VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# ROZHRANÍ PRO TVORBU HERNÍCH AGENTŮ V PROSTŘEDÍ JASON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAKUB RUSNÁK
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# ROZHRANÍ PRO TVORBU HERNÍCH AGENTŮ V PROSTŘEDÍ JASON

INTERFACE FOR CREATION OF GAME AGENTS IN JASON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAKUB RUSNÁK
AUTHOR

VEDOUCÍ PRÁCE Ing. KRÁL JIŘÍ,
SUPERVISOR

BRNO 2015

# Abstrakt

Práce se zabývá tvorbou rozhraní pro vytváření agentů v jazyce Jason pro hru Unreal Tournament 2004. Čtenář se seznámí s teorií agentních systémů, s hrou Unreal Tournament a prostředím GameBots. V práci je popsán návrh tohoto rozhraní a jeho implementace. Řeší také problém hledání cesty na mapě. Popisuje vytvoření vlastního bota pro hraní deathmatch módu. Na závěr je vyhodnocena efektivita implementovaného řešení.



# Abstract

Thesis deals with creation of interface for creating agents in Jason programming language for Unreal Tournament 2004. Reader is introduced to theory of agent systems, UnrealTournament game and GameBots environment. The paper describes design of this interface and it's implementation. It also deals with the problem of finding path for bots in game. It desribes creation of your own bot using this interface. Finally, efficiency of solution is evaluated.



# Klíčová slova

multiagentní systém, umělá inteligence, Jason, AgentSpeak, BDI agent, Unreal Tournament, GameBots, hledání cesty

# Keywords

multi-agent system, artificial intelligence, Jason, AgentSpeak, BDI agent, Unreal Tournament, GameBots, pathfinding

# Citace

Jakub Rusnák: Rozhraní pro tvorbu herních agentů
v prostředí Jason, bakalářská práce, Brno, FIT VUT v Brně, 2015

# Rozhraní pro tvorbu herních agentů v prostředí Jason

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jiřího Krále.

. . . . . .
Jakub Rusnák
5. května 2015

## Poděkování

Rád bych poděkoval svému vedoucímu bakalářské práce, panu Ing. Jiřímu Králi, za cenné rady a připomínky.

# Obsah

# Kapitola 1

# Úvod

Umelá inteligencia vzbudzuje záujem v čoraz viac odvetviach vývoja softwaru a výskumu. Preniká aj do počítačových hier, pri vytváraní rôznych botov alebo pri asistovaní hráčovi s jeho úlohou. Logické programovanie zastáva pri programovaní inteligentných agentov významnú úlohu. Jeho vyučovanie však môže byť problematické, pretože sa odlišuje od funckionálneho alebo objektového programovania, s ktorým študenti zvyčajne začínajú. Zaujímavou formou osvojenia si zásad logického programovania môže byť tvorba inteligentných agentov (botov) pre hry. Práve týmto sa zaoberá táto práca. Cieľom je vytvoriť rozhranie medzi jazykom Jason [6], hrou Unreal Tournament 2004 [10] a prostredím GameBots [11].

Agenti sú inteligentné entity, ktoré pozorujú svet svojimi senzormi a ovplývňujú ho efektormy. Reagujú na podnety z prostredia a na základe jeho stavu robia rozhodnutia, aby uspeli vo svojich cieľoch.

Typickým prostredím pre takýchto agentov sú počítačové hry. Hráči čoraz viac požadujú, aby ich počítačový protivníci alebo asistenti boli inteligentnejší. Neustále sa zvyšujú požiadavky na agentov a to posúva ich vývoj dopredu. Ich vlastnosti, najmä schopnosť riešiť zložité problémy, reaktivita na zmeny prostredia, či samostatné rozhodovanie sa dá v hrách veľmi dobre uplatniť.

Úvod do problematiky ponúka sekcia 2. Obsahuje informácie o tom, čo sú to agenti, multiagentné systémy a ich implementáciu. Popisuje tiež hru, s ktorou budeme pracovať a jej modifikáciu GameBots.

V kapitole 3 je popísaný návrh rozhrania spájajúceho tieto komponenty. Kapitola 4 vysvetľuje, ako sú implementované. To, ako si vytvoriť vlastného agenta s použitím rozhrania si popíšeme v kapitole 6.

Pri pohybe agenta na mape riešime problém hľadania cesty grafom. Mapy sú tvorené množinou navigačných bodov. Je potrebné nájsť tú najlepšiu cestu a to ešte k tomu rýchlo, aby agent nestál pridlho na jednom mieste. To je obsahom kapitoly 5.

Implementované riešenie sme podrobili rôznym testom. Zamerali sme sa najmä na časovú náročnosť jednotlivých úkonov. Výsledky sú zhodnotené v sekcii 7.

# Kapitola 2

# Teoretický základ

## 2.1 Agent

Pri programovaní sa stretávame s problémom, keď potrebujeme aby náš program reagoval na udalosti a mal istú dávku samostatnosti. Jeho riešenie vo funkcionálnych jazykoch často nie je jednoduché, ani elegantné. Systémy, v ktorých potrebujeme reagovať na udalosti sa nazývajú reaktívne systémy. V posledných desaťročiach vedci skúmali ako možno takéto systémy efektívne implementovať. Vyvinuli agentné systémy, v ktorých centre stojí agent.

Agent je reaktívny, autonómny systém. Je umiestnený v prostredí a spolu s ďalšími agentami tvoria multiagentný systém. Prostredie sa často dynamicky a nedeterministicky mení. Agent musí na tieto zmeny včas reagovať. Vníma svoje prostredie senzormi a môže ho ovplyvňovať efektormi. Je autonómny, pod čím rozumieme, že dokáže existovať samostatne a tiež sa vie samostatne rozhodovať. Ak narazí na problém, sám zvolí jeho vhodné riešenie. To platí aj o úlohách, ktoré mu môže užívateľ alebo iný agent zadať. Aby vo svojom snažení uspel, aktívne sleduje svoje ciele. Je však žiaduce aby sa správal racionálne. Ak mu bude jasné, že tieto ciele nedokáže splniť, či už kôli stavu prostredia, alebo už nemajú význam, tak svoju snahu vzdá.

Niekedy chceme aby agenti medzi sebou komunikovali. Môžu si navzájom vymieňať informácie, koordinovať svoje akcie alebo delegovať ciele na iných agentov. Takéto sociálne schopnosti sú veľmi užitočné, pretože môžu zjednodušiť a zefektívniť prácu. Pokrokom v rozvoji multiagentných systémov bolo vytvorenie AgentSpeaku [15].

## 2.2 AgentSpeak

AgentSpeak je logický programovací jazyk pre agentné systémy, ktorý používa BDI architektúru (z angl. Belief-Desire-Intention). BDI model vychádza z modelu praktického uvažovania človeka. Agent má predstavy o prostredí a stave, v ktorom sa nachádza. Tie nemusia odpovedať skutočnému prostrediu, nemusia byť pravdivé a môžu byť dokonca navzájom protichodné. Agentove túžby sú ciele, ktoré by chcel v budúcnosti dosiahnuť. Zatiaľ ich však nenapĺňa. Napríklad preto, že prostredie nie je v stave, ktoré je pre ich úspech nevyhnutné, alebo je agent práve zaneprázdnený inou činnosťou. Zámery agenta sú túžby, ktoré sa rozhodol realizovať a aktívne sa snaží aby uspeli. Realizuje to vykonávaním akcií, ktoré ovplyvňujú prostredie.

V AgentSpeaku majú predstavy formu formulí predikátovej logiky. Atomickou predstavou je predikát vo forme $b(t_1, t_2 \ldots t_n)$. Atomická prestava alebo jej negácia sú literály.

Rozlišujeme medzi dvoma typmi cieľov, dosahovacie a testovacie. Dosahovacie ciele sú uvedené znakom `!`. Nimi chce agent dosiahnuť taký stav prostredia, aby platil daný predikát. Testovacie ciele sa odlišujú znakom `?`. Vracajú unifikáciu medzi ich premennou a jednou z agentových predstáv.

Tento systém je riadený udalosťami. Sú generované pri pridávaní alebo odoberaní predstáv ($+b$, $-b$), nadobúdaní alebo neúspechu dosahovacích cieľov ($+!g$, $-!g$) a cieľov testovacích ($+?g$, $-?g$). Interpreter spravuje zoznam udalostí a vyberá jednu udalosť, na ktorú bude agent daný úvahový cyklus reagovať.

Základom programu v tomto jazyku sú plány. Skladajú sa zo spúšťacej udalosti, kontextu a súboru akcií alebo cieľov v tvare `spúšťacia udalosť: kontext <- telo.`. Keď interpreter vyberie udalosť, hľadá k nej plány, ktoré na ňu reagujú. V prípade, že také relevantné plány nájde pozrie sa na ich kontext. Kontext je súbor logických podmienok, ktoré musia byť pravdivé, aby sa plán mohol uskutočniť. Zvyčajne reprezentujú stav prostredia, ktorý musí byť splnený aby mal plán šancu na úspech. Teraz máme množinu aplikovateľných plánov, ktoré reagujú na istú udalosť a ich kontexty dovoľujú ich použitie. Vyberieme jeden a začneme ho uskutočňovať.

Plán je súbor akcií alebo cieľov, ktoré sa vykonávajú postupne. Tieto akcie môžu byť akcie ovplyvňujúce prostredie (efektormi), môžu pridávať alebo odoberať predstavy, ale aj dosahovať a testovať ciele. Vybratý plán sa stane zámerom. Je reprezentovaný ako zásobník plánov. Ak sa pri vykonávaní plánu vyskytne požiadavka na dosiahnutie ďalšieho cieľa, nájde sa relevantný plán a vloží sa na vrch tohto zásobníku. Plán uspeje ak uspejú všetky jeho akcie.

Jazyk používa tiež pravidlá. Používajú sa na odvodzovanie nových znalostí. Príkladom môže byť `my_location(X,Y,Z):- my_id(ME) & location(ME,X,Y,Z).`. Na ľavej strane sa musí nachádzať jeden literál. Aby platil, podmienky na pravej strane musia uspieť.

## 2.3 Jason

Jason [6] je rozšírením AgentSpeaku v jazyku Java. Dodržiava princípy BDI modelu. Prináša aj niekoľko noviniek naproti AgentSpeaku.

Boli pridané anotácie. Sú to listy termov uvedené v hranatých zátvorkách. Majú najmä informatívny charakter, ale programátor ich môže s výhodou použiť k praktickým účelom. Viažu sa ku konkrétnej znalosti a môžu upresňovať jej význam alebo pôvod. Dôležitou anotáciou je `[source(s)]`. Udáva zdroj predstavy. Pre vnemy z prostredia má tvar `[source(percept)]`, pre vlastné poznámky `[source(self)]`. Anotácie môžeme použiť pri programovaní na skontrolovanie zdroja predstavy alebo jej doby platnosti.

Premenné je možné použiť všade tam, kde je očakávaná atomická formula, napríklad vo všetkých častiach plánov.

Plány môžu mať štítky, čo sú atomické formule s anotáciami. Štítky je možno použiť k jednoduchšej identifikácii plánu v kóde.

Dôležité časti Jasonu sú prostredie, agenti a úvahový cyklus. Prostredie môže byť abstrakciou reálneho prostredia, v ktorom agent operuje, alebo môže byť simulované. Je tvorené triedou `Environment`. Táto trieda spravuje vnemy z prostredia a tiež môže poskytovať akcie pre všetkých agentov.

Agent je tvorený bázou znalostí, množinou plánov, výberovými funkciami, aktualizačnou funkciou, ktorá obnovuje agentovu bázu znalostí, reviznou funkciou, ktorá upravuje vnemy do želanej podoby a štruktúry pre čakajúce udalosti a zámery. Takmer všetko si môžu užívatelia prispôsobiť podľa svojej potreby. V súbore s príponou `.asl` je kód agenta v

tomto jazyku. Obsahuje sadu počiatočných znalostí a cieľov, ktoré má agent pri spustení programu a plány.

Úvahový cyklus riadi celý program. Cyklus sa skladá z niekoľkých častí. Najprv aktualizuje agentove predstavy, aby odpovedali súčasnému stavu prostredia. Pre všetky zmeny sú generované odpovedajúce udalosti. Prijmú sa správy od ostatných agentov. Následne sa vyberie udalosť, na ktorú bude agent tento cyklus reagovať. Nájde sa plán a vytvorí sa nový zámer. Zvyčajne má agent niekoľko zámerov súčasne. Vyberie sa jeden z nich a vykoná sa akcia na vrchole jeho zásobníka.

## 2.4 Unreal Tournament 2004

Unreal Tournament 2004 je futuristická, akčná, strieľacia hra pre jedného alebo viacerých hráčov. Bola vytvorená spoločnosťami Epic Games [3] a Digital Extremes [1]. Svojho času bola veľmi populárna a získala si mnoho fanúšikov. Za svoj úspech vďačí výbornej hrateľnosti, kvalitnej grafiky a početnými hrateľnými módmy.

Centrom hry je zápas, ktorý sa odohráva na nejakej mape. Mapa obsahuje sieť navigačných bodov a súbor dostupných vecí. Sú nimi zbrane, náboje, lekárničky, adrenalín a štíty. Každá z týchto vecí má svoju objavovaciu pozíciu. Keď hráč vec zoberie, zmizne zo svojej pozície a objaví sa tam opäť po uplynutí časového intervalu. Čím je vec vzácnejšia, tým dlhšie trvá jej obnovenie.

Hráči sa môžu po mape voľne pohybovať. Pohyb zahŕňa beh, zakrádanie sa, prikrčenie, výskok, dvojitý výskok, dodge (rýchle uhnutie sa) a odskok od steny. Niektoré mapy ponúkajú translokátor, ktorým sa hráči premiestnujú na akúkoľvek pozíciu v dosahu.

Hráči zbierajú zbrane a predmety na mape. Tie sa automaticky pridávajú do inventára (zbrane) alebo sa okamžite aplikujú (adrenalín, lekárnička a iné). Každý hráč má 100 životov, špeciálnymi lekárničkami sa dá zvýšiť až na 199. Pri hre získava adrenalín, buď zbieraním piluliek alebo zabíjaním iných hráčov. Pribúda od 0 do 100 a pri maxime môže hráč použiť vylepšenia, ktoré po zapnutí adrenalín odčerpávajú. Hráči môžu tiež získať brnenie zbieraním predmetov s modelom štítu. Hodnota sa pohybuje od 0 do 150 a chová sa ako extra životy. Po smrti má hráč základný počet životov, žiadne brnenie a dve základné zbrane, ale adrenalín pretrváva s hodnotou akú mal hráč v čase smrti. Kategórie vecí na mape:

- zbraň — Hra ponúka široké spektrum zbraní. Niektoré sa hodia na ničenie vozidiel, iné na boj zblízka či na diaľku, niektoré závisia na zvolenom hernom móde alebo na nastavení zápasu. Každá zbraň sa po zodvihnutí pridá do inventára s istým množstvom počiatočných nábojov. Zbrane majú dva módy strieľania, primárny a sekundárny, ktoré závisia od jej typu.

- náboj — Ďalšími objektami sú náboje. Väčšina zbraní má na mape rozmiestnené svoje náboje, typicky v okolí spawnu zbrane.

- lekárnička — Na mape sa nachádzajú lekárničky (Health Pack), ktorými je možné si živote opäť doplniť pri zranení o 25 bodov, nemožno však presiahnuť hodnotu 100. Ďalším typom lekárničky je Mini Health Pack. Pridáva len 5 životov, zvyšuje však maximum do hodnoty 199. Na niektorých mapách sa nachádza Mega Health Pack, ktorý pridáva 100 životov a tiež zvyšuje ich maximálny počet.

- adrenalín — Pilulky pridávajú 2 body adrenalínu. Pri dosiahnutí tohoto maxima môžu hráči využívať špeciálne kombá, napríklad neviditeľnosť, rýchlosť, regenerácia, besne-

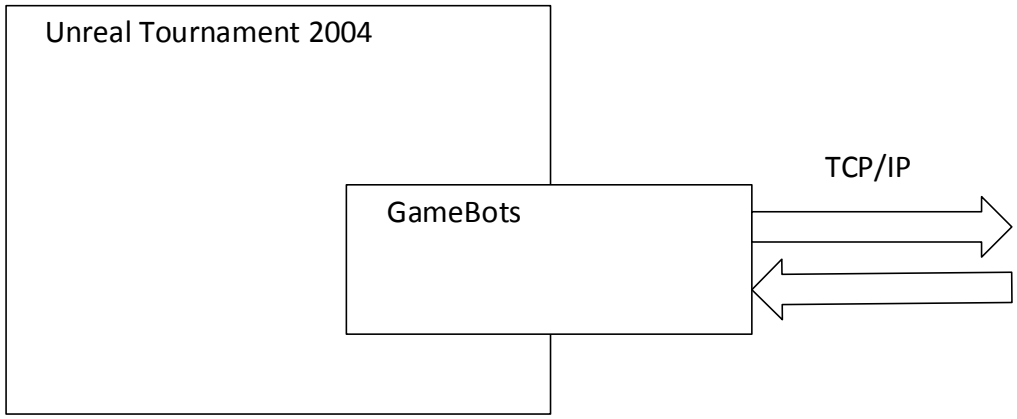
nie a zmenšenie, ktoré postupne odčerpávaju adrenalín. Zabitie protihráča tiež generuje adrenalín.

- brnenie — Mapy obsahujú tiež štíty, ktoré chránia hráčov pred zranením. Malý štít pridáva 50 (do maxima 100), veľký 100 (do maxima 150) ochrany. Správajú sa ako extra životy.
- dvojitý damage — Je to vzácna vec, ktorá po dobu 30 sekúnd zvýši všetky spôsobené zranenia na dvojnásobok.

Tvorcovia hry umožnili používateľom vytvárať vlastné mapy, zbrane, či dokonca módy. Vďaka tejto možnosti bolo vytvorených množstvo zaujímavých máp a módov. Jednou z takýchto modifikácií je GameBots [11].

## 2.5 GameBots

GameBots bol vytvorený na University of Southern California na Information Sciences Institute. Jeho tvorcami sú Andrew N. Marshal a Gal Kaminka. Bol vytvorený zo zámerom skúmať umelú inteligenciu. Bol ďalej vylepšovaný a optimalizovaný Michalom Bídom, Martinom Černým, Jakubom Gemrotom a Cyrilom Bromom na univerzite v Prahe. Je to mód pre UT2004, ktorý umožňuje ovládať botov pomocou textového protokolu. Po pripojení bota na server sa mu posielajú textové správy o jeho stave a stave okolného prostredia. Týmto spojením môže agent ovládať svojho bota súborom textových príkazov. Kompletný zoznam správ možno nájsť na [5].

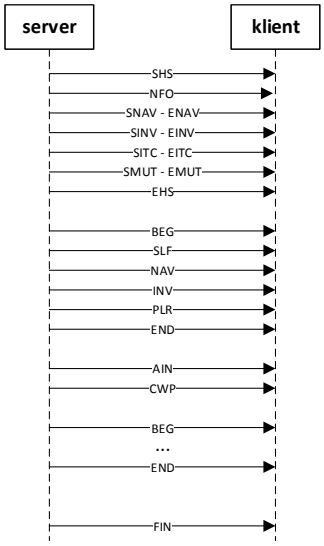



Obrázek 2.1: Diagram Unreal Tournamentu s GameBots.

### 2.5.1 Textový protokol

Všetky správy majú textový formát v tvare: `PLR {Name Arnold} {Location 0,0,0} {Rotation 32000,0,0} {Health 100} {Height 170.59} {IsHuman False}`

Správy sa skladajú z typu (v tomto prípade PLR), ktorý je na začiatku správy až do prvého bieleho znaku. Za ním nasleduje zoznam atribútov správy (niektoré správy atribúty




Obrázek 2.2: Správy posielané serverom.

nemajú). Atribút je uzavretý v zložených zátvorkách a má názov atribútu od ľavej zátvorky do prvého bieleho znaku. Hodnota atribútu je zvyšok po uzatváraciu zátvorku.

Datové typy používané v GameBots sú celé a reálne čísla, reťazce znakov, vektory (poloha pozostávajúca z hodnôt pre x, y a z súradnicu) a uhly. Uhol je rozdelený na unreal jednotky. Pre rotáciu 360 stupňov máme 65535 unreal jednotiek.

### 2.5.2 Pripojenia

GameBots pozná tri typy pripojení. Pre bota, server a pozorovateľa. Je treba poznať adresu servera a jeho port.

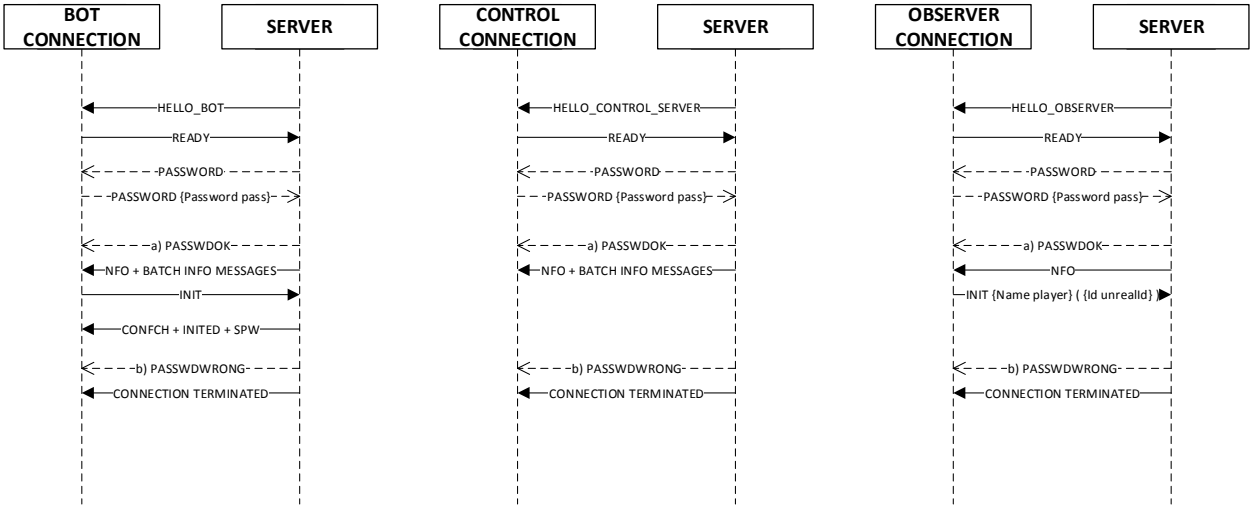
Po pripojení sa posielajú inicializačné správy 2.3, ktoré závisia od typu pripojenia. Týmto spojením je tiež možno bota ovládať. Východzia hodnota portu pripojenia pre bota je 3000. Server posiela klientovi tieto správy 2.2:

- handshake — Je séria správ posielaných na začiatku spojenia. Je uzatvorená do dvojice správ. Začína sa správou SHS, končí EHS. Obsahuje zoznam vecí, ktoré sa počas celého kola nemenia. Sú to napríklad navigačné body, veci na mape a ich kategórie alebo mutátory.

- synchrónne správy — Po handshaku začne server posielať v pravidelných intervaloch informácie o stave bota. Jeho polohu, životy, adrenalín, veci, ktoré vidí a tak ďalej. Tieto správy sú ohraničené správami BEG a END.

- asynchrónne správy — Okolo bota sa dejú udalosti, ktoré nemožno predvídať. Tie sa posielajú asynchrónne a zvyčajne ich tvorí jedna správa s informáciami o danej udalosti. Príkladom môže byť zasiahnutie hráča či zodvihnutie zbrane.

Pripojenie pre ovládanie servera môže kontrolovať priebeh hry. Používa sa na zmeny mapy, vyhadzovanie hráčov, spúšťanie príkazov v konzole. Pripája sa na port 3001. Po odoslaní




Obrázek 2.3: Inicializačné správy.

handshaku dostáva synchrónne správy o všetkých hráčoch na serveri. Aby toto pripojenie fungovalo musí byť nastavené `bAllowControlServer=true` v súbore GameBots2004.ini.

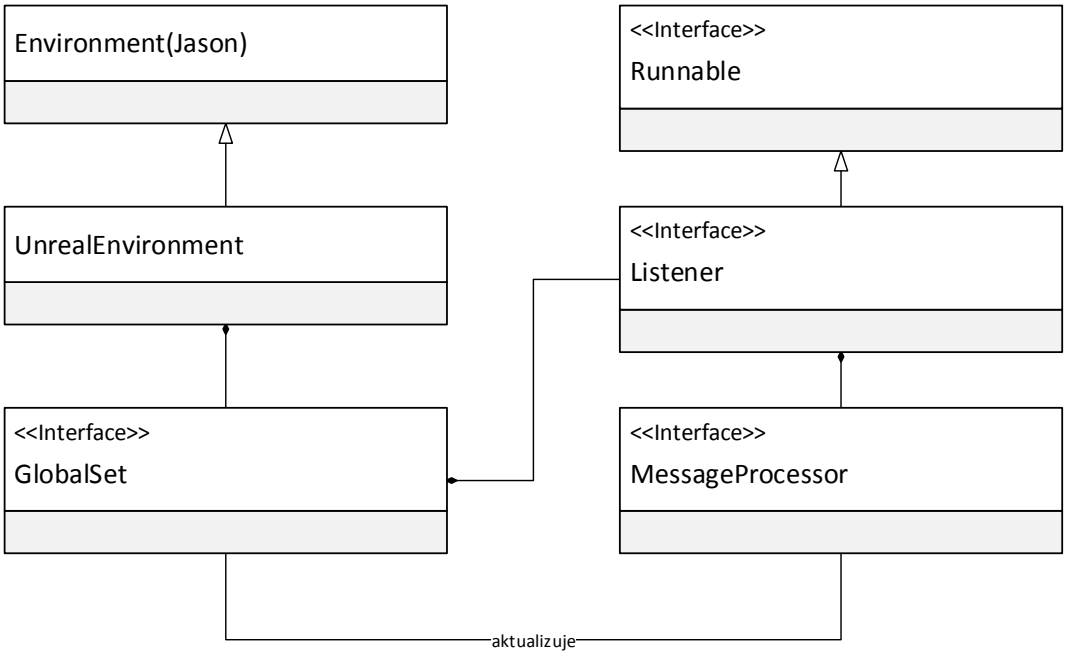
Pozorovateľ sa pripojuje na port 3002. Po zvolení hráča sa posielajú informácie o všetkých akciách, ktoré hráč vykonal. Na toto pripojenie je treba nastaviť `bAllowObservingServer=true` v súbore GameBots2004.ini.

# Kapitola 3

# Návrh

Na prepojenie Jasonu s GameBots upravíme triedu pre prostredie (`Environment`) a triedu pre agentov (`AgArch`) tak, aby sa dokázali pripojiť a prijímať správy zo serveru a tiež aby poskytovali užitočné funkcie svojim agentom.

## 3.1 Prostredie




Obrázek 3.1: UML diagram prostredia.

`UnrealEnvironment` je triedou prostredia. Dedí od triedy Jasonu pre prostredie, `Environment`, a preto preberá niektoré jej funkcie. Požiadavky na túto triedu sú:

- Obsahuje informácie o mape. Sú nemenné, rovnaké a dostupné pre všetkých agentov.
- Poskytuje globálne vnemy (predstavy).

- Informácie o adrese a porte servera.
- Užitočné akcie pre agentov.

Prostredie poskytuje agentom informácie o zápase. Sú uložené v globálnom sete. Ten ich spravuje, aktualizuje a poskytuje užívateľom. Obsahuje zoznamy navigačných bodov, vecí a ich kategórií, mutátory, pohybujúce sa objekty a informácie o mape a hernom móde. Okrem toho, prostredie pridáva agentom vnemy o stave zápasu.

Obsahuje tiež niekoľko akcií pre agentov.

## 3.2 Set informácií

Združuje v sebe informácie o zápase alebo o botovi. Sú to zoznamy navigačných bodov, vecí na mape a ich kategórií, mutátorov, pohybujúcich sa objektov (mover), informácií o mape, informácií o stave bota (životy, pozícia, adrenalín a iné), vecí, čo bot vidí, udalosti, ktoré sa okolo neho stali a podobne. To aké informácie bude set obsahovať závisí od jeho účelu.

Napríklad prostredie má svoj vlastný set, ktorý obsahuje globálne informácie o práve prebiehajúcom zápase. Tieto informácie musia byť prístupné všetkým agentom, ktorý by s nimi chceli pracovať.

Agenti majú svoj vlastný set, ale informácie v ňom sa týkajú výhradne ich. Je v nich aktuálny stav bota a jeho okolia.

## 3.3 Spravovanie spojenia

Informácie o hre získavame z GameBots serveru. Ten ich svojim klientom posiela vo forme textového protokolu.

Po pripojení a odoslaní inicializačných správ, server posiela handshake. To je skupina správ uvedená správami SHS–EHS. Tu sa nachádzajú globálne informácie o zápase a mape. Tieto budú užitočné pre triedu prostredia.

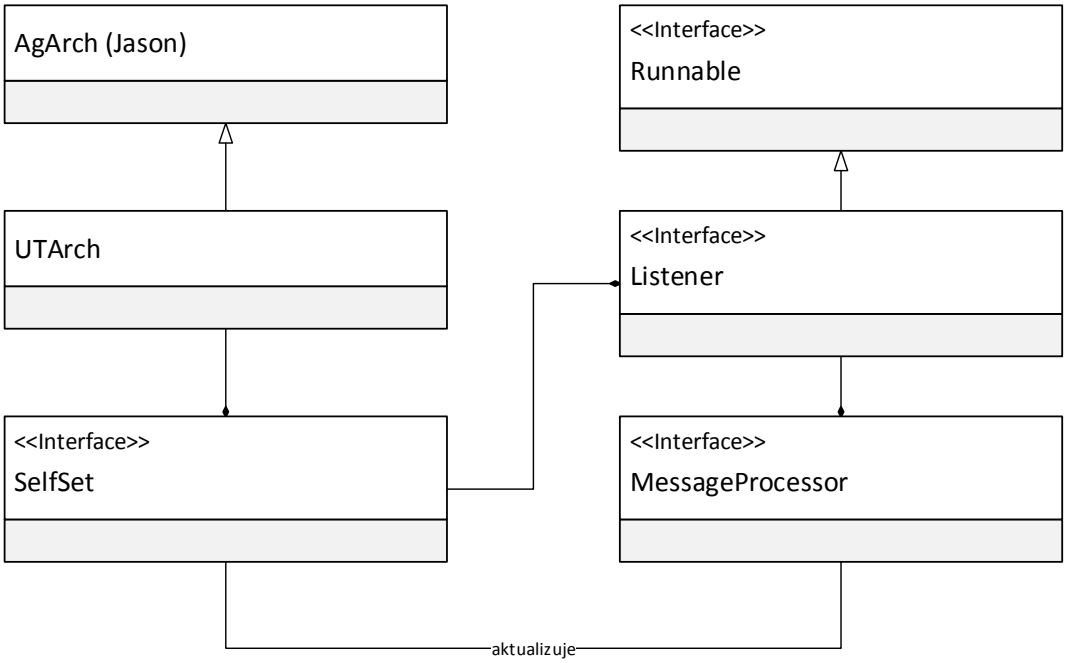
Ak sa pripájame na port pre bota, po odoslaní správy INIT sa v hre vytvorí bot, ktorého môžeme ovládať. GameBots okamžite začne posielať správy o jeho stave a udalostiach, ktoré sa stali v jeho okolí. Tieto informácie budú zaujímať jednotlivých agentov.

O spravovanie spojenia sa stará trieda `Listener`. Okrem pripojenia na GameBots server, spracováva prijímané správy a z reťazca znakov vytorí objekt správy (GBmessage 4.1) a predá ho ďalej na spracovanie. `Listener` beží na samostatnom vlákne aby svojimi vstupno-výstupnými operáciami neblokoval beh programu.

## 3.4 Spracovanie správ

Správy, ktoré vytvorí `Listener` je treba vytriediť, spracovať a naplniť nimi set informácií. Triedenie je potrebné, pretože nie všetky správy sú užitočné, niektoré len uvádzajú skupinu iných správ. Tiež nás zvyčajne nezaujímajú všetky správy. Prostredie potrebuje len informácie o zápase, kdežto agenta zaujímajú informácie, ktoré sa týkajú jeho. Rôzny agenti môžu dokonca chcieť prijímať rôzne správy. Toto triedenie je úlohou `MessageProcessor`u. Keď mu príde na spracovanie správa a je relevantná, vytvorí z nej objekt a uloží ho do setu informácií, ktorý spravuje.

## 3.5 Architektúra agenta




Obrázek 3.2: UML diagram agenta.

Architektúra sa stará o agentovu reakciu s jeho okolím (pozri Overal Agent Architecture [12]). Dôležitou súčasťou je set informácií, obsahujúci agentov stav, veci, ktoré vidí, udalosti, ktoré nastali a zbrane, ktoré má v inventári. Set má vlastný `Listener` a `MessageProcessor`, ktoré ho aktualizujú.

Informácie zo setu sa používajú na aktualizovanie agentovej báze znalostí o vnemy z prostredia.

Agent potrebuje akcie, ktorými by svoj svet mohol ovplyvňovať. Preto obsahuje základné príkazy pre ovládanie bota v Unreal Tournamente prostredníctvom GameBots protokolu.

# Kapitola 4

# Implementácia

V tejto kapitole si popíšeme ako sú jednotlivé časti rozhrania implementované. Začneme datovými typmi, ktoré používa a ďalej popíšeme prostredie a agenta.

## 4.1 GBmessage

Vytvorí z textovej správy posielanej GameBots serverom objekt, charakterizovaný svojím typom a mapou kľúčov a hodnôt. Pre podrobnosti ohľadom štruktúry GameBots správy pozri sekciu 2.5.1. Takéto správy spracúva `MessageProcessor`.

## 4.2 UTobject

Základný objekt, s ktorým pracujeme. Je charakterizovaný typom a mapou atribútov. Niekedy môže obsahovať zoznam ďalších objektov. Platí to napríklad pre navigačné body, kde okrem informácií o danom bode ešte obsahuje zoznam susediacich uzlov s informáciami o ich prepojení. Jeho hodnoty sú po inicializácii nemenné.

## 4.3 Location

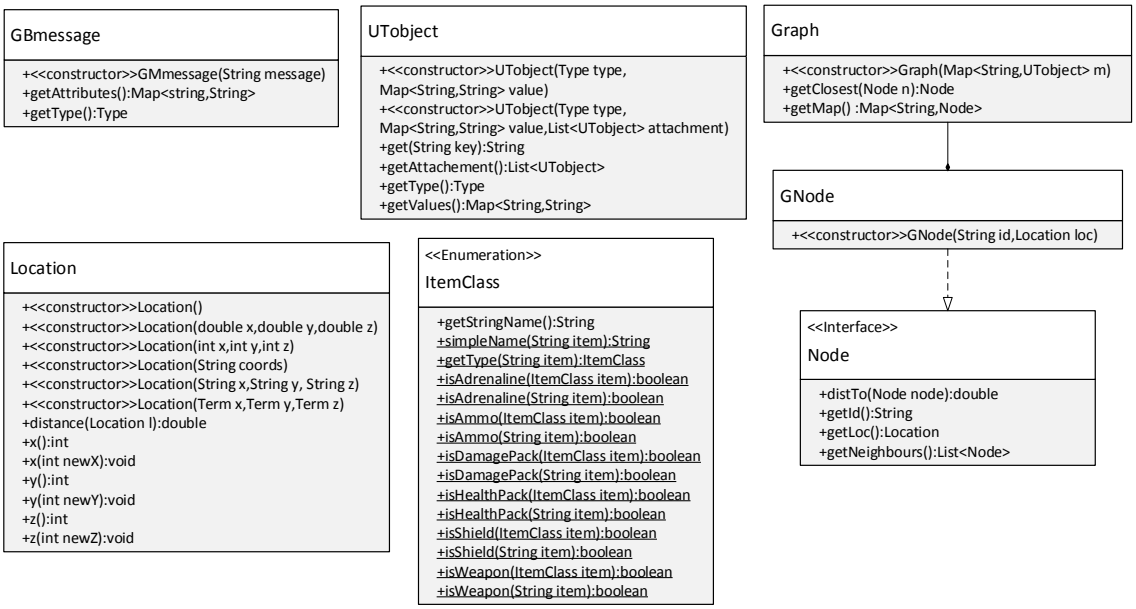
Udáva pozíciu na mape v 3D súradniciach. Informácie sú uložené vo forme celých čísel, hoci Unreal Tournament používa čísla desatinné. Odchýlka je však taká malá, že ju v praxi takmer nepostrehneme. Celé čísla navyše urýchlia výpočty a umožnia presnejšie porovnávanie dvoch bodov.

Trieda poskytuje funkciu na výpočet Euklidovskej vzdialenosti medzi touto lokáciou a bodom, zadaným v parametri.

Ponúka širokú škálu konštruktorov, aby ju bolo možné jednoducho používať zo všetkých oblastí implementácie.

## 4.4 ItemClass

Tento vymenovaný typ slúži na klasifikáciu vecí. Veci zahŕňajú zbrane, náboje, lekárničky, adrenalín, brnenie a dvojnásobný damage. Poskytuje funkcie na určenie typu veci. Obsahuje aj zjednodušené meno objektu v reťazci. Vstupy môžu byť aj reťazce z GameBots




Obrázek 4.1: UML diagram tried pre dáta.

správ, napríklad Id alebo Type atribút a metóda `getType(String)` vráti typ veci. Napríklad pre XPickups.MiniHealthPack alebo DM-Corrugation.MiniHealthPack0id37 vráti `ItemClass.MINIHEALTHPACK`.

## 4.5 Graph

Vytvorí mapu uzlov (`Node`) z mapy navigačných bodov Unreal Tournamentu. Túto mapu používa pathfinder. Uzol má svoje id, pozíciu a zoznam susedov.

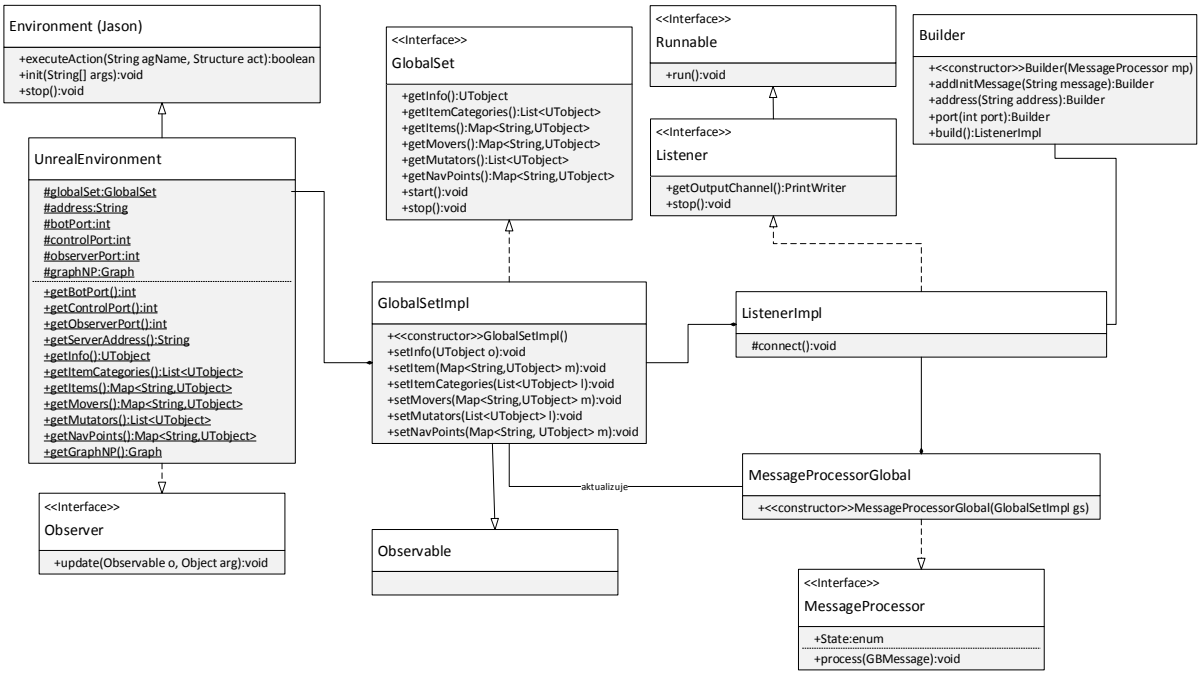
Metóda `getClosest(Node n)` nájde najbližší navigačný bod od bodu zadanom v parametri.

## 4.6 UnrealEnvironment

Prostredie je reprezentované triedou `UnrealEnvironment`. Jeho najdôležitejšou komponentou je set informácií `GlobalSet`, ktorý obsahuje informácie o zápase. Klienti k nim môžu pristupovať pomocou statických metód triedy prostredia. Rovnaké metódy sú aj pre informácie o adrese a porte servera.

Trieda spravuje globálny set a používa ho na získanie informácií o hre. Set implementuje triedu `Observable` a prostredie je upozornené pri zmene hodnôt. Informácie, ktoré obsahuje sú popísané v kapitole 4.7.

Keď sa pridajú navigačné body mapy, vytvoríme z nich graf. Ten používa algoritmus na hľadanie cesty. Agenti ho môžu využiť, ak chcú nájsť cestu zo svojej pozície na nejaký navigačný bod mapy. Je prístupný akciou `get_path(X,Y,Z,[dest(ID,Xd,Yd,Zd)])`. X, Y, Z sú štartovacie súradnice. Štvrtý argument je zoznam destinácií, ktoré chce agent na svojej ceste navštíviť. Algoritmus nájde túto cestu a pridá ju do agentovej báze znalostí ako súbor literálov tvaru `path(ID,N,X,Y,Z,F)`. ID je identifikátor bodu (pre navigačné body je to Id




Obrázek 4.2: UML diagram prostredia.

zo správ poslaných GameBots, pre ciele identifikátor zadaný užívateľom, začiatočný bod má ako identifikátor agentovo meno), N je poradové číslo začínajúce 0, F je flag pre pohyb. Je to typ prechodu medzi dvoma navigačnými bodmi, charakterizovaný celým číslom definovaným v [9]. Ak má hodnotu 1 prechod je jednoduchý, no napríklad pri 9 už treba aj vyskočiť.

Ďalšia akcia je `game_in_progress`, ktorá jednoducho vracia pravdu, ak je spustený nejaký zápas na serveri.

Prostredie agentom poskytuje aj niekoľko vnemov, ktoré súvisia s prebiehajúcou hrou. Ak prebieha zápas, v agentovej báze znalostí sa objavý vnem `match(ID)`, kde ID je názov mapy. Ak zápas skončil, agenti dostanú `match_end`.

## 4.7 GlobalSet

Obsahuje nasledujúce informácie:

- Navigačné body - majú pozíciu v 3D súradniciach, informáciu o veciach, ktoré sa na nich objavujú, zoznam susediacich uzlov.
- Veci - majú identifikátor, miesto kde ich možno získať, typ, množstvo.
- Kategórie vecí - každá vec má svoju kategóriu, tá udáva vzdialenosť na akú je zbraň vhodná, informácie o primárnej a sekundárnej streľbe, či množstvo nábojov alebo života, ktoré pridáva.
- Mutátor - má meno a id.
- Movery - alebo pohybujúce sa objekty, obsahujú informácie o pozícii, type, rýchlosti.

- Informácie o mape - majú názov mapy, typ hry, časový limit, podmienky ukončenia hry, napríklad pri deathmatchi požadované skóre.

Pre podrobnejšie informácie pozri GameBots protokol [5].

Tieto informácie pochádzajú zo servera a preto má `GlobalSet` vlastný `Listener`, ktorý toto spojenie udržiava. Pripája sa na adresu a port, ktoré získa od prostredia. Set rozširuje triedu `Observable`, aby prostredie mohlo reagovať na jeho zmeny.

## 4.8 ListenerImpl

Je to východzia trieda `Listener`a. Vytvára sa pomocou staviteľa, `Builder`a (návrhový vzor). Zadá sa `MessageProcessor`, ktorý bude spracovávať správy. Ďalšie parametre sú nepovinné, ako adresa a port servera a inicializačné správy. Ak ich užívateľ nenastaví, adresu získa z `UnrealEnvironment`u, takisto ako port (použije sa port pre spojenie bota) a nepošlú sa žiadne inicializačné správy.

Ak sa spojí zo serverom, správy ktoré prijme, spracuje do `GBmessage` a predá ďalej do spracovača správ. V prípade, že sa spojenie uzavrie (skončil zápas alebo je server offline) predá do `MessageProcessor`u hodnotu `null`.

Beží na samostatnom vlákne. Hneď po spustení sa pokúsi pripojiť na server. Prichádzajúce správy spracúvava až dokým nie je prerušený alebo spojenie uzavreté. Ak chce užívateľ `Listener` ukončiť, zavolá metódu `stop()`, ktorá uzavrie spojenie zo serverom.

## 4.9 MessageProcessorGlobal

Používa sa v `GlobalSet`e. Spracováva len správy z handshaku, ostatné ignoruje. Slúži na aktualizovanie informácií o mape. Prijme `GBmessage`, vytvorí z nej `UTobject` a uloží ho do globálneho setu informácií, kde bude užívateľom k dispozícii. Spracováva nasledujúce správy: NFO, NAV, INV, ITC, MUT, MOV.

## 4.10 UTArch

Je trieda agenta pre interakciu s okolím. Tiež má set informácií, ale odlišný od toho v prostredí. Je to `SelfSet` a obsahuje informácie, týkajúce sa agenta. Podrobnosti v nasledujúcej sekcii.

Dedí od triedy pre architektúru agenta z Jasonu, takže preberá jej zodpovednosti. Sú nimi spracovanie vnemov z prostredia, vykonávanie akcií a prijímanie správ od ostatných agentov.

Informácie o stave agenta a okolnom prostredí získava zo `SelfSet`u. Na ich základe pridáva zmeny do báze znalostí. Jason na to volá funkciu `perceive()`. V nej pre každý objekt v `SelfSet`e zavoláme funkciu `getPercepts(UTobject o)`, ktorá vráti zoznam vnemov z daného objektu. Napríklad pre navigačné body to bude `nav_point(NP)`, `location(ID,X,Y,Z)` a `visible(ID)`. Zoznam všetkých vnemov:

- `my_id(ID)` - Identifikátor bota v Unreal Tournamente.
- `location(ID,X,Y,Z)` - Pozícia entity danej identifikátorom. Môže to byť hráč, zbraň, navigačný bod.
- `inventory(Weapon)` - Zbraň, ktorú má v inventáry.

- `visible(ID)` - Identifikátor veci, ktorú vidím. Môže to byť hráč, vec, navigačný bod.
- `player(P)` - Entita P je hráč.
- `nav_point(NP)` - Entita NP je navigačný bod.
- `end_game` - Hra skončila.
- `connect` - Úspešne sa pripojil na server.
- `spawned` - Bot sa objavil na mape.
- `picked(ITEM,AMOUNT)` - Zodvihol vec ITEM s množstvom AMOUNT.
- `died` - Bot zomrel.

Aby agent mohol hrať hru, poskytneme mu set akcií, ktorými môže ovládať svojho bota. Sú vo funkcii `act(...)`, ktorú interpreter Jasonu volá, keď chce agent vykonať nejakú akciu. Základné akcie sú:

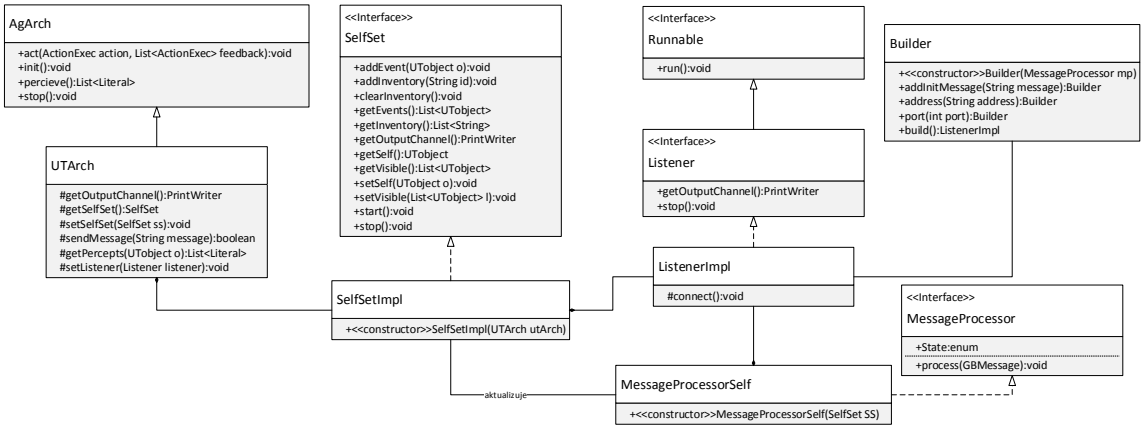
- `turn_to(X,Y,Z)` - Otoč sa na pozíciu zadanú súradnicami v parametri.
- `jump` - Vyskoč.
- `jump(DD,DELAY,FORCE)` - Zložitejší výskok. Parametre odpovedajú správe pre GameBots. DD bude True, ak chceme, aby bot vykonal dvojitý skok. DELAY určí oneskorenie, s ktorým vyskočí druhý krát. FORCE je vektor, určujúci aký veľký bude skok.
- `connect` - Pripoj sa na server.
- `disconnect` - Odpoj sa od servera.
- `stop_mov` - Zastav pohyb.
- `move(X,Y,Z)` - Pohni sa na pozíciu X, Y, Z.
- `stop_shoot` - Zastav streľbu.
- `shoot(ID)` - Začni strieľať na vec, určenú identifikátorom ID.
- `cwp(W)` - Zmeň zbraň na W. Zbraň musí byť v inventáry.

## 4.11 SelfSet

Združuje v sebe informácie o botovi.

- Stav bota - počet životov, brnenia, adrenalínu.
- Objekty, ktoré vidí - navigačné body, zbrane, lekárničky, hráči.
- Udalosti - zasiahnutie, zodvihnutie veci.
- Inventár - zoznam zbraní, ktoré môže bot použiť.

Pre podrobnejšie informácie pozri GameBots protokol [5]. Tieto informácie prichádzajú z GameBots servera. Je ich treba prijímať a spracovávať, preto má set instanciu `Listener`u a `MessageProcessor`u. `ListenerImpl` sme si už popísali vyššie, spracovač správ sa však trochu odlišuje od toho z prostredia.




Obrázek 4.3: UML diagram agenta.

## 4.12 MessageProcessorSelf

Pracuje na podobnom princípe ako `MessageProcessorGlobal`, líši sa tým, aké správy ho zaujímajú. Ignoruje správy z handshaku, tie má agent k dispozícii z prostredia. Spracováva skupiny synchronizovaných správ uvedené správami BEG-END. Obsahujú SLF (stav bota), INV (veci, ktoré vidí), NAV (navigačné body, ktoré vidí), PLR (hráči, ktorých vidí). Pravidelne chodí aj správa UPD, kde sa aktualizuje pozícia bota rýchlejšie ako v správe SLF (správa SLF chodí každých 250 $ms$, UPD 50 $ms$). Ostatné správy sú asynchrónne a sú považované za udalosti.

`MessageProcessorSelf` spracováva správy: NAV, INV, SLF, UPD, PLR, SPW, AIN, DIE, FIN.

## 4.13 Vnútorné akcie

Tieto akcie neovplyvňujú prostredie. Sú použité na výpočty, ktoré chce agent vykonať v Jave. Akcie sú vykonávané naraz a v jednom úvahovom cykle. Užitočné vnútorné akcie sa nachádzajú v balíku `ias`.

- distance(X1,Y1,Z1,X2,Y2,Z2,D) - určí vzdialenosť medzi dvoma bodmi. Ak je D premenná, unifikuje sa s výsledkom. Ak je číslo, porovná sa s výsledkom a vráti pravdu, ak je vzdialenosť bodov menšia ako D.
- random_location(X,Y,Z) - unifikuje parametre s pozíciou náhodného navigačného bodu na mape. Tieto body získa z `UnrealEnvironment`.
- jump(X1,Y1,Z1,X2,Y2,Z2,H,V) - pracuje zo skokmi. Zadáme mu pozíciu, z ktorej chceme skočiť a pozíciu, na ktorú chceme skočiť. Vypočíta z nich horizontálnu a vertikálnu vzdialenosť skoku a unifikuje ju s premennými H (horizontálna) a V (vertikálna).

## 4.14 Nástroje

V balíčku `utils` sú užitočné nástroje pri práci s prostredím.

### 4.14.1 GameBotsActions

Obsahuje všetky akcie, ktoré GameBots poskytuje. V parametri zadávame atribúty správy. Pri zadaní hodnoty `null` sa atribút k správe nepridá.

Napríklad na príkaz `jump(String doubleJump, String delay, String force)` bot vyskočí. Ak zavoláme akciu `GameBotsActions.jump(null, null, null)` dostaneme správu "JUMP". Pri zadaní prvého parametra na dvojitý skok, `GameBotsActions.jump(''True'', null, null)`, dostaneme správu "JUMP {DoubleJump True}"a bot vykoná dvojitý výskok.

### 4.14.2 GameBotsUtils

Nástroje, ktoré súvisia s GameBots serverom.

- `sendMessage(PrintWriter out, String message)` - Pošle na výstupný kanál správu z parametra. Správy pre GameBots musia byť ukončené dvojicou znakov `\r\n`. Metóda ich pridá (správa v parametri nemusí obsahovať ukončovacie znaky) a správu okamžite odošle.

### 4.14.3 JasonUtils

Nástroje, ktoré súvisia s Jasonom.

- `convert(String str)` - Syntax Jasonu kladie obmedzenia na formu literálov. Premenné začínajú veľkým písmenom, atómy malým, bodka oddeľuje názvy balíkov vo vnútorných akciách. Metóda odstáni špeciálne znaky okrem '_' a '∼' a tiež prekonvertuje písmená na malé.
- `equals(String s1, String s2)` - Najprv reťazce prekonvertuje a potom určí, či sú zhodné. Ak je jeden z parametrov `null` vráti nepravdu.
- `percept(String name, String ...args)` - Metóda vytvorí literál z parametrov. Názov literálu je prvým parametrom ostatné, nepovinné, sú jeho termy. Parametre pred spracovaním najprv prekonvertuje metódou, uvedenou vyššie.
- `makePathPercept(List<Node> path)` - Z cesty, nájdenej pathfinderom, vytvorí zoznam literálov pre agenta, v tvare `path(ID,N,X,Y,Z,F)[source(pathfinder),expires(clearpath)]`.

### 4.14.4 NavPointUtils

Slúži na prácu s navigačnými bodmi.

- `findClosest(Location source)` - Nájde najbližší navigačný bod k zadanej pozícii.
- `findClosest(String id)` - Nájde najbližší bod k bodu s daným identifikátorom.
- `getNavPointById(String id)` - Nájde navigačný bod so zadaným identifikátorom. Použije metódu pre porovnanie `JasonUtils.equals(String s1, String s2)`.
- `hasItem(UTobject np)` - Zistí, či sa na danom navigačnom bode objavuje vec.
- `getAdrenalineNP()` - Vráti zoznam navigačných bodov s adrenalínom.
- `getAmmoNP()` - Vráti zoznam navigačných bodov s nábojmi.

- `getDamageNP()` - Vráti zoznam navigačných bodov s dvojitým damagom.
- `getHealthNP()` - Vráti zoznam navigačných bodov s lekárničkami.
- `getShieldNP()` - Vráti zoznam navigačných bodov so štítmy.
- `getWeaponNP()` - Vráti zoznam navigačných bodov so zbraňami.

### 4.14.5 ItemUtils

Nástroje na prácu s vecami.

- `findItemById(String id)` - Prehľadá veci z prostredia a vráti tú, ktorá má zhodný identifikátor s parametrom. Na porovnanie sa použije `JasonUtils.equals(s1,s2)`.
- `findItemsByType(String type)` - Vráti zoznam vecí z prostredia, ktoré odpovedajú typu veci, zadanom v parametri.
- `isAdrenaline(String item)` - Určí, či je vec adrenalín.
- `isAmmo(String item)` - Určí, či je vec náboj.
- `isDamagePack(String item)` - Určí, či je vec dvojnásobný damage.
- `isHealthPack(String item)` - Určí, či je vec lekárnička.
- `isShield(String item)` - Určí, či je vec brnenie.
- `isWeapon(String item)` - Určí, či je vec zbraň.
- `simpleName(String name)` - Vytvorí jednoduché meno z reťazca zadaného v parametri.
- `getAdrenaline()` - Zoznam adrenalínov na mape.
- `getAmmo()` - Zoznam nábojov na mape.
- `getDamage()` - Zoznam dvojitého damagu na mape.
- `getHealth()` - Zoznam lekárničiek na mape.
- `getShield()` - Zoznam štítov na mape.
- `getWeapons()` - Zoznam zbraní na mape.

# Kapitola 5

# Hľadanie cesty

Hľadanie cesty v grafe hrá dôležitú úlohu pri pohybe agenta. Každá mapa obsahuje sieť navigačných bodov. Sú navzájom prepojené. Cesta, ktorú nájdeme nemusí byť najkratšia ani najlepšia. Hlavne aby bola nájdená čo možno najrýchlejšie, aby bot nestál na mieste.

Pri hľadaní cesty sa najčastejšie používa A* algoritmus. Dokáže rýchlo nájsť najkratšiu cestu. Pomáha si pri tom heuristikou. Dobrá heuristika dokáže celý proces urýchliť. Je daná funkciou $f(n) = g(n)+h(n)$. $g(n)$ reprezentuje cestu zo začiatku do vrchola $n$. $h(n)$ je odhadovaná vzdialenosť do cieľa. V našom prípade to bude euklidovská vzdialenosť. Pseudokód algoritmu z [14]:

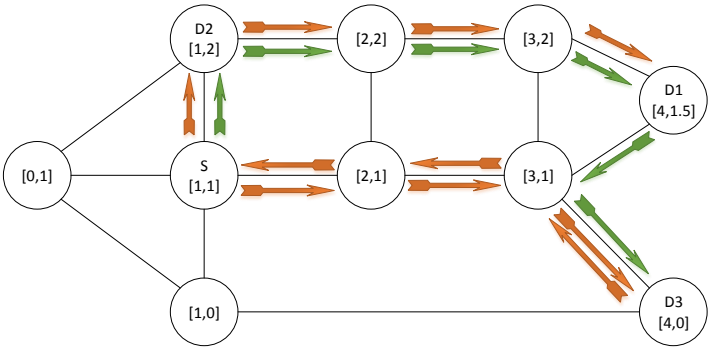
```
OPEN = priority queue containing START
CLOSED = empty set
while lowest rank in OPEN is not the GOAL:
  current = remove lowest rank item from OPEN
  add current to CLOSED
  for neighbors of current:
    cost = g(current) + movementcost(current, neighbor)
    if neighbor in OPEN and cost less than g(neighbor):
      remove neighbor from OPEN, because new path is better
    if neighbor in CLOSED and cost less than g(neighbor):
      remove neighbor from CLOSED
    if neighbor not in OPEN and neighbor not in CLOSED:
      set g(neighbor) to cost
      add neighbor to OPEN
      set priority queue rank to g(neighbor) + h(neighbor)
      set neighbors parent to current

reconstruct reverse path from goal to start
by following parent pointers
```

## 5.1 Viacero destinácií

Predpokladajme, že máme viacero destinácií, na ktoré sa chceme dostať. Mohli by sme z nich vybrať náhodne jednu, dostať sa do nej a ísť na ďalšiu. Takýto pohyb by však nebol príliš efektívny (obrázok 5.1 červená cesta). Alternatívnym riešením by bolo, nájsť najbližšiu z

nich a vybrať sa tam. Z nej si nájdeme ďalšiu najbližšiu a tento proces opakujeme dovtedy, pokiaľ sa nedostaneme do všetkých cieľov. Na obrázku 5.1 je znázornená cesta s náhodne vybranými bodmi červenou. Najprv sa vyberieme do bodu D3, potom do D2 a skončíme na D1. Ako vidíme, existuje aj lepšia cesta. Je vyznačená zelenou a nájdeme ju tak, že si zo zoznamu destinácií vždy vyberáme tú najbližšiu.




Obrázek 5.1: Cesta s viacerými destináciami.

## 5.2 Rôzne body

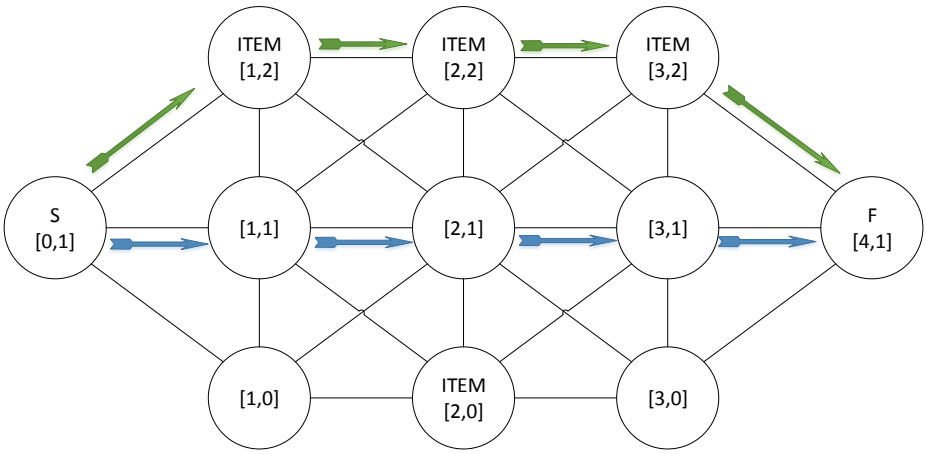
V navigačných bodoch sú rozmiestnené veci. Sú nimi zbrane, náboje, lekárničky a adrenalín. Pri prechode mapou botovi nemusí vadiť, ak sa od svojej optimálnej trasy odchýli, aby si vzal užitočnú vec. Je to dokonca žiadané, keďže zbieranie vecí poskytuje výhodu nad protivníkom.

Mohli by sme akumulovať odmenu za zodvihnuté veci, ako sa bot pohybuje po jednotlivých trasách. Potom by sme vybrali tú, s najväčším ziskom vzdialenosť/odmena. To si však vyžaduje nájdenie viacerých (alebo všetkých) ciest a ich porovnanie. My však chceme nájsť cestu čo najrýchlejšie.

Iný prístup by mohol byť manipulácia so vzdialenosťou bodov. Keď v algoritme A* určujeme cenu do suseda $g(n)$, spočítame ju ako precestovanú vzdialenosť plus vzdialenosť medzi týmito dvoma bodmi $cost = g(current) + movementcost(current, neighbor)$. Ak by sa na susedovi nachádzala nejaká vec, mohli by sme ju skrátiť a ovplyvniť tak cestu. Napríklad skrátením o koeficient v rozmedzí $< 0,1)$, by sa zdala cesta bližšie k štartu a tým pádom lepšia. Cesta už nebude najkratšia, ale za toto predĺženie budeme odmenený vecami, ktoré na nej získame. Navyše, ak zvolíme vhodné koeficienty, tak sa od trasy, ktorú by normálne našiel algoritmus, príliš nevzdialime.

Na obrázku 5.2 je modrou vyznačená cesta bez ohodnotenia vecí. Jej dĺžka je 4. Zelenú cestu nájde algoritmus, ak ohodnotenie veci bude $0,5$. Keby boli body prázdne, vzdialenosť by bola 4.828, teda dlhšia ako modrá. No pri koeficiente $0,5$ sa skráti na 3.121, čo je pre bota výhodnejšie.

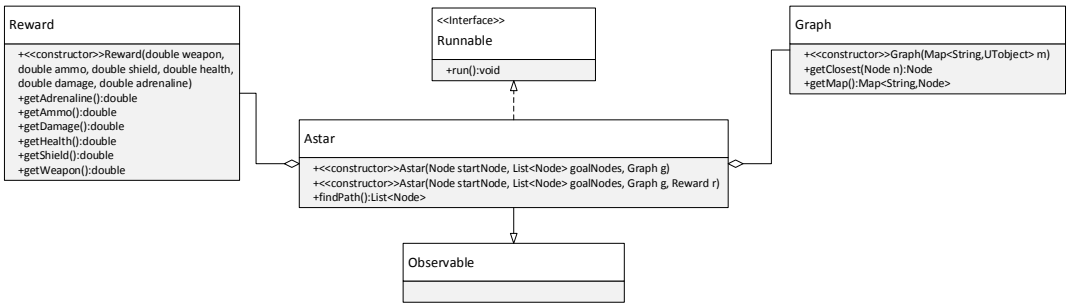
Zmeny vo vzdialenostiach majú aj negatívny efekt. Môžu predĺžiť dobu výpočtu tým, že spôsobujú expandovanie viacerých uzlov. Preto je vhodné voliť koeficienty opatrne.




Obrázek 5.2: Cesta so zbieraním vecí.

## 5.3 Astar

Trieda `Astar` implementuje vyššie zmienený algoritmus. Používa sa na hľadanie cesty v sieti navigačných bodov Unreal Tournamentu. Uml diagram na obrázku 5.3.




Obrázek 5.3: Uml triedy `Astar`.

Algoritmus inicializujeme zadaním počiatočného bodu, zoznamu cieľov a grafom všetkých bodov mapy. Ak chceme, aby algoritmus bral do úvahy veci na navigačných bodoch, dáme mu triedu `Reward`. Tá obsahuje koeficienty pre rôzne objekty mapy.

Trieda implementuje rozhranie `Runnable` a beží na samostatnom vlákne. Po spustení vlákna sa okamžite začne výpočet funkciou `findPath()`. Táto metóda postupne hľadá cestu pre všetky ciele. Používa algoritmus A* implementovaný v metóde `execute(ANode start, ANode goal)`. Vyberie najbližší bod, nájde k nemu cestu a pripojí ju k výsledku.

Algoritmus:

```
START = start_node
path = START

while goals not empty
 CLOSEST_NODE = closest goal from goals
 p = A* find path from START to CLOSEST_NODE
 path = join path and p
 notify observers with found path
 remove goal from goals
 START = CLOSEST_NODE

return path
```

Aby klienti vedeli, kedy algoritmus skončí, rozširuje triedu `Observable`. Teraz sa môžu klienti zaregistrovať ako pozorovatelia a budú upozornený, keď A* nájde cestu do cieľa. Keďže užívateľ môže zadať viacero cieľov, výpočet cesty môže trvať dlhšie a bot sa chce pohybovať okamžite. Algoritmus preto upozorní pozorovateľov vždy, keď nájde cestu do jednej z destinácií.

# Kapitola 6

# Programovanie agenta

V tejto kapitole sa pozrieme na to, ako si vytvoriť vlastného bota. Ako herný mód zvolíme deathmatch. Jeho cieľom je získať určité skóre. Kto ho dosiahne prvý, vyhráva. Za zabitie súpera sa bod pridá, za smrť odoberie. Mapy pre tento mód sú charakteristické rovnomerným rozmiestnením zbraní, nábojov, lekárničiek a adrenalínu. Zvyčajene majú aj balíček dvojitého damagu. Typickými zbraňami sú guľomet, brokovnica, raketomet, šoková a ostreľovacia puška, organická a linkovacia zbraň. Mapy nie sú príliš rozsiahle, aby hráči na seba často narážali. Neobsahujú vozidlá ani továrne.

Pre tento herný mód chceme vytvoriť agenta. Nazvime ho Arnold. Arnold chce vyhrať každý zápas, na ktorý sa prihlási. Na to, aby mohol hru hrať, musí vnímať svoje okolie, pohybovať sa po mape a útočiť na protivníkov. No najprv sa musí pripojiť na server.

### 6.0.1 Pripojenie na server

`UnrealEnvironment` pridáva agentom vnem `match(ID)`, ak zápas prebieha a pri ukonční zápasu agenti dostanú vnem `match_end`. Môžeme na túto informáciu reagovať pripojením alebo odpojením od servera. Na nadviazanie spojenia použijeme akciu `connect`. Tá spustí `SelfSet`, ktorý sa pripojí na adresu a port servera, ktoré získa z prostredia. Akciou `disconnect` sa agent odpojí a uzavrie spojenie.

Arnold sa chce pripojiť na každú hru, ktorá prebieha. Vytvoríme mu plán na pripojenie k serveru:

```
+match(_):not connected<-
        +connected;
        game_in_progress;
        connect.

+match_end:connected<-
        -connected;
        disconnect.
```

Arnold najprv zistí, či je hra spustená, akciou prostredia `game_in_progress` a potom sa pripojí. Ak zápas skončil, odpojí sa a uzavrie spojenie.

Teraz sa už vieme pripojiť a odpojiť od servera. Potrebujeme však vnemy o prostredí, aby sme vedeli, čo sa okolo nás deje.

### 6.0.2 Vnemy

Trieda `UTArch` agentom poskytuje základné vnemy. Ich zoznam v kapitole 4.10. Môžeme však pridať aj vlastné.

Arnold by chcel vedieť aj to, koľko má životov a adrenalínu. Na to budeme musieť upraviť funkciu pre aktualizáciu vnemov agenta. Vytvoríme si triedu `SampleUTArch`, ktorá dedí od triedy pre architektúru `UTArch`. Na pridanie vlastného vnemu prepíšeme metódu `getPercepts(UTobject o)` aby vracala zoznam aj s informáciou o Arnoldovom živote a adrenalíne. Tieto veci sú obsiahnuté v správe SLF.

```
@Override
protected List<Literal> getPercepts(UTobject o){
 List<Literal> result = super.getPercepts(o);
 switch(o.getType()){
 case SLF:
  result.add(JasonUtils.percept("health", o.get("Health")));
  result.add(JasonUtils.percept("adrenaline", o.get("Adrenaline"))
      );
 break;
 default:
 break;
 }
return result;
}
```

Najprv zavoláme pôvodnú funkciu, ktorá nám vráti zoznam vnemov z objektu o. Do tohoto zoznamu pridáme vnemy, ktoré chceme, v našom prípade Arnoldove životy a adrenalín. O ich pridanie do agentovej báze znalostí sa postará funkcia `perceive()`.

Teraz má agent informácie o svojom zdraví v báze znalostí.

```
+health(H): H < 25<-
    !find_healthpack.

+adrenaline(H): H == 100<-
    !do_combo.
```

Na nové informácie môže reagovať. Ak bude mať Arnold menej ako 25 života, dá si za cieľ nájsť lekárničku a svoje zranenia vyliečiť. Ak bude mať 100 adrenalínu, môže zapnúť kombo, aby ho využil.

### 6.0.3 Pohyb na mape

Arnold je pripojený k serveru, má informácie o prostredí, ale stojí na mieste. Nevie ako a kde sa má pohnúť. Najjeduchšie bude ísť na miesto prvého navigačného bodu, ktorý uvidí:

```
+nav_point(ID):location(ID,X,Y,Z)<-
    move(X,Y,Z).
```

Akcia `move(X,Y,Z)` je definovaná v základnom sete akcií v `UTArch`. Presunie bota na pozíciu zadanú v parametroch. GameBots ponúka rozšírenú akciu `move(firstLocation, secondLocation,`

`focusTarget, focusLocation)`. Po dosiahnutí prvej lokácie sa bot pohne okamžite pohne na druhú. Arnold by sa chcel, po navštívení navigačného bodu, vrátiť na pozíciu, z ktorej začínal. Namiesto dvojnásobného volania `move(X,Y,Z)`, využije príkaz na pohyb s dvoma destináciami. Keďže `UTArch` takúto akciu neponúka, musíme si ju vytvoriť prepísaním metódy `act(...)`. Nová akcia bude mať tvar `move(X1,Y1,Z1,X2,Y2,Z2)`.

```
@Override
public void act(ActionExec action, List<ActionExec> feedback){
 boolean result = false;
 Structure act = action.getActionTerm();
 switch(act.getFunctor()){
 case "move":
  if(act.getArity() == 6){
   Location loc1 = new Location(act.getTerm(0), act.getTerm(1),
       act.getTerm(2));
   Location loc2 = new Location(act.getTerm(3), act.getTerm(4),
       act.getTerm(5));
   result = sendMessage(GameBotsActions.move(loc1.toString(), loc2
       .toString(), null, null));
  }else{
   super.act(action, feedback);
   return;
  }
  break;
  default:
   super.act(action, feedback);
   return;
  }
  action.setResult(result);
  feedback.add(action);
}
```

Z parametrov vytvoríme lokácie a pošleme ich príkazom na pohyb na server. Ak teraz zavoláme akciu na pohyb zo 6-timi parametrami, bot sa najprv rozbehne do prvej, potom do druhej.

```
+nav_point(ID):location(ID,X,Y,Z)<-
    ?my_id(ME);
    ?location(ME,X1,Y1,Z1);
    move(X,Y,Z,X1,Y1,Z1).
```

Hoci sa už Arnold vie prejsť po mape, jeho pohyb zatiaľ nie je veľmi užitočný. Chceli by sme, aby sa vedel dostať na každý bod mapy. Takúto cestu si bude musieť naplánovať. `UnrealEnvironment` má akciu `get_path(X,Y,Z,[dest(ID,Xd,Yd,Zd)])`. Po zadaní začínajúcej pozície a zoznamu destinácií, dostane agent do báze znalostí cestu vo forme série navigačných bodov. Majú tvar `path(ID,N,X,Y,Z,F)[source(pathfinder),expires(clearpath)]`. Bod je zadaný svojím identifikátorom a pozíciou. N je poradové číslo cesty začínajúce 0. F udáva typ prechodu medzi navigačnými bodmi.

Arnold sa chce pohybovať hneď po spawnutí na náhodnú pozíciu na mape.

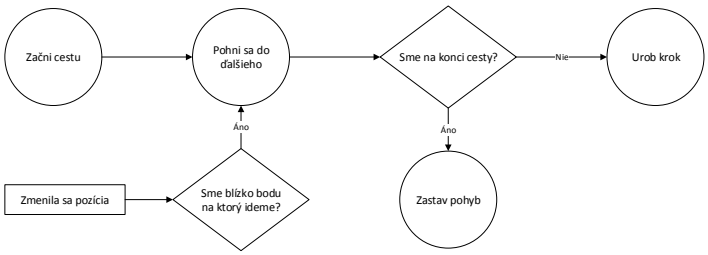
```
+spawned<-
    ias.random_location(X,Y,Z);
    +dest(rand,X,Y,Z);
   .findall(dest(ID,X,Y,Z),dest(ID,X,Y,Z),L);
   ?my_id(ME);
   ?location(ME,Xme,Yme,Zme);
   get_path(Xme,Yme,Zme,L).
```

Využijeme vnútornú akciu z balíčka `ias`, ktorá unifikuje premenné s náhodnou pozíciou na mape. Pridáme si destináciu do báze znalostí. Zistíme svoju pozíciu, nájdeme všetky destinácie, ktoré máme a akciou `get_path` si vyžiadame od prostredia cestu. Prostredie ju nájde a agent ju dostane vo forme série literálov.

Keď už máme cestu, môžeme sa začať pohybovať. Literálom `path_num` si označíme poradové číslo bodu na ceste, na ktorý ideme.

```
+path(ID,0,X,Y,Z,F)<-
          +path_num(1)[expires(clearpath)];
          !move_to_next.
```

Teraz musíme zaistiť, aby sa bot vedel po ceste pohybovať. Diagram pohybu znázornený na obrázku 6.1. Prechody medzi bodmi na mape sú rôzne. Medzi najdôležitejšie patria: jednoduchý pohyb, skoky a výťahy. V Unreal Tournamente rozlišujeme dva typy skokov, jednoduchý a dvojitý. Skoky pri normálnej gravitácii sú znázornené na diagrame 6.2. Výťahy sú pohybujúce sa objekty. Keď na ne hráč vkročí, začnú sa pohybovať, dokým nedorazia na výstupný bod. Tam sa na chvíľu zastavia a potom sa vrátia na pôvodnú pozíciu 6.3. Pri každej zmene pohybu alebo bodu cesty budeme kontrolovať, či už nie sme na pozícii, na ktorú ideme. Ak áno pohneme sa na ďalšiu, až dokým nie sme na konci cesty. Kód v prílohe E.

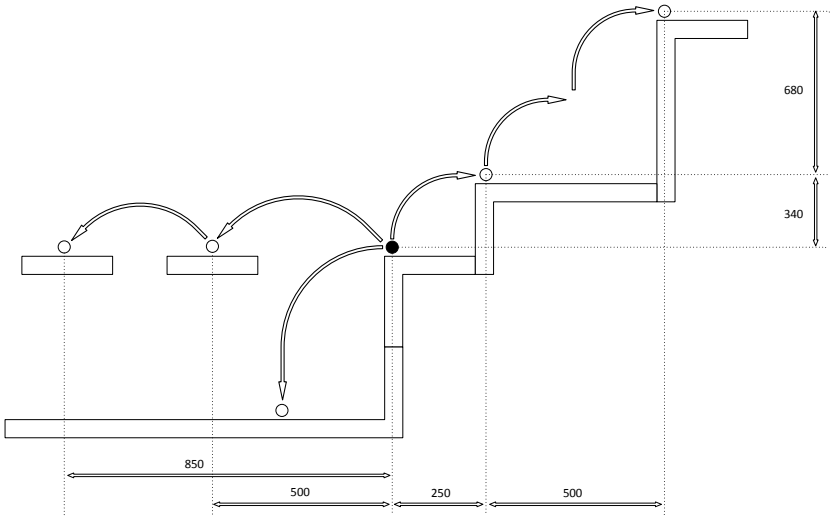



Obrázek 6.1: Diagram pohybu.

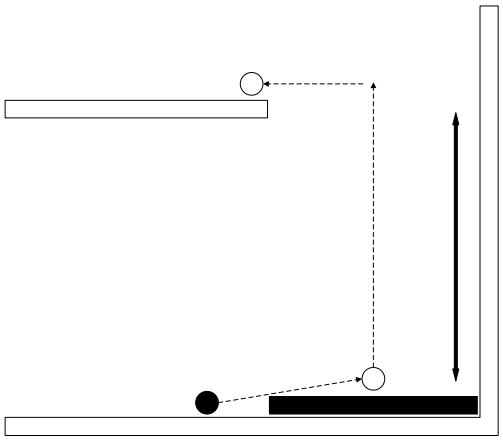
Kroky `step(X,Y,Z,F)` budú predstavovať prechod z jedného bodu na druhý. F je typ prechodu definovaný v [9]. Vzniká ich kombináciou. Ak sa má po nich bot vedieť pohybovať, bude musieť zvládnuť aspoň tie najzákladnejšie.

- 256 je `forced` prechod. Na ten sa bot nedokáže dostať, ak nemá povolené špeciálne pohyby, ako napríklad lietanie (povolenia sú v súbore `GameBots2004.ini`). Ak na takýto prechod narazí, zastaví svoj pohyb a pohľadá inú cestu.
- 32 je špeciálny prechod, zvyčajne výťah. Keď naň bot vstúpi, musí počkať, kým ho vyvezie hore a potom vystúpiť.

- 16 sú dvere, cez ktoré jednoducho prejde.
- 8 sú skoky. Znázornené na diagrame 6.2.
- 1 je jednoduchý pohyb.




Obrázek 6.2: Skoky.



Obrázek 6.3: Výťahy.

Použili sme niekoľko pravidiel. Tie zjednodušujú kód a združujú zložitejšie volania. `my_position` zistí pozíciu bota, `near` určí, či sme blízko bodu, na ktorý sa chceme dostať, `next` vráti ďalší bod na ceste a `on_mover` určí, či sme na výťahu.

```
my_position(X,Y,Z):- my_id(ME) & location(ME,X,Y,Z).
near:- my_position(Xme,Yme,Zme) & path_num(N) & path(_,N,X,Y,Z,_)
   & ias.distance(X,Y,Z,Xme,Yme,Zme,100).
next(N+1,X,Y,Z,F):- path_num(N) & path(_,N+1,X,Y,Z,F).
on_mover:- path_num(N) & path(ID,N,_,_,_,_) & mover(ID).
```

Teraz je už Arnold schopný pohybovať sa po mape. Ešte zostáva jedna drobnosť, a to vysporiadať sa s anotáciou `clearpath`. Ako sme spomenuli v úvode od Jasonu, anotácie majú informatívny charakter. My ju však chceme použiť k revízii znalostí. Keď sa pridá do báze znalostí, odstránime všetky literály, ktoré majú anotáciu `expires(clearpath)`. K takýmto akciám slúži funkcia `brf(Literal beliefToAdd, Literal beliefToDel, Intention i)` z triedy Jasonu pre agenta `Agent`. Ak sa do báze znalostí pridal `clearpath`, prejdeme všetky znalosti a vyberieme tie, ktoré odpovedajú kritériu. Následne ich odstránime.

```
public class SampleUTAgent extends Agent{
 @Override
 public List<Literal>[] brf(Literal beliefToAdd, Literal
    beliefToDel, Intention i) throws RevisionFailedException{
  List<Literal>[] result = super.brf(beliefToAdd, beliefToDel, i);
  try{
   if(beliefToAdd != null && "clearpath".equals(beliefToAdd.
      getFunctor())){
    List<Literal> toBeDeleted = new LinkedList<Literal>();
    Term clrAnnot = ASSyntax.parseTerm("expires(clearpath)");
    for(Literal b : getBB()){
     if(b.hasAnnot(clrAnnot)){
      toBeDeleted.add(b);
     }
    }
    for(Literal l : toBeDeleted){
     delBel(l);
    }
   }
  }catch(ParseException e){}
  return result;
 }
}
```

Báza znalostí je teraz zbavená informácii o pohybe po jeho ukončení a vieme sa pohybovať po mape. Čo ak Arnold na potulkách mapou narazí na nepriateľa?

### 6.0.4 Útočenie

Ak bot zazrie hráča, dostane túto informáciu v synchronizovaných správach. `MessageProcessor` ju spracuje a aktualizuje `SelfSet`. Funkcia `perceive()` pridá do agentovej báze znalostí vnem `player(ID)`. Keď máme túto infomáciu, môžeme na ňu reagovať.

```
+player(P):not target(_)<-
    +target(P);
    shoot(P).

-player(P):target(P)<-
    stop_shoot;
    -target(P).
```

Akcia `shoot(ID)` slúži na strieľanie. Stačí zadať identifikátor hráča, na ktorého chceme strieľať a Unreal Tournament zabezpečí mierenie sám. Ak hráč zmizne zo zorného poľa ukončíme streľbu.

# Kapitola 7

# Testovanie

Aby sa dalo rozhranie použiť v praxi, malo by spĺňať isté podmienky. Agenti sa musia rozhodovať a ovládať svojho bota v reálnom čase. To znamená, že úvahový cyklus musí prebiehať čo najrýchlejšie, rovnako ako algoritmus hľadania cesty, spracovanie správ či vkladanie vnemov z prostredia.

Implementáciu otestujeme na počítači:

- notebook MSI MS-16GA
- procesor Intel Core i5-3230M, 2.60 GHz
- pamäť 4 GB
- systém Windows 8.1, 64-bit
- Java verzie 1.8.0_25
- GameBots server beží na samostatnom počítači. Rýchlosť spojenia 2 ms.

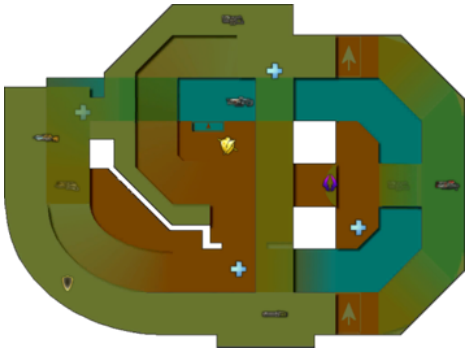
Budeme skúmať dobu úvahového cyklu, aktualizovanie vnemov, vykonanie akcie a spracovanie správy. Ďalej zistíme, ako na ne vplýva počet agentov a veľkosť mapy. Experimenty vykonáme s dvoma, piatimi, desiatimi a pätnástimi agentami na troch rôznych mapách pre deathmatch. 15 hráčov zvolíme ako maximum, pretože mapy sú navrhnuté najviac pre 10 až 16 hráčov. Vyberieme si tri mapy pre testovanie. Budú to DM-Deck17, DM-DE-Grendelkeep a DM-Oceanic. Po dobu desiatich minút budeme zbierať štatistiky. Vykonáme päť meraní a vyhodnotíme výsledky.

Otestujeme tiež algoritmus na hľadanie cesty. Na týchto troch mapách vykonáme 100 000 náhodných prechádzok. Urobíme 5 meraní, pričom počet destinácií jednej cesty sa bude pohybovať od 1 do 5. Otestujeme aj vplyv ohodnotenia navigačného bodu podľa veci, ktorá na ňom leží a ako vplýva na dobu nájdenia cesty.

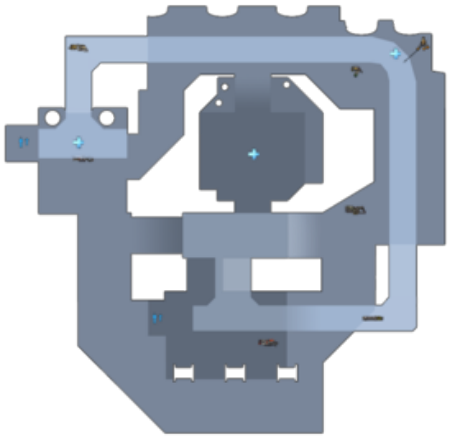
Mapy:

- DM-Deck17 - mapa má 13 zbraní, 34 nábojov, 40 vecí a 276 navigačných bodov.
- DM-DE-Grendelkeep - mapa má 7 zbraní, 28 nábojov, 26 vecí a 179 navigačných bodov.
- DM-Oceanic - mapa má 6 zbraní, 17 nábojov, 3 veci a 98 navigačných bodov.


(a) Mapa DM-Deck17 [16].



(b) Mapa DM-DE-Grendelkeep [17].



(c) Mapa DM-Oceanic [13].

Obrázek 7.1: Plány máp.

### 7.0.5 Úvahový cyklus

Každý úvahový cyklus agent aktualizuje svoje vnemy z prostredia a vykoná jednu akciu zo svojich zámerov. Tento postup sa neustále opakuje. Aby sa agent rozhodoval v reálnom čase, musí byť čo najrýchlejší.

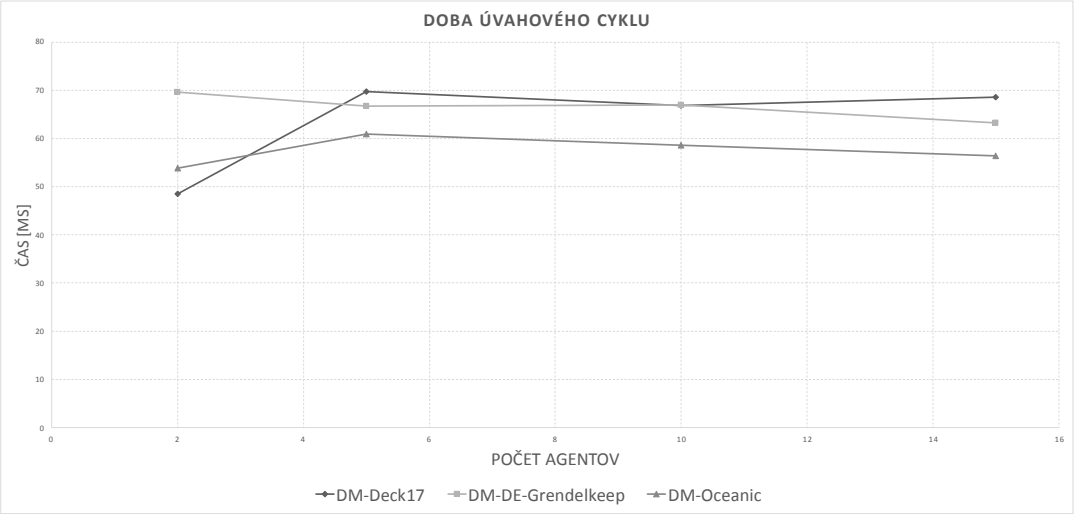
Doba cyklu závisí od náročnosti výpočtov jeho krokov. Toto zahŕňa aktualizovanie vnemov, generovanie a vybratie udalostí, nájdenie plánu, vytvorenie zámeru a vykonanie akcie. Tiež bude závisieť od počtu agentov, ktorý sú spustený na jednom počíťači.

Túto dobu budeme merať a zistíme, aký rýchly je úvahový cyklus v našej implementácii a ako závisí od počtu agentov či mapy. Namerané hodnoty sú v tabuľke 7.1.

Z grafu 7.2 vidno, že počet agentov ani typ mapy nemá na doby cyklu výraznejši vplyv a drží sa okolo priemernej hodnoty 65 ms. Na to, aké rýchle je vytvorenie vnemov a vykonanie akcie sa pozrieme v nasledujúcich sekciách.

| agentov | DM-Deck17 | DM-DE-Grendelkeep | DM-Oceanic |
|---|---|---|---|
| 2 | 48,5 | 69,6 | 53,8 |
| 5 | 69,7 | 66,7 | 60,9 |
| 10 | 66,9 | 66,9 | 58,6 |
| 15 | 68,5 | 63,2 | 56,3 |

Tabulka 7.1: Doba úvahového cyklu v milisekundách.




Obrázek 7.2: Graf závislosti doby úvahového cyklu od počtu agentov a mapy.

### 7.0.6 Aktualizácia vnemov

Agenti si aktualizujú svoju bázu znalostí o vnemy z prostredia každý úvahový cyklus. Toto zahŕňa získanie informácií zo `SelfSetu` a vytvorenie literálov. Čas, potrebný na tieto akcie, dostaneme odmeraním času stráveným v tejto metóde, delené počtom jej volaní.

$$t_{percieve} = \frac{SPENT\_TIME}{INV}$$

Vytvorenie vnemov závisí od toho, koľko informácií obsahuje `SelfSet`. Sú to informácie o botovi, viditeľných objektoch a udalostiach, ktoré sa okolo neho stali. Čím ich bude viac, tým dlhšie bude trvať vytvorenie vnemov. Doba tiež závisí od počtu vnemov z jednej veci.

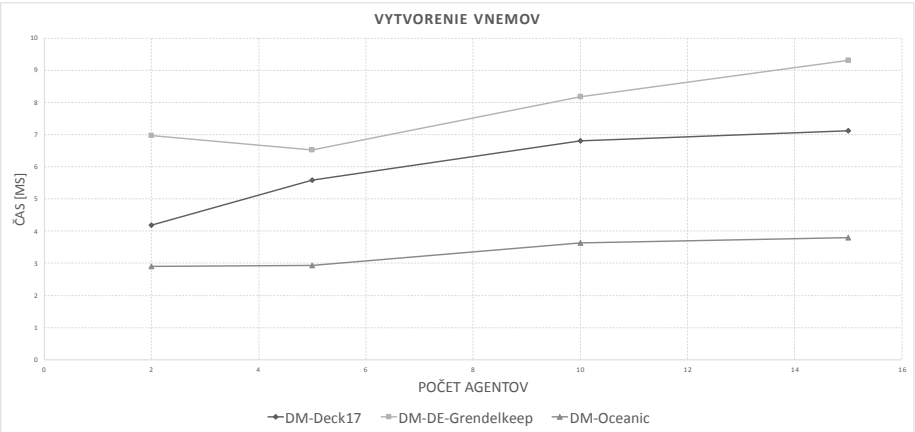
Zistíme, ako na ne vplýva počet agentov a typ mapy. Namerané hodnoty sú v tabuľke 7.2, graf hodnôt je znázornený na obrázku 7.3.

Čas vytvorenia vnemov sme skúmali na troch rôznych mapách. Najviac času zabralo na mape Grendelkeep. Z testovaných máp má priemerný počet vecí a navigačných bodov. Je však tvorená tromi otvorenými priestanstvami. To znamená, že agenti majú väčšie zorné pole a teda vnímajú viac vecí a aj na seba častejšie narážajú. Ostatné dve mapy sú tvorené najmä uzavretými chodbami, hoci najväčšia mapa, Deck17, má v strede veľkú miestnosť. Preto sú vnemy z nej spracovávané dlhšie ako v Oceanic.

Čas sa tiež zvyšuje s počtom agentov. Tí ovplyvňujú hru svojou činnosťou a spôsobujú, že agenti vnímajú viac informácií o hráčoch a tiež spôsobujú viac udalostí. Doba potrebná na vytvorenie sa zvyšuje približne o 0,1 ms na jedného hráča. Celkový čas vytvorenia vnemov tvorí približne 10 % úvahového cyklu.

| agentov | DM-Deck17 | DM-DE-Grendelkeep | DM-Oceanic |
|---|---|---|---|
| 2 | 4,2 | 7,0 | 2,9 |
| 5 | 5,6 | 6,5 | 2,9 |
| 10 | 6,8 | 8,2 | 3,6 |
| 15 | 7,1 | 9,3 | 3,8 |

Tabulka 7.2: Doba vytvorenia vnemov v milisekundách.




Obrázek 7.3: Graf závislosti doby vytvorenia vnemov od počtu agentov a mapy.

### 7.0.7 Vykonávanie akcií

Počas hrania, agenti posielajú príkazy na server a ovládajú nimi svojho bota. Tieto akcie sa musia vykonať rýchlo, lebo okolie okolo bota sa dynamicky mení a o chvíľu už nemusia mať zmysel. Odmeráme priemerný čas strávený v metóde `act(...)`, aby sme zistili, ako dlho trvá vykonanie jedného príkazu.

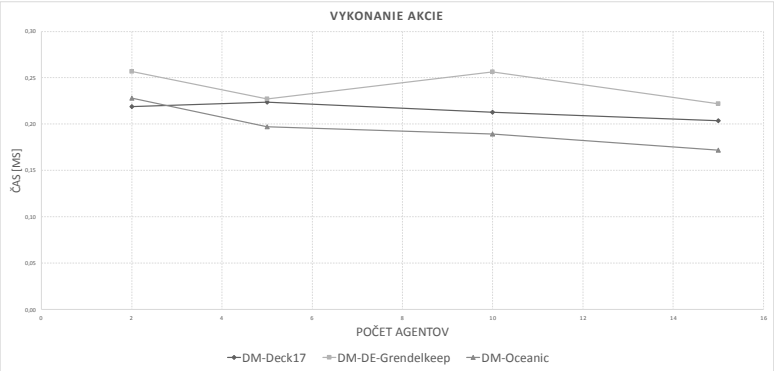
Vykonanie akcie v podstate zahŕňa len poslanie textového príkazu na GameBots server. Očakávame teda, že táto doba bude konštantná.

Namerané hodnoty sú v tabuľke 7.3, graf hodnôt je znázornený na obrázku 7.4a.

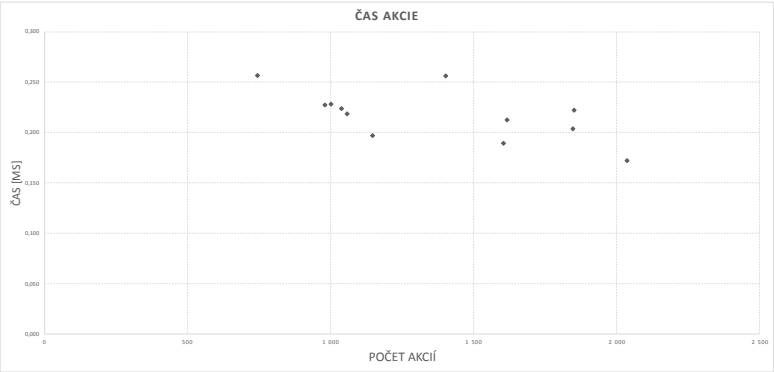
Z experimentov vyplýva, že priemerný čas jednej akcie je 0,22 ms a nezávisí od mapy ani počtu agentov 7.4a.Obrázok 7.4b ukazuje, že nezávisí ani na počte vykonaných akcií.

| agentov | DM-Deck17 | DM-DE-Grendelkeep | DM-Oceanic |
|---|---|---|---|
| 2 | 0,22 | 0,26 | 0,23 |
| 5 | 0,22 | 0,23 | 0,20 |
| 10 | 0,21 | 0,26 | 0,19 |
| 15 | 0,20 | 0,22 | 0,17 |

Tabulka 7.3: Doba vykonania akcie v milisekundách.




(a) Graf závislosti doby vykonania akcie od počtu agentov a mapy.




(b) Graf závislosti doby vykonania akcie od počtu vykonaných akcií.

Obrázek 7.4: Čas vykonania akcií.

### 7.0.8 Spracovanie správ

Z prostredia GameBots prichádza množstvo správ a `MessageProcessor` ich musí všetky spracovať. Sú to informácie o stave prostredia okolo bota, ktoré nevyhnutne potrebuje na svoje rozhodovanie. Musíme ich spracovať čo najrýchlejšie.

Budeme merať iba spracovávanie správ v `SelfSete`. Ten ich prijíma viac a je dôležitejšie, aby mal agent informácie z neho včas. `GlobalSet` a jeho `MessageProcessorGlobal` spracovávajú iba správy z handshaku, ktorý sa posiela na začiatku hry a ani nie je potrebné, aby boli spracované tak rýchlo. Okrem toho sú si tieto dva procesory také podobné, že čas strávený spracovaním jednej správy je porovnateľný.

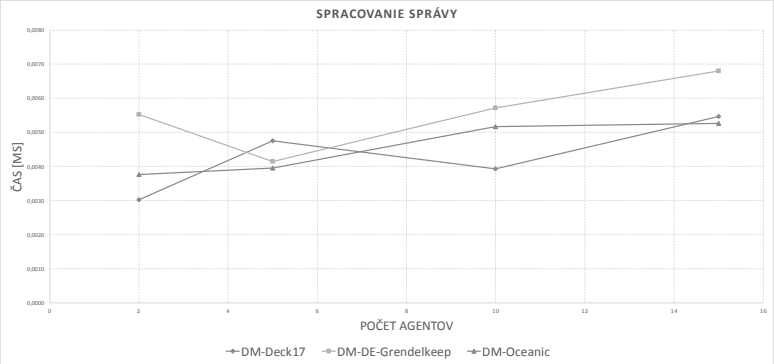
Čas spracovania správy by mal závisieť od počtu prijatých správ. Informácie o botovi sú posielané v pravidelných intervaloch. Ich počet bude ovplyvnený počtom vecí, ktoré vidí a počtom udalostí, ktoré sa stali. Niektoré správy ignorujeme (nespracovávame), takže čas bude závisieť aj od toho, koľko správ transformujeme na objekty pre agenta.

Graf 7.5a zobrazuje závisloť spracovania správy od mapy a počtu agentov. S vzrastajúcim počtom agentov v hre rastie aj doba spracovania jednej správy. Viac hráčov spôsobuje posielanie viacerých správ. Mapa na ňu výraznejšie nevplýva.

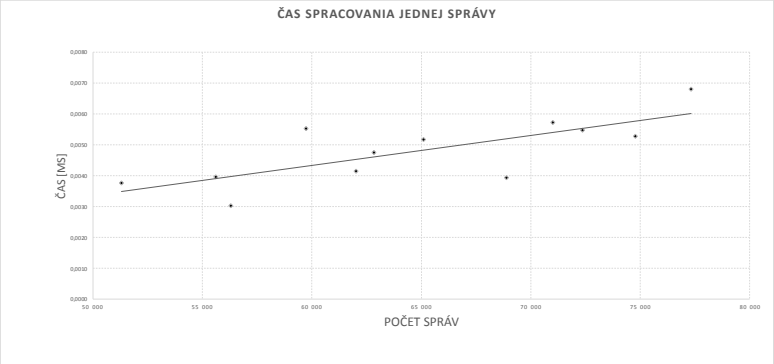
Graf 7.5b ukazuje lineárnu závislosť doby spracovania od počtu prijatých správ. Pre každých 5000 správ, čas spracovania jednej správy vzrastie o 0,5 nanosekundy.

| agentov | DM-Deck17 | DM-DE-Grendelkeep | DM-Oceanic |
|---|---|---|---|
| 2 | 0,00302 | 0,00553 | 0,00376 |
| 5 | 0,00475 | 0,00414 | 0,00395 |
| 10 | 0,00393 | 0,00572 | 0,00517 |
| 15 | 0,00547 | 0,00680 | 0,00527 |

Tabulka 7.4: Doba spracovania jednej správy v milisekundách.




(a) Graf závislosti doby spracovania správy od počtu agentov a mapy.




(b) Graf závislosti doby spracovania správy od počtu prijatých správ.

Obrázek 7.5: Čas spracovania správy.

### 7.0.9 Hľadanie cesty

Agenti si hľadajú cestu do cieľovej destinácie sieťou navigačných bodov. Používajú na to algoritus A*. Modifikovali sme ho, aby bral do úvahy aj veci na navigačných bodoch. Vie tiež nájsť cestu cez viacero destinácií.

Budeme ho testovať na všetkých troch mapách. Vykonáme 100 000 náhodných prechádzok s počtom destinácií od 1 do 5. Určíme rýchlosť nájdenia jednej cesty a jej závislosť od počtu destinácií, veľkosti mapy a koeficientov pre veci na mape.

Najprv sa pozrieme ako na rýchlosť vplýva mapa a počet destinácií. Otestujeme algoritmus na mapách Deck17, Grendelkeep a Oceanic. Líšia sa navzájom v počte navigačných bodov. Najväčšia je Deck17, očakávame teda, že čas, potrebný na nájdenie cesty bude najväčší. Výsledok experimentov je zobrazený v grafe 7.6a. Pozoruje exponenciálnu závislosť rýchlosti hľadania cesty od počtu navigačných bodov na mape.

Bot môže chcieť počas svojej cesty navštíviť viacero destinácií. Závislosť doby nájdenia cesty od ich počtu je zobrazená na obrázku 7.6b. Čas pre nájdenie cesty sa zvyšuje lineárne s počtom destinácií.

Nakoniec budeme skúmať závislosť ohodnotenia bodu na základe veci, ktorú obsahuje. Otestujeme algoritmus na jednej mape (DM-Deck17) s rôznymy hodnotami koeficientov. Najprv budú mať všetky veci ohodnotenie 0,5, potom 0,1, 0,01, 0 a nakoniec bude mať rôznu hodnotu (zbraň 0,01, náboj 0,2, lekárnička 0,1, arenalín 0,5, damage 0,01, brnenie

0,1). Namerané hodnoty sú v tabuľke 7.6 a zobrazené v grafe 7.6c. Ukazuje sa, že tieto koeficienty nemajú vplyv na nájdenie cesty ale závisia skôr od času matematických výpočtov pre vzdialenosť. Pre hodnoty 0,5 a 0 je asi dvakrát rýchlejšie, pretože ich násobenie s dĺžkou cesty je rýchlejšie ako desatinné keoficienty.

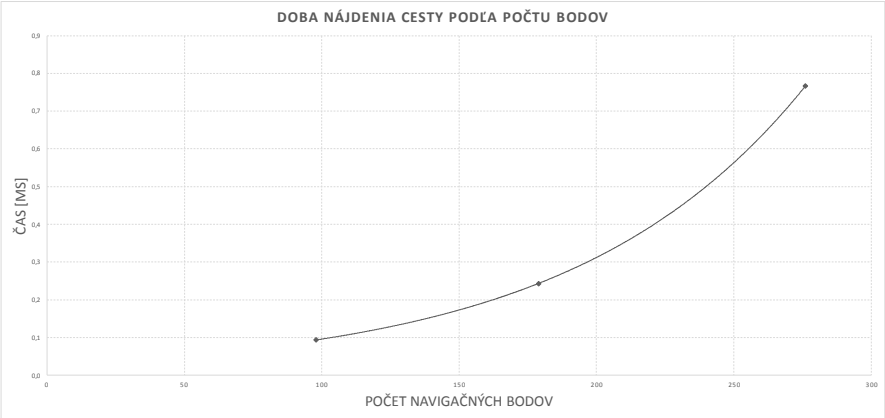
Nájdenie cesty v grafe najviac závisí od veľkosti mapy. Rastie exponenciálne s počtom navigačných bodov. Môže ho mierne ovplyvniť aj voľba keoficientov (ak chceme brať do úvahy veci na navigačných bodoch). Napriek tomu je hľadanie cesty celkovo veľmi rýchle, keďže mapy majú pomerne málo navigačných bodov a keoficienty toto hľadanie výrazne nezdržujú.

| počet destinácií | DM-Deck17 | DM-DE-Grendelkeep | DM-Oceanic |
|---|---|---|---|
| 1 | 0,767 | 0,243 | 0,094 |
| 2 | 1,409 | 0,416 | 0,141 |
| 3 | 1,988 | 0,573 | 0,190 |
| 4 | 2,626 | 0,716 | 0,239 |
| 5 | 3,037 | 0,854 | 0,279 |

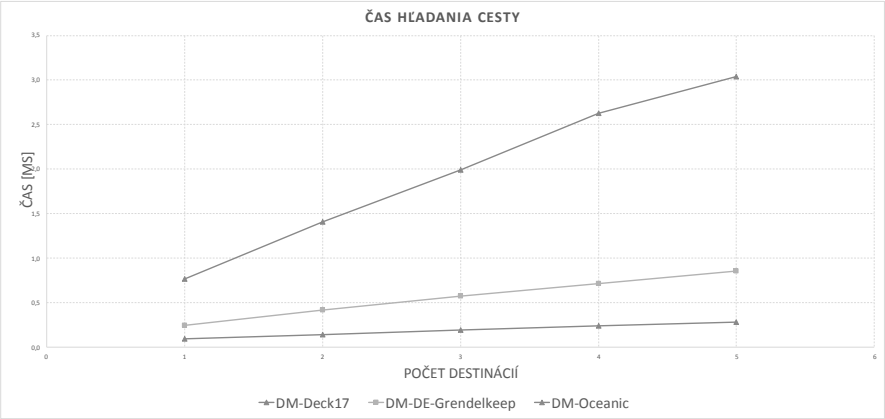
Tabulka 7.5: Doba nájdenia jednej cesty bez ohodnotenia vecí v milisekundách.

| počet destinácií | 0,5 | 0,1 | 0,01 | 0 | rôzne |
|---|---|---|---|---|---|
| 1 | 0,66165 | 1,28428 | 1,35533 | 0,59443 | 1,28943 |
| 2 | 1,21955 | 2,07648 | 2,1787 | 1,07577 | 2,12772 |
| 3 | 1,78522 | 2,96273 | 2,98489 | 1,65353 | 2,87141 |
| 4 | 2,19014 | 3,59804 | 3,62535 | 2,09256 | 3,4751 |
| 5 | 2,70454 | 4,06877 | 4,36798 | 2,75983 | 4,49717 |

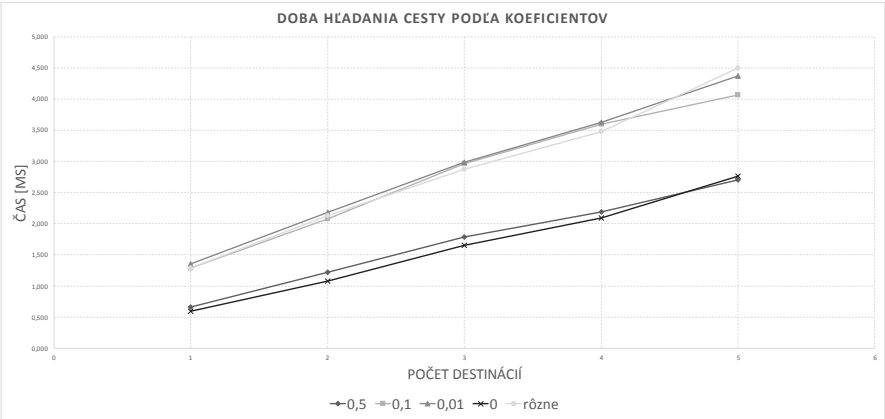
Tabulka 7.6: Rýchlosť hľadania cesty pri rôznych keoficientoch v milisekundách.




(a) Graf závislosti doby nájdenia cesty od počtu navigačných bodov mapy.




(b) Graf závislosti doby nájdenia cesty od počtu destinácií a veľkosti mapy.




(c) Graf závislosti doby nájdenia cesty od rôznych koeficientov.

Obrázek 7.6: Čas hľadania ciest.

# Kapitola 8

# Záver

Cieľom práce bolo vytvoriť rozhranie pre tvorbu agentov v jazyku Jason. Týto agenti by mali hrať hru Unreal Tournament a ovládať svojho bota prostredníctvom prostredia GameBots. Aby sa dalo použiť v praxi, malo by byť ľahko použiteľné a najmä efektívne. Agenti sa musia rozhodovať v reálnom čase.

Pri vytváraní tejto práce som sa musel zoznámiť s BDI architektúrou, jazykom AgentSpeak a jeho rozšírením Jason, teóriou multiagentných systémov a prostredím GameBots, ktoré slúži na ovládanie botov pre Unreal Tournament prostredníctvom textového protokolu. Mojou úlohou bolo navrhnúť rozhranie, ktoré by tieto komponenty spájalo do jednoduchého rozhrania.

Implementované riešenie sa skladá z dvoch hlavných častí. Je tu prostredie obsahujúce informácie o zápase a mape. Spravuje ich a poskytuje všetkým agentom. Ďalej sú tu agenti, ktorý ovládajú svojho bota v hre. Majú o ňom informácie a príkazmy textového protokolu GameBots mu hovoria, čo má robiť. Obe komponenty majú set informácií. V prostredí obsahuje globálne informácie, agenti v ňom majú stav svojho bota a jeho okolia. Tieto sety spravujú svojho poslucháča, ktorý zodpovedá za pripojenie na GameBots server a prijímanie správ, ktoré posiela. Správy sú spracovávané spracovačom správ, ktorý z nich vytvorí objekty a aktualizuje príslušný set informácií.

Trieda pre agentov obsahuje sadu základných vnemov a akcií. S ich využitím sme vytvorili agenta, ktorý dokáže hrať mód deathmatch. Experimentami sme zistili, že doba úvahového cyklu sa pohybuje okolo 65 milisekúnd. To je postačujúce na to, aby sa mohol agent rozhodovať v reálnom čase. Táto doba nezávisí od mapy ani počtu agentov.

Správy prichádzajúce zo servera sú tiež spracovávané rýchlo. Jedna správa za 5 nanosekundu. Pri počte správ 60 000 za 10 minút, ich spracovanie zaberie 300 milisekúnd.

Ďalší vývoj by som zameral na rozšírenie množiny vnemov a akcií poskytovaných agentov a na prispôsobenie pre ďalšie herné módy.

# Literatura


[1] Digital Extremes website. http://www.digitalextremes.com/.

[2] Eclipse. https://eclipse.org/.

[3] Epic Games website. http://epicgames.com/.

[4] GameBots stranka.
http://pogamut.cuni.cz/main/tiki-index.php?page=GameBots.

[5] GameBots user documentation.
http://pogamut.cuni.cz/pogamut_files/latest/doc/gamebots/ch06.html/.

[6] Jason, a Java-based interpreter for an extended version of AgentSpeak.
http://jason.sourceforge.net/wp/.

[7] Jason plugin pre eclipse.
http://jason.sourceforge.net/mini-tutorial/eclipse-plugin/.

[8] Java download. https://java.com/en/download/.

[9] Typy prechodov medzi navigačnými bodmi.
http://wiki.beyondunreal.com/Legacy:ReachSpec.

[10] Unreal Tournament. http://www.unrealtournament.com/blog/.

[11] Bida, M.; Cerny, M.; Gemrot, J.; aj.: Evolution of GameBots project.
*In: Herrlich*, ročník 7522, 2012: s. 397-400.

[12] Bordini, R. H.; Hübner, J. F.; Wooldridge, M.: *Programming Multi-Agent Systems in AgentSpeak using Jason*. John Wiley & Sons, 2007, ISBN 978-0-470-02900-8.

[13] Kaithofis: Oceanic map. 2007.
URL http://liandri.beyondunreal.com/File:DM-Oceanic-Pickups.png

[14] Patel, A.: Amit's A* Pages.
http://theory.stanford.edu/ amitp/GameProgramming/index.html.

[15] Rao, A.: *AgentSpeak(L): BDI agents speak out in a logical computable language*. Springer Berlin Heidelberg, 1996, ISBN 978-3-540-60852-3.

[16] Wormbo: Deck17 map. 2007.
URL http://liandri.beyondunreal.com/File:Deck17-Pickups.png

[17] Wormbo: Grendelkeep map. 2007.
URL http://liandri.beyondunreal.com/File:Grendelkeep-Pickups.png

# Příloha A

# Obsah CD

- **src** obsahuje zdrojové súbory
- **jar** obsahuje implementáciu v jar súbore
- **tex** obsauje zdrojové súbory textu práce v LaTeXe
- **example** obsahuje ukážkovú implementáciu agenta s použitím vytvoreného rozhrania

# Příloha B

# Vnemy a akcie agenta.

Vnemy:

- `my_id(ID)` - Identifikátor bota v Unreal Tournamente.
- `location(ID,X,Y,Z)` - Pozícia entity danej identifikátorom. Môže to byť hráč, zbraň, navigačný bod.
- `inventory(Weapon)` - Zbraň, ktorú má v inventáry.
- `visible(ID)` - Identifikátor veci, ktorú vidím. Môže to byť hráč, vec, navigačný bod.
- `player(P)` - Entita P je hráč.
- `nav_point(NP)` - Entita NP je navigačný bod.
- `end_game` - Hra skončila.
- `connect` - Úspešne sa pripojil na server.
- `spawned` - Bot sa objavil na mape.
- `picked(ITEM,AMOUNT)` - Zodvihol vec ITEM s množstvom AMOUNT.
- `died` - Bot zomrel.
- `path(ID,N,X,Y,Z,F)` - Navigačný bod cesty. ID je identifikátor bodu (pre navigačné body je to Id zo správ poslaných GameBots, pre ciele identifikátor zadaný užívateľom, začiatočný bod má ako identifikátor agentovo meno), N je poradové číslo začínajúce 0, X, Y, Z sú súradnice bodu a F je flag pre pohyb.
- `match(ID)` - Práve prebieha zápas na mape s názvom ID.
- `match_end` - Zápas sa skončil.

Akcie:

- `turn_to(X,Y,Z)` - Otoč sa na pozíciu zadanú súradnicami v parametri.
- `jump` - Vyskoč.

- `jump(DD,DELAY,FORCE)` - Zložitejší výskok. Parametre odpovedajú správe pre GameBots. DD bude True, ak chceme, aby bot vykonal dvojitý skok. DELAY určí oneskorenie, s ktorým vyskočí druhý krát, FORCE je vektor, určujúci aký veľký bude skok.
- `connect` - Pripoj sa na server.
- `disconnect` - Odpoj sa od servera.
- `stop_mov` - Zastav pohyb.
- `move(X,Y,Z)` - Pohni sa na pozíciu X, Y, Z.
- `stop_shoot` - Zastav streľbu.
- `shoot(ID)` - Začni strieľať na vec, určenú identifikátorom ID.
- `cwp(W)` - Zmeň zbraň na W. Zbraň musí byť v inventáry.
- `game_in_progress` - Zistí, či práve prebieha nejaká hra.
- `get_path(X,Y,Z,[dest(ID,Xd,Yd,Zd)])` - Nájde cestu do zoznamu destinácií dest(...). X, Y, Z je štartovacia pozícia.

# Příloha C

# Použitie

Pre vytvorenie bota budeme potrebovať nainštalovanú hru Unreal Tournament 2004 [10], Javu [8], GameBots [4], Jason [6], Eclipse [2] a jeho plugin [7]. Každá z týchto stránok ponúka návod na ich inštaláciu.

Teraz potrebujeme pridať rozhranie do cesty zostavenia projektu a nastaviť classpath premennú v súbore .mas2j.

```
environment: unrealEnvironment.UnrealEnvironment
classpath: "lib/UnrealFramework.jar";
```

Pre prispôsobenie vlastného agenta, vytvoríme triedu, ktorá dedí od UTArch. Tá poskytuje základnú funkcionalitu. Tú si môžeme rozšíriť. Napríklad chceme vložiť vlastný set informácií s novým `MessageProcessor`om.

```
public class MySelfSet extends SelfSetImpl{
        public MySelfSet(UTArch utArch){
                super(utArch);
                setListener(new ListenerImpl.Builder(new
                    MyProcessor(this)).addInitMessage(
                    GameBotsActions.ready()).build());
        }
}
```

```
public class SampleUTArch extends UTArch{

        public SampleUTArch(){
                setSelfSet(new MySelfSet(this));
        }
}
```

Keď chce agent vykonať akciu, volá sa metóda `act( ...)`. Pre pridanie vlastnej akcie ju prepíšeme. Vložíme volanie rodičovskej implementácie, ak chceme využívať akcie z nej. Zoznam akcií v prílohe B.

```
@Override
public void act(ActionExec action, List<ActionExec> feedback){
        boolean result = false;
        Structure act = action.getActionTerm();
```

```
        switch(act.getFunctor()){
        case "move":
                if(act.getArity() == 6){
                        Location loc1 = new Location(act.getTerm
                            (0), act.getTerm(1), act.getTerm(2));
                        Location loc2 = new Location(act.getTerm
                            (3), act.getTerm(4), act.getTerm(5));
                        result = sendMessage(GameBotsActions.move(
                            loc1.toString(), loc2.toString(), null,
                             null));
                }else{
                        super.act(action, feedback);
                        return;
                }
        break;
        default:
                super.act(action, feedback);
                return;
        }
action.setResult(result);
feedback.add(action);
}
```

Pri aktualizácii vnemov volá Jason funkciu `perceive()`. Implementácia v triede `UTArch` vyberie zo `SelfSetu` informáciu o botovi, viditeľné objekty a udalosti a pre každý objekt zavolá metódu `getPercepts( ...)`. Tá vráti zoznam vnemov z danej veci. Ak chceme pridať vlastný vnem, prepíšeme metódu tak, aby vracala zoznam aj s novým literálom.

```
@Override
protected List<Literal> getPercepts(UTobject o){
        List<Literal> result = super.getPercepts(o);
        switch(o.getType()){
        case SLF:
                result.add(JasonUtils.percept("health", o.get("
                    Health")));
                result.add(JasonUtils.percept("adrenaline", o.get(
                    "Adrenaline")));
        break;
        }
        return result;
}
```

# Príloha D

# Výsledky experimentov

| DM-Deck17 | | | | | | | |
|---|---|---|---|---|---|---|---|
| agentov | správ | čas správy | vnemov | čas vnemu | akcií | čas akcie | čas cyklu |
| 2 | 56 291 | 0,00302 | 12 706 | 4,18 | 1 058 | 0,22 | 48,50 |
| 5 | 62 826 | 0,00475 | 8 877 | 5,59 | 1 038 | 0,22 | 69,72 |
| 10 | 68 886 | 0,00393 | 9 647 | 6,81 | 1 617 | 0,21 | 66,86 |
| 15 | 72 359 | 0,00547 | 9 251 | 7,11 | 1 847 | 0,20 | 68,52 |

Tabulka D.1: Výsledky z mapy Deck17. Čas v milisekundách.

| DM-DE-Grendelkeep | | | | | | | |
|---|---|---|---|---|---|---|---|
| agentov | správ | čas správy | vnemov | čas vnemu | akcií | čas akcie | čas cyklu |
| 2 | 59 732 | 0,00553 | 8 648 | 6,98 | 744 | 0,26 | 69,60 |
| 5 | 62 014 | 0,00414 | 9 145 | 6,53 | 980 | 0,23 | 66,68 |
| 10 | 71 010 | 0,00572 | 9 113 | 8,18 | 1 403 | 0,26 | 66,90 |
| 15 | 77 323 | 0,00680 | 9 532 | 9,31 | 1 852 | 0,22 | 63,19 |

Tabulka D.2: Výsledky z mapy Grendelkeep. Čas v milisekundách.

| DM-Oceanic | | | | | | | |
|---|---|---|---|---|---|---|---|
| agentov | správ | čas správy | vnemov | čas vnemu | akcií | čas akcie | čas cyklu |
| 2 | 51 297 | 0,00376 | 11 134 | 2,90 | 1 001 | 0,23 | 53,80 |
| 5 | 55 612 | 0,00395 | 9 882 | 2,93 | 1 147 | 0,20 | 60,92 |
| 10 | 65 103 | 0,00517 | 10 245 | 3,64 | 1 605 | 0,19 | 58,60 |
| 15 | 74 785 | 0,00527 | 10 706 | 3,80 | 2 037 | 0,17 | 56,33 |

Tabulka D.3: Výsledky z mapy Oceanic. Čas v milisekundách.

| mapa | počet ciest | počet destinácií | čas bez reward | čas s reward |
|---|---|---|---|---|
| DM-Deck17 | 100 000 | 1 | 0,767 | 0,662 |
| DM-Deck17 | 100 000 | 2 | 1,409 | 1,220 |
| DM-Deck17 | 100 000 | 3 | 1,988 | 1,785 |
| DM-Deck17 | 100 000 | 4 | 2,626 | 2,190 |
| DM-Deck17 | 100 000 | 5 | 3,037 | 2,705 |
| DM-DE-Grendelkeep | 100 000 | 1 | 0,243 | 0,199 |
| DM-DE-Grendelkeep | 100 000 | 2 | 0,416 | 0,331 |
| DM-DE-Grendelkeep | 100 000 | 3 | 0,573 | 0,481 |
| DM-DE-Grendelkeep | 100 000 | 4 | 0,716 | 0,569 |
| DM-DE-Grendelkeep | 100 000 | 5 | 0,854 | 0,684 |
| DM-Oceanic | 100 000 | 1 | 0,094 | 0,105 |
| DM-Oceanic | 100 000 | 2 | 0,141 | 0,150 |
| DM-Oceanic | 100 000 | 3 | 0,190 | 0,201 |
| DM-Oceanic | 100 000 | 4 | 0,239 | 0,250 |
| DM-Oceanic | 100 000 | 5 | 0,279 | 0,290 |

Tabulka D.4: Výsledky algoritmu A*. Čas v milisekundách.

| počet destinácií | 0,5 | 0,1 | 0,01 | 0 | rôzne |
|---|---|---|---|---|---|
| 1 | 0,66165 | 1,28428 | 1,35533 | 0,59443 | 1,28943 |
| 2 | 1,21955 | 2,07648 | 2,1787 | 1,07577 | 2,12772 |
| 3 | 1,78522 | 2,96273 | 2,98489 | 1,65353 | 2,87141 |
| 4 | 2,19014 | 3,59804 | 3,62535 | 2,09256 | 3,4751 |
| 5 | 2,70454 | 4,06877 | 4,36798 | 2,75983 | 4,49717 |

Tabulka D.5: Výsledky algoritmu A* s rôznymi koeficientami pre mapu Deck17. Čas v milisekundách.

# Příloha E

# Kód agenta

```
{include("movement.asl")}
//=====================================
//Connections
@connect[atomic]
+match(_):not connected<-
        +connected;
        game_in_progress;
        connect.

@disconnect[atomic]
+match_end:connected<-
        -connected;
        disconnect.

//=====================================
//Weapons
@change_weapon
+picked(W):not .substring("shieldgun",W)<-
        cwp(W).

//=====================================
//Spawn, Death
@spawned
+spawned<-
        !wander.
@died
-died<-
        !stop_movement;
        -clearpath.
@movement_stopped
-moving<-
        !wander.
@wander_around
+!wander<-
```

```
        -dest(_,_,_,_);
        ias.random_location(X,Y,Z);
        +dest(rand,X,Y,Z);
        !start_movement.
-!wander<-
        true.
@shoot
+location(P,_,_,_):player(P)<-
        shoot(P).
@stop_shoot
-player(P)<-
        stop_shoot.
```

```
my_position(X,Y,Z):- my_id(ME) & location(ME,X,Y,Z).
near:- my_position(Xme,Yme,Zme) & path_num(N) & path(_,N,X,Y,Z,_)
   & ias.distance(X,Y,Z,Xme,Yme,Zme,100).
next(N+1,X,Y,Z,F):- path_num(N) & path(_,N+1,X,Y,Z,F).
on_mover:- path_num(N) & path(ID,N,_,_,_,_) & mover(ID).

//=================================
//start and stop movement
@start_movement[priority(1),atomic]
+!start_movement:not moving<-
        +moving[expires(clearpath)];
        .findall(dest(ID,X,Y,Z),dest(ID,X,Y,Z),L);
        ?my_position(Xme,Yme,Zme);
        get_path(Xme,Yme,Zme,L).
@stop_movement[atomic]
+!stop_movement<-
        stop_mov;
        +clearpath[expires(clearpath)].

@start_path[atomic]
+path(ID,0,X,Y,Z,F)<-
        +path_num(1)[expires(clearpath)];
        !move_to_next.
//=================================
//Updates
@location_change
+location(ME,_,_,_): my_id(ME) & near & not on_mover<-
        !move_to_next.
@location_change_on_mover
+location(ME,X,Y,Z): my_id(ME) & on_mover & next(_,X1,Y1,Z1,_) &
   ias.jump(X,Y,Z,X1,Y1,Z1,H,V) & V < 100 & V > -100<-
        !move_to_next.

@at_destination
-path_num(N):path(ID,N,_,_,_,_) & dest(ID,_,_,_)<-
```

```
        -dest(ID,_,_,_).

@path_num_and_near
+path_num(N):near<-
        !!move_to_next.

@path_num
+path_num(N)<-
        .succeed_goal(antistuck(_,_,_,_));
        ?my_position(X,Y,Z);
        !!antistuck(N,X,Y,Z).

//=====================================
//Antistuck
@antistuck
+!antistuck(N,X,Y,Z)<-
        .wait(5000);
        ?path_num(N1);
        ?my_position(X1,Y1,Z1);
        N == N1;
        ias.distance(X,Y,Z,X1,Y1,Z1,50);
        !stop_movement.
@antistuck_restart[priority(1)]
-!antistuck(_,_,_,_):path_num(N) & my_position(X,Y,Z)<-
        !!antistuck(N,X,Y,Z).
@antistuck_finish
-!antistuck(_,_,_,_)<-
        true.

//=====================================
//Movement
@move_to_next[priority(2),atomic]
+!move_to_next:next(N,X,Y,Z,F)<-
        -+path_num(N)[expires(clearpath)];
        !step(X,Y,Z,F).
@move_to_next_and_finished[priority(1),atomic]
+!move_to_next:path_num(N) & not path(ID,N+1,_,_,_,_) & dest(ID,_,
    _,_)<-
        -dest(ID,_,_,_);
        !stop_movement.
@move_to_next_def[priority(0)]
+!move_to_next<-
        true.

//=====================================
//Steps
@step_forced[priority(10),atomic]
+!step(X,Y,Z,F):F >= 256<-
```

```
        !stop_movement.

@step_special[priority(7),atomic]
+!step(X,Y,Z,F):F >= 32<-
        move(X,Y,Z);
        !step(X,Y,Z,F-32).

@step_door[priority(8),atomic]
+!step(X,Y,Z,F):F >= 16<-
        move(X,Y,Z);
        !step(X,Y,Z,F-16).

@step_jump[priority(7),atomic]
+!step(X,Y,Z,F):F >= 8<-
        ?path_num(N);
        ?path(_,N-1,Xs,Ys,Zs,_);
        ias.jump(Xs,Ys,Zs,X,Y,Z,Horizontal,Vertical);
        !jump(X,Y,Z,Horizontal,Vertical);
        !step(X,Y,Z,F-8).
@jump_too_high[priority(7),atomic]
+!jump(X,Y,Z,H,V):V > 340<-
        !stop_movement.
@jump_too_far[priority(6),atomic]
+!jump(X,Y,Z,H,V):H > 850<-
        !stop_movement.
@jump_up[priority(5),atomic]
+!jump(X,Y,Z,H,V):V > -10 & V < 180 & H < 250<-
        move(X,Y,Z);
        .wait(75);
        jump;
        move(X,Y,Z).
@jump_up_double[priority(4),atomic]
+!jump(X,Y,Z,H,V):V > -10 & V < 340 & H < 500<-
        move(X,Y,Z);
        .wait(75);
        jump.
@jump[priority(3),atomic]
+!jump(X,Y,Z,H,V):H < 500<-
        move(X,Y,Z);
        .wait(75);
        jump.
@jump_far[priority(2),atomic]
+!jump(X,Y,Z,H,V):H < 850<-
        move(X,Y,Z);
        .wait(75);
        jump(true,0.7,null).
@jump_down[priority(1),atomic]
+!jump(X,Y,Z,H,V): V < -10<-
```

```
        move(X,Y,Z).

@step[priority(6),atomic]
+!step(X,Y,Z,F):F >= 1<-
        move(X,Y,Z);
        !step(X,Y,Z,F-1).

@step_successful[priority(1),atomic]
+!step(X,Y,Z,F):F < 1<-
        true.

@step_dont_know_how[atomic]
+!step(X,Y,Z,F)<-
        ?path_num(N);
        ?path(ID,N,_,_,_,_);
        ?path(ID1,N-1,_,_,_,_);
        .print(F," from ",ID1," to ",ID).
```