FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Pavel Šmejkal

Artificial Intelligence for Children of the Galaxy Computer Game

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.

Study programme: Computer science

Study branch: Computer Graphics and Game Development

Prague 2018

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature

I would like to thank to my supervisor Jakub Gemrot for priceless insights and guidance, and to Filip Dušek, who allowed me to use his game and always answered my questions.

Title: Artificial Intelligence for Children of the Galaxy Computer Game

Author: Pavel Šmejkal

Department / Institute: Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: Even though artificial intelligence (AI) agents are now able to solve many classical games, in the field of computer strategy games, the AI opponents still leave much to be desired. In this work we tackle a problem of combat in strategy video games by adapting existing search approaches: Portfolio greedy search (PGS) and Monte-Carlo tree search (MCTS). We also introduce an improved version of MCTS called *MCTS considering hit points* (MCTS_HP). These methods are evaluated in context of a recently released 4X strategy game Children of the Galaxy. We implement a combat simulator for the game and a benchmarking framework where various AI approaches can be compared. We show that for small to medium combat MCTS methods are superior to PGS. In all scenarios MCTS_HP is equal or better than regular MCTS due to its better search guidance. In smaller scenarios MCTS_HP with only 100 millisecond time limit outperforms regular MCTS with 2 second time limit. By combining fast greedy search for large combats and more precise MCTS_HP for smaller scenarios a universal AI player can be created.

Keywords: artificial player, Monte-Carlo Tree Search, computer game, Children of the Galaxy

Contents

1 Introduction

In the recent years great artificial intelligence (*AI*) problems in games such as Go, and two player Texas hold'em poker have been solved (Silver et al. 2016), (Bowling et al. 2015). Even the best human players in these games are beaten by AI agents. The research is now turned more towards computer strategy games which pose new, greater challenges. In these games AI agents (*bots*) win against human players in small subproblems, usually concerning reflexes and speed as shown by the OpenAI Dota 2 bot[1]. However, bots are mostly incompetent when playing the whole game which also involves high-level decision making and planning (Yoochul and Minhyung 2017).

A lot of research has been done on real time strategy (*RTS*) games which simulate military operations, one of the most popular RTS being StarCraft (*SC*). However, in this thesis we focus on the genre of turn based 4X[2] strategy games. 4X strategy is a multiplayer computer game where the victory can be achieved in multiple ways such as eliminating all opponents in warfare using military units, discovering and conquering sizeable portion of the game map using civil units, or establishing cultural superiority over other players. To do this, players must gather variety of resources, research new technologies, and expand their empires. Games of this genre include Civilization IV[3], Galactic Civilizations III[4], and Master of Orion[5].

Compared to classical games such as Go, strategy games have much larger branching factors (even exponentially growing), their rules and goals are not so clear, and most actions do not have any apparent short-term effect. In addition, in 4X games players rule over a civilization and communicate with each other using diplomacy. This means that for an AI to be enjoyable to play against it should not only play optimally but also behave according to its long-term plans and motivations; i.e., behave human-like. Due to their play time, considerably longer than in RTS games, it is common to play 4X games against AI which puts much more pressure on the AI agents.

---

[1] https://blog.openai.com/dota-2/ [Accessed 18 April 2018]

[2] 4X stands for eXplore, eXpand, eXploit, eXterminate

[3] https://www.civilization.com/civilization-4/ [Accessed 18 April 2018]

[4] https://www.galciv3.com/ [Accessed 18 April 2018]

[5] http://masteroforion.com/ [Accessed 18 April 2018]

As the name suggests, in turn-based games players take turns to play the game and typically, players can play only during their turn. In 4X games there is usually no time limit for a turn, meaning that a turn can span indefinitely until the player chooses to end it. It may seem that this way we have much more computation time for the AI than in RTS games where we can see turns as frames and each frame is only about 16 to 40 milliseconds long. It is true, the AI in 4X games has more time, but it also has much more to do as we can see by simple comparison of branching factors which are estimated to about 35 for chess, 360 for Go, and 10^6 for StarCraft (Synnaeve 2012). In 4X games even a simple task of moving 10 units, which is just a small subtask of the whole turn, can get us to branching factor of $126^{10} \sim 10^{21}$. Opposed to RTS games which are almost exclusively played one on one, 4X games are commonly played in free for all format in about 8 players. Now, if we would allocate 60 seconds for all the AI players' turns (meaning 60 seconds in which the human player just waits) it leaves us with 8.6 seconds per AI, which is 215 times more time for tens of orders of magnitude higher branching factor than RTS games have.

As a result, AI agents in 4X as well as in RTS games are usually reactive and unable to beat human players unless given unfair advantage (more resources, unlimited vision etc.) which was frequently done in the past. Nowadays, the players want fair challenge; therefore, AI architecture in commercial games leans towards more complex algorithms, some of which we will explore in this thesis. AI in games is also interesting for academia since it poses unfamiliar problems which are unlike the ones in classical games.

In RTS games the research is motivated by annual competitions such as AIIDE StarCraft competition[6] or MIT's BattleCode[7], this also provides standard testbeds for this genre such as StarCraft and MicroRTS[8]. There is, however, no such thing for 4X games. Some research has been done on the Civilization series, but no annual competitions are being held in the academic sphere and standard testbed of the size of StarCraft does not exist.

As our environment we chose Children of the Galaxy (*CotG*), a 4X indie strategy game developed by Filip Dušek the main developer behind the EmptyKeys

[6] https://www.cs.mun.ca/~dchurchill/starcraftaicomp/ [Accessed 18 April 2018]

[7] https://www.battlecode.org/ [Accessed 18 April 2018]

[8] https://github.com/santiontanon/microrts [Accessed 18 April 2018]

studio[9]. CotG is still in active development, currently in Early access on Steam[10]. Source code for the AI part of the CotG is freely available on GitHub under MIT license but to develop and test it the original game is still necessary. This is to allow the community to improve the AI in the game since current AI is very basic. It is very convenient for us because we can implement interesting algorithms and at the same time develop an AI system to a commercial game and help the developers.

As such the goals of this thesis are following. Decompose the AI architecture for CotG and from this decomposition we choose a subproblem of micromanagement of units (moving individual units, e.g., to setup a battle formation or to avoid enemy fire) which is well suited for our thesis and valuable for the community and later game development. This problem usually involves military units; therefore, we will call it *combat*. We tackle this problem by implementing a simplified standalone simulation of the game since it is impractical to solve it using only the game itself. In this simulation framework several search techniques are implemented and empirically evaluated. We also adapt one of these techniques – Monte-Carlo tree search (*MCTS*) in script space and improve it by changing the way how playout value is calculated; therefore, providing more information to it. This way it better suits our combat goals (lose as few ships as possible, lose as few hit points (*HP*) as possible) and limitations (high branching factor and limited search time). This framework comes with full standalone testing mechanism which allows new developers explore the genre of 4X strategy AI.

Rest of the thesis is organized as follows: First, we present the current state of the CotG AI and our decomposition of the game from the AI's perspective. Next, we look at related work to the subproblem of combat in strategy games. Then we lay out our general approach and viable solutions to this problem. After that we discuss limitations of these methods and our improvements. Then we present our architecture and implementation of these techniques in the context of CotG. We conclude by evaluating different combat algorithms in the environment of our simulator and discussing the results.

[9] http://emptykeys.com/ [Accessed 18 April 2018]

[10] http://store.steampowered.com/ [Accessed 18 April 2018]

2 Focus analysis

In this chapter we present the game Children of the Galaxy. Then the current AI architecture in CotG is discussed. Then we decompose the game form the AI perspective and present a subproblem which is the focus of this thesis.

2.1 CotG from player's perspective

Before we explore the game from the AI perspective let us first explain the game and define some terminology. The game is situated in a single galaxy where each player controls one of three races. To win the game, a player can either eliminate all opponents, colonize most of the galaxy, or research and build a Dyson sphere.

The game is played on a grid of hexes which can be seen from two views representing two levels of detail. In a *galaxy view* (in Figure 2.1) players can see individual solar systems. Each hex in this view can be magnified to a *solar system view* (in Figure 2.2) where players can see a single solar system with its star and planets. In the solar system view units can move and fight each other. Planets rotate around their stars and can be colonized (if unoccupied) or attacked (if in enemy's possession). From here units can go to *warp* which moves them to the galaxy view where they can travel greater distances but cannot interact with each other. A unit in the galaxy view can *explore* the hex it occupies, which moves it to the solar system view of this hex. This can be done even if there is no star; in that case there may be asteroids or just an empty space.

Each colonized planet has stats such as population, production, and research. Players may build structures on their planets to improve these stats. Each planet can also produce units which are either civil (colonization ship, builder) or military (battleship, destroyer etc.). Military units cost rare resources which motivates players to colonize new planets and mine on asteroids as well as it limits the number of military units a player can have. Players can also research technologies in rich technology tree. Some technologies will allow them to build new types of military units where other technologies may permit colonization of new planet types.

Figure 2.1 – Galaxy view of the game. Color of the hexes indicates which player's region they belong to. In the top right corner, we can see a mini map of the whole galaxy. Hexes with stars have a label with the star's name and solar system details next to them.

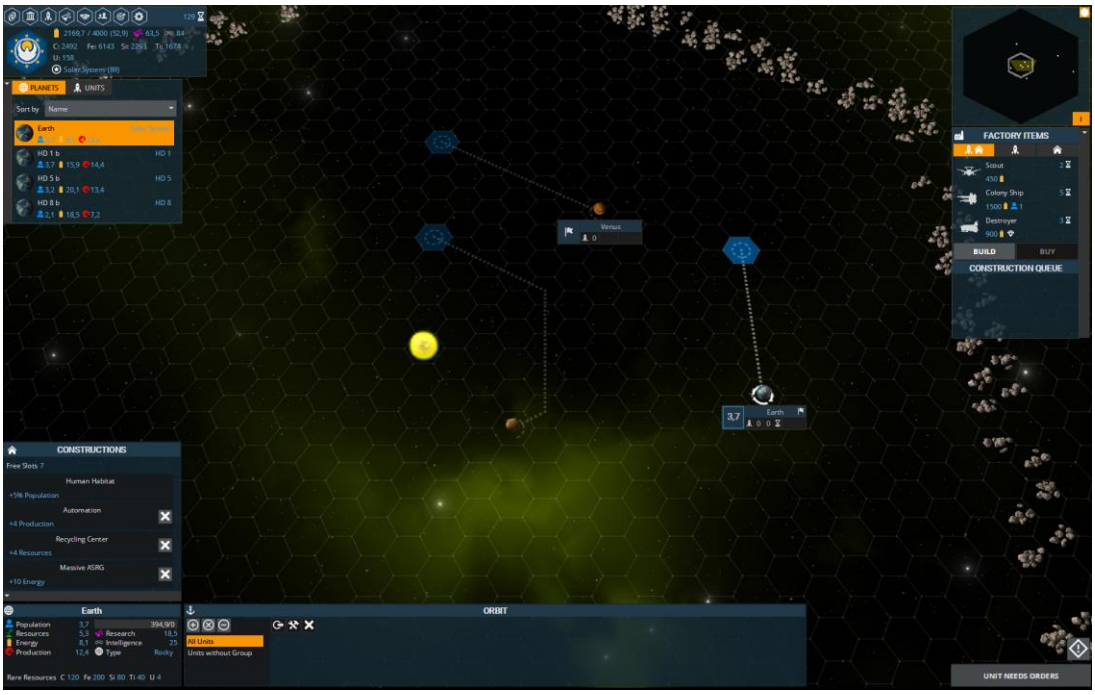

Figure 2.2 – View of a single solar system. We can see a star, asteroids, and planets with indicators of their positions on the next turn.

2.2 CotG AI architecture

To design, program, and release a 4X strategy game is immensely challenging task for a single developer to achieve. CotG is mostly made by a single developer and

therefore, some compromises must have been made. One of these is rather smart decision to offload the work on the AI system to the community by making the AI library opensource. Even then a game with no AI and no structure to program one would have a very hard time. Therefore, a framework for AI is provided and a basic AI is implemented in it by the developer.

The whole AI is programmed using behavior trees (Champandard and Dunstan 2012) which are one of the modern standard ways to implement an AI in games. Behavior trees have a great advantage of clear visual representation of the code flow and therefore, allow a designer to make the AI behave as he wants. This, however, leads to some pitfalls such as the fact that this AI can be a little too predictable since it only knows what the designer implemented. To partially avoid this, the behavior trees have some non-determinism in them – sometimes the AI chooses its next step randomly with some bias. It is also difficult to maintain such an AI during the development when features and game mechanics may change drastically. Another important thing is that the actual execution nodes in the tree are C# classes which are not limited in any way and can do an arbitrary amount of computation.

Another limitation of the current system is that to save time, memory and code complexity, the behavior trees do not implement a *running* state. This state basically saves the state of the tree and when it is next time queried it continues from the *running* node. The running state, however, is one of the biggest advantages of the behavior trees and its omission degrades them in some ways to much simpler "if-then rules" (Gemrot 2017).

2.3 Decomposition of 4X game from the AI perspective

To decompose the game, we studied how other strategy games solve the AI problem. Bots in RTS games (such as UAlbertaBot[11] in StarCraft) often decompose the game to, at first glance, independent problems; e.g., strategy, production, micromanagement etc. Each of these problems then has a manager and these managers communicate with each other. This is problematic, because no two parts in such a complex game are truly independent and sequential update of all managers soon proves too complex to develop and maintain. Much better solution to the AI problem would be to coordinate these

[11] https://github.com/davechurchill/ualbertabot/wiki [Accessed 18 April 2018]

managers and distribute resources based on their priorities. Such a system of tasks and priorities is presented in the Total War series and mentioned in (Welch 2007).

We can divide this architecture into three main parts; static systems, task systems, and production and research systems. Static systems can be asked for information about current game state such as enemy presence using influence maps, or they may provide static computations.

In 4X games each civilization usually has a different personality. This projects to their end game goals and these are realized by production and research which are managed by their respective systems. In Rome II: Total War each civilization has a predesigned template of what does it seek to do in the game. E.g., Roman empire may be designed to care mostly about military and in that field, it may prefer infantry to cavalry. When asked what to research next the agent tries to match these preferences. If it has too much military more cultural or civic research is done etc.

The tasks system composes of many simple task generators which produce tasks based on their preferences. A task is a small goal without a specific plan, e.g., attack Cartago, defend Rome etc. These tasks are then rated by all the managers which leads to a list where each task has a priority and resources needed for its execution. A key point here is that different task generators may generate the same task which gives us a sort of voting system. We then want to select the best subset of all the tasks. In Rome II: Total War this is done using MCTS algorithm. To explore this system in its full detail please refer to (Gosling and Andruszkiewicz 2014).

2.4 Application to Children of the Galaxy

To implement such a system in CotG we need to look at the game from a high-level perspective. We suggest decomposing the AI as follows. Divide the game to task systems, production and research systems, and standalone systems as seen in Figure 2.3.

Standalone systems are either static or are updated once an action is performed. One of them is *Micro estimator* which, when queried estimates outcome of given battle and recommends best unit moves. Another one is *Strategic context* which contains information about long term plans. Its main component is a Threat map which is an influence map of enemy presence which can be queried when reasoning about defense of friendly regions.

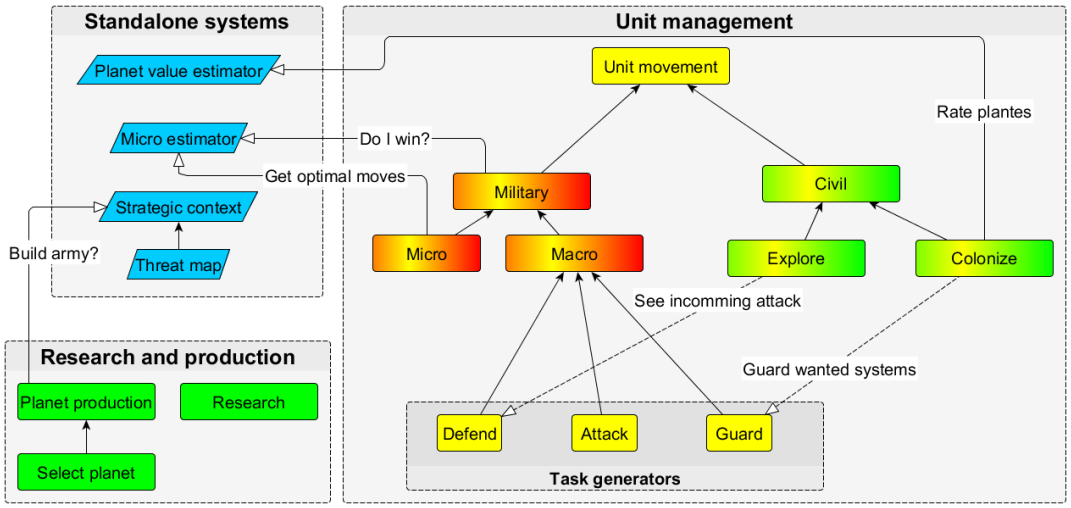

Figure 2.3 – High-level decomposition of CotG AI. Blue are standalone systems to be queried. Green are production and research systems not handled by tasks. Yellow and yellow with gradient are task managed systems. Dashed arrows indicate hints to Task generators. Full arrows with hollow point indicate query. Regular arrows indicate composition.

Production systems query Strategic context to see whether army production should be prioritized. If not, they behave according to predesigned templates based on behavior desired for each civilization. Resources are limited and therefore, it is important to produce units on right planets; that is done by *Select planet* system. *Research* system is entirely template based.

Task systems mostly care about units positioning which involves attacking, colonization, defending etc. We can split it into two parts which do not share resources (units). Colonization plus exploration which uses Colonization ships and Scouts. And Military which uses all kinds of military units. Tasks which can be generated are Defend, Attack, and Guard.

2.5 Outcome and focus

When designing programmatic solutions for this system we realized that the quality of most decisions would heavily depend on the quality of the Micro estimator (it is important to know the risk when sending a fleet somewhere). E.g., we could value some task very highly but with bad combat estimation we could easily over or under estimate our chances and then commit to all the wrong tasks. Therefore, instead of relying on a bad micro estimator we decided to solve the problem bottom up and implement a reliable one. Another advantage of implementing Micro estimator lies in the fact that it is a relatively standalone system; therefore, it would not be so prone to breaking during the ongoing game development. E.g., when implementing tasks, even

minor changes in game mechanics and diplomacy systems could break it. Micro estimator on the other hand relies only on the core mechanics of movement and fighting.

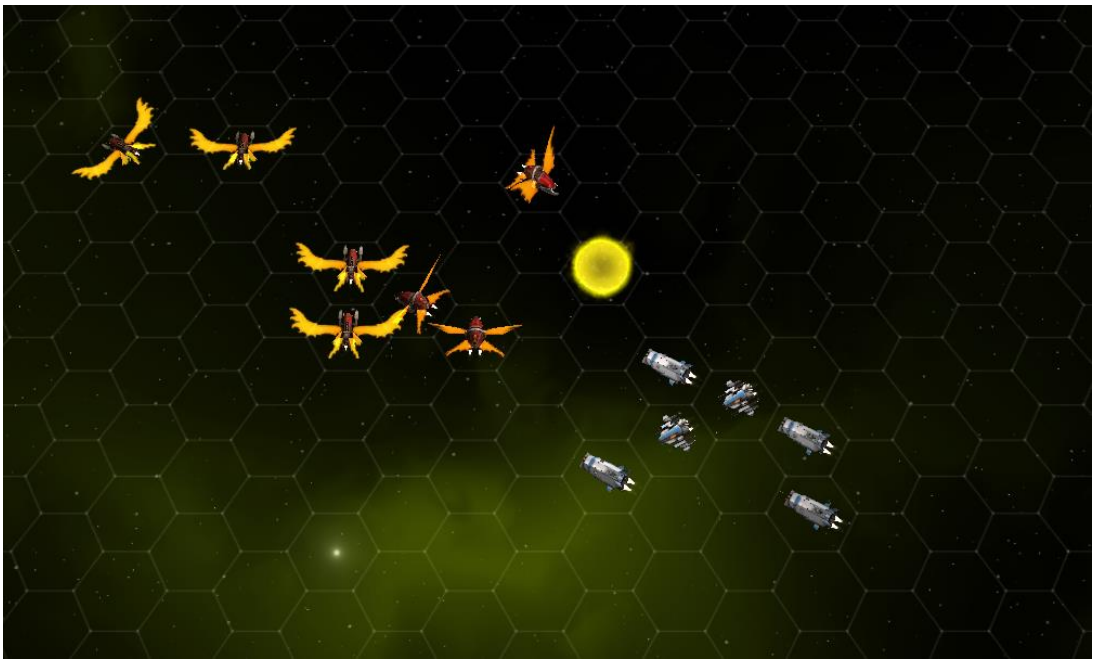

Figure 2.4 – Solar system view with a small combat scenario. Yellow/red ships are one army and blue/grey ships the other army.

The Micro estimator solves the problem of combat which takes place in the solar system view as seen in Figure 2.4. In CotG, combat is a tuple (U_0, U_1, P_0, P_1, E) where U_i is a set of units for player i positioned in the environment E. P_i is an agent controlling units U_i. And E is an environment in which the combat takes place. Environment contains information about the position of the sun, impassable hexes, passable hexes, asteroids, and planets. A combat is played in turns. At turn k agent $P_{k \bmod 2}$ performs all actions it wants (moves units, attacks enemy units etc.) and then turn $i + 1$ is initiated. Player wins combat when all enemy units are eliminated. From the turn-based nature of the game, draws are not possible in combat. This means that combat is a deterministic, turn-taking, two-player, zero-sum game with perfect information as defined by Russel and Norvig (2010).

The focus of this thesis is to implement a standalone micro estimator component to the game. Analyze possible approaches to combat in strategy games and implement viable solutions. Design and implement evaluation framework and workflow, to compare our solutions. Demonstrate possible integration of the combat component to the current game AI system.

3 Related work

Although research in RTS games is very active and many testbeds and competitions exist, the research in the field of 4X strategy games is rather sparse. Most of it is done on the Civilization series and its clones and it is mostly unrelated to our work because it solves much higher lever problems such as city placement as in (Wender and Watson 2008) or generally playing the game, e.g., in (Branavan et al. 2011).

In this section we first present similarities of RTS games to turn-based strategy (*TBS*) games and show that some research done in RTS can be applied to our problem. Then we present one way to solve the problem which is search and show several search algorithms used in strategy games. After that, other methods than search are shown. We conclude by showing related work in field of game simulators and testbeds for strategy games research.

3.1 Similarities of 4X to RTS combat

Because combat is much more low level than general game playing we can relate it to micromanagement in RTS games. At the first glance, the biggest difference between TBS and RTS is the continuity of time. However, computer systems work in discrete fashion and so do computer games. Therefore, in RTS games players' actions are executed in order – all actions of the first player, all actions of the second player. This means that for the AI agent RTS is like TBS, but each turn is called *frame* and has fixed time limit of 16 to 40 milliseconds. The difference lays in the granularity of actions. In RTS the actions are much smaller such as to move several pixels whereas in TBS actions involve moving several hexes or full execution of attack.

If we look at CotG and for example StarCraft we can find even more similarities. In both games, if we look only at combat we have a set of our units, set of enemy units and some environment with terrain and obstacles. Units collide with each other and various units move at different speeds, have different health pools, and damage. In SC there is limited vision (fog of war) and some units have special abilities, but these factors are almost never accounted for in research. Movement in SC is limited to one step per frame and units have acceleration, deceleration, and turn speed. This is however, again usually simplified in research and in simulators as well as projectile speed for ranged attacks. These simplifications are done to speed up the simulation and to enable more iterations.

3.2 Scripted approach

AI in commercial computer games often servers a few simple purposes. Companions in adventure games should follow the player and occasionally provide hits. Enemies in first-person shooter games should appear to behave organized with the right amount of shooting around the player's head while getting killed by him. In RTS games, the campaign AI should just tell the story while the player wins all the missions. Even in games considered difficult such as Dark Souls[12] or Cuphead[13] the enemies are controlled by a simple AI which allows the player to learn its patterns and defeat it. All this is possible by carefully designing what can the AI do and when does it do it. It would be rather disappointing if in a military WW2 game, the AI decided that peace is a better option for Germany after all.

In these games the AI is usually programmed using static rules. Common architectures are *finite state machines* or *behavior trees*. In these architectures the behavior is usually divided into scripts which are then selected at the right time (using static rules) and executed. Examples of such scripts may be *GoToPlayer*, *Flank*, *Attack, Retreat* etc. Within these scripts pathfinding or best target selection may be performed. Then, when a predefined condition is met, e.g., player is in weapons range, appropriate script is executed, e.g., shoot the player. In our work we use scripts as well and more formal definition is provided in section 4.4.1. In some situations, however, we can see that this is not what we want to do. Maybe the player is too strong, and the AI agent is alone, maybe at that point waiting for reinforcements would be the better thing to do and unless the designer specifically programmed this behavior it will not be executed. But as we said earlier, we often value predictability over optimal behavior in games. In some games, however, it is the other way around.

3.3 Search approach

Strategy games are usually played because of the game mechanics and the challenge. In these games a good AI playing optimally is desired and search techniques are commonly used. Some commercial games using search for their AI are Frozen

[12] https://en.bandainamcoent.eu/dark-souls/ [Accessed 18 April 2018]

[13] http://www.cupheadgame.com/ [Accessed 18 April 2018]

Synapse[14], Rome II: Total War, F.E.A.R. (Orkin 2006), Fable Legends (Mountain 2015). Search techniques have also an added benefit of automatic adaptation when the game mechanics slightly change, which happens quite often in computer games for balancing purposes.

3.3.1 General structure of the problem

Since the problem at hand has similarities with chess, one of very prominent ways to solve it in RTS games is using adversarial planning to find a sequence of actions. Advantages to this approach are natural agent adaptation to current situation and the fact that search algorithms are mostly domain agnostic and require less expert knowledge as pointed out by Churchill et al. (2012).

Generally, search-based approaches build a tree where each node represents a game state. Children of a node are game states generated from the parent node by applying all valid actions to the parent game state. Even levels of the tree correspond to the first player's actions and odd to the second one's. E.g., each node is generated by a so-called *player-action*, i.e., an action a player makes. In games where the player may control multiple units or can generally do more actions in his turn this player-action is composed of multiple so-called *unit-actions* (Ontañón 2013).

In simple games such as tic-tac-toe we can recursively build the whole game tree where the current game state is in the root and leaves correspond to terminal states. Then we can fully search this tree and always chose moves that lead to optimal outcome (victory). In more complex games such as chess more cleaver ways to search the tree were invented such as iterative deepening where we stop at certain level and evaluate quality of given game state using some heuristic function. Another, recently very popular search method is Monte-Carlo tree search which performs asymmetric search.

Search algorithms solve the problem of which player-action to choose in current game state by looking ahead and trying multiple actions and opponent reactions. Now we can look at application of these methods in the domain of strategy games.

[14] http://www.mode7games.com/blog/2010/11/02/artificial-intelligence-2/ [Accessed 18 April 2018]

3.3.2 Search algorithms

Churchill et al. (2012) try to solve combat in the RTS game StarCraft using improved Alpha-Beta pruning called Alpha-Beta Considering Durations (*ABCD*). This algorithm addresses the fact that not all units can perform actions each turn (some actions need to "cool down") and how do deal with simultaneous actions in the search tree. It utilizes move ordering where first few actions in a new node are proposed by scripted behaviors and the rest is ordered in some predefined manner. It also utilizes iterative deepening where the heuristic function is a deterministic playout using static scripted behavior. The authors show that ABCD, with proper heuristics, outperforms random and scripted behaviors in small scale combat – from 2 to 8 units on each side in real-time.

As a continuation, Churchill and Buro (2013) present two new search algorithms. The first one is Upper confidence bound for trees (*UCT*) Considering Durations (*UCTCD*) which is standard UCT algorithm which, same as ABCD, considers durative and simultaneous actions. The second algorithm is called Portfolio Greedy Search (*PGS*). It is an iterative algorithm which works with a portfolio of predefined behaviors (scripts). Given a game state, it assigns the same script to all units. Then it iteratively, for each unit changes its script, performs a playout and if it ends better than the previous assignment it keeps this new script; if not, the last script is used. This is called *improvement* and it is done in turns for both armies – first, one army is improved and then the other one. We implement this algorithm and explain it in greater detail in chapter 4.4.3. In the paper the authors show that this approach outperforms state of the art approaches such as ABCD and UCTCD for medium (16v16) to large scale (50v50) combat scenarios in real-time.

Further improvements to the UCT were proposed by Justeen et al. (2014). Inspired by PGS they modify UCTCD to search in the space of script assignments instead of action assignments. This means that unit-actions in each player-action are generated only using predefined scripts. We base our algorithms on this approach and explain it in chapter 4.4.2. Another improvement proposed in this paper was to create clusters of units and assign the same script to all units in each cluster. These novel approaches proved ineffective in small scale combat but highly effective in medium to large scale combat against UCTCD. Performance against PGS was not tested.

In most player-actions there are some good and some bad unit-actions. E.g., unit attacking and destroying enemy's key unit is a good action. We would like to identify these unit-actions and when creating new player-actions in the search tree we would like to prefer these "good" unit-actions to the bad ones. E.g., we think that attacking with unit U_1 is a good action but we are not sure what to do with unit U_2; therefore, we want to try to stick with attack for U_1 and try different actions for U_2. This was motivation for so called Naïve sampling for Combinatorial multi-armed bandit (*CMAB*) problems done by Ontañón (2013) and showcased in a custom game called *MicroRTS*. The approach decomposes a single CMAB where we search a tree of player-actions for n units, to $n + 1$ Multi-armed bandit (*MAB*) problems. One for each unit (with unit-actions) and one global for player-actions. The unit-action trees serve for exploration of new actions whereas the global player-action tree serves for exploitation and evaluation of currently known player-actions. To sample a tree means to somehow select a leaf node from it. At each step of the algorithm we choose (in $\epsilon - greedy$ fashion) if we sample the unit-action trees or the global tree. If we decide to sample the unit-action trees we end up with n unit-actions from which we compose a new player-action which is added to the global tree. If we sample the global tree we get a single player-action as a result. From this player-action a playout is performed, and results of this playout are backpropagated in the global tree as well as in the unit-action trees. When sampling a unit-action tree, at each node we either create a new unit-action or we take the current best unit-action (again $\epsilon - greedy$). When the global tree is sampled we again in $e - greedy$ fashion go through the tree selecting best vs random player-action. This approach was shown to be competitive to classical UTC and ABCD algorithms in small scale game and outperforms them in medium scale. Further improvements of this algorithm motivated by the success of AlphaGo (Silver et al. 2016) were shown by Ontañón (2016) where Bayesian learning was used to rate quality of actions which is then used in the search. This improved algorithm outperforms the original one and is suitable for games with even larger branching factors. In the largest scenario the median branching factor was 112 and maximum branching factor encountered was $2.9 * 10^{15}$.

3.4 Other approaches to combat

In this section we present three methods not utilizing adversarial search. One of the most used pathfinding algorithms in games A* is usually used in search-based approaches. A* has some great qualities such as that the path found is guaranteed to be optimal if the heuristic function used is admissible. It is, however, still a linear algorithm which in the worst case (when no path exists) needs to go through all nodes. Another approach to the spatial reasoning and pathfinding problems are potential fields which are suitable for navigating many units through an environment or for repeated navigation. Liu et al. (2013) apply potential fields with genetic algorithms to combat in StarCraft. In their case the potential field has several parameters which control unit interactions, e.g., friendly units should be repulsed a little to avoid collisions, and they should be attracted to enemy units in some way and repulsed in another way to achieve the best combat positioning. To determine these coefficients, the authors compare hill-climbing approaches which are fast but do not yield high quality solutions to genetic algorithms which return much better solutions but are slow. A good balance between quality and speed is achieved by using Case-injected genetic algorithms. The targeting problem is entirely left to the default StarCraft AI and is not solved in context of this work.

Similar approach to potential fields – influence maps are used for combat reasoning by Bergsma and Spronck (2008) in their custom made 4X turn-based strategy game *Simple wars*. In their work, each type of object in the game environment has an influence map assigned. These influence maps are then combined (layered) by a neural network. Output of this layering algorithm are two influence maps, one which indicates the preferability to move to each tile and another one doing the same for attacking each tile. The reasoning algorithm then simply selects the tiles with the highest influence from all possible targets for the move and attack actions. To determine the influence spreading and the weights for the layering an evolutionary algorithm was used. This approach was then compared to fixed scripted strategies which it outperformed as was shown on a small-scale combat scenario.

Reinforcement learning techniques are very popular due to their universality and the fact that they are unsupervised. In (Wender and Watson 2012) the authors use variations of Q-Learning and State-Action-Reward-State-Action algorithms to learn hit and run behavior in StarCraft. In their scenario one fast unit is put against several

slower units which can easily destroy the fast one when it faces them directly. The fast unit should learn the hit and run behavior and destroy all the enemy units. The experiments were performed in the actual game (not in a simulation such as SparCraft) where the learning algorithm receives a simplified state and only two actions Fight and Retreat. These actions are implemented via scripted behaviors, e.g., Fight action when executed tries to attack the weakest enemy around and the Retreat action calculates retreat vector based on surrounding environment and units. The resulting behavior can win the scenario almost all the time when given enough time to learn.

3.5 Game simulators

One big disadvantage of search algorithms is that a forward model of the game is needed, this means that we need to be able to perform actions on a clone of a game state or we need to have the ability to revert these actions. Game state usually has neither of these properties. Furthermore, game state and actions usually contain much more information that the AI needs and using these in the search algorithms would severely hurt the performance. Therefore, when developing a search-based AI another simplified game simulation is developed as well for the AI to work with. This has an added benefit that we can easily evaluate different AIs in separate standalone applications.

To evaluate search algorithms in StarCraft a simulator called SparCraft was introduced by Churchill et al. (2012) and was used extensively in research. It is a standalone simplified simulation of combat in StarCraft. Many features of the full game are missing such as: units do not collide with each other, no obstacles are present, units have no acceleration and turning speed, the entire battlefield is revealed, projectiles have no travel time and others. This limits its utilization in the real game because it cannot be queried to return the next best move (since most of its moves are not valid in the real game). For example, UAlberta bot uses SparCraft just to estimate which army is stronger. Lack of collision detection greatly speeds up the computation, but it also simplifies the game too much since arbitrary number of units can stand at the same position. The simulation of units is also not entirely accurate because it was implemented without knowledge of the original game's source codes as discussed by Schneider and Buro (2015). A Java implementation called JarCraft also exists (Justesen et al. 2014).

A different approach to research AI in RTS games is to entirely ignore existing games and implement an open-source game with simplified rules and graphics which is designed AI-first. Examples of such games are MIT's BattleCode or MicroRTS introduced in (Ontañón 2013). Using such games has an undisputable advantage of controlled environment because we have access to the source code and the game can be designed AI-first. This means that we may not have to have two separate implementations of the game mechanics and game state. The game code, however, may not be production grade since it is usually developed only by a handful of researchers. These games are not designed to be played by human players; therefore, no human expert players exist to compare the AI agents against. This also means that no datasets of game logs of human players playing the game exist.

3.6 Conclusion

In the previous section we looked at various methods used in specific games and made an overview of techniques which are relevant to our work. As we can see, problems in strategy games are mostly very large and general algorithms need to be heavily modified and adapted to specific games. In the context of this thesis, we have chosen to explore and adapt search-based techniques because of their high potential and flexibility to ongoing game development of CotG. They also fit very well to the current behavior tree-based AI architecture. Due to the complexity of the game and immense branching factor we use combination of search and scripted approaches such as PGS and modified MCTS. Since these approaches need a forward model we also develop a simplified game simulation which provides such a model and allows us to benchmark different AI methods independently on the game.

4 Search techniques for modeling combat

In related work we overviewed several search methods to the problem of combat in various strategy games. In this section we focus on a single game – CotG which is described and analyzed in sections 4.1 and 4.2. Then we describe domain-agnostic search techniques we decided to implement it this context. Since we use modified versions of MCTS algorithm the original version is described in section 4.3.2. Then we show that the search can be done not only in actions but also in scripts which is a focus of section 4.4 where we first ground what we mean by scripts (4.4.1) and then MCTS in script space (4.4.2) and previously mentioned PGS (4.4.3) algorithms are explained in greater detail. We conclude by introducing our own improvement of the MCTS algorithm – MCTS considering HP which was designed for CotG domain, but it should be generalizable to different games as well.

4.1 CotG combat domain

First, let us briefly introduce the problem of combat in CotG and ground some basic terminology, more formal and abstract definition was presented in section 2.5. As we said earlier CotG can be played in more than two players. We, however, simplify the combat to a two-player game because in most scenarios players are either cooperating or they are neutral, e.g., a player is not in war with us and is just a bystander to the conflict. Scenarios such as 1vs1vs1 are not very common.

Combat in CotG takes place in the solar system view (see 2.1) and is played (as the rest of the game) in turns. During his turn, a player can issue orders to his units (*ships*). Ships have different stats, for us the most important are *hull*, which is also referred to as hit-points (*HP*) which represents the amount of damage a unit can take before being destroyed. *Shields* which reduce damage done to hull when a unit is attacked. *Damage* which is a strength of attack of given unit. *Attack-range* is the reach of unit's weapons. *Power* represents movement capabilities of a unit.

Orders important for combat are *Attack* and *Move*. Attack can be issued to a unit with positive damage (i.e., has weapons) which did not attack this turn, and which has enemy units in its attack-range. Friendly units cannot be attacked. Move can be issued to a unit with positive power. For each hex a unit moves it loses one power. Power is recharged at the start of each turn. E.g., a unit with power 4 can move 4 hexes each turn. Minimal move distance is one hex.

Orders can be issued repeatedly and to different units. E.g., we can order a unit to move 1 hex away, move back, attack, move away again etc. as long as these orders are valid (it has enough power and attack capabilities). Orders can also be interleaved, e.g., we can issue some orders to unit 1, then to unit 2 and then again to unit 1 if these orders are valid. Anytime during the turn a player may end the turn. This means it is enemy's turn now and the player can not issue any more orders until his next turn.

4.2 Branching factor analysis

To be able to decide which search techniques are viable for our domain we first need to analyze its complexity which for search algorithms usually means branching factor of the search tree. Consider a movement with a single unit on a hex grid. For unit which can move n hexes far the number of reachable hexes is given by a simple formula:

$$hexCount(r) = 3 * r * (r + 1) = 3r^2 + 3r \qquad \forall r \in \mathbb{N}$$

Visualization of the range is shown in Figure 4.1.

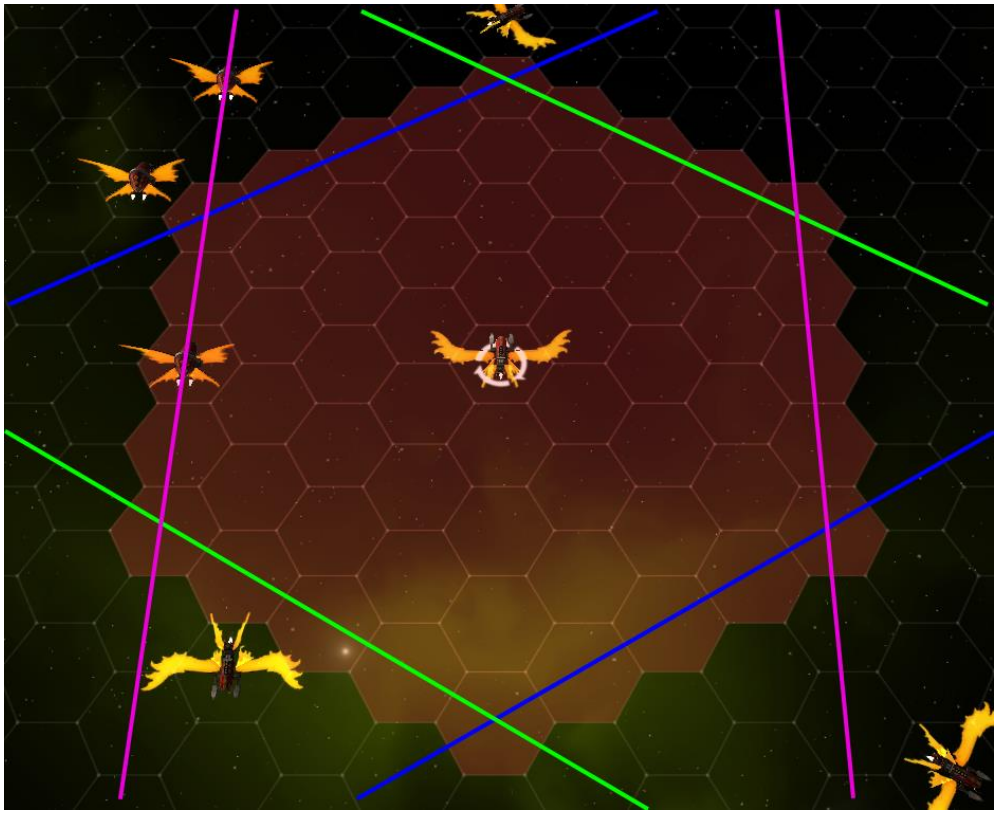

Figure 4.1 – Red hexes show range of 4 around a unit. This range is determined as an intersection of six half-planes indicated by parallel lines on the image. Magenta lines show $\boldsymbol{x}$ axis limits for distance of 4. Blue lines are $\boldsymbol{y}$ axis limits and green are for $\boldsymbol{z}$.

This gives us 60 possibilities for a single unit's movement for a unit with power 4 and thus a branching factor of 60 when moving a single unit. Now, let us consider so called "hit and run" behavior. E.g., a unit moves close to an enemy, attacks and moves away. If we move 2 hexes towards an enemy unit we have 18 possible positions. There may be k enemy units we can attack and then we may want to retreat again to 18 positions. Which leaves us with $k * 18^2$ possible ways to perform this maneuver.

If we add more units, their actions may interleave. E.g., unit A goes from $(0, 0)$ to $(1, 1)$, unit B goes from $(2, 2)$ to $(0, 0)$ and unit A goes from $(1, 1)$ to $(2, 2)$. This is important if we want to regroup units.

To simplify things, we force units to perform actions in a fixed order and each unit must perform all its actions at once. A player-action is in our case a vector of all unit-actions[15] issued by a given player at a given turn (Ontañón 2013). E.g., for a set of n units $\{u_1, u_2, \dots, u_n\}$, a player action is a vector of unit-actions $(a_1, a_2, \dots, a_n)$ where an action a_i corresponds to an action of unit u_i and actions in this vector are executed in order. Unit-action is a function $a: G \rightarrow G$. Where G is a set of all possible game states. Unit-action a for a given game state returns new game state where action a is performed.

Actions correspond to orders described earlier (see 4.1); therefore, we have two actions, move and attack. Move issues a move order to a unit from source hex to a target hex. Attack issues an attack order to a unit from source hex to a target hex.

Each node in our search tree corresponds to a player-action. The resulting branching factor for this search tree is:

$$\prod_{u \in units} actionCount(u)$$

Where $actionCount(u)$ returns number of actions a unit u can perform this turn. E.g., for 7 units where each one can only move 4 hexes away and fixed execution order we would have $60^7 \approx 2.8 * 10^{12}$ actions – branching factor. The simplifications we present greatly reduce our tactical capabilities and the resulting branching factor is still very large; therefore, we need select smart search techniques which can deal with this complexity.

[15] Note that unit-actions do not have to be atomic, e.g., (move), (attack); but they can be also composite, e.g., (move, attack, move) etc.

4.3 General tree-search techniques

In this section we briefly present evolution of adversarial search techniques from simple Minimax algorithm to regular MCTS algorithm which is basis for our methods presented in later chapters.

4.3.1 Minimax and Alpha-Beta pruning

One of the most basic adversarial search techniques is algorithm called Minimax (Russel, Norvig 2010 p.165). Consider a game played by two players, *Max* and *Min*. In the game, *Max* starts and then players take turns. Points for winning are awarded at the end of the game. At each point of the game the player on turn can choose one of several actions to make which implicitly creates a game where the nodes are states and edges are actions. If we create this whole game tree we can award points to the leaf nodes and these points can be then propagated upwards through the tree where at each level, we select either maximum of children (if *Max* chooses an action) or minimum of children (if *Min* chooses an action). The minimax algorithm creates such a tree and then at the root level, when next action to play needs to be selected it returns the action with maximum value (if playing as *Max*). Because the tree is fully expanded this action is the optimal play in given situation.

The Minimax approach is usually demonstrated on 3x3 tic-tac-toe game where there are fewer than 9! terminal nodes. However, on larger problems such as chess where there are over 10^{40} nodes the fully expanded game tree is impossible to construct. In these cases, a simple yet very powerful enhancement of Minimax called Alpha-Beta pruning is employed (Russel, Norvig 2010 p.167). It is based on an observation that some branches will never be selected by rational players and can be safely pruned away, while still returning optimal solutions. To avoid the need to reach a terminal state to be able to propagate the result, a heuristic evaluating a non-terminal state is commonly used and the search is terminated early (in that case optimality is no longer assured).

Alpha-Beta pruning with enhancements such as iterative deepening, transposition tables and move ordering is known to perform well in games with smaller branching factors where a good heuristic function is known such as chess (Černý 2016 p.76). In strategy games some heuristic functions are known, such as sum of unit values or scripted playout. The bigger problem is the branching factor of these games which is enormous even after simplifying the game as we saw in section 4.2.

4.3.2 Monte-Carlo tree search

MCTS is an algorithm which tries to stochastically determine the value of nodes to avoid building the whole minimax tree. Opposed to Alpha-Beta pruning, search done in MCTS is not uniform but rather guided towards the most promising nodes which allows it to handle much bigger branching factors. MCTS algorithm starts with only the current game state as the root node. It is iterative, and each iteration has four steps:

1. *Selection* – the game tree is traversed from the root. The most promising child is selected recursively, until a node which is not fully expanded (does not have all possible children) is found.
2. *Expansion* – a new child node is created for the node found in selection.
3. *Simulation* – a random playout from the new child node is performed.
4. *Backpropagation* – the result of the playout is propagated from the child node to the root node. Number of playouts and wins is updated for each node along the path.

To select the most promising child various techniques may be used, probably the most common one is called *Upper-confidence bounds for trees* (*UCT*). It sees the child selection as arm selection in n-armed bandit problem (Auer et al. 2002). The algorithm selects child node which maximizes the value of upper-confidence bounds formula UCB1:

$$UCB1(i) = \frac{w_i}{n_i} + c * \sqrt{\frac{\ln(n_p)}{n_i}}$$

Where i is the current node, w_i is the number of wins in the current node, n_i is the number of visits of the current node, n_p is the number of visits of the parent of the current node, and c is a constant usually determined empirically.

The biggest advantage of MCTS is that the search is asymmetrical which helps to reduce time spent with bad moves and is key to handle huge branching factors. Another important feature is that no heuristic function is needed. The algorithm estimates the value of a position based on the number of ways it is possible to win from it. MCTS can also be stopped at any time giving the best result for the time it has been given. This is crucial feature for real-time applications, such as games. An extensive survey and background of MCTS methods can be found in (Browne et al. 2012).

Application of plain MCTS to CotG has two key issues. The first one is that if we were to perform a random playout it may never finish. In games such as Go, each action the player performs moves it one step closer to the end of the game and the number of turns is limited to several hundred. In CotG there is no turn limit like that and most actions just move units. Units which recharge shields over time may be never destroyed. This could be solved by replacing random playout by a guided one which would prefer some actions over others. The second issue is that MCTS requires action generation and playouts to be as fast as possible to have many of them to be able to guide the search. In games such as Go, valid actions are relatively easy to enumerate and state update after choosing one may be done fast as well. To enumerate all move actions in CotG we need to perform some sort of pathfinding to see which ones are valid. Caching the pathfinding may help but it would be very memory intensive and as we discuss later, in dynamic environments such as CotG caching may even slower the search. Same goes for caching of game states in the search tree, due to the sheer number of them it would not be very advantageous. If we combine this with the branching factor which is tens of orders of magnitude higher than Go this seems like a dead end.

Other search methods we mentioned earlier such as Naïve MCTS could work but even there the algorithms were shown on much smaller problems and a cleaver move ordering was necessary.

4.4 Search in script space

If we take a high-level look at actions in CotG combat, we can see that many actions fall into the same category and from each category only a few actions seem reasonable to perform. Imagine two units are in combat, one is severely hurt and the only thing it can do is flee. If the search algorithm chooses moves that go closer to the enemy, the unit will be destroyed fast and these will quickly prove bad. But from the moves which move the unit away from enemy most moves are good, but apparently (in large open space) the best one is to move as far as possible. The search algorithm does not have this knowledge; therefore, it must try all the moves at least once and this in turn unnecessarily increases the branching factor. The situation is obviously much worse for more units. But what if we could help the search algorithm choosing from redundant moves by implementing common behaviors and performing local search? We do this by using scripts.

4.4.1 Scripts

Scripted behaviors are the simplest and most commonly used in computer games (Churchill et al. 2012). In this work we focus on scripts controlling a single unit although scripts controlling groups of units are also possible. There is no strict definition of what script can or cannot do, but in our case, we work with scripts which are modular, state-less, and have a clear goal.

We define script as a function $s: G \times \mathbb{N} \rightarrow A$ where G is a set of all possible game states and A is a set of all possible actions; i.e., for a game state and index of a unit it returns an action this unit should perform. Some examples of scripts are:

- *Attack-Closest* – Find the closest unit and attack it. If the closest unit is not in weapons range, go towards it.
- *Attack-Weakest* – Find the weakest unit in weapons range and attack it. If there is none, go to the closest unit.
- *Attack-Value* – Find unit u with the highest $v(u) = \frac{damage(u)}{hp(u)}$ value in weapons range and attack it. If there is none, go to the closest unit.
- *No-Overkill-Attack-Value (NOKAV)* – Similar to Attack-Value but if a target unit was already assigned a lethal damage by another unit we ignore it. E.g., if two units were to deal damage to the same enemy and the enemy would die after the first damage is done, the other unit's damage would be wasted – we try to avoid this situation.
- *Kiter* – If there are no units in weapons range, go to the closest unit. If there are units in weapons range, attack the one NOKAV selects. If you attacked, move away from the enemy.
- *Regroup* – Find the center of gravity of positions of all friendly units and move towards it.

Note that there are multiple ways how to design the kiter behavior. For example, Churchill et al. (2013) use Attack-Closest instead of NOKAV target selection. We choose to use NOKAV because Attack-Value was proven to be the optimal targeting strategy for 1 vs n scenarios (Furtak and Buro 2010) and it usually dominates Attack-Closest in n vs m scenarios as in (Churchill et al. 2012).

As we can see, some sort of search or iteration is part of all the scripts presented above; e.g., "move to closest unit" does not only require us to find it but also to find a viable spot somewhere near it and then find a path to this spot if it is reachable, if not,

new spot needs to be found etc. This search is, however, local and focused – we know exactly what we are looking for and usually some sort of heuristic function is used to try the best candidates first.

4.4.2 MCTS in script space

To perform MCTS[16] in script space, instead of assigning actions to units we assign scripts to them. For a set of n units $\{u_1, u_2, \dots, u_n\}$ a node in the search tree is a game state with these units and an edge between two nodes is a vector of scripts $(s_1, s_2, \dots, s_n)$ where script s_i returns a unit-action for unit u_i. We can easily map this vector of scripts to player-action by giving current game state to each script which returns a unit-action and we can modify this state for the next script. E.g., a unit-action for unit u_i is $a_i = s_i(g_{i-1}, i)$, where $g_i = a_i(g_{i-1})$ and g_0 is the initial game state in the node[17]. Player-action is then a vector of unit-actions $(a_1, a_2, \dots a_n)$.

A playout is performed by generating player-actions from random vectors of scripts. We first generate a vector of scripts $(s_1, s_2, \dots, s_n)$ where $\forall i \in 1 \dots n: s_i \in S$ and s_i is chosen uniformly randomly and S is a set of scripts the MCTS uses. Then from this vector, we generate a player-action as explained above and apply it to the game state. Then we do the same thing for the other player and iterate until we reach a terminal state.

Compared to the action space search, the branching factor is greatly limited by the number of scripts the units can choose from. For u units and s scripts we have a branching factor of s^u, e.g., for 3 scripts and 10 units we have $59{,}049$ which is much more reasonable number that the one we had in action space.

There are many enhancements and variations to the standard MCTS algorithm, some of which could be used to our modified script-space version. These include *History heuristic* which orders moves based on previous results or *Initial state-value estimates* which uses heuristic function to estimate initial value of states (Sturtevant 2008). Since playouts are expensive, we tried to improve the MCTS by cutting the playout after fixed number of rounds and evaluating the state using heuristic

[16] The exact algorithm we use is an adaptation of UCT. When not explicitly specified, we use UCT and MCTS interchangeably.

[17] For definition of script function see section 4.4.1 and for definition of action see section 4.2.

function which sums estimates army value for each player. This, however, did not improve our results; therefore, we perform full playouts.

4.4.3 Portfolio Greedy Search

PGS is algorithm introduced by Churchill and Buro (2013) where it was applied to StarCraft. We use it in its original version. Like MCTS in script space, PGS tries different script assignments for units and selects the best one encountered. In contrast to MCTS, PGS applies greedy hill-climbing search, i.e., it improves the currently best assignment and can get stuck in local optima or improve in circles. Following paragraph is from the original paper:

Portfolio Greedy Search takes as input an initial RTS combat state, a set of scripts to be searched called a portfolio*, and two integer values I and R. I is the number of improvement iterations we will perform, and R is the number of responses we will perform. As output it produces a player move, like the output of Alpha-Beta or UCT. The algorithm can be broken down into three main procedures:*

The main procedure is *PortfolioGreedySearch*. It first initializes all units of player 1 with predefined default script and then runs *GetSeedPlayer* for both players which gives us initial script assignments. Assignment of player 1 is then improved in *Improve* procedure. Then in a loop player 2 is improved and after it player 1. This loop is repeated *R* times. In the end, a script assignment for player 1 is returned. From this assignment we can again easily generate player action as described in 4.4.2.

The *GetSeedPlayer* procedure iterates over all scripts in the portfolio and in each iteration assigns current script to all units and performs a playout with given assignment against the other player's assignment. A single script assignment which performs best is chosen.

The *Improve* procedure is where the actual search happens. Given a script assignment it tries to improve it in hill-climbing fashion. It iterates through the units and for each unit tries in turn all scripts in the portfolio. With a new script assigned to a unit it performs a playout. If the playout ends up being better than the previous best, the new script assignment is kept. If it is not better, we keep the previous best and try new script or unit. This s done *I* times. *Improve* procedure may be terminated if a predefined time limit is reached. This time limit is reset each time we run the *Improve* procedure. Due to the greedy nature of the algorithm if we terminate the *Improve* early we still get the best assignment found in given time.

5 MCTS considering HP

In chapter 4, we explored and explained various regular and adapted search algorithms we decided to use to solve combat in CotG. We analyzed complexity of the domain and observed that plain search algorithms are not usable in our case. Therefore, the algorithms we decided to use are PGS and adapted MCTS in script space. In this section we present our enhancement of MCTS in script space called MCTS considering hit points (*MCTS_HP*). It is an interesting improvement which could be used in other games of similar type or generally to solve problems of similar nature.

The idea behind MCTS is that given enough random playouts, better actions will statistically lead to more wins than the worse ones. In UCT the search is guided based on this, and the more playouts we have, the more time we spend in promising actions. The playout and tree traversal speeds are therefore crucial, and a lot of optimization effort is put into these areas such as in (Černý 2016). In case of strategy games, high iteration speeds are, however, not always achievable due to the immense complexity of the game. For example, Justesen et al. (2014) were able to achieve between 5 to 100 iterations during 40ms with their UCT algorithm in small to medium size combat in simplified simulation of StarCraft. In CotG, we deal with the same problem. Even with simplified game simulation and reduced branching factor by using scripts we are not able to perform thousands of iterations in a 100ms time-frame. Because the iteration count is not high enough we try to guide the search by specifying the problem a little better and modifying MCTS accordingly.

The rest of the chapter is organized as follows. First, we clarify what is the actual goal in problem of combat. Then we present MCTS_HP and show how does it fit better to this problem. We conclude by discussing related work to this approach which we were able to find.

5.1 Problem clarification

In standard UCT algorithm when a random playout is finished either 1 or -1 is returned, representing which player won the playout[18]. This value is then backpropagated and added to each node's value for all nodes on the path to the root as

[18] 0 instead of -1 is also sometimes returned. 1/0 relates closer to "number of wins". 1/-1 is closer to symmetrical value of playout in the min-max sense.

explained in 4.3.2. This means that the UCT algorithm tries to maximize the number of wins – actions which lead to more wins are better. But it is oblivious to the way how was the victory achieved as well as to how exactly does the winning state look. That is fine when the AI plays the whole game and nothing other than win/lose is recorded such as in Go. In many cases, however, the UCT is used only for part of the game and win/lose is just an interpretation of the whole game state.

Combat in strategy games is one such example. It is important to win the combat scenario, but it usually does not mean we won the whole game. Better definition of the problem is that we want to maximize enemy losses while minimizing ours. To account for this, we propose Monte-Carlo tree search considering hit points algorithm.

5.2 MCTS_HP algorithm

MCTS_HP is based on MCTS in script space (see 4.4.2) which is an adaptation of UCT. The algorithms are identical except for the playout and backpropagation part. In MCTS after performing playout we return -1 if player p_1 has no units left and 1 if player p_2 has no units left, which gives us one bit of information. In MCTS_HP playout returns:

$$playoutValue = hp(p_1) - hp(p_2)$$

Where $hp(x)$ returns sum of HP of all units of player x. Note that either $hp(p_1)$ or $hp(p_2)$ must be zero, otherwise the game state would not be terminal.

During backpropagation, this value returned from playout is in each node n on the path from leaf to root mapped to interval $[-1,1]$ and added to the node's current value as usual. The mapping is performed as follows:

$$nodeValue_n(playoutValue) = \frac{playoutValue}{hp_n(p)}$$

Where $hp_n(p)$ returns sum of HP of all units of player p for the state in node n. p is p_1 for $playoutValue > 0$ and p_2 otherwise. This means that p_1 tries to maximize the amount of HP remaining for his units and minimize it for enemy units. We can be sure that the $nodeValue$ is always in the $[-1,1]$ interval because units cannot gain HP; therefore,

$$playoutVaule \leq hp_{node}(p) \quad \forall node \in path$$

Where $path$ contains all nodes on the path from the leaf where the playout was performed to the root. Example of this mapping for one node can be seen in Figure 5.1.

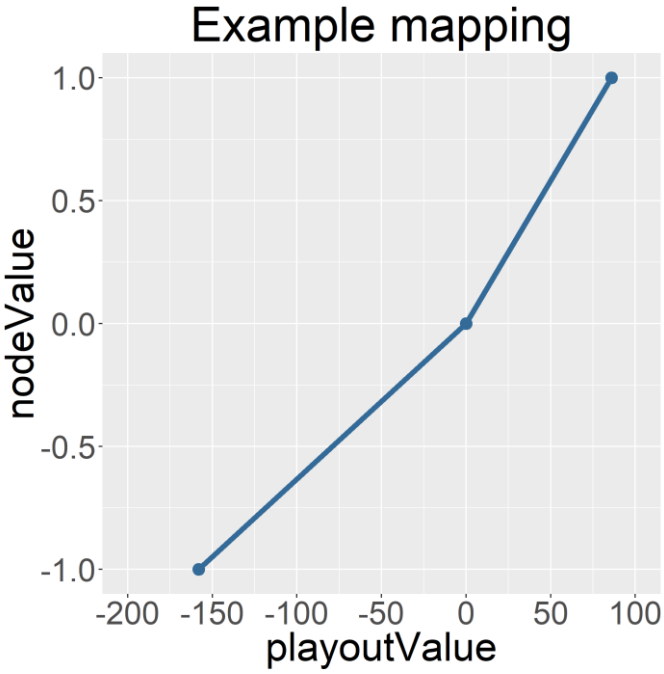

Figure 5.1 – Example mapping of playout value into interval $[-\mathbf{1}, \mathbf{1}]$ for a node $\boldsymbol{n}$ with $\boldsymbol{hp_n(p_1) = 158}$ and $\boldsymbol{hp_n(p_2) = 86}$. On the $\boldsymbol{x}$ axis the leftmost point has value of **158** and the rightmost point has value of **86**.

The fact that we perform the mapping again in each node means that in different nodes the value of a playout may mean different things. For example, in a scenario depicted in Figure 5.2, we perform a playout from node with $hp_{leaf}(p_1) = 10$ and the playout ends up having $playoutValue = 9$, the mapped value in the leaf will be $nodeValue = 0.9$ which looks like a very good result. If we did not remap and just propagated this value up the tree, all the nodes on the path would increase their value by 0.9 and their chances to be selected would increase. What if, however, having 10 HP was not very good in the first place? It could happen so that we started with 210 HP and ended up with only 10 and then managed to barely win with 9 HP. By remapping the HP value at each node, we can preserve this information and in the starting node with $hp_{start}(p_1) = 210$ this will be considered very weak win.

As we see in Figure 5.1, the mapping is linear for both subintervals $[-1,0)$ and $[0,1]$. In case of this mapping very weak wins and very weak loses, i.e., $nodeValue$ in small interval around 0 change the actual value of a node only slightly. However, the binary result (win/loss) is still completely opposite. This may be problematic for

games where we care more about the binary result which is usually not the case for combat in strategy games. Other mapping functions may be used for slightly different behaviors. To give more value to the win/loss result we could push the mapping interval away from 0 and map for example to interval $[-1, -0.8] \cup [0.8,1]$ or we could just use different function than linear.

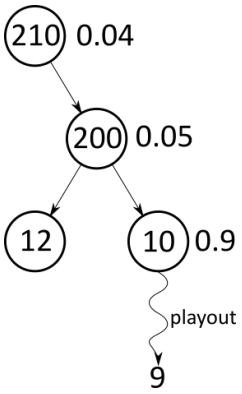

Figure 5.2 – Example of possible backpropagation and mapping in MCTS_HP. Numbers in the nodes are HP remaining, i.e., $\boldsymbol{hp_n(p_1)}$. The numbers next to nodes represent normalized values, i.e., $\boldsymbol{nodeValue}$ of given playout with $\boldsymbol{playoutValue = 9}$.

We should note that application of this approach is not limited just to script space searches. This approach would work the same in a regular MCTS in action space and other MCTS variations. And it is applicable to problems where the goal is not only to win, but we also care for quality of the victory and it is quantifiable (such as HP in our case). Values other than HP could also be used in our case such as unit value, e.g., how expensive or high-tech a unit is.

5.3 Related work

Since this enhanced MCTS method proved to be very useful and to our knowledge was not used in context of strategy games we researched other areas where MCTS is used. We found usage of a similar technique in (Tom and Müller 2009) where the authors primarily tested the Rapid Action Value Estimation heuristics on a game called Sum of Switches (*SOS*). Explanation of the game from the original paper follows.

SOS is a number picking game played by two players. The game has one parameter n. In $SOS(n)$ players alternate turns picking one of n possible moves. Each move can only be picked once. The moves have values $\{0, \ldots, n-1\}$, but the values are hidden from the players. The only feedback for the players is whether they win or lose the overall game. After n moves, the game is over. Let s_1 be the sum of all first

player's picks, $s_1 = p_{1,1} + \ldots + p_{1,\frac{n}{2}}$*, and* s_2 *the sum of second player's picks,* $s_2 = p_{2,1} + \ldots + p_{2,\frac{n}{2}}$*. Scoring is similar to the game of Go. The komi* k *is set to the perfect play outcome,* $k = (n-1) - (n-2) + \ldots = \lfloor n/2 \rfloor$*. The first player wins iff* $s_1 - s_2 \geq k$*.*

In the experiments, the authors use enhanced UCT algorithm where one of the enhancements is called *Score Bonus*. It is used to distinguish between strong and weak wins by instead of returning 0 or 1 from the playout, a value in interval $[0; \gamma]$ for losses and in interval $[1-\gamma; 1]$ for wins is returned. The values are scaled linearly in these intervals. Values 0.1, 0.05, and 0.02 were tried as the parameter γ, however, the Score Bonus was unable to improve the gameplay of UCT in SOS.

The authors do not elaborate why did the Score Bonus fail to improve performance in SOS, although they mention that value of $\gamma = 0.02$ slightly improved performance in 9x9 Go. Results of our approach are contradictory, but our problem and method are different as well. First, we use much wider range of values from the whole interval $[-1,1]$ and more aggressively exploit the entire range of different states. It is also worth noting that in our case the algorithm closely matches the problem. In SOS and Go, the MCTS solves the whole game whereas in our case the MCTS solves just a subproblem, the result of which matters to the rest of the game. Another factor is the number of iterations. It is common in games such as Go for the MCTS to have thousands to millions of playouts. In our case we have around hundreds to thousands of playouts with much higher branching factor, which means that the regular MCTS cannot sample the search tree so well and using MCTS_HP for better guidance helps.

6 Implementation

In this chapter we explore implementation details of algorithms and solutions presented earlier. This section combined with user documentation and XML documentation in the source code generated to HTML should also serve as programmer documentation[19]. First, we present current AI behavior tree system implementation and our solution to visualizing and editing these trees. Then we show the architecture and interesting details in our combat simulator implementation. In the end, we show our benchmarking framework and its architecture.

6.1 Original AI system

CotG is developed using Xenko game engine[20] which is a modern high-level game engine written in C#. The AI module is available on GitHub[21] under MIT license, however, to build and develop the AI project the game's binaries are still needed therefore one needs to own the game.

6.1.1 Behavior trees

The AI in the game is programmed in behavior trees, which are encoded as Extensible Markup Language (*XML*) files with *.beh* extension. This format is convenient for its implicit tree structure. AI behavior files located in the game's directory `Content/Gameplay/AI/` are loaded by the game at startup and deserialized. For each XML tag in the tree a corresponding class either in the game binaries or in the AI module needs to exist. This class must derive from the `BehaviorComponentBase` class. Class architecture provided by the game is shown in Figure 6.1.

The tree can be built from standard blocks such as `Inverter`, `Selector`, or `Sequence`. Trees can also contain references to other trees, so they can be easily reused and be more readable. This is done by `BehaviorReference` node. However, these are only flow control nodes. The actual logic is implemented in `Behavior` nodes. For more information about behavior trees and their implementation please see (Champandard and Dunstan 2012).

[19] Both user documentation and HTML documentation available in the attachment.

[20] https://xenko.com/

[21] https://github.com/EmptyKeys/Children-of-the-Galaxy

To add a custom behavior, we must implement a public class which derives from the `BehaviorComponentBase` class to the `EmptyKeys.Strategy.AI` project. Let this class be called *CustomBehavior*. Then we can add an XML tag `<CustomBehavior />` to the behavior tree XML file. When a node in the tree is executed, the `Behave` method is invoked on it. This method returns success or failure state[22] which is then passed to the control elements such as selector or sequence. This means that in our *CustomBehavior* class we would override the `Behave` method and implement our behavior there.

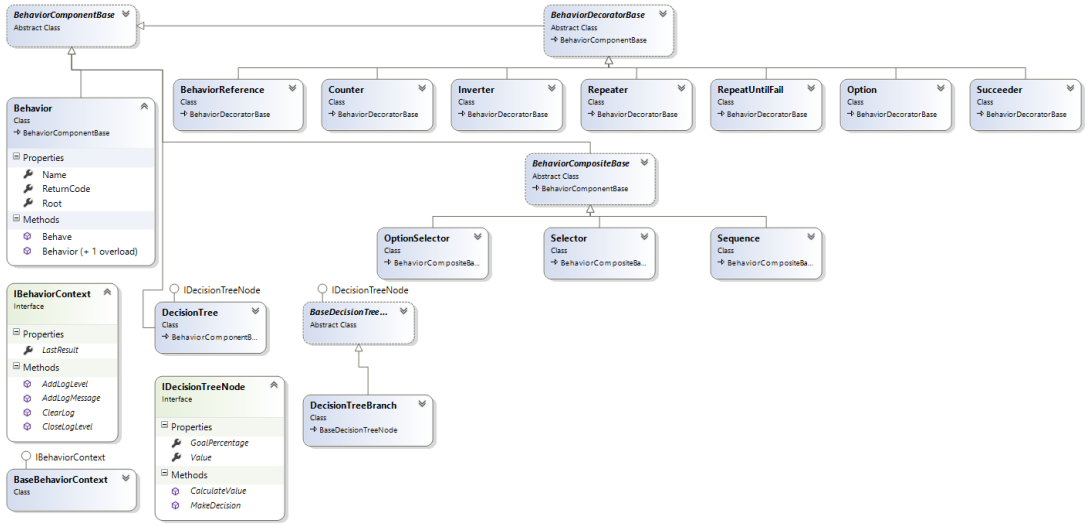

Figure 6.1 – Class hierarchy provided by the game to develop an AI.

Taken from https://github.com/EmptyKeys/Children-of-the-Galaxy.

Behaviors without a shared state would make for very weak and cumbersome system. To account for this, when a behavior is executed, the `Behave` method receives an `IBehaviorContext` object as a parameter. This object serves as a state to otherwise state-less behavior nodes. This object is shared between behaviors and they can share data this way. E.g., one behavior finds an attack target, it saves this target to the context and the next behavior responsible for attack execution can load this target and perform attack at it. Since the behaviors are executed sequentially, race conditions and synchronization do not have to be considered.

Following XML snippet shows a part of `ExploreEmptySpace` behavior.

[22] As mentioned in section 2.2, *running* state is not supported.

```
<?xml version="1.0"?>
<Behavior Name="ExploreEmptySpace">
  <Sequence>
    <Behaviors>
      <Inverter>
        <UnitIsInGalaxyEnvironment />
      </Inverter>
      <FindUnscannedAnomaly />
      <BehaviorReference ReferenceName="ScanAnomaly" />
    </Behaviors>
  </Sequence>
  <!-- more exploration, omitted -->
</Behavior>
```

This snippet runs a sequence of behaviors. First, it checks if a unit is in a galaxy view or solar system view. If it is not in the galaxy, a not yet scanned anomaly is found. This information is saved to the behavior context[23]. Then a *ScanAnomaly* behavior tree is referenced which takes care of scanning of the anomaly found and saved in the context. This tree is implemented in corresponding XML file `ScanAnomaly.beh`. Nodes `UnitsIsInGalaxyEnvironment` and `FindUnscannedAnomaly` are implemented as custom classes in the AI module – written by the AI developer.

6.1.2 Behavior tree editor

Tools are very important part of any development process. XML format provides nice structure for smaller behavior trees, but for larger ones where the leaves may be several levels deep, purely text-based approach with references to other files is error prone and difficult to navigate in. Therefore, it is a frequent practice to have a visual behavior tree editor[24]. However, developing such a tool is a lengthy process and thus CotG does not have one. That is why we decided to implement it at least partially to better

[23] This is not visible from the tree, but we know it by inspecting the code for FindUnscannedAnomaly class.

[24] One such example can be found in widely used game engine Unreal Engine 4 https://docs.unrealengine.com/en-us/Engine/AI/BehaviorTrees [Accessed 9 May 2018].

understand the inner workings of current AI and potentially help future developers to read and edit behavior trees for the game.

Our workflow for editing behavior trees is divided to visualization and writing. Visualization takes a behavior tree in XML format and transforms it to a GML format[25] which can be then visualized by existing graph editors such as yEd[26]. One such visualization can be seen in Figure 6.2.

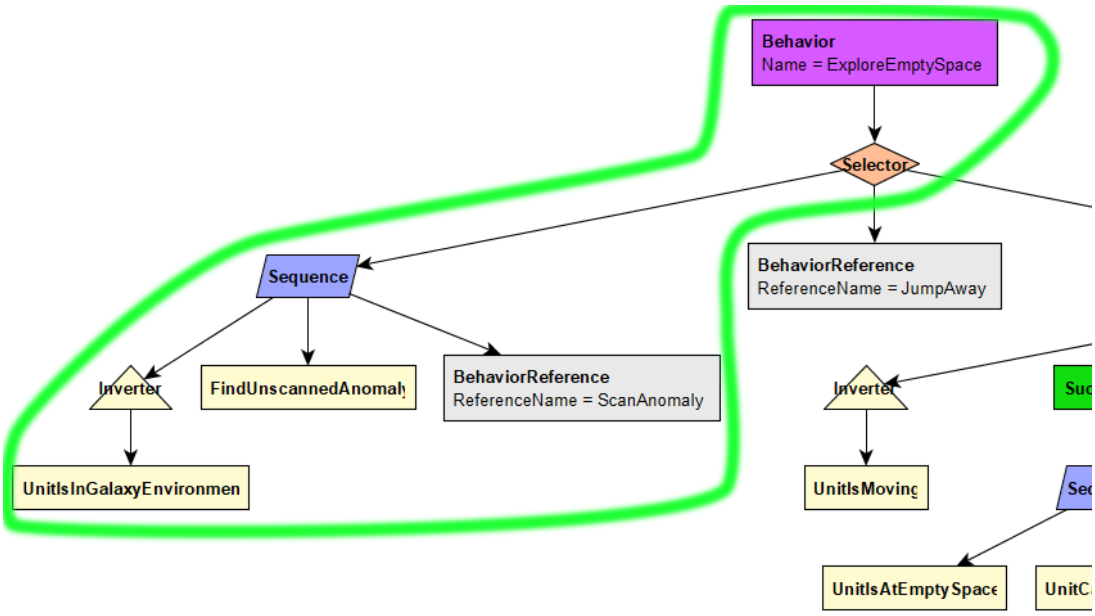

Figure 6.2 – Explore empty space transformed to GML and visualized by yEd. In green we can see ExploreEmptySpace code snippet from 6.1.1. Full image in high resolution can be found in the attachment.

Our visualizer can also expand references which means we can clearly see a whole behavior without opening multiple files. Using editor such as yEd allows us to utilize its strengths such as automatic tree construction and ability to select and move nodes around. Which leads us to writing part of our workflow. In yEd we have created a custom palette of nodes which can be used to construct behavior trees by simply dragging, dropping, and editing a text in the node. A node created this way can be then simply connected to the tree and ordered in relation to its siblings. Resulting graph may be then again saved to GML format and transformed to the XML format the game understands.

[25] http://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf [Accessed 9 April 2018]

[26] https://www.yworks.com/products/yed [Accessed 9 April 2018]

Visualization and writing is taken care of by a set of Python scripts we created. All scripts, user documentation, and yEd palette can be found in a Bitbucket Git repository[27]. When integrated to a context menu in Windows this workflow is almost as good as having a full behavior tree editor. Unfortunately, yEd is freeware but not opensource; therefore, we could not integrate our scripts into the editor.

6.2 Children of the Galaxy Micro Simulator (CMS)

To use search algorithms, we need a forward model of the game. This is usually done by implementing simplified simulation of the game with the necessary game mechanics. CMS is a library containing this simplified simulation. It is added as a project to the original AI module's Visual Studio 2015 solution. This solution can be found in a Bitbucket Git repository[28]. It has no dependencies on the game which makes it ideal testbed for AI agents. If we want to use it from the game, we simply add it as a dependency and then we can simulate the game from our current game state using one of the AI agents implemented in CMS.

CMS is programmed in C# to allow easy integration and usage from the original AI module. At first, we considered C++ as a better fit for this type of work because we wanted the simulation to be as fast as possible. C++ has non-trivial interoperability with C#, which would not be a problem for us, but it may prevent not so experienced community developers from using our simulator. And since the game is multi-platform, compilation would be more difficult as well. Therefore, to attract the widest array of potential programmers we decided to use slower but easier C#.

The CMS library is state-less and provides a clean API. It offers variety of AI players in a namespace `CMS.Players`. To use it we simply create one of these players, transform the full game state to a simplified game state represented by a class `CMS.GameEnvironment`. Then we call a `MakeActions` method on the player we created and pass the game environment to it. This method performs simulation based on the type and configuration of the player we created and returns a collection of actions (class `ActionCms`) the player considers to be the best in given state. This collection corresponds to a player-action. The transformation of the full game state to

[27] https://bitbucket.org/smejkapa/behvisualizer/src/master/ [Accessed 9 April 2018]

[28] https://bitbucket.org/smejkapa/cog-ai [Accessed 9 April 2018]

`CMS.GameEnvironment` is performed on the AI module side (not in CMS). This allows us to have one-way dependency. For this purpose, we implemented static class `EnvironmentFactory` in the AI module.

The CMS library depends on *Optimized priority queue* library[29] for the A* and *QuickGraph* library[30] for debugging visualization purposes, both libraries are included in the project as NuGet dependencies.

6.2.1 Game state and environment

All state CMS works with is encapsulated in the `GameEnvironment` class. UML diagram of hierarchy of state classes in CMS is in Figure 6.3.

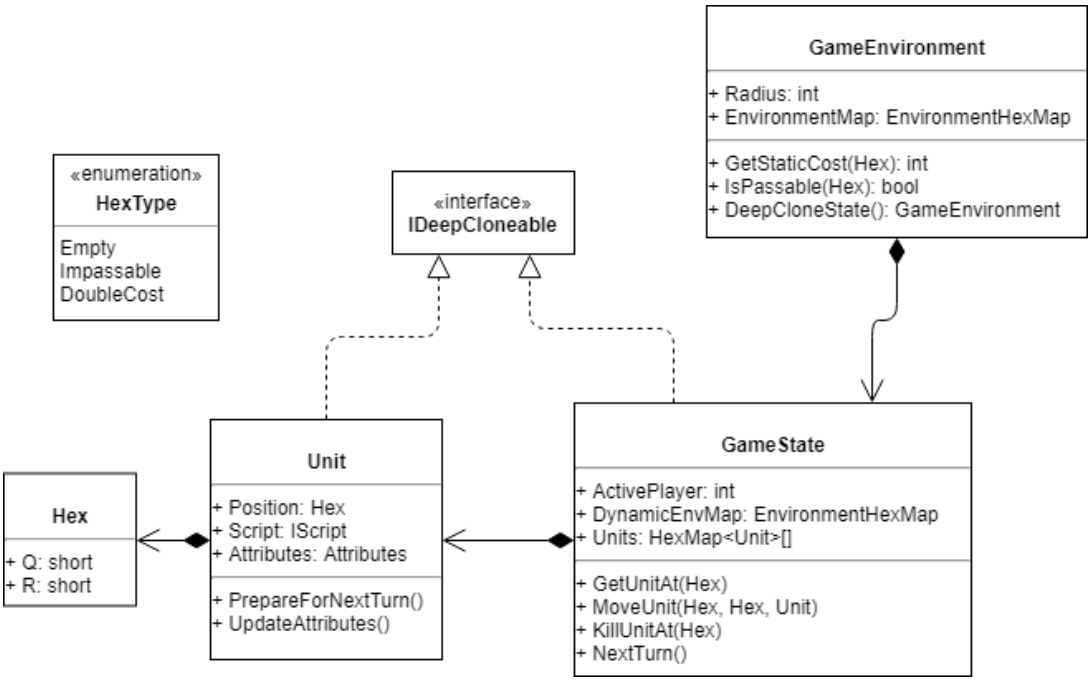

Figure 6.3 – UML diagram of state classes in CMS.

`GameEnvironment` class itself contains static information about the environment such as radius of current solar system and positions of asteroids and sun. Dynamic information about current game state and environment are encapsulated in the `GameState` class. It contains information about units and active player. Information about individual hexes are saved in `EnvironmentHexMap` class which is a C# Dictionary where the key is a `Hex` struct and the value is a `HexType` enumeration. This

[29] https://github.com/BlueRaja/High-Speed-Priority-Queue-for-C-Sharp [Accessed 9 April 2018]

[30] https://github.com/YaccConstructor/QuickGraph [Accessed 9 April 2018]

class is used to store information about whether a hex is passable or not and if it is, how much does a move action through it cost. `GameState` combines static information from `GameEnvironment` and dynamic information from units' positions to a single `EnvironmentHexMap`. This way when we perform pathfinding only a single hash-table lookup per hex is required.

The `Hex` struct contains only Q and R 16-bit integer axial coordinates[31] of the hex and is optimized for fast (and unique) hash code generation. By shifting the first coordinate by 16 bits to the left and using logical *OR* to the result and the second coordinate we store both in 32-bit integer and each 32-bit number corresponds to exactly one (Q, R) pair. To achieve best data locality and lower allocation needs, `Hex` is a C# struct.

Since executing actions on a game state is potentially expensive operation involving pathfinding and several calculations for each unit, instead of executing and reverting actions during search – as common in Chess or Go we decided to copy the whole game state for each search node instead, which is common in computer games. Therefore, the `GameState` class implements `IDeepCloneable` interface. Since the static information in `GameEnvironment` do not change during the search it is not cloneable itself, but it provides `DeepCloneState` method which returns new instance of `GameEnvironment` which shares static data and contains a new deep cloned instance of `GameState`.

The game progresses to a next turn by calling `NextTurn` method on a `GameState`. This method modifies the `GameState` and calls appropriate methods on all units. Since we do not have access to source codes of the game and it is very important for this process to be the same as in the game, we utilized one big advantage of C# binaries and decompiled the bytecode to more readable C# code using JetBrains C# decompiler[32]. Based on this decompiled code we were able to implement our simulation according to the mechanics in the game.

[31] https://www.redblobgames.com/grids/hexagons/#coordinates-axial [Accessed 26 April 2018]

[32] https://www.jetbrains.com/decompiler/ [Accessed 9 April 2018]

6.2.2 Player and AI architecture

When using CMS, we can either create a `Player`, give it a `GameEnvironment` and ask for next actions. Or we can create a `Game`, give it two `Player`s and `GameEnvironment` and then we can simulate combat with these settings, for example to estimate our chances in upcoming battle. Implementations of the abstract class `Player` usually do not execute the AI logic themselves but serve as an API. These classes then contain different AI implementations from the `CMS.Micro` namespace. This allows us to combine more AI algorithms to one `Player` and hide the inner workings of the `Player` from the user. E.g., if we knew that some algorithm's performance is dependent on the size of the battle we could create a player which would use different algorithms depending on how many units are there in the combat scenario. Another advantage is that we can have unified API for the user in the `Player` class but different APIs for different AI algorithms. UML diagram of hierarchy of players and AI is in Figure 6.4.

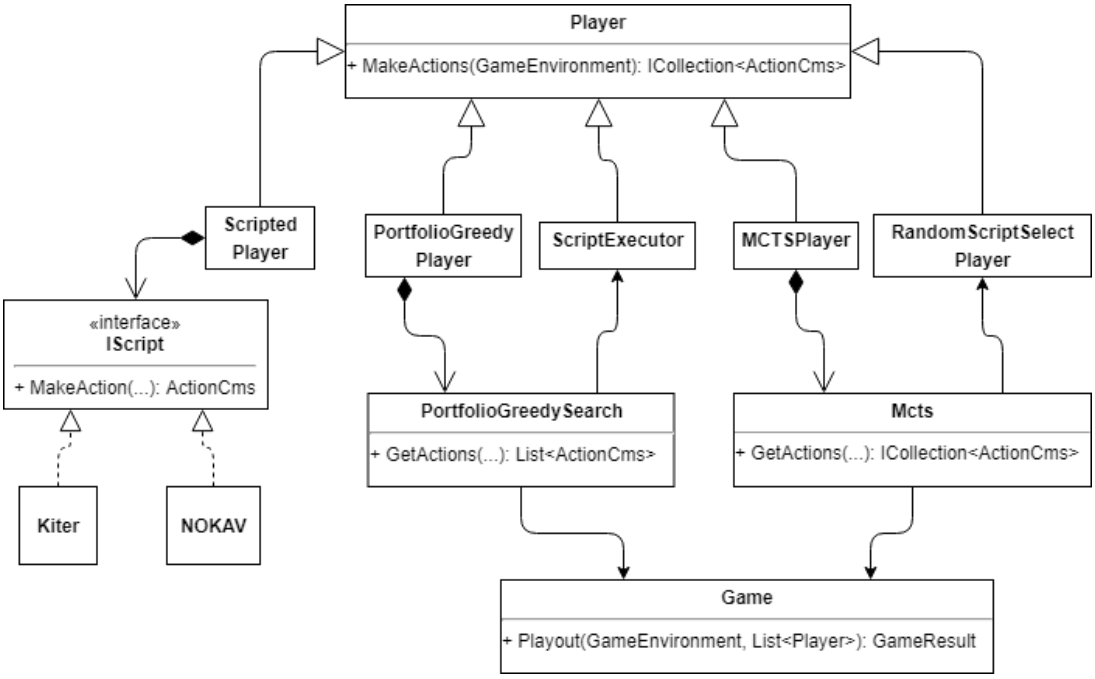

Figure 6.4 – UML diagram of hierarchy of AI classes and Player and Game API classes.

6.2.3 Scripted player

The simplest player we have is a `ScriptedPlayer`. It contains a single implementation of `IScript` interface. When asked for a player-action it simply executes its script for all units for given player. This operation corresponds to how we explained generation of player-action from script assignment in section 4.4.2. In this

case all units have the same script. The `ScriptedPlayer` class is generic where the generic parameter must be an `IScript` and determines which script will this `ScriptedPlayer` execute. Scripts we currently implement are *Kiter* and *NOKAV* introduced in section 4.4.1. We chose these two scripts because they performed very well in previous research and they are orthogonal to each other. The *Kiter* script implements the hit and run tactics and is more defensive and cunning where the *NOKAV* script is more like front line heavy fighting and absorbing damage.

6.2.4 Portfolio Greedy player

`PortfolioGreedyPlayer` uses our implementation of PGS algorithm introduced in chapter 4.4.3. We implemented this algorithm based on information provided in the original paper and based on its implementation in SparCraft. Portfolio of scripts is a `List` of `IScript`s and these scripts are then assigned directly to units, i.e., class `Unit` has s field of type `IScript`. When a playout is necessary the PGS clones the environment, creates an instance of the class `Game` and two `ScriptExecutor` players. Then a `Playout` method on the `Game` is called, a simulation is executed, and result returned. Based on this result an improvement is potentially performed.

The `ScriptExecutor` is a very simple class which, when asked for a player-action, just iterates through all the units and executes a script given unit has assigned. We chose this approach because it simplifies the playout considerably. If we did not want to add an `IScript` field to a unit, we would have to work with some sort of a map which would assign scripts to units. Simplest version of this would be to have a list of scripts for our list of units. Unit at position i in the unit list would have according script at position i in the script list. But then when executing the playout, we would have to pass these maps to the `Player` and the `Player` would have to somehow update them when units die. In our case we just iterate through the list of units both while executing playout and when improving the script assignments. The performance impact on other search approaches is negligible since the class `Unit` already has quite a few fields and an 8-byte pointer does not make much difference in our case.

Another improvement compared to the original algorithm we made was playout caching. This means that we have a hash table where the key is a script assignment for all units in the environment and the value is a result of a playout with this script assignment. Since the scripts are deterministic and the game state does not

change between playouts, instead of doing them again we can simply fetch the result of the playout from the cache. We should note that this improvement made a dramatic difference in performance of this algorithm and made it competitive to our other approaches.

6.2.5 MCTS player

`MctsPlayer` uses our implementation of MCTS or MCTS_HP algorithms introduced in chapters 4.4.2 and 5 respectively. Both AI's are implemented in the `Mcts` class. The search tree is composed of `MctsNode` nodes and has a tree structure in memory. To avoid memory allocations and improve data locality, a fixed preallocated array is often used and the search tree in MCTS is saved there. We also considered this approach as well as object pooling approach but when we profiled our implementation most time is spent while performing playouts and these optimizations would yield only minor performance improvements (if any).

Each `MctsNode` has a pointer to its parent as well as pointers to all its children. Generating children is not a cheap operation because actions of the child need to be applied to the parent's game state. The number of child nodes (leaves) is exponential and most of them are not visited at all. Therefore, we use lazy action generation. Each `MctsNode` has an `IEnumerable` of all its child nodes which are constructed on demand in the enumerable. To allow this we have an `IActionGenerator` interface which is used to create this enumerable. For MCTS in script space an implementation `ScriptActionGenerator` is used. An enumerator from the enumerable holds all the state necessary – it knows which child node to generate next. This is done by the standard `yield return` statement from C#. When traversing the tree, we call the `TryGetNextChild` method on each node. This method checks if the node is fully expanded. If not, a new child node is generated. If it is fully expanded, we use UCB to select the best child and search the tree recursively.

Like in PGS, the playout is performed by the method `Game.Playout`, but in this case `RandomScriptSelectPlayer` is used to perform it. This player simply choses a random script for each unit and then generates an action from this script. To improve performance, we tried to cutoff the playout early and evaluate given game state by a heuristic function. This indeed improved the speed, but overall results were worse; therefore, we use full playouts.

6.2.6 Pathfinding

One of the most utilized systems in our simulation is pathfinding implemented in a class `Pathfinder`. We use standard A* algorithm optimized for our needs. We use A* because we want to always use the optimal path as appropriate for strategy game.

By profiling the application, we found out that even after all our optimizations pathfinding takes more than one third of all time. This is quite common in applications which use more sophisticated pathfinding algorithms such as A* as for example in (Andruszkiewicz 2015). To improve performance either precomputation or caching is usually used. Since our environment is dynamic and units can collide and block each other we cannot precompute all possible shortest paths in the environment or use more sophisticated search techniques such as Jump Point Search+ (Rabin 2015). Therefore, we implemented caching of previous pathfinding results using two-level cache. When a pathfinding query is issued, we create a hash from all the occupied hexes in current environment and try to find it in the first level cache. If such an entry exists we get the second level cache where we try to find whether a query with the same start and end points was issued before. If it was, we return it; if not, we compute it and save it to the cache. However, even when we have the same shared cache throughout multiple searches we were able to achieve an average cache hit rate of only about 10% to 30%. And the overhead of hashing the game environment, equality checks, and looking up potential existing entries was usually higher that the benefit of the cache. This is mainly because the environment is relatively big, and units can move to many different hexes which results in many different game states. This cache system is, however, implemented in the simulator and may be turned on by defining a compilation symbol `USE_PFATHFINDING_CACHE`.

To improve performance of A* we also tried to use pooling to the nodes A* uses. This again did not prove to yield any significant performance boost. Pooling can be turned on by defining compilation symbol `USE_A_STAR_NODE_POOLING`.

6.3 CMS benchmark

To quickly evaluate different AI players, we designed a simple command line interface framework for the CMS which is in the project `CMS.Benchmark`. This project is configurable by XML files which allow us to define players, battles, environments, and benchmarks without the need to recompile the code. These configs can be then

distributed with the executable and run from command line. Results of these benchmarks are output to the command line, csv, and PostgreSQL[33] database server from which we can transform them by a set of SQL queries and visualize results of these queries by the R[34] software. In this chapter we briefly introduce architecture of this benchmarking project. How to configure and use this project is described in the user documentation. The project depends on *Npsql* library[35] to interact with the PostgreSQL server, the library is included as a NuGet dependency. User documentation containing explanations of the configuration can be found in the attachment.

`CMS.Benchmark` is programmed in C# and compiles to an executable application. It depends on the CMS library since it uses all the logic from it. The main class is `BenchmarkRunner` which has a single method `Run` accepting a string representing an id of the set of benchmarks to run. First, it locates a file with the benchmark set, parses it, and creates benchmarks (class `Benchmark`) according to it. These benchmarks are then executed in serial order.

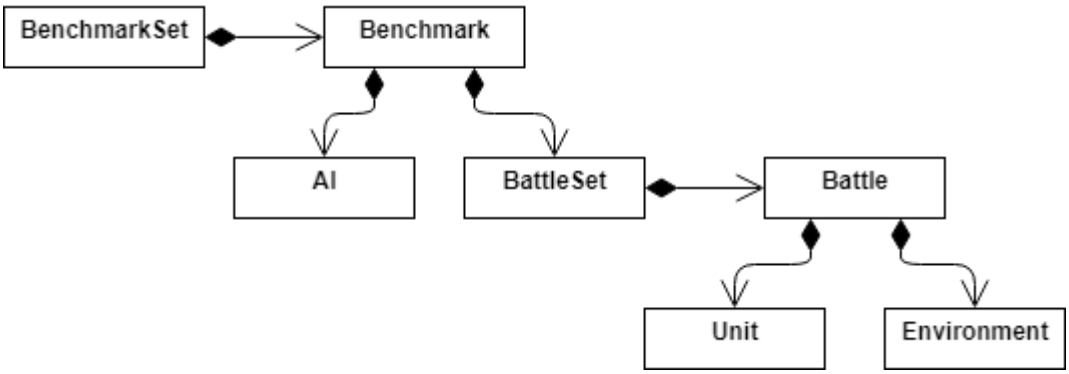

Figure 6.5 – Shows references between different XML configuration file types in the configuration hierarchy.

To parse the XML config files we provide a set of factory methods in the namespace `CMS.Benchmark.Config`. Since our configuration setup is hierarchical as shown in Figure 6.5 the static factory methods correspond to this hierarchy. E.g., there is a method creating a Battle which accepts the battle's id and it calls methods creating

[33] https://www.postgresql.org/ [Accessed 9 May 2018]

[34] https://www.r-project.org/ [Accessed 9 May 2018]

[35] http://www.npgsql.org/ [Accessed 9 May 2018]

Unit and Environment with their respective ids. The main class and method of these factories is `BenchmarkFactory.MakeBenchmarkSet` which accepts a string id of a `BenchmarkSet` and returns the complete assembled set of benchmarks. The XML files are all validated against their according XML schema definitions (*XSDs*). If an error in an XML file is encountered an exception is thrown and caught and a helpful error message is displayed in the console describing where and in which file, the error was encountered.

This configuration may seem complicated at first, but its hierarchical structure allows us to reuse many files such as AI definitions which means that if we have a set of benchmarks created and all we want to do is to change one AI we can simply change this one file and all the benchmarks referencing it will be changed. This structure also allows us to easily split benchmarks to smaller groups. E.g., we can simply split a battle set into two parts and run the first part now and the second one later. No other files than the battle set need to be modified.

The project could be easily multithreaded to utilize full potential of the computer it runs on by running multiple benchmarks at once. We opted not to do this and rather use process level parallelism where we simply run multiple processes with different configurations. This way it is easier to control the utilization of the processor and if we split the configurations (or have more of them) we can even distribute the processes across multiple computers which are all send data to one central SQL server. However, multithreading support may be introduced in future versions.

We can easily create different configurations by hand by editing the files. However, to have a large set of different randomized benchmarks we use a Python script `BenchmarkMaker/main.py` which can be also found in the main repository.

7 Experiments

In this chapter we compare the AI approaches presented in chapter 4 in our test environment introduced in chapter 6.3. First, we introduce our methodology. Then we show results and time performances of the AI players in several tournaments. We conclude by discussing these results in detail.

To evaluate different AI payers, we performed several tournaments where everyone played against everyone else. Each tournament was played with different number of units to show how various AIs scale with that parameter. Unit counts are 3, 5, 7, 9 representing small scale combats, 16, 32 representing medium scale combats and 48, 64 representing large scale combats. The battles were asymmetrical in sense of positioning. For each army, a center point was chosen randomly in the environment and around this point units were randomly distributed. This was done automatically by a script. Two basic types of units were involved in these battles. *Destroyer* which is a unit with high damage output but relatively short range and low HP. And *Battleship* which is a unit with medium weapon range, lower damage, but high HP and shields. These units are in their default configuration without any upgrades. Our results present average of different ratios of these units to show common case scenarios, e.g., an army in 5vs5 scenario may contain 3 battleships and 2 destroyers, or 1 battleship and 4 destroyers etc.

Because the game is turn-based, one player needs to go first which may be a great advantage. Therefore, we evaluate each battle scenario from both sides. First, one player controls army 1 and the other player controls army 2. Afterwards, we switch sides, i.e., army 1 always goes first but once player 1 controls it and once player 2 does. We call this battle where players switch sides *symmetric battle* and if a player manages to win both battles in a symmetric battle we say that he achieved a *sym-win* where winning each sub battle is simply called *win*. The fact that one army goes first may be too advantageous and even a relatively bad player may win from given position meaning that many symmetric battles end in no sym-wins, but each player has one win. Therefore, we decided to bring the scenario closer to reality of the game and we also sum HP of all the ships of the winning player. If a symmetric battle ends 1:1 we look at how many HP did each player's ships end with and the one with more HP left

achieves a sym-win[36]. To clarify our methodology and sym-wins, consider an example in Table 7.1.

Symmetric battle id	Battle 1: Player A starts	Battle 2: Player B starts	Sym result
1	A wins, HP: 10	B wins, HP: 20	B has more HP
2	A wins, HP: 12	B wins, HP: 15	B has more HP
3	A wins, HP: 5	A wins, HP: 2	A wins both
4	A wins, HP: 4	A wins, HP: 1	A wins both
5	A wins. HP: 8	B wins, HP: 11	B has more HP
Total	A: 7 wins, HP: 42 B: 3 wins, HP: 46		A: 2 sym-wins B: 3 sym-wins

Table 7.1 – Example benchmark with 5 symmetric battles = 10 battles in total. Player A wins 7 battles, Player B wins 3 battles. However, since the Player B wins these battles with more HP, he achieves 3 sym-wins whereas the player A has only 2. In this example we can see the players are similarly good, but the win count suggests that the Player A is much better. This shows how counting only wins can be misleading as well as counting only sym-wins without the total HP remaining.

Another statistic we measure and show here is HP remaining; e.g., after a battle we sum HP of all ships of the winning player (losing player does not have any ships) and save this value to the database. In the plots, this is summed over all the battles played in a given tournament. As we explain in chapter 5.1, how many units remain after a battle and in what condition they are is very important for players in strategy games. This number gives us complementary information to the sym-win number. E.g., sym-wins tell us how many sym-battles did a player win, and HP remaining tells us how good these victories were. It is possible to have high HP remaining by dominating the opponent in a few battles but then the sym-win count will be low. It is also possible to barely sym-win many sym-battles but then the HP remaining will be lower.

Algorithms we chose for our tournament were the two scripts Kiter and NOKAV, where the NOKAV script behaves very similarly to the current AI present in the game. PGS with different response and iteration counts which is encoded as PGS_I_R where I is improvement count and R is response count. Specific

[36] If the symmetric battle ends 0:2 or 2:0 the previous rules for sym-win apply.

configurations are PGS_1_0, PGS_3_3, PGS_5_5. PGS_1_0 was chosen because it was used in the original paper and the other two were chosen to see how higher iteration and response counts influence the performance. All PGS algorithms have time limit per Improvement set to 500 milliseconds (see 4.4.3). MCTS and MCTS_HP with three different execution time limits were chosen. The encoding is straight forward MCTS_TIME or MCTS_HP_TIME where TIME stands for time in milliseconds. 100ms represents very fast execution which could be used many times during one turn, 500ms is medium time and 2s is present to show how could the MCTS perform if further optimized or running on faster computers. All experiments were performed on Intel Core i7 2600K @ 4.5Ghz with 16GB of DDR3 RAM, SSD, and running Windows 10 Professional.

If not stated otherwise, error bounds in the graphs represent 95% confidence interval. Because the MCTS algorithms are not deterministic we run each battle 5 times and consider them as separate battles. All the algorithms which need a portfolio of scrips use the two scripts we implement Kiter and NOKAV. Based on preliminary results we use a very simple move ordering for MCTS methods where we try the assignments involving the Kiter script first. All the benchmark configurations and resulting graphs can be found in the attachment.

7.1 Results

In this chapter we present win and sym-win rate results of series of round robin tournaments. All charts presented in this chapter and several more (for example for 3vs3) are also included in the attachment.

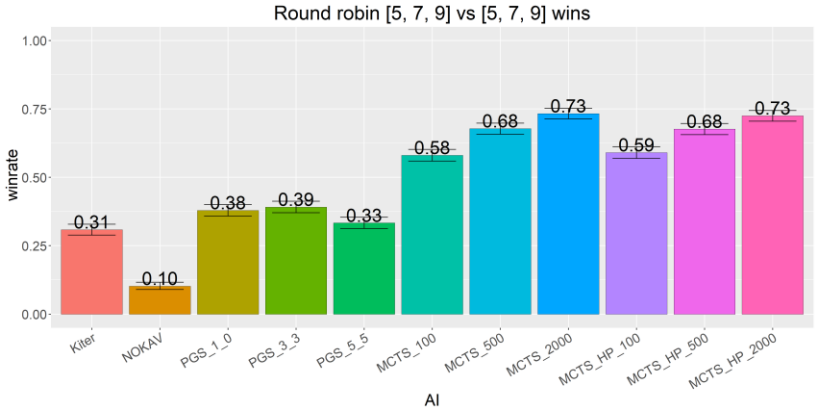

Figure 7.1 – Win rates for 5v5, 7vs7, and 9vs9 small scale combat round robin tournaments.

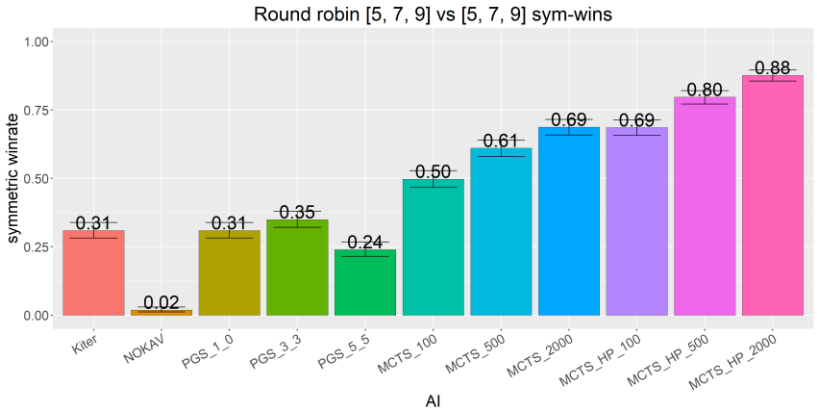

Figure 7.2 – Sym-win rates for 5v5, 7vs7, and 9vs9 small scale combat round robin tournaments.

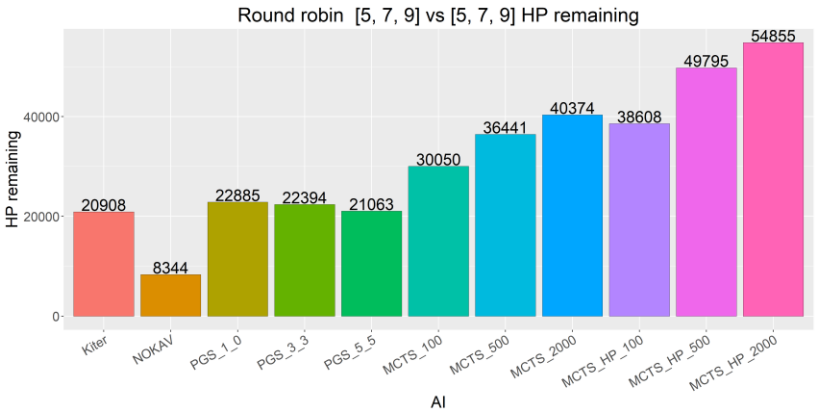

Figure 7.3 – Summed HP remaining for the winning side from all battles.

We start by showing results for small combat of 5, 7, and 9 units at each side in Figure 7.1, Figure 7.2, and Figure 7.3. We ran three separate tournaments but since the results were very similar we added them together and they represent “small combat”. Smaller scenarios such as 3vs3 were also tried but there were not that many tactical opportunities to exploit and all the AI players behaved similarly.

In this case, we can see the win rate scales well with time for MCTS methods and is overall higher than PGS methods because even the MCTS with 100ms time limit has enough time to compute good moves. In sym-wins we see MCTS_HP dominates other approaches; even the MCTS with 2 seconds time limit only matches the MCTS_HP with 100ms limit. The HP remaining plot shows correlation with sym-wins, which means that the sym-wins are usually strong, e.g., if an AI sym-wins it does not leave the battle with just a few HP.

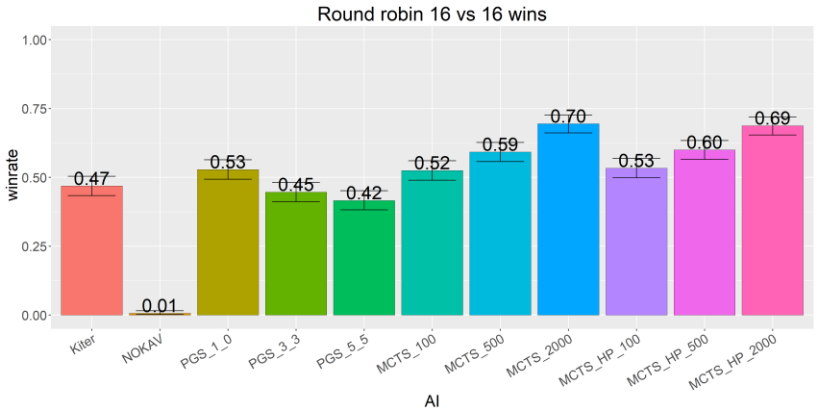

Figure 7.4 – Win rates for 16vs16 medium scale combat round robin tournament.

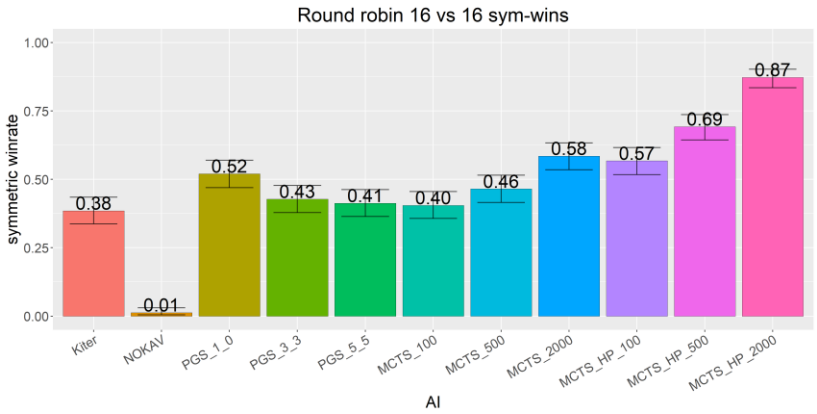

Figure 7.5 – Sym-win rates for 16vs16 medium scale combat round robin tournament.

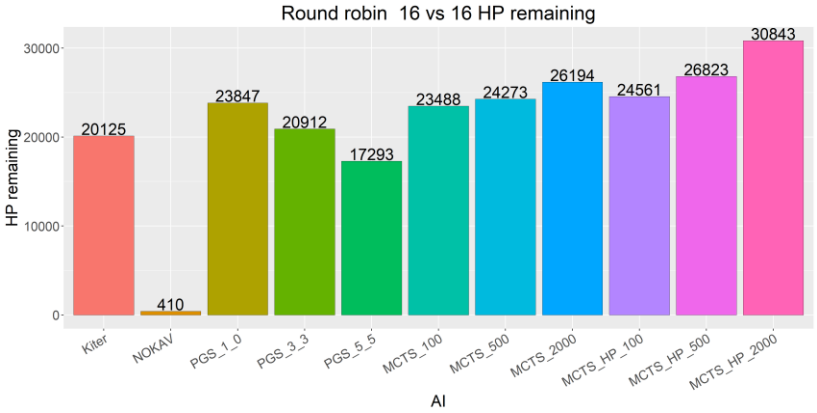

Figure 7.6 – Summed HP remaining for the winning side from all battles for 16vs16.

Results for the 16vs16 combat tournament are shown in Figure 7.4 and Figure 7.5, and Figure 7.6. When we move to medium scale combat the win rates are much closer and the PGS methods match the MCTS methods with 100ms time limit, but overall, MCTS still achieves more wins. In sym-wins the PGS matches the regular MCTS and even the MCTS_2000 is not very far from the best PGS result. MCTS_HP, however, still

easily wins. The HP results show that the wins are not as strong as for the small-scale combat, however, we also need to bear in mind that these numbers are absolute and for higher unit counts will be larger, and the relative differences will be smaller.

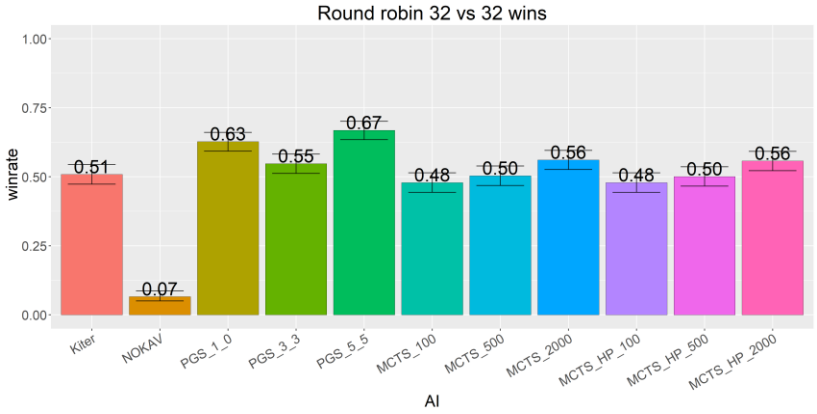

Figure 7.7 – Win rates for 32vs32 medium scale combat round robin tournament.

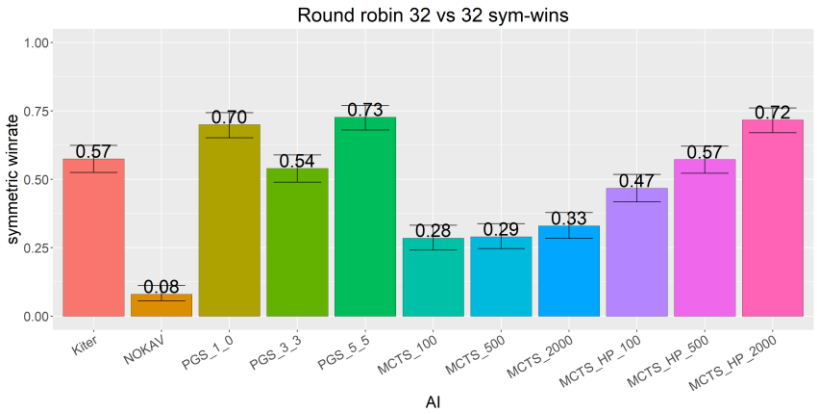

Figure 7.8 – Sym-win rates for 32vs32 medium scale combat round robin tournament.

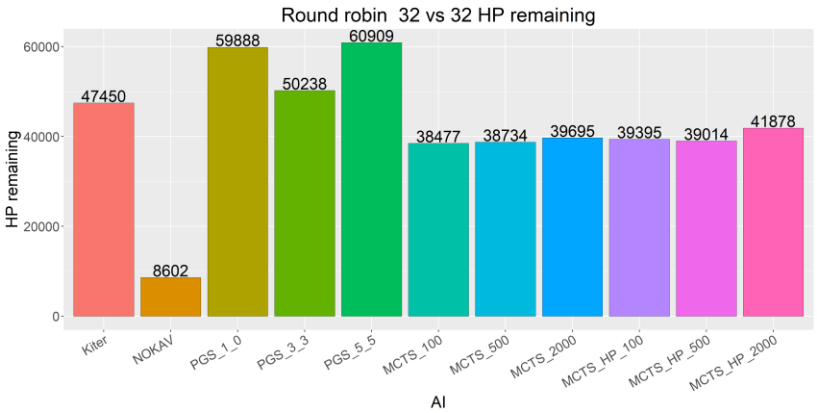

Figure 7.9 – Summed HP remaining for the winning side from all battles for 32vs32.

Results for the 32vs32 combat tournament are shown in Figure 7.7, Figure 7.8, and Figure 7.9. When we double the unit count we can see that MCTS methods start to

lose and faster and simpler greedy PGS methods gain. Win rates are close but PGS is clearly better. In sym-win rates the picture is even clearer. MCTS does not have enough time for the search in such a big combat scenario and loses. The only competition remains in MCTS_HP_2000 which still manages to achieve top level sym-win rate. However, the HP results show that the PGS was able to win many high-quality battles since it has many more HP remaining than even the MCTS_HP_2000.

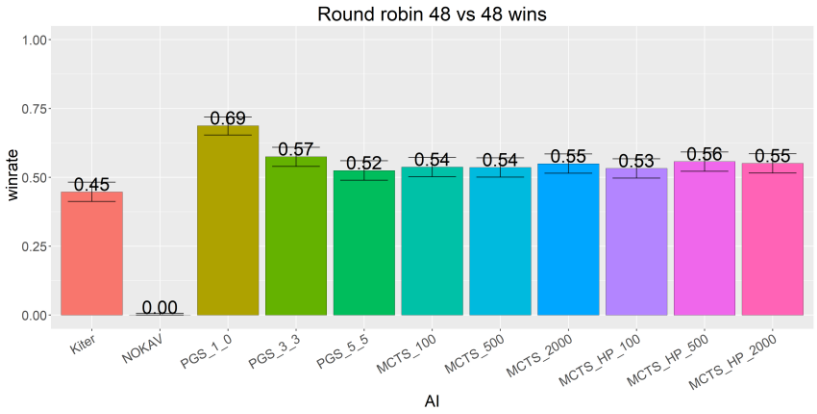

Figure 7.10 – Win rates for 48vs48 large scale combat round robin tournament.

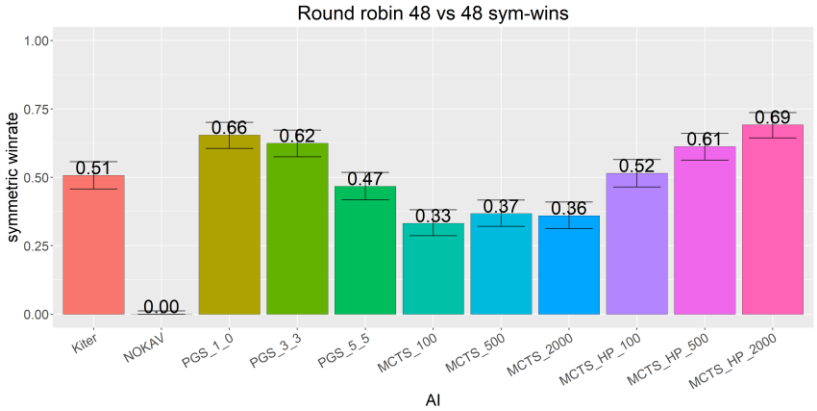

Figure 7.11 – Sym-win rates for 48vs48 large scale combat round robin tournament.

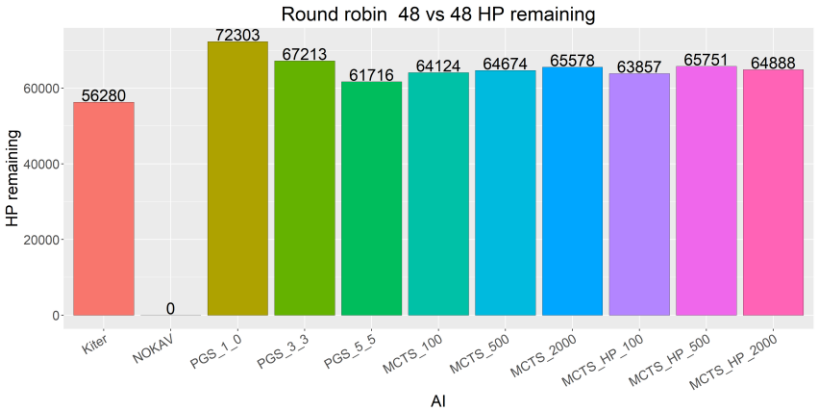

Figure 7.12 – Summed HP remaining for the winning side from all battles for 48vs48.

Results for the 48vs48 combat tournament are shown in Figure 7.10, Figure 7.11, and Figure 7.12. When we add 16 more units and enter the area of large scale combat we can see the MCTS techniques behave similarly to previous scenario. However, the differences in HP remaining are not so drastic. This may be because the scenarios generated were different and did not favor the PGS as much. Also, the PGS time limit could start playing a bigger role and some better assignments could have been skipped.

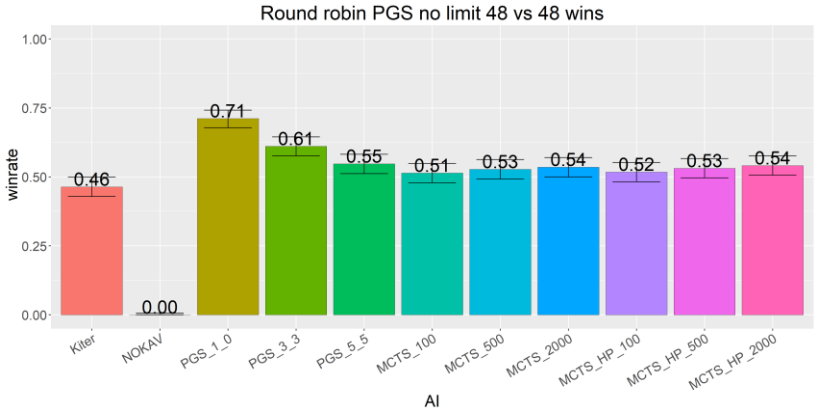

Figure 7.13 – Win rates for 48vs48 large scale round robin tournament. PGS algorithms are without time limit.

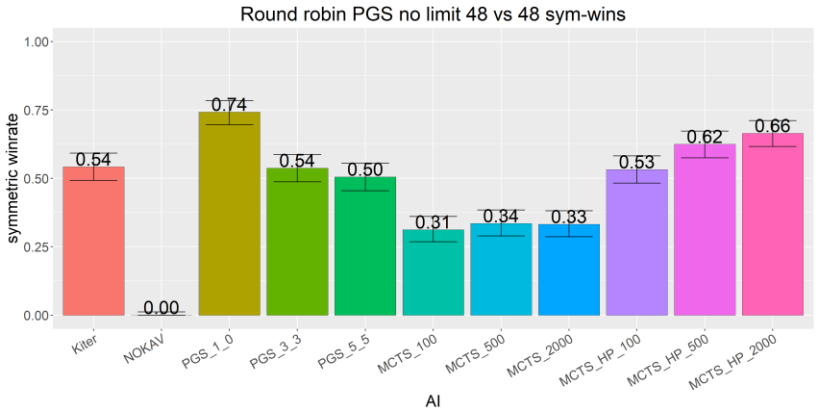

Figure 7.14 – Sym-win rates for 48vs48 large scale round robin tournament. PGS algorithms are without time limit.

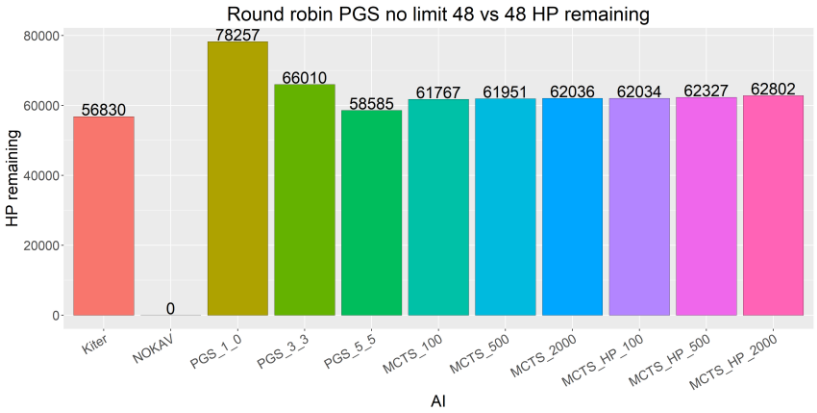

Figure 7.15 – Summed HP remaining for the winning side from all battles for 48vs48. PGS algorithms are without time limit.

To see the influence of the time limit to PGS approaches we decided to run the same 48vs48 tournament again, but this time without the time limit for PGS algorithms. Results can be seen in Figure 7.13, Figure 7.14, and Figure 7.15. PGS_1_0 still performs the best achieving 2 more percent points in win rate, 8 more percent points in sym-win rate and it doubled the gap in HP remaining between itself and the MCTS approaches. This performance gain, however, did not come cheap as we will see in the next chapter.

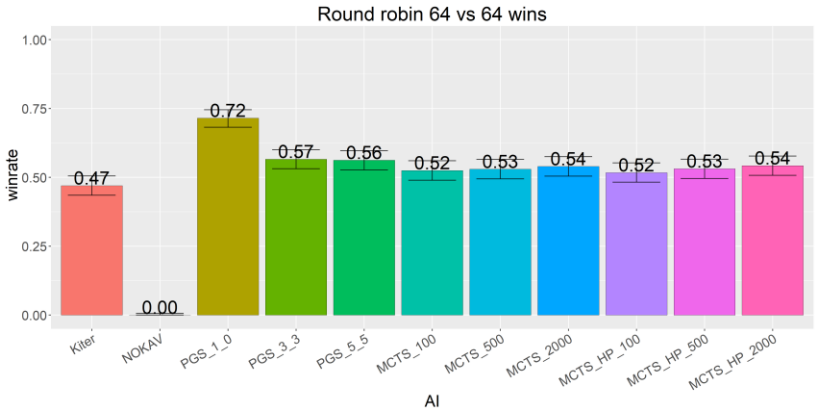

Figure 7.16 – Win rates for 64vs64 large scale combat round robin tournament.

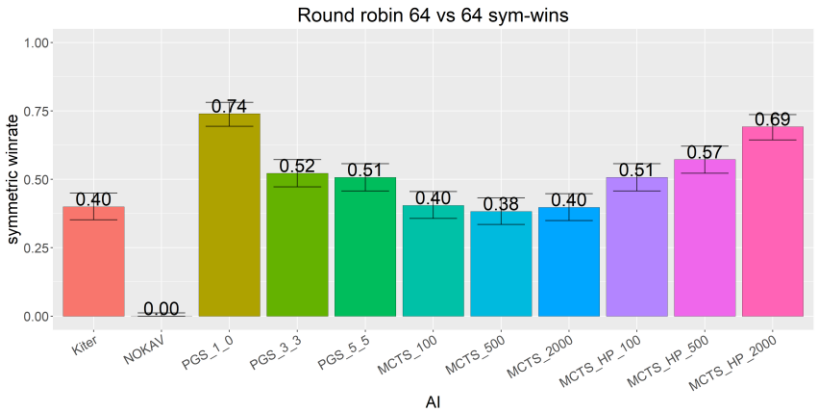

Figure 7.17 – Sym-win rates for 64vs64 large scale combat round robin tournament.

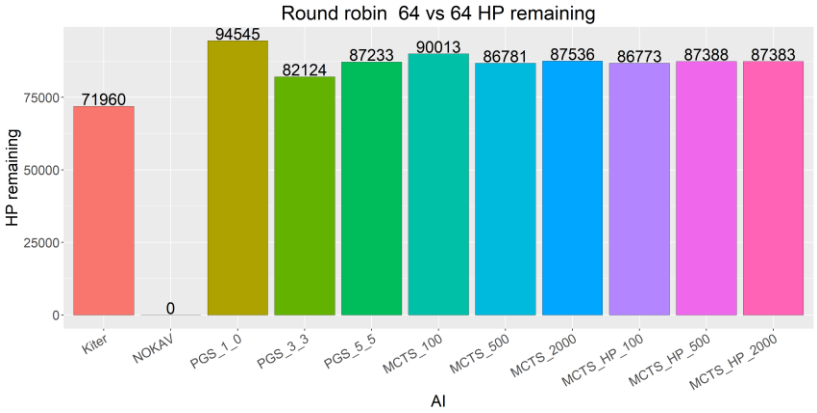

Figure 7.18 – Summed HP remaining for the winning side from all battles for 64vs64.

Finally, results for 64vs64 tournament are shown in Figure 7.16, Figure 7.17, and Figure 7.18. Here, we can see established trend where MCTS techniques are outperformed by PGS. As with the 48vs48 scenario the results are not so strong for the PGS methods probably because of the time limit for PGS. Also, having more units does not necessarily have to mean that the simulation will be slower. For example, if

we had many strong units close to each other, they could easily eliminate themselves in early stages of the playout and the playout could be faster overall; therefore, the MCTS could have more iterations to compute. However, the established trend is clearly visible and PGS is the better approach.

7.2 Execution time results

Since execution time is a crucial factor when deciding which AI method to use, we measured execution times of the whole search for each AI. In the game playout each time we ask an AI player to make its actions we measure execution time of this method. Several statistics (minimum, maximum, median, average) of these times are then saved to the database for each battle. Snice we perform multiple battles for each unit count we plot statistics aggregated over all these battles for each unit count tournament; i.e., average time in our plots is average over average times for all battles we have, the same goes for median. Maximum is chosen as a 90th percentile of all maximum values to discard potential unreasonable peaks in the execution time (in plots is represented as p90_max_time). Since a NvsN combat scenario also includes smaller combats which emerge as units die, we are more interested in the maximum execution times than median or average ones because the maximum times usually correspond to the full NvsN scenario. That is why we also include the avg_max_time statistic which represents average of maximums.

The execution times of ScriptedPlayers are under 2ms even for the largest scenario; therefore, we do not include them in our statistics. Execution times for MCTS methods are fixed by the time limit of each approach (100ms, 500ms, 2000ms); therefore, we indicated these important times in our plots by which the MCTS approaches are represented[37]. Time plots are presented in Figure 7.19, Figure 7.20, Figure 7.21, Figure 7.22, Figure 7.23, Figure 7.24, Figure 7.25, Figure 7.26, and Figure 7.27.

[37] Since we check the time limit overflow only occasionally, the MCTS methods can sometimes take more time than the limit. This is, however, negligible and min/max/avg/median statistics are all very close to the limit.

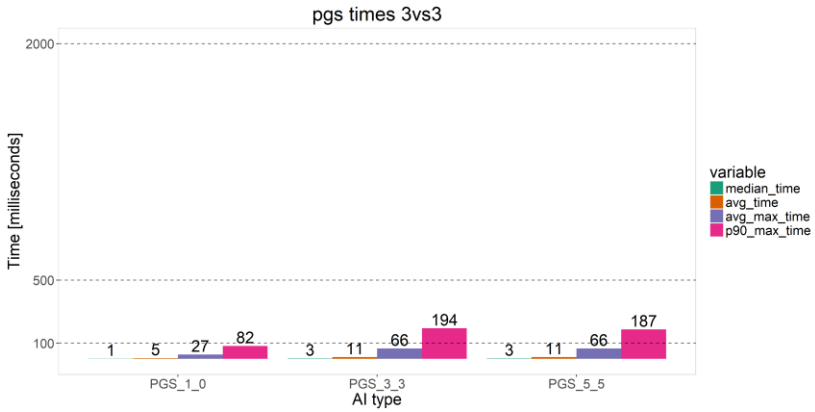

Figure 7.19 – Time plot of execution times of PGS approaches for 3vs3 combat.

In the 3vs3 combat we can see that all PGS approaches are very fast and on average execute in under 100ms. This is due to fast playouts. However, as we discussed in the results this small combat is not very interesting.

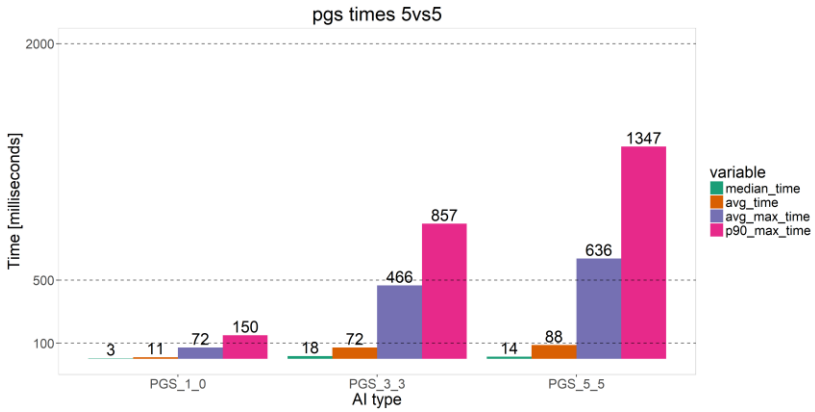

Figure 7.20 – Time plot of execution times of PGS approaches for 5vs5 combat.

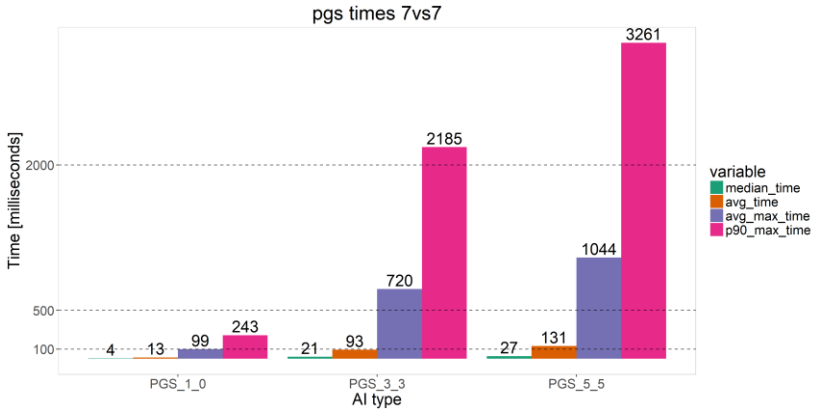

Figure 7.21 – Time plot of execution times of PGS approaches for 7vs7 combat.

For 5 and 7 units the execution times go up greatly. PGS_1_0 remains under 100ms on avg_max but with more iterations the PGS peaks at more than 2 seconds execution time for the 7vs7 scenario. Avg_max times are relatively reasonable and even for

PGS_5_5 stop around 1s. These scenarios are, however, still dominated by the MCTS methods.

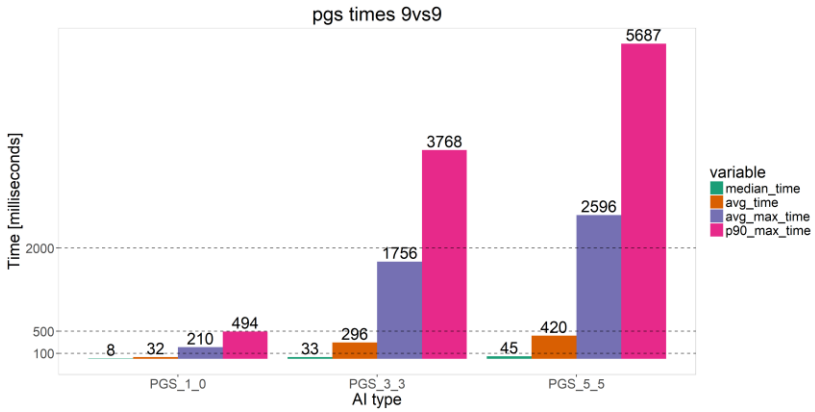

Figure 7.22 – Time plot of execution times of PGS approaches for 9vs9 combat.

9vs9 combat follows the trend of small combats where the execution times for PGS are lengthy and the performance is poor.

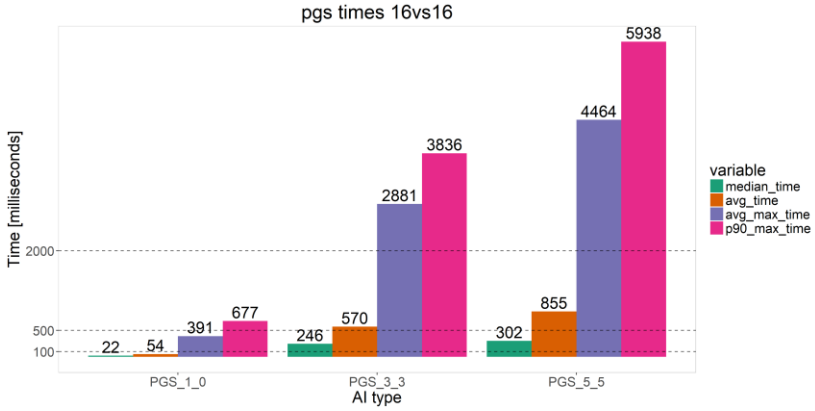

Figure 7.23 – Time plot of execution times of PGS approaches for 16vs16 combat.

In 16vs16 even the median and average times start to climb up for the PGS_3_3 and PGS_5_5. Peak times for these algorithms are much higher than what is acceptable with PGS_5_5 peaking at around 6 seconds execution time. PGS_1_0 is, however, still reasonably fast with avg_max still well under 500ms. If we look at the performance of these approaches, the PGS_1_0 may be a viable alternative to the plain MCTS approaches.

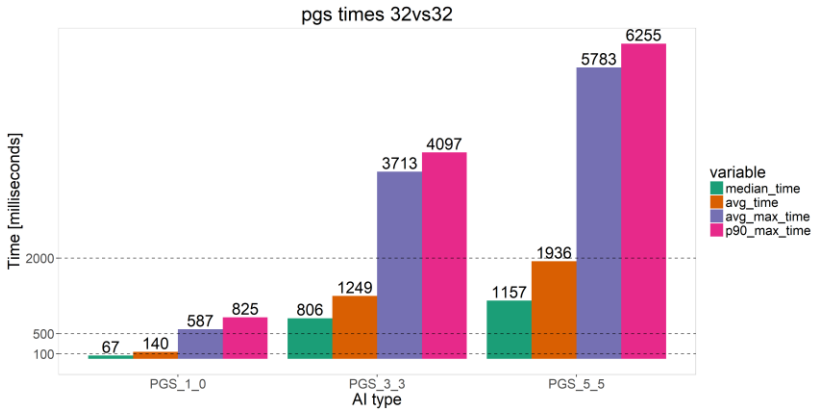

Figure 7.24 – Time plot of execution times of PGS approaches for 32vs32 combat.

In 32vs32 battles the only viable PGS approach time-wise remains to be the PGS_1_0, which is also competitive with MCTS approaches performance-wise. Even at peak times the PGS_1_0 does not exceed 1s but performs on par with MCTS_HP with 2s time limit.

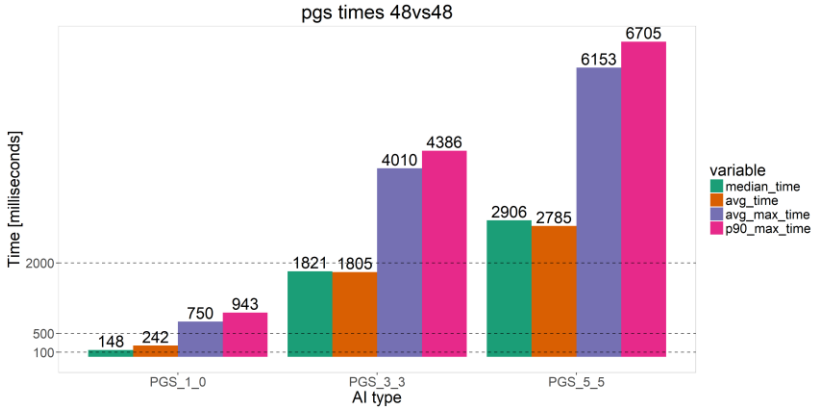

Figure 7.25 – Time plot of execution times of PGS approaches for 48vs48 combat.

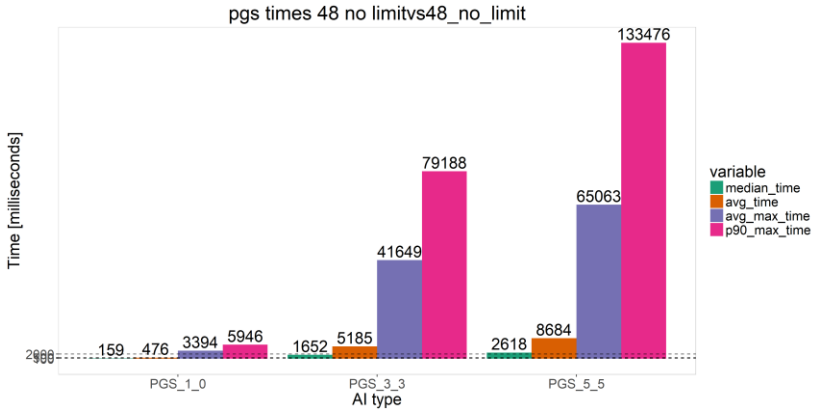

Figure 7.26 – Time plot of execution times of PGS approaches without time limit for 48vs48 combat.

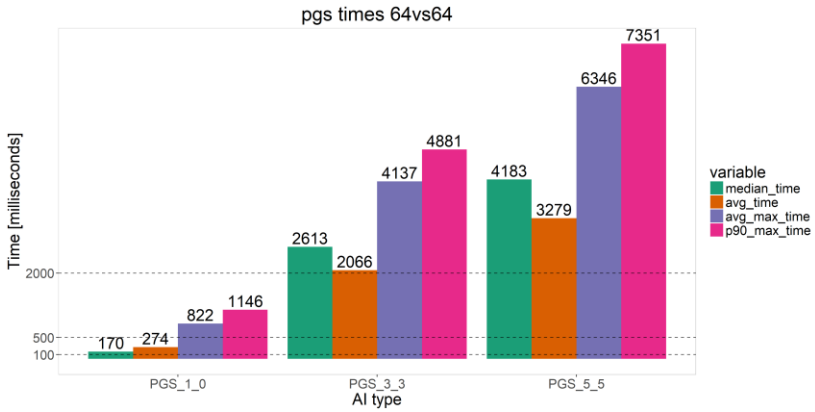

Figure 7.27 – Time plot of execution times of PGS approaches for 64vs64 combat.

In the large-scale combat scenarios the times grow even higher. However, the PGS_1_0 still performs on average maximum time under one second, which makes still usable for these combat sizes. PGS without time limit shows completely unbound time results, where the 90th percentile of PGS_5_5 shows execution time of more than two minutes. PGS_1_0 has also very high times and would be no longer very useful.

7.3 Discussion

As we expected, the more complex MCTS approaches work well on small to medium scale combat and at larger sizes greedy PGS approaches prevail. MCTS_HP proved to be equal or superior to the regular MCTS in all scenarios. Often even the MCTS_HP with 100ms for computation defeated the regular MCTS with 2 seconds for computation. The NOKAV script, representing the AI currently implemented in the game, was easily outperformed by all other methods.

It sometimes happens that the Kiter script wins more games than the search methods. For PGS, this is due to its greedy nature. Especially the versions with more iterations may generate results which assume too advanced opponent and a simple Kiter script may win against them. Another thing to consider is that there probably is not a dominating strategy (script assignment) which would win against all other strategies. Therefore, for each script assignment there exists a counter-assignment and the PGS algorithm may search in circles where the opponents try to counter each other. For MCTS, this may be due to the time limit which stopped the search too soon and the MCTS could not sufficiently sample the actions.

One interesting observation is that MCTS_HP does not achieve higher win rates compared to MCTS given the same time limit[38]. This may be because our mapping for MCTS_HP is linear and as discussed in section 5.2 does not distinguish much between weak win and weak loss. E.g., if the mapped values are close to 0 the UCB formula is not affected much and it does not matter whether the AI won or lost. This, however, corresponds to how combat in strategy games works. If from a whole army of units only one remains after a combat, we cannot say we decisively won but the result is closer to a draw and it is not very different from situation when only one enemy unit remained. What we consider a success is that even though we search for action which give us stronger wins, we do not sacrifice the actual win rate since it remains on par with the regular MCTS search techniques. It may be possible to improve it by using different mapping functions which, however, remains as a future work. When implemented in an agent paying the full game, this may not be a problem at all, since ending each battle with a little more HP than regular MCTS the MCTS_HP agent would soon have many more units, which should lead to full game victory.

Since the authors of the PGS algorithm tried only the PGS_1_0 in their work, we did not know how the versions with higher iteration counts would work. Surprising is that increasing the response and improvement counts does not seem to reliably increase win rates. This stems from the nature of the greedy approach but still, the simplest and fastest PGS_1_0 which just improves the player's script assignment once against a fixed opponent works well and sometimes better than the variations with more iterations.

The execution times of PGS variations leave a lot to be desired. Even with our caching optimization the PGS variations with more iterations do not perform well compared to how much time the computation takes. This makes the PGS_1_0 the best PGS variation in view of performance/time ratio. This fact is, however, a little disappointing considering that from full complexity of the algorithm only a single improvement against a fixed opponent strategy is used.

When measuring performance of different algorithms there are many dimensions we can optimize. One of them is pure win and sym-win rate maximization. Another one is execution time which we usually want to minimize. And then we have

[38] However, higher win rates are achievable since given more time the MCTS reaches them.

something we call execution time and performance stability where we want the algorithm to achieve stable results with stable (or constant) execution time. In computer games we usually want to find a balance between these, but the execution time and stability requirements are commonly the most crucial ones because we cannot let the players wait and we want the AI to behave similarly in terms of performance all the time. Therefore, we decided to use the 500-millisecond limit for PGS approaches and in the comparison, we can see how important this time limit is. If we pursued only the win rates we could remove or increase the time limit, however, then, as we saw, we would get very unstable algorithm which could take from a few hundred milliseconds up to several seconds and this could be frustrating to players. When limited, the algorithm still performs very well, and we get much smoother execution times which is preferred in our situation.

Based on these results we propose to use PGS_1_0 for large combats and when some units are destroyed we can switch to more accurate MCTS_HP which even with a few hundreds of milliseconds time limit outperforms all other methods.

8 Conclusion and future work

In this work we have described possible AI architecture for a 4X TBS game Children of the Galaxy. Using our custom-made behavior tree visualizer and analyzing the current architecture we have chosen a subproblem of combat because it is one of the key parts of the game and quality combat reasoning is important for higher level modules in the AI. We explored combat in context of RTS and TBS games. To solve the problem in CotG we have decided to use search methods. Common search techniques were, however, not applicable to our case because of the immense complexity and branching factors. Therefore, we adapted two search-based algorithms which worked well in previous research in RTS games, Portfolio greedy search improved by caching previous playouts and modified version of MCTS which in our case works in space of script assignments rather than in space of action assignments.

We also presented an improved version of this MCTS algorithm MCTS_HP which improves how the search is guided towards more promising parts of the tree by improved specification of the actual problem at hand. By considering not only whether the random playout ended in win or loss but also considering how good win or how bad loss was achieved we were able to overcome the fact that playouts are expensive and therefore, the iteration counts are relatively low. We propose this version of MCTS could be used in scenarios where the problem solved by the MCTS is only a subproblem of a bigger problem or where we do not care only about win or loss but where the output of the playout could be more precisely specified.

To evaluate these algorithms and integrate them to the CotG game environment we implemented a combat simulation CMS which has no dependencies on the game and can be used as a standalone testbed for future research in TBS combat. To show this we implemented CMS.Benchmark standalone application which uses CMS and allows us to run benchmarks of AI agents in controlled and fast environment. The CMS.Benchmark framework is easily configurable using XML files. Output to an SQL database is possible which drastically simplifies the management of data generated by the framework as well as enables simple distributivity to multiple computers with one central SQL server where all the data is being saved. Desired data can be then selected using SQL queries and plotted using the R software.

Using CMS.Benchmark framework, we have conducted a series of experiments with all the implemented AI approaches. These experiments showed that MCTS_HP

is strictly better than a regular MCTS on script space and is well suited for small to medium scale combat. Three PGS variations were also evaluated and the simplest one proved to be viable option for medium to large scale combat where the MCTS methods require too much time to operate effectively. The other two PGS variations with more iteration counts proved to have questionable performance and unreasonable execution time requirements even with playout caching enabled. Since our MCTS algorithms had only about a hundred iterations for more complex scenarios and low time limits we also shown that MCTS is viable option even in these scenarios where it is not possible to perform hundreds of thousands of playouts.

A sample integration of the CMS to the CotG game was also performed in a testing mission. Full state transformation and API usage is implemented in this scenario and the CMS has full control over the ships. Transforming this sample implementation to more subtle usage from the game AI would be trivial and is part of the future work.

8.1 Future work

One of the main aims of this work was to implement a part of the AI system for CotG game. While working on this system we were in touch with the main developer of the game and one of the future work for us is to discuss and plan more complete integration of our system to the game while encouraging the community around the game to use our simulator in their own AI modifications. This will consist of creating a reasonably universal AI player which will direct the units during combat and a fast combat estimation system which will be used to approximate the AI's chances. CMS is ready to serve in both roles, now it is just necessary to implement the AI systems in the game. After this implementation, very interesting future work would be to perform playtesting of various AI approaches against human players. Example battle of our current sample implementation can be watched as a video in the attachment.

Since we now have a benchmarking framework and working game simulation it would be great to use these tools to implement and benchmark more AI approaches for CotG and TBS games in general. One interesting approach may be a modified version of the Naïve MCTS algorithm. To enable faster playout and more iterations some pathfinding methods other than A* could be explored. Variations of the MCTS_HP with different mapping function could be also explored and compared to see the effect of giving more (or less) value to win/loss states. MCTS_HP could also

be modified to consider not HP but for example value of a unit, which could work better for high quality units with low HP counts.

In our case the script space search proved very useful. It is, however, very dependent on the quality and implementation of underlying scripts. As a future work different scripts and implementations should be explored. One way to go is a set of very simple but fast scripts as opposed to our current set of two relatively complex scripts.

Our improved version of MCTS dominated the regular MCTS approach in our tests. It would be very interesting to see how this algorithm would perform in other games. Note that in our case MCTS_HP searches in script space but as discussed earlier the same approach would work in regular action space as well. Our assumption is that in games such as StarCraft or Kingdom Come: Deliverance[39] where the combat is just a subproblem of the whole game and it is very important how well or bad does it end, MCTS_HP would improve the performance and guide the search better. A full game playing agent with MCTS_HP algorithm for combat should perform much better because accumulating units over time should lead to ultimate victory, e.g., if I finish battles with more units I will have more units for future battles and I am in a better spot.

[39] Explored in (Černý 2016).

Bibliography

ANDRUSZKIEWICZ, Piotr, 2015. Optimizing MCTS Performance for Tactical Coordination in TOTAL WAR: ATILLA. nucl.ai Conference [online]. 2015. [Accessed 20 May 2016]. Available from: http://archives.nucl.ai/recording/optimizing-mcts-performance-for-tactical-coordination-in-total-war-atilla/

AUER, Peter, CESA-BIANCHI, Nicolo and FISCHER, Paul, 2002. Finite-time Analysis of the Multiarmed Bandit Problem. Machine Learning. 2002. Vol. 47, p. 235–256.

BERGSMA, Maurice HJ; SPRONCK, Pieter. Adaptive Spatial Reasoning for Turn-based Strategy Games. In: AIIDE. 2008.

BOWLING, Michael, et al. Heads-up limit hold'em poker is solved. Science, 2015, 347.6218: 145-149.

BRANAVAN, S. R. K.; SILVER, David; BARZILAY, Regina. Non-linear monte-carlo search in civilization ii. In: IJCAI. 2011. p. 2404-2410.

BROWNE, Cameron B., et al. A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in games, 2012, 4.1: 1-43.

CHAMPANDARD, Alex J.; DUNSTAN, Philip. The behavior tree starter kit. Game AI Pro: Collected Wisdom of Game AI Professionals, 2012, 72-92.

CHURCHILL, David; BURO, Michael. Portfolio greedy search and simulation for large-scale combat in StarCraft. In: Computational Intelligence in Games (CIG), 2013 IEEE Conference on. IEEE, 2013. p. 1-8.

CHURCHILL, David; SAFFIDINE, Abdallah; BURO, Michael. Fast Heuristic Search for RTS Game Combat Scenarios. In: AIIDE. 2012. p. 112-117.

ČERNÝ, Martin. Reducing Complexity of AI in Open-World Games by Combining Search-based and Reactive Techniques. 2016.

FURTAK, Timothy; BURO, Michael. On the Complexity of Two-Player Attrition Games Played on Graphs. In: AIIDE. 2010.

GEMROT, Jakub. Controlling Virtual People. 2017.

GOSLING, Tim and ANDRUSZKIEWICZ, Piotr, 2014. Divide and Conquer, The Campaign AI of Total War: ROME II. Game/AI Conference Vienna [online]. 2014. [Accessed 20 May 2016]. Available from: https://archives.nucl.ai/recording/divide-and-conquer-the-campaign-ai-of-total-war-rome-ii/

MOUNTAIN, Gwaredd, 2015, Tactical Planning and Real-time MCTS in Fable Legends. nucl.ai conference [online]. 2015. [Accessed 18 April 2018]. Available from: https://archives.nucl.ai/recording/tactical-planning-and-real-time-mcts-in-fable-legends/

JUSTESEN, Niels, et al. Script-and cluster-based UCT for StarCraft. In: Computational Intelligence and Games (CIG), 2014 IEEE Conference on. IEEE, 2014. p. 1-8.

LIU, Siming; LOUIS, Sushil J.; NICOLESCU, Monica. Comparing heuristic search methods for finding effective group behaviors in RTS game. In: Evolutionary Computation (CEC), 2013 IEEE Congress on. IEEE, 2013. p. 1371-1378.

ONTANÓN, Santiago. Informed monte carlo tree search for real-time strategy games. In: Computational Intelligence and Games (CIG), 2016 IEEE Conference on. IEEE, 2016. p. 1-8.

ONTANÓN, Santiago. The combinatorial multi-armed bandit problem and its application to real-time strategy games. In: Ninth Artificial Intelligence and Interactive Digital Entertainment Conference. 2013.

ORKIN, Jeff. Three states and a plan: the AI of FEAR. In: Game Developers Conference. 2006. p. 4.

RABIN, Steve, 2015, JPS+: Over 100x Faster than A*. [online]. 2015. [Accessed 26 April 2018]. Available from: https://www.gdcvault.com/play/1022094/JPS-Over-100x-Faster-than

RUSSEL, Stuart J. and NORVIG, Peter, 2010. Artificial Intelligence: A Modern Approach. 3rd. Upper Saddle River: Prentice Hall. ISBN 978-0-13-604259-4.

SCHNEIDER, Douglas; BURO, Michael. StarCraft unit motion: Analysis and search enhancements. In: Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference. 2015.

SILVER, David, et al. Mastering the game of Go with deep neural networks and tree search. nature, 2016, 529.7587: 484-489.

STURTEVANT, Nathan R. An analysis of UCT in multi-player games. In: International Conference on Computers and Games. Springer, Berlin, Heidelberg, 2008. p. 37-49.

SYNNAEVE, Gabriel, 2012, Bayesian Programming and Learning for Multi-Player Video Games Application to RTS AI, Doctor of Philosophy, Grenoble University

TOM, David; MÜLLER, Martin. A Study of UCT and its Enhancements in an Artificial Game. In: Advances in Computer Games. Springer, Berlin, Heidelberg, 2009. p. 55-64.

WELCH, Ed, 2007, Designing AI Algorithms For Turn-Based Strategy Games. [online]. 2007. [Accessed 18 April 2018]. Available from: https://www.gamasutra.com/view/feature/129959/designing_ai_algorithms_for_.php

WENDER, Stefan and WATSON, Ian, 2008. Using reinforcement learning for city site selection in the turn-based strategy game civilization IV. In: 2008 IEEE Symposium on Computational Intelligence and Games, CIG 2008. 2008. p. 372–377. ISBN 9781424429745.

WENDER, Stefan; WATSON, Ian. Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft: Broodwar. In: Computational Intelligence and Games (CIG), 2012 IEEE Conference on. IEEE, 2012. p. 402-408.

YOOCHUL Kim; MINHYUNG Lee. 2017. Humans Are Still Better Than AI at StarCraft – for Now. MIT Technology Review [online]. 2017. [Accessed 18 April 2018]. Available from https://www.technologyreview.com/s/609242/humans-are-still-better-than-ai-at-starcraftfor-now/

List of Figures

List of Tables

List of abbreviations

ABCD – Alpha-Beta considering durations

AI – Artificial intelligence

CMAB – Combinatorial multi-armed bandit problem

CMS – Children of the Galaxy micro simulator

CotG – Children of the Galaxy

HP – Hit points

MAB – Multi-armed bandit problem

MCTS – Monte-Carlo tree search

MCTS_HP – Monte-Carlo tree search considering hit points

NOKAV – No-Overkill-Attack-Value

PGS – Portfolio Greedy Search

RTS – Real-time strategy

SC – StarCraft

SOS – Sum of Switches

TBS – Turn-based strategy

UCB1 – Upper confidence bounds

UCT – Upper confidence bound for trees

UCTCD – Upper confidence bound for trees considering durations

XML – Extensible markup language

XSD – XML schema definition

Attachments

In this section a directory structure for the attached CD and online zip archive is explained.

- **/CMS/** – Contains CMS.dll with its dependencies ready to be used in the game.
- **/CMS.Benchmark/** – Contains build of the benchmark project with sample data. May be executed by running `run_all_benchmarks.bat`.
- **/results/** – Contains csv files generated from our database which were used to create the graphs. `full_battles_table.csv` contains full dump of the whole battles table.
- **/images/** – Contains all graphs used in this thesis and some more showing more results.
- **/videos/** – Contains a gameplay video of MCTS_HP_100 versus NOKAV. MCTS_HP_100 controls yellow/red ships.
- **/benchmarks/** – Contains configurations of all benchmarks we used in this thesis. Each configuration is in a zip archive. To run it simply copy and paste the Resources directory to the CMS.Benchmark directory (overwriting the original) and run `run_all_benchmarks.bat`.
- **/src/cog-ai/** – Contains source code for the solution containing CMS, CMS.Benchmark, and the game AI module projects. Also contains the *Content* directory in which original as well as new and modified behavior trees are.
- **/src/behvisualizer/** – Contains Behavior tree visualizer and writer scripts.
- **/doc/** – Contains user documentation and generated HTML documentation for the EmptyKeys.Strategy.AI solution. The HTML documentation can be accessed by opening the `index.html` in html directory.
- **/Thesis.pdf** – Text of this thesis in pdf format.
- **/Abstract_en.pdf** – Abstract to this thesis in pdf.
- **/readme.txt** – File containing this explanation of the attachment.