Game State and Action Abstracting Monte Carlo Tree Search for General Strategy Game-Playing

Alexander Dockhorn, Jorge Hurtado-Grueso, Dominik Jeurissen, Linjie Xu, Diego Perez-Liebana
School of Electronic Engineering and Computer Science
Queen Mary University of London, London, UK
{a.dockhorn, diego.perez}@qmul.ac.uk

***Abstract*—When implementing intelligent agents for strategy games, we observe that search-based methods struggle with the complexity of such games. To tackle this problem, we propose a new variant of Monte Carlo Tree Search which can incorporate action and game state abstractions. Focusing on the latter, we developed a game state encoding for turn-based strategy games that allows for a flexible abstraction. Using an optimization procedure, we optimize the agent's action and game state abstraction to maximize its performance against a rule-based agent. Furthermore, we compare different combinations of abstractions and their impact on the agent's performance based on the Kill the King game of the STRATEGA framework. Our results show that action abstractions have improved the performance of our agent considerably. Contrary, game state abstractions have not shown much impact. While these results may be limited to the tested game, they are in line with previous research on abstractions of simple Markov Decision Processes. The higher complexity of strategy games may require more intricate methods, such as hierarchical or time-based abstractions, to further improve the agent's performance.**

***Index Terms*—Action Abstraction, Game State Abstraction, General Strategy Game-playing, Stratega, Monte Carlo Tree Search, N-Tuple Bandit Evolutionary Algorithm**

I. INTRODUCTION

Developing artificial agents for strategy games comes with unique challenges that need to be addressed to receive proficient and engaging AI agents. The most prominent one is the enormous game-tree complexity of strategy games, which hinders the applicability of search-based algorithms. Additionally, many strategy games require agents to control multiple units at the same time, resulting in a large combinatorial action space. Similarly, such games often require many moves until completed, which in turn results in an enormous search depth.

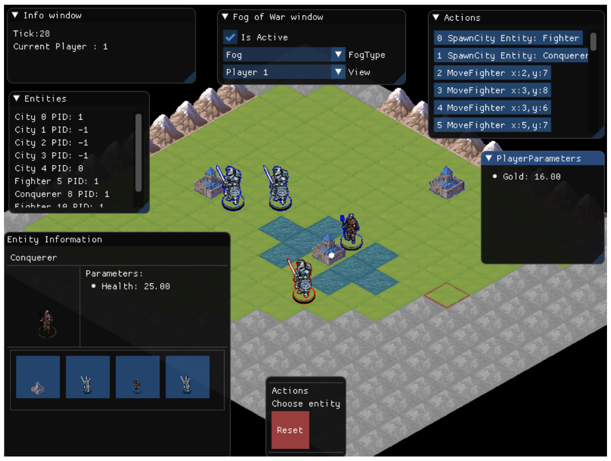
Action and game state abstractions represent two strategies for reducing the game tree's complexity. While action abstractions reduce the set of available actions to a feasible subset, game state abstractions map the states to a smaller subset. Ultimately, those abstractions define what the agent is able to perceive and which actions it will consider. Naturally, the design of these abstractions has a great impact on an agent's performance by reducing the complexity of the search tree. Both have previously been used in conjunction with search algorithms such as Monte Carlo Tree Search (MCTS) [1] and the Rolling Horizon Evolutionary Algorithm (RHEA) [2] and have resulted in considerable performance improvements. Nevertheless, the abstractions in these studies have been provided by human experts. Creating or optimizing abstractions while learning to play a game will be an interesting next step to enhance the abilities of artificial intelligence agents.

In our recent work [2], we explored the use and optimization of action abstractions for turn-based multi-unit strategy games. Therefore, we proposed the Portfolio RHEA (PRHEA) and compared its performance with other action abstracting search-based methods. In its original form, the RHEA algorithm is using evolutionary optimization to optimize action sequences. To boost the agent's performance, we replaced the action assignment, with a unit-script assignment, whereas each script returns an action according to the given game-state. In our study, we optimized the agent's portfolio (the set of scripts), to match the agent's preferences and further emphasize different play-styles. The optimization increased the agents' performance and resulted in more diverse agent behavior.

In this study, we want to expand on our previous approach by also taking game state abstraction into account. Our contributions can be summarized as follows:

- **Flexible state encoding for general strategy games:** We define a flexible state encoding for general strategy games, that allows taking an arbitrary set of parameters into account. Based on such a representation we propose a state abstraction that ignores a set of elements to aggregate similar states.
- **State and action abstracting MCTS:** Given the proposed state-abstraction we propose an MCTS algorithm that makes use of action and state abstractions to reduce the complexity of the search task. To maximize the performance, we propose an optimization scheme for both types of abstractions.
- **Exploring the impact of state and action abstractions:** Finally, we compare the agents' performance and explore the performance gains of both abstraction types.

In Section II we review the STRATEGA framework. Sections III and IV provide an overview of abstractions in search-based methods in the context of artificial intelligence agents. Based on our action-abstracting MCTS algorithm [1], we propose a new variant that can combine action and game state abstraction (see Section V). Therefore, we first define how states of strategy games can be encoded and abstracted, after which we explain how abstracted states can be used to reduce the complexity of



```
Tick:28
Current Player : 1

City 0 PID: 1
City 1 PID: -1
City 2 PID: -1
City 3 PID: -1
City 4 PID: 0
Fighter 5 PID: 1
Conquerer 8 PID: 1
Fighter 10 PID: 1

0 SpawnCity Entity: Fighter
1 SpawnCity Entity: Conquerer
2 MoveFighter x:2,y:7
3 MoveFighter x:3,y:8
4 MoveFighter x:3,y:6
5 MoveFighter x:5,y:7

Gold: 16.00

Conquerer
Parameters:
• Health: 25.00
```

Fig. 1: Graphical User Interface of STRATEGA

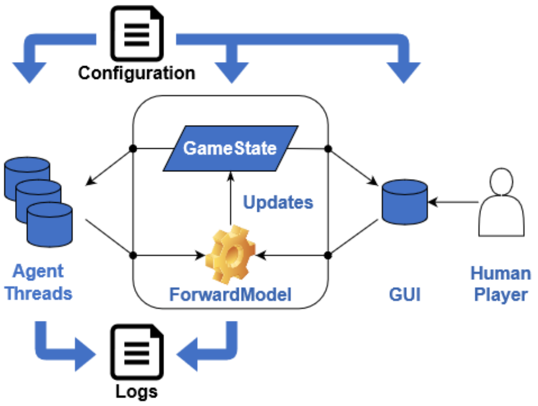


Fig. 2: Overall structure of the framework.

the search tree. Our evaluation in Section VI is split in two parts. First, we describe how the underlying abstractions can be optimized Section VI-B, after which we test and compare the performance of resulting agents Section VI-C. We conclude the paper with an outlook of future work in Section VII.

II. STRATEGA

STRATEGA is a fast and flexible framework for researching AI in complex strategy games. It allows the creation of a wide variety of turn-based and real-time strategy games through YAML configuration files. This permits testing new game and agent configurations without recompiling the whole framework. STRATEGA provides a common API for agent development, which gives access to the game's forward model. This makes it an excellent tool for research on search-based agents.

Figure 2 shows the components of the STRATEGA framework. The game runner is the main component, it contains a game state and a forward model used to update the game whenever it receives a new set of actions. To retrieve an agent's actions, it receives the game's forward model and an observation of the current game state, in which an optional fog of war hides information out of the player's vision range.

The engine can be executed in two different modes:

- **GUI:** Displays the game state and allows the user to analyze or play the game (cf. Figure 1). If there is no human player involved, the GUI will run in spectator mode which allows rendering the battle from the perspective of each player.
- **Arena:** Runs the game in headless mode and simulates a round-robin tournament to test the performance between combinations of agents in different environments. Results of each game will automatically be logged.

In this work, we used the turn-based game-mode *Kill the King* provided by the framework. Here, each player receives a king and a set of additional units for fighting. The players' main goal is to keep their king alive while trying to defeat the opponent's king. This game mode shares some aspects similar to chess, where losing any additional unit does not end the game but can reduce the choices of the player.

More information on our project can be found under:

https://gaigresearch.github.io/afm/

The current state of the framework can be accessed at:

https://github.com/GAIGResearch/Stratega

III. ACTION ABSTRACTIONS

Given a game state, an AI agent tries to find an optimal action, whereas, in this work, optimal means the action that increases the agent's chance of winning the game the most. In case the action space is very large, a full exploration of all actions becomes infeasible. Instead, we can focus the search on a set of promising actions. This concept is reflected in action abstractions, which reduce the complexity of the search by reducing the size of the action space to a smaller subset (cf. Figure 3). Portfolio-based search methods are a popular type of action abstraction, which have shown to considerably improve the performance of strategy game-playing agents [3], [4]. A portfolio consists of a set of scripts, whereas each script selects an action from the given action space based on the agent's observation of the current game state. Instead of searching for the optimal action, a portfolio-based agent will search for the best action returned by any of the scripts, thereby reducing the branching factor of the search to the size of the portfolio.

Portfolio-based search methods vary in the way they encode and optimize the script selection. Portfolio Greedy Search (PGS), Portfolio Online Evolution (POE), and Portfolio Rolling Horizon Evolutionary Algorithm (PRHEA) have previously been tested in the context of the STRATEGA framework and have shown to yield better performance than search-based agents operating on the original action-space [2]. PGS [3] uses a hill-climbing procedure to search for the best script in the current game state. In [4], Justesen et al. have proposed a script- and cluster-based UCT algorithm for Starcraft, in which scripts have been assigned to unit-groups. This approach significantly improved the agent's performance. POE [5] replaces hill-climbing procedure of PGS with an evolutionary algorithm to evolve a script assignment for each unit. Therefore, an

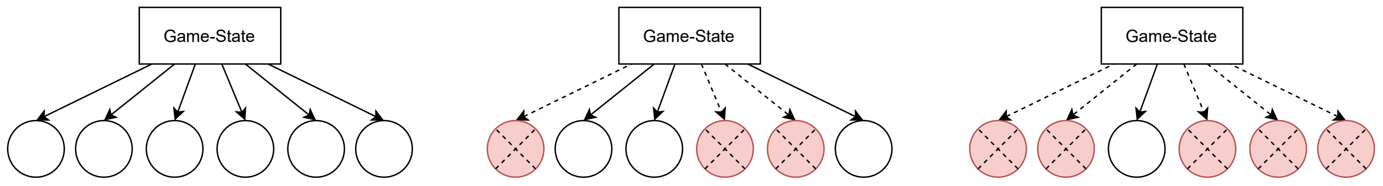


Fig. 3: Action abstractions of various degrees. (left) consider all actions and add their respective child-nodes to the search-tree; (middle) action abstraction ignoring some of the nodes (marked in red); (right) greedy agent ignoring all but one child-nodes.

individual is encoded as a vector of unit-script assignments. An individual's fitness is estimated by simulating the game while repeatedly retrieving the actions from the script assignments and evaluating the resulting game state. PRHEA [2] extends on this by optimizing a sequence of script assignments per unit. In contrast to POE, which operates on a turn-by-turn basis, PRHEA encodes the next n actions, whereas n is called the "horizon". Every time an action is returned, the first action of an individual's action sequence will be removed and a new random action will be appended. Thereafter, evolutionary optimization is used to further improve the individuals of the population. This allows the agent to react to updated observations before returning the next action.

Next to improving an agent's performance, portfolio-based methods allow generating agents of diverse play styles. Perez et al. [1] have used a parameterized version of Portfolio MCTS in combination with MAP-Elites to select and prioritize scripts during the agent's search. In the context of the turn-based strategy game Tribes, the algorithm has shown to generate a wide range of strategies while maintaining high performance.

Another way of reducing the search space is the application of opponent modeling. Here, the agent builds a model of its opponent to predict its future actions. Thereby, we can reduce the number of considered child states during the opponent's turns to a likely subset. Therefore, we argue that opponent modeling can be considered to be a special type of action abstraction that only applies to the opponent's turn. Goodman and Lucas [6] have shown that opponent modeling can increase the performance of MCTS in a real-time strategy game.

IV. Game State Abstractions

Another interesting approach to reduce the complexity of decision-making in games with large state and action spaces is state abstraction, which aims to decrease the amount of states to be explored in the game (or its Markov Decision Process; MDP). This concept may refer to any way of reducing the amount of information needed to describe each state, or to group or cluster different states into similar ones. One of the first works on state abstraction applied to games is [7], [8], where Bulitko et al. propose a method for abstractions in graphs for complex problems such as pathfinding. In this early work, the authors produce different levels of abstractions in a graph, from a grounded level network to the most abstracted version where real-time search algorithms can navigate the full space. Each graph is an image of the next under a given abstraction operator, which maps a subset of states in the lower level graph to a single abstract state in the upper level one. This allows generating a hierarchy of representations where the search is conducted in the most abstracted space and solutions (paths) are progressively refined to the ground level.

Later, Childs et al. [9] studied the effect of transpositions and move groups in MCTS. First, transpositions are states that can be reached through several trajectories during the search, which can allow the tree to merge states that are equivalent or very similar (i.e. game states with the same abstract representation, cf. Figure 4). While this is simple, making moves from these states is less obvious. Then, the authors introduce the concept of move groups, which permit clustering actions that yield a similar outcome, reducing the branching factor and showing its efficacy in artificial game trees and the game Go.

Johanson et al. [10] define a state-space abstraction as a many-to-one mapping between the real state in the game and those from a smaller and artificially constructed abstract game. The main idea behind state-space abstraction is that a game-playing algorithm (such as SFP or Counterfactual Regret Minimization - CFR) operates in the space of abstracted states aiming to achieve an optimal strategy so that it approximates to Nash Equilibrium in the non-abstracted space. The authors evaluate several abstraction methodologies in Texas Hold'em Poker and determined that the distribution-aware techniques are able to provide a great advantage in large game abstractions.

This grouping of states is common through the literature. McMahon et al. [11] by grouping states into *equivalence classes*, each of which captures similar progress made towards specific goals. The authors apply this work to motion planning, where each equivalence class represents a feasible and safe trajectory to be followed; then, the search method selects the class with the lowest cost to be expanded next. In [12], Hostetler et al. propose the concept of state aggregation in MCTS to deal with MDPs with high branching factors. The authors propose aggregating similar states in terms of decision-making, by considering similar states that share the same optimal action. Later on, in [13], the authors introduce the concept of adaptive state abstractions, where these abstractions are progressively refined and adapted through the game. The authors compared these adaptive abstractions with several fixed variants and the grounded (not abstracted) state space, on 12 different experimental problems. Their results show that, despite an increase in the implementation complexity and memory footprint, the adaptive abstractions were able to outperform the other abstractions, including the grounded case.

State abstraction have previously been applied in the context

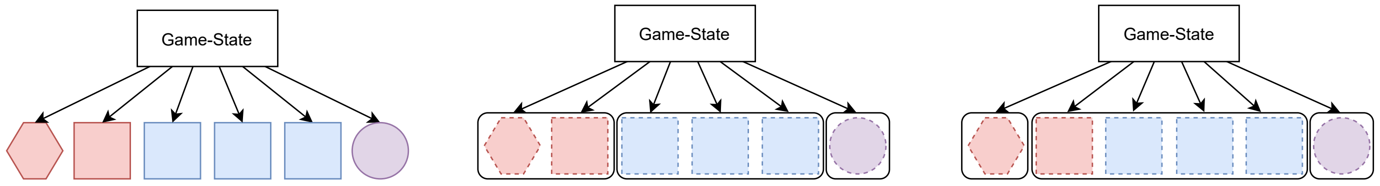


Fig. 4: Game state abstractions based on different properties of the game state. (left) visualization of the search tree showing child nodes that differ in the two attributes shape and color; (middle) grouping nodes by color; (right) grouping nodes by shape.

of strategy games [14]–[16]. Especially in combination with Deep (Reinforcement) Learning agents, various state encodings have shown to significantly impact an agent's performance. While referenced works leave the internal representation of the state to the learning capabilities of a neural network, we want to study how changing the state encoding reflect on an agent's planning capabilities. To the best of our knowledge, none of the related works have applied such analysis to complex strategy games, which count on large state spaces with intricate information.

V. Method

In the following, we present our ideas on incorporating game state abstractions with MCTS. Therefore, we need to answer three questions, i.e. (1) "How can a game state be encoded?", (2) "How can we create an abstract representation of a game state?", and (3) "How can we make use of this abstraction in MCTS?". These three questions will be addressed in the subsections below.

A. Flexible State Encoding and Evaluation Function

First of all, we need to define how we can flexibly represent a complex game state of a turn-based strategy game. Our design choices are heavily influenced by the inner workings of the Stratega framework. Since this framework aims to be a general strategy framework that is already capable of modeling a diverse set of strategy games, we believe that this procedure can easily be adapted for other games of this genre.

In the following, we assume that a game consists of three components:

- **Map:** An $m \times n$ grid of tiles, whereas each tile type has a unique identifier. Parts of the map may be hidden by a fog of war, which replaces tiles that are not uncovered by the agent with a default tile.
- **Entities:** A set of units, buildings, etc which each have a unique identifier, a position on the map, and a set of properties. Entities may be hidden by a fog of war, which means that the agent cannot perceive the entity nor any of its properties.
- **Entity Properties:** Anything that is used to encode the properties of an entity, e.g. health points, attack points, etc. Each property needs to define a range of valid values.

When we observe a game state, we want to represent it as an ordered set of vectors. Therefore, we need to retrieve a list of all observable entity properties. For each parameter, we will fill a vector based on the property values of observed entities. Therefore, we first sort all observed entities by their *id* and then add their property values to the respective vectors. Those vectors might differ in length in the case of entities that exhibit different properties, e.g. buildings might be unable to attack, therefore they will not need an attack value. Similarly, we can add the units' positions and the observed tile types to this representation. This makes the comparison of two states very simple. We just need to check if their resulting set of vectors are the same. Furthermore, it allows us to ignore certain elements of the game state to form an abstract state representation.

To evaluate a state, we first map observed parameters per entity into the range of $[0, 1]$ and use a weighted sum to aggregate the resulting values into a single score value. Here, we use a modified version of the Uniform Rational Quantization (URQ) function [17] to map an observed parameter value x from the range $[x_{min}, x_{max}]$ to the range of $[0, 1]$.

$$\text{URQ}(x, u) = \frac{u \cdot (x - x_{min})}{u \cdot (x - x_{min}) - x + x_{max}} \tag{1}$$

The resulting value is multiplied by a weight, which allows highlighting the importance of a parameter. Furthermore, we multiply the resulting value with -1 in case the corresponding entity is controlled by an opposing player. Finally, we calculate the sum of all such values to rate the state.

The resulting state heuristic can easily be optimized for any game of the Stratega framework. Nevertheless, we provide a default heuristic, which can be used for the Kill the King game, which has shown acceptable performance in preliminary evaluations. In the upcoming experiments, we chose a weight of 1 for all parameters except for a unit's *Health* value, for which we use 10. Similarly, we chose a u value of 1 for all but a unit's *Health*, which is using a u value of 5 to focus on killing units that are low on health points. Additionally, we added a reward of 1000 in case the game has been won or deduct this value in case the game has been lost.

B. Game State Abstractions for Monte Carlo Tree Search

The MCTS algorithm consists of four phases, i.e. (1) Selection, (2) Expansion, (3) Simulation, and (4) Backpropagation. Starting with the root node that represents the current state, we simulate the outcome of available actions and add their resulting states as child nodes. During selection, we traverse the tree downwards to select a node that represents a promising game state. In vanilla MCTS, all actions available in that node are considered during the expansion phase. Once an action has been chosen, it is applied to the current game state and added as a child node of the previously selected node. To incorporate game state and action abstractions we modify the expansion phase, in which a selected node is expanded by a child node.

As proposed in our previous paper [1], we incorporate action abstractions by reducing the action space to the actions returned by a portfolio of scripts. A total of six scripts have been implemented in the STRATEGA framework and define our agent's portfolio. The script implement the following logic:

- **Attack Closest**: The selected unit attacks the closest opponent. If no unit is in attack range, the units walks to the closest opponent unit to prepare an attack. In case none of the opponent's units is visible, act randomly.
- **Attack Weakest**: Attack the weakest visible opponent unit or walk closer to it. In case none of the opponent's units is visible, act randomly.
- **Run Away**: Walk to the position that maximizes the distance to all known opponent units. In case none of the opponent's units is visible, act randomly.
- **Run To Friends**: Walk to the tile that minimizes the distance to all friendly units. In case the current unit is the last friendly unit remaining, act randomly.
- **Use Special Ability** Each game's configuration can define special abilities such as healing or attack moves with higher damage. The script uses a random special ability of the selected unit. In case no special ability is available it selects a random action .
- **Random**: Use a random action. Required to ensure a small chance of selecting any action, and therefore, ensures that the whole search space can be explored.

To further reduce the branching factor, we incorporate game state abstractions to combine states with a similar abstract state representation. Therefore, each node will store the game state as well as its abstract representation. Every time, we want to expand a node, we first simulate the outcome of the action and then compare its abstract state representation with existing child nodes of the selected node. In case no matching abstract state can be found, we add the node as usual. If a similar abstract state is already present, we add the action to the existing child node and add another rollout to the respective node.

This state aggregation procedure can be applied to vanilla MCTS and Portfolio MCTS. It effectively groups game states of similar abstract representation. This reduces the branching factor and yields a better estimate of the existing nodes by aggregating their rollouts. The impact of state aggregation depends on a variety of parameters, e.g., the chosen abstract state representation, the number of actions per game state as well as the difference of their outcome. A typical example of a good abstract representation is the reduction of redundancy, e.g. grouping mirrored states in a board game. In terms of general strategy games, it is hard to predict which components of the game will result in a redundant encoding. Therefore, we will devise the proposed flexible state encoding to explore which elements can be left out without degrading the agent's performance.

Similarly, the impact of the action abstraction depends on the quality and the diversity of actions returned by the portfolio's scripts. By adding or removing scripts, we can drastically modify the set of behaviors an agent can express. This can make the agent more efficient but also reduce its performance considerably. Similar to the game state abstraction, we will optimize the action abstraction by modifying the agent's portfolio as part of the upcoming evaluation.

VI. EVALUATION

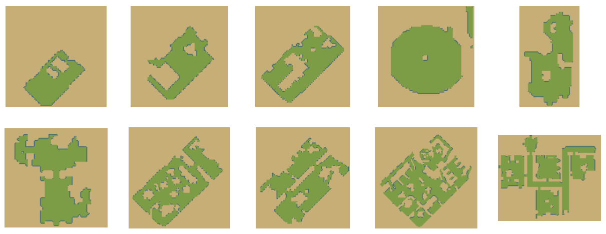
To get an impression of the proposed agent's performance we want to compare the different abstraction variants of the MCTS algorithm with each other and with a rule-based agent as a common baseline. Throughout our experiments, we will be using the *Kill the King* game of the STRATEGA framework and aim to analyze the impact of various abstraction styles on the agent's search and its resulting performance. In total, we will be considering four variants of MCTS, i.e. vanilla MCTS, Portfolio MCTS (P-MCTS), state-abstracting MCTS (S-MCTS), and a hybrid considering both abstractions (H-MCTS). The rule-based agent, also called *Combat Agent*, is focusing on grouping its units while quickly approaching the opponent. During combat, it tries to target units that can be killed using a single attack and prefers to attack units that are already weakened. It has previously been used in an early version of the framework and showed to outperform vanilla MCTS, RHEA, and a greedy agent [18].

In the following, we will first discuss our preparations for a diverse set of maps to be played, after which we introduce our procedure for optimizing the agents' parameters, and finally present the results of a simulated round-robin tournament among the optimized agents and the rule-based baseline.

A. *Generating Diverse Map*

For both evaluations, we have used maps from Nathan Sturtevant's set of path-finding benchmarks [19]. Specifically, we parsed Baldur's Gate 2 and Dragon Age Origin maps to use them in the STRATEGA framework. In total, we selected 40 maps which vary in size and represent diverse fighting landscapes for our agents. Smaller maps tend to represent different room layouts, while larger maps tend to be more open. We split the set of maps into 10 maps for optimization and 30 maps for the performance evaluation. Figures 5a and 5b show the maps of the two resulting map sets.

To test the agents' capability to adhere to different fighting styles, we have considered 5 army compositions in our evaluation. Figure 5c lists the army compositions created for our experiments. During parameter optimization, only the first army composition has been used, while our second evaluation phase covers all 5 army compositions. To position the units we have first selected two central positions among all movable cells of a map. At those positions, the player's king will be placed. Furthermore, we split cells into two groups according to their closer king. The remaining units per player will be placed randomly in these cell sets, which ensures that they will be closer to the player's king than the opponent's king


(a) 10 Training Maps

(b) 30 Test Maps

	King	Warrior	Archer	Healer
Army 1	1	3	3	3
Army 2	1	×	×	3
Army 3	1	×	10	×
Army 4	1	10	×	×
Army 5	1	5	5	×

(c) Army compositions used during our evaluation.

Fig. 5: Overview of maps and unit compositions that have been used during optimization (training) and evaluation (test). The training set consists of 10 map layouts (a) and only includes 1 version per map in which army 1 has been randomly positioned. The test set consists of the 30 map layouts (b), whereas each map layout is included 5 times, once for each unit composition in (c).

and therefore have a chance to defend him. For the test set, we receive 5 different versions per map (each representing an army composition) for a total of 500 maps used during performance evaluation.

B. Optimization of the Agents' Parameters

The four MCTS variants vary strongly in the way they structure their search tree. Therefore, we want to carefully optimize their respective parameters, so we can make a fair comparison of their resulting performance. For this purpose, we will be using the N-Tuple Bandit Evolutionary Algorithm (NTBEA) [20], which balances the exploration of previously unseen parameter combinations and the exploitation of promising combinations.

For the vanilla MCTS agent, we optimize three parameters, i.e. the exploration constant of the UCB selection criterion ($\{0.1, 1, 10, 100\}$), the rollout length ($\{1, 5, 10, 15, 20, 25, 30\}$), and the script used to simulate the opponent's turn (attack closest, attack weakest, random, skip). In the case of the action-abstracting MCTS, we additionally optimize its portfolio set consisting of the following script (attack closest, attack weakest, run away, run to a friend, use an ability, random), whereas each script can either be included in or left out of the portfolio. This results in a total of 9 parameters. For the state-abstracting variant, we use the three base parameters of MCTS and on top of that, decide about 8 properties of the game state that can be included or left out in the game state abstraction, i.e. the map, the unit positions, the units' properties including current health, max health, attack damage, attack range, heal amount, and movement points. Finally, the parameters of the hybrid agent consist of the 3 base parameters, the 6 scripts of the portfolio, and the 8 game state properties, resulting in a total of 17 parameters.

Given a budget of 50 iterations, we search for the best parameter combination per agent. During each iteration, the algorithm selects the most promising one out of 10 neighboring parameter combinations and evaluates its fitness by simulating 20 games against the rule-based *Combat Agent*. During these 20 games, the agents will play the 10 training maps, two times each, with interchanged starting positions. The agent's points are increased by 3 for each win and remain unchanged after a loss. After 100 turns the game is automatically terminated and results in a draw. In the latter case, the agent's points are increased by 1.

Figure 6 shows the points scored during each iteration of the optimization. The graphs clearly show the impact of action abstractions, since both action-abstracting algorithms (P-MCTS and H-MCTS) perform much better than their competitors. The portfolio of P-MCTS consists of the two attack-based scripts, while H-MCTS is additionally using the "run to friend" script. On top of that, the H-MCTS algorithm has removed most properties from the game state encoding. Only the attack damage, the unit's healing range as well as its movement points will be used to differentiate game states. Our proposed hybrid algorithm was able to match the performance of P-MCTS after a few iterations and from there on quickly

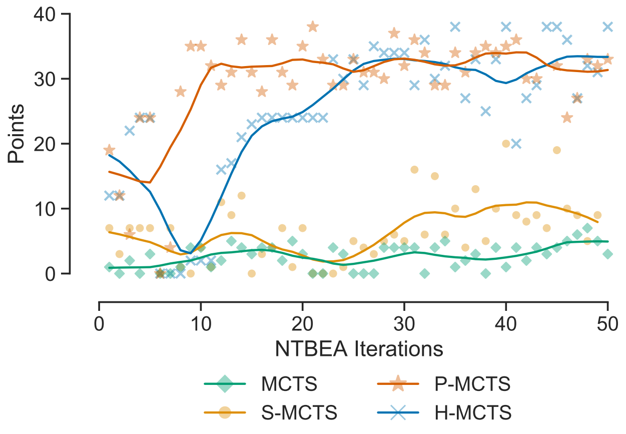


Fig. 6: Results of the parameter optimization using NTBEA. Shown points are the results of 20 games, whereas the agent receives 3 points per win, 1 point per draw and 0 points for each lost game. A local regression (width = 10) has been added for an easier comparison.

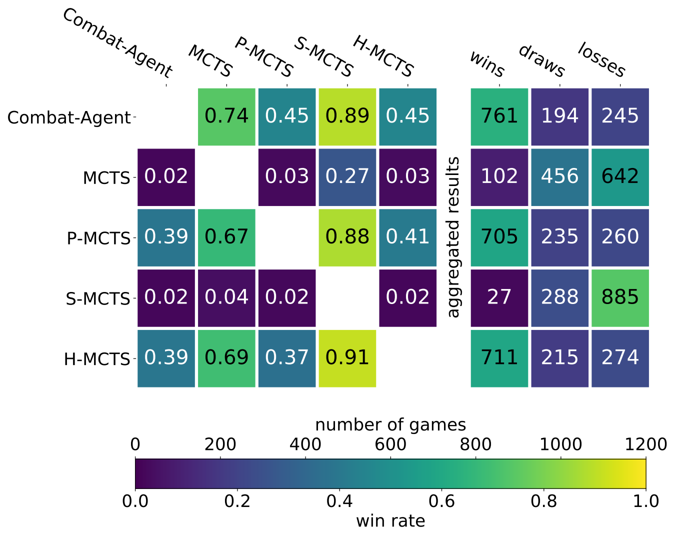


Fig. 7: Performance comparison in a round-robin tournament of 300 games per match-up throughout the three tested game-modes. Each cell (i, j) shows the win rate of the agent in row i against the agent in column j. The columns to the right show the total number of wins, draws, and losses per agent.

stabilized in performance. The game state abstracting S-MCTS has performed slightly better than vanilla MCTS. There is only a marginal difference in the performance of these two agents, which is not surprising when analyzing the abstraction used by the optimized S-MCTS agent. It basically selected not to remove any properties from the game state encoding; therefore, it produces similar search trees to vanilla MCTS. All algorithms prefer to use shorter rollouts of length 1 or 5, while the evaluation during optimization have rarely shown much difference between these two options.

Regarding our evaluation, we want to note that the size of the search space and its dimensionality considerably differs for each optimization problem. While the search space of MCTS consists of 112 parameter combinations (and 3 dimensions), P-MCTS includes 7168 (9 dimensions), S-MCTS 28672 (11 dimensions), and H-MCTS a total of 1835008 possible parameter combinations (17 dimensions). Using a fixed number of iterations results in a disadvantage for the more complex optimization tasks. We have observed that NTBEA quickly identified useful action abstractions but had a harder time in finding beneficial game state abstractions. Based on our observation, other optimization algorithms might perform better than NTBEA for such high-dimensional optimization tasks.

C. Performance Comparisons

Given the optimized parameter sets per agent, we want to compare their game-play performance given a larger variety of unit compositions and test maps. Our test set includes 30 maps for which we include 5 versions per map, one for each of the unit compositions shown in Figure 5c. This results in a total of 150 scenarios, which are played twice with interchanged roles for a total of 300 games per match-up.

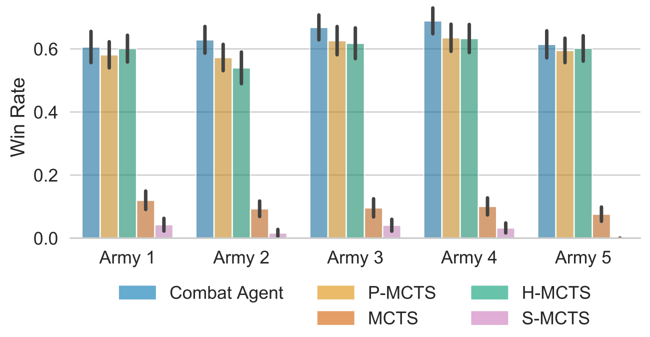


Fig. 8: Results of the round-robin tournament. Average win rate per combination of agents and unit compositions and its standard deviation.

Figure 7 shows the win rates for each match-up in the simulated round-robin tournament and the resulting average win rates per agent as well as their total number of wins, draws, and losses. Our results show that the Combat Agent performed best again all its competitors, closely followed by P-MCTS and H-MCTS. Agents that did not make use of action abstraction, i.e. MCTS and S-MCTS, have performed much worse and lost their respective match-ups against the three top-performers.

Furthermore, we tested if the performance of each agent varies given the different unit compositions. Figure 8 shows the average win rate of each agent per army. The graphs show that the performance is mostly unaffected by the starting army composition. Therefore, we expect that there is no significant overfitting with respect to the single army composition used during training.

VII. Conclusion

In this paper, we explored implementations of action and game state abstractions for games of the Stratega framework. We proposed a flexible state encoding capable of including or excluding a varying set of game-relevant properties. This can be used to aggregate states with similar encoding during the agent's search process. Adding this aggregation process to a Portfolio MCTS agent resulted in a hybrid search algorithm that is taking action and game state abstractions into account.

We explored the impact of those abstractions by first optimizing agents against a rule-based baseline and then simulating a round-robin tournament between all agents. The action abstraction has shown to drastically improve the performance of our MCTS agent, while game state abstractions have not shown to significantly impact the agent's performance. While these results are limited to our observations on playing a single game mode, they might hint that the devised abstraction type is insufficient for addressing the complex state spaces of strategy games. This would be in line with previous research on MDPs, which has shown that some state abstractions are limited to simple MDPs.

While game state abstractions have not performed well in this setting, we still believe that there is much potential in this overall application. Time-based and hierarchical [7], [8] abstractions might prove useful in addressing the complex state spaces found in the strategy game genre. Overall, those abstractions may allow agents to create longer plans and therefore further improve their performance.

In terms of action abstractions, we have seen that the 6 rule-based scripts used in this study have shown to be a simple and efficient way of improving a search-based agent's performance. However, the resulting agent has not been capable of defeating the rule-based combat agent, therefore, raising the question of how the portfolio set can be adapted to achieve similar or even better performance. Therefore, we would like to explore the optimization or creation of scripts using machine-learning methods. Especially interesting would be an analysis of the trade-off between the number of scripts used and the agent's resulting performance. Minimizing the number by choosing or generating a set of scripts that complement each other will be an interesting task for future work.

Acknowledgments

This work was supported by UK EPSRC grant EP/T008962/1.

References

[1] D. Perez-Liebana, C. Guerrero-Romero, A. Dockhorn, and L. Xu, "Generating Diverse and Competitive Play-Styles for Strategy Games," *IEEE Conference on Games, COG*, 2021.

[3] D. Churchill and M. Buro, "Portfolio Greedy Search and Simulation for Large-scale Combat in StarCraft," in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*. IEEE, Aug. 2013.

[2] A. Dockhorn, J. Hurtado-Grueso, D. Jeurissen, L. Xu, and D. Perez-Liebana, "Portfolio Search and Optimization for General Strategy Game-Playing," in *IEEE Congress on Evolutionary Computation (CEC)*, 2021.

[4] N. Justesen, B. Tillman, J. Togelius, and S. Risi, "Script- and cluster-based UCT for StarCraft," in *2014 IEEE Conference on Computational Intelligence and Games*. IEEE, Aug. 2014. [Online]. Available: https://doi.org/10.1109/cig.2014.6932900

[5] C. Wang, P. Chen, Y. Li, C. Holmgård, and J. Togelius, "Portfolio Online Evolution in StarCraft," *Aiide*, pp. 114–120, 2016.

[6] J. Goodman and S. Lucas, "Does it matter how well i know what you're thinking? opponent modelling in an rts game," in *2020 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2020, pp. 1–8.

[7] V. Bulitko, N. Sturtevant, and M. Kazakevich, "Speeding up learning in real-time search via automatic state abstraction," in *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 3*, ser. AAAI'05. AAAI Press, 2005, p. 1349–1354.

[8] V. Bulitko, N. Sturtevant, J. Lu, and T. Yau, "Graph abstraction in real-time heuristic search," *Journal of Artificial Intelligence Research*, vol. 30, pp. 51–100, Sep. 2007.

[9] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in monte carlo tree search," in *2008 IEEE Symposium On Computational Intelligence and Games*, 2008, pp. 389–395.

[10] M. Johanson, N. Burch, R. Valenzano, and M. Bowling, "Evaluating state-space abstractions in extensive-form games," in *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, 2013, pp. 271–278.

[11] J. McMahon and E. Plaku, "Sampling-based tree search with discrete abstractions for motion planning with dynamics and temporal logic," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, Sep. 2014.

[12] J. Hostetler, A. Fern, and T. Dietterich, "State aggregation in monte carlo tree search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, no. 1, Jun. 2014.

[13] ——, "Sample-based tree search with fixed and adaptive state abstractions," *Journal of Artificial Intelligence Research*, vol. 60, pp. 717–777, Dec. 2017.

[14] L. Han, J. Xiong, P. Sun, X. Sun, M. Fang, Q. Guo, Q. Chen, T. Shi, H. Yu, and Z. Zhang, "Tstarbot-x: An open-sourced and comprehensive study for efficient league training in starcraft II full game," *CoRR*, vol. abs/2011.13729, 2020. [Online]. Available: https://arxiv.org/abs/2011.13729

[15] D. Lee, H. Tang, J. O. Zhang, H. Xu, T. Darrell, and P. Abbeel, "Modular architecture for starcraft ii with deep reinforcement learning," in *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[16] N. Justesen and S. Risi, "Learning macromanagement in starcraft from replays using deep learning," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, Aug. 2017. [Online]. Available: https://doi.org/10.1109/cig.2017.8080430

[17] C. Schlick, "Quantization techniques for visualization of high dynamic range pictures," in *Photorealistic rendering techniques*. Springer, 1995, pp. 7–20.

[18] A. Dockhorn, J. H. Grueso, D. Jeurissen, and D. Perez-Liebana, ""Stratega": A General Strategy Games Framework," *Artificial Intelligence for Strategy Games Decision, AIIDE 2020 Workshop*, 2020.

[19] N. Sturtevant, "Benchmarks for grid-based pathfinding," *Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144 – 148, 2012. [Online]. Available: http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

[20] S. M. Lucas, J. Liu, and D. Perez-Liebana, "The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation," in *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2018, pp. 1–9.