MCTS-based Planning for Grand Strategy Games

Manuel António Felizardo Roxo

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 videogame. We will focus on TripleA [5], an open source grand strategy video game engine which features a number of different maps. We implemented different MCTS variants, Bridge Burning MCTS [4] and Non Exploring MCTS [4] , as well as an evolutionary algorithm called Online Evolutionary Planning [4]. 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

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 the significant number of actions causes 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.

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) [1] present a good approach to handling planing in this case, as it provides good characteristics for dealing with the issues presented.

Some MCTS variants have been developed to target turn-based adversarial games specifically [4], and have been proven to increase the performance of the algorithm by a big margin under these conditions. 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.

2 Related Work

2.1 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 [3].

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.

2.1.1 Pruning Strategies.

Unwinnable battles, which almost always lead to bad outcomes, were 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.

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.

2.2 Playing Multi-Action Adversarial Games [4]

J.Togelius et al addressed the problem of playing turn based multi-action adversarial games [4]. 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 [4] 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.

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}.

2.2.1 Non-exploring MCTS.

The first Monte Carlo tree search variant adapted for multi-action adversarial games introduced in [4] is the 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.

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.

2.2.2 Bridge-burning MCTS (BB-MCTS).

BB-MCTS is another MCTS variant introduced in [4]. 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=1/2. 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.

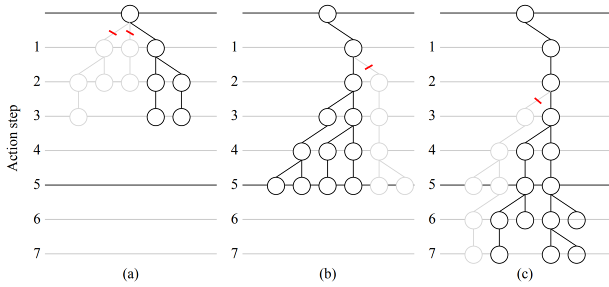

Figure 1. Tree structure as evolved by the Bridge Burning algorithm [4]

Fig. 1 illustrates how nodes are pruned by the Bridge burning algorithm in a multi-action game with 5 actions in a turn.

2.2.3 Online Evolutionary Planning (OEP).

The main algorithm proposed in [4] is an evolutionary algorithm called 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. 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.

Algorithm 1 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
```

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.

3 Case Study - TripleA game

TripleA is a turn-based grand strategy game engine based on the Axis and Allies board game. 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.

3.1 Map

Each game features a discrete map divided into areas, or territories, and units are moved around the map between adjacent areas.

Fig. 2 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.

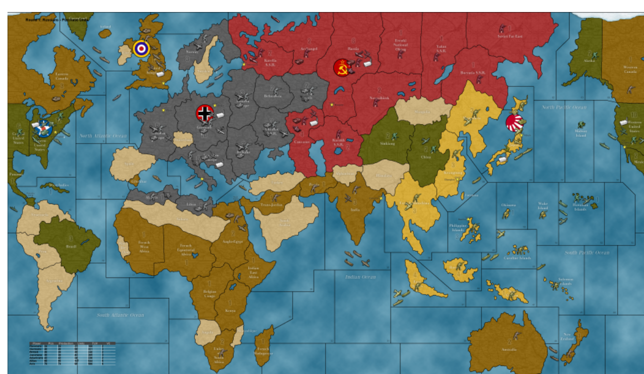

Figure 2. TripleA standard map

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

3.2 Units

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.

3.3 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

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".

3.3.1 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.

3.4 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.

3.5 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.

3.5.1 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.

4 Implementation

The algorithms proposed in [4] 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 [4] 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 [4], 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 section.

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.

We decided to run combat deterministically in our model, always getting the most likely outcome, which allows for quality single execution playout results, and saves computational budget for the algorithms to run more iterations.

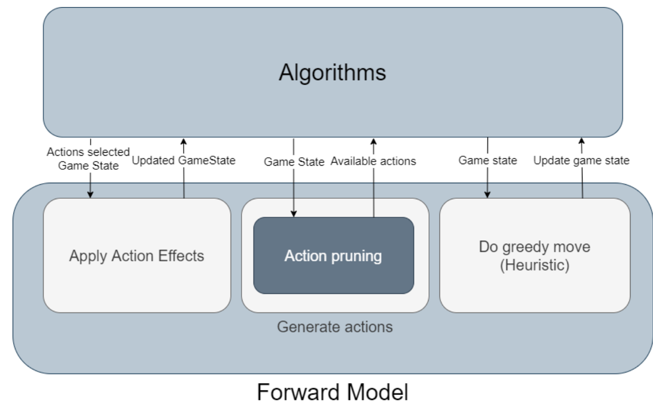

Figure 3. How algorithms interact with the forward model

Fig 3 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.

4.1 Pruning strategies

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 [3], 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 [4], and implement them.

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

4.1.1 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 [3].

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 [3].

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 [3], where sub-trees leading to identical world states were pruned.

Territory action generation. Our approach to action generation is to generate actions for each territory iteratively and generate a possible action for each possible set of units that can move to it.

Using this approach, territories will be ordered and the tree search will generate possible actions for each node based on a single territory, which will change as the tree progresses. For example, on the root node, actions correspond to the first territory. On nodes with depth 2, actions correspond to the second territory, and so on.

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.

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 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.

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).

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.

4.2 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.

4.3 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 2 is used.

Algorithm 2 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)
```

4.4 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.

4.4.1 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.

4.4.2 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.

Algorithm 3 Procreate

```
1: function PROCREATE(genes)
2:     actions = {}
3:     newGenes = {}
4:     for gene g: genes do
5:         actions.add(new Action(t,set))
6:         gene g2 = genes.random()
7:         newGenes.add(g.procreate(g2))
8:     genes.addAll(newGenes)
9:     return genes
   =0
```

4.4.3 Procreate.
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.

4.4.4 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.

4.5 BB-MCTS and NE-MCTS

The Bridge Burning and the Non Exploring MCTS variants are implemented as described in the original paper [4]. 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.

5 Results

In this section 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.

5.1 Experimental setup

5.1.1 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.

5.2 Win rates and agent performance

Results. Table 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 2 presents the results of a similar test setting, but for an agent execution time of 5 seconds.

Table 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 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.

In 2, the results obtained for running agent for a longer period of time of 5 seconds, differ noticeably from the results in 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.

Table 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%

1 second of execution time. These results contrast the results obtained in [4] slightly. Compared to the results in [4], the results we obtained show an improvement in performance for the BB MCTS, and a a decrease for OEP. In the result section of [4], 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 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 section, 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.

5.2.1 Agents vs Hard AI.

Results. Across the board in 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 videogame AIs, which is the case for many of the current AI solutions in grand strategy video games.

6 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.

References

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

[4] 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

[5] TripleA contributors 2001-20019,<https://triplea-game.org/>