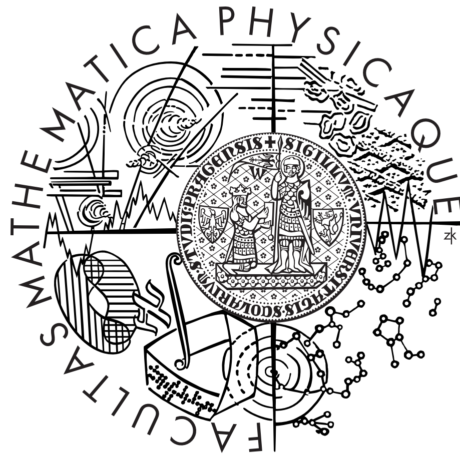
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Bc. Václav Obrázek

Vývoj chování inteligentních agentů

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Roman Neruda, CSc.
Studijní program: Informatika
Studijní obor: Teoretická informatika

Praha 2015

Poděkování

Rád bych zde touto cestou vyjádřil poděkování vedoucímu diplomové práce Mgr. Romanu Nerudovi, CSc. za jeho dobré rady, cenné připomínky a hlavně za čas, který mi věnoval.

Rád bych též poděkoval rodině a přátelům za jejich podporu při studiu.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

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 v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne .... Podpis autora

Název práce: Vývoj chování inteligentních agentů

Autor: Bc. Václav Obrázek

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Mgr. Roman Neruda, CSc.

Abstrakt: Tato práce se zabývá vývojem inteligentního chování agentů pro prostředí reálné počítačové hry pomocí evolučních algoritmů. Byla zvolena hra Unreal Tournament 2004, která má díky knihovně Pogamut dobrou podporu pro ruční tvorbu agentů. Jako řídící struktura pro agenty byly zvoleny yaPOSH reaktivní plány. Jelikož prostředí reálné hry kvůli časovým i hardwarovým nárokům není zcela vhodné pro účely umělé evoluce, bylo vytvořeno odlehčené prostředí LightEnv, které simuluje jen některé aspekty důležité pro vývoj agentů. Vývoj probíhal pomocí technik genetického programování s automaticky definovanými funkcemi upravených pro potřeby yaPOSH reaktivních plánů v prostředí LigthEnv. Vytvořená chování pro scénáře death match a team death match dokázala porazit pevně naprogramované strategie i po přenesení do hry Unreal Tournament 2004. V rámci scénáře team death match se podařilo vyvinout chování dobře využívající týmovou komunikaci.

Klíčová slova: evoluční algoritmy, genetické programování, inteligentní agenti, yaPOSH

Title: Evolution of behaviors for intelligent agents

Author: Bc. Václav Obrázek

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc.

Abstract: This thesis deals with agent behavior evolution for the environment of a real computer game using evolutionary algorithms. The game Unreal Tournament 2004 was chosen, due to its ease of use for creating agents manually with the Pogamut suite of tools. As a decision making structure for the agents yaPOSH reactive plans were used. Due to the demanding needs on the hardware and time a real computer game is not considered to be very suitable for artificial evolution. To overcome this fact a light-weighted environment LightEnv, that simulates only those aspects that are important for agent evolution, was created. The evolution was based on genetic programming modified for use with yaPOSH reactive plans. The evolved agent behavior for death match and team death match game scenarios exceeded the preprogrammed ones and was successfully transferred to Unreal Tournament 2004 environment. In the team death match scenario an interesting behavior that utilizes agent communication was evolved.

Keywords: evolutionary algorithms, genetic programming, intelligent agents, yaPOSH

Obsah

1 Úvod **3**
1.1 Struktura práce . . . 3
1.2 Příbuzné práce . . . 4

2 Teoretické aspekty vývoje chování inteligentních agentů **5**
2.1 Agent . . . 5
2.1.1 Architektura agenta . . . 6
2.1.2 Multiagentní systémy . . . 7
2.2 Behavior-Oriented Design . . . 7
2.3 Genetické programování . . . 11
2.3.1 Evoluční algoritmy . . . 11
2.3.2 Genetické programování . . . 12

3 Vývoj chování agentů **15**
3.1 Návrh . . . 15
3.1.1 Návrh LightEnv . . . 15
3.1.2 Návrh obecné evoluce . . . 17
3.1.3 Evoluce týmu agentů . . . 18
3.1.4 Genetické programování pro yaPOSH . . . 18
3.1.5 Infrastruktura . . . 21
3.2 Implementace . . . 22
3.2.1 LightEnv . . . 22
3.2.2 Evoluce . . . 24
3.2.3 Evoluce týmu agentů . . . 24
3.2.4 GP pro yaPOSH . . . 25
3.2.5 Infrastruktura . . . 25
3.2.6 Závěr . . . 26

4 Experimenty **27**
4.1 Experiment 1 on 1 DM . . . 27
4.1.1 Příprava . . . 28
4.1.2 Výsledky . . . 28
4.1.3 Validace v UT . . . 32
4.2 Experiment 2 on 1 TDM . . . 32
4.2.1 Příprava . . . 32
4.2.2 Výsledky . . . 32
4.2.3 Validace v UT . . . 35
4.3 Experiment 3 on 3 TDM . . . 35
4.3.1 Příprava . . . 36
4.3.2 Výsledky . . . 36
4.3.3 Validace v UT . . . 37

5 **Závěr** 40
5.1 Výsledky . . . 40
5.1.1 Vyvinutá chování . . . 40
5.1.2 LightEnv . . . 40
5.1.3 Přenositelnost chování . . . 41
5.1.4 Výkon . . . 41
5.2 Další rozvoj tématu . . . 42
5.2.1 Integreace s knihovnou Pogamut . . . 42
5.2.2 Využití Navigation Mesh . . . 42
5.2.3 Scénáře . . . 42
5.2.4 Tým . . . 42
5.2.5 Rozšíření metodiky BOD a POSH reaktivních plánů . . . 42

Seznam použité literatury 45

Seznam tabulek 47

Seznam použitých zkratek 48

Přílohy 49

1. Úvod

Cílem této práce je vyvinout inteligentní chování pro agenty ve virtuálním prostředí realistické počítačové hry. Pro tyto účely byla vybrána hra Unreal Tournament 2004 (UT), pro níž lze díky knihovně Pogamut [6] dobře vytvářet agenty ručně. Nicméně pro účely umělé evoluce již tolik vhodná není, velkými překážkami jsou:

- vysoký nárok na výkon - simulace probíhá v komplexním prostředí s propracovanou fyzikou,
- časová náročnost - simulace běží v reálném čase.

Pro překonání těchto překážek bylo navrženo a implementováno odlehčené prostředí LightEnv (LE), které simuluje jen některé aspekty důležité pro vývoj chování agentů. Zejména odpadá složitá simulace fyziky a 3D zobrazení.

Agenti budou řízeni pomocí POSH reaktivních plánů. Tato struktura poskytuje rychlou odezvu na podněty z prostředí, zároveň ale umožňuje vytvářet komplexní chování. Bude proveden návrh úprav technik evolučních algoritmů pro práci nad stromovou strukturou POSH reaktivních plánů a tyto budou implementovány. Fitness pro jedince bude získávána spouštěním agentů v LightEnv.

Vývoj agentů bude probíhat v prostředí LightEnv, pro validaci bude výsledek přenesen do Unreal Tournamentu 2004.

Někdy není česká terminologie ustálená, a občas ani vhodný překlad neexistuje, proto se v těchto případech uchýlíme k použití termínů anglických. Věříme ale, že čitelnost textu tímto neutrpí.

1.1 Struktura práce

Práce je strukturována do kapitol, kapitoly jsou děleny na sekce a případně podsekce tak, aby se zvýšila čitelnost textu i orientace v něm.

První kapitola obsahuje úvod a v části 1.2 jsou popsány příbuzné práce.

Kapitola 2 se věnuje teoretickým aspektům vývoje chování inteligentních agentů, popisuje současná řešení a zasazuje tuto práci do kontextu. Jelikož tato oblast je velice široká, omezíme se pouze na koncepty, technologie a prostředky, které jsou využity v dalších částech této práce.

V kapitole 3 popíšeme detaily návrhu a implementaci specifických nástrojů pro vývoj chování agentů. Jedná se o prostředí pro vývoj LightEnv, návrh umělé evoluce a celkové infrastruktury.

Kapitola 4 je věnována experimentům, kdy s použitím vytvořených nástrojů vyvíjíme inteligentní chování agentů. Součástí popisu jsou počáteční podmínky experimentu a validace v prostředí Unreal Tournament 2004.

V závěru (kapitola 5) shrneme výsledky a nastíníme další možný rozvoj tématu.

1.2 Příbuzné práce

Téma vývoje inteligentního chování agentů již bylo mnohokrát zpracováno, zde se pokusíme práci zasadit do širšího kontextu příbuzných prací. Výčet samozřejmě není úplný, snažíme se zacílit na práce využívající podobné nástroje a přístupy.

- Van Hoorn, Togelius a Schmidhuber ve svém článku [18] vyvíjejí agenty do hry Unreal Tournament 2004. Jimi zvolená architektura je hierarchická s inkrementálním učením jednotlivých modulů. Tato architektura si vedla lépe než referenční monolitická.

- Galli, Loiacono a Lanzi v [4] vytvářejí automatickou strategii pro výběr vhodné zbraně založenou na aktuální herní situaci v počítačové hře Unreal Tournament III.

- Witzany ve své práci [20] vyvíjí adaptivního protihráče do hry Unreal Tournament 2004 pomocí Q-Learning algoritmu. Výkon vyvinutých botů byl srovnatelný s ostatními podobnými implementacemi.

- Zelinka ve své práci [23] vytváří tým botů do hry Unreal Tournament 2004 pomocí nástroje yaPOSH knihovny Pogamut. Tento tým hraje scénář kradení vlajek – *Capture the Flag* (CTF). Nástroj yaPOSH se ukázal jako velmi vhodný, vytvořený tým byl úspěšnější než ostatní. Absence podpory pro paralelní chování v yaPOSH vedla na vytvoření části logiky v čisté Javě.

- Kadlec [9] vytváří boty do Unreal Tournamentu 2004 pomocí umělé evoluce. Nízkoúrovňové chování vyhýbání se nepřátelským projektilům je řízeno neuronovou sítí, rozhodování na vyšší úrovni zajišťuje strom chování, vytvořený pomocí techniky genetického programování.

Tato diplomová práce se věnuje vývoji inteligentního chování agentů v prostředí počítačové hry Unreal Tournament 2004. Řídící strukturou budou POSH reaktivní plány, stejně jako zvolil Zelinka v [23] pro ruční návrh CTF týmu. Agenty budeme vyvíjet pomocí technik umělé evoluce, v tomto směru kroky provedl Kadlec v [9], který ale nepoužil POSH reaktivní plány. Problém časové a výpočetní náročnosti simulace v kombinaci s nutností mnoha běhů pro evoluční algoritmy překonáváme vytvořením zjednodušeného prostředí LightEnv.

2. Teoretické aspekty vývoje chování inteligentních agentů

Tato kapitola se věnuje rešerši a popisu stávajících řešení relevantních k tématu této diplomové práce: *„Vývoj inteligentního chování agentů"*. Techniky zde popsané jsou použity a rozšířeny pro specifické použití v kapitole 3.

V první části popíšeme agenta, jeho vztah k prostředí a několik běžných architektur agenta. V druhé části se věnujeme metodice Behavior-Oriented Design (BOD) a návrhu agenta podle ní. Třetí část popisuje genetické programování.

2.1 Agent

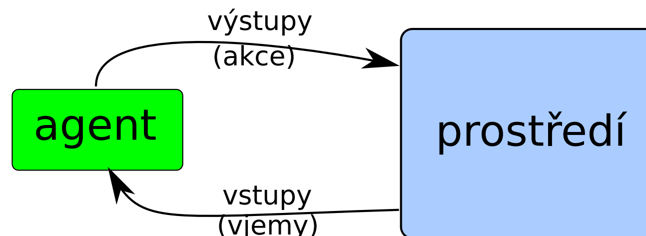
Přestože existuje mnoho významů pojmu *agent* (více Nwana [14]), zde budeme agenta chápat jako softwarový systém mající vlastnosti dle Wooldridge [22]:

1. **autonomie:** agent je schopen práce bez přímého řízení, má kontrolu nad svými akcemi,
2. **sociální interakce:** agent může interagovat s jinými agenty nebo lidmi,
3. **reaktivita:** agent reaguje na vstupy (vjemy) z prostředí pomocí výstupů (akcí), které mohou prostředí ovlivnit,
4. **intencionalita:** agent má dlouhodobý cíl, kterého se snaží dosáhnout.

Tato definice klade důraz i na *prostředí*, v rámci něhož agent akce vykonává, a na jejich vzájemný vztah. Agent získává vstupy (vjemy) z prostředí a pomocí svých výstupů (akcí) prostředí ovlivňuje. Tedy nelze zcela oddělit agenta od prostředí. Vztah agenta a prostředí je zobrazen na obrázku 2.1. Obecně vzato agent nemusí mít vlastní reprezentaci, nemusí tedy existovat entita v prostředí, která by agenta představovala.

Obvykle očekáváme dynamické prostředí, tedy takové, které se mění samo a ne jenom prostřednictvím akcí agenta.

Agent zasazený do počítačové hry, který má reprezentaci v prostředí hry, se obvykle nazývá *bot*.



Obrázek 2.1: Vztah agenta a prostředí.

2.1.1 Architektura agenta

V této části popíšeme některé architektury agentů, na které navážeme v další části metodikou BOD.

Reaktivní architektura Reaktivní architektura poskytuje agentovi spíše základní funkcionalitu tak, aby splňoval výše zmiňované vlastnosti, hlavní důraz je kladen na reaktivitu. Tato architektura se vyznačuje svojí rychlou odezvou na podněty.

Agent vybírá akci na základě vstupů z prostředí a vnitřního stavu. Nicméně neplánuje dopředu a ani nemá model prostředí.

Mezi reaktivní architektury se řadí například rozhodovací stromy v podobě *if-then* pravidel nebo dopředné neuronové sítě. Přestože oba tyto příklady jsou principiálně odlišné, mají některé podobné vlastnosti, zejména rychlou odezvu na podněty z prostředí.

Deliberativní architektura Tato architektura umožňuje agentovi plánovat dopředu. Agent si zpravidla udržuje vlastní reprezentaci světa, včetně svých znalostí o něm.

Tato architektura umožňuje vykonávat i složité úkoly, avšak může být výpočetně složitější.

Jako příklad můžeme uvést agenty využívající STRIPS plánovače nebo zejména v případě multiagentních systému oblíbené modely Belief-Desire-Intention (BDI).

Hybridní architektury Hybridní architektury kombinují výhody reaktivních i deliberativních architektur. Poskytují rychlou odezvu, ale umožňují i složitější plánování. Základem je rozdělení do vrstev, které zpravidla zajišťují určitou vymezenou funkcionalitu.

Vrstvy mohou být uspořádány *horizontálně* nebo *vertikálně*. V horizontálním uspořádání jsou všechny vrstvy připojeny na vstupy i výstupy, efekty jednotlivých vrstev se skládají dohromady. V případě vertikálního uspořádání je pouze jedna vrstva připojena na vstupy a výstupy. Informace se pak propagují mezi vrstvami.

V současné době často užívanou technikou je některá z forem hybridní třívrstvé architektury ([5],[8]). Základem je rozdělení do tří vrstev:

- Reaktivní vrstva.
  Cílem reaktivní vrstvy je poskytnout velmi rychlou odezvu na vnější podnět. Moduly v této vrstvě většinou nemívají vnitřní stav. Implementačně se může jednat o prosté mapování hodnot s prahem ze vstupů na výstupy.

- Sekvenční vrstva.
  Tato vrstva zajišťuje vykonávání abstraktnějších akcí z vrstvy plánovací.

- Plánovací vrstva.
  Zde máme na mysli abstraktnější plánování, které velmi často může být i časově náročné. Agent může vytvářet složité plány do budoucna tak, aby plnil své cíle.

Tato architektura má dobré vlastnosti, agent je schopen rychle reagovat v případě potřeby - řízení převezme reaktivní vrstva, ale i vytváří a sleduje složitější abstraktní plány do budoucna. Její nevýhodou je ovšem složitá implementace, je třeba řešit komunikaci mezi vrstvami, a každá z nich navíc používá jiné technologie i paradigmata.

Subsumpční architektura Tato architektura, jak ji popisuje Brooks v [2], staví na relativně jednoduchých nezávislých modulech, které je snadné navrhnout. Tyto moduly jsou uspořádány do vrstev, které určují jejich hierarchii. Každá vrstva se snaží zajistit některý z agentových cílů, přičemž vyšší vrstva může pomocí úprav vstupů a výstupů zahrnout cíl vrstvy nižší.

2.1.2 Multiagentní systémy

Multiagentní systém (MAS) je systém, v němž určitý počet agentů interaguje. Dle definice agenta v sekci 2.1 interakce může probíhat jedině přes sdílené prostředí. Zejména tedy komunikace mezi agenty by měla být do jisté míry omezená (a to i v rámci týmu), tak aby rozhodovací procesy v rámci agentů byly dostatečně autonomní. [15]

Agenty v MAS lze sdružovat do týmů. Scénář (úloha v prostředí) pak může klást cíle pro týmy spíše než pro jednotlivé agenty.

Obecně můžeme scénáře, které agenti mají řešit, rozdělit na:

- **kooperační:** agenti mohou dosáhnout svých cílů lépe pomocí spolupráce,
- **kompetitivní:** agenti se snaží dosáhnout rozdílných cílů, které se vylučují.

Příklady MAS i vhodných úloh uvádí Vidal v [19].

2.2 Behavior-Oriented Design

Metodika Behavior-Oriented Design (BOD), jak ji popisuje Bryson v [3], je vysoce modulární metodika pro tvorbu inteligentních agentů jako jsou virtuální bytosti. V této části popíšeme tuto metodiku, jak se odlišuje od jiných rozšířených technik a v čem tkví její výhody. Je potřeba zdůraznit, že velkou roli hraje nejen kvalita výsledků – chování agentů, ale i nenáročnost popisu tohoto chování v dané architektuře. Architektura by měla umožnit snadnou tvorbu i velmi komplexního chování. V tomto tkví silná stránka BOD.

BOD vychází částečně z architektur uvedených výše, ale zejména staví na principech převzatých z paradigmatu objektově orientovaného návrhu [3]. Základním stavebním kamenem jsou zapouzdřené moduly obsahující veškeré vstupy, akce, učení i paměť a mohou tedy obsahovat i velmi složitou logiku. Tyto moduly jsou seskupeny do hierarchické stromové struktury ne nepodobné reaktivním plánům. Celek je tedy reaktivní, na daný podnět ale lze odpovědět i komplexním chováním.

Dále je kladen důraz na využití standardních programovacích jazyků za použití objektového návrhu. Oproti třívrstvým architekturám úplně chybí ekvivalent reakční bezstavové vrstvy.

Návrh agenta Základním principem při návrhu agenta podle metodiky BOD je iterativní dekompozice problému. Na nejvyšší úrovni je potřeba rozhodnout a pojmenovat, co má agent dělat. Tato chování je nutno postupně dekomponovat až na jednotlivé akce a vjemy. Akce a vjemy ve smyslu BOD zapouzdřují vstupy a výstupy z prostředí a mohou přidávat další netriviální logiku.

Proces návrhu komplexních chování je pomocí metodiky BOD velmi snadný a přehledný. Výsledné plány je možné snadno reorganizovat a dle potřeby upravovat.

POSH Metodika BOD sama o sobě neobsahuje implementaci, může využívat POSH (Parallel-rooted, Ordered, Slip-stack Hierarchical) dynamické reaktivní plány [1].

Kromě základních akcí (*action*) a vjemů (*sense*) je stromová struktura POSH popsána pomocí:

- **drive collection** (DC) Drive collection představuje nejvyšší úroveň stromu a sdružuje základní obecná chování, která označujeme termínem *drive* (D). V každém kroku se agent musí rozhodnout, které chování z DC má vykonávat. Toto dobře podporuje rychlost odezvy agenta, ale na rozdíl od třívrstvé architektury, přepíná se na vysoké úrovni a tudíž výsledné chování může být i velmi komplexní.

  Jako modelový příklad lze uvést jednoduchý scénář *predátor-kořist*. V tomto scénáři se v prostředí pohybují dva agenti, *predátor* a *kořist*. Kořist se snaží sbírat *potravu*, která se vyskytuje náhodně v prostředí. Predátor se snaží ulovit kořist. Pokusíme se navrhnout agenta kořist pomocí BOD.

  Kořist se pohybuje v prostředí a hledá potravu. Toto by odpovídalo jednomu chování v rámci DC, akce může implementovat i sofistikované hledání potravy na neprozkoumaných místech. Pokud však kořist uvidí predátora, utíká se schovat. V DC by toto chování mělo mít vyšší prioritu. Samotná akce provádějící schování může například vypočítat nejkratší cestu a podobně. Pokud přidáme paměť, bude výsledné chování: *pokud kořist nedávno zahlédla predátora, utíká se schovat.* Vjem, který toto chování *schovávej se* spouští, je vjem *byl nedávno zahlédnut predátor* a prováděná akce je *schovej se.* Toto chování lze již považovat za inteligentní. Velkou výhodou BOD je, že náš slovní popis lze velmi dobře převést na reálný plán, vizte obrázek 2.2 a 2.3 na straně 10 ukazující výsledný plán v grafickém zobrazení a v textové podobě.

  Rovněž se pohybujeme na vysoké úrovni abstrakce, konečná implementace je v rámci akcí a vjemů.

- **action pattern** (AP) Action Pattern bychom česky mohli nazvat *vzorec chování*, technicky se jedná o sekvenci akcí. Jako příklad v našem prostředí s predátorem a kořistí můžeme uvést sekvenci dvou jednoduchých akcí *dojdi k potravě*, *sněz potravu.* Tento AP bychom mohli zařadit do DC jako chování *sháněj potravu* spouštěné vjemem *vidíš potravu.* V rámci hierarchie v DC by toto chování mělo být na druhém místě za *schovávej se* a před *prohledávej prostředí.*

- **competence** (C) Kompetence jsou svojí strukturou velmi podobné DC. Mohou tvořit druhé a další patro stromu a nejsou tedy vyhodnocovány úplně pokaždé. V terminologii POSH plánů položky z *competence* označujeme *competence element* (CE), česky *kompetence* a *prvky kompetence*.

  V našem příkladu predátor-kořist můžeme pomocí kompetence rozšířit chování, které zajišťuje schovávání před predátorem. Pokud predátor kořist zahlédl, měla by se dát na rychlý ústup. Pokud se ale predátor dívá jiným směrem, kořist se zkusí nenápadně odplížit.

  Výsledný plán pro příklad predátor-kořist je zobrazen v grafické podobě na obrázku 2.2 a v textové podobě na obrázku 2.3.

Aby bylo možné zajistit dostatečně rychlou odezvu, je třeba, aby akce byly snadno přerušitelné, pokud dojde k přepnutí chování v DC. Paralelní vykonávání akcí v základu není, korektní POSH reaktivní plán nám určí jednu akci.

Pokud je to možné a kontext prostředí to dovolí, vykonávání přerušené akce může být znovu spuštěno nikoliv od začátku, ale od místa přerušení.

yaPOSH Implementace POSHe podle BOD v knihovně Pogamut [6] se jmenuje yaPOSH (yet another POSH). Z uživatelského hlediska je k dispozici plugin pro vývojové prostředí (IDE) NetBeans [13], který zajišťuje vizualizaci plánu a umožňuje i jeho úpravy. Výsledný plán pro kořist z předešlého příkladu je graficky znázorněn na obrázku 2.2 a jeho odpovídající textová verze na obrázku 2.3. Oboje bylo vytvořeno pomocí pluginu v NetBeans.

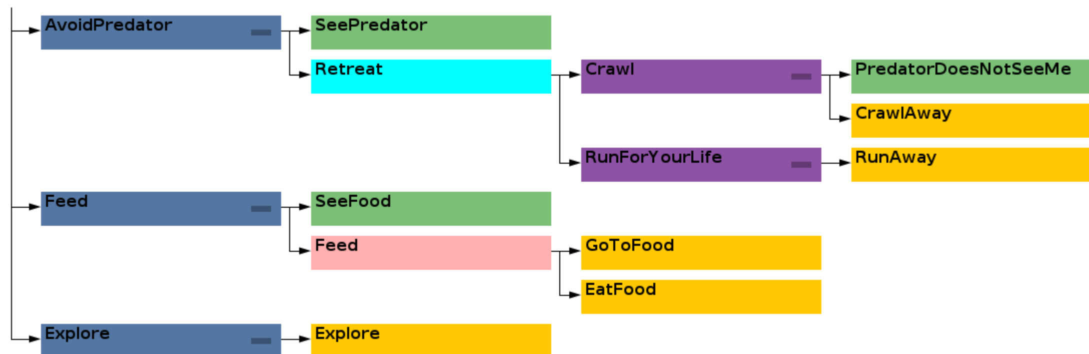
Definice kompetencí (C) a vzorců chování (AP) jsou v rámci popisu plánu 2.3 mimo hlavní strom DC a jsou z něj jen odkazovány. Rovněž lze jednotlivé AP a C odkazovat mezi sebou, pokud ovšem nevznikne cyklus. Toto výraznou měrou přispívá k snadnému návrhu i úpravám a zvyšuje znovupoužitelnost kódu.

Grafický nástroj navíc podporuje práci s plánem pomocí uživatelsky přívětivé metody Drag and Drop. Kořen stromu, drive collection, není v rámci grafického pulginu zobrazen. Nejvyšší úroveň tedy tvoří chování (drive), zobrazené modrou barvou. Vjemy jsou odlišeny zelenou, akce žlutou. Vzorce chování (action pattern) IDE zobrazuje barvou lososovou. Kompetence se skládají ze dvou vrstev, první je jméno kompetence, barva světle modrá, druhá vrstva obsahuje jednotlivé prvky kompetence (competence element), které jsou obarveny na fialovo. Oproti textové reprezentaci ukazuje tento grafický nástroj plán plně rozvinutý.

Akce jsou v Pogamutu vyhodnocovány s výsledkem:

- *RUNNING_ONCE* akce proběhla a skončila,
- *RUNNING* akce proběhla ale během jednoho cyklu logiky ještě neskončí, tedy při dalším vyhodnocení plánu bude pokračovat,
- *FINISHED* akce skončila, plánovač pokračuje ve vyhodnocování.

Při každém cyklu logiky (vyhodnocení plánu) se hledá akce, která se má vykonat. Tyto cykly jsou v Pogamutu od sebe vzdáleny přibližně 250 ms.

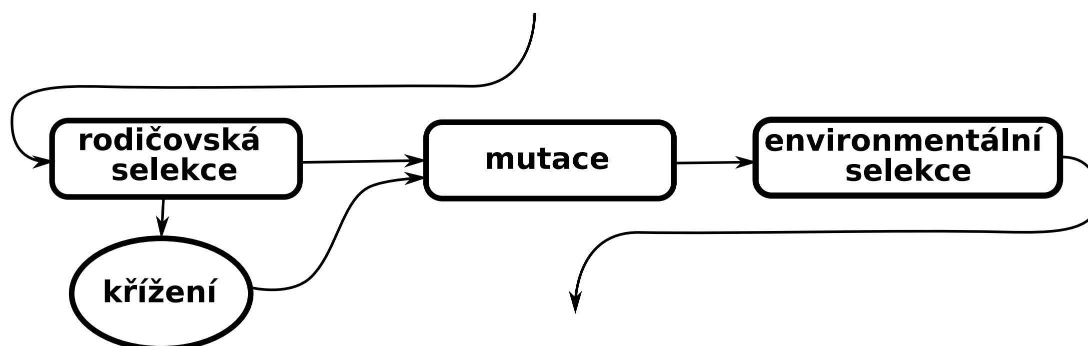


Obrázek 2.2: Ukázka plánu pro problém predátor-kořist z pluginu IDE NetBeans

```
(
        (C Retreat
                (elements
                        ((Crawl (trigger ((PredatorDoesNotSeeMe))) CrawlAway))
                        ((RunForYourLife RunAway))
                )
        )
        (AP Feed (GoToFood EatFood))

        (DC life
                (drives
                        ((AvoidPredator (trigger ((SeePredator))) Retreat))
                        ((Feed (trigger ((SeeFood))) Feed))
                        ((Explore Explore))
                )
        )
)
```

Obrázek 2.3: Ukázka textové reprezentace plánu pro problém predátor-kořist



Obrázek 2.4: Generační krok

2.3 Genetické programování

Termín genetické programování (GP) označuje soubor specifických technik a postupů pro prohledávání prostoru potenciálních řešení různých úloh a řadí se mezi evoluční algoritmy. V této sekci nejprve obecně popíšeme evoluční algoritmy a poté se zaměříme speciálně na genetické programování.

2.3.1 Evoluční algoritmy

Evoluční algoritmy jsou heuristické prohledávací algoritmy, u kterých je patrná silná inspirace přirozenou evolucí. Pracujeme zde s pojmy jako je *populace*, *jedinec*, *křížení*, *mutace*, *selekce*.

Jedincem rozumíme kandidátské řešení daného problému, resp. předpis pro sestavení takového řešení. Množinu jedinců nazýváme populace. Proces prohledávání označujeme *vývoj* a probíhá v krocích zvaných *generace*.

Populace bývá iniciována náhodnými jednici, představující náhodné body prohledávaného prostoru.

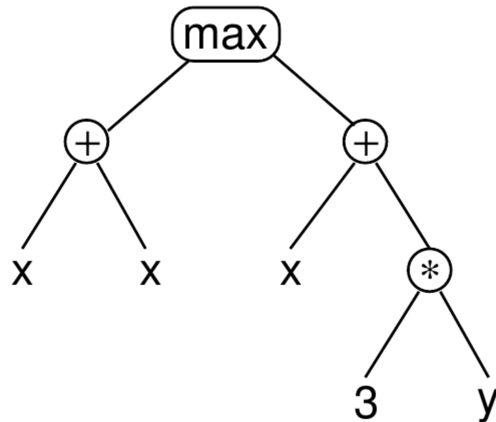
Mezi dvěma jednotlivými generacemi dochází k úpravám populace pomocí mutací, selekce a křížení. Tyto nazýváme souhrnně *operátory*. Abychom zajistili správný běh evoluce, každého jedince ohodnocujme pomocí funkce zvané fitness. Tato funkce udává kvalitu jednice. Pro jedince reprezentujícího optimální řešení problému by měla být hodnota fitness samozřejmě nejvyšší.

Selekce (výběr jedinců z populace) probíhá na základě hodnoty fitness. Jedince vybíráme pro křížení a pro postup do další generace, tedy kvalitní jedinci by měli mít větší šance na rozmnožení a i na přežití. Obvyklý generační krok s rodičovskou selekcí, křížením, mutacemi a environmentální selekcí je na obrázku 2.4.

Mezi obvyklé selekce patří *turnajová selekce* a *ruletová selekce*. V případě turnajové selekce vybíráme opakovaně z populace dva nebo více jedinců, ze kterých postupuje ten nejlepší, tedy ten s nejvyšší fitness. Ruletová selekce vybírá jedince s pravděpodobností úměrnou velikosti jejich fitness.

Prohledávání prostoru zajišťuje křížení a mutace. Křížením vytváříme zpravidla dva nové jedince ze dvou rodičů. Předpokládá se, že kombinací kvalitních rodičů, mohou vzniknout kvalitní potomci. Náhodnost přidávají mutace, které mohou provést lokální změnu uvnitř jedince.

Elitismus je způsob jak zachovat velmi kvalitní jedince napříč generacemi



Obrázek 2.5: Syntaktický strom. Převzato z [16].

tak, aby nebyli ohroženi náhodnými mutacemi nebo křížením. Určitá malá část populace, obvykle nejlepší jedinci, je beze změny přesouvána do další generace. Tito jedinci se mohou i nadále účastnit křížení.

Evoluční algoritmy představují přírodou inspirované heuristické prohledávací algoritmy, které fungují dobře v celé řadě úloh. Podstatnou roli hraje i prvek náhody, který umožňuje opustit suboptimální řešení.

2.3.2 Genetické programování

V této části rozebereme specifika genetického programování (GP) s ohledem na evoluční algoritmy, jak byly popsány v části 2.3.1. Kvalitní úvod do problematiky genetického programování je Field Guide to Genetic Programming [16], do větší hloubky se GP věnuje Koza v [10].

Genetické programování je speciální technika spadající do oblasti evolučních algoritmů. Jedincem je program obvykle reprezentovaný pomocí syntaktického stromu. Operátory jsou uzpůsobené pro práci nad touto strukturou. Je vhodné zdůraznit na úvod, že na rozdíl od tradičních EA nemusí mít jedinec v GP předem danou fixní velikost, právě naopak, v průběhu evoluce se do určité míry podporuje nárůst velikosti jedinců.

Jedinec Jedinec v GP je program, který je reprezentován v podobě syntaktického stromu. Strom tvoříme dynamicky pomocí operátorů ze základních prvků, nazývaných *primitiva.*

Primitiva Primitiva rozdělujeme na *terminály* a *neterminály* (funkce). Ukázku syntaktického stromu pro výraz $max(x + x, x + 3 * y)$ naleznete na obrázku 2.5. Terminály jsou $x, 3, y$ a funkce $max, +, *$ všechny s aritou 2. Samozřejmě funkce mohou mít různou aritu a je třeba zajistit typovou správnost, tak aby vznikal korektní syntaktický strom. V tomto příkladu pracujeme jen s číselným typem, takže je vše v pořádku.

Mutace Mutace v GP lze rozdělit podle toho, zda mění stromovou strukturu nebo ne. Mezi měnící patří:

- nahrazení podstromu novým podstromem,
- nahrazení podstromu terminálem,
- nahrazení terminálu podstromem,
- prohození (swap) dvou podstromů.

Mutace, které strukturu nemění, jsou např:

- úprava konstanty,
- nahrazení terminálu jiným terminálem,
- nahrazení funkce jinou funkcí se stejnou aritou.

Aby vznikl validní jedinec je vždy potřeba zajistit typovou kontrolu.

Křížení V tomto odstavci popíšeme jednoduché jednobodové křížení, samozřejmě složitější varianty jsou možné. Do jednobodového křížení vstupují dva rodiče a vznikají dva potomci. U tohoto křížení dochází k výměně náhodně vybraných podstromů rodičů. Opět je potřeba zachovat typovou správnost.

Selekce Selekce probíhá standardně na základě fitness.

Bloat Stromová struktura jedince umožňuje (teoreticky) neomezenou velikost, takže potenciálně mohou vznikat i velmi komplexní programy. Toto chování je žádoucí. Nicméně kvůli prvku náhody mohou po několika generacích začít vznikat části programu (podstromy jedince), které mohou mít na výsledek jedince jen malý, někdy dokonce žádný, vliv. Takovéto na první pohled zbytečné části kódu se označují anglickým termínem *bloat*[1].

Nadměrná tvorba bloatu může mít negativní vliv na rychlost výpočtu, proto lze zařadit operátory pro jeho odstraňování nebo příliš velké jedince znevýhodnit pomocí snížení jejich fitness.

Určitý nárůst velikosti je nicméně nutný, protože zpravidla hledáme netriviální řešení. Po skončení evoluce se jedinci často ručně upravují do lépe čitelné podoby.

ADF Rozšíření základního GP je možné například pomocí automaticky definovaných funkcí (ADF) [11]. Některé části výpočtu se mohou často opakovat, to obvykle řešíme deklarací funkce. ADF umožňuje automaticky vytvářet takové funkce v rámci GP.

Prakticky libovolný podstrom je možno vzít a uložit jako ADF, která následně funguje jako nová funkce nebo terminál. Na úrovni jednice nebo populace tedy definujeme množinu F, která je z počátku zpravidla prázdná, prvky z této množiny nazýváme automaticky definované funkce. Do množiny F prvky (podstromy)

[1] Český termín pro bloat není ustálen.

v průběhu výpočtu přidáváme heuristicky nebo náhodně. Tyto prvky $f \in F$ nazýváme automaticky definované funkce[2]. Všechny operátory nyní pracují nejen na množinách terminálů a neterminálů, ale mohou stejným způsobem využít i množinu F automaticky definovaných funkcí.

ADF přinášejí možnost znovupoužití kódu a to zcela automaticky. Navíc je lze sdílet mezi jedinci, ať už přímo nebo prostřednictvím speciálních křížení nebo mutací nebo použitím množiny automaticky definovaných funkcí sdílené na úrovni celé populace.

2.3.2.1 Úspěchy GP

Genetické programování bylo úspěšné v celé řadě úloh, Koza [12] uvádí více-ro příkladů, mezi nimi například systém pro návrh analogových zesilovačů, kdy vytvořená schémata byla po vyčištění od bloatu často shodná s patentovanými komerčně úspěšnými návrhy zesilovačů.

Sipper [17] se s úspěchem zabývá deskovými hrami, simulacemi a hlavolamy, uvádí mimo jiné úspěchy virtuálního robota vytvořeného pomocí GP v *HaikuBot leage* v soutěži Robocode. Vytvořený robot porazil všechny lidmi vytvořené a skončil na prvním místě.

Důležité faktory pro úspěšné použití GP, jak je uvádí Koza v [12], jsou:

1. není dobře znám význam vstupních parametrů,
2. řešení má neznámou velikost a strukturu,
3. máme k dispozici velké množství testovacích dat,
4. existuje dobrý simulátor, ale řešení nelze z něj přímo získat,
5. konvenční matematické metody nefungují,
6. stačí aproximace,
7. malé zlepšení je považováno za úspěch.

Genetické programování staví na evolučních algoritmech, vyvíjení jedinci jsou ale programy pro tvorbu řešení dané úlohy v podobě syntaktických stromů. Tato neobvyklá struktura umožňuje dosáhnout velmi dobrých výsledků.

[2]Terminál je funkce s nulovou aritou.

3. Vývoj chování agentů

V této kapitole specifikujeme detaily zadání a popíšeme požadavky a nároky, které budou kladeny na řešení. V první sekci se budeme věnovat návrhu a popisu vhodných technik založených na technikách prezentovaných v kapitole 2. Následovat bude sekce popisující implementaci jednotlivých řešení. Podrobné technické detaily naleznete v příloze 1.

3.1 Návrh

Tato sekce se věnuje návrhu řešení pro evoluci chování agentů - botů do hry Unreal Tournament 2004. Pro evoluci využijeme techniky založené na genetickém programování, pro zvýšení rychlosti a usnadnění simulace bude vytvořeno zjednodušené prostředí napodobující UT, nazvané *LightEnv*. Plány vzniklé evolucí bude možné snadno spouštět v obou prostředích.

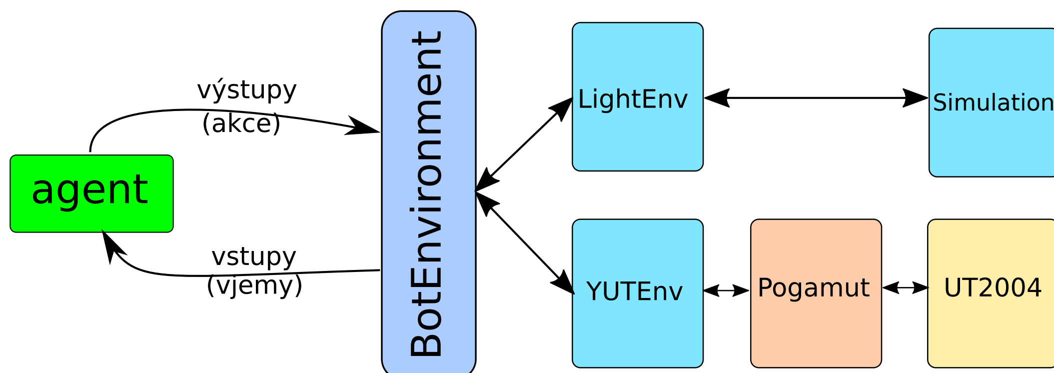
Na obrázku 3.1 je zobrazen celkový návrh architektury ve vztahu k agentovi. Lze porovnat s obrázkem 2.1 na straně 5 ilustrujícím v rámci definice agenta jeho vztah s prostředím.

Agent přijímá vstupy z univerzálního abstraktního prostředí *BotEnvironment*, které unifikuje vstupy z jednotlivých prostředí. Akce jsou propagovány stejným způsobem zpět. Pro agenta je tedy nerozlišitelné, které konkrétní prostředí je právě používáno, z jeho pohledu se vždy jedná o pevně dané *BotEnvironment*.

Při převodu akcí a vjemů z univerzálního *BotEnvironment* do konkrétního prostředí (*LightEnv*, *YUTEnv*) je třeba zachovat věrně logický význam i vlastnosti tak, aby chování agentů v obou prostředí bylo co možná nejvíce podobné. Tomuto požadavku je podroben návrh *LightEnv*.

3.1.1 Návrh LightEnv

Návrh prostředí reflektuje potřeby projektu, tedy běh simulace má být rychlý, ale prostředí zároveň musí poskytovat dostatečně bohatou škálu vjemů a akcí, které navíc musí do značné míry odpovídat prostředí Unreal Tournament 2004.



Obrázek 3.1: Návrh prostředí a vztah k agentovi

Prostředí nemusí obsahovat pokročilou fyziku, ani 3D zobrazení světa, nicméně by mělo poskytnout agentovi (botovi) podobné vjemy jako prostředí Unreal Tournament 2004. Bot funguje na určitém stupni abstrakce, například pro orientaci využívá navigační graf[1] místo přímé vizuální percepce prostředí, a tedy funkční mechanismus navigace založené na navigačním grafu by měl být dostačující.

Pogamut a UT Většinu prostředků pro práci s agentem zajišťuje knihovna Pogamut [6], jsou to zejména moduly navigace a viditelnost, kde se nevyžívá prostředí UT přímo. Navigace je z UT pouze inicializována, viditelnost dokonce používá předzpracovaná data. V rámci LightEnv josu tedy tyto moduly znovu implementovány. Simulace prostředí probíhá v UT, včetně fyziky, boje a zobrazování. LightEnv implementuje tyto prvky zjednodušeně.

Porovnání vlastností prostředí můžete nalézt v tabulce 3.1 na straně 17.

Zobrazení Prostředí LightEnv pro účel vývoje i debuggingu může být zobrazeno. Tento pohled stačí dvojrozměrný z ptačí perspektivy. Prostředí rovněž může být spuštěno bez jakéhokoliv zobrazování.

Fyzika V LE není třeba simulovat fyziku, pro pohyb lze vyžít informace z navigačního grafu. Čas na rozdíl od UT nemusí plynout pseudospojitě, stačí nám diskrétní body, které odpovídají časům evaluace plánu.

Pohyb, navigace, viditelnost Pohyb mimo navigační graf je podporován, ale jen v omezené míře. Bot se může pohybovat po přímce mezi dvěma přímo spojenými navigačními body. Bot tedy není vázán jen na navigační body, protože ty mohou být i daleko od sebe, a v každém kroku simulace chceme vědět jeho přesnou polohu. Není řešena detekce kolizí. Rychlost pohybu by měla odpovídat UT, tedy přesun z jednoho navigačního bodu do jiného by měl trvat přibližně stejnou dobu v obou prostředích.

Navigace může probíhat po navigačním grafu, který je zkonstruován na základě exportu bodů z navigace z knihovny Pogamut. Chování navigace by mělo být podobné, zejména nalezené trasy by se neměly lišit.

Viditelnost by měla co nejvíce reflektovat stav v UT, opět lze vyexportovat informace o viditelnosti z Pogamutu.

Předměty Ve hře UT je velké množství předmětů, budeme simulovat pouze tři kategorie: *lékárnička*, *štít* a *zbraň*. Tyto předměty se většinou nachází na navigačních bodech, jediná výjimka jsou upuštěné předměty. Je třeba umožnit botům v LE předměty sbírat a využívat jejich vlastností, očekáváme, že to může mít zásadní vliv na kvalitu chování jedinců. Předměty se též po určité době znovu objevují ve hře.

Sbírání předmětů se děje automaticky, pokud bot proběhne daným navigačním bodem a předmět může sebrat (např.: nemá plné zdraví v případě lékárničky).

[1] Týká se Pogamutu verze 3.6.1 z května 2014, nová verze 3.7.0 z února 2015 přidává podporu pro Navigation Mesh

Boj Jisté zjednodušení boje je též žádoucí, není třeba simulovat různé typy projektilů. Měla by ale zůstat rozdílná síla zbraní, jejich dostřel a velikost zásobníku. Předpokládáme funkční mechanismus štítu a zdraví.

Komunikace Jak již bylo diskutováno v kapitole 2, stačí omezená komunikace pro boty, zde tedy zasílání předdefinovaných zpráv.

POSH Boti by měli být řízeni pomocí POSH plánu, který bude fungovat jak v LE tak v UT s Pogamutem. Pro evaluaci i reprezentaci plánu lze využít implementaci z knihovny Pogamut, což zajistí stejné vlastnosti, co se plánů týče, v obou prostředích.

Knihovna Pogamut umožňuje do jisté míry vytvářet paralelně vykonavatelné akce. Jedná se například o akce *StartShooting*, která způsobí, že bot bude střílet až do zavolání akce *StopShooting*, nebo všechny akce, které využívají navigaci, jež bota naviguje v každém cyklu i bez volání příslušné akce. Toto neodpovídá dobře metodice BOD a definici agenta v předešlé kapitole, proto tyto akce nebudeme používat. Dle požadavků bude bot vykonávat vždy jen jednu akci, nejen na úrovní POSH reaktivních plánů, ale i na úrovni prostředí. Detailně je problematika paralelizace akcí rozebrána v části 5.2.5 na straně 42.

	Unreal Tournament 2004 + Pogamut	**LightEnv**
čas	pseudospojitý	diskrétní
zobrazení	3D	2D
pohyb	v prostoru	po grafu
navigace	graf	graf
viditelnost	graf	graf
boj	plná simulace	omezené
komunikace	textové zprávy / serializované Java objekty	předdefinované zprávy

Tabulka 3.1: Porovnání prostředí

3.1.2 Návrh obecné evoluce

Implementace obecných evolučních algoritmů bude vycházet z popisu v části 2.3.1 na straně 11. Základem je *populace* složená z *jedinců*. Vše, co pracuje s populací, bude pro nás *operátor*. Z tohoto konceptu budeme vycházet.

Operátory Na obecné úrovni je vše, co pracuje s populací, operátor. Klademe důraz na to, aby operátory byly co možná nejjednodušší a měli jasně vymezenou funkcionalitu.

Můžeme vytvořit sadu operátorů i bez znalosti konkrétní implementace jedinců, které budou zajišťovat:

- **výběr jedinců na základě fitness** (environmentální selekce) - tento operátor vybere z populace nejlepších k jedinců dle hodnoty fitness, ostatní odstraní,

- **výpis populace a fitness pro účely ladění i logování** - předpokládáme výpis statistik do souboru. Zajímá nás fitness nejlepšího jedince, průměrná fitness. Samozřejmě lze rozšířit o výpis dalších informací,
- **aplikace mutací** - operátory pracují s populací, mutace ale s jedinci, abychom zachovali čistotu návrhu, vytvoříme operátor pro aplikaci mutace,
- **elitismus** - tento operátor bude zajišťovat zachování elity v rámci populace, kdy část kvalitních jedinců bude přenesena do další generace.
- **aplikaci křížení na vybrané jedince** - podobně jako mutace, křížení pracuje s jedinci, tento operátor umožní výběr jedinců (rodičovská selekce) a jejich křížení. Konkrétní implementace opět závisí na jedincích.

Fitness bude reprezentována číselnou hodnotou vyjadřující kvalitu jedinců na základě jejich chování v prostředí. O její výpočet se bude starat speciální operátor, který ji nastaví.

Generační krok Vše, co mění populaci jedinců, je pro nás *operátor*. V rámci generačního kroku se postupně aplikují všechny specifikované operátory na populaci. Díky tomuto lze vytvořit prakticky libovolný generační krok, který nemusí odpovídat tradičnímu pojetí, jak je zobrazeno v předešlé kapitole na obrázku 2.4 na straně 11.

3.1.3 Evoluce týmu agentů

Zde navrhneme specifika evoluce pro tým agentů s ohledem na návrh evoluce výše. Agenti v rámci týmu mohou být buď homogenní nebo heterogenní.

Jedinec ve smyslu evolučních algoritmů je tým agentů. Pro výpočet fitness ovšem potřebujeme informace nejen z týmu, ale i z konkrétních agentů. Informace na úrovni týmu může být třeba počet ukradených vlajek ve hře Capture the Flag (CTF), na úrovni agenta pak uběhnutá vzdálenost nebo počet zastřelených nepřátel. Tyto informace je třeba agregovat do celkové fitness týmu.

Mutace budou probíhat na úrovni agentů. Křížení rovněž, i když pro heterogenní tým lze přidat speciální operátory, které budou vyměňovat jedince mezi týmy.

3.1.4 Genetické programování pro yaPOSH

Agenti - boti budou řízeni pomocí POSH reaktivních plánů. Pro účely evoluce těchto plánů byly zvoleny techniky vycházející z genetického programování.

Struktura plánu Řízení botů zajišťují POSH reaktivní plány. Plán má sice logickou strukturu stromu, ale implementačně se jedná a sadu mělkých zakořeněných stromů. Grafický nástroj v IDE Netbeans zobrazuje logickou stromovou strukturu, vizte obrázek 2.2 na straně 10, implementačně však pracujeme s vícero paralelně zakořeněnými stromy, jejichž textovou reprezentaci ukazuje obrázek 2.3 na straně 10. Opakovaně lze použít daný AP nebo C a změna v grafickém editoru na jednom místě se projeví na všech ostatních.

Drive Collection	Competence	Action Pattern	Trigger
DC -> D DC -> TD D -> a D -> a D D -> a TD TD -> T D	C -> CE C -> TCE CE -> a CE -> a CE CE -> a TCE TCE -> T CE	AP -> AE AE -> a AE -> a AE	T -> s T -> s T

Tabulka 3.2: Gramatiky popisující strukturu neterminálů

Z hlediska implementace se tedy nejedná o jeden strom, tak jak je zobrazen v grafickém editoru, ale máme sadu mělkých paralelně zakořeněných stromů, což musíme reflektovat v návrhu GP pro yaPOSH.

V POSH reaktivních plánech rovněž záleží na pořadí synů a není stanoven jejich maximální počet. Každý vjem, *sense* (S), musí být součástí *triggeru* (T), přičemž více vjemů v rámci *triggeru* vytvářejí sdruženou podmínku spojenou pomocí logického AND.

Návrh vychází z implementace yaPOSH plánů v knihovně Pogamut, protože předpokládá využití právě této implementace, nicméně teoreticky by mělo být možné tento návrh aplikovat i na jiné implementace.

ADF Struktura yaPOSH reaktivních plánů je ale velmi podobná rozšíření GP zvanému ADF [11]. Za hlavní strom GP můžeme považovat *drive collection* (DC), stromy pro *competece* (C) a *action patterny* (AP) budeme brát jako ADF. Evoluce ale bude muset probíhat i v rámci stromů C a AP.

Terminály V rámci GP pracujeme se sadou terminálů a neterminálů. Zde bude sada terminálů dynamická, kromě základních akcí se bude rozšiřovat o definice C a AP, souhrnně je nazveme akce a budeme označovat A. Další množinu terminálů tvoří vjemy, označujeme S.

Neterminály Neterminály v klasickém významu GP zde uplatnění nenajdou. Raději vytvoříme dynamické neterminály, které mohou měnit svoji strukturu. Pro DC, AP, C a pro T bude vždy jeden dynamický neterminál. Struktura neterminálů je popsána gramatikou v tabulce 3.2, kde $a \in A$ je akce z množiny akcí (tedy i s definicemi AP a C) a $s \in S$ je vjem.

Gramatika formálně popisuje pravidla, která struktura neterminálů musí splňovat. Plán obsahuje právě jednu DC a libovolný počet AP a C mající strukturu dle neterminálů v tabulce 3.2. Navíc máme podmínky na AP a C, ty se sice mohou odkazovat na ostatní AP a C, ale pouze tak, aby nevznikl cyklus.

Dynamická podoba neterminálu se využije při tvorbě nových AP a C, navíc nám pomůže při návrhu mutací.

Textová reprezentace plánu obsahuje navíc názvy prvků, závorky a formátovací odsazení. Odpovídající neterminály pro plán z příkladu predátor-kořist jsou zobrazeny v tabulce 3.3. Terminály zde jsou základní *GoToFood, EatFood, CrawlAway, RunAway, Explore* a dynamicky přidané *Feed, Retreat.* Plán obsahuje jednu C a jednu AP.

AP	neterminál: $a\ a$ `(AP Feed (GoToFood EatFood))`
C	neterminál: $s\ a\ a$ `(C Retreat (elements ((Crawl (trigger ((PredatorDoesNotSeeMe))) CrawlAway)) ((RunForYourLife RunAway)) ) )`
DC	neterminál: $s\ a\ s\ a\ a$ `(DC life (drives ((AvoidPredator (trigger ((SeePredator))) Retreat)) ((Feed (trigger ((SeeFood))) Feed)) ((Explore Explore)) ) )`

Tabulka 3.3: Ukázka plánu a odpovídající neterminály

Mutace Mutace budou buď provádět modifikaci C, DC nebo AP tak, aby stále odpovídaly gramatice, tedy aby existoval nějaký dynamický neterminál odpovídající jejich struktuře, nebo lze použít modifikace na úrovni celého stromu (přidání nebo odebrání C a AP).

Strukturálně podobné jsou dvojice DC - C a dvojce AP - T, tedy i mutace pro tyto budou podobné.

Základní návrh sady mutací pro C resp. DC:

1. přidání CE (resp. D) do C (resp. DC),
2. odebrání CE (resp. D) z C (resp. DC),
3. vytvoření nového T pro CE resp. D,
4. změna pořadí CE (resp. D) v C (resp. DC).

Sada mutací pro AP resp. T:

1. přidání a (resp. s) do AP (resp. T),
2. odebrání a (resp. s) z AP (resp. T),
3. změna pořadí a (resp. s).

Dále bude přítomna sada mutací:

1. záměna akce za jinou v AP, C, DC,
2. záměna vjemu za jiný v T,
3. odebrání T z AP resp. C,
4. vytvoření nového AP nebo C.

Očekáváme, že bude vznikat bloat, na jeho snižování navrhujeme použít buď tradiční zhoršení fitness příliš velkým jedincům, nebo speciální mutace, které by zmenšovaly složitost plánů:

1. zkrácení DC (resp. C) na určitou délku odstraňováním D (resp. CE),
2. zkrácení AP,
3. zjednodušení T,
4. odstranění nepoužitých AP,
5. odstranění nepoužitých C.

Nicméně určitá nadbytečnost je žádoucí, tyto mutace jsou čistě pro zabránění nepřiměřenému růstu velikosti, které by mohlo vést až k selhání výpočtu.

Křížení Pro strukturu yaPOSH reaktivních plánů lze křížení provést, zde navrhujeme dvě speciální:

1. sdílení AP a/nebo C mezi jedinci,
2. jednobodové křížení na úrovni DC, se zachováním relevantních AP a C v potomcích.

Opět je potřeba zachovat korektní strukturu odpovídající nějakému neterminálu a vyhnout se logickým chybám, jako je cyklus nebo chybějící akce.

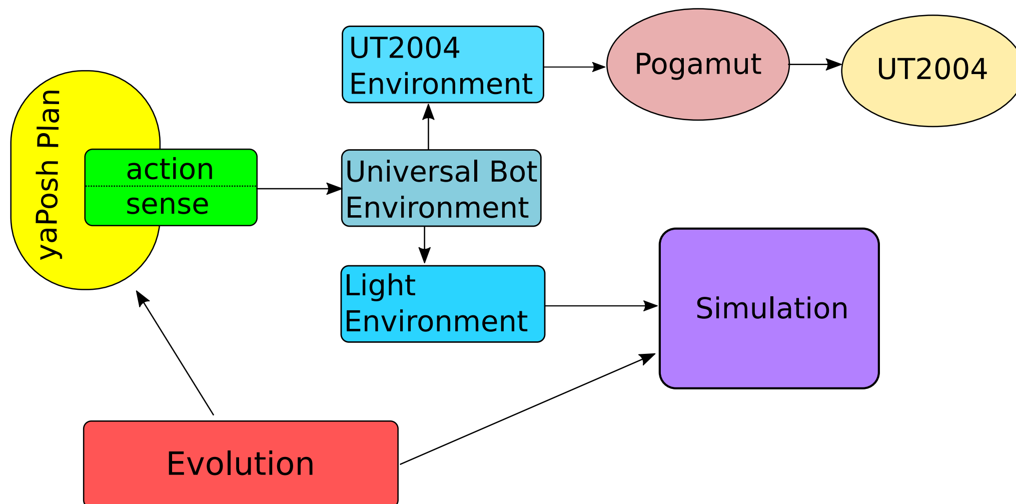
Operátory i neterminály jsou poněkud odlišné od tradičního GP, což je dáno strukturou yaPOSH plánů. Naopak koncept ADF je zde využit ve velké míře a značná část evoluce bude probíhat právě na této úrovni. Každopádně tento návrh umožňuje kvalitní implementaci.

3.1.5 Infrastruktura

Aby bylo možné spouštět yaPOSH plány jak v LightEnv i v UT, je potřeba vytvořit patřičnou infrastrukturu. Pro maximální efektivitu, by prostředí LightEnv mělo být co možná nejméně závislé i na knihovně Pogamut, nicméně předpokládáme využití plánovače pro yaPOSH plány. Celkový návrh ukazuje obrázek 3.2. Budeme pracovat s univerzálními akcemi, překlad do prostředí LightEnv a UT zajistí speciální mezivrstva, což reflektuje obecný návrh vztahu agent-prostředí. Toto navíc umožní snadno vytvářet plány a testovat je jak v jednom tak v druhém prostředí. Návrh umožňuje rozšíření o další prostředí.

Evoluce sice bude oddělená, ale bude pracovat s yaPOSH reaktivními plány a pro evaluaci fitness bude muset mít přístup do simulace v rámci LightEnv. Nepředpokládáme běh evoluce s prostředím UT, jinak bychom návrh provedli více obecný.

Dále je potřeba zajistit spouštění vyvinutých plánů v rámci UT pro účely validace.



Obrázek 3.2: Celkový návrh infrastruktury

3.2 Implementace

Tato část popisuje implementaci s ohledem na předešlý teoretický návrh, technické detaily lze nalézt v dokumentaci v příloze 1 a zdrojové kódy v příloze 2. Implementace je napsána v jazyce Java tak, aby byla zachována konzistence s použitými nástroji, zejména s knihovnou Pogamut [6]. Kód je strukturován do balíků, každý z nich zajišťuje samostatnou funkcionalitu.

Nejprve popíšeme implementaci odlehčeného prostředí LightEnv, následují části věnované popisu implementace obecné evoluce, evoluce týmu agentů a genetického programování pro potřeby POSH reaktivních plánů. Následuje přehled a popis celkové infrastruktury pro vývoj agentů.

Pro ucelený přehled doporučujeme prostudovat obrázek 3.2 zobrazující celkový návrh.

3.2.1 LightEnv

Při implementaci prostředí LightEnv byla pouze minimálně využita knihovna Pogamut [6], LightEnv má být maximálně nezávislé. Zdrojové kódy (příloha 2) pro LightEnv jsou sdruženy v balíku *lightEnv* a obsahují mimo jiné vše potřebné pro běh simulace i její zobrazování.

3.2.1.1 Simulace

Základem prostředí LightEnv je simulace *Simulation*, která zastřešuje veškerou logiku, tedy pohyb botů, navigaci, viditelnost, boj, spawn předmětů i botů. Simulace běží v krocích odpovídajícím 250 ms herního času. Toto reflektuje vlastnosti Pogamutu, kdy logika bota je aktivována ve stejných intervalech. Nicméně simulace prostředí v UT používá mnohem kratší interval.

Kalendář Pro objevování předmětů a botů (spawn) je použit mechanismus kalendáře, který umožňuje naplánovat událost pro simulaci do budoucna. Nevyužíváme reálný čas, ale pracuje s počtem kroků od začátku simulace.

Navigace Modul navigace zastřešuje hledání cest a navádění botů po nalezené cestě. Inicializace probíhá z exportovaných dat z knihovny Pogamut. Umístění navigačních bodů a jejich vlastnosti tedy velmi dobře odpovídají situaci v UT.

Nejkratší cesta je nalezena pomocí Floyd-Warshalova algoritmu s předpočítanou maticí, tedy samotný výpočet během simulace je rychlý. Následné navádění je již jednoduché.

Viditelnost Viditelnost je inicializována z uložených dat vytvořených knihovnou Pogamut. Jedná se o serializovaný Java objekt reprezentující informace o viditelnosti. Pro rychlejší běh simulace opět probíhá předzpracování při inicializaci.

Bot Bot je v prostředí modelován pomocí třídy *SimulationBot*, která udržuje aktuální informace o něm. Narozdíl od Pogamutu nemá tento bot vlastní modul navigace a viditelnosti, ale tyto jsou společné na úrovni simulace. Očekáváme totiž tvorbu i odstraňování botů mezi běhy evoluce, zatímco simulace zůstane. Nedochází tedy k náročné inicializaci těchto modulů vícekrát během evoluce.

Komunikace Prostředí umožňuje botům zasílat v rámci týmu předdefinované zprávy. Tyto zprávy nesou časové razítko tak, aby staré mohly být zahazovány. Zpráva obsahuje informace o odesílateli, zejména jeho současnou polohu. Lze je samozřejmě rozšířit i o další informace.

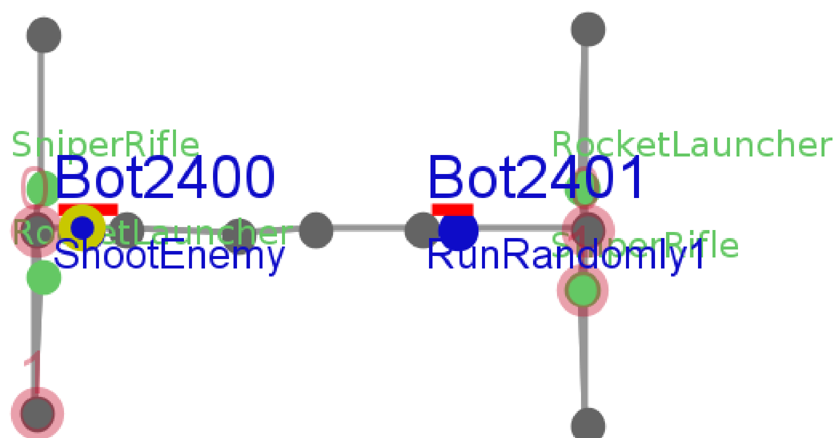
Na základě časového razítka lze stanovit platnost zprávy. Bot tedy nemusí okamžitě asynchronně zprávu zpracovat, může se periodicky dotazovat zda byla v poslední době nějaká zpráva. Výchozí platnost zpráv byla stanovena na 15 herních vteřin.

3.2.1.2 Zobrazení

Modul pro zobrazení je veskrze minimalistický, slouží pouze pro rychlou analýzu chování a debugging. Byla využita knihovna Swing. Zdrojové kódy jsou umístěny rovněž v balíku *lightEnv* stejně jako simulace, protože se nepředpokládá využití tohoto zobrazovače jinde.

Zobrazovač neobsahuje žádné ovládací prvky, slouží výhradně pro vizualizaci simulace, nicméně lze pomocí myši zobrazení posouvat a přibližovat i oddalovat. Ukázka je na obrázku 3.3, kde jsou dva boti bojující proti sobě na malé mapě *CTF-1on1-Joust*. Kromě základního zobrazení navigačních bodů a spojnic mezi nimi (šedá), botů (modrá) a předmětů (zelená), jsou zobrazeny další užitečné informace: jméno bota (modře nad pozicí bota) a jeho zdraví (červená), akce, kterou bot zrovna vykonává (modrá pod pozicí bota), naplánovaná trasa bota (růžová) a zda bot právě střílí (žlutá) nebo ne. Na příkladu na obrázku 3.3 bot *Bot2401* právě provádí akci *RunRandomly1* - běhá náhodně. Jeho trasa do budoucna je zobrazena růžovou barvou a míří doprava a dolů. Bot *Bot2400*, který vykonává akci *ShootEnemy*, tedy střílí, má rovněž naplánovanou trasu (doleva a dolů).

Samozřejmě by bylo možné funkcionalitu zobrazovače rozšířit, ale pro naše účely bohatě stačí takto navržený.



Obrázek 3.3: Vizualizace simulace v prostředí LightEnv na mapě CTF-1on1-Joust.

3.2.2 Evoluce

Evoluci na obecné úrovni zajišťuje balík *ea*. Implementace vychází z návrhu v části 3.1.2 a je dostatečně obecná pro použití i v jiných úlohách. Základem je generická populace `Population<T extends Individual>` a obecný generický operátor `Operator<T extends Individual>`, od něž se ostatní operátory odvozují.

Dále zde definujeme sadu tříd a rozhraní pro běh evoluce. Kromě základní sady pro odvozování mutací, křížení a inicializace populace, máme předdefinované šikovné operátory pro třídění podle fitness, turnajový selektor, elitismus, logování i ladící výpisy, které pracují dle návrhu na úrovni obecných jedinců a fitness.

Generační krok probíhá postupným vykonáváním jednotlivých operátorů v pořadí, v jakém byly definovány. Lze tedy dosáhnout prakticky libovolného generačního kroku pomocí vhodných operátorů. Navíc je dokonce možné snadno dosáhnout dynamicky se měnící velikosti populace, operátory mají nad ní plnou kontrolu.

Dále máme k dispozici abstraktní parametrizovatelné mutace, které jsou navrženy pro načítání parametrů ze souboru. Očekáváme použití za pomoci Java Reflection API, kdy parametry jsou předány jako pole Stringů. Zůstává zde prostor pro další rozšíření, například načítání úplně všech parametrů evoluce ze souboru.

3.2.3 Evoluce týmu agentů

K evoluci týmu agentů je potřeba přistupovat specificky, jak bylo diskutováno v části 3.1.3. Implementace je sdružena v balíku *ea.special*.

Evoluce pracuje s populací jedinců `TeamIndividual`, kteří ale zabalují jedince `PlanIndividual`. Tyto dvě úrovně odpovídají návrhu. Obecná implementace evoluce musela být rozšířena o mechanizmy umožňující například vytvořit snadno mutaci pro `PlanIndivual`, která se bude propagovat z vyšší úrovně - jedinec `TeamIndividual`. Toto zajišťuje `MutationProxy` a třídy od ní odvozené. V soula-

du s návrhem jsme schopni vytvářet mutace jak pro jedince na úrovni týmu tak na úrovni yaPOSH plánů, navíc lze pracovat jak s homogenními tak nehomogenními týmy a s různými strategiemi pro aplikaci mutací.

Pro výpočet fitness na úrovni týmů i na úrovni plánů se používají třídy odvozené od `TeamFitness` (balík *ea.special.fitness*). Umožňují různé strategie spouštění simulace. Měřené veličiny na úrovni plánů jsou získávány ze simulace a jsou to:

- uběhnutá vzdálenost,
- počet sebraných předmětů,
- zranění způsobené nepřátelům,
- počet zabití nepřátel,
- počet úmrtí.

Tyto veličiny během simulace sbírá `PlanReporter` z činnosti botů. Na úrovni týmu jsou tyto agregovány do výsledné fitness týmu. Tato agregace může navíc přidávat další informace z týmového scénáře a je možné nastavovat váhu jednotlivých veličin.

3.2.4 GP pro yaPOSH

Genetické programování pro yaPOSH má svá specifika, zejména to je výrazné využití ADF. Implementace mutací je v balíku *ea.special.mutation*, další operátory například pro křížení jsou v balíku *ea.special.operator*.

Mutace i operátory vycházejí z návrhu v části 3.1.4. Detaily lze nalézt v dokumentaci v příloze 1. Jako struktura pro udržování plánu byla zvolena třída `PoshPlan` z knihovny Pogamut, protože umožňuje dobrou manipulaci s plánem a před jeho vykonáváním není třeba plán konvertovat. Zároveň se jedná o poměrně uzavřenou samostatnou část, takže tato závislost není na škodu celkovému konceptu.

3.2.5 Infrastruktura

Naším cílem bylo vytvořit dostatečně obecnou univerzální infrastrukturu, která by umožnila snadné spouštění plánu jak v prostředí UT2004 s knihovnou Pogamut, tak v rámci našeho prostředí LightEnv. Implementace odpovídá návrhu (vizte obrázek 3.2) a umožňuje napojení dalšího jiného prostředí[2].

Základem je tedy abstraktní třída `BotEnvironment`, která definuje vstupy a výstupy z jednotlivých prostředí. V balíku *universal* je dále sada tříd poskytující abstrakci nezávislou na konkrétním prostředí.

Univerzální akce a vjemy jsou umístěny v balících *Yuniversal.action* resp. *Yuniversal.sense* a jsou potomky `UniversalAction` resp. `UniversalSense`. Jelikož využíváme implementaci yaPOSH plánovače z knihovny Pogamut, je struktura akcí dána. Z programátorského hlediska je práce s akcemi a vjemy trochu

[2] Jako proof-of-concept bylo vytvořeno napojení do UT pro boty řízené univerzálně programy v čisté Javě. Toto napojení (balík *UT2004Env*) se po technické stránce liší od napojení pro boty řízené pomocí yaPOSH reaktivních plánů (balík *Yut2004*). Navíc tyto boty lze spouštět i v rámci LightEnv.

méně pohodlná než při programování čistě Javového bota, ale to je vykoupeno snazší manipulací s logickými bloky v rámci plánu.

Tyto akce a vjemy nevyužívají mechanismu kontextu `Context` z knihovny Pogamut, ale zavádí vlastní, založený na `BotEnvironment`. Toto neumožňuje využít stávající akce a vjemy, což ale ani nebylo cílem. Naopak dovoluje použít stejný plán v různých prostředích, což je požadovaná funkcionalita.

Jelikož se v rámci yaPOSH plánovače používá Java Reflection API a akce i vjemy jsou definovány pouze textově, není možné snadno refaktorovat odpovídající třídy.

3.2.6 Závěr

Implementace všech částí je rozdělena do logických celků a struktura balíků toto rozdělení reflektuje. V předešlých částech byla podrobně implementace jednotlivých celků, pro technické detaily doporučujeme dokumentaci v příloze 1 nebo zdrojové kódy, příloha 2.

Implementace je napsána s ohledem na možná rozšíření, od jednoduchých (další akce a vjemy) až po komplexní (přidání dalšího jiného prostředí). Díky dokumentaci v podobě JavaDoc, by měla být tato práce snadná.

4. Experimenty

Tato kapitola se věnuje vývoji chování agentů pomocí technik a nástrojů prezentovaných v předešlých kapitolách.

Experimenty probíhaly na více strojích, ale jako referenční byl používán počítač s procesorem Intel®Core™i7-2860QM (2.5 GHz, 4 jádra, 8 threadů) osazený 8 GB RAM s operačním systémem Ubuntu 14.04 LTS (Linux 3.13.0-49-generic, 64bit). Detaily a rozbor výkonu jsou popsány v sekci 5.1.4 na straně 41.

Každý experiment začíná popisem úlohy, následuje návrh přístupu k řešení. Samotný experiment je ilustrován grafy vývoje fitness a měřených veličin. Následuje rozbor vzniklého chování a jeho evaluace v prostředí UT. Detaily i výstupy jsou v příloze 3.

Scénáře Pro experimenty byly vybrány základní scénáře *death match* (DM) a *team death match* (TDM). Tyto herní scénáře představují boj na život a na smrt dvou a vícero hráčů, v našem případě botů. Hra typu TDM sdružuje boty do týmů, skóre se počítá nejen jednotlivcům ale i celému týmu.

Týmová hra by měla vést ke spolupráci hráčů tak, aby získávali strategickou výhodu nad nepřítelem.

Měřené veličiny Pro hodnocení kvality chování měříme u botů nejen počet zabitých nepřátel a počet smrtí, ale i uběhnutou vzdálenost, počet sebraných předmětů a způsobené zranění nepřátelům. Tyto veličiny se používají i pro výpočet fitness a pomáhají nám při analýze vývoje chování jedinců v populaci.

Fitness Fitness botům určujeme na základě měřených veličin jako jejich vážený součet, obecně pro bota b a měřené veličiny $v_0, v_1, ...v_{n-1}$ je fitness $F(b)$ dána předpisem:

$$F(b) = \sum_{i=0}^{n-1} k_i v_i(b)$$

Hodnoty $k_i \in \mathbb{R}$ pro $i \in \{0, 1, ..., n-1\}$ volíme v závislosti na dané úloze.

Pro tým T o m botech je fitness týmu $E(T)$ dána předpisem:

$$E(T) = \sum_{b \in T} F(b)$$

Průběh evoluce hodně závisí na volbě fitness, vhodnou volbou parametrů lze docílit rychlejšího vývoje.

4.1 Experiment 1 on 1 DM

První scénář pro ověření možností vývoje byl zvolen *1 on 1 DM*, tedy boj na život a na smrt jednoho bota proti druhému. Mapa byla zvolena minimalistická *Joust*. Tato mapa například vůbec neobsahuje lékárny ani štíty.

4.1.1 Příprava

Pro evoluci bylo potřeba zvolit sadu akcí a vjemů, které bude mít bot k dispozici. Vynecháme akce a vjemy pracující s týmovou komunikací a akce zahrnující sbírání lékárniček a štítů, protože tyto se na mapě nevyskytují. Omezíme se tedy na základní sadu akcí: *ShootEnemy, RunForWeapon, RunRandomly, Explore, Hide, SwitchWeapon* a vjemů: *SeeEnemy, NumberOfVisibleEnemies, HasMoreWeapons, Health, Shield.*

Evoluce běžela 400 generací, což se ukázalo jako dostatečný počet pro nalezení dobrých řešení.

Fitness Jelikož v simulaci hraje velkou roli náhoda, pro validní výsledky je potřeba ji spouštět několikrát. Toto bylo reflektováno při volbě metodiky výpočtu fitness, kdy používáme `FixedOpponentFitness` s pěti opakováními simulace proti různým botům. Takto dostáváme docela stabilní výsledky, s menším rozptylem fitness. Jako protivníci byli zvoleni *DoNothingBot* a *FightBot*, jak jména napovídají, první nedělá vůbec nic, druhý bojuje a pohybuje se.

Fitness je nastavena tak, aby byl kladen důraz na boj, ale aby i pohyb po mapě přinášel určitou (menší) výhodu. Výpočet hodnoty je proveden podle vzorečku:

$$distanceRan/10 + enemiesKilled * 1000 + damageDone * 100 - timesDead * 100$$

Zde má uběhlá vzdálenost docela velkou váhu, proporcionálně je totiž mnohem větší než ostatní[1].

Ukázalo se totiž, že příliš nízká váha uběhnuté vzdálenosti, vede ke špatným chováním typu *SittingDuck*[2]. Takto navíc je možno evoluci dobře nastartovat s poměrně jednoduchým cílem vyvinout prozkoumávání, následovat mohou boj a další pokročilé techniky. Heuristický předpoklad je, že prohledávání prostředí bude více úspěšné, než čekání na místě.

4.1.2 Výsledky

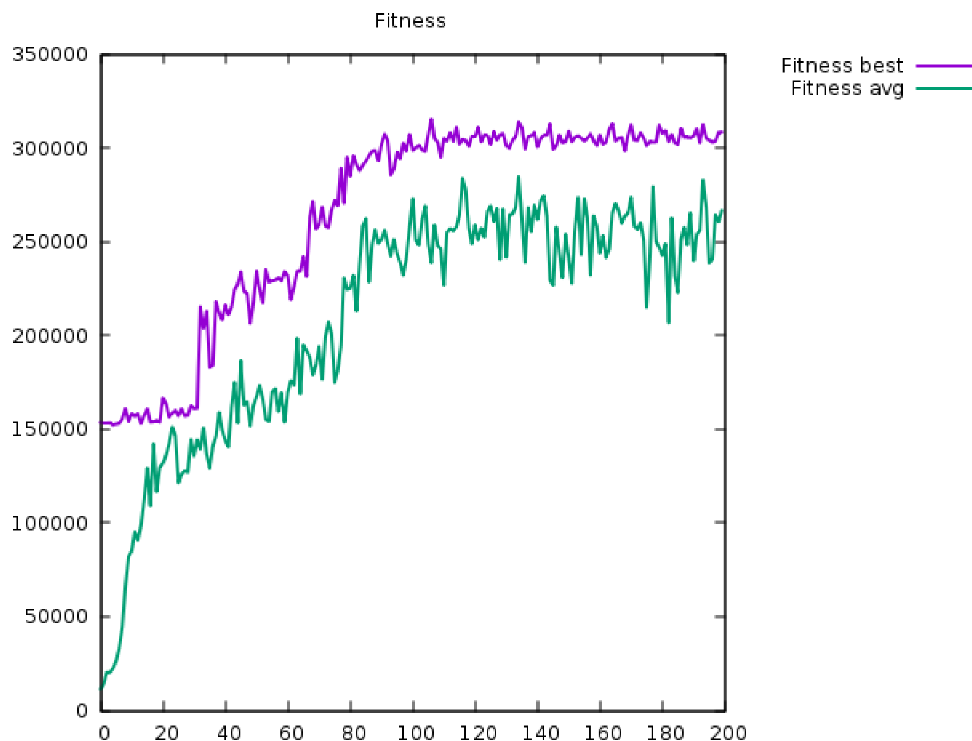
Vývoj fitness je na obrázku 4.1, zobrazuje pouze prvních 200 generací, v další evoluci již k výraznějším změnám v tomto případě nedošlo. Vývoj měřených veličin pro nejlepšího jedince je vynesen v grafu na obrázku 4.2, průměr je pak na obrázku 4.3. Tyto grafy mají logaritmické měřítko na ose y, aby bylo možno zobrazit všechny veličiny najednou.

Zaměříme se na výsledky nejlepšího jedince, kolem třicáté generace je vidět patrný nárůst fitness, odpovídající nárůstu způsobeného zranění, a tedy i počtu zabitých nepřátel. Zároveň je vidět pokles uběhnuté vzdálenosti. Počet smrtí zůstává stále přibližně stejný. Vyvinulo se zatím nedokonalé chování, které kombinuje pohyb po mapě s bojem, ale netvoří ještě zásadnější strategickou převahu.

Zajímavé je okolí šedesáté generace, kdy dochází k poklesu počtu zabití, ale i smrtí, bot se tedy střetu s nepřítelem spíše vyhýbal.

[1]Bereme maximální teoreticky možnou uběhlou vzdálenost, která je řádově větší než například maximální možný počet zabití. Pokud by bot celou dobu jen běhal urazí vzdálenost v řádech milionů vzdálenostních jednotek, zatímco pokud by dokázal zabíjet s maximální efektivitou nedostane se přes desítky zabití.

[2]Chování kdy bot čeká na místě, až ho protivník objeví.



Obrázek 4.1: Experiment 1, vývoj fitness 1on1

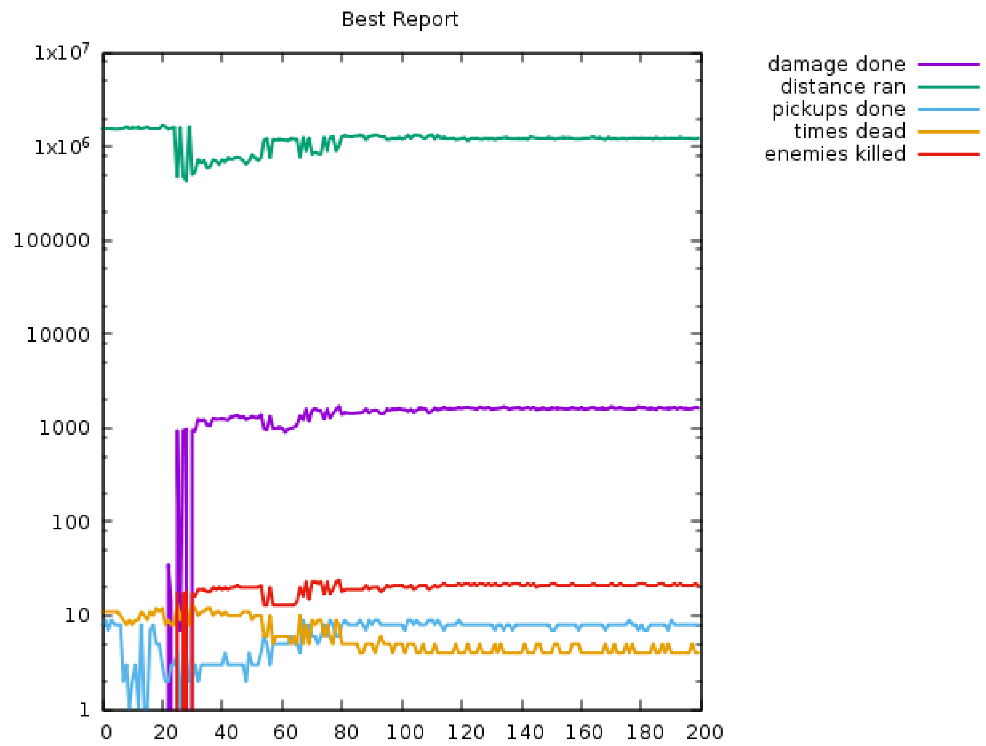
Velký nárůst nastává postupně mezi zhruba sedmdesátou a stou generací, kdy se chování víceméně ustálilo. Je vidět nárůst v počtu sebraných předmětů, pokles úmrtí a nárůst počtu zabití, ale též uběhnuté vzdálenosti. Chování tedy lze označit za velmi dobré, bot měl v boji strategickou výhodu.

Výsledný plán pro odstranění nedosažitelných a nepoužívaných částí je ve výpisu na obrázku 4.4.

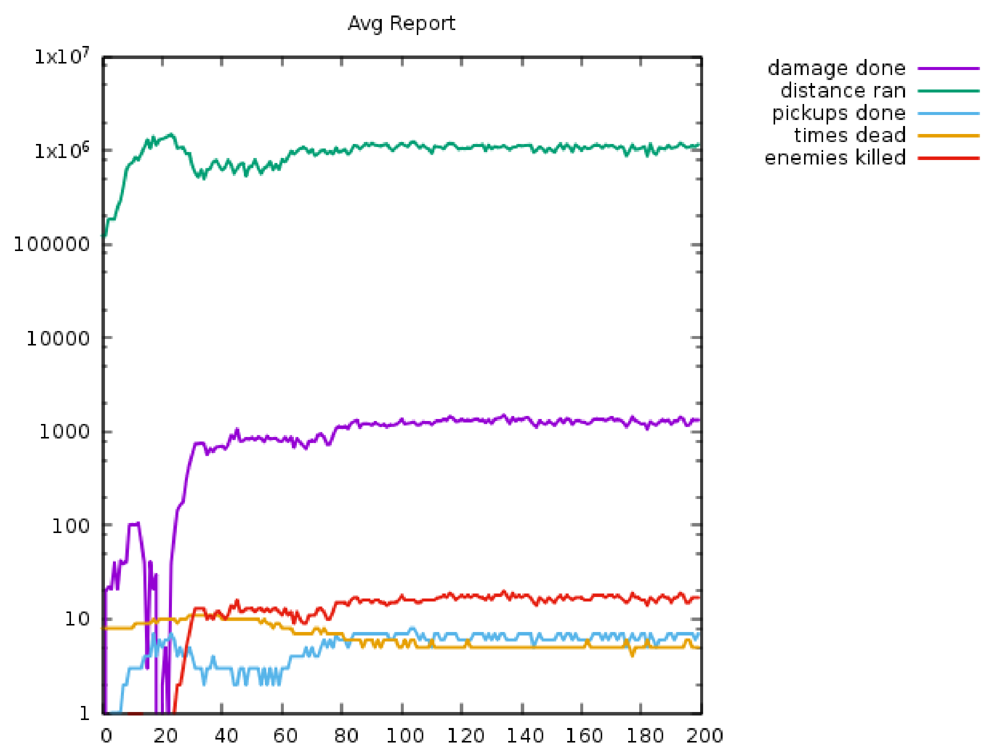
Jako nejvyšší úroveň máme chování D3973, které zajišťuje boj, pokud bot vidí nepřítele. Druhé chování D3974 zajišťuje prohledávání mapy (akce *Explore*) a přepnutí zbraně na lepší (akce *SwitchWeapon*). Akce *ShootEnemy* v rámci AP33 selže, pokud není v dostřelu žádný nepřítel, a pro bota to znamená pouze jeden cyklus prodlevu, což fitness nedokáže zachytit.

Vytvořené chování zajišťuje botovi strategickou výhodu vhodným přepínáním zbraní. Toto se děje v rámci delších AP, které zajišťují i další prvky chování. Jelikož akce *SwitchWeapon* trvá pouze jeden cyklus, a botovi tedy neškodí její častější (i když zbytečné) vykonávání.

Zajímavé je přepínání zbraní, které se děje vždy po skončení akce *Explore*, a je tedy voláno velmi často. Tato akce je tedy velmi často potřeba a nejspíš chybí vhodný vjem, který by ji dokázal iniciovat v DC. Sadu botových vjemů bychom mohli rozšířit o vjem *WasWeaponPickup*, který by reprezentoval, zda byla nedávno sebrána zbraň.



Obrázek 4.2: Experiment 1, vývoj měřených veličin pro nejlepšího jedince



Obrázek 4.3: Experiment 1, průměrný vývoj měřených veličin

```
(
  (C C79
    (elements
      ((CE34 (trigger ((Yuniversal.sense.Health 69 <=)))
              AP1
      ))
    )
  )
  (AP AP1 (Yuniversal.action.Explore
           Yuniversal.action.SwitchWeapon))

  (AP AP3 (AP1 AP33))

  (AP AP33 (Yuniversal.action.ShootEnemy C79))

  (DC life
    (drives
      ((D37393
        (trigger (
          (Yuniversal.sense.NumberOfVisibleEnemies 1 >=)
          (Yuniversal.sense.Shield 86 <=))
          )
        Yuniversal.action.ShootEnemy))
      ((D37394 AP3))
    )
  )
)
```

Obrázek 4.4: Plán v textové podobě

4.1.3 Validace v UT

V prostředí UT jsme zachovali stejné podmínky. Vyvinuté plány byly spouštěny proti stejnému soupeři jako v prostředí LightEnv. Ukázka z jednoho zápasu je v příloze 4.

Dle subjektivního hodnocení se bot choval inteligentně, nad naprogramovaným nepřítelem měl převahu. V některých případech došla oběma botům munice, což se v prostředí LightEnv nestávalo. To je dáno zjednodušeným vyhodnocováním střelby v LightEnv, kdy pravděpodobnost zásahu neklesá s rostoucí vzdáleností, a boti tedy neplýtvají municí. Jako zjednodušený model byly zbraně vybaveny menší kapacitou, než jakou mají předlohy v UT.

4.2 Experiment 2 on 1 TDM

Jako druhý scénář byl zvolen death match nevyrovnaných týmů. Budeme vyvíjet tým o dvou jedincích, který bude bojovat proti jednomu nepříteli. Cílem je porovnat výsledky s předešlým experimentem a zhodnotit, jaký vliv může mít týmová spolupráce. Byla zvolena mapa *Albatross*, která patří mezi velké mapy a obsahuje velké množství předmětů včetně lékárniček, štítů a zbraní.

4.2.1 Příprava

Jelikož vyvíjený tým obsahuje více jedinců, přidáme akce a vjemy pro týmovou spolupráci:

- zasílání zpráv: akce *SendMessage*,
- zpracování zpráv: vjem *WasMessage*,
- týmová spolupráce: akce *RunToSender*.

Také připojíme akce pro cílené sbírání předmětů: *RunForHealth, RunForShield, RunForWeapon*.

Evoluci jsme nechali běžet 500 generací. Týmy byly homogenní, tedy každý jedinec v týmu měl byl řízen stejný plánem. Dále byl snížen počet opakování zápasů pro vyhodnocení fitness na tři, ale zachovali jsme opakování proti dvěma různým nepřátelům: *DoNothingBot* a *FightBot*.

Fitness pro bota byla mírně upravena oproti předešlému příkladu:

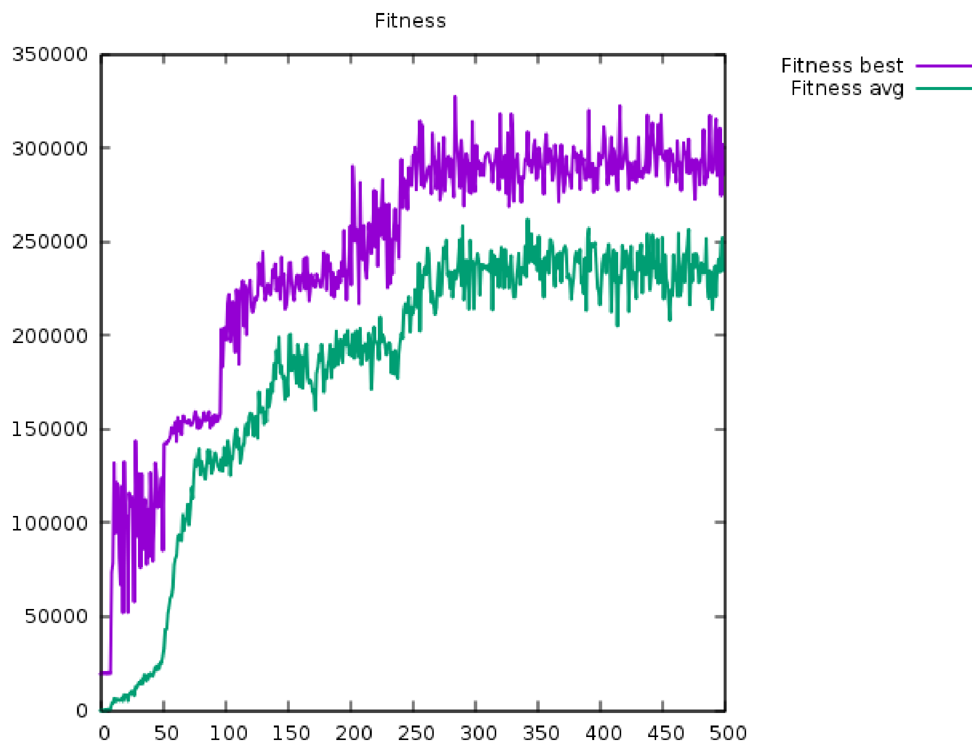
$$distanceRan/50 + 1000 * enemiesKilled + damageDone * 100 - timesDead * 100$$

Fitness týmu se počítá jako součet fitness členů, stejně tak měřené veličiny.

4.2.2 Výsledky

Podařilo se vyvinout inteligentní chování, průběh experimentu ilustruje obrázek 4.5 ukazující vývoj fitness týmů.

Zhruba do padesáté generace byly zkoušeny náhodné strategie, z obrázků 4.6 a 4.7 vývoje měřených veličin pro nejlepší tým, respektive vývoje průměru těchto veličin v populaci, lze vyvozovat, že v raných stádiích evoluce jedinci buď dokázali



Obrázek 4.5: Experiment 2, vývoj fitness

střílet nebo běhat, ale mezi těmito chováními nedokázali přepínat. Vzhledem k nastavené fitness, byli upřednostňováni jedinci, kteří dokázali střílet. Velký rozptyl fitness je způsoben bojem proti botovi *DoNothingBot*, kdy úspěšnost týmu závisela na tom, kde se boti objevili. Jelikož *DoNothingBot* se nepohybuje, mohla snadno nastat situace, kdy se během celé simulace vůbec nic nestalo. Úspěšnost proti botovi *FightBot* též závisela na náhodě. *FightBot* se sice pohybuje, ale v rozlehlé mapě může trvat dlouho, než narazí na vyvíjené boty.

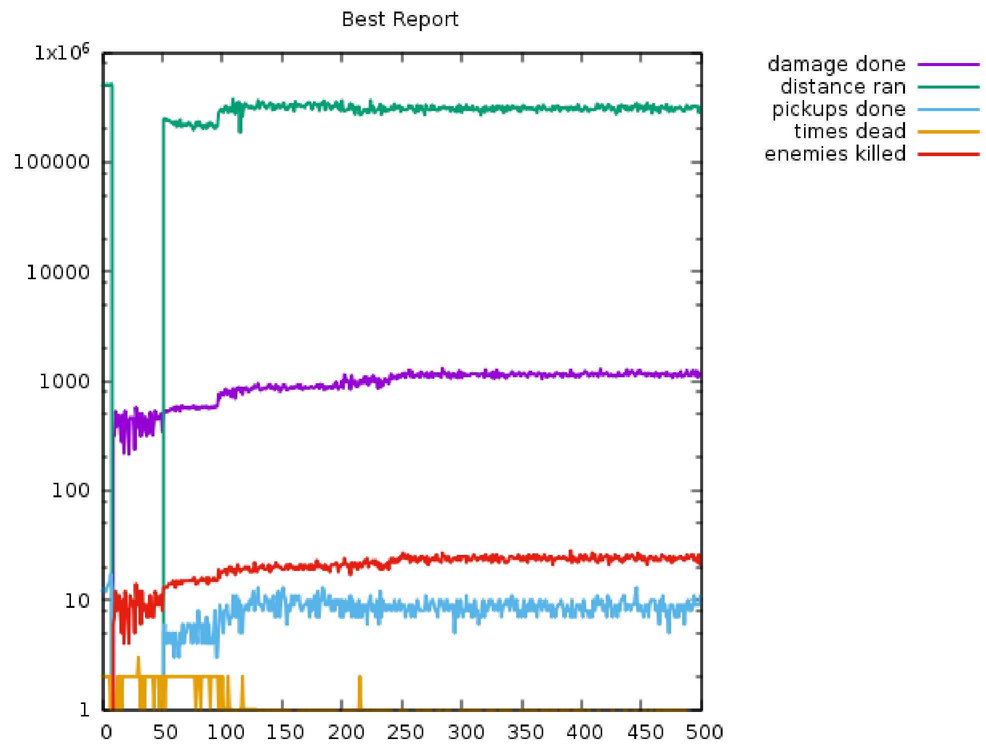
Ve zhruba padesáté generaci se objevilo chování, které zahrnovalo přepínání mezi prozkoumáváním a bojem. Nárůst fitness je způsoben nárůstem uběhlé vzdálenosti (a tedy i počtu sebraných předmětů) u nejlepšího týmu. Zároveň vzrostl počet zabitých nepřátel, tato strategie totiž funguje spolehlivě i proti nepříteli *DoNothingBot*, který by jinak nemusel být objeven.

Další nárůst před stou generací a kolem dvě stě padesáté je způsoben zefektivněním boje a odstraněním nevhodných aspektů chování.

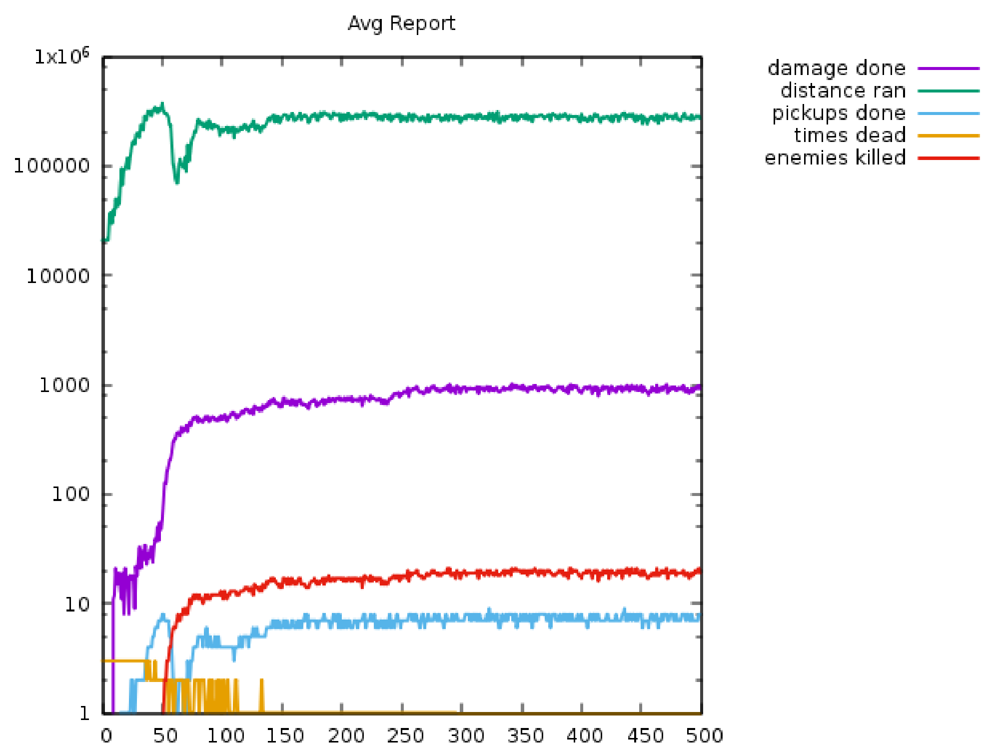
Výsledný plán po zjednodušení (obrázek 4.8) je překvapivě jednoduchý, ukazuje se, že přesila zajišťuje botům velkou strategickou výhodu. Zároveň vzhledem k rozloze mapy a množství předmětů je náhodným prohledáváním možné dosáhnout dobrých výsledků.

Pro přehlednost byli prvky v plánu na obrázku 4.8 pojmenovány. Chování je jednoduché, problém nastává při vysokém stavu štítu, kdy se bot zastaví. Je zde tedy prostor pro další zlepšení. Boti nevyužívají komunikace, jejich přesila jim dává dostatečnou výhodu.

Vyvinuté chování umožňuje tři základní věci, a sice střílet po nepřátelích, jsou-li v dohledu, přepínat zbraně a prohledávat mapu. Cílené sbírání předmětů



Obrázek 4.6: Experiment 2, vývoj měřených veličin nejlepšího týmu



Obrázek 4.7: Experiment 2, vývoj měřených veličin - průměr

```
(
  (AP SwitchWeaponAndExplore
    (Yuniversal.action.SwitchWeapon
     Yuniversal.action.RunRandomly1)
   )

   (DC Evolved
     (drives
       ((Fight (trigger ((Yuniversal.sense.SeeEnemy)
                         (Yuniversal.sense.Shield 80 <=)) )
               Yuniversal.action.ShootEnemy
       ))

       ((Explore (trigger ((Yuniversal.sense.Shield 83 <=) ))
                          SwitchWeaponAndExplore))
       )
   )
)
```

Obrázek 4.8: Experiment 2, zjednodušený plán v textové podobě

zde není, náhodný pohyb stačí.

4.2.3 Validace v UT

Vyvinutí boti měli převahu i v rámci prostředí UT, ale objevilo se několik problematických oblastí.

První problém je v navigaci. Boti se občas zaseknou, například protože se jim nepodaří skok. Bot se pokouší manévr několikrát zopakovat, což působí velmi nepřirozeně, a navíc tím ztrácí čas. Nezasekne se napořád, navigace v Pogamutu má vestavěné mechanismy detekce těchto situací. Tyto události byly na mapě *Albatross* více patrné než na *Joust* z minulého experimentu.

Druhá oblast, která by zasluhovala prohloubení, je specifické využití zbraní. V našem návrhu nemá bot velkou kontrolu nad zbraní, se kterou střílí, nemůže ani upravit podle ní své chování. Vhodnost využití zbraně záleží na situaci a toto naši boti nemohou reflektovat.

I přes uvedené nedostatky byli vyvinutí boti úspěšní.

4.3 Experiment 3 on 3 TDM

Třetí scénář si klade za cíl vyvinout chování pro tým o třech botech v týmovém scénáři death match. Toto již je dostatečně velký tým, aby se mohlo vyvinout zajímavé týmové chování, proto v sadě terminálů budou přítomny akce pro využití týmové spolupráce. V předešlém experimentu se týmové chování nepodařilo vyvinou, předpokládáme, že je to dáno přesilou vyvíjeného týmu, proto zde volíme vyrovnané počty.

4.3.1 Příprava

Prostředí Na prostředí klademe několik požadavků tak, aby byl připraven prostor pro evoluci. Zvolená mapa by měla být velká tak, aby i pro celkem šest botů nedocházelo ke střetům příliš často. Další požadavek je na viditelnost, kdy spíše preferujeme mapy více uzavřené. V neposlední řadě je pro vývoj týmového chování potřeba zvolit mapu, která má i přes předešlé dobrou průchodnost. Byla zvolena mapa *DM-Rankin*, která požadavky splňuje.

Akce Sada akcí zahrnuje jak základní akce pro chování botů *RunRandomly, Explore, RunForWeapon, RunForHealth, RunForShield, Hide, ShootEnemy, SwitchWeapon*, tak rozšířené akce pro týmovou komunikaci: *SendMessage1, SendMessage2, SendMessage3, RunToSender.*

Vjemy Vjemy pro vyvíjené boty jsou: *SeeEnemy, Health, Shield, NumberOfVisibleEnemies, HasMoreWeapons, WasMessage1, WasMessage2, WasMessage3.*

Fitness Fitness funkce opět reflektuje naše cíle, snažíme se vyvinout chování, které bude zahrnovat pohyb po mapě, sbírání předmětů a boj (zabíjení nepřátel a co možná nejméně vlastních smrtí). Samozřejmě je potřeba v počátcích zvýhodňovat chování, která splňují alespoň část tohoto komplexního úkolu.

Fitness byla tedy zvolena:

$$distanceRan/10 + 1000 * enemiesKilled + 100 * damageDone - 500 * timesDead$$

4.3.2 Výsledky

Vývoj fitness jedinců je zobrazen na obrázku 4.9. Okolo zhruba padesáté generace je patrný první velký nárůst. Z grafu vývoje měřených veličin nejlepšího jedince na obrázku 4.10 a průměru na obrázku 4.11 lze usuzovat, že původní náhodné strategie byly nahrazeny lepšími.

Díky dobré volbě fitness byli v rámci prvních generací dobře hodnoceni jedinci, kteří se dovedli pohybovat po prostředí. Ve zhruba čtyřicáté páté generaci se vyvinulo stacionární bojové chování, indikováno propadem uběhlé vzdálenosti na nulu a nárůstem počtu zabití nepřátel. Toto chování má lepší fitness než jen prozkoumávání prostředí. Toto chování bylo rychle nahrazeno (zhruba padesátá generace) komplexnějším, které kombinuje prohledávání prostředí s bojem a je ohodnoceno vyšší fitness. Lze pozorovat nárůst uběhlé vzdálenosti a počtu sebraných předmětů, ostatní veličiny ale zůstávají na stejné úrovni, bot bojuje stále stejně dobře.

Další výrazná změna nastává u zhruba sté generace. Fitness funkce začíná mít větší rozptyl mezi jednotlivými generacemi, který může být způsoben prvkem náhody a úspěšnost nových chování závisí na vstupních podmínkách o něco více. Nicméně na grafu vývoje měřených veličin (obrázek 4.10) je patrné, že počet zabitých nepřátel překonal počet smrtí, tým má tedy pozitivní skóre a překonává naprogramovaný tým. Velký rozptyl fitness je dán její velkou citlivostí na tyto veličiny, malý rozdíl má podstatný vliv na velikost fitness.

Plán Nejlepší vyvinuté strategie využívají týmovou komunikaci, zjednodušený pojmenovaný plán pro jednu z nich můžete nalézt na obrázku 4.12. Tímto plánem byli řízeni všichni tři boti v týmu.

Nejvyšší prioritu má chování *Fight* spouštěné vjemem *SeeEnemy*, které zajišťuje v případě zahlédnutí nepřítele týmovou spolupráci a boj. Vykonává se AP *FightTogether*, kdy bot nejprve pošle týmovou zprávu *Message3* a začne střílet.

Druhé chování *Arm* není téměř vůbec spouštěno, což je dáno kombinací vjemů ve spouštěči. Pokud bot slyšel zprávu *Message1* a nemá více zbraní, běhá náhodně. Zpráva *Message1* je posílána pouze na konci AP *Gather*.

Chování *HelpOthers* zajišťuje pomoc při boji a shromažďování botů. Je spouštěno byla-li zachycena zpráva *Message3*, která je posílána, než bot zahájí palbu. V tomto chování je volán AP *Gather*, kdy bot doběhne na místo, ze kterého byla zpráva poslána.

Čtvrté chování s nejnižší prioritou nemá žádný spouštěč a zajišťuje pouze prohledávání okolí pomocí AP *Explore*. V tomto AP bot nejprve běží na náhodné místo, pak přepne zbraň a nakonec běží pro lékárnu.

Výsledné chování Výsledné chování je potřeba popisovat na úrovni celého týmu. Boti se pohybují po prostředí, pokud jeden zahlédne nepřítele, pošle zprávu a ostatní se k němu rozběhnou po nejkratší cestě. Pokud cestou nenarazí na jiného soupeře, dorazí bot k již zraněnému nepříteli a dokonce může vzniknout přesila až tři na jednoho. Toto chování hodnotíme velmi pozitivně, poskytuje týmu velkou strategickou převahu.

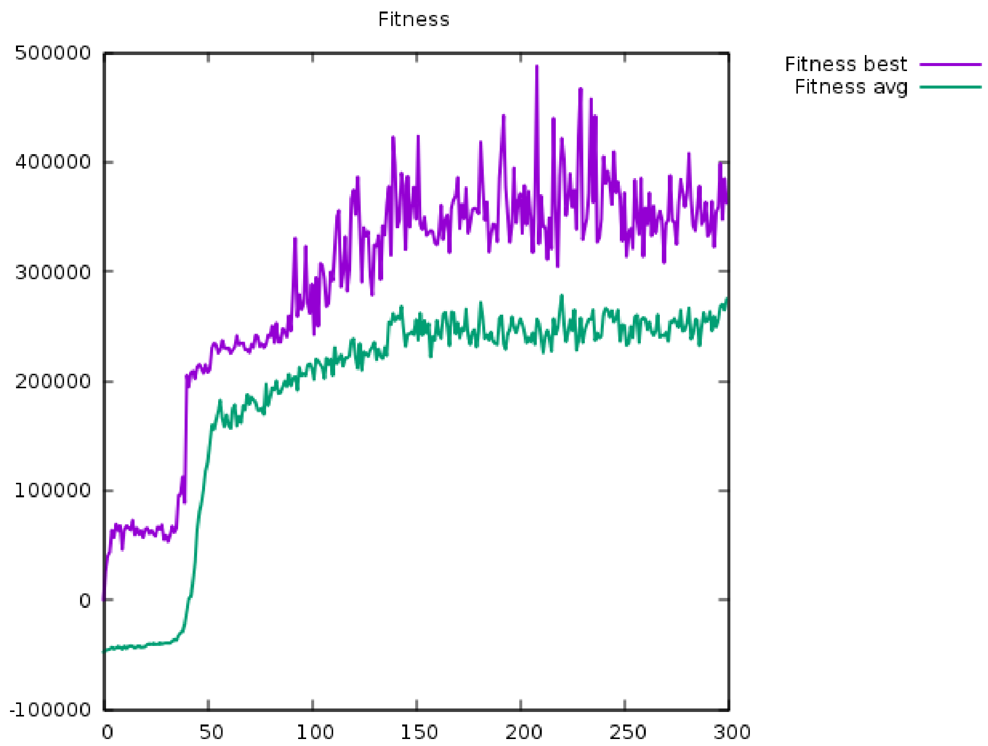
V některých situacích boti setrvávají na jednom místě v hloučku, což je dáno dobou platnosti zpráv (výchozí hodnota je 15 herních vteřin), ale nezhoršuje to zásadně výsledné chování.

Boti spoléhají spíše na náhodné prohledávání prostředí, než na cílené sbírání předmětů, zde by mohl být prostor pro další vylepšení.

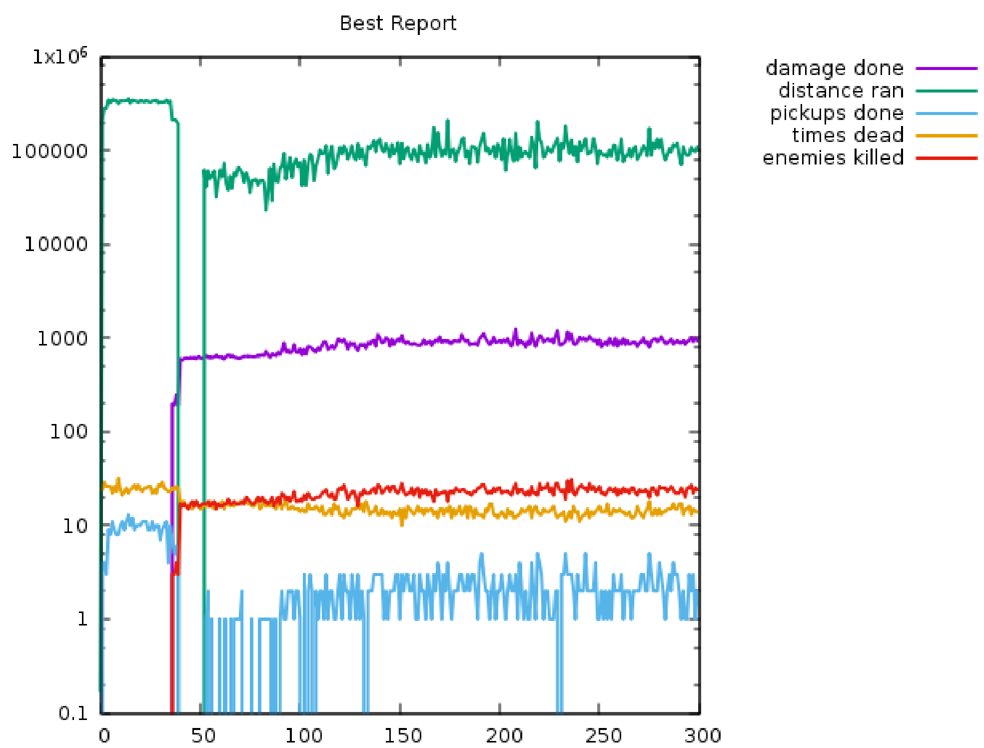
4.3.3 Validace v UT

Tým se po přenesení do prostředí UT choval dobře, týmová spolupráce fungovala. Po souboji agenti často zůstávali až po dobu zhruba pěti sekund v hloučku, než se rozeběhli. Udržování týmu pohromadě se zdálo být velmi dobré, nevadila ani určitá prodleva, než se boti rozeběhli dál v rámci chování *Explore*.

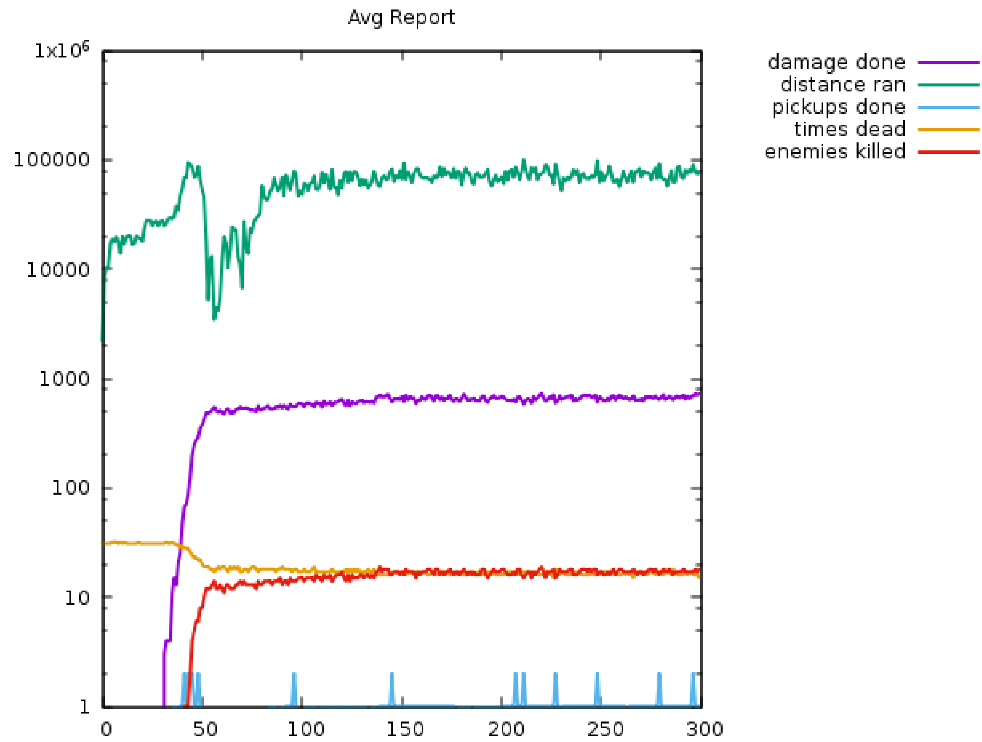
Oproti LightEnv se zdálo méně časté spouštění posledního chování *Explore*. Mohlo by to být způsobeno mírně odlišným chováním navigace a pohybu v UT, kdy navigace může občas selhat, a boti se například na chvíli zaseknou. Lze pozorovat v přiložených videích v příloze 4. Zdálo se, že boti tráví více času v chováních s vyšší prioritou.



Obrázek 4.9: Experiment 3, vývoj fitness



Obrázek 4.10: Experiment 3, vývoj měřených veličin nejlepšího týmu



Obrázek 4.11: Experiment 3, vývoj měřených veličin - průměr

```
(
  (AP FightTogether (SendMessage3 ShootEnemy))

  (AP Gather (RunToSender SendMessage1))

  (AP Explore (RunRandomly1 SwitchWeapon RunForHealth))

  (DC life
    (drives
      ((Fight (trigger ((SeeEnemy)))
               FightTogether))
      ((Arm (trigger ((HasMoreWeapons false ==)
                      (WasMessage1)))
            RunRandomly1))
      ((HelpOthers (trigger ((WasMessage3)))
                   Gather))
      ((Explore Explore))
    )
  )
)
```

Obrázek 4.12: Experiment 3, zjednodušený plán v textové podobě

5. Závěr

V této kapitole shrneme poznatky z předešlých částí a nastíníme další možný rozvoj tématu.

5.1 Výsledky

Tato část shrnuje nejen výsledky experimentů, ale v širším kontextu této práce hodnotí návrh i implementaci jednotlivých řešení.

Pomocí vytvořeného zjednodušeného prostředí LightEnv se podařilo vyvinout kvalitní chování pro agenty, která mohou být přenesena do prostředí hry Unreal Tournament 2004 se zachováním jejich vlastností. Prostředí LightEnv se ukázalo jako vhodné pro potřeby umělé evoluce a byl ověřen koncept přenosu vyvinutých chování mezi prostředími.

5.1.1 Vyvinutá chování

Herní scénáře zahrnovali jak scénáře pro jednoho bota (*death match*), tak i pro vícero (*team death match*). Ve všech případech se podařilo vyvinou chování lepší než ta, která byla předprogramována.

V rámci scénáře *death match* i v týmovém scénáři nevyrovnaných týmů (*2 on 1 death match*) boti získali převahu pomocí využití mechanismu výběru zbraně. Vyvinuté yaPOSH reaktivní plány byly ještě ručně upraveny do více čitelné podoby.

Pro scénář vyrovnaných týmů (*3 on 3 TDM*) se podařilo vyvinou chování využívající týmovou komunikaci, čímž si boti zajistili strategickou výhodu. V rámci tohoto chování si boti aktivně přicházeli na pomoc v případě ohrožení.

5.1.2 LightEnv

Návrh prostředí LightEnv umožnil rychlý běh evoluce, více v sekci 5.1.4. Prostředí nabízí velmi omezenou fyziku a jednoduchou dvourozměrnou vizualizaci.

Pro vývoj agentů byl při návrhu kladen důraz na navigaci odpovídající navigaci v Pogamutu. Obě jsou založeny na navigačním grafu a ukázalo se, že tato abstrakce je dostatečná i pro celkový pohyb agenta.

Modul viditelnosti navržený stejným způsobem ale agenti nevyužívali naplno, akce používající viditelnost se neobjevily v žádném z vyvinutých plánů, v tomto směru je tedy prostor pro zlepšení. To může být dáno i omezenou sadou navržených akcí. Předpokládáme, že vytvořením dalších, komplexních akcí využívajících viditelnost by bylo možné chování zlepšit.

Mechanismus sbírání předmětů a jejich znovuobjevování se ukázal být dostatečný.

Určité rozšíření by bylo potřeba učinit pro mechanismus boje, zejména doladit vlastnosti jednotlivých modelovaných zbraní tak, aby co možná nejvíce odpovídaly předlohám ve hře Unreal Tournament 2004.

I přes značné zjednodušení oproti hře Unreal Tournament 2004 je LightEnv vhodné pro vývoj a testování složitých chování agentů, která navíc mohou být jednoduše přenesena přímo do prostředí této hry.

5.1.3 Přenositelnost chování

Přenositelnost chování mezi prostředími dosáhla dobré úrovně, boti vyvinutí v rámci LigthEnv se chovali dobře i v prostředí hry Unreal Tournament 2004. Samozřejmě zůstává prostor pro zlepšení, ale například týmová spolupráce pomocí komunikace dosáhla v experimentu maximální možné úrovně s ohledem na dostupné akce a vjemy.

Nicméně byly objeveny menší odlišnosti v některých aspektech chování přenesených z LightEnv do hry Unreal Tournament 2004.

První odlišnost se projevila v pohybu botů. Navigaci v prostředí hry Unreal Tournament 2004 zajišťuje knihovna Pogamut. Navigace je do jisté míry nespolehlivá, v určitých situacích se může bot zaseknout, ať už srážkou s jiným botem, nezdařilým skokem nebo jen chybou v navigačním grafu. Tento fenomén se projevuje zaseknutím bota nebo opakováním zdánlivě nesmyslného pohybu. Navigace obsahuje metody pro detekci a řešení těchto situací, nicméně toto chování působí rušivě a nepříliš inteligentně. V tomto směru je LightEnv o něco lepší, navigace je zcela spolehlivá a takovéto chyby zde nenastávají.

Velká odlišnost nastává v rámci soubojů, prostředí LightEnv tyto řeší velmi zjednodušeně. Nejsou různé typy projektilů ani nepřesné zásahy, které jsou možné a časté v prostředí UT. V tomto směru by bylo vhodné rozšířit simulaci v LightEnv.

5.1.4 Výkon

Díky návrhu prostředí LightEnv byl běh simulace velmi rychlý, bylo možné experimenty snadno spouštět opakovaně a doba běhu byla i při vyšším počtu generací přijatelně krátká.

Na referenčním stroji jsme dosáhli u prvního experimentu výkonu zhruba 400 generací za hodinu. Každá generace obsahovala vyhodnocení fitness pomocí pěti opakování pětiminutové simulace a to proti dvěma protivníkům. Dostáváme se tedy zhruba na 300 herních hodin za jednu hodinu, což považujeme za velký úspěch. Jak uvádí Kadlec v [9], změnou rychlosti simulace v UT dochází k výrazné změně vlastností prostředí z pohledu vyvíjených botů. Simulace by tedy měla běžet v reálném čase rychlostí jedna herní hodina za hodinu. LightEnv se ukázalo být až třistakrát rychlejší při zachování vlastností důležitých pro vývoj botů.

S rostoucím počtem botů v simulaci se výkon snižoval, nejvíce času trvalo vyhodnocování yaPOSH reaktivního plánu, který ale nebyl navržen s ohledem na tento způsob využití. Jistého zlepšení by bylo možné dosáhnout při použití protivníků naprogramovaných v čisté Javě.

Problém nastává při přílišném nárůstu bloatu, což je vlastnost GP, kdy výrazně poklesne výkon a celá evoluce může selhat kvůli nedostatku paměti. I z tohoto důvodu je třeba bloat držet pod kontrolou.

5.2 Další rozvoj tématu

5.2.1 Integreace s knihovnou Pogamut

Využití zjednodušeného prostředí LightEnv se ukázalo jako velmi dobré. Jelikož pracuje s yaPOSH reaktivními plány z Pogamutu, bylo by vhodné tento projekt do Pogamutu začlenit.

Mohlo by se podařit dosáhnout až takového stupně integrace s Pogamutem, kdy by plán při procesu ručního návrhu pomocí pluginu v IDE NetBeans mohl být na pozadí rychle vyhodnocován, a uživatel by měl téměř okamžitě očekávané výsledky bota řízeného tímto plánem. Samozřejmě je třeba zajistit validitu takovýchto odhadů, ale jak tato práce ukázala, je možné toto provést.

Také by bylo vhodné umožnit integraci opačným směrem, tedy přidat v pluginu v IDE NetBeans podporu pro ladění plánů pro boty spouštěné v LightEnv. V současné chvíli je pouze v rámci vizualizace zobrazena akce, kterou bot vykonává.

5.2.2 Využití Navigation Mesh

Implementace LightEnv pracuje s Pogamutem verze 3.6.1 z května 2014. Nová verze 3.7.0 z února 2015 přináší podporu pro *navigation mesh* (navigační mřížka).

Jako možné rozšíření by bylo vhodné provést analýzu vlastností, které navigation mesh botům přináší, a zda a jakým způsobem provést úpravy LightEnv, aby reflektovalo nové možnosti.

5.2.3 Scénáře

V této práci jsme se zaměřili na herní scénáře *death match* a *team death match*, nicméně se nabízí rozšíření na další týmové scénáře, jako je *capture the flag* nebo *domination*, kdy boti řeší složitější úlohy. Samozřejmě bude třeba rozšířit sadu akcí a vjemů, i funkcionalitu zjednodušeného prostředí LightEnv, které je na to díky svému návrhu dobře připraveno.

5.2.4 Tým

Pro evoluci byly použity homogenní týmy, pro komplexní scénáře by bylo vhodné pracovat s týmy heterogenními. Boti se pak mohou mnohem lépe specializovat, ale evoluce je o to komplikovanější.

5.2.5 Rozšíření metodiky BOD a POSH reaktivních plánů

Metodika BOD se ukázala jako velmi kvalitní nástroj pro tvorbu chování agentů, POSH reaktivní plány bylo možné vyvíjet pomocí technik založených na genetickém programování. Nicméně při práci byly zjištěny jisté nedostatky. Tato část se věnuje možným návrhům pro zlepšení.

Nedostatky Jako velký nedostatek při tvorbě chování agenta striktně podle metodiky BOD s využitím POSH reaktivních plánů se nám ukázala absence paralelního chování. Dekompozice na základní akce a vjemy je velice dobrá, ale pro

některá chování tato dekompozice může selhat kvůli absenci paralelních akcí. Na tyto problémy upozorňuje i Zelinka v [23].

Jako modelový příklad může posloužit boj v rámci prostředí hry Unreal Tournament 2004. Při dekompozici bychom potřebovali vyjádřit fakt, že bot má v případě zahlédnutí nepřítele zároveň střílet a uhýbat střelám protivníka, což v současném stavu nelze, bot může pomocí POSH reaktivního plánu vykonávat jen jedno z uvedeného.

Současné možnosti řešení V současné době se využívají dva možné přístupy, první z nich je uzavření soubojového chování do samostatné akce. Tato akce potom řeší všechny aspekty boje. Velkou nevýhodou je, že není možné toto chování upravovat na úrovni POSH reaktivních plánů. Navíc neproběhla dekompozice až na úroveň základních prvků podle BOD. Akce *bojuj* je však příliš komplexní. Tento přístup použil Zelinka ve své práci [23].

Druhý přístup, který využívá i knihovna Pogamut, je umožnění paralelního vykonávání efektů akce. Máme na mysli například akce *StartShooting* způsobující, že agent začne střílet a přestane až po volání akce *StopShooting* nebo akce využívající navigaci. Tyto akce není třeba volat v každé iteraci, přesto se jejich efekty projevují. Toto není v souladu s modelem agenta v prostředí, neboť to porušuje transparentnost vazby výstupu agenta (tedy akce) na efekt v prostředí (projev akce).

Návrh řešení Jako třetí variantu navrhujeme rozšíření POSH reaktivních plánů o *paralelní akce* (PA). Tyto paralelní akce v POSH reaktivním plánu by měly stejnou strukturu jako mají kompetence (C), a to včetně spouštěčů (triggerů), jejich význam by však byl jiný. Zatímco kompetence vybírá postupně akci, která má být vykonána, PA by spustilo postupně všechny akce se splněnými spouštěči v rámci aktuálního cyklu logiky. Z pohledu prostředí se ale bude zdát, že akce proběhly najednou. Tento přístup řeší i většinu konfliktů efektů akcí. Spouštění akcí by mělo probíhat odspodu tak, aby nakonec převážily efekty nejvrchnějších akcí, a byl tak zachován hierarchický přístup. Nicméně předpokládáme využití PA především na nekonfliktní akce.

Díky těmto *paralelním akcím* je možné pomocí dekompozice elegantně vytvořit například i bojové chování. Ukázka možné textové reprezentace takového plánu je na obrázku 5.1. Paralelní akce *FightAndDodge* se sestává ze tří paralelních prvků: *Shoot* zajišťuje střelbu po nepříteli, *Dodge*, umožňuje uskakovat před projektily, pokud jsou nějaké detekovány, a *GetBetterShot*, přibližující bota k nepříteli pro možnost lepšího zásahu. Pokud by nastal konflikt mezi *Dodge* a *GetBetterShot* zvítězí *Dodge*, protože je výše v hierarchii.

Námi navržený přístup zapadá do metodiky BOD a měl by umožnit snadné vytváření složitějších chování vyžadujících paralelní vykonávání akcí.

```
(
  (PA FightAndDodge
    (elements
      ((Shoot ShootEnemy))
      ((Dodge (trigger ((IncomingBullet)))
              DodgeBullets
          ))
          ((GetBetterShot RunToEnemy))
    )
  )

  (DC life
    (drives
      ((Fight
        (trigger ((SeeEnemy)))
        FightAndDodge
          ))
      ((Explore RunRandomly))
    )
  )
)
```

Obrázek 5.1: Návrh jednoduchého plánu využívajícího paralelní akce

Seznam použité literatury

[1] Brom, C., Gemrot, J., Bída, M., Burkert, O., Partington, S., Bryson, J., *POSH Tools for Game Agent Development by Students and Non-Programmers.* Proceedings of CGAMES 2006, s. 126-135, 2006.

[2] Brooks, R. *A robust layered control system for a mobile robot*, IEEE Journal of Robotics and Automation, s. 14–23, 1986.

[3] Bryson, J. *The Behavior-Oriented Design of Modular Agent Intelligence.* Agent technologies, infrastructures, tools, and applications for e-services, s. 61-76. Springer, 2003.

[4] Galli, L., Loiacono, D., Lanzi, P. *Learning a Context-aware Weapon Selection Policy for Unreal Tournament III.* Proceedings of the 5th international conference on Computational Intelligence and Games, s. 310-316, IEEE Press, 2009. ISBN: 978-1-4244-4814-2

[5] Gat, E. *Three-layer architectures.* Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems, s. 195–210. MIT Press, 1998.

[6] Gemrot, J., Kadlec, R., Bida, M., Burkert, O., Pibil, R., Havlicek, J., Zemcak, L., Simlovic, J., Vansa, R., Stolba, M., Plch, T., Brom, C. *Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents.* Agents for Games and Simulations, s. 1-15, Springer, 2009.

[7] Havlíček, J. *Tools for virtual agent behavior specification in POSH*, diplomová práce, Univerzita Karlova, Matematicko-fyzikální fakulta, 2013.

[8] Hexmoor, H., Horswill, I., Kortenkamp, D. *Software architectures for hardware agents.* Journal of Experimental & Theoretical Artificial Intelligence. Volume 9, Issue 2, s. 147-156, 1997.

[9] Kadlec, R. *Evolution of intelligent agent behaviour in computer games*, diplomová práce, Univerzita Karlova, Matematicko-fyzikální fakulta, 2008.

[10] Koza, J. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, 1998. ISBN 0-262-11170-5

[11] Koza, J. *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, 1994. ISBN 0-262-11189-6

[12] Koza, J. *Human-competitive results produced by genetic programming.* Genetic Programming and Evolvable Machines archive. Volume 11 Issue 3-4, s. 251-284, 2010.

[13] *Netbeans IDE*, The Smarter and Faster Way to Code, [software] [online]. 2015. [cit. 2015-02-20]. URL: <`http://netbeans.org`>.

[14] Nwana, H. *Software Agents: An Overview.* Knowledge Engineering Review, Volume 11, No. 3, s. 1-40, 1996.

[15] Panait, L, Luke, S. *Cooperative Multi-Agent Learning: The State of the Art.* Autonomous Agents and Multi-Agent Systems 11, no. 3, s. 387-434, 2005.

[16] Poli, R., Langdon, W., McPhee, N. *A field guide to genetic programming*, [online]. 2008 [cit. 2014-01-10]. URL: <`http://www.gp-field-guide.org.uk`>.

[17] Sipper, M. *Evolved to Win*, Lulu, 2011. ISBN 978-1-4709-7283-7

[18] Van Hoorn, N., Togelius, J., Schmidhuber, J. *Hierarchical Controller-learning in a First Person Shooter.* Proceedings of the IEEE Symposium on Computational Intelligence and Games, s. 294-301, IEEE Press, 2009. ISBN: 978-1-4244-4814-2

[19] Vidal, J. *Fundamentals of Multiagent Systems with NetLogo Examples*, [online] 2009 [cit. 2015-01-10]. URL: <`http://multiagent.com/p/fundamentals-of-multiagent-systems.html`>.

[20] Witzany, T. *Adaptive Agent in a FPS Game*, bakalářská práce, Univerzita Karlova, Matematicko-fyzikální fakulta, 2013.

[21] Wooldridge, M. *Intelligent Agents.* Multi-Agent Systems (second edition), s 3-50, MIT Press, 2013.

[22] Wooldridge, M., Jennings, N. *Intelligent Agents: Theory and Practice.* Knowledge Engineering Review 10(2), 1995.

[23] Zelinka, M. *Using yaPOSH for CTF team behaviour*, bakalářská práce, Univerzita Karlova, Matematicko-fyzikální fakulta, 2014.

Seznam tabulek

Tabulka 3.1 - Porovnání prostředí (strana 17)
Tabulka 3.2 - Gramatiky popisující struktury neterminálů (strana 19)
Tabulka 3.3 - Ukázky plánů a odpovídající neterminály (strana 20)

Seznam použitých zkratek

ADF - Automatically defined functions, automaticky definované funkce. Technika genetického programování.
AP - Action Pattern, vzorec chování.
BDI - Belief-Desire-Intention, model deliberativního agenta.
BOD - Behavior-Oriented Design, design se zaměřením na chování. Metodika návrhu a tvorby chování inteligentních agentů.
C - Competence, kompetence.
CTF - Capture the Flag, boj o vlajky. Herní mód, kdy soupeřící týmy se snaží navzájem si ukrást vlajku.
DC - Drive Collection, kolekce chování.
DM - Death Match, boj na život a na smrt. Herní mód, kdy protivníci bojují proti sobě.
EA - Evoluční algoritmy.
GP - Genetic Programming, genetické programování. Technika umělé evoluce.
IDE - Integrated Development Environment, integrované vývojové prostředí.
PA - Paralelní akce.
POSH [reaktivní plán] - Parallel-rooted, Ordered, Slip-stack Hierarchical [reaktivní plán].
TDM - Team Death Match, týmový boj na život a na smrt. Týmový herní mód.
UT - Unreal Tournament 2004.
yaPOSH [reaktivní plán] - Yet another Parallel-rooted, Ordered, Slip-stack Hierarchical [reaktivní plán].

Přílohy

CD-ROM

1. *Dokumentace*. Dokumentace ve formátu JavaDoc.
   Umístění: `cdrom/javdoc/`.
2. *Zdrojové kódy*. Zdrojové kódy pro projekt.
   Umístění: `cdrom/bot/`.
3. *Experimenty*. Data sesbíraná v rámci experimentů.
   Umístění: `cdrom/experimenty/`.
4. *Video*. Videa s ukázkami běhu.
   Umístění: `cdrom/video/`.