UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

STROJOVÉ UČENIE POHYBOV HUMANOIDNÉHO ROBOTA LILLI

Bakalárska práca

2023 Lukáš Kostrian

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY

STROJOVÉ UČENIE POHYBOV HUMANOIDNÉHO ROBOTA LILLI

Bakalárska práca

Študijný program: Aplikovaná informatika
Študijný odbor: Informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Vedúci: Mgr. Pavel Petrovič, PhD.

Bratislava, 2023 Lukáš Kostrian

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Lukáš Kostrian
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Strojove učenie pohybov humanoidného robota Lilli
Machine Learning of movements of humanoid robot Lilli

Anotácia: Humanoidný robot Lilli má 25 stupňov voľnosti, výšku okolo 70 cm a je vyrobený z laserom vyrezaných drevených dielov, má stereovidenie a výkonný zabudovaný počítač. V predchádzajúcich záverečných prácach bol vytvorený simulovaný model, URDF špecifikácia robota, otestovaná inverzná kinematika v simulácii, analyzovala sa jeho možná chôdza, robota možno naučiť jednoduché postupnosti pohybov.

Cieľ: Cieľom práce je najskôr sprevádzkovať a zhodnotiť simuláciu robota Lilli podľa predchádzajúcej diplomovej práce. Študent ďalej analyzuje možné spôsoby učenia pohybov robota pomocou strojového učenia. Následne navrhne a implementuje metódu pre automatické naučenie sa chôdze alebo iných pohybov v simulácii. V prípade úspešného výsledku a dostatku ľudských a časových zdrojov overí a porovná naučené pohyby v simulácií s reálnym robotom a prípadne upraví simuláciu, aby bola vierohodnejšia.

Literatúra: Tomáš Kosec: Algoritmy riadenia humanoidného robota, bakalárska práca, FMFI UK, 2020.
Gabriel Halasi: Humanoid Robot Lilli, diplomová práca, FMFI UK, 2020.
Howie Choset a kol.: Principles of Robot Motion, MIT Press, 2005.

Kľúčové slová: humanoidný robot, kinematika, strojové učenie

Vedúci: Mgr. Pavel Petrovič, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 01.09.2022

Dátum schválenia: 30.11.2022 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Čestné vyhlásenie:

Vyhlasujem, že predloženú bakalársku prácu som vypracoval samostatne. Použil som literatúru, ktorú uvádzam v zozname použitej literatúry.

Bratislava 1. 6. 2023 .....

Poďakovanie:

Týmto by som chcel poďakovať môjmu školiteľovi, Mgr. Pavlovi Petrovičovi, PhD., za jeho sústavnú podporu, za jeho nesmierne nadšenie z práce, za jeho cenné rady a promptné odpovede, a hlavne za všetok čas, ktorý investoval do pomoci na tejto práci.

Abstrakt

Cieľom práce bolo nájsť také polohy kĺbov, pomocou ktorých by humanoidný robot Lilli vedel chodiť po dvoch nohách v prostredí simulátora CoppeliaSim. V simulátore CoppeliaSim sme napísali genetický algoritmus v jazyku Lua, ktorý hľadal potrebné polohy kĺbov, ktoré musel robot v poradí vykonať. Tie boli reprezentované ako polia, obsahujúce hodnoty uhlov, aké majú mať používané kĺby.

Úspešné riešenie, ktoré sme našli, reprezentovalo jeden krok pomocou troch polôh z toho prvá poloha bola počiatočná a genetickým algoritmom nemenená. Robota sme v simulátore inicializovali s už jednou nohou vykročenou. Krok druhou nohou sme vytvárali automaticky z kroku prvej nohy tak, aby boli oba kroky symetrické. Robot v simulátore striedal obe nohy na chôdzu. Vedel prejsť pomocou dvadsiatich krokov vzdialenosť 1,7 metra.

Porovnávali sme efektívnosť parametrov genetického algoritmu na hľadanie riešenia. Zamerali sme sa na crossover funkcie a silu mutácie. Z porovnávaných crossoverov bol najlepší 1-point crossover, ktorý vykonával kríženie na každej polohe zvlášť s pravdepodobnosťou 50 %. Z porovnávaných síl mutácie sme najlepšie výsledky dosiahli s mutáciou, ktorá menila jedincovi hodnotu každého uhla v každej z jeho polôh s pravdepodobnosťou 10 % o hodnotu $\pm 5°$.

Kľúčové slová: humanoidný robot, CoppeliaSim, strojové učenie, genetický algoritmus

Abstract

The objective of this thesis was to find joint positions, that would lead humanoid robot Lilli to walk on two legs in environment of CoppeliaSim simulator. In this simulator we wrote a genetic algorithm in programming language Lua, which searched for joint positions the robot needed to perform to walk. Those positions were represented as arrays of integers, containing values of angles for joints that were in use.

The successful solution, that we found, represented a single step using three joint positions, where the first position was the starting position and was not changed by the genetic algorithm. We initialized the robot in the simulator with one foot already in the front. Step, made with the second leg, was generated automatically by symmetrically inversing joint positions used by the first leg. The robot in simulator alternated between legs to walk and was able to cross 1,7 meters, making twenty steps.

We compared the efficiency of some parameters of genetic algorithm for this task. We compared the efficiency of crossover and mutation strength. Out of all crossovers, that we compared, the best was 1-point crossover, which was performing crossover on each position separately with 50% probability. From various mutation strengths we compared, the best results were achieved with mutation, that was changing individual's values for each angle with 10% probability and strength $\pm 5°$.

Keywords: humanoid robot, CoppeliaSim, machine learning, genetic algorithm

Obsah

Úvod

Humanoidní roboti v dnešnej dobe začínajú byť viac a viac populárni a používaní v praxi. Stroje, o ktorých naši predkovia snívali, sa v dnešnej dobe stávajú realitou. Prvé zmienky o humanoidných robotoch sú staré už stáročia, najstaršie z nich sú starogrécke mýty z 4. storočia pred naším letopočtom. Avšak predstava o strojoch, ktoré sa podobajú na človeka a slúžia mu, sa zjavovali všade po svete. Už naši predkovia si totiž uvedomovali nezvratné výhody, ktoré humanoidné roboty ľuďom prinášajú.

Humanoidní roboti sa svojim vzhľadom snažia priblížiť človeku alebo aspoň jeho časti. Ich najdôležitejšie využitie je na práce, ktoré sú pre človeka nemožné alebo príliš nebezpečné. Rovnako dôležité je aj ich využitie na pracovných pozíciách, ktoré sú fyzicky náročné, avšak nevyžadujú vyšší level intuitívneho myslenia. Tieto práce by totiž v budúcnosti mohli byť plne vykonávané humanoidnými robotmi. Pre medicínu sú humanoidní roboti rovnako dôležití. Časti humanoidných robotov sú využiteľné ako náhradné končatiny pre ľudí, ktorí nehodou o nejakú časť tela prišli alebo boli inak zranení a ich bežné pohybové schopnosti boli obmedzené. Robotickými náhradami, aké existujú už aj v dnešnej dobe, sa títo ľudia vedia začleniť do bežného života a naďalej bežne fungovať aj napriek mnohým zdravotným ťažkostiam. Kvôli týmto a aj mnohým ďalším dôvodom je vývoj v oblasti humanoidných robotov taký dôležitý.

V našej práci sa sústredíme iba na jeden z mnohých problémov, ktoré sú spájané s humanoidnými robotmi a ich plnohodnotnému nasadeniu do praxe. Sústredíme sa na to, aby sme humanoidného robota Lilli naučili chodiť po dvoch nohách. Hľadáme možné riešenie na to, ako by mohol humanoidný robot Lilli chodiť, pomocou strojového učenia v robotickom simulátore CoppeliaSim, ktorý je jedným z popredných robotických simulátorov súčasnosti. Taktiež sa pokúšame nájsť také parametre genetického algoritmu, ktoré proces hľadania chôdze zefektívni čo najviac. Výsledkom našej práce nie je len chôdza pre jedného humanoidného robota, ale taktiež genetický algoritmus, ktorý bude využiteľný na efektívne hľadanie chôdze pre iné modely alebo za iných podmienok. Stanovením rôznych účelových funkcií a obmedzení môže ten istý algoritmus generovať rôzne riešenia, napríklad hľadať chôdzu do určitého smeru, pričupenú chôdzu alebo aj iné pohyby.

1 Východiská práce

V tejto kapitole spomenieme teóriu, ktorú využívame v práci. Na začiatku kapitoly píšeme o softvéri CoppeliaSim, v ktorom pracujeme, ďalej píšeme o humanoidnom robotovi Lilli a spomenieme diplomovú prácu, na ktorú táto práca nadväzuje. Na záver kapitoly opíšeme algoritmy, ktoré budeme používať a už existujúce práce, ktoré sú podobné tejto práci.

1.1 CoppeliaSim

CoppeliaSim, známy v minulosti ako V-REP, je jedným z najpoužívanejších robotických simulátorov súčasnosti. Slúži na odsimulovanie správania robota v zadanom prostredí, aby neskôr mohol byť daný robot odskúšaný a zaradený do prevádzky mimo simulátora. CoppeliaSim je využívaný predovšetkým v automatizovanom a poloautomatizovanom priemysle. Taktiež sa využíva aj na študijné účely a vo výskume v oblastiach robotiky. Softvér je dostupný zdarma na osobné a študijné účely, v prípade komerčných účelov je potrebné si zakúpiť licenciu. Informácie spomenuté v tejto kapitole čerpáme z užívateľskej príručky pre CoppeliaSim [1].

1.1.1 Scéna a objekty

Hlavnou časťou samotnej simulácie je scéna. Tá reprezentuje jednu situáciu správania robota v prostredí. Scén môže mať užívateľ otvorených aj viac naraz a preklikávať sa medzi nimi, avšak zobrazená súčasne môže byť práve jedna. Scéna sa ukladá vo formáte ".ttt". Nachádzajú sa v nej objekty, ktoré tvoria samotnú scénu. Medzi najčastejšie objekty v scéne patria objekty ako kamera, svetlo, simulovaný robot a prostredie simulácie. Každý objekt má svoju pozíciu a orientáciu v danej scéne. Objekty v softvéri sa dajú ukladať do stromovej štruktúry. To znamená, že jeden objekt môže mať viacero dcérskych objektov, ktoré sú s ním spojené a tak isto každý dcérsky objekt môže mať svoje ďalšie dcérske objekty, ktoré k nemu prislúchajú. Takáto stromová štruktúra objektov sa nazýva model. Objekt, ktorý je koreňom tejto stromovej štruktúry, musí byť korektne označený ako základňa modelu. Modely sa dajú ukladať a načítavať zo súborov vo formáte ".ttm". Model nevie existovať a nedá sa zobraziť samostatne. Na jeho zobrazenie je potrebné ho vložiť do existujúcej scény.

1.1.2 Typy objektov

CoppeliaSim pozná niekoľko typov objektov. Každý objekt má svoj názov, ktorý sa dá zmeniť dvojklikom. Tak isto má každý objekt svoje objektové vlastnosti (Scene Object Properties), ktoré sa dajú zobraziť a meniť dvojklikom na ikonku daného objektu. Ďalej spomenieme bližšie informácie o niektorých objektoch simulátora.

1.1.2.1 Geometrický tvar (Shape)

Ide o najzákladnejší typ objektov. Sú reprezentované neohybnou mriežkou. Môže ísť o primitívne geometrické útvary (primitive shape), konvexné geometrické útvary (convex shape) alebo aj o úplne náhodné tvary (random shape) pričom platí, že čím komplikovanejší je daný geometrický útvar, tak tým viac zaobchádzanie s ním spomaľuje simuláciu. Z tohoto dôvodu sa odporúča využívať hlavne primitívne a konvexné tvary.

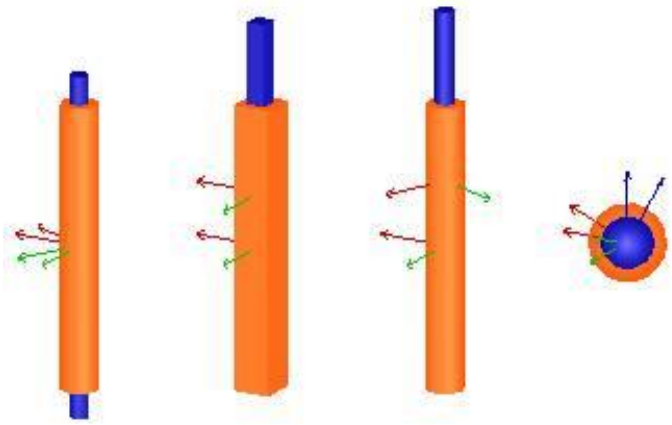
Takýto objekt môže mať nastavené rôzne dynamické vlastnosti. Vlastnosť dynamic znamená, že objekt sa vie hýbať v simulácii. Objekt je ovplyvňovaný gravitáciou a môže byť posunutý objektami, ktoré doňho narazia. Druhá vlastnosť respondable znamená, že objekt vie naraziť do iného objektu. Každý respondable objekt má dve masky, lokálnu masku, ktorú používa pri kolízii s objektami z rovnakého modelu a globálnu masku, ktorú využíva pri kolízii s ostatnými objektami. Každá maska je reprezentovaná ôsmimi bitmi. Dva objekty do seba môžu naraziť iba v prípade, že sú obidva respondable a bitový AND ich práve používaných masiek je rozdielny od nuly.

1.1.2.2 Kĺb (Joint)

Zabezpečujú rotáciu a posunutie medzi svojim rodičovským objektom a svojimi dcérskymi objektami. V CoppeliaSim existujú štyri druhy:

- Revolute joint – má jeden stupeň voľnosti, zabezpečuje rotáciu dcérskeho objektu po jednej z osí x, y, z
- Prismatic joint – má jeden stupeň voľnosti, zabezpečuje transláciu (posunutie) dcérskeho objektu v určenom smere
- Screw – má jeden stupeň voľnosti, ide o kombináciu revolute joint a prismatic joint, zabezpečuje rotáciu a transláciu dcérskeho objektu v smere danej osi pripomínajúci pohyb zaťahovanej skrutky
- Spherical joint – má tri stupne voľnosti, zabezpečuje rotáciu dcérskeho objektu po povrchu gule, čiže v smere každej z troch osí x, y, z, v niektorých prípadoch

je zameniteľný za tri revolute joints, každý zabezpečujúci rotáciu po inej osi



Obrázok 1: Revolute joint, prismatic joint, screw a spherical joint [1]

1.1.2.3 Senzory

Senzory slúžia na zistenie stavu okolitého prostredia v danom momente simulácie.

Proximity sensor slúži na zistenie toho, či sa v pozorovateľnom priestore senzora nachádza detekovateľný objekt a v akej vzdialenosti sa tento objekt nachádza. CoppeliaSim vie týmto spôsobom odsimulovať ultrazvukový alebo aj infračervený senzor.

Na rozpoznávanie intenzity svetla a farby objektov slúži vision sensor. Jeho využitie avšak spomaľuje výpočtovú rýchlosť v porovnaní s proximity sensor.

Posledným typom senzorov sú force sensors, ktoré slúžia na spojenie dvoch objektov a vedia merať silu, ktorá je na ne vyvíjaná. Vedia nasimulovať zlomenie daného spojenia v prípade, že sa prekročí nastaviteľná hodnota maximálnej sily, ktorá môže byť vyvíjaná na daný povrch.

1.1.2.4 Automaticky vytvorené objekty

Pri vytvorení novej scény v CoppeliaSim sa do scény automaticky pridajú niektoré objekty. Jedným z takýchto objektov je kamera. Tá umožňuje užívateľovi sa pozerať na simuláciu z rôznych uhlov. Je nevyhnutnou súčasťou scény, pretože bez nej by simulácia bežala na pozadí a užívateľ by nevidel, čo sa deje. Novovytvorená scéna automaticky obsahuje dve kamery. Jedna zaznamenáva pohľad zboku, druhá zase pohľad zhora. Záber z kamery sa dá kedykoľvek posunúť, otočiť alebo priblížiť pomocou myši. Záber z kamery, ktorý užívateľ práve vidí, je možné meniť alebo môže mať užívateľ zobrazených viacero záberov z rôznych kamier naraz. Do scény je možné pridať ľubovoľne veľa kamier. Kamera môže byť statická alebo sa môže dynamicky posúvať za nastaveným objektom, ktorý má sledovať.

Druhým automaticky vytvoreným objektom je svetlo, ktoré zabezpečuje osvetlenie scény. Jedna scéna môže mať maximálne osem svetiel. Bez tohto objektu je celá scéna tmavá a objekty nevrhajú tieň. Scéna je však stále viditeľná vďaka prirodzenému malému osvetleniu, ktoré vytvára prostredie samotné a jeho intenzita je nastaviteľná.

1.1.3 Programovanie v CoppeliaSim

CoppeliaSim umožňuje užívateľom upravovať skoro každý krok simulácie pomocou scriptov. Rozlišuje základné tri druhy: sandbox script, add-on a embedded script. Sandbox script a add-on sú si veľmi podobné, oba totiž bežia sústavne na pozadí aplikácie a rozširujú funkcionalitu celého programu. V práci využívame embedded scripty a v nasledujúcej časti ich rozoberieme viac do hĺbky.

1.1.3.1 Embedded script

Embedded script sú scripty pripojené ku scéne, modelom alebo objektom. Existujú dva druhy. Customization scripty slúžia na úpravu scény a objektov mimo behu simulácie. Naopak simulation scripts bežia iba v čase simulácie.

Najhlavnejším scriptom zo simulation scripts je main script. Ten je vytvorený automaticky a prislúcha scéne. Zabezpečuje chod celej simulácie ako aj spúšťanie ostatných scriptov počas behu simulácie a v akom poradí sa majú vykonávať. Bez neho nemôže simulácia bežať. Je modifikovateľný, avšak nie je odporúčané ho meniť, keďže kód v main scripte sa generuje automaticky a zabezpečuje, že aj scripty novo vložených modelov do scény budú spracované správne. Modifikovanie main scriptu môže spomaliť beh simulácie.

Child scripts sú scripty prislúchajúce objektom scény a sú spúšťané main scriptom. Majú vlastnosť, že pri kopírovaní a načítavaní modelu do scény sa s ním načítajú aj všetky jeho scripty. Táto vlastnosť umožňuje, že scéna sa dá ľahko rozšíriť o už hotové naprogramované modely, ktoré sú ihneď funkčné.

Každý simulation script má od svojho vytvorenie štyri základné systémové funkcie, s ktorými pracuje main script pri spúšťaní child scriptov. Funkcia sysCall_init sa spustí iba raz a to pri inicializácii objektu v scéne. Funkcia sysCall_actuation sa spustí v každom kroku simulácie. V tejto funkcii by mali byť príkazy, ktoré ovládajú pohyby objektov. Funkcia sysCall_sensing sa tak isto spustí v každom kroku simulácie. Táto funkcia slúži na spracovanie vstupov zo senzorov. Funkcia sysCall_cleanup sa spustí práve raz a to tesne pred skončením simulácie.

1.1.3.2 Programovacie jazyky

Scripty v CoppeliaSim sa dajú priamo v aplikácii písať v dvoch programovacích jazykoch, nimi sú Python a Lua. Oba jazyky sú aplikáciou rozšírené o CoppeliaSim príkazy, ktoré slúžia na prácu s aplikáciou a simuláciou. Všetky CoppeliaSim príkazy majú prefix sim. CoppeliaSim taktiež podporuje programovanie cez vzdialené API alebo aj cez ROS.

Rozdiel medzi programovaním v Pythone a v Lua je minimálny. Jedným z rozdielov je, že Lua kód je spustený v rovnakom vlákne ako samotný CoppeliaSim. Naopak, Python kód je spustený v samostatnom procese a komunikuje s CoppeliaSim pomocou socketov. Lua kód je kvôli tomu rýchlejší.

1.1.4 Simulácia

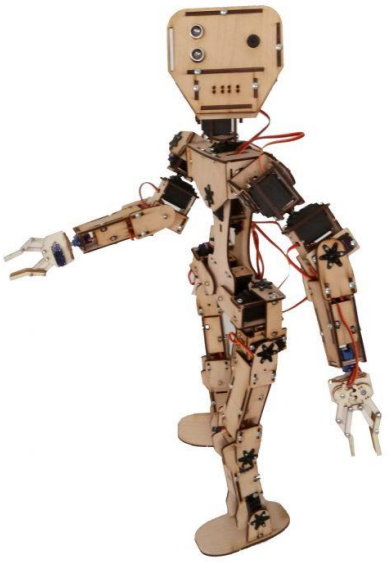
Simulácia v CoppeliaSim prebieha v krokoch trvajúcich konštantný čas, ktorý sa dá na začiatku simulácie nastaviť. V jednom kroku simulácie sa najprv spustí main script, ktorý vykoná všetko potrebné a spustí všetky child scripts, následne sa vykreslí celá scéna simulácie v aktuálnom kroku. Vykresľovanie scény sa dá užívateľom vypnúť a simulácia bude bežať iba na pozadí, čím bude0 rýchlejšia. Po skončení jedného kroku simulácie nasleduje hneď ďalší krok simulácie, dokým nebude simulácia zastavená nejakým scriptom alebo manuálne používateľom. Simulácia nebeží v reálnom čase, ale v simulačnom čase, ktorý je podstatne rýchlejší ako skutočný čas. Čas simulácie sa dá užívateľom zrýchliť alebo spomaliť aj v priebehu behu simulácie. Užívateľ vie takisto nastaviť, aby čas simulácie zodpovedal reálnemu času.

Výpočet fyziky v simulácii zabezpečuje physics engine. CoppeliaSim podporuje päť physics engines, z ktorých si užívateľ môže vybrať. Sú nimi *Bullet physics library, Open Dynamics Engine* (ODE), *MuJoCo, Vortex Studio* a *Newton Dynamics*. Výpočet fyziky v simulácii je veľmi komplexný a z toho dôvodu každý z physics engines bude dávať čiastočne rozdielne výsledky. Preto je pre simuláciu dôležité si vybrať správny physics engine, ktorý najviac vyhovuje danej simulovanej situácii a ktorý podporuje funkcionality, ktoré chceme v simulácii využiť. Novovytvorená scéna má prednastavený physics engine Bullet 2.78.

1.2 Humanoidný robot Lilli

Robot Lilli je humanoidný robot, teda ide o robota, ktorý sa snaží svojim výzorom napodobniť vzhľad človeka. Robota navrhol Per R. Ø. Salkowitsch a prvýkrát prototyp robota Lilli ukázal v roku 2018 na Maker Faire, ktorý sa konal vo Viedni. Robot má torzo, hlavu, dve

horné a dve dolné končatiny, ktoré by mu mali slúžiť na vzpriamenú chôdzu. Na výšku má 75 centimetrov a má 25 stupňov voľnosti, ktoré zabezpečujú servá riadené Arduinom, Raspberry Pi alebo iným mikropočítačom. V krku má dve servá slúžiace na otáčanie hlavy, jedno servo má v trupe, ktoré slúži na nakláňania do strán, v každej ruke má šesť serv, z toho tri v ramene a po jednom v lakti, v zápästí a v dlani. V každej nohe má päť serv, z čoho tri v panvovom kĺbe, jedno v kolene a jedno v členku. Vnímanie okolitého prostredia má vďaka stereovideniu, ktoré je umiestnené v hlave robota. Ide o open source framework, čo znamená, že všetky potrebné údaje k robotovi Lilli sú voľne dostupné na stiahnutie, zhotovenie a následné programovanie pre každého. Robota Lilli treba zostaviť z preglejkových dielov vyrezaných na CNC stroji a následne zaskrutkovať k sebe.



Obrázok 2: Humanoidný robot Lilli [10]

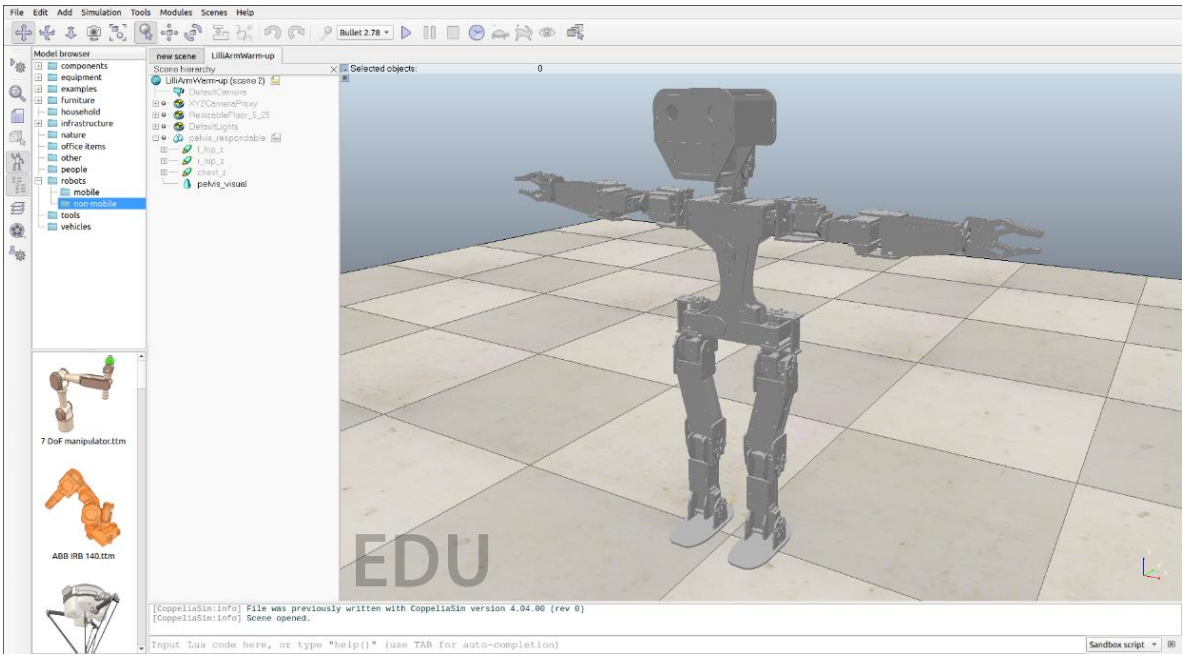
1.2.1 URDF model

URDF je skratka znamenajúca Unified Robotics Description Format. Ide o XML špecifikáciu slúžiacu na zadefinovanie modelu robota, ktorá sa dá následne zobraziť a využívať v robotických simulátoroch ako napríklad v CoppeliaSim alebo Gazebo. V URDF modeli sú zadefinované tvary všetkých objektov robota, ich hmotnosť a ťažisko, vlastnosti kĺbov robota, ako napríklad jeho sila alebo maximálna rýchlosť, ale aj dynamické vlastnosti jeho jednotlivých častí ako napríklad správanie sa objektu pri kolízii s iným objektom.

1.2.2 URDF model robota Lilli

URDF model pre robota Lilli vytvoril G. Halasi v roku 2020 ako súčasť svojej diplomovej práce [2]. V tomto modeli je každý vyrezaný diel z pôvodného robota Lilli

zadefinovaný ako geometrický útvar. Každý z 25 stupňov voľnosti je v modeli reprezentovaný ako revolute joint so svojím vlastným menom. Toto meno vyjadruje, po ktorej osi daný kĺb rotuje, kde os *y* rastie smerom dozadu od robota a klesá smerom dopredu od robota, os *x* rastie smerom doľava od robota a klesá smerom doprava od robota a os *z* rastie smerom od podlahy nahor. Prefix *l* alebo *r* v názve kĺbov určuje, či sa jedná o ľavú končatinu, v prípade prefixu *l*, alebo o pravú končatinu, v prípade prefixu *r*. Tieto prefixy majú iba párové kĺby, čiže ide o kĺby v rukách a v nohách. Základňou celého modelu je torzo robota s názvom *pelvis_respondable*, na ktoré sú napojené všetky ostatné tvary a kĺby. Robot je v počiatočnom stave vtedy, keď všetky jeho kĺby sú na pozícii nula. Robot v počiatočnom stave stojí na oboch nohách a má upažené ruky. Na ovládanie robota používal G. Halasi Java Remote API. Na záver práce vytvoril niekoľko scén na testovanie pohybov robota, konkrétne na testovanie pohybov rúk a nôh v súboroch LilliArmWarm-up.ttt a LilliLegWarm-up.ttt.



Obrázok 3: URDF model robota Lilli v CoppeliaSim v počiatočnom stave

1.3 Genetický algoritmus

Genetický algoritmus, ďalej spomínaný už iba ako GA, patrí do triedy evolučných algoritmov. Je to metóda na riešenie optimalizačných problémov, ktorá je založená na procese prirodzeného výberu známeho z evolúcie v prírode. GA v tejto kapitole opisujeme tak, ako ho opísali Russell a Norvig vo svojej knihe Artificial Intelligence: A Modern Approach [3].

1.3.1 Jedinec

Genetický algoritmus musí mať jednotne definovaných jedincov. Každý jedinec je

reprezentovaný postupnosťou symbolov nad konečnou abecedou, v ktorých je zakódovaná jeho genetická informácia.

1.3.2 Účelová funkcia

Účelová funkcia slúži na ohodnotenie jedincov z populácie. Ide o funkciu, ktorá dostane na vstupe jedinca, teda preddefinovanú postupnosť znakov, a vráti na výstupe reálne číslo, ktoré zodpovedá kvalite jedinca na riešenie daného problému.

V genetickom algoritme sa pokúšame fitness buď minimalizovať, teda nájsť jedinca s čo najnižšou fitness, alebo sa snažíme fitness maximalizovať, teda nájsť jedinca s čo najväčšou fitness.

1.3.3 Začiatok GA

Na začiatku genetického algoritmu sa vygeneruje náhodná začiatočná množina jedincov, ktorá sa nazýva populácia. Každý jedinec v populácii je teda náhodne vygenerovaná postupnosť symbolov z danej konečnej abecedy a následne sa ohodnotí pomocou účelovej funkcie. Je mu teda priradené reálne číslo (fitness), ktoré určuje jeho kvalitu na riešenie daného optimalizačného problému.

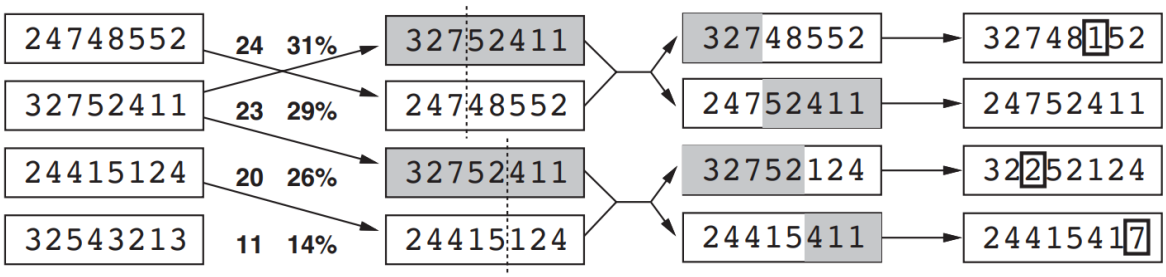
1.3.4 Ďalšia generácia

Ďalšia generácia populácie sa vytvorí z predchádzajúcej generácie na základe hodnoty fitness jedincov. Postupne sa vyberajú dvaja náhodní jedinci z populácie, pričom pravdepodobnosť, s akou je jedinec vybraný, je priamoúmerná jeho fitness (stať 1.3.7.4) v prípade, že sa snažíme fitness maximalizovať. Platí, že jeden jedinec z pôvodnej populácie môže byť vybraný ľubovoľne veľakrát. Títo dvaja jedinci následne prejdú krížením a mutáciou za účelom vytvorenia nového jedinca. Takto sa vyberajú jedinci z pôvodnej populácie, dokým nie je vytvorený rovnaký počet jedincov do novej populácie ako bolo v predchádzajúcej populácii. Nová populácia sa opäť ohodnotí pomocou účelovej funkcie a znovu sa opakuje proces tvorby novej generácie.

1.3.5 Kríženie a mutácia

Po výbere dvoch jedincov z populácie, z ktorých sa bude vytvárať nový jedinec do ďalšej generácie, nastáva kríženie týchto jedincov. Kríženie prebieha tak, že sa zvolí miesto v zozname symbolov reprezentujúcich jedinca, ktoré bude tvoriť pomyselnú hranicu. Následne nový jedinec bude vytvorený tak, že časť svojej genetickej informácie, teda zoznam symbolov,

dostane od jedného z vybraných jedincov až po zvolené miesto predelu a druhú časť dostane od druhého zvoleného jedinca. Takto vytvorený jedinec teda má jednu časť svojho zoznamu symbolov od jedného rodiča a druhú časť od druhého rodiča. Následne takto vytvorený nový jedinec prejde mutáciou. To znamená, že každý symbol v jeho postupnosti symbolov bude s veľmi malou pravdepodobnosťou náhodne zmenený za iný symbol z danej abecedy.



Obrázok 4: Kríženie a mutácia [3]

1.3.6 Koniec GA

Ukončenie genetického algoritmu zvykne byť jedným z dvoch spôsobov alebo ich kombináciou.

Prvý spôsob je, že sa určí maximálny počet generácií, ktorými chceme prejsť a akonáhle prídeme na generáciu s poradovým číslom zodpovedajúcim maximálnemu počtu generácií, tak po nej už nebude nasledovať ďalšia generácia. Algoritmus vráti najlepšieho jedinca danej poslednej generácie, ktorý je výsledným hľadaným jedincom daného optimalizačného problému.

Druhý spôsob ukončenia genetického algoritmu je za podmienky nájdenia hľadaného výsledného jedinca dostatočnej kvality. To znamená, že sa určí hodnota cieľovej fitness, akú chceme dosiahnuť a akonáhle generácia obsahuje jedinca s aspoň takou fitness, ako je cieľová fitness, tak algoritmus skončí a vráti na výstupe daného jedinca s požadovanou fitness.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new_population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Obrázok 5: Pseudokód genetického algoritmu [3]

1.3.7 Heuristiky

V GA sa využívajú mnohé heuristiky, ktoré slúžia na vylepšenie daného GA. Pri zvolení správnych heuristík môžeme docieliť skrátenie doby, ktorú náš GA bude potrebovať, dokým nájde výsledného hľadaného jedinca. Ďalej spomenieme niektoré heuristiky.

1.3.7.1 K-point crossover

K-point crossover, tak ako ho spomína David B. Fogel vo svojej knihe [4], na rozdiel od kríženia spomínaného v predchádzajúcej kapitole (stať 1.3.5), ktoré sa nazýva 1-point crossover, je kríženie jedincov, v ktorom sa namiesto jedného miesta predelu v zozname symbolov jedinca vyberie niekoľko miest. Premenná *k* určuje počet miest, ktoré rozdelia zoznam symbolov jedinca na niekoľko úsekov. Výsledný jedinec dostane striedavo každú takto vytvorenú časť od iného z daných dvoch vybraných jedincov.

1.3.7.2 Uniform crossover

Uniform crossover, podľa David B. Fogela [4], je kríženie jedincov, pri ktorom každý jeden symbol výsledného jedinca je vybraný náhodne od jedného z jeho dvoch vybraných predkov. Pre každý symbol je šanca výberu rodiča, od ktorého zdedí daný symbol, rovnaká.

1.3.7.3 BLX-alpha crossover (Blend crossover)

Ide o algoritmus kríženia prvýkrát publikovaný v práci Real-Coded Genetic Algorithms and Interval-Schemata od autorov L. J. Eshelman, J. D. Schaffer [5]. Algoritmus hovorí, že po výbere rodičov označených G1 a G2, treba vybrať náhodné číslo gamma z intervalu [-alpha, 1 + alpha], kde alpha je nejaké malé číslo, napríklad 0,5. Následne hodnota potomka G sa vypočíta ako G = gamma * G1 + (1 - gamma) * G2.

1.3.7.4 Tournament selection

Slúži na výber jedincov z danej populácie, ktorí budú vytvárať jedincov do novej populácie. Výber jedincov spomínaný v predchádzajúcej kapitole (stať 1.3.4) sa nazýva Roulette wheel selection a jeho najväčšou nevýhodou je, že ak jedinci v populácii majú veľmi podobnú fitness, tak výsledný výber jedincov je takmer úplne náhodný. V Tournament selection sa vždy náhodne vyberú dvaja jedinci z populácie a do fázy kríženia sa dostane jedinec s lepšou fitness, teda väčšou pri maximalizovaní fitness a menšou pri minimalizovaní fitness. Tento proces sa zopakuje aj pri výbere druhého rodiča do fázy kríženia.

1.3.7.5 Elitizmus

Elitizmus rieši problém, ktorý nastáva pri výbere jedincov na kríženie, keďže najlepší jedinci nemusia byť do procesu kríženia vybraní ani raz. Pri používaní tejto heuristiky sa do ďalšej generácie odloží najlepší jedinec alebo niekoľko najlepších jedincov a tí postupujú do ďalšej generácie nezmenení. Tým sa zaručí, že najlepšia dosiahnutá fitness nebude v ďalšej generácii nikdy horšia ako bola v predchádzajúcej.

1.4 Súvisiaci výskum

V práci Humanoid Robot Walking Optimization using Genetic Algorithms od autorov Villela a Colombini [6] sa jej autori snažili naučiť chodiť humanoidného robota NAO od spoločnosti SoftBank Robotics [7] v simulátore V-REP, čo je predchodcom CoppeliaSim, za pomoci genetického algoritmu. V tejto práci bol každý kĺb robota reprezentovaný funkciou, ktorá zobrazovala polohu kĺbu na čas simulácie. Funkcia každého kĺbu bola optimalizovaná pomocou genetického algoritmu. Účelová funkcia hodnotila jedincov na základe prejdenej vzdialenosti za čo najnižší čas. Jedinci taktiež dostávali bonus k fitness, ak pri chôdzi nezabáčali do strán a chôdza bola priama. Výber jedincov do ďalšej generácie robili pomocou Roulette Wheel Selection. Kríženie robili pomocou jemne upravenej verzie BLX-alpha crossover (stať 1.3.7.3). Pravdepodobnosť mutácie mali nastavenú na 5 %. Výsledný jedinec vedel kráčať rýchlosťou 54 cm/s, čo v pomere na jeho výšku 58 cm znamená, že vedel prejsť

93 % svojej výšky za sekundu. Dokopy vedel prejsť až 200 metrov bez pádu. Následne experiment pokračoval s pokusom naučiť robota chodiť v priestore s kladným a záporným prevýšením za pomoci toho istého algoritmu. Tento pokus bol takisto úspešný avšak výsledný jedinec dosiahol o niečo nižšiu rýchlosť chôdze.

2 Cieľ práce a návrh riešenia

Za cieľ práce sme si dali dokázať hypotézu, že humanoidný robot Lilli je spôsobilý na chôdzu po dvoch nohách. Táto hypotéza nebola vyriešená ešte v žiadnej inej práci, ale predpokladá sa, že humanoidný robot Lilli by mal byť spôsobilý na chôdzu. Práca nadväzuje na prácu G. Halasiho [2], ktorý vytvoril URDF model robota Lilli a na prácu T. Koseca [8], ten vo svojej práci pripravil podmienky pre implementáciu pohybu a postavil pomocnú konštrukciu, ktorá má robotovi Lilli robiť oporu pri chôdzi.

2.1 Výber prostriedkov

Experimenty budeme uskutočňovať v prostredí robotického simulátora CoppeliaSim, v ktorom využijeme URDF model robota Lilli zostrojeného G. Halasim (stať 1.2.2). Budeme využívať prednastavený výpočtový engine Bullet 2.78. Ten ako uvádzajú vo svojej práci I. Bzhikhatlov a S. Perepelkina [9] je vhodný na simulovanie chôdze humanoidného robota. Spôsob, akým má robot Lilli chodiť, sa pokúsime nájsť pomocou genetického algoritmu. Keďže očakávame časovú náročnosť behu genetického algoritmu, budeme celý genetický algoritmus programovať priamo v prostredí CoppeliaSim v jazyku Lua. V prostredí CoppeliaSim sa dá písať kód aj v Pythone, avšak jeho nevýhoda je, že je pomalší od kódu napísanom v Lua. Aj z tohto dôvodu sme sa rozhodli pre jazyk Lua. Kód nášho programu s genetickým algoritmom budeme písať ako embedded simulation non-threaded script. Lua script pripojíme k jednému z automaticky generovaných objektov v scéne. Nemôžeme script pripojiť k samotnému modelu robota Lilli, keďže budeme chcieť model odstraňovať zo scény počas simulácie a tým by sme tak prišli o vykonávanie scriptu.

2.2 Reprezentácia chôdze

Chôdzu budeme reprezentovať ako cyklickú postupnosť polôh kĺbov robota v čase. Každá poloha je reprezentovaná n-prvkovým polom, kde n zodpovedá počtu používaných kĺbov pri chôdzi. V tomto poli prislúcha každý prvok poľa jednému kĺbu a jeho hodnota určuje polohu kĺbu v stupňoch, v ktorej sa má daný kĺb nachádzať. Postupnosť takto reprezentovaných polôh bude predstavovať chôdzu, kde po uplynutí daného časového intervalu prejde simulovaný robot z jednej polohy do nasledujúcej. Celá postupnosť bude cyklická, teda po vykonaní poslednej polohy bude nasledovať zase prvá poloha. Jeden cyklus postupnosti

polôh bude predstavovať dva kroky a to jeden krok ľavou nohou a jeden krok pravou nohou, ktoré sa budú cyklicky opakovať. Súčasťou chôdze nie je stabilné zastavenie robota na konci chôdze.

2.3 Priaznivý výsledok

Za priaznivý výsledok budeme považovať takú postupnosť polôh, pomocou ktorej bude vedieť simulovaný robot prejsť aspoň tri kroky, pričom bude striedať nohy, teda bude vedieť vykonať viac ako jeden celý cyklus postupnosti polôh, a pri tom nespadnúť. Jedinec bude taktiež robiť efektívne kroky, a teda každým krokom sa posunie dopredu. Nezaujíma nás rýchlosť robota počas chôdze, ale prejdená vzdialenosť.

2.4 Implementácia genetického algoritmu

2.4.1 Reprezentácia jedinca

Jedinec bude reprezentovaný ako dvojica. Prvým prvkom dvojice je číslo, ktoré reprezentuje fitness jedinca a druhým prvkom dvojice je dvojrozmerné pole reprezentujúce cyklickú postupnosť polôh kĺbov robota, ktorú budeme nazývať gén. Kĺbov, ktorými bude môcť robot hýbať počas chôdze, bude 9. Sú nimi kĺby *l_hip_x* a *r_hip_x*, ktoré zabezpečujú pohyb celej nohy dopredu a dozadu, kĺby *l_knee_x* a *r_knee_x*, ktoré zabezpečujú pohyb predkolenia dopredu a dozadu, *l_ankle_x* a *r_ankle_x* zabezpečujúce pohyb členka dopredu a dozadu, *l_shoulder_y* a *r_shoulder_y* zabezpečujúce zase pohyb rúk od tela smerom do upaženia a kĺb *chest_z*, ktorý zabezpečuje naklonenie hornej časti trupu do strán. Prvých šesť kĺbov slúži na pohyb dopredu, posledné tri kĺby slúžia na udržiavanie stability počas chôdze.

2.4.2 Genetický algoritmus

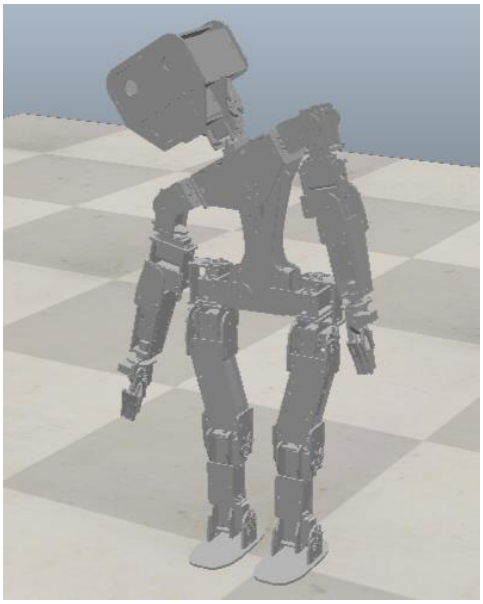
V sysCall_init funkcii sa najprv vytvorí náhodná počiatočná populácia. Následne v sysCall_actuation funkcii prebieha zvyšok genetického algoritmu, teda ohodnocovanie jedincov a tvorba novej generácie z predchádzajúcej generácie.

2.4.2.1 Ohodnocovanie jedincov

Na ohodnotenie jedinca danou účelovou funkciou je potrebné vykonať jeho postupnosť polôh do momentu, dokým simulovaný robot nespadne.

Na začiatku sa načíta do scény model robota Lilli, ktorý je jemne upravený, aby

nezačínal zo svojej počiatočnej pozície, zadefinovanej G. Halasim vo svojej práci, ale začínal s pripaženými rukami a nakloneným torzom na pravú stranu, aby bol navážený na pravú nohu a mohol ľavou nohou začať chôdzu.



Obrázok 6: Počiatočná pozícia robota

Počas behu simulácie sa v každom kroku simulácie vykoná funkcia sysCall_actuation. Kvôli tomu sme celú funkciu spomalili a kód sa vykoná iba raz za nami definovaný krátky interval, ktorý je nastavený na 0,1, čo znamená, že sa funkcia sysCall_actuation vykoná iba v každom druhom kroku simulácie. Pri načítaní robota sa interval navýši, aby mal robot čas sa nastaviť z počiatočnej pozície do nami pripravenej naklonenej štartovacej pozície. Potom, počas testovania jedinca, sa interval zníži a jedna poloha sa vykoná zakaždým, keď uplynie interval, dokým simulovaný robot nespadne.

Robot je považovaný za padnutého, keď sa *z*-ová súradnica jeho torza nachádza pod hranicou 20 cm, teda jeho torzo klesne o viac ako polovicu od pôvodnej výšky stredu torza, ktorá sa nachádza vo výške 45 centimetrov.

Každý jedinec v populácii po tom, čo sa načíta do scény nanovo model robota Lilli, sa najprv načíta do globálnych premenných, teda jeho gén sa načíta do globálnej premennej. Potom tento jedinec vykonáva pravidelne po skončení intervalu ďalšiu polohu zo svojho génu, dokým nespadne. Keď spadne, zapamätá sa jeho konečná *y*-ová súradnica, ktorá bude určovať jeho fitness. Budeme používať hodnotu vynásobenú -1, keďže inak by sme dostávali záporné hodnoty kvôli tomu, že *y*-ová súradnica klesá smerom dopredu od robota a v našom GA chceme maximalizovať fitness. Na konci testovania chôdze jedinca sa aktuálny model robota Lilli vymaže zo scény, aby sa mohol pri ďalšom jedincovi načítať model v počiatočnej polohe.

2.4.2.2 Ďalšia generácia

Po tom, čo sa ohodnotí každý jedinec v populácii a zapamätá sa jeho fitness, vytvorí sa nová populácia z pôvodnej populácie krížením a mutáciou. Výber jedincov na kríženie a mutáciu budeme robiť pomocou tournament selection, kde každý jedinec môže byť vybraný aj viackrát. Na kríženie jedincov budeme používať k-point crossover, kde *k* je nastaviteľný parameter a skúsime porovnať efektívnosť algoritmu pre náš problém pri 1-point a 2-point crossoveri. Kríženie bude prebiehať na každej polohe samostatne, čiže pri výbere dvoch jedincov sa každá n-tá poloha prvého jedinca skríži s n-tou polohou druhého jedinca podľa práve zvoleného k-point crossoveru. Pravdepodobnosť na to, že sa na danej polohe vykoná kríženie bude 50 %. Na konci kríženia vzniknú dvaja noví jedinci tak, ako je opísané v kapitole Kríženie a mutácia (stať 1.3.5). Každý jedinec následne prejde s malou pravdepodobnosťou mutáciou, ktorá zmení s malou pravdepodobnosťou ktorúkoľvek hodnotu uhla pre náhodne zvolený kĺb v géne jedinca o malý uhol náhodným smerom. Takto vytvorená nová generácia opäť prejde ohodnotením všetkých jej jedincov.

2.4.2.3 Výstupy programu

Program bude na konci každej generácie zapisovať do samostatných textových súborov najlepšieho jedinca, fitness akú tento jedinec dosiahol a priemernú fitness celej populácie. Tieto súbory budeme otvárať vo funkcii sysCall_init. Po každej generácii flush-neme údaje do súborov, aby sme mohli sledovať práve najlepšiu fitness GA počas jeho behu. Zapisované údaje sú nám potrebné na to, aby sme vedeli zistiť efektívnosť algoritmu, prípadne porovnať efektívnosť programu pri spustení s rôznymi parametrami.

2.4.2.4 Ukončenie genetického algoritmu

Genetický algoritmus nebude mať podmienku, pri ktorej by mal skončiť, ale bude vypínaný ručne, buď po dosiahnutí priaznivého výsledku alebo keď prestane fitness najlepšieho jedinca narastať.

2.5 Ovládanie robota v simulácii

Robot v simulátore sa ovláda pomocou handles. Každému objektu je priradený jeho vlastný handle, ktorý je reprezentovaný prirodzeným číslom. Na zistenie hodnoty handle pre daný objekt využijeme príkaz sim.getObject(string objectPath), ktorý po zadaní relatívnej cesty k objektu, vráti handle daného objektu.

Na ovládanie kĺbov robota využijeme príkaz sim.setJointTargetPosition(int objectHandle,float targetPosition), ktorý posunie kĺb daného handle na zadanú pozíciu.

Parameter targetPosition určuje pozíciu kĺbu v radiánoch. Keďže hodnoty v géne jedinca sú v uhloch, budeme vždy pri používaní tohto príkazu robiť konverziu z uhlov na radiány. Na rozdiel od príkazu sim.setJointPosition, príkaz sim.setJointTargetPosition ráta s parametrami kĺbu, ako je napríklad sila kĺbu a maximálna rýchlosť.

Pri načítaní modelu robota Lilli v simulácii na začiatku ohodnocovania jedinca si uložíme handles všetkých kĺbov do poľa. Hodnoty handles z poľa budeme následne využívať pri pohybe kĺbov počas simulácie. Po tom, čo sa vymaže model robota zo scény na konci ohodnocovania a znovu sa načíta pri ohodnocovaní ďalšieho jedinca, treba hodnoty handles načítať do poľa nanovo, keďže novopridanému modelu priradí softvér nové hodnoty handles.

2.6 Príprava pred prvým experimentom

Ešte pred tým, než sme začali s experimentálnou časťou práce, sme sa najprv zamerali na to vytvoriť jedinca, ktorý by imitoval primitívnu chôdzu.

Tento jedinec pozostával zo štyroch polôh. Prvá poloha dostane robota do pozície, kedy je navážený na pravú nohu, keďže z tejto polohy začína aj náš vytvorený model. Druhá pozícia urobí ľavou nohou malý pohyb dopredu. Tretia pozícia sa naváži na ľavú nohu. A posledná, štvrtá, pozícia spraví pravou nohou malý pohyb dopredu.

Vytvorili sme podľa tohto návrhu jedinca, ktorého postupnosť polôh sme nastavili tak, aby zodpovedali jednotlivým polohám návrhu. Tento jedinec nebol schopný chôdze, ale vedel na mieste prekračovať z jednej nohy na druhú. V žiadnom momente pohybu nebol jedinec v stave, že by dvihol jednu nohu zo zeme, teda obe nohy mali stále kontakt so zemou a iba ich posúval po podlahe.

Pri spúšťaní tohto návrhu sme zistili, že výpočtový engine Bullet 2.78 nie je pre náš problém vyhovujúci. V tomto výpočtovom engine bol robot schopný posúvať nohy dopredu po podlahe bez toho, aby bol trením šľapy o podlahu spomaľovaný. V takomto prostredí by totiž simulovaný robot nemusel dvíhať nohy zo zeme, a teda nezodpovedal predstave o chôdzi v reálnom svete. Nastavili sme preto výpočtový engine na Bullet 2.83, ktorý vytvára trenie šľapy o podlahu oveľa väčšie, a tak bude musieť simulovaný robot dvihnúť nohu na to, aby spravil krok. Tým pádom tento výpočtový engine simuluje fyzikálne vlastnosti simulácie realistickejšie pre náš problém.

Pohyby jedinca aj napriek zmene na iný engine nepôsobili v porovnaní s reálnym robotom Lilli realisticky, pretože boli príliš rýchle aj napriek tomu, že model robota

v simulátore má nastavenú maximálnu rýchlosť kĺbu zodpovedajúcu rýchlosti servo motorov v reálnom robotovi. Vytvorili sme preto funkciu *multiply_gene*, ktorá pridá do pohybu jedinca medzikrok medzi každé dve polohy, pričom pridaná nová poloha je presne v strede medzi dvomi polohami, medzi ktoré bola vložená. Funkcia má jeden parameter *multiply*, ktorý určuje, koľkokrát sa má medzi každé dve polohy vložiť nová medzipoloha. Keď pomenujeme pôvodný počet polôh vrátane počiatočnej a konečnej polohy ako *n* a hodnotu parametra *multiply* ako *m*, tak výsledný počet polôh sa bude rovnať $(n-1) \times 2^m + 1$. Tieto pridané polohy slúžia na to, aby pohyby robota v simulácii boli realistickejšie v porovnaní so skutočným modelom robota.

Simulovaný robot potreboval okolo 60 polôh na to, aby jeho pohyby pôsobili reálne v porovnaní so skutočným robotom Lilli. Rozhodli sme sa, že dĺžku postupnosti polôh jedinca nastavíme na 16 a budeme spúšťať $multiply = 2$. Dĺžku génu jedinca sme navýšili na 16, aby pohyb simulovaného robota nebol priamočiary, ako tomu bolo vo verzii s dĺžkou 4.

Tento návrh jedinca budeme využívať pri vytvorení počiatočnej populácie, kde každý jedinec bude vytvorený ako kópia tohto jedinca, pričom každý uhol v jeho géne bude zmenený o nami definovanú hodnotu. Tým sa pokúsime dosiahnuť to, aby populácia nezačínala s úplne náhodnými jedincami, ale s jedincami podobnými nášmu návrhu chodiaceho jedinca.

3 Experimentálna časť práce

V tejto časti opíšeme všetky významné experimenty, ktoré sme vyskúšali. O každom experimente povieme, aký bol predpokladaný výsledok pred experimentom, ako sme experiment realizovali a aké parametre sme pre daný experiment zvolili, aký bol skutočný výsledok experimentu a aký záver sme si z daného experimentu zobrali do ďalších pokusov. Okrem spomínaných experimentov sme uskutočnili aj mnohé iné, avšak tieto neúspešné a chybné experimenty v tejto kapitole spomínať nebudeme.

3.1 Úvodné experimenty

Pokusy sme začali sériou menších pokusov, aby sme overili korektnosť nami naprogramovaného genetického algoritmu. Taktiež nám tieto pokusy slúžili na lepšie pochopenie celej problematiky a úpravu parametrov a kódu pri väčších nasledujúcich pokusoch.

3.1.1 Predpoklad

Od úvodných experimentov očakávame, že nájdeme jedinca, ktorý bude schopný spraviť aspoň jeden krok. Taktiež očakávame, že lepšie pochopíme komplexnosť daného problému. Predpokladáme aj to, že nájdeme nedostatky nášho kódu, s ktorými sme doteraz nepredpokladali, aby sme ich vedeli opraviť v nasledujúcich experimentoch. Očakávame aj, že nami naprogramovaný genetický algoritmus bude korektný a fitness jedinca bude aj pre takéto menšie experimenty narastať.

3.1.2 Realizácia

Experimenty budeme realizovať na lokálnom počítači, spustený bude iba jeden proces súčasne. Budeme zobrazovať beh simulácie na obrazovke a sledovať evolúciu jedincov, aby sme vedeli odpozorovať prípadné nekorektné správanie jedincov a opraviť ho v neskorších pokusoch.

Budeme mať veľmi malú populáciu 32 jedincov. Jedinci budú mať gén dĺžky 16 a bude znásobený s $multiply = 2$. Využijeme 2-point crossover na každej polohe vykonávaný samostatne. Šanca na to, že jedinec bude zmenený mutáciou, bude 10 % a šanca pre každý uhol, že bude zmenený so silou $\pm 5°$ bude 10 %.

Fitness budeme rátať ako prejdenú vzdialenosť stredu torza robota od začiatku ohodnocovania jedinca až po moment, kedy jedinec padne, teda z-ová súradnica stredu torza klesne pod hranicu 20 centimetrov.

3.1.3 Výsledok

Najlepší jedinci prvotných spustení programu sa po desiatkach generácií začali správať tak, že sa odrážali oboma nohami čo najviac dopredu a spadli. Jedinci sa nepokúšali o chôdzu.

3.1.4 Záver

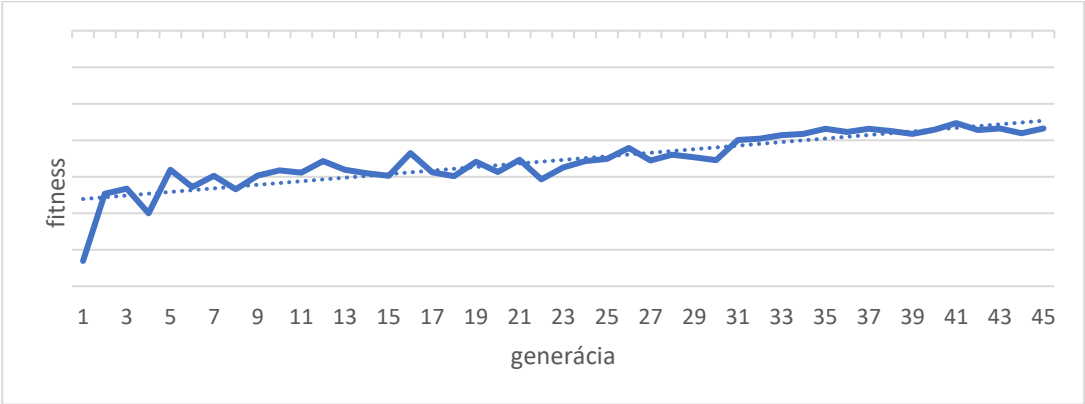
Dôvodom bola chybná účelová funkcia, ktorá uprednostňovala takéto správanie oproti čiastočnej chôdzi. Aj napriek tomu, že by chodiaci jedinec vedel prejsť väčšiu vzdialenosť ako jedinec skákajúci dopredu, nemá za takýchto podmienok šancu sa evolučne zlepšiť, keďže nedosahuje takú fitness ako skákajúci jedinci.

Experimenty preto budeme opakovať, avšak zmeníme podmienky tak, že jedinec bude považovaný za spadnutého v momente, keď stred jeho torza klesne pod výšku 40 centimetrov. Taktiež zmeníme počítanie fitness tak, že jedincova výsledná fitness bude prejdená vzdialenosť od načítania modelu robota po poslednú vykonanú polohu, ktorá neviedla k pádu jedinca. Tým zabezpečíme, že skákajúci jedinci budú dosahovať nižšiu fitness ako jedinci, ktorí sa pokúšajú o chôdzu.

3.1.5 Výsledok upraveného experimentu

Najlepší jedinec týchto experimentov vedel robiť minimálne pohyby na mieste pripomínajúce prvotné štádiá chôdze. Fitness najlepšieho jedinca naprieč generáciami narastala.

Priložený je graf jedného behu programu, ktorý zobrazuje fitness najlepšieho jedinca v každej generácii. Program bežal 45 generácií.



Obrázok 7: Fitness najlepšieho jedinca z úvodných experimentov v každej generácii

3.1.6 Záver upraveného experimentu

Nepodarilo sa nám nájsť jedinca, ktorý by vedel spraviť aspoň jeden stabilný krok badateľnej dĺžky, avšak aktuálni najlepší jedinci mali predpoklad na to chodiť, kebyže prejde algoritmus viacerými generáciami. Opravili sme kód a účelovú funkciu tak, aby bola lepšia pre náš problém. Fitness jedinca naprieč generáciami rástla, čím sme potvrdili korektnosť algoritmu.

3.2 Chôdza robota ako celok

Na základe predchádzajúcich pokusov sme sa rozhodli, že budeme spúšťať simuláciu na vzdialenom zariadení, budeme spúšťať viacero procesov naraz a s väčšou populáciou, aby sme efektívnejšie prehľadávali možné riešenia. Taktiež už nebudeme spúšťať simuláciu aj s jej vizualizáciou, keďže tá iba spomaľuje výpočty. Vyskúšame do simulácie pridať nové prvky. Skúsime pridať do chôdze prefix, ktorý by mal zjednodušiť hľadanie riešenia. Taktiež skúsime zmeniť gén jedinca na symetrický a tým zjednodušiť zložitosť problému.

3.2.1 Predpoklad

Od experimentu očakávame, že nájdeme jedinca, ktorý bude schopný chôdze pozostávajúcej z aspoň dvoch krokov. Očakávame, že sa nám podarí zistiť, koľko polôh potrebuje jedinec na spravenie kroku, aby sme v neskorších spusteniach programu mohli zredukovať počet polôh v postupnosti polôh jedinca. Pridáme nové parametre prefix a symetriu do chôdze, porovnáme ich efektívnosť a očakávame, že verzie programu s týmito prvkami budú dosahovať lepšie výsledky. Skúsime rôzne hodnoty pre *multiply* a očakávame, že zistíme, aký vplyv má obmena tohto parametra.

3.2.2 Realizácia

Experiment budeme realizovať na vzdialenom počítači, na ktorý sa pripojíme pomocou ssh protokolu. Na počítači bude bežať 10 procesov súčasne. Spravíme päť rôznych verzií programu, každý z nich bude mať obmenené parametre a bude spustený na dvoch procesoch.

Procesy budeme spúšťať pomocou príkazu xvfb-run, ktorý zabezpečí, že náš program bude bežať vo virtuálnom prostredí. Bez príkazu xvfb-run by sa po zrušení ssh spojenia zastavila simulácia. Budeme ho spúšťať s flagom -a, ktorý zabezpečí, že sa vždy nájde voľný server, na ktorom sa simulácia spustí. Bez neho by sme mohli spustiť iba jeden proces pomocou xvfb-run súčasne. Samotný coppeliaSim.sh, pomocou ktorého sa spúšťa CoppeliaSim, budeme spúšťať s flagom -h, ktorý otvorí náš program v headless móde, teda bez vizualizácie, a s flagom -s, ktorý ihneď spustí simuláciu pri zapnutí programu. Taktiež musíme pridať .ttt súbor, ktorý chceme v simulátore otvoriť. Príkaz spúšťame na pozadí pomocou &.

Využijeme program z predchádzajúceho experimentu. Navýšime počet jedincov na 300. Účelová funkcia bude pozostávať z prejdenej vzdialenosti od začiatku ohodnocovania až po poslednú polohu, ktorá neviedla k pádu robota. Taktiež bude jedinec získavať fitness, keď jeho chôdza bude smerovať rovno, teda sa nevychýli jeho *x*-ová súradnica z počiatočnej pozície o viac ako $\pm 20cm$. V prípade vybočenia z tohto intervalu bude jedinec fitness strácať. Využijeme 2-point crossover vykonaný na každej polohe samostatne. Šanca na to, že bude na danej polohe vykonaný, bude 50 %. Šanca na mutáciu jedinca bude 10 % a takémuto jedincovi bude každý uhol s pravdepodobnosťou 10 % zmenený o $\pm 5°$.

Do niektorých verzií programu sme pridali do génu jedinca prefix. To znamená, že gén jedinca s prefixom začínal postupnosťou štyroch polôh, ktoré sa za celé testovanie jedinca vykonajú iba raz a to na začiatku. Po týchto štyroch polohách nasledujú zostávajúce polohy, ktoré sa budú cyklicky opakovať. Týmto sme chceli dať jedincovi možnosť sa nastaviť do ľubovoľnej polohy pred tým, než začne s opakovaním tých istých polôh cyklicky.

Ďalej sme do niektorých verzií programu pridali symetricky dopočítanú druhú polovicu génu jedinca, teda jedinec evolúciou menil iba prvých 8 polôh a ostatných 8 bolo symetricky dorátaných. Symetricky dorátané polohy sme dostali tak, že pri párových kĺboch hodnota, v akej bol ľavý kĺb, sa dala do pravého a naopak, hodnota pravého kĺbu sa dala do ľavého. Hodnota polohy torza sa obrátila na inverznú polohu, teda vynásobila -1. Takto upravená postupnosť sa pridala na koniec pôvodnej postupnosti dĺžky 8.

Spustili sme 5 verzií programu, každý na dvoch procesoch. Procesy sme nechali bežať

500 generácií. Jednotlivé verzie programu sa líšili v parametroch génu jedinca a v hodnote parametra *multiply*. Jednotlivé verzie mali nasledovné parametre:

1. verzia – prefix dĺžky 4, gén dĺžky 16, $multiply = 2$
2. verzia – gén dĺžky 8 symetricky dorátaný do 16, $multiply = 2$
3. verzia – gén dĺžky 16, $multiply = 2$
4. verzia – prefix dĺžky 4, gén dĺžky 16, $multiply = 3$
5. verzia – gén dĺžky 8 symetricky dorátaný do 16, $multiply = 3$

3.2.3 Výsledok

Po skončení behu procesov sme si zobrali z každého procesu jedinca s najlepšou fitness a spustili sme si jeho chôdzu s vizualizáciou, aby sme vedeli zistiť, či sme našli jedinca, ktorý spĺňa naše podmienky pre chodiaceho jedinca. Porovnávali sme výsledky jednotlivých verzií programu. Pri porovnávaní sme si zobrali hodnoty najlepšej a priemernej fitness oboch procesov pre danú verziu a spravili sme priemer týchto hodnôt, aby sme dostali lepšie výsledky.

3.2.3.1 Najlepší jedinci z jednotlivých verzií

Prvá verzia programu našla najlepšieho jedinca, ktorý vedel spraviť tri kroky. Kroky robil príliš rýchlo a štvrtý krok nevedel stihnúť spraviť kvôli rýchlosti svojej chôdze a obmedzenou rýchlosťou servo motorov. Jedinec dosiahol jednu z najlepších fitness naprieč všetkými procesmi, avšak aj napriek tomu by sa nedal považovať za chodiaceho jedinca.

Druhá verzia programu našla jedinca, ktorý vedel spraviť iba jeden krok. Jedinec nebol ničím špeciálny a nedal by sa označiť za chodiaceho jedinca.

Tretia verzia programu našla najlepšieho jedinca, ktorý robil jeden krok so zastavením. Jedinec vedel spraviť niekoľko krokov, avšak krok robil vždy iba jednou nohou a druhú nohu dotiahol po zemi k prvej nohe. Jedinec nedosiahol takú dobrú fitness ako jedinci z iných procesov, pretože sa po každom kroku otáčal do strán o viac ako 90°, a tak jeho najväčšia dosiahnutá celková vzdialenosť bola malá. Efektívnosť chôdze jedinca bola príliš nízka, takže jedinca nebolo možné označiť za chodiaceho.

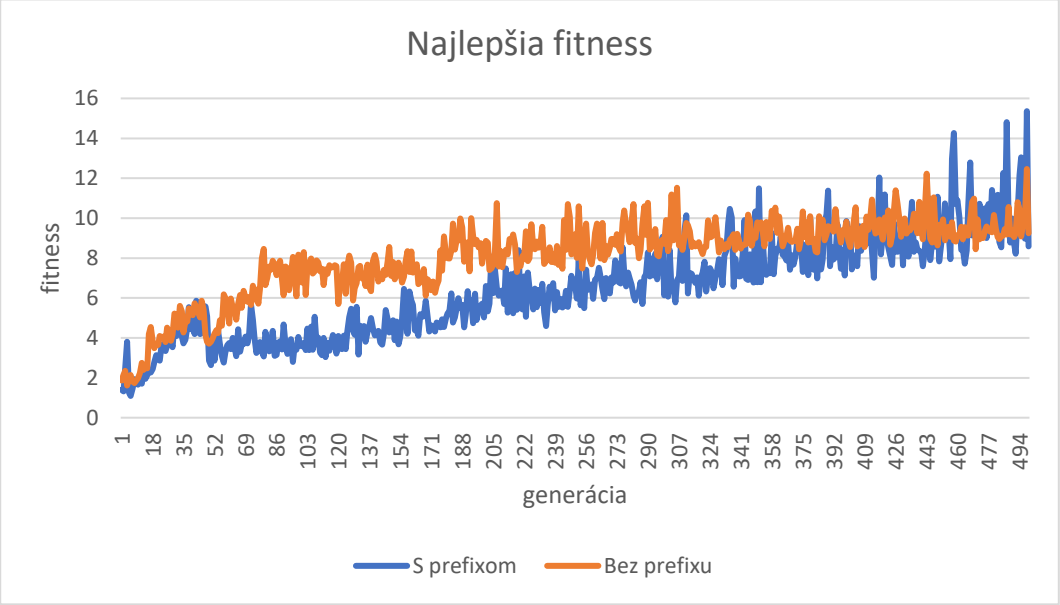
Štvrtá verzia programu našla najlepšieho jedinca, ktorý vedel spraviť dva kroky, každý jednou nohou. Jedinec vedel prejsť všetky polohy vo svojom géne, nevedel avšak spraviť nasledujúci krok, teda prvý krok pri druhom prechádzaní génu. Jedinec sa z tohto dôvodu nedá považovať za chodiaceho.

Piata verzia programu našla najlepšieho jedinca spomedzi všetkých verzií. Jedinec bol podobný najlepšiemu jedincovi z tretej verzie programu, avšak na rozdiel od neho, robil kroky efektívnejšie a neotáčal sa pri tom do strán o viac ako 90°. Jedinec robil kroky iba jednou nohou a druhú nohu dotiahol po zemi k prvej. Na tento pohyb jedinec využíval iba prvé dve polohy zo svojho génu a ostatné polohy nevyužíval, teda počas nich sa nijako badateľne nehýbal. V tomto prípade taktiež nešlo o chodiaceho jedinca podľa našej definície.

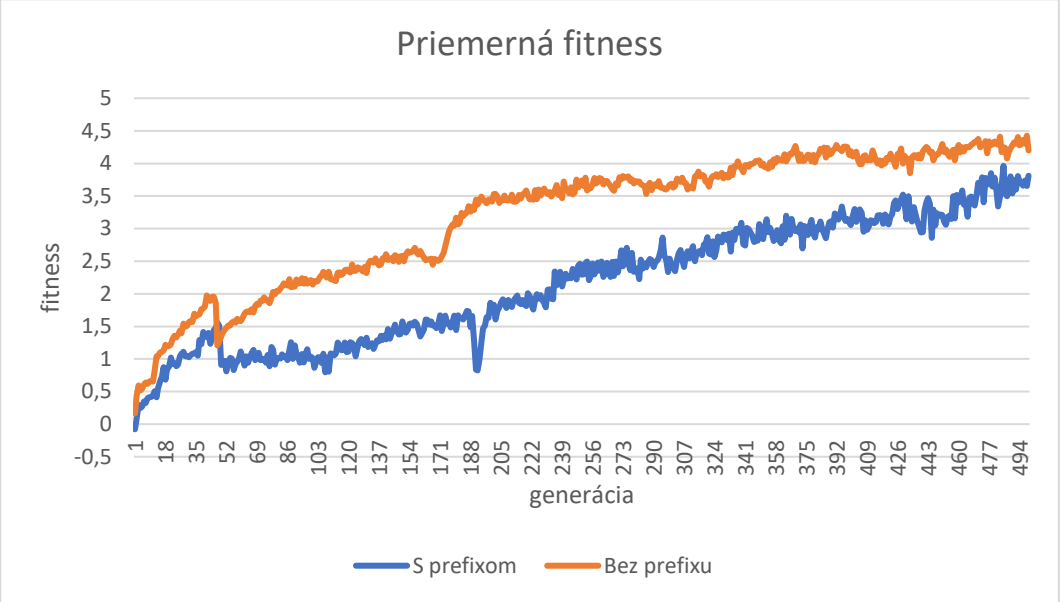
Jedinec z piatej verzie programu sa najviac priblížil našej predstave, pretože polohy vo svojom géne vedel vykonať viac ako raz, vedel spraviť viac ako tri kroky a vedel ich spraviť efektívne. Nevedel avšak robiť kroky druhou nohou.

3.2.3.2 Porovnanie efektívnosti prefixu

Porovnali sme prvú a tretiu verziu programu naprieč všetkými generáciami. Prvá verzia je verzia s prefixom a tretia verzia je verzia bez prefixu. Porovnávali sme najlepšiu fitness a priemernú fitness jedincov jednotlivých verzií v každej generácii. Dostali sme nasledovné výsledky.



Obrázok 8: Najlepšia fitness v priemere v každej generácii pri porovnávaní verzií s prefixom a bez prefixu

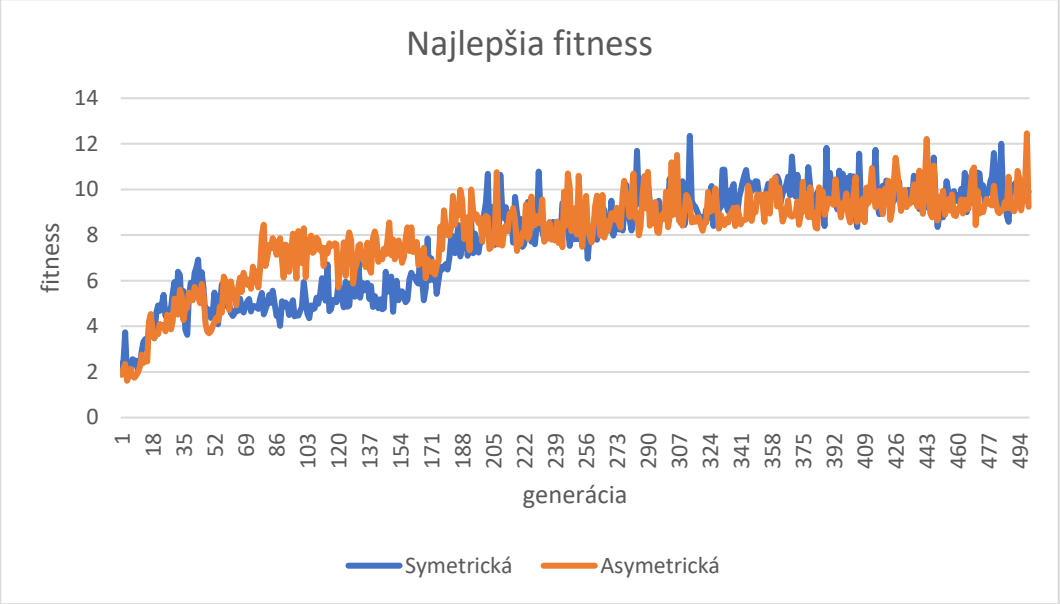


Obrázok 9: Priemerná fitness v každej generácii pri porovnávaní verzií s prefixom a bez prefixu

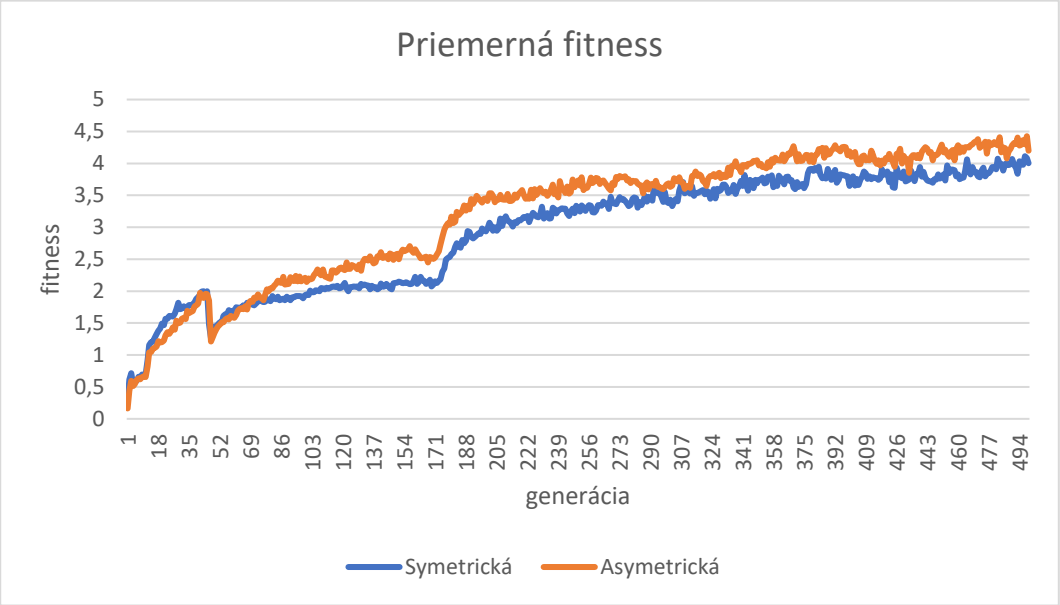
Verzia bez prefixu dosahovala v priemere lepšie výsledky ako verzia s prefixom. Vo výsledku avšak verzia s prefixom našla lepšieho jedinca ako verzia bez prefixu. Aj napriek tomu sa ale ukázalo, že náš predpoklad sa nenaplnil a náš program nevedel efektívne využiť prefix a s prefixom dosahoval v priemere horšie výsledky.

3.2.3.3 Porovnanie efektívnosti symetrie

Porovnali sme aj druhú a tretiu verziu, aby sme zistili, či symetrická verzia programu, kde sa krok druhou nohou doráta automaticky, je lepšia oproti pôvodnej verzii, kde jedinec má kontrolu nad evolúciou oboch polôh. Dostali sme nasledujúce výsledky.



Obrázok 10: Najlepšia fitness v priemere v každej generácii pri porovnávaní symetrických a asymetrických verzií



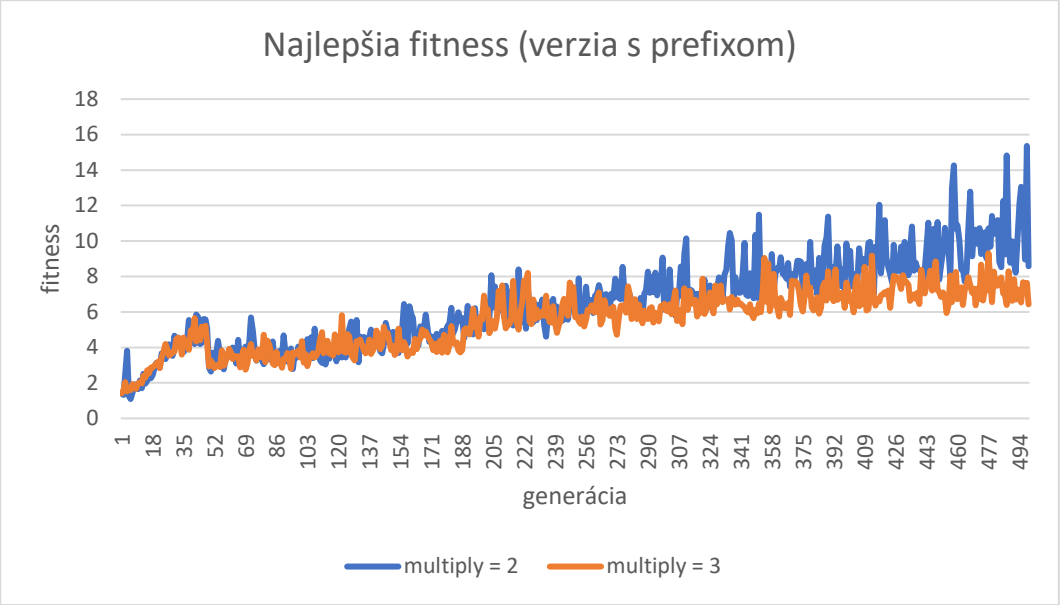
Obrázok 11: Priemerná fitness v každej generácii pri porovnávaní symetrických a asymetrických verzií

Rozdiel medzi symetrickou a asymetrickou verziou programu je minimálny. Asymetrická verzia je o niečo lepšia v priemere ako symetrická. Na druhej strane symetrická verzia našla vo výsledku viacej lepších jedincov ako asymetrická verzia, dôvodom takéhoto výsledku avšak môže byť iba primalý počet porovnávaných procesov. Náš predpoklad sa nenaplnil a symetrické dorátavanie génu nezlepšilo efektivitu nášho programu.

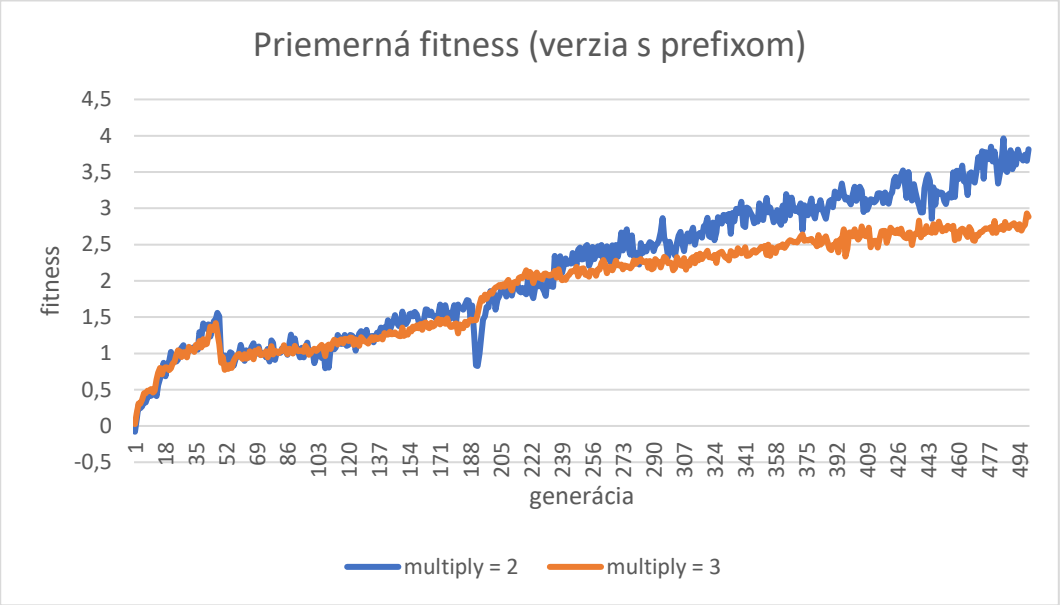
3.2.3.4 Porovnanie rôznej dĺžky vynásobenia génu

Porovnali sme prvú a štvrtú verziu programu a druhú a piatu verziu programu, aby sme zistili, aký vplyv má väčšia hodnota parametra *multiply*. Porovnali sme verzie s prefixom

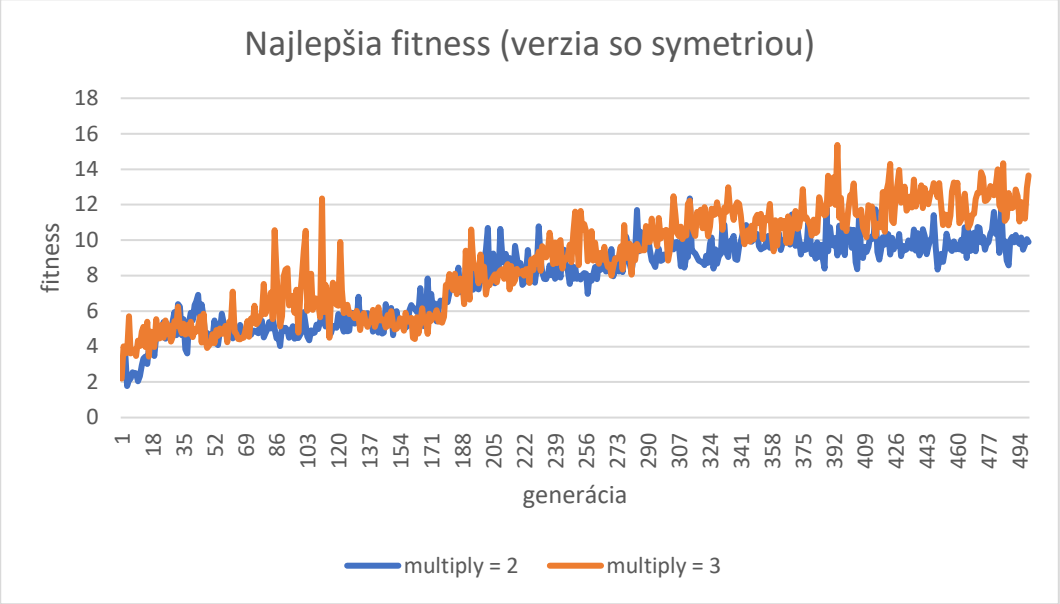
a verzie so symetriou s hodnotami parametra $multiply = 2$ a $multiply = 3$. Dostali sme takéto výsledky.



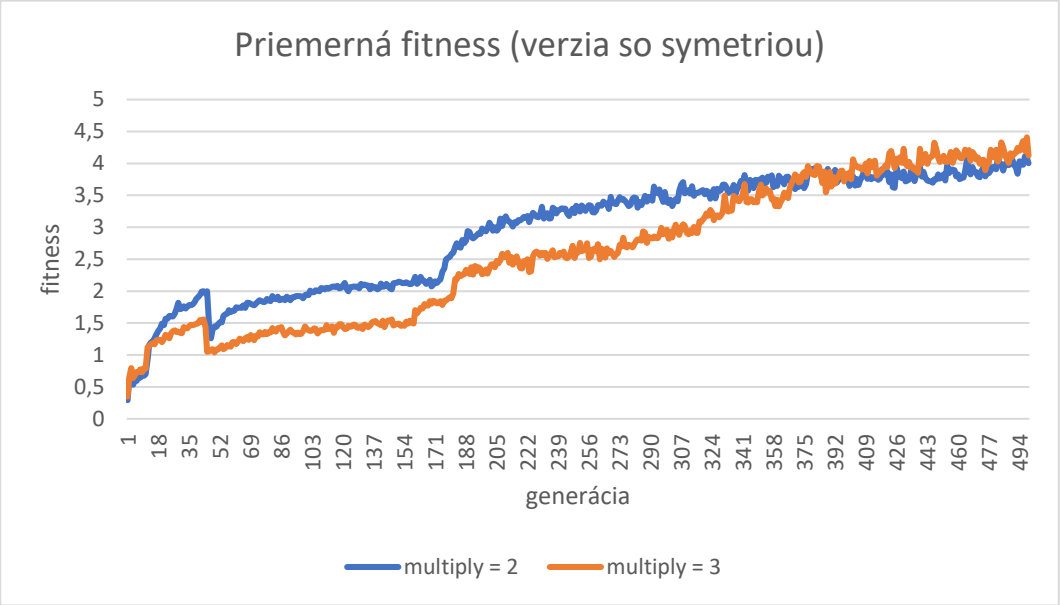
Obrázok 12: Najlepšie fitness v priemere v každej generácii pri porovnávaní rôznej hodnoty parametra multiply vo verziách s prefixom



Obrázok 13: Priemerné fitness v každej generácii pri porovnávaní rôznej hodnoty parametra multiply vo verziách s prefixom



Obrázok 14: Najlepšie fitness v priemere v každej generácii pri porovnávaní rôznej hodnoty parametra multiply vo verziách so symetriou



Obrázok 15: Priemerné fitness v každej generácii pri porovnávaní rôznej hodnoty parametra multiply vo verziách so symetriou

Obmena parametra *multiply* má na rôzne verzie programu iný dopad, v niektorých prípadoch by sme možno vedeli s vyšším parametrom *multiply* dosahovať aj lepšie výsledky.

3.2.4 Záver

Našli sme jedinca, ktorý sa približuje našej predstave chodiaceho jedinca, má však nedostatky, ktoré by sme chceli do budúcich verzií vylepšiť. Jedinec nám dal lepšiu predstavu o tom, že na jeden krok stačia jedincovi práve dve polohy a do ďalších verzií programu vieme znížiť dĺžku génu a tým zjednodušiť zložitosť celého problému.

Pridanie prefixu do génu jedinca zvýšilo zložitosť problému, a preto predpokladáme, že nezlepšilo dosahovanie lepších výsledkov. Pridanie symetrie nemalo na evolúciu väčší vplyv z hľadiska efektivity, avšak symetrické výsledky boli vo vizualizácii efektnejšie. Zmena parametra *multiply* môže mať na evolúciu pozitívny dopad aj napriek tomu, že celú chôdzu navýšenie tohto parametru spomalí, keďže nás zaujíma iba celková prejdená vzdialenosť.

3.3 Chôdza robota rozdelená na tri časti

V nasledujúcom experimente sme sa chceli viacej priblížiť našej predstave o chôdzi. Keďže pokusy o to spraviť chôdzu symetrickú a pridať do nej prefix boli neúspešné, rozhodli sme sa tieto časti vyvíjať v samostatných genetických algoritmoch. Prvý algoritmus bude hľadať prefix, teda bude začínať v počiatočnej polohe robota a bude sa snažiť ľavou nohou spraviť krok dopredu a zastaviť. Druhý algoritmus bude začínať v polohe s ľavou nohou vpredu a pravou nohou vzadu a bude mať za cieľ presunúť zadnú nohu dopredu a skončiť v polohe symetricky opačnej k svojej počiatočnej polohe. Postupnosť polôh na spravenie kroku z pozície s ľavou nohou vzadu a pravou nohou vpredu nebudeme získavať pomocou samostatného genetického algoritmu, ale využijeme na to postupnosť nájdenú druhým algoritmom, ktorá bude symetricky opačná. Z výsledkov týchto dvoch algoritmov chceme potom vyskladať chôdzu.

3.3.1 Prvý krok

Vytvorili sme genetický algoritmus, ktorým sa pokúsime nájsť postupnosť polôh, ktoré musí simulovaný robot vykonať na to, aby z počiatočnej polohy, v ktorej má nohy vedľa seba, spravil jeden krok a zastavil v polohe s ľavou nohou vpredu a v tejto polohe bez spadnutia zastal.

3.3.1.1 Predpoklad

Predpokladáme, že sa nám podarí nájsť krok z polohy znožmo, ktorý budeme môcť využiť vo finálnej chôdzi. Taktiež vyskúšame spustiť experimenty s 1-point crossoverom a porovnáme ich výsledky s výsledkami s 2-point crossoverom.

3.3.1.2 Realizácia

Experiment budeme spúšťať na vzdialených počítačoch pomocou príkazu xvfb-run podobne ako v predchádzajúcom experimente. Dokopy spustíme 12 procesov, z ktorých 8 bude na kríženie využívať 2-point crossover a 4 budú využívať 1-point crossover. Daný k-point crossover bude vykonávaný na každej polohe zvlášť a šanca, že sa na danej polohe vykoná, bude 50 %. Procesy budú mať rôzne účelové funkcie.

Fitness bude jedinec získavať za prejdenú vzdialenosť pravej šľapy od počiatočnej polohy po poslednú polohu, ktorá neviedla k pádu jedinca. Jedinec stráca fitness, ak nedokončil celú svoju postupnosť polôh bez pádu, pričom jedinec sa považuje za padnutého, ak jeho stred torza klesol pod výšku 30 centimetrov. Bude strácať fitness aj za to, keď pohne pravou nohou dopredu o viac ako je stanovený limit. Taktiež bude jedinec strácať fitness za to, keď posunie ľavú nohu príliš dopredu, teda za hranicu stanoveného limitu.

Spustíme šesť procesov s limitom 15 centimetrov na ľavú nohu a limitom 5 centimetrov na pravú nohu, z nich štyri procesy budú využívať kríženie 2-point a dva procesy 1-point. Spustíme dva procesy s limitom 10 centimetrov pre ľavú nohu a 2 pre pravú a dva procesy s limitom 7,5 centimetrov pre pravú nohu a 2 pre ľavú, v oboch prípadoch bude jeden proces využívať 1-point a druhý 2-point crossover. Posledné dva procesy budú využívať 2-point crossover a limit na ľavú nohu budú mať 5 centimetrov a na pravú 1 centimeter.

Počet jedincov bude 300 a genetický algoritmus bude bežať dokým nenájde jedinca, ktorý vie spraviť krok z polohy znožmo. Jedinci nebudú vytvorení úplne náhodne, ale budú vytvorení z prvých dvoch polôh najlepšieho jedinca z predchádzajúceho experimentu s odchýlkou $\pm 5°$ na každej hodnote uhlu. Gén jedinca bude pozostávať z dvoch polôh. Šanca na mutáciu jedinca bude $10\ \%$ a takto zmenenému jedincovi sa každý uhol so šancou $10\ \%$ zmení o $\pm 2°$. Budeme používať hodnotu parametra $multiply = 6$. Kvôli tomu, aby nenastávali priveľké odchýlky v hodnote najlepšej fitness naprieč generáciami, využijeme elitizmus, teda najlepší štyria jedinci budú pokračovať do ďalšej generácie nezmenení, avšak budú taktiež nanovo ohodnotení účelovou funkciou.

Kvôli zložitosti a stochastickosti simulovaných fyzikálnych javov môže byť ohodnotenie toho istého jedinca rôzne pri viacerých ohodnoteniach programom. Aj kvôli tomuto sa bude diať, že najlepšia fitness môže naprieč generáciami klesať aj napriek tomu, že použijeme elitizmus. Preto bude každý jedinec spustený trikrát a výsledná fitness jedinca sa vyráta ako priemerná fitness z týchto troch pokusov.

3.3.1.3 Výsledok

Zo všetkých dvanástich spustených procesov našlo vyhovujúcu postupnosť polôh šesť procesov za 100 generácií. Po týchto 100 generáciách sme experiment ukončili ako úspešný.

Výsledné riešenie našli tri zo štyroch procesov s 1-point crossoverom, pričom každý proces s rozdielnymi parametrami na maximálne hranice polôh pre nohy. Riešenie tiež našli tri z ôsmich procesov s 2-point crossoverom, pričom každý proces s rozdielnymi parametrami na maximálne hranice polôh pre nohy. Dva procesy s limitom polohy pre ľavú nohu 5 centimetrov

a pre pravú nohu 1 centimeter nenašli riešenie.

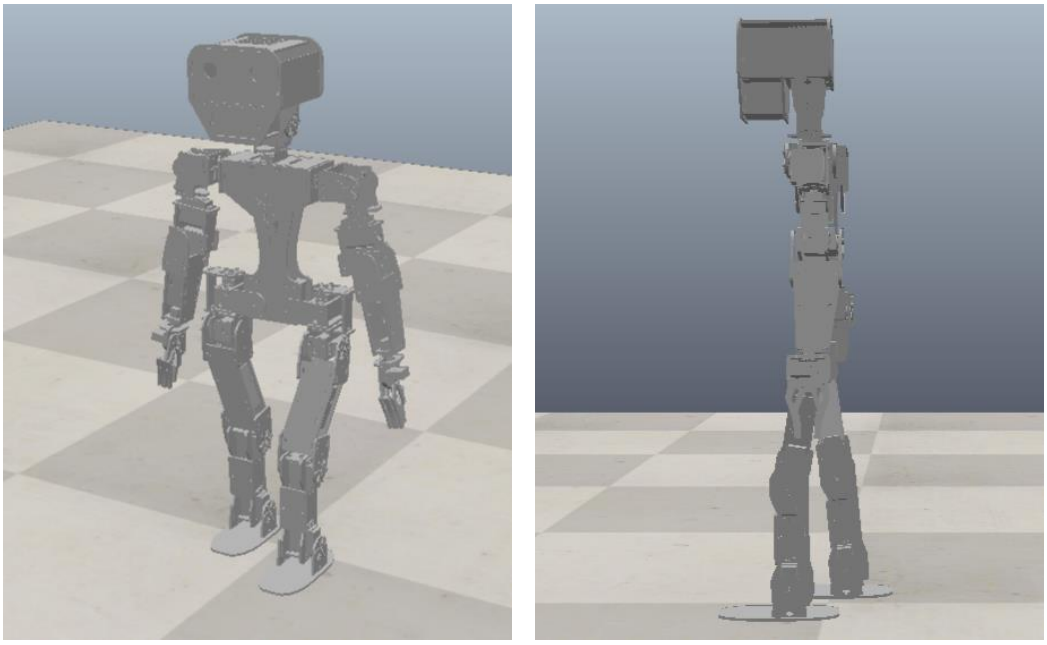
V priemere sa rýchlejšie k riešeniu dopracovali procesy s 1-point crossoverom a to za približne 60 generácií. Procesy s 2-point crossoverom sa k riešeniu v priemere dostali za 75 generácií.

3.3.1.4 Záver

Našli sme niekoľko vyhovujúcich jedincov s rôznou šírkou kroku. Porovnali sme efektívnosť 1-point a 2-point crossoveru a vyšlo nám, že pri takto malom počte polôh je 1-point crossover efektívnejší.

3.3.2 Upravené modely robota Lilli

V ďalšom experimente budeme potrebovať, aby načítaný model do scény nemal nohy pri sebe. Vytvorili sme preto ďalšie štyri modely robota Lilli, v ktorých má robot Lilli ľavú nohu vpredu, pravú vzadu a je v stabilnej polohe. Každý z týchto modelov má inú šírku rozkročenia nôh. Všetky modely majú oproti pôvodnému modelu zmenené polohy kĺbov *l_hip_x, r_hip_x, l_knee_x, r_knee_x* tak, aby ich šľapy stáli celou podrážkou na podlahe. Skúšali sme aj modely s vystretými kolenami, ale s týmito modelmi sme dosahovali horšie výsledky.



Obrázok 16: Upravený model s pokrčeným kolenom

Modely sme hľadali pomocou príkazu sim.setObjectPosition, ktorým sme najprv celý model nastavili na súradnicu $z = 50$, čím sme robota zdvihli zo zeme o 5 centimetrov. Tento príkaz sme spustili v customization scripte, aby sa vykonal mimo behu simulácie. Vďaka tomu,

že sa príkaz vykonal mimo behu simulácie, tak na model robota nepôsobí gravitácia a pomocou príkazov sim.setJointPosition vieme nastaviť jeho kĺby, ktorých stabilitu vieme overiť spustením simulácie. Na rozdiel od behu programu, v ktorom na ovládanie kĺbov využívame príkaz sim.setJointTargetPosition, v tomto prípade musíme využiť sim.setJointPosition, keďže príkaz sim.setJointTargetPosition ráta s fyzikou, a teda funguje iba počas behu simulácie.

V takto vytvorenom modeli musíme nechať príkaz sim.setObjectPosition vo funkcii sysCall_init, ktorý celý model dvihne o 5 centimetrov vyššie, aby načítaný model dokázal nastaviť polohu svojich nôh. V prvom kroku simulácie potom robot začne byť ovplyvňovaný gravitáciou a spadne na zem, avšak vďaka tomu, že sme robota dvihli iba o malú vzdialenosť, 5 centimetrov, ostane stáť.

Keďže v účelovej funkcii nášho programu si budeme pamätať počiatočnú pozíciu šliap načítaného modelu, musíme po načítaní modelu počkať krátky čas, aby simulácia stihla jedinca načítať. Ak by sme to nespravili, tak načítaný model by sa nemusel stihnúť dať do svojej počiatočnej polohy skôr, než by sme uložili súradnice jeho šliap do premenných a tým pádom by sme pracovali s nesprávnymi hodnotami.

3.3.3 Druhý krok

Nadviazali sme na výsledok predchádzajúceho experimentu a naprogramovali sme genetický algoritmus, ktorý má za úlohu nájsť postupnosť polôh, ktoré robot potrebuje na to, aby spravil druhý krok chôdze. Robot začína v rovnovážnej polohe s ľavou nohou vpredu a pravou nohou vzadu. Jeho cieľom je presunúť pravú nohu dopredu, zastaviť a zotrvať v koncovej polohe, ktorá je symetrická k jeho počiatočnej polohe.

3.3.3.1 Predpoklad

Predpokladáme, že nájdeme krok z konečnej polohy predchádzajúceho experimentu, teda z polohy, kde má robot ľavú nohu vpredu. Tento krok bude končiť v polohe s pravou nohou vpredu, ktorá bude symetrická k jeho počiatočnej polohe. Za pomoci tejto nájdenej postupnosti polôh a symetricky opačnej postupnosti polôh vytvorenej z nej očakávame, že zostavíme celú chôdzu.

3.3.3.2 Realizácia

Experimenty sme spúšťali na vzdialených počítačoch pomocou príkazu xvfb-run. Spúšťali sme 24 procesov, z toho 12 procesov realizovalo kríženie pomocou 1-point a 12 procesov pomocou 2-point. Crossover bude vykonávaný na každej polohe zvlášť s pravdepodobnosťou 50 %, že sa na sa jednotlivej polohe vykoná. Všetky ostatné nastavenia

programu boli v každom experimente rovnaké.

Oproti predchádzajúcemu experimentu navýšime veľkosť populácie na 500. Genetický algoritmus bude bežať dokým nenájde vyhovujúceho jedinca, teda jedinca, ktorý sa dostane do konečnej polohy, zotrvá v nej a pri tom nespadne. Jedinci budú vytváraní úplne náhodne. Ich gén bude pozostávať zo štyroch polôh, pričom prvá a posledná poloha sú navzájom symetricky opačné a tieto polohy nebudú menené krížením a mutáciou. Mutácia bude s pravdepodobnosťou 10 % a na každom uhle bude následne šanca 10 %, že sa zmení o $\pm 5°$. Pridali sme do mutácie s pravdepodobnosťou 10 %, že jedincovi sa do jeho génu pridá na predposledné miesto nová poloha, ktorá vznikne z predposlednej polohy upravením každého uhla o $\pm 5°$. Na rozdiel od predchádzajúceho experimentu, bude každý jedinec účelovou funkciou ohodnotený iba raz, pretože viacnásobné ohodnocovanie jedincov iba spomalilo beh simulácie a nedosahovalo lepšie výsledky. Elitizmus bude nastavený na 20 a v každej ďalšej generácii sa vygeneruje 50 nových náhodných jedincov.

V prvotných experimentoch sme skúšali pridať jedincovi ďalšie dva parametre, ktoré sme krížili pomocou uniform crossoveru. Parametre, ktoré sme pridali, boli hodnota parametra *multiply* a model, aký sa ma pri jeho spustení načítať. Podľa neho sa nastaví počiatočná a konečná poloha v géne. Využili sme pri tom štyri modely robota Lilli, ktoré sme si pre tento experiment vytvorili. Experimenty boli neúspešné a vo finálnom experimente sme tieto parametre jedincovi odobrali a ich hodnoty boli v programe fixne nastavené. Parameter *multiply* sme nastavili na 2 a načítavali sme model s najnižším rozkročením nôh.

3.3.3.3 Hľadanie účelovej funkcie

Účelovú funkciu sme každým experimentom menili, dokým sme nedospeli k funkcii, ktorá najlepšie hodnotila náš problém. V každej verzii jedinec stratil veľa fitness, pokiaľ sa mu nepodarilo dokončiť všetky polohy vo svojom géne, aby sme vedeli z priebežných výsledkov odsledovať, kedy sme našli výsledného jedinca.

Začali sme s funkciou, ktorá po vykonaní každej ďalšej polohy v géne prirátala jedincovi k zatiaľ získanej fitness vzdialenosť pravej šľapy od počiatočnej polohy. Jedinec strácal fitness za to, keď pohol ľavou nohou o viac ako 1 centimeter. Jedinci v experimentoch s touto fitness príliš rýchlo hýbali nohou ďaleko dopredu, aby maximalizovali získanú fitness. Títo jedinci nemali predpoklad na to dokončiť celý krok podľa našej predstavy.

V ďalšom experimente sme preto upravili účelovú funkciu tak, že jedinec strácal fitness, ak pravú nohu presunul príliš ďaleko. Taktiež sme upravili získavanie fitness a to tak, že jedinec získal konštantnú hodnotu fitness, ak poloha jeho pravej šľapy bola väčšia alebo rovná

predchádzajúcej polohe šľapy. Tým sme chceli docieliť, aby jedinci nehýbali nohou príliš rýchlo a aby vyhrávali jedinci, ktorí dokončia celú svoju postupnosť polôh. Začali avšak vyhrávať jedinci, ktorí mutáciou získali viac polôh a hýbali nohou pomaly dopredu, a potom prudko nohou pohli dozadu. Ani jeden z týchto jedincov nedokončil celý krok.

Účelová funkcia finálneho experimentu teda vyzerala tak, že jedinec strácal fitness, keď pohol ľavou nohou. Strácal fitness, keď pravou nohou pohol príliš ďaleko alebo keď ňou pohol dozadu oproti predchádzajúcej polohe. Získaval konštantnú fitness za každú vykonanú polohu, ktorá pravú nohu presunula viac dopredu alebo ňou nepohla.

3.3.3.4 Výsledok

Takto vykonané experimenty s týmito parametrami skončili neúspešne. Najlepší jedinci, akých sme našli, boli vo verzii experimentu, v ktorom jedincovou súčasťou bol parameter *multiply*. Jedinci z tohto experimentu sa evolúciou dostali až k hodnote $multiply = 0$ a vedeli dokončiť krok, avšak krok robili nerealisticky rýchlo oproti reálnemu robotovi Lilli, a tak aj z tohto dôvodu sme tieto parametre z evolúcie v ďalších experimentoch vynechali.

3.3.3.5 Záver

Zistili sme, že nevieme nájsť chôdzu s takými vlastnosťami, aby zodpovedala realite našim doterajším programom. Výsledky, ktoré vedia prísť do konečného stavu, tak robia rýchlosťou, ktorá nezodpovedá reálnej rýchlosti robota. Vyskúšali sme nájsť chôdzu aj manuálne, mimo behu simulácie, ale tiež neúspešne.

3.3.3.6 Úprava doterajšej verzie

Na základe doterajších výsledkov sme sa rozhodli upraviť doterajšiu verziu chôdze a to tak, že sme zmenili kĺby, ktoré robot počas chôdze využíva. Odstránili sme kĺby *l_shoulder_y* a *r_shoulder_y*, ktoré mal robot využívať na vyvažovanie. Poloha týchto kĺbov je automaticky nastavená tak, aby sa ruky držali čo najbližšie k telu. Pridali sme dva nové kĺby *l_hip_z* a *r_hip_z*. Spustíme kód z predchádzajúceho experimentu, v ktorom zmeníme kĺby, ktoré simulovaný robot ovláda. Dokopy spustíme 33 procesov, z ktorých 21 bude realizovať kríženie pomocou 2-point crossoveru a 12 procesov pomocou 1-point crossoveru na každej polohe s pravdepodobnosťou 50 %, že sa vykoná pre danú polohu.

3.3.3.7 Výsledok upravenej verzie

Zo všetkých 33 procesov v zmenenom experimente s pridanými kĺbmi sa podarilo 24 procesom nájsť postupnosť polôh potrebných na druhý krok za 60 generácií. Z procesov využívajúcich 2-point crossover našlo úspešne druhý krok 12 procesov a z procesov

využívajúcich 1-point crossover tiež 12 procesov aj napriek tomu, že bolo s 1-point crossoverom spustených menej procesov. Každú výslednú postupnosť polôh úspešných procesov sme samostatne spustili a zhodnotili jej efektívnosť v porovnaní so skutočným modelom Lilli. Zo všetkých procesov 2 procesy s 2-point crossoverom a 3 procesy s 1-point crossoverom sme označili ako realistický krok, ktorý by vedel vykonať aj skutočný robot Lilli.

3.3.3.8 Záver upravenej verzie

Podarilo sa nám nájsť viacerých jedincov, ktorí vedeli spraviť druhý krok chôdze z polohy s ľavou nohou vpredu a pravou nohou vzadu.

3.3.4 Spojenie chôdze do celku

Skúsili sme spojiť jednotlivé tri časti do finálnej chôdze, ktorá pozostávala z prvého kroku ľavou nohou z polohy s nohami pri sebe. Ďalej nasledoval krok pravou nohou z polohy s ľavou nohou vpredu a krok ľavou nohou z polohy s pravou nohou vpredu, ktoré sa až do pádu robota opakovali.

Mnohé postupnosti polôh nájdené doterajšími experimentami avšak nedosahovali dostatočné výsledky, keďže neboli evolúciou testované na viac ako jeden krok. Rozhodli sme sa preto experiment zopakovať so zmenou, že simulovaný robot bude vykonávať viac ako len jeden krok. Gén jedinca bude stále pozostávať iba z jedného kroku a krok opačnou nohou bude vytvorený zo symetricky opačnej postupnosti, akú má jeho gén. Máme za cieľ zdokonaliť doterajšie výsledky a nájsť jedincov, ktorí vedia stabilne vykonávať chôdzu.

3.3.4.1 Realizácia vylepšovania chôdze

Experimenty sme spúšťali na vzdialených počítačoch pomocou príkazu xvfb-run. Spúšťali sme 24 procesov, z toho 12 procesov realizovalo kríženie pomocou 1-point a 12 procesov pomocou 2-point. Crossover bol robený na každej polohe samostatne so šancou 50 % na jeho vykonanie. Populácia bude veľkosti 500. Mutácia bude s pravdepodobnosťou 10 % a uhly zmutovaných jedincov sa so šancou 10 % zmenia o $\pm 5°$. Šanca na pridanie novej polohy jedincovi bude 10 %, tá potom vznikne z predposlednej polohy upravením každého uhla o $\pm 5°$. Každý jedinec bude účelovou funkciou ohodnotený iba raz. Elitizmus bude nastavený na 20 a do novej generácie bude pridaných 50 nových náhodných jedincov.

Jedinci budú vytváraní úplne náhodne. Ich gén bude pozostávať z troch polôh, pričom prvá poloha ostáva nemenná a vychádza z počiatočnej polohy robota, ktorý je v polohe s ľavou nohou vpredu. Z experimentu vynecháme prvý krok, ktorý robot musí spraviť z polohy s oboma nohami pri sebe. Pri testovaní jedinca budú na koniec jeho génu pridané ďalšie tri

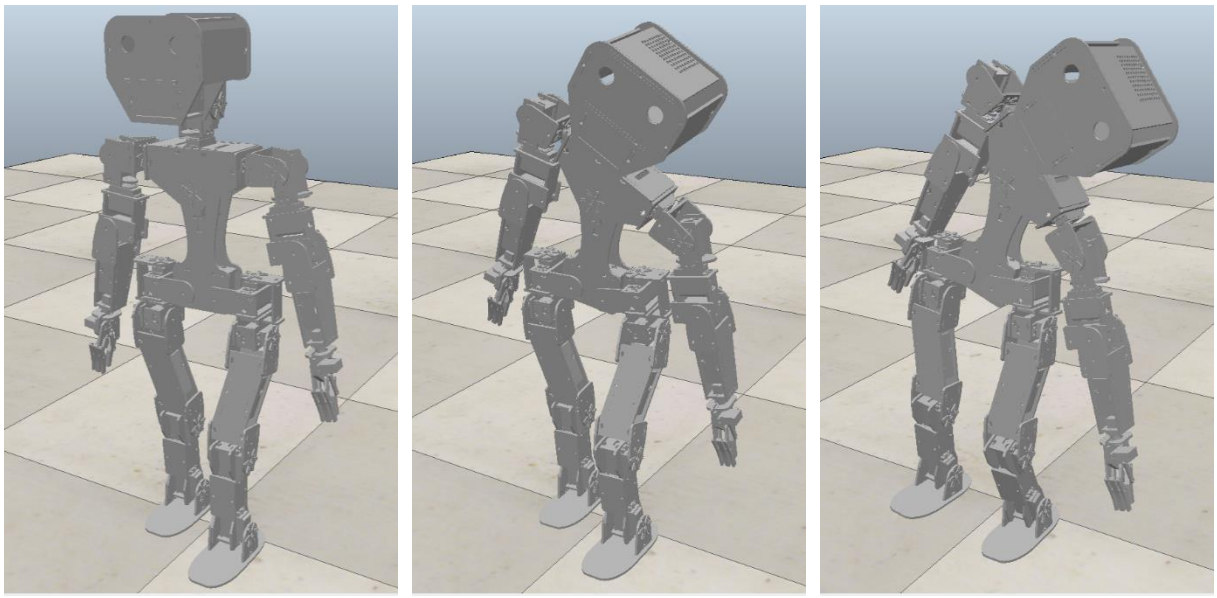
polohy, ktoré budú k jeho polohám symetricky opačné. Jedinec svoj gén vykoná 10-krát, teda spraví 10 krokov ľavou nohou a 10 krokov pravou nohou. Jedinec bude ihneď zastavený, ak spadne.

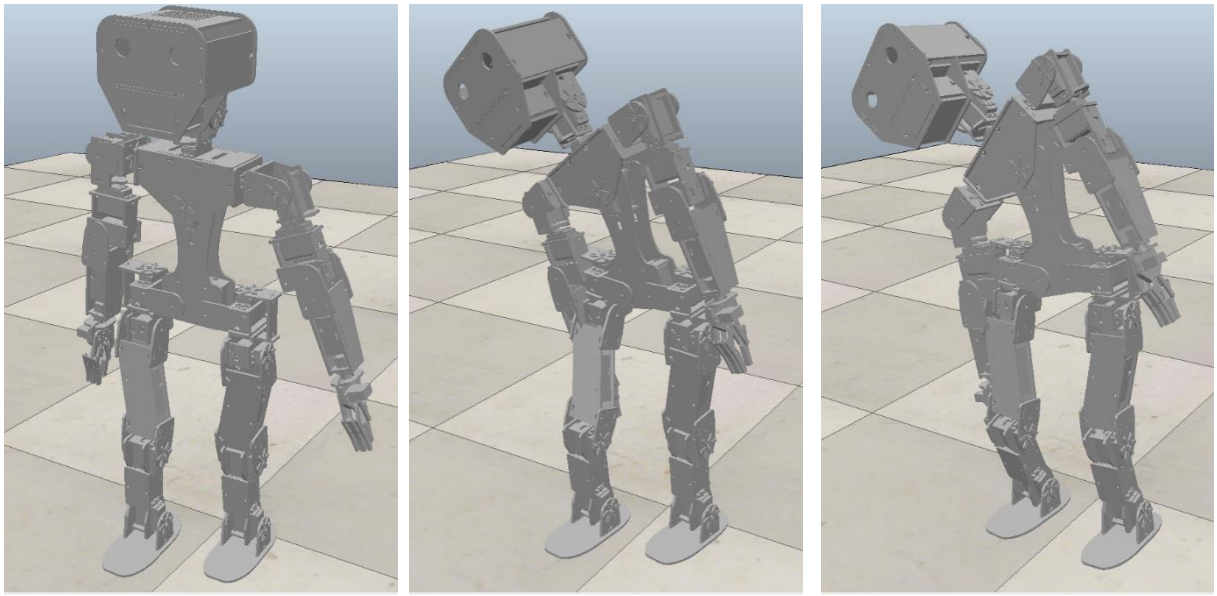
Účelová funkcia bude pozostávať z dvoch častí. Ak jedinec ešte nespravil prvý krok, bude fitness strácať, keď pohne ľavou nohou. Stráca fitness, aj keď pravou nohou pohne príliš ďaleko alebo keď ňou pohne dozadu oproti predchádzajúcej polohe. Získava fitness, keď pohne pravou nohou dopredu oproti predchádzajúcej polohe. Pokiaľ jedinec už spravil prvý krok, tak získava konštantnú fitness za každú vykonanú polohu, ktorá neviedla k pádu jedinca. Na konci ohodnocovania dostane fitness podľa prejdenej vzdialenosti podľa osi *y*.

3.3.4.2 Výsledok

Zo všetkých 24 procesov našlo za 70 generácií chôdzu 11 procesov. Najlepší jedinci z týchto procesov vedeli svojou postupnosťou polôh prejsť 20 krokov. Z týchto procesov, ktoré našli chôdzu, bolo 8 procesov s 1-point crossoverom a 3 procesy s 2-point crossoverom.

Všetkých najlepších jedincov z každého procesu, čo našli chodiaceho jedinca, sme si samostatne spustili a zhodnotili jeho efektívnosť a efektnosť chôdze. Následne sme vybrali troch najlepších jedincov, ktorých sme spustili 5000-krát a zmerali sme ich priemernú prejdenú vzdialenosť. Druhý a tretí najlepší jedinci prešli priemernú vzdialenosť 90 cm a 70 cm. Najlepší nájdený jedinec prešiel priemernú vzdialenosť 1,3 metra za daných podmienok. Najväčšiu vzdialenosť, akú vedel dvadsiatimi krokmi prejsť bolo 1,7 metra.





Obrázok 17: Šesť polôh tvoriacich chôdzu

3.3.4.3 Záver

Podarilo sa nám nájsť chodiaceho jedinca, ktorý spĺňa našu predstavu o chodiacom jedincovi. Jedinec vie spraviť viac ako tri kroky, strieda pri chôdzi obe nohy a chôdzu vykonáva efektívne, teda každým krokom sa posunie viac dopredu.

3.4 Testovanie parametrov genetického algoritmu

Zobrali sme si náš experiment, ktorý hľadal chôdzu z pozície s ľavou nohou vpredu a pravou nohou vzadu a reprezentoval jedincov pomocou troch polôh, z ktorých prvá poloha bola nemenná. Tieto polohy reprezentovali krok pravou nohou a následne krok ľavou nohou bol u jedinca vytvorený automaticky ako symetricky opačná postupnosť polôh. Tieto polohy boli opakované, dokým jedinec nespadol alebo nevykonal maximálny povolený počet krokov. Bližšia špecifikácia experimentu a jeho parametrov je v kapitole 3.3.4.1.

Rozhodli sme sa porovnať efektivitu tohto genetického algoritmu s rôznymi parametrami. Pozorovali sme efektívnosť rôznych crossoverov a rôznej sily mutácie.

3.4.1 Porovnanie crossoverov

Porovnávali sme efektívnosť rôznych crossoverov na hľadanie chôdze naším GA. Porovnali sme 1-point a 2-point crossover, ktorý sme využívali v predchádzajúcom experimente. Tieto crossovery sú vykonávané na každej polohe zvlášť s pravdepodobnosťou 50 %, teda na každej z dvoch polôh nastane s 50 % pravdepodobnosťou daný k-point crossover. Tak isto sme porovnali 1-point a 2-point crossover vykonávaný na celom géne, teda časť génu,

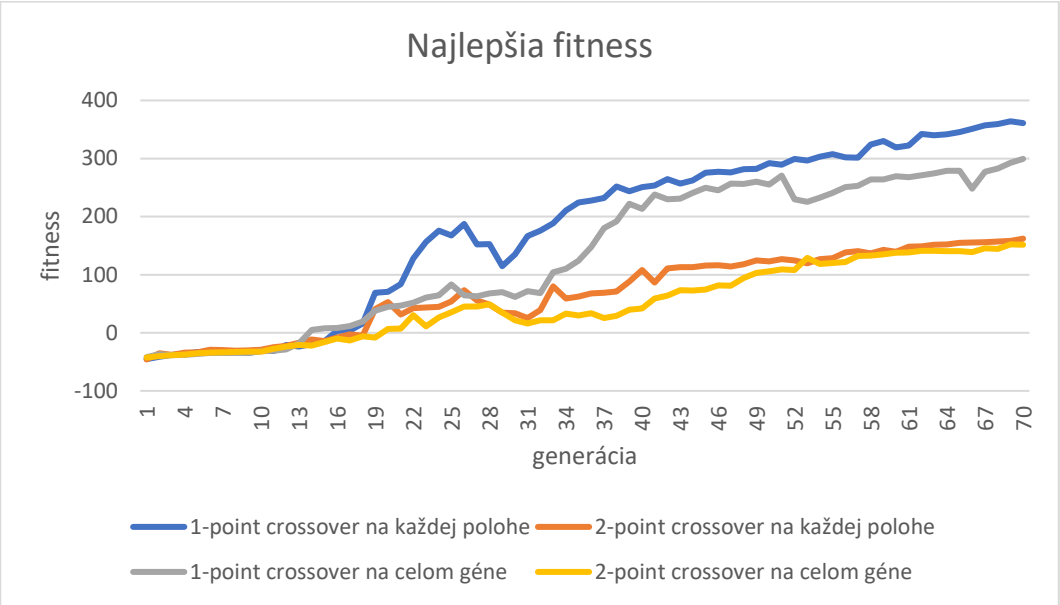
ktorá je menená crossoverom, bude pri krížení braná ako jedno pole. Aj v tomto prípade bola šanca na crossover 50 %.

3.4.1.1 Realizácia

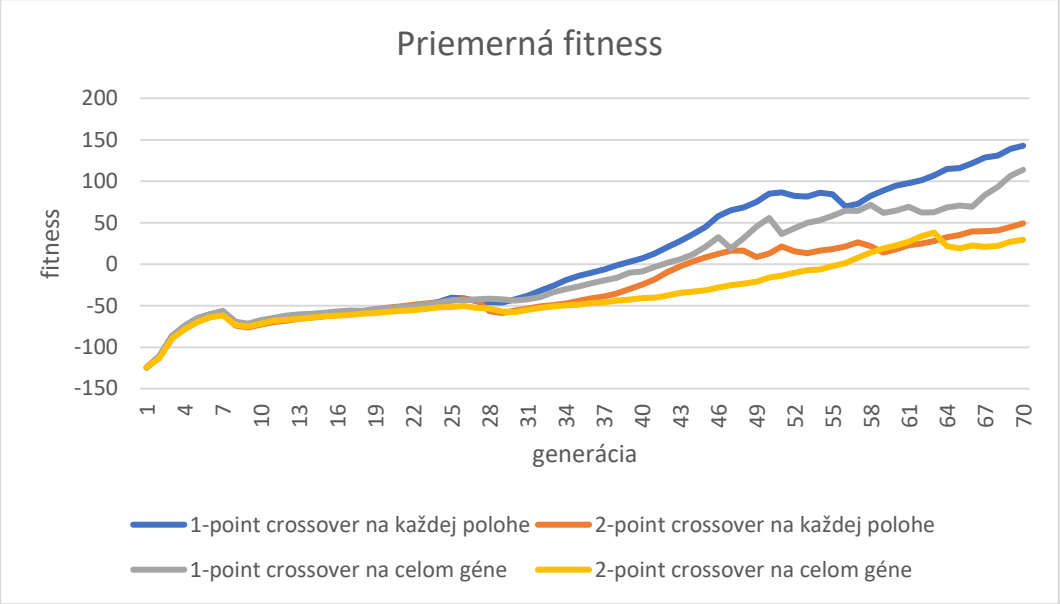
Pre každý zo štyroch rôznych crossoverov sme spustili 12 procesov na vzdialenom počítači príkazom xvfb-run. Každý proces bežal 70 generácií a do súboru zapisoval na konci každej generácie fitness najlepšieho jedinca a priemernú fitness populácie.

3.4.1.2 Výsledok

Spravili sme grafy, ktoré ukazujú priemernú fitness najlepšieho jedinca vo všetkých generáciách a priemer z priemernej fitness populácie v každej generácii naprieč všetkými procesmi pre každý porovnávaný crossover.



Obrázok 18: Najlepšia fitness v priemere v každej generácii pri porovnávaní crossoverov



Obrázok 19: Priemerná fitness v každej generácii pri porovnávaní crossoverov

3.4.1.3 Záver

1-point crossover v priemere dosahoval vyšší nárast účelovej funkcie naprieč generáciami ako 2-point crossover. Taktiež v priemere dosahoval vyššie hodnoty fitness. Najlepšie hodnoty fitness v priemere dosahoval 1-point crossover, ktorý bol vykonávaný na každej polohe.

3.4.2 Porovnanie sily mutácie

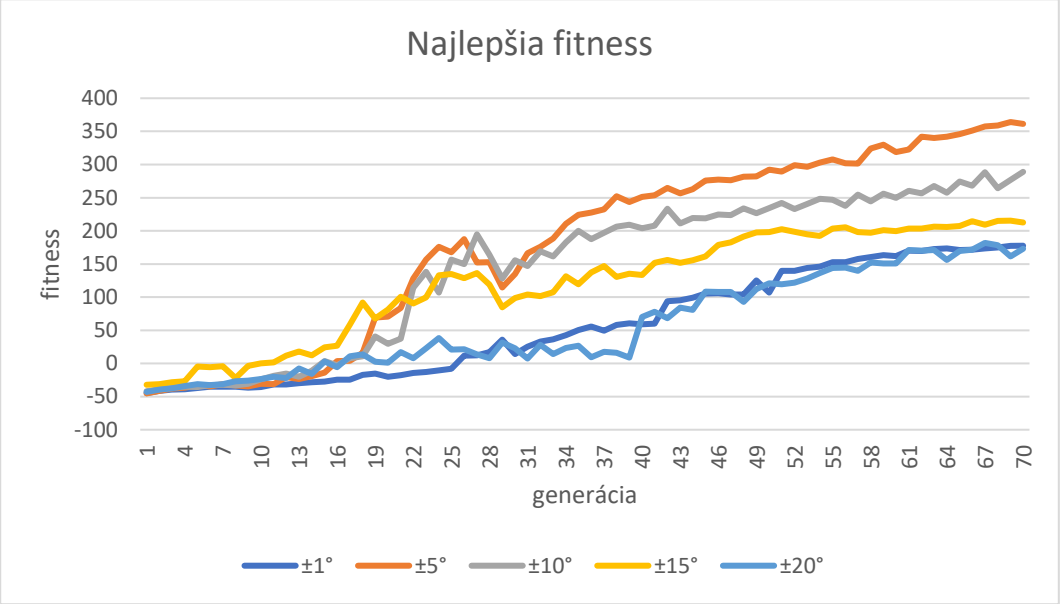
Porovnávali sme efektívnosť rôznej sily mutácie na hľadanie chôdze naším GA. Porovnali sme hodnoty sily $\pm 1°$, $\pm 5°$, $\pm 10°$, $\pm 15°$ a $\pm 20°$.

3.4.2.1 Realizácia

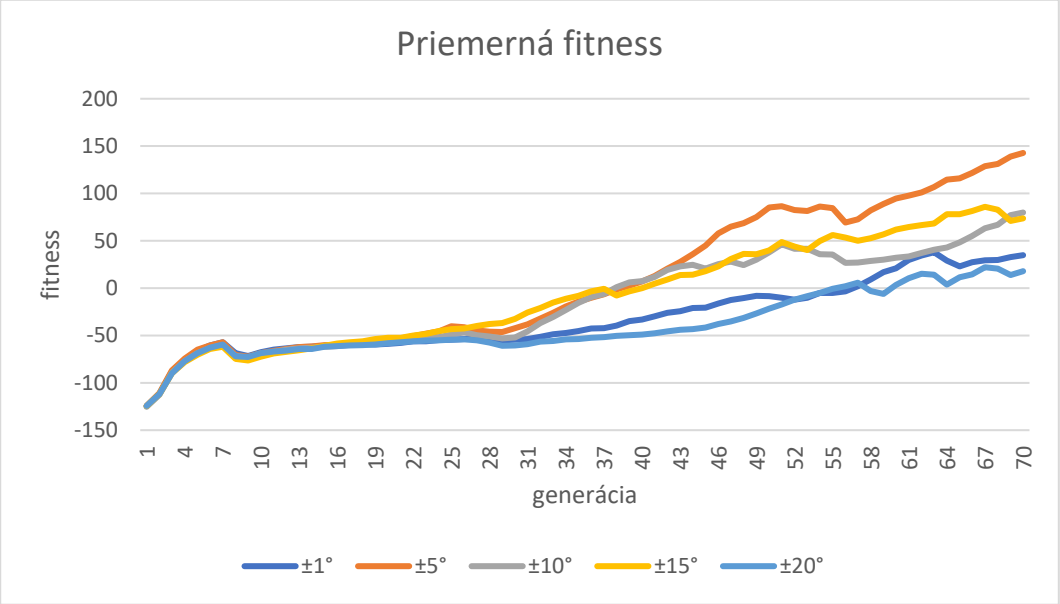
Pre každú rôznu hodnotu sily mutácie sme spustili 12 procesov na vzdialenom počítači príkazom xvfb-run. Každý proces bežal 70 generácií a do súboru zapisoval na konci každej generácie fitness najlepšieho jedinca a priemernú fitness populácie. Každý proces bežal s 1-point crossoverom vykonávanom na každej polohe s pravdepodobnosťou 50 %. Pravdepodobnosť mutácie bola 10 % a v jedincovi, ktorý mal byť zmenený mutáciou, sa zmenil každý uhol s pravdepodobnosťou 10 % o danú hodnotu sily mutácie.

3.4.2.2 Výsledok

Vygenerovali sme grafy, ktoré porovnávajú priemernú fitness najlepšieho jedinca vo všetkých generáciách a priemer z priemernej fitness populácie v každej generácii naprieč všetkými procesmi pre každú porovnávanú silu mutácie.



Obrázok 20: Najlepšia fitness v priemere v každej generácii pri porovnávaní rôznej sily mutácie



Obrázok 21: Priemerná fitness v každej generácii pri porovnávaní rôznej sily mutácie

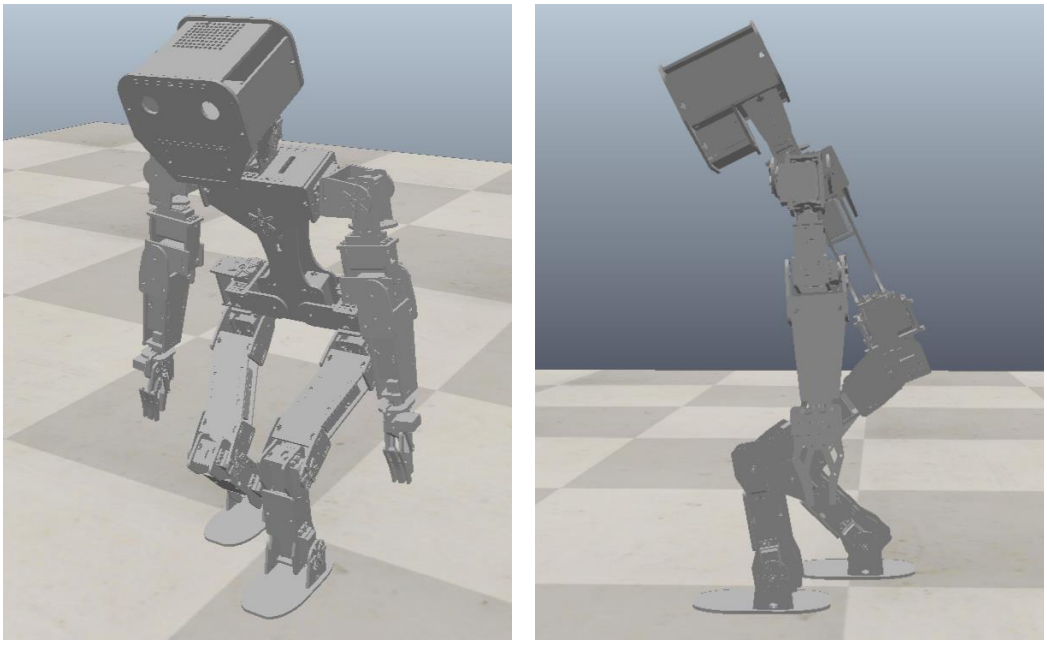
3.4.2.3 Záver

Najlepšie výsledky sme dosiahli so silou mutácie $\pm 5°$. Príliš nízka sila mutácie, $\pm 1°$ a príliš vysoká sila mutácie, $\pm 20°$, dosahujú podstatne horšie výsledky.

3.5 Zohnutá chôdza

Rozhodli sme sa otestovať náš doterajší genetický algoritmus na hľadanie chôdze s obmenenými podmienkami. Pokúsili sme sa nájsť chôdzu pre robota v zohnutej polohe. Vytvorili sme na tento účel tri modely, každý s iným stupňom zohnutia. Oproti pôvodnému

modelu, ktorý začína s torzom vo výške 45 cm, naše vytvorené modely mali torzo vo výške 36,5 cm, 35 cm a 33,5 cm. Nakoniec sme v experimente využívali model s torzom vo výške 36,5 cm.



Obrázok 22: Zohnutý model robota Lilli

3.5.1 Predpoklad

Predpokladáme, že sa nám naším doterajším programom podarí nájsť chôdza pri zmene pôvodného modelu na model, v ktorom je robot Lilli zohnutý.

3.5.2 Realizácia

Použili sme náš program, ktorý je popísaný v kapitole 3.3.4.1 Spustili sme 10 procesov na vzdialenom počítači pomocou príkazu xvfb-run.

Verzia programu bola zmenená tak, že sa mutáciou nepridávali nové polohy. Taktiež jedinec počas behu GA mohol spraviť maximálne 10 krokov. Jedinec bol zastavený, pokiaľ sa vystrel, teda jeho torzo sa dostalo nad výšku 43 cm, a bol penalizovaný mínusovou fitness.

Jedinci boli tvorení z troch polôh, ktoré reprezentovali jeden krok a druhý krok chôdze bol vytvorený z tejto postupnosti tým, že sa symetricky otočila. Prvá poloha postupnosti bola nemenná a vychádzala z počiatočnej polohy použitého modelu robota. Generáciu tvorilo 500 jedincov, ktorí neboli tvorení úplne náhodne, ale vznikli úpravou počiatočnej polohy a symetricky otočenej počiatočnej polohy o $\pm 20°$. Využívali sme 1-point crossover robený na každej polohe s pravdepodobnosťou 50 %. Mutácia na jedincovi bola nastavená na 10 %

a zmutovanému jedincovi bol každý jeho uhol so šancou 10 % zmenený o $\pm 5°$. Účelová funkcia a tvorba novej generácie bola robená rovnako ako v experimente opísanom v kapitole 3.3.4.1.

3.5.3 Výsledok

Procesy sme zastavili po päťdesiatich generáciách. Viacerými procesmi sme našli chodiaceho jedinca, ktorý spĺňal naše predstavy o chodiacom jedincovi, teda striedal na chôdzu obe nohy, robili efektívne kroky, ktorými sa posúval dopredu a vedel spraviť viac ako tri kroky. Chodiaceho jedinca našlo 5 z 10 spustených procesov. Týchto 5 procesov našlo jedinca, ktorý vedel spraviť 10 krokov bez toho, aby spadol. Najlepší jedinec vedel prejsť dvadsiatimi krokmi vzdialenosť 1,6 metra.

3.5.4 Záver

Podarilo sa nám naším programom nájsť chodiaceho jedinca aj za pozmenených počiatočných podmienok, teda za použitia modelu, ktorý je zohnutý o 8,5 centimetra.

Záver

Cieľom práce bolo nájsť takú postupnosť pohybov, ktorá by umožnila humanoidnému robotovi Lilli chodiť v prostredí robotického simulátora CoppeliaSim pomocou genetického algoritmu. Tiež sme v práci porovnali efektívnosť rôznych parametrov genetického algoritmu pri hľadaní nášho riešenia. Skúsili sme nájsť pomocou nášho genetického algoritmu aj chôdzu s iným modelom, konkrétne so zohnutým modelom robota Lilli.

Najprv sme analyzovali robotický simulátor CoppeliaSim, prácu s ním a ako v ňom efektívne navrhnúť genetický algoritmus, ktorý bude počas behu simulácie testovať jedincov každej generácie. Po navrhnutí genetického algoritmu sme začali hľadať postupnosť pohybov, ktoré musí robot v simulátore vykonať na to, aby chodil. Skúsili sme viaceré stratégie na hľadanie výsledných pohybov. Vyhovujúce pohyby sme nakoniec našli, keď sme pomocou genetického algoritmu hľadali iba jeden krok z počiatočnej polohy s jednou nohou vpredu a zvyšok chôdze sme zostavili upravením tohto nájdeného kroku, aby vedel robot spraviť krok aj druhou nohou. Striedaním krokov každou nohou sme získali výslednú chôdzu.

Chôdzu pre robota sme vždy hľadali s predstavou, aby čo najvierohodnejšie reprezentovala chôdzu v reálnom svete. Preto by sa v budúcnosti dalo v práci pokračovať otestovaním nami nájdenej postupnosti polôh na reálnom robotovi Lilli so zistením, či bude vedieť robot Lilli chodiť v reálnom svete tak dobre, ako vo svete simulátora.

Možností na to pokračovať práci je viacero aj v rámci simulátora. Robota Lilli by bolo možné naučiť ďalším užitočným pohybom, predovšetkým pohybom, ktoré by viedli ku zastaveniu chôdze bez toho, aby robot stratil rovnováhu a spadol. Taktiež by bolo možné naučiť robota Lilli zabáčať počas chôdze do strán na rozdiel od priamej chôdze, ktorú sme sa snažili nájsť v tejto práci. Pomocou týchto pohybov a stereovidenia, ktoré by bolo nutné pridať do modelu robota Lilli v simulátore, by sa robot Lilli v budúcnosti vedel využiť na hľadanie optimálnej cesty v bludisku.

Cieľ našej práce bol úspešne splnený. Výslednou nájdenou postupnosťou pohybov vedel simulovaný robot chodiť po dvoch nohách a vedel dvadsiatimi krokmi prejsť vzdialenosť až 1,7 metra.

Použité zdroje

[1] CoppeliaSim: CoppeliaSim User Manual Version 4.5, 2023

[2] Halasi G.: Humanoid Robot Lilli, diplomová práca, Comenius University in Bratislava, 2020

[3] Russell S.J. & Norvig P.: Artificial Intelligence: A Modern Approach, Prentice Hall, 1995

[4] Fogel D.B.: An Introduction to Simulated Evolutionary Optimization, IEEE Transactions on Neural Networks and Learning Systems, vol. 5, no. 1, January, 1994

[5] Eshelman L.J. & Schaffer J.D.: Real-Coded Genetic Algorithms and Interval-Schemata, Foundations of Genetic Algorithms vol. 2, p. 187-202, 1993

[6] Villela L.F.C. & Colombini E.L.: Humanoid Robot Walking Optimization using Genetic Algorithms, diplomová práca, Universidade Estadual de Campinas, 2017

[7] SoftBank Robotics: NAO^6 User Guide, 2019

[8] Kosec T.: Algoritmy riadenia humanoidného robota, bakalárska práca, Comenius University in Bratislava, 2020

[9] Bzhikhatlov I. & Perepelkina S.: Research of robot model behaviour depending on model parameters using physic engines bullet physics and ODE, IEEE, 2017 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM)

Použité obrázky

[10] Personal Robotics (2023)

Dostupné na odkaze: https://personalrobotics.eu/robots-2/

Prílohy

Všetky súbory sú dostupné na githube: https://github.com/Robotics-DAI-FMFI-UK/cu-lIIIlI. Názov priečinka je *bipedal movement*.

V priečinku sa nachádzajú súbory: Lilli.ttt, v ktorom je uložená scéna a napísaný genetický algoritmus, niekoľko .ttm súborov obsahujúcich rôzne modely robota Lilli a súbor README. Všetky súbory sú open-source.