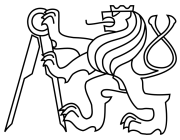
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	MCTS library for unit movement planning in real-time strategy game StarCraft
Student:	Mykyta Viazovskyi
Supervisor:	RNDr. Michal Čertický, Ph.D.
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of summer semester 2017/18

Instructions

The aim of the thesis is to design and implement a library for AI (Artificial Inteligence) agents playing real-time strategy game StarCraft, implementing Monte-Carlo Tree Search (MCTS) for unit movement planning. Usage of the library will be demonstrated and evaluated.

1. Introduce the field of RTS (Real Time Strategy) game AI development.
2. Make an overview of available resources and libraries for Starcraft AI.
3. Design and implement a new library for Starcraft AI. The design will allow further extensions - adding new functionality to the library.
4. The library will support the following two methods:
   4.1. Basic graph search-based planning of unit movement.
   4.2. Planning based on MCTS Considering Durations.
5. Show the basic use of the library on few expressive examples.
6. Perform experiments and compare the performance with alternative solutions.
7. Document your solution and discuss possible further extensions.

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 11, 2017

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Bachelor's thesis

MCTS library for unit movement planning in real-time strategy game Starcraft

Bc. Mykyta Viazovskyi

Supervisor: RNDr. Michal Čertický, Ph.D.

16th May 2017

Acknowledgements

I would like to thank my supervisor, RNDr. Michal Čertický, Ph.D., for his tournament organization that led me to the idea and his continuous support during the work on this thesis. My sincere thanks go to the Department faculty members for their help and support, and especially the teaching staff, who instilled the decent amount of knowledge used for the thesis and beyond. I am thankful to my partner and my friends who supported me through this project. Last but not the least, I am also grateful to my family for their endless support and encouragement during the whole period of my study.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 16th May 2017 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Viazovskyi, Mykyta. *MCTS library for unit movement planning in real-time strategy game Starcraft.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2017.

Abstrakt

Existuje společnost vývojářů umělé inteligence, kteří zkouší své nápady a pilně pracují, aby vytvořili neporazitelného protivníka pro živou strategickou hru Starcraft, což dokázali v Šachách a Go.

Tato práce předvádí využití knihovny pro Monte CarloTree Search Considering durations algorytmus, který byl prvně navrhnut Albertem Uriarte a Santagem Ontãńon z Drexelské univerzity. Tento algorytmus prokazuje vynikající výsledky v řízení armády v živých strategických hrách. Jako menší náhradu přidáme do knihovny vyhledávání Negamax. Naše využití algorytmu je vypracováno jako statická knihovna ++, která může být připojena k jakémukoli možnému botovi. Instalace je velmi jednoduchá a nenáročná. V průběhu práce vyhodnocujeme algoritmy, porovnáváme je a demonstrujeme jejich využití. Tyto algoritmy jsou založeny a testovány na platformě UAlberta bot.

Klíčová slova MCTSCD knihovna, Starcraft, strategie v reálném čase, prohledávání stavového prostoru, umělá inteligence videoher.

Abstract

There is a live community of AI developers that are trying their ideas and putting effort to create an unbeatable rival for real-time strategy game Starcraft, as it was done with Chess and Go.

This work presents an implementation of the library for the Monte Carlo Tree Search Considering Durations algorithm, that was recently proposed by Alberto Uriarte and Santiago Ontañón from Drexel University. It appears to bring outstanding results in real-time strategy army control. As a smaller substitute, we add a Negamax search to the library. Our implementation of the algorithms is designed as a static C++ library, which could be easily plugged in-to any possible bot. The setup is simple and intuitive. During the course of the work we evaluate the algorithms, compare them and demonstrate their usage. The algorithms are based and tested on UAlberta bot framework.

Keywords MCTSCD library, Starcraft, real-time strategy, state space search, video game AI.

Contents

List of Figures

Introduction

"The only way to get smarter is
by playing a smarter opponent."

Fundamentals of Chess 1883

Probably many of us have already heard of exponential growth. Technology skyrocketed at the end of the previous century and the amount of information is growing exponentially. In 1985 maximum hard disk capacity was around 10 MB, 1 GB in 1995, more than 100 GB in 2005 and in 2016 it was 16 TB. The world's technological per capita capacity to store information has roughly doubled every 40 months since the 1980s; as of 2012, every day 2.5 exabytes (2.5×10^{18}) of data are generated. In the age of hyper-innovation the number of world-changing inventions developed within the next 10 years will be equal to those produced in the previous 100 years. And in order to assist us to cope with the ever-increasing information flow, the artificial intelligence comes into play. In many fields strict algorithms are beaten by artificial intelligence (AI) in performance, in ability to grasp high dimensionality and even in presentation of interesting insights that previously have remained hidden. Image and speech recognition have had a big advancement due to neural networks. The Big Data is handling such complex challenges as: analysis, capture, data curation, search, sharing, storage, transfer, visualization, querying, updating and information privacy. The world encloses an unimaginably enormous amount of digital information which is getting ever bigger ever more rapidly.[18] AI allows us to do many things that previously were not possible: spot business trends, prevent diseases, combat crime and so on.

Nowadays gamers are pushing the limits of creativity, attention, reaction and many other game-related skills. Even modern built-in AI is not interesting to play with for an average player. But machine learning is evolving, thus providing new opportunities for both computer opponent and general computational comprehensiveness.

The Starcraft bot development is a field, where pleasure and science go hand in hand. Creating a more advanced bot contributes makes it more interesting to play. And as the skill of human play rises, he needs to have a more advanced enemy. Thus, the aim of the work could be seen as a contribution to both science and gamers' enjoyment.

Therefore this topic was decided to be a contribution to the combination of AI and game development. This will help to achieve better results in both fields with the enthusiasm of the respective communities.

Challenges

Real-Time Strategy (RTS) games pose a significant challenge to AI mainly due to their enormous state space and branching factor, and because they are real-time and partially observable.

Early research in AI for RTS games identified the following six challenges[4]:

- resource management;
- collaboration (between multiple AIs);
- opponent modelling and learning;
- decision making under uncertainty;
- spatial and temporal reasoning;
- adversarial real-time planning.

Some of these fields are being developed, while others stay untouched (e.g. collaboration). With the current implementation we tackle the last three of these challenges.

As mentioned above, the size of the state space in RTS games is much larger than that of traditional board games such as Chess or Go. This is one of the reasons they were chosen as a path of general AI research. As mentioned at BlizzCon 2016, Starcraft 2 will be the next platform to be conquered by Deepmind. Research scientist Oriol Vinyals states, that "StarCraft is an interesting testing environment for current AI research because it provides a useful bridge to the messiness of the real-world. The skills required for an agent to progress through the environment and play StarCraft well could ultimately transfer to real-world tasks".[17]

Additionally, the number of actions that can be executed at a given instant is also much larger. Thus, standard adversarial planning approaches, such as game-tree search, are not directly applicable. As it could be anticipated, planning in RTS games is approachable with the layer of high abstraction: the game is not perceived as individual units and locations, but rather unified groups and modules supervising every responsibility.[10]

Adversarial planning under uncertainty in domains of the size of RTS games is still an unsolved challenge. The uncertainty consists of two parts. On the one hand, the game is partially observable, and players cannot see the whole map, but need to scout in order to see what is happening in the dark region. This kind of uncertainty can be lowered by good scouting, and knowledge representation (to infer what is possible given what has been seen). On the other hand, there is also uncertainty arising from the fact that the games are adversarial, and a player cannot predict the actions that the opponent(s) will execute. For this type of uncertainty, the AI, as the human player, can only build a sensible model of what the opponent is likely to do.[10]

Certainly, this great amount of challenges made RTS games a fruitful playground for new AI algorithms.

Many of the algorithms are tested at the tournament, as there is a big variety of strategy approaches. The Student StarCraft AI Tournament (SSCAIT) currently has 57 bots and 124 registered bot authors[12]. In addition to SSCAIT, there are two other important SC AI competitions - both collocated with research conferences: AIIDE and CIG[6]. Currently, there are 5 types of tournaments being held[1].

Chapter 1

Aim of the thesis

This thesis is aimed at helping to create accessible tools for AI game development. The particular target used in our thesis is Blizzard's Starcraft real-time strategy game, which is an excellent example of complex, deep and challenging competition platform. Even though the Starcraft was released in 1998, there are not that many Starcraft bot development tools available up to this moment. We review them in the following chapter.

The reason this topic was selected is the shortage of building blocks that a regular developer can use. The articles on various strategy improvements either lacked implementation or crudely integrated into the author's bot. This is exactly the case of MCTSCD, which we want to make accessible to general public.

The core of the thesis is to make an easily attached algorithm of search through the vast game state space. The user will be given a set of functions and classes, which will grant him an ability to get the most effective way to retreat, attack or defend. The answer will be based on many dimensions of the state space, like: number and position of our army, number and position of the enemy army, map layout, chokepoints, etc. We believe that implementation of the library would assist fellow researchers or even ordinary geeks to get new inspiration and push the limits of AI while playing their favourite game.

In video games the frame is a picture which is usually shown to the user every $\frac{1}{24}$ seconds. The architecture of video games forces the logic to process before the frame is displayed. Because of this real-time constraint, the algorithm has to show results quickly, without stopping the frame for too long. Otherwise the picture will look intermittent. For further discussion see Section 3.1.

1.1 Thesis structure

This work is structured in the following manner.

At first, we investigate the state of the art for Starcraft bot development. We list the main and popular items that are ready to be used. These are mostly map tools, data mining tools and a generic bot framework.

Secondly, we discuss the design of the library and the algorithms. The most important parts are illustrated in figures.

Further, we describe the implementation details. The MCTSCD will be tested against a Negamax algorithm. So, we state what is common for the two, and then dive into implementation of each individual algorithm. As promised, there are simple setup instructions, which are at the end of the chapter.

Finally, we test the implementations. We argue about the testing ground and show the performance of two methods.

Chapter 2

Resources and libraries overview

There are various libraries for the Starcraft. Most of them cover map analysis or offer some base for a rapid start for bot development. But the most significant framework is the Brood War Application Programming Interface (BWAPI). BWAPI is a free and open source C++ framework that is used to interact with the popular Real Time Strategy (RTS) game Starcraft: Broodwar. This framework allows many people to channel their thoughts and ideas to the creation of Artificial Intelligence in Starcraft.

BWAPI only reveals the visible parts of the game state to AI modules by default. Information on units that have gone back into the fog of war is denied to the AI. This enables programmers to write competitive non-cheating AIs that must plan and operate under partial information conditions. BWAPI also denies user input by default, ensuring the user cannot take control of game units while the AI is playing.[9]

With BWAPI one is able to:

- Write competitive AIs for Starcraft: Broodwar by controlling individual units.
- Read all relevant aspects of the game state.
- Analyze replays frame-by-frame, and extract trends, build orders and common strategies.
- Get comprehensive information on the unit types, upgrades, technologies, weapons, and more.
- Study and research real-time AI algorithms in a robust commercial RTS environment.

Thus, the appearance of BWAPI was the crucial point of the bot development for Starcraft.

Figure 2.1: BWTA terrain partition. Green: unpassable line. Red: the most narrow distance between regions.

Next, there is the Broodwar Terrain Analyzer 2 (BWTA2), a fork of BWTA, an add-on for BWAPI that analyzes the map and computes the regions, chokepoints, and base locations. The original BWTA was not able to analyze all kinds of Starcraft maps, especially the ones from tournament. The BWTA2 is a basis for any bot development, as we can easily access and understand the basic blocks of the current map, get all possible locations, compute the distance to our next target etc. As our library provides a way for developers to search for best shifts through the map, it is heavily relying on BWTA2.[15]

There is an alternative to BWTA2, Brood War Easy Map (BWEM), which is a C++ library that analyses Brood War's maps and provides relevant information such as areas, choke points and base locations. It is built on top of the BWAPI library. It first aims at simplifying the development of bots for Brood War, but can be used for any task requiring high level map information. It can be used as a replacement for the BWTA2 add-on, as it performs faster and shows better robustness while providing similar information.[7]

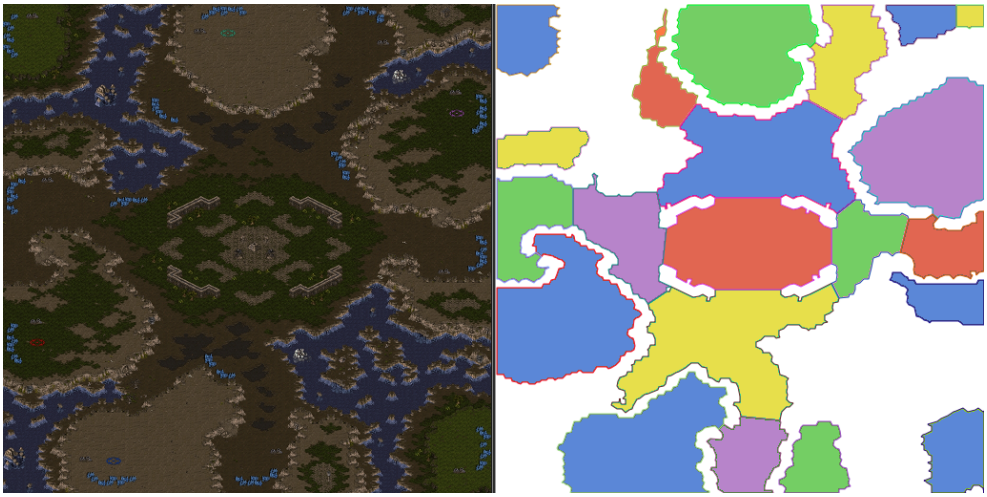


Figure 2.2: BWTA region split

Though BWEM has some limitations[7]:

- BWEM doesn't provide any geometric description (polygon) of the computed areas.
- It is not compatible with older versions of BWAPI, which might be useful if the bot is not being updated to the newest version of the library.

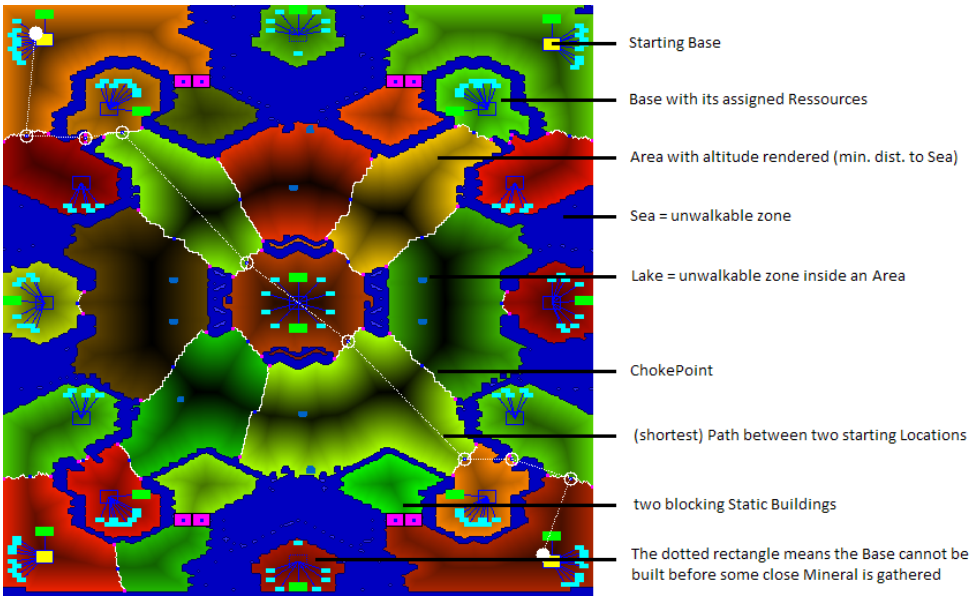


Figure 2.3: BWEM region split

The BWAPI Standard Add-on Library (BWSAL) is a project that aims at developing several add-ons for BWAPI that will be useful for a wide variety of AIs, including different managers per each aspect of the game: workers, buildings, build orders, supply, bases, technologies, upgrades, scouts, defence, general information and unit groups. It is similar to the UAlbertabot, but it is providing only individual managers, not the whole solution.[8]

What is rather unique in its presence is the StarCraftBenchmarkAI. This is a benchmark for Starcraft Intelligent Agents. It allows different metrics to evaluate the performance of the agents: survivor's life, time survived, time needed to complete the goal, units lost. These metrics are applicable to various scenarios, where agents can perform different tasks, like navigation though obstacles, kiting enemy or strategy placement and recovery.[14]

UAlbertaBot has been written and its code made public by Dave Churchill of the University of Alberta, Canada. UAlbertaBot is an AI capable of playing Random, and in that possesses D-level game play in all three races. It is very well documented and recommended as a starting platform for new bot-coders. It is essentially a framework that could be easily expanded into a bot of our choice. It has simple configuration, well-done documentation and design. It was used for testing purposes for our library. The framework contains not only the bot, but also a combat simulator, called SparCraft, which allows us to make predictions on the forthcoming battle. And the last feature of the UAlbertaBot is the Build Order Search System (BOSS), which helps with selecting and comparing a proper build order, hence giving more strategy options. We will more closely consider the UAlbertaBot in the Section 3.1.[5]

Another interesting project is the TorchCraft. It is a bridge between Starcraft and Torch, a scientific computing framework with wide support for machine learning algorithms.[11]

Chapter 3

Library design

3.1 Library format

As for the C++ library for the AI development there are two options to select from: statically- and dynamically-linked. The library was decided to be static for the following reasons:

- it will be compiled directly into bot executable
- it saves execution latency
- if search functionality is needed, most probably it is to be provided constantly (throughout the whole match)
- it is ensured that all functionality is up to date (no versioning problem)

The BWAPI interface is a shared library itself due to the reverse-engineered nature of the solution, which could add some delay on its own. As with the CPU-bound projects like Starcraft and BWAPI, it is always good to think about performance. Even though on a regular basis we may allow some delay for the bot and game response (as those are bound), it is forbidden, during the official tournaments, to cross some predefined limit.

Aside from the obvious causes to suffer defeat like a crash or loss of all units and buildings, the bot is considered defeated if it has spent:

- more than 1 frame longer than 10 seconds, or
- more than 10 frames longer than 1 second, or
- more than 320 frames longer than 55 milliseconds.

One could argue, that use of static library could be a waste of space. But as long as it provides essential part of bot logic, and if that logic is used, it is presumably going to be used constantly during the whole match. Additionally,

the size of the library would be negligible (25 MB) in relation to Starcraft size and the size of modern average data storage devices.

As the library is intended to provide only individual algorithms, its best option is going to be used from one of the managers of the bot.

With the current MCTSCD algorithm, its best place would be some squad manager, to control the movement of separate groups of units. Though guidelines on the bot structure are not defined in Starcraft, most developers either use UAlbertaBot or have similar approach. The UAlbertaBot has the CombatManager module, which we are basing our tests on (Figure 3.1).

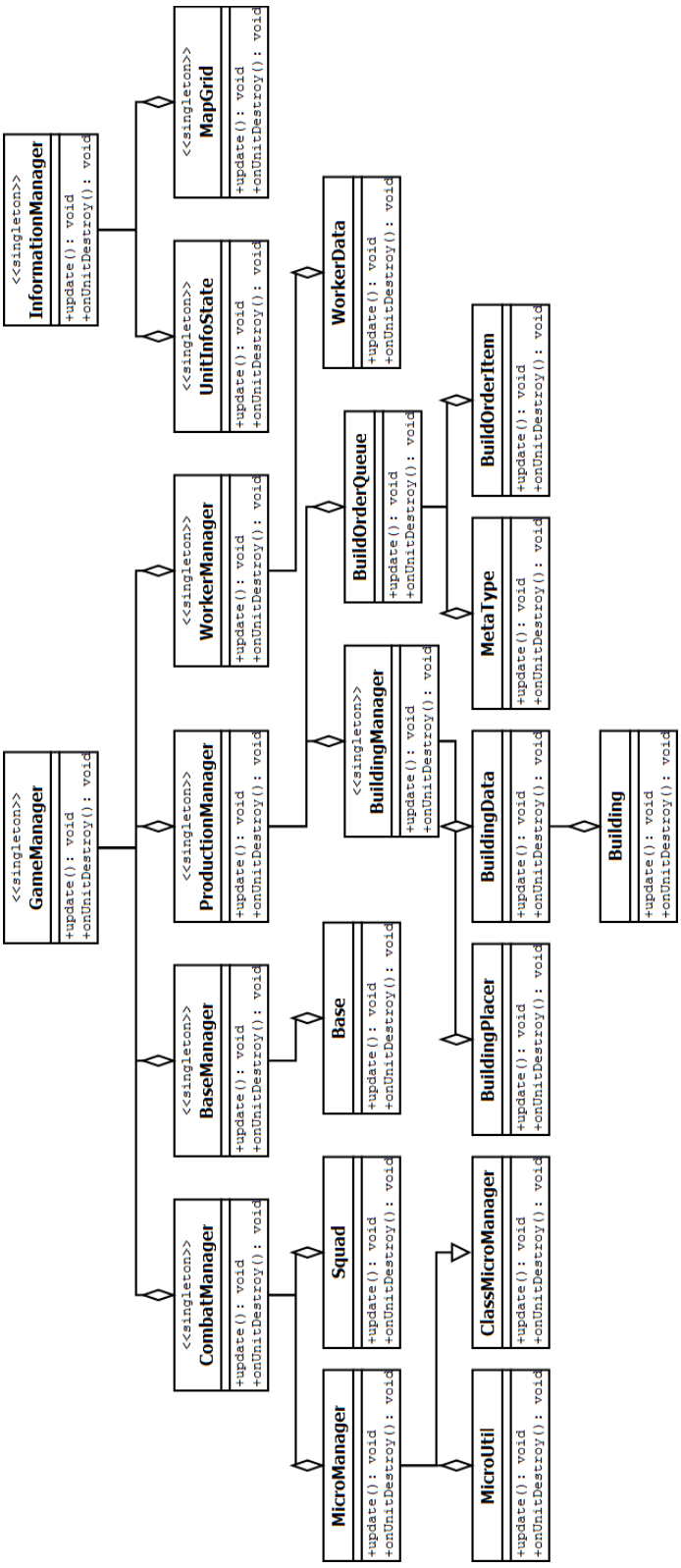


Figure 3.1: Dave Churchill's UAlbertaBot structure

3.2 Library architecture

Now it is time to check the class structure. Overall, the project consists of 23 classes. The most crucial ones are: *AbstractGroup, ActionGenerator, CombatSimulator, EvaluationFunction, GameNode, GameState, MCTSCD, RegionManager* and *UnitInfoStatic* (see Figure 3.3). These are the absolute core of the library, and we should examine each of them.

AbstractGroup. This class keeps the parameters of the abstract group, where all units of the same type and from the same region are joined. We cover the strategy decomposition in the following chapter. Aside from aforementioned specification, the group also has to note the order to be executed, the average hit points, size of the group, and the frame bounds, in which the group will exist.

ActionGenerator is responsible for noting all possible actions for the player and his army, and retrieve different sets of actions. The actions could have the following parameters: number, randomness, bias, player and abstraction level.

CombatSimulator. Whenever in the search there is a situation, where two rival groups collide, the outcome is resolved in a combat simulation. This is an interface, because there are different approaches to the simulation: *Target-Selection Lanchester's Square Law Model* (CombatSimLanchester), *Sustained Damage per Frame (DPF) Model* (CombatSimSustained) and *Decreasing DPF Model* (CombatSimDecreased)[16].

EvaluationFunction is a simple yet important class for the search. In our implementation it evaluates the number of units of each type times the "destroy score", the amount of score points awarded the player gets for the unit kill (at the end of the match).[9]

GameNode is part of the search core. It represents the tree node, and holds many parts related to evaluation, such as: actions (what actions are to be executed if player is at this node of the tree), *totalEvaluation* (a numerical representation of how advantageous current actions are), *totalVisits* (how many times the node has been visited), *gameState* (what is the actual situation on the map when the node is visited) and *actionsProbability* (how likely it is to have the selected actions appear in the game). *totalVisits* is the crucial point to evaluate the search tree before the simulation.

GameState is a view of the actual game. First of all, it is the army of both sides, that has to be reflected in the state. Next, there are the regions in combat, to check the expected end frame of the current state and, consequently, be able to forward the game state in future. This bit contributes to the *Considering Durations* part of the algorithm name.

MCTSCD is the search itself. It is controlled by the depth, number of iterations and maximum simulation time. All these values have to be provided by the user. All these parameters are discussed in the section 4.3.

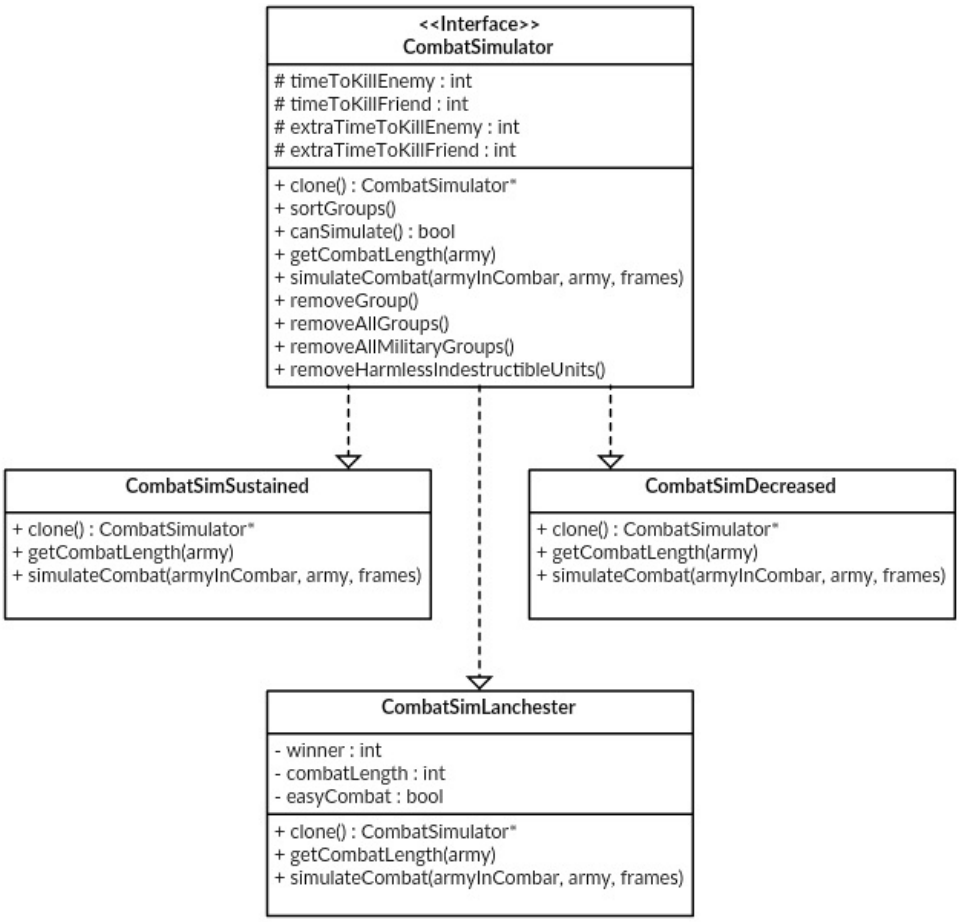


Figure 3.2: Combat simulation design class model

RegionManager is a static entity (not in terms of C++ though). An object of this class has to be created once, but is used throughout the game. It represents the regions, chokepoints and all that is connected to them. It helps all other modules get instant information about the map, such as:

- What is the region, the Id of which we dispose (regionId).
- What is the id of the given region (regionFromId).
- Given X and Y coordinates, what region it is (regionIdMap).
- *same applies to chokepoints*

UnitInfoStatic. Similar to the previous class, this one is fixed after its creation, but provides useful information about all the available units in

the game. It offers the following information: typeDPF (a mapping from unit types to their damage output), DPF (similar mapping, but in regards to ability to attack air and/or ground units) and HP (hit points against air and/or ground attacks).

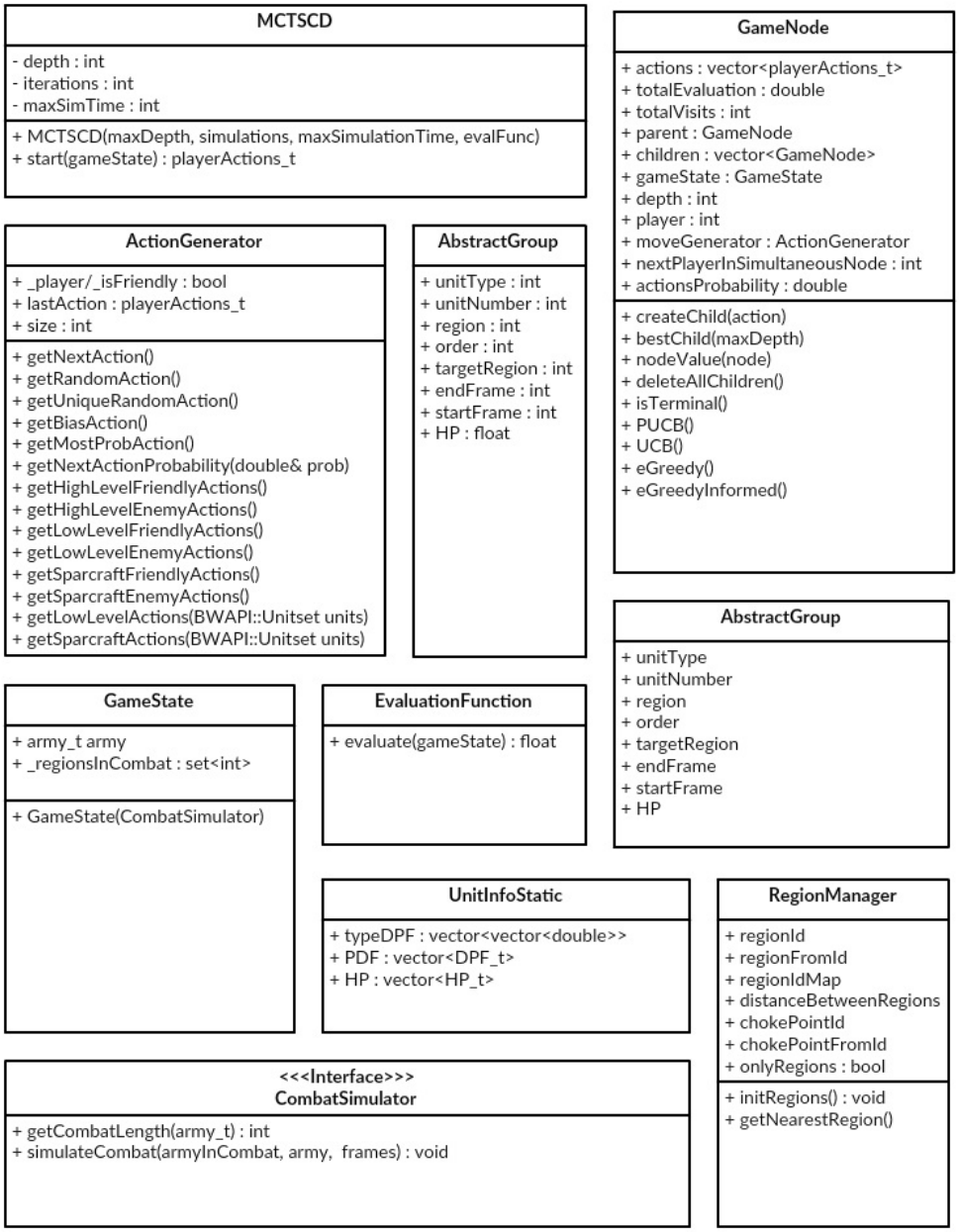


Figure 3.3: Class diagram

3.3 Algorithm structure

The backbone of the algorithm developed by Uriarte and Ontañón is on the Figure 3.4 [13].

The core of the algorithm is the MCTSCD class. It performs the search itself based on possible actions in the GameNode and the game estimate from EvaluationFunction.

Building block of the search tree is the GameNode. Every GameNode takes care of the actions (order-region bundle), an evaluation of the current game state, number of visits of the node, the evaluated GameState and identification of whose turn it is. This class aggregates the game situation and possible actions in form of GameState and ActionGenerator respectively.

The GameState class is responsible for the representation of the game in terms of army and regions that are in combat, ongoing or in the future. GameState uses CombatSimulator interface to approximate the result of combats that are to happen in that state.

As already stated, the CombatSimulator interface helps with estimation of the combat outcome. Currently, one of the following implementations of the interface could be selected: CombatSimLanchester, CombatSimSustained and CombatSimDecreased (see Section 3.2).

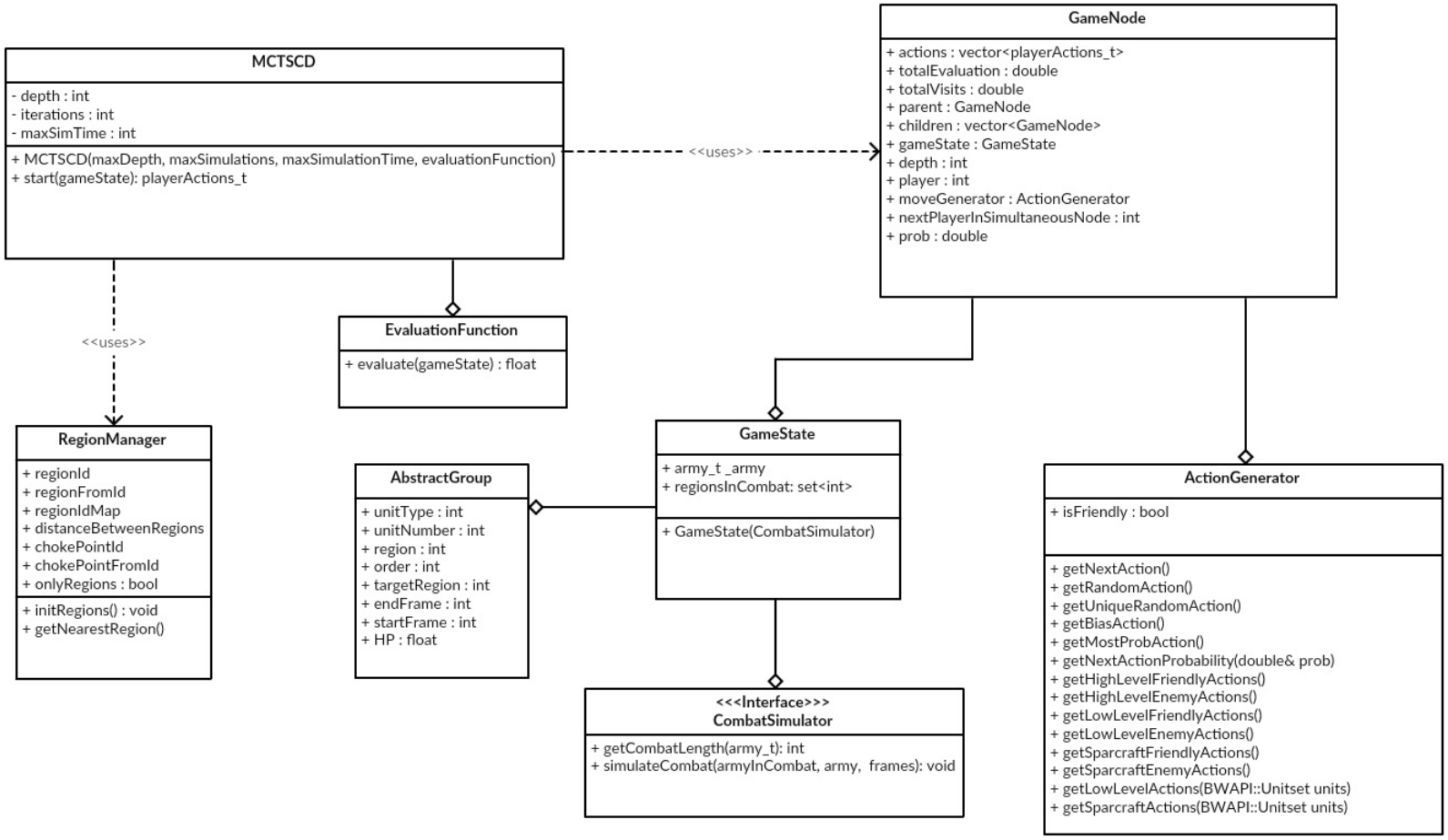


Figure 3.4: Search class diagram

Chapter 4

Implementation

Combat is a gist mechanic in most RTS games. During a combat each player commands his units to defeat the opponent. Thus, a unit's positioning could be a critical moment of the match. The unit movement planning is at the core of general army strategy. There could often be such circumstances, in which both rivals have almost the same resources, map control and army size. That would be the case when positioning determines everything.

To address the location of the army properly we represent the map as a graph, where each adjacent region (set of tiles) corresponds to a graph node. Strategic decisions are made across the graph. Furthermore, instead of considering every individual unit, we classify the same unit type on the same region as one entity. The entities in the same region (graph node) are recognized as being next to each other, and those in different regions – not being able to reach one another. The map from the Figure 4.1 would have 20 regions, 5 of which are not connected by land, which gives us a graph with 15 nodes, and 5 separate ones. Those nodes are only accessible by air units.

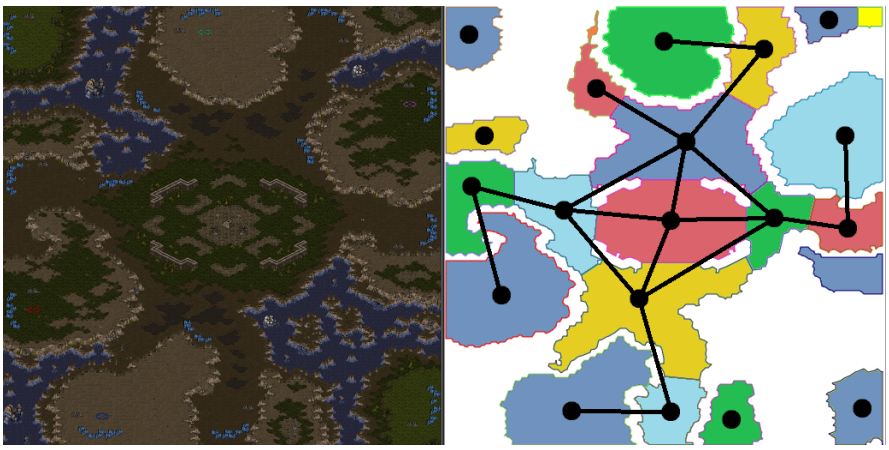


Figure 4.1: Map graph

In order to prevail in the RTS game it is necessary to evaluate the vast game state space of all possible moves. This considers not only building placement and unit recruiting, but also the movement of every unit. In Starcraft, there is a virtual 200 units upper bound (400 for Zerg race) per player, which means 4^200 moves per frame, or $4^200 \times 24 \times 60$ moves per minute. Obviously, such complexity must be overcome with some abstraction. We offer an abstract representation of the game state with army and time bounds, which could be shifted to the future state through simulation. This helps to greatly decrease the branching factor in the game-tree search.

It is important to remark the high- and low-level representation of the movements. Initially, all attack units are added to the game state. They are joined into groups, and mapped to the map graph. Secondly, the search with simulations is performed on the game node, unfolding the tree as the search proceeds. And finally, the set of actions is returned, which has to be assigned back to the corresponding groups that were considered in the first step. This gives us the deliberation of the abstract state and exactness of the detached groups.

Every group must be uniform in terms of type and place. So all together group has to store: owner, type of unit, unit number, average health, currently performed action, start frame, end frame and the region (see Figure 3.3). Based on these and other features the search is able to perform proper simulation and reasoning.

The small battle in Starcraft could look like the one on the Figure 4.2. Clearly, 21 units are far from the upper bound (200 per player). But granted that every unit could move, stay idle or attack, it gives us total 3^{21} actions just on this small field! Conversely, in abstract domain this is just 3^2 actions.

Figure 4.2: Unit abstraction

4.1 Negamax algorithm planning

As our library is going to offer a unit movement algorithm, in order to show the benefits and advantages we decided to compare it against some simpler algorithm.

After extensive research, it became clear that the Negamax algorithm was right for the job.

We have a slightly modified Negamax to correspond to RTS game reality. Negamax search is a variant form of Minimax search that relies on the zero-sum property of a two-player game.[19] Thus, if our move has the same consequences as opponent's move, but with negative sign, then it's a zero-sum game. In most situations of the game, Starcraft is a zero-sum game (with rare exceptions, which could be fixed with proper evaluation function). But we are focusing on the movement search, or so to say, strategic positioning, which is clearly a zero-sum.

The heuristic value of the node is evaluated as the number of our units in the region versus the number of enemy units. This allows us to have the simplified strategy of arranging our army in such a way that we would have the dominance at the region. That will help to crush the enemy locally and have the advantage afterwards.

The Negamax search task is to search for the best node score value of the player who is playing from the root node. The pseudocode below (Code 4.1) shows the Negamax base algorithm[3], where we could limit the maximum search depth:

Code 4.1: Negamax pseudocode

```
1 function negamax(node, depth, color)
2     if depth = 0 or node is a terminal node
3         return color * the heuristic value of node
4
5     bestValue := −∞
6     foreach child of node
7         v := −negamax(child, depth − 1, −color)
8         bestValue := max(bestValue, v)
9     return bestValue
```

The root node inherits its score from one of its children. The child node that ultimately sets the root node's best score also represents the best move to play[3]. Although this Negamax function shown only returns the node's best score as bestValue, our Negamax implementation keeps both the evaluation and the node, which is keeping the game state value. In the basic Negamax the only important information from non-root nodes is the best score. And the best move isn't necessary to retain nor return for those nodes.

The calculation of the heuristic score could be perplexing. At first, the color of the user has to be provided. This is the arithmetic detail to alternate the heuristic result. In this implementation, the heuristic value is always calculated from the point of view of player A, whose color value is one. Higher heuristic values mean more favourable situations for player A. This behaviour is similar to the regular Minimax algorithm. The heuristic score is not necessarily the same as a node's return value, bestValue, due to result negation by Negamax and the color argument. The Negamax node's return value is a heuristic result from the point of view of the node's current player.

Alterations of Negamax may omit the color parameter, which is our case. The heuristic evaluation function returns values from the point of view of two players, noting the size of the army for every player[3]

4.2 Planning based on MCTS Considering Durations

Monte Carlo tree search relies on the Monte Carlo method. The idea behind the method is to continuously sample random elements to obtain results. It is using randomness to address deterministic problems. Additionally, Monte Carlo methods can be used to solve any problem having a probabilistic interpretation.[20]

Generally, Monte Carlo methods have the following structure:

- Define a domain of possible inputs.
- Generate inputs randomly from a probability distribution over the domain.
- Perform a deterministic computation on the inputs.
- Aggregate the results.

Even though the Monte Carlo tree search is based on the Monte Carlo principle, it requires a search tree. Instead of just running random simulations from the current node, it uses the results of the simulations to compare simulations and propagate the search recursively through the search tree.

MCTS is an alternative to Negamax, that helps to oppose the high branching factor. The initial idea of the MCTS is to simulate the current state of the game until some point, which could be final result or intermediate, but defined step. The key in the algorithm is to balance between the exploration and exploitation of the tree. In these terms, exploration is looking into undiscovered subtrees, and exploitation is expanding the most promising nodes. There is a variability of policies to simulate the game until the logical stop, the default one being the uniform random actions of the player.[13]

What is impressive about the MCTS is that it is able to run almost indefinitely (as it explores the great state space), and be stopped at any moment. This behaviour resembles the human thinking process, when we are allowed to consider the problem for some time, and then give the final result. Possibly, the result would have been better, if given extended time, but it is the constraint that we always have to face.

Ability to "think" and stop is the biggest difference between Negamax and MCTS. To balance out the Negamax, we must manually adjust the depth of the search to fit into computation bounds.

The other positive difference of MCTS is the use of heuristic selection to explore the search tree. It does not unroll all possible results, which helps to avoid potentially poor decisions. If we know, that some move is highly undesired, it is indifferent what could be done out of it. This is the essential part of what makes the algorithm so effective in finding the favourable solutions.

Code 4.2: MCTS Considering Durations

```
1  function MCTSSearch(s0)
2      n0 := CreateNode(s0, 0)
3      while withing computational budget do
4          nl := TreePolicy(n0)
5          4 := DefaultPolicy(nl)
6          BACKUP(nl, 4)
7      return (BestChild(n0)).action
8
9  function CreateNode(s, n0)
10     n.parent := n0
11     n.lastSimult := n0.lastSimult
12     n.player := PlayerToMove(s, n.lastSimult)
13     if BothCanMove(s) then
14         n.lastSimult := n.player
15     return n
16
17 function DefaultPolicy(n)
18     lastSimult := n.lastSimult
19     s := n.s
20     while withing computational budget do
21         p := PlayerToMove(s,lastSimult)
22         if BothCanMove(s) then
23             lastSimult := p
24         simulate game s with a policy and player p
25     return s.reward
```

4.3 Library usage

To start with the library application, it would be needed to import header files and create corresponding classes of the search.

There is an option to provide several arguments for the Monte Carlo search: depth, number of iterations and maximum simulation time. The depth limits how deep the search should go down the tree. Upon reaching this limit, the node will be considered terminal. Number of iterations tells us how many children will the root node have. This parameter sets how many additional tries of the search are going to be performed. Depending on the computational budget this could be high in case the machine is able to perform far more operations than it is necessary for the game, or vice versa.

All these arguments are to be provided to the MCTSCD on initialization. The search object has to be created once, and then invoked every time the user wants to have strategic results. It is up to him, if he wants the algorithm to give suggestions every frame, or on some defined intervals. For example, it is possible to run very long search, and rely on its results for several frames, as it would make predictions further to the future, than the short one. On the other hand, it is possible to run short searches every frame, to keep the army ideal positioning up to date. The first approach might need some computation rebalance on the slow machine.

Chapter 5

Testing

The library testing has been performed with UAlbertaBot, which offers a basic bot structure, has good architecture and is easy to modify. It has been empirically proven that this framework is a great place to start. Many bot creators got inspired by it[12].

5.1 Case study

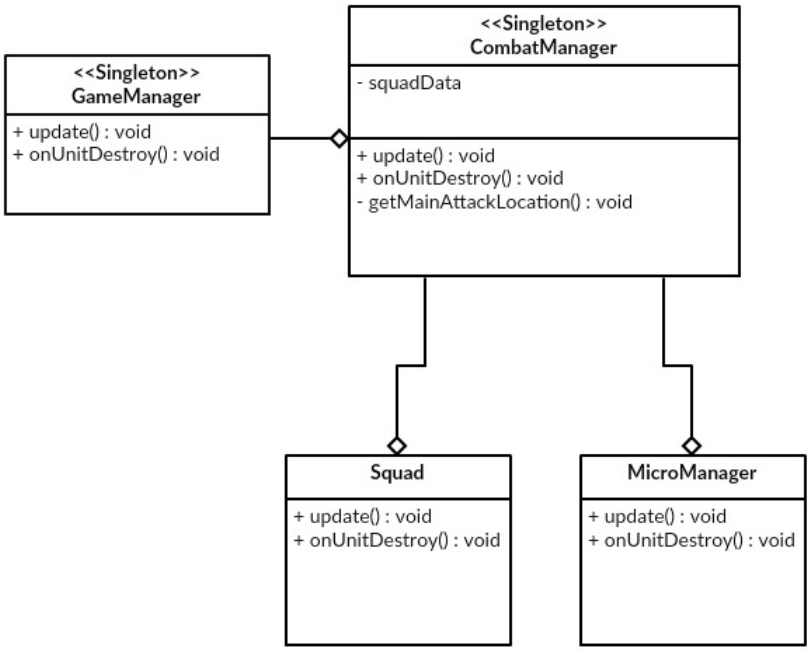
For the proper placing, we had to investigate the structure of both UAlbertaBot and the search, how those two could be merged effectively with the least amount of effort for the end user.

Because our search does not use the squads of the user, it is necessary to bind the user squad to the internal unit grouping.

The CombatManager module (see Figure 5.1) of the UAlbertaBot is the correct place to start with search integration. It serves a function of controlling the combat units of a player. The execution of modules is hierarchical: the BWAPI library is calling the *onFrame* function of UAlbertaBot, which gets propagated to GameManager (Figure 3.1) and then to CombatManager. The manager has access to the *squadData*, a set of all squads of the player. This gives the control over the unit distribution across groups, which is exactly what is needed for the search.

The division of units into groups inside the search is based on unit type and the region. The closer the squad formation is to the group, the more precise army coordination is achieved. Given the unit set the search does the mapping from to internal structure on its own, but the user has to keep track of the set, as it will have to be joined with the returned actions correspondingly. The more squad units mapping conforms with unit type and place, the better the search results are. Thus, we decided to split the army into several sets, based on their location. When the result is computed, it is assigned to the corresponding squad.

Figure 5.1: Combat manager diagram



The *updateAttackSquads* method is the proper place to embody the search results. The result of each algorithm is the vector of locations corresponding to the vector of squads that the player currently owns.

5.2 Test setup

As long as the game is partially observed, the MCTSCD would not be able to give decent results on the strategy. Thus, we would enable the whole game information from the BWAPI, so bots would see each other from the very beginning, and would not need to scout the map.

This *fog of war* is another complexity dimension of the game, which could be partially solved by thorough map investigation. But that would require an additional manager to be written (or the old one improved). Still, to refine the results of the matches and avoid additional random element in the measurements, we enable the entire map vision for both opponents.

Testing has been performed on the following basis. Both algorithms are compiled into bots, and run to play against the in-game AI. The UAlbertaBot is able to outplay the default version of the bot, but offers a scripted behaviour. This could be noticed, remembered and exploited. With the search based on simulation, the results are not always victorious, but they are much less exploitable (if at all). Thus, the players will benefit from more advanced play, and AI will become more general.

We decided to make 160 iterations of competitive play to show comparative performance of the MCTSCD and Negamax.

It is important to select a proper map to compare the performance of the searches. Ideally the map graph has to be fully connected and have the highest distance between the nodes, but Starcraft maps are limited in size (128×128 tiles because of the memory restrictions – the game was released in 1998). In addition to that, the map has to offer some path finding challenges and strategic areas which would allow players deliberate tactical formations. All in all, we selected the maps that are as close to the ideal, as possible. They have many regions, and these regions are mostly connected.

In every match bots play the same race (Zerg) and build order (that means they build same buildings and recruit the same units). This is done to ensure as minimal random factor, as possible.

The in-game AI is set to play Protoss race. It has stable build order and doesn't depend on player actions. This is also selected to decrease randomness.

5.3 Results

Let us discuss the results of the comparison. The bots were set to log the frame at which the game was finished (*frame*), the total number of points for created units (*unit score*), the total amount of points for killed units (*kill score*) and the winner. Each bot played 160 games, statistic data was cleared from outliers – Negamax was not able to reach his enemy several times due to UAlbertaBot limitation. Based on the winning data, the win ratio for MCTSCD is exactly 50% and $23,7\%$ for Negamax. Thus, we can safely conclude that MCTSCD is twice better than Negamax for this case.

5.3.1 Unit score

The unit score metric shows us the number of points for all our units at the end of the game. The meaning behind this result is how well did the bot expand during the game. In other words, how well did it survive the enemy attacks.

As could be noted from Figure 5.2, the Monte Carlo method is leading until the 30000 frame mark (21 minutes on normal game speed). Moreover, it finishes the game sooner with bigger number of units (having fewer units lost). It is an incontestable leader in this case.

5.3.2 Kill score

On the contrary, the kill score metric shows how many units were killed by the bot. And again, the MCTSCD has bigger score up to 28000 frames and its average converges with Negamax at 40000 frames (see Figure 5.3).

As we can see, there is a clear split of data into two parts for each algorithm. This is due to the success in the battle – if you won the battle, you are more likely to win the next one and eventually the whole game. Likewise, losing a battle brings more threats to lose the game. Such split was not present for unit score, as making more units doesn't directly influence the number of units to be made afterwards.

Higher kill score of the Negamax is not an index of better performance in the long game. As its results are shifted to the right, this means that algorithm failed to finish the game earlier, therefore it tortures the opponent longer.

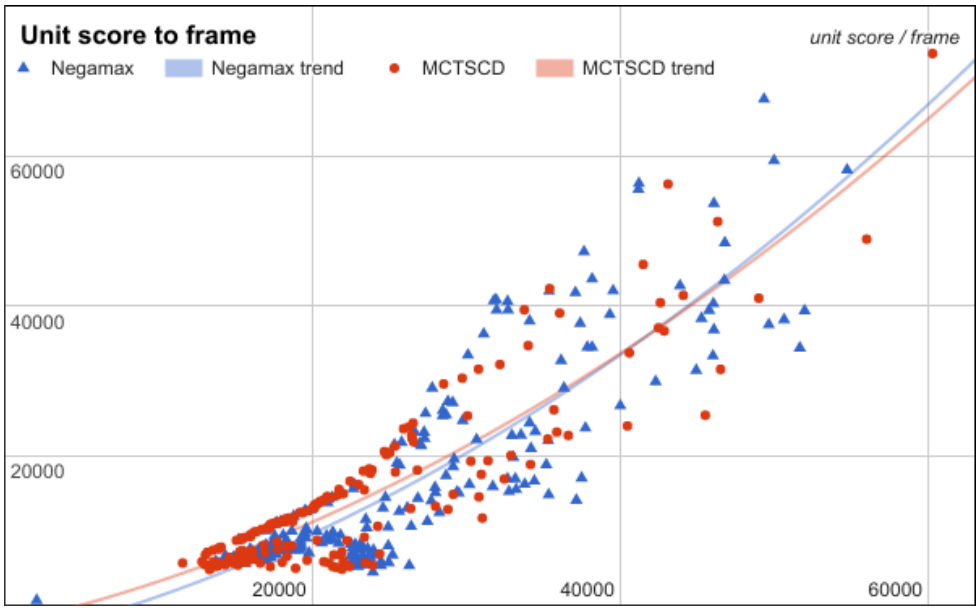


Figure 5.2: Unit score per frame

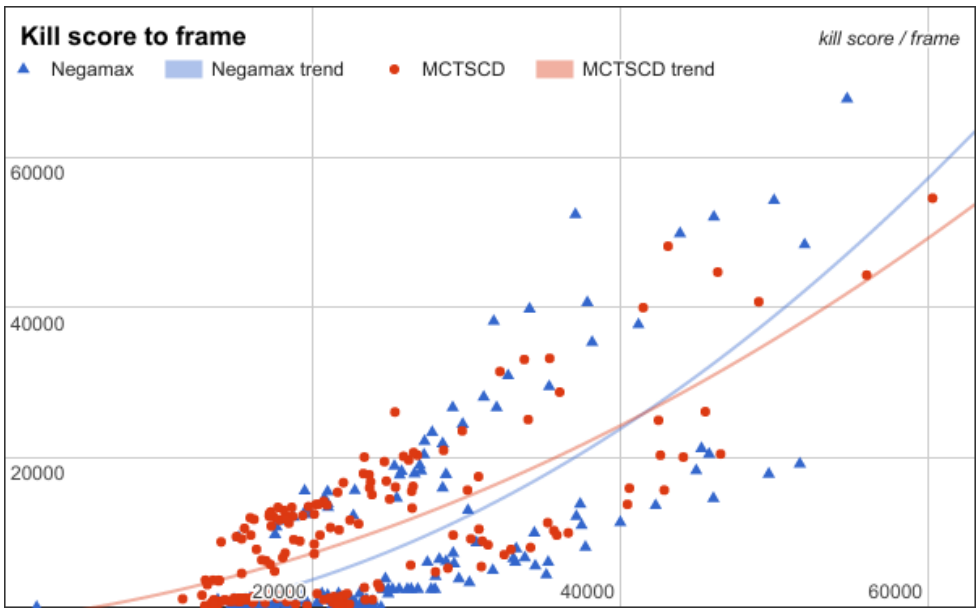


Figure 5.3: Kill score per frame

Conclusion

The purpose of the thesis was to make a tool for Starcraft bot development, specifically, library for Monte Carlo Tree Search Considering Durations algorithm and test it against an alternative.

The key result is that researchers can now have an easy access to state space search tools, and they can focus on different aspects of the game AI; there are numerous challenges to be solved. The task is going to be completed clearly and effortlessly saving time and efforts for more inspiration.

As a part of our future work, we would like to put some additions to the library. There are many future possible extensions for the library, as it just began to exist. Several algorithms were developed in the recent past: Alpha-Beta Considering Durations, Portfolio Greedy search, Upper Confidence Bound and others. As RTS games have high dimensionality, there are many places for algorithm application.

We would like to add different measurement and tuning possibilities to the library, so that every parameter of the algorithm and all that is connected could be easily reachable, logged and analyzed.

Finally, we would definitely collaborate with our clients for the future research. As the purpose of the product is to be used by other people extensively, it has to be convenient and understandable as much as possible. In addition to that we would have everything thoroughly commented and documented as this helps with extension of the problem and lets new people understand what is going on behind the scenes.

Bibliography

[1] Starcraft AI. Starcraft AI, the resource for custom starcraft brood war AIs, May 2017. URL: http://www.starcraftai.com.

[2] arXiv. arxiv.org e-print archive, May 2017. URL: https://arxiv.org.

[3] Dennis M. Breuker. *Memory versus Search in Games*. PhD thesis, Maastricht University, October 1998.

[4] Michael Buro. Real-Time Strategy Games: A New AI Research Challenge. In *IN PROCEEDINGS OF THE 18TH INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 1534–1535. International Joint Conferences on Artificial Intelligence, 2003.

[5] Dave Churchill, May 2017. URL: https://github.com/davechurchill/ualbertabot.

[6] David Churchill, Mike Preuss, Florian Richoux, Gabriel Synnaeve, Alberto Uriarte, Santiago Ontañnón, and Michal Čertický. *StarCraft Bots and Competitions*, pages 1–18. Springer International Publishing, Cham, 2016. URL: http://dx.doi.org/10.1007/978-3-319-08234-9_18-1, doi:10.1007/978-3-319-08234-9_18-1.

[7] Igor Dimitrijevic. BWEM library, May 2017. URL: http://bwem.sourceforge.net.

[8] Fobbah. Bwapi standard add-on library ("v2"), May 2017. URL: https://github.com/Fobbah/bwsal.

[9] Adam Heinermann. An API for interacting with Starcraft: Broodwar, May 2017. URL: http://bwapi.github.io.

[10] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Trans. Comput. Intellig. and AI in Games*, 5(4):293–311, 2013. URL: http://dx.doi.org/10.1109/TCIAIG.2013.2286295, doi:10.1109/TCIAIG.2013.2286295.

[11] Gabriel Synnaeve, Nantas Nardelli, Alex Auvolat, Soumith Chintala, Timothée Lacroix, Zeming Lin, Florian Richoux, and Nicolas Usunier. Torchcraft: a library for machine learning research on real-time strategy games. *arXiv preprint arXiv:1611.00625*, 2016.

[12] SSCAIT team. SSCAIT: Bots and Score, May 2017. URL: http://sscaitournament.com/index.php?action=scores.

[13] Alberto Uriarte and Santiago Ontañón. Game-tree search over high-level game states in rts games. In *AIIDE*, 2014.

[14] Alberto Uriarte and Santiago Ontañón. A benchmark for starcraft intelligent agents. In *AIIDE*, 2015.

[15] Alberto Uriarte and Santiago Ontañón. Improving terrain analysis and applications to rts game ai. In *AIIDE*, 2016.

[16] Alberto Uriarte and Santiago Ontañón. Combat models for RTS games. *CoRR*, abs/1605.05305, 2016. URL: http://arxiv.org/abs/1605.05305.

[17] Oriol Vinyals. DeepMind and Blizzard to release StarCraft II as an AI research environment, May 2017. URL: https://deepmind.com/blog/deepmind-and-blizzard-release-starcraft-ii-ai-research-environment.

[18] Wikipedia. The Free Encyclopedia, May 2017. URL: https://en.wikipedia.org/wiki/Big_data.

[19] Wikipedia. The Free Encyclopedia, May 2017. URL: https://en.wikipedia.org/wiki/Negamax.

[20] Wikipedia. The Free Encyclopedia, May 2017. URL: https://en.wikipedia.org/wiki/Monte_Carlo_method.

Appendix A

Acronyms

MCTSCD Monte Carlo Tree Search Considering Durations

AI Artificial Intelligence

RTS Real Time Strategy

BWTA Broodwar Terrain Analyzer

BWAPI Brood War Application Programming Interface

DPF Damage per Frame

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `staralgo` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format