TÉCNICO LISBOA

MCTS-based Planning for Grand Strategy Games

Manuel António Felizardo Roxo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Pedro Alexandre Simões dos Santos
Prof. João Miguel de Sousa de Assis Dias

Examination Committee

Chairperson: Prof. David Manuel Martins de Matos
Supervisor: Prof. Pedro Alexandre Simões dos Santos
Member of the Committee: Prof. Rui Filipe Fernandes Prada

January 2021

Acknowledgments

Firstly, I would like to thank my parents and family for supporting me throughout my academic years, as well as for their care and encouragement.

I would also like to thank my supervisors, Prof. João Miguel de Sousa de Assis Dias and Prof. Pedro Alexandre Simões dos Santos, for helping and guiding me through the development of this thesis, and for sparing time every week to meet and talk about the thesis' progress. Without their continuous assistance and guidance this thesis would not be possible.

Lastly, I would like to thank my friends for their continuous support and friendship, and for always being there for me.

To each and every one of you – Thank you.

Abstract

Using planning in grand strategy video games is a difficult task. These games are characterized by having a sizeable and complex search space and many other constraints. In contrast, there is not much computational budget available for running an AI in this setting as many resources are spent on running the game itself. The purpose of this thesis is to conceive and design different planning systems based on state-of-the-art planning algorithms, mostly based on Monte Carlo Tree Search (MCTS), as well as domain specific pruning strategies, and apply them to a grand-strategy video-game. We will focus on TripleA [18], an open source grand strategy video game engine which features a number of different maps. We implemented different MCTS variants, Bridge Burning MCTS [17] and Non Exploring MCTS [17] , as well as an evolutionary algorithm called Online Evolutionary Planning [17]. These agents were tested against each other as well as against the game's current AI solutions. Our results show that in the practical setting used, our agents are able to beat the game's AI solutions consistently.

Keywords

MCTS; Grand-Strategy; AI; Evolutionary algorithms;

Resumo

Usar planeamento to contexto de jogos de grand strategy é uma tarefa difícil. Estes tipos de jogos são caracterizados por terem um espaço de pesquisa vasto e complexo e muitos outros constragimentos face á construção de um sistema de planeamento. Em contraste, não existe muito tempo de execução disponível para correr uma IA nestes cenários, visto que muitos dos recursos são usados para correr o jogo em si. O objetivo desta tese é desenvolver diferentes sistemas de planeamento para jogos de grand strategy, baseados em algoritmos state-of-the-art, especialmente variantes do algorithmo Monte Carlo Tree Search (MCTS), e enriquecê-los com técnicas de corte do espaço de estados. Vamos focar-nos no jogo TripleA, um engine open-source de video-jogos grand strategy, que tem um conjunto de mapas diferentes. Foram implementadas diferentes variantes do MCTS, nomeadamente os algoritmos Bridge Burning MCTS e Non Exploring MCTS, bem como um algoritmo evolucionário chamado Online Evolutionary Planning. Estes agentes foram testados uns contra os outros, bem como contra as soluções de IA existentes no jogo atualmente. Os resultados obtidos mostram que os agentes conseguem derrotar consistentemente a IA do jogo no cenário usado.

Palavras Chave

MCTS; Grand-Strategy; Inteligência Artificial; Algoritmos evolucionários;

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

MCTS	Monte Carlo Tree Search
OEP	Online Evolutionary Planning
BB-MCTS	Bridge Burning MCTS
NE-MCTS	Non Exploring MCTS

1

Introduction

Grand strategy video-games focus on the management of an entire nation state, where the player has to direct all its resources and coordinate its military strategy in order to achieve a goal, including political, economic and military conflict. These video-games usually focus on war, typically over a long period of time. Some examples of these video-games are The Hearts of Iron, Europa Universalis, Supreme Ruler Ultimate and Total War series. These games usually feature a discretized map. Some games focus on real life events, and feature real geographical maps, like Hearts of Iron, whose map features the entire world, and where regions are divided into areas of different granularities, with provinces being the lower-granularity type, and then making up states and countries. Other games, like some entries in the Total war series, are based on fictional environments, and feature made-up locations, but usually maintain a similar discretized type of representation. Some games might have an hexagonal representation, like the Civilization series, or even a grid like one. Over the course of the game, the player has to make important political decisions that have impact on different scopes of the game, influence the production of resources and manage them, and interact with combat by deploying units and planning combat, ultimately deciding the outcomes of war.

Using planning in the context of strategy games, such as the grand strategy games mentioned, raises a series of problems, mostly due to the sizeable processing time that is required for such tasks, because of the significant number of actions that cause a big branching factor, and the contrasting small computational budget that is available. This thesis aims to design a system capable of providing intelligent decisions for grand-strategy games, and apply it. More specifically, the focus is for our solution to perform under the following conditions:

- Large search space and branching factor.
- Small computational budget (Most of this budget is spent on running the games).
- Stochastic environment (There is usually a level of uncertainty and randomness to actions in grand strategy games).

For the implementation, we decided to focus on TripleA, a grand strategy game based on the board game Axis and Allies. This game is a good candidate for this thesis because it presents all the challenges inherent to the thesis' objective and motivation, while also being open source, and highly adaptable to different scenarios. This facilitates the development of the algorithms and also provides a good test bed for different scenarios. It's a turn-based game, and being a grand strategy game, it is characterized by a large action space and branching factor. Additionally, the outcome of combat in the game is influenced by dice rolls, granting stochastic outcomes and randomness to the game.

Approaches based on Monte Carlo Tree Search (MCTS) present a good approach to handling planing in this case, as it provides good characteristics for dealing with the issues presented.

- By being an anytime algorithm, it is able to provide valid solutions to the problem even if it is interrupted before it ends, with the quality of the solution improving as more time is used. This characteristic makes it a great algorithm to use with a flexible computational and time budget for decisions.
- The integration of multi-armed bandit solutions in MCTS, UCT for example, cause the search tree to grow asymmetrically, as the method concentrates on the more promising subtrees. This makes MCTS well suited for large search spaces with high branching factor, as it can focus and explore the most interesting parts of the search space.
- Pruning strategies and offline knowledge introduced by variations of MCTS are usually used to better guide the search and increase the quality of actions produced in small amounts of time.
- By using playouts and sampling, MCTS becomes able to simulate the possible outcomes of random actions over the course of simulations, learning to deal with them well.

Some MCTS variants have been developed to target turn-based adversarial games specifically [17], and have been proven to increase the performance of the algorithm by a big margin under these conditions. Additional variations introduced to MCTS give it flexibility, allowing to focus and target specific areas. These variations will be explored and contextualized within our problem in the related work setting. While turn based adversarial games have a smaller branching factor compared to grand strategy games, the use of pruning strategies can lower the game's branching factor and adapt the algorithms to better suit our scenario. Another key aspect of making MCTS thrive is the implementation and adaption of the algorithm using case specific knowledge which can improve the quality of the algorithm. One of the focuses is also to explore these alternatives and see how they affect the performance of the algorithm.

2

Background

Contents

2.1 Monte Carlo Tree Search (MCTS)

Monte carlo methods are a broad class of computational algorithms that rely on repeated sampling to obtain numerical results. Their underlying concept is using randomness to solve problems that might be deterministic in principle. Monte Carlo Simulation is a subset of Monte Carlo methods that tries to obtain statistical properties of some phenomenon by sampling repeatedly and is the basys behind Monte Carlo Tree Search.

Monte Carlo Tree Search (MCTS) [1] is a method for finding optimal decisions in a given domain, by taking random samples in the decision space and building a search tree according to the results. This algorithm is based on two fundamental concepts, the fact that, given enough time, the true value of an action might be approximated using random simulation and that the obtained values may be used efficiently to adjust the policy towards a best-first strategy. The basic MCTS algorithm iteratively builds a search tree using these principles while within computational budget. In this search tree the nodes represent the game state, with current game state at the root node, and child nodes represent states attainable by taking action x from the father node. The focus of the algorithm is analysing the most promising moves, expanding the search tree based on random sampling of the search space in the given domain. The main application of monte carlo methods in this algorithm is the use of playouts, or simulations, to play the game with random decisions from a given state until the end, and using the result to weigh the nodes in the search tree. It is composed of 4 steps, the Selection, Expansion, Playout and Backpropagation steps.

Algorithm 1 MCTS algorithm pseudocode

```
1: function MCTSSEARCH(s_o)
2:     create root node v_o with state s_0
3:     while within computational budget do
4:         v_l ←TreePolicy(v_0)
5:         Δ ←DefaultPolicy(s(v_l))
6:         Backup(v_l, Δ)
7:     return a(BestChild(v_0, 0))
```

The selection step iteratively descends upon the tree using some selection criteria until a leaf node is reached. The purpose of this step is to find the most relevant node that is expandable. A node is expandable if it has child nodes that haven't been expanded and the relevance of said node is decided according to the selected decision criteria, usually taking into account the current weight of the node and its number of visits.

On the Expansion step, one or more of the unexpanded child nodes of the node selected in the previous step are added to the tree and one of them is selected for simulation in the next step. These unexpanded nodes correspond to game states attainable by selecting actions yet unexplored from the selected node.

The simulation step is responsible for completing a playout, starting from the game state of the expanded node. A default policy is used to decide which actions are taken at each step. The basic algorithm suggests the use of a random policy as the default policy in this step.

Finally, the backpropagation step sees that the result obtained at the end of the simulation step is correctly impacting nodes on the path from the root node to the expanded node, updating their statistics (their number of visits and victories) based on the outcome. Fig. 2.1 illustrates one iteration of Monte Carlo Tree Search and its steps.

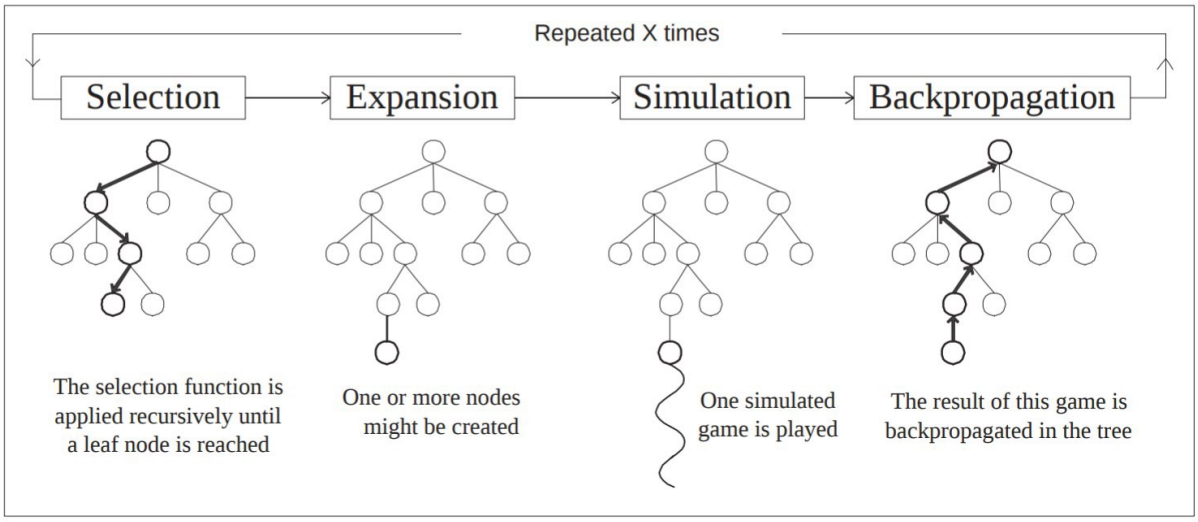

Figure 2.1: Monte Carlo Tree Search steps [15]

The search ends when a set number of iterations is ran or when a given threshold on computation time is exceeded. When the algorithm finishes, the best action is returned by the algorithm. There are multiple ways to select the best child of the root node and its corresponding action:

- Max child - The max child is the child that has the highest value.
- Robust child - The robust child is the child with the highest visit count.
- Robust-max child - The robust-max child is the child with both the highest visit count and the highest value. If there is no robust-max child at the moment, more simulations are played until a robust-max child occurs.

2.2 Upper Confidence bounds applied to Trees (UCT)

The most popular algorithm in the MCTS family is the Upper Confidence for Trees (UCT) algorithm, proposed by Kocsis and Szepesvári in [1]. The main idea behind UCT is to reduce the sampling on all

stages of the tree by identifying sub-optimal actions and using the optimality of an action as the main criteria when selecting nodes.

This algorithm aims to improve the performance of the vanilla MCTS algorithm by refining the node selection process using the UCB1 bandit-algorithm in MCTS selection. This modification ensures that the exploration exploitation problem that arises when exploring the tree is handled, balancing between testing alternatives that look the most favoring so far, and exploring unpromising alternatives that might not have been explored sufficiently, and so are perceived incorrectly. Given enough time, the UCT algorithm theoretically converges to Minimax. Each node selection iteration is treated as a multi-armed bandit problem, and the node that maximizes the formula 2.1 is chosen.

$$UCT = X_j + C * \sqrt{\frac{ln(n)}{n_j}} \tag{2.1}$$

In this formula, Xj is the win ratio of the child, n is the number of times the parent node has been visited, nj is the number of times the child node has been visited and C is a constant to adjust the exploration.

3

Related work

Contents

In this chapter we are going to talk about research that is relevant to this thesis. We will talk about state-of-the-art MCTS based algorithms and variations used for different settings, as well as industry specific implementations relevant to our case.

3.1 Real-world implementations of MCTS in TOTAL WAR- ATTILA

Total War: Attila is the ninth standalone game in the Total War series, released by creative assembly, in 2015. For this game, an AI approach using MCTS was implemented, which was described in the nucl.ai 2015 conference, on the ”Optimizing MCTS Performance for Tactical Coordination in TOTAL WAR- ATTILA” presentation [13].

This is a grand strategy video game with a mix of real time battles and turn based strategy. The implementation focuses on the campaign side of the game, where the player has to manage the economy by managing construction, taking care of public order in their settlements, establish alliances and trade agreements, pursue wars, recruits new forces and conquer new regions. Each game has a large dimension, with many factions and regions, making decision making in its context a complicated problem with a big search space. Each army can move only a set distance per turn on the hex based map, that becomes increasingly complicated by featuring different terrain features.

The purpose behind using MCTS in this game comes from its ability to handle the large search space, to allow for fine-grained control of performance, and the fact that it's easily extensible. This allows MCTS to perform even in future iterations of the game, and be used as a tool in future games with minimal effort. In this implementation the standard UCT algorithm is used as its basis, using a set of different domain specific alterations that are explained further ahead, and are applied in order to make the use of MCTS viable. The solution developed focuses on the tactical coordination sub-problem. It takes as input the set of units, and outputs an ordered list of actions to be performed, tying to protect its assets and maximize enemy casualties while minimizing its own.

Due to the complexity of the search space, it would be too costly to search more than one turn ahead of the current one because of the available execution time for the algorithm. Because of this constraint, the developers developed a solution which is able to look at most one turn ahead.

3.1.1 World Representation.

The world representation for the search subsystem is separate from the game engine. It is a compact and simplified representation of the game state, where unnecessary information, like general names and each specific building in enemy camps, is omitted. Some difficult to analyze concepts like public order, a measure of how happy the populace of a province is, are abstracted. If public order deteriorates to a certain level in a settlement, a rebellion can spawn, which will grow overtime and possibly take

over a settlement. Since MCTS analyzes only one turn ahead, modelling this occurrence is difficult. To encourage AI to enforce actions that will affect public order, the likelihood of a rebellion is assessed and a modifier is added to the score to account for this, instead of modelling this concept in a more complex and expensive way.

3.1.2 Threat Analysis.

Dealing with threats, deciding the immediate danger they present and on which locations, is difficult. For each army in the game, a cached influence map which can be composited very quickly and represents what it can threaten in one turn, is created. This means that the influence map for each enemy army can be merged quickly and serve as a good indicator of enemy threat at each location. If the expected enemy strength is higher than the AI's own army strength, then the fight is assumed to be a lost one, and is not considered. This approach always assumes a worst case scenario, and it might be the case that not all enemy armies will cooperate to destroy its units. For example, there might be a situation where AI decides to besiege 3 different settlements and there's an enemy army that can threaten each and every one of those, but it isn't able to destroy all of them in one turn, only being able to get to one. Because of this worst-case scenario assumption, AI is no longer able to maintain and go for a trade-off in this situation, where it might otherwise have been able to acquire two settlements by losing one army. Performance issues are the main bottleneck for implementing MCTS in this context, and the developers argue that this change is a valuable one, as the trade-off between quality and performance makes it worth. The tree space that is cut from the search improves the quality of the solution because the remaining sub-tree is more likely to return better solutions, given the search time.

Even with that change, the initial performance (time of MCTS execution) was disappointing, not being able to provide decidisions in effective time. In spite of this, the quality (value of chosen actions) of results was impressive.

3.1.3 Pruning Strategies.

While many targets were unreachable, a big chunk of the execution time was being used on path finding queries between interacting agents. In order to reduce the portion of time used with path finding queries in this scenario, influence maps were used to check if targets are reachable, and queries were only performed after.

Unwinnable battles, which almost always lead to bad outcomes, were also seen as an opportunity to prune, in this case the action space. If an army target can't be defeated, then it can be safely pruned away as no optimal action comes from targeting it.

Actions where the likelihood of success is below a certain threshold are also pruned away. Addition-

Figure 3.1: Influence maps for threat analysis in Total War: Attila [13]

ally, plans that have actions that depend on chance are post processed to execute these types of actions as early as possible. The objective of this change is to commit as few resources as possible when there might be need for re-planning. This way, the re-planning that has to be done in case of failure can be done as early as possible and with the AI having committed the smaller amount of resources necessary.

There are also many sub-trees that result in identical world states. An example might be plans that include actions that are unrelated to each other (independent effects), which can be ordered in any way, and still produce the same outcome in the final state. The search tree was divided to reduce the number of these equivalent tree paths. An arbitrary ordering scheme was implemented to prevent duplicates, by using a unique character index, specific to each unit, and prohibiting characters with a lower index from acting against someone if a character with a larger index has already moved against it. This change removes multiple sub-trees for the same set of actions.

Actions that affect other actions are also explored closer to the root node, and the search was divided into sub-phases. For example, actions that impact the combat units and might increase their strength are always done before battle so that the odds of winning are increased.

These changes allowed the AI to focus its computational budget, but even after this step, there were still too many sub-trees with identical outcomes and most of the time was spent on the pathfinder. Influence maps were then used once again, this time to determine maximum target sets and permit armies to act against all those targets, regardless of the distance, which allows the skipping of path-finding queries all together. If an action sequence whose score is better than the current best solution

is found, then the validity of all the actions performed in that sequence are validated by backtracking up the tree. If invalid actions are found, then the sequence is pruned.

3.2 Rapid Action Value Estimate (RAVE)

In order to produce a low-variance estimate of action-value of each action and state pair, UCT has to sample each action multiple times. This represents a problem, especially in settings with sizeable state and action spaces where its learning rate becomes really slow. Rapid Action Value Estimation (RAVE) [2] tries to atone for this issue by creating an abstraction of actions from states. In this variation, the value of selecting an action A in state S is determined by episodes where a is selected in any consequent state, instead of only considering episodes where action a was selected immediately. The algorithm is thus able to gather larger amounts of samples for each action in the same time-frame, speeding the learning rate, and decreasing the variance of the action-values. One problematic side-effect of this method is the bias since the value of an action is tied to the state it is used on. Because of this, a weight B is used to balance this metric, promoting its impact in the early iterations, and reducing it the longer the algorithm runs.

In the original paper, results are compared to at-the-time state of the art approaches that didn't use UCT, resulting in win-rates ranging from 69% to 92% depending on the number of iterations.

3.3 PoolRAVE

PoolRave is a MCTS RAVE improvement advanced by Rimmel et al in [3]. This improvement tries to artificially insert domain knowledge into the playout phase by using RAVE-good plays. When deciding which action to play with the default policy in the playout phase, the best action according to RAVE, A, is selected instead with a fixed probability p. In this case, the Rave values are those of the last node with at least 50 simulations. By doing this, some of the actions through playouts are well-guided, while still maintaining the exploration/randomness factor of MCTS, depending on p. The results in the paper demonstrate slight increases in performance over the standard RAVE-MCTS for the game of havannah, and more accentuated improvements in GO.

3.4 Move Average Sampling Technique (MAST)

Move Average Sampling Technique (MAST), used by Cadia Player [7] in its approach to the 2008 general video-game playing competition, is similar to PoolRave, with both algorithms only differing in the way in which they store statistics for actions. In PoolRave, action A's statistics for state S are influenced not

only by the value associated with A at S, but also by the value of selecting A at subsequent states. In MAST there is no association between states and actions, it simply stores a mean value for each action, independent of the state it was taken at. Both algorithms then use these statistics to guide simulations, taking the best action according to these values with probability p, choosing with the default policy otherwise. MAST can thus be seen as a simplification of the PoolRave algorithm. However, MAST can outperform PoolRave when the number of playouts is small, as proven with practical results in [12].

3.5 Last-good-reply (LGR) and N-gram

Last-good-reply(LGR) and N-gram are quite similar MCTS playout improvements based on the same principles. The principle behind both of these improvements is the idea that if a move is a good reply to another, in a specific state, it might also be a good move in a different state, which is a notion foreign to vanilla MCTS. LGF does this by associating a move A as a good reply to a different move B, if move A is played after move B in a playout and said playout results in a win. LGF can also associate a move to more than one previous sequential moves. This variant, called LGF-N, denotes a last-good-reply algorithm where, instead of associating move A as a reply to a single move B, it is associated to the N previous moves made. Additionally, in [6], the notion of forgetting is advanced in the setting of LGF, presenting Last-good-reply with forgetting (LGRF), a similar algorithm where the last good reply is forgotten if it is played in a losing playout.

While the last-good-reply variation was advanced by Peter Drake in [4] directly in the context of the Monte-Carlo tree search, N-grams were presented earlier, in 1950, by Shannon in [5], in the context of predicting words. N-words are introduced as a concept to predict the Nth word of a phrase, given that the N-1 previous words are known.

In [6], Stankiewicz et al. study the effects of all these different LGR variations, using N $\in \{1, 2\}$ for each variant, in a practical setting, using the game of havannah and testing against UCT-RAVE. They also test the conjunction of N-grams with LGF and the use of offline knowledge in the UCT formula. In the tested setting LGRF-1 shows significantly better win-rates than the alternatives, displaying even further improvements when used in conjunction with N-gram. While the use of offline knowledge is tightly connected to the game of havannah in this case, lacking generalization to different games, it still highlights a good opportunity to use offline knowledge in different contexts as a way to significantly improve MCTS performance using these variants.

3.6 Progressive Bias and Progressive Unpruning

Progressive bias and progressive unpruning [8] make use of domain and heuristic knowledge in order to direct the search and reduce the branching factor, respectively. Progressive bias makes use of heuristic knowledge to better guide the search taken in MCTS. To ensure that the strategy still converges to a selection strategy, the influence of the heuristic knowledge in the search process is dictated by the current number of iterations, having a higher impact the less games have been played. The UCT selection formula is changed to the formula 3.1 where $f(n_i)$ is added.

$$v_i + C \times \sqrt{\frac{lnN}{n_i}} + f(n_i) \tag{3.1}$$

In the original paper, $f(n_i)$ is set to $\frac{H_B}{n_i}$, where H_B represents the heuristic knowledge and n_i the number of visits to the current node, as in the standard UCT formula.

Progressive unpruning [8] also makes use of this heuristic knowledge, but operates in a different setting. Instead of guiding the search of tree, it prunes the action space in function of the available runtime. This solution consists of, firstly, reducing the branching factor artificially when the selection function is applied, and then increasing it progressively as more time becomes available. Often times MCTS performs poorly if there is not much time available and there is a high branching factor. This variant tries to solve this problem by limiting MCTS to explore only the most promising actions, in function of the available time.

3.7 Fast Evolutionary MCTS

Evolutionary algorithms are population-based optimization algorithms based on biological evolution that use some of its concepts, such as mutation. Possible solutions to the optimization problem play the role of individuals, and the fitness function is used to determine the quality of the solutions. These algorithms are usually run in iterations, with each iteration evaluating a set of individuals, and deciding which are given continuity. If an individual provides a good evaluation by the fitness function, it is granted continuity in some way, which is usually domain specific. An example of these algorithms might be finding the best weights for a neural network, where each individual is characterized by its values for each weight. If an individual has good fitness, it might be selected for continuity and even mutated (a new individual is generated based on the original with slight variation) in future iterations.

The main idea behind Fast Evolutionary MCTS [9] is using an evolutionary algorithm to rapidly adapt the behaviour of the MCTS algorithm. Each individual is in practice used as policy, that will be used to guide the playout, and the results of these playouts will be then used as the fitness function to create statistical data to determine how good of an individual (or policy) it is. Features associated with the

state are extracted, and each individual is characterized by a vector of parameters that map to the set of features, giving a probability distribution over the set of actions, which represents the policy. MCTS is thus able to learn a policy on the problem's set of states by using playouts to create and evaluate individuals that decide which actions are best.

Algorithm 2 Fast Evolutionary MCTS algorithm pseudocode

```
1: function MCTSSEARCH(s_o)
2:     create root node v_o with state s_0
3:     while within computational budget do
4:         for w in evo.CurrentGeneration() do
5:             initialize statistics object stats (e.g average)
6:             for i=1 to K do
7:                 v_l ← TreePolicy(v_0)
8:                 Δ ← Playout(v_l, w)
9:                 Backup(v_l, Δ)
10:                UpdateStats(stats, Δ)
11:            evo.setFitness(w, Δ)
12:        evo.newGeneration()
13:    save evo.CurrentGeneration()
14:    return a(BestChild(v_0, 0))
```

These individuals are kept in a population, from where they are extracted to be used and evaluated. Each population corresponds to an iteration of the algorithm. In order to generate the individuals for the next iteration, or the next population, the individuals with highest fitness are selected for continuity, which can be done in different ways. Commonly, this is achieved by crossing over the different individuals selected (merging each individual with another by creating a new one with a portion of each one's weights) and then mutating new individuals (varying its weights slightly, usually one of the weights).

Every individual of the evolutionary algorithm is retrieved sequentially from the current generation and evaluated by K playouts of the MCTS algorithm. This is done because evaluating each individual in a single playouts would introduce bias. When each individual in the current generation has been iterated upon, the next generation is created. When execution of the base MCTS algorithm ends, the current generation is saved to be used in future MCTS iterations, and the best child according to MCTS is returned. The algorithm's pseudo-code is presented in algorithm 2. By using playouts to calculate fitness for each individual the algorithm has access to a large quantity of data, and is able to improve the weights rapidly. In the original paper, features were hard coded according to the problem, and each individual maintains a vector of parameters for each action. Each action's value is calculated according to formula 3.2.

$$a_i = \sum_{j=1}^{n} w_{ij} f_i \tag{3.2}$$

A softmax function is then used to, using each action's value, calculate the probability distribution for

each action.

3.8 Knowledge-Based Fast Evolutionary MCTS

Knowledge-Based Fast Evolutionary MCTS [10], an improvement on Fast Evolutionary MCTS, adds the concept of knowledge base to the genetic evolution environment, defining the fitness with which to classify and evaluate the different individuals. This method was originally developed for General Video game Playing [14], a scenario where AI solutions are tested against various unknown video games, and the actual game rules and logic can change from game to game, but the activation of such rules corresponds to the collision of the avatar (character being controlled by the agent) and other game elements like enemies, items, etc. Because of this each piece of knowledge, or event, corresponds to these collisions and the algorithm tries to learn in this way which collisions are good, and which are bad. Two notions are introduced, curiosity and experience. The former refers to discovering the effects of collisions with unknown effects thus far, and the latter represents the exploitation of collisions that provided a score gain in the past. While UCT addresses the exploration/exploitation problem in the search tree, KBFE does this with playouts. It does so by influencing the fitness of Fast Evolutionary MCTS to benefit, not only actions that seem good, but also actions that present new knowledge.

For each event, the number of occurrences and the average score change caused by it, are kept as statistics, which are updated when each playout ends. When a playout finishes, different metrics are calculated, all with the purpose of updating the knowledge base:

Δ R - The effective score change between the beginning and the end of the playout.

Δ Z - A measure of curiosity that measures the increase in all event occurrences, or collisions, over the playout, being higher when playouts produce more events, and benefiting events that have been rarely seen.

Δ D - Is a distance metric that measures the change in distance to each type of sprite. This metric increases if, during the playout, the avatar gets closer to unknown sprites or ones that provided a score increase. This metric is thus used to value the character's approximation to good or unknown collisions. The following equation defines how the score for the playout is obtained, using these 3 values:

$$Reward = \begin{cases} \Delta R & : \Delta \text{ R} \neq 0 \\ \alpha \times \Delta Z + \beta \times \Delta D & : \text{Otherwise} \end{cases} \tag{3.3}$$

3.9 AlphaGo

AlphaGo [11] is a computer program that plays the board game Go developed by DeepMind, which represents arguably the most significant AI achievement of recent times. AlphaGo became the first Go

computer program to beat a human professional on a full-sized board in 2015, and later, in 2016, it beat one of the best human player, Lee Sedol, which was well-documented in a famous documentary.

AlphaGo uses Monte Carlo tree search together with two neural networks, that take the game state as input. The first neural network, the “policy network”, determines the probability of selecting each move by creating a distribution over the possible moves. This information is used to select which actions are explored and expanded in the search tree and effectively prune the action space. The second, the ”value network”, tries to predict the chances of winning the game, given the current state. It is used to evaluate intermediate states in playouts, thus removing the need to run games until a terminal state, and increasing the performance of the search. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. The policy network was trained initially by supervised learning to accurately predict human expert moves, and was subsequently refined by policy-gradient reinforcement learning, while the value network was trained to predict the winner of games played by the policy network against itself.

Both neural nets complement Monte Carlo tree search and are able to be used in conjunction with it really well. The ”policy network” allows for the pruning of most actions in MCTS, reducing the search space by a big margin. The ”value network” removes the need for the algorithm to run games extensively until the end during the playout phase, while still providing a good evaluation of the game states, thus improving MCTS performance drastically.

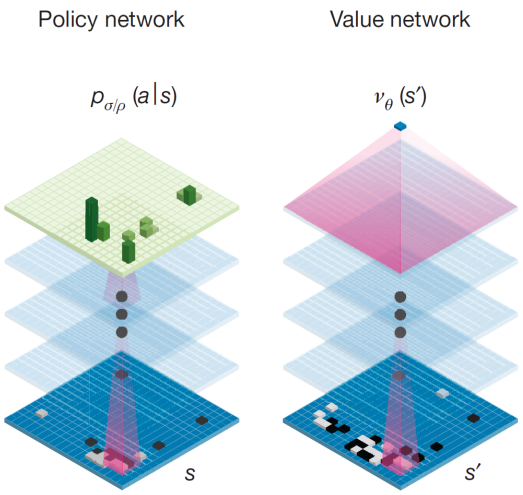

Figure 3.2: AlphaGo’s neural networks [16]

3.10 Playing Multi-Action Adversarial Games [17]

J.Togelius et al addressed the problem of playing turn based multi-action adversarial games [17]. This type of games include many strategy games with extremely high branching factors as players take multiple actions each turn. While Go and Chess have a branching factor of 300 and 30 respectively, most turn-based multi-action adversarial games have a branching factor with way higher magnitude since these games have multiple actions and multiple units.

In this paper three new algorithms which target the these types of games are introduced. The performance of different MCTS based algorithms is studied, as well as the introduction of two new MCTS variant, as well as the Online Evolutionary Planning algorithm.

The testbed game used for testing the performance of the different algorithms studied in [17] is the game Hero Academy. Hero Academy is a two player turn based tactics game inspired by chess. Each player has a number of units and spells which can be deployed and used on grid-shaped map of 9×5 squares. There are different class of units which feature different roles in the game and have access to different actions. Additionally, the game map also features special cells which unlock certain actions. The most central mechanic of the game if the usage of action points (AP). Each turn the player is given 5 AP, which can be spent to perform actions in the respective turn.

Figure 3.3: Hero Academy, the testbed for [17]

In this paper, Hero Academy itself serves as the forward model and the fitness of an action sequence is calculated as the difference between the values of both players' units. Both the units on the game board as well as those still at the players' disposal are taken into account. The assumption behind this

particular fitness function is that the difference in units serves as a good indicator for which player is more likely to win.

Complexity analysis

The action point mechanic of Hero academy causes the number of future game states to be significantly higher than in most other games, which makes the game challenging for decision making algorithms. The branching factor of the game is hard to calculate precisely, but is estimated by the authors, counting the number of possible actions in a recorded game. The branching factor is estimated to be 60 per action on average, which results in 60^5=7.78×10^8 branching factor per turn. Additionally, using the average game round duration of 40 rounds, the game-tree complexity is estimated to be $((60^5)^2)^4 0 = 1.82 \times 10^{711}$. The state space of the game is estimated to be 1.5×10^{199}.

3.10.1 Non-exploring MCTS

The first Monte Carlo tree search variant adapted for multi-action adversarial games introduced in [17] is the Non Exploring MCTS (NE-MCTS). This variant uses a non exploring policy in the tree search, and deterministic playouts, while still ensuring that each children of a node is still visited at least once before any of them is expanded further. Due to the complexity of multi-action adversarial games, vanilla MCTS isn't able to expand further than the current turn in the search tree. By removing exploration in the search phase, the resulting tree is more unbalanced and guided into better performing actions more heavily. A limited form of exploration is still present, since all children of a node are visited before a new one is expanded. The idea behind this algorithm is to heavily guide the search tree toward promising action sequences in order to efficiently obtain complete action sequences in the large search.

Since playouts are deterministic, the fact that some nodes might be visited only once has no impact on the result. The result obtained can not be corrupted by bad luck outcomes.

3.10.2 Bridge-burning MCTS (BB-MCTS)

Bridge Burning MCTS (BB-MCTS) is another MCTS variant introduced in [17]. Contrary to the Non Exploring MCTS variant, and similarly to vanilla MCTS, this variant makes use of exploration, both during the playout and the tree search phases. Playouts are ran using an e-greedy policy with e=0.5 and the exploration factor in the tree search is C=$1/\sqrt{2}$. In this variant, in order to guide the tree efficiently enough for multi-action adversarial games, the time budget is split into a number of sequential phases equal to the number of actions in a turn, with each phase locking a new move. In the end of each phase, all but the most promising node from the root are pruned and will never be added again, and the most promising node from the root acts as the root node for the next phase. The search behaves similarly to

MCTS during each phase. This approach can be seen as an aggressive progressive pruning strategy, which ignores parts of the search space in order to enable the search to reach deeper plies of the tree. The name Bridge Burning emphasizes that the nodes are aggressively pruned and can never be visited again.

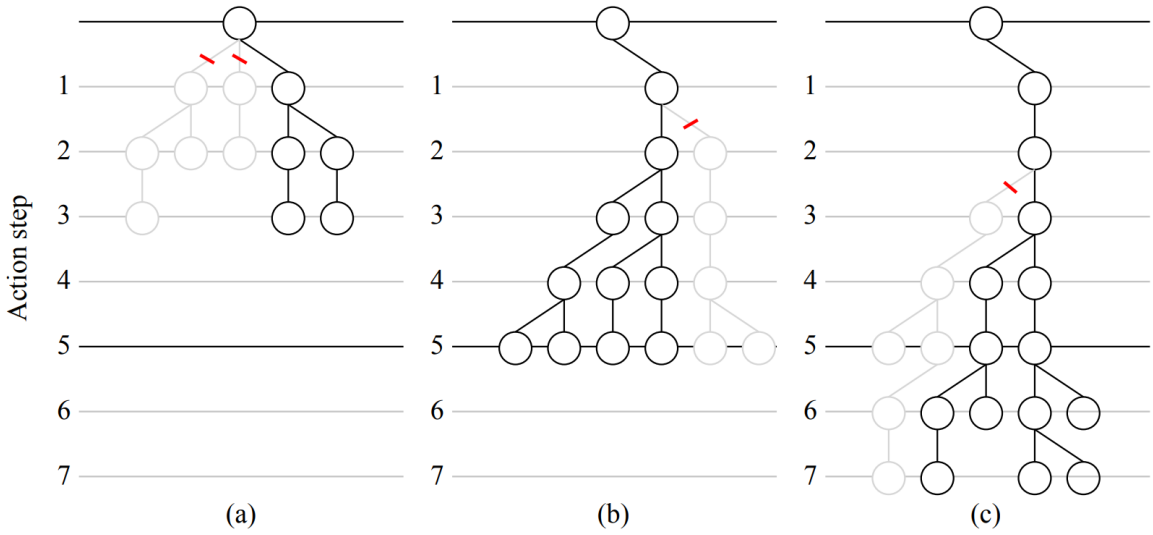

Figure 3.4: Tree structure as evolved by the Bridge Burning algorithm [17]

Fig. 3.4 illustrates how nodes are pruned by the Bridge burning algorithm in a multi-action game with 5 actions in a turn.

3.10.3 Online Evolutionary Planning (OEP)

The main algorithm proposed in [17] is an evolutionary algorithm called Online Evolutionary Planning (OEP), which aims to evolve optimal action sequences every turn.

An exhaustive search is not able to explore the entire space of action sequences within a reasonable time frame and may miss many interesting choices. As stated in the Fast Evolutionary MCTS section, evolutionary algorithms iteratively optimize an initially randomized population of candidate solutions. Because of this, an evolutionary algorithm can explore the search space in a very different way.

In this algorithm, each genome represents action sequences equal to an entire turn. In the paper's practical setting a genome is modeled to have five actions, which are described by type and one or more locations.

The initial population is composed of random genomes, which are created by repeatedly selecting random actions based on the given forward model. This process is repeated until no more action points are left. After the creation of the initial population, the population is improved over a large number of generations until a given time budget is exhausted.

After the evaluation, the genomes with the lowest scores are removed from the population. The remaining genomes are used for procreation, generating new genomes. To achieve this, each one of the remaining genomes is paired with another randomly selected genome and the new genome is created through uniform crossover. Crossing two genomes randomly can lead to illegal action sequences. To avoid this problem, the crossover checks for legality of each move when combining the two sequences.

The heuristic function used to calculate the fitness of genomes in OEP was based on playouts. This was done with the intent of incorporating information about possible counter moves. Due to the large branching factor of the game, stochastic playout evaluations were tested and determined to be unreliable, so deterministic playouts were used instead.

Algorithm 3 Online Evolutionary Planning (OEP) algorithm pseudocode

```
1:  function ONLINEEVOLUTIONARYPLANNING(State s)
2:      Genome[] pop = ∅
3:      INIT(pop, s)
4:      while time left do
5:          for w in evo.CurrentGeneration() do
6:              State clone = CLONE(s)
7:              clone.update(g.actions)
8:              if g.visits = 0 then
9:                  g.value = EVAL(clone)
10:             g.visits++
11:         sort pop in descending order by value
12:         pop = first half of pop
13:         pop = PROCREATE(pop)
14:     return pop[0].actions
15:
16: function INIT(Genome pop, State s)
17:     for x = 1 to POP SIZE do
18:         State clone = CLONE(s)
19:         Genome g = new Genome()
20:         g.actions = RANDOMACTIONS(clone)
21:         g.visits = 0
22:         pop.add(g)
23:     return pop[0].actions
24:
25: function RANDOMACTIONS)(State s)
26:     Action[] actions = ∅
27:     Boolean p1 = s.p1                                  ▷ Who's turn is it
28:     while s is not terminal  s.p1 = p1  do
29:         Action a = random available action in s
30:         s.update(a)
31:         actions.push(a)
32:     return actions
```

4

Case Study - TripleA game

Contents

In this chapter we introduce the testbed used for the experiments in this thesis and for which the algorithms were implemented, which is an open-source grand strategy video-game called TripleA.

4.1 TripleA game

TripleA is a turn-based grand strategy game engine based on the Axis and Allies board game. TripleA games involve various scenarios between world powers depicted on maps of varying size and complexity. Many maps for individual games are historical, being primarily based on relative strength and position of world powers leading up to the Second World War.

TripleA comes with multiple maps, has a considerable pool of individually developed maps which can be download, and also allows for the creation of new maps. The game features multiplayer and several AIs for single-player mode.

TripleA presents a great test bed for the development of this thesis. All of the game's characteristics align with the performance challenges that we aim to target. The flexibility of the game, by being more of an engine, and not a game, allows for the adaptation of test scenarios, and we can create our own game map and gameplay adapted to our case. From the pool of available open source grand strategy games, it presented the best characteristics for our goal. In fig. 4.1 the standard game window is shown.

Figure 4.1: TripleA game window

4.2 Map

TripleA games involve various scenarios between world powers depicted on maps of varying size and complexity. Each game features a discrete map divided into areas, or territories, and units are moved around the map between adjacent areas. Land areas are called territories while sea areas are called sea zones.

Fig. 5.3 demonstrates one of the default game maps available. Each land territory is always controlled by some player, and can change hands if an enemy land unit conquers and occupies it. The color of the territory on the map indicates who controls it.

Each territory has a production value which determines how many units can be produced there per turn and determines how much income that territory provides per turn to the player controlling it. Certain territories contain “Factory” units, which allow the controlling player to produce units in that territory. A territory with a factory may produce a number of units up to the production value of the territory, if the player owned that territory at the beginning of their turn.

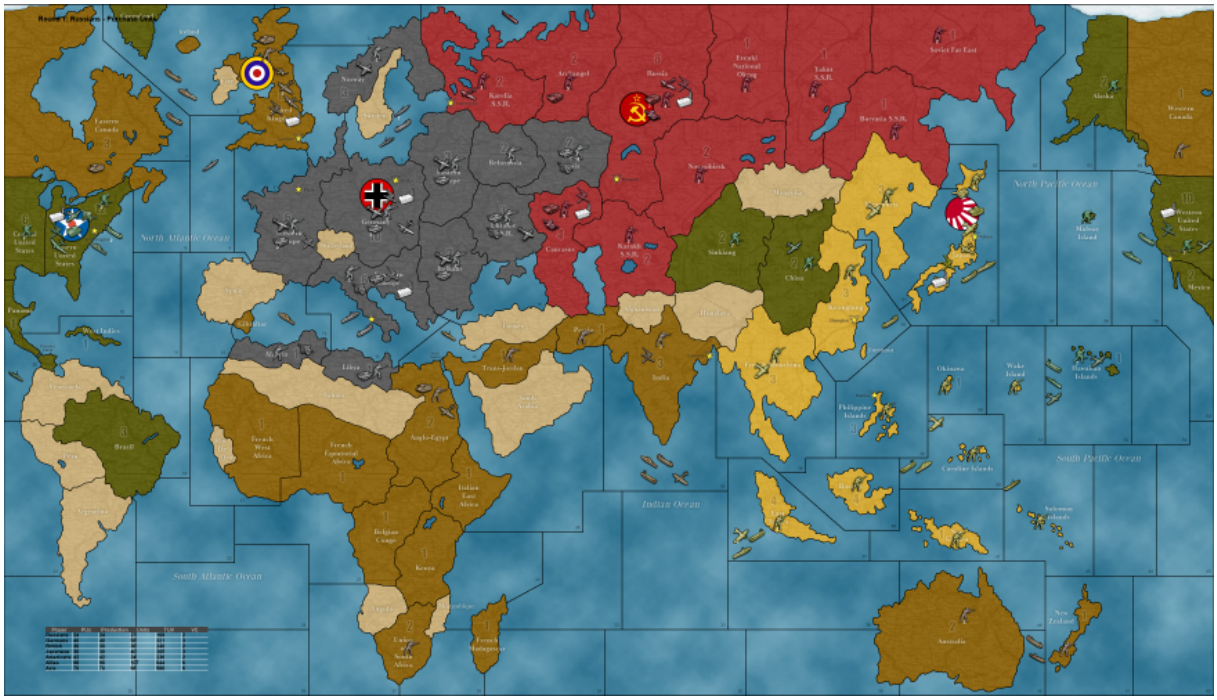

Figure 4.2: TripleA standard map

Each of the nations in TripleA has a capital territory. If an enemy player captures one of these territories, there are drastic consequences, such as losing the capacity to produce new units, and conquering an enemy capital is even a win condition in some games.

4.3 Units

As most games on TripleA are grand strategy games, a unit does not represent a single infantry or tank, but instead represents an entire army that is based around infantry or tanks. By general rule, each unit's actions are limited to movement during the unit movement phases. If a unit moves to an enemy territory while in the combat phase, then that unit also attacks the enemy territory. There are some special units, such as factories, which behave differently.

Each TripleA unit has certain properties, which are expressed as a set of numbers. These properties are:

- Attack - Firepower when a unit is attacking. The unit gets one hit by rolling that number or less on a 6-sided die.
- Defense - Firepower when a unit is defending. The unit gets one hit by rolling that number or less on a 6- sided die.
- Movement - The number of map territories that the unit can move each turn. A unit with a Movement of 1 can move to one adjacent area, and so forth.
- Cost (PUs) - Cost is how many production units (PUs) must be spent to produce one of that unit.
- Hit-Points - How many hits this unit must suffer before it dies. Almost all units in TripleA have only 1 hitpoint
- In some cases, special properties.

Below some of the unit types featured in the game are described, especially the ones relevant to our practical setting.

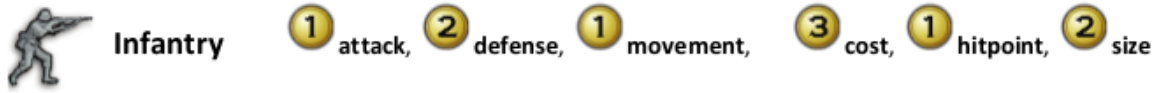

Figure 4.3: TripleA infantry

Special Abilities: Infantry may receive support. This means when they are paired with Artillery units on a 1-to-1 basis, the infantry units will receive +1 attack power.

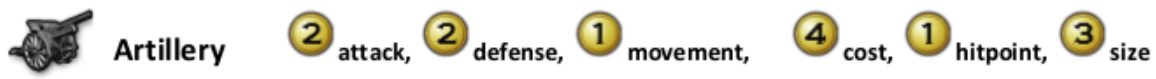

Figure 4.4: TripleA artillery

Special Abilities: Artillery may give support. So as an example, if you had 3 infantry and 2 artillery, then 2 of the infantry would receive support and have their attacks increased to 2, while the 3rd infantry would stay at 1 attack. The support is not cumulative, so 1 infantry with 2 artillery would only receive 1 support.

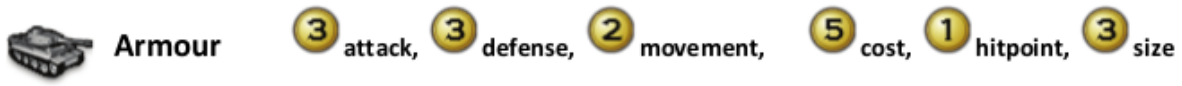

Figure 4.5: TripleA tank/armour

Special Abilities: Armour has the ability to “Blitz” through empty enemy territories. This means that while normally units must stop in the first enemy territory they enter, armour has the ability to keep moving if the enemy territory is empty of enemy units. The blitzing unit conquers this territory, and then can continue moving, going deeper into enemy territory or even returning to friendly territory.

Figure 4.6: TripleA factory

Special Abilities: Factories allow their owner to produce units in the territory the factory is located in. The Factory will allow production of any number of units, up to the Production value of the territory it is in. Having multiple factories in a territory does not increase this limit. Factories do not participate in combat, and if an attacker conquers the territory they will automatically capture any factories in the territory.

4.4 Sequence of play

Each game features two or more players, and the game is sequenced into rounds. During every round each player has a turn, which is composed of steps. The standard sequence of steps for each player turn is:

- Purchase - The player purchases units using his Production values
- Combat Move - The player moves his units offensively, allowing him to attack enemy territories
- Battle (resolving combat) - Pending battles from the combat move phase are resolved
- Non-Combat Move - The player moves his units with movement left after the combat move phase. The player isn’t allowed to move offensively and attack enemy territories

- Placement - The player places the units acquired during the purchase phase.
- End of Turn

A player may choose to do nothing in certain phases, and, depending on the circumstances, some phases may not occur. Since the game works as an engine more than a game itself, and features lots of different maps, it's hard to get a good estimate on the complexity without considering a single map's setting. The map used in the experimental setup is described in the results chapter, as well as the complexity analysis of the setting in question. In this setting some elements of the game were abstracted, such as the placement of units, focusing the algorithms on the movement of units and combat.

Normally "Victory" is determined by one player surrendering the game once they believe it is impossible for them to win. However, most maps include default 'victory conditions', which the players can play too if no surrender is forthcoming. Normally victory is determined by controlling a certain number of strategically important territories, called "Victory Cities".

4.4.1 Combat

When the player moves into an enemy territory with defending units during the combat movement phase, it results in a battle between both set of units. Combat is resolved in the following way:

- Dice are rolled for attacking units.
- Defender selects casualties. If all defending units die, then automatically all defending units are selected.
- Dice are rolled for defending units.
- Attacker selects casualties.
- Selected casualties are removed for both attacker and defender.

4.5 Game characterization

During gameplay, the entire map and placed units are visible to all players at all times, and there is no information about a player that is hidden from other players, the game is fully-observable.

As described in the sequence of play section, players take multiple actions in turns, and a player knows the previous moves taken by other players when he is taking his actions, so the game is also sequential and multi-action.

In the scope from which we approach the game, focusing on combat, the outcome of most actions is known, but some actions are stochastic. Actions like movement and deployment of units have clear

outcomes with no uncertainty, the unit simply moves to its assigned position. The combat outcomes are not certain, however. As explained in the sequence of play section, battles are decided using dice rolls to calculate successful and failed hits, resulting in an outcome that can be predicted to be in a given window, but still not certain.

4.6 Existing AI in the game

When starting a game, one can pick between controlling each player manually, or alternatively, there are three AI choices available, easy, fast and hard AI. Each AI has a different performance level, with easy AI being the weakest and fast AI being a weaker and faster version of the hard AI, which is the best performing alternative.

4.6.1 Easy Ai

Easy AI is the weakest AI option in the game. In the movement phase, this AI has a rigid set of hard coded rules which decide how to move it's units. One example is moving a single unit to each neighboring territory that is undefended.

In the combat phase, it prioritizes enemy territories it can just walk into, and tries to take those, and then moves into defended enemy territories it can reasonably expect to take. To calculate differences in strength between defenders and attackers, it uses a simple formula, and attacks with the minimum amount of units it considers necessary to take said territory.

In the non combat movement phase, this AI moves its units towards the closest enemy capitol, or toward the nearest enemy territory, if the first option is not available. However, if its own capital is in danger it doesn't move aggressively.

4.6.2 Fast and Hard AI

The Fast and Hard AIs follow a number of iterative steps in order to decide which actions to take in the movement phase. These AIs use calculators to estimate the total unit value of taking actions. The difference between the two is that the fast AI uses a simpler, faster calculator, while the hard AI uses a more costly and more effective calculator. The Fast AI's calculator predicts the outcome of the battle directly, based on the strength of both the defending and the attacking units. The Hard AI's battle calculator simulates the battle a number of times, using the game's battle mechanics, and returns the average result across all executions as the battle result.

The steps these AIs take in order to determine the combat movement are as following:

- Determine whether capital is threatened and I should be in a defensive stance;

- Find the maximum number of units that can attack each territory and max enemy defenders;
- Remove territories that aren't worth attacking and prioritize the remaining ones;
- Determine which territories to attack;
- Determine which territories can be held and remove any that aren't worth attacking;
- Determine how many units to attack each territory with;
- Determine if capital can be held if I still own it;

5

Implementation

Contents

The algorithms proposed in [17] are state-of-the-art for multi-action adversarial games, which, in a way, our test bed tripleA can be seen as. The main difference between tripleA and the testbed in [17] is the considerably higher branching factor per turn of tripleA, since grand strategy games tend to deal with a considerable amount of game units, as well as the stochasticity of the game. If we remove those factors from play, then tripleA can also be seen as a turn based multi-action adversarial game, similar to Hero Academy.

We adapted our problem, using action pruning strategies, and implemented the algorithms present in [17], the Non-Exploring MCTS (NE-MCTS) algorithm, the Bridge Burning MCTS (BB-MCTS) algorithm, and the Online Evolutionary Planning (OEP) algorithm, which are described in the related work chapter.

In this chapter we discuss how the implementations of the different agents were developed. We describe how and which algorithms were implemented, as well as the pruning strategies used, and also talk about how the forward model was developed and adjusted to our case.

5.1 Development challenges

The implementation of the algorithms using TripleA as a test bed proved to be extremely troublesome, and raised a lot of challenges, especially due to the effort required in order to understand the underlying game code, and a lot of unexpected bugs appeared which proved troublesome to fix.

The first decision to be made was how to approach the development of the forward model, whether to implement a new forward model from scratch, or re-use some of the game's data structures and use the game itself as the forward model. The forward model works as the middle-man between the algorithms and the game. The algorithms implemented need to obtain possible actions for each game state, as well as obtain states resulting from applying actions' effects, and simulate playouts. The forward model provides these features to the algorithms, being used extensively by them and a key element in their performance. It was ultimately decided that the best option was reusing as most of the game's code as possible. Since the game's Hard AI simulated 1 turn ahead in order to decide which units to purchase, some of the code already implemented would be similar, and this seemed like the better alternative.

In hindsight, developing a new forward model from scratch seems like a better option now, as the source-code raised a lot of bugs and took a lot of effort and digging into in order to "get right". Additionally, by creating our own forward model from scratch, we might have been able to get a more optimized version which would be able to run more iterations per time frame. Having said this, the forward model developed works as intended, and is able to perform efficiently enough for the algorithms and agent to return a quality response with small computational budget, which is the goal of the thesis.

5.2 Forward Model

The forward model was implemented making use of most of the game's data structures, adjusting and optimizing the source code where we could, and adding new code where needed.

5.2.1 Game state

A copy of the Gamedata data structure, which holds all game information, is created. Each world model object holds a copy of this structure, which contains all the game data corresponding to the current game state. Code was implemented to efficiently clone Gamedata objects, which allows for the efficient exploration of the action space with tree search and generation of new nodes.

This data structure allows access to all elements of the game and since it uses the game's original code, a lot of the existing methods present in the source code can be re-used. This data structure contains following elements:

- Game map (GameMap)- This data structure represents the game map. It allows for the creation of routes between territories and can be used to check which territories neighbor which. It contains the following elements:
  - Connections between different territories.
  - List of all the game's territories (Territory). Each territory contains the following data:
    - Owner (GamePlayer).
    - Units (List<Unit>) that are placed in this territory.
    - Additional descriptive information, such as if the territory is a capital or not.
- Players (List<GamePlayer>).
- Units (List<Unit>) - The list of units in the game. Unit data structure contains the following elements:
  - Owner (GamePlayer).
  - Descriptive information, such as it's stats, whether unit can blitz, etc.
  - Movement left in current round.
- Game sequence - Contains information about the current step and round. Used to tell which player should move. Contains game step elements, which represent a single step in the current round. These game step elements contain delegate objects which are used to interact with the GameData object, and make the changes necessary for the game to ”progress”.

To abstract the algorithms from the game's inner logic, the forward model handles the underlying behaviour of the game, such as resetting unit's movement, changing owners, handling battles, etc. The forward model is also responsible for advancing the game state into game steps where the algorithms operate.

5.2.2 State expansion - Battles

One important factor of developing the forward model was how to decide the result of battles between units. Battles in the game are decided by dice rolls, which makes them stochastic, and thus, difficult to handle in the forward model. When deciding how to deal with the stochasticity of the game's battles in the forward model and algorithms, there seemed to be two possible solutions.

The first alternative was keeping battle's results stochastic, working similarly to the real game, and running playouts multiple times for each state, instead of only once, attributing the average playout value to the node. This way, the aggregate result will give a better estimate of the real quality of a given node.

When running playouts for algorithms with no exploration in this phase, this approach seems like a good alternative, as only the outcome of battle will vary, and not the actions chosen for each game state themselves. This leads to a clear estimate of how different outcomes of the occurring battles will change the final state's quality. However, when using this approach with a non deterministic playout heuristic, the randomness of the actions chosen, together with the stochastic outcome of battles, makes the outcome too random, and makes it harder to get a clear estimate with a small number of playout executions. When using this approach, multiple playout executions have to be ran.

The main problem with this approach is the additional computational budget that the algorithms consume in order to run multiple playouts per tree node.

The second approach was running combat deterministically, always getting the most likely outcome out of battles. This can be done by using the game's Fast AI combat calculator to get an estimate of the result for each battle. This solution is computationally cheaper, and seems like a better alternative to deal with non deterministic playout policies. The main problem of this solution is that it cannot assess the value of worst and best case scenarios by getting only the more likely outcome in battles. This is a small trade-off for the benefits of this alternative when compared to the first one. With this approach, the quality of a node can be assessed by running a single playout, which saves computational budget for the algorithms to run more iterations. Additionally, since not all algorithms implemented have deterministic playout policies, this approach is also better suited for those algorithms.

Since the computational budget for running the algorithms is small, we opted for the latter option, so for our model, the outcome of battles is not stochastic, and always returns the most likely outcome. In the experimental setup, however, battles still work as intended, and have stochastic outcomes dictated by dice rolls.

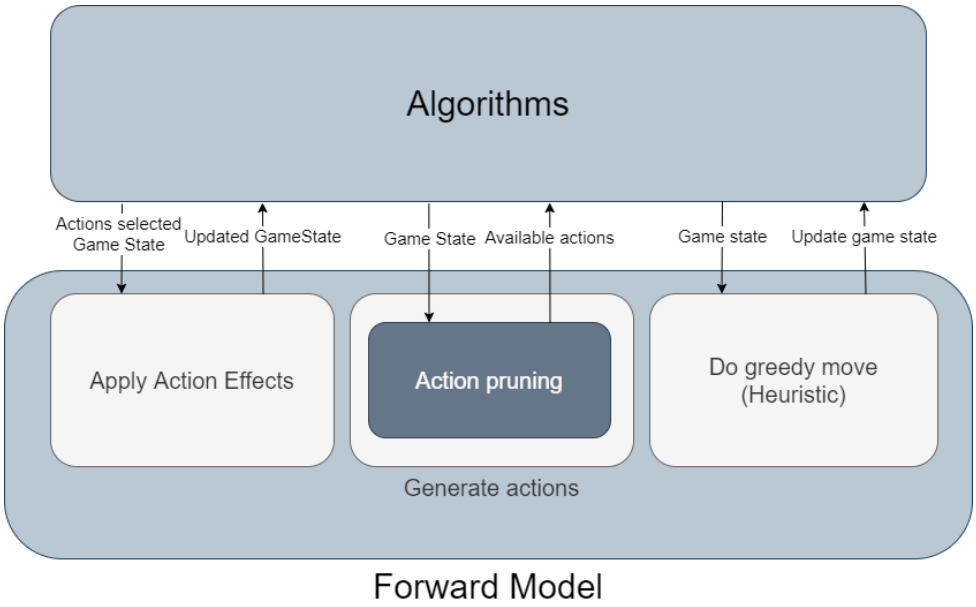

Figure 5.1: How algorithms interact with the forward model

Fig 5.1 is a high level illustration of how algorithms interact with the forward model. The forward model is used to apply action's effect to game states, to generate possible actions for states and also to generate the best deterministic action sequence for the playout phase. In order to optimize the execution of algorithms, both the pruning of actions and the action selection for deterministic playouts is handled by the forward model directly. The pruning of actions as well as the playout heuristic are described next.

5.3 Pruning strategies

During the development of the thesis, and as stated in the related work section, different algorithms and MCTS variations were reviewed, as well as case specific and practical implementation in the industry of the MCTS based AI in Total War: Attila [13].

One of the main differences between the test beds of the different algorithms reviewed and our case is the extremely high branching factor and the number of possible game states in our scenario. Without action pruning, the number of possible actions is too large for the algorithms to be able to effectively explore search space in a realistic time frame, not even coming close to exploring an entire turn in the search phase, for tree search algorithms

By making use of domain-specific knowledge for TripleA, as well as some of the techniques introduced in [13], we will try to adapt the search problem in order to prune the action space as heavily as possible, thus making the game's setting one that can be effective explored by the algorithms introduced in [17], and implement them. The way in which we developed our forward model also abstracts the

game's stochasticity from the algorithm's perspective.

Next we describe in detail the pruning strategies and decisions made.

5.3.1 Action generation and pruning

The generation of actions for game states was developed with the purpose of reducing complexity as well as we could, eliminating redundant game states and ineffective actions, based on strategies employed in [13].

We consider an action to be ineffective if it leads to an instantaneous disadvantage for the player, such as attacking an enemy territory with a small number of units relative to the defender, resulting in a great loss of units. This happens because units tend to have better attacking stats and deal more damage when defending. Because of this, when attacking, its best to overpower enemy units and take them out as soon as possible by having strength in numbers, since the dice rolls favor the defending units. In order to improve the quality of the algorithm, action sequences leading to these types of results will be considered illegal and effectively pruned. This is similar to the pruning of unwinnable battles used in [13].

We consider redundant game states to be game states that are similar to others that can be obtained via different action sequences. Even though the action sequence is different, the resulting game state is equal, and there is no interest in exploring more than one of them. Pruning these states is also a strategy used in [13], where sub-trees leading to identical world states were pruned.

When deciding how to target the generation of possible actions for the algorithms, there were two possible approaches.

Unit action generation

The first approach is to generate actions for each unit iteratively and generate a possible action for each territory that the unit can move to. Using this approach, units will be ordered and the tree search will generate possible actions for each node, based on a single unit which will change as the tree progresses. For example, on the root node, actions correspond to the first unit. On nodes with depth 2, actions correspond to the second unit, and so on. This ordering schema is similar to ones employed in [13], and can effectively reduce the state space by removing some duplicate states.

However, this approach leads to some problems. Firstly, even though the unit ordering scheme will stop some redundant similar states from forming in the tree, there will still occur redundant game states, even from different action sequences. This happens because if two units of similar type are in the same territory, then the game state resulting from attacking another territory with either one of them, is similar. This action generation and ordering scheme makes it difficult to handle this specific type of redundancy.

Secondly, this approach makes it harder to prune ineffective action sequences. When deciding which action sequences lead to ”bad” attacks when using this action generation schema, the whole action sequence has to be considered. This is so because we can only get the total number of units attacking a territory after we have generated actions for every unit in question. It is hard to classify an action as ineffective when considering units individually, as we only get the resulting attacking units per enemy territory in the end of action sequence.

This approach has a small branching factor per action (each territory the unit can move to), but leads to a great number of actions per turn, and since the whole action sequence has to be considered, some areas of the tree would be heavily explored before being classified as ineffective, leading to wasted potential.

Territory action generation

The second approach to action generation is to generate actions based on each territory. Like in the previous approach, actions would be generated sequentially, only with this method, they would be based on territories and not units. In this case, multiple actions are generated for each territory, and possible actions for a territory correspond to different sets of units that can move to said territory.

With this approach, the pruning of ineffective actions becomes much more manageable. When generating an action we can tell immediately if that action leads to an ineffective action sequence or not. By calculating the strength of defending units, it is possible to assess if an attack will fail (because we have access to all the units that will move to the territory), and not generate actions that lead to a combat loss.

Additionally, since all units moving to a territory are considered simultaneously, it becomes possible to further cut down on redundant game states, as we can choose not to generate actions with similar sets of units moving to the same territory.

For these reasons, it was decided to follow the second approach.

Combat movement phase

Action generation for the combat movement phase generates possible actions for one territory at a time. Given a state and a territory, it returns a set of different actions corresponding to attacking the territory with different sets of units. It works in the following way:

- Territory t corresponds to the target territory that was selected for movement/attack in the current node.
- All allied units that can attack this territory are selected, and subsequently the firepower of these units, as well as the firepower of the enemy units that defend territory t, is calculated. Based

Algorithm 4 Generate actions combat movement

```
1: function GENERATEACTIONS(Territory t)
2:     actions = {}
3:     units = allUnitsThatCanAttack(t)
4:     if units.canSuccessfullyAttack(t) then
5:         attackCombinations = attackCombinations(units, t)
6:         for unitSet:attackCombinations do
7:             actions.add(new Action(t, unitSet))
8:     actions.add(new Action(t, {}))
9:     return actions
10:
11: function ATTACKCOMBINATIONS(units, Territory t)
12:     unitsToAttack={}
13:     unitSets={}
14:     for Unit u:units do
15:         unitsToAttack.add(u)
16:         if unitsToAttack.canSuccfessfullyAttack(t) then
17:             unitSets.add(unitsToAttack.copy())
18:     return unitSets
```

on the estimated firepower, if the firepower of the defending units is greater than the firepower of attacking units, then the territory is considered unable to be conquered, and a single action is generated for t, with a list of empty units. This represents skipping the territory since it can't be attacked successfully.

- If the territory is able to be attacked successfully, actions are generated for t. Each action has a different subset of all the units that can move to t, which can vary in size, with the condition that each subset must also be able to successfully attack the territory.

  By creating multiple actions with different sets of units varying in size, the search space explores multiple options. If a territory is attacked with few units, then it leaves some units open to attack other territories. If the territory is attacked by an overwhelming number of units, then the attack will be efficient and cost a minimal amount of casualties, but there may be no additional units to attack other territories. Generating actions in this way ensures that the tree search can explore the effects of these choices and have alternatives.

  Additionally, an action with an empty set of units is also generated, corresponding to ignoring the territory, which can also be an optimal decision, even if the territory is able to be conquered.

When experimenting with the generation of possible sets to attack a territory, we also experimented with generating random subsets of the list of total units, instead of our current approach described in the ATTACKCOMBINATIONS() function in 4. This change didn't increase the quality of the algorithms, so we maintained our original, simpler alternative.

Non combat movement phase

During this phase, there is no obvious pruning strategy that can be implemented. One thing that was concluded from playing the game was that usually it was a good idea to move all units in a territory together. Unless units from one territory must be split in order to defend two or more adjacent territories, this is the case. Because of this, and in order to reduce the branching factor of actions in this phase, actions are generated in the following way.

- Territory t corresponds to the to the territory which actions must be generated for in the current node.
- For each allied territory t2, a set S with all units placed on t2 that can move to t, is generated. This set is then split into two sets, each containing half of the units in S. This is done so because sometimes it might be a good idea to split forces in order to defend more than one territory. By doing this, forces in a territory are split in half and considered independently (both halves can still move to the same territory).
- After this phase, all sets created are placed in a list L, and a new action is created for each permutation of L from size 0 to size(L) (represents each possible combination of units that can be moved to t, based on the sets created).

Algorithm 5 Generate Actions non combat movement

```
1:  function GENERATEACTIONS(Territory t)
2:      actions = {}
3:      unitTerritorySets = UnitSets(t)
4:      permutations = Permutations(unitTerritorySets)
5:      for Permutation p:permutations do
6:          actions.add(new Action(t,p))
7:      return actions
8:
9:  function UNITSETS(Territory t)
10:     sets = {}
11:     for Territory t2: PlayerOwnedTerritories do
12:         unitsToMove = {}
13:         units=t.units()
14:         for Unit u:units do
15:             if u.canMoveTo(t2) then
16:                 unitsToMove.add(u)
17:         sets.add(unitsToMove.subList(0, unitsToMove.size()/2))          ▷ First half of the territory list
18:         sets.add(unitsToMove.subList(unitsToMove.size()/2, unitsToMove.size()))     ▷ Second half of the territory list
19:     return sets
```

Normally, the order in which actions are generated for different territories would be irrelevant. However, it was noted that sometimes, when the execution time was not long enough to check all tree nodes

across all depths, territories for which actions were generated first were explored more extensively, and obtained better quality actions.

For this reason, an ordering scheme was implemented in this phase. During this phase, territories are ordered based on proximity to enemy territories, with the exception of capitols, which are always first in the list. The logic behind this ordering schema is that capitols and territories bordering enemy lands are more sensitive and prone to enemy attacks, and thus deserve more attention.

This is not a problem during the combat movement phase, and there's no need to order territories during said phase as only territories that can be attacked have corresponding actions, and the consequent number of actions in the sequence is smaller than during the non combat movement phase.

5.4 Playout phase heuristic

The heuristic function used to guide actions during the playout phase of the algorithms is based on the weak AI implementation on the game's source code. The playouts used in the implemented algorithm's description are deterministic, which means the same actions are always selected for each state. Because of this, after testing the performance of the game's Weak AI and noticing that it was quite fast, we saw an opportunity to use it as the basis for our own playout heuristic. This method does not attribute a value for each different possible action, and instead generates the actions to take on each state directly. It can be seen as an heuristic that attributes value 1 to actions that should be taken, and value 0 to actions that shouldn't be taken. Since playouts are deterministic, there's no point in considering and attributing values to actions that won't be taken either way.

Using this heuristic increases the overhead of running the algorithms, but increases the quality of the rewards obtained for states after the playout phase. Since most of the algorithms implemented use deterministic playouts and have no exploration during this phase, we see this as a positive trade-off. It works in the following way during different phases of movement:

Combat movement phase heuristic

A list containing all enemy territories is created and is ordered according to priority of capture. The newly ordered list of territories is iterated upon, following the steps described below. This logic gives priority to territories placed first on the list.

- Where it is possible, a single unit is placed on undefended enemy territories;
- For each enemy territory in the list, the strength of defending units is calculated, as well as the strength of possible attackers that can move to the target territory. Based on the perceived strength of both sets of units, if the territory is unable to be conquered successfully, then it is ignored.

Otherwise, the smaller number of attacking units that is expected to conquer the territory with ease is moved there, freeing other units to attack or defend other territories.

Non combat movement phase heuristic

During the non combat movement phase, the goal is to move units closer to nearby enemy capitols. The algorithm iterates over allied territories, and executes the following steps:

- If the current territory is a capitol, and it is at risk of being lost, then it does move units that are placed there, and skips this territory instead
- All units that can still move and are placed in the current territory are selected
- If possible, move units towards the closer enemy capitol Otherwise, move units in the direction of the closest enemy territory

5.5 State reward function

When running playouts, one important thing to note is that sometimes a terminal state might take too long or even be impossible to reach. Taking this into account, playouts are ran for a maximum depth of 10 rounds, simulating turns until the current state is terminal, or until that depth is reached. Because of this, it is important that our reward function is able to properly evaluate the quality of both terminal and non-terminal states, since we might not always obtain a terminal state by running playouts. When the resulting state is obtained, reward function 6 is used.

Algorithm 6 Reward function

```
1:  function REWARD(State s, int depth)
2:      create root node v_o with state s_0
3:      if s.isTerminal() then
4:          if s.isWin() then
5:              return ( 0.6 + 0.4 × ( 10 - depth ) / 10 )
6:          else
7:              return ( -0.6 - 0.4 × ( 10 - depth ) / 10 )
8:      else
9:          alliedUnitN = state.getAlliedUnits().size()
10:         enemyUnitN = state.getEnemyUnits().size()
11:         totalSize = alliedUnitN + enemyUnitN
12:         return ( alliedUnitN - enemyUnitN ) / ( totalSize × 2 )
```

Terminal state reward function

The reward function works differently based on if the final state is terminal or not. If the final state terminal, then the reward function gives a positive value for states where the player is the winner, and a negative value for states where the player lost, with respective intervals of [0.6,1] and [-1,-0.6]. Depending on how long it took until the current state resulted in a win or loss, the reward function changes the value attributed to the state, valuing wins that happened faster more highly, and similarly, giving a higher value to losses that happened in later rounds than to instantaneous losses.

Non-Terminal state reward function

If the final state reached during the playout phase is not terminal however, then the reward is calculated based on the number of units that both players have. The resulting value is contained in the interval [-0.5,0.5], based on whether the player has more, or less units than the opposing player. This reward function cedes more impact to playouts where a terminal state is obtained, meaning that a winning playout will always be valued higher than a non terminal one where the player has a unit advantage, and a losing one will always be valued lower than a non terminal one where the player is at a unit disadvantage. The reason behind this value function for non terminal states is the simple logic that if a player has more units than the adversary, then he is in a advantageous position.

5.6 OEP

Generation of actions for OEP works similarly to the other implemented algorithms, being generated based on territories. This means that when adapted to TripleA, an OEP genome will be composed by a set of actions corresponding to a different territory each, and the units that will be moved there. The population size was set to 50, with a survival rate of 0.5 and a mutation probability of 0,1.

5.6.1 Genome creation

When creating a new genome, actions should be generated randomly. To achieve this, the list of possible move territories is ordered randomly, and actions are generated for each of the territories sequentially. For each territory, a random action from the list of possible actions (generated by the forward model) for that territory is added to the action sequence. It is important to randomize the order of the territories because territories for which actions are generated first might have access to a higher number of units. This happens because the territories added latter cannot use units that are already contained within actions in the sequence.

After generating the action sequences for each genome, the state resulting from applying the actions in the genome to be evaluated to the current game state, is obtained from the forward model.

5.6.2 Evaluation

In order to evaluate the quality of a genome, an evaluation function is used on the state resulting from applying the actions in the genome. This evaluation function works similarly to a deterministic playout in the MCTS algorithm. Playouts are guided using the playout phase heuristic described in this section, and ran for a maximum depth of 10 rounds or until a terminal state is reached. After this, the reward function described is used to calculate the value of the state.

5.6.3 Procreate

Algorithm 7 Procreate

```
1: function PROCREATE(genes)
2:     newGenes = {}
3:     for gene g: genes do
4:         gene g2 = genes.random()
5:         newGenes.add(g.procreate(g2))
6:     genes.addAll(newGenes)
7:     return genes
   =0
```

Offspring between two genomes is generated in the following way:

- Randomly order actions corresponding to each territory.
- Iterate over the territories, randomly selecting one of the two genome's action corresponding to the current territory, and adding that action to the new genome.
- When a selected action leads to an illegal sequence of actions (Some of the units corresponding to the new action are already used), then a new random action corresponding to the same territory is generated instead, using units from the pool of available units.

5.6.4 Mutation

Each newly created genome is replaced based on probability E, in our case 0.1, with a randomly generated action (obtained from the forward model) corresponding to the same territory.

Fig 5.2 illustrates how the OEP algorithm works on a high level, and how it interacts with our forward model.

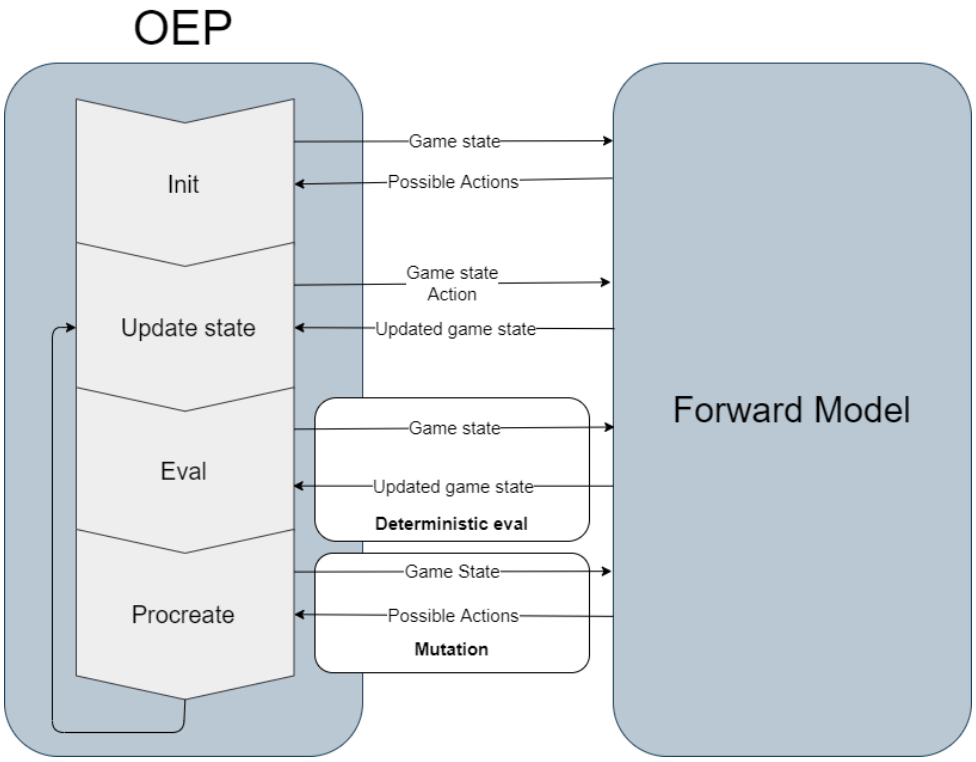

Figure 5.2: OEP and forward model interactions

5.7 BB-MCTS and NE-MCTS

The Bridge Burning and the Non Exploring MCTS variants are implemented as described in the original paper [17]. When obtaining possible actions for each state, actions are obtained from the forward model, which employs the pruning strategies directly. Similarly, the forward model handles the deterministic playout actions. The reward function used in playouts is the one mentioned.

The Bridge Burning algorithm does not have deterministic playouts, containing exploration in this phase. Because of this, and due to our action generation and playout schema, the exploring factor for playouts in this algorithm works in the following way: For each move step, with probability E, instead of the playout action being determined deterministically by our heuristic, the action sequence in the current turn is performed randomly instead. These random actions are still selected from a pool of actions generated using our action pruning strategies.

Fig 5.3 is a high level illustration of how the Non Exploring and the Bridge Burning MCTS algorithms interact with the forward model.

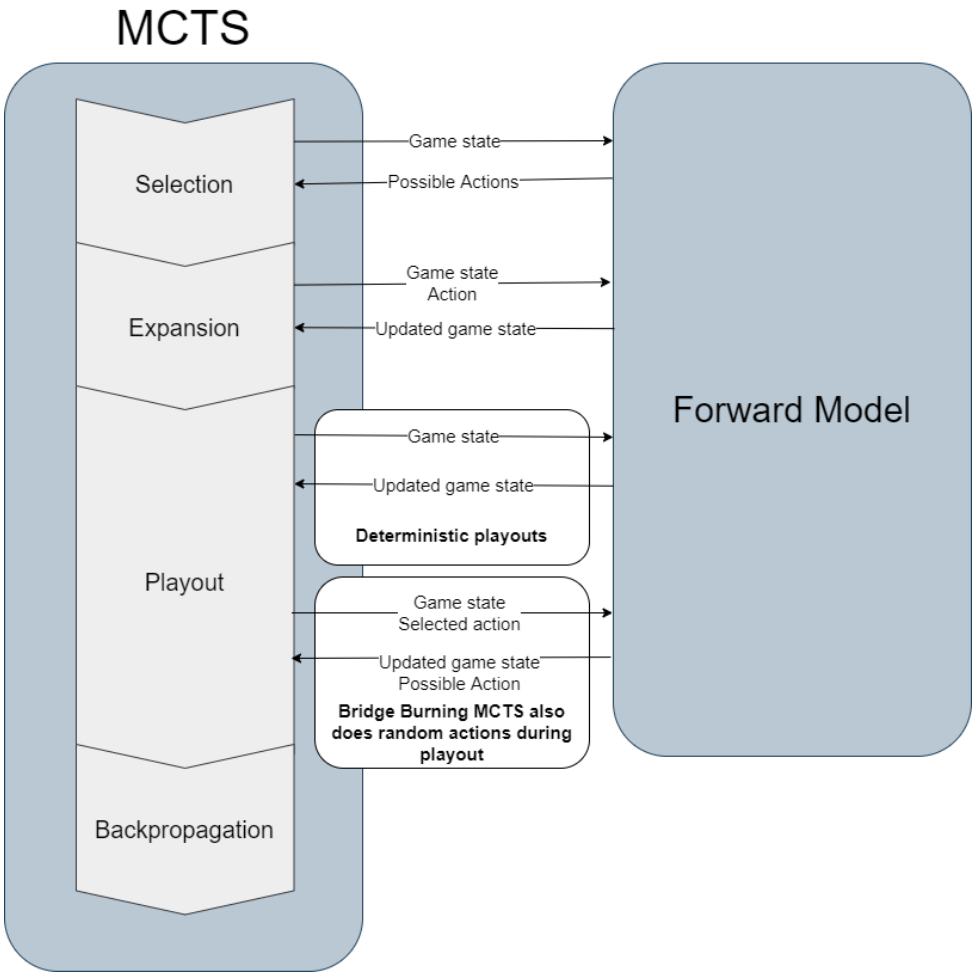

Figure 5.3: BB-MCTS/NE-MCTS and forward model interactions

6

Results

Contents

In this chapter we present the performance results obtained for the different agents implemented and described in the implementation setting, as well as against the game's existing AI. An analysis of the performance of the forward model as well as of the pruning strategy is also performed.

6.1 Experimental setup

6.1.1 Practical setting

To simplify the development of the forward model, as well as the algorithms, the implemented agent's area of effect was limited to the unit movement, which includes the Combat and Non Combat movement phases, ignoring the placement and purchasing of units. The air and amphibious combat units were also removed from consideration. Additionally, the game will be played on a map featuring two players in order to get the win rates between different agents. This decision simplifies the development of the forward model for the game significantly, and eases the development and implementation of the algorithms and agents as well. While this abstraction removes some complexity from the game, the resulting complexity is still in alignment with the target problem, being characteristic of most grand strategy games, and significantly higher than more common multi-action adversarial turn based games.

The algorithms were implemented and tested on a custom map, designed to cater to our practical setting. The map features 14 territories, and each player starts the game with 28 units, 19 infantry, 6 artillery and 3 tanks. Each territory had its production value set to 0, which means that no player will be able to produce units. The implemented algorithm's decision making is thus limited to the combat and non combat movement phases.

We compare the performance of each agent against each other, as well as against the game's most complex AI option, the Hard AI. We ran tests for different computational time budgets of 1 and 5 seconds per move and for each different match up 200 games were ran, 100 for each agent starting on different sides. Even though the vanilla game doesn't have draws, games that lasted for more than 35 rounds were counted as a draw, as when the game got to this phase, it usually meant that neither player would be able to come out victorious. These test were ran on 16 GB of ram and on an AMD Ryzen 5 3600X 6-core processor with 3.80Ghz.

We decided to run tests for 1 second because this execution time resembles that of the game's Hard AI, against which we will also match our agents. Additionally, we also did performance tests for an execution time of 5 seconds because we wanted to compare how the performance of these algorithms would change in function of the execution time. This computational budget of 5 seconds allowed the agents to perform deeper searches of the search space, while still causing the games to finish in a reasonable time for the gathering of data. We did not match our agents against the game's Hard AI in the 5 seconds of execution setting because there's not as much relevance (because this execution time

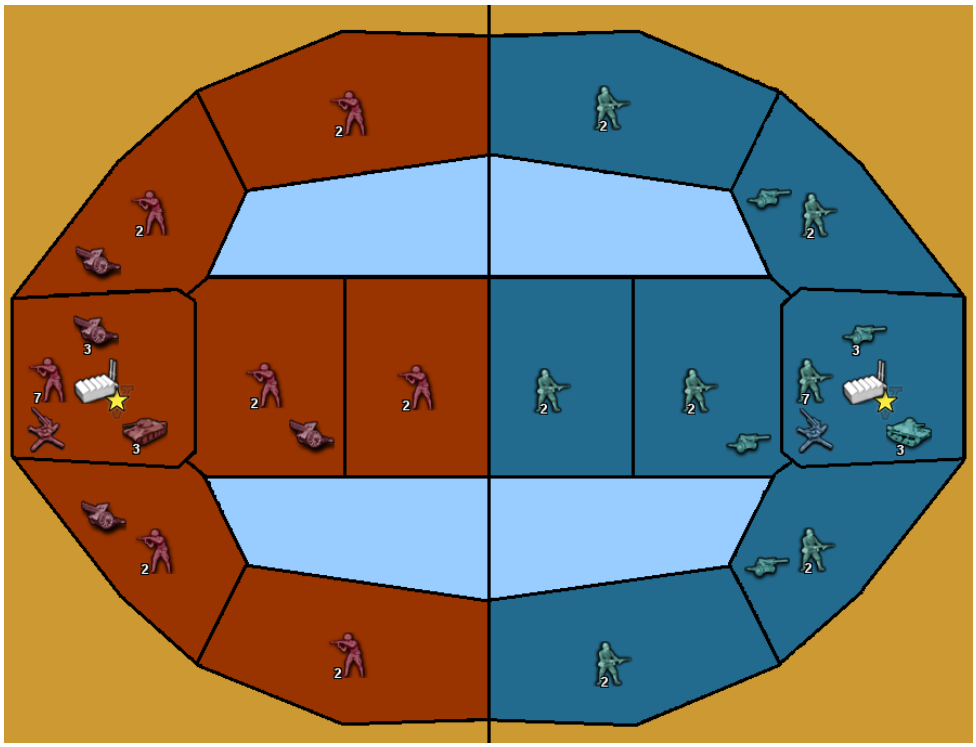

Figure 6.1: The practical setting used to run games between different agents

far exceeds that of the Hard AI).

One concern of using this setting to test the agents against the game's hard AI was that since no territory has production value, the hard AI would try to gain advantage by conquering territories and gain no real advantage because they would provide no production value, and therefore, no units in future turns. However, the game's hard AI takes this factor into consideration when deciding how to move and which territories to conquer. It uses the territory's production value as a way to measure the value in conquering it, taking the difficulty to conquer it in consideration as well. Because of this, even though one of the hard AI's aspects of play is irrelevant, the setting does not result in a direct disadvantage for the AI, as it "knows" that the territories have no real value in being held or conquered.

6.1.2 Practical setting complexity analysis

In this setting each territory has between 2 and 4 neighboring territories. The units used are tanks, artillery and infantry units. Each infantry and artillery unit can move to an adjacent territory or remain in its current location, meaning that it can take between 3 and 5 actions per turn, depending on its current location. Tank units have the blitzing ability, which allows them to move to territories located two tiles from their current location. A tank unit has on average between 5 to 8 actions per turn. The practical setting used features 19 infantry units, 6 artillery, and 3 tanks.

It is quite difficult to estimate the branching factor per turn precisely, but using each unit's average possible actions, the lower bound for initial states can be estimated to be $3^{infantryN+artilleryN} \times 5^{tankN} = 3^{19+6} \times 5^3 = 1.06 \times 10^{14}$. the upper bound is much greater, being $5^{infantryN+artilleryN} \times 8^{tankN} = 5^{19+6} \times 8^3 = 1.53 \times 10^{20}$ This calculation assumes that actions are handled individually for each unit in the game. However, as described in the previous section, we opted to generate actions based on territories, and not units, which changes not only the branching factor per action, but also the depth of the action sequence.

It is also hard to estimate the value for the branching factor over the game when using our action generation format, as we would have to count the number of possible sets of units that can move to each territory individually. To do so, we have to analyze each specific game state and see how many possible combinations of units can move to each territory. We will calculate the branching factor for the initial state's non combat movement phase, and also analyze how our action generation schema and pruning impacts the branching factor in this case.

Fig 6.2 features the practical setting's map with the red player's territories numbered. We can count how many units can move to each specific territory. For territories 1,5 and 7, there are 6 units that can move there (1 artillery, 2 infantry, 3 tanks), for territories 2, 4 and 6, there are 21 units that can move there (5 artillery, 13 infantry and 3 tanks) and for territory 3 there are 9 units that can move there (3 artillery and 6 infantry). Taking these numbers, and calculating the number of different possible sets of units that can be moved to each territory, for territories 1,5 and 7, there are $2^6 = 64$ combinations, for territories 2,4 and 6 there are $2^{21} = 2,1 \times 10^6$ combinations, and for territory 3 there are $2^9 = 512$ combinations. The average branching factor per action(territory) is thus $(2,1 \times 10^6 \times 3 + 64 \times 3 + 512)/7 = 9 \times 10^5$.

We will now do the same calculation for the action pruning schema implemented. Territories 1, 5 and 7 each have units that can have units move there from 2 different territories. Territories 2, 3, 4 and 6 have units that can move there from 3 different territories. In our non combat action generation, for each territory that can move units to another, 3 different sets of units are generated, one containing 0 units, one containing half the units, and another containing all the units there, and for each combination of these sets corresponding to different territories, an action is generated. Because of this, for territories 1, 5 and 7, $3 \times 3 \times 3 = 9$ actions are generated, and for territories 2, 3, 4 and 6, $3 \times 3\times = 27$ actions are generated. This results in a branching factor of 9 or 27 per action, depending on the territory in question, with the average being $(9 \times 3 + 27 \times 4)/7 = 19,3$.

Using the branching factor per action obtained for both when using and not using our pruning schema, we can calculate the ratio by which the branching factor per action was reduced when using pruning. In this specific case, our pruning strategy reduced the average branching factor per action by $(9 \times 10^5/19,3) = 4,66 \times 10^4$. One "double-edged sword" that can be identified by these calculations is the fact that the number of actions generated for each territory does not depend on the number of units. For

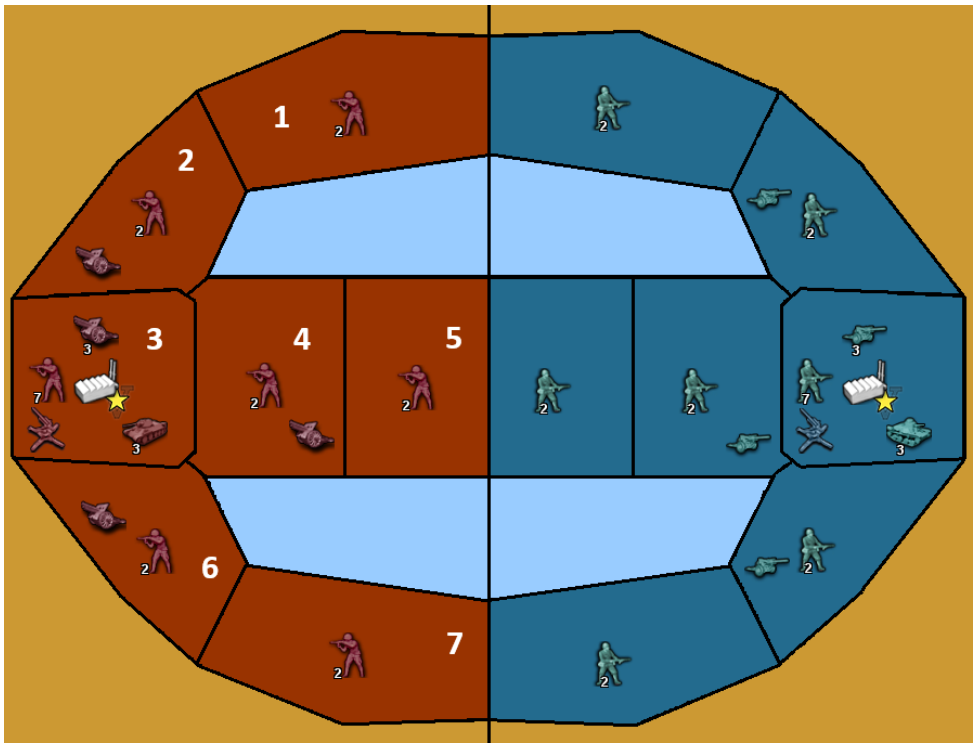

Figure 6.2: The practical setting with numbered maps

example, we can see that for territories 3, 4 and 6, there are $2,1 \times 10^6$ possible actions (high density of units), while for our action generation format only 27 will be considered (because units can only be moved there from 3 territories). This can be benefitial and increase the scalability of the algorithm, but can also be problematic, as a lot of optimal decisions might not be considered. This highlights an opportunity to change and further optimize this mechanic in future work.

6.2 Win rates and agent performance

Results

Table 6.1 presents the results obtained for each matchup between the agents OEP, Non exploring MCTS, Bridge burning MCTS and also TripleA's Hard AI. 200 games were played for each matchup, 100 with each agent on the starting side. The first line represents the win rate when counting a tie as 0.5 wins for each agent, and below is the percentage of wins, ties and losses, respectively, for each match up. We display the results in this way in order to facilitate the analysis, because we consider presenting the number of ties to be important, while the results in the first column give us a more explicit win rate. For tests with 1 second of execution time, the agents were also compared against the game's Hard AI,

as this execution time is similar to it's execution time. Table 6.2 presents the results of a similar test setting, but for an agent execution time of 5 seconds.

Table 6.1: Win rates of agent on the left most column vs the agents on the top row. 1 second of execution time. For each match up, the top row represents the win rate when counting a tie as 0.5 wins for each player, and the bottom row contains the number of win, ties and losses for that match up, respectively

	OEP	Non Exploring MCTS	Bridge Burning MCTS	Hard AI	Average
OEP		37.75% 32.5% / 10.5% / 57%	47.5% 42.5% / 10% / 47.5%	60.5% 49% / 23% / 28%	48.58%
Non Exploring MCTS	62.25% 57% / 10.5% / 32.5%		51% 48.5% / 5% / 46.5%	72,25% 61% 22.5% 16.5%	61.92%
Bridge Burning MCTS	52.5% 47.5% / 10% / 42.5%	49% 46.5% / 5% / 48.5%		72.75% 62% 21.5% 16.5%	58.1%
Hard AI	39.5% 28% / 23% / 49%	27,75% 16.5% 22.5% 61%	27,25% 16.5% 21.5% 62%		31.5%

In 6.1, between all implemented agents, the most noticeable difference is present in the Non Exploring MCTS vs OEP match up, with the Non exploring algorithm achieving a win rate of 62,5% and winning almost twice as many games as OEP. The Bridge burning algorithm does not show any clear advantage or disadvantage over any of the other agents, having a win rate of 49% against Non exploring MCTS and 51% against OEP. Taking these results into account, it can be noted that in this scenario, the best performing agent is the Non exploring variant, followed by Bridge Burning and lastly the OEP algorithm.

Table 6.2: Win rates of agent on the left most column vs the agents on the top row. 5 seconds of execution time. For each match up, the top row represents the win rate when counting a tie as 0.5 wins for each player, and the bottom row contains the number of win, ties and losses for that match up, respectively

	OEP	Non Exploring MCTS	Bridge Burning MCTS	Average
OEP		46.5% 43% / 7% / 50%	38.25% 33.5% / 9.5% / 57%	42.4%
Non Exploring MCTS	53.5% 50% / 7% / 43%		47% 44% / 6% / 50%	50.25%
Bridge Burning MCTS	61.75% 57% / 9.5% / 33.5%	52% 50% / 6% / 44%		56.88%

In 6.2, the results obtained for running agent for a longer period of time of 5 seconds, differ noticeably from the results in 6.1. By increasing the execution time, the perfomance of both the Bridge burning and the OEP algorithm increases when compared to the Non exploring MCTS. The win rate of Bridge

Burning over the other agents increases, especially against OEP, and while the performance of OEP decreases against the Bride Burning variant, it increases drastically against the Non exploring variant.

Analysis

1 second of execution time

These results contrast the results obtained in [17] slightly. Compared to the results in [17], the results we obtained show an improvement in performance for the BB MCTS, and a a decrease for OEP. In the result section of [17], it is observed that the performance of the OEP algorithm against other algorithms is improved for longer action sequences. One of the strong aspects of this algorithm is the rapid creation of complete action sequences. While the tree search algorithms have to explore a lot of intermediate nodes in order to reach the end of a round, OEP creates complete sequences on the get go. The action abstraction used when testing these algorithms, as described in the implementation setting, is based on territories, and increases the branching factor while decreasing the number of actions required per action sequence. Taking into account the underlying behaviour of the algorithms, this action sequence length property can explain the difference in results obtained, as the action abstraction used here plays to the strengths of BB and NE MCTS, when compared to OEP.

5 second of execution time

The changes in performance observed in 6.2 make sense, given the impact that the increase in execution time has on the algorithms. The Non exploring algorithm has no real advantage over running for 1 or 5 seconds. Since the tree search has no exploration factor in the Non Exploring algorithm, the tree search will almost always only expand child nodes for one node per depth, even though it visits all child nodes of expanded nodes at least once. This happens because the tree search will always select the most promising node in the tree search when in the selection phase. The only exception to this effect, is when the value of a node decreases due to backpropagation of child nodes, and is then lower than one of its sibling nodes, in which case the sibling node will be chosen in further tree searches instead.

Because of this, if the search is able to get to the end of the action sequence with 1 second of execution, the same nodes will be visited when the algorithm is executed for 5 seconds. One change to the algorithm that may increase the performance of the algorithm in these scenarios is the progressive introduction of some exploration in the tree search once all action sequence ending nodes have been explored. This change would work similarly to the progressive unpruning MCTS variant described in the related work chapter, by introducing new areas of the tree to the search space that would be unexplored otherwise, if there is available computation time.

Since the Bridge Burning algorithm splits search of different depths of the tree based on the total

time budget, it can benefit from the increased execution time, unlike the Non Exploring algorithm. This is proven by the results obtained.

6.2.1 Agents vs Hard AI

Results

Across the board in 6.1, all developed agents show high win rates against the game's Hard AI, with the Bridge Burning and Non Exploring MCTS variants displaying a significantly higher win rate than OEP. These results were obtained from running the agents with computational budget similar to the one used by the Hard AI.

The Bridge Burning and Non Exploring MCTS variants won 4 times more games than the Hard AI in their match up against it, with ties making up approximately 20% of all games, while OEP won approximately twice as many games as the Hard AI in that match up, with similar number of ties.

Analysis

The experimental setting used abstracts some elements of the game, such as air and amphibious combat and the placement of units, which are game aspects that the game's AI takes into consideration. For this reason, this setting may reduce the effectiveness of the game's AI, and the win rates achieved and the performance of the agents developed could be significantly weaker when playing in a different setting featuring those elements. However, as explained when describing the experimental setup, the game's Hard AI takes the production value of a territory into consideration when deciding to try to conquer it or not, so, while our setting may reduce the effectiveness of the AI, it does not cause it to make decisions that are inherently bad. Additionally, our setting is comprised of only two players, while most games are comprised of a significantly higher number of players. Such a setting increases the complexity of the problem, and may cause further decreases in performance for the implemented algorithms.

Even when taking all these factors into consideration, the win rate of the developed agents over the game's AI can still be considered significant, and is of a higher degree than expected. Even with the presence of these factors, these results still highlight the possibility for these types of algorithms to outperform existing AI implementations in the industry. It shows that MCTS and other search algorithms, when paired with powerful pruning strategies and domain knowledge, have the potential to outperform hard coded and rigid video-game AIs, which is the case for many of the current AI solutions in grand strategy video games.

6.3 Analysis of iteration and forward model efficiency

While it can be hard to achieve good efficiency on simulation based algorithms in complicated settings such as grand-strategy games, it is crucial that the algorithms perform efficiently due to the lack of computational budget available. For this reason, we also do an analysis of performance of the forward model and its reflection on the efficiency of the algorithms

The tree search algorithms achieved an average number of iterations per second of 84 iterations. In the first iterations the average number of iterations is around 60, while in latter rounds the tree search achieves an average of 110 iterations. This change in performance as the game progresses happens because playouts in later stages reach terminal states with a smaller depth. The OEP algorithm is able to run an average of 9 iterations per second, with a population size of 50; In [17], the forward model and tree search algorithms run much more efficiently, being able to run on average 43000 iterations per second, and the OEP algorithm runs an average of 615 generations with a population size of 100.

This difference highlights the importance of having an efficient forward model. Based on these averages, the forward model in [17] runs approximately 500 times faster than our model.

Even though Hero Academy is quite simpler that TripleA, which causes a foward model for said game to be faster than for TripleA, there are two additional reasons that can explain the poor performance of our model. Firstly, our model was developed making use of the source code. Even though the code was optimized as much as possible, there still is some overhead that could be eliminated by developing the model from scratch. By doing so, a simpler version could be made that would replicate all the necessary mechanics, while eliminating all unnecessary components. Even though this is the case, the pruning strategies employed counter acted the effect of this lack of performance. The tree search algorithms were still able to reach action sequence ending nodes, and the quality of the actions sequences generated was able to beat the game's Hard AI in our scenario.

The second cause for slow performance of the model, is the playout heuristic used. This heuristic does a thorough search of the current state, and is more complex than the heuristic used in [17]. Using this heuristic represents a trade off, cutting on execution speed while increasing the quality of the state evaluations made after the playout phase. When deciding how to approach the playout heuristic, this tradeoff made sense. Since the algorithms and agents implemented are largely based on deterministic playouts, then having a good quality playout is a key aspect for the success of these algorithms.

7

Conclusion

Using planning in grand strategy video games is a difficult task, especially due to the complex search space and the small computational budget available for decision-making in this setting. In this thesis we planned and implemented different state-of-the-art algorithms based on MCTS over the setting of a grand strategy video game. In order to make the large search space of grand strategy video games approachable for the implemented algorithms, different pruning strategies using domain knowledge were employed. These pruning strategies were effective and were able to reduce the size of the search space significantly, being a key aspect in the performance of the agents.

The implemented algorithms performed differently under varying conditions and execution times, with the Non Exploring MCTS algorithm performing better under higher time constraints, while the Bridge Burning MCTS algorithm was the best performing with higher computational budget. The pruning strategies employed gave an edge to both Bridge Burning MCTS and Non Exploring MCTS over OEP, but the latter still performed considerably well.

All implemented algorithms outperformed the testbed's existing AI when running with a similar computational budget in the experimental setting. These results highlight the possibility of search algorithms as the basis of AI agents for these types of games.

The developed forward model was hard to develop and optimize, and even thought it ran well enough for the agents to produce good action sequences with small computational budget, it's performance was still lacking in comparison to models developed for different, simpler games. A better efficiency model would increase the quality of the algorithms further.

For future work, when considering the development of an AI for TripleA specifically, the model should be adapted to contain the different parts of the game that were abstracted, and the agents should be tested under that setting. The development of a simpler model from scratch would also increase it's performance and allow for experimentation with a lighter pruning of the search space. These algorithms can also be experimented with on a different setting. The Non Exploring MCTS algorithm may also benefit and be enriched from progressive unpruning strategies.

Bibliography

[1] Kocsis, Levente and Szepesvári, Csaba. (2006). Bandit Based Monte-Carlo Planning. Machine Learning: ECML. 2006. 282-293. 10.1007/11871842_29.

[2] Gelly, Sylvain and Silver, David. (2007). Combining Online and Offline Knowledge in UCT. ACM International Conference Proceeding Series. 227. 10.1145/1273496.1273531.

[3] Arpad Rimmel, Fabien Teytaud, Olivier Teytaud. Biasing Monte-Carlo Simulations through RAVE Values. The International Conference on Computers and Games 2010, Sep 2010, Kanazawa, Japan. ffinria-00485555f

[4] Drake, Peter. “The Last-Good-Reply Policy for Monte-Carlo Go.” ICGA Journal 32 (2009): 221-227

[5] Shannon, Claude E.. “Prediction and entropy of printed English.” (1951).

[6] Baier, Hendrik, and Peter D. Drake. ”The power of forgetting: Improving the last-good-reply policy in Monte Carlo Go.” IEEE Transactions on Computational Intelligence and AI in Games 2.4 (2010): 303-309.

[7] Bjornsson, Yngvi, and Hilmar Finnsson. ”Cadiaplayer: A simulation-based general game player.” IEEE Transactions on Computational Intelligence and AI in Games 1.1 (2009): 4-15.

[8] Chaslot, Guillaume and Winands, Mark and Herik, H. and Uiterwijk, Jos and Bouzy, Bruno. (2008). Progressive Strategies for Monte-Carlo Tree Search. New Mathematics and Natural Computation. 04. 343-357. 10.1142/S1793005708001094.

[9] Lucas S.M., Samothrakis S., Pérez D. (2014) Fast Evolutionary Adaptation for Monte Carlo Tree Search. In: Esparcia-Alcázar A., Mora A. (eds) Applications of Evolutionary Computation. EvoApplications 2014. Lecture Notes in Computer Science, vol 8602. Springer, Berlin, Heidelberg

[10] Perez Liebana, Diego and Samothrakis, Spyridon and Lucas, Simon. (2014). Knowledge-based Fast Evolutionary MCTS for General Video Game Playing. IEEE Conference on Computatonal Intelligence and Games, CIG. 10.1109/CIG.2014.6932868.

[11] Silver, D., Schrittwieser, J., Simonyan, K. et al. Mastering the game of Go without human knowledge. Nature 550, 354–359 (2017)

[12] Joris Duguépéroux, Ahmad Mazyad, Fabien Teytaud, Julien Dehos. Pruning playouts in Monte-Carlo Tree Search for the game of Havannah. The 9th International Conference on Computers and Games (CG2016), Jun 2016, Leiden, Netherlands. ffhal-01342347f

[13] Andruszkiewcz, P. Nucl.ai (2015). Optimizing MCTS Performance for Tactical Coordination in Total War: Attila.

[14] Diego Perez, Spyridon Samothrakis, Julian Togelius, Tom Schaul, and Simon Lucas. The General Video Game AI Competition, 2014. www.gvgai.net

[15] Chaslot, Guillaume and Bakkes, Sander and Szita, Istvan and Spronck, Pieter. (2008). Monte-Carlo Tree Search: A New Framework for Game AI. Bijdragen.

[16] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016)

[17] N. Justesen, T. Mahlmann, S. Risi and J. Togelius, "Playing Multiaction Adversarial Games: Online Evolutionary Planning Versus Tree Search," in IEEE Transactions on Games, vol. 10, no. 3, pp. 281-291, Sept. 2018, doi: 10.1109/TCIAIG.2017.2738156

[18] TripleA contributors 2001-20019,¡https://triplea-game.org/¿

Code of Thesis

The code for the thesis is present in:

https://github.com/ManuelFelizardo/MCTSGrandStrategy.git