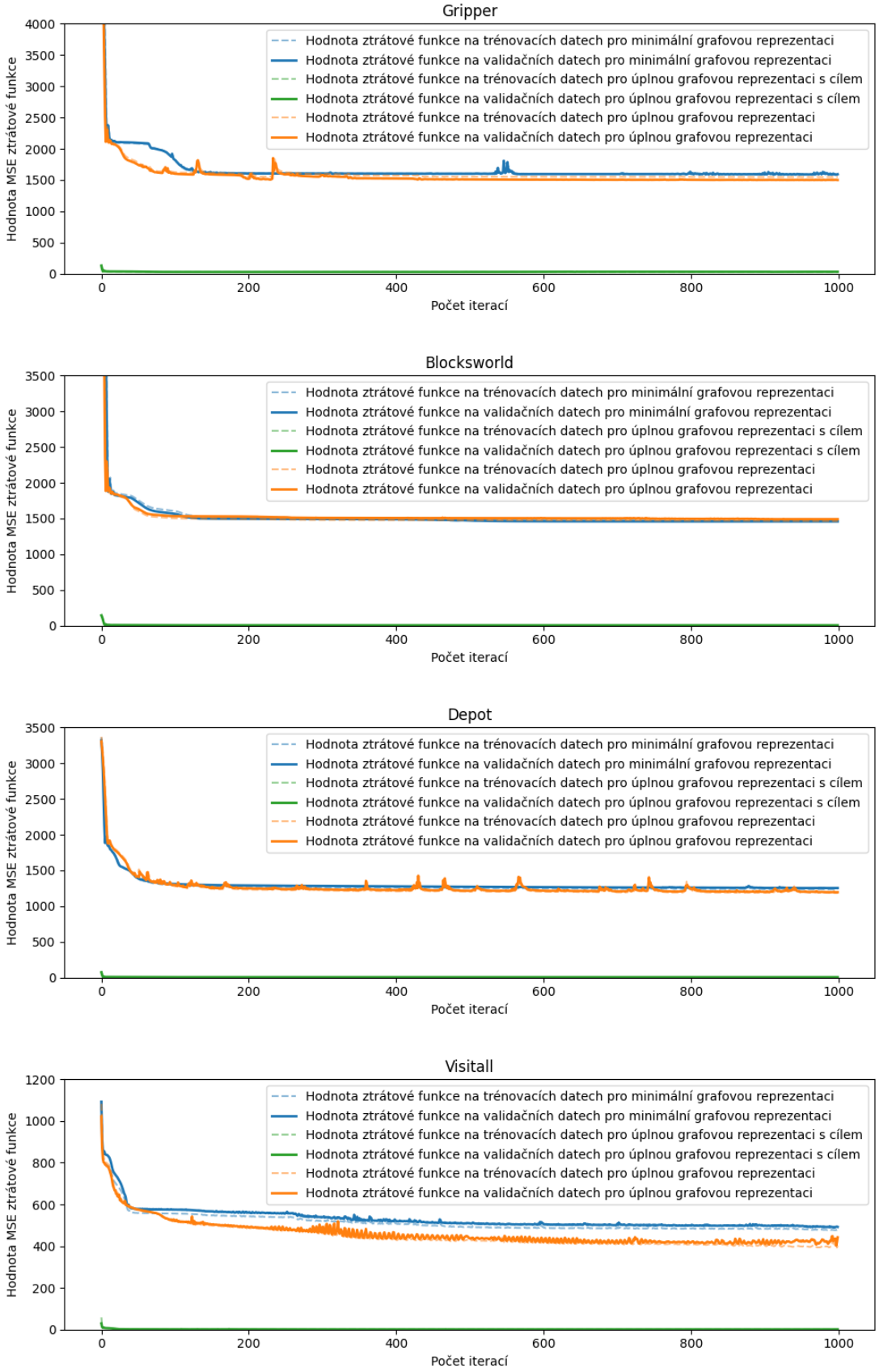
Čas pro vyřešení jednoho problému byl omezen na 30 minut. Po ukončení prohledávání, úspěšně či neúspěšně, byly zaznamenány 4 hodnoty popisující kvalitu řešení. Tyto metriky jsou následně využívány k analýze výsledků.

- Délka nalezeného plánu
- Počet uzavřených stavů
- Počet otevřených stavů
- Celkový počet stavů

4.2.2 Výsledky plánovacích experimentů

Hlavní metrikou používanou pro porovnání výsledků je procentuální úspěšnost vyřešených problémů[3]. Tabulka 4.2 zobrazuje porovnání úspěšnosti natrénovaných modelů a standardních heuristik na všech sadách testovacích problémů. Z tohoto porovnání nejlépe vychází heuristika h_{ff}, jediné kde je poražena je doména visitall, kterou ovládly modely trénované pomocí rankovací ztrátové funkce. Tabulka 4.3 porovnává délky nalezených řešení, ale vzhledem k tomu že, pro plánování nebyl použit optimální plánovač, tak naměřené hodnoty vypovídají pouze o schopnosti heuristik navigovat GBFS stavovým prostorem. Tabulka 4.4, 4.5 a 4.6 zobrazují průměrné počty navštívených, otevřených a celkově vytvořených stavů v průběhu prohledávání stavového prostoru. Z těchto hodnot lze vypozorovat efektivitu heuristiky, případně odhadnout větvící index domény.

[2] Bližší popis je vždy u konkrétní domény.
[3] V anglické literatuře se používá pojem „coverage“, neboli pokrytí



Obrázek 4.4. Porovnání grafových reprezentací při tréninku pomocí MSE ztrátové funkce.

Sada problémů	Doména	MSE ztrátová funkce MGR	FGR	FGG	Rankovací ztrátová funkce MGR	FGR	FGG	h_{max}	h_{FF}
IPC	blocksworld	40.00%	34.29%	34.29%	40.00%	34.29%	34.29%	97.14%	**100.00**%
	depot	9.09%	9.09%	9.09%	9.09%	9.09%	9.09%	31.82%	**54.55**%
	gripper	25.00%	20.00%	20.00%	25.00%	20.00%	20.00%	25.00%	**100.00**%
	visitall	25.00%	35.00%	80.00%	70.00%	65.00%	**95.00**%	10.00%	55.00%
Vlastní sada	blocksworld	11.11%	5.56%	11.11%	5.56%	11.11%	11.11%	83.33%	**100.00**%
	depot	14.29%	0.00%	0.00%	14.29%	14.29%	0.00%	**100.00**%	**100.00**%
	gripper	27.27%	18.18%	9.09%	36.36%	18.18%	18.18%	27.27%	**100.00**%
	visitall	25.00%	66.67%	91.67%	**100.00**%	**100.00**%	83.33%	0.00%	83.33%

Tabulka 4.2. Úspěšnost při řešení problémů.

Sada problémů	Doména	MSE ztrátová funkce MGR	FGR	FGG	Rankovací ztrátová funkce MGR	FGR	FGG	h_{max}	h_{FF}
IPC	blocksworld	15.57(14)	14.67(12)	14.67(12)	15.14(14)	14.83(12)	14.67(12)	30.88(34)	65.34(35)
	depot	14.50(2)	18.00(2)	15.50(2)	15.00(2)	18.00(2)	16.00(2)	31.29(7)	53.67(12)
	gripper	33.20(5)	24.50(4)	32.00(4)	26.80(5)	22.00(4)	23.00(4)	24.00(5)	85.20(20)
	visitall	60.60(5)	120.43(7)	99.94(16)	123.64(14)	82.69(13)	156.47(19)	38.50(2)	102.91(11)
Vlastní sada	blocksworld	15.00(2)	17.00(1)	16.00(2)	17.00(1)	16.00(2)	17.00(2)	26.87(15)	40.33(18)
	depot	17.00(1)	-	-	19.00(1)	25.00(1)	-	19.86(7)	20.43(7)
	gripper	46.00(3)	38.00(2)	44.00(1)	39.00(4)	33.00(2)	48.00(2)	33.33(3)	51.64(11)
	visitall	91.33(3)	206.75(8)	86.18(11)	130.50(12)	96.75(12)	85.80(10)	-	137.00(10)

Tabulka 4.3. Průměrná délka nalezených řešení(počet nalezených řešení).

Sada problémů	Doména	MSE ztrátová funkce MGR	MSE ztrátová funkce FGR	MSE ztrátová funkce FGG	Rankovací ztrátová funkce MGR	Rankovací ztrátová funkce FGR	Rankovací ztrátová funkce FGG	h_{max}	h_{FF}
IPC	blocksworld	104542.66	102407.83	110072.49	112179.77	89990.57	107172.43	16535.03	5157.71
	depot	56483.14	32815.27	11665.27	53461.36	36407.27	10993.91	47599.73	282269.23
	gripper	70248.55	60779.10	26032.10	77738.45	65109.65	17622.85	194608.30	717.20
	visitall	56524.10	678.55	148.20	640.40	108.50	192.65	443258.20	67950.60
Vlastní sada	blocksworld	176766.22	156785.72	109771.94	182093.39	156407.89	117786.94	110577.44	1081.33
	depot	110849.00	116164.86	531.57	95385.86	73305.00	400.57	2563.14	227.57
	gripper	123194.91	175034.27	78475.64	149276.45	214697.73	45262.09	361870.55	242.64
	visitall	132494.75	1914.75	215.67	387.83	102.25	136.00	583026.75	64534.58

Tabulka 4.4. Průměrný počet prozkoumaných stavů.

Sada problémů	Doména	MSE ztrátová funkce MGR	MSE ztrátová funkce FGR	MSE ztrátová funkce FGG	Rankovací ztrátová funkce MGR	Rankovací ztrátová funkce FGR	Rankovací ztrátová funkce FGG	h_{max}	h_{FF}
IPC	blocksworld	259799.71	222162.26	234230.03	247884.63	213199.91	222993.43	104872.74	5821.83
	depot	281828.18	82491.23	14173.32	300748.41	79774.95	15946.68	216547.68	87200.00
	gripper	127436.15	24397.80	12371.65	178611.75	26984.40	20812.40	440865.75	8400.00
	visitall	9221.50	692.35	319.05	706.60	256.35	412.95	248270.40	102265.20
Vlastní sada	blocksworld	436138.33	370987.78	302677.67	419928.44	376537.67	298198.67	454744.22	1618.72
	depot	166608.00	175171.14	2091.00	214233.14	196455.86	2231.86	10790.00	607.71
	gripper	184718.09	70882.00	30975.91	159074.09	83660.18	57039.91	315305.64	1518.36
	visitall	16465.42	1633.50	215.25	510.83	219.00	258.00	345090.75	79837.00

Tabulka 4.5. Průměrný počet otevřených stavů.

Sada problémů	Doména	MSE ztrátová funkce MGR	MSE ztrátová funkce FGR	MSE ztrátová funkce FGG	Rankovací ztrátová funkce MGR	Rankovací ztrátová funkce FGR	Rankovací ztrátová funkce FGG	h_{max}	h_{FF}
IPC	blocksworld	359650.11	318572.46	325355.03	352812.94	297671.11	321690.80	121407.77	9540.86
	depot	317163.68	102856.68	20888.18	341186.73	98893.23	21631.64	259475.41	106887.64
	gripper	186957.85	61372.35	31315.05	237243.70	78476.15	31652.15	635474.05	9117.20
	visitall	40422.55	1282.75	463.10	1279.50	364.85	603.25	549156.00	170191.60
Vlastní sada	blocksworld	600636.61	517308.33	396380.11	589063.11	521330.67	405489.17	565321.67	2612.78
	depot	215745.00	234697.00	2528.43	275152.57	247174.71	2578.57	13320.57	831.29
	gripper	274738.18	169664.55	82263.82	264444.64	254659.36	85979.82	677176.18	1761.00
	visitall	85568.17	3202.58	367.08	882.83	321.25	389.58	755552.67	144349.83

Tabulka 4.6. Průměrný celkově vytvořených stavů.

Kapitola 5
Diskuze nad výsledky

Vytvořené modely, i přes svojí relativně malou velikost dokázaly, že jsou schopné se na trénovací datech, naučit odhadovat heuristickou funkci. Tato skutečnost je vypodložena konvergenčními grafy, ze sekce 4.1.2, na kterých je vidět pokles hodnot ztrátové funkce na trénovacích i na validačních datech. Dalším důležitým poznatkem z těchto grafů je, že i přes vysoký počet iterací, proběhla většina zlepšení během prvních několika iterací.

Při pohledu na Obrázek 4.4 je vidět že MSE ztrátová funkce má problémy naučit modely, které využívají grafové reprezentace bez přidaného cíle. To může být vysvětleno tím, že tyto modely nezískávají ze vstupního grafu dostatečnou informaci, nutnou pro aproximaci hodnoty optimální heuristiky. V případě rankovací funkce (Obrázek 4.3) nejsou rozdíly mezi hodnotami tak vysoké, což je možné vysvětlit tím, že tlak není vyvíjen na naučení konkrétní „vzdálenosti“ od cílového stavu, ale na vytvoření pořadí mezi stavy.

Pokud se zaměříme na Obrázek 4.1 a 4.2, můžeme pozorovat, že všechny modely dosahují při tréninku nejlepších výsledků na doméně visitall. Tento trend pokračoval, také při experimentální evaluaci, kdy se 4 ze 6 natrénovaných modelů dosáhli lepších výsledků, než heuristická funkce h_{FF}. Doména visitall dělá problémy oběma klasickým heuristikám a důvod spočívá v jejím návrhu, který je popsán v sekci 3.1.4.

Tabulka 4.4, 4.5 a 4.6 přináší překvapivé informace o efektivitě řešení domény visitall modely trénovanými rankovací funkcí. Tyto modely předvedly značnou míru škálovatelnosti, jelikož byly učeny pouze na problému o velikosti 4×4, ale dokázali v časovém limitu vyřešit problémy z IPC o velikosti 30×30 a model využívající úplnou grafovou reprezentaci s cílem byl schopen vyřešit i problém o velikosti 48×48.

Z provedených experimentů můžeme obecně říci, že výsledky dosažené modely, trénované právě pomocí rankovací funkce, byly přinejhorším stejné, ale ve většině případů lepší než výsledky dosažené modely trénovaných pomocí MSE.

Kapitola 6
Závěr

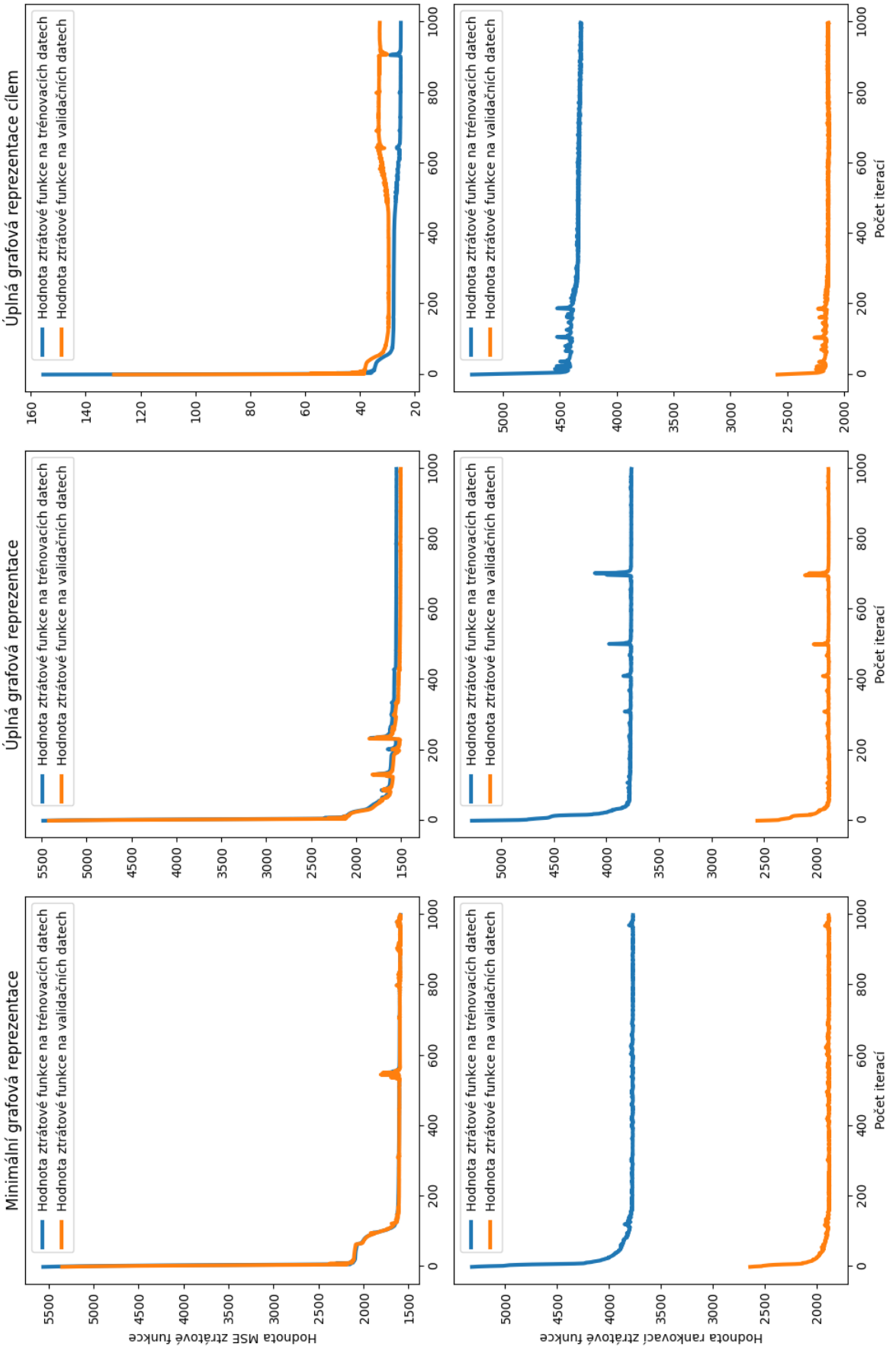
Hlavním cílem práce bylo vytvoření doménově nezávislé grafové reprezentace plánovacího problému, kterou bude možné použít pro trénování heuristických funkcí pomocí grafových neuronových sítí.

Za tímto účelem byly navrženy a implementovány tři reprezentace stavů a architektura grafové neuronové sítě. Byla také upravena rankovací ztrátová funkce. Následně byly jednotlivé navržené části experimentálně otestovány a porovnány se standardními metodami klasického plánování.

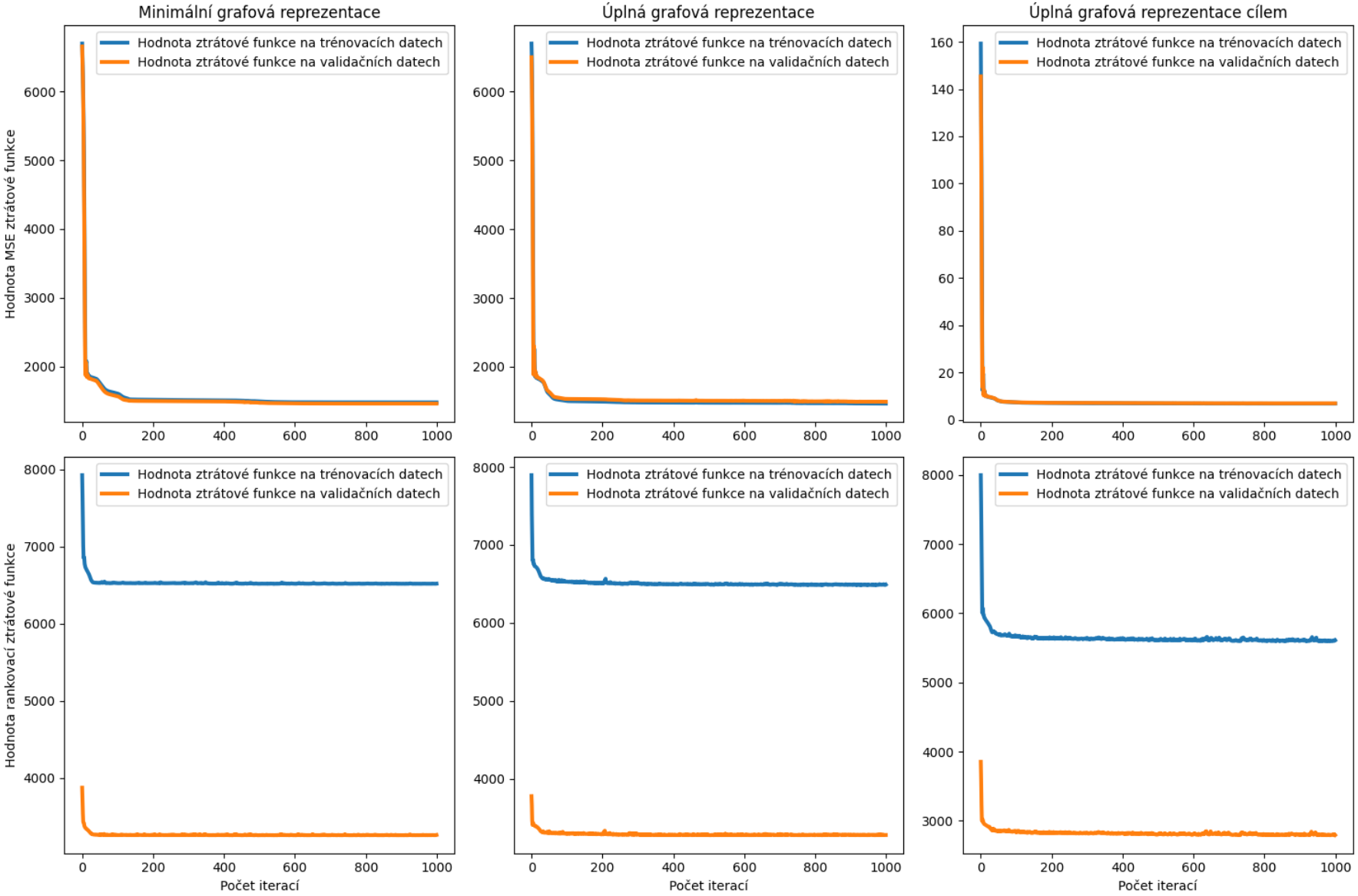
Při experimentech byl objeven nečekaný potenciál modelů při plánování na doméně visitall a potvrzena důležitost návrhu ztrátové funkce pro trénování heuristické funkce.

Na práci lze navázat rozšířením experimentů zejména ve směru porovnání více architektur grafových neuronových sítí. Případně se lze věnovat vylepšení vlastností rankovací ztrátové funkce.

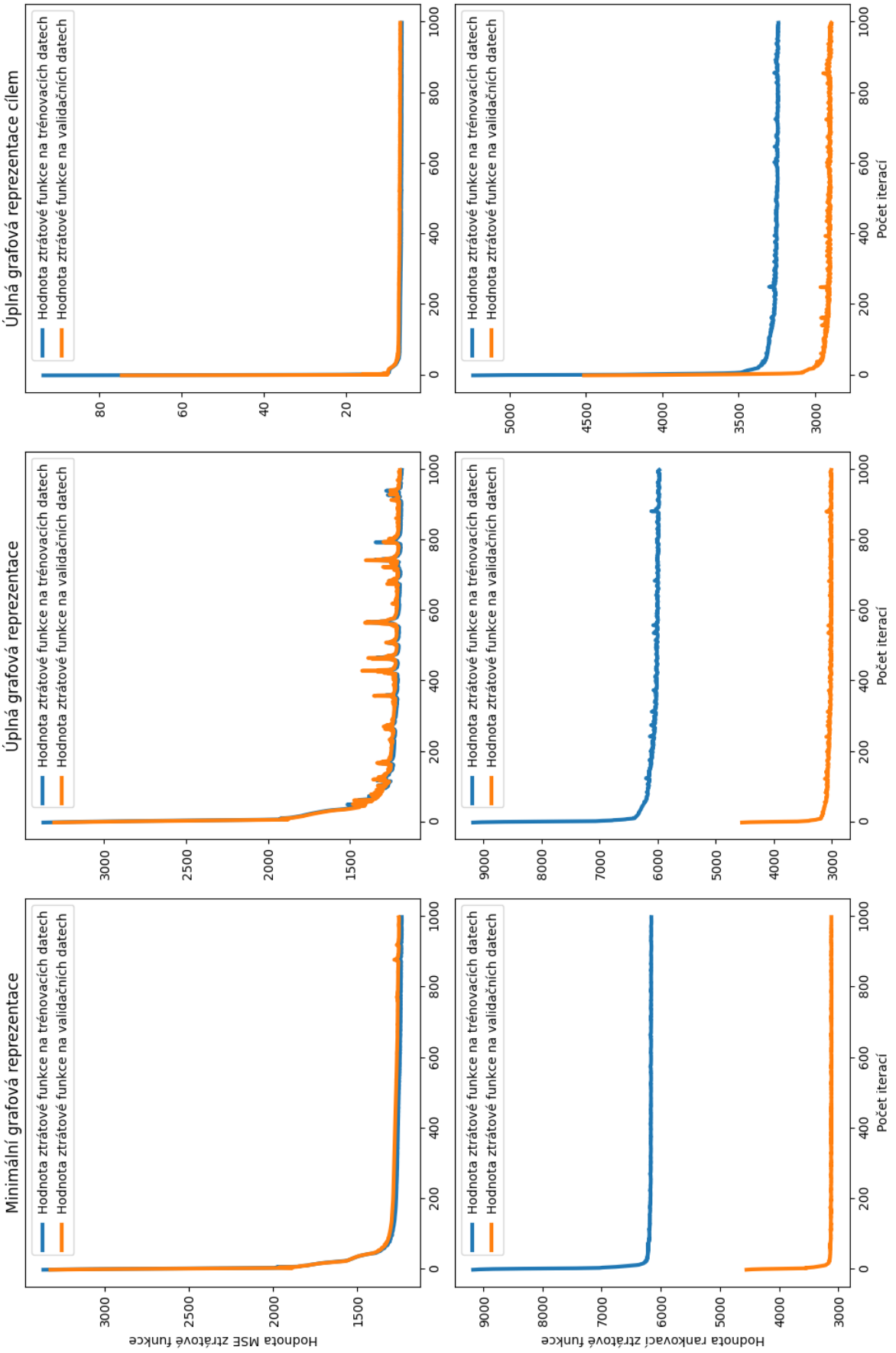
Příloha A
Obrázky



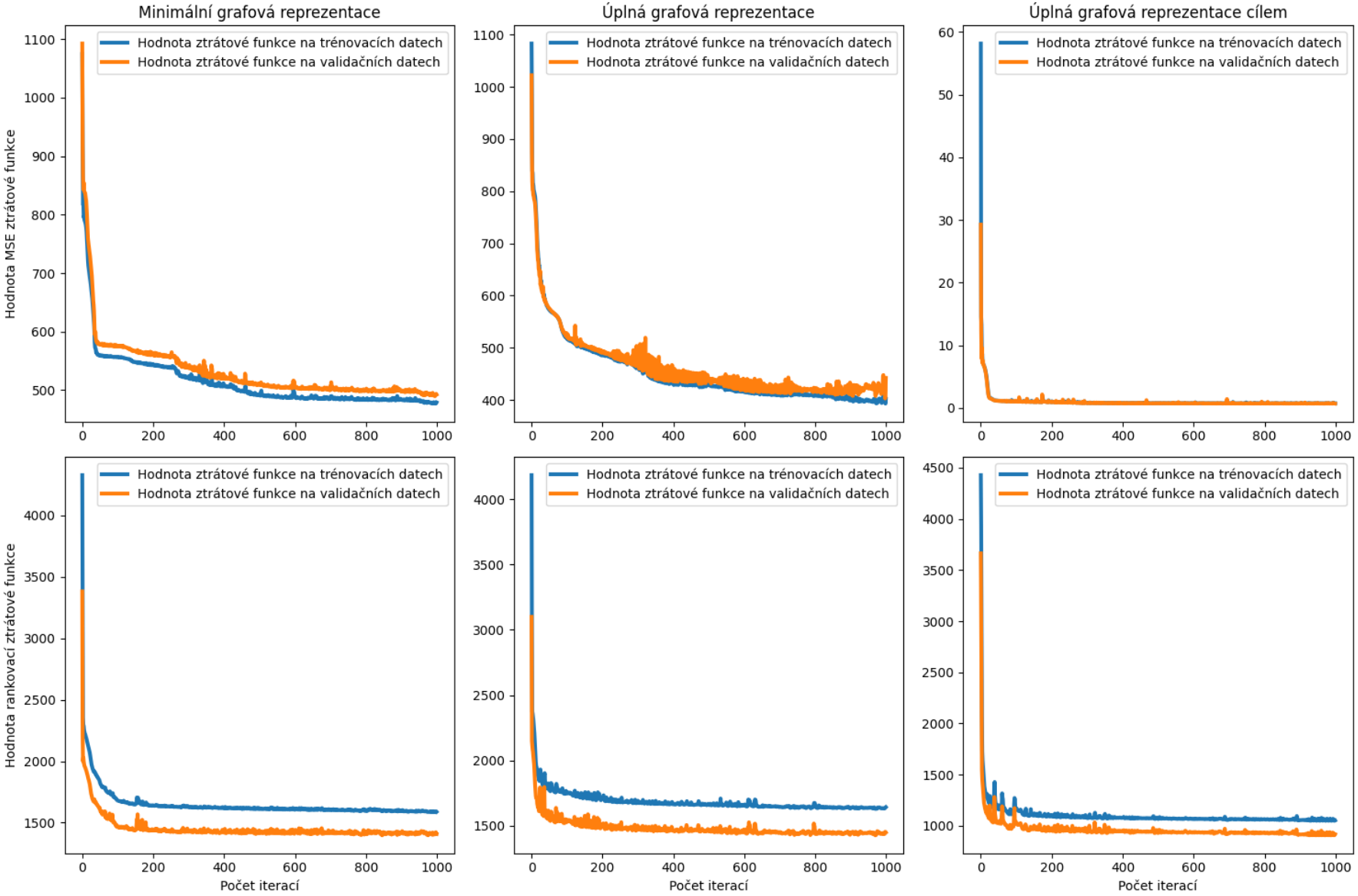
Obrázek A.1. Průběh hodnoty ztrátové funkce při tréninku modelů a doméně Gripper.



Obrázek A.2. Průběh hodnoty ztrátové funkce při tréninku modelů a doméně Blocksworld.



Obrázek A.3. Průběh hodnoty ztrátové funkce při tréninku modelů a doméně Depot.



Obrázek A.4. Průběh hodnoty ztrátové funkce při tréninku modelů a doméně Visitall.

Literatura

[1] David E. Smith. The Case for Durative Actions: A Commentary on PDDL2.1. *J. Artif. Intell. Res.*. 2003, 20 149–154. DOI 10.1613/jair.1997.

[2] Drew V. McDermott. The 1998 AI Planning Systems Competition. *AI Mag.*. 2000, 21 (2), 35–55. DOI 10.1609/aimag.v21i2.1506.

[3] Ping Huang, Huijuan Zhu, Lei Zheng a Ying Wang. *Text Sentiment Analysis based on BERT and Convolutional Neural Networks.* In: *NLPIR 2021: 5th International Conference on Natural Language Processing and Information Retrieval, Sanya, China, December 17 - 20, 2021*. ACM, 2021. 1–7. `https://doi.org/10.1145/3508230.3508231`.

[4] Alex Krizhevsky, Ilya Sutskever a Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. In: Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Leon Bottou a Kilian Q. Weinberger, eds. *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. 2012. 1106–1114. `https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html`.

[5] Masataro Asai a Alex Fukunaga. *Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary.* In: Sheila A. McIlraith a Kilian Q. Weinberger, eds. *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*. AAAI Press, 2018. 6094–6101. `https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16302`.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg a Demis Hassabis. Human-level control through deep reinforcement learning. *Nat.*. 2015, 518 (7540), 529–533. DOI 10.1038/nature14236.

[7] Alberto Fernandez a Sergio Gomez. Portfolio selection using neural networks. *CoRR*. 2005, abs/cs/0501005

[8] Edward Groshev, Aviv Tamar, Siddharth Srivastava a Pieter Abbeel. Learning Generalized Reactive Policies using Deep Neural Networks. *CoRR*. 2017, abs/1708.07280

[9] Sam Toyer, Felipe W. Trevizan, Sylvie Thiebaux a Lexing Xie. ASNets: Deep Learning for Generalised Planning. *CoRR*. 2019, abs/1908.01362

[10] William Shen, Felipe W. Trevizan a Sylvie Thiebaux. Learning Domain-Independent Planning Heuristics with Hypergraph Networks. *CoRR*. 2019, abs/1911.13101

[11] Simon Ståhlberg, Blai Bonet a Hector Geffner. Learning General Optimal Policies with Graph Neural Networks: Expressive Power, Transparency, and Limits. *CoRR*. 2021, abs/2109.10129

[12] Dorina Thanou, Philip A. Chou a Pascal Frossard. Graph-based compression of dynamic 3D point cloud sequences. *CoRR*. 2015, abs/1506.06096

[13] Steven M. LaValle. *Planning algorithms*. Cambridge University Press, 2006. ISBN 978-0-521-86205-9.

[14] Edwin P. D. Pednault. *ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus*. In: Ronald J. Brachman, Hector J. Levesque a Raymond Reiter, eds. *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89). Toronto, Canada, May 15-18 1989*. Morgan Kaufmann, 1989. 324–332.

[15] J. Scott Penberthy a Daniel S. Weld. *UCPOP: A Sound, Complete, Partial Order Planner for ADL*. In: Bernhard Nebel, Charles Rich a William R. Swartout, eds. *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, USA, October 25-29, 1992*. Morgan Kaufmann, 1992. 103–114.

[16] Maria Fox a Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res.*. 2003, 20 61–124. DOI 10.1613/jair.1129.

[17] Stefan Edelkamp a Jorg Hoffmann. *PDDL2. 2: The language for the classical part of the 4th international planning competition*. . Technical Report 195, University of Freiburg.

[18] Alfonso Gerevini a Derek Long. *Plan constraints and preferences in PDDL3*. . Technical Report 2005-08-07, Department of Electronics for Automation ….

[19] Daniel L Kovacs. Complete BNF description of PDDL 3.1. *Language Specification, Department of Measurement and Information Systems, Budapest University of Technology and Economics: Budapest, Hungary*. 2011,

[20] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain.. *Psychological review*. 1958, 65 (6), 386.

[21] Xianlin Wang, Yuqing Liu a Haohui Xin. Bond strength prediction of concrete-encased steel structures using hybrid machine learning method. *Structures*. 2021, 32 2279-2292. DOI 10.1016/j.istruc.2021.04.018.

[22] David E Rumelhart, Geoffrey E Hinton a Ronald J Williams. Learning representations by back-propagating errors. *nature*. 1986, 323 (6088), 533–536.

[23] Diederik P Kingma a Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 2014,

[24] John Duchi, Elad Hazan a Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization.. *Journal of machine learning research*. 2011, 12 (7),

[25] Tijmen Tieleman, Geoffrey Hinton a others. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*. 2012, 4 (2), 26–31.

[26] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce a Alexander B. Wiltschko. A Gentle Introduction to Graph Neural Networks. *Distill*. 2021, DOI 10.23915/distill.00033. https://distill.pub/2021/gnn-intro.

[27] Shaked Brody, Uri Alon a Eran Yahav. How Attentive are Graph Attention Networks?. *CoRR*. 2021, abs/2105.14491

[28] Michaela Urbanovska a Anton n Komenda. Neural networks for model-free and scale-free automated planning. *Knowl. Inf. Syst.*. 2021, 63 (12), 3103–3138. DOI 10.1007/s10115-021-01619-8.