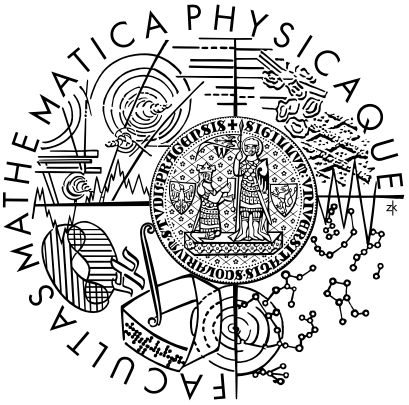
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

Diplomová práce

Pracovní verze



Jan Kratochvíl

Systém na vývoj robotů s vizuálním vnímáním

Kabinet software a výuky infromatiky

Vedoucí diplomové práce: RNDr. František Mráz, Csc.

Studijní program: Informatika

Poděkování

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne x. xx 2006

Jan Kratochvíl

Obsah

Title: Platform for robots development with visual perception

Author: Jan Kratochvíl

Department: Department of Software and Computer Science Education

Supervisor: RNDr. František Mráz, CSc.

Supervisor's e-mail address: mraz@ksvi.mff.cuni.cz

Abstract: TODO

Keywords: software, robots, genetic algorithms, neural networks

Název práce: Systém na vývoj robotů s vizuálním vnímáním

Autor: Jan Kratochvíl

Katedra: Kabinet software a výuky informatiky

Vedoucí diplomové práce: RNDr. František Mráz, CSc.

e-mail vedoucího: mraz@ksvi.mff.cuni.cz

Abstrakt: TODO

Klíčová slova: software, robot, genetické algoritmy, neuronové sítě

Kapitola 1

Úvod

Cílem této práce je navrhnout a implementovat systém umožňující vývoj řízení pro roboty s vizuálním vnímáním. Robotika je mladou vědou zahrnující jak výrobu a návrh, tak i metody řízení robotů. Přestože vzhledem i schopnostmi se roboti velmi liší, zachovávají několik společných vlastností. Jsou to mechanické pohyblivé struktury podléhající jisté formě řízení. Návrhem struktury se v této práci nebudeme zabývat a přejdeme rovnou k řízení jednoduchých autonomních robotů. Autonomní robot je inteligentní stroj, který je schopen vykonávat úlohy samostatně bez lidské pomoci. Příkladem může být například autonomní helikoptéra [**?**]. Nejdůležitější vlastností autonomního robota je jeho schopnost reagovat na změny prostředí. Za tuto vlastnost vděčí svému řídícímu systému. A právě návrhem struktury řízení a jejím učením se v této práci budeme zabývat.

Některé předprogramované systémy, jako například kráčející Johnie [**?**], zvládají obtížné úkoly. Jejich velkou nevýhodou je, že nejsou schopny se učit. Jak je tedy donutit, aby se učily v závislosti na zkušenosti? Bohužel zpětnovazebné učení (reinforcement learning) je omezeno na jednoduché reaktivní chování a s reálnými roboty nepodává tak dobré výsledky. Roboti potřebují dostávat informace o stavu okolního prostředí, stav mohou zjišťovat z moha různých senzorů, jako je kamera, sonar, polohovací zařízení nebo GPS. Senzory mohou být nepřesné, a proto je důležité správně rozlišit šum od podstatné informace. Dále je vhodné si vjemy alespoň krátkodobě zapamatovat v dynamických adaptivních strukturách.

Úspěšným přístupem k simulaci robotů s vizuálním vnímáním je použití neuronových sítí a genetických algoritmů. Z myslitelných neuronových sítí požadovaným vlastnostem nejlépe odpovídají rekurentní neuronové sítě.

Implementovat cíl této práce pomocí fyzických robotů je finančne, technicky i časově velmi náročné, proto je těžištěm práce implementace virtuálního prostředí. Přitom je potřeba, aby byl systém schopný simulovat pro robota jeho senzory (například kameru nebo sonar) a zároveň výstupy robota (motorické akce). Roboti, kterých může být v simulaci více, se mohou pohybovat v plně dynamickém trojrozměrném prostoru, tedy stejně jako ve skutečnosti. Důraz byl kladen na snadnou konfigurovatelnost ze strany experimentátora, tedy poskytnutí možnosti nastavit co nejvíce parametrů z uživatelského prostředí.

Celá práce je rozdělena do pěti kapitol. První kapitola poskytuje stručný úvod do problematiky učení robotů. Následuje kapitola, kde jsou nejdřív podrobně sepsány požadavky na implementovaný systém a vzápětí jsou probrány již existující systémy, které řeší podobné problémy. V závěru druhé kapitoly jsou podrobně specifikovány cíle této práce. Třetí kapitola pojednává o neuronových sítích a genetických algoritmech a podává tak teoretické základy, které jsou potřebné pro pochopení této práce jako celku. V závěru je stručně zmíněno paralelní zpracování, které je výhodné použít z důvodů velké výpočetní náročnosti. Kapitoly čtyři a pět pojednávají o samotné implementaci frameworku a o tom jak ho lze použít na několika příkladech. Kapitola pět na praktickém příkladu ukazuje použití vybudovaného systému. V poslední kapitole jsou shrnuty výsledky a možnosti jak na tuto práci navázat.

Kapitola 2

Přehled a motivace

Učení robotů s vizuálním vnímáním je velmi složité téma zasahující do mnoha oblastí informatických věd. Proto je v sekci 2.1 velmi stručně vysvětleno, kterým partiím z učení robotů se budeme věnovat. V následující sekci 2.2 vysvětlíme, co budeme požadovat od systému, který je vyvinut jako součást této práce. Po definici požadavků jsou v 2.3 probrány existující systémy řešící podobné problémy. Na úplný závěr kapitoly jsou podrobněji vytyčeny cíle této práce a určeny prostředky jak jich dosáhnout.

2.1 Učení robotů

Učit robota inteligentně reagovat na vnější prostředí a na základě vizuálních vjemů provádět určité akce je velmi obtížný úkol, ovšem metod vyvinutých pro jejich učení je mnoho. Existuje mnoho specializovaných cílů učení. Cílem této práce je učit roboty jednat na základě vizuálních vjemů, například dojet k cíli a po cestě překonat různé překážky, to vše jen při využívání vizuálních vjemů z prostředí.

V počátcích robotiky se výzkum soustředil na úlohy o plánování cesty ve známém prostředí, tedy na použití grafových algoritmů na předem známé mapě. V osmdesátých letech přišel V. Lumelsky [15, 16] s jiným možným přístupem, hledání v předem neznámém prostředí pro automat s malou pamětí a s dotykovým senzorem a znalostí svých povrchových souřadnic. Tento přístup se běžně označuje jako Bug algoritmy [9]. Nicméně tento přístup je stále příliš statický.

Zobrazení vstupů z vizuálních zařízení na vstupy pro kontrolery je velmi složitá funkce a proto se jí budeme snažit aproximovat. Velmi dobrou schopnost aproximace složitých funkcí vykazují neuronové sítě a proto se v roce 1982, kdy došlo

k oživení zájmu o problematiku neuronových sítí, začaly využívat i pro řízení robotů. Problémem ovšem zůstává, jak danou aproximační funkci nalézt, v našem případě tedy jak vytrénovat příslušnou neuronovou síť. Učící modely založené na učících případech je mnoho. Mezi ně patří i Back propagation a Hebbova učení, jejich nevýhody jsou ovšem především v pomalé konvergenci a v uvíznutí v lokálním minimu. Další možnou metodu učení poskytují genetické algoritmy, které jsou schopny poskytnout robustní řešení na učení různých druhů neuronových sítí. Podrobnější popis učení robotů je popsán v kapitole 3.

2.2 Požadavky na systém

Návrh systému na učení robotů je složitý a proto je vhodné rozebrat si ho na dílčí podúlohy. Nejprve si shrneme, co od cílového systému požadujeme, jednotlivé požadavky jsou popsány v následujících sekcích.

- zobrazení 3D světa (viz 2.2.1),
- co nejvěrnější simulace 3D světa (viz 2.2.2),
- vazba na reálné roboty, pomocí jednotného rozhraní (viz 2.2.3),
- rychlé zpracování (viz 2.2.4),
- podpora pro řídící systémy (viz 2.2.5),
- podpora učících algoritmů (viz 2.2.6),
- schopnost běžet na vzdáleném stroji (viz 2.2.7).

2.2.1 Zobrazení 3D světa

Od systému požadujeme schopnost jednoduše definovat prostředí, ve kterém se bude robot pohybovat, a to potom věrně zobrazit pomocí kamer, které má robot k dispozici. Vzhledem k tomu, že hardwarově akcelerované zobrazení je v poslední době, především díky hernímu průmyslu, již standardem, budeme od systému vyžadovat, aby jej využíval.

Dalším požadavkem na systém pro zobrazení, který může umožnit robustnější návrh systému, je schopnost po samotném zobrazení obraz upravovat, například ke generování šumů na kameře, které nám umožní simulovat reálné podmínky. Skutečný robot má totiž jen výjimečně čistý obraz. Navíc je možné požadovat černobílý obraz, který se opět dá simulovat pomocí úpravy obrazu.

Samotné zobrazení nemusí být nutně fotorealistické.

2.2.2 Simulace 3D světa

Další podstatnou částí jsou fyzikálně věrné interakce mezi objekty. Je žádoucí, aby se robot při nárazu správně odrazil, překážka, do které robot narazí, správně spadla atd. Při simulaci pevných těles je kladen důraz na stabilitu a na rychlost zpracování. Dále je vyžadována podpora základních typů spojů mezi pevnými tělesy. Simulovat budeme především roboty s pásy případně kolečky a proto je třeba aby systém uměl simulovat chování vozidel, respektive robotů, a to jak pásových tak kolečkových.

2.2.3 Vazba na reálné roboty

Vzhledem k záměrům práce je vhodné, aby se naučený robot dal vyzkoušet i v reálném světě mimo simulaci. K dosažení tohoto cíle je potřeba, aby systém pro řízení robota komunikoval pomocí jednotného rozhraní, které samo rozhodne jestli pošle příkaz robotu v simulovaném světě či v reálném. Přirozeným důsledkem tohoto požadavku je to, že simulovaný robot nesmí poskytovat informace, které by reálný robot nemohl mít. Například simulovaný robot nesmí poskytnout informace o své absolutní pozici, pokud nemá GPS modul.

2.2.4 Rychlé zpracování

Vzhledem k tomu, že od systému budeme požadovat mnohonásobnou simulaci tak je velmi žádoucí, aby byl systém co nejvíce optimalizován na rychlost.

2.2.5 Řídící systémy

Od řídícího systému budeme požadovat, aby se na základě vstupních informací z vnějšího prostředí (kamera, sonar, ...) rozhodl o dalším konání robota. Dále je třeba, aby byl tento systém snadno rozšiřitelný, například formou přídavných modulů, a také snadno konfigurovatelný, aby usnadňoval vykonávání experimentů. V neposlední řadě musí být systém schopný spolupracovat s učícími algoritmy popsanými níže. V této práci jsou pro řídící systémy použity neuronové sítě. O motivacích vedoucích k tomuto rozhodnutí více v kapitole 3.

2.2.6 Učící algoritmy

Učící algoritmy musí úzce spolupracovat s řídícím systémem a naučit ho chování, které jistou formou požaduje uživatel. Opět platí, že tyto algoritmy mají být rozšiřitelné a snadno konfigurovatelné. V této práci jsou pro učení použity genetické algoritmy, o kterých je více v kapitole 3.

2.2.7 Schopnost běžet na vzdáleném stroji

Tento požadavek vychází z předpokladu časové náročnosti zpracování velkého počtu potřebných simulací, zejména v kombinaci s použitím genetických algoritmů. Je tedy vhodné úlohu urychlit využitím více počítačů najednou. Tato vlastnost klade skrytý, ale podstatný technický požadavek na zobrazování, které musí být schopno zobrazovat i bez existence okna do kterého by mělo zobrazovat.

2.3 Systémy řešící podobné problémy

V této části se pokusíme shrnout a stručně popsat systémy, které by šlo ve velmi omezené míře použít k dosažení podobných výsledků jaké nabízí tato práce.

2.3.1 Gazebo a Player

Projekt Player/Stage [4] umožňuje výzkum robotických a senzorových systémů. Server „Player" je pravděpodobně nejpoužívanější rozhraní pro řízení robotů. Jeho simulační prostředí Stage a Gazebo jsou rovněž velmi rozšířené. Stage i Gazebo jsou robotické simulátory pro venkovní prostředí. Umožňují simulaci celé populace robotů. Základní rozdíl mezi těmito simulátory je v tom, že Stage pracuje v dvourozměrném prostředí zatímco Gazebo v třírozměrném.

Gazebo tedy vytváří 3D virtuální svět včetně fyzikálních interakcí. Samo o sobě vůbec neřeší učení robotů. Pro řešení problému učení robotů ho lze použít pouze jako nejnižší vrstvu, která odstiňuje problémy s použitím reálného robota.

Player je knihovna umožňující přístup k ovládání robotů. Díky ní je možno odstínit simulátor Gazebo. Tedy při používání Playera nám nezáleží na tom, zda přistupujeme k robotu simulovaném v Gazebu či k reálnému robotu. Tedy tento systém se vůbec nestará o logiku řízení robotů ani o jejich učení. Vztah mezi Playerem a Gazebem je tedy jasně daný, Gazebo simuluje svět a Player ovládá roboty v daném světě.

Výhody

- stabilní fyzikální simulace světa - autorům se povedlo odladit parametry fyzikální simulace a pomocí knihovny ODE [22] připravili prostředí s reálnými fyzikálními interakcemi.
- rozšiřitelnost jednotlivých částí - Gazebo i Player lze snadno rozšířit o další zařízení. Například lze v pár krocích snadno přidat novou kameru.
- open source - volně dostupné zdrojové kódy usnadňují rychlé proniknutí do programu a hlavně umožňují úpravy dané aplikace.

Nevýhody

- velmi nízkoúrovňové - systém umí pouze simulovat roboty, vůbec neřeší jak roboty řídit.
- samo o sobě nepodporuje paralelizaci - systém nemá žádné prostředky pro paralelizaci, navíc ani nepodporuje zpracování simulace s vizuální informací bez hardwarového displaye. (Tedy nelze pouštět úlohu na vzdáleném počítači)
- nepřipravenost na požadavky k zasahování do běhu simulace zvenku - Gazebo se snaží působit jako vrstva, která má být plně nahraditelná reálným světem, tudíž neposkytuje informace, které se pro učení robotů mohou hodit, ale v reálném světě nejdou získat. Například získání absolutní pozice objektů.

2.3.2 Marie

MARIE [5] je návrhářský nástroj pro mobilní a autonomní robotické aplikace. Umožňuje integraci mnoha heterogenních softwarových elementů. Tento nástroj je založen na distribuovaném modelu, tedy umožňuje pouštění jednotlivých komponent na různých strojích, architekturách či platformách. Tedy systém se snaží navrhnout obecné rozhraní pro běžné úlohy vyskytující se v aplikacích zabývajících se robotickým výzkumem a implementovat metody tohoto rozhraní pro nejpoužívanější systémy. A tím tyto systémy propojit mezi sebou.

Systém je velmi komplexní a i proto trvá velmi dlouho než do něj uživatel pronikne a i poté je poměrně obtížné s daným systémem pracovat. Nicméně samotní autoři MARIE již integrovali systémy Gazebo, Player, Stage, CARMEN, RobotFlow, ARNL.

Výhody

- mnoho integrovaných systémů - systém sám o sobě podporuje mnoho exis-

tujících systémů pro vývoj robotů. V podstatě všechny běžně používané.

- podpora paralelního zpracování - je na úrovni obecných rozhraní. Tedy je dokonce možné pustit rozpoznávání obrazu na jiném počítači než samotné řízení robota.

- široký záběr - tedy snaha o integraci všech možných úkolů robotických simulací do jednoho velkého balíčku. Tedy je velmi pravděpodobné, že výzkumník nebude potřebovat použít jiný systém k výzkumu než je tento.

Nevýhody

- příliš obecné - je velmi obtížné systém používat a ještě obtížnější donutit systém aby dělal jednu konkrétní úlohu, kterou experimentátor zrovna potřebuje. Vyžaduje dlouhou dobu na proniknutí do systému.

- rané stadium implementace - systém je zatím nasazen jen v několika málo projektech a stále není v produkční verzi.

2.3.3 Pyro

Pyro [3] je zkratka pro Python robotics. Cílem projektu je poskytnout prostředí pro jednoduché zkoumání pokročilých témat umělé inteligence, aniž by se uživatel musel zabývat nízkoúrovňovými problémy s hardwarem. Jak název napovídá Pyro využívá python, a tím samozřejmě vymoženosti interpretovaných jazyků. Tedy lze měnit kusy kódu bez nutnosti opětovné kompilace. Pyro si rozumí s několika robotickými simulátory, v současné době jsou to Robocup soccer, Player / Stage, Gazebo a Kheppera.

Výhody

- podpora genetických algoritmů a neuronových sítí - systém podává základní rozhraní pro práci s neuronovými sítěmi a poskytuje metody učení pomocí genetických algoritmů.

- zaměření na fyzické roboty - je velmi intenzivně testován s mnoha různými hardwarovými roboty a tudíž systém propojení mezi počítačem a roboty je velmi dobře odladěn.

- skriptování pomocí pythonu - dynamicky interpretovaný jazyk pro určení chování robotů umožňuje snadnou změnu chování robota v průběhu vykonávání jeho akcí. Navíc není nutné pouštět kompilaci systému při každé drobné změně.

Nevýhody

- neschopnost paralelního zpracování - systém nepodporuje paralelní zpracování robotů s vizuální informací. Neumí pouštět simulaci na vzdáleném počítači bez hardwarového displaye.

- pomalá interpretace jazyka python - nevýhodou skriptovacího jazyka je rychlost. Pomalé zpracování může být na obtíž v případě nutnosti realtime zpracování a potřeby rychlých reakcí.

2.3.4 Lobot

Projekt Lobot vznikl jako softwarový projekt na Matematicko-Fyzikální fakultě Univerzity Karlovy [24]. Projekt má podobné cíle jako tato diplomová práce. Nicméně díky snaze vše si naimplementovat pomocí vlastních zdrojů není projekt rozšířen a neexistují žádné ovladače pro roboty, se kterými autoři projektu nepočítali. Navíc integruje pouze 2D fyzikální simulaci.

Výhody

- řešení všech požadavků v jednom balení - systém se snaží řešit učení robotů. Vše svými vlastními prostředky tak, aby experimentátor nepotřeboval žádnou další knihovnu.

- podpora paralelního zpracování - systém umí běžet na vzdáleném stroji a dokonce nemá problém ani se simulací vizuálních vjemů bez nutnosti vlastnit hardwarový display.

Nevýhody

- 2D fyzikální simulace - fyzika i detekce kolizí jsou pouze ve 2D, nelze tedy například simulovat přelézání překážek, let helikoptéry atd.

- neudržovaný projekt - projekt skončil současně s obhajobou, od té doby je vývoj pozastaven.

- vlastní rozhraní pro komunikaci s roboty - toto je velká nevýhoda, systém podporuje LEGO roboty a žádné jiné.

Na závěr tabulka shrnuje popsané systémy a jejich vlastnosti.

[1]Pouze pomocí externích knihoven
[1]Pomocí Gazeba

	Gazebo	Marie	Pyro	Lobot
zobrazení 3D světa	Ano	Ano[1]	Ano[2]	Ano
fyzikální simulace	Ode	Ano [1]	Ode	Vlastní 2D
vazba na reálné roboty	Player	Ano [1]	Ano	Lego robot
podpora učících algoritmů	Ne	Ano [1]	Ano	Ano
paralelizace	Ne	Ano [1]	Ne	Ano
podpora řídících systémů	Ne	Ano [1]	Ano	Ano

Tabulka 2.1: Porovnání existujících systémů

2.4 Cíle

Cílem této práce je navrhnout a naimplementovat systém pro vývoj robotů s vizuálním vnímáním, který bude umět simulovat reálné prostředí, a to včetně fyzikálních interakcí. Simulace musí zahrnovat i simulaci vozidel, tedy fyzikálních interpretací robotů na pásech, případně na kolech. K simulaci bude připraveno jednotné rozhraní pro definici a řízení robotů. Nad tímto rozhraním bude postaven systém pro učení robotů genetickými algoritmy. Samotné řízení robotů bude pomocí neuronových sítí, pro které bude v systému poskytnut nízkoúrovňová podpora. Použití systému bude předvedeno na jednoduchých příkladech. Dalším cílem je studie jednotlivých postupů při genetickém učení a zkoumání vlivů prostředí na učící se roboty.

2.4.1 Realizace cílů

Vzhledem k tomu, že celkově si práce dává za cíl velmi komplexní úkol, není v silách jedince všechny dílčí podúlohy řešit od začátku. Jedním z velmi obtížných úkolů je dosažení věrné simulace reálného světa, který se chová podle fyzikálních zákonů. Řešení tohoto problému se tedy vyhneme použitím vhodné knihovny. Nejobtížnější krok ovšem nastává v případě rozhodování o rozhraní pro řízení robotů. Chceme totiž zaručit aby byl v naší práci využitelný i robot, který bude navržen až po dokončení této práce nebo takový, který s využitím v této práci vůbec nepočítá. Z tohoto důvodu je velmi žádoucí použít pro řízení robotů některé již existující a pokud možno rozšířené rozhraní.

2.4.1.1 Zvolené knihovny

Pro simulaci reálné fyziky byl z existujících systémů (havok,meqon,ageia,ode,...) vybrán systém **ODE** [22], jako jediný je totiž volně dostupný. K odstínění problémů s integrací různých robotů je použit projekt **PlayerStage [4]**, který má zdaleka nej-

větší komunitu uživatelů. Částí tohoto projektu je i Gazebo, které je rovněž použito, avšak značně upravené pro potřeby této práce. Pro řízení robotů byly vybrány neuronové sítě, které jsou implementovány pomocí knihovny **Annie** [2], která byla pro účely této práce rozšířena. Podpora paralelního zpracování je implementována pomocí **PVM [20].** Zajištění samotné konfigurovatelnosti systému je pomocí souborů ve formátu XML.

2.4.1.2 Prostředí

Implementace využívá projekt PlayerStage z čehož plyne omezení na nutnost použití POSIX platformy. Celá práce je testována pouze pod GNU/Linux, ale z principu by měla být zkompilovatelná na jakékoliv POSIX platformě.

2.4.2 Přínos práce

Hlavním přínosem této práce je vytvoření systému, který bude schopen pomocí genetických algoritmů paralelně zpracovávat velké populace jedinců při zachování velkého stupně rozšiřitelnosti a konfigurovatelnosti systému, včetně vyhodnocování vizuálních vjemů.

Kapitola 3

Teoretické základy

Tato kapitola popisuje teoretické základy, které jsou základem pro implementační rozhodnutí provedená v následujících kapitolách. Kapitola je rozdělena na 3 části, první popisuje řídící jednotku, tedy systém který řídí robota. Druhá část popisuje učící mechanismus, tedy systém jakým způsobem lze řídící jednotku naučit chovat se podle našich představ. Poslední část této kapitoly se věnuje možnostem distribuovaného zpracování.

3.1 Řídící jednotka

Existuje mnoho způsobů jak řídit robota. Naivní přístup je pokusit se nastavit přesná pravidla, která určují jak se v dané situaci chovat. Tedy pokusit se vystihnout každou možnou situaci a určit na ní správnou reakci. Tímto přístupem jsme schopni řešit jednoduché úlohy u nichž víme, že nikdy nedojde k nečekané události. Nevýhodou je neschopnost přizpůsobit se razantním změnám v prostředí a nulová schopnost učení se z chyb.

Další možností je sestavit řídící program ve tvaru stromové struktury. Program je konstruován ze dvou množin symbolů: z množiny T terminálních symbolů, které reprezentují vstupy, výstupy, konstanty, případně nulární funkce, a z množiny F neterminálních symbolů, které reprezentují funkce. Je dobré si uvědomit, že funkcí muže být i podmíněný operátor **If-Then-Else**. Takovéto stromové struktury pak lze modifikovat a učit plnit naší úlohu pomocí genetického programování. Více o této problematice v kapitole 5 v [27].

Na řízení robota lze nahlížet jako na složitou transformaci senzorických vstupů robota na jeho motorické jednotky. Takováto transformace se dá realizovat také

pomocí neuronových sítí. Tento postup byl vybrán v této práci a proto bude popsán podrobně ve zbytku této sekce.

3.1.1 Neuronové sítě

Neuronová síť (NN - neural network) je síť mnoha navzájem bohatě propojených jednoduchých jednotek. Graf takového propojení se často označuje jako topologie sítě. Jednotky nazýváme neurony, protože velice zjednodušeně modelují skutečné neurony v centrální nervové soustavě a jejich formální návrh je inspirován vlastnostmi mozku. Více viz například [25, 27].

3.1.2 Neuron

Do neuronu se sbíhá n spojů, které reprezentují buď výstupy z jiných neuronů nebo podněty z vnějšího okolí. Po každém z těchto vstupů, řekněme i-tém, přichází v daném časovém okamžiku hodnota (reálné číslo) x_i. Ke každému spoji je navíc přiřazeno reálné číslo w_i, které reprezentuje váhu daného spoje. A nakonec ke každému neuronu je přiřazeno reálné číslo ϑ, které nazýváme práh. Vážený součet $\xi = \sum_{i=1}^{n} w_i x_i - \vartheta$´udává celkový podnět, tzv. potenciál neuronu. Na tento potenciál neuron reaguje odezvou $z = f(\xi)$, kde f je tzv. přenosová funkce.

Přenosová funkce obvykle bývá neklesající. V neuronových sítích se nejčastěji používá několik základních druhů přenosových funkcí.

- **Sigmoida** - nejobvyklejší prahovací funkce definována jako $f(\xi) = \frac{1}{1+e^{-\lambda\xi}}$, kde λ určuje „strmost" této funkce.
- **Prahovací funkce -** $f(\xi) = 1$ pro $\xi \geq 0$ a $f(\xi) = -1$ pro $\xi < 0$.
- **Lineární funkce -** $f(\xi) = a\xi$

Neurony dále můžeme dělit podle umístění v síti na vstupní, skryté a výstupní. Samozřejmě neuron může být vstupní a zároveň výstupní.

3.1.3 Struktura neuronové sítě

Ze strukturálního hlediska existují dva druhy neuronových sítí, dopředné a rekurentní. Dopředné sítě jsou takové, kde informace putuje pouze jedním směrem, od vstupních neuronů k výstupním. V rekurentních sítích jsou dva různé typy spojů, dopředné a zpětné, které umožňují propagovat informaci tam a zpět. Dopředné sítě byly úspěšně použité v úlohách na řízení robotických kontrolerů [28], nicméně

tyto sítě potřebují velký počet vstupních neuronů, což znamená delší výpočetní čas a jsou málo odolné vůči externím šumům [28].

Rekurentní sítě jsou atraktivní převážně tím, že jsou schopny řešit úlohy na které dopředné sítě nestačí, například protože mají schopnost dočasné paměti. Dva jednoduché typy rekurentních sítí jsou Elmanova síť [10] a Jordanova síť [12], které jsou popsány dále v textu.

3.1.4 Učení rekurentních neuronových sítí

Jednoduchý postup jak učit tyto sítě je pomocí algoritmu *back propagation* [21] s tím, že všechny zpětné vazby jsou vyloučeny z učení a mají přednastavené konstantní váhy s kterými experimentuje sám uživatel. Tento postup ovšem není příliš praktický vzhledem k tomu, že zpětné váhy vůbec neučíme. Pro učení rekurentních neuronových sítí se dají použít i genetické algoritmy a toho využijeme v této práci.

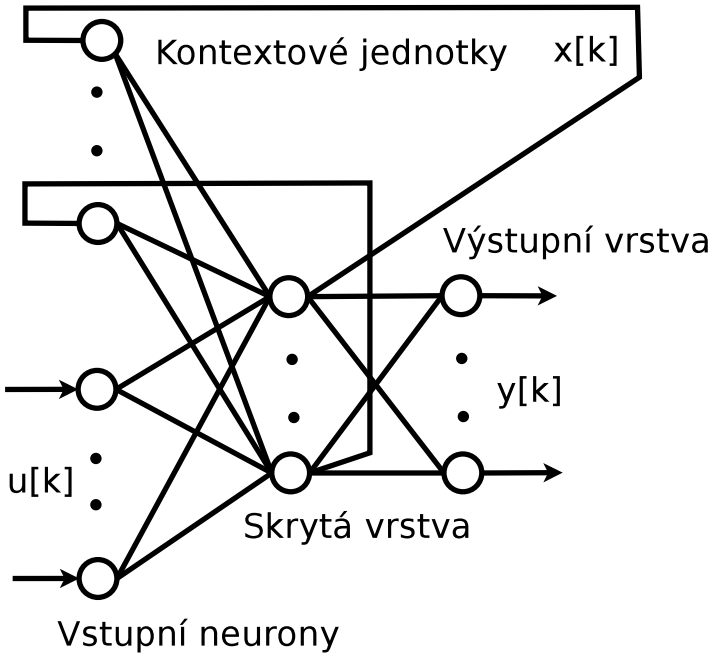
3.1.5 Použité neuronové sítě

3.1.5.1 Elmanova síť

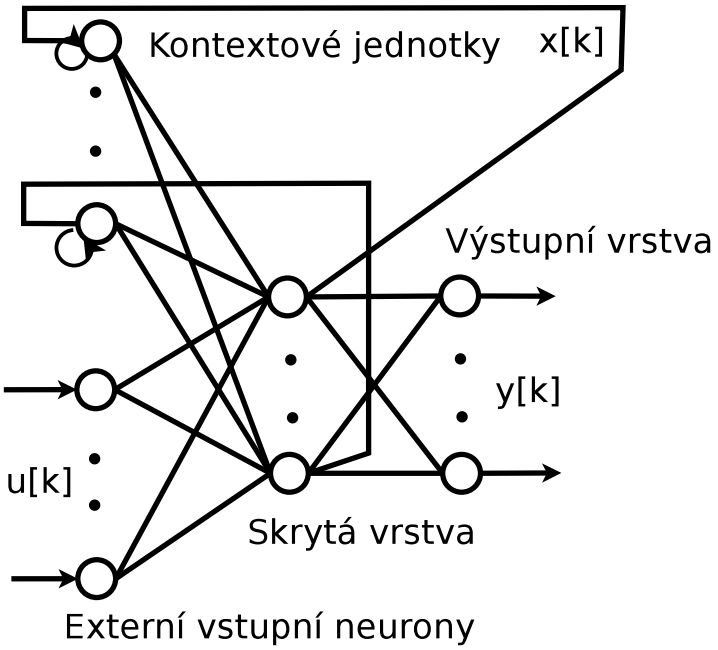
Standardní elmanova síť (viz obrázek 3.1) je složena ze tří vrstev neuronů. První vrstva je rozdělena do dvou skupin, v první jsou externí vstupní neurony a ve druhé interní vstupní neurony, jinak také nazývané kontextové jednotky. Vstupy do kontextových jednotek obstarávají neurony z druhé vrstvy, nazývané skrytá vrstva. Výstupy z kontextových jednotek a z externě vstupních neuronů jsou napojeny na skrytou vrstvu. Kontextové jednotky jsou známy také jako paměťová jednotka, protože uchovávají hodnoty minulého výstupu skryté vrstvy.

3.1.5.2 Upravená Elmanova síť

Vychází ze standardní Elmanovy sítě (viz 3.1.5.1), liší se pouze v zavedení *self-feedback* (tedy vazba sama na sebe) vazby na kontextových jednotkách, čímž se zvýší dynamická paměť celé sítě. Celé zavedení nové vazby je inspirováno Jordanovou sítí (3.1.5.3). Váhy nových spojů zafixujeme na hodnoty mezi 0 a 1 před začátkem učení. Zapojení je vidět na obrázku 3.2.



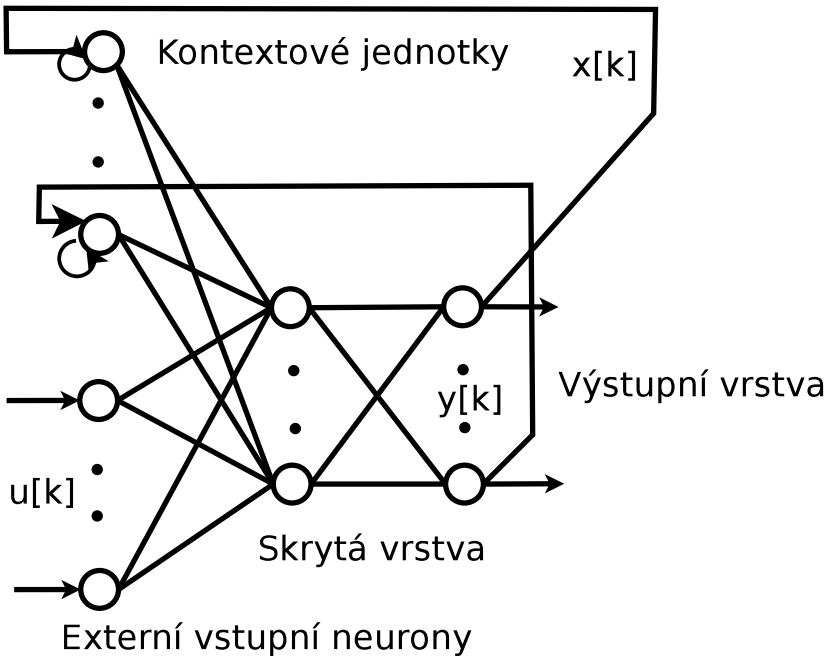
Obrázek 3.1: Elmanova síť - základní varianta



Obrázek 3.2: Elmanova síť - rozšíření o *self-feedback* vazby

3.1.5.3 Jordanova síť

Tato síť má stejně jako Elmanova síť tři vrstvy. Velmi se podobá upravené Elmanově síti s tím rozdílem, že zpětná vazba není ze skryté vrstvy, ale z výstupní vrstvy. Viz obrázek 3.3.



Obrázek 3.3: Jordanova síť - základní varianta

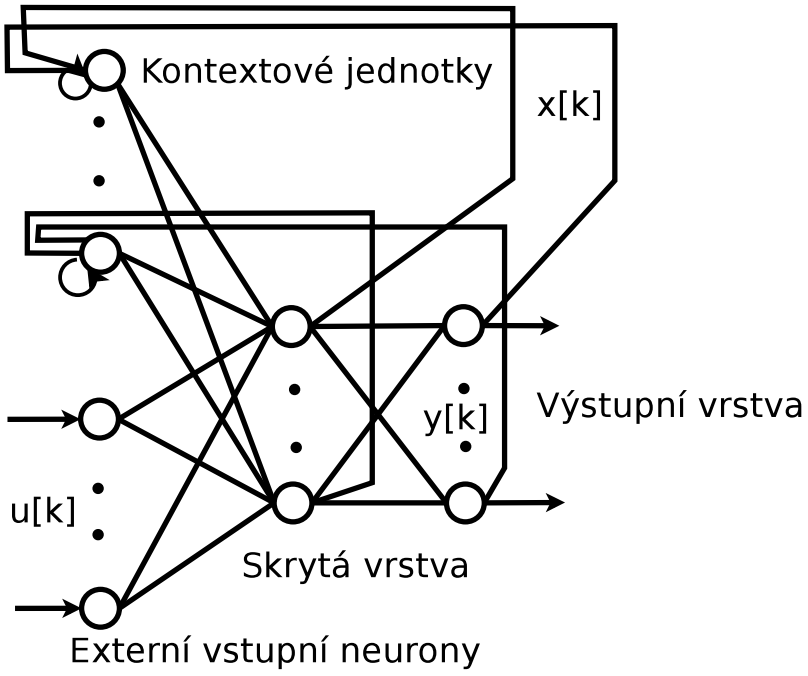
3.1.5.4 Upravená Jordanova síť

Tato síť vychází z Jordanovy sítě. Vzniká rozšířením o zpětné vazby i ze skryté vrstvy. Viz obrázek 3.4.

3.2 Učící mechanismus

Nyní máme popsán řídící mechanismus a je třeba rozhodnout jak ho učit. Vzhledem k tomu že jsme si za mozek vybrali neuronovou síť tak se budeme dále věnovat jen metodám učení neuronových sítí.

Nejznámější metoda učení vícevrstvých neuronových sítí je jistě metoda zpětného šíření, back-propagation. Tato metoda dává uspokojivé výsledky v dopředných sí-



Obrázek 3.4: Jordanova síť - rozšíření o zpětné vazby ze skryté vrstvy

tích. V sítích se zpětnou vazbou ovšem přímo použít nejde. Pokud by měla být použita, tak se musejí speciálně vyřešit zpětné vazby. A to tak, že zpětné vazby nastaví uživatel a učit se budou pouze dopředné. To je ovšem velmi nepraktické. Existují i upravené verze metody zpětného šíření pro rekurentní sítě, popsané například v [14].

Proto v našem případě pro naučení těchto sítí použijeme genetické algoritmy (GA), viz sekce 3.2.1, a to pro získání vah dopředných i zpětných vazeb. GA slouží jako usměrněné náhodné vyhledávání, což je poměrně efektivní pro optimalizaci nelineárních funkcí [13]. Jednou z výhod GA, která je odlišuje od ostatních optimalizačních metod (např od gradientních), je jejich schopnost opustit lokální extrém.

3.2.1 Genetické algoritmy

Genetický algoritmus je heuristický postup, který se snaží aplikací principů evoluční biologie nalézt řešení složitých problémů, pro které neexistuje použitelný exaktní algoritmus. Genetické algoritmy, resp. všechny postupy patřící mezi tzv. evoluční algoritmy používají techniky napodobující evoluční procesy známé z biologie — dědičnost, mutace, přirozený výběr a křížení — pro „šlechtění” řešení zadané úlohy.

Princip práce genetického algoritmu je postupná tvorba generací různých řešení daného problému. Při řešení se uchovává tzv. populace, jejíž každý jedinec, reprezentovaný genem, představuje jedno řešení daného problému. Jak populace probíhá evolucí, řešení se zlepšují. Tradičně je řešení reprezentováno binárními čísly, nicméně používají se i jiné reprezentace (strom, pole, matice, ...). Typicky je na začátku simulace (v první generaci) populace složena z naprosto náhodných členů. V přechodu do nové generace je pro každého jedince spočtena tzv. fitness funkce, která vyjadřuje kvalitu řešení reprezentovaného tímto jedincem. Podle této kvality jsou stochasticky vybraní jedinci, kteří jsou modifikováni (pomocí mutací a křížení), čímž vznikne nová populace. Tento postup se iterativně opakuje, čímž se kvalita řešení v populaci postupně vylepšuje. Algoritmus se obvykle zastaví při dosažení postačující kvality řešení, případně po předem dané době.

Algoritmus 1 Genetický algoritmus

```
Vytvoř úvodní populaci jedinců
while neplatí podmínka ukončení do
  vyhodnoť fitness funkci všech jedinců
  vyber členy populace pro reprodukci
  vytvoř novou populaci pomocí křížení a mutace
end while
```

Algoritmus 1 je záměrně velmi obecný, pro každou z jeho částí je možno používat více metod. Další důležitou volbou je rozhodnutí jak reprezentovat jedince dané populace. Vše je v krátkosti popsáno v následujících sekcích.

3.2.2 Reprezentace jedince

Cílem našeho snažení je naučit neuronovou síť pomocí genetických algoritmů. Tedy již u reprezentace jedince se budeme zabývat jen možnostmi jak kódovat neuronovou síť. Dobrá volba reprezentace, spolu se správnou volbou rekombinačních operátorů může velmi urychlit konvergenci ke správnému řešení.

Při reprezentaci neuronové sítě se musíme rozhodnout zda bude náš gen určovat pouze váhy v pevně dané síti, či zda bude určovat i topologii sítě. Výhodou pevně dané topologie je větší stabilita, ovšem pokud je původní topologie špatně navržená tak se ze špatného stavu pomocí evoluce nemůžeme dostat. Na druhou stranu při variabilní topologii sítě je pro nalezení dobrého řešení nutné tvořit populace o mnoha jedincích.

3.2.2.1 Váha spoje

Váha mezi dvěma neurony je reálné číslo. Vzhledem k tomu že velikost reprezentace jednotlivých genů ovlivňuje celkovou velikost prohledávaného prostoru a čím větší prostor tím hůře se v něm najde nejlepší řešení, je rozumné reálné číslo reprezentovat v méně než 32 bitech. Reprezentace B bity dává 2^B možností. Navíc lze využít znalost o aktivační funkci. Je velmi praktické zvolit B jako 2^n, pro nějaké n přirozené. Vzhledem k tomu že je potom vhodněji reprezentovatelné. Pokud se podíváme na sigmoidální funkci, popsanou v sekci 3.1.2, tak si můžeme všimnout že tato funkce má maximální derivaci v okolí nuly. Zatímco pro hodnoty v absolutní hodnotě vzdálené od nuly jsou si již hodnoty této funkce velmi podobné. Proto se vyplatí v naší reprezentaci omezit jen na malý interval, který navíc nebude do B-bitového čísla kódován rovnoměrně, ale pomocí nějaké transformační funkce, například funkce inverzní k sigmoidální funkci.

3.2.2.2 Neuronová síť

Pokud máme reprezentovánu váhu na jednom spoji, stále ještě musíme reprezentovat všechny váhy v dané síti. Samotná reprezentace všech vah může být v poli, kde pozice v poli jednoznačně určuje, o kterou váhu jde. Cílová reprezentace je tedy v podstatě matice bitů o rozměrech LxB, kde L je počet spojů v dané sítí.

3.2.3 Fitness funkce

Ohodnocení daného jedince má za úkol určit jak moc se danému jedinci vedlo. Základní požadavek je, aby funkce ohodnocení byla monotónní. Tedy aby setřídění jedinců podle hodnot fitness funkce odpovídalo skutečnosti, tedy aby roboti kteří řeší danou úlohu lépe dostali větší ohodnocení. Tento jednoduchý předpoklad je bohužel velmi těžké splnit a hledání dobré fitness funkce, která správně ohodnotí vykonanou práci bývá velmi zdlouhavé.

3.2.4 Selekce

Nejpopulárnější metody pro selekci jedinců z populace jsou ruleta a turnaj, které jsou popsané níže.

3.2.4.1 Ruleta

Probíhá tak, že se pro ohodnocená řešení spočítá $\vartheta = \sum_{i=1}^{N} \omega_i$ kde ω_i je hodnota fitness funkce pro i-tého jedince. Poté N-krát vybíráme nového jedince kde i-tý jedinec bude zvolen s pravděpodobností $p(i) = \frac{\omega_i}{\vartheta}$. Selekce pomocí rulety dává navíc zřejmý požadavek na nezápornost fitness funkce.

3.2.4.2 Turnaj

Turnajová selekce probíhá tak, že vyberem několik náhodně zvolených jedinců z populace a z nich vezmem s určitou pravděpodobností nejlepšího jedince, tedy toho s největší hodnotou fitness funkce.

3.2.4.3 Výběr selekce

Již z letmého pohledu je vidět, že turnajová selekce má oproti ruletě například tyto výhody:

- Turnaj lze pořádat i když není vyhodnocena celá populace.
- Ruletová selekce je více ovlivněna absolutním hodnotou fitness funkce a to je vzhledem k tomu že v typických případech neznáme optimální hodnotu fitness funkce nepříjemné.

Ve většině dostupné literatury vychází turnajová selekce mnohem lépe než ruleta, nebo jiná myslitelná selekce, proto bude v následujícím textu zvažována již jen tato.

Pro zachování nejlepších jedinců se standardně ještě přenáší několik nejlepších jedinců do nové populace, tzv. elitismus. Jinak by se mohlo stát, že nejlepší jedinec se zkříží a nejlepší hledaný jedinec by tak byl ztracen.

3.2.5 Rekombinační operátory

Pro zanesení změn do populací se používají dva druhy operátorů, křížení a mutace. Ne všichni jedinci ovšem musejí projít těmito genetickými operacemi. Operace křížení se provede s určitou pravděpodobností P_x, zpravidla mezi 0,6-1, zatímco mutace se provádí s nižší pravděpodobností P_m, typicky mezi 0,05-0,15.

3.2.5.1 Křížení

Křížení typicky vezme dva geny a zkombinuje je do nového. Na bitových řetězcích se zpravidla používá tzv. jednobodové křížení, které pracuje, tak že náhodně vybere křížící bod x z intervalu $(0, L)$ kde L je velikost řetězce. Nový prvek vznikne, tak že z prvního genu se vezme prvních x bitů a z druhého zbývajících $L - x$. V našem případě ovšem nemáme bitové řetězce ale přímo řetězce B-bitových čísel. Budeme tedy volit křížící bod vždy tak aby $x = B*n$ pro nějaké n přirozené. Kdyby bylo povoleno křížit i uvnitř reprezentace konkrétní váhy jednoho neuronu, tak by docházelo k znehodnocení váhy jejíž hodnota by byla v křížícím bodě.

Existuje spousta dalších druhů křížení, ve zkratce například jednobodové křížení, dvoubodové křížení, „cut and splice", jejich popis lze nalézt například v knize [13]. První dvě metody zachovávají délku genu, třetí délku mění.

3.2.6 Mutace

Mutace má za úkol daného jedince náhodně poupravit, ovšem s tím že by měla pokud možno zachovat co nejvíce jeho původních vlastností. Základní podoba mutace je taková, že nad každým bitem v celé populaci provedeme náhodný experiment s pravděpodobností úspěchu P_m. Pokud experiment uspěje tak daný bit invertujeme. Alternativou k tomuto postupu je volba jedince s pravděpodobností $PS * P_m$, kde PS je velikost populace, a inverze jednoho náhodně zvoleného bitu tohoto jedince, více v [26].

Pro naše potřeby jsou však tyto metody nevhodné. Změna náhodného bitu v $B-$bitovém řetězci, který reprezentuje reálné číslo, může způsobit příliš velký rozdíl oproti původní hodnotě. Proto je lepší použít plíživou mutaci (creep mutation), která využije znalost transformace z bitového řetězce v reálné číslo. V algoritmu 2 volíme N, které udává počet změněných hodnot v celém genu, dále musíme zvolit x, které říká jaká je maximální odchylka od původní hodnoty. Funkce *uniformRand* dá náhodné číslo z intervalu zadaného parametrem, s tím že rozdělení pravděpodobnosti je rovnoměrné.

Algoritmus 2 Creep mutace

if $uniformRand(0,1) < P_m$ **then**
 for i:=1 **to** N **do**
  $temp$:=transformuj B-bitové číslo na reálné
  $temp := temp + uniformRand(-x, x)$
  převeď $temp$ zpět do binární reprezentace
 end for
end if

3.2.7 Ukončení

Jako poslední problém nám zbylo rozhodnutí kdy ukončit vývoj a další jedince již negenerovat. Z časových důvodů budeme ukončovat simulaci po dosažení, předem pevně stanoveného, počtu generací. Jako sekundární podmínka bude nalezení dostatečně dobrého jedince.

3.3 Distribuované zpracování

Vzhledem k tomu, že výpočet fitness funkce je časově velmi náročná operace je nutné uvažovat o nějaké formě distribuovaného zpracování. Přínosem distribuovaného zpracování by mělo být zrychlení až Nkrát pokud budeme pracovat na N počítačích místo na jednom. Ve skutečnosti to ovšem není tak jednoduché. Do hry totiž vstupuje režie na distribuci úloh. Naštěstí ovšem genetické algoritmy patří do skupiny snadno distribuovatelných úloh.

Jenda z nejjednodušších metod je popsána algoritmem 3, který je drobnou modifikací algoritmu 1.

Algoritmus 3 Paralelní Genetický algoritmus

```
Vytvoř úvodní populaci jedinců
while neplatí podmínka ukončení do
  do paralelně
    vyhodnoť fitness funkci všech jedinců
  end paralelně
  vyber nejlepší páry pro preprodukci
  vytvoř novou populaci pomocí křížení a mutace
end while
```

Jinou možnou metodou je provádět N úloh nezávisle na N procesorech. A nejlepší výsledek z těchto N běhů prohlásíme za výsledek. Vzhledem k tomu že genetické algoritmy jsou stochastické tak je stejně potřeba několik běhů k dosažení rozumného výsledku, takže i tato metoda má své uplatnění.

Další propracovanější techniky pro paralelní genetické algoritmy, vycházejí z pozorování že přirozené populace mají územní strukturu. A vývoj probíhá v částečně oddělených skupinkách jedinců. Dvě nejznámější kategorie jsou ostrovy [8, 23] a mřížky [17].

Vzhledem k tomu, že provádění výpočtu fitness funkce je v našem případě zdaleka nejnáročnější operace, tak nám postup popsaný algoritmem 3 přinese lineární zrychlení výpočtu. Tedy nevyplatí se hledat složitější řešení.

Kapitola 4

Framework

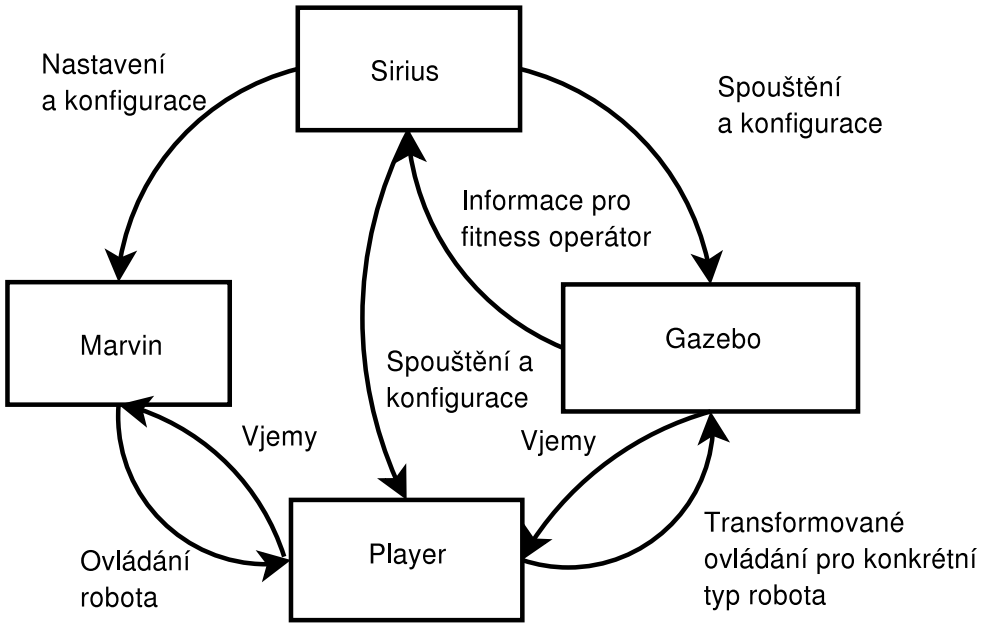
V následující kapitole si popíšeme softwarový návrh a implementaci frameworku (systému). Kapitola je tedy rozdělena na dvě hlavní části na návrh a implementaci. Obě části jsou rozvrženy do menších podčástí, které se snaží kopírovat rozvržení zavedené v předchozí kapitole. Řídící jednotku zde zastupuje softwarová komponenta nazvaná *Marvin*, zatímco učící mechanismus je zastoupen komponentou *Sirius*. Distribuované zpracování je řešeno formou speciálního modulu v komponentě Sirius a je tedy popsáno u ní. Součástí návrhu, a především implementace, jsou i pomocné prográmky pro snazší práci s výstupy celého frameworku. Ty jsou popsány v sekci o implementaci.

4.1 Návrh

Systém je rozdělen na dvě hlavní části. Marvin je část starající se o řízení robota, zatímco Sirius je určen k učení robotů. Hlavní důraz je při návrhu dáván na snadnou rozšiřitelnost obou částí a na hladké propojení daných komponent. Dalším požadavkem je snadná konfigurovatelnost systému, která umožní experimentátorovi provádět spoustu pokusů bez nutnosti překompilovat kód.

K těmto částem patří ještě externí programy Gazebo a Player, které působí jako samostatné aplikace. Gazebo je simulátor 3D světa a Player je komunikační rozhraní, které přeposílá rozkazy buď do simulátoru, nebo k reálným robotům. Jejich použití je zachyceno v komunikačním schématu na obrázku 4.1. Sirius se stará o spouštění zbylých částí, tedy Gazeba, Playera a Marvina. Navíc nastavuje parametry marvinovi, který dále přes Playera ovládá fyzickou reprezentaci sebe sama v Gazebu. Opačným směrem, tedy z Gazebo do Marvina jdou informace do senzorů (opět přes Playera). Sirius si přímo komunikací s Gazebem zjišťuje, co se děje v simulaci,

a tuto informaci využívá pro fitness operátor (viz sekce 4.1.2).



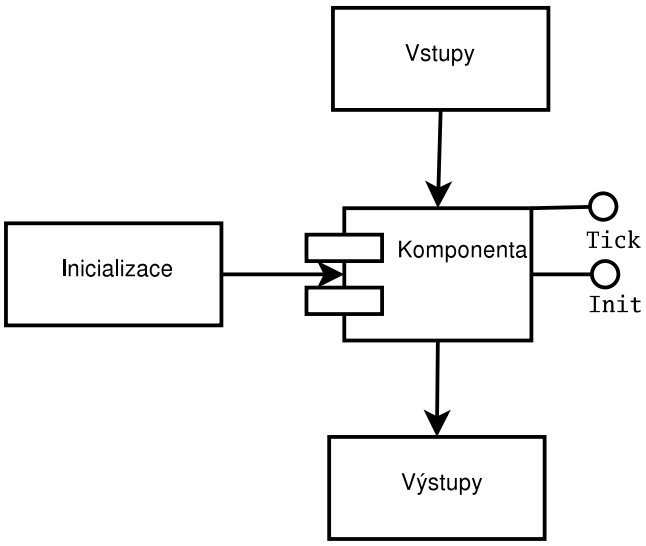
Obrázek 4.1: Komunikační schéma frameworku

4.1.1 Řídící jednotka (Marvin)

Každý robot se skládá z malých částí, komponent, které jsou mezi sebou propojeny pomocí vstupů a výstupů. Komponenta (obrázek 4.2) je základním kamenem robota v simulaci. Celý robot je pomocí nich sestaven. Rozšiřitelnost řídící jednotky je umožněna tím, že lze implementovat nové komponenty. Komponenta pro zbytek systému funguje jako černá krabička, která pouze dává k dispozici své vstupy a výstupy. Každý vstup respektive výstup má uveden druh dat které přijímá respektive produkuje. Připojit výstup z komponenty na vstup jiné jde, pouze pokud mají stejné typy. Vstupy do komponenty se navíc dělí na běžné a na inicializační. Běžné vstupy jsou takové, do kterých jsou připojeny výstupy z jiných komponent, oproti tomu inicializační vstupy se použijí pouze jednou při inicializaci komponenty. Každá komponenta navíc musí poskytnout metodu *Tick*, která přečte vstupy a vygeneruje výstupy, a metodu Init, která načte vstupy z inicializačních vstupů a nastaví si vnitřní prostředí tak, jak má. Výstupy z komponenty může odebírat libovolný počet jiných komponent.

Komponenty lze dále dělit do několika druhů, podle určení, k jakému účelu jsou zamýšleny:

- Zařízení - komponenta která se stará o nízkoúrovňové zařízení. Tato kom-



Obrázek 4.2: Komponenta

ponenta interně ovlivňuje běh simulace, případně čte stav simulace. Typické komponenty typu zařízení nemají buď externí vstupy, například kamera, která vstupy dostává ze simulace, nebo výstupy, například polohovací zařízení, které vstupy interpretuje jako impulzy do motorů a nemusí podávat žádné výstupy.

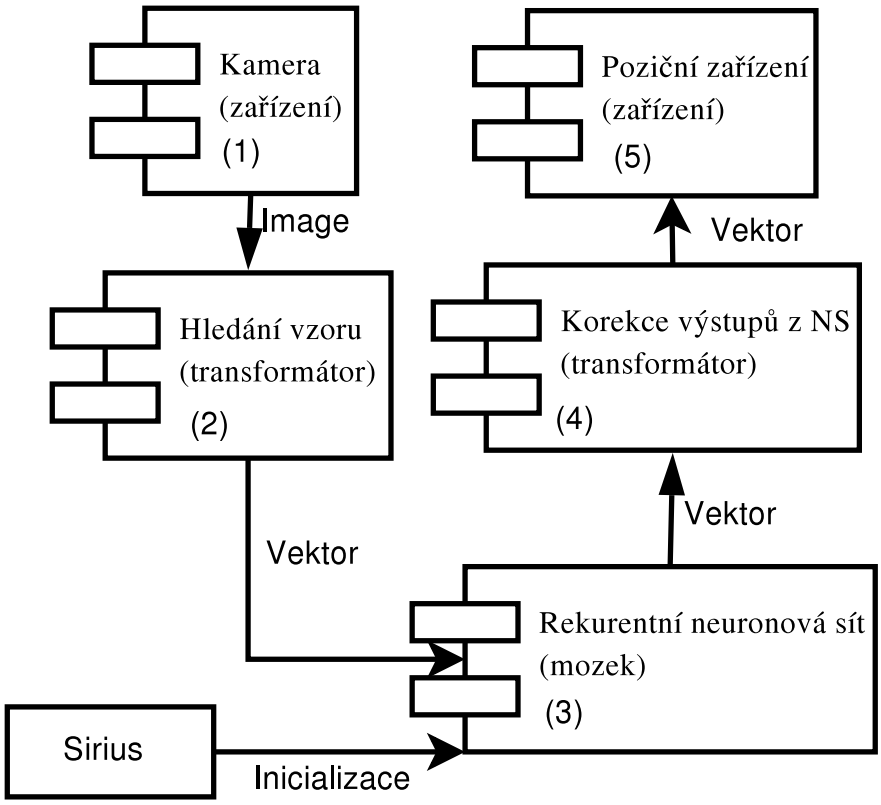
- *Mozek* - komponenta ,která na základě vstupů rozhodne a vygeneruje výstup. Navíc má inicializační vstupy, kterými může například nastavit rozložení neuronové sítě. Tato komponenta se od ostatních v logice odlišuje právě tím, že by to měla být komponenta, kterou se budeme snažit učit pomocí učícího systému. Požaduje tedy inicializační vstupy, které zásadně ovlivňují její chování. Typickým příkladem komponenty typu mozek je právě neuronová síť.

- *Transformátor* - komponenta která předem daným algoritmem zpracuje vstup a vygeneruje výstup. Příkladem může být například vyhledávač vzoru v obrazu, různé datové filtry, náhodný výběr z množiny nebo například převaděč dat jednoho typu na druhý atp.

Komponenty jsou ve výsledku propojeny v acyklickém orientovaném grafu. Každý takový graf musí mít alespoň jeden vrchol, do kterého nevede hrana a alespoň jeden vrchol, z kterého nevede hrana. Na každém takovém grafu lze provést topologické třídění, tedy uspořádat vrcholy grafu do lineárního seznamu tak, spoje v síti jdou vždy z levé části seznamu do pravé. Jeden *Tick* robota probíhá tak, že se provede *Tick* na každé jeho komponentě v pořadí daném topologickým uspořádáním. Tím je zajištěno, že každá komponenta má v době provádění metody *Tick*

připravena vstupní data z komponent jí předcházejících.

4.1.1.1 Příklad



Obrázek 4.3: Příklad jednoduchého robota

Na obrázku 4.3 je vidět jednoduché zapojení robota. Nejdříve přijde obraz z kamery (1), komponenta (2) se v něm pokusí najít vzor. Výstup dané komponenty je vektor, který je předán na vstup neuronové sítě (3). Výstup z neuronové sítě jde do (4), kde se transformuje do použitelné formy pro poziční zařízení (5), která již přímo ovládá motor robota. Jeden výstup může sloužit jako vstup několika zařízením, proto lze například výstup rekurentní neuronové sítě poslat ještě do ladící komponenty, která bude výstupy zaznamenávat do externího souboru.

Samotná konfigurace robota je plně popsána v XML, jehož formát a popis jednotlivých položek je popsán v příloze B.

4.1.2 Učení robotů (Sirius)

Učení robotů je implementováno pomocí genetických algoritmů s ohledem na možnost paralelizace. Sirius je samostatná aplikace, která načítá konfigurační soubory a provede inicializaci jednotlivých rozhraní a spustí provádění hlavního simulačního rozhraní.

Rozhraní v této aplikaci vycházejí z potřeby implementovat genetické algoritmy vycházející z algoritmu 3. Tedy pro jednotlivé kroky genetických algoritmů (popsáno v sekci 3.2.1) existuje vždy obecné rozhraní. Rozšiřitelnost této části je zajištěna možností implementovat vlastní rozhraní. Vysoké konfigurovatelnosti je dosaženo možností měnit jednotlivá rozhraní a také tím, že rozhraní mají poskytnut přístup do konfiguračního XML souboru, mohou tedy definovat nové položky v XML a načítat hodnoty určené uživatelem. Další základní funkcí rozhraní je schopnost serializace, tedy schopnost uložit svůj kompletní stav na disk, případně poslat po síti. Tuto vlastnost hojně využívá paralelní implementace genetických algoritmů.

Jednotlivé typy rozhraní:

- Evolution - základní rozhraní definující celou aplikaci. Vstupní bod celé aplikace. Provádí genetické algoritmy použitím ostatních rozhraní.
- Simulation - rozhraní pro simulaci. Zajišťuje spouštění Gazeba, Playera a Marvina. Simuluje běh jedné populace, tím zajistí zjištění hodnot fitness pro danou populaci. Spolupracuje s rozhraním Fitness.
- Initialization - rozhraní pro inicializaci populace, umožní nastavit parametry prvotní populaci. Nastaví základní parametry jako velikost populace, dobu po kterou má být jedinec simulován. Navíc otestuje zda jsou vstupní parametry kompatibilní se zbytkem nastavení. Tedy jestli například rekombinační operátory umí pracovat s nastaveným typem genu. Implementační detaily viz 4.2.3.1.
- Fitness - rozhraní pro hodnocení jedinců. Pomocí přímého přístupu k simulátoru Gazebo může číst informace přímo ze simulace a v závislosti na pozorovaných údajích hodnotit, jak dobře daný jedinec plní zadanou úlohu. Jejím výstupem je hodnota, absolutní číslo, které určuje míru toho, jak dobře byla zadaná úloha plněna. Tedy fitness rozhraní v podstatě definuje, co má daný jedinec dělat a co se tedy má naučit. Implementační detaily viz 4.2.3.3.
- Selection - rozhraní které na vstupu dostane populaci, kde každý jedinec má přiřazenu svoji fitness a na základě určitých pravidel vytvoří populaci novou. Toto rozhraní může i měnit počet jedinců v simulaci. Implementační detaily viz 4.2.3.4.

- Recombination - rozhraní pro přeměnu některých jedinců v stávající populaci. Podčástí tohoto rozhraní jsou metody pro mutace a pro křížení. Křížení vezme dva geny a z nich vyprodukuje jiné dva geny. Mutace oproti tomu dostane jeden gen a v něm provede nějaké změny. Tyto operace se ovšem nad jedinci z populace provádějí pouze s určitou pravděpodobností, a proto musí rozhraní umožnit konfigurovat i pravděpodobnosti s kterými se bude tato operace vykonávat. Implementační detaily viz 4.2.3.5.

- StopCondition - podmínka ukončení určuje, kdy zastavit běh algoritmu. K určení, jestli ještě pokračovat, má podmínka ukončení veškeré informace o právě vytvořené populaci a i o všech již předtím vytvořených. Může tedy například sledovat vývoj a zastavit, pokud se v posledních několika generacích již hodnoty fitness moc nemění. Implementační detaily viz 4.2.3.6.

Pokud mluvíme o jedinci dané populace, myslíme tím neprázdnou množinu robotů, kteří se účastní simulace, ovšem samotná definice pojmu jedinec závisí na konkrétní implementaci daných rozhraní, především rozhraní Simulation.

Pro běh programu je nutné určit následující:

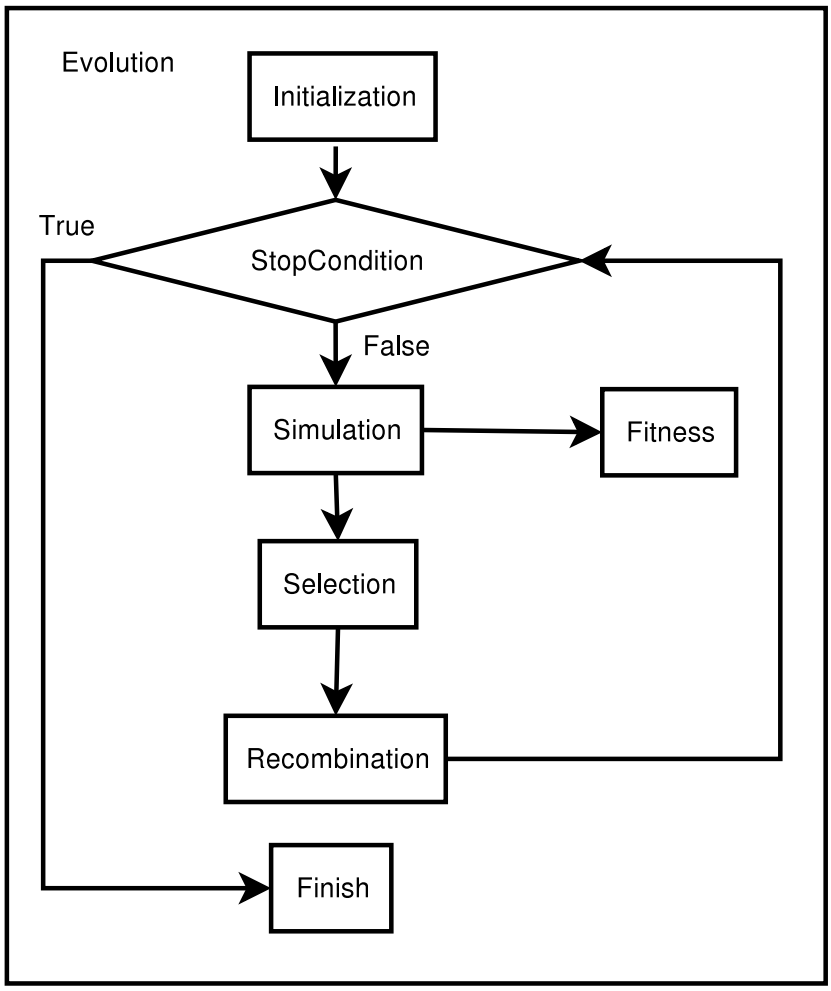
- Zvolit rozhraní - je nutné v konfiguraci vybrat instance rozhraní které chceme použít a to pro všechny výše popsané typy. Navíc je potřeba pro každé rozhraní nastavit správnou konfiguraci.

- Prostředí pro vývoj - definovat svět ve kterém bude žít náš jedinec. Samotné určení světa je necháno pouze na konfiguraci Gazeba. Podrobná specifikace definice světa je v dokumentaci [1].

Výstupem této aplikace jsou inicializační vstupy pro mozky učených robotů. Propojení jednotlivých rozhraní je na obrázku 4.4, ze kterého lze vyčíst pořadí volání jednotlivých rozhraní. Navíc je zde znázorněn Finish, který pouze naznačuje, že evoluce byla dokončena.

4.2 Implementace

Pro nejnižší simulační vrstvu implementace byl využit systém Gazebo. Celé prostředí je napsáno v programovacím jazyce C++. Další rozšíření je možné psát v C++ a přidávat formou dynamických knihoven.



Obrázek 4.4: Průběh evolučního algoritmu

4.2.1 Gazebo a Player

Pro simulaci robotů byl použit systém Gazebo a pro samotné ovládání robotů je použito abstraktní rozhraní Player. Díky těmto knihovnám by mělo být možné systém přenést do reálného prostředí na skutečné roboty. Více o Gazebu je v sekci 2.3.1.

Vzhledem k chybám a nedostatkům v implementacích Gazeba a Playera bylo nezbytné tyto knihovny pro potřeby této práci upravit. K hlavním úpravám patří:

- Oprava chyb v komunikaci mezi Gazebem a Playerem, která způsobovala ztrátu dat.
- Přidáno vykreslování mimo okno do Gazeba, za použití MesaGL[18]. Které je využíváno při paralelním zpracování na vzdáleném počítači.
- Přidání pozičních zařízení pro statické objekty. Gazebo je zaměřeno na to, aby se dalo použít jako náhrada reálného světa, tudíž nepodává informace, které v reálném světě nejdou zjistit. Pro vývoj robotů pomocí genetických algoritmů je však při hodnocení chování robota vhodné mít nějaké informace navíc, například absolutní polohu některých objektů.
- Přidání zpráv informujících o plné inicializaci Gazeba / Playera. Při paralelní simulaci je třeba pustit Gazebo i Playera bez asistence uživatele. Navíc je potřeba pustit je ve správném pořadí a ve správný čas. Tedy až v době, kdy je předchozí nainicializováno. Proto je v rámci této práce do Gazeba i do Playera přiděláno rozhraní pro komunikaci, pomocí kterého lze zjistit jejich stav. Lze potom tedy pustit Gazebo poté počkat dokud se nenainicializuje, pak až následně pustit Playera a opět počkat na jeho inicializaci.

4.2.2 Řídící jednotka (Marvin)

Základní třída *Marvin* slouží k propojení s Gazebem a Playerem a k držení acyklického orientovaného grafu komponent. Při každém kroku robota zpracovává komponenty podle topologického uspořádání. Navíc poskytuje Marvin pro zařízení schopnost načítat konfiguraci z XML a ukládat ji do XML.

Vstupy a výstupy jsou implementovány pomocí slotů. Slot je buď vstupní a nebo výstupní, navíc má určen datový typ, tedy co v daném slotu může být. Vstupní sloty jsou pro komponenty pouze pro čtení, zápis do nich může provádět pouze řídící třída, tedy Marvin. Oproti tomu výstupní data jsou zpřístupněna pro komponentu pouze pro zápis a číst z nich může pouze hlavní třída Marvin.

Popis formátu pro Marvina a jeho komponenty je v příloze B.

4.2.3 Učení robotů (Sirius)

Program Sirius je více specializovaný než Marvin, nicméně i tak umožňuje velkou rozšiřitelnost, díky možnosti implementovat vlastní jednotlivá rozhraní, která jsou popsána v sekci 4.1.2. Základem programu je evoluční algoritmus který je popsán algoritmem 4. Další důležitou částí je definování typu genomu a napojení genomu na inicializační vstupy Marvina, více v sekci 4.2.3.1. Běh algoritmu nejprve inicializuje úvodní generaci jedinců G v čase $t = 0$. Poté musí spočítat fitness pro úvodní populaci a poté v cyklu, dokud není splněna ukončovací podmínka, vytváří a ohodnocuje nové generace. Rozhraní Simulation je rozděleno do tří částí - *Begin*, *Advance* a *End*. *Begin* zinicializuje Gazebo a Playera a připraví ho k použití. *Advance* pro každého jedince v simulaci spočte jeho fitness pomoci Fitness rozhraní. A konečně *End* ukončí Gazebo a Playera.

Algoritmus 4 Evoluční algoritmus

```
t := 0
Initialize(G(t))
Fitness(G(t))
Simulation::Begin()
while not StopCondition(G(t)) do
  Simulation::Advance(G(t))
  Fitness(G(t))
  t := t + 1
  G(t) := Selection(G(t-1))
  Recombine(G(t))
end
Simulation::End()
```

V následujících sekcích jsou popsána jednotlivá rozhraní. Každé rozhraní je zděděno od *třídy I_Framework*, která poskytuje základní metody pro serializaci, inicializaci a deinicializaci. Serializací rozumíme schopnost uložit svůj stav do bufferu, respektive načíst svůj stav z bufferu. Serializace je potřebná jak z důvodů pozastavení výpočtu a pokračování později tak kvůli paralelizaci. V případě paralelního zpracování totiž často dochází k případům kdy je potřeba přenést konfiguraci celého rozhraní po síti na jiný počítač. Inicializace poskytuje prostor pro přípravu datových struktur a deinicializace dává prostor k úklidu.

```
class I_Framework:
{
  virtual ~I_Framework();
  virtual void Init( Population &pop );
  virtual void Deinit();
  virtual void LoadData( const InputBuffer &input );
  virtual void SaveData( OutputBuffer &output ) const;
```

```
}
```

4.2.3.1 Initialization

Toto rozhraní musí naplnit strukturu *Population* a inicializovat každého jedince. Typická implementace nastaví velikost populace, pravděpodobnostní meze pro křížení či mutace, vytvoří jedince a nastaví jim počáteční náhodný genom. Každý inicializátor musí implementovat rozhraní *I_Initializer*. Inicializace Marvina neprobíhá pomocí genomu, Marvin o genomu nic neví, ale pomocí transformátoru, který převede data z genomu na vstupní data pro komponentu Mozek. To znamená, že v rámci inicializační fáze musí implementátor poskytnout správný transformátor, pomocí metody *SetTransform*.

```
class I_Initializer: public I_Framework
{
public:
  virtual void Initialize( Population &p ) = 0;
  virtual void SetTransform( Transform *trans );
}
```

Jako příklad inicializátorů jsou implementovány *RandomInitializer* a *LoadFromFileInitializer viz příloha C.*

4.2.3.2 Simulation

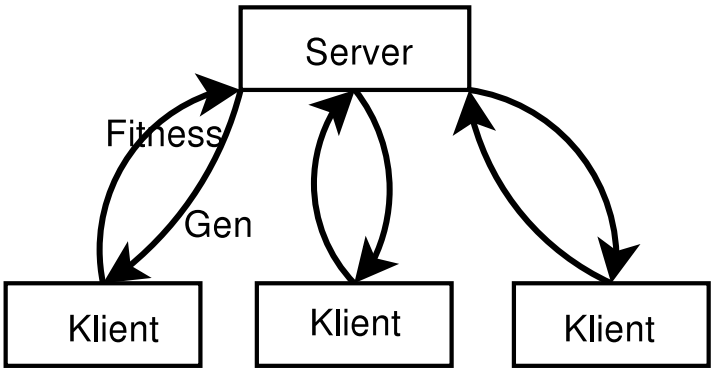
Simulační část vpustí jedince do simulace a nechá je řešit danou úlohu, která je určena fitness operátorem (viz 4.2.3.3). Standardní implementace spouští jedince sekvenčně. Pro inicializaci mozků používá transformační generátor zadaný do rozhraní *I_Initializer*.

```
class I_Simulation : public I_Framework
{
public:
  virtual void Advance( Population &p ) = 0;
  virtual void Begin() = 0;
  virtual void End() = 0;
};
```

4.2.3.2.1 Paralelní simulace Vzhledem k časové náročnosti simulace jedinců a nutné velikosti jednotlivých populací je implementována i distribuovaná verze simulačního rozhraní. Ta je implementována pomocí PVM [20], které umožňuje

spouštění dávkových úloh na mnoha počítačích. Nejmenší a již nedělitelnou jednotkou pro paralelní zpracování je simulace jednoho jedince v jedné generaci. Tedy posun simulace o jednu generaci probíhá tak, že se rozdělí simulace jedinců na více počítačů, samozřejmě čím více počítačů tím lépe. Ovšem nemá smysl mít více počítačů než je jedinců v generaci.

Samotná PVM simulace při zahájení (Begin) inicializuje klienty a v kroku Advance přeposílá své požadavky na klienty a čeká na výsledky. Navíc systém musí řešit situace kdy vzdálený klient neodpovídá. V takových případech se pokusí o vyřešení úlohy na jiném klientu a pokusí se restartovat rozhraní na počítači na kterém daná úloha selhala.



Obrázek 4.5: Server-klient komunikace

4.2.3.3 Fitness

Fitness operátor může každého jedince sledovat po celou dobu běhu simulace. Při startu simulace se mu zavolá *BeginSimulation*, při kterém si vynuluje hodnoty, v průběhu simulace se mu volá *StepSimulation*, a při ukončení simulace se zavolá *EndSimulation*.

```
class I_Fitness
{
public:
  virtual void BeginSimulation( World &w ) = 0;
  virtual void EndSimulation() = 0;
  virtual void StepSimulation( Marvin &m ) = 0;
}
```

Fitness operátor je jeden ze základních prvků který ovlivňuje co chceme robota naučit, proto je obtížné ho implementovat v obecné rovině. Pro test je implementován *DistanceFitness* operátor, který dává příznivé skóre pokud se robot přiblížil k

požadovanému objektu.

Fitness operátor je jediné místo kde je porušen zákaz čtení dat o simulaci, která robot ve fyzickém světě nemůže mít. Například absolutní pozice objektů v simulaci. Vzhledem k tomu, že toto je jediné místo odkud je povoleno přistupovat na simulaci tak lze tuto závislost odstranit napsáním fitness operátoru, který tyto informace nevyužívá. Nicméně přímý přístup do simulace Gazebo nepodporuje,) a proto byl do simulátoru přidělán v rámci implementace této práce.

4.2.3.4 Selection

Operátor selekce vybírá z populace jedinců s určitou pravděpodobností ty nejlepší. Pro test je implementována turnajová selekce s uchováním několika dosud nejlepších řešení, tzv. *elitismus*, *TournamentSelection*.

```
class I_Selection
{
public:
  virtual void Select( Population &p ) = 0;
}
```

4.2.3.5 Recombination

Rekombinační operátor definuje pravděpodobnosti pro křížení, respektive mutování jedinců a samotné křížící, respektive mutační operace.

```
class I_Recombine
{
public:
  virtual double GetMutateProb() = 0;
  virtual double GetCrossoverProb() = 0;
  virtual void Mutate( Genome &g ) = 0;
  virtual void Crossover( Genome &g ) = 0;
}
```

4.2.3.6 StopCondition

O ukončení se lze rozhodnout na základě vlastností populace, tedy už jsme našli jedince který plní úlohu uspokojivě, nebo podle počtu již hotových generací.

```
class I_StopCondition
```

```
{
public:
  virtual bool StopCondition( Population &p ) = 0;
}
```

Kapitola 5

Experimenty s frameworkem

Systém je navržen tak aby byl velmi rozšiřitelný a působí spíše jako stabilní kostra pro vývoj robotů. Pro demonstraci tohoto frameworku bylo vyvinuto několik modulů a nad nimi provedeny některé pokusy. Teoretické odůvodnění použitých metod je v kapitole 3. V této části si tedy popíšeme jak daný framework použít a k jakým výsledkům jsme došli.

Nejprve si tedy popíšeme úkol který danému robotovi uložíme, poté si v sekci 5.2 podrobněji rozebereme nastavení, které použijeme pro tento experiment a nakonec si popíšeme výsledky pro různé nastavení genetických algoritmů a pro různé neuronové sítě.

5.1 Úkol

Použití frameworku si ukážeme na jednoduchém příkladu, kde robot bude uvržen do arény a bude mít za úkol dojet k jednoznačně identifikovatelnému objektu, například k velké od pozadí odlišitelné kouli. V terénu mohou být nerovnosti, které ovšem robot umí překonat, mohou mu však zaclonit výhled na cíl. Robot se naviguje pouze pomocí kamery. Pohyb obstarávají 4 kola s náhonem na přední nápravu.

5.2 Řešení

5.2.1 Fáze

Úlohu budeme řešit pomocí postupu známého jako inkrementální evoluce, tedy rozdělíme si ji na jednodušší úlohy a to tak, že první úloha bude nejlehčí a další bude řešit vždy složitější úkol, nakonec ta poslední bude řešit přímo naši zadanou úlohu.

- najdi kuličku a dojeď ke kuličce v prostředí bez teréních nerovností,
- najdi kuličku a dojeď ke kuličce v prostředí s překážkami,
- najdi kuličku a dojeď k pohyblivé kuličce v prostředí s překážkami.

5.2.2 Sestavení robota

Robot bude sestaven z následujících komponent:

- Camera (*zařízení*) - toto zařízení nemá žádný vstup, svůj výstup generuje na základě vizuálních vjemů, které získává přímo od rozhraní Playera. Tedy s největší pravděpodobností bude výstup pocházet od simulátoru Gazebo.Rozlišeni kamery je 320x200x16.
- ShapeFinder (*transformátor*) - na vstupu dostává obraz z kamery a na výstupu vydá informaci o nalezeném vzoru. Výstupní vektor je tvaru (f, x, y, c, r), kde první hodnota je stav zda jsme vzor našli či nikoliv. Další dvě hodnoty (x, y) určují kde na obrazu se vzor nachází, c říká jak moc má vzor správnou barvu a nakonec r určuje jak moc si je ShapeFinder jistý podávaným výsledkem. Pokud vzor v obrazu není nalezen tak ShapeFinder vratí v prvním parametru 0 a v dalších lineárně sníženou hodnotu z minulého kroku. ShapeFinder v první fázi vyhledává kruhovou oblast (kulčka) a v druhé fázi navíc obdélníkovou oblast (zeď).
- RecurrentNetwork (*mozek*) - tato komponenta bude na základě vstupu z předchozí komponenty rozhodovat co má robot dělat. Právě tuto komponentu budeme učit pomocí genetickcýh algoritmů. Vnitřní struktura komponenty je rekurentní síť, kterou lze inicializovat pomocí vstupního vektoru vah (tento vektor bude výstupem učícího algoritmu). Právě změnou této komponenty provedeme několik pokusů, budeme testovat 4 druhy sítí - Elmanovu a Jordanovu síť a jejich modifikace, tyto sítě jsou popsány v 3.1.1. Síťe budou mít pět vstupů a pět výstupů.

- Constraints (*transformátor*) - výstup z neuronové sítě budeme interpretovat jednotným způsobem. A tento způsob bude určovat právě tato komponenta. Na vstupu je vektor pěti čísel (a, b, c, d, e), na výstup potřebujeme dostat dvě čísla. Impulz do motorů ($speed$) a rychlost otáčení ($turnrate$). Spočtení hodnot bude podle následujících vzorečků $speed = \sqrt{a^2 + b^2 + c^2}$ a $turnrate = 3 * (d - e)$. Tedy rychlost bude velikost vektoru prvních tří hodnot ze vstupu a otáčivost bude jednoduché lineární zobrazení posledních dvou hodnot ze vstupu.

- Position (*zařízení*) - vezme vstup a rovnou ho předá do nižší vrstvy kterou je rozhraní pro Playera. Player už sám vydá správnému robotovi impulz k pohybu podle předaných parametrů.

5.2.3 Mozek

Ze sekce 3.1.1 o neuronových sítí víme, že pro učení robotů je vhodnější použít rekurentní sítě než dopředné. Proto i mozek v našem Marvnovi bude formován rekurentní sítí. Porovnáme schopnost naučit se daný úkol tyto čtyři následující sítě:

- Elmanova, viz kapitola 3.1.5.1
- Elmanova s úpravou, viz kapitola 3.1.5.2
- Jordanova, viz kapitola 3.1.5.3
- Jordanova s úpravou, viz kapitola 3.1.5.4

5.2.4 Genetické algoritmy

Pro trénování neuronové sítě použijeme genetické algoritmy (GA). GA se v oblasti neuronových sítí používají převážně ve třech oblastech: trénování vah sítě, určení struktury sítě a hledání optimálního učícího algoritmu [6]. V této práci se omezíme na první případ, struktura sítě bude známá dopředu a o následné učení robota se již nestaráme. Dále je třeba rozhodnout o rozhraních popsaných v následujících sekcích.

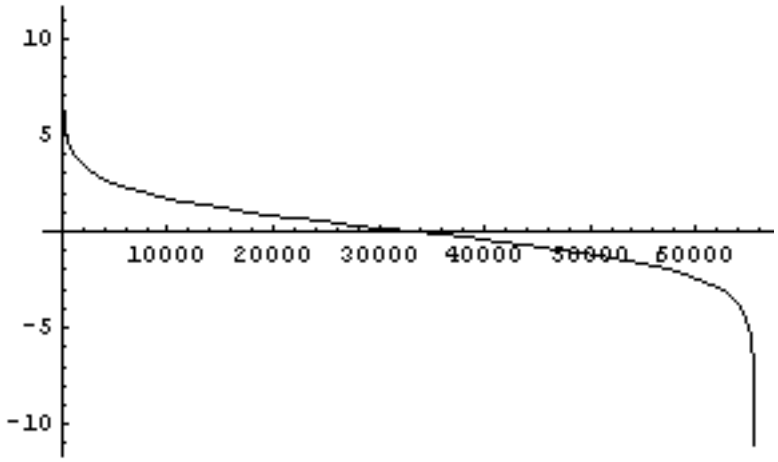
5.2.4.1 Populace

Zvolit velikost populace je nelehký úkol. Při zpracování na jednom počítači trvá simulace příliš dlouho. Při distribuovaném zpracování máme větší prostor k experimentům, přesto však nejsme neomezeni. Rozhodně nemá cenu zkoušet simulovat

populaci o méně než 50 jedincích.

5.2.4.2 Reprezentace jedinců

Vzhledem k tomu že struktura sítě je pevně daná potřebujeme reprezentovat pouze váhy v síti. Pro reprezentaci čísla zvolíme 16 bitové číslo které budeme na reálné číslo převádět pomocí funkce podobné inverzní k funkci sigmoidální. Tedy konkrétně $f(x) = ln(\frac{x+1}{2^{16}+1} - 1)$, viz obrázek 5.1. Pokud budeme potřebovat N vah potom sestavíme řetězec o délce N reálných čísel, kde každé číslo bude reprezentovat jednu váhu.



Obrázek 5.1: Dekódovací funkce

5.2.4.3 Inicializace

Inicializace pro první fázi bude náhodným generátorem, tedy pro každého jedince vygenerujeme náhodnou matici, reprezentující jeho mozek. V ostatních fázích využijeme hodnoty z předchozích fází.

5.2.4.4 Fitness operátor

Pro určení jak si který jedinec vede budeme hodnotit několik aspektů. První a nejdůležitější bude jak daleko bude jedinec od cíle po skončení přiděleného času, druhým bude rychlost jakou se jedinec přibliží k cíli a nakonec se vezme v potaz i průměrná doba pohybu správným směrem. Naopak negativní hodnoty bude nabývat takový jedinec který se bude od cíle vzdalovat.

5.2.4.5 Operátor selekce

Pro výběr nejlepších jedinců použijeme turnajovou selekci [7] se zachováním dvou nejlepších jedinců. Turnajová selekce probíhá v turnajových kláních, vždy tak že se náhodně vybere n jedinců a z nich se do nové populace s největší pravděpodobností dostane ten nejlepší. Každý jedinec může počítat s tím že se do turnaje dostane přibližně n-krát. Tedy nejlepší jedinec by měl souboj vyhrát n-krát a nejhorší jedinec by měl všech svých n soubojů prohrát.

5.2.4.6 Rekombinační operátor

V použitém přikladu aplikujeme jak křížení tak mutaci. Křížení s pravděpodobností okolo 0.85 a mutace s pravděpodobností okolo 0.05.

Křížení provedeme jednobodové, tedy pro dvě křížené neuronové sítě to bude znamenat že si prohodí váhy v některých svých spojích. Tímto nevznikají nové váhy ve spojích, toto je plně ponecháno na mutaci.

Mutace náhodně změní L čísel v celém řetězci, kde $0 < L \leq N$. Změna probíhá pomocí creep mutace s parametrem 0.5, která je popsaná algoritmem 2.

5.2.4.7 Podmínka ukončení

Ukončení nastane pokud najdeme jedince, který plní daný úkol dostatečně dobře a nebo pokud proběhl předem daný počet generací.

5.3 Experimenty

5.3.1 Fáze 1

V první fázi jsme si dali za úkol najít kuličku, která je vždy na stejném místě. Je nutno nastavit pokus tak aby všichni roboti v populaci měli stejné podmínky. Výsledky by jistě negativně ovlivnilo kdyby měl každý robot jinou startovní pozici, tedy obecně by byl různě daleko od vyhledávané kuličky, navíc některý robot by kuličku již od začátku viděl a jiný ne. Proto budeme roboty pouštět vždy ze stejného místa, ale abysme se vyhnuli přeučení a specializaci na jednu jedinou pozici a orientaci tak jedince vypustíme do simulace, v rámci vyhodnocování jeho fitness hodnoty, vždy třikrát pokaždé s jiným natočením vůči kuličce. Tento problém by se dal řešit i úpravou fitness funkce, tedy tak aby byl robot s lepší počáteční

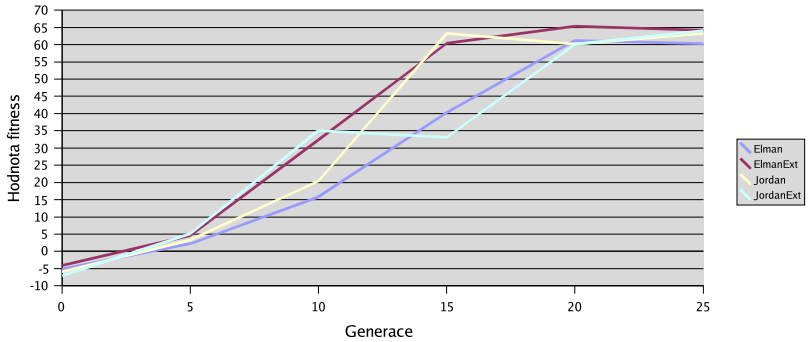
pozicí penalizován a naopak aby robot s horší pozicí dostal bonus. Tento přístup má ovšem tu nevýhodu, že spoléhá na to, že člověk správně odhadne penalizační funkci, proto zůstaneme u prvního přístupu.

V našem pokusu nejprve porovnáme jak se danou úlohu učili jednotlivé sítě při stejném počtu generací, v tabulce 5.1 jsou porovnány hodnoty z fitness funkcí. U každé sítě je vždy průměrná a nejlepší hodnota fitness funkce. Hodnota je získána ze simulace populace o velikosti 50 jedinců, vždy jde o průměr z pěti puštění. V následujících grafech je vyjádřen vztah mezi průměrnými (obrázek 5.2) a maximálními (obrázek 5.3) hodnotami.

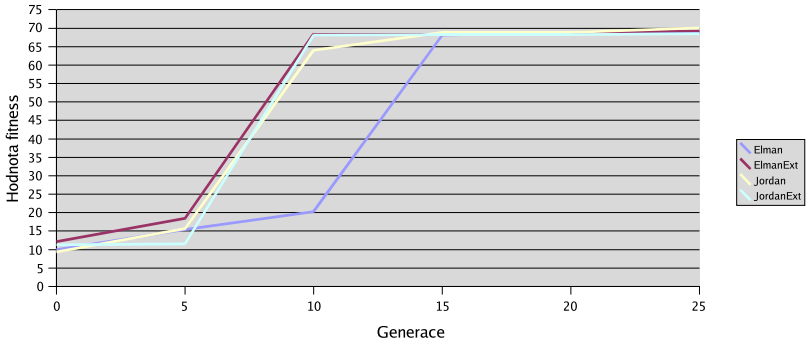
	Elman		ElmanExt		Jordan		JordanExt	
Gen	Avg	Best	Avg	Best	Avg	Best	Avg	Best
0	-5.2	10	-4.1	12.1	-5.9	9.3	-7.1	11.3
5	2.2	15.4	4.6	18.4	3.3	15.6	5.1	11.5
10	15.8	20.3	32.4	68.4	20.4	64.0	35.1	68.1
15	40.3	68.1	60.4	68.4	63.3	68.9	33.1	68.3
20	61.2	68.9	65.3	68.9	60.2	68.9	60.1	68.3
25	60.3	68.9	64.2	69.3	63.2	70.1	64.1	68.5

Tabulka 5.1: Fáze 1, porovnání fitness

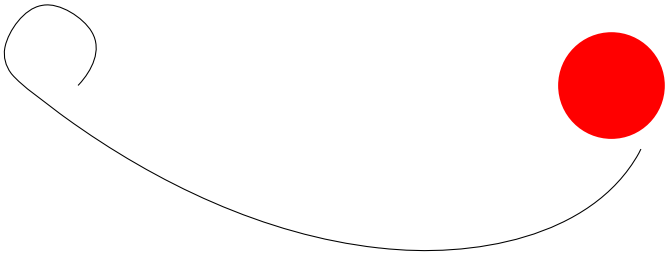
Z grafů je vidět že všechny sítě se požadovanou úlohu naučily v prvních 20 generacích. Proto do dalších fází nemá cenu vycházet z pozdějších generací, kde už je potlačena variabilita samotných jedinců a je větší pravděpodobnost, že v nich bude více jedinců řešící úlohu stejným způsobem. Na obrázku 5.4 je znázorněna trasa nejlepšího jedince. Kulička je znázorněna červeným kruhem a trasa černou čárou, je vidět že se robot nejdřív otočil a pak se teprve vydal za kuličkou. Tato strategie se projevila jako vítězná především díky tomu, že se robotovi podařilo u kuličky zastavit, tedy neposunul kuličku a nedostal postih.



Obrázek 5.2: Fáze 1, graf průměrných fitness



Obrázek 5.3: Fáze 1, graf maximálních fitness



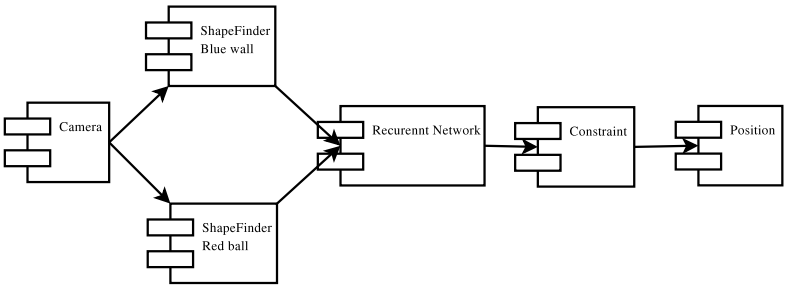
Obrázek 5.4: Fáze 1, trasa nejlepšího jedince

5.3.2 Fáze 2

Druhá fáze vychází z první a je zde snaha naučit robota dojet ke kuličče v prostředí s překážkami. Do simulačního prostředí vložíme zeď, a aby to robot neměl tak lehké tak ji vložíme mezi něj a kuličku. Tuto úlohu s výše popsaným nastavením zkusíme řešit pouze pro Elmanovu a Jordanovu síť bez jejich rozšíření. Využijeme naučené chování z předchozí fáze, ovšem vzhledem k tomu že od 20 generace dál už se nevyskytovalo zlepšení tak pro druhou fázi použijeme populaci z 20 generace. Z tabulky 5.2 je vidět, že se něco nepovedlo. Při takto malých fitness hodnotách je zřejmé, že se robotům nedaří úkol splnit. Při studování chování naučených robotů lze pozorovat, že jim chybí informace o zdi, vzhledem k tomu že tato informace nejde do neuronové sítě tak je to pro mozek robota jako kdyby ji neviděl. Tedy vítězná strategie z předchozí fáze, otáčej se dokud nevidíš kuličku a pak se k ní vydej, v tomto případě nemohla zabrat. Neuronovou síť, tedy mozek, robota modifikujeme tak, že přidáme šestý vstup do sítě, který napojíme na všechny kontextové jednotky sítě. Nový vstup bude dostávat informace z nové komponenty WallDetector, která bude zjišťovat zda má robot ve výhledu zeď, či nikoliv[1]. Nové zapojení Marvina tedy bude vypadat tak jako na obrázku 5.5.

	Elman		Jordan	
Gen	Avg	Best	Avg	Best
0	-12.2	2	-4.3	2.3
5	-11.2	1.2	-3.1	2.5
10	-11.8	3.1	-3.3	2.6
15	-3.3	3.2	-3.1	3.3
20	-2.2	3.2	-3.1	3.4
25	1.1	3.3	-3.1	3.5

Tabulka 5.2: Fáze 2 první pokus, porovnání fitness



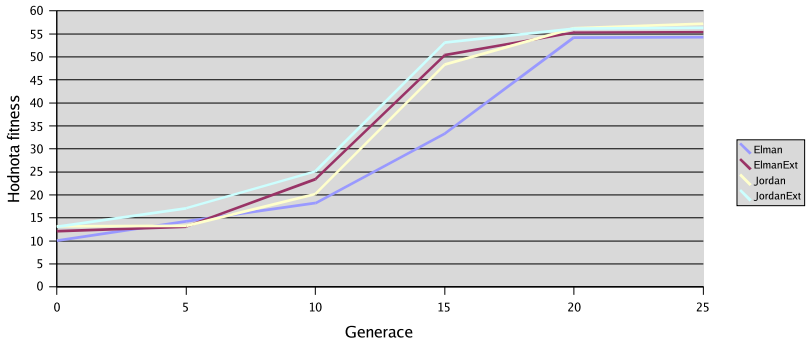
Obrázek 5.5: Zařízení pro robota ve fázi dvě

[1]Díky šikovně zvolenému rozšíření není potřeba znova předělávat fázi 1. Protože ve fázi 1 by nebyla zeď detekována (žádná tam není), a tudíž by šestý vstup měl nulový přínos, tedy jako kdyby tam nebyl.

Při tomto zapojení zkusíme sledovat již opět všechny 4 sítě. Opět navážeme na 20 generaci z předchozí fáze. Tabulka 5.3, ukazuje hodnoty fitness obdobně jako tabulka pro fázi 1. Grafické znázornění je opět v grafech průměrných hodnot (obrázek 5.6) a maximálních hodnot (obrázek 5.7). Opět lze pozorovat, že rozšířená Jordanova síť (JordanExt) se úlohu naučila nejrychleji. Na obrázku 5.8 je opět znázorněna trasa nejlepšího jedince. Z trasy je vidět že se robot naučil dojet k levému okraji zdi a když zeď ztratil z výhledu tak se začal otáčet dokud neviděl kuličku, k té pak vyrazil a dokroužil k ní, aniž by do ní narazil.

	Elman		ElmanExt		Jordan		JordanExt	
Gen	Avg	Best	Avg	Best	Avg	Best	Avg	Best
0	10.1	15.1	12.1	15.3	13.2	16.3	13.1	17.3
5	14.2	16.4	13.1	24.4	13.3	16.6	17.1	22.5
10	18.2	20.3	23.4	24.4	20.2	23.6	25.1	62.1
15	33.3	62.1	50.4	63.1	48.3	63.1	53.1	64.1
20	54.2	62.9	55.3	63.1	56.2	63.1	56.1	64.1
25	54.3	62.9	55.4	63.1	57.2	63.1	56.3	64.1

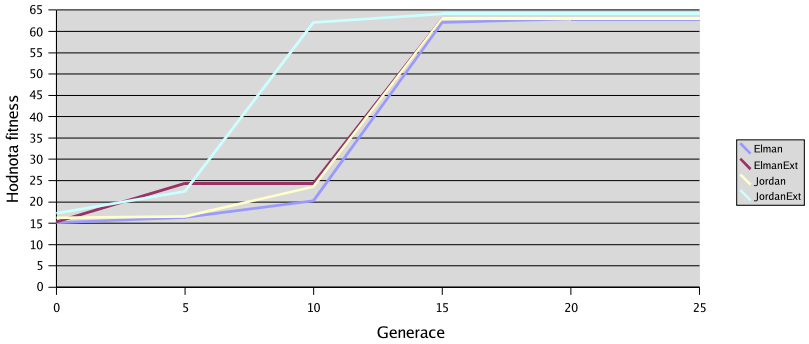
Tabulka 5.3: Fáze 2 druhý pokus, porovnání fitness



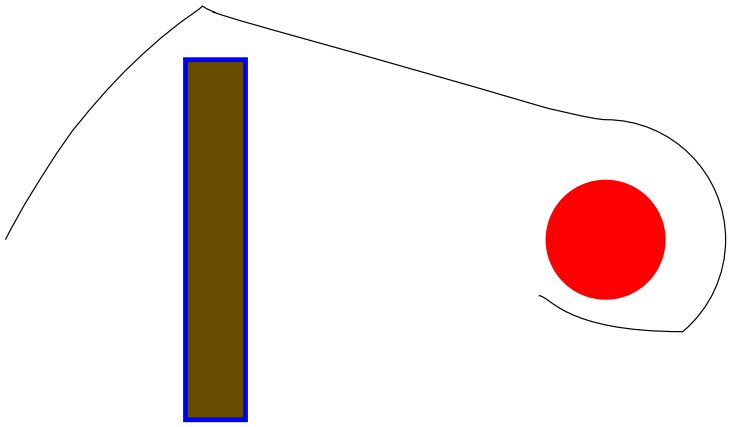
Obrázek 5.6: Fáze 2, graf průměrných fitness

5.3.3 Fáze 3

Ve fázi 3 přidáme k předchozí jen malou změnu a to sice tu, že kuličkou začneme pohybovat. Tohoto docílíme tak, že upravíme rozhraní fitness. Rozhraní fitness je pro tento úkol to nejpřirozenější místo, již proto že je to jediné místo odkud z programu přímo ovlivňujeme simulaci. Při zkoumání výsledků z této fáze je třeba si nejdříve všimnout, že již nestačilo 25 generací ale k rozumnému výsledku se dospělo až po 35 generacích. Navíc úlohu se nepodařilo naučit neupravenou elmanovu síť, navíc z vývoje hodnot průměrných fitness se zdá, že ani větší počet gene-



Obrázek 5.7: Fáze 2, graf maximálních fitness

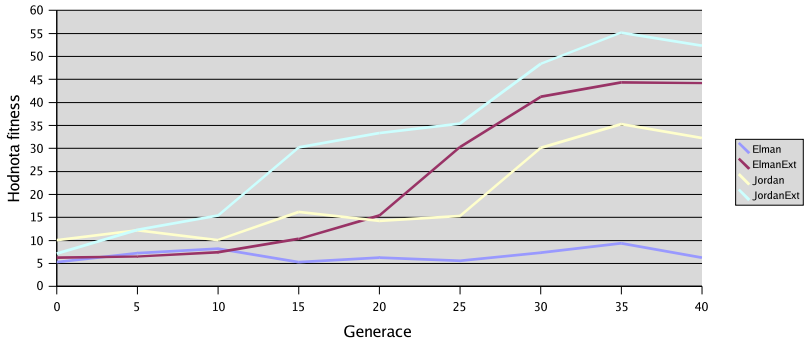


Obrázek 5.8: Fáze 2, trasa nejlepšího jedince

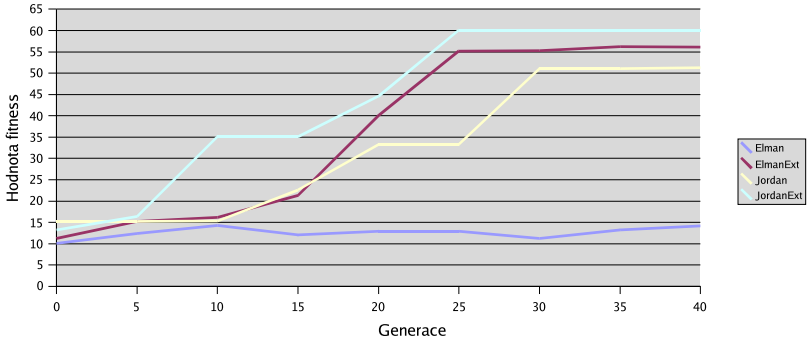
rací by elmanově síti nepomohl. Fáze 3 opět vychází z populace z předchozí fáze a to z patnácté generace. V tabulce 5.4 jsou uvedeny naměřené hodnoty fitness. Nejlépe a nejrychleji se naučila danou úlohu řešit rozšířená Jordanova síť. Grafické znázornění průběhu vývoje je znázorněno v grafech průměrných hodnot (obrázek 5.9) a maximálních hodnot (obrázek 5.10).

	Elman		ElmanExt		Jordan		JordanExt	
Gen	Avg	Best	Avg	Best	Avg	Best	Avg	Best
0	5.3	10.1	6.3	11.2	10.1	15.2	7.1	13.2
5	7.2	12.4	6.5	15.2	12.2	15.2	12.3	16.4
10	8.2	14.3	7.4	16.2	10.1	15.3	15.4	35.1
15	5.3	12.1	10.3	21.3	16.2	22.6	30.2	35.1
20	6.3	12.9	15.4	40.1	14.2	33.3	33.3	44.6
25	5.6	12.9	30.3	55.2	15.3	33.3	35.4	60.1
30	7.3	11.2	41.2	55.3	30.1	51.1	48.4	60.1
35	9.4	13.2	44.3	56.2	35.3	51.1	55.2	60.1
40	6.3	14.2	44.2	56.1	32.3	51.3	52.3	60.1

Tabulka 5.4: Fáze 3, porovnání fitness



Obrázek 5.9: Fáze 3, graf průměrných fitness



Obrázek 5.10: Fáze 3, graf maximálních fitness

Kapitola 1

Závěr

Cílem práce bylo implementovat systém pro vývoj robotů s vizuálním vnímáním, s důrazem na široké možnosti konfigurovatelnosti. Bylo potřeba navrhnout systém, který by umožnil vyvíjet roboty v čistě virtuálním prostředí.

Cíl práce byl splněn včetně rozšíření o paralelní zpracování, které umožňuje urychlit vývoj robotů použitím více počítačů. Systém funguje i na pracovních stanicích za běžného provozu — běží s nízkou prioritou — počítá i s tím, že nepřítel vypne pracovní stanici, případně že administrátor násilně ukončí jeden z běžících procesů. Systém umožňuje vývoj robotů pomocí konfigurovatelných genetických algoritmů a dává podporu pro řízení robotů neuronovými sítěmi. Pro rozpoznávání obrazu byla vyvinuta komponenta pro hledání vzorů. Pro vizuální vjemy lze použít libovolné zařízení používáné systémem Gazebo (tzn. kamera, sonar, laser).

V porovnání s existujícími systémy, které jsou popsané v kapitole **??**, poskytuje tento systém jako jediný možnost paralelního zpracování v kombinaci s 3D vizualizací a plnou fyzikální simulací. Přitom umí transparentně pracovat i s reálnými roboty.

Na příkladu z kapitoly **??** je předvedeno, že je systém schopen naučit roboty vykonávat určenou akci – vyhnout se zdi a dojet ke kuličce, která se může i pohybovat. Užití je předvedeno včetně kompletního nastavení programů a analýzy výsledků. K vývoji robotů byla použita paralelní verze programu, využito bylo 40 klientů a jeden server. Vyzkoušení jedne gen-

erace jedinců trvalo přibližne 15 minut a generací pro jeden pokus bylo 25 až 40. V rámci pokusu bylo ověřeno, že systém se dovede vypořádat s výpadky klientských stanic. Jako druhotný výsledek práce se povedlo ověřit, že přidání *self-feedback* vazeb do Elmanovi sítě (viz sekce **??**) rozšířilo její možnosti a umožnilo ji naučit se složitější úlohu (viz **??**).

Tento systém lze dále rozšiřovat o mnoho zařízení pro program Marvin a nebo o rozhraní pro program Sirius. Bylo by jistě zajímavé pokusit se využít natrénované sítě na skutečných robotech. Systém umožňuje i simulaci více robotů, proto by bylo zajímavé vyzkoušet učit model kořist – dravec. Systém neklade v podstatě žádné meze, limitujícím faktorem je pouze uživatelova fantazie.

Dodatek A

Uživatelská dokumentace

Uživatelská dokumentace provede uživatele instalací programu poté odkáže do správných příloh na nastavení programu a nakonec řekne jak se dá program pustit.

A.1 Instalace

Framework je závislý na několika externích knihovnách, které musejí být nainstalovány před instalací samotného frameworku. Nejprve je potřeba nainstalovat knihovnu ODE.

A.2 Nastavení

Veškeré nastavení programu se provádí pomocí XML souborů. Podrobný popis formátu těchto souborů je v přílohách B a C. Tím lze konfigurovat vše kromě konfigurace světa pro simulátor Gazebo, popis formátu konfiguračního souboru pro Gazebo lze nalézt na domovských stránkách projektu [1]. Příklady nastavení všech těchto částí je v příloze D.

A.3 Spouštění

Při spouštění samotného učení je potřeba rozlišit dva případy a to sice zda učíme pouze pomocí jednoduché neparalelní simulace či pomocí paralelní simulace. V prvním případě stačí spustit příkaz

```
./sirius /cesta/ke/konfiguraci/konfigurace.xml
```

ve druhém je ovšem třeba nejprve pustit PVM a přidat do něj co nejvíce hostitelských počítačů. O konfiguraci a spouštění PVM lze více přečíst na domácích stránkách projektu[20]. Po spuštění PVM je už možné pustit program stejně jako v přechozím případě.

K prohlížení a analýze výsledku slouží dva programy.

- replay - Program určený k přehrávání práce robotů naučených v simulaci. Následující příklad spustí přehrávání jedince z populace 12, který má druhou nejlepší fitness. Konfigurace jedince se bere z marvinconf.xml, takže pokud nejsou inicializační vstupy kompatibilní tak se bude robot s největší pravděpodobností chovat zmateně. Pozor pro replay je nutné pustit Gazebo i Playera vlastnoručně.

  ```
  #Pust' Gazebo s rozhraním pro zobrazování výsledků
  #na obrazovku.
  wxgazebo worldfile.xml
  #Pust' Playera a připoj se k puštěnému Gazebu
  player -g 0 gazebo.cfg
  #A nakonec spust' replay
  ./replay marvinconf.xml -p \
  /cesta/k/populacim/pop12.pop -r 2
  ```

- popviewer - Program na prohlížení fitness hodnot. Pomocí něho lze zjistit jaké jsou hodnoty fitness a jak se chovali nejlepší jedinci.

  ```
  #Zjisti statistické informace ze všech generací
  ./popviewer -s /cesta/k/pop/pop
  #Zjisti statistické informace o jedné populaci
  ./popviewer -p /cesta/k/pop/pop13.pop
  ```

Dodatek B

XML formát pro Marvina

Formát XML pro program Marvin definuje jednoduchou strukturu. Kořenový element se bude vždy jmenovat *marvin*. Uvnitř tohoto elementu jsou očekávány několikanásobné výskyty elementů *device* a *connection*.

B.1 Device

Element device popisuje jedno zařízení. Tento element má dva povinné atributy, *name* a *id*. Atribut *name* určuje které zařízení konfigurujeme, musí to být jedno ze zařízení které poskytuje program Marvin. Atribut *id* určuje jaké id bude mít toto zařízení.

Podelementy jsou *initializers*,*inputs* a *outputs*. Initializers určuje jaké inicializační vstupy daná komponenta má. Inputs určují jaké vstupy a jakého druhu jsou v komponentě a nakonec Outputs určují jaké poskytuje komponenta výstupy. Všechny tyto elementy mohou mít jako své podelementy, elementy typu slot, který je popsán níže.

B.1.1 Slot

Slot je element který popisuje vstup, výstup nebo inicializační vstup komponenty. Každý slot má povinný atribut *id*, který daný slot v kombinaci s typem slotu a *id* zařízení jednoznačně identifikuje. Popis jednoho slotu sestává z elementů *required* a *item*. Element required určuje zda je povinné aby tento slot byl zapojen, tedy pokud je požadován a není zapojen, program vyhlásí chybu již v inicializaci. Element *item* určuje datový typ, který slot umí přijímat, respektive který vydává, navíc je

možné v položce *item/data* určit *implicitní data pro daný vstup, která ovšem mohou být programem přetížena.*

B.2 Connection

V elementech connections je určeno propojení jednotlivých zařízení. Tedy napojení výstupů komponent na vstupy jiných. Propojení je provedeno pomocí dvou elementů *source* a *target*. Oba tyto elementy mají dva povinné atributy - *device* a *slot*, pomocí kterých je určeno z kterého slotu na kterém zařízení se má vzít výstup (v případě elementu *source*), respektive vstup (v případě elementu target).

Nakonec uvedeme fiktivní příklad s jednoduchým zapojením.

```
<marvin>
    <device name='Motor' id='0'>
        <initializers/>
        <inputs>
            <slot id='0'>
                <required value='1'/>
            </slot>
        </inputs>
        <outputs/>
    </device>
    <device name='Mozek' id='1'>
        <initializers/>
        <inputs>
            <slot id='0'>
                <required value='1'/>
            </slot>
        </inputs>
        <outputs>
            <slot id='0'>
                <required value='1'/>
                <item>
                    <name value='Vector'/>
                </item>
            </slot>
        </outputs>
    </device>
    <device name='Kamera' id='2'>
        <initializers/>
        <inputs>
```

```
            <slot id='0'>
                <required value='1'/>
            </slot>
        </inputs>
        <outputs>
            <slot id='0'>
                <item>
                    <name value='Vector'/>
                        <data/>
                </item>
             </slot>
        </outputs>
    </device>
    <connection>
        <source device='1' slot='0'/>
        <target device='0' slot='0'/>
     </connection>
    <connection>
        <source device='2' slot='0'/>
        <target device='1' slot='0'/>
    </connection>
</marvin>
```

Dodatek C

XML formát pro Sirius

Program Sirius předpokládá v configuračním XML souboru výskyt kořenového elementu *sirisu*. A 7 podelementů *initializer, simulation, fitness, selection, recombine, stopcondition* a *transform*. Každý z těchto podelementů má jeden atribut *name,* který určuje které z implementací daného rozhraní se má použít. Dale popíšeme dostupné parametry pro existující implementace rozhraní.

C.1 Initializer

Inicializační rozhraní jsou implementována dvě. Inicializace náhodnými hodnotami a inicializace načtením předchozí populace. Obě sdílejí společný konfigurační základ, a to elementy.

- *PopulationsSize* - v atributu *value* udává očekávanou velikost populace.
- *SliceTime* - v atributu *value* udává dobu po kterou má simulace trvat. Doba je určena počtem tiků robota.
- *Generation - v* atributu *value* je kolikátou generací se má začít. Jde jen o číslování první generace.
- GenomeLength - v *atributu value* je určena délka genu, který má inicializovat.
- SaveDirectory - v atributu *value je cesta k adresáři do kterého se mají ukládat mezivýsledky.*

Implementace inicializačního rozhraní pomocí náhodných čísel nezavádí žádný nový parametr, implementace pro načítání již existující populace zavádí nový ele-

ment *LoadDirectory* ve kterém je určen adresář a soubor ze kterého se má načíst úvodní populace jedinců.

C.2 Simulation

Implementace simulačních rozhraní sdílí element *MarvinConfig*. Existující implementace Basic, pro simulaci na jednom počítači, další element nezavádí. Oproti tomu PVM, pro paralelní simulaci, zavádějí nový element *catchout*.

- *MarvinConfig - v atributu* file *je jméno konfiguračního souboru pro Marvina a v atributu* path *je cesta k tomuto souboru.*
- *catchout* - v atributu *value je buď 0, která říká že nechceme dostávat výstup od klientských aplikací, nenulová číselná hodnota zaručuje odchytávání výstupu od klientů.*

C.3 Fitness

Implementovány jsou dvě rozhraní tohoto typu. *Distance* a *DistanceWithWall*. Ani jedna z těchto komponent nezavádí konfigurační elementy.

C.4 Selection

Implementováno je pouze rozhraní pro turnajovou selekci, *TournamentSelection*, a to nezavádí žádné konfigurační elementy.

C.5 Recombine

Rekombinační operátor je implementován jen jeden *OneCrossPointArra*y a ten nezavádí žádné konfigurační elementy.

C.6 StopCondition

Bylo implementováno jedno rozhraní typu *StopCondition a to GenerationsCount*, které udává jeden element MaxGenerations.

- *MaxGenerations - v atributu* value je určeno po kolikáté generaci se má zastavit výpočet.

C.7 Transform

Transformační rozhraní je jedno *GSA2Vec*, které nemá žádné konfigurační elementy.

Dodatek D

Příklady nastavení

D.1 Fáze 1

Ukázka nastavení bude vždy pro případ s Elmanovou sítí, pro jiné druhy sítí je změna jednoduchá, výměnou zařízení s *id* 1.

D.1.1 Nastavení Marvin

```
<marvin>
<!-- Poziční zařízení, nemá žádné výstupy pouze vstupy. -->
    <device name='PositionGazebo' id='0'>
<!-- Toto zařízení nelze konfigurovat. -->
        <initializers/>
<!-- Vstup je jeden a je povinný -->
        <inputs>
            <slot id='0'>
                <required value='1'/>
            </slot>
        </inputs>
        <outputs/>
    </device>
<!-- Neuronová síť -->
    <device name='ElmanNet' id='1'>
        <!-- Inicializace -->
        <initializers>
            <slot id='0'>
                <required value='1'/>
```

```
                <item>
                    <name value='Vector'/>
                    <data>
                        <items>
<!-- Počet vstupních neuronů -->
                            <item value='5'/>
<!-- Počet neuronů ve skryté vrstvě -->
                            <item value='5'/>
<!-- Počet kontextových jednotek -->
                            <item value='5'/>
<!-- Počet výstupních neuronů -->
                            <item value='5'/>
<!-- Počet spojů celkem -->
                            <item value='85'/>
<!-- Počet neuronů celkem -->
                            <item value='20'/>
                        </items>
                    </data>
                </item>
            </slot>
            <slot id='1'>
                <required value='1'/>
                <item>
                    <name value='Vector'/>
                        <data>
                            <items/>
                        </data>
                 </item>
            </slot>
           <slot id='2'>
                <required value='1'/>
                <item>
                    <name value='Vector'/>
                    <data>
                        <items/>
                    </data>
                </item>
            </slot>
        </initializers>
        <inputs>
            <slot id='0'>
                <required value='1'/>
            </slot>
```

```
    </inputs>
    <outputs>
        <slot id='0'>
            <required value='1'/>
            <item>
                <name value='Vector'/>
                <data/>
            </item>
        </slot>
    </outputs>
</device>
<device name='SpeedConstraints' id='2'>
    <initializers/>
    <inputs>
        <slot id='0'>
            <required value='1'/>
        </slot>
    </inputs>
    <outputs>
        <slot id='0'>
            <item>
                <name value='Vector'/>
                    <data/>
            </item>
         </slot>
    </outputs>
</device>
<device name='CameraGazebo' id='3'>
    <initializers/>
<inputs/>
<outputs>
    <slot id='0'>
        <required value='1'/>
            <item>
                <name value='Image'/>
            </item>
        </slot>
    </outputs>
</device>
<device name='ImageBallFinder' id='4'>
    <initializers/>
    <inputs>
        <slot id='0'>
```

```
                <required value='1'/>
            </slot>
        </inputs>
        <outputs>
            <slot id='0'>
                <required value='1'/>
                <item>
                    <name value='Vector'/>
                    <data/>
                </item>
            </slot>
        </outputs>
    </device>
<!-- Propojení zařízení -->
<!-- ElmanNet:0 -> SpeedConstraints:0 -->
    <connection>
        <source device='1' slot='0'/>
        <target device='2' slot='0'/>
     </connection>
<!-- SpeedConstraint:0 -> PositionDevice:0 -->
     <connection>
        <source device='2' slot='0'/>
        <target device='0' slot='0'/>
    </connection>
<!-- CameraGazebo:0 -> ImageBallFinder:0 -->
    <connection>
        <source device='3' slot='0'/>
        <target device='4' slot='0'/>
    </connection>
<!-- ImageBallFinder:0 -> ElmanNet:0 -->
    <connection>
        <source device='4' slot='0'/>
        <target device='1' slot='0'/>
    </connection>
</marvin>
```

D.1.2 Nastavení Sirius

```
<sirius>
<!-- Inicialize původní populace náhodnými
     hodnotami -->
    <initializer name='Random'>
        <PopulationSize value="5"/>
```

```
        <SliceTime value="300"/>
        <Generation value="0"/>
        <GenomeLength value="105"/>
        <SaveDirectory value=
         "/cesta/k/adresari/pro/ulozeni/populace"/>
    </initializer>
<!-- Simulace pomocí PVM simulačního rozhraní -->
    <simulation name='PVM'>
<!-- elman.xml je soubor, který je popsaný výše -->
        <MarvinConfig
            file='elman.xml'
            path='/cesta/k/konfiguraci/pro/marvina/'/>
<!-- odkomentováním tohoto řádku, docílíme
     přijímání výstupů na serveru od klientů -->
        <!--<catchout value="1"/>-->
    </simulation>
<!-- Volba fitness -->
    <fitness name='Distance'>
    </fitness>
<!-- Volba selekce -->
    <selection name='Tournament'>
    </selection>
<!-- Volba rekombinačního operátoru -->
    <recombine name='OneCrossPointArray'>
    </recombine>
<!-- Volba podmínky ukončení -->
    <stopcondition name='GenerationsCount'>
        <MaxGenerations value="20"/>
    </stopcondition>
<!-- Volba vhodné transformace -->
    <transform name="GSA2Vec">
    </transform>
</sirius>
```

D.1.3 Nastavení Gazebo

```
<?xml version="1.0"?>
<gz:world>
```

<!– Sekce vhodná pouze pro pozorování stavu simulace. Při samotné evoluci není potřeba a jen zdržuje –>

```
    <model:ObserverCam>
```

```
        <id>userCam0</id>
        <xyz>5 15 5</xyz>
        <rpy>0 10 -90</rpy>
        <imageSize>640 480</imageSize>
        <updateRate>1000</updateRate>
        <renderMethod>OSMESA</renderMethod>
    </model:ObserverCam>
```

<!– Zdroj světla ve scéně. –>

```
    <model:LightSource>
        <id>light1</id>
        <xyz>0.000 10.000 100.000</xyz>
        <attenuation>2 0 0</attenuation>
    </model:LightSource>
    <model:LightSource>
        <id>light1</id>
        <xyz>0.000 -10.000 100.000</xyz>
        <attenuation>1 0 0</attenuation>
    </model:LightSource>
```

<!– Podlaha. –>

```
    <model:GroundPlane>
        <id>ground1</id>
        <color>0.0 0.5 0.0</color>
        <textureFile>grid.ppm</textureFile>
    </model:GroundPlane>
```

<!– Kulička kterou budeme hledat. –>

```
    <model:SimpleSolid>
        <id>sphere1</id>
        <shape>sphere</shape>
        <xyz>10 0 0</xyz>
        <size>1</size>
        <color>1 0 0 </color>
        <fiducial>1</fiducial>
    </model:SimpleSolid>
```

<!– Náš kontrolovaný robot. –>

```
    <model:Pioneer2AT>
        <id>robot1</id>
        <xyz>0 0 0</xyz>
```

```
            <rpy>0 0 0</rpy>
            <model:SonyVID30>
                <id>camera1</id>
                <xyz>0 0 0.40</xyz>
                <rpy>0 0 0</rpy>
                <updateRate>10</updateRate>
                <renderMethod>OSMESA</renderMethod>
                <nearClip>0.01</nearClip>
            </model:SonyVID30>
        </model:Pioneer2AT>
    </gz:world>
```

D.2 Fáze 2

Fáze 2 vychází z fáze 1 a pouze ji lehce upravuje. Zde si popíšeme jen provedené úpravy, pro každý program zvlášť.

D.2.1 Nastavení Marvin

V rámci úprav pro druhou fázi došlo k přidání jednoho zařízení a dvou propojení. Ukážeme tedy pouze konfigurace přidaných částí. Nejprve přidáme zařízení.

```
    <device name='ImageWallFinder' id='5'>
        <initializers/>
        <inputs>
            <slot id='0'>
                <required value='1'/>
            </slot>
        </inputs>
        <outputs>
            <slot id='0'>
                <required value='1'/>
                <item>
                    <name value='Vector'/>
                    <data/>
                </item>
            </slot>
        </outputs>
    </device>
```

A nakonec je potřeba nové zařízení připojit ke stávajícím.

```
<!-- CameraGazebo:0 -> ImageWallFinder:0 -->
<connection>
    <source device='3' slot='0'/>
    <target device='5' slot='0'/>
</connection>
<!-- ImageWallFinder:0 -> ElmanNet:1 -->
<connection>
    <source device='5' slot='0'/>
    <target device='1' slot='1'/>
</connection>
```

D.2.2 Nastavení Sirius

V nastavení program Sirius dojde pouze k záměně ve třech elementech. A to sice *initializer, simulation* a *fitness.*

Initializer je vyměněn z náhodného na načítací z disku.

```
<initializer name='Load'>
    <PopulationSize value="5"/>
    <SliceTime value="300"/>
    <Generation value="0"/>
    <GenomeLength value="105"/>
    <SaveDirectory value=
    "/cesta/k/adresari/pro/ulozeni/populace"/>
    <LoadDirectory value=
    "/cesta/k/adresari/pro/nacteni/populace"/>
</initializer>
```

V elementu simulation dojde pouze k nahrazení jména souboru elman.xml za elman2.xml.

```
<simulation name='PVM'>
    <MarvinConfig
        file='elman2.xml'
        path='/cesta/k/konfiguraci/pro/marvina/'/>
</simulation>
```

A nakonec fitness operátor bude nový, který bude brát v potaz existenci zdi.

```
<fitness name='DistanceWithWall'>
</fitness>
```

D.2.3 Nastavení Gazebo

V nastavení simulátoru Gazeba pro tuto fázi pouze přidáme objekt pro zeď.

```
<model:SimpleSolid>
    <id>wall</id>
    <shape>box</shape>
    <xyz>4 0 0</xyz>
    <size>1 4 2</size>
    <color>0 0 1</color>
    <fiducial>1</fiducial>
    <mass>100</mass>
</model:SimpleSolid>
```

D.3 Fáze 3

Ve fázi 3 byla pouze změněna konfigurace programu Sirius. A do programu Marvin bylo přidáno nové zařízení, bez vstupů a výstupů, které pohybuje s kuličkou.

Dodatek E

Seznamy

Následující sekce obsahuje seznam obrázků, tabulek a odkazy na literaturu.

Seznam obrázků

Seznam algoritmů

Seznam tabulek

Literatura

[1] Nate Koenig Andrew Howard. Gazebo documentation. http://playerstage.sourceforge.net/doc/Gazebo-manual-0.5-html/.

[2] Ondřej Pacovský Asim Shankar. Artificial neural network library. http://annie.sf.net/.

[3] Yanco H. Kumar D. Blank, D.S. and Meeden L. Pyro. http://pyrorobotics.org/.

[4] Andrew Howard Nate Koenig Brian Gerkey, Richard Vaughan. Player/stage. http://playerstage.sf.net.

[5] François Michaud Carle Côté and team. Marie. http://marie.sourceforge.net/index.html.

[6] David J. Chalmers. The evolution of learning: An experiment in genetic connectionism. 1990.

[7] Lance D. Chambers. *Practical Handbook of Genetic Algorithms: New Frontiers*. CRC Press, Inc., Boca Raton, FL, USA, 1995.

[8] James P. Cohoon, Shailesh U. Hegde, Worthy N. Martin, and Dana S. Richards. Punctuated equilibria: A parallel genetic algorithm. In *ICGA*, pages 148–154, 1987.

[9] Martin Dlouhý. Bug algoritmy. http://robotika.cz/guide/bug-alg/cs.

[10] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[11] Carnegie Mellon Robotics Institute. Autonomous helicopter project. www.cs.cmu.edu/afs/cs/project/chopper/www/, 1998.

[12] Michael I. Jordan. Attractor dynamics and parallelism in a connectionist

sequential machine. pages 112–127, 1990.

[13] Davis L. *Handbook of Genetic Algorithms*. NY:Van Nostrand Reinhold, 1991.

[14] Mikko Lehtokangas. Constructive backpropagation for recurrent networks. *Neural Process. Lett.*, 9(3):271–278, 1999.

[15] V. Lumelsky and T. Skewis. Incorporating range sensing in the robot navigation function. *T-SMC*, 20:1058–1069, 1990.

[16] Vladimir J. Lumelsky and Alexander A. Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[17] Bernard Manderick and Piet Spiessens. Fine-grained parallel genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 428–433, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[18] Brian Paul. `http://www.mesa3d.org/`.

[19] Prof. Dr.-Ing. Friedrich Pfeiffer i.R. Prof. Dr.-Ing. habil. Heinz Ulbrich. Tum biped walking robot. `http://www.amm.mw.tu-muenchen.de/index_e.html`.

[20] Pvm. Parallel virtual machine. `http://www.csm.ornl.gov/pvm/`.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland, et al., editors, *Parallel Distributed Processing: Volume 1: Foundations*, pages 318–362. MIT Press, Cambridge, 1987.

[22] Russell Smith. Open dynamics engine. `http://www.ode.org`.

[23] Reiko Tanese. Parallel genetic algorithms for a hypercube. In *ICGA*, pages 177–183, 1987.

[24] TODO. Lobot.

[25] Jiří lažanský a kolektiv Vladimír Mařík, Olga Štěpánková. *Umělá inteligence (1)*. Academia, Praha, 1993.

[26] Jiří lažanský a kolektiv Vladimír Mařík, Olga Štěpánková. *Umělá inteligence (3)*. Academia, Praha, 2001.

[27] Jiří lažanský a kolektiv Vladimír Mařík, Olga Štěpánková. *Umělá inteligence (4)*. Academia, Praha, 2003.

[28] Liu Xing and Duc T. Pham. *Neural Networks for Identification, Prediction, and Control*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.