VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Srovnání paradigmat strojového učení

Comparison of Machine Learning Paradigms

Martin Kaleta

Bakalářská práce

Vedoucí práce: Ing. Adam Albert

Ostrava, 2023

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Č.j.VSB/22/078976
vsboee22026093

Zadání bakalářské práce

Student: **Martin Kaleta**

Studijní program: B0613A140014 Informatika

Téma: Srovnání paradigmat strojového učení
Comparison of Machine Learning Paradigms

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je shrnou teorii strojového učení v multiagentních systémech, provést shrnutí základních paradigmat. V případové studii student vytvoří příklady algoritmů pro porovnání jejich rysů a funkcionality.

Práce bude obsahovat:
1. Shrnutí teorie strojového učení na základě symbolické reprezentace znalostí, genetických algoritmů, pravděpodobnosti a umělých neuronových sítí.
2. Případová stude, ve které student naimplementuje metody strojového učení na základě symbolické reprezentace a genetických algoritmů. Vstupem algoritmů budou data reprezentována pomocí klauzulí jazyka Prolog.

Seznam doporučené odborné literatury:

[1] LUGER, George F. Artificial intelligence: structures and strategies for complex problem solving. 6th ed. Boston: Pearson Addison-Wesley, c2009. ISBN 978-0-321-54589-3.
[2] MITCHELL, Tom M. Machine Learning. New York: McGraw-Hill, c1997. ISBN 00-704-2807-7.
[3] KUBÍK, Aleš. Inteligentní agenty. Brno: Computer Press, 2004. ISBN 80-251-0323-4.
[4] POOLE, David L., MACKWORTH Alan K. Artificial intelligence: foundations of computational agents. 2nd pub. Cambridge: Cambridge University Press, 2010. ISBN 978-0-521-51900-7.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Adam Albert**

Datum zadání: 01.09.2021

Datum odevzdání: 30.04.2023

Garant studijního programu: doc. Mgr. Miloš Kudělka, Ph.D.

V IS EDISON zadáno: 07.11.2022 12:18:12

Abstrakt

Cílem práce je shrnout teorii strojového učení v multiagentních systémech, jejich základních paradigmat a pomocí implementace porovnat základní rysy a funkcionality vybraných metod strojového učení. První část práce se zabývá představením strojového učení a popisem jednotlivých paradigmat. Druhá část je věnovaná případové studii.

Klíčová slova

strojové učení, agent, multiagentní systém, symbolická reprezentace znalostí, genetické algoritmy, pravděpodobnost, umělé neuronové sítě

Abstract

The aim of this thesis is to summarize the theory of machine learning in multi-agent systems, their basic paradigms and compare the basic features and functionalities of selected machine learning methods through implementation. The first part of the thesis deals with the introduction of machine learning and the description of each paradigm. The second part devoted to a case study.

Keywords

machine learning, agent, multiagent system, symbol-based, genetic algorithms, stochastic, connectionist approaches

Poděkování

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, protože bez nich by tato práce nevznikla.

Obsah

Seznam použitých zkratek a symbolů

KNN	–	Algoritmus k-nejbližších sousedů
DBSCAN	–	Prostorové shlukování aplikací se šumem na základě hustoty
MAS	–	Multiagentní systém
GA	–	Genetické algoritmy
ANN	–	Umělé neuronové sítě
PDP	–	Paralelně distribuované zpracování
HMM	–	Skrytý Markovův model

Seznam obrázků

Seznam algoritmů

Kapitola 1

Úvod

Cílem této práce je poskytnout srovnání různých paradigmat strojového učení v multiagentních systémech. Poskytnout charakteristiku jednotlivých paradigmat, včetně jejich výhod, nevýhod a vhodnosti použití při řešení různých druhů problémů. Celá práce je rozdělena na dvě části. První část je věnována shrnutí teorie strojového učení na základě symbolické reprezentace znalostí, genetických algoritmů, pravděpodobnosti a umělých neuronových sítí. Druhá část se věnuje případové studii, která obsahuje popis a implementaci algoritmu založeném na symbolické reprezentaci znalostí a genetických algoritmech.

Celá práce je strukturována následovně. Kapitola 2 popisuje co je to multiagentní systém a obsahuje definici agenta, jeho popis a základní rozdělení agentů. V kapitole 3 je stručně popsáno, co je to strojové učení, jaké jsou jeho základní rozdělení podle způsobu učení a jaké jsou základní přístupy zpracování dat ve strojovém učení. Kapitola 4 zaměřena na popis a charakteristiku 4 základních přístupů strojového učení.

Kapitola 5 obsahuje případovou studii, ve které jsem implementoval dva algoritmy. Prvním z nich je Winstonův algoritmus 5.1 aplikovaný na rozpoznávání tvarů, který je založen na symbolické reprezentaci znalostí. Algoritmus je možné použít na celou řadu problémů. Druhým z nich je algoritmus inspirovaný přirozeným výběrem živočišných druhů tj. evoluční biologií, který vyhledává cestu v bludišti 5.2. Kapitola 6 obsahuje závěr a shrnutí mé práce.

Kapitola 2

Multiagentní systémy

Multiagentní systém (MAS) [2] je decentralizovaný systém skládající se z několika autonomních, inteligentních agentů, kteří vzájemně spolupracují v prostředí na dosažení společného cíle. Klíčovou vlastností pro MAS je racionalita. Tím se rozumí, že agenti jsou schopni plnit úkoly v prostředí, ve kterém se nacházejí. Inteligence agentů se může lišit a může být velmi omezená (např. agent, který pouze přepíná mezi dvěma stavy) nebo velice rozvinutá. Takovým příkladem může být osobní auto, které pomocí svých senzorů pozoruje okolí a podle situace dynamicky rozhodne, co provede (např. senzor, který hlídá bezpečnou vzdálenost mezi vozidly a podle toho upraví rychlost).

S MAS se setkáme tam, kde centralizované systémy s předem daným chováním nevyhovují. Nejčastěji se jedná o systémy, kde je potřeba dynamického chování (např. řízení leteckého provozu, pak dále použití v oblasti počítačových sítí při řízení vyvažování zátěže sítě). V takových situacích nemusí centralizovaný systém správně fungovat a došlo by k jeho zhroucení. Když použijeme MAS a nastane neočekávaná situace, během které dojde k selhání několika agentů, tak nedojde k selhání celého systému, protože ostatní agenti jsou stále schopni autonomně na danou situaci reagovat.

Tímto lze říct, že mezi základní vlastnosti MAS patří autonomnost, robustnost, decentralizace a adaptace. Podrobněji je dále problematika MAS a agentů popsaná v [2].

V další části bude dále definován a popsán pojem agent.

2.1 Agent

Podle [2] je *agent* definován následovně:

Definice 1 *Agent je entita zkonstruovaná za účelem kontinuálně a do jisté míry autonomně plnit své cíle v adekvátním prostředí na základě vnímání prostřednictvím senzorů a prováděním akcí prostřednictvím aktuátorů. Agent přitom ovlivňuje podmínky v prostředí tak, aby se přibližoval k plnění cílů.*

Agenta můžeme popsat jako jakéhosi zástupce, který je pověřen k vykonávání určitých úkolů. Měl by být schopen fungovat samostatně bez nutnosti zásahu člověka, případně minimálním. K tomu, aby mohli agenti úspěšně plnit své úkoly, tak se musí neustále učit a zlepšovat svou činnost. Agenta lze konstruovat jak v hardwarové podobě, tak softwarově.

Agenty lze rozdělit do několika skupin podle jejich racionality, inteligence a také podle jejich vnitřní stavby jednotlivých částí (složitosti organizace vnitřních komponent) [2]. Lze je rozdělit do těchto 4 skupin:

- reaktivní
- deliberativní
- sociální
- hybridní

Kapitola 3

Strojové učení

3.1 Úvod

Strojové učení je jednou z vědních disciplín umělé inteligence, která se zabývá algoritmy a technikami, jakými je agentovi umožněno se učit [2]. Na základě získaných zkušeností je schopen se zlepšovat ve svém chování, zefektivnit svou schopnost přizpůsobit se změnám okolního prostředí, a podávat tak přesnější výsledky. Strojové učení jako vědní obor není příliš jednoduché správně zařadit, protože se mezi sebou jednotlivé obory prolínají. Můžeme mezi ně zařadit obor pravděpodobnosti a statistiky, optimalizace či dolování dat.

Techniky strojového učení se využívají v mnoha odvětvích. Můžeme se s ním setkat v medicíně, jako je včasné odhalení chorob podle nasbíraných dat o předchozích pacientech, v bankovnictví při rozpoznávání nelegálního užití kreditních karet nebo při rozpoznávání hlasů, psaného textu.

3.2 Rozdělení

Tato část se zaměřuje na základní rozdělení strojového učení a jejich charakteristiku. Strojové učení lze rozdělit podle několika charakteristik do různých kategorií, a to podle druhu učení, způsobu zpracování nebo získávání dat. Nejjednodušeji lze strojové učení rozdělit podle druhu zpětné vazby do třech skupin:

- *učení s učitelem* (supervised learning)
- *učení bez učitele* (unsupervised learning)
- *učení posilováním* nebo také *zpětnovazebné učení* (reinforcement learning)

Učení s učitelem realizuje svou činnost na základě trénovacích vzorků, které jsou označeny ohodnocením. Podobně funguje *učení bez učitele*, jen v tomto případě chybí pro trénovací vzorky jejich

ohodnocení. U *učení s posilováním* se využívá zpětné vazby za správně chování a trest za nežádoucí chování.

3.2.1 Učení s učitelem

V [5] je *učení s učitelem* definováno následovně:

Definice 2 (Supervised Learning) *Učení s učitelem je metoda, která předpovídá funkční závislostí mezi vstupními a výstupními hodnotami.*

Agentovi jsou předkládány příklady z trénovací sady dat ke zpracování. Jednotlivé příklady jsou popsány sadou vstupních a výstupních atributů. Vstupní atributy popisují charakteristiky nějakého objektu (např. auta) a výstupní atribut může popisovat, co přesně to je (např. o jaký druh auta se jedná).

Tento postup se dá přirovnat k standardnímu učení ve školách, kdy se nám vyučující snaží vysvětlit nějaký problém. Když provedeme nějakou chybu, tak nás opraví a pokusí se nás navést k správnému řešení.

Mezi nejznámější metody využívající tento typ učení patří umělé neuronové sítě a rozhodovací stromy.

3.2.2 Učení bez učitele

Učení bez učitele rovněž anglicky Unsupervised learning je metoda učení, která obdobně jako učení s učitelem využívá sadu trénovacích dat, ale s jediným rozdílem, že data neobsahují hodnoty výstupních atributů. Tímto se eliminuje role učitele a nezbývá nic jiného než se naučit rozpoznávat souvislosti samostatně. Na nalezení vhodné hypotézy musí daný algoritmus přijít sám principem pokus a omyl.

Ve vstupní datech jsou hledány podobnosti mezi atributy, podle kterých jsou rozděleny do tzv. shluků. Shluky jsou oblasti tvořené daty, které jsou si vzájemně podobné. Pro nalezení podobnosti se využívá vzdálenosti mezi jednotlivými objekty, které jsou reprezentovány ve formě vektorů.

Příkladem algoritmů, které staví na tomto popisovaném principu učení je K-Means, KNN.

Učení bez učitele je dále podrobněji popsáno v [1, 6, 7].

3.2.3 Učení posilováním

Podle [6] je *učení s posilováním* definováno následovně:

Definice 3 (Reinforcement Learning) *Při učení posilováním se agent učí na základě řady posílení - odměn nebo trestů.*

Jedná se o metodu učení pomocí zpětné vazby, kdy je agent umístěn do nějakého systému (stavového prostoru), ve kterém se učí chovat metodou odměňovaní za správné chování a trestáním za to špatné. Agent se snaží najít nejlepší možné chování. Například získání bodů na konci šachové partie napoví agentovi, že provedl správnou akci [6].

Často se s touto metodou učení setkáme ve videohrách, ale i v robotice. Mezi algoritmy, které využívají tuto metodu učení je například Q-learning nebo Deep Q-Networks.

3.3 Základní přístupy

V této kapitole se zaměříme na základní druhy přístupů zpracování dat strojovým učením. Jedná se hlavně o tyto tři způsoby:

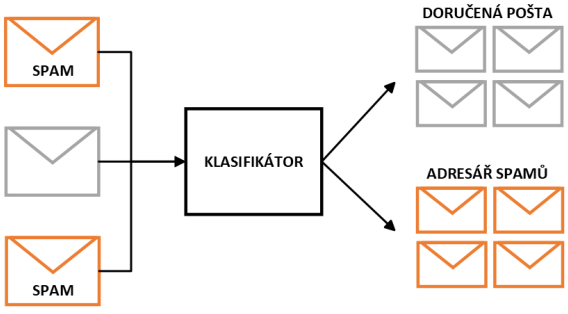
- klasifikace (classification)
- regrese (regression)
- shlukování (clustering)

3.3.1 Klasifikace

Jedná se o přístup strojového učení, který kategorizuje vstupní data do tříd podle vybraných kategorií. Klasifikátor k tomu používá vstupní datovou sadu, kterou rozdělí na tréninkovou a testovací. Z tréninkové sady získá zkušenosti a ty potom uplatní na testovací sadu k zařazení vzorku do vhodné kategorie, kde hodnota výstupního atributu je diskrétní.

Klasifikovat lze jak binárně, tak i pro více tříd. K tomuto přístupu existuje řada algoritmů, které se ho snaží řešit. Příkladem algoritmů, které pomáhají řešit tento přístup jsou rozhodovací stromy, náhodné stromy (Random Forests) a umělé neuronové sítě.

S klasifikací se dnes setkává většina lidí na světě a ani o tom nemá povědomí. Používá se při rozlišení nevyžádané pošty v e-mailové komunikaci, která je znázorněna na obrázku 3.1. Obrázek demonstruje klasifikaci příchozí pošty. Každá příchozí pošta projde klasifikátorem, který rozhodne,



Obrázek 3.1: Klasifikace příchozí e-mailové komunikace.

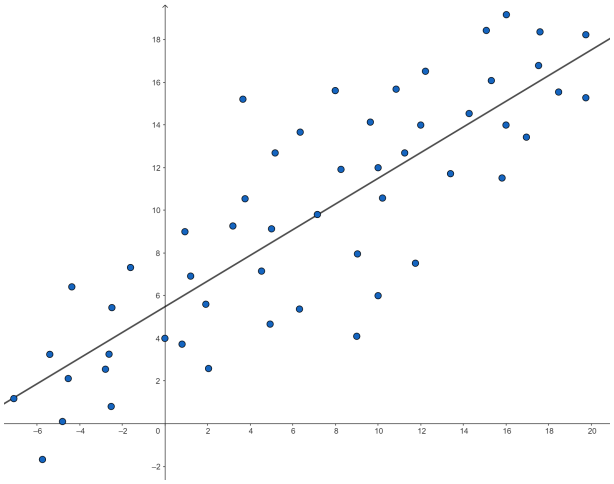
jestli se jedná chtěný nebo nechtěný e-mail. Dále v bankovnictví, kdy nám pomáhá klasifikovat nové klienty banky do různých profilů, ve zdravotnictví k predikci nemocí a v mnoha dalších odvětvích.

3.3.2 Regrese

Regrese je dalším přístupem, který se snaží odhadnout spojitou hodnotu výstupu podle vstupních dat. Snaží se vytvořit vztah mezi jednou závislou proměnnou a několika nezávislými proměnnými. Nejběžnějším typem regrese, který se používá je lineární regrese. Tento přístup patří do učení s učitelem.

Používá se nejčastěji tam, kde potřebujeme předpovědět nějakou číselnou hodnotu. Například odhadování ceny domů, platů zaměstnanců nebo marketingového rozpočtu pro další rok.

Na obrázku 3.2 je vyobrazen vztah mezi jednou nezávislou proměnnou na ose x a závislou proměnnou na ose y. Například kdyby osa x reprezentovala venkovní teplotu a osa y počet prodaných kusů zmrzliny, tak můžeme předpovědět, kolik asi musíme objednat kusů zmrzliny pro další den, když bude taková teplota.



Obrázek 3.2: Grafické znázornění lineární regrese v datové analýze.

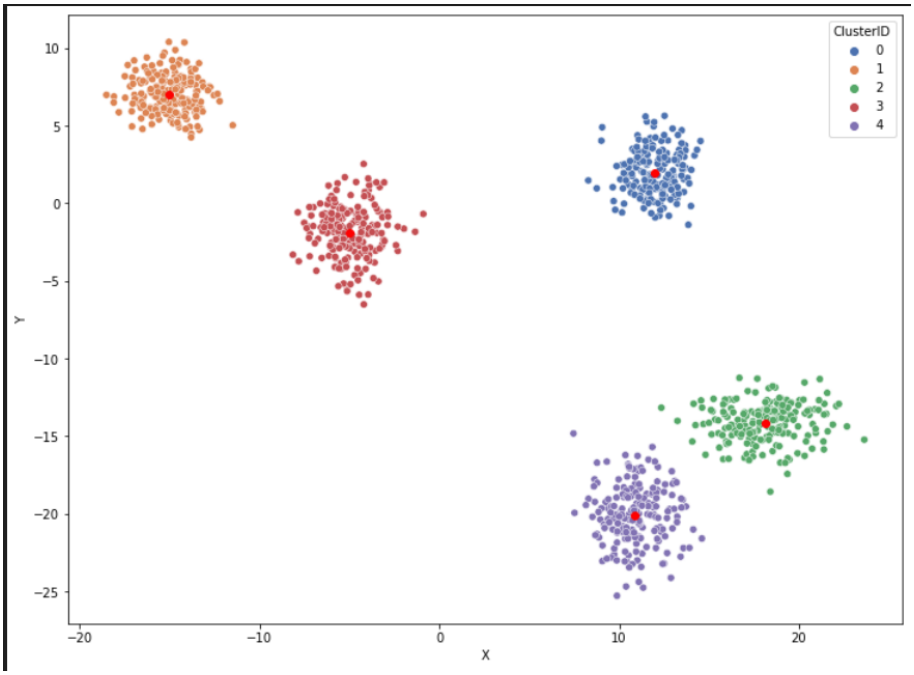
3.3.3 Shlukování

Shlukování neboli Clustering je přístup strojového učení využívající způsobu učení bez učitele, který seskupuje data do shluků podle jejich podobných rysů. Podobnost se mezi nimi určuje nejčastěji pomocí vzdálenosti mezi objekty. Pak tedy nejbližší objekty spolu tvoří cluster, který se postupně rozšiřuje o další objekty.

Existuje široká řada algoritmů, které se zde aplikují. Mezi nejznámější můžeme zařadit K-Means a DBSCAN.

Na obrázku 3.3, který znázorňuje graf shlukování si můžete všimnout 5 vzniklých shluků, kde každý z nich se skládá z několika podobných objektů, kdy nejvíce u středu se nacházejí objekty,

které jsou si nejvíce podobné. Příklad využití shlukování může být v marketingu, kdy se společnosti snaží zjistit, na jaké skupiny lidí zaměřit svou reklamu.



Obrázek 3.3: Příklad shlukovaní nad kolekcí dat.

Kapitola 4

Paradigma

V této kapitole si shrneme teorii 4 základních paradigmat strojového učení, kterými jsou symbolická reprezentace znalostí, genetické algoritmy, pravděpodobnost a umělé neuronové sítě. Jednotlivé přístupy si zde popíšeme z hlediska základních charakteristik, využití, výhod či nevýhod a jejich používaných algoritmů.

4.1 Symbolická reprezentace znalostí

V této kapitole si rozebereme první ze čtyř paradigmat, kterým je symbolická reprezentace znalostí. Symbolická reprezentace znalostí je dána množinou symbolů a gramatikou. Obsahuje pravidla definovaná pomocí logických výrazů jako je například predikátová logika 1.řádu. Pomocí pravidel jsou následně reprezentovány entity (objekty) a vztahy mezi nimi. Jednotlivé algoritmy se snaží odvodit nové a užitečné zobecnění, které lze vyjádřit pomocí symbolů nebo pravidel [1].

Jak již bylo zmíněno data jsou reprezentovány ve formě výrazů, například predikátové logiky prvního řádu nebo v jiné formě, která využívá symbolický zápis. Zápis pomocí symbolů a pravidel je pro člověka mnohem čitelnější a dokáže z něho pochopit jeho význam. Jednoduchým příkladem může být *míč*, který může být definován tvarem, velikostí, barvou. Například míč, který má červenou barvu a je malý může být definován pomocí jednoduchých pravidel takto:

$$velikost(x, \text{malý}) \wedge barva(x, \text{červená}) \wedge tvar(x, \text{kulatý}) \tag{4.1}$$

4.1.1 Konceptuální učení

V [1] se uvádí, že konceptuální učení je typickým problémem induktivního učení. Pokouší se odvodit obecnou definici nějakého konceptu[1] (například kočky, psa, čehokoliv), která by umožnila agentovi správně rozpoznat budoucí případy tohoto konceptu. Je to proces učení, který rozpoznává podob-

[1]Podle [1, 4] je koncept třída objektů, které popisuje hledaná hypotéza.

nosti v datech a podle nich klasifikuje. Pro své učení využívá množiny příkladů (popis konceptů), ze kterých získává zkušenosti a buduje obecnou hypotézu.

Konceptuální učení je základem pro algoritmy *Version space search* (prohledávání prostoru verzí) a rozhodovací stromy [1].

4.1.1.1 Version space search

Prohledávání prostoru verzí (Version space search) je metoda, která se používá k prohledávání prostoru konceptů (příkladů, jak pozitivních i negativních), ze kterých se agent snaží nalézt hypotézu popisující vstupní data. Hledá symbolicky reprezentovaný popis, který je upravován na základě vstupních příkladů. Jedná se o inkrementální učení, kdy jsou příklady předkládaný agentovi postupně. K tomu, aby agent mohl nalézt hypotézu potřebuje 2 základní operace. Jsou jimi operace generalizace (zobecnění) a specializace (upřesnění). Vyhledávání v prostoru hypotéz využívá toho, že operace generalizace a specializace definuje částečné uspořádání na konceptech, které se používá k vedení vyhledávání [1].

Mezi základní operace generalizace podle [1] můžeme zařadit tyto:

1. Nahrazení konstanty proměnnou, například konstantu barvy míče nahradíme proměnnou, která reprezentuje jakoukoliv barvu.

$$barva(x, \text{červená}) \Rightarrow barva(x, y)$$

   Vzniká obecnější hypotéza, která není soustředěna pouze na míče červené barvy, ale na libovolně barevné míče.

2. Odstranění podmínky z výrazu, která specializovala hypotézu a vylučovala některé správné příklady.

$$tvar(x, \text{kulatý}) \wedge velikost(x, \text{malý}) \wedge barva(x, \text{červená})$$
$$\Rightarrow$$
$$tvar(x, \text{kulatý}) \wedge barva(x, \text{červená})$$

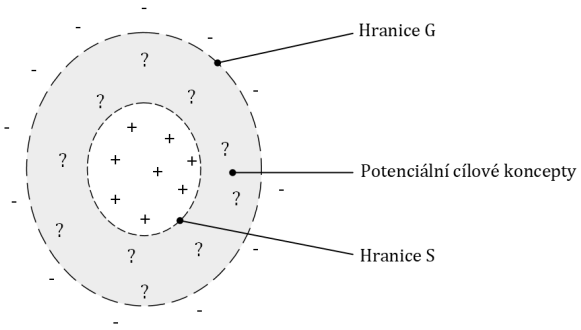
3. Přidání podmínky k výrazu, například definujeme, že míč může mít jen červenou nebo žlutou barvu.

$$tvar(x, \text{kulatý}) \wedge velikost(x, \text{malý}) \wedge barva(x, \text{červená})$$
$$\Rightarrow$$
$$tvar(x, \text{kulatý}) \wedge velikost(x, \text{malý}) \wedge [barva(x, \text{červená}) \vee barva(x, \text{žlutá})]$$

4. Nahrazení atributu obecnější hodnotou z jeho rodiče, například hodnotu atributu barva nahradíme obecnou typem primární barva.

$$barva(x, \text{červená}) \Rightarrow barva(x, primarni\_barva)$$

Na základě operací generalizace a specializace vznikly algoritmy, které řeší prohledávání prostoru hypotéz (version space) a snaží se nalézt výslednou hypotézu. První dva algoritmy *Specific to General Search* a *General to Specific Search* se snaží zmenšovat prostor hypotéz ve směru prohledávání od obecného ke specifickému a od specifického k obecnému. Třetím z nich je *Candidate Eliminations algoritmus*, který využívá oba směry prohledávání předchozích dvou algoritmů. Candidate Eliminations obsahuje dvě množiny konceptů. Množinu maximálně obecných konceptů G, kterou specializuje a množinu specifických konceptů S, kterou generalizuje. Tím dojde podle obrázku 4.1 pomocí konvergence k eliminaci všech negativních konceptů a zahrnutí nových potenciálních cílových konceptů.

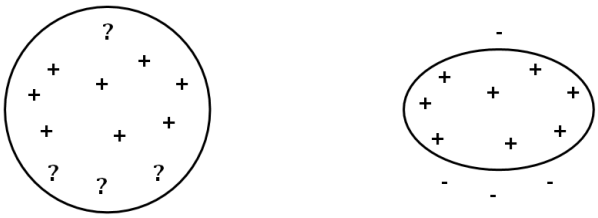


Obrázek 4.1: Konvergující hranice S a G.

Všechny výše uvedené algoritmy používají sadu, jak pozitivních i negativních příkladů výsledného konceptu. Pouze z pozitivních příkladů lze zobecňovat a nalézt hypotézu. S tímto vzniká problém přehnaného zobecnějí (Overgeneralization) znázorněného na obrázku 4.2, kdy bude hypotéza příliš obecná a bude zahrnovat i příklady, které nespadají pod námi hledaný koncept. Z toho důvodu se využívají negativní příklady, pomocí kterých se hypotéza specializuje a zabraňuje vzniku příliš obecné hypotézy. V ideálním případě musí být naučený koncept dostatečně obecný, aby pokryl všechny pozitivní příklady, a zároveň dostatečně specifický, aby vyloučil negativní příklady.

Toto téma bude dále popsáno v případové studii, která se zabývá implementací a popisem Winstonova algoritmu pro rozpoznávání tvarů.

Podrobněji se o prohledávání prostoru verzí dočtete v [1, 5].



Obrázek 4.2: Úloha negativních příkladů
v předcházení přílišné generalizace.

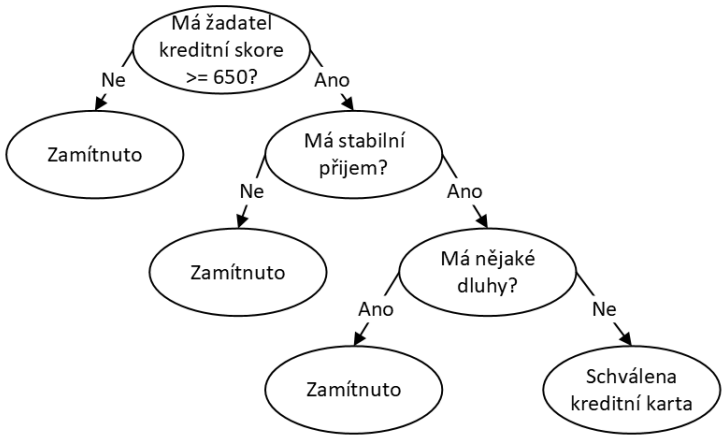
4.1.2 Rozhodovací stromy

Rozhodovací stromy patří mezi nejznámější symbolické metody strojového učení. Umožňují vytvářet modely pro klasifikační a regresní úlohy. Při vytváření rozhodovací stromu se postupuje principem *rozděl a panuj*. Rozhodovací strom je tvořen uzly, které jsou dvojího typu (listové a testovací uzly) a hranami. Každý testovací uzel uvnitř stromu obsahuje test na jeden atribut a hrany představují jeho hodnoty. Jinak řečeno se dá říct, že rozhodovací strom je tvořen testy na atributy, které odpovídají programátorské podmínce *if...then...else*. Hrany vycházející z testovacích uzlů jsou ohodnoceny hodnotou příslušných atributů. Každý listový uzel reprezentuje rozhodnutí, například ano/ne.

Mezi základní algoritmy k vytváření rozhodovacích stromů patří ID3 a TDIDT (Top Down Induction of Decision Trees) [1, 14].

Rozhodovací stromy se například používají v bankovnictví jako na obrázku 4.3, kdy se podle několika kritérií rozhodne, jestli banka může klientovi poskytnout kreditní kartu.

Více se o rozhodovacích stromech dočtete v [1, 14].



Obrázek 4.3: Příklad rozhodovacího stromu pro schválení žádosti o kreditní kartu.

4.2 Genetické algoritmy

Genetický algoritmus (GA) je metoda, která se snaží napodobovat proces přirozené evoluce, kterou popsal Charles Darwin (1858) ve své teorii o vývoji druhů a přirozeném výběru. Základní myšlenkou je napodobit vývoj živého organismu a podle něho vytvořený algoritmus aplikovat při řešení složitých problémů, kde dochází i k změnám prostředí, na které člověk nedokáže vhodně reagovat. Podobně jako umělé neuronové sítě jsou GA založeny na biologické metafoře, kdy učení chápou jako soutěž mezi populací vyvíjejících se jedinců.

GA se podle [1, 9] řadí do skupiny *evolučních algoritmů*, které se obecně používají k řešení složitých optimalizačních úloh a vyhledávacích problémů pomocí technik inspirovanými evolucí (dědičnost, mutace, selekce, křížení a mnoho dalších). Také jsou schopny pracovat s velkým množstvím dat a najít v nich informace, které by jinak byly obtížné nalézt.

Vznik prvních GA se datuje do 60. let dvacátého století, kde jejich průkopník John Holland z Michiganské univerzity poprvé v [8] popsal základní principy a vlastnosti těchto algoritmů. Při svém výzkumu Holland definoval 2 cíle. Prvním z nich bylo zlepšit porozumění procesu přirozené adaptace a za druhé navrhnout systém s podobnými vlastnostmi jako mají systémy přírodní.

V GA se informace nebo vlastnosti reprezentují pomocí genetického kódu jedince. Tento kód může být reprezentován různými způsoby, v závislosti na tom, jaký problém řeší. Může se jednat například o bitovou posloupnost čísel.

Následující pojmy jsou nezbytné pro další pochopení základních principů GA:

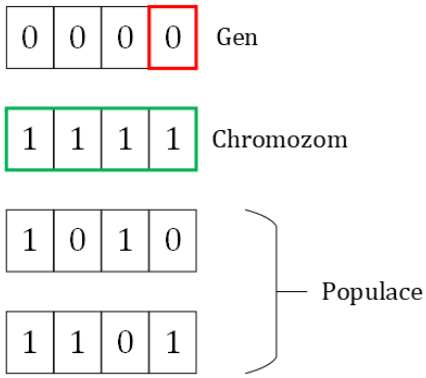
- gen
- chromozom
- populace
- fitness hodnota

Gen je nejmenší část *chromozomu*, která je dále už nedělitelná. *Chromozom* je soubor informací, které v sobě nese každý jedinec. Nejčastěji jsou tyto informace reprezentovány jako řetězce nul a jedniček, ale může jít také o matice, vektory nebo křivky.

Dalším důležitým pojmem je *populace*. *Populaci* můžeme chápat jako skupinu jedinců v rámci jedné *generace*. Všechny tyto zmiňované pojmy jsou na obrázku 4.4 znázorněny.

Fitness hodnota je číslo, které vyjadřuje kvalitu každého jedince. Pro každý problém je potřeba sestavit fitness funkci, která jako výsledek vrací požadovanou číselnou hodnotu. Fitness funkce hodnotí, jak dobře přispěje jedinec k další generaci řešení.

Podrobněji jsou dále GA popsány v [1, 8, 9].



Obrázek 4.4: Reprezentace genu, chromozomu a populace.

4.2.1 Operace

Jak již bylo zmíněno, GA využívají techniky podobné těm, které se vyskytují při přirozeném vývoji. Jsou jimi tyto operace:

- křížení (crossover)
- mutace (mutation)
- selekce (selection)
- inverze (inversion)
- výměna (exchange)

Z těchto vyjmenovaných technik se však řadí v [8] mezi základní a nejčastěji používané jen *křížení*, *mutace* a *selekce*. Zbylé operace se využívají jen zřídka. Tyto operace se aplikují nad celou generací a výstupem je nová generace. Dokud se nenaleznou jedinci s námi požadovanými vlastnosti, tak se tyto kroky opakují.

V následujících podkapitolách si popíšeme zmíněné 3 základní operace používané k nalezení vhodných jedinců.

4.2.1.1 Selekce

Tato technika se využívá k výběru nejlepších jedinců, kteří se mohou stát rodiči pro další generaci. Existuje zde několik metod selekce. Jedná se o ruletovou, turnajovou selekci, metodu ořezáním a náhodným výběrem. Většina těchto metod pracuje s kvalitou jedince (fitness hodnota). Každý jedinec má svoji fitness hodnotu jeho chromozomu.

- **Vážená ruleta** je metoda, kdy každý jedinec má šanci být vybrán pro příští generaci, která je úměrná jeho fitness hodnotě. Každý jedinec dostane podíl na pomyslné ruletě podle své fitness hodnoty. Poté je prováděno losování pro výběr nových rodičů.

- **Turnajová selekce** vybírá náhodně z populace skupinu jedinců. Minimálně musí však vybrat 2 a z nich následně vybere nejlepšího jedince s nejvyšší fitness hodnotou.
- **Ořezáním** funguje tak, že se všichni jedinci v populaci seřadí podle fitness hodnoty a pak jsou vybráni ti nejlepší podle zvoleného parametru.
- **Náhodný výběr** je nejjednodušší metodou, která nijak neřeší hodnoty fitness a náhodně vybírá z celé populace.

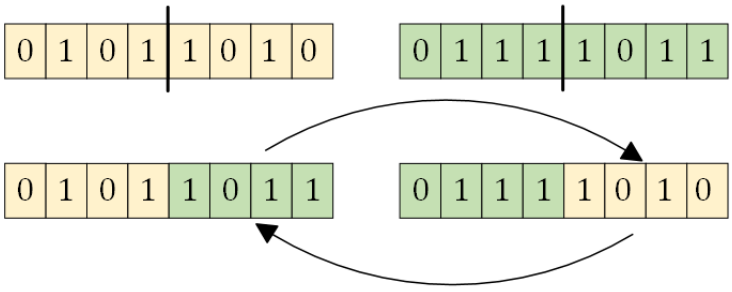
Podle [8, 9] je selekce důležitou technikou, protože nám umožňuje přenést nejlepší vlastnosti rodičů na své potomky, a tím přispět k postupnému zlepšování kvality populace.

4.2.1.2 Křížení

Křížení je operace, která umožňuje kombinovat vlastnosti rodičů, a tím vytvářet nové jedince, kteří jsou kombinací genů svých rodičů. Přesně řečeno vezmeme chromozomy dvou rodičů, vyměníme mezi nimi jejich části, čímž nám vzniknou dva noví potomci. Tento proces je inspirován přirozeným křížením organismů.

Existuje několik metod křížení, které se využívají v GA [8, 9]. Mezi ně patří například jednobodové a vícebodové křížení. Jednobodové rozdělí chromozom na dvě části a ty se mezi potomky vymění. U vícebodového je možné provádět libovolné kombinace z více než dvou rodičů.

Na následujícím obrázku 4.5 je graficky znázorněno, jak se provede křížení nad dvěma bitovými řetězci, které budou rozděleny v jednom bodě na dvě části. Jak znázorňují šipky na obrázku, tak budou koncové části rodičů vyměněny mezi sebou, a tím dojde k vzniku nových potomků.



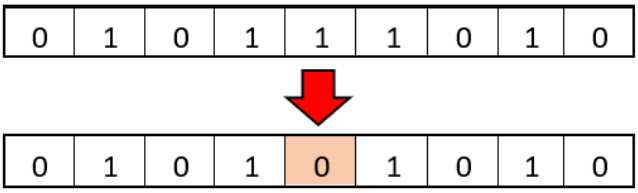
Obrázek 4.5: Příklad použití křížení na dva bitové řetězce.

4.2.1.3 Mutace

Mutace je poslední operací GA [8, 9]. Je to proces, kdy se mění genetický kód jedince. Tato technika vezme jednoho potomka a náhodně mu změní některý jeho aspekt. Může například vybrat jeden jeho bit a náhodně ho změnit z 0 na 1 nebo naopak. Mutace je důležitá, protože hned při výběru počáteční populace může být vyloučena podstatná část řešení, a tím pádem by populace jedinců

pouze reprodukovala své genetické vlastnosti a nedocházelo by k žádným změnám. S mutací se nám mohou objevit nové kombinace, které můžou mít lepší vlastnosti než původní potomci.

Na obrázku 4.6 je znázorněna mutace, kdy dochází k změně bitu 1 na hodnotu 0, a tím k vytvoření nového jedince.



Obrázek 4.6: Příklad mutace na bitovém řetězci.

4.2.2 Princip činnosti genetického algoritmu

Když je použit GA k řešení problémů, probíhá během toho několik různých fází. Jednotlivé fáze si teď popíšeme podle algoritmu 1 z [1, 9]. Prvotní fází je inicializace. Během ní se musí vytvořit počáteční populace řešení, která se skládá z jednotlivých jedinců (chromozomů). Ještě než budou jedinci použiti, tak se většinou musí přeložit jejich prezentace do podoby, která umožňuje operace kombinace (selekce, křížení, mutace). Nejčastější používanou reprezentací bývá bitový řetězec. Tímto je vytvořena počáteční generace.

Druhou fází je fitness funkce. Provede se vyhodnocení kvality každého jedince pomocí zmiňované fitness funkce. Pokud vznikl jedinec splňující požadované vlastnosti k řešení problému, tak v tomto místě algoritmus končí.

Třetím krokem je selekce. Výběr jedinců na základě fitness hodnoty, kteří předají své vlastnosti do další generace. Poté následuje operace křížení a mutace.

Posledním krokem v činnosti GA je návrat zpět k vyhodnocení fitness funkce a rozhodnutí, jestli bylo nalezeno požadované řešení. Pokud ne, tak se proces opakuje po několik generací.

Celý tento postup činnosti GA si můžeme shrnout do několika kroků:

1. Inicializace
2. Vyhodnocení fitness funkce
3. Selekce
4. Křížení
5. Mutace
6. Návrat do bodu 2.

Algoritmus 1: Obecná podoba genetického algoritmu

```
t = 0;
inicializace P(t);
while Dokud není nalezen vhodný jedinec do
    vyhodnotit fitness hodnotu jedinců v P(t);
    selekce z P(t) na základě fitness hodnoty;
    provedení operace křížení a mutace;
    nahradit původní generaci novou;
    t = t + 1;
end
```

4.2.3 Rozšíření genetických algoritmů

V předchozích částech jsme si popsali charakteristiky a principy základního modelu GA na nejnižší úrovni, který popsal Holland při svém výzkumu. Poté přišla řada odborníků (Goldberg, Mitchell, Koza), kteří se zapříčinili ke vzniku nových oblastí výzkumu jako genetické programování, umělý život, kde se techniky GA aplikují na složitější reprezentace (například části počítačového programu) a umožňuji jejich širší použití. Holland používal ve své práci jednoduchou, nízko-úrovňovou bitovou reprezentaci. Řetězec bitů složený z nul a jedniček.

Genetické programování rozšiřuje oblast GA. Je to metoda, která je schopna optimalizovat zdrojový kód programu a vytvářet nové programy. Zaměřuje se tedy na návrh a implementaci programů. Průkopníkem tohoto je John Koza (1992) [1], který přišel se svým návrhem, kde se počítačový program může vyvíjet postupnou aplikací genetických operátorů. Počátečním krokem genetického programování je vytvoření počáteční populace, která je složena z náhodně vygenerovaných programů skládajících se z částí potřebných pro nalezení řešení. Části se mohou skládat z běžných aritmetický, logických operací, matematických funkcí. Po provedení inicializace se provedou podobné kroky, které jsme si už předtím popsali. Provede se operace selekce, křížení, mutace nad kódem programu. Algoritmy pro tyto operace však musí být uzpůsobeny pro vytváření počítačových programů, protože se manipuluje s hierarchicky uspořádanými moduly. Na závěr je stanovena fitness hodnota každého programu. Například podle toho, jak si vedl dobře při řešení zadaných problémů. Tímto se určí, které programy přežijí a budou využity k vytváření nových potomků.

Genetické programování má řadu využití. Od optimalizace a tvorby algoritmů, dále využití při projektování, návrhu inženýrských nástrojů, vývoji nových léčiv, chemických látek.

Podrobněji je problematika genetického programování a dalších technik popsána v [1].

4.3 Stochastické a dynamické modely učení

V této kapitole se seznámíme s přístupem strojového učení, založeným na pravděpodobnosti. Tento přístup využívá teorii pravděpodobnosti, souhrnně Stochastiku (obor pravděpodobnosti a statistiky)

k modelování závislostí mezi různými proměnnými a k vytváření předpovědí na základě předchozích pozorování. Pomocí teorie pravděpodobnosti můžeme určit pravděpodobnost výskytu jevů, případně popsat, jak se navzájem tyto jevy ovlivňují.

K použití pravděpodobnosti v oblasti strojové učení, k pochopení a předvídání jevů existují 2 hlavní důvody [1]. Prvním z nich je modelování složitých vztahů v měnícím se světě, které se nejlépe zachycují pomocí stochastických modelů a za druhé události mohou být skutečně pravděpodobnostně spolu provázány. Díky toho měla pravděpodobnost a stochastika značný vliv na návrh a účinnost algoritmů strojového učení.

Součástí pravděpodobnosti existují dva druhy modelů učení [1]. Prvním z nich jsou *Stochastické modely* učení, které se zabývají náhodností a pravděpodobnostním rozdělováním na základě dat. Jsou schopny odhadnout pravděpodobnost několika možných událostí. Druhým typem jsou *Dynamické modely* učení zabývají se učením, které se mění v průběhu času (přizpůsobit se změnám na základě času).

Konkrétně mezi modely učení, které využívají pravděpodobnostní přístup můžeme zahrnout například *Bayesovské sítě* a *Markovovy Modely*.

Modely učení, založené na pravděpodobnosti, se používají v řadě oblastech. Příkladem může být zpracování řeči, analýza jazyka, kde je pomocí pravděpodobnosti vybírán nejlepší možný překlad nebo nejlepší stavba slov pro daný kontext. Dále při diagnostice zdravotního stavu může být pravděpodobnost využita pro analýzu dat a následné predikci vzniku nemocí. Tímto je možné u pacientů předpověď podle jejich zdravotních anamnézy, jaké onemocnění se u nich mohou objevit.

Jednou z hlavních nevýhod pravděpodobnosti ve strojovém učení je podle [1] jejich výpočetní náročnost. Jejich další nevýhodou jsou vyšší nároky na trénovací množinu dat než běžné metody strojového učení.

4.3.1 Bayesovské sítě

Bayesovské sítě jsou pravděpodobnostní modely využívající grafovou reprezentaci [1]. Podporují interpretaci nových zkušeností na základě dříve naučených vztahů (porozumění současným událostem je funkcí naučených jevů z předchozích událostí). Bayesovská sít je orientovaný acyklický graf (v grafu neexistuje cyklus), který pomocí hran zachycuje závislosti mezi proměnnými a systémem náhodných pravděpodobnostních distribucí. Každý uzel má přiřazenou pravděpodobnostní distribuci (podle Bayesova pravidla) a ta popisuje pravděpodobnostní závislost mezi vybranými proměnnými [12].

Bayesovo pravidlo je věta teorie pravděpodobnosti, kterou využívají Bayesovské sítě při stanovení hodnot pravděpodobnostní distribuce na základě hodnot jiných uzlů (proměnných) v sítí. Tento princip popisuje, jak se pravděpodobnost jedné události mění v závislostí na jiné. Tento vztah lze zapsat takto:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)} \tag{4.2}$$

- P(A|B) je podmíněná pravděpodobnost jevu A za předpokladu, že nastal jev B.
- P(B|A) je podmíněná pravděpodobnost jevu B podmíněná výskytem jevu A.
- P(A) a P(B) jsou pravděpodobnosti jevů A a B.

Proces učení těchto sítí se skládá z 2 hlavních kroků. Jde o strukturální a parametrické učení. Strukturální učení rozhoduje, které proměnné budou součástí sítě a jaké budou mezi nimi závislosti, zatímco parametrické učení řeší pravděpodobnostní rozdělení pro každou proměnnou v síti.

Bayesovské sítě mají řadu využiti, jedním z nich bývá diagnostika a predikce, například v medicíně. Hlavní výhodou Bayesovských sítí je jejich schopnost zpracovávat složité vztahy mezi proměnnými, práce s neúplnými daty.

Teorie bayesovských sítí je dále popsána v [1, 6, 12, 13].

4.3.2 Markovovy modely

Jedná se o pravděpodobnostní modely, které popisují náhodný proces měnící se v závislosti na čase. Základem těchto modelů je vlastnost *Markov Property*, která se odborně označuje jako *Markovova vlastnost* [1, 10, 11]. Vlastnost obecně říká, že pravděpodobnost přechodu z jednoho stavu do následujícího závisí pouze na aktuálním stavu, a ne na předchozích stavech. Nezáleží tedy na tom, jak jsme se do aktuálního stavu dostali, ale pouze jen na stavu bezprostředně předcházejícím. Není potřeba si pamatovat historii, stačí jen aktuální stav. Díky tomu je možné reprezentovat procesy jako konečné stavové automaty, kde ke každé hraně je připsána její pravděpodobnost. Existují i Markovovy modely vyšších řádů, ve kterých závisí na konkrétním počtu předcházejících stavů.

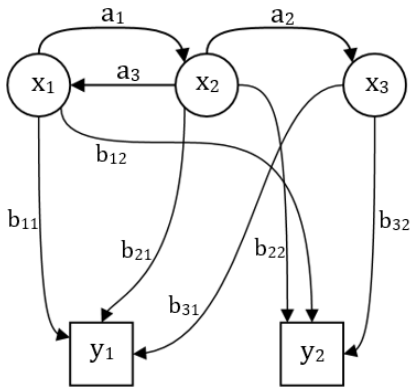
Existuje několik druhů Markovových modelů, můžeme zmínit tyto:

- Markovův řetězec
- Skryté Markovovy modely

Skryté Markovovy modely nebo zkráceně HMM jsou zobecněním Markovova řetězce. V Markovově řetězci reprezentoval každý stav jednu pozorovatelnou událost (například počasí v určitou dobu). HMM je tvořen 2 druhy stavů, prvním z nich jsou skryté stavy, které nejsou viditelné a druhým typem stavů, které jsou pozorovatelné stejně jak u Markovova řetězce. HMM lze popsat jako pozorovatelný stochastický proces pozorovaný dalším skrytým procesem.

Na obrázku 4.7 je znázorněn HMM se svými skrytými stavy x_n, pravděpodobnostními přechody a_n, pozorovatelnými výstupními stavy y_n a jejich výstupními přechody b_n.

Aplikaci HMM lze demonstrovat na problému uren, které obsahují míčky. Urny jsou umístěny v místnosti, kam nemá pozorovatel přístup a jen vidí posloupnost vytažených míčku, ale neví, ze kterých uren byly vytaženy. Markovovy modely se používají v mnoha oblastech. Můžeme se s nimi setkat například v biologii, kde se s jejich pomocí analyzují a modelují DNA sekvence. Používají se také při analýze, generování hlasu nebo psaného textu.



Obrázek 4.7: Příklad skrytého Markovova modelu.

Problematika Markova modelu je dále podrobněji popsána v [1, 10, 11].

4.4 Umělé neuronové sítě

Umělá neuronová sít (ANN) je modelem, který je inspirován nervovou soustavou, snaží se o napodobení činnosti lidského mozku. Je to jeden z přístupů, kde se uplatňuje biologie, podobně jako při GA. Někdy jsou ANN označovány jako konekcionistické nebo paralelně distribuované systémy (PDP) [1]. Slovo paralelně se zde aplikuje, protože ANN je složena z vrstev propojených umělých *neuronů*, kde každý neuron ve vrstvě zpracovává své vstupní data současně a nezávisle na ostatních. Vazby neboli spojení mezi neurony jsou reprezentovány vahami, které se průběžně upravují podle toho, jestli neuron vede k správné nebo špatné odpovědi. Neuron je tedy základním stavebním kamenem (základní výpočetní jednotka) při tvorbě ANN a v pozdější části si ho rozebereme více.

Kromě vlastností neuronu je ANN dále charakterizována globálními vlastnostmi jako je topologie sítě, algoritmus učení, schéma kódovaní.

Nejčastějším použitím pro ANN přístup je klasifikace, rozpoznávání vzorů, predikce, optimalizace, filtrace šumu [1]. Praktickým příkladem může být rozpoznávání, obrazů nebo hlasů, překlad textu, analýza dat, predikce nemocí. Metody ANN jsou také vhodné k řešení problémů, které nedokážou symbolické přístupy vyřešit. Typicky u úloh, kde není přesně definována syntaxe nebo jsou vyžadovaný schopnosti založené na vnímání.

Výhodou ANN je jejich výpočetní výkon, díky kterému jsou schopny tréninku sítě na velkém množství dat. Další výhodou je použiti na širokou řadu úloh (klasifikace, regrese, detekce atd.). Dále dokážou v datech najít takové vzory, které člověk nebo běžné metody strojového učení nedokážou identifikovat.

Mezi nevýhody určitě patří jejich nadměrné přizpůsobení určitým datům, když jsou ANN trénovány na příliš malé sadě dat. Nastane to, že ANN bude příliš specializovanou na konkrétní sadu

dat a později nebude schopna se vyvíjet na nových datech. Dalším problémem bývá špatné nastavení jejich parametrů, kdy dojde k špatnému vytrénování sítě. V poslední řadě patří mezi zásadní problém komplexnost sítě, kdy příliš rozsáhlou síť bude velice náročné navrhnout.

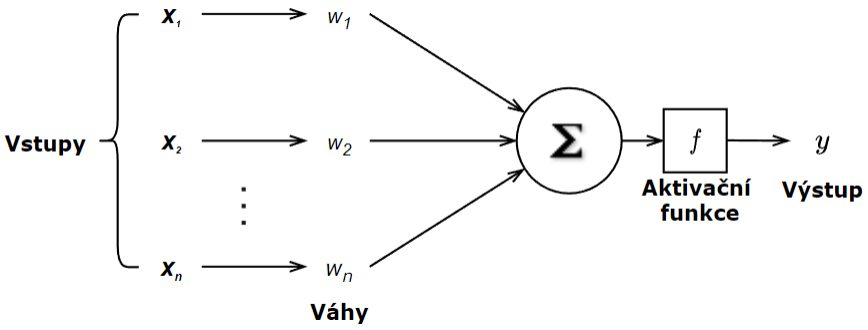
4.4.1 Model neuronu

Jedním z prvních modelů neuronu byl ten, který navrhli McCulloch a Pitts (1943) [1] a jejich model se dodnes používá. Základním prvkem ANN je neuron, který je představen na obrázku 4.8. Vstupy neuronu $x_1, x_2, ...x_n$ přijímají vstupní signály. Může jít například o již zpracovaná data předchozích vrstev neuronů nebo data ze senzorů. Podoba těchto vstupních dat bývá často ve formě vektorů, matic a je určena v závislosti na typu řešeného problému. Ke každému vstupu odpovídá jeho váha (koeficient), která popisuje jeho význam. Na základě $w_1, w_2, ...w_n$ vah se potom rozhoduje, které vstupy upřednostnit při výpočtu.

Po vstupech přichází na řadu výpočetní část neuronu, kde probíhají dva procesy, které jsou podle [15, 16, 17] popsány následovně. Matematicky je lze funkcí neuronu zapsat takto:

$$y = f(\sum_{i=1}^{n} w_i x_i - \theta) \tag{4.3}$$

Prvním z procesů je vnitřní potenciál neuronu. Vnitřní potenciál je součet vstupujících signálů škálovaných váhou spojení w_i. Prahová hodnota θ určuje, při jaké hodnotě bude neuron aktivován (probuzen). Když bude součet všech vstupů menší než prahová hodnota, tak nedojde k aktivaci a výstup zůstane nezměněn. Druhým procesem je aplikování aktivační funkce f, která vypočte výslednou výstupní hodnotu y.



Obrázek 4.8: Model umělého neuronu.

4.4.2 Návrh umělé neuronové sítě

Při návrhu ANN je potřeba dodržet několik kritérií podle toho, co přesně má síť vykonávat. Jednotlivé kritéria lze souhrne popsat těmito body:

- definování cíle
- vhodný výběr typu a struktury sítě
- nastavení parametrů
- příprava dat a trénink sítě

Prvním krokem je stanovit cíl řešení. Dále je důležitý výběr vhodného typu sítě, jestli půjde o použití perceptronu nebo vícevrstvé sítě a k tomu vhodně zvolit strukturu navrhované sítě (množství vstupů, výstupů, skrytých vrstev). Dalším důležitým požadavkem je před připravení dat a následné trénování na těchto datech. Data většinou nelze použít v surovém stavu a musíme je upravit (normalizovat) a poté rozdělit na dvě množiny - trénovací a testovací. Síť se následně trénuje na trénovací množině dat a během toho se vyhodnocuje chybová funkce, která určitě, jak dobře daná síť řeší konkrétní problém. Podle zjištěné chybovosti se upravují jednotlivé váhy.

4.4.3 Typy sítí

Jednou z prvních sítí podle [1] byl *Perceptron* a jednalo se o sít složenou pouze z jednoho neuronu. Model umělého neuronu jsme si popsali už dříve. Tato síť dokázala řešit jen jednoduché, lineárně separovatelné (binárně klasifikační) problémy. McCulloch a Pitts demonstrovali použití Perceptronu na učení booleovských funkcí AND a OR. Při použití Perceptronu na učení složitějších, lineárně neseparovatelných problémů, jakým je například booleovská funkce XOR se zjistilo, že nejde tyto problémy tímto způsobem řešit. Tohle vedlo k návrhu sítí s větší výpočetní sílou, složených z několika perceptronů tzv. vícevrstvé perceptronové sítě.

Vícevrstvá síť je složena z vstupní, výstupní vrstvy a libovolného počtu skrytých vrstev, které se starají o výpočetní část. Podle obrázku 4.9 si lze vícevrstvou síť představit jako orientovaný graf. Z každé vrstvy sítě, která obsahuje libovolný počet neuronů (uzlů) vedou vždy spojení do vyšších vrstev. Počet skrytých vrstev a počet neuronů v nich závisí na složitosti funkce, jakou má síť vykonávat a na typu sítě.

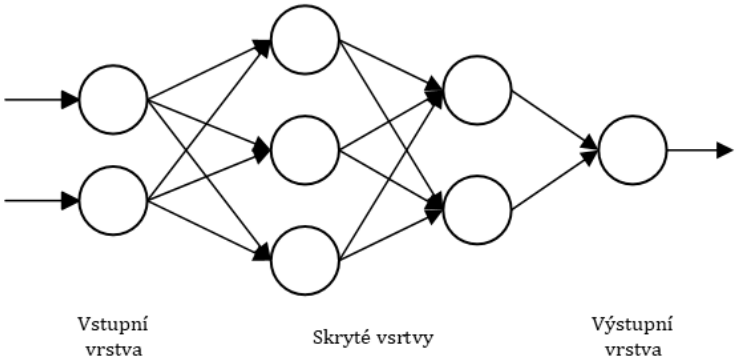
Existuje celá řada dalších druhů ANN, které se liší v jejich architektuře, algoritmech učení [18]. Dalšími typy sítí je například Hopfieldova nebo Kohoenova sít.

4.4.4 Proces učení umělých neuronových sítí

Proces učení začíná připravením dostatečné datové sady příkladů, na kterých bude ANN trénována. Kdyby nebyla množina dat dostatečně velká mohlo by dojít k tomu, že se síť příliš specializuje na konkrétní příklady a později se nedokáže vyvíjet dále nad novými příklady.

Na začátku učení se ve většině případech nastavují vstupní váhy na náhodné hodnoty.

Pro své učení podle [18] využívají ANN nejčastěji algoritmus zpětného šíření chyby (Backpropagation). Jeho cílem je minimalizovat výstupní chybovou funkci (chybu nebo odchylku), která vzniká



Obrázek 4.9: Vícevrstvá neuronová síť.

jako rozdíl mezi očekávaným výstupem (z trénovací sady dat) a výstupem z ANN. Podle této chyby se upravují jednotlivé váhy neuronů, aby se docílilo lepšího výsledku při dalším opakování učení. Proces učení se vždy provádí v několika iteracích.

Kapitola 5

Případová studie

Tato kapitola je zaměřena na případovou studii jejíž cílem byla implementace algoritmů na základě symbolické reprezentace znalostí a GA. Oba zmiňované přístupy byly popsány v předchozích kapitolách teoretické části práce.

Cílem prvního algoritmu založeném na symbolickém přístupu bylo hledat obecnou hypotézu (předpis), pomocí kterého je možné rozpoznávat objekty spadající do třídy oblouků. Algoritmus přijímá na vstupu příklady objektů, které jsou definované jako jednoduché klauzule jazyka Prolog [20] a snaží se podle příkladů oblouku vytvářet obecnou hypotézu, která bude správně rozhodovat, jestli se jedná o oblouk nebo ne.

Druhý algoritmus je zaměřen na hledání cesty v bludišti přístupem GA. Algoritmus pracuje s populací agentů, kteří zkoušejí projít bludištěm k cíli a postupně se během toho vyvíjí podle agentů, kteří byli efektivní v průchodu bludištěm. Základním principem algoritmu je genetický vývoj, kdy postupnou evolucí se agenti v populaci zlepšují až se dostanou k cílové pozici v bludišti.

Pro implementaci obou algoritmů jsem zvolil programovací jazyk Python a jeho knihovny, konkrétně ve verzi interpretu 3.11. Jeho výhodou je jeho přenositelnost, což znamená, že je spustitelný prakticky, na kterémkoliv operačním systému. Jedná se o vyšší programovací jazyk, patřící mezi skriptovací jazyky umožnující psát lépe čitelné a pochopitelné programy oproti nízkoúrovňovým programovacím jazykům jako je například jazyk symbolických adres nebo programovací jazyk C, které poskytují nízkou abstrakci a jsou snadněji převeditelné na strojové instrukce.

V následujících dvou podkapitolách budou oba zmíněné algoritmy více popsány.

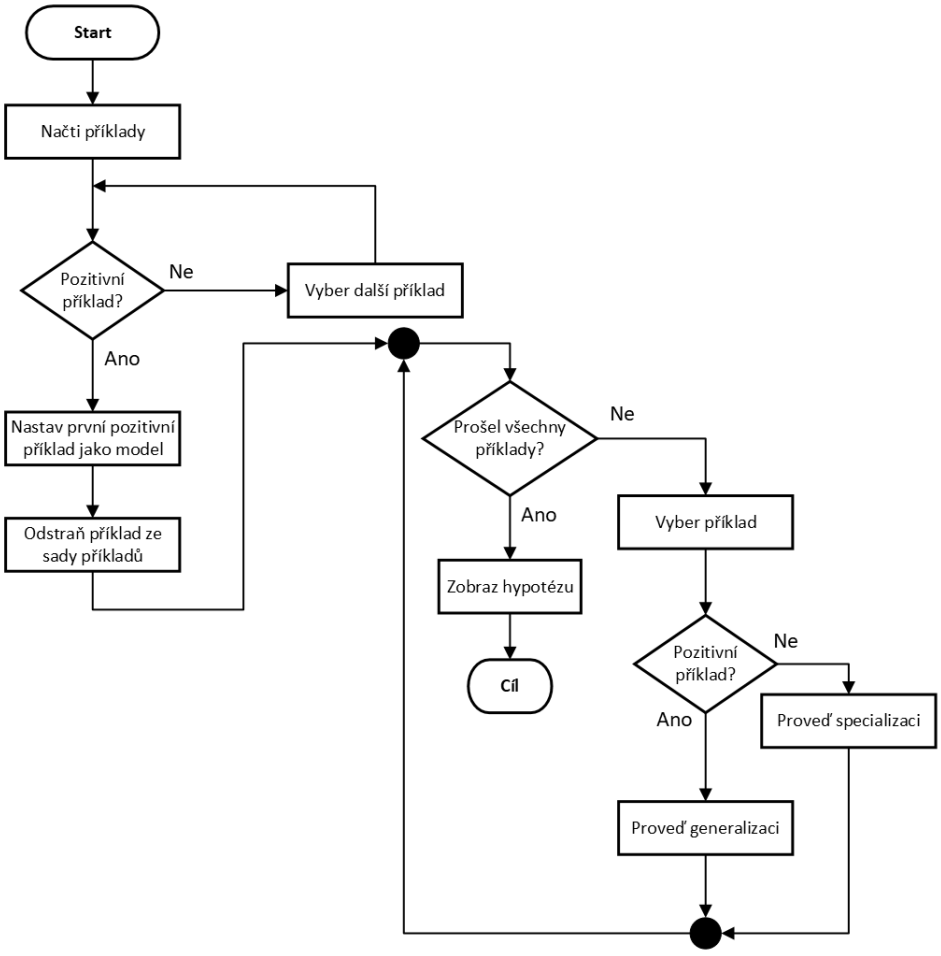
5.1 Rozpoznávání tvarů

V této části případové studie bylo za úkol na implementovat algoritmus založený na symbolické reprezentaci znalostí, která byla popsána v teoretické části v kapitole 4.1. Konkrétně se jedná o Winstonův algoritmus, který patří mezi algoritmy využívající typ učení s učitelem. Cílem algoritmu

je nalézt obecnou hypotézu, pomocí které bude možné rozpoznávat oblouk složený z trojrozměrných těles.

Algoritmus si nejprve musí zvolit nějaký příklad jako jeho počáteční hypotézu, kterou následně bude upravovat, dokud bude mít příklady pro trénování. Jako počáteční hypotézu (model) si zvolí první pozitivní příklad ze sady příkladů, která mu byla poskytnuta jako vstupní data. Následně postupně vybírá příklady ze vstupní sady příkladů a porovnává je s hypotézou. Pokud je vybrán pozitivní příklad reprezentující oblouk, tak se provádí operace generalizace, kdy se algoritmus snaží zobecňovat hypotézu, aby zahrnovala více variací oblouku. V druhém případě, když je vybrán negativní příklad, tak je prováděna operace specializace, kdy je hypotéza upřesněna, aby vylučovala příklady, které nepředstavují oblouk. Negativní příklady jsou používány proto, aby se zabránilo vytvoření příliš obecné hypotézy, která by rozpoznávala i něco jiného než jen pouze oblouk.

Takto probíhá celý postup opakovaně, dokud algoritmus neprojde všechny příklady. Na obrázku 5.1 je znázorněn vývojový diagram popisovaného Winstonova algoritmu pro rozpoznávání tvarů.



Obrázek 5.1: Vývojový diagram Winstonova algoritmu pro rozpoznávání tvarů.

5.1.1 Vstupní data

Vstupem pro tento algoritmus jsou data reprezentována pomocí jednoduchých klauzulí jazyka Prolog. Prolog je logický programovací jazyk, který se skládá z predikátů, logických výrazů a pravidel, pomocí kterých popisuje vztahy mezi různými objekty a pravdivostními hodnotami. Zápis dat pomocí jazyka Prolog byl zvolen kvůli toho, že algoritmy založené na symbolické reprezentaci znalostí pracují s daty popsanými pomocí logických výrazů a pravidel.

Podrobněji je jazyk Prolog popsán v publikaci [20].

V tomto případě obsahem vstupních dat je trénovací sada příkladů, která je uložena v csv souboru, kde jednotlivé příklady popisují různě definované oblouky a tvary podobné obloukům. Příklady jsou poté v programu načteny ze souboru a převedeny na seznam, kde každá položka odpovídá jednomu příkladu. Oblouk zapsaný pomocí jazyka Prolog je definován následovně:

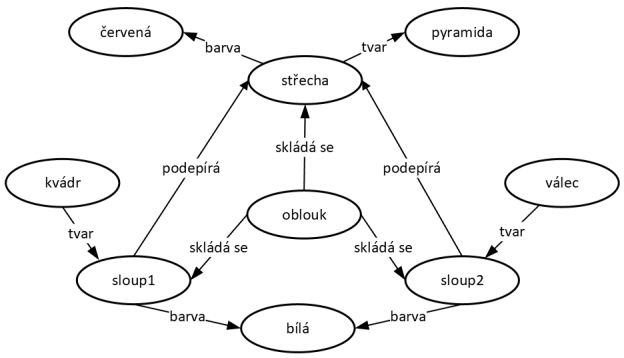
$$\begin{gathered}\text{skládá\_se(oblouk, sloup1)} \wedge \text{skládá\_se(oblouk, sloup2)} \wedge \text{skládá\_se(oblouk, střecha)} \\ \wedge \\ \text{tvar(sloup1, kvádr)} \wedge \text{tvar(sloup2, válec)} \wedge \text{tvar(střecha, pyramida)} \\ \wedge \\ \text{barva(sloup1, bílá)} \wedge \text{barva(sloup2, bílá)} \wedge \text{barva(střecha, červená)} \\ \wedge \\ \text{podepírá(sloup1, střecha)} \wedge \text{podepírá(sloup2, střecha)}\end{gathered}$$

Zápis oblouku v jazyce Prolog je tvořen pomocí jednoduchých pravidel, které jsou spojeny pomocí konjunkce. Dříve uvedený zápis, ale není zcela správný. Korektně zápis oblouku pomocí Prolog klauzulí vypadá trochu odlišně, kde konjunkce jsou nahrazeny čárkami jako v tomto případě:

$$\begin{gathered}\text{skládá\_se(oblouk, sloup1)}, \text{skládá\_se(oblouk, sloup2)}, \text{skládá\_se(oblouk, střecha)}, \\ \text{tvar(sloup1, kvádr)}, \text{tvar(sloup2, válec)}, \text{tvar(střecha, pyramida)}, \\ \text{barva(sloup1, bílá)}, \text{barva(sloup2, bílá)}, \text{barva(střecha, červená)}, \\ \text{podepírá(sloup1, střecha)}, \text{podepírá(sloup2, střecha)},\end{gathered}$$

Podle výše uvedených zápisů lze oblouk převést do grafické podoby jako je tomu na obrázku 5.2, kde je oblouk složen ze 3 těles, které mají určité vlastnosti jako je tvar, barva a čeho se dotýkají. V tomto případě *sloup1* a *sloup2* podepírají střechu, a tím vytvářejí oblouk. Samozřejmě je možné definovat další pravidla, a tím upřesnit definici oblouku. Například sloupy by měly mezi sebou mít volný prostor a mít stejnou výšku.

Stejným způsobem jsou dále popsány další varianty oblouků a jiných tvarů.



Obrázek 5.2: Grafické znázornění zápisu oblouku.

5.1.2 Operace generalizace a specializace

Jak bylo zmíněno dříve v 4.1, Winstonův algoritmus pro rozpoznávání tvarů je založen na sadě operací, které vedou k budování hypotézy během procházení všech příkladů. Mezi 2 hlavní operace, které agent používá při úpravě hypotézy je generalizace (zobecnění) a specializace (upřesnění).

Pro budování správné hypotézy je nejlepší zvolit sadu příkladů, která obsahuje jak pozitivní i negativní příklady oblouku. Pokud by byla hypotéza tvořena jen pomocí pozitivní příkladů, tak by mohlo dojít k tomu, že hypotéza bude příliš obecná a mohla by zahrnout i příklady, které nemusí odpovídat oblouku. S pomocí negativních příkladů je hypotéza upřesněna pouze na oblouky.

Specializace se používá v momentě, kdy je potřeba porovnat hypotézu s negativním příkladem, který neodpovídá oblouku. Pokud je při porovnání nalezen zásadní rozdíl mezi hypotézou a negativním příkladem, tak přichází na řadu modifikace hypotézy (v dalších částech označované jako model). Podle [5] pro modifikaci hypotézy zde existuje několik úprav:

- pokud model obsahuje pravidlo, které není součástí negativního příkladu, tak toto pravidlo označ jako povinné (**require-link**)
- pokud negativní příklad obsahuje pravidlo, které není v modelu, tak toto pravidlo přidej do modelu a označ jako zakázané (**forbid-link**)

Generalizace probíhá obdobným způsobem jako specializace jen v tomto případě se porovnává hypotéza s pozitivním příkladem. Pro každý nalezený rozdíl jsou následně prováděny tyto heuristiky[1], které modifikují hypotézu:

- pokud pro pravidlo, které má rozdílnou hodnotu modelu i pozitivním příkladu existuje společná obecná hodnota, tak tuto hodnotu použij v modelu (**climb-tree**)
- pokud obsahuje model i příklad hodnotu v pravidlu, která se vylučuje, tak toto pravidlo odstraň (**drop-link**)

[1] Jedná se o metodu používanou k řešení problémů nebo rozhodování v situacích, kdy neexistuje jednoznačné řešení.

- pokud model i pozitivní příklad obsahují ve stejném pravidlu rozdílnou hodnotu, pro kterou neexistuje žádná obecná hodnota, tak rozšiř toto pravidlo o hodnotu z příkladu(**enlarge-set**)
- pokud je v modelu pravidlo, které chybí v pozitivním příkladu, tak ho odstraň, protože je nepotřebné (**drop-link**)
- pokud se model a příklad liší v číselné hodnotě nebo intervalu hodnot, tak vytvoř interval hodnot nebo tento interval modifikuj (**close-interval**)

Všechny zmíněné heuristiky (úpravy) si podrobněji popíšeme v dalších částech práce.

5.1.2.1 Require-link

Cílem této heuristiky je označit v modelu pravidla, které chybí v porovnávaném negativním příkladu. Z takto označených pravidel se následně stávají povinná pravidla, které musí být součástí modelu, aby bylo možné správně rozpoznávat oblouk. Nelze je odstranit, ale je možné na ně aplikovat heuristiky zaměřené na generalizaci. Označení pravidla jako povinného může být realizováno například použitím prefixu *must-be*.

5.1.2.2 Forbid-link

V této heuristice jde o podobný princip jako v Require-link. Jen je zde rozdíl v tom, že se používá ve chvíli, kdy v modelu chybí pravidlo, které je v negativním příkladu. Toto chybějící pravidlo je následně přidáno do modelu a označeno opět prefixem (*must-not-be*) jako zakázané.

Pokud při testování agent narazí na příklad oblouku, který obsahuje pravidlo, které je zakázané, tak bude o něm rozhodnuto, že se nejedná o oblouk.

5.1.2.3 Climb-tree

Někdy nastane situace, kdy jsou pravidla v modelu příliš specifická a je potřeba je zobecnit. Úkolem této heuristiky je najít pro rozdílné hodnoty stejného pravidla nějakou obecnou hodnotu, kterou by bylo možné upravit pravidlo a udělat, tak model obecnější. Dalo by se říct, že máme nějakou abstraktní třídu, ze které dědí několik potomků. Například máme abstraktní třídu Tvar, ze které dědí obdélník, kruh, trojúhelník a mnoho dalších geometrických tvarů.

5.1.2.4 Enlarge-set

Tato heuristika je používána v případech, kdy máme v modelu a pozitivním příkladu stejné pravidlo s rozdílnou hodnotou, pro které neexistuje žádná společná obecná hodnota. Je nutné modifikovat pravidlo tak, aby registrovala i jinou hodnotu než, která je momentálně nastavena v modelu. Řešením je použít disjunkci, pomocí které se přidá další hodnota do pravidla.

5.1.2.5 Drop-link

Odstranění pravidla z modelu je potřeba, když součástí modelu je pravidlo, které není v pozitivním příkladu nebo pokud se dvě související hodnoty pravidla mezi sebou vylučují.

Nejčastěji nastává situace, kdy je právě odstraněno pravidlo z modelu, které je nepotřebné k rozpoznávání. Příkladem může být barva, která je pro střechu definována v modelu, ale v následujícím pozitivním příkladu toto pravidlo chybí. Dojde k jejímu odstranění v modelu, protože je zanedbatelné, jakou bude mít střecha barvu.

V kódu 5.1 je ukázka, jak je kontrolováno pravidlo modelu a negativního příkladu, jestli neobsahuje vylučující se hodnoty. Pokud by takové hodnoty obsahoval, tak je vrácena hodnota **True** a bude pravidlo odstraněno z modelu.

```
# prochází slovník obsahující pravidlo a k němu exclude hodnoty
for ex in agent.exclude_link_dict:
    # klíč slovníku je složen z názvu pravidla a k čemu patří
    ex_split = ex.split('-')
    # pokud je dané pravidlo ve slovníku
    if ex_split[0] in link.property_name and ex_split[1] in link.property_name:
        # pokud je hodnota z modelu a z příkladu ve slovníku, tak vrátí True
        if link.property_value in agent.exclude_link_dict[ex] and
            example_link_value in agent.exclude_link_dict[ex]:
            return True
    return False
```

Zdrojový kód 5.1: Nalezení vylučujících se hodnot v pravidle.

5.1.2.6 Close-interval

Tato heuristika je používaná v případech, kdy pravidla v modelu i pozitivním příkladu obsahují číselné hodnoty nebo intervaly. Pokud je nalezeno pravidlo, které se pouze liší v číselných hodnotách, tak je z těchto hodnot vytvořen interval. V případě, že některé pravidlo v modelu již obsahuje interval, tak je pouze tento interval upraven, tak aby zahrnoval novou hodnotu z pozitivního příkladu. V kódu 5.2 je ukázka vytvoření intervalu hodnot pro pravidlo v modelu.

```
for link in example:
    # pokud je nalezeno v příkladu stejné pravidlo jako je modelu
    if l1.property_name in link:
        # získání hodnoty z pravidla příkladu
        e_property_value = link_get_value(link)
        # pokud hodnota z modelu < hodnota z příkladu, vytvoř interval
        if l1.property_value < e_property_value:
```

```
            self.model = list(map(lambda x: x.replace(l1.link, 'must-be-' + l1.
                property_name + ',' + l1.property_value + '-' + e_property_value +
                ')'), self.model))
        # pokud hodnota z modelu > hodnota z příkladu, vytvoř interval
        elif l1.property_value > e_property_value:
            self.model = list(map(lambda x: x.replace(l1.link, 'must-be-' + l1.
                property_name + ',' + e_property_value + '-' + l1.property_value +
                ')'), self.model))
        break
```

Zdrojový kód 5.2: Vytvoření intervalu v pravidlu modelu.

5.1.3 Výsledek

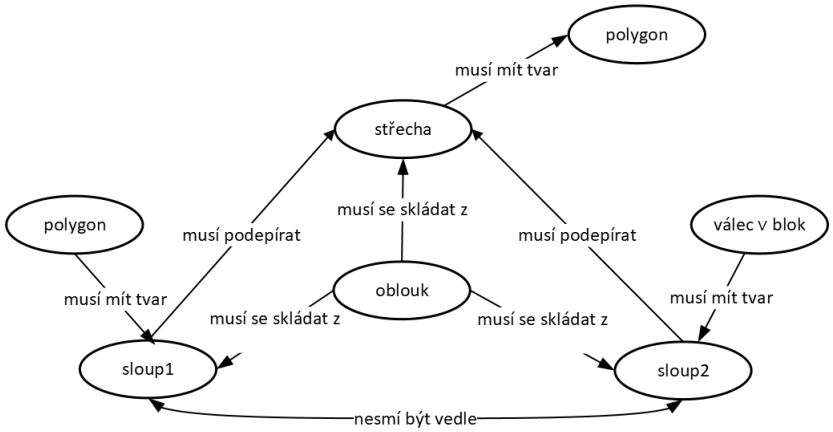
Výsledkem této části případové studie byl Winstonův algoritmus pro rozpoznávání tvarů, konkrétně rozpoznávaní oblouku. Algoritmus vytváří obecnou hypotézu neboli předpis pomocí příkladů, které jsou mu postupně předkládány. Na vstupu algoritmu byla trénovací datová sada složená z 20 příkladů popsaných pomocí jazyka Prolog. Polovina ze všech příkladů popisovala oblouk a ostatní příklady popisovaly jiný tvar. Součástí vstupu byly také informace o vylučujících se hodnotách v pravidlech. Následně se podle nalezené hypotézy testují vybrané příklady z testovací sady příkladů, jestli je dokáže hypotéza správně identifikovat. Testovací sada příkladů se liší od trénovací sady příkladů v tom, že algoritmus neví, zda mu byl předložen pozitivní nebo negativní příklad. Na obrázku 5.3 je představen příklad jednoduché hypotézy, která rozpoznává oblouk.

Testování jsem provedl náhodně pro celou sadu testovacích příkladů, kdy jsem postupně vybíral příklady a porovnával je s hypotézou, jestli jednotlivé pravidla definované v hypotéze odpovídají vybranému příkladu. Součástí testování je kontrola toho, jestli příklad obsahuje všechny povinné pravidla a neobsahuje pravidla, které nesmí být v příkladu. Dále je prováděna kontrola hodnot pravidel. Pokud se jedná o číselnou hodnotu, tak se kontroluje, jestli například patří do intervalu, který je definován v hypotéze. Dále je prováděna kontrola pro obecné hodnoty, jestli hodnota příkladu a modelu spadá pod nějakou obecnější hodnotu, která je společná pro oba. Poté co jsou všechna pravidla ověřena, tak je zobrazen výsledek s informací, jestli se jedná o oblouk nebo ne. Tento výsledek je na závěr porovnán s očekávaným výsledkem pro daný příklad. Algoritmus je ve funkčním stavu, schopný rozpoznávat v této chvíli tvary.

Hlavní výhodou symbolické reprezentace znalostí je její relativně jednoduchá čitelnost a pochopitelnost oproti reprezentaci dat v ANN nebo GA, kde o datech prakticky nic nevíme. Data jsou reprezentována pomocí symbolů a logických pravidel, jak bylo dříve popsáno v kapitole 4.1. Vytváří, tak popis nějakého objektu, v tomto případě popis oblouku, který je definován několika vlastnostmi.

Podle [3] patří mezi nevýhody symbolického přístupu citlivost na šum v datech. Pokud je hledána hypotéza na trénovací sadě dat, která obsahuje špatně definována nebo neúplná data, tak

může nastat případ, kdy výsledná hypotéza nebude korektně rozpoznávat objekty. Například bude hypotéza příliš specifická nebo v druhém případě příliš obecná, kvůli toho, že v datech chyběly nebo byly navíc pravidla, které by omezily množinu možných případů daného objektu.



Obrázek 5.3: Příklad nalezené výsledné hypotézy.

5.2 Hledání cesty v bludišti

Cílem tohoto algoritmu je nalezení cesty mezi dvěma body v dvourozměrném bludišti libovolné velikosti na základě GA, které byly popsány v kapitole 4.2. Základem algoritmu je počáteční populace agentů, kde každý agent obsahuje náhodně vygenerované pole (chromozom), které určuje, jak se má v bludišti postupně pohybovat. Jednoduše řečeno, agent vykonává vygenerované kroky, dokud nerazí do zdi, nenajde cílovou pozici nebo pokud se nepokusí vrátit zpět na pozici, kterou už jednou navštívil. V kódu A.1 je ukázka toho, jak je kontrolováno, jestli je možné udělat následující krok a jeho následné provedení agentem. Z části kódu si můžete všimnout, že je zabráněno zpětnému pohybu agenta tím, že se kontroluje, jestli nová pozice agenta není součástí seznamu, který ukládá již navštívené pozice. Pokud tato situace nastane, tak se agent nezastaví, ale pokusí se najít jiný možný krok a ten provede. Tím se docílí toho, že bude agent více prozkoumávat bludiště než, aby se hned zastavil.

Když agenti skončí s pohybem po bludišti, tak přichází na řadu vyhodnocení fitness hodnoty každého agenta. Hodnota fitness je měřítkem toho, jak úspěšný byl agent v procházení bludištěm.

K tomu, aby bylo možné získat ohodnocení každého agenta je potřeba fitness funkce, která vyhodnocuje jak blízko se agent dostal k cíli. Existuje celá řada metrik zabývajících se vzdáleností, jak jako je například Euklidova nebo Manhattanská vzdálenost, které se vyjadřují vzdáleností mezi dvěma body. Pro tento problém jsem zvolil Manhattanskou vzdálenost. Vzdálenost je v tomto případě odvozována podle pravoúhlého systému ulic ve městě New York. Výpočet této metriky je

definován jako součet absolutních hodnot rozdílů souřadnic bodů a lze ji zapsat takto:

$$d_{Manhattan}(A, B) = |x_1 - x_2| + |y_1 - y_2| \tag{5.1}$$

Tento druh vzdálenosti jsem vybral kvůli toho, že se agenti pohybují po čtvercové mřížce a vzdálenost mezi dvěma body bude představovat počet dílků mezi nimi.

Poté co je pro všechny agenty z populace vypočtena fitness hodnota, tak dochází k selekci, křížení a mutaci nad populací agentů. Během operace selekce jsou nejdříve všichni agenti v populaci seřazeni od nejmenší po největší podle jejich fitness hodnoty, která udává, jak daleko jsou od cílové pozice. Následně je vybráno z populace několik lepších agentů, kteří jsou použiti ke křížení s ostatními agenty. Vybere se lepší agent a provede křížení se všemi horšími agenty s tím, že je eliminována možnost provést křížení na stejném agentovi.

Jak už název naznačuje operace křížení se snaží vzít dva jedince jako rodiče a vytvořit z nich nového, lepšího jedince, který dědí informace z obou rodičů. V tomto případě, nový agent získá pole kroků, které je částečně složené z jeho rodičů. Z kódu 5.3 si můžete všimnout, že je nejprve náhodně vygenerováno celé číslo, podle kterého jsou pole rodičovských agentů rozdělena na dvě části a následně spojena do nového pole jako na obrázku 4.5 s tím, že bude ve většině případů zvýhodňován první agent. Nově vytvořené pole je následně uloženo do seznamu polí, do kterého se postupně ukládají nově vytvoření potomci. Takto probíhá proces křížení dokud není vytvořen stejný počet potomků jako bylo agentů.

```
def crossover(array1, array2):
    # vygenerování náhodné pozice k rozdělení pole
    cut_point = random.randint(0, len(array1) - 1)
    # 95% času se bude podmínka splněna
    if random.random() <= 0.95:
        # vzniká nové pole složené z částí array1 a array2
        new_arr = np.concatenate((array1[:cut_point], array2[cut_point:]))
    else:
        # vzniká nové pole složené z částí array2 a array1
        new_arr = np.concatenate((array2[:cut_point], array1[cut_point:]))
    return new_arr
```

Zdrojový kód 5.3: Křížení dvou agentů.

Dále je na nutné provést mutaci nad potomky, kteří prošli křížením. Účelem mutace je zavést do populace rozmanitost. Mutace umožňuje například pomocí náhodné změny některých hodnot přinést do populace nová řešení, která by nemohla vzniknout jen pomocí selekce a křížení. Pro mutaci jsem zvolil postup, kdy si nejprve stanovím pro kolik agentů z populace bude mutace prováděna, a náhodně nechám vybrat, na kterých pozicích přesně se má v seznamu obsahující všechna pole

kroků provést mutace. Poté už jen přepočítám pozici na konkrétního agenta, na kterého má být aplikována mutace a na pozici v jeho poli směrů, která má být změněna. Pomocí mutace je možné zabránit uvíznout populaci v místech, které si myslí, že jsou vhodné pro procházení bludiště, ale nevedou k žádnému lepšímu výsledku. Například se může jednat o mrtvé konce v bludišti, které mohou být optimálními řešeními, ale vedou k uvíznutí.

```
    total_elements = array.size
    amount_to_mutate = int(total_elements * MUTATION_RATE)
    indices = random.sample(range(total_elements), amount_to_mutate)

    # pro každou indices najdi pozici v poli a proved mutaci
    for i in indices:
        # vypočte řádek
        agent_row = i // NUM_MOVES
        # vypočte sloupec
        turn_column = i % NUM_MOVES
        current_value = array[row, column]
        # uloží na vybranou pozici jinou hodnotu pohybu
        array[row, column] = random.choice(DIRECTION_OPTIONS[DIRECTION_OPTIONS !=
            current_value])
```

Zdrojový kód 5.4: Ukázka mutace agentů.

Po dokončení křížení a mutace přichází na řadu poslední část algoritmu, kdy je vytvořena nová generace agentů, která je předvedena na kódu 5.5. Nejprve jsou nastaveny počítadla, které zaznamenávají aktuální tah a počet již vytvořených generací. Dále je vytvořena nová populace agentů, kterým je přiřazeno pole kroků ze seznamu, který obsahuje pole pro jednotlivé agenty.

```
def next_generation(self, moves_lists):
    # nastavení počítadel
    self.turn = 1
    self.generation += 1
    # vytvoření nových agentů
    self.agents = [Agent() for _ in range(NUM_AGENTS)]
    # přiřazení agentům pole tahů z moves_lists
    for i, m in enumerate(moves_lists):
        self.agents[i].move_array = m
    print(f"## {self.generation} generation created ##")
```

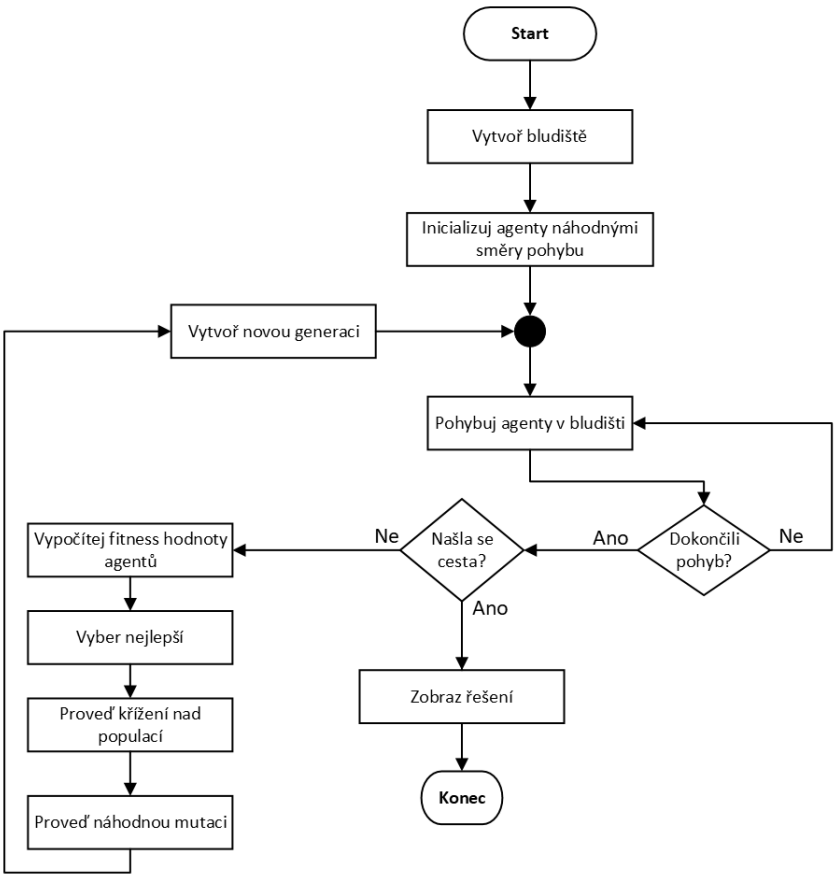
Zdrojový kód 5.5: Ukázka vytváření nové generace agentů.

Popisovaný algoritmus se následně opakuje, dokud určité procento agentů nenalezne cílovou pozici nebo skončí, když do definovaného počtu generací nenajde žádný agent řešení.

Na závěr, když je nalezena cesta bludištěm, tak je na obrazovku do nového okna vykreslena výsledná cesta jednoho z agentů.

Součástí programu jsou globální parametry, které slouží k nastavení bludiště (např. jeho velikost) nebo k definování počtu agentů, počtu kroků nebo třeba maximálního počtu generací k nalezení řešení.

Zjednodušený vývojový diagram celého programu je znázorněn na obrázku 5.4.

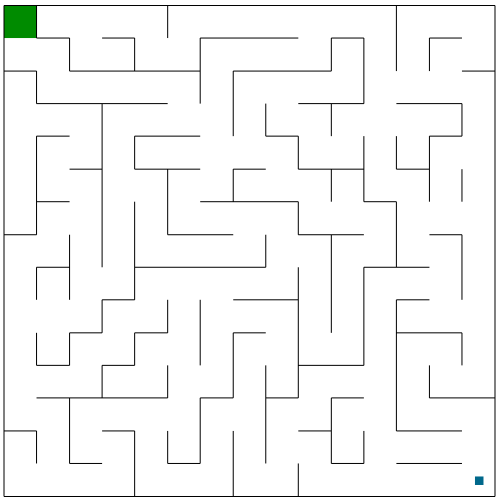


Obrázek 5.4: Vývojový diagram programu hledající cestu v bludišti.

5.2.1 Vytváření bludiště

Pro generování bludiště a jeho vykreslování na obrazovku jsem použil modul pyamaze, který slouží k jednoduchému generování náhodného bludiště. Modul je postaven na frameworku Tkinter, který se používá k tvorbě grafických uživatelských rozhraní a s jeho pomocí je schopen vygenerované bludiště včetně nalezené cesty zobrazit na obrazovku. Součástí modulu jsou před připravené metody a funkce, kterými je možné ovlivnit vzhled, ale také nastavovat parametry generovaného bludiště. Umožňuje

například generovat bludiště libovolné velikosti a složitosti. Na obrázku 5.5 je ukázka náhodně vygenerovaného bludiště pomocí modulu pyamaze, na kterém je zelenou barvou vyznačena cílová pozice v bludiště a modře znázorněna startovní pozice pro hledání cesty v bludišti. Kompletní dokumentaci modulu pyamaze lze najít na [19].



Obrázek 5.5: Ukázka náhodně vygenerovaného bludiště pomocí modulu pyamaze.

5.2.2 Reprezentace vstupních dat

Důležitým rozhodnutím při implementaci algoritmu byla volba toho, v jaké podobě budou data popisující, jakým směrem se má agent v bludišti pohybovat. Proto jsem se rozhodl reprezentovat jeden krok pohybu jedním ze 4 základních směrů podle světových stran (sever, jih, východ a západ). Hlavním důvodem, proč jsem zvolil tento způsob reprezentace byl kvůli toho, že modul pyamaze umožňuje pomocí základních směrů světových stran provádět pohyb hráče po bludišti a za druhé po vygenerování bludiště vzniká slovník všech pozic popisující, jakými směry je možné se pohybovat.

Dále je potřeba vytvořit pole, které obsahuje jednotlivé kroky. Pomocí náhodného výběru ze 4 možností popsaných dříve se pro každého agenta vytvoří počáteční pole kroků o určité délce. Vygenerované pole kroků potom vypadá jako na obrázku 5.6, kde každý krok v poli určuje jakým směrem se má agent pohybovat.

```
[ 'W', 'N', 'E', 'N', 'N', 'N', 'W', 'W', 'W' ]
```

Obrázek 5.6: Příklad vygenerované pole kroků.

5.2.3 Testování

Následující část demonstruje, jak byla testována funkčnost implementovaného programu pro hledání cesty v bludišti pomocí operací, které jsou odvozeny z genetiky. V první řadě byl program otestován na malých čtvercových bludištích o velikosti jedné strany 5 až 20 dílků. V tomto případě program splnil očekávání a nalezl cestu během několika generací. Byly vyzkoušeny různá nastavení parametrů ovlivňující funkčnost celého programu jako například počet agentů, kteří procházejí bludištěm, nebo parametry, které ovlivňují kolikrát bude prováděna mutace nad populací agentů.

Poté však s rostoucí velikostí bludiště nastaly komplikace. Problémem bylo to, že v některých případech program nedokázal najít řešení pro bludiště a agenti se postupně zasekli v mrtvém bodě. Ve většině případech se zasekli ve slepých uličkách bludiště. Z tohoto jsem následně vyvodil teorii, že při výběrů agentů ke křížení můžou být vybráni i agenti, kteří se dostali do slepých uliček, ale podle fitness hodnoty jsou blíže k cílové pozici než ostatní. Někdy je tento problém označován jako problém lokálního optima, kdy je dosaženo nejlepšího lokálního výsledku, ale ne nutně nejlepšího globálního. V tomto případě nejlepším lokálním výsledkem je cesta, která se dostal nejblíže k cíli, ale končí slepou uličkou. Jedním možných řešení, ale ne úplně korektním by bylo spustit program nad stejným bludištěm znova a předpokládat, že se nestane stejná situace jako dříve.

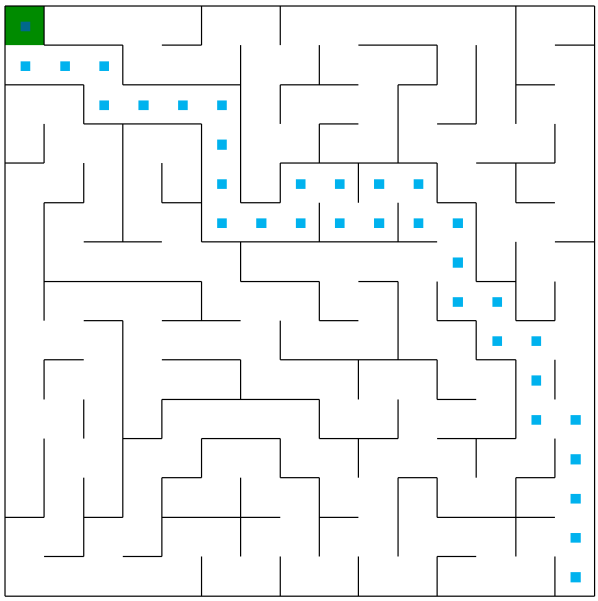
Proto jsem se nejprve pokusil tento problém eliminovat pomocí přenastavení globálních parametrů (například počet agentů, maximální počet generací, rychlost mutace), které mají vliv na algoritmus a následně modifikovat celý algoritmus. Oba způsoby, ale nevedly žádnému lepšímu výsledku. Proto jsem jako řešení zvolil generovat jednodušší bludiště, které neobsahují mnoho slepých uliček.

Po vyřešení problému popsaného dříve, program funguje správně i pro bludiště větší velikosti. Několikrát bylo provedeno testování pro různě velká bludiště až do velikosti 70x70 dílků. Na závěr jsem provedl test, při kterém se pokaždé, když byla nalezena cesta vygenerovalo nové bludiště. Takto jsem nechal program hledat cestu pro přibližně 300 bludišť, kdy pouze v jednom případě se to programu včas nepovedlo. Na obrázku 5.7 je ukázka nalezené cesty v bludišti velikosti 15x15 dílků, kde zeleně označený dílek reprezentuje cílovou pozici.

Z testování lze vyvodit několik výhod a nevýhod GA. Hlavní výhodou GA je jejich síla při řešení optimalizačních úloh a schopnost prohledávat širokou množinu možných řešení.

Mezi nevýhody bych zařadil nutnost ladění a testování různých kombinací nastavení parametrů, které ovlivňuji činnost GA. Další nevýhodou, která byla popsána během testování je uvíznutí v lokálním optimu, na které jsou tyto algoritmy citlivé. Dále jsou GA časové a výpočetně náročné. Jejich náročnost roste v závislosti na řešeném problému a velikosti populace, kdy je potřeba pro celou populaci provádět mnoho operací.

K nalezení cesty v bludišti existuje dále celá řada dalších algoritmů, které však nejsou založeny na GA. Jedná se o algoritmy, které byly konkrétně navrženy k hledání nejkratší cesty grafu. Zmínil



Obrázek 5.7: Ukázka nalezené cesty v bludišti.

bych zde například Dijkstrův algoritmus, který slouží k nalezení nejkratší cesty mezi dvěma uzly v kladně ohodnoceném grafu.

Kapitola 6

Závěr

Cílem této práce bylo nejprve shrnout teorii strojového učení v multiagentních systémech a na základě symbolické reprezentace znalostí, GA, pravděpodobnosti a ANN. Všechny tyto vyjmenované pojmy byly popsány v teoretické části mojí práce.

V případové studii byly podle zadání vytvořeny a následně popsány algoritmy založené na symbolické reprezentaci znalostí a GA. Oba vytvořené algoritmy se povedlo dovést do funkčního spustitelného stavu, na kterých lze demonstrovat charakteristiky GA a symbolické reprezentace znalostí. Prvním z nich byl algoritmus založený na symbolickém přístupu, přesněji Winstonův algoritmus, který vytváří obecnou hypotézu k rozpoznávání tvarů. Během testování GA pro hledání cesty v bludišti bylo zjištěno, že s rostoucí náročností bludiště a větším množstvím slepých uliček dochází k tomu, že se agenti přestávají dostatečně vyvíjet. Tento problém byl nakonec vyřešen zjednodušením generovaného bludiště.

Oba algoritmy jsou dále možné rozšířit. Jedním z možných příkladů rozšíření pro první algoritmus je jeho zobecnění, aby ho bylo možné použít pro rozpoznávání širší množiny problémů.

Druhý algoritmus by bylo možné rozšířit pro řešení úloh zabývajících se optimalizací a hledání nejkratší nebo nejefektivnější cesty v podobné struktuře jako je bludiště. Hlavní výhodou GA je jejich síla při řešení optimalizačních úloh.

Literatura

[1] LUGER, George F. *Artificial intelligence: structures and strategies for complex problem solving*. 6th ed. Boston: Pearson Education, 2008. ISBN 978-0-321-54589-3.

[2] KUBÍK, Aleš. *Inteligentní agenty*. Brno: Computer Press, 2004. ISBN 80-251-0323-4.

[3] POOLE, David L. a Alan K. MACKWORTH. *Artificial intelligence: foundations of computational agents*. 2nd pub. Cambridge: Cambridge University Press, 2010. ISBN 978-0-521-51900-7.

[4] MITCHELL, Tom M. *Machine learning*. Boston: McGraw-Hill, 1997. ISBN 00-704-2807-7.

[5] MENŠÍK, M., M. DUŽÍ, A. ALBERT, V. PATSCHKA a M. PAJR. Machine learning Using TIL. *Frontiers in Artificial Intelligence and Applications*. 2020, (321), 344-362.

[6] RUSSELL, Stuart J. a Peter NORVIG. *Artificial intelligence: a modern approach*. 3rd ed. Upper Saddle River: Prentice Hall, 2010. Prentice Hall series in artificial intelligence. ISBN 978-0-13-604259-4.

[7] HONZÍK, Petr. *Strojové učení* [online]. Brno, 2006 [cit. 2022-12-02]. Dostupné z: `http://vision.uamt.feec.vutbr.cz/STU/others/Honzik%20-%20Strojove_uceni_S.pdf`

[8] BHASIN, Harsh a Surbhi BHATIA. *International Journal of Computer Science and Information Technologies: Application of Genetic Algorithms in Machine learning* [online]. 2. 2011 [cit. 2023-01-16]. ISSN 0975-9646.

[9] OLIVKA, Petr. *Genetické algoritmy* [online]. [cit. 2023-01-16]. Dostupné z: `https://poli.cs.vsb.cz/edu/isy/down/ga.pdf`

[10] HLAVÁČ, Václav. *Rozpoznávání s markovskými modely* [online]. České vysoké učení technické v Praze, institut informatiky, robotiky a kybernetiky. [cit. 2023-01-24]. Dostupné z: `http://people.ciirc.cvut.cz/~hlavac/TeachPresCz/31Rozp/62MarkovianPR-Cz.pdf`

[11] BYSTRÝ, Vojtěch. *Markovovy modely v Bioinformatice* [online]. Masarykova Univerzita, Brno. [cit. 2023-01-24]. Dostupné z: `https://is.muni.cz/el/fi/jaro2017/PB051/um/prednaskaHMM1.pdf`

[12] MATOUŠEK, Kamil. *Pravděpodobnostní reprezentace neurčitosti: Bayesovské sítě* [online]. Fakulta Elektrotechnická, České vysoké učení technické Praha. [cit. 2023-01-24]. Dostupné z: https://cw.fel.cvut.cz/old/_media/courses/a7b33sui/bayesovske_site.pdf

[13] BERKA, Petr. *Bayesovská klasifikace* [online]. In: . [cit. 2023-01-24]. Dostupné z: https://sorry.vse.cz/~berka/docs/izi456/kap_5.6.pdf

[14] BERKA, Petr. *Dobývání znalostí z databází: Rozhodovací stromy* [online]. [cit. 2023-01-25]. Dostupné z: https://sorry.vse.cz/~berka/docs/izi456/kap_5.1.pdf

[15] VOLNÁ, Eva. *Neuronové sítě 1* [online]. In: . Ostravská univerzita, Ostrava, 2008 [cit. 2023-01-30]. Dostupné z: https://web.osu.cz/~Volna/Neuronove_site_skripta.pdf

[16] VONDRÁK, Ivo. *Neuronové sítě* [online]. Fakulta elektrotechniky a informatiky, VŠB - Technická univerzita Ostrava, 2009 [cit. 2023-01-30]. Dostupné z: http://vondrak.cs.vsb.cz/download/Neuronove_site.pdf

[17] VALENTA, Ondřej a Václav MATOUŠEK. *Umělé neuronové sítě* [online]. In: . Katedra informatiky a výpočetní techniky, Západočeská univerzita, Plzeň, 2018, [cit. 2023-01-30]. Dostupné z: https://www.kiv.zcu.cz/studies/predmety/uir/predn/P5_NN/FThema5_2018.pdf

[18] SIEGER, Tomáš. Neuronové sítě. In: *ČVUT FEL CourseWare Wiki* [online]. [cit. 2023-01-30]. Dostupné z: https://cw.fel.cvut.cz/b181/_media/courses/a6m33dvz/dvz2017-05-nnet.pdf

[19] AHSAN NAEEM, Muhammad. Pyamaze. In: *Github* [online]. [cit. 2023-04-06]. Dostupné z: https://github.com/MAN1986/pyamaze

[20] BRATKO, Ivan. *Prolog: programming for artifical intelligence*. 3rd ed. Harlow: Pearson, 2001. International computer science series. ISBN 02-014-0375-7.

Příloha A

Zdrojový kód pohybu agenta

```
def next_move(self, move, maze_map, turn):
    if self.can_walk is False:
        return
    next_position = self.next_position(move)
    # pokud je možné se daným směrem pohybovat a nová pozice ještě nebyla navští
        vena
    if maze_map[(self.x, self.y)][move] == 1 and next_position not in self.
        visited_positions:
        self.move(move)
    # není možné daným směrem jít nebo na daném místě už byl
    else:
        # vyhledá dostupné směry pohybu
        possible_moves=[k for k, v in maze_map[(self.x, self.y)].items() if v==1]
        if len(possible_moves) == 1:
            self.can_walk = False
            self.fitness += 300
        else:
            # zjistí se možné cesty, které nebyly navštíveny
            correct_moves = self.correct_moves(possible_moves)
            # vyloučí se cesta, která už byla použita
            difference = set(correct_moves) - set(move)
            difference = [i for i in difference]
            # pokud existuje více možných cest
            if len(difference) > 1:
                # náhodně se vybere jedna cesta a ta se použije k pohybu
                tmp = random.choice(difference)
```

```
            # uloží se na pozici aktuálního tahu, nový pohyb
            self.move_array[turn] = tmp
            self.move(tmp)
        # pokud neexistuje žádná možná cesta, tak se nemůže dále pohybovat
        elif len(difference) == 0:
            self.can_walk = False
        else:
            # existuje pouze jedna možná cesta k pohybu
            self.move_array[turn] = difference[0]
            self.move(difference[0])
```

Zdrojový kód A.1: Ověření a pohyb agenta.