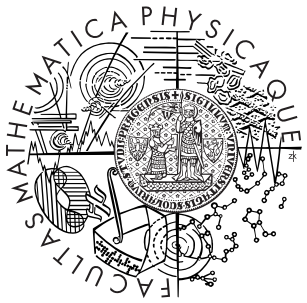
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Tomáš Hubík

Umělá inteligence pro strategické hry

Katedra teoretické informatiky a matematické logiky

Mgr. Ondřej Sýkora
Informatika: Programování

2010

Na tomto místě bych chtěl poděkovat Mgr. Ondřeji Sýkorovi za vedení bakalářské práce a za poskytnutí mnoha cenných rad a materiálů. Dále bych chtěl poděkovat svému bratrovi za testování platformy a dobré připomínky. V neposlední řadě děkuji mým rodičům a přátelům za trpělivost a podporu.

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

V Praze dne 26. června 2010 Tomáš Hubík

Obsah

Název práce: Umělá inteligence pro strategické hry
Autor: Tomáš Hubík
Katedra: Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Mgr. Ondřej Sýkora
e-mail vedoucího: ondrasej@centrum.cz

Abstrakt: V předložené práci se zabývám návrhem jednoduché tahové strategické hry a implementací platformy na testování algoritmů pro tuto hru. Další částí práce je implementace několika různých algoritmů pro tuto platformu. Naimplementoval jsem jeden algoritmus založený na principu analýzy mapy a herního prostředí bez jakékoliv predikce a prohledávání stavového prostoru. Dále dva algoritmy založené na prohledávání a rozhodování s pomocí upraveného Minimaxového algoritmu. Poslední dva algoritmy jsou inspirované metodou Monte Carlo plánování.

Klíčová slova: umělá inteligence, tahové strategické hry, Monte Carlo plánování, testování algoritmů umělé inteligence

Title: Artificial intelligence for stretegic games
Author: Tomáš Hubík
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Mgr. Ondřej Sýkora
Supervisor's e-mail address: ondrasej@centrum.cz

Abstract: In the present work I devote to simple turn-based strategic game design and implementation of a platform for testing algorithms for this game. Another part of the work is implementation of various types of algorithms for this platform. I have implemented one algorithm based on map and game environment analysis without any prediction or searching the game state space. Next two algorithms are based on searching the game state and making decisions using modified Minimax algorithm. The last two algorithms are inspired by method called Monte Carlo Planning.

Keywords: artificial intelligence, turn-based strategy games, Monte Carlo Planning, artificial intelligence algorithms testing

Kapitola 1

Úvod

1.1 Umělá inteligence

1.1.1 Historie

Podle knížky Umělá inteligence 1 [1] si otázku „Mohou stroje myslet?" pokládali významní filosofové jako Descartes, Pascal nebo Hobbes již v 17. století. V této době se ovšem řešila pouze existenční filosofická otázka, zda ano, či ne, ale nikdo se již nezabýval tím, jak strojového myšlení dosáhnout.

První významné praktické výsledky v tomto oboru přinesl až matematik a fyzik Kurt Gödel ve 30. letech 20. století. V roce 1930 publikoval větu o úplnosti predikátové logiky prvního řádu a o rok později ještě zásadnější dvě věty o neúplnosti axiomatických formálních systémů s aritmetikou. Tím ukázal, že není možné navrhnout soubor axiomů, pomocí kterých by se dala zodpovědět každá otázka formulovaná uvnitř formálního systému. Poukázal tak na možnost existence neřešitelných problémů.

Velký rozmach výpočetní techniky po druhé světové válce zájem a úsilí o napodobení intelektuálních schopností člověka ještě znásobil. Systémy v té době byly navrhovány a ověřovány spíše experimentálně. Bylo několik možností, jak k návrhu přistupovat. Jsou využívány například techniky vycházející z detailní analýzy fungování nervové soustavy živých organismů. Z těchto analýz vychází například neuronové sítě a genetické algoritmy, které si popíšeme dále. O použití neuronových sítí a genetických algoritmů ve spojitosti s počítačovými hrami pojednává knížka *AI Techniques for Game Programming* [2]. Dalším směrem byla snaha abstrahovat mentální procesy lidského mozku na úrovni psychologické a kognitivní. Z toho těží metody reprezentace

a využívání znalostí ve stavovém prostoru, modely metod učení a další.

V roce 1950 byl pro tento obor velmi populární. Alan Turing v té době formuloval slavný Turingův test[1] a ve stejné době John von Neumann řekl, že stroje v dohledné době překonají intelektuální schopnosti člověka.

Ovšem za počátek oboru umělá inteligence se považuje rok 1956, kdy John McCarthy zorganizoval konferenci, na které se řešila domněnka, že „každé hledisko učení nebo jakýkoliv jiný příznak inteligence může být v principu tak přesně popsán, že může být vyvinut stroj, který ho simuluje" [3]. Díky návrhu McCarthyho se celému vznikajícímu oboru začalo říkat *umělá inteligence*.

1.1.2 Pojem umělá inteligence

Vzhledem k tomu, jak je pojem umělá inteligence mladý, není dodnes jednoznačně definován a skrývá se pod ním mnoho významů a oborů: Filosofie (jak v mozku vzniká mysl, logika, metody uvažování), matematika (jak formálně odvodit platné závěry, co lze spočítat), ekonomie (maximalizace zisku, rozhodování), neurověda (jak mozek zpracovává informace), psychologie (jak lidé a zvířata myslí, behaviorismus), počítačové inženýrství (stroje na zpracování informací, efektivní počítače), teorie řízení (maximalizace efektu v čase), lingvistika (reprezentace znalostí) a mnohé další.

Minského definice

Jednu z definic vyslovil Marvin Minsky v roce 1967 [4]: „Umělá inteligence je věda o vytváření strojů nebo systémů, které budou při řešení určitého úkolu užívat takového postupu, který — kdyby ho dělal člověk — bychom považovali za projev jeho inteligence."

Tato definice vychází z Turingova testu a říká vlastně, že umělá inteligence řeší takové úlohy, na které by člověk musel uplatnit svoji inteligenci. Z této definice vychází, že řešit něco inteligentně znamená řešit to jako člověk.

[1] Touringův test je test rozlišitelnosti stroje a člověka. Stroj projde testem, pokud není člověk na základě písemné komunikace schopen rozpoznat, zda na druhé straně je člověk nebo stroj.

Definice Richové

Další definice pochází od dvojice Rich a Knight z roku 1991 [5] a říká toto: „Umělá inteligence se zabývá tím, jak počítačově řešit úlohy, které dnes zatím zvládají lidé lépe."

Tato definice přímo říká, že umělá inteligence je tedy vázána na aktuální stav v oblasti informatiky a s časem se její náplň mění. Bohužel nezahrnuje úlohy, které je třeba vyřešit a člověk je řešit neumí.

Kotkova definice

Tuto definici vyslovil Kotek v roce 1983 [6]: „Umělá inteligence je vlastnost člověkem uměle vytvořených systémů vyznačujících se schopností rozpoznávat předměty, jevy a situace, analyzovat vztahy mezi nimi a tak vytvářet vnitřní modely světa, ve kterých tyto systémy existují, a na tomto základě pak přijímat účelná rozhodnutí za pomoci schopností předvídat důsledky těchto rozhodnutí a objevovat nové zákonitosti mezi různými modely nebo jejich skupinami."

Kotkova definice tedy zahrnuje i podobory jako počítačové vidění, reprezentace, formalizace a uchovávání znalostí a řešení úloh v ní.

1.1.3 Umělá inteligence a hry

Umělá inteligence má velmi široké spektrum záběru. Přes akademickou sféru, plánování výroby až po zábavní průmysl a počítačové hry.

V počítačových hrách má umělá inteligence za úkol dotvářet herní svět a suplovat lidského hráče jako protivníka nebo partnera. Přesto, že je ve hrách umělá inteligence často až na druhém místě za grafikou, hraje velmi důležitou roli a její podcenění může celou hru zcela degradovat.

Podle knihy *AI Game Development* [7] lze způsoby implementace umělé inteligence v počítačových hrách rozdělit na dva typy:

Tradiční přístup

Zde je umělá inteligence přímo součástí hry a má tedy přístup ke všem informacím ve hře. Takto implementované algoritmy dosahovaly často velmi dobrých výsledků také proto, že měly přístup k informacím, ke kterým třeba hráč přístup neměl, což jim dávalo velkou výhodu. Nevýhoda tohoto řešení

ovšem byla, že umělá inteligence byla se hrou tak spjata, že ji většinou nebylo možné v budoucnu příliš modifikovat nebo vyměňovat.

Moderní přístup

Tento přístup zcela odděluje hru a umělou inteligenci. Umělá inteligence je pak zabalena do agentů komunikující se hrou pomocí rozhraní. Výhoda tohoto řešení je, že se algoritmy mohou vyvíjet a testovat naprosto odděleně od hry.

Toto řešení se ale v praxi příliš nepoužívá a je rozšířeno hlavně v akademické sféře.

Testovací platforma, která je hlavní náplní této práce, je postavena právě na tomto přístupu.

1.2 Cíle práce

Cílem této práce je navrhnout a vytvořit platformu na testování různých algoritmů a přístupů k umělé inteligenci ve strategických tahových hrách. Pojmem tahová strategická hra se zabýváme podrobněji v kapitole 2.

Platforma by měla umožňovat jednoduché porovnávání algoritmů, co se týče kvality herní strategie i rychlosti rozhodování. Dále by měla umožňovat ukládat průběhy her pro budoucí analýzu odehraných strategií. Také by zde měla být možnost napojení více klientů přes počítačovou síť.

Dalším cílem bylo navrhnout několik algoritmů pro tuto platformu. V práci se zabýváme výhradně taktickou, bojovou částí hry, která je asi nejdůležitějším problémem této oblasti. V průběhu hry budou algoritmy vybírat konkrétní akce a příkazy, které by měly vést k úspěchu a výhře algoritmu.

1.3 Související práce

Bohužel jsme nenašli žádnou podobnou platformu, která by umožňovala jednoduše implementovat vlastní algoritmy umělé inteligence a pak je vzájemně proti sobě nebo proti lidskému hráči testovat.

Projekty existují hlavně pro real-time strategické hry (RTS). Velká část algoritmů a přístupů používaných v těchto hrách se dá použít i v tahových hrách, ovšem obráceně toto již neplatí, a to hlavně pro potřebu rozhodnout se velmi rychle. V případě prodlevy v tahové strategii riskujeme pouze

znechucení uživatele, ale v případě prodlevy v RTS může dojít i k porážce. Proto jsou takové platformy pro náš problém těžko použitelné. Přesto uvedu nějaké příklady takových projektů.

Wargus

Wargus je v podstatě předělaná hra Warcraft 2 [8] tak, aby fungovala s herním enginem Stargus, což je volně šiřitelný herní engine pro 2D strategické hry. Stargus umožňuje do velké míry změnou parametrů měnit herní prostředí, a proto je na testování algoritmů v různých podmínkách docela vhodný. Bohužel se ale zdá, že je projekt od roku 2007 neaktualizovaný a tím pádem bez jakékoliv podpory. Více se o enginu Stargus dá dočíst například na Wikipedii [9], kde jsou informace o něm pěkně shrnuty. Pokračovatelem projektu Stargus je engine Bos Wars [10].

FEAR

FEAR je platforma pro experimentování s různými algoritmy umělé inteligence. Bohužel ale tato platforma nedisponuje žádnými nástroji na měření jejich výkonu a kvality. Další nevýhodou je, že je projekt již od roku 2007 nepodporovaný. Domovská stránka projektu je: *http://fear.sourceforge.net* [11]. Hodně informací o projektu je napsáno také v knížce *AI Game Development* [7].

1.4 Struktura práce

V následující kapitole si podrobně popíšeme, co všechno se skrývá pod pojmem tahová strategická hra a jaký má význam pro reálný život. V další části této kapitoly uvedeme jednoduchý přehled metod přístupů k algoritmům.

V kapitole 3 popíšeme prostředí a pravidla hry, na jejímž základě je implementována celá platforma a pro kterou se budou vytvářet jednotlivé algoritmy. Dále zde popíšeme možnosti nastavení celé platformy.

Celá kapitola 4.1 je věnována architektuře platformy a to jak klientské, tak serverové části. Pro lepší názornost je zde text doplněn jednoduchými UML diagramy. Architektura je zde popsána pouze rámcově a podrobnější informace o vnitřní struktuře celé platformy jsou k nalezení v programátorské dokumentaci.

Kapitola 5 se věnuje již jednotlivým algoritmům implementovaným v platformě. Nejprve je zde popsáno, s jakými strukturami algoritmy pracují, pak jak se nový algoritmus do platformy implementuje a nakonec jsou zde popsány podrobně jednotlivé algoritmy. Tato kapitola také obsahuje grafy a výsledky měření z jednotlivých her, kde jsme testovali různé algoritmy proti sobě. Nakonec zde nastíníme možnost dalšího vývoje v této oblasti.

V kapitole 6 shrneme informace o implementaci platformy a informace o jednotlivých algoritmech a zkušenosti s nimi.

Kapitola 2

Tahové strategické hry

2.1 Vlastnosti tahových strategických her

Tahové strategické hry se dělí do mnoha kategorií podle počtu hráčů, počtu jednotek ovládaných hráčem, pozorovatelnosti prostředí (zda je vidět celé herní pole, nebo pouze jeho část), přítomnosti náhody (zda je další stav určen současným stavem a akcí agenta nebo do hry zasahuje například ještě náhoda), reprezentace prostředí (například dělení herního prostředí na čtvercovou síť), nebo také celkovým zaměřením. Tedy zda je možné stavět základnu a vyrábět jednotky nebo je hra zaměřena pouze na boj bez možnosti obnovy ztracených jednotek. V tahových strategických hrách je více rozšířená druhá možnost. Možnost stavět základnu bývá většinou doménou real-timových strategických her (RTS), jako je například Age of Empires [12], Warcraft [8] a další. Oproti RTS se tento žánr zásadně liší v omezení počtu úkonů, které může daný hráč v daný okamžik hry provést. Tím pádem se mezi hráči smazávají výhody, například v rychlých reakcích a schopnosti zvládat ovládání více jednotek najednou, protože zde má každý hráč na svůj tah neomezené množství reálného času. Zde se hra dělí na „tahy", kde v každém tahu může hrát pouze jeden hráč a dostane určitý počet bodů (někdy nazývaných akční body), které může „utratit" za akce, které v daném tahu provede. Body mohou být vázány buď na celý tah a hráče, nebo na jednotlivé jednotky. Po skončení tahu se dostává na řadu následující hráč a tak dále.

My se budeme z pohledu umělé inteligence — tak, jak ji dělí kniha *Artificial Intelligence: A Modern Approach* [13] — zabývat prostředími, která jsou plně pozorovatelná, deterministická, sekvenční (akce hráče závisí na

jeho předchozích akcích), statická (prostředí se nemění, zatímco hráč přemýšlí), diskrétní a multi-agentní (tedy hráč není v herním světě sám, ale má soupeře). Dále bude hra zaměřena pouze na bojovou složku bez možnosti jednotky dále vyrábět, obnovovat, či dokonce stavět základnu. A nakonec akční body budou vázány na celého hráče a budou sdíleny všemi jednotkami.

2.1.1 Problémy k řešení

Z pohledu umělé inteligence je třeba řešit mnoho aspektů hry. Ty se dají rozdělit na krátkodobé a dlouhodobé podle toho, jak velkou část hry, tedy například kolik tahů, zabírají.

Krátkodobé

Sem patří například hledání nejkratší cesty na herním plánu, což vede k lepšímu využití akčních bodů. Pak lokální boje a jejich vedení, predikce, co může protivník učinit v následujícím tahu a snažit se mu v tom bránit. Neúspěch v této oblasti většinou neznamená prohru v celé hře, ale pouze menší ztrátu. Opakované neúspěchy ale mohou mít na celkový výsledek zásadní vliv.

Dlouhodobé

Do této oblasti zapadá dlouhodobé plánování strategie a taktiky. Patří sem třeba analýza celého herního plánu a vytipování strategicky důležitých míst a jejich obsazování. Selhání v této oblasti, na rozdíl od minulé, již většinou povede k neúspěchu v celé hře. A naopak v případě dobrého zvládnutí této části může vyhrát hráč i proti poměrně velké početní převaze. Tato skutečnost není pouze imaginární problém vytvořený v počítačových hrách, ale je známa také v reálném světě. Nejedna bitva byla vyhrána z důvodu dobré znalosti terénu a využitím taktických schopností tehdejších vojevůdců byla mnohdy rozdrcena i mnohem početnější armáda.

2.1.2 Význam

Schopnost správně se rozhodnout v imaginárním světě tahových strategických her, tak vzdáleném od reálného života, nemusí být přínosem pouze pro herní průmysl. Poznatky a metody získané řešením úloh tohoto typu se

dají použít i v reálném světě, například při plánováni strategie pro vojenské akce.

2.2 Metody řešení strategických her

Bohužel není moc dobrých zdrojů, odkud by se daly čerpat kvalitní návrhy na algoritmy, protože ty nejlépe použitelné se vyskytují v komerční sféře, kde si je každá firma vyvíjí sama, a také pečlivě střeží jakožto součást obchodního tajemství. Mimo to nejlepší algoritmy většinou nepoužívají jednu konkrétní metodu přístupu, ale používají jich celou řadu, protože pro každou oblast a případ se hodí jiný algoritmus a jejich kombinací se dá dosáhnout lepšího výsledku. Částečně se dá čerpat z opensource her, kde je ovšem velmi často velmi špatná dokumentace, ze které se princip fungování umělé inteligence studuje velmi obtížně. Ze zdrojového kódu také často není na první pohled vidět, jaké myšlenky daný algoritmus používá, a to obzvláště když používá kombinace různých metod a algoritmů nebo optimalizace specifické pro danou hru. Nejlepším zdrojem jsou knížky, jako je například *AI Game Programming Wisdom* [14], kde jsou algoritmy a metody poměrně přehledně a dobře popsány. Dále existuje rozsáhlá série sborníků *Game Programming Gems*, která má v této době 8 dílů ([15], [16], [17], [18], [19], [20], [21], [22]).

Nyní uvedeme alespoň stručný přehled metod, které se používají k implementaci umělé inteligence do tahových strategických her. Podrobněji jsou jednotlivé metody popsány v knize *AI for Game Developers* [23].

2.2.1 Algoritmy založené na skriptech

Pod pojmem skript si zde můžeme představit nějaký zjednodušený programovací jazyk, který bude popisovat chování a vlastnosti jednotlivých entit ve hře.

Skriptování je jedna z nejjednodušších metod implementace umělé inteligence. Můžeme pomocí něj například jednoduše popsat, kudy mají jednotky chodit, co mají dělat, když narazí na nepřítele a tak dále. Výhodou je, že takto jednoduše implementovaný oponent se může na první pohled chovat velmi chytře, ale tento způsob také skýtá obrovskou nevýhodu. Takto vytvořený hráč se bude chovat vždy stejně a po jisté době bude již velmi dobře predikovatelný. Dále je takový skript velmi závislý na programátorovi a jeho

důvtipu. Mimo to bývají skripty závislé na daném herním plánu a nedají se již používat jinde.

Kvůli těmto omezením se skripty nejčastěji používají jako doplněk herního prostředí, kde slouží pro tvoření příběhu. Ve skriptech pak bývá zachycena jistá posloupnost akcí, které musí hráč udělat a hra na ně nějakým předem definovaným způsobem reaguje. Jde o jakési plnění úkolů a misí, které jsou předem definované.

2.2.2 Konečné automaty

Konečné automaty jsou zřejmě jednou z nejstarších metod, jak implementovat počítačové hráče. Například duchové v počítačové hře *Pac Man* [24] jsou řízeny konečnými automaty.

Díky své jednoduchosti se konečné automaty relativně dobře implementují a ladí, a proto jsou autory her často používány.

Algoritmus s konečným automatem tedy jako svůj vstup dostane herní stav. Podle herního stavu automat vybere přechodovou funkci a aplikuje ji na svůj aktuální stav. Dále se chová podle toho, v jakém stavu se právě nachází. Toto chování může být definováno například pomocí skriptů.

2.2.3 Fuzzy logika

Fuzzy logika je založena na myšlence, že reálný svět není pouze černobílý. Proto se snaží nahradit Booleovskou logiku, kde jsou pouze 2 hodnoty — pravda a nepravda. Touto logikou, kde jsou hodnoty již reálná čísla mezi 0 a 1 vyjadřující míru pravdivosti. Tím se rozhodování v tomto systému více podobá rozhodování člověka, který také analyzuje situace podle různých kritérií, která nejsou diskrétní, ale vyjadřují nějakou míru. Například když se člověk rozhoduje, zda je někdo vysoký, většinou jsou zde hodnocení jako malý, vysoký, docela vysoký, relativně malý a tak dále. Tedy nezajímá ho přesná hodnota, ale jen nějaká míra, podle které se dále rozhodne.

Fuzzy logika se do jisté míry dá nahradit standardní Booleovskou, ale její výhoda je v jisté intuitivnosti právě kvůli větší korespondenci s reálným světem. Dále je dokázáno [25], že pravidla psaná pomocí fuzzy logiky potřebují o 50% až 80% méně pravidel pro vyřešení stejných úloh.

V počítačových hrách se fuzzy logika používá v oblastech, kde je třeba vyjádřit nějakou míru. Například v simulátorech automobilových závodů se pomocí fuzzy logiky může programovat řízení automobilu oponenta. Příklad

fuzzy logiky může vypadat takto: „Když silnice zatáčí hodně doleva, zatoč hodně doleva, silnice zatáčí mírně, zatoč mírně." To nám dává jistou plynulost a realističnost chování oponentů.

Fuzzy logika samozřejmě podporuje i složitější formule, jako je to v logice Booleovské. Vyhodnocují se na podobném principu, ale výsledkem této formule bude opět reálné číslo vyjadřující míru jejich pravdivosti.

2.2.4 Algoritmy založené na pravidlech

Algoritmy založené na systémech pravidel jsou dnes zřejmě nejpoužívanější metodou. Tato metoda je svým způsobem velmi podobná algoritmům založeným na skriptech, ale zde jsou pravidla psána s mnohem menším polem působnosti. Tím je dosažena jejich opětovná znovupoužitelnost a obecnost. Jsou to systémy založené na množině podmínek ve tvaru if-then. Podmínky jsou buď použity přímo k nějakým akcím, nebo pouze k odvozování. Například když uvidím vojáka, odvodím, že nepřítel má kasárna.

Algoritmy založené na pravidlech mají podobné nevýhody, jako algoritmy založené na skriptech. Jsou závislé na programátorech, kteří musí vymyslet co nejoptimálnější strategii a zachytit jí v těchto pravidlech. Dále musí pomocí pravidel pokrýt co nejvíce herních stavů, což bývá velmi náročné na jejich počet a následnou údržbu.

2.2.5 Prohledávací algoritmy

Prohledávání stavového prostoru je jedna z nejuniverzálnějších a také myšlenkově nejméně náročných metod. Je založená na generování všech možných akcí ve hře, a poté hodnocení jejich výsledku. Nakonec se vybere taková akce, která vede k nejlepšímu předpokládanému výsledku. Nevýhodou tohoto přístupu je předpoklad existence dobré ohodnocovací funkce, která je pro správnou funkci celého algoritmu esenciální. Jediný případ, kdy ohodnocovací funkce není potřeba, je prohledání celého stavového prostoru, kde je již vidět, jaký hráč kdy vyhraje. To lze ale pouze u velmi jednoduchých her. Prohledávání stavového prostoru může být v případě her s velkým stavovým prostorem velmi časově i paměťově náročné, proto se používá spíše u jednodušších deskových her. Často se zde používají různá prořezávání a heuristiky, aby se prohledávaný prostor co možná nejvíce zmenšil.

2.2.6 Pravděpodobnostní algoritmy a Bayesovské sítě

Bayesovskou sítí zde rozumíme acyklický orientovaný graf, kde uzly reprezentují náhodné proměnné a hrany reprezentují závislosti mezi nimi. Dále máme na náhodných proměnných sdružené pravděpodobnostní rozdělení. Každý vrchol je pak podmíněně závislý pouze na svých potomcích.

Výhodou této metody je, že zde lze implementovat jistou formu učení, při které se zdokonalují pravděpodobnostní hodnoty jednotlivých podmínek, a tak i ve hrách se algoritmus může adaptovat na svého protivníka.

Přes tuto výhodu nejsou Bayesovské sítě v počítačových hrách moc rozšířené. Když už jsou použité, tak nejčastěji na predikci akcí protivníka. Své pole působnosti má tato metoda hlavně v expertních systémech profesionálnějšího zaměření. Používají se například v odhadování diagnózy pacienta ve zdravotnictví. Zde máme systém podmíněných pravděpodobnostních jevů ve tvaru: Pravděpodobnost příznaku za podmínky nemoci. Dále máme od pacienta seznam příznaků a snažíme se ze systému pravidel odvodit pravděpodobnost nemoci za podmínky všech příznaků.

2.2.7 Neuronové sítě

Neuronová síť je z pohledu umělé inteligence sítí navzájem pospojovaných buněk tří kategorií. Vstupní, kam přijde nějaký signál, například nějak zakódovaný problém, vnitřní a výstupní, kde by mělo nakonec vyjít řešení nebo akce vedoucí k řešení. Buňky po příchodu signálu tento signál modifikují podle své vnitřní funkce a výsledek rozešlou na výstupy. Každá buňka má uvnitř nějakou parametrizovanou matematickou funkci. Modifikací parametrů funkcí uvnitř buněk a pospojováním dostatečného množství těchto buněk by měla být síť schopna požadované funkce.

Podobně jako u Bayesovských sítí, i neuronové sítě jsou schopny učení. Ve skutečnosti to je hlavní metoda jejich ladění. Zde se dávají síti různé vstupy, a podle toho, zda je výstup žádoucí či nikoli, se parametry nějak mění.

Nejčastěji se tento přístup používá k rozpoznávání obrazu. V případě her se používají spíše na řešení obecnějších problémů, jako třeba zda útočit nebo se krýt. Pokud by měla síť řešit obecnější problémy, rostou velmi prudce nároky na její učení.

2.2.8 Genetické algoritmy

Genetické algoritmy jsou, spolu s neuronovými sítěmi, inspirovány přírodou a reálným světem. Konkrétně tento typ algoritmů je inspirován evolucí.

Jako takové se nepoužívají přímo na hledání řešení problémů, ale spíše se pomocí nich hledají nějaké optimální parametry pro jiné algoritmy. Nejprve se parametry vygenerují třeba náhodně, a pak se mezi sebou různě kombinují ty, co dávaly dobré výsledky s tím, že by výsledná kombinace mohla být ještě lepší. Do kombinace se často vkládá nějaký prvek náhody, aby byl počet možných vzniklých kombinací větší.

Tato metoda je další z těch, které jsou schopny se hráči přizpůsobit a tak odbourávat stereotyp, který mohou způsobovat algoritmy založené na skriptech nebo systémech pravidel.

2.3 Shrnutí

My se budeme zabývat tahovými strategickými hrami bez přítomnosti náhody, tudíž pravděpodobnostní algoritmy a fuzzy logika pro nás nejsou vhodné. Pravidla těchto her jsou také příliš složitá na sestavení neuronové sítě. Další nevýhodou je nedostatek dat, na kterých by se neuronová síť mohla učit. Z výše uvedeného přehledu jsou tedy nejvhodnější algoritmy založené na pravidlech a prohledávací algoritmy, kterým se budeme věnovat podrobněji.

Kapitola 3

Testovací platforma

Pro testování jednotlivých algoritmů a metod přístupu k implementaci umělé inteligence jsme navrhli a vytvořili jednoduchou strategickou hru, pro kterou budeme algoritmy psát. Hra je inspirována běžnými strategickými tahovými hrami, jako jsou Steel Panthers [26], Panzer General [27], Battle for Wesnoth [28], nebo Combat Tanks [29]. Pravidla jsme udělali co nejjednodušší, aby zbytečně nezatěžovala implementaci jednotlivých algoritmů nepotřebnými detaily. Zároveň jsou pravidla dostatečně složitá, aby se na nich daly demonstrovat výhody a nevýhody jednotlivých algoritmů v různých situacích.

Nyní si popíšeme herní prostředí a pravidla hry.

3.1 Herní prostředí

Herní prostředí se skládá z hracího pole a objektů na něm. Existují dva druhy objektů — jednotky a bonusy. Každý objekt má na mapě své místo, které nesdílí s žádným jiným objektem. Hráč pak může ovládat své jednotky podobně, jako by ovládal figurky v šachách. Jednota je tedy jediná entita, se kterou může ve hře hráč manipulovat a tvořit tak herní strategii.

3.1.1 Hrací pole

Hrací pole je tvořené dvourozměrnou mapou, která reprezentuje terén. Celá je rozdělena na dlaždice umístěné do čtvercové sítě. Každý objekt ve hře se nalézá na právě jedné dlaždici herního pole. Existují tři druhy dlaždic:

Země Země je jediný druh dlaždic, na kterém se mohou vyskytovat objekty. Můžeme se po ní pohybovat i střílet. Tento typ dlaždic se dále dělí na deset druhů podle ceny pohybu po nich. Na mapě je znázorněna zelenou barvou, a čím dražší je po dlaždici pohyb, tím je odstín zelené tmavší.

Voda Tento druh dlaždic slouží pouze jako překážka pro pohyb. Nemohou se na ní ani vyskytovat žádné objekty. Střelba je ovšem přes vodu možná a na mapě je znázorněna modrou barvou.

Zeď Zeď je překážka. Nelze přes ni ani střílet, ani chodit a ani se na ní nemohou vyskytovat žádné objekty. Na mapě je tento druh dlaždice znázorněn hnědou barvou.

Obrázek 3.1 ukazuje příklad hracího pole s jednotkami.

3.1.2 Typy jednotek

Zde si popíšeme typy objektů, se kterými se můžeme ve hře setkat.

Unit

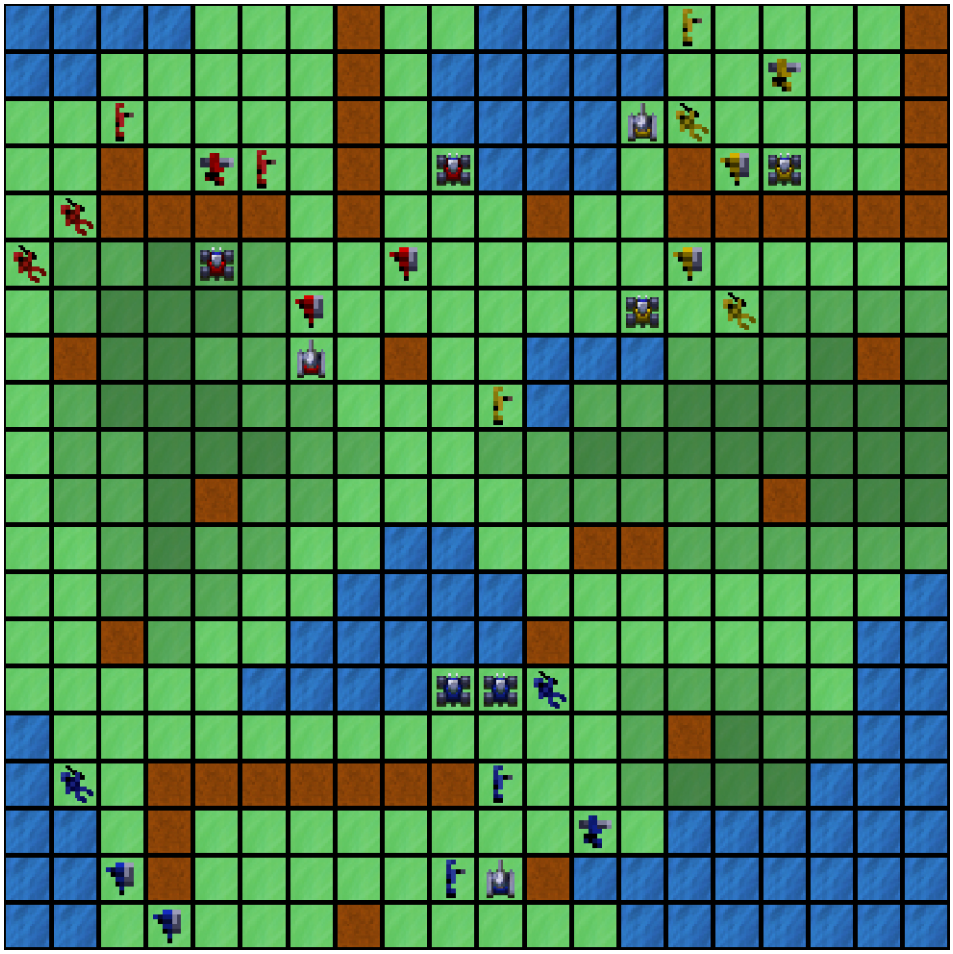
Prvním objektem je objekt typu *unit*, neboli jednotka. Je to objekt, který se používá ve hře k boji. Každá jednotka má svého majitele a ten ji ovládá.

Jednotka se může po mapě pohybovat nebo střílet na ostatní nepřátelské jednotky. Jednotky mají množství parametrů, kterými se liší:

Cena Počet bodů, který musí hráč zaplatit při zakoupení této jednotky.

Dostřel Počet políček, jak daleko jednotka dostřelí. Vzdálenost se počítá jako úhlopříčka čtyřúhelníku se svislými a vodorovnými stranami určeného dvěma body — útočící jednotkou a jednotkou, na kterou je útočeno.

Zdraví Počet bodů zdraví nám určuje kondici jednotky. Každý typ jednotky má různou počáteční hodnotu tohoto parametru a tato hodnota se zmenšuje s každým zásahem od nepřátelské jednotky. Když tento parametr klesne na nulu, jednotka je zničena a odstraněna z mapy.



Obrázek 3.1: Ukázka hracího pole s jednotkami

Štít Tento parametr určuje odolnost jednotky před zásahy nepřítelem. Čím větší je to číslo, tím je jednotka odolnější. Při zásahu nepřítelem se od celkového počtu bodů zdraví, který by měl jednotce ubrat, odečte právě tato hodnota, a až výsledný počet se skutečně odečte. Ovšem nejmenší počet bodů, který zásah může ubrat, je jeden bod. Tedy, i kdyby po odečtení štítu vyšlo záporné číslo, bude jednotce ubrán od zdraví jeden bod. Štít se v průběhu hry nijak nemění (s výjimkou změny nějakým bonusem, kdyby takový existoval).

Zbraň Parametr „zbraň" určuje, kolik by jednotka ubrala jiné jednotce bodů ze zdraví, kdyby ji zasáhla a zasažená jednotka by neměla žádný štít.

Cena pohybu Počet akčních bodů, který hráče stojí posunout jednotkou o jedno políčko, na kterém je pohyb nejlevnější. Tedy cena pohybu po prvním typu země.

Cena střelby Počet akčních bodů, který hráče stojí jeden výstřel s jednotkou. Cena je stálá bez ohledu na vzdálenost cíle.

Bonus

Dalším typem objektu je *bonus*. Je to speciální typ objektu, který nějakým způsobem ovlivní buď jednotku, nebo celého hráče. Aktivuje se vstoupením jednotky na políčko s bonusem.

Bonusy jsou statické, nemohou se pohybovat. Po aktivaci se uplatní jejich efekt a pak bonus zaniká. Ve hře je pouze jeden implementovaný bonus a to *First Aid Kit*, tedy lékárnička, která jednotce, jež ji aktivovala, přidá 50 bodů ke zdraví.

3.1.3 Akce ve hře

Ve hře je 6 možných akcí, které může hráč vykonávat. Tři jsou pro část hry, kde se kupují jednotky a tři pro bojovou část (části jsou podrobněji popsány v kapitole 3.1.4).

Nákup jednotek

Nákup První možná akce, kterou může hráč ve hře udělat je tato. Nakupují a umisťují se s ní jednotky, které si hráč vybírá z předem daného seznamu všech jednotek.

Prodej Další akcí je prodej jednotek. Tato akce prodá danou jednotku, tedy odstraní ji z mapy, a vrátí hráči počet bodů, odpovídající její hodnotě. Používá se v případě, že se v průběhu kupování jednotek hráč spletl nebo si rozmyslel strategii.

Konec nákupu Tato akce se použije, když má hráč již jednotky nakoupené, rozmístěné a připravené k boji. Po provedení této akce jde na

řadu s nákupem následující hráč. Až poslední hráč nakoupí jednotky, hra přechází do bojové části.

Bojová část

Pohyb Příkaz pohyb slouží k přemístění jednotky z jednoho místa na druhé.

Útok Tato akce se používá, když chce hráč zaútočit svojí jednotkou na nějakou nepřátelskou jednotku.

Konec tahu Poslední akce ukončuje hráčův tah. Když hráč již nechce dělat žádná další rozhodnutí, nebo na ně nemá dostatečný počet akčních bodů, použije tento příkaz.

3.1.4 Pravidla hry

Hra má poměrně jednoduchá pravidla. Jedná se o hru 2 až 9 hráčů, kde hraje každý proti každému. Celá hra se dělí na dvě části. První je výběr a nákup jednotek, druhá je pak samotná bojová část. Každá z těchto částí má jiná pravidla.

Nákup jednotek

Nákup jednotek probíhá jednoduše. Každá jednotka má svoji cenu a většinou lepší jednotka je dražší. Každý hráč na začátku dostane určitý počet bodů, za které si vybere jednotky. Hráči mají na mapě svá políčka, na která zakoupené jednotky umístí. Žádní dva hráči nesdílí stejné políčko, a umístění takových políček je součástí mapy a jsou na ni vázána. V průběhu nákupu jednotek vidí každý hráč políčka všech hráčů. Tedy kromě svých i ta, na která budou moci umístit své jednotky ostatní hráči, ale žádný hráč v této fázi nevidí, jaké jednotky kdo koupil a kam je umístil.

Bojová část

Tato část je hlavní částí celé platformy. Každý hráč má již své jednotky nakoupené a umístěné na hracím poli. Na hracím poli taktéž mohou být i bonusy. Celá hra se dělí na tahy, kdy v jednom tahu hraje pouze jeden hráč. Každý má veškeré informace o celém dění na hracím poli, včetně všech jednotek a jejich parametrů. Na začátku každého tahu dostane každý hráč

akční body, které může utratit v průběhu tahu za jednotlivé akce. Hráč může dělat pouze ty akce, na které má dostatek bodů.

Útočit může hráč pouze na nepřátelské jednotky. Vlastní jednotky tedy zničit nemůže. Při střelbě nesmí být v dráze žádná zeď, ani jakákoliv jiná jednotka. Vyhodnocení trajektorie střely probíhá pomocí spojnice útočníka a cíle, přičemž spojnice nesmí procházet skrz žádnou zeď, nebo jednotku. Není možné střílet ani mezi zdmi, které se dotýkají pouze rohem, protože dráha letu se vyhodnocuje jako spojitá plocha, kde se políčka trajektorie musí dotýkat celou stranou.

Pohyb je umožněn do čtyř stran – nahoru, dolu, doleva a doprava, přičemž není omezen pouze pohybem o jedno políčko, ale může jít i o několik políček najednou, kde výsledný stav je stejný, jako kdyby se pohyb složil z jednotlivých kroků. V případě, že je na cestě cizí jednotka, nelze jít přes ni — musí se obejít. Aplikace se sama stará o hledání nejkratší cesty mezi dvěma body přičemž respektuje daná pravidla. V případě, že je na cestě bonus, nebude pouhým průchodem aktivován. K jeho aktivaci je třeba se na něm zastavit.

Vyhrává hráč, který jako první zničí všechny jednotky všech ostatních hráčů.

Obecná pravidla

Žádný hráč nemůže vlastnit více jednotek, než kolik by si dokázal koupit. Dále se počet jednotek, které vlastní jeden hráč, dá omezit pomocí parametrů.

3.2 Parametry a nastavení platformy

Celá platforma se dá z velké části nastavovat pomocí dvou externích souborů s parametry a typy jednotek. Jedná se o soubory *parameters.ait* a *unit_types.ait*. Zde si popíšeme pouze rámcově, co je obsahem těchto souborů. Přesný popis formátu souboru je uveden v programátorské dokumentaci a uživatelské příručce.

3.2.1 parameters.ait

V tomto souboru jsou uloženy parametry platformy včetně parametrů všech algoritmů umělé inteligence a jsou automaticky načteny při spuštění

programu. Parametry jsou v souboru rozděleny do čtyř kategorií.

AI Tato kategorie se dělí ještě na další podkategorie. Podkategorii *GENERAL*, kde jsou parametry společné pro všechny algoritmy. Do této kategorie spadá například skupina parametrů ovlivňujících výpočet hodnocení jednotlivých herních stavů (viz. 5.1.2). Pak jsou zde podkategorie pro každý algoritmus, ale jejich obsah si popíšeme až u jednotlivých algoritmů v kapitole 5. Při vytvoření nového algoritmu je tedy třeba všechny parametry, kterými se má algoritmus konfigurovat, uvést do této kategorie. Platforma si je sama při spuštění načte.

GAME V této kategorii jsou parametry týkající se herního jádra. Je tu parametr omezující maximální počet hráčů, parametr omezující maximální počet jednotek, který může jeden hráč vlastnit. Poslední dva parametry určují počet bodů, který každý hráč dostane pro nákup jednotek, a nakonec počet akčních bodů pro jednotlivé tahy hry.

GUI Tyto parametry se týkají grafického rozhraní platformy a v současnosti jsou zde pouze dva parametry, a to šířka dlaždice a výška dlaždice, reprezentující rozměry jednoho políčka mapy v grafickém rozhraní.

NETWORK Poslední kategorie parametrů se týká nastavení síťového rozhraní. Do této kategorie zatím nezapadá žádný parametr a je zde z důvodu možnosti rozšíření, například o počet opakovaných pokusů pro připojení k serveru a tak dále.

3.2.2 unit_types.ait

Zde jsou uloženy všechny typy jednotek. Každá jednotka zde má definován svůj název a dále všechny své parametry, jako je cena, dostřel, zdraví, štít, zbraň, cena pohybu a cena střelby. Nakonec tu jsou parametry důležité pro grafické rozhraní. Název souboru s ikonou jednotky pro jednotlivé hráče ve formátu gif, a jako poslední zde je název souboru obsahující zvuk střelby ve formátu wav.

Přidání nebo změna jednotek je tedy velmi snadná. Stačí vložit další řádek do tohoto souboru, kde nadefinujeme název jednotky, její parametry a názvy souborů s ikonami a zvukem, vytvoříme ikony a platforma si při spuštění vše načte. Do zdrojového kódu tedy není nutné vůbec zasahovat.

3.2.3 Herní plán

Herní plán je také načítán z externích souborů. Zde jsou dva druhy těchto souborů. První nese příponu „.map" a obsahuje informace o terénu. Druhým souborem je soubor s příponou „.obj" a nese případné informace o rozmístění objektů na mapě. Pro hru je vyžadován pouze soubor s terénem. Soubor s objekty není povinný. Přesný formát těchto souborů je popsán v uživatelské a programátorské dokumentaci.

Kapitola 4

Architektura systému

Vzhledem k účelu platformy byl při jejím návrhu kladen největší důraz na její rozšiřitelnost a nastavitelnost.

K implementaci platformy jsme zvolili jazyk C++, a to hlavně pro relativně velké možnosti optimalizace, která je v této oblasti zásadní. Pro zjednodušení práce jsme si vybrali multiplatformní framework Qt, který je použit na implementaci grafického rozhraní, síťové rozhraní a vícevláknové implementace některých algoritmů. Nejvíce používaná část tohoto frameworku je mechanismus slotů a signálů. Toto řešení se nám velmi osvědčilo, protože lze tímto způsobem vytvářet neblokující aplikace a volání bez nutnosti práce s velkým množstvím vláken. Dále jsme použili knihovnu qwt, což je knihovna usnadňující kreslení grafů a různých statistik využívající taktéž frameworku Qt.

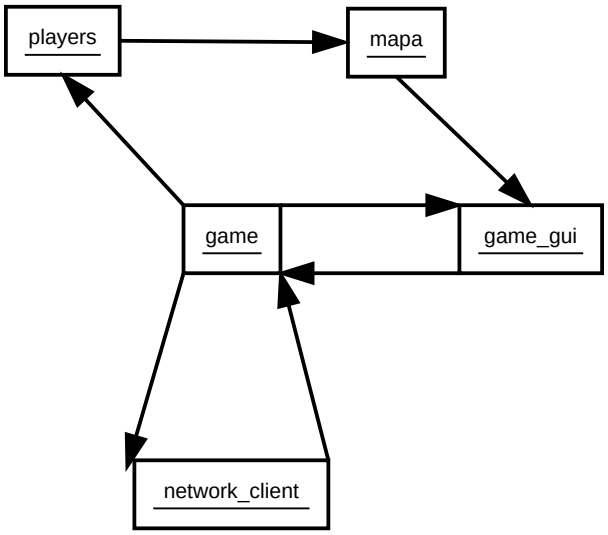
Celá platforma se skládá ze dvou částí. První je klient, který je samostatná jednotka, na které běží veškerá herní logika a sám o sobě je klient schopný testování algoritmů i hry umělé inteligence proti člověku a nakonec hry lidského hráče proti lidskému hráči. Druhou částí je server, který slouží pouze k posílání zpráv mezi klienty, a tím pádem umožňuje hru přes síťové rozhraní.

Nyní si popíšeme architekturu obou částí. Popisovat budeme pouze hlavní části a systém spíše z pohledu návrhu. Podrobný popis architektury je uveden v programátorské dokumentaci včetně diagramů UML. Zde uvedené diagramy nejsou kompletní a budou sloužit spíše jako doplněk textu pro jeho lepší názornost.

Hlavní částí platformy, kde je soustředěna všechna logika, je klient, který také popíšeme jako první.

4.1 Architektura klienta

Klient je rozdělen na několik komponent, kde každá komponenta je pro jednodušší komunikaci implementována jako singleton. Tento způsob implementace komponent se pak v průběhu vývoje systému velmi osvědčil a velmi zrychlil jeho vývoj. Jedná se o následující komponenty: *game*, což je hlavní jádro hry, starající se o koordinaci ostatních komponent, *players*, která reprezentuje hráče a jejich jednotky, *mapa*, která reprezentuje herní plán, tedy jsou v ní uloženy terén a objekty. Komponenta *game_gui* slouží pro zobrazování průběhu hry, ale v případě lidského hráče také k získávání příkazů pro jednotky a ovládání celé platformy. Poslední komponentou je *network_client*, což je komponenta starající se o spojení se serverem a posílání a přijímání zpráv. Tato komponenta je vytvořena a používána pouze při síťové hře. Na obrázku 4.1 je schematicky znázorněna celková architektura klienta.



Obrázek 4.1: Schéma propojení komponent klienta

Komunikace mezi komponentami *game* a *game_gui* probíhá pomocí zpráv. Pomocí zpráv funguje také spojení jádra hry s algoritmy, které jsou součástí hráčů obsažených v komponentě *players*. Celá komunikace probíhá následovně. Od uživatele, algoritmu nebo od serveru prostřednictvím komponenty *network_client* přijde zpráva. Tedy příkaz, co provést s jednotkou. Jádro hry příkaz zpracuje a dá povel dané jednotce, aby požadovanou akci provedla.

Toto probíhá již pomocí běžného volání funkcí *move(unsigned int x, unsigned int y)* a *fire(unsigned int x, unsigned int y)*, jak bude vidět dále. V případě úspěchu jednotka provede danou akci, což se projeví také na hracím plánu a celá akce se zobrazí zpět v grafickém rozhraní. Herní jádro akci uloží pro budoucí využití na analýzu hry a čeká na další příkaz. V případě, že se hraje přes síť, tak provedený příkaz ještě pošle komponentě *network_client*, aby ho přeposlala serveru, který ho následovně přepošle ostatním klientům.

Mimo těchto hlavních pěti komponent se používají ještě pomocné: *parameters*, kde jsou uloženy veškeré parametry platformy načítané z externího souboru, *prototypes*, což je komponenta obsahující prototypy všech jednotek a používá se v části hry, kde se nakupují jednotky a *ai_algorithms*, která obsahuje prototypy všech algoritmů umělé inteligence a používá se při nastavování hráčů před hrou.

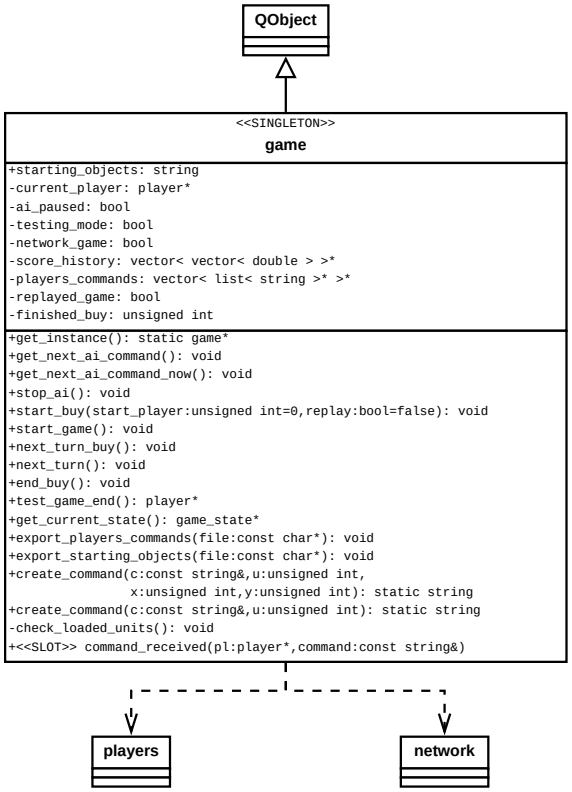
Nyní si podrobněji popíšeme jednotlivé komponenty.

4.1.1 game

Hlavní komponentou celé platformy starající se o koordinaci ostatních komponent je singleton *game*. Veškeré zprávy, pomocí kterých komunikují jednotlivé komponenty prochází právě tudy, proto je tato komponenta ideální také pro ukládání průběhu hry a počítání hodnocení jednotlivých tahů hráčů. V herním průmyslu se takový uložený průběh hry nazývá replay a používá se k opětovnému přehrání hry a její následné analýze. Vzhledem k tomu, že se jedná o tahovou strategii, může se hráč může v průběhu svého tahu dostat do zdánlivě nevýhodných pozic a přitom skončit na konci tahu velmi dobře, proto jsou ukládány pouze hodnocení stavu hry na koncích jednotlivých tahů. Ostatní hodnocení jsou v tomto smyslu irelevantní a mohly by být i zavádějící.

Strukturu komponenty znázorňuje diagram 4.2. Ukazatel *current_player* je ukazatel na aktuálního hráče reprezentovaného třídou *player*. Dále je zde několik proměnných indikujících stav platformy, tedy například zda je hra právě pozastavena či nikoliv (toto platí pouze pro hry s umělou inteligencí nebo přehrávané hry z „replayů"), pak zda je hra pouze lokální nebo síťová a tedy je třeba příkazy posílat i komponentě starající se o síťové spojení, a nakonec, zda jde o reálnou hru nebo pouze o testování algoritmu, a tedy zda se budou prováděné akce i zobrazovat v grafickém rozhraní či nikoliv. Mimo to hra udržuje aktuální herní stav pro předání algoritmům umělé inteligence a také pro výpočet hodnocení stavu pro uložení a následovnou grafickou

vizualizaci průběhu na konci hry.



Obrázek 4.2: Architektura komponenty game

Celá hra začíná příkazem *start_buy()*, kde se nastaví všechny počáteční hodnoty, inicializuje se grafické rozhraní a začne první fáze hry, což je nákup jednotek. Další řízení průběhu hry je již závislé na přijatých zprávách od algoritmů umělé inteligence, prostřednictvím grafického rozhraní od uživatele nebo zpráv přijatých ze síťového rozhraní. Veškeré přijímání a posílání zpráv je implementováno pomocí mechanismu slotů a signálů z frameworku Qt. Veškeré zprávy tedy odchytává slot *command_received(player* pl, const string& command)*, což znamená, že každá komponenta vysílající zprávy musí být na tento slot napojena. Dále pokud hra není replay, tak se příkaz uloží pro budoucí analýzu herní strategie, a pokud se jedná o síťovou hru, je přeposlán komponentě *network_client*.

Příkazů sloužících k ovládání hry je šest:

BUY u x y Tento příkaz je používán pro nákup jednotek v nakupovací části hry a má tři parametry: **u** je identifikační číslo jednotky k nákupu z komponenty *prototypes* a **x** a **y** jsou souřadnice, kam se zakoupená jednotka má umístit. Po přijetí tohoto příkazu je zavolána metoda *buy_unit(unit* u, unsigned int x, unsigned int y)* na aktuálním hráči, která jednotku koupí.

SEL u Příkaz používaný pro prodávání již nakoupených jednotek. Jediný parametr **u** určuje identifikační číslo jednotky, která se má prodat, ale nyní již ne v komponentě *prototypes*, ale v rámci jednotek aktuálního hráče. Přijetím tohoto příkazu je, obdobně jako v případě nákupu, zavolána funkce *sell_unit(unsigned int u)* na aktuálním hráči.

END : Tento příkaz je používán pro signalizaci, že hráč dokončil nakupovací fázi hry. Tím je zavolána funkce *end_buy()*, která inicializuje nakupovací fázi u následujícího hráče, nebo pokud již všichni hráči tuto fázi absolvovali, začne bojovou fázi příkazem *start_game()*, která opět provede veškeré inicializace včetně změny grafického rozhraní.

MOV u x y Pohyb jednotek je reprezentován tímto příkazem, který má tři parametry: **u** je identifikační číslo jednotky v rámci hráče a **x** a **y** jsou souřadnice, kam se jednotka má pohnout.

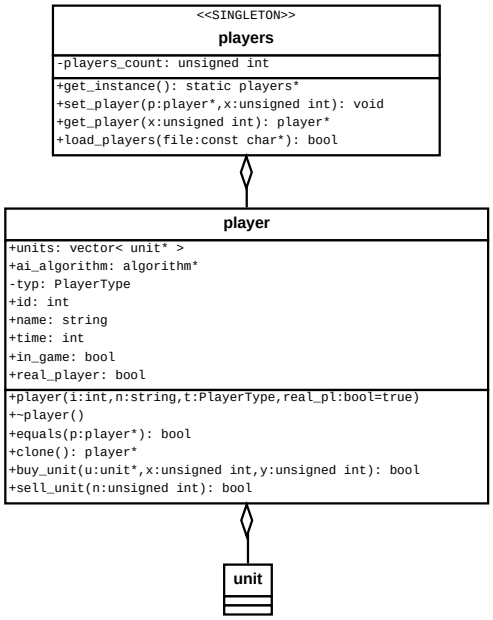
ATT u x y Toto je zpráva pro útok a má také tři parametry: **u** je opět identifikační číslo jednotky v rámci hráče, ale **x** a **y** jsou nyní souřadnice, kam má jednotka daná identifikačním číslem útočit.

NXT Tento příkaz signalizuje, že aktuální hráč ukončil svůj tah. Tím se zavolá funkce *next_turn()*, která inicializuje další tah pro následujícího hráče.

4.1.2 players

Tato komponenta pouze obsahuje pole s jednotlivými hráči a stará se o něj. Každý hráč obsahuje pole naplněné svými jednotkami, tedy objekty *unit*.

Schéma komponenty je zobrazeno na diagramu 4.3.

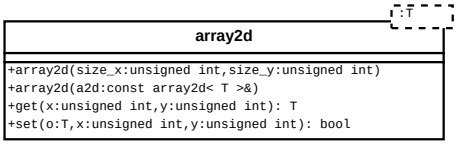


Obrázek 4.3: Architektura komponenty players

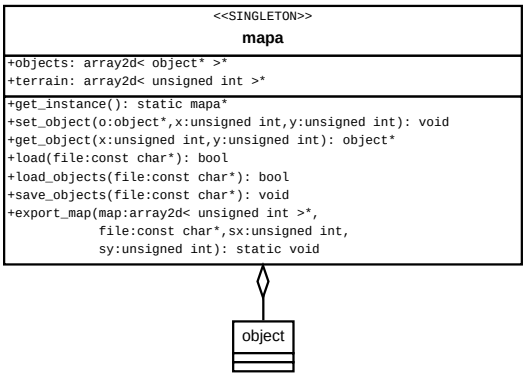
4.1.3 mapa

Zde je uložena reprezentace herního plánu. Ta je rozdělena na dvě části. První je reprezentace terénu a druhá je reprezentace umístění objektů na herním plánu. Obojí je realizováno pomocí dvourozměrného pole *array2d* (diagram 4.4). Tato komponenta také obsahuje funkce potřebné pro načítání dat do obou polí z externích souborů.

Diagram 4.5 zobrazuje architekturu komponenty *mapa*.



Obrázek 4.4: Architektura třídy array2d



Obrázek 4.5: Architektura komponenty mapa

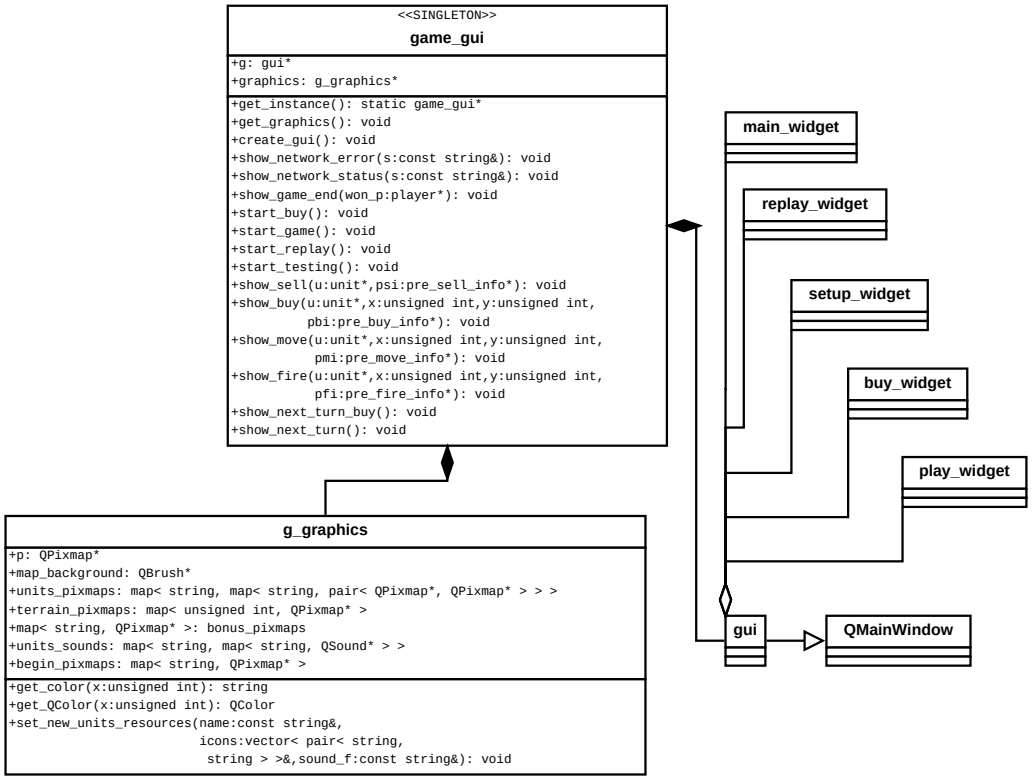
4.1.4 game_gui

Komponenta *game_gui* reprezentuje v této platformě celé grafické rozhraní. Poskytuje několik hlavních funkcí na ovládání grafického rozhraní, které většinou pouze přeposílají požadavek právě aktivní komponentě. Tím je zapouzdřena vlastní implementace celého grafického rozhraní, a to nám umožňuje jej jednoduše nahradit třeba rozhraním textovým, bez nutnosti zasahovat do kódu jádra aplikace. Sama obsahuje dvě podkomponenty, jak ukazuje diagram 4.6. Jedná se o podkomponenty *gui* a *g_graphics*, které si nyní popíšeme samostatně.

gui

Tato podkomponenta, jak je vidět na diagramu 4.6, přímo reprezentuje hlavní okno aplikace. Sama obsahuje ukazatele na jednotlivé widgety[1], aby mezi nimi bylo možné jednoduše přepínat v případě nastavování hry. Po inicializaci obsahuje hlavní okno pouze tlačítkové menu, pomocí kterého se určuje další směr použití platformy. Většina z následných widgetů není zajímavá, protože obvykle pouze konfigurují některé parametry hry, jako je třeba počet a typ hráčů, terén, objekty, jaké algoritmy se budou používat a tak podobně. Tyto widgety jsou podrobněji včetně diagramů popsány v programátorské dokumentaci. Zde si popíšeme pouze dva hlavní, a těmi jsou

[1]Widget v kontextu grafického rozhraní znamená ovládací prvek, případně jejich skupinu. V našem případě budeme hovořit o widgetech jako o části okna grafického rozhraní.



Obrázek 4.6: Architektura komponenty game_gui

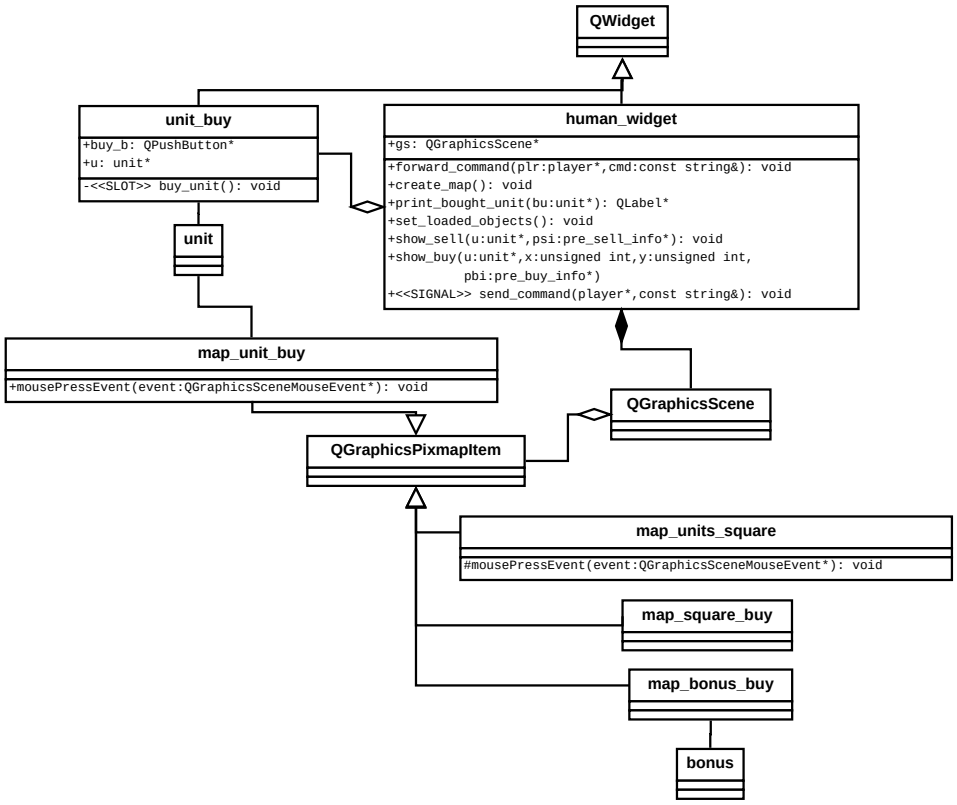
human_widget sloužící pro nákup jednotek uživatelem, a *play_widget*, sloužící pro vizualizaci bojové části hry.

human_widget

Hlavní částí tohoto widgetu je třída *QGraphicsView*, zobrazující třídu *QGraphicsScene* (obě z frameworku Qt), která reprezentuje celé hrací pole. Celá scéna je složena z objektů reprezentujících jednotlivé dlaždice a jednotky ve hře. Dlaždici, kam je možné umístit jednotku, reprezentuje třída *map_units_square* a při kliknutí myší generuje příkaz **BUY u x y**, dále je zde obyčejná dlaždice s terénem, *map_square_buy*, a dlaždice *map_bonus_buy*, která má v této fázi hry pouze vizualizační funkci. Zakoupená jednotka je reprezentována třídou *map_unit_buy* a při pravém kliknutí myší generuje příkaz **SEL u**. Jaká jednotka se bude kupovat je určeno polem widgetů *unit_buy*, které obsahuje seznam jednotlivých jednotek s výpisem jejich parametrů a

tlačítkem „Buy", které nastaví parametr **u** do případného příkazu **BUY u x y**.

Závislosti jednotlivých widgetů a entit znázorňuje diagram 4.7.



Obrázek 4.7: Architektura widgetu human_widget

play_widget

Podobně jako u třídy *human_widget*, i zde je hlavní částí *QGraphicsView* zobrazující *QGraphicsScene*, která reprezentuje celé hrací pole. Celá scéna je opět složena z objektů reprezentujících jednotlivé dlaždice a jednotky ve hře, ale nyní mají dlaždice jiný účel. Jsou zde tři třídy reprezentující objekty herního pole. První je *map_unit*, která reprezentuje objekt *unit*, tedy jednotku hráče. Po kliknutí levým tlačítkem myši se jednotka aktivuje a další akce budou spojeny s ní. Pravý klik na nepřátelskou jednotku (tedy opět na *map_unit*) se generuje příkaz **ATT u x y**, kde **u** je identifikační

číslo aktivované jednotky a **x** a **y** jsou souřadnice cíle. Pouhé najetí myší na tuto jednotku zobrazí indikátor, zda by byl případný útok možný. Dále pravé kliknutí myší na dlaždici *map_square*, reprezentující terén, generuje příkaz **MOV u x y**, kde **u** je opět id jednotky a **x** a **y** jsou souřadnice destinace. Obdobně jako u útoku, i zde pouhé najetí myší zobrazí nejkratší cestu do destinace s cenou pohybu reprezentovanou třídou *p*, pokud je takový pohyb možný. Levý klik na dlaždici *map_square* deaktivuje aktivní jednotku. Posledním typem dlaždic je *map_bonus*, který reprezentuje objekt *bonus* a má pouze vizualizační funkci stejně jako u widgetu *human_widget*.

Dále tento widget obsahuje sérii tlačítek pro zobrazení pomocných struktur herního stavu, které používají algoritmy umělé inteligence pro své rozhodování. Tyto vizualizace pak umožňují efektivnější analýzu jednotlivých algoritmů a jejich tahů.

Závislosti jednotlivých widgetů ilustruje diagram 4.8.

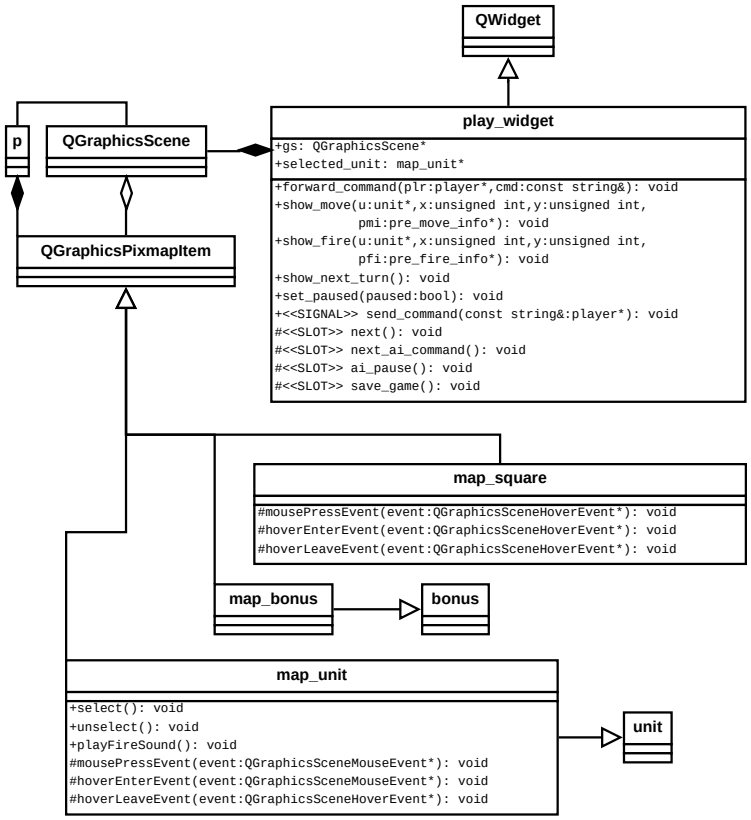
g_graphics

Tato podkomponenta obsahuje veškeré grafické a zvukové entity používané v platformě. Většina těchto entit je uspořádána do map indexovaných podle jejich použití. Například je zde mapa obsahující ikony všech jednotek pro každého hráče indexované jejich názvy.

Veškeré entity, kromě těch, které patří jednotkám, jsou načteny již při spuštění platformy. Entity související s jednotlivými jednotkami jsou definovány v konfiguračním souboru *unit_types.ait* a jsou načteny při načítání jednotlivých typů jednotek.

4.1.5 network_client

O celé síťové rozhraní a komunikaci se serverem se stará právě tato komponenta. Obsahuje třídu *Client*, která reprezentuje TCP Socket napojený na server. Klient komunikuje se serverem pomocí zpráv reprezentovaných obyčejným textovým řetězcem. Vzhledem k malému počtu typů zpráv a jejich jednoduchému obsahu je tento způsob dostačující. Samozřejmě by se zprávy daly reprezentovat pomocí nějaké třídy a jejích potomků s funkcí umožňující serializaci, ale v tomto případě se jedná o zbytečně komplikované řešení. Třída *Client* obsahuje metodu *receive_message()* implementovanou opět pomocí mechanismů Qt slotů a signálů, která parsuje příchozí zprávy a volá příslušné další akce. Zpráva se skládá ze dvou částí. Identifikace typu zprávy, pak oddělovač, kterým byl zvolen znak '=' a nakonec samotný obsah zprávy.



Obrázek 4.8: Architektura widgetu play_widget

Nyní si popíšeme všech osm typů zpráv rozdělených podle toho, zda mohou být přijaty nebo odeslány.

Přijímané zprávy

ACCEPTED Tato zpráva obsahuje pouze identifikační číslo klienta, které mu přiděluje server. Je to zpráva, kterou se potvrzuje přijetí klienta do síťové hry.

SET_PLAYER Toto je zpráva obsahující informace o ostatních klientech připojených do hry. Obsahuje identifikační číslo a jméno připojeného klienta. Jako odpověď na tuto zprávu klient odešle potvrzení ve tvaru „**OK=3**", jehož funkce je popsána dále.

MAP Zpráva obsahující informaci o terénu. Tedy mapu načítanou komponentou *mapa*. Formát obsahu zprávy je totožný s formátem souboru obsahující terén (přípona .map). Odpovědí na tuto zprávu je potvrzení ve tvaru „**OK=0**", jehož popis je taktéž dále.

OBJECTS Zpráva obsahující informaci o objektech rozmístěných po herním plánu. Její obsah je opět načítán komponentou *mapa* a obsah zprávy odpovídá formátem opět souboru s těmito informacemi (přípona .obj). Potvrzení přijetí této zprávy je podobné jako u mapy a je ve tvaru „**OK=1**".

START_BUY Tato zpráva obsahuje identifikační číslo hráče, který začne s nákupem jednotek, což je defaultně 0. Dále hra začne funkcí k tomu určenou v komponentě *game* a to *start_buy()*.

ACTION Touto zprávou již probíhá ovládání samotné hry. Může obsahovat jakýkoliv příkaz popsaný u komponenty *game* (4.1.1) a tento příkaz je této komponentě okamžitě přeposlán.

ERROR Tento typ zpráv obsahuje jakoukoliv chybovou hlášku informující o chybě nebo jiné události vzniklé na serveru. Obsah hlášky je jednoduše zobrazen pomocí funkce *show_network_error(const string& s)* v komponentě *game_gui*.

Odesílané zprávy

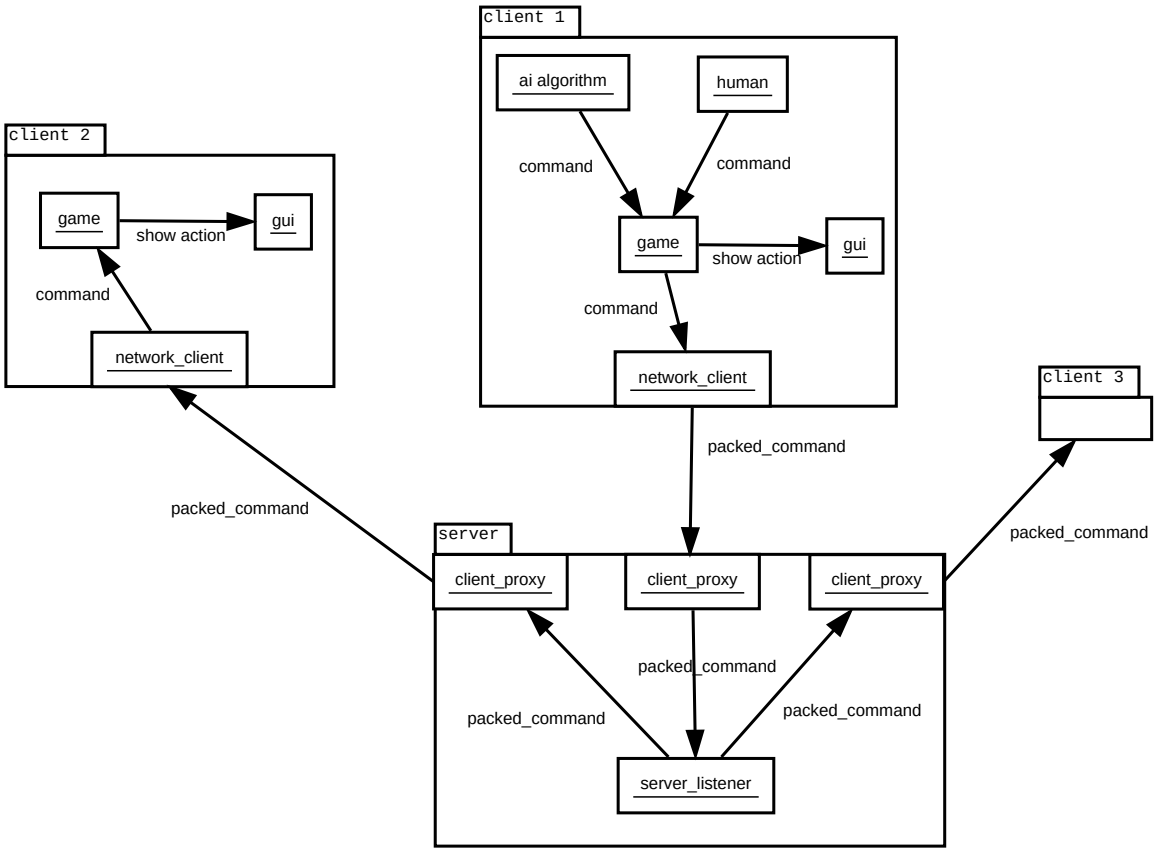
SET_PLAYER Obsah této zprávy je stejný jako u přijímané. Tuto zprávu odešle klient po připojení k serveru a registruje se tak u něj. Server si zprávy schovává a přeposílá ostatním klientům.

OK Tato zpráva obsahující potvrzení o přijetí informací o terénu, objektech a ostatních hráčích připojených ke hře. Její význam je popsán v sekci server_listener popisující fungování serveru (4.2.1).

ACTION Opět zpráva se stejným obsahem jako u přijímaných. Je generována komponentou *network_client* po provedení jakékoliv akce a přeposílána serveru, který ji pošle ostatním klientům. Při generování zprávy je do ní pouze přidáno identifikační číslo odesílajícího hráče a celkový tvar je pak následující: „**ACTION=**<*ID*>**:**<*COMMAND*>".

ERROR Pokud nastane na klientovi nějaká chyba, její textový popis je zakódován do této zprávy a odeslán serveru, kde je hláška zobrazena.

Schéma toku zpráv v aplikaci a přes síťové rozhraní je zobrazeno na diagramu 4.9. Zde je akce generována buď algoritmem umělé inteligence, nebo lidským hráčem.



Obrázek 4.9: Schéma toku zpráv při hře přes síťové rozhraní

4.1.6 parameters

Tato podpůrná komponenta má na starosti pouze jednoduchý přístup k parametrům platformy načítaným ze souboru *parameters.ait*, jehož struktura je popsána v sekci 3.2.1. Parametry jsou rozděleny do čtyř skupin.

Obecné parametry platformy, parametry grafického rozhraní, umělé inteligence a parametry síťového rozhraní.

Obecné parametry platformy jsou uloženy v mapě *game_parameters* a jsou indexovány názvem parametru. Hodnoty jsou zde pouze celočíselné.

Parametry grafického rozhraní mají své místo v mapě *gui_parameters* stejného typu, jako je mapa pro obecné parametry.

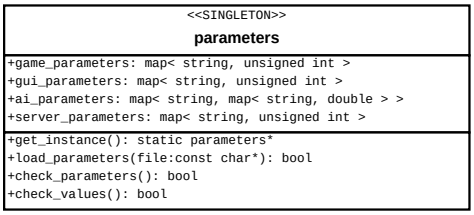
Umělá inteligence má mapu trochu odlišnou. Dále se dělí na obecné parametry umělé inteligence, které platí pro všechny algoritmy, a pak na parametry pro jednotlivé algoritmy. Proto je tato kategorie implementována mapou s klíčem „**GENERAL**" pro obecné parametry a názvem algoritmu pro parametry ostatní. Hodnotami v této mapě jsou opět mapy, kde je již klíč přímo název daného parametru a hodnota je tentokráte typu *double*.

Poslední kategorií jsou parametry síťového rozhraní, které nesou název *network_parameters*, a je to opět mapa indexovaná názvem parametru a hodnoty jsou zde opět celá čísla.

Dále tato komponenta obsahuje dvě kontrolní metody pro načtené parametry. První metodou je *check_parameters()*, která kontroluje pouze přítomnost důležitých parametrů v komponentě, a tedy potažmo v souboru, a druhou je *check_values()*, která kontroluje jejich hodnoty na povolené meze. Obě funkce jsou spouštěny po načtení parametrů ze souboru, které probíhá funkcí *load_parameters(const char *file)* ihned po spuštění aplikace a v případě výskytu chyby nedovolí aplikaci dalšího běhu.

Komponenta obsahuje konstantu *DEFINED_PLAYERS_RESOURCES*, která určuje maximální možný počet hráčů omezený počtem definovaných barev a ikon jednotek těchto hráčů. V současné době je platforma připravena na tři hráče.

Diagram s číslem 4.10 ukazuje architekturu této komponenty.



Obrázek 4.10: Architektura komponenty parameters

4.1.7 prototypes

Komponenta *prototypes* je další podpůrnou komponentou. Na rozdíl od komponenty *parameters*, která je používána v celém průběhu používání aplikace, tato je používána pouze v nakupovací fázi hry. Komponenta obsahuje prototypy všech objektů, které se mohou vyskytovat na herním plánu a z těchto objektů se klonují jednotlivé instance. Toto řešení je výhodné, protože je možné všechny objekty včetně parametrů konstruktoru definovat jednoduše na jednom místě. Obsahuje tedy dvě mapy, kde první, *units*, obsahuje prototypy jednotek a druhá, *bonuses*, prototypy bonusů. Obě mapy jsou indexovány názvy jednotlivých objektů.

Prototypy jednotek jsou načítány ze souboru *unit_types.ait*, jehož obsah je popsán v programátorské dokumentaci a také v sekci 3.2.2. Načítání vykonává funkce *load_types(const char *file)*.

Prototypy bonusů již takto jednoduše načteny být nemohou, protože každý bonus má jinou roli a tato role by se musela v souboru popisovat nějakým skriptovacím jazykem, což by bylo velmi složité. Proto se nový bonus vytváří jako potomek třídy *bonus* s reimplementovanou funkcí *effect(unit *u)*, která určuje, co bude daný bonus dělat. Prototyp bonusu se do mapy vkládá v konstruktoru a pro každý nový bonus je třeba ho sem přidat manuálně.

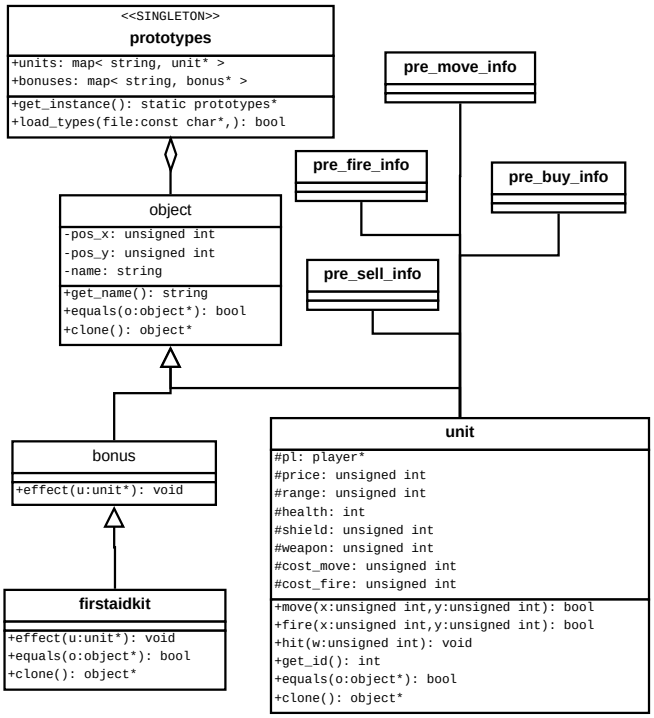
Architektura této komponenty a vztahy objektů jsou zobrazena na diagramu 4.11. Třídy *pre_move_info*, *pre_fire_info*, *pre_buy_info* a *pre_sell_info* jsou pomocné třídy pro grafické rozhraní, které mu dávají některé důležité informace o objektech před jejich změnou danou akcí.

4.1.8 ai_algorithms

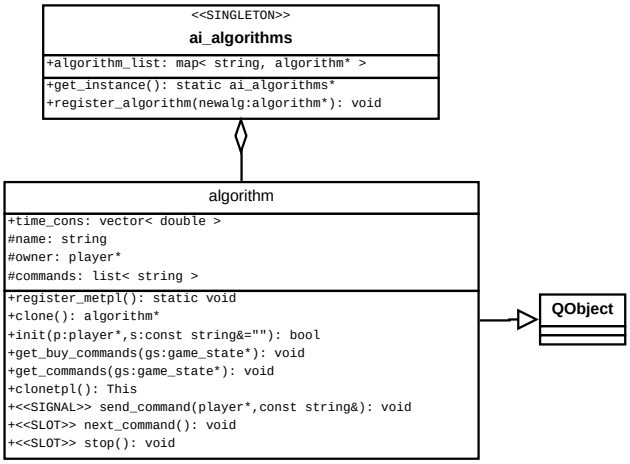
Tato komponenta má podobnou funkci jako *prototypes* a je navrhována hlavně s ohledem na budoucí rozšiřování platformy o nové algoritmy.

Komponenta obsahuje mapu s třídami *algorithm* indexovanou názvy jednotlivých algoritmů. Do mapy se algoritmus přidá zavoláním statické metody *register_me()* implementované v každém algoritmu. Tato metoda je volána v konstruktoru komponenty pro každý algoritmus. Tedy při implementaci nového algoritmu je třeba přidat i volání registrační funkce do tohoto konstruktoru.

Vnitřní architektura je ukázána na diagramu 4.12.



Obrázek 4.11: Architektura komponenty prototypes a hierarchie objektů



Obrázek 4.12: Architektura komponenty ai_algorithms

4.2 Architektura serveru

Server je, stejně jako klient, rozdělen na několik komponent, kde každá komponenta je implementována jako singleton. Protože je server poměrně jednoduchý a není v něm téměř žádná herní logika, vystačí si pouze s jednou komponentou pro logiku serveru a jednou pro grafické rozhraní. Jedná se o komponenty s názvy *server_listener*, což je hlavní komponenta, a stará se o celý síťový provoz, a *server_gui* sloužící pro zobrazování stavu spojení mezi klienty a nastavování serveru. Architektura této komponenty je vidět na obrázku 4.9.

4.2.1 server_listener

Jak již bylo napsáno, tato komponenta se stará o veškerý síťový provoz. Obsahuje třídu *Server*, která naslouchá příchozím spojením a obsluhuje je. Každý klient má přidělen svůj proxy objekt, který se stará o příchozí zprávy od něj. Tedy existuje tolik instancí objektu *client_proxy*, kolik je klientů.

Každá příchozí zpráva je opět obsluhována pomocí systému slotů a signálů. Zde se u příchozí zprávy zavolá slot *receive_message()* na proxy objektu, který ke zprávě přidá informaci o odesílateli a přepošle ji objektu *Server*, který ji již dekóduje a vykoná příslušné akce.

Zprávy přijímané a odesílané serverem jsou již popsány v sekci 4.1.5 u popisu architektury klienta a jeho síťového rozhraní.

Připojení klienta probíhá pomocí slotu *add_client()*. Pokud ještě není obsazen požadovaný počet klientů připojených do hry, je novému klientovi vytvořen proxy objekt a napojeny všechny sloty a signály na něj. Zpět se jako potvrzení pošle zpráva **ACCEPTED** s identifikačním číslem klienta, které mu server přidělil, pak zpráva **MAP** s informacemi o terénu a případně zpráva **OBJECTS** s informacemi o objektech na mapě. Klient mu jako odpověď pošle informace o sobě ve zprávě **SET_PLAYER**. Tyto zprávy si server uchovává a po připojení všech klientů jim je rozešle, aby každý věděl o každém.

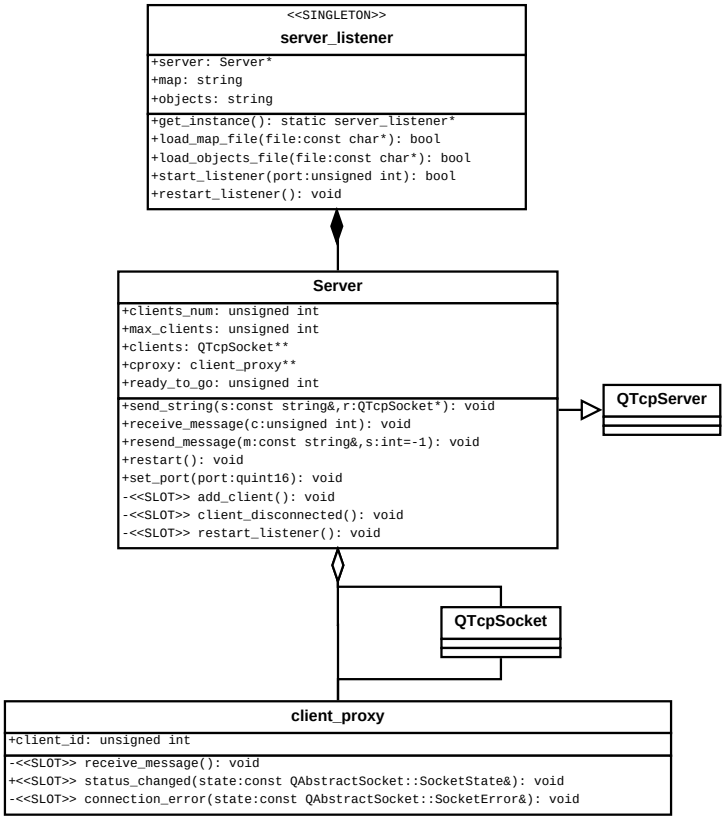
V případě, že je již maximální počet klientů, který má server obsloužit, připojen, server klientům, kteří by se chtěli připojit, pošle pouze chybovou hlášku a spojení uzavře.

Vyhodnocování, zda všichni klienti přijali všechny potřebné zprávy, probíhá pomocí zpráv **OK**, kde parametr **0** znamená přijatou mapu, parametr **1** přijaté objekty a parametr **3** přijaté informace za každého hráče. Server

tedy tyto zprávy počítá a podle toho vyhodnocuje, zda již bylo vše klienty přijato a hra může začít. To signalizuje rozesláním zprávy **START_BUY**.

V průběhu hry se server stará pouze o přeposílání zpráv **ACTION** ostatním klientům.

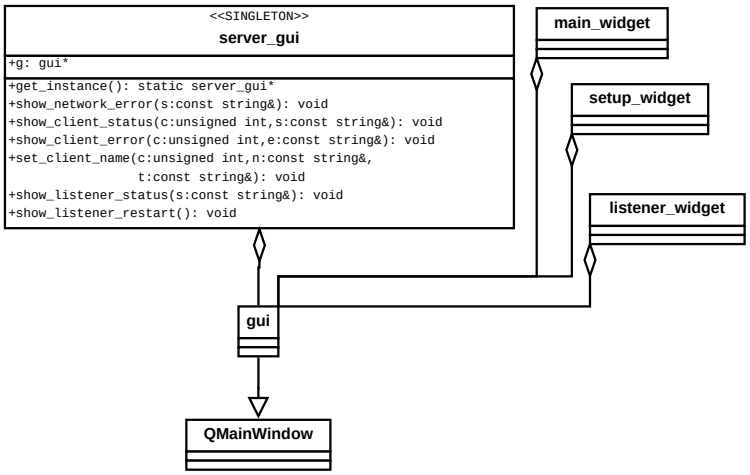
Celkovou architekturu této komponenty znázorňuje diagram 4.13.



Obrázek 4.13: Architektura komponenty server_listener

4.2.2 server_gui

Komponenta *server_gui* reprezentuje grafické prostředí serveru a stejně jako u klienta, i zde poskytuje několik hlavních funkcí na jeho ovládání. Sama obsahuje pouze jednu podkomponentu — *gui*. Celková architektura této komponenty je zobrazena na diagramu 4.14.



Obrázek 4.14: Architektura komponenty server_gui

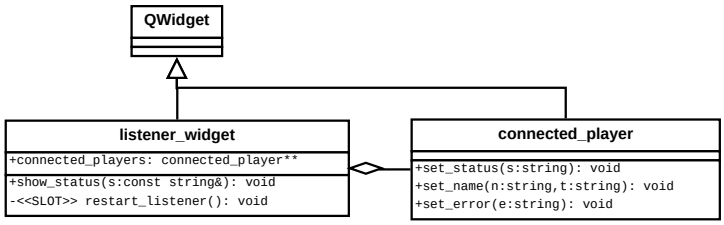
gui

Tato podkomponenta zastupuje hlavní okno aplikace. Sama obsahuje ukazatele na jednotlivé widgety, tedy architektura grafického rozhraní serveru je téměř stejná jako u klienta. Po inicializaci obsahuje hlavní okno taktéž pouze tlačítkové menu, kde si můžeme vybrat mezi zapnutím komponenty *server_listener* nebo zobrazením informací o aplikaci. Před spuštěním komponenty *server_listener* je třeba provést nějaká nastavení, jako je načtení map, zadání portu, na kterém má server poslouchat, a také určení požadovaného počtu hráčů, na které má server čekat. Toto se provádí ve widgetu s názvem *setup_widget*.

Po spuštění listeneru se zobrazí widget *listener_widget*, který zobrazuje informace o připojených klientech. Každý má svůj widget *connected_player*, který zobrazuje stav spojení a případné chybové hlášky. Celkovou architekturu tohoto informačního widgetu ukazuje diagram 4.15.

4.3 Možnosti rozšíření

Platforma se dá ještě rozšiřovat různými způsoby. Například způsob napojování algoritmů na platformu je nyní trochu nepohodlný, takže by bylo vhodné mít možnost algoritmy vyvíjet zvlášť bez zdrojových kódů celé platformy, a poté je do platformy vkládat jako moduly. Na tento způsob



Obrázek 4.15: Architektura komponenty widget_listener

rozšiřování je již platforma připravena a načítání modulů by bylo možné z frameworku Qt použít například *QPluginLoader*, který je k těmto účelům vhodný.

Kapitola 5

Algoritmy umělé inteligence

V této kapitole si popíšeme, jak se vytváří nový algoritmus pro testovací platformu, a pak také jednotlivé ukázkové algoritmy, které jsou v platformě již implementované.

Pro srovnání kvality jednotlivých algoritmů měříme dvě veličiny. První je rychlost rozhodování, která je vypisována v průběhu hry do konzolového výstupu a na konci hry je zobrazena přehledně graficky pro celý průběh hry. Druhou je kvalita tahů. Tedy hodnocení stavů na konci každého tahu. Celkový průběh hry může být taktéž zobrazen na konci hry.

Nejprve si popíšeme základní pojem, který je v aplikaci v souvislosti s umělou inteligencí používán. Jedná se o pojem herní stav.

5.1 Herní stav

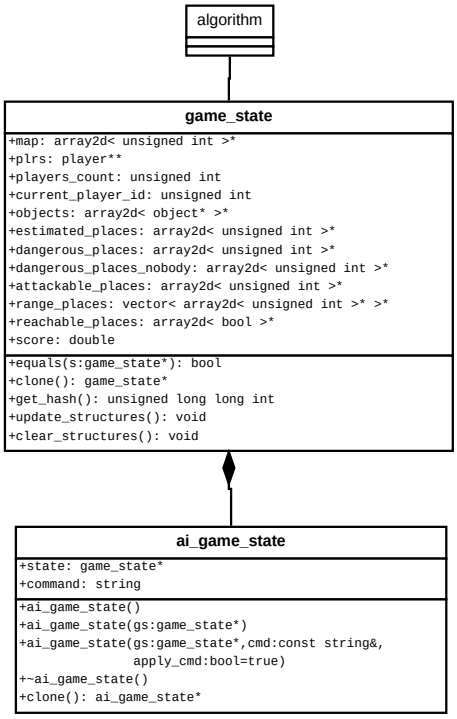
Herní stav je struktura, ve které jsou obsaženy veškeré informace určující stav hry v daném časovém okamžiku. Mimo tyto informace může ještě obsahovat pomocné mapy analyzující herní plán z různých pohledů a podle různých kritérií. Tyto mapy mají stejnou velikost jako původní herní plán. Herní stav tedy obsahuje celý herní plán s informacemi o terénu a objektech, seznam všech hráčů a jejich jednotek, číslo aktuálního hráče a také ohodnocení stavu vůči aktuálnímu hráči. K výpočtu ohodnocení herního stavu je potřeba několik pomocných map analyzujících herní stav.

Herní stav je také struktura, kterou dostane každý algoritmus ve svém tahu, a na jeho základě pak vygeneruje akce, které mají být v daném tahu provedeny. Tuto strukturu je také možné klonovat a v rámci algoritmu dále měnit a využívat, protože struktury v ní obsažené jsou pak zcela nezávislé na

zbytku platformy. V platformě je tento objekt pojmenován jako *game_state*.

Dále je poskytován objekt *ai_game_state*, který obsahuje normální herní stav a je ještě obohacen o informaci, jakým příkazem se do daného stavu došlo. Tato třída najde své využití hlavně pro potřeby stavění rozhodovacích stromů.

Architektura herního stavu je zobrazena na diagramu 5.1.



Obrázek 5.1: Architektura herního stavu

5.1.1 Pomocné struktury

estimated_places

Mapa hodnotící políčko z hlediska možnosti boje. Každé políčko obsahuje číslo vyjadřující počet jiných políček, ze kterých je možné na dané políčko dostřelit s dosahem uvedeným v souboru s parametry platformy. Uvažují se pouze políčka, na kterých by se mohla nějaká jednotka potenciálně vyskytovat. Zároveň ale také vyjadřuje, na kolik políček se dá z daného dostřelit,

protože když já mohu být zasažen z nějaké vzdálenosti a místa, tak také mohu se stejným dosahem — až na výjimky, jako je střelba přes roh zdi — na dané místo dostřelit. Tato mapa je tedy nezávislá na konkrétním umístění objektů na herním plánu a závisí pouze na terénu, konkrétně tedy pouze na překážkách v něm. Při odhadu se uvažují pouze políčka vyhodnocená jako dostupná přes výpočet mapy *reachable_places* popsané dále.

dangerous_places

Tato mapa znázorňuje reálnou nebezpečnost míst na mapě vzhledem k aktuálnímu hráči. Každé políčko tedy obsahuje číslo určující maximální zranění, které by mohla dostat od nepřátel jednotka jedním výstřelem, která by zde stála a neměla žádný štít.

dangerous_places_nobody

Zde je situace podobná jako u *dangerous_places* s tím rozdílem, že zde neuvažujeme existenci ostatních jednotek jako překážek. To má tu výhodu, že odhalíme nebezpečí i v případě, že by měl protivník několik jednotek, které by se vzájemně kryly a stačilo by s první uhnout, aby na nás měl dostřel.

attackable_places

Tato mapa hodnotí hru z ofenzivní stránky a na rozdíl od *dangerous_places*, zde každé políčko zobrazuje maximální zranění, které by mohla dostat nepřátelská jednotka od aktuálního hráče jedním výstřelem, která by zde stála a neměla žádný štít.

range_places

Toto není ve skutečnosti jedna mapa, ale celé pole map, kde jedna mapa patří jedné jednotce aktuálního hráče. Zde je mapa hodnocena opět z ofenzivního hlediska a jedno políčko vyjadřuje počet nepřátelských jednotek, které by z daného políčka bylo možné danou jednotkou zasáhnout. Tato mapa nám tedy ukazuje, kam se s jakou jednotkou musíme dostat, aby bylo možné zaútočit na nepřítele.

reachable_places

Tato mapa jen jednoduše vyjadřuje, jaká políčka jsou dosažitelná jakoukoliv jednotkou a jaká ne. Tato informace je využívána k vypočítání mapy *estimated_places*.

5.1.2 Ohodnocení herního stavu

Ohodnocení herního stavu se skládá ze tří složek. První složka hodnotí velikost armád – *army_balance*, druhá zranitelnost armády – *army_vulnerability* a třetí její umístění vůči ostatním armádám – *army_distances*. Každé složce se dá nastavit váha pomocí parametrů z mapy *ai_parameters* v kategorii „**GENERAL**". Parametr nastavující váhu složce *army_balance* nese název „**army_balance_coef**", pro složku *army_distances* se parametr jmenuje „**army_distances_coef**" a pro poslední složku, *army_vulnerability*, to je „**army_vulnerability_coef**". Výsledné ohodnocení je pak *army_balance*, od kterého se odečte složka *army_vulnerability* a *army_distances*, vše vynásobené koeficienty určující váhu složek. Pokud nastavíme nějakou váhu složky na nulu, nebude se tato složka vůbec počítat. To je výhodné například při přehrávání her, kde můžeme vypustit složku hodnotící vzdálenost armád, protože je její vliv marginální a pouze by zpomalovala výpočet. Nyní si popíšeme, co jednotlivé složky znamenají.

army_balance

Tato složka je založena na porovnávání předpokládané škody, kterou je aktuální hráč schopen nepřátelům způsobit a pak předpokládané průměrné výdrže armády nepřátel.

Předpokládaná škoda je počítána pomocí průměrně síly útoku a odhadovaného počtu zbývajících kol ve hře. Průměrná síla útoku zahrnuje i cenu výstřelu a je to součet tolika sil jednoho výstřelu, kolikrát je schopna daná jednotka v jednom tahu vystřelit. Odhadovaný počet zbývajících kol ve hře vychází z výdrže armády aktuálního hráče, a průměrné síly nepřátelských armád.

Předpokládaná výdrž armády je vypočítána jako součet výdrží jednotlivých jednotek. Výdrž zahrnuje aktuální zdraví a štíty všech jednotek.

Výsledná složka se spočítá jako rozdíl předpokládané škody, kterou je aktuální hráč schopen způsobit, a průměrné výdrže armády nepřátel.

Průměrné výdrže a síly nám zde dávají lepší odraz reality a reflektují aspekt, že ve hře hrají všichni proti všem. Ve výsledku nám tato složka bude i určovat, jakého nepřítele je vhodné ničit a tedy bude simulovat vytváření spojenectví mezi slabými armádami. Vezměme si příklad, kde proti sobě hrají dvě slabé a jedna silná armáda. Z pohledu slabé armády bude druhá slabá armáda snižovat průměrnou sílu a výdrž nepřátel. Pokud by tedy slabá armáda zničila druhou slabou, paradoxně si pohorší, protože nyní budou průměrné výdrže a síly nepřátel mnohem větší, protože zde zůstane pouze silná armáda jako jediný nepřítel. Proto použití této ohodnocovací funkce v prohledávacím algoritmu způsobí, že hra bude vypadat, jako kdyby se obě slabé armády spojily proti silné. Spojenectví bude trvat až do vyrovnání sil.

army_vulnerability

Touto složkou je vyjádřeno průměrné nebezpečí na jednotku, ve kterém celá armáda je. Využívá se zde pomocné struktury *dangerous_places* z herního stavu.

Průměr nám zde opět dává takovou vlastnost, že u velké armády jedna jednotka v nebezpečí nevadí, ale jak se armáda zmenšuje, tím více se snaží skrývat před nepřítelem.

army_distances

Poslední složka vyjadřuje průměrnou vzdálenost všech jednotek od místa, ze kterého může daná jednotka útočit na nepřítele. Zde je využíváno pomocné struktury *range_places* z herního stavu.

Tato složka je zde z důvodu, aby se armády snažili k sobě přibližovat a bojovat. Má za úkol pouze rozlišovat mezi dvěma stavy, kde jsou na tom armády stejně co do velikosti, a proto má na výsledné skóre pouze minimální vliv.

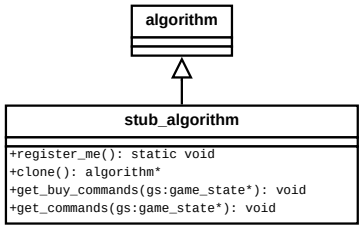
5.2 Vytváření nového algoritmu

Nový algoritmus je jednoduché vytvořit. Veškeré potřebné funkce a třídy jsou uvedeny v hlavičkovém souboru *ai.h*. Stačí pouze udělat potomka třídy *algorithm* a v něm implementovat dvě základní funkce: *get_buy_commands(const game_state* gs)* a *get_commands(const game_state* gs)*, kde každá funkce dostane jako svůj parametr aktuální herní stav. Vzhledem k takto

udělanému návrhu je možné aplikace psát také vícevláknově bez nějakých omezení. Takovým příkladem jsou algoritmy *Search algorithm MT* (viz. 5.5) a *Monte Carlo algorithm MT* (viz. 5.7). Algoritmy pomocí těchto funkcí generují příkazy do seznamu, který je součástí třídy *algorithm*. Z tohoto seznamu je pak zasílají hernímu jádru na jeho signál.

Uživatel má k implementaci algoritmu k dispozici řadu funkcí pro simulování jednotlivých tahů na fiktivním herním stavu a testování proveditelnosti těchto tahů, aniž by je musel skutečně provést. Všechny tyto funkce jsou uzavřeny do jmenného prostoru *ai_support_computations*.

Minimální kostra algoritmu pro jednoduché vytvoření nového se jmenuje *stub_algorithm*, jeho schéma zobrazuje diagram 5.2 a zdrojové kódy jsou v souborech *ai_stub_alg.h* a *ai_stub_alg.cpp*.



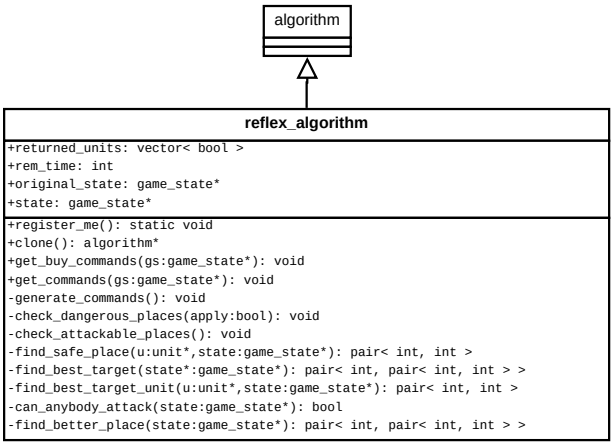
Obrázek 5.2: Architektura ukázkového algoritmu stub_algorithm

5.3 Reflex algorithm

Hlavní myšlenkou tohoto algoritmu je statická analýza herního stavu bez jakéhokoliv prohledávání stavového prostoru. To dává algoritmu poměrně velkou rychlost. Podle rozdělení metod řešení algoritmů z kapitoly 2.2 spadá tento do kategorie algoritmů založených na pravidlech.

Rozhodování je plně deterministické a algoritmus se tedy na stejných herních stavech rozhodne vždy stejně. Na rozdíl od algoritmů založených na prohledávání, tento je schopen začít hrát dříve, než dopočítá všechny akce až do konce svého tahu. Celkový čas strávený výpočtem tady není zcela relevantním údajem, ale aby bylo hodnocení algoritmů konzistentní, i zde měříme čas od obdržení herního stavu až po vygenerování všech akcí do konce tahu. Algoritmu tedy není dovoleno hrát dříve, než výpočet skončí.

Architektura datových struktur, které tento algoritmus využívá, je zobrazena na diagramu 5.3.



Obrázek 5.3: Architektura algoritmu Reflex

Celý běží v cyklech s tím, že každý cyklus má 3 fáze. Na konci cyklu se zjistí, zda algoritmus našel nějaké akce, a když ano, veškeré akce se aplikují na testovaný stav a algoritmus běží s takto upraveným stavem opět od fáze 1 znovu. V podstatě to znamená, že v ideálním případě algoritmus hledá do té doby, dokud má ještě nějaké body, aby mohl udělat nějakou smysluplnou akci.

5.3.1 Fáze 1

První fáze zkontroluje pro každou hráčovu jednotku, zda není v nebezpečí, tj. zda neexistuje nějaká protivníkova jednotka, která na ni má dostřel. Pro každou jednotku, která je v takovém nebezpečí, zkusí najít jiné, co nejbezpečnější, místo.

Bezpečnější místo se hledá ve čtverci určeném umístěním jednotky a nejdelší možnou vzdáleností, kterou by mohla ujít, kdyby byl terén rovný. Tedy zbývající body se vydělí cenou za pohyb přes jedno políčko. Tím máme zajištěno, že se tato část algoritmu nebude zhoršovat při rostoucí velikosti mapy.

Bezpečnost místa se vyhodnocuje podle počtu protivníkových jednotek, které na něj mají momentálně dostřel. Pokud jsou dvě místa pro danou jednotku stejně bezpečná, vybere si bližší z nich. Pro analýzu bezpečnosti daného políčka je využívána tabulka z pomocných map ze stavu s názvem *dangerous_places_nobody*.

Nalezené akce jsou používány pouze pro odhad počtu bodů potřebných pro přesun všech jednotek do bezpečí. Se stejným stavem, ale zmenšeným počtem bodů právě o tento odhad se spouští fáze 2.

5.3.2 Fáze 2

Druhá fáze je již ofenzivní. Nejprve se zkontroluje, zda nemůžeme s nějakou jednotkou útočit, a podle toho, zda můžeme či nemůžeme, se tato fáze dělí na další 2 fáze. Fáze 2.1 nastává, když útočit můžeme, fáze 2.2, když nemůžeme. Testování, zda útočit můžeme nebo ne, se provádí opět pomocí pomocných struktur uložených v každém stavu. V tomto případě se využívá tabulka *attackable_places*.

Akce nalezené v této fázi jsou již aplikovány na testovaný stav, a až pak algoritmus přechází do fáze 3.

Fáze 2.1

Když teď víme, že existuje alespoň jedna jednotka, která může někam útočit (*attackable_places* nám dává pouze existenční informaci, ale již ne konkrétně o kterou jednotku se jedná), tak opět najdeme v nějakém smyslu nejideálnější dvojici naší jednotky a cíle. Hledání dvojice funguje tak, že se pro každou naši jednotku najdou všechny protivníkovy jednotky, na které má dostřel, a vybrána je ta s nejméně životy. Ze všech takovýchto dvojic se opět vybere ta, kde má protivník nejméně životů, a tím pádem velkou šanci na zničení.

Fáze 2.2

Pokud neexistuje žádná jednotka, na kterou můžeme útočit, opět pro každou jednotku najdeme nějaký optimální cíl. Ten se hledá pomocí pomocné mapy *range_places*. Hledá se nejbližší místo, ze kterého můžeme někam útočit, a pokud najdeme takových míst více, vybereme z nich to, odkud se může útočit na nejvíce protivníkových jednotek. Po nalezení takového místa musíme najít, kam se přesuneme, abychom mu byli blíže, protože místo může být tak vzdálené, že není možné se na něj v jednom tahu dostat. Hledání takového místa je postupné. Algoritmus se podívá vždy o jedno políčko vedle do všech čtyř směrů a najde takové, které je cílovému nejblíže. Nakonec vybere takovou jednotku, která je schopna se přiblížit k nepříteli co nejlevněji.

Pokud je políčko, na které se má přesunout, v dosahu nepřítele, odečítá se za něj dvojnásobek času, což nám dává rezervu na návrat zpět z „nepřátelského území". Spolu s fází 1 nakonec víme, že se jednotka dokáže se zbylým časem jistě vrátit zpět na bezpečné místo, pokud takové fáze 1 našla nebo pokud na počátku jednotka v nebezpečí nebyla. Samozřejmě, že bezpečné místo se může nacházet blíže a tím pádem může být rezerva na návrat zpět zbytečně velká a vedla by k plýtvání akčními body. Proto běží celý algoritmus v cyklech.

Tato fáze skýtá ještě nebezpečí, že jedna jednotka spotřebuje veškeré body na opakovaném pokusu dostat se k nepříteli. Může se totiž stát, že se bude jednotka přibližovat stále přes „nepřátelské území" a na konci zjistí, že vystřelit nemůže a vrátí se. Ovšem v dalším cyklu může zůstat dostatek bodů, aby to jednotka zkusila znovu, a tím spotřebovala zbylé body, proto je každá jednotka, která se jednou vrátí, vyloučena z dalších pohybů jiných, než krycích do bezpečí. Tím pádem dáváme v dalším cyklu šanci jiným jednotkám na pohyb.

5.3.3 Fáze 3

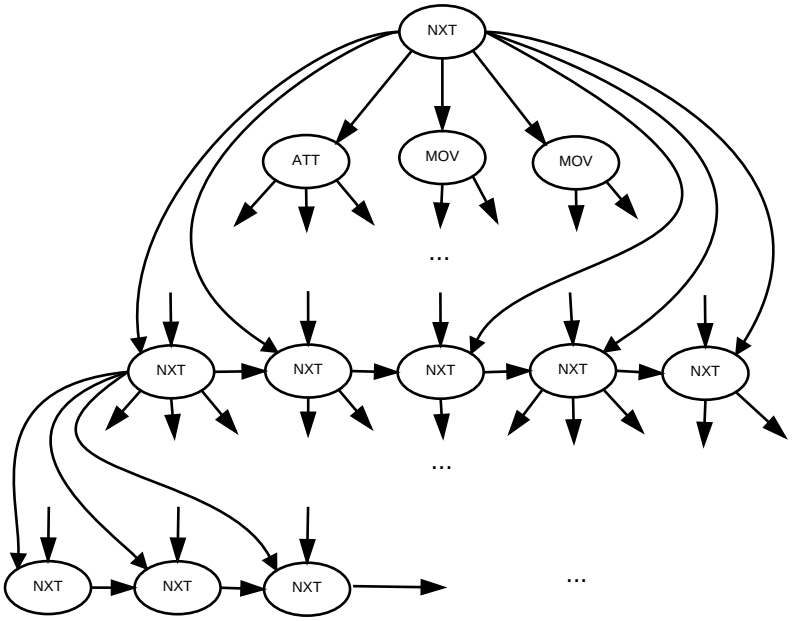
Před vstupem do této fáze se ke zbývajícím bodům na tah přičtou zpět body získané odhadem ve fázi 1. Poté proběhnou stejné mechanismy, jako ve fázi 1, ale nalezené akce se již nepoužívají pouze pro odhad, ale tentokrát se skutečně uloží do seznamu akcí algoritmu a tím pádem dostaneme jednotky před koncem tahu do bezpečí.

5.4 Search algorithm

Tento algoritmus je založen na prohledávání stavového prostoru. Prohledávací algoritmy jsou podrobně rozebrány v knížce *Artificial Intelligence: A Modern Approach* [13] v kapitole 3.4 od stránky 73. Algoritmus prohledává celý stavový prostor do hloubky, která je nastavena v parametrech. Číslo hloubky v podstatě znamená počet odehraných tahů. Hloubka 1 tedy označuje prohledání stavů do konce tahu aktuálního hráče, hloubka 2 pak prohledává do konce tahu jeho následníka a tak dále.

Na začátku tedy vezmeme zkoumaný stav a uděláme z něj vrchol stromu (třída *ai_states_node*), kde synové budou vždy stavy, do kterých se dá dostat z otce pomocí jedné akce. V každém uzlu je také uložena akce, kterou byl uzel ze svého otce vytvořen. Potom postupně expandujeme rekurzivně

jednotlivé stavy. Pokud se dostaneme do stavu, kde končí tah, propojím ho s jeho posledním otcem, který je na konci tahu (*level_childern* a *level_parent*), což poznáme podle toho, že nese příkaz určující konec tahu. A dále tento vrchol vložíme do spojového seznamu obsahující všechny konce tahů v dané hloubce (*state_levels*). Struktura takového stromu s hloubkou prohledávání 2 je znázorněna na obrázku 5.4. Této struktury pak využíváme při závěrečném hledání ideální sekvence tahů.



Obrázek 5.4: Struktura prohledávacího stromu stavového prostoru

Takto popsané prohledávání ovšem generuje velké množství duplicitních stavů, které celé prohledávání velmi zpomalují. Proto jsme navrhli hashovací funkci na jednotlivé stavy, pomocí které zjišťujeme, zda jsme daný stav již neprohledávali. Pro rychlejší provedení této kontroly si ukazatele na jednotlivé stavy ukládáme ještě do spojového seznamu (*searched_states_list*), který před prohledáním nového stavu projedeme a zkontrolujeme, zda se zde stav již nevyskytuje.

Po prohledání všech stavů do dané hloubky začneme procházet strom od předposlední vrstvy. Zajímají nás pouze vrcholy, kde končí tah. V každé vrstvě ohodnotíme vrcholy tak, že předpokládáme optimální strategii protihráčů. Hodnota stavu bude odpovídat hodnotě jednoho z jeho synů, který

má nejvyšší hodnocení vůči hráči, který v něm hrál, ale tuto hodnotu přepočítáme vůči hráči, který se právě rozhoduje, jaké tahy udělá. Budeme pak vybírat takové stavy, které jsou pro daného hráče nejméně nevýhodné — protihráč na nich může získat nejmenší skóre — což odpovídá optimalitě strategií protihráčů. Až tímto způsobem dojdeme do vrstvy, která nás zajímá, tedy do vrstvy, kde končí náš tah, vybereme ze všech možných zakončení takové, které pro nás bude nejlepší. Nejlepší zakončení se vyhodnocuje stejným způsobem, jako zde bylo popsáno. Vybereme tedy takové zakončení, které nám v hloubce, do které jsme stavový prostor prohledali, zajistí co možná nejlepší výsledek. Od takto nalezeného stavu již jednoduše projdeme všechny vrcholy až k začátku tahu, přičemž si ukládáme příkazy, pomocí kterých jsme se do něj dostali.

Takto navržený algoritmus dokáže hrát téměř optimálně na velmi malých mapách s malým množstvím jednotek, kde je možné stavový prostor prohledat do dostatečné hloubky. Pokusy dokázaly, že již hloubka 2 je dostačující na rozumnou hru. Výsledek algoritmu ovšem velmi záleží na dobré ohodnocovací funkci. Tu používáme ze standardního setu funkcí ze jmenného prostoru *ai_support_computations*.

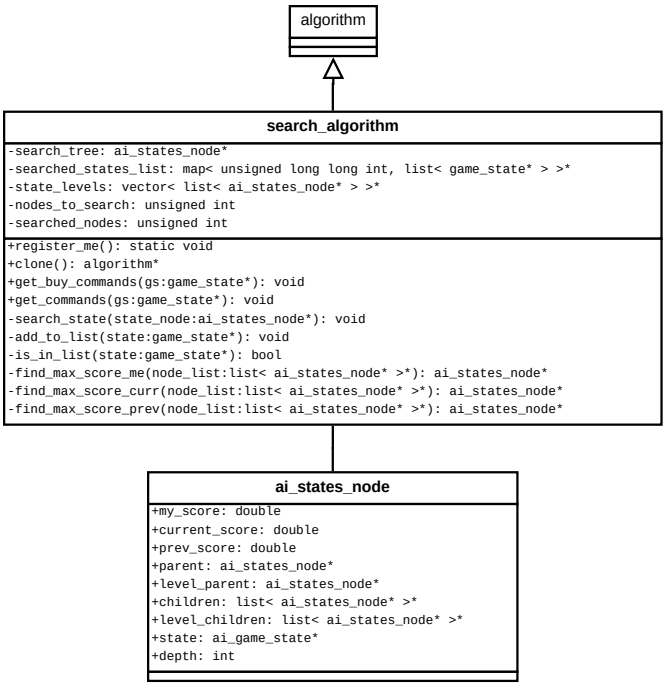
I přes vyřazování duplicitních stavů je stavový prostor hry tak velký, že je na běžných mapách velikosti například 30 x 30 políček, zcela nepoužitelný a vhodný spíše na experimentální účely a pro hledání různých prořezávání a heuristik.

Tento algoritmus má v parametrech sekci „**SEARCH**", kde se nastavuje parametr „**depth**", což je hloubka prohledávání.

Architektura algoritmu je vidět na diagramu 5.5.

5.5 Search algorithm MT

Search algorithm MT je v podstatě pouze vícevláknová mutace algoritmu Search. Abychom dosáhli bezpečnosti vícevláknového přístupu ke sdíleným proměnným, museli jsme použít velké množství zámků. Práce s nimi byla ovšem nepohodlná, tak jsme implementovali třídu *QRWLocker*, která při vytvoření sama zamkne příslušný zámek předaný jako parametr konstruktoru a při zániku jej odemkne. Když budeme tuto třídu používat v každé funkci, kde je potřeba přístup ke sdílené proměnné tak, že ji vytvoříme na stack, nemůže se nám pak stát nepříjemné opomenutí zamknutých zámků, protože při ukončení funkce se nám při zániku této třídy zámek automaticky odemkne.



Obrázek 5.5: Architektura algoritmu Search

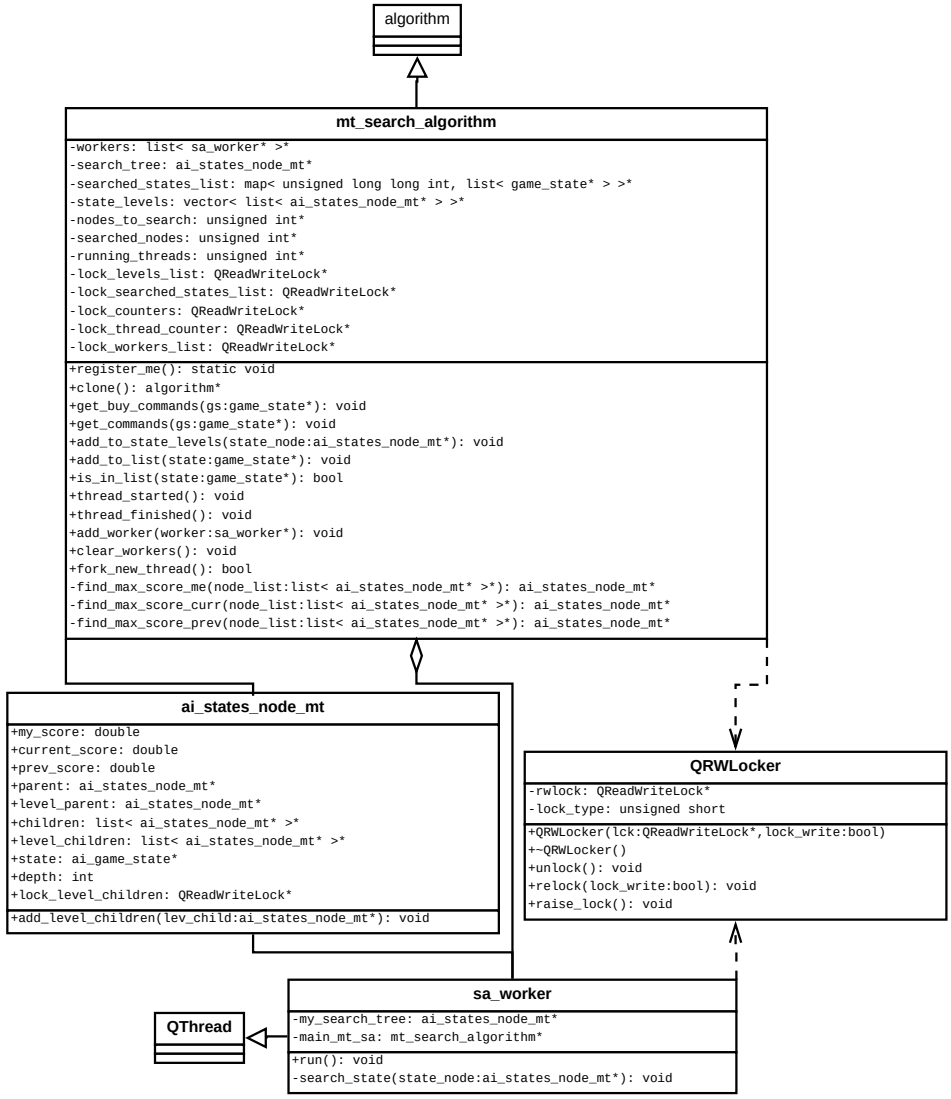
Algoritmus na začátku s pomocí frameworku Qt vyhodnotí optimální počet vláken pro daný počítač a tohoto počtu se dále drží. Celý výpočet probíhá v třídě *sa_worker*, která dostane uzel (*ai_states_node_mt*) pro expandování. Jakmile dojde výpočet do fáze, že by se měl expandovat další uzel, zkontroluje se, zda běží dostatečné množství vláken, a když ne, uzel je předán nové třídě *sa_worker*. Pokud dostatečné množství vláken běží, je uzel expandován uvnitř stávajícího vlákna. Jakmile dané vlákno prohledá svůj podstrom do požadované hloubky, zaniká a uvolňuje místo novým vláknům. Výpočet končí po zániku všech vláken.

Tento algoritmus je na vícejádrových počítačích za standardních podmínek o poznání rychlejší, než jeho jednovláknová verze, ale stále je na větších mapách nepoužitelný. Na druhé straně je nevýhodou tohoto algoritmu poměrně velká režie s vlákny a častý přístup ke sdíleným proměnným jej zpomaluje. Proto jsme měřeními zjistili, že na mapách o velikosti 5 x 5 s dvěma jednotkami je díky této režii a častému přístupu ke sdíleným proměnným, dokonce pomalejší, než jeho jednovláknová verze. Takto malé mapy jsou

ovšem extrémním případem, a v běžné hře se nevyskytují.

Sekce v parametrech náležící tomuto algoritmu nese jméno „**SEARCH_MT**". Podobně, jako u jednovláknové verze, i zde je parametr „**depth**", omezující hloubku prohledávání.

Architektura této verze algoritmu je zobrazena na diagramu 5.6.



Obrázek 5.6: Architektura algoritmu Search MT

5.6 Monte Carlo algorithm

Monte Carlo algoritmus je další algoritmus založený na prohledávání stavového prostoru. Jak jsme již zjistili, stavový prostor je tak velký, že je nereálné ho prohledávat celý. Proto jsme se inspirovali metodou s názvem Monte Carlo Planning.

5.6.1 Monte Carlo plánování

Monte Carlo plánování je metoda založená na náhodném prohledávání. Podrobněji se touto oblastí zabývá článek *Monte Carlo Planning in RTS Games* [30]. Monte Carlo plánování funguje tak, že zvolíme počet náhodných plánů vygenerovaných během výběru a pak počet testů jednoho plánu proti náhodným plánům protivníka. Simulace začíná vygenerováním náhodného plánu pro daného hráče. Poté ho otestuje daným počtem opět náhodně vygenerovaných plánů pro protivníka. Původní plán ohodnotí pesimisticky. Tedy ze všech testů vybere ten, který pro něj dopadl nejhůře. Takto generuje další plány a nakonec z nich vybere ten s největším hodnocením.

Je tedy vidět, že celá metoda je závislá na existenci kvalitní ohodnocovací funkce a dále na dobrém generátoru náhodných plánů. Pokud by generátor produkoval plány, které jsou si velmi podobné, celá metoda by neměla smysl. Dále se často pod pojmem plán skrývá spíše nějaká obecnější strategie, než přímo sled akcí. Díky tomuto zobecnění je generátor méně náchylný na vytváření podobných plánů, ale vzniká nám tím problém s definováním obecných strategií.

5.6.2 Implementace

Tento algoritmus využívá hlavní myšlenku z Monte Carlo plánování a kombinuje jí s myšlenkou Search algoritmu. Generování stavů probíhá tak, že do hloubky 1, tedy do konce tahu prvního hráče, se generují jednotlivé plány stejným způsobem jako u Search algoritmu s tím, že je každý stav expandován maximálně do šířky definované v parametrech. To odpovídá generování různých plánů v Monte Carlo plánování. Ve větších hloubkách již generujeme do jiné šířky, definované opět v parametrech, a to odpovídá testování vygenerovaného plánu. Výběr stavů funguje následovně. Vypočítáme hodnocení stavu jako jeho skutečné hodnocení umocněné preferenčním koeficientem z parametrů, což ve výsledku způsobuje větší preferenci hodně

dobrých stavů a znevýhodnění hodně špatných stavů. Dále je hodnota vynásobena náhodně vygenerovaným číslem z intervalu od hodnoty zadané v parametrech po 1. To nám určuje míru náhodnosti výběru. Pak takto spočítané hodnocení ještě vynásobíme koeficientem znevýhodňujícím hodnocení stavu, kde končí tah. To vychází z myšlenky, že v průběhu tahu se hráč může dostat do nevýhodných situací, ale na konci tahu by měl skončit co nejlépe, tedy kritéria pro výběr stavu na konci tahu by měla být přísnější. Z takto ohodnocených stavů vybereme určitý počet nejlepších a ty dále expandujeme. Původní funkce ohodnocující stav je zde tedy brána jako heuristická funkce, která určuje, který stav by se měl expandovat, takže výběr stavů zde není zcela náhodný, jako u originálního Monte Carlo plánování.

Stavba stromu a konečný způsob výběru akcí je totožný s výběrem v Search algoritmu.

Nevýhoda tohoto řešení je, že musíme hodnocení stavu počítat pro každý uzel, takže je tento algoritmus velmi náchylný na rychlost spočítání ohodnocovací funkce. Na ni jsou tedy kladeny ještě větší nároky, než u běžného prohledávání, kde se počítalo hodnocení pouze u stavů, kde končil tah.

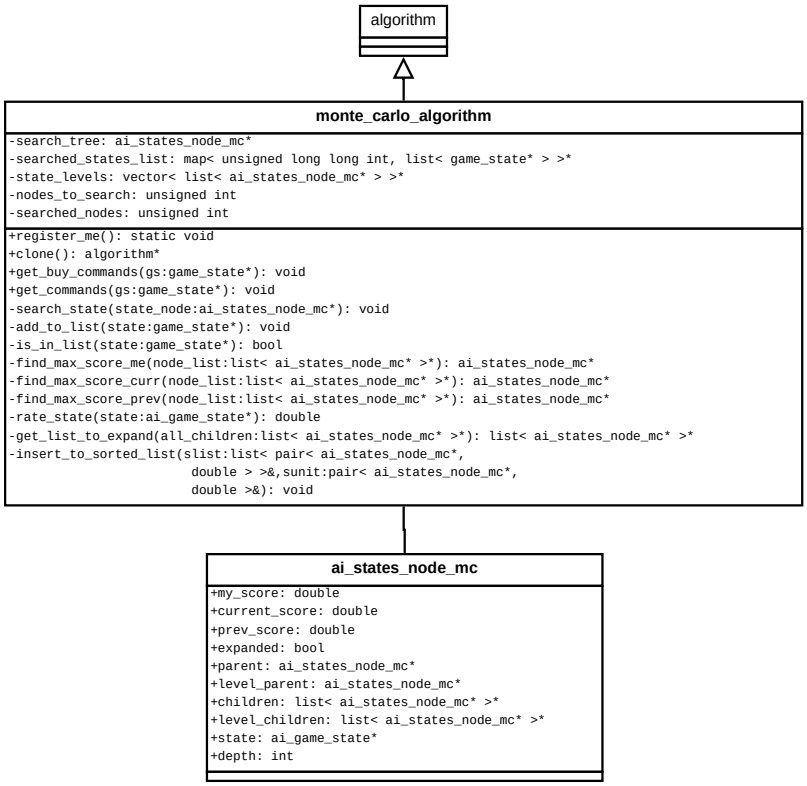
Jak již bylo napsáno, tento algoritmus se dá velmi přizpůsobovat pomocí parametrů. Ty jsou uloženy v sekci „**MONTE_CARLO**". Parametr omezující hloubku prohledávání nese jméno „**depth**", počet expandovaných dětí každého stavu do hloubky 1 je uložen ve „**width_plans**", ve větších hloubkách je to „**width**". Koeficient znevýhodňující stavy, kde končí tah, se jmenuje „**nxt_coef**", koeficient preferující dobré stavy a znevýhodňující špatné je „**pref_coef**" a nakonec koeficient omezující náhodnost výběru stavu se jmenuje „**rand_coef**".

Architekturu tohoto algoritmu ukazuje diagram 5.7.

Bohužel stavový prostor je tak obrovský, že ani tento algoritmus není na větších mapách moc použitelný, i když je to o mnoho lepší, než u obyčejného prohledávání.

5.7 Monte Carlo algorithm MT

Podobně jako Search algoritmus, i tento má svoji vícevláknovou mutaci. Pro dosažení této mutace bylo použito stejných technik, jako u Search MT algoritmu. Zde má ale tato mutace větší efekt v tom, že zde expanze stavu trvá o poznání déle kvůli výběru stavů pro expanzi, a tedy nutnosti počítání ohodnocovací funkce pro každý stav. Proto není přístup ke sdíleným proměnným tak častý a algoritmus jimi není tolik limitován.



Obrázek 5.7: Architektura algoritmu Monte Carlo

Parametry, které tento algoritmus využívá, jsou stejné, jako u jednovláknové verze. Zde jsou uloženy v sekci „**MONTE_CARLO_MT**".

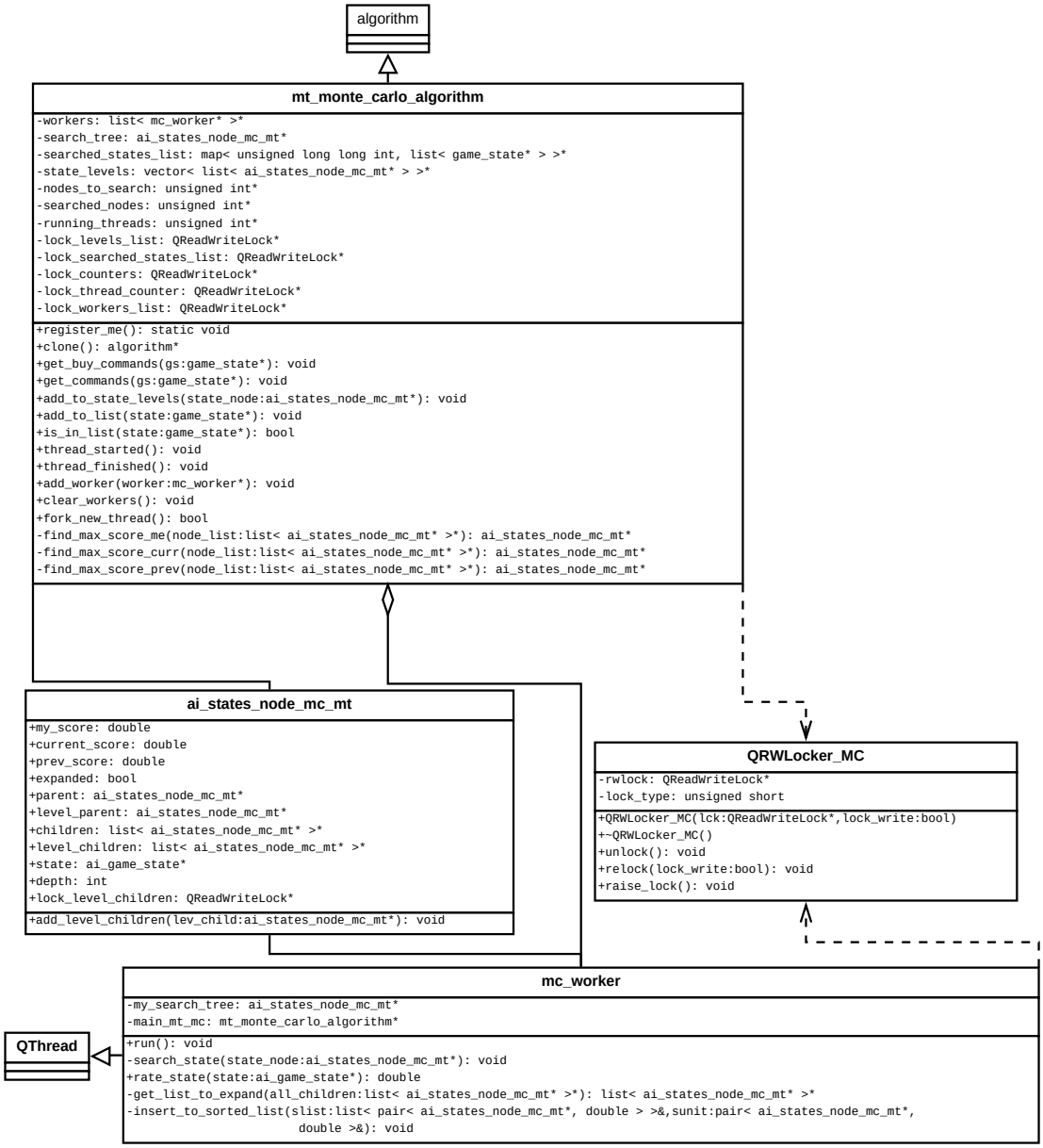
Architektura této vícevláknové verze je zachycena na diagramu 5.8.

5.8 Porovnání a shrnutí

Jako nejlepší algoritmus se ukázal Reflex algorithm, který dokazuje, že dobře navržená sada pravidel může hrát lépe, než prohledávací algoritmus, kde není stavový prostor možné prohledat do dostatečné hloubky.

Srovnání algoritmů bylo počítáno pro pevné nastavení jejich parametrů, které jsou popsány v tabulce 5.1.

Rychlost algoritmů je u prohledávacích algoritmů velmi závislá na aktuálním stavu a možnostech jeho větvení. Jednotlivé časy jsme měřili na mapě



Obrázek 5.8: Architektura algoritmu Monte Carlo MT

velikosti 10 x 10 políček s třemi jednotkami pro každého hráče. Search MT algoritmus má průměrnou dobu rozhodování na takové mapě okolo 55 minut.

Název algoritmu	Název parametru	Hodnota parametru
Obecné parametry	army_balance_coef	1
	army_vulnerability_coef	20
	army_distances_coef	0.1
	test_range	25
Reflex algorithm	žádné parametry	
Search algorithm	depth	1
Search algorithm MT	depth	1
Monte Carlo algorithm	depth	1
	width	2
	width_plans	5
	nxt_coef	0.9
	pref_coef	3
	rand_coef	0.9
Monte Carlo algorithm MT	depth	1
	width	2
	width_plans	5
	nxt_coef	0.9
	pref_coef	3
	rand_coef	0.9

Tabulka 5.1: Nastavení parametrů jednotlivých algoritmů pro testování

Expanduje takto přes 200 000 herních stavů. Monte Carlo MT algoritmus je na tom v tomto případě lépe a rozhodnutí mu trvá okolo 20 minut. Prohledá takto desítky tisíc herních stavů. Výhoda tohoto algoritmu je, že i na větších mapách se doba rozhodování díky omezenému počtu stavů pro expanzi, příliš nezvětší. Reflex algoritmus je na této mapě tak rychlý, že je jeho doba rozhodování na hranici měřitelnosti a rozhodne se do jedné vteřiny.

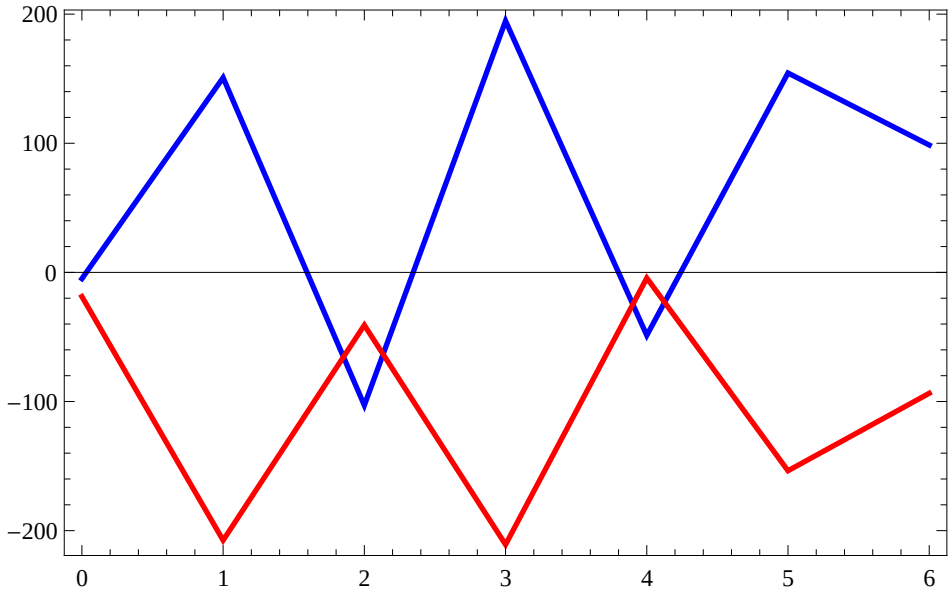
Pro lepší názornost uvedeme několik grafů ukazujících průběh hry na malé mapě ze souboru *10x10_2_small.map*. Objekty byly použity vždy ze souboru *10x10_2_small.obj*. Seznam těchto her je uveden v tabulce 5.2, kde jsou i uvedeny názvy jednotlivých souborů pro opětovné přehrání hry. Hru začíná vždy modrý, následuje červený. V grafu není znázorněno hodnocení posledního tahu, protože zde by toto hodnocení dávalo obrovské číslo, které by graf znehodnotilo. Proto může na grafu na konci mít lepší hodnocení hráč, který nakonec prohrál.

Graf číslo 5.9 ukazuje průběh zápasu číslo 1, kde hrál algoritmus Search MT proti algoritmu Monte Carlo MT. Je zde vidět výhoda začínajícího hráče a velká vyrovnanost obou algoritmů. Až do konce nemá ani jeden z nich velkou převahu. V grafu číslo 5.10 vidíme hru číslo 2 algoritmu Monte Carlo

MT proti algoritmu Reflex. Tento graf ukazuje, že dobře navržený systém pravidel, který je v algoritmu Reflex, může konkurovat i prohledávacím algoritmům. Hra číslo 3 dvou algoritmů Reflex je znázorněna na grafu číslo 5.11. Je zde vidět, že již od začátku má modrý malou převahu díky výhodě prvního tahu. Nakonec také vyhraje.

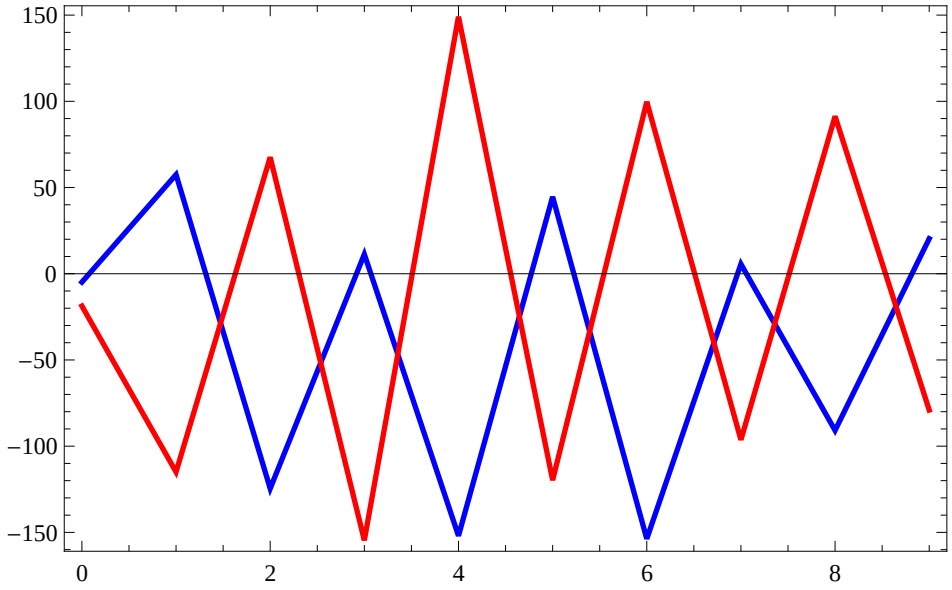
	1. algoritmus	**2. algoritmus**	**Replay**	**Vítěz**
1	Search MT	Monte Carlo MT	10x10_small_SE-MC	1
2	Monte Carlo MT	Reflex	10x10_small_MC-RE	1
3	Reflex	Reflex	10x10_small_MC-RE	1

Tabulka 5.2: Srovnání algoritmů na mapě 10x10_2_small.map s objekty 10x10_2_small.obj



Obrázek 5.9: Zápas algoritmu Search MT proti algoritmu Monte Carlo MT na malé mapě

Dalším grafem průběhu je graf 5.12 ukazující průběh hry tří algoritmů Reflex. Hru opět začíná modry, následuje červený a nakonec žlutý. Skok uprostřed grafu je způsoben vyřazením červeného hráče ze hry. Replay je



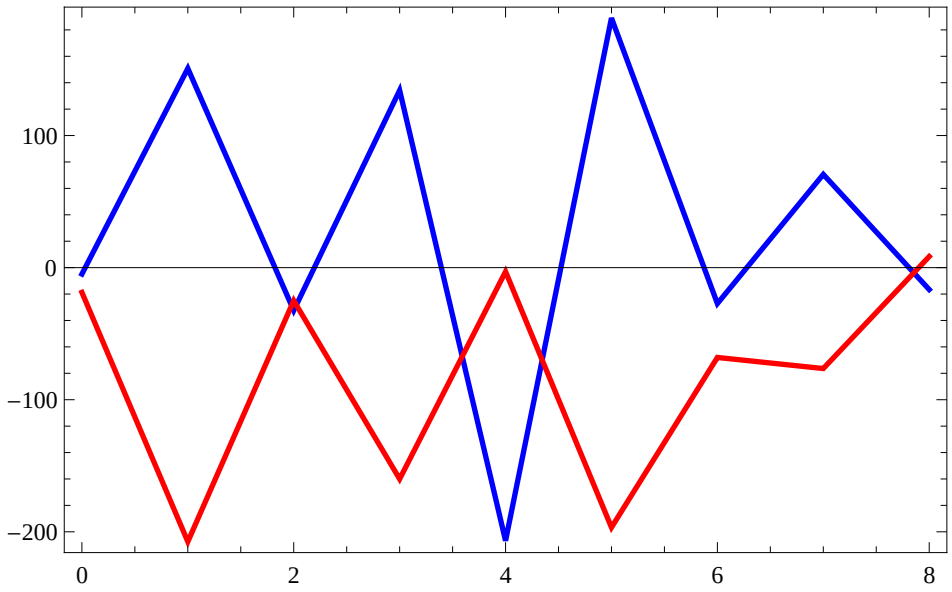
Obrázek 5.10: Zápas algoritmu Monte Carlo MT proti algoritmu Reflex na malé mapě

opět možné přehrát pomocí souborů mapy *20x20_3_hillslakes.map*, objektů *20x20_3_hillslakes.obj* a příkazů *20x20_hillslakes_RX-RX-RX.rpl*.

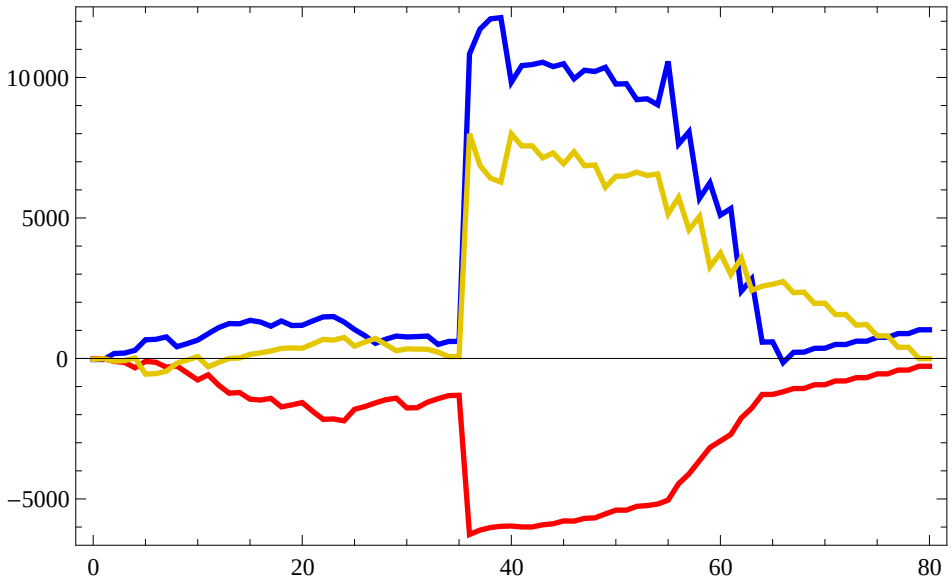
Poslední hra ukazuje rychlost rozhodování algoritmu Reflex v sekundách. Hrají zde dva proti sobě na větší mapě. Pro přehrání replaye jsou potřeba soubory *32x23_2_wide.map*, *32x23_2_wide.obj* a *32x23_wide_RX-RX.rpl*. Průběh hry ukazuje graf číslo 5.13 a doby rozhodování graf 5.14. Zde je vidět, jak čas potřebný k rozhodnutí s ubývajícími jednotkami klesá. Průměrný čas se tedy pohybuje okolo pěti sekund. Vygenerování prvního příkazu trvá ale opět méně, než sekundu.

5.9 Možnosti rozšíření

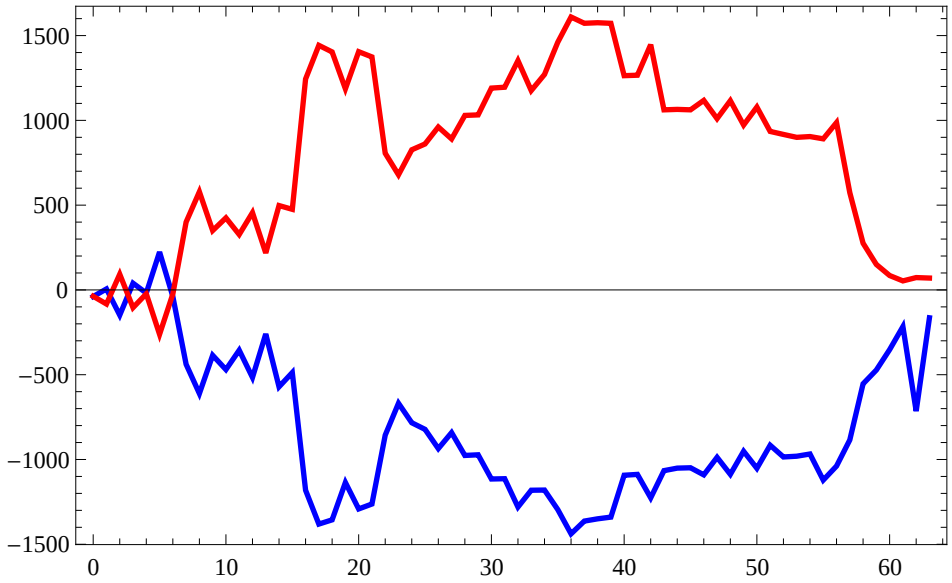
Dále by bylo možné rozšiřovat algoritmus Reflex o predikci soupeřových tahů. V současné době je jeho největší slabina právě tato oblast. Toho by se dalo docílit prohledáváním do hloubky jedné akce u každé jednotky. Takové prohledávání by znamenalo pouze malý nárust doby rozhodování vzhledem k velmi malému počtu stavů, který by se takto vygeneroval.



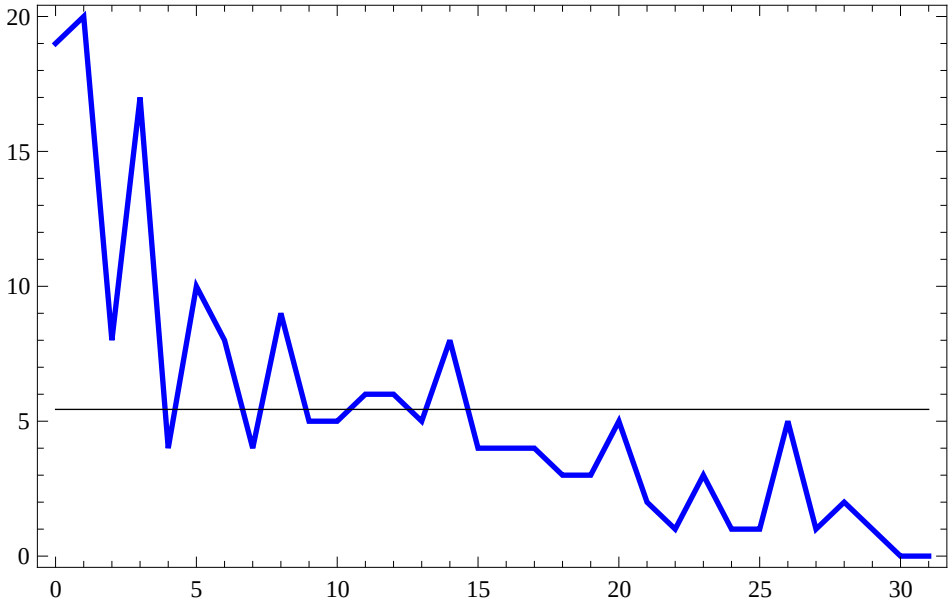
Obrázek 5.11: Zápas algoritmu Reflex proti algoritmu Reflex na malé mapě



Obrázek 5.12: Zápas tří algoritmů Reflex na střední mapě



Obrázek 5.13: Zápas dvou algoritmů Reflex na větší mapě



Obrázek 5.14: Doba rozhodování algoritmu Reflex na větší mapě

Kapitola 6

Závěr

6.1 Testovací platforma

K vývoji aplikace se nám velmi osvědčil framework Qt, který umožňuje vyvíjet dobré okenní aplikace, má podporu vláken i sítí, takže svými možnostmi plně dostačoval vývoji celé platformy. Jednou z hlavních výhod této platformy je již zmiňovaný systém slotů a signálů. Vzhledem k velké rozšířenosti tohoto frameworku v praxi, má velmi dobrou podporu a dokumentaci, takže jsme většinu svých problémů rychle vyřešili.

V této práci jsme navrhli jednoduchá pravidla tahové strategické hry, která jsme pak implementovali do podoby testovací platformy na algoritmy umělé inteligence. Platforma je velmi dobře konfigurovatelná, a tak pouhou změnou parametrů jednotek můžeme docílit zcela jiného charakteru hry.

Do platformy lze snadno implementovat další algoritmy, které se mezi sebou dají pak porovnávat a testovat jejich výhody v různých situacích.

Program poskytuje funkce pro analýzu strategií různých algoritmů a hráčů. Je možné si hry přehrávat znovu, sledovat vývoj hry a vývoj časů potřebných pro rozhodnutí jednotlivých algoritmů. Tato data lze také exportovat do textových souborů pro zpracování jinými programy a statistickými nástroji.

Jako vedlejší funkce zde je i možnost hry přes síťové rozhraní s pomocí serveru, který jsme též implementovali.

6.2 Algoritmy

Do vzniklé platformy jsme implementovali celkem 5 algoritmů umělé inteligence. Reflex algorithm, založený na systému pravidel, Search algorithm s jeho vícevláknovou verzí založenou na prohledávání stavového prostoru hry a Monte Carlo algorithm inspirovaný metodou Monte Carlo Planning, který je taktéž ve vícevlákové verzi. Ukázalo se, že stavový prostor hry je tak veliký, že klasické prohledávací algoritmy zde nejsou moc použitelné, protože prohledávání stavového prostoru trvá příliš dlouho. Vygenerování akcí pro jeden tah trvá na běžných mapách v řádu hodin. Pouze algoritmus inspirovaný metodou Monte Carlo Planning vykazuje lepší výkonnostní výsledky. Při nastavení stejné hloubky prohledávání vyhrál častěji algoritmus Search. Výhodou algoritmu Monte Carlo je, že je schopen za stejnou dobu prohledat stavový prostor do větší hloubky. Proto když srovnáváme délky běhů algoritmů, tak zde algoritmus Monte Carlo jednoznačně vítězí a větší hloubka prohledávání převažuje to, že algoritmus neprohledává stavový prostor tak do šířky.

Jednoznačně nejlepším algoritmem, co se rychlosti rozhodování týče, je algoritmus Reflex, který je schopen svůj první krok vygenerovat na velkých mapách v řádu jednotek vteřin. To je již přijatelná doba i pro hraní proti uživateli.

Naimplementované algoritmy se na při hře chovají rozumně. Zajímavé záznamy z her jsou k dispozici na přehrání na přiloženém CD. Hra člověka proti algoritmu Reflex je na členitějších mapách docela vyzývavá, protože se zde ukáží přednosti tohoto algoritmu a jeho schopnosti dobře analyzovat herní plán.

Seznam obrázků

Seznam tabulek

Literatura

[1] Mařík V., Štěpánková O., Lažanský J. a kol.: *Umělá inteligence 1*, Academia, 1993.

[2] M. Buckland: *AI Techniques for Game Programming*, Premier Press, 2002.

[3] F. Rose: *Into the Heart of the Mind*, Harper and Row, 1984.

[4] Minsky M.: *Computation. Finite and Infinite Machines*, Prentice-Hall, 1967.

[5] Rich E., Knight K.: *Artificial Intelligence - Second Edition*, McGraw Hill, Inc., 1991.

[6] Kotek Z., Mařík V., Zdráhal Z.: *Metody rozpoznávání a umělá inteligence*, ČSVTS FE VŠSE, 1983.

[7] A. J. Champandard: *AI Game Development*, New Riders, 2003.

[8] Skupina autorů, Moby Games — Warcraft universe [online], aktualizováno 19. 5. 2010 [cit. 19. 5. 2010]. Dostupné z WWW: <*http://www.mobygames.com/game-group/warcraft-universe*>.

[9] Wikipedia contributors, Wikipedia: the free encyclopedia — Stargus [online], aktualizováno 22. 11. 2009 [cit. 26. 11. 2009]. Dostupné z WWW: <*http://en.wikipedia.org/wiki/Stratagus*>.

[10] Bos Wars Team, Bos Wars [online], aktualizováno 11. 4. 2010 [cit. 15. 4. 2010]. Dostupné z WWW: <*http://www.boswars.org*>.

[11] Foundations for Genuine Game AI, FEAR [online], aktualizováno 11. 6. 2007 [cit. 4. 2. 2010]. Dostupné z WWW: <*http://fear.sourceforge.net*>.

[12] Skupina autorů, Moby Games — Age of Empires series [online], aktualizováno 19. 5. 2010 [cit. 19. 5. 2010]. Dostupné z WWW: <*http://www.mobygames.com/game-group/age-of-empires-series*>.

[13] S. J. Russel, P. Norwig: *Artificial Intelligence: A Modern Approach (Second Edition)*, Pearson Education, 2003.

[14] Edited by S. Rabin: *AI Game Programming Wisdom*, Charles River Media, 2002.

[15] Edited by M. Deloura: *Game Programming Gems 1*, Charles River Media, 2000.

[16] Edited by M. Deloura: *Game Programming Gems 2*, Charles River Media, 2001.

[17] Edited by D. Treqlia: *Game Programming Gems 3*, Charles River Media, 2002.

[18] Edited by A. Kirmse: *Game Programming Gems 4*, Charles River Media, 2004.

[19] Edited by K. Pallister: *Game Programming Gems 5*, Charles River Media, 2005.

[20] Edited by M. Dickheiser: *Game Programming Gems 6*, Charles River Media, 2006.

[21] Edited by S. Jacobs: *Game Programming Gems 7*, Charles River Media, 2008.

[22] Edited by A. Lake: *Game Programming Gems 8*, Charles River Media, 2010.

[23] D. M. Bourg a G. Seemann: *AI for Game Developers*, O'Reilly, 2004.

[24] Skupina autorů, Moby Games — Pac Man [online], aktualizováno 19. 5. 2010 [cit. 19. 5. 2010]. Dostupné z WWW: <*http://www.mobygames.com/game/pac-man*>.

[25] T. Masters: *Practical Neural Network Recipes in C++*, Academic Press, 1993.

[26] Skupina autorů, Moby Games — Steel Panthers [online], aktualizováno 19. 5. 2010 [cit. 19. 5. 2010]. Dostupné z WWW: <*http://www.mobygames.com/game/dos/steel-panthers*>.

[27] Skupina autorů, Moby Games — Panzer General [online], aktualizováno 19. 5. 2010 [cit. 19. 5. 2010]. Dostupné z WWW: <*http://www.mobygames.com/game/panzer-general*>.

[28] Skupina autorů, Moby Games — Battle for Wesnoth [online], aktualizováno 19. 5. 2010 [cit. 19. 5. 2010]. Dostupné z WWW: <*http://www.mobygames.com/game/battle-for-wesnoth*>.

[29] Skupina autorů, Moby Games — Combat Tanks [online], aktualizováno 19. 5. 2010 [cit. 19. 5. 2010]. Dostupné z WWW: <*http://www.mobygames.com/game/combat-tanks*>.

[30] M. Chung, M. Buro a J. Schaeffer: *Monte Carlo Planning in RTS Games*, CIG 2005.

Příloha A

Obsah přiloženého CD

Přiložené CD obsahuje ve složce **text** tuto práci ve formátu PDF, programátorskou dokumentaci k celé platformě ve formátu HTML a uživatelský manuál ve formátu PDF. Celý projekt se zdrojovými kódy v projektu Microsoft Visual Studia 2008 je uložen ve složce **sources** v podsložce **AI_Tester**. Všechny potřebné knihovny pro zkompilování projektu — tedy Qt Framework 4.6.0 a qwt 5.2.0 — jsou umístěny ve složce se zdrojovými kódy v podsložce **3rd-party**. Dále je zde obsažen instalátor platformy, který vyžaduje Microsoft Installer 3.5 a Microsoft .NET Framework 2.0. Ten je umístěn ve složce **installer**. Program se instaluje spuštěním souboru **setup.exe**. Prerekvizity potřebné k instalaci jsou zde v podsložce **3rd-party**. Instalovat prerekvizity z tohoto CD není nutné, protože v případě jejich absence je instalátor schopen si je stáhnout z internetu a nainstalovat sám. Pro testování algoritmů umělé inteligence bylo přibaleno několik ukázkových map a uložených pozic objektů na nich, které jsou ve složce **sources** a podsložce **AI_Tester** ve složkách **maps** a **objects**. Ve složce **replays** jsou umístěny ukázkové průběhy her. Po nainstalování aplikace pomocí instalátoru budou tyto mapy a replaye přístupné taktéž ve složce, kam byla platforma nainstalována. Pro jednodušší vytváření map je taktéž přibalen jednoduchý editor ve formátu XLSX, tedy vyžaduje Microsoft Excel 2007. Ten je zde ve složce **tools**.

Pro lepší orientaci v obsahu CD jsme připravili jednoduchý html soubor s odkazy s názvem *index.html*.

Příloha B

Zdrojové kódy

Vzhledem k použití opensource verze platformy Qt, jsou zdrojové kódy celé aplikace publikovány pod licencí GNU General Public License verze 3.0. Domovská stránka projektu je umístěna na Google Code pod adresou *http://code.google.com/p/ai-tester/*. Zde je také k dispozici repositář s aktuální verzí zdrojových kódů, které je možné stáhnout pomocí příkazu *svn checkout http://ai-tester.googlecode.com/svn/trunk/*. V sekci *Downloads* jsou umístěny soubory s programátorskou a uživatelskou dokumentací a instalátor aplikace pro systém Windows.