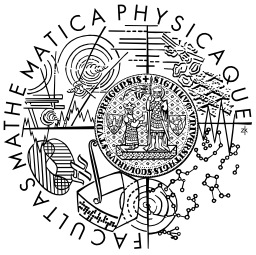


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Marek Bečvář

Evoluce robotů v simulovaném fyzikálním prostředí

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. František Mráz, CSc.

Studijní program: Informatika

Studijní obor: Informatika se specializací Umělá inteligence

Praha 2023

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Poděkování.

Název práce: Evoluce robotů v simulovaném fyzikálním prostředí

Autor: Marek Bečvář

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. František Mráz, CSc., Katedra softwaru a výuky informatiky

Abstrakt: Abstrakt.

Klíčová slova: klíčová slova

Title: Evolution of robots in a simulated physical environment

Author: Marek Bečvář

Department: Department of software and computer science education

Supervisor: RNDr. František Mráz, CSc., Department of software and computer science education

Abstract: Abstract.

Keywords: key words

Obsah

Úvod

V dnešní době stále přibývá možností, kde se snažíme aplikovat metody umělé inteligence pro řešení různorodých problémů. Na řadu z těchto problémů se nám může nabízet hned několik možných řešení. Problém ale může nastat, pokud si nejsme jistí, nebo třeba vůbec není možné přesně definovat, co vlastně by mělo být správným řešením.

Pro tyto problémy se nám často hodí využívat metod evolučních algoritmů. Jedná se o optimalizační algoritmy inspirovaných v přírodě – specificky Darwinovou evoluční teorií, které napodobováním přírodních procesů hledají dle našich požadavků ta nejlepší řešení.

Zacházení s těmito algoritmy ale nemusí být vůbec jednoduché a podobně jako u dalších optimalizačních metod a metod strojového učení je jejich běh ukryt pod množstvím parametrů, které spolu souvisí často špatně předvídatelným způsobem.

Z tohoto důvodu je cílem této práce vytvořit platformu, která bude přístupná uživatelům různých úrovní specializace, umožňující tvořit a provádět experimenty s evolučními algoritmy.

S tímto cílem volíme tvořit experimenty s roboty ve virtuálním prostředí, což se na tento problém velmi dobře hodí. Uživatel díky robotům intuitivně chápe složitost problému a každý posun v řešeném problému je interaktivně pozorovatelný v daném prostředí ještě za doby řešení.

Cílem pro projekt je, aby uživatel dostal kontrolu nad experimenty a mohl tak získat lepší přehled o práci s evolučními algoritmy a pochopil tak množství parametrů a jejich vzájemných souvislostí, se kterými se můžeme při tvorbě experimentů setkat. Projekt bude obsahovat různorodou řadu problémů, na které bude potřeba využít vícero různých přístupů, což umožní dále rozšířit pochopení problémů evolučních algoritmů. Nejtěžšími pak mohou být problémy, vyžadující využití neuronových sítí, což může být pro uživatele díky tomuto projektu jednoduchým, prvním využitím pokročilého algoritmu pro neuroevoluci (evoluční vývoj neuronových sítí) – NEAT.

Pro lehce pokročilého uživatele bude projekt dále nabízet možnosti nahlédnutí do útrob projektu, prezentující jak se s evolučními algoritmy zachází v samotném zdrojovém kódu, což mu zároveň umožní tvořit vlastní pokročilé experimenty a upravovat a rozšiřovat připravenou databázi částí evolučních algoritmů o vlastní.

Práce je rozdělena do čtyř hlavních kapitol. V kapitole 1 si představíme a vysvětlíme základní pojmy využívané v této práci jako jsou evoluční algoritmy, neuronové sítě a další. Dále v kapitole 2 blíže popíšeme specifikaci projektu a přesně jakých cílů tímto projektem chceme dosáhnout. V kapitole 3 již rámcově představíme jak je celý projekt interně poskládaný a poskytneme tak základní náhled na to, jak knihovna pracuje uvnitř. V poslední kapitole 4 na příkladech předvedeme typy experimentů, které si uživatel bude moci sám ve výchozí verzi vyzkoušet, a které bude mít možnost hned upravovat a testovat. Zároveň v této kapitole ukážeme způsob, jakým tyto experimenty jdou statisticky vyhodnocovat

a rozebereme výsledky ukázkových experimentů. V závěru pak bude následovat shrnutí výsledků práce a budou navržena možná rozšíření projektu.

1. Základní pojmy

V této kapitole vysvětlíme a rozebereme důležité pojmy, se kterými se v dalším popisu práce budeme setkávat. Znalost těchto pojmů je potřebná pro pochopení důvodů volby daných vybraných technologií a pro pochopení základního rozboru implementace řešení, kterou popíšeme v následujících kapitolách.

V této kapitole nejdříve vysvětlíme základní teorii evolučních algoritmů (oddíl 1.1) a dále si v oddíle 1.1.1 ukážeme již existující knihovny pracující nebo umožňující pracovat s genetickými algoritmy. Poté se v oddílu 1.2 podíváme na základ teorie umělých neuronových sítí a v oddílu 1.3 popíšeme neuroevoluci, kategorii pokročilých evolučních algoritmů sloužící přímo k vývoji umělých neuronových sítí (algoritmy NEAT v oddílu 1.3.1 a HyperNEAT v oddílu 1.3.2). Dále popíšeme simulátory prostředí (oddíl 1.4) a fyzikální simulátory (oddíl 1.4.2), které využijeme pro simulaci při vyvíjení našich robotů.

1.1 Evoluční algoritmy

Evoluční algoritmus je označení pro stochastické vyhledávací algoritmy, které svými procesy napodobují principy Darwinovy evoluční teorie o přirozeném výběru a přežívání nejlepších jedinců. Pomocí těchto algoritmů se často snažíme optimalizovat nějaké procesy, nebo vlastnosti systémů, a tedy evoluční algoritmy můžeme označovat i jako optimalizační algoritmy. Mohou být úspěšné v řešení komplexních problémů (např. aproximace NP-úplných problémů) a problémů se složitě definovanou hodnotící funkcí. Stochastická vlastnost těchto algoritmů způsobuje, že algoritmy nezaručují nalezení optimálního řešení, ale jsou schopné se tomuto řešení přiblížit nějakým kvalitním suboptimálním řešením.

Mezi známé varianty evoluční algoritmů patří:

- *Genetické algoritmy (GA)* – kódující řešení pomocí vektorů binárních hodnot,
- *Evoluční strategie (ES)* – pro reprezentaci řešení, na rozdíl od GA, využívají ES reálné hodnoty a vektory reálných hodnot,
- *Genetické programování (GP)* – reprezentuje řešení skládáním složitějších operací (např. pravidel bezkontextových gramatik, stromů s funkcemi a konstantami).

Jednotlivá řešení v evolučních algoritmech jsou nazývána jako **jedinci** s vlastními genetickými informacemi (**genotyp**). Kvalitu genotypu (označujeme jako **fitness**) je možné pro daný problém otestovat, vyhodnocením předem definované *hodnotící funkce*. Fitness je potom číselná hodnota, kterou se v procesem evolučního algoritmu snažíme maximalizovat/minimalizovat (dle definice hodnotící funkce). Množinu takových jedinců v evolučních algoritmech nazýváme **populace**. Celý vývoj evolučního algoritmu potom probíhá v cyklech, kdy v každé iteraci je otestována kvalita genotypu celé populace. Tyto jednotlivé iterace nazýváme **generace**.

Genetické operátory Vedle inicializace populace, která je obvykle provedena náhodně, jsou důležitými částmi evolučních algoritmů tzv. **genetické operátory**. Jedná se o procesy silně inspirované přírodními jevy pozorovanými při evoluci živých organismů. Tak jako se vyvíjí organismy v přírodě, za účelem udržení těch nejvhodnějších vlastností pro dané prostředí, vyvíjí se i jedinci v evolučních algoritmech, aby se přibližovali optimálnímu řešení zadaného problému (optimalizovali hodnotící funkci). Proces vývoje v evolučních algoritmech je pak zajištěn třemi genetickými operátory – **selekce**, **křížení** a **mutace**. Všechny budou blíže popsány v následujících odstavcích. Určité varianty evolučních algoritmů nemusí využívat všechny genetické operátory.

Selekce Selekce je proces, který vybírá jedince z populace, kteří buď přímo postoupí do další generace, nebo budou vybráni jako rodiče (zdroj genetických informací) pro další generaci. Selekce má upřednostňovat jedince s vyšší kvalitou genotypu, což by mělo v průběhu vývoje produkovat kvalitnější jedince, a tedy směřovat vývoj k optimálnímu řešení. Při vývoji může být aplikována ve dvou bodech. Pomocí selekce můžeme vybírat rodičovské jedince, kteří budou použiti pro tvorbu potomků, ze kterých se následně stane nová populace pro další generaci (*mating selection=selekce pro křížení*). Zároveň můžeme selekci využít pro sestavení nové generace výběrem z nejlepších jedinců původní populace a nově vytvořených potomků (*environmentální selekce*).

Nejpoužívanějšími způsoby implementace operátoru selekce jsou *turnajová selekce* a *ruletová selekce*.

Při *turnajové selekci* náhodně vybereme určitý počet jedinců z populace a dle jejich ohodnocení (fitness) je rozřadíme (uspořádáme turnaj mezi jedinci). V tuto chvíli nejčastěji selekce vybere vítěze turnaje (nejlepšího jedince z turnaje). Existují i pravděpodobnostní varianty *turnajové selekce*, kdy pořadí v turnaji určuje pravděpodobnost zvolení daného jedince (vítěz turnaje má nejvyšší pravděpodobnost zvolení, další v pořadí mají klesající šanci na zvolení). Výhodou *turnajové selekce* je, že nedává žádná omezení na fitness jedinců. Stačí nám, když jdou jednotlivé hodnoty fitness porovnat a rozlišit tak, která je lepší a která horší.

Ruletová selekce je abstrakce ruletového kola, kde každý jedinec má šanci na zvolení (šance, že jeho hodnota padne na ruletovém kole) přímo úměrnou jeho fitness. Tato selekce se potom provádí opakovaně, dokud nevybereme požadovaný počet jedinců, nad celou populací a ne pouze nad náhodně vybranou podmnožinou jedinců z populace (jak tomu bylo u *turnajové selekce*). Pokud se v populaci nachází jedinec s dominantní hodnotou fitness, která výrazně překonává ostatní jedince v populaci, může se stávat, že bude docházet k opakovanému výběru jednoho a toho samého jedince, což může vést k velmi rychlému zúžení prohledávaného prostoru a tedy nalezení neoptimálního řešení (vývoj může uvíznout v tzv. *lokálním optimu*).

Křížení Křížení je proces, při kterém nějakým stylem kombinujeme genetické informace dvou (nebo více) jedinců. Tímto procesem vznikají potomci původních jedinců, kteří se mohou stát novou populací v následující generaci. Výběr jedinců pro křížení zajišťuje **selekce**. Křížení umožňuje důkladnější prohledávání prostoru možných řešení mezi již prohledanými možnostmi. Existuje mnoho variant křížení. Často využívanými variantami křížení jsou např. *jednobodové křížení*, *vícebodové*

křížení nebo *uniformní křížení*.

Při *jednobodovém křížení* (pro dva rodičovské jedince) vybereme náhodně jedno místo (bod dělení), ve kterém oba genotypy rozdělíme. Genetickou informaci potomků následně vytvoříme tak, že složíme zpět rozdělené části genotypů různých rodičů (např. první potomek vznikne složením první části genotypu prvního rodiče a druhé části genotypu druhého rodiče a druhý potomek naopak). *Vícebodové křížení* probíhá stejně, pouze na začátku volíme vícero bodů, ve kterých genotypy rozdělíme a tak máme více možností, jak genotyp složit zpět.

Uniformní křížení při tvorbě potomků najednou prochází celý genotyp všech rodičů a u každé hodnoty uniformně náhodně vybere, z jakého rodiče se daný kus genetické informace zkopíruje do potomka. U tohoto typu křížení lze jednoduše použít i víc než dva rodiče pro tvorbu potomků.

Jedná se o příklady metod křížení, často používané pro **genetické algoritmy**. Algoritmy pracující s reálnými hodnotami nebo jinak složitými strukturami mohou využívat jiné metody, pracující specificky s danými hodnotami genotypu (příkladem může být *křížení průměrováním* reálných hodnot dvou nebo více rodičů).

Mutace Při mutaci náhodně měníme malé části genotypu jedinců. Obvykle probíhá po vygenerování nových potomků pomocí **křížení**. Pomocí mutace evoluční algoritmy rozšiřují prostor prohledávaných řešení. Správně zvolená hodnota mutace zabraňuje uvíznutí evolučního algoritmu v lokálních optimech. Příliš silná mutace ale může způsobovat velké změny v genotypech a tím zabránit vývoji jakýchkoli výhodných vlastností jedinců, které by vedli k optimalizaci řešení. Hodnotou mutace označujeme pravděpodobnost, že část genotypu bude pozměněna.

Styl, jakým se genotyp mutuje, je závislý na typu evolučního algoritmu. Pro genetické algoritmy s genotypem tvořeným vektorem binárních hodnot se může jednat o např. náhodný *bit flip* (tedy změnu dané hodnoty z 0 na 1 nebo naopak). U jiných typů evolučních algoritmů je možné používat např. přiřazení nové náhodně vygenerované hodnoty z rozsahu povolených hodnot pro danou část genotypu, nebo posun o malou náhodně vygenerovanou hodnotu.

Elitismus Jelikož je evoluční vývoj proces postavený na náhodě, mohlo by se stát, že náhodnou mutací přijdeme o nějaké potřebné vlastnosti z nejlepších jedinců a těch už nikdy nebudeme moci dosáhnout. Elitismus umožní tyto aktuálně nejlepší vlastnosti mezi generacemi zachovat. Jedná se o speciální proces, řadící se většinou k **selekci**, který umožňuje zachovat určité množství jedinců mezi generacemi beze změny. Obvykle jeden nebo malé množství nejlepších jedinců je po každé generaci přesunuto beze změny do další (tedy neprojdou žádnou mutací a jejich genotyp je stejný mezi generacemi).

Následující ukázka pseudokódu předvede jednoduchý příklad běhu evolučního algoritmu.

```
POP = náhodná inicializace populace
otestuj populaci a vypočti fitness
dokud problém není vyřešen:
    PARENTS = pomocí selekce vyber množinu rodičů
    OFFSPRING = křížením (z PARENTS) vytvoř množinu potomků
    MUT_OFFSPRING = aplikuj mutaci na potomky z OFFSPRING

    POP = z potomků v MUT_OFFSPRING vytvoř populaci další generace
    otestuj novou populaci POP a vypočti fitness
```

V ukázce můžeme vidět jednotlivé kroky jednoduchého evolučního algoritmu. Nejdříve inicializujeme populaci a zjistíme fitness jedinců. Následně dokud se nesplní podmínka pro ukončení evolučního vývoje (např. dosažení specifické fitness hodnoty), bude cyklicky v generacích probíhat vývoj populace, dokud se neobjeví takový jedinec, který zadaný problém vyřeší.

Možnosti paralelizace Všechny části evolučních algoritmů jsou ve své podstatě velmi jednoduché a tedy i rychlé na výpočet. Jediný bod, který může vývoj zpomalit je samotné testování jedinců. Například, když jedinec reprezentuje robota, tak jeho hodnocení vyžaduje simulaci robota po určitou dobu. Jelikož každý robot musí být simulován zvlášť, je tento krok v našem evolučním algoritmu zdaleka ten nejnáročnější co se týče doby výpočtu. Naštěstí ve většině příkladů použití evolučních algoritmů, je tento krok možné poměrně jednoduše paralelizovat a i v našem případě tomu tak je. Jednotliví jedinci jsou v absolutní většině evolučních algoritmů naprosto nezávislí na ostatních. Pokud to simulační prostředí dovolí, je možné nechat každého robota simulovat nezávisle na ostatních ve vlastním vlákně (procesu) a tak využít moderní vícevláknové procesory. Tato změna je poměrně jednoduchá na implementaci a absolutní většina evolučních algoritmů může tohoto využít pro mnohonásobné urychlení celého procesu evolučního vývoje.

1.1.1 Existující implementace

Pro vývoj řízení robotů budeme využívat evoluční algoritmy. Naše knihovna tedy bude implementovat několik alespoň základních genetických operátorů, používaných při vývoji jedinců a co nejjednodušeji umožňovat jejich konfiguraci před spouštěním jednotlivých experimentů. Naším cílem je co možná nejvíce zpřístupnit knihovnu, která má být výsledkem této práce, aby uživatel se základní znalostí genetických algoritmů a programovacího jazyka byl schopný pochopit běh algoritmu a v případě potřeby mohl jednoduše provádět zásahy do jeho běhu. Není tedy naším cílem najít tu nejefektivnější knihovnu, nýbrž tu, která přinese výhody jako přehlednost a snadnou úpravu algoritmů, bez větších obtíží s implementací a pochopením knihovny.

Dále představíme několik knihoven implementujících nebo usnadňujících implementaci genetických algoritmů nebo jejich částí. Celkem se podíváme na dvě knihovny – DEAP (sekce 1.1.1.1) a Inspyred (sekce 1.1.1.2). Existují další podobné knihovny (Pyevolve, PyGAD), které ale oproti DEAP a Inspyred nepřináší mnoho dalších užitečných možností.

1.1.1.1 DEAP

DEAP (*Distributed Evolutionary Algorithms in Python*) (DEAP) je open-source Python knihovna pro rychlou tvorbu a prototypování evolučních algoritmů. Snaží se tvorbu evolučních algoritmů zjednodušit pomocí přímočarého postupu, podobného pseudokódu, který je se základní znalostí knihovny poměrně jednoduchý na porozumění.

Knihovna je tvořena ze dvou hlavních struktur `creator`, který slouží k vytváření genetických informací jedinců z libovolných datových struktur a `toolbox`, který je seznamem nástrojů (genetických operátorů), které mohou být použité při sestavování evolučního algoritmu. Dalšími menšími strukturami jsou `algorithms` obsahující 4 základní typy algoritmů a `tools` implementující další základní operátory (části operátorů), které je posléze možné přidávat do `toolbox`. Pomocí těchto základních stavebních bloků mohou uživatelé poměrně jednoduše začít tvořit skoro libovolné evoluční algoritmy (Fortin a kol., 2012).

Následuje ukázka kódu tvorby základních částí evolučního algoritmu pro *One Max* problém, popsaná v oficiální dokumentaci knihovny DEAP. V *One Max* problému máme populaci jedinců, kteří reprezentují vektor binárních čísel předem zvolené délky. Cílem je potom vyvinout takového jedince, který má na všech pozicích vektoru nastavené jedničky.

Nejprve v kódu importujeme potřebné části knihovny. Dále využijeme strukturu `creator` pro tvorbu specifických tříd, které budeme potřebovat pro popis jedinců v našem evolučním algoritmu.

`Creator` `Creator` je třída sloužící jako *factory* pro uživatelem definované třídy. Tedy s její pomocí můžeme vytvářet nové třídy za běhu programu. To se hodí, protože různé problémy mohou vyžadovat velmi rozdílné typy jedinců. Tvorba třídy probíhá pomocí metoda `create`, která jako argumenty přijímá jméno vytvářené třídy, dále třídu, od které nová třída bude dědit a poté může následovat řada argumentů, které mohou být využity jako další argumenty naší nové třídy.

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
```

První řádek popisuje tvorbu třídy `FitnessMax`, která dědí od třídy `Fitness` knihovny DEAP a zároveň obsahuje argument `weights`, který specifikuje, že fitness bude maximalizovat jediný cíl (pomocí DEAP můžeme specifikovat hned několik cílů najednou, ve kterých chceme, aby se jedinci zlepšovali, přičemž jednotlivým cílům můžeme přiřadit různé váhy).

Na druhém řádku vytváříme třídu jedince `Individual`, která dědí od třídy `list` a bude obsahovat parametr `fitness`, do nějž přiřadíme vytvořenou třídu `FitnessMax`.

`Toolbox` Dále využijeme vlastní třídy pro tvorbu specifických typů, reprezentujících jedince a celou populaci. `Toolbox` se stane úložištěm pro všechny objekty, které budeme při tvorbě evolučního algoritmu používat. Do tohoto úložiště můžeme objekty přidávat metodou `register` a můžeme je odebrat metodou `unregister`.

```
# založení úložiště
toolbox = base.Toolbox()

# generátor atributů pro jedince
toolbox.register("attr_bool", random.randint, 0, 1)

# inicializace struktur jedince a populace
toolbox.register("individual",
                 tools.initRepeat,
                 creator.Individual,
                 toolbox.attr_bool,
                 100)
toolbox.register("population",
                 tools.initRepeat,
                 list,
                 toolbox.individual)
```

Nejdříve jsme vytvořili `toolbox` jako úložiště pro naše funkce. Dále jsme přidali generátor `toolbox.attr_bool()`, tvořený z metody `random.randint()`, který když zavoláme, náhodně vygeneruje celé číslo mezi 0 a 1.

Dále jsme přidali dvě inicializační metody `toolbox.individual()` pro inicializaci jedinců a `toolbox.population()` pro inicializaci celé populace.

Pro vytvoření jedince používáme metodu `tools.initRepeat()`, jejíž první argument přijímá kontejner (v našem případě třídu jedince, kterou jsme definovali dříve jako potomka třídy `list`). Ten bude při inicializaci naplněn metodou`toolbox.attr_bool()` zavolanou 100 krát, jak specifikují následující dva argumenty. Při inicializaci celé populace budeme postupovat stejně, jen jsme ještě v tento moment nespecifikovali, kolik jedinců bude do populace vytvořeno.

Hodnotící funkce Hodnotící funkce je pro tento problém jednoduchá. Potřebujeme pouze spočítat, kolik jedniček obsahuje daný jedinec (vektor binárních čísel).

```
def evalOneMax(individual):
    return sum(individual)
```

Genetické operátory Knihovna umožňuje dva přístupy, jak můžeme využívat genetické operátory. Buď je můžeme volat přímo z `tools`, nebo je nejdříve registrujeme do úložiště `toolbox` a z něho je budeme používat. Registrace je považována za vhodnější variantu, protože to do budoucna zjednodušuje změny používaných operátorů.

```
toolbox.register("evaluate", evalOneMax)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutFlipBit, indpb=0.05)
toolbox.register("select", tools.selTournament, tournsize=3)
```

Hodnotící funkci realizuje metoda `toolbox.evaluate()`, fungující jako alias na dříve vytvořenou hodnotící funkci. Metoda `toolbox.mate()` je v tomto příkladu alias za `tools.cxTwoPoint()`, což je v knihovně implementovaná metoda provádějící dvoubodové křížení. Podobně tvoříme i funkce pro mutaci jedinců (v tomto případě binární mutaci – `tools.mutFlipBit`), kde argument `indpb` určuje pravděpodobnost mutace jednotlivých parametrů jedince a nakonec funkci pro selekci (turnajová selekce s turnajem mezi třemi jedinci).

Evoluce Když jsou všechny části připravené, vlastní evoluční algoritmus se sestaví kombinací jednotlivých definovaných částí, aplikováním registrovaných funkcí na populaci nebo jedince dle potřeby.

Na inicializované populaci se provádí evoluce, dokud nějaký z jedinců nesplní zadaný úkol, nebo dokud evoluce nedosáhne určitého počtu generací.

Pro stručnější popis zbytek kódu vynecháme, protože jsme si již předvedli všechny části spojené s definováním evolučního algoritmu, které jsou specifické pro práci s knihovnou DEAP. Úplnou ukázku lze najít v oficiální dokumentaci knihovny (DEAP).

Podle článku (Fortin a kol., 2012), který porovnává několik Python modulů pro usnadnění práce s evolučními algoritmy, je DEAP nejefektivnější, tedy tvoří nejkratší kód, podle počtu řádků potřebných pro implementaci algoritmu řešícího *One Max* problém z ukázky.

1.1.1.2 Inspyred

Inspyred (Garrett, 2012) poskytuje většinu z nejpoužívanějších evolučních algoritmů a dalších přírodou inspirovaných algoritmů (simulace reálných biologických systémů – např. optimalizace mravenčí kolonií) v jazyce Python.

Knihovna přichází s funkční implementací řady základních evolučních algoritmů ve formě komponentů (Python funkcí), které si uživatel může sám upravovat, rozšiřovat, nebo je úplně nahradit vlastními funkcemi. Při tvorbě algoritmu pak uživatel skládá dohromady několik komponent, které ovlivňují, jak celý vývoj bude probíhat. Těmito komponenty jsou:

a) komponenty specifické danému problému:

- `generator` – určuje jak se generují řešení (jedinci),
- `evaluator` – definuje jak se vypočítává fitness jedinců,

b) komponenty specifické danému danému evolučnímu algoritmu:

- `observer` – definuje jak uživatel sleduje evoluci,
- `terminator` – rozhoduje, kdy by měla evoluce skončit,
- `selector` – rozhoduje, kteří jedinci by se měli stát rodiči další generace,
- `variator` – definuje jak jsou potomci vytvořeni z rodičovských jedinců,
- `replacer` – volí, kteří jedinci mají přežít do další generace,

- `migrator` – určuje jak se přenáší jedinci mezi různými populacemi/generacemi,
- `archiver` – definuje jak jsou jedinci ukládání mimo stávající populaci.

Libovolná z těchto komponent pak může být nahrazena odpovídající vlastní implementací (Tonda, 2020).

Následuje jednoduchý příklad z dokumentace knihovny Inspyred, který nám rychle představí možnou práci s knihovnou. Budeme řešit problém srovnatelný s problémem *One Max* představeným u příkladu knihovny DEAP. Nyní je cílem, aby hodnota vektoru interpretována jako binární zápis celého čísla byla co nejvyšší (opět chceme, aby výsledný jedinec měl na všech místech vektoru nastavené jedničky).

Na začátku se importuje knihovna Inspyred a modul `random`.

`Generator` Pro řešení problému vytvoříme vlastní generátor jedinců populace. Všechny generátory knihovny Inspyred mají vždy stejné dva argumenty:

- `random` – objekt pro generování náhodných čísel,
- `args` – slovník dalších argumentů, které můžeme libovolně přidat.

```
def generate_binary(random, args):
    bits = args.get('num_bits', 8)
    return [random.choice([0, 1]) for i in range(bits)]
```

Zde vytváříme vlastní funkci `generate_binary`, která bude sloužit jako generátor jedinců. V této funkci nejdříve do proměnné `bits` načteme hodnotu z argumentu `num_bits` (s jeho definicí se setkáme později) a poté vytvoříme jedince jako seznam, který naplníme náhodně zvolenými binárními hodnotami.

`Evaluator` Podobně jako generátor, tak i pro vyhodnocení fitness jedinců vytvoříme vlastní funkci. Funkce tohoto typu opět vyžadují dva argumenty:

- `candidate` – jedinec, kterého ve funkci vyhodnocujeme, a
- `args` – slovník dalších argumentů, které můžeme libovolně přidat.

V knihovně se často setkáme s dekorátory metod. Přesněji metody, které tvoříme pro `evaluator` vyžadují dekorátor `@evaluator`. Dekorátory se používají, protože vytváříme metody, které budou použité v rámci jiných interních metod (např. naše vlastní vyhodnocovací funkce pracuje pouze s jedním jedincem, ale funkce se bude při vyhodnocení fitness interně používat pro celou populaci).

```
@inspyred.ec.evaluators.evaluator
def evaluate_binary(candidate, args):
    return int("".join([str(c) for c in candidate]), 2)
```

Vyhodnocení jedince tedy vezme všechny prvky jeho vektoru, přečte je a vyhodnotí výstup jako binární zápis nějakého celého čísla. Toto číslo je potom výstupní ohodnocení pro daného jedince.

Genetický algoritmu Vytvořili jsme všechny potřebné části, specifické pro tento problém a tedy je můžeme použít pro vytvoření výsledného genetického algoritmu. Knihovna Inspyred nabízí několik různých typů evolučních algoritmů (genetické algoritmy, evoluční strategie, simulované žíhání a další). Pro tento problém vybereme základní formu genetického algoritmu, která je nám v této knihovně dostupná.

```
rand = random.Random()
ga = inspyred.ec.GA(rand)
ga.observer = inspyred.ec.observers.stats_observer
ga.terminator = inspyred.ec.terminators.evaluation_termination
```

Zde nejprve vytváříme objekt pro generování náhodných čísel, který bude využíván v algoritmu. Na druhém řádku pak vytváříme objekt samotného genetického algoritmu. Jak jsme zmínili výše, všechny algoritmy mají určité komponenty, které uživatel může měnit za jiné, nebo je celé sám upravovat. Pro ukázku zde měníme komponenty `observer` a `terminator`. Pro `observer` volíme připravený `stats_observer`, což je metoda, která v průběhu algoritmu bude vypisovat statistiky z běhu evoluce. Výstup tohoto `observeru` bude mít následující podobu:

```
Generation Evaluation   Worst    Best   Median   Average     Std Dev
---------- ---------- ------- ------- -------- --------- -----------
         0        100       6    1016    564.5    536.02  309.833954

Generation Evaluation   Worst    Best   Median   Average     Std Dev
---------- ---------- ------- ------- -------- --------- -----------
         1        200      29    1021    722.5    675.22  247.645576
```

kde `Generation` je číslo aktuální generace, `Evaluation` udává počet vyhodnocení jedinců a hodnoty `Worst`, `Best`, `Median`, `Average` a `Std Dev` udávají popořadě fitness hodnotu nejhoršího z jedinců, nejlepšího z jedinců, hodnotu mediánu a průměru fitness hodnot a standardní odchylku fitness hodnot.

Zároveň měníme i `terminator` za funkci `evaluation_termination`, která jednoduše zahlásí, že evoluce má skončit, pokud evoluce dosáhla maximálního počtu vyhodnocení, což je hodnota určená argumentem `max_evaluations`, který je zvolen při volání metody `evolve` níže.

Evoluce Samotné spuštění a vyhodnocení evoluce je pak velmi jednoduché.

```
final_pop = ga.evolve(evaluator=evaluate_binary,
                      generator=generate_binary,
                      max_evaluations=1000,
                      num_elites=1,
                      pop_size=100,
                      num_bits=10)

final_pop.sort(reverse=True)
```

Genetický algoritmus se lehce spustí pomocí metody `evolve`, které předáme požadované parametry jako náš zvolený `evaluator` a `generator`, dále pro vybraný `terminator` volíme maximální počet vyhodnocení, které chceme při vývoji dovolit. Dále je možné pro tyto algoritmy specifikovat jevy jako třeba elitismus. Nakonec `pop_size` určuje velikost populace, se kterou bude evoluce pracovat a vkládáme zde dříve zmíněný argument `num_bits`, určující velikost vektoru jedince.

Další informace o příkladu a jednotlivých funkcích lze nalézt v oficiální dokumentaci Inspyred (Garrett, 2012).

1.1.1.3 Porovnání

Při srovnání těchto knihoven jsme převážně sledovali jak intuitivní práce s nimi je. Cílem této práce je vytvořit co možná nejvíce otevřenou platformu, se kterou bude moci uživatel provádět experimenty při vývoji řízení robotů. Uživatel se základní znalostí evolučních algoritmů by tak měl být schopný jednoduše pochopit všechny části knihovny a pokud by měl potřebu, sám si doplnit nějaké specifické části.

Z vlastního pohledu, ačkoli knihovna DEAP umožňuje tvorbu asi libovolného evolučního algoritmu velmi kompaktním způsobem, potřeba pochopit a seznámit se s procesem tvorby vlastních tříd a objektů, na kterém je DEAP postavený, je poměrně velkou překážkou pro možného nového uživatele naší knihovny, který by si mohl chtít upravit proces evolučních algoritmů dle svých požadavků. Ačkoli méně kompaktní, řešení knihovny Inspyred 1.1.1.2, které dělí algoritmy do základních stavebních bloků, kde každá část může být se základní znalostí Pythonu pozměněna, se mi pro náš účel zamlouvá více. Inspyred ale zároveň implementuje řadu dalších algoritmů, které by v našem případě vůbec nemusely být využitelné a pouze by mohly zvyšovat minimální množství znalostí potřebných k práci s naší knihovnou.

Na základě vyzkoušených a předvedených knihoven, které se dle různých zdrojů (Fortin a kol., 2012) zdály jako nejvhodnější pro naše využití, jsme se rozhodli inspirovat se knihovnou Inspyred (stylem, jakým knihovna dělí algoritmus na základní stavební bloky) a vytvořit vlastní implementaci většiny základních stavebních bloků, které bude možné použít při tvorbě vlastních evolučních algoritmů v naší knihovně. Tyto bloky budou co možná nejvíce obecné a snadno konfigurovatelné funkce s předepsaným výstupním typem, aby uživatel mohl snadněji porozumět implementaci každého z bloků a zároveň aby měl možnost vytvářet vlastní bloky (Python funkce) a ty jednoduše zapojit do evolučního algoritmu. Právě tak jak tomu je v knihovně Inspyred.

Celý projekt je směřován zároveň jako možný studijní materiál, a tak navíc věřím, že pro studenty bude možnost vidět v kódu funkční implementaci jednotlivých částí algoritmů tak, jak běží na pozadí experimentů přínosnější a ulehčí to jejich další experimenty, třeba i s implementací vlastních bloků evolučních algoritmů.

1.2 Neuronové sítě

Umělé neuronové sítě jsou matematickou abstrakcí biologických neuronů a jejich chování v nervové soustavě. Umělé neurony jsou zjednodušeně vrcholy v grafu, propojeného hranami s váhami (podobně jako reálné neurony jsou propojeny synapsemi). Každý neuron počítá svůj potenciál jako vážený součet příchozích vzruchů přicházejících korespondujícími vstupními hranami. Nakonec je na potenciál neuronu aplikována aktivační funkce dále transformující potenciál na výstupní hodnotu neuronu.

Neuron Nejmenší jednotkou neuronových sítí je jeden neuron. Pro neuron s n vstupy, jejichž hodnoty označíme x_1,...,x_n a váhy korespondujících hran označíme w_1,...,w_n, potom výstupní hodnotu tohoto neuronu, označenou O, můžeme spočítat jako,

$$O = f(\xi) \tag{1.1}$$

kde $\xi = \sum_{i=1}^{n} x_i w_i + b$ je potenciál neuronu, f je aktivační funkce na neuronu a b je práh (*bias*) daného neuronu, což je hodnota, která se přičítá k váženému součtu, sloužící jako posun výsledku.

Aktivační funkce Aktivační funkce slouží k transformaci potenciálu neuronu na výstupní hodnotu. Nejčastěji se jedná o nelineární transformaci pomocí nelineární funkce. Dnes nejpoužívanějšími aktivačními funkcemi jsou:

- *logistická sigmoida* (obrázek 1.1),

$$f(\xi) = \frac{1}{1 + e^{-\xi}} \tag{1.2}$$

- *hyperbolický tangens* (obrázek 1.2),

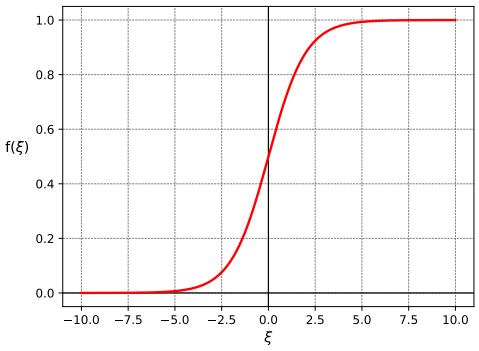
$$f(\xi) = tanh(\xi) = \frac{2}{1 + e^{-2\xi}} - 1 \tag{1.3}$$

- *ReLu (Rectified Linear Unit)* (obrázek 1.3).

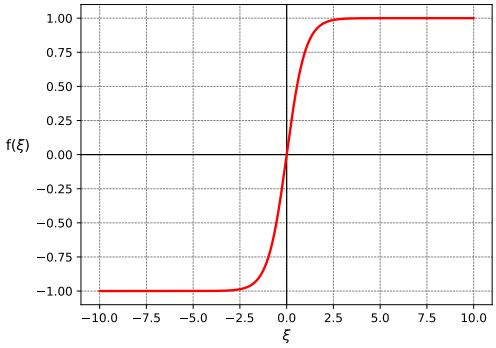
$$f(\xi) = \max(0, \xi) \tag{1.4}$$

Neuronová síť Architektura neuronové sítě je tvořena ze třech základních typů neuronů – vstupní (ze kterých pouze vycházejí spojení), výstupní (do kterých pouze přicházení spojení) a skryté (mají vstupní i výstupní spoje). Neuronová síť pak jde popsat jako orientovaný graf. Pokud se jedná o graf bez orientovaných cyklů můžeme síť označit za *dopřednou neuronovou síť*. Pokud obsahuje nějaké orientované cykly, označujeme ji jako *rekurentní neuronovou síť*. Dále se budeme zaměřovat na *dopředné vrstevnaté neuronové sítě*.

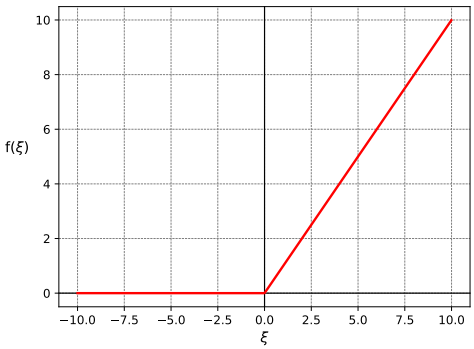
Dopředné neuronové sítě jsou často rozdělené do vrstev a nazývají se vrstevnaté neuronové sítě. V těchto sítích vstup přechází po orientovaných hranách vrstvami neuronů, kde každý neuron z předchozí vrstvy je spojen hranou s každým vrcholem následující vrstvě. První vrstva se nazývá *vstupní vrstva*. Je tvořena ze *vstupních neuronů* a slouží pro vstup parametrů (vstupů) do sítě. Vstupní



Obrázek 1.1: Funkce *sigmoida*



Obrázek 1.2: Funkce *tanh*



Obrázek 1.3: Funkce *ReLU*

neuron dostane hodnotu konkrétního vstupu a tuto hodnotu posílá beze změny dále (vstupní neurony nepoužívají aktivační funkci). Poslední vrstva se nazývá *výstupní vrstva*. Je tvořena z *výstupních neuronů* a výstup těchto neuronů je zároveň výstupem celé sítě. Každá neuronová síť musí nutně mít alespoň jeden vstupní a alespoň jeden výstupní neuron. Libovolná další vrstva mezi *vstupní* a *výstupní vrstvou* se nazývá *skrytá vrstva*. Těchto vrstev může být v síti teoreticky libovolné množství.

Následující rovnice předvedou jeden dopředný průchod neuronovou sítí transformující vstupní vektor hodnot na výstupní vektor. V tomto příkladu si představíme průchod dat jednoduchou sítí, skládající se ze vstupní vrstvy (výstupy n vstupních neuronů značíme $x_1,...,x_n$), jedné skryté vrstvy neuronů (výstupy k neuronů v této vrstvě označíme $h_1,...,h_k$, s odpovídajícím vektorem prahů $b_1,...,b_k$ a váhami $w_{j,i}$ pro $i = 1,...,k$ a $j = 1,...,n$) a výstupní vrstvy (výstupy m neuronů výstupní vrstvy označíme jako $y_1,...,y_m$, matici vah $w'_{j,i}$ pro $j = 1,...,k$ a $i = 1,...,m$ a vektor prahů $b'_1,...,b'_m$).

Výstup skrytých neuronů se spočítá jako,

$$h_i = f(\sum_{j=1}^{n} x_j w_{j,i} + b_i), \quad i = 1,...,k \tag{1.5}$$

kde $w_{j,i}$ je váha hrany ze vstupního neuronu j do skrytého neuronu i a f je aktivační funkce neuronů skryté vrstvy. Výstupy výstupních neuronů se spočítají

jako,

$$y_i = a(\sum_{j=1}^{k} h_j w'_{j,i} + b'_i), \quad i = 1,...,m \tag{1.6}$$

kde $w'_{j,i}$ je váha hrany z j-tého skrytého neuronu do i-tého výstupního neuronu a a je aktivační funkce neuronů výstupní vrstvy.

Trénování neuronových sítí Trénování (neboli učení) neuronových sítí je často velmi složitý proces, při kterém se snažíme nastavit hodnoty vah jednotlivých spojů mezi neurony tak, abychom pro konkrétní vstup na *vstupní vrstvě*, dostali požadovaný výstup na *výstupní vrstvě*.

Toto je nejčastější požadavek pro neuronové sítě při tzv. *učení s učitelem* (*supervised learning*). Při tomto učení síť dostává dvojice vektorů (x, t), kde x je vektor vstupních hodnot a t je vektor požadovaných výstupů. Následně se x nastaví jako potenciál vstupních neuronů a síť spočítá výstupy na výstupních neuronech. Vektor výstupních neuronů (obvykle značený y) se porovná s požadovaným výstupem t. Na základě rozdílů (chyby) těchto hodnot se provede úprava vah jednotlivých spojů tak, aby se při opakovaném výpočtu sítě chyba zmenšila. Tohoto lze dosáhnout často používaným algoritmem zpětného šíření chyby (*backpropagation*), který počítá derivace chybové funkce podle vah tak, aby vždy prováděl úpravy vah spojů ve směru klesající chybové funkce.

Tento přístup ale přestává fungovat, pokud nedokážeme vytvořit vstupní trénovací dvojice *(vstup, požadovaný výstup)*, a tedy bychom nevěděli jakým směrem váhy spojů upravovat. Toto je překážka v mnoha praktických problémech, na které by se neuronové sítě hodilo použít. Možným řešením je nevyužívat učící metody založené na propagaci výsledné chyby sítě, ale použít nějakou hodnotící funkci, která ohodnotí kvalitu konfigurace dané sítě (např. po simulačním běhu). Toto nás vede k možnosti využití evolučních algoritmů pro trénovaní neuronových sítí evolučním algoritmem.

1.3 Neuroevoluce

Neuroevoluce Lehman a Miikkulainen (2013) je technika pro evoluční vývoj umělých neuronových sítí pomocí principů evolučních algoritmů (popsaných sekci 1.1). Vývoj pomocí neuroevoluce je obecnější než vývoj pomocí klasických trénovacích metod, jelikož vývoj, na rozdíl od trénovacích metod založených na principu *učení s učitelem*, může probíhat i na sítích proměnlivé architektury a nepotřebuje znát korektní výstupy pro daný vstup sítě. Pro trénovaní stačí, když jsme schopni nějakým způsobem ohodnotit kvalitu řešení, k jakému se s využitím dané konfigurace (architektura a váhy spojení) sítě dostaneme. Díky tomuto se neuroevoluce hodí pro vývoj neuronových sítí v případech, kdy nejsme schopni přesně určit správné výstupy sítě pro dané vstupy a pouze můžeme pozorovat kvalitu chování daného vyvíjeného systému.

Vývoj neuronových sítí Vývoj pomocí neuroevoluce probíhá stejně jako u jiných evolučních algoritmů. Neuronová síť je zakódovaná do genotypu jedinců, množina těchto jedinců potom tvoří populaci, procházející vývojem skrz opakované generace. V každé generaci je genotyp každého jedince dekódován a z těchto

informací je vytvořena neuronová síť podle dekódované architektury a vah spojení. Pro každého jedince je jeho dekódovaná síť následně otestována v testovacím prostředí, kde se ohodnotí kvalita jedince (fitness).

Jednoduchým typem kódování může být uložení hodnot vah všech hran v neuronové síti do jediného vektoru. Takový vektor se použije jako genotyp jedince. To umožňuje optimalizaci vah sítě s fixní architekturou. Tento typ kódování může být ale výpočetně velmi náročný, kvůli rychle narůstající délce genotypu s velikostí architektury (počet vrstev a počet neuronů v síti) neuronové sítě.

Pokročilé metody Výše uvedené problémy můžeme řešit hned několika způsoby. Různé styly zakódování neuronových sítí do genotypu umožňují vývoj mnohem rozsáhlejších sítí, a přitom zachovávají udržitelně malou velikost genetické informace jedinců.

Některé metody Gomez a kol. (2008) navrhovaly postup, jakým můžeme omezit vývoj z celých neuronových sítí na pouze menší komponenty, které dále mohly být spojovány dohromady s ostatními jedinci v kooperativní evoluci. To umožnilo evoluční vývoj sítí rozsáhlejších architektur s menšími nároky na velikost genotypu.

Jiné metody se poté zaměřily na vývoj jak vah, tak topologie neuronové sítě. Navíc se ukázalo, že současný vývoj topologie často přináší lepší výsledky, než vývoj pouze vah sítě. Vývoj v těchto metodách začíná s nejzákladnější strukturou sítě, která se podle nároků problému sama vyvíjí a rozšiřuje, dokud tyto změny přináší kvalitativní zlepšení. Často využívaným algoritmem využívající těchto metod je algoritmus NEAT (*NeuroEvolution of Augmenting Topologies*), který si představíme v následujícím oddílu.

1.3.1 NEAT

NEAT (*NeuroEvolution of Augmenting Topologies*) (Stanley a Miikkulainen, 2002) je neuroevoluční algoritmus vyvíjející najednou váhy synapsí i topologii neuronové sítě, který se díky své výkonnosti stal jedním z nejznámějších algoritmů používaných pro účely evolučního vývoje neuronových sítí. Autoři jeho efektivitu připisují třem základním principům, se kterými algoritmus pracuje:

1. značkování genetických informací pomocí tzv. *historických značek*, umožňující smysluplné křížení genotypů napříč různými topologiemi,

2. ochrana nových genetických informací v populaci pomocí rozdělení do druhů,

3. postupný vývoj topologie sítí od nejjednodušších struktur ke složitějším.

S těmito principy se NEAT ukázal jako algoritmus, který často nachází efektivní sítě rychleji než ostatní algoritmy a výkony překonal nejlepší neuroevoluční algoritmy pracující s neuronovými sítěmi s fixní topologií.

Algoritmus NEAT používá tzv. *přímé kódování*, tedy genotyp každého jedince přímo popisuje celou topologii sítě (všechny neurony a všechny hrany mezi neurony). Každý genotyp obsahuje seznam *genů synapsí* a seznam *genů neuronů*.

Seznam genů neuronů popisuje vstupní, výstupní a skryté neurony sítě, které mohou být spojené hranami. Každý gen synapse obsahuje následující informace:

- vstupní neuron – neuron, do kterého synapse vchází,
- výstupní neuron – neuron, ze kterého synapse vychází,
- hodnotu váhy synapse,
- příznak, jestli je spojení v síti použito,
- *historická značka* – číslo, popisující kdy v historii byla daná hrana do sítě přidána; umožňuje nacházet odpovídající geny synapsí při křížení.

Algoritmus začíná s nejzákladnější strukturou, připomínající jednoduchý perceptron, pouze s předem daným počtem vstupních a výstupních neuronů a hranami mezi nimi. Tato jednoduchá topologie je rozšiřována pomocí následujících genetických operátorů.

Mutace Mutace v algoritmu NEAT má schopnost měnit jak spojení v síti, tak její strukturu. Mutace synapsí sítě zahrnují jednoduchou změnu váhy spojení nebo změnu příznaku použití daného spojení v síti. Struktura sítě může mutovat dvěma způsoby. První typ mutace přidává nový gen synapsí, spojující dva doposud nespojené neurony. Druhý typ mutace přidává nový neuron. Tato mutace probíhá tak, že na místo existující synapse se přidá nový neuron. Původní synapse se označí jako nepoužívaná a namísto toho se vytvoří dvě nové rozdělující tu původní. Váha spojení je v tomto procesu zachována. Při těchto operacích je vždy novým genům synapsí navýšena jejich *historická značka* (používá se globální čítač, jehož číslo je s každým novým genem navýšeno). Růst sítí probíhá právě díky mutaci.

Křížení Křížení využívá *historické značky*. Dva jedinci se nejdříve hranami zarovnají pomocí jejich *historických značek*. Stejná čísla historických značek totiž v síti značí stejnou strukturu. Synapse, která se v obou jedincích schoduje se dědí náhodně z jednoho rodiče. Synapse, kterou má jen jeden z rodičů se dědí pouze když je v lepším z rodičů a pokud je v nějakém jedinci hrana neaktivní a v druhém aktivní, s danou pravděpodobností se tento stav v novém jedinci změní. Algoritmus je tímto stylem schopný vytvářet velké množství různých topologií. Tyto nové topologie, přestože jsou často důležité pro řešení zadaného problému, mají jen velmi malou šanci se v populaci menších topologií udržet, protože původní menší topologie jsou v době vzniku větší topologie často optimalizovanější než tyto nové topologie. Proto NEAT využívá systém, kterým chrání tyto nové topologie před vyhynutím z populace.

Ochrana nových druhů Nové topologie jsou chráněny rozřazením genotypů do odlišných druhů. Jednotlivé genotypy poté primárně soutěží s jedinci stejného druhu a nové druhy tak mají šanci se vyvinout a optimalizovat na jejich úroveň. NEAT pro výpočet odlišností jedinců opět využívá *historické značky*, pomocí kterých hledá společné hrany a vypočítá vzdálenost dvou genomů. Geny rozdělíme do několika kategorií. Buď se shodují (mají stejnou *historickou značku*), a pak jsou

označené jako shodné (*matching genes*), nebo se neshodují. Neshodné geny, které mají historické značky v rozmezí hodnot *historických značek* druhého z jedinců, se nazývají disjunktní (*disjoint*) a zbylé geny se nazývají přesahující (*excess*). Hodnota vzdálenosti dvou jedinců se poté počítá jednoduchou lineární kombinací počtu genů jako:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{1.7}$$

kde N je celkový počet genů ve větším z jedinců, E je počet přesahujících genů, D je počet disjunktních genů, $\overline{W}$ je průměrný rozdíl vah shodujících se genů a koeficienty c_1, c_2 a c_3 umožňují nastavovat důležitost těchto tří faktorů (Stanley a Miikkulainen, 2002).

Na začátku generace se pak vytváří seznam druhů. Pokud se objeví genom, který nezapadá do žádného z druhů, je pro něj vytvořen jeho vlastní nový druh.

Fitness Rozdělení do druhů má vliv i na fitness jedinců. Kvalita každého jedince se při výpočtu fitness dělí počtem jedinců stejného druhu (vzorec (1.8)). To zároveň omezuje druhy v ovládnutí celé generace a dál ochraňuje nové topologie.

$$f'_i = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{1.8}$$

kde sh je binární indikátor s hodnotou 1, pokud je vzdálenost $\delta(i,j)$ menší než daný práh δ_t, jinak má hodnotu 0. f_i je fitness hodnota jedince a f'_i jeho upravená hodnota.

Minimalizace dimenzionality jedinců Jak již bylo zmíněno, NEAT inicializuje celou populaci jedinců s minimální topologií, obsahující pouze nutný počet vstupních a výstupních neuronů, které jsou navzájem plně propojené a neobsahuje žádné skryté neurony. Nové topologie vznikají díky genetickým operátorům a všechna zvětšení genotypů jsou tedy v evoluci opodstatněná. Díky tomuto NEAT samovolně vede k vývoji minimálních topologií. To umožňuje, že tento algoritmus je často výkonnější než ostatní, protože prohledává minimální potřebný prostor pro nalezení řešení, oproti neuroevolučním algoritmům, které používají fixní topologie neuronových sítí.

1.3.2 HyperNEAT

Pro porovnání algoritmus HyperNEAT (*Hypercube-based NEAT*) (Stanley a kol., 2009) (Eplex) je neuroevoluční algoritmus rozšiřující algoritmus NEAT. HyperNEAT slouží pro vývoj umělých neuronových sítí fixní topologie.

Na rozdíl od algoritmu NEAT, používá HyperNEAT nepřímou reprezentaci vah sítě. Tyto váhy jsou reprezentovány pomocí jiné neuronové sítě (*Compositional Pattern-Producing Network*, zkráceně *CPPN*), která jako vstup dostává pozice dvou neuronů v prostoru a vrací váhu jejich spojení (synapse). Tímto stylem může *CPPN* být využita pro reprezentaci sítě libovolné topologie.

Síť *CPPN* je poté v algoritmu HyperNEAT vyvíjena pomocí NEAT.

Díky této nepřímé reprezentaci vah má HyperNEAT schopnost efektivně vyvinout velmi rozsáhlé neuronové sítě s předem určenou strukturou (takto dokáže napodobit regularitu velkého množství spojů v lidském mozku).

Naznačení průběhu HyperNEAT algoritmu:

1. Zvolit konfiguraci vyvíjené sítě (vstupní a výstupní neurony a rozložení skrytých neuronů),
2. Inicializovat NEAT algoritmus s *CPPN* sítěmi,
3. Běh NEAT algoritmu, dokud není nalezeno řešení:
   (a) pomocí *CPPN* daného jedince vytvoř synapse pro původní síť,
   (b) síť otestuj v testovacím prostředí pro výpočet kvality řešení,
   (c) s fitness hodnotami pokračuj ve vývoji genotypů popisujících *CPPN* sítě.

Protože v práci nebudeme HyperNEAT používat, detailnější popis vynecháme.

1.4 Simulované prostředí

Jelikož chceme vyvíjet řízení robotů založené na interakcích s prostředím, je pro tuto práci důležité vybrat vhodný simulátor prostředí. Například při vývoji počítačových her se využívají různé výpočetně výhodné simulátory prostředí. My však chceme, aby se chování objektů v simulátoru virtuálního prostředí co nejvíce blížilo chování objektů v reálném prostředí. Proto budeme využívat pouze robustní simulátory prostředí, založené na fyzikálních zákonech. Budeme rozlišovat mezi *simulátorem prostředí*, který eviduje objekty (a vlastnosti objektů jako třeba rozměry, váha, atd.), a *simulátorem fyziky*, který řeší interakce mezi objekty tak, aby odpovídaly fyzikálním zákonům.

Přáli bychom si mít možnost jednoduše konfigurovat co nejvíce vlastností prostředí a zároveň mít co nejjednodušší přístup k morfologii simulovaných robotů. Zároveň chceme, abychom měli možnost do morfologie robotů nějakým způsobem zasahovat i v průběhu evolučního vývoje a aktivně ji za běhu měnit. Jelikož plánujeme v různých prostředích provádět experimenty s různými typy robotů, používajícími různé styly pohybu (typy motorů, kloubů, tvarů končetin, atd.), je potřebné, aby fyzikální simulátor (*fyzikální řešič=solver*) byl schopný simulovat i složitější typy robotů. Takovými mohou být právě třeba kráčející roboti, neboli roboti používající k pohybu končetiny připomínající nohy, na rozdíl od jednodušších typů robotů, kteří se mohou pohybovat pomocí kol, jejichž simulace bývá mnohdy jednodušší.

Stejně tak, jak potřebujeme umožnit simulaci složitějších robotů, protože nebudeme mít možnost vlastnoručně kontrolovat každý parametr, který bude při vývoji robotům přiřazen, potřebujeme zajistit, aby fyzikální simulátor zvládal velké rozsahy parametrů a simulace zůstala s těmito parametry stabilní. Zároveň chceme, aby simulátor v prostředí byl deterministický, což umožní, že předváděné experimenty můžeme dle potřeby opakovat a výsledky tak náležitě prezentovat.

Evoluční algoritmy jsou velmi lehce paralelizovatelné a tedy pro urychlení procesu vývoje a experimentů bude pro nás výhodné, pokud by simulace zvládala paralelní běh na více vláknech (více simulací, každá na vlastním vlákně). V neposlední řadě pro lehčí integraci do vlastního modulu bude užitečné, aby modul spravující zvolený simulátor byl open-source, což nám dá volnost v případě, že si budeme chtít chování systémů v prostředí nějak vlastnoručně upravit.

V oddíle 1.4.1 popíšeme několik simulátorů prostředí, které využívají fyzikální simulátory. Samotné fyzikální simulátory představíme v oddíle 1.4.2.

1.4.1 Simulátory prostředí

Při hledání simulátorů prostředí, které by vyhovovali našim požadavkům a umožňovali kontrolu a ovládání prostředí prostřednictvím zvoleného jazyka Python, jsme narazili na několik možností. Omezený výčet těchto simulátorů zde popíšeme – Gazebo (v oddílu 1.4.1.1), Webots (v oddílu 1.4.1.2) a CoppeliaSim (v oddílu 1.4.1.3).

1.4.1.1 Gazebo

Gazebo (OpenRobotics) je sada open-source víceplatformních knihoven pro vývoj, výzkum a aplikaci robotů, která vznikla v roce 2002. Umožňuje kompletní kontrolu nad simulací dynamického 3D prostředí s více agenty a generování dat ze simulovaných senzorů. Fyzikálně korektní interakce v prostředí pak od začátku projektu zajišťuje známý fyzikální simulátor ODE (viz sekce 1.4.2.1), nad kterým Gazebo tvoří abstraktní vrstvu, umožňující snazší tvorbu simulovaných objektů různých druhů. V dnešní době je stále výchozím fyzikálním simulátorem ODE, nicméně uživatel si již může vybrat celkem ze čtyř různých fyzikálních simulátorů – Bullet (sekce 1.4.2.2), Simbody, Dart (sekce 1.4.2.3) a ODE. Uživatel s knihovnou pracuje prostřednictvím grafického rozhraní založené na knihovně Open Scene Graph používající OpenGL, nebo prostřednictvím příkazové řádky. Prostředí a roboti mohou být tvořené buď z grafického rozhraní prostředí, nebo v textovém formátu XML. Limitací Gazebo je pak chybějící možnost rozdělit simulace mezi vícero vláken kvůli vnitřní architektuře spojené s fyzikální simulací (Koenig a Howard, 2004).

1.4.1.2 Webots

Webots (Webots) je open-source víceplatformní, robustní a deterministický robotický simulátor vyvíjený od roku 1998. Umožňuje programování a testování virtuálních robotů mnoha různých typů a jednoduchou následnou aplikaci softwaru v reálných robotech. Simulátor je možné použít pro simulaci prostředí s vícero agenty najednou. Agenti spolu mohou komunikovat a to jak lokálně tak i globálně. Výpočty fyzikálních interakcí zajišťuje fyzikální simulátor ODE. Pro vývoj robotů a prostředí je možné využít řady programovacích jazyků a to C, C++, Python, Java, MATLAB nebo ROS (*Robot Operating System*). Prostředí umožňuje práci v grafickém rozhraní a vizualizaci simulací pomocí OpenGL. Knihovna dále nabízí využití připravených modelů robotů, vlastní editor robotů a map a možnosti vložení vlastních robotů z 3D modelovacích softwarů v CAD formátu (Michel, 2004).

1.4.1.3 CoppeliaSim

CoppeliaSim (Rohmer a kol., 2013) (CoppeliaRobotics) (původně známý pod jménem *V-REP = Virtual Robot Experimentation Platform*) je víceplatformní simulační modul pro vývoj, testování a jednoduchou aplikaci softwaru pro roboty. Dovoluje vývoj ovladačů pomocí 7 různých programovacích jazyků a ulehčuje jejich aplikace v simulovaných a skutečných robotech. Simulaci ovladačů je možno jednoduše rozdistribuovat mezi vícero vláken dokonce vícero strojů, což urychluje vývoj a snižuje nároky na procesor v době simulace. Navíc je možné vyvíjený ovladač nechat v době simulací běžet na samotném (bezdrátově) připojeném robotovi, co dále ulehčuje přenos finální verze ovladačů z vývoje do skutečného světa. Prostředí umožňuje práci s širokou řadou typů objektů, druhů kloubů, senzorů a dalších objektů obvykle používaných při vývojích robotických ovladačů. Obsahuje snadno použitelný editor prostředí a robotů samotných s řadou předem vytvořených modelů, které může uživatel hned využít. Modely zároveň mohou být přidány v řadě různých formátů (XML, URDF, SDF). Prostředí podporuje pět různých fyzikálních simulátorů (Bullet, ODE, MuJoCo (v sekci 1.4.2.4), Vortex (v sekci 1.4.2.5) a Newton), mezi kterými si uživatel může vybrat dle potřeb přesnosti (reálnosti), rychlosti a dalších možností jednotlivých fyzikálních simulátorů (Nogueira, 2014).

1.4.2 Fyzikální simulátory

V této podkapitole se podíváme na základní popis a možné výhody a nevýhody jednotlivých fyzikálních simulátorů, na které jsme narazili při hledání simulátorů prostředí.

1.4.2.1 ODE

ODE (*Open Dynamics Engine*) (Smith) je víceplatformní open-source fyzikální simulátor, jehož vývoj začal v roce 2001. Je vhodný pro simulaci pevných těles s různými druhy kloubů a pro detekci kolizí. Byl navržen pro využití v interaktivních nebo real-time simulacích, upřednostňujících rychlost a stabilitu nad fyzikální přesností (Smith a kol., 2007). Vyžaduje používat menší simulační kroky kvůli stabilitě. Hodí se pro simulaci vozidel, kráčejících robotů a virtuálních prostředí. Má široké využití v počítačových hrách a 3D simulačních nástrojích jako jsou CoppeliaSim (v sekci 1.4.1.3), Gazebo (v sekci 1.4.1.1), Webots (v sekci 1.4.1.2) a dalších.

1.4.2.2 Bullet

Bullet je open-source fyzikální knihovna, podporující detekci kolizí a simulaci pevných a měkkých těles. Bullet je používán jako fyzikální simulátor pro hry, vizuální efekty a robotiku (Coumans). Byl použit jako hlavní fyzikální simulátor pro simulaci NASA *Tensegrity* robotů (s vlastními úpravami pro simulaci měkkých těles, kvůli nerealistickým metodám řešení simulace provazů) (Izadi a Bezuijen, 2018).

1.4.2.3 Dart

Dart (*Dynamic Animation and Robotics Toolkit*) je víceplatformní otevřená knihovna pro simulace a animace robotů. Od předchozích se odlišuje stabilitou a přesností, díky zobecněné reprezentaci souřadnic pevných těles v simulaci. Na rozdíl od ostatních fyzikálních simulátorů, aby dal vývojáři plnou kontrolu nad simulací, umožňuje Dart plný přístup k interním hodnotám simulace. Zároveň se díky línému vyhodnocování hodí pro vývoj real-time ovladačů pro roboty (Lee a kol., 2018).

1.4.2.4 MuJoCo

MuJoCo (*Multi-Joint Dynamics with Contact*) (DeepMind, 2021) je open-source fyzikální simulátor pro vývoj v oblasti robotiky, biomechaniky a dalších. Často je využíváno pro testování a porovnávání různých metod navrhování robotických systémů jako jsou třeba evoluční algoritmy nebo metody zpětnovazebného učení (Salimans a kol., 2017). V simulacích je pro roboty možné nakonfigurovat využití mnoha druhů aktuátorů, včetně těch simulujících práci svalů a k dispozici je i velké množství kloubů. Simulátor zároveň umožňuje velký nárůst v rychlosti běhu simulace za pomoci plné podpory paralelizace na všech dostupných vláknech a stabilitě simulace i při velmi velkých simulačních krocích (Todorov a kol., 2012). Zároveň nabízí jednoduchý styl, jakým si může uživatel konfigurovat všechny detaily simulace a samotných simulovaných robotů pomocí jednoduchých XML konfiguračních souborů (XML formát modelů *MJCF*). V komplexním rozboru řady četně používaných fyzikálních simulátorů byl simulátor MuJoCo hodnocen jako jeden z nejlepších co se týče stability, přesnosti a rychlosti simulací. Další výhodou zlepšující přesnost tohoto simulátoru je, že MuJoCo pro simulaci používá kloubní souřadnicový systém, který předchází narušení fyzikálních pravidel a tedy nepřesností v kloubech (Erez a kol., 2015).

1.4.2.5 Vortex

Vortex je uzavřený, komerční fyzikální simulátor určený pro tvorbu reálnému světu odpovídajících simulací. Obsahuje mnoho parametrů, umožňující nastavení reálných fyzikálních parametrů dle potřeb, většinou industriálních a výzkumných aplikací (CoppeliaRobotics) (Yoon a kol., 2023).

1.4.2.6 Porovnání simulátorů

V dnešní době se nám nabízí velké množství potencionálních kandidátů, vhodných k využití pro naši aplikaci. Prakticky každý z open-source simulátorů, které jsme našli a představili, by bylo možné použít pro simulaci robotů složitosti, jakou máme předběžně v plánu. Hlavním z rozhodujících faktorů pro tento projekt bude jak jednoduše půjde prostředí používat pro vývoj pomocí genetických algoritmů. Chceme tedy nějaký jednoduchý přístup k simulaci a ovládání robotů, rychlost a přesnost simulace.

Opět většina ze simulátorů prostředí toto nabízí. Osobně se nám ale nejvíce zalíbilo MuJoCo. Díky nedávnému převedení fyzikálního simulátoru MuJoCo do open-source a změně prostředí (nejprve do **OpenAI Gym** a nyní do **Farama**

Foundation Gymnasium) jsme dostali možnost využít jednoduché Python API pro ovládání robotů a zároveň konfiguraci celé simulace.

Tato abstrakce od vlastní simulace je pro tuto práci velmi přínosná, protože se především chceme zajímat o vývoj řízení robotů pomocí evolučních algoritmů. Řešit zároveň složité ovladače robotů, které by se mohly lišit pro různé typy robotů, by mohlo bezdůvodně komplikovat celý proces spojení evolučních algoritmů s řízením robotů. Takové věci by pak mohly být problematické pro možného uživatele, který by si chtěl sám evoluční algoritmy upravovat.

MuJoCo se zároveň ukazuje jako jeden z nejlepších volně dostupných fyzikálních simulátorů dnes. Z výsledků článku testujících různé vlastnosti známých fyzikálních simulátorů Erez a kol. (2015) vychází, že MuJoCo má navrch jak v rychlosti, tak ve kvalitě simulací. Zároveň interně využívá kloubní souřadnicový systém, který je přesnější, protože zabraňuje nepřesnostem v kloubech. To se hodí o to více, když v této práci chceme vyvíjet hlavně kráčející roboty, u kterých můžeme mít i větší počty kloubů.

Simulátor MuJoCo a roboti, které můžeme používat, je zároveň možné jednoduše konfigurovat pomocí vlastního XML formátu a spojení s Python API navíc umožní tyto konfigurace provádět jak často bude potřeba.

2. Specifikace

Vývoj pomocí evolučních algoritmů je možné nejlépe představit pomocí experimentů, na kterých může uživatel sám pozorovat změny, kterými postupný iterativní evoluční vývoj nachází možná řešení na zadaný problém. Je ale složité vytvořit takový systém, ve kterém by uživatel mohl snadno ovládat interní části evolučních algoritmů a tak vytvářet vlastní různorodé experimenty. A právě tyto experimenty mohou být zásadní pro pochopení specifických zákoutí aplikace evolučních algoritmů.

Proto cílem tohoto projektu je návrh knihovny, která by uživatelům, přicházejících z různých oborů, umožnila bližší pochopení a seznámení se s evolučními algoritmy pomocí vlastních interaktivních experimentů při vývoji robotů v simulovaném prostředí.

I jednoduché problémy, které od robotů můžeme požadovat vyřešit (např. ujití co největší možné vzdálenosti za daný čas), poskytují pro roboty různé složitosti (různé morfologie, počtu kloubů, atd.) dobrou představu v nárůstu obtížnosti daného problému. Tímto zároveň experimenty s různými roboty vynucují využití různých pokročilejších metod pro dosažení požadovaných cílů daného experimentu.

V následující sekci 2.1 zabývající se funkčními požadavky si představíme jednotlivé vlastnosti, které od takového systému budeme požadovat.

2.1 Funkční požadavky

Tento projekt cílí vytvořit systém umožňující uživatelům vytvářet vlastní experimenty s evolučním vývojem robotů v simulovaném fyzikálním prostředí. Uživatel by měl být schopný před spuštěním experimentu podrobně pochopit a upravit co nejvíce částí evolučního vývoje, který bude v době experimentu probíhat. Uživatel musí být v době běhu experimentu schopný sledovat průběžné výsledky z jednotlivých generací a vizualizovat dosavadní výsledky v simulovaném prostředí. Po dokončení experimentu musí být možné uložit výsledky ve formě dále zpracovatelné, např. pro statistický rozbor většího množství experimentů s možností vizualizace dat nejlepších jedinců finálních generací a podobně.

Systém bude z hlavní části vytvořený v programovacím jazyce Python, vytvářející uživateli přístupnější kód a umožňující rychlejší experimentování a prototypování nápadů. Python je vhodný, jelikož se pro tento systém nesnažíme o maximální efektivitu nebo rychlost experimentů, ale o čitelnost celého systému a schopnost vytvářet s naší knihovnou vlastní experimenty. Zároveň nám to umožní zpřístupnit otestovaný program uživatelům na operačních systémech Windows i Linux.

Univerzálnost navrženého řešení umožní uživateli přistupovat k našemu systému třemi způsoby popsanými níže. Každý způsob přístupu k systému má své vlastní požadavky, které s sebou přináší.

Grafické rozhraní Pro uživatele, kteří preferují jednoduchý přístup zprostředkovaný interaktivním grafickým rozhraním, musí systém umožňovat vytvářet a

konfigurovat experimenty dostatečné složitosti z prostředí tohoto grafického rozhraní. Uživatel tímto způsobem bude dále schopný pozorovat průběžné výsledky evolučního vývoje a vizualizovat průběžná nejlepší řešení, které evoluční vývoj najde.

Python knihovna Pro uživatele, kteří chtějí navrhovat vlastní experimenty, ale nechtějí všechno programovat od základů. Systém pro tvorbu experimentů vývoje robotů v simulovaném prostředí bude tvořit otevřenou Python knihovnu, kterou uživatel může připojit ke svému projektu a pomocí naší knihovny jednoduše vytvářet experimenty s požadovanou volností konfigurace jednotlivých parametrů evolučního vývoje.

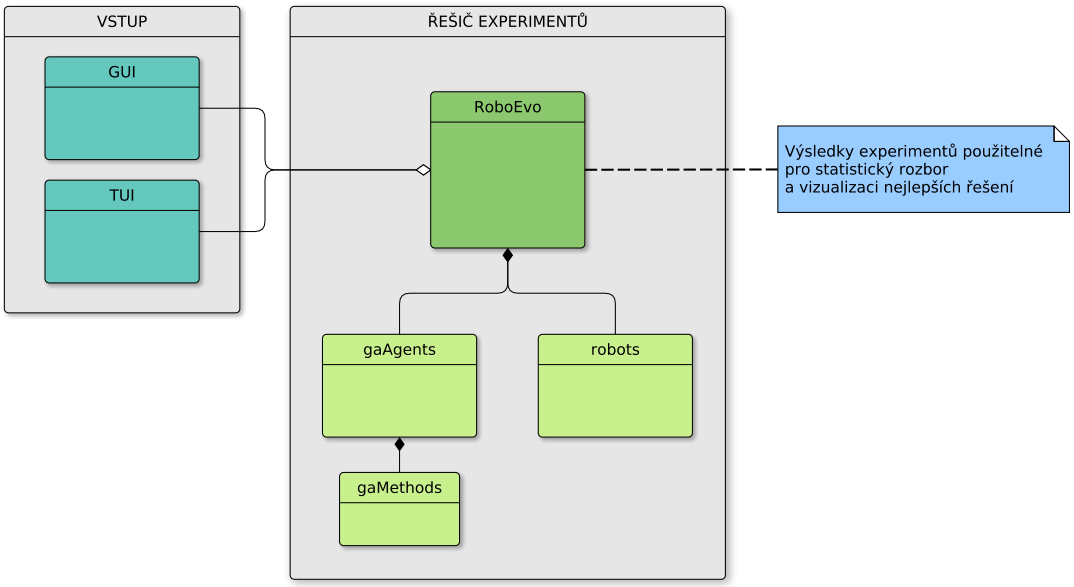
Knihovna bude mít dostatečnou dokumentaci na to, aby takový uživatel byl schopen provádět pokročilou konfiguraci experimentů přímo v kódu knihovny, a aby tyto experimenty bylo možné provádět z kódu, bez omezení výstupů systému nebo vizualizace řešení.

Rozšiřování knihovny Pro pokročilé uživatele bude navržený systém rozšiřitelnou platformou. Dokumentace představí a vysvětlí technologie využité při vývoji této knihovny a kód knihovny bude sestavený tak, aby byl dobře přístupný a jednoduše rozšířitelný.

3. Implementace projektu

V předchozí kapitole jsme prošli funkční požadavky, očekávané od vyvíjeného souboru programů. Následuje rozbor jednotlivých modulů, které vznikly při vlastní implementaci.

Programovací jazyk Celý projekt je napsána v programovacím jazyce **Python**. Cílem projektu je vytvořit čitelnou rozšířitelnou platformu, která bude uživateli jednoduše dostupná. Pokud uživatel bude mít potřebu vytvořené moduly jakkoli měnit nebo rozšiřovat, implementace v Pythonu toto bez problémů umožní. Jednoduchá čitelnost Pythonu spojená s rychlostí, jakou mohou být prováděny iterace změn, bez potřeby zdlouhavého překladu celé knihovny, se nám zdají býti dostatečně užitečné vlastnosti volbu Pythonu jako jazyka pro tento projekt.



Obrázek 3.1: Struktura projektu

Struktura projektu Obrázek 3.1 popisuje na jaké části je projekt rozdělen. Centrální částí je modul *RoboEvo*, pomocí kterého knihovna provádí evoluční experimenty. Tento modul se pro přehlednost a rozšířitelnost kódu skládá z několika menších částí – *gaAgents* (popisující agenty a vlastní genetické operátory) a *robots* (udržující jednotný způsob přístupu k různým robotům).

Jak popisují funkční požadavky (v sekci 2.1), projekt umožňuje několik možných způsobů práce s naší knihovny. Dva základní možné přístupy jsou za pomoci grafického, nebo textového rozhraní. Tyto přístupy jsou v projektu rozděleny do *GUI* (*Graphical User Interface*) a *TUI* (*Text-based User Interface*) modulů. Uživatel, který bude chtít pracovat s kódem části knihovny zaměřené na vytváření a provádění experimentů s evolučním vývojem, se dále může zaměřit na hlavní modul *RoboEvo* (a s ním spojené pomocné moduly).

Dále v této kapitole v sekci 3.1.1 popíšeme centrální modul *RoboEvo* pracující s několika dalšími pomocnými moduly, jejichž implementace popíšeme v dalších oddílech. Představíme si modul *robots* propojující roboty ze simulátoru *MuJoCo* (*MuJoCo* bylo popsáno v základních pojmech v oddílu 1.4.2.4) s ostatními třídami v oddílu 3.1.2. Dále si v oddílu 3.1.3 představíme třídu agentů. Popis třídy implementující vlastní genetické operátorů se nachází v oddílu 3.1.4. V sekci 3.2 představíme třídu `Experiment` modulu *experiment_setter*, sloužící k uchování a předvolbě parametrů pro experimenty, usnadňující tak provádění většího množství experimentů. Jako poslední si představíme moduly umožňující uživateli práci s knihovnou buď pomocí grafického prostředí (v sekci 3.3), nebo pomocí příkazové řádky (v sekci 3.4).

3.1 RoboEvo a pomocné moduly

V této sekci popíšeme hlavní modul *RoboEvo* a jeho pomocné moduly *gaAgents*, *gaOperators* a *robots*. Soubor těchto modulů tvoří hlavní část celého projektu, podporující celý proces provádění evolučních experimentů s roboty. Vytvořené rozdělení modulů bylo zvoleno pro zlepšení čitelnosti a zjednodušení rozšiřování kódu, kde nyní každý modul zprostředkovává velmi specifickou roli v procesu evolučního vývoje, a tudíž je pro uživatele jednoduché tyto části upravovat. Tato část projektu je zároveň zcela oddělena od zpracování uživatelského vstupu, který do hlavního modulu vstupuje z vnějších modulů až ve chvíli zahájení experimentu.

3.1.1 Modul RoboEvo

Modul *RoboEvo* je centrální modul tohoto projektu, sloužící pro spouštění a běh experimentů s evolučním vývojem robotů.

Každý experiment se skládá z několika nezávislých částí. Experiment může využívat různé typy evolučních agentů s různými genetickými operátory a může se snažit vyvíjet různé typy robotů. Jak bylo zmíněno výše, tyto částí jsou pro přehlednost, čitelnost a rozšiřitelnost kódu oddělené do vlastních menších implementací, rozšiřujících hlavní modul (jednotlivé implementace budou popsány v dalších oddílech). Oddělené moduly umožňují jednoduše kombinovat různé agenty s různými roboty.

Implementace modulu *RoboEvo* Tento modul obsahuje funkce sloužící jak k inicializaci evolučních experimentů, tak k samotnému běhu evolučních algoritmů, včetně propojení s knihovnou *Gymnasium* od Farama Foundation (popsáno v sekci 1.4.2.6), zprostředkovávající simulaci fyzikálního prostředí pro testování jedinců.

Hlavní funkcí, která z vnějšího vstupního prostředí (např. *GUI*, *TUI*, vlastní modul uživatele) přijímá parametry pro spuštění experimentů, je funkce pojmenovaná `run_experiment`. Povinným parametrem této funkce jsou parametry experimentu, které jsou vloženy do jednoduché třídy `ExperimentParams` (z modulu *experiment_params*) obsahující následující hodnoty:

- `robot` – zvolený robot z modulu `robots`,

- `agent` – zvolený agent z modulu `gaAgents`,
- `population_size` – velikost populace jedinců v evolučním algoritmu,
- `generation_count` – počet generací, po které bude evoluční algoritmus běžet,
- `show_best` – příznak určující, zda po doběhnutí evolučního algoritmu chceme v simulovaném prostředí zobrazit řešení nejlepšího jedince,
- `save_best` – příznak určující, zda po doběhnutí evolučního algoritmu chceme uložit nejlepšího jedince,
- `save_dir` – cesta ke složce, kam chceme uložit data z běhu evolučního algoritmu (pokud neexistuje, je složka automaticky vytvořena po doběhnutí algoritmu),
- `show_graph` – příznak určující, zda je při běhu algoritmu vykreslován graf zobrazující fitness hodnoty (minimální, průměrnou a maximální) v jednotlivých generacích,
- `note` – případná poznámka, kterou může uživatel speciálně odlišit název dat, ukládaných po doběhnutí algoritmu.

Funkce `run_experiment` zpracovává tyto parametry a zajišťuje vše potřebné pro běh experimentu. V přípravě probíhá spouštění výpočetních jednotek pro paralelizaci testovacího prostředí (umožňující ohodnocení populace jedinců paralelně). Následně proběhne spuštění evolučního algoritmu se zvolenými parametry, po kterém funkce uloží data vygenerovaná evolučním algoritmem – fitness hodnoty jedinců v každé generaci, celou populaci jedinců z poslední generace a (volitelné) nejlepší řešení na konci experimentu. Tato data jsou uložená do složky, dostupné na cestě popsané v parametru `save_dir`.

Běh evolučního algoritmu Funkce `run_experiment` zajišťuje spuštění evolučního algoritmu se zvolenými parametry. Samotný běh evolučního algoritmu je poté zajištěn funkcí `run_evolution`. V rámci této funkce provádíme všechny kroky evolučního algoritmu (jak byly popsány v základních pojmech evolučních algoritmů v sekci 1.1). Navíc zde pro jedince vytváříme simulační prostředí, ve kterých budou jedinci testováni při výpočtu fitness.

Po výpočtu fitness přichází na řadu genetické operátory, které jsou vždy specifické pro zvolený evoluční algoritmus. V naší implementaci jsou zvolené operátory specifikované v třídě agenta. Podrobněji třídu agentů popíšeme v dalším oddíle 3.1.3.

V rámci této funkce se zároveň sbírají důležitá data o vývoji fitness hodnot napříč všemi generacemi a pokud to uživatel povolil, jsou aktuální data v průběhu algoritmu vykreslována do jednoduchého grafu.

3.1.2 Modul robots

Pro přehledné rozdělení všech částí evolučního algoritmu, zlepšení čitelnosti a tvorby experimentů oddělujeme i třídu popisující roboty a práci s roboty do vlastního modulu *robots*.

Roboti a řízení robotů Roboti knihovny *Gymnasium* jsou popisováni pomocí XML konfiguračních souborů, specifikující různé vlastnosti jak samotných robotů, tak prostředí, ve kterém se pohybují.

Robot vznikne spojováním jednoduchých tvarů pomocí kloubů. Tyto klouby mohou být různých typů (např. pantový, kulovitý), kde každý typ se liší počtem os, ve kterých spojeným částem těla povoluje pohyb (např. pantový v jedné ose). Kloubu dále můžeme nastavit povolený rozsah (úhel ve stupních), ve kterém se bude moci pohybovat.

Následuje ukázka konfigurace pantového kloubu (`type="hinge"`) v jednom z výchozích robotů. Tento kloub je pohyblivý kolem osy z (`axis="0 0 1"`), interně pojmenován `hip_1` s rozsahem od -30 do 30 stupňů.

```
<joint axis="0 0 1" name="hip_1" pos="0.0 0.0 0.0" range="-30 30"
 type="hinge"/>
```

Do kloubů můžeme vložit různé aktuátory, pomocí kterého můžeme kloubem pohybovat (*MuJoCo* nabízí velké množství různých aktuátorů, jejichž celý popis je nad rámec této práce – více informací je možné najít v oficiální dokumentaci *MuJoCo* (MuJoCo)).

Většina z robotů ale využívá aktuátory jen dvou typů. Prvním je `motor` (umožňující vedle jiných specifikovat atribut převodu motoru (*gear*)) – aktuátor využitý výchozím robotem *AntV3* popsán později v oddílu 3.1.2.1). Následuje ukázka z XML konfigurace tohoto robota popisující konfiguraci aktuátor motoru. Důležitým je argument `joint`, který přiřazuje daný aktuátor ke zvolenému kloubu a argument `ctrlrange`, který udává rozsah hodnot (nastavení aktuátoru), která jsou přímo mapována na povolený rozsah kloubu.

```
<actuator>
  <motor ctrllimited="true" ctrlrange="-1.0 1.0" joint="hip_1"
   gear="100"/>
  ...
</actuator>
```

Druhým typem je `position` (abstrakce servomotoru), umožňující nastavit reálný atribut `kp` (*velocity feedback gain*), se kterým se můžeme potkat při nastavování PID ovladačů (*proportional-integral-derivative*). Tento aktuátor je v práci využíván v konfiguraci vlastního robota *SpotLike* (opět popsán později). Tento typ aktuátoru jsme využili, protože *SpotLike* je již složitější robot s dvanácti stupni volnosti a možnost ručního naladění servomotorů byla nutná. To umožnilo robotovi stabilní postoj bez příliš velkých korekcí při změnách nastavení servomotorů (jinak vedoucích k oscilacím).

Definovaný seznam aktuátorů (část v ukázce výše) poté v simulovaném prostředí vytváří způsob, jak robota ovládat. V pořadí, ve kterém jsme aktuátory definovali v XML souboru, můžeme robotovi v simulovaném prostředí posílat seznam hodnot odpovídající délky (počet aktuátorů). Aktuátory jsou následně nastavovány na specifikované hodnoty. Tedy například robot *SpotLike* s dvanácti stupni volnosti očekává seznam dvanácti hodnot (pokud má aktuátor nastavenou hodnotu argumentu `ctrllimited`, tak je libovolná vstupní hodnota vždy za běhu omezena na povolený rozsah specifikovaný argumentem `ctrlrange`).

Ovládání robota je poté sekvence n-tic reálných čísel (kde n je počet aktuátorů), které robotovi posíláme v každém kroku simulace. Je tedy možné si představit, jak složité je vytvořit i pro jednoduchého robota s minimálním počtem aktuátorů takové ovládání, které ho například rozpohybuje v určitém směru.

Třída `BaseRobot` Modul *robots* je vytvořen podobným stylem jako modul *gaAgents* (v sekci 3.1.3), tedy obsahuje jednu hlavní třídu `BaseRobot`, tvořící šablonu pro odvozené třídy jednotlivých robotů. Tato implementace zásadně zjednodušuje možné připojení vlastního robota do naší knihovny což umožní využívat vlastní roboty v experimentech s evolučním vývojem.

Třída `BaseRobot` obsahuje několik metod využívaných všemi odvozenými třídami a jednu abstraktní metodu, kterou každá z odvozených tříd musí implementovat sama (informace o robotovi, sloužící k jeho prezentaci v grafické aplikaci – popsané v sekci 3.3). Zároveň definuje všechna pole nesoucí informace o daném robotovi (identifikační hodnota *MuJoCo* prostředí, slovník jmen částí těla robota, které mohou být použité při evolučním vývoji a odkaz na náhledový obrázek, zobrazující robota, pro *GUI*). Tyto informace jsou uložené o každém robotovi voláním konstruktor rodičovské třídy s potřebnými parametry.

Následuje seznam veřejných metod základní třídy `BaseRobot`:

- `create(body_part_mask, individual, tmp_file = None)` – metoda, která na vstupu dostane argument masky částí těla popisující, které části budou moci být měněny evolučním algoritmem (seznam hodnot – buď hodnoty `False`/0 pro části těla, které se nemají měnit, nebo specifikování rozsahu ve formátu `tupple` – `(min, max)`) a argument jedince, ze kterého získává délky části těla daného jedince (používané při vývoji morfologie robota). Tyto hodnoty metoda použije při úpravě konfiguračního XML souboru, který popisuje robota v simulačním prostředí, a vrací nově vytvořený soubor s konfigurací (pokud metoda dostane argument `tmp_file`, tak konfiguraci vygeneruje do tohoto souboru),
- `create_default()` – stejná funkcionalita jako předchozí metoda, ale pro zjednodušení zápisu vytvoří konfiguraci robota s výchozím nastavením,
- `body_part_names()` – metoda vracející jména všech dostupných končetin robota.

Konfigurace vlastního robota Roboti v simulátoru *MuJoCo* jsou popisování konfiguračními soubory ve formátu XML (dokumentace k modelování v *MuJoCo* (MuJoCo)). Pro naši implementaci využíváme tento formát pro tvorbu šablony konfiguračních souborů, kde na místo některých číselných hodnot v konfiguraci můžeme dosadit speciální značky, které naší knihovně umožní konfigurační soubor upravovat za běhu algoritmu. Toto nám umožní vytvářet experimenty, které vedle pohybu mohou zároveň vyvíjet i morfologii robotů (např. velikost specifikovaných končetin).

Speciální značky mohou být ve tvaru např `$L_FRONT(0.75)$` nebo `@...@`.

Značka ohraničená symbolem $ se může nacházet na pozici teoreticky libovolné číselné hodnoty v konfiguračním souboru. Značka označuje a pojmenovává

hodnotu (zde např. interně pojmenovaná jako `L_FRONT` s výchozí hodnotou 0.75), kterou bude možno zvolit pro vývoj pomocí evolučního algoritmu.

Značka ohraničená symbolem @ označuje část konfiguračního souboru, která může obsahovat základní aritmetické operace, které budou vyhodnoceny po dosazení hodnot za všechny značky ohraničené symbolem $. Umožňuje tedy provádět jednoduché výpočty v konfiguračních souborech.

Značky jsou při vytváření robota vyhledávány pomocí regulárních výrazů očekávajících značky předvedeného formátu.

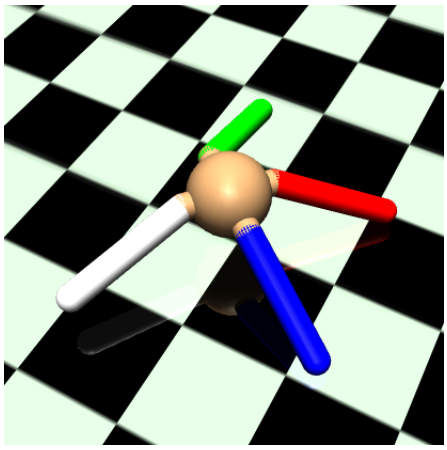
Pokud máme vytvořenou konfiguraci robota s korektním XML formátem v souboru (např. soubor *custom_stick_ant.xml*), můžeme ho jednoduše přidat mezi použitelné roboty, vytvořením odpovídající třídy v modulu *robots* (podle následující ukázky – tvorba robota *StickAnt*). Tato třída dále potřebuje určit cestu k náhledovému obrázku (v tomto případě *Basic-Ant.jpg*) a identifikační název *MuJoCo* prostředí, ve kterém bude robot testován (nyní všichni roboti využívají stejné prostředí – *custom/CustomEnv-v0*).

```
class StickAnt(BaseRobot): # tvorba robota StickAnt
    def __init__(self):
        # všechny soubory se nachází ve stejné složce jako
        # tento modul
        DIR = os.path.dirname(__file__)

        # cesta k šabloně konfiguračního souboru robota
        source_file = DIR+"/custom_stick_ant.xml"

        # cesta k náhledovému obrázku robota
        picture_path = DIR+"/Basic-Ant.jpg"


        # MuJoCo prostředí, ve kterém robot bude spuštěn
        environment_id = "custom/CustomEnv-v0"

        # 'custom/CustomEnv-v0' označuje naše prostředí
        # vytvořené tak, aby jednoduše podporovalo všechny základní
        # druhy robotů (bez robotů využívající NEAT)

        # volání inicializační funkce třídy BaseRobot,
        # která drží všechny informace v jednotném formátu
        super(StickAnt, self).__init__(source_file,
                                       picture_path,
                                       environment_id)

    # implementace vlastní abstraktní metody pro popis robota pro GUI
    @property
    def description(self):
        return """ Libovolně rozsáhlý popis robota """
```

3.1.2.1 Implementovaní roboti

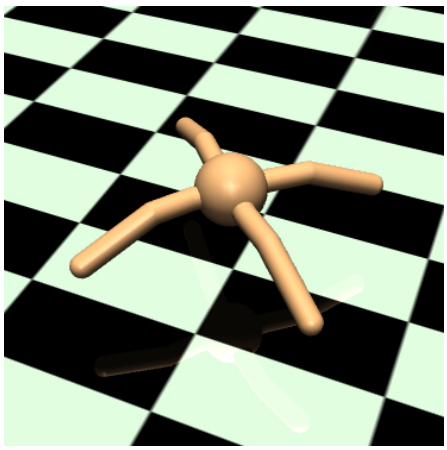
Knihovna obsahuje tři implementované roboty. Roboti by měli být dostatečně rozdílní v obtížnosti ovládání, aby se na nich mohl demonstrovat rozdíl v efektivitě pokročilých evolučních algoritmů.

Robot *StickAnt* Základním robotem je robot pojmenovaný *StickAnt* (ukázka na obrázku 3.2). Výchozím robotem pro jeho tělo je výchozí *AntV3* z knihovny *Gymnasium*. Morfologie tohoto robota je velmi jednoduchá. Jeho tělo se skládá z koule fungující jako torso a čtyř jednoduchých končetin, kde každá je jedním pantovým kloubem připojena na torso.



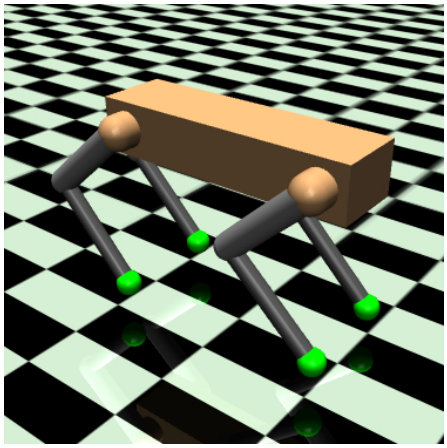
Obrázek 3.2: Robot *StickAnt*

Robot *AntV3* Jedná se o výchozího robota knihovny *Gymnasium* (ukázka na obrázku 3.3)). Morfologie tohoto robota je pokročilejší než morfologie robota *StickAnt*. Každá noha má navíc jeden pantový kloub, který můžeme brát jako koleno. Ovládání tohoto robota již začíná být složitější, kvůli tomu, že robot se může převrátit a spadnout.



Obrázek 3.3: Robot *AntV3*

Robot *SpotLike* Robot inspirovaný pokročilým robotem pojmenovaným *Spot* od firmy *Boston Dynamics* (popsaný v článku (Guizzo, 2019)). Jedná se o čtyřnohého robota s morfologií těla, která připomíná psa (ukázka na obrázku 3.4). Každou nohu ovládají tři klouby (dohromady tedy 12 stupňů volnosti pro celého robota), což z tohoto robota dělá toho nejobtížnějšího na ovládání. Čtyři vysoké nohy jsou zároveň vratké a tudíž je o to těžší vyvinout stabilní pohyb, který ho udrží nepadat.



Obrázek 3.4: Robot *SpotLike*

Roboti knihovny Gymnasium Naše knihovna umožňuje jednoduché přidání robotů, volně dostupných v knihovně Gymnasium. Většina těchto robotů má ve svém prostředí definované složitější cíle, kterých mají dosáhnout (např. balancování kyvadla ve vertikální poloze). Tato prostředí jsou často smysluplně využitelná pouze pro experimenty s vývoj pomocí algoritmů neuroevoluce. Několik vybraných robotů jsou jako ukázky již implementováni v naší knihovně – *Walker2D*, *InvertedPendulum*, *InvertedDoublePendulum*.

3.1.3 Modul *gaAgents*

Modul *gaAgents* je kolekcí několika tříd popisující agenty využívané evolučními algoritmy. Každá třída agenta povinně obsahuje několik funkcí, které specifikují jak vypadá genotyp jedinců vytvořených podle tohoto agenta, jakým způsobem se generuje populace takových jedinců, jaké genetické operátory budou při evolučním vývoji použité a jakým stylem probíhá transformace genotypu jedince na nastavení odpovídajících aktuátorů robota. Agent zároveň uchovává informaci o zvoleném typu evoluce, který určuje, zda evoluční algoritmus může vyvíjet řízení robota, jeho morfologii nebo obojí.

Agent pro evoluční algoritmus vytváří jedince, jejichž genotyp se skládá ze dvou vektorů. První z vektorů vždy obsahuje hodnoty, pomocí kterých agent generuje na základě vstupu z testovacího prostředí odpovídající nastavení pro aktuátory robota. Druhý z vektorů obsahuje hodnoty popisující délku těch částí

těla robota, u kterým jsme povolili jejich vývoj pomocí evolučního algoritmu. Pokud jsme nepovolili vývoj žádné části těla, tento vektor zůstane prázdný.

Třída agenta Hlavní třídou tohoto modulu, tvořící šablonu pro všechny další definované agenty, je třída `BaseAgent`. Všechny třídy popisující agenty mají povinnost odvozovat od této třídy základního agenta. Součástí této třídy je několik abstraktních metod (metody, které odvozená třída má povinnost implementovat, aby byla použitelná). Výčet abstraktních metod agenta:

a) metody využívané evolučním algoritmem:

- `generate_population(population_size)` – metoda vytvářející populaci jedinců požadované velikosti,
- `get_action(individual, step)` – metoda zprostředkovávající transformaci části genotypu jedince (kterou funkce získá ze vstupního parametru `individual`) na nastavení pro aktuátory daného robota na základě vstupu z testovacího prostředí (simulační krok v testovacím prostředí – z parametru `step`),
- `selection(population, fitness_values)` – genetický operátor – metoda s parametry populace jedinců a jejich fitness hodnotami, která nějakým způsobem (s využitím genetických operátorů selekce) vybere jedince a vrátí jejich seznam,
- `crossover(population)` – genetický operátor – metoda s parametrem populace jedinců (označující skupinu rodičů), na které provede křížení genotypů a vrátí vytvořenou skupinu potomků,
- `mutation(population)` – genetický operátor – metoda do které jako parametr vstoupí skupina jedinců (potomků křížení), na kterých provede mutaci genotypu a vrátí zmutované potomky,
- `switch_evo_phase()` – metoda využívána z hlavního modulu, který řídí evoluční algoritmus, sloužící pro vyhlášení změny typu evolučního vývoje (přechod mezi vývojem řízení a morfologie při odděleném vývoji řízení a morfologie),

b) metody využívané pro vstupní rozhraní:

- `description()` – metoda, do které můžeme vložit text, sloužící jako rozsáhlejší popisek agenta v *GUI*.

Využití abstraktních metod se pro tuto implementaci hodí, protože tímto způsobem můžeme v experimentech jednoduše zaměňovat typy využívaných agentů a měnit tak průběh evolučního vývoje.

Vlastní agent Uživatel může jednoduše přidávat vlastní agenty, vytvořením třídy odvozené od třídy `BaseAgent` a implementováním potřebných metod. Pro zjednodušení tohoto procesu, uživatel nemusí vlastnoručně programovat všechny tyto metody, ale může využít připravené genetické operátory, implementované v pomocném modulu *gaOperators*, který si představíme v následujícím oddílu 3.1.4. Pokud uživatel nenajde takovou funkci, která by přesně odpovídala jeho požadavkům, může si potřebný algoritmus dopsat sám, za dodržení pravidel specifikovaných implementací třídy agenta.

Existující agenti V následujícím seznamu krátce představíme všechny dostupné agenty připravené pro experimenty:

- ***StepCycleHalfAgent*** – agent, jehož genotyp je vektor předem zvolené délky pro polovinu aktuátorů robota, kde hodnoty v genotypu přímo popisují hodnoty nastavení aktuátorů. Tato nastavení jsou periodicky opakována v periodě zvolené délky vždy pro polovinu aktuátorů a pro druhou polovinu jsou symetricky přenesena a nastavena na opačné hodnoty (vynásobením hodnoty minus jedničkou – všechny aktuátory mají povolený rozsah hodnot od -1 do 1).

- ***StepCycleFullAgent*** – agent, podobný agentovi *StepCycleHalfAgent*, generující do svého genotypu nastavení pro všechny aktuátory robota pro určitý počet kroků. Tato nastavení se opět periodicky opakují.

- ***SineFuncFullAgent*** – agent, jehož genotyp popisuje parametry sinusové funkce pro každý aktuátor robota (amplituda, frekvence, posun v ose x a posun v ose y). Nastavení aktuátorů je pak vygenerované výpočtem sinus funkce pro každý aktuátor v daném simulačním kroku (z celočíselného parametru`step` funkce `get_action`), podle následující rovnice:

$$\text{nastavení aktuátoru} = f(step) = A \cdot \sin(\frac{2\pi \cdot step}{T} + \delta_x) + \delta_y \tag{3.1}$$

  kde T je perioda sinus funkce, A je amplituda funkce a δ_x a δ_y jsou její posuny. Hodnota parametru `step` může být pro plynulejší přechody mezi nastaveními aktuátorů dělena na menší hodnoty.

- ***SineFuncHalfAgent*** – agent, podobný jako *SineFuncFullAgent*, který má parametry sinusových funkcí pouze pro polovinu aktuátorů. Druhou polovinu generuje opět přenesením opačných hodnot z první poloviny. Pro výpočet hodnoty nastavení aktuátorů využívá stejné funkce, jako *SineFuncFullAgent* (funkce (3.1)).

- ***TFSAgent*** – složitější agent než *SineFuncFullAgent*, využívající genotyp popisující zkrácenou Fourierovu transformaci pro každý aktuátor na generování požadovaného nastavení. Genotyp obsahuje parametry pro amplitudy a posuny pro vybraný počet sinus funkcí. Výpočet nastavení jednoho aktuátoru potom vypadá dle následující funkce (3.2):

$$\text{nastavení aktuátoru} = f(step) = \sum_{i=1}^{N} A_i \cdot \sin \frac{i \cdot step \cdot 2\pi}{T} + \delta_i \tag{3.2}$$

  kde N je pevný počet sinus funkcí, na které součet omezíme, T je pevně zvolená perioda, A_1,...,A_N jsou amplitudy sčítaných sinusoid a δ_i,...,δ_N jsou jejich posuny,

- ***NEATAgent*** – agent, implementující algoritmus neuroevoluce – NEAT (*NeuroEvolution of Augmenting Topologies*), popsaný v základních pojmech v sekci 1.3.1. Naše implementace využívá python knihovnu pro NEAT – *neat-python* (McIntyre a kol.).

Kromě zcela náhodného agenta a agenta využívajícího NEAT, se agenti snaží pro urychlení evolučního vývoje předpokládat, že vhodné řešení bude v podobě nějakého periodického pohybu a generují tedy výstupy pro nastavení aktuátorů podle nějakých periodických funkcí.

3.1.4 Modul gaOperators

Modul *gaOperators* slouží pro usnadnění tvorby evolučních algoritmů implementací řady nejpoužívanějších genetických operátorů použitelných v metodách agentů (popsaných v předešlé sekci 3.1.3).

Seznam implementovaných operátorů (podrobný popis operátorů v sekci 1.1):

- `roulette_selection(pop, fitness_values)` – implementace *ruletové selekce*, s argumenty populace jedinců a jejich fitness hodnoty. Vyžaduje, aby všechny hodnoty `fitness_values` byly nezáporné,
- `tournament_selection(pop, fitness_values, k)` – implementace *turnajové selekce*, s argumenty populace jedinců, jejich fitness hodnoty a hodnotu k určující velikost turnaje,
- `tournament_prob_selection(pop, fitness_values, prob, k)` – implementace pokročilé *turnajové selekce*, kde umístění jedince v turnaji mezi k náhodně vybranými jedinci určuje jeho pravděpodobnost na zvolení podle vzorce:
$$p(X) = prob \cdot (1 - prob)^{(X-1)} \tag{3.3}$$
kde $X = 1,...,k$ je pozice, na které se daný jedince umístil v turnaji a *prob* je vstupní parametr funkce, určující pravděpodobnost na zvolení prvního. Toto rozdělení pravděpodobností je normalizováno tak, aby pro každou volbu k se sečetlo na jedničku. Následně je vektor pravděpodobností využit při náhodném výběru jednoho z jedinců v turnaji,
- `crossover_uniform(pop, agent)` – implementace základního operátoru uniformního křížení, popsaného v sekci 1.1, s argumenty populace jedinců a odkazem na zvoleného agenta, který pro metody udržuje další potřebné informace (např. sílu mutace jedinců, příznak označující povolení vývoje morfologie),
- `crossover_single_point(pop, agent)` – implementace operátoru jednobodového křížení, popsaného v sekci 1.1,
- `uniform_mutation(pop, agent)` – implementace jednoduchého operátoru uniformní mutace (popsané v sekci 1.1), využívající parametry agenta, který specifikuje pravděpodobnost mutace samotného jedince, pravděpodobnost mutace akcí a pravděpodobnost mutace částí těla (*individual mutation probability*, *action mutation probability*, *body mutation probability*),
- `uniform_shift_mutation(pop, agent)` – implementace upraveného operátoru uniformní mutace, využívajících stejných parametrů pravděpodobností jako předchozí operátor `uniform_mutation`. Tento operátor provede mutaci dané hodnoty vygenerováním hodnoty malé změny z povoleného rozsahu (příklad – náhodně zvolená $\delta \in [\frac{min}{0.05}, \frac{max}{0.05}]$), kterou přičte k původní

mutované hodnotě ($a = a' + \delta$, kde a' je mutovaná hodnota před mutací a a hodnota po aplikování mutace). Výsledná hodnota je poté omezena do povoleného rozsahu.

3.2 Třída experimentů

Pro usnadnění vytváření, ukládání a výběru experimentů jsme v modulu *experiment_setter* implementovali vlastní jednoduchou třídu `Experiments`. Tato třída má za úkol být seznamem nadefinovaných experimentů, které mohou být rychle načteny buď podle zvoleného názvu experimentu, nebo podle jména funkce, vytvářející experiment.

Hlavní datovou strukturou třídy je slovník, který má jako klíče názvy experimentů a jako hodnoty korespondující parametry experimentu (objekt třídy `ExperimentParams`). Tento slovník je naplněn v době inicializace třídy a uživatel do něj může přidat vlastní experimenty (je potřeba vytvořit záznam ve slovníku experimentů s hodnotou odpovídajících parametrů experimentu).

Třída `Experiments` obsahuje, všechny experimenty předvedené v této práci, ze kterých uživatel může brát inspiraci při vytváření vlastních. Samotný experiment je popsán vždy ve vlastní funkci vracející parametry experimentu. Jak může vypadat tvorba experimentu můžeme vidět v následující ukázce.

```
def exp10_TFS_spotlike(self, run=True):
    robot = robots.SpotLike()
    agent = gaAgents.TFSAgent(robot, ...)
    note = "exp1.0"

    # __create_batch_dir - Funkce, která na základě zvoleného robota,
    # agenta a zvolené poznámky vytvoří cestu, kam se uloží
    # výsledky experimentu.
    batch_dir = self.__create_batch_dir(robot, agent, note)

    # tvorba parametrů experimentu
    params = ExperimentParams(robot,
                              agent,
                              population_size=100,
                              generation_count=500,
                              show_best=False,
                              save_best=True,
                              save_dir=batch_dir,
                              note="")

    # pokud je experiment tvořen přímo voláním funkce, vypiš info
    if run:
        self.__exp_start_note()

    return params
```

Tato třída výrazně usnadňuje spouštění experimentů pro vstupní prostředí. Navíc díky dalším metodám (pro kontrolu správného výběru experimentu a pro

výpis všech vytvořených experimentů) uživatel může pracovat s experimenty, aniž by měl potřebu znát obsah této třídy.

3.3 Grafické rozhraní

Pro nastavování a spouštění experimentů může uživatel využít jednoduché grafické aplikace. Tento přístup je vhodný zejména pro prototypování nápadů na experimenty. Uživatel v aplikaci může rychle vybírat a měnit parametry jak agentů, robotů, tak samotného genetického algoritmu. Oproti tomu, použití grafického rozhraní nemusí být vhodné v případech, že uživatel bude chtít provádět větší množství předem připravených experimentů, při kterých běh grafického rozhraní není potřeba.

Implementace GUI Implementace používá modul *PySimpleGUI* (pys), umožňující jednoduchý a rychlý vývoj grafických aplikací v Pythonu.

PySimpleGUI umožňuje vytvářet prvky rozhraní (např. tlačítka, vstupní textová pole, ale i celá okna) uvnitř vlastních funkcí a prvky pak vracet jako hodnotu funkce. To oproti jiným grafickým knihovnám zlepšuje celkovou čitelnost a přehlednost kódu. Prvky rozhraní musí být v okně vložené do rozložení (*layout*). Tato rozložení ale mohou být jednoduše uložená v seznamech, díky schopnosti Pythonu pracovat se seznamy heterogenních dat.

Naše grafická aplikace je rozdělená do několika sekcí:

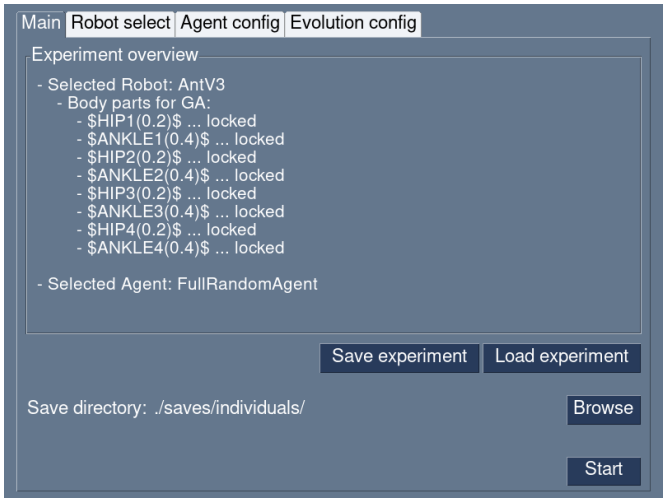
- *Main* – přehled informací o nastavovaném experimentu a spouštění experimentu (ukázka na obrázku 3.5),
- *Robot select* – sekce pro výběr robota (ukázka na obrázku 3.6),
- *Agent config* – sekce pro výběr a nastavení parametrů agenta (ukázka na obrázku 3.7),
- *Evolution config* – podrobnější úprava genetických operátorů a parametrů evolučního algoritmu (ukázka na obrázku 3.8).

Vytváření jednotlivých sekcí je oddělené do vlastních funkcí, které jsou pak při inicializaci aplikace volány z rozložení hlavního okna.

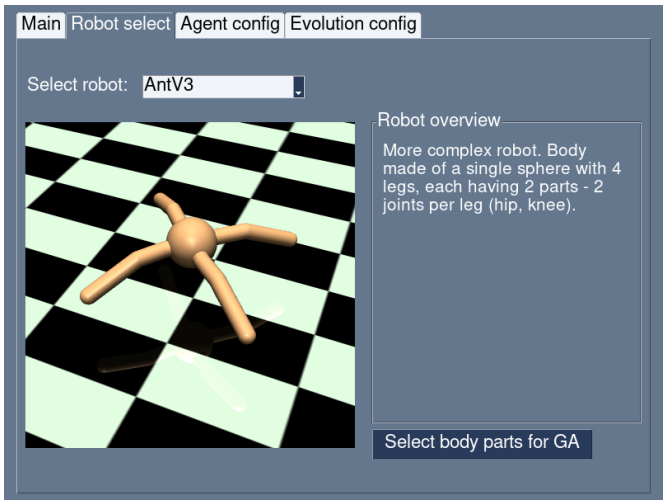
Běh aplikace je tak, jak tomu u grafických aplikací bývá, zajištěn nekonečným cyklem, který se opakuje vždy při příchodu nějaké události do aplikace (např. kliknutí myší na tlačítko v okně). Modul *PySimpleGUI* pracuje na základě posílání zpráv o událostech. Tyto zprávy jsou po obdržení zpracovávány uvnitř nekonečného cyklu a dle potřeby vyřešeny.

Určité sekce využívají napojení na pomocné moduly z centrálního modulu *RoboEvo* (popsané výše v sekci 3.1.1). Tohoto využíváme hlavně, abychom umožnili rozsáhlejší modifikace jednotlivých pomocných modulů (např. popis a nastavování parametrů agenta).

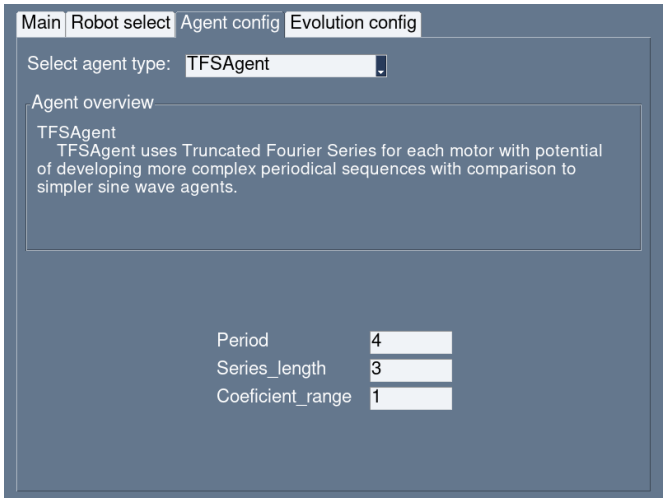
Pokud uživatel spustí vybraný experiment, proběhne proces, ve kterém se z navolených hodnot v aplikaci vytvoří parametry pro spuštění experimentu (objekt třídy `ExperimentParams`), které následně předá modulu *RoboEvo*. Zároveň



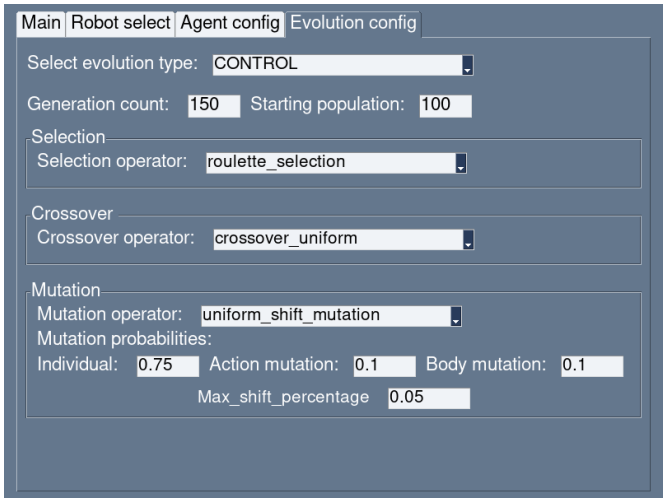
Obrázek 3.5: Úvodní okno aplikace



Obrázek 3.6: Okno pro volbu robota pro experiment



Obrázek 3.7: Okno pro volbu agenta pro experiment



Obrázek 3.8: Okno pro úpravu specifických nastavení agenta

se vytvoří nové okno, které vykresluje graf a vypisuje informace o právě běžícím experimentu. Z tohoto okna může uživatel v libovolném okamžiku zažádat o prezentaci jedince s dosud nejlepším řešením.

3.4 Textové rozhraní

Vedle grafického rozhraní lze knihovnu ovládat i pomocí zadávání příkazů do příkazové řádky. Tento přístup ke knihovně se může hodit při vypracování větších nebo vícero experimentů, kdy očekáváme, že experimenty poběží delší dobu (několik hodin). Pro tento účel nepotřebujeme a ani by nebylo nejvýhodnější po celou dobu sledovat okno grafické aplikace.

Textové rozhraní (TUI) je tedy vytvořené hlavně za účelem spouštění experimentů. Proto toto rozhraní úzce spolupracuje se třídou experimentů (popsanou výše v sekci 3.2). Rozhraní vytváří jednoduchý způsob, jak vybírat a spouštět vyhodnocení dostupných experimentů.

Ovládání TUI Textové rozhraní ovládá uživatel z příkazové řádky s různými vstupními argumenty. Těmito argumenty jsou následující:

- `--experiment` – argument, který obdrží textový vstup specifikující jméno jednoho nebo více experimentu (oddělených mezerou), jehož parametry chceme načíst a spustit (pracující s modulem *experiment_setter* popsaného v oddíle 3.2),
- `--experiment_names` – při uvedení tohoto argumentu program vypíše názvy všech dostupných vytvořených experimentů z modulu *experiment_setter* a následně se ukončí,
- `--batch` – argument číselné hodnoty, specifikující kolikrát se má nakonfigurovaný experiment opakovaně spustit (používané pro statistické vyhodnocení výsledků experimentů),

- `--batch_note` – textový argument umožňující připojit vlastní poznámku k názvu složky, do které se experimenty z několikanásobného spuštění ukládají (argument nemá žádný efekt pro experimenty z modulu *experiment_setter*),
- `--open` – textový argument, který obdrží cestu k uloženým datům nejlepšího jedince z libovolného předchozího experimentu, umožňující vizualizaci řešení daného jedince,
- `--no_graph` – argument, který značí, že za běhu algoritmu nemá být vykreslován graf průběhu fitness hodnot v jednotlivých generacích.

Ukázky možných vstupů Textové rozhraní úzce spolupracuje s modulem *experiment_setter*, který udržuje definované experimenty. Jedním z užitečných parametrů je `--experiment_names`, který TUI nechá vypsat názvy všech definovaných experimentů.

```
>>> TUI.py --experiment_names
<<< List of created experiments:
     - exp10_TFS
     - exp11_TFS_spot
     - exp12_TFS_ant
     ...
```

Pokud již známe jeden nebo více experimentů, které chceme spustit, můžeme je spustit výběrem parametru `--experiment` a vypsáním seznamu zvolených experimentů oddělených mezerou.

```
>>> TUI.py --experiment exp11_TFS_spot exp_12_TFS_ant
<<< Starting experiment - exp11_TFS_spot
    ...
```

Ve spojení s parametrem `--experiment` můžeme vybrat další parametry. Těmi mohou být parametr `--batch` (`--batch 5` bude opakovat běh všech zvolených experimentů 5krát), nebo parametr `--no_graph`, který zabrání průběžnému vykreslování grafů z běhu experimentu nebo parametr `--note`, umožňující upravit název složky, do které se budou data ukládat.

Posledním často používaným parametrem je `--open`, pomocí kterého si můžeme v simulačním prostředí přehrát běh nejlepšího jedince ze zvoleného předchozího experimentu.

```
>>> TUI.py --open saved_files/runs/run1/individual.save
<<< (Simulace zvoleného jedince)
```

Implementace TUI Samotná implementace rozhraní je již velmi jednoduchá. Zpracovává uživatelské vstupy z argumentů, které buď předává dál do korespondujících funkcí, nebo hlásí a řeší problémy, které mohli při zadávání argumentů nastat.

4. Experimenty a výsledky

V této kapitole se podíváme na tři typy experimentů, které s naším systémem můžeme provádět. Knihovna implementuje několik různých přístupů jak roboty řídit a jak je vyvíjet pomocí evolučních algoritmů. Následující experimenty předvedou vzorek z těchto přístupů.

Cílem všech následujících experimentů je pomocí evolučních algoritmů vyvinout zvoleného robota tak, aby byl schopný stabilního pohybu v simulovaném prostředí v předem určeném směru. Každý jedinec začíná svůj simulační běh v prostředí v počátku na souřadnicích (0,0). V našem experimentu chceme, aby se robot pohyboval ve směru rostoucí x-ové souřadnice. Kvalita jedinců je pak jednoduše vypočtena dle následující rovnice:

$$fitness = x - 0.5 \cdot |y| \tag{4.1}$$

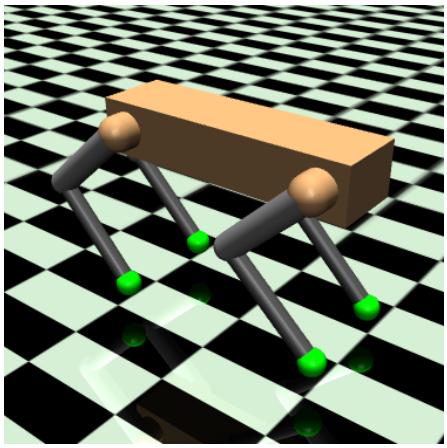
kde (x,y) jsou souřadnice bodu v simulovaném prostředí, do kterého jedinec dorazil (buď do vypršení limitovaného času na simulaci, nebo do dosažení podmínky předčasně ukončující simulační běh – např. pádu robota).

V prvním experimentu v sekci 4.1 si ověříme přirozený předpoklad, že pro řízení jednoduchých robotů nám stačí základní evoluční algoritmy a pro složitější roboty (s větším množstvím stupňů volnosti) potřebujeme pokročilé přístupy. V následujících dvou experimentech v sekci 4.2 popíšeme experimenty demonstrující možnost evolučního vývoje jak řízení tak morfologie robotů.

4.1 Vývoj řízení robotů

V této sekci se zaměříme na vývoj řízení robotů. Řízení robota je systém, který na základě vstupů z prostředí (senzory robota, čas, atd.) generuje signály do výkonných prvků robota (typicky motory).

Průběh vývoje řízení robota je ovlivněn zvoleným agentem, který definuje genotyp jedinců a určuje, jaké genetické operátory budou při vývoji využívány. Podrobnější popis agentů se nachází v sekci 3.1.3.

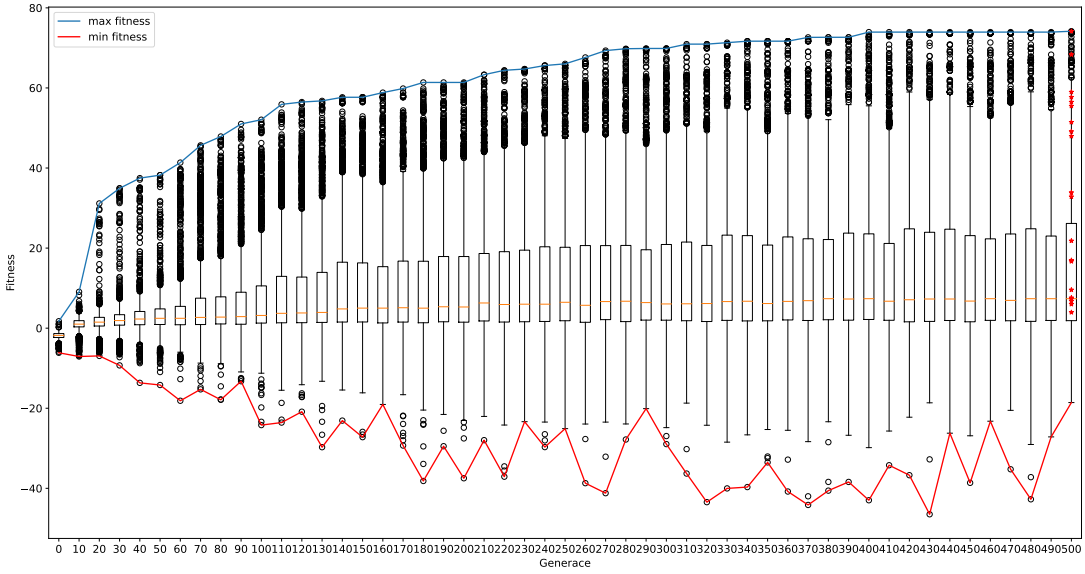


Obrázek 4.1: Robot *SpotLike*

První pokus se bude snažit vyvinout řízení pro pokročilého robota v projektu označovaném jako *SpotLike* (na obrázku 4.1, blíže popsaný v implementaci v sekci 3.1.2.1). Jedná se o robota kráčejícího na čtyřech nohách, kde každá noha má 3 stupně volnosti (tedy celkem 12 pro celého robota). Můžeme ho tedy řadit mezi roboty, u kterých již bude obtížnější vyvinout stabilní pohyb v určeném směru.

Kráčení, kterého bychom u robotů chtěli dosáhnout, si můžeme představit jako periodický pohyb, kde každá noha cyklicky opakuje stejné pohyby. Proto se pro vývoj řízení pokusíme využít agenty, kteří interně podle parametrů generují periodické hodnoty pro motory robotů.

Volba parametrů velikosti experimentu Pro tento typ experimentu inicializujeme populaci se **100 jedinci**. Abychom zjistili, kolik generací je potřeba pro rozumné výsledky, necháme nejprve proběhnout delší experiment, ve kterém vývoj evolučního algoritmu poběží po **500 generací**. Abychom vyloučili vliv náhodné inicializace populace jedinců v genetickém algoritmu, pro vyhodnocení je celý běh algoritmu **20 krát** nezávisle opakován a výsledky prezentujeme pomocí následujícího typu grafu.



Obrázek 4.2: Velký experiment, 20 opakování algoritmu po 500 generacích, robot *SpotLike*, agent *TFSAgent*

Krabicový graf Pro vizualizaci dat využíváme krabicový graf (boxplot). Tento graf vizualizuje statistické rozložení dat pomocí kvartilů. Uprostřed se nachází *krabice* (obdélník), který je shora ohraničena 3. kvartilem a zespodu 1. kvartilem. Rozsah pokrytý obdélníkem tedy obsahuje 50% pozorovaných hodnot. Uvnitř *krabice* se nachází (v našich grafech oranžovou barvou) čára, naznačující hodnotu mediánu. Dále krabicové grafy nad a pod střední *krabicí* vykreslují tzv. *vousy*, což vizualizuje variabilitu dat nad 3. a pod 1. kvartilem. Dále se v grafu mohou nacházet odlehlé body (*outliers*), které jsou vykresleny jako samostatné kroužky.

Grafy vznikají složením dat z průběhů fitness hodnot několika opakování daného evolučního algoritmu. Jeden krabicový diagram pro danou generaci zobrazuje rozložení hodnot fitness všech jedinců dané generace ze všech běhů. V grafu

4.2 je to celkem 2000 hodnot. V poslední generaci dále červené značky označují hodnotu maximální fitness v jednotlivých bězích.

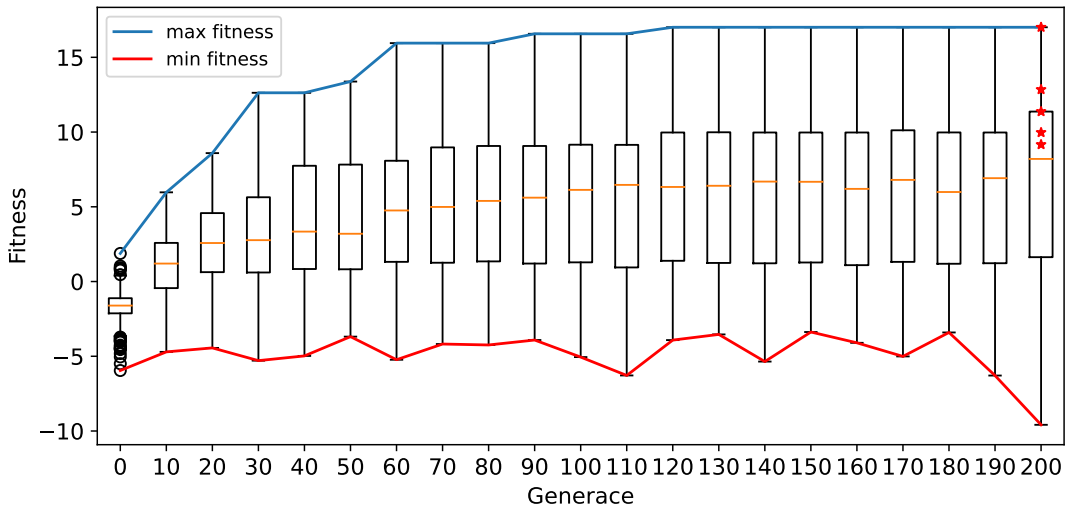
V našich grafech pro přehlednost dále vykreslujeme hodnotu maximální (modrou čarou) a minimální (červenou čarou) fitness v dané generaci.

Krabicový graf na obrázku 4.2 zobrazuje vyhodnocení popsaného velkého experimentu. Z grafu můžeme vidět, že největší růst fitness proběhl přibližně v prvních 200 generacích. Optimalizace dále pokračovala a byla schopná nacházet řešení s lepším hodnocením, ale pro účely vyhodnocování našich experimentů bude dostačující, když evoluci v tomto experimentu omezíme na 200 generací a algoritmus necháme vývoj pětkrát nezávisle zopakovat. To se nám ukázalo jako dostatečné množství dat, které se podobá předvedenému velkému experimentu a můžeme tak urychlit statistické vyhodnocení.

Průběh předvedeného experimentu v obrázku 4.2 trval přibližně 12 hodin i s využitím paralelizace ohodnocení jedinců mezi 12 vláken na výkonném osobním počítači. Díky omezení dalších experimentů na menší počet generací a opakování algoritmu jsme schopni na stejném systému následující experimenty provést v řádech desítek minut.

První experiment Nejprve se pokusíme řízení robota *SpotLike* (popsán v sekci 3.1.2.1) vyvinout pomocí evolučního algoritmu, který kóduje nastavení aktuátorů pomocí základních periodických funkcí (agent *SineFuncFullAgent* popisující tento algoritmus je popsán v implementaci v sekci 3.1.3). Každý aktuátor robota má v tomto případě přiřazenou vlastní periodickou funkci a genotyp jedinců specifikuje parametry těchto periodických funkcí (4 parametry pro každý kloub – amplituda, frekvence, x a y posun). Funkce jsou popsány rovnicí (3.1) v sekci implementace agentů.

Evoluční algoritmus běžel **200 generací** se **100** náhodně inicializovanými jedinci. Pro vyhodnocení byl celý běh evolučního algoritmu pětkrát opakován vždy s novou náhodně vygenerovanou počáteční populací.



Obrázek 4.3: Vývoj fitness populace v experimentu se základním agentem *SineFuncFullAgent* a robotem *SpotLike*

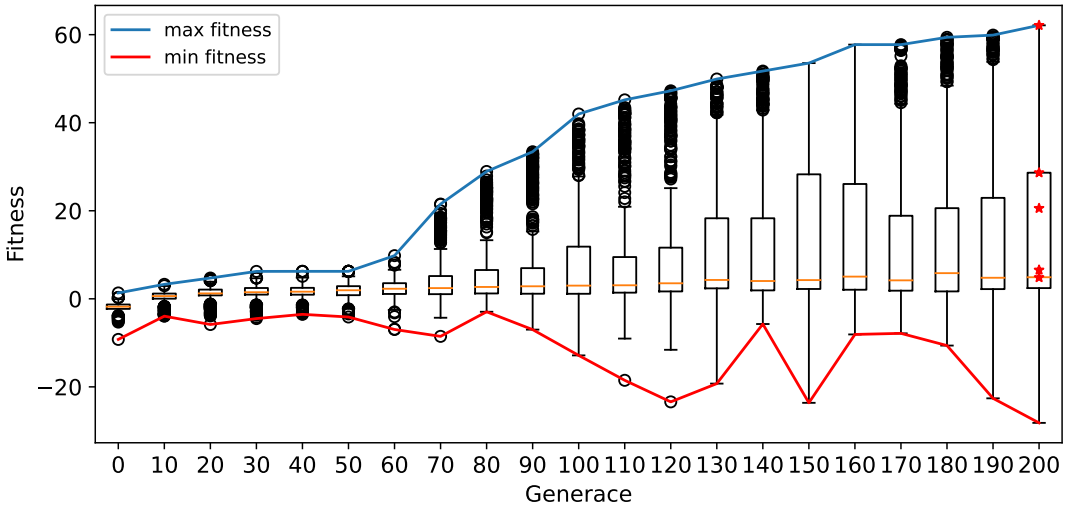
Graf na obrázku 4.3 zobrazuje vývoj fitness hodnot za běhu výše popsaného

evolučního algoritmu. Data jsou vytvořena kombinací záznamů o fitness hodnotách v dané generaci ze všech pěti nezávislých běhů.

Z grafu se ukazuje, že tento přístup vývoje řízení dosáhl maximální hodnotu fitness okolo 15 a populace na této hodnotě stagnovala a nebyla schopna většího posunu.

Pokročilý agent Pro porovnání jsme zvolili pokročilého agenta *TFSAgent*, který generuje signály do aktuátorů pomocí omezených Fourierových řad (agent byl popsán v sekci 3.1.3). Tento agent je na úkor malého zvětšení genotypu, oproti předchozímu agentovi schopný generovat mnohem komplexnější periodické funkce popsané skládáním několika funkcí sinus.

Stejně jako v předchozím běhu, algoritmus běžel 200 generací se 100 jedinci a byl opět pětkrát zopakován.



Obrázek 4.4: Vývoj fitness populace v experimentu s pokročilým agentem *TFSAgent* a se stejným robotem *SpotLike*

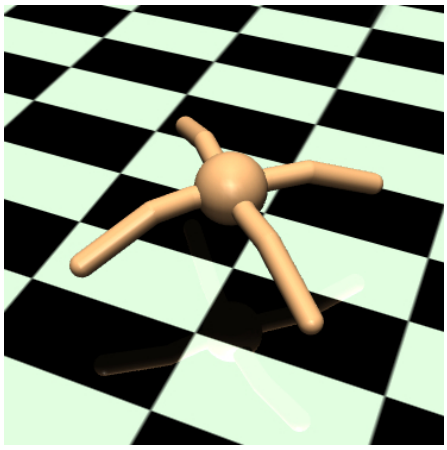
Graf 4.4 vývoje fitness hodnot z experimentu s pokročilým agentem ukazuje, že tento agent již byl schopný vyvinout stabilní pohyb i pro zadaného komplexního robota. Maximální fitness hodnota, které při vývoji agent dosáhl, se pohybovala okolo 60. Zároveň podle krabicového grafu vidíme, že poměrně velká část populace byla schopná dosáhnout výsledků, přesahující nejlepší výsledky jednoduššího agenta.

Z experimentů jsme zároveň dostali i nejlepšího jedince z poslední generace evolučního algoritmu. Jelikož naše simulované fyzikální prostředí je deterministické, můžeme řešení nejlepšího jedince zpětně vizualizovat.

Ruční kontrolou těchto výsledků jsme dále zjistili, že pouze část (dva z pěti běhů) dosáhly takového pohybu, který bychom od robota této morfologie očekávali. Tělo v robota v těchto případech (až na menší odchylky) směřovalo rovně, způsobem připomínající chůzi čtyřnohých zvířat podobné morfologie. Zbylé běhy vyvinuly stabilní, ale ne zcela estetickou chůzi. Roboti se v těchto případech posouvali stranou využitím většího rozsahu v rotaci (*kyčelních*) kloubů. Pravděpodobně kvůli lepší stabilitě robota.

Osobně si myslím, že vývoj estetického pohybu pro tohoto robota je s malou úpravou hodnotící funkce možný. Ta by například mohla penalizovat rotaci těla od požadovaného směru pohybu. Chůze stranou je kvůli rozsahu (*kyčelních*) kloubů (hlavně v ose délky těla robota) mnohem snazší na vyvinutí a tak tento způsob chůze tvoří silné lokální optimum. Agenti velmi rychle konvergují ke způsobům chůze, které jsou stabilní, což jim navyšuje šanci urazit větší vzdálenost bez pádu. Chůze stranou je oproti vratké chůzi rovně mnohem stabilnější. Navržená jednoduchá úprava hodnotící funkce by měla být schopna toto lokální optimum penalizovat a tedy přinutit estetičtější pohyb.

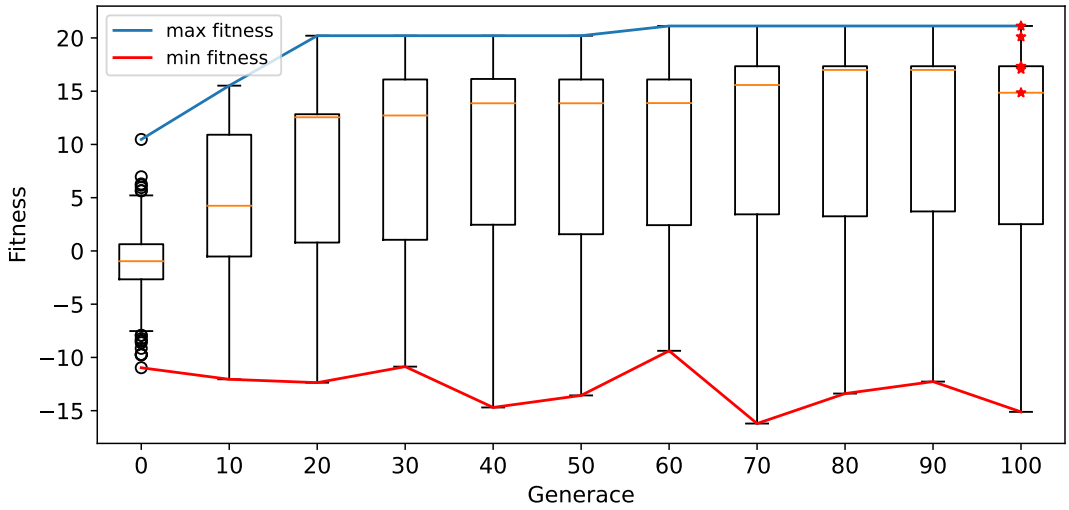
Ve srovnání s předchozími pokusy jsme se pomocí stejných agentů snažili vyvinout řízení jednoduššího robota označeného jako *Ant* (zobrazeného na obrázku 4.5, blíže popsán v implementaci v sekci 3.1.2.1). Můžeme ho brát jako jednoduššího, protože obsahuje menší počet kloubů (8 stupňů volnosti) v porovnání s robotem *SpotLike*. Zároveň jeho morfologie umožňuje snazší pohyb všemi směry s menším rizikem pádu.



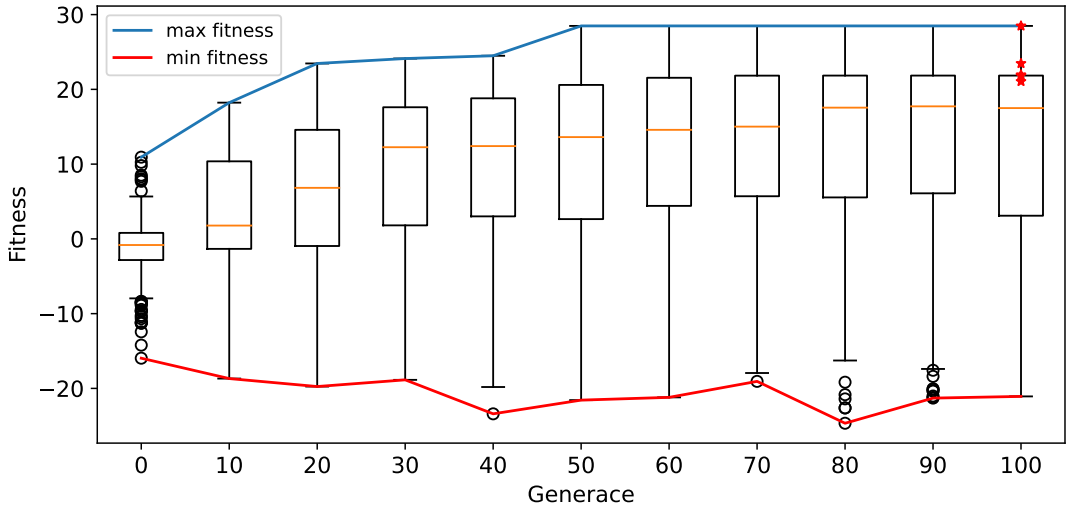
Obrázek 4.5: Robot *AntV3*

Řízení jsme vyvíjeli pomocí agentů se stejnými parametry jako v případě s pokročilým robotem. Celý běh byl opět pro statistické vyhodnocení pětkrát opakován se stejnou velikostí náhodně vygenerované populace (**100 jedinců**). Jelikož jsme kvůli menšímu počtu stupňů volnosti očekávali schopnost agentů vyvinout řízení rychleji, zkrátili jsme délku evoluce na **100 generací**.

Z vývoje fitness (v grafu 4.6 a 4.7) můžeme pozorovat, že oba agenti byli schopní u tohoto jednoduššího robota dosáhnout přijatelných výsledků i s omezeným počtem generací. Přijatelné výsledky jsou částečně podpořeny faktem, že i náhodné konfigurace mají dobrou šanci tohoto robota rozpohybovat v požadovaném směru a tedy již velmi brzy získat dobré ohodnocení. To se zároveň projevuje i na minimální fitness. Pro tohoto robota je totiž stejně jednoduché dostat takovou konfiguraci aktuátorů, které ho rozpohybují v opačném směru (jedinec z definice hodnotící funkce obdrží záporné ohodnocení).



Obrázek 4.6: Vývoj fitness populace se základním agentem *SineFuncHalfAgent* a jednodušším robotem *AntV3*



Obrázek 4.7: Vývoj fitness populace s pokročilým agentem *TFSAgent* a jednodušším robotem *AntV3*

4.2 Vývoj řízení a morfologie robotů

V této sekci předvedeme dva další typy experimentů, které naše knihovna podporuje – simultánní vývoj (v sekci 4.2.1) a oddělený vývoj (v sekci 4.2.2) řízení a morfologie.

Tyto experimenty se od předchozích liší tím, že evolučnímu algoritmu povolíme vyvíjet i zvolené části těla robota. To teoreticky umožní z těla původního robota vyvinout optimálnější morfologii pro zadaný problém.

Vývoj těla je implementací umožněn díky speciálním značkám v XML konfiguračních souborech (popsaných v konfiguraci vlastního robota v sekci 3.1.2).

4.2.1 Simultánní vývoj řízení a morfologie

V prvním příkladu předvedeme experiment, ve kterém umožníme evolučnímu algoritmu vyvíjet zároveň řízení i morfologii robota. Tento experiment by teoreticky mohl pomáhat v optimalizaci morfologie robotů za hranici představivosti jejich autorů.

Pro vývoj morfologie volíme při inicializaci agenta *masku* těla robota (popsána v parametru `body_part_mask` v sekci 3.1.3). Tato maska popisuje, které části těla mají při vývoji zůstat beze změny a zároveň určuje povolené rozsahy hodnot pro části těla otevřených pro evoluční vývoj.

Kvalita jedinců byla v tomto experimentu hodnocena stejně jako v předchozích experimentech. Cílem pro jedince je tedy dojít co nejdále v určeném směru. Přesný výpočet fitness je popsán rovnicí (4.1).

Pro experiment jsme zvolili robota *AntV3* (popsán v implementaci v sekci 3.1.2.1). Tento typ experimentů je ale samozřejmě povolen pro libovolného robota, který umožňující evoluční vývoj alespoň jedné části těla.

Vývoj robota *AntV3* Jak bylo řečeno v této ukázce jsme zvolili robota *AntV3*. Pro vývoj robota byl využit agent *SineFuncHalfAgent* (popsaného v sekci 3.1.3). *AntV3* je robot se složitější morfologií končetin a z tohoto důvodu ho vybíráme pro demonstraci tohoto typu vývoje.

Tento experiment je nakonfigurovaný v modulu *experiment_setter* pod názvem `exp2_body_para`. Pro evoluční algoritmus byla zvolena velikost populace na **150 jedinců** (větší velikost populace zvolena kvůli rozšíření prohledávaného prostoru, ve kterém evoluce hledá optimální řešení) a evoluci byl umožněn vývoj po **200 generací**. Zároveň byl zvolen povolený rozsah délek částí končetin pro vývoj. Robot *AntV3* (na obrázku 4.5, popsán v sekci 3.1.2.1) má 4 končetiny složené ze dvou částí – pro jednoduchost je označme jako *stehno* a *lýtko*. V základní konfiguraci je délka *stehna* pro tohoto robota 0.2 (pro vývoj zvolen rozsah mezi 0.1 a 0.5) a délka *lýtka* je 0.4 (pro vývoj umožníme rozsah mezi 0.15 a 0.5). Délka libovolné části těla je při vývoji nezávislá na ostatních.

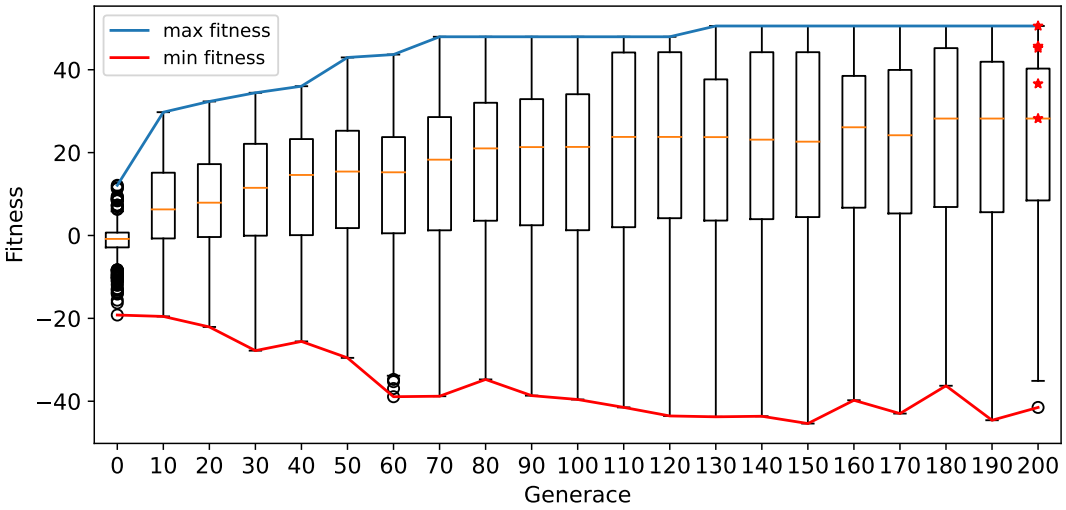
Z grafu na obrázku 4.8 můžeme pozorovat, že tento typ vývoje zvládl stabilně produkovat vyšší fitness hodnoty než srovnatelný předchozí experiment (využívající stejného agenta i robota) nevyužívající vývoje morfologie (graf na obrázku 4.7). Můžeme tedy odhadovat, že mohou existovat optimálnější konfigurace končetin tohoto robota, než jaká je výchozí.

Díky možnosti přehrávání nejlepších řešení jednotlivých běhů algoritmu můžeme zároveň prozkoumat, k jakým konfiguracím vývoj dospěl. Ukázalo se, že vývoj dospěl buď k robotům s velmi dlouhými končetinami, nebo (jak je na obrázcích 4.9) k robotům s několika dlouhými (*odrazovými*) končetinami a jednou kratší končetinou (sloužící jako *kormidlo*).

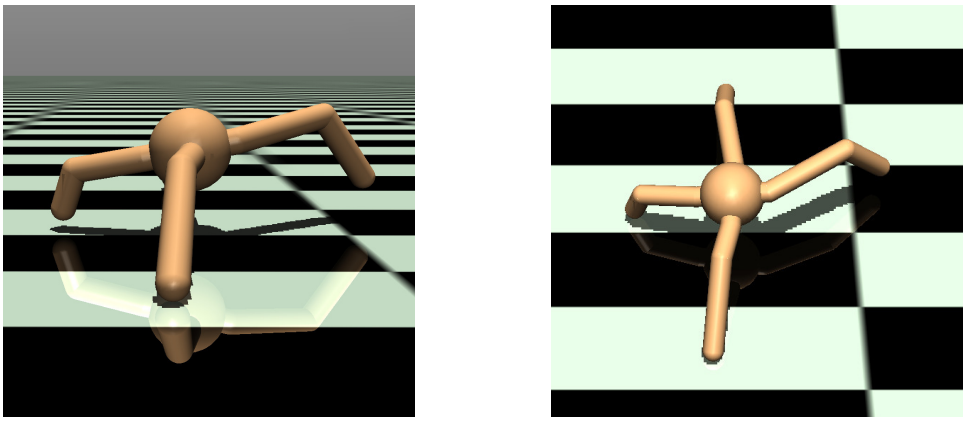
Obrázky 4.9 ukazují nejúspěšnějšího robota z předvedeného experimentu, který je schopný své tělo díky kratší zadní noze naklonit, a je tak schopný poměrně rychle skákat vpřed.

4.2.2 Oddělený vývoj řízení a morfologie

V tomto experimentu předvedeme druhý typ vývoje řízení a morfologie robota a to oddělený vývoj. V tomto typu evolučního vývoje se nejprve provede vývoj



Obrázek 4.8: Vývoj fitness populace při současném vývoji řízení a morfologie s agentem *SineFuncHalfAgent* a robotem *AntV3*



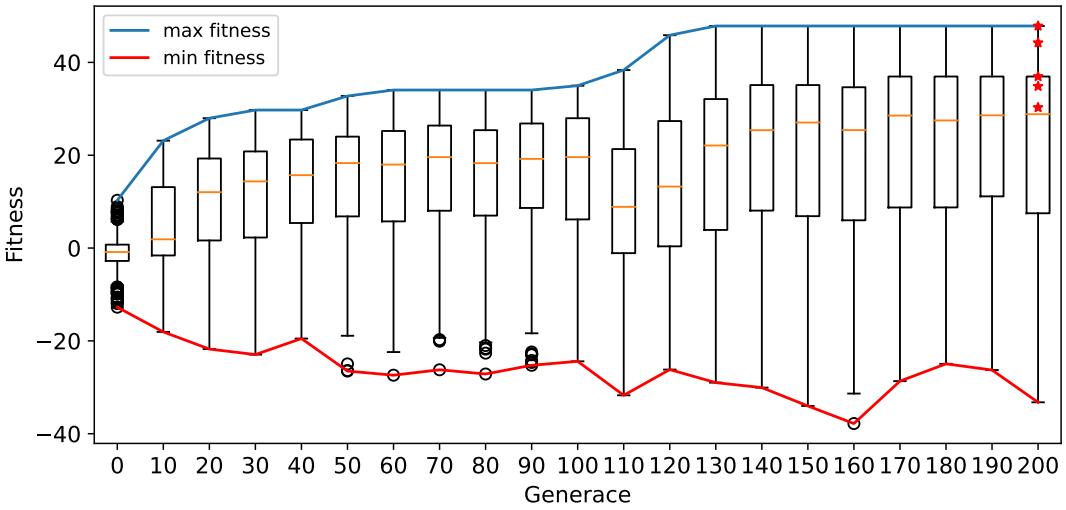
Obrázek 4.9: Příklad z výsledků současného vývoje řízení a morfologie robota *AntV3*

řízení robota s výchozí morfologií. Následně se vývoj řízení zafixuje ve stavu populace z poslední generace a začne druhá část vývoje, ve kterém se vyvíjí pouze morfologie robota.

Jedinci tak dostanou možnost nejprve vyvinout samotný pohyb (jak tomu bylo v prvním experimentu) a následně evolučním vývojem optimalizovat tělo pro specifický pohyb. Tímto přístupem by teoreticky jedinci měli být schopni dosáhnout lepších výsledků, než při vývoji samotného řízení.

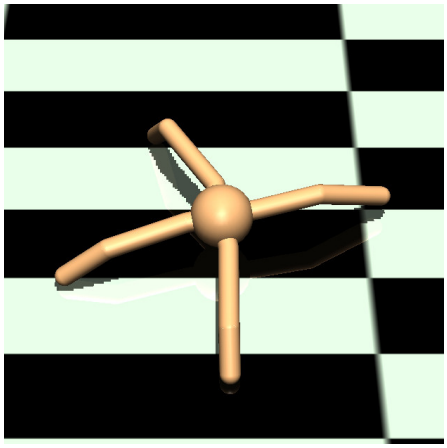
Pro tento experiment jsme zvolili stejného robota (s totožnými rozsahy velikostí končetin) a stejného agenta jako v předchozím experimentu. Konfigurace experimentu je v modulu *experiment_setter* pod názvem `exp2_body_serial`. Evoluční algoritmus běžel nejprve 100 generací, při kterých se vyvíjelo řízení robota a následně 100 generací pro vývoj morfologie.

V grafu na obrázku 4.10 můžeme pozorovat vývoj fitness v jednotlivých generacích. Můžeme vidět, že po 100. generaci přišel malý pokles průměrných fitness hodnot, což bylo zapříčiněno začátkem vývoje morfologie robota. Ten začíná vygenerováním zcela náhodných konfigurací těla robota (v povoleném rozsahu délek



Obrázek 4.10: Vývoj fitness populace při odděleném vývoji řízení a morfologie s agentem *SineFuncHalfAgent* a robota *AntV3*

končetin), a tedy může chvilkově vést k horším výsledkům. Časem ale vidíme, že evoluce byla schopná optimalizovat tělo robota pro již vyvinutý pohyb a tak celkově vylepšit výsledky jedinců.



Obrázek 4.11: Příklad z výsledků odděleného vývoje řízení a morfologie robota *AntV3*

Na obrázku 4.11 můžeme vidět nejlepšího jedince z experimentu s odděleným vývojem řízení a morfologie. Jedná se zároveň o dobrý příklad morfologie, ke které se řešení často blížila. Pro tento typ pohybu se robotům hodilo mít mnohem delší části končetin vedoucí přímo od těla (*stehno*), které svým pohybem jsou schopny posunout robota dále a tak optimalizovat uraženou vzdálenost.

Závěr

Seznam použité literatury

Python guis for humans. URL `https://www.pysimplegui.org/`. Accessed: May 1, 2023.

CoppeliaRobotics. https://www.coppeliarobotics.com. URL `https://www.coppeliarobotics.com`. Robot simulation software.

Coumans, E. Bullet real-time physics simulation. URL `https://pybullet.org/wordpress/`. Accessed: March 19, 2023.

DEAP, P. Deap documentation. URL `https://deap.readthedocs.io/`. Accessed: March 19, 2023.

DeepMind (2021). Mujoco. URL `https://mujoco.org/`. Accessed: March 26, 2023.

Eplex. URL `http://eplex.cs.ucf.edu/hyperNEATpage/`. Accessed: April 10, 2023.

Erez, T., Tassa, Y. a Todorov, E. (2015). Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 4397–4404. IEEE.

Fortin, F.-A., De Rainville, F.-M., Gardner, M.-A. G., Parizeau, M. a Gagné, C. (2012). Deap: Evolutionary algorithms made easy. *The Journal of Machine Learning Research*, **13**(1), 2171–2175.

Garrett, A. (2012). Inspyred: Bio-inspired algorithms in python. URL `https://pythonhosted.org/inspyred/`. Accessed: March 26, 2023.

Gomez, F., Schmidhuber, J., Miikkulainen, R. a Mitchell, M. (2008). Accelerated neural evolution through cooperatively coevolved synapses. *Journal of Machine Learning Research*, **9**(5).

Guizzo, E. (2019). By leaps and bounds: An exclusive look at how boston dynamics is redefining robot agility. *IEEE Spectrum*, **56**(12), 34–39.

Izadi, E. a Bezuijen, A. (2018). Simulating direct shear tests with the bullet physics library: A validation study. *PLOS one*, **13**(4), e0195073.

Koenig, N. a Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE.

Lee, J., X. Grey, M., Ha, S., Kunz, T., Jain, S., Ye, Y., S. Srinivasa, S., Stilman, M. a Karen Liu, C. (2018). Dart: Dynamic animation and robotics toolkit. *The Journal of Open Source Software*, **3**(22), 500.

Lehman, J. a Miikkulainen, R. (2013). Neuroevolution. *Scholarpedia*, **8**(6), 30977. doi: 10.4249/scholarpedia.30977. revision #137053.

McIntyre, A., Kallada, M., Miguel, C. G., Feher de Silva, C. a Netto, M. L. neat-python.

Michel, O. (2004). Cyberbotics ltd. webots™: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, **1**(1), 5.

MuJoCo. Mujoco modeling. URL `https://mujoco.readthedocs.io/en/stable/modeling.html`. Accessed: May 1, 2023.

Nogueira, L. (2014). Comparative analysis between gazebo and v-rep robotic simulators. *Seminario Interno de Cognicao Artificial-SICA*, **2014**(5), 2.

OpenRobotics. URL `https://gazebosim.org/`. Accessed: March 26, 2023.

Rohmer, E., Singh, S. P. N. a Freese, M. (2013). Coppeliasim (formerly v-rep): a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*. www.coppeliarobotics.com.

Salimans, T., Ho, J., Chen, X., Sidor, S. a Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Smith, R. URL `http://ode.org/`. Accessed: March 26, 2023.

Smith, R. a kol. (2007). Open dynamics engine.

Stanley, K. O. a Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary computation*, **10**(2), 99–127.

Stanley, K. O., D'Ambrosio, D. B. a Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, **15**(2), 185–212.

Todorov, E., Erez, T. a Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE.

Tonda, A. (2020). Inspyred: Bio-inspired algorithms in python. *Genetic Programming and Evolvable Machines*, **21**(1-2), 269–272.

Webots. http://www.cyberbotics.com. URL `http://www.cyberbotics.com`. Open-source Mobile Robot Simulation Software.

Yoon, J., Son, B. a Lee, D. (2023). Comparative study of physics engines for robot simulation with mechanical interaction. *Applied Sciences*, **13**(2), 680.

A. Přílohy

A.1 První příloha