Strategy Game-Playing with Size-Constrained State Abstraction

Linjie Xu, Diego Perez-Liebana
School of EECS
Queen Mary University of London, London, UK
{linjie.xu, diego.perez}@qmul.ac.uk

Alexander Dockhorn
Faculty of EECS
Leibniz University Hannover, Hannover, Germany
dockhorn@tnt.uni-hannover.de

***Abstract*—Playing strategy games is a challenging problem for artificial intelligence (AI). One of the major challenges is the large search space due to a diverse set of game components. In recent works, state abstraction has been applied to search-based game AI and has brought significant performance improvements. State abstraction techniques rely on reducing the search space, e.g., by aggregating similar states. However, the application of these abstractions is hindered because the quality of an abstraction is difficult to evaluate. Previous works hence abandon the abstraction in the middle of the search to not bias the search to a local optimum. This mechanism introduces a hyper-parameter to decide the time to abandon the current state abstraction. In this work, we propose a size-constrained state abstraction (SCSA), an approach that limits the maximum number of nodes being grouped together. We found that with SCSA, the abstraction is not required to be abandoned. Our empirical results on 3 strategy games show that the SCSA agent outperforms the previous methods and yields robust performance over different games. Codes are opensourced at https://github.com/GAIGResearch/Stratega.**

***Index Terms*—Game artificial intelligence, state abstraction, monte carlo tree search, planning**

I. Introduction

Strategy games have helped advance the development of Artificial Intelligence (AI) to achieve significant progress in competing with human players [1, 2], AI-AI cooperation [3, 4, 5] and human-AI cooperation [3, 6, 7, 8]. Most of this progress depends on deep reinforcement learning (DRL). However, DRL agents have their neural networks trained and tuned for a specific game, making it difficult to apply these agents to other game variants. In contrast, search-based algorithms such as Monte Carlo Tree Search (MCTS) have shown outstanding performance in general video game-playing [9, 10, 11]. The ability to play different game variants is important because real-world games are frequently updated by their developers. Therefore, in this work, we focus on search-based methods for strategy game playing.

One of the most challenging problems for search-based algorithms is the combinatorial search space. Unfortunately, strategy games typically have a combinatorial search space. In strategy games such as Starcraft, a number of units (e.g. buildings, and armies) are distributed on the map. The state space of these games is defined as the combination of unit properties (e.g. positions, health points). This combinatorial space increases exponentially with the number of game components (including the unit number and unit property etc.) [12, 13]. On top of that, most strategy video games have a large set of unit variants and each unit has a diverse set of properties. Together, they produce large state and action spaces, resulting in a much larger branching factor compared to other games. With a large branching factor, MCTS finds it difficult to explore the tree deeply for accurate action-value approximation and thus fails to perform well in these games.

State abstraction [14, 15] is a powerful technique that helps MCTS solve large-scale planning problems. State abstraction methods focus on simplifying the search space, which is often achieved by aggregating similar states. In strategy games, state abstraction [16, 17, 18, 19] has been applied to reduce the search space and gain significant performance improvements. However, one of the issues that hinder the application of state abstraction is a lack of data for approximating the state abstraction, resulting in a possible poor-quality state abstraction. To avoid this state abstraction to degrade the performance, Xu et al. [19] proposed an *early stop* mechanism to abandon the constructed state abstraction at an early stage. However, this approach introduces a hyperparameter whose range depends on the training budget, making it difficult to select an appropriate value.

In this paper, we propose the size-constrained state abstraction (SCSA), a novel approach to address the negative effect of a potential poor-quality state abstraction. SCSA limits the maximal number of nodes in the same node group and does not need the *early stop*. Meanwhile, its hyperparameter is less sensitive to the previous approach. Finally, we evaluate the SCSA agent in 3 strategy games using a common value of this size limit. It outperforms all the baseline agents in 2 simple games and achieves results competitive to Elastic MCTS [19] in another more complex game.

The main contributions of this work are listed below:

1) We proposed a novel approach to address planning with a poor-quality state abstraction in strategy game-playing.
2) Our empirical results show that the proposed method achieves outstanding performance in 3 strategy games of different complexity.
3) We analyzed the compression rate under the SCSA and Elastic MCTS [19]. SCSA shows a lower compression

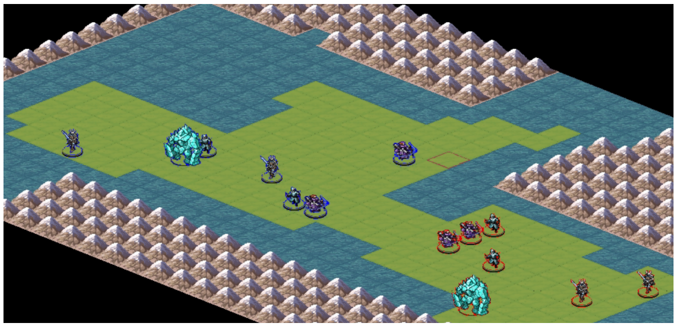


Fig. 1: A screenshot of *Kill The King* game. In this case, each player has one *king*, two *warriors*, two *archers* and two *healers*

Fig. 2: A screenshot of *Push Them All* game. Each player has three *pushers* that try to push enemy units into holes.

Fig. 3: A screenshot of *Two Kingdom* game. The blue side spawned a *worker* to mine the gold and the red side spawned three warriors to protect the *king*.

rate, revealing a trade-off between memory usage and agent performance under the state abstraction.

II. Related Work

State abstraction for MCTS recently gained much interest from the community. Jiang et al. [14] proposed to aggregate same-layer tree nodes with Markov decision process homomorphism approximated from samples. This method shows a promising performance in the board game Othello. Anand et al. [20] proposed a state-action abstraction method that aggregates state-action pairs instead of states (tree nodes). Anand et al. [21] propose progressive state abstraction that updates the state abstraction more frequently instead of per batch. Hostetler et al. [15] proposed a progressive refinement method to construct state abstraction. Baier et al. [22] proposed *abstraction over opponent moves* to aggregate tree nodes having the same opponent moving history. Sokota et al. [23] proposed *abstraction refinement* to reject similar states to be added in the tree. These methods prove the effectiveness of state abstraction in tackling large branching factors in MCTS. However, their application is mainly limited to planning problems and board games. This work instead focuses on more complex strategy games.

In the early study, hand-crafted state abstraction were applied to help strategy game play. Chung et al. [16] used a handcrafted state abstraction to divide the game map into tiles. Synnaeve et al.[17] proposed a mechanism to separate the map in StarCraft to regions that are connected through checkpoints. Uriarte et al.[24] also used the technique developed by Synnaeve et al. [17] but further removed combat-irrelevant units from the map. Although these hand-crafted state abstractions can significantly reduce the size of state space for some games, they rely on human heuristics and thus fails to generalize to different games.

Except for hand-crafted state abstraction, automatic state abstraction are also explored in strategy game-playing in recent years. A parameter optimization method [25] was leveraged to search unit features that can be removed from the unit vectors. By removing some features, states having all other features the same are merged. Dockhorn et al. [18] proposed to represent game states with a combination of unit vectors. Our work is closely related to Xu et al. [19], where an elastic MCTS method is proposed for strategy game-playing. In elastic MCTS, the state abstraction is first constructed in a batch manner, similar to Jiang et al. [14]. Later, the constructed state abstraction is abandoned and abstract nodes are split into ground tree nodes. Our work does not need to abandon the state abstraction, and thus is more consistent with state abstraction usage in planning [14, 15, 20, 21].

III. The Stratega platform

Stratega [26] is a general strategy game platform for testing AI agents. To evaluate the general performance of the proposed method, we select 3 two-player turn-based strategy games from the Stratega platform. They are *Kill The King (KTK), Push Them All (PTA)* and *Two Kingdoms (TK)*. We next introduce the details of these games.

In *KTK* (Figure 1), each player controls a set of units including a *king*. The goal of this game is to kill the opponent's *king*. We instantiate the army for each player as a *king*, a *warrior*, an *archer*, and a *healer*. All units have the move action. Based on that, the *king* and the *warrior* can attack neighbour enemy units. The *archer* can attack enemy units in range. The *healer* can heal ally units. Following Xu et al. [19], each unit also has an *Do-nothing* action. The action space size for a 4-unit army is about 10^5.

In *PTA* (Figure 2), a player controls units to push enemy units in different directions. The unit being pushed will move its position toward the corresponding direction. To win this game, all the enemy units need to be pushed into holes distributed in the map. Each player has 3 *pusher* units. The action set for each *pusher* is *[Move, Push, Do-nothing]*, resulting in an action space of $(4+1) \times 4 \times 4 = 80$. The first term indicates moving in 4 directions or not moving, the second term is selecting a neighbour unit (there are 4 neighbour grids) and the last term is pushing the enemy unit in 4 different directions. With 3 *pushers*, the final action space is $80^3 = 512,000$.

The *TK* game (Figure 3) is more complex. It consists of technologies, resources, unit spawning, and combat. At the beginning of a gameplay, each player has a *castle* and a *king*. The aim is the same as *KTK*, i.e. kill the opponent *king*. However, a set of units need to be spawned from the *castle*. A technology *Mining* is required to be researched for spawning *worker*. The research takes one round to be finished. The *Worker* unit can collect gold from gold veins and *Warrior, Night, Wizard* and *Healer* can be spawned with gold.

IV. Background

A. Monte Carlo Tree Search

MCTS [27] is a method to solve sequential decision-making problems with a forward model. The forward model is used to roll out the game. I.e., given a state and a valid action under this state, the forward model returns the next state. Using the forward model, MCTS builds up a tree to approximate the value for actions under the current state. In the generated tree, each node represents a game state and each branch represents a valid action of its source node. We next introduce the 4 stages for building up this tree: *selection, expansion, rollout* and *back-propagation*.

The *selection stage* selects a tree node as an input for the subsequent stages. The *selection* starts from the root node and keeps selecting a branch to the next layer until a target node is reached. The target node could be a leaf node (a node with no children) or a node where not all its actions have been added to the tree as branches. To select the next-layer node, node values (e.g. the UCB value [28]) for all its children are calculated and the node with the highest UCB is selected. Depending on the node type of the target node, MCTS enters different stages. If the target node is a terminal state, it enters the *back-propagation* directly. In another case, the target node is a non-terminal state, an action that has not yet been added as a branch. By running this action in the forward model, the next state is returned and is added as a new child. Based on this new state, a roll-out policy takes a sequence of actions until a pre-determined depth or a terminal state is reached. This is the *rollout* stage. The output state from rollout is evaluated by a state evaluation function to obtain a score. This score is used by the *back-propagation* stage. In the *back-propagation* stage, the score from the target state is added to all states in the trajectory of selection. i.e. a node sequence from the root node to the target node.

Each MCTS iteration consists of these 4 or 3 stages (the roll-out stage is skipped if the selected node is a terminal state). The computation budget in this work is set as the maximum number of forward model calls. After running out of the budget, a recommendation policy selects an action to execute in the game. A common recommendation policy is selecting the branch leading to a node with the highest visit count.

B. Monte Carlo Tree Search with Unit Ordering

In strategy games where many units are distributed on the map, the action space is the combination of all unit actions, which can easily reach a high complexity. e.g. in *KTK*, the combinatorial action space reaches a magnitude of 10^5. To reduce the action space, Xu et al. [19] propose the MCTS with unit ordering (MCTS$_u$). In MCTS$_u$, the move ordering of units is randomly initialized and is fixed throughout the whole game. Each node controls only one unit and its children control the subsequent unit in the move order. With this setting, the tree becomes deeper but narrower. MCTS$_u$ has shown a strong performance in the multi-unit strategy games.

C. State Abstraction and Approximate MDP Homomorphism

A Markov Decision Process (MDP) is defined as $\langle \mathbb{S}, \mathbb{A}, R, P, \gamma \rangle$, where the $\mathbb{S}$ is the state space, $\mathbb{A}$ the action space, $R : \mathbb{S} \times \mathbb{A} \mapsto \mathbb{R}$ the reward function, $P : \mathbb{S} \times \mathbb{A} \mapsto \mathbb{S}$ the transition function and $\gamma \in \mathbb{R}$ is a discount factor. A state abstraction for an MDP is $< \mathbb{S}_\phi, \mathbb{A}, R, \hat{P}, \gamma >$, where the $\mathbb{S}_\phi$ is the abstract state space. Each abstract state includes a set of states. The $\hat{P} : \mathbb{S}_\phi \times \mathbb{A} \mapsto \mathbb{S}_\phi$ defines a transition function based on abstract states.

A key step to construct state abstraction is defining a state mapping function $\phi : \mathbb{S} \mapsto \mathbb{S}_\phi$ that maps a ground state to an abstract state. The function ϕ can be implemented by defining similarity between states and aggregating similar states to the same abstract state. Approximate MDP homomorphism [29] is a typical state similarity measurement. For two states s_1 and s_2, it is defined by the approximate error of reward function ϵ_R and the approximate error of transition function ϵ_T:

$$\epsilon_R(s_1, s_2) = \max_{a \in \mathbb{A}} |R(s_1, a) - R(s_2, a)| \tag{1}$$

$$\epsilon_T(s_1, s_2) = \max_{a \in \mathbb{A}} \sum_{s'_\phi \in \mathbb{S}_\phi} \left| \sum_{s' \in s'_\phi} T(s'|s_1, a) - \sum_{s' \in s'_\phi} T(s'|s_2, a) \right| \tag{2}$$

where $T(s'|s, a)$ is the transition probability, ϵ_R measures the maximal difference between reward functions of the given states and ϵ_T measures the worst-case total variation distance between state transition distributions.

D. Elastic Monte Carlo Tree Search

The elastic MCTS method [19] is built upon MCTS$_u$. It aggregates tree nodes with approximate MDP homomorphism. The constructed node groups are split into ground tree nodes with *early stop*. Algorithm 1 and Algorithm 2 (without the blue parts) provide pseudocode for elastic MCTS.

For every B MCTS iteration (line 6 in Algorithm 1), elastic MCTS checks all the tree nodes that have not yet been added in an abstraction node and calculates their approximate MDP homomorphism errors (line 8-9 in Algorithm 2). If the errors between the candidate state s_1 and an abstraction node $\hat{s}$ are below the pre-determined error thresholds η_R and η_T, s_1 is added into $\hat{s}$ (line 13). If there is no abstract node that matches this condition, a new abstract node is created with the s_1 as the only member node (line 15). The *early stop* shows in line 4-5 from Algorithm 1. It splits all abstract nodes into ground nodes once the MCTS iteration reaches an *early stop* threshold α_{ES}.

V. Method

Based on MCTS, our method automatically groups tree nodes by the approximate MDP homomorphism. Following Jiang et al. [14] and Xu et al. [19], SCSA groups tree nodes from the same layer at every batch (a fixed number of MCTS iterations). At each iteration, the MCTS samples one trajectory that consists of a sequence of nodes, starting from the root node to a leaf node. To approximate the MDP homomorphism, a batch of samples is required for calculating the approximate errors (Equation 1). Therefore, for every B iteration(s), SCSA checks every tree node that has not yet joined a node group to expand the current abstraction. There are two approaches for a node to be added to the existing abstraction, depending on the approximate MDP homomorphism errors. If the approximate errors between this candidate node and a node group are below the thresholds, this candidate node is added to the corresponding node group, becoming a *member node* of this group. When the approximate errors between the candidate node and all same-layer groups are found higher than the thresholds, a new node group is created and this node becomes the only member node.

It is found that a large number of samples are required to obtain high-quality state abstractions [14]. In strategy games where the search space are large, it is infeasible to obtain enough samples. Under limited samples, the constructed state abstraction might be of unstable quality. Moreover, it is difficult to evaluate the quality of the constructed state abstraction. Xu et al.[19] discovered that abandoning the existing state abstraction in the middle of MCTS running can bring significant performance improvement. In contrast to their approach [19], SCSA does not abandon the state abstraction. Instead, a global size constraint is defined to limit the maximum number of member nodes for every node group. Below, we introduce the abstraction construction in detail.

The pseudocodes of the SCSA algorithm are shown in Algorithm 1 and Algorithm 2, with highlighted lines in red representing the removed part from Elastic MCTS, and the lines highlighted in blue are newly introduced by the SCSA method. The computation budget constant is N_{FM}, meaning the maximum number of available forward model calls. In Algorithm 1, lines 6-7 presents the early stop with a threshold α_{ES} [19]. Our method removes this part.

We first introduce the hyperparameters, N_{FM} the computation budget, B the batch size, η_R reward function error, η_T transition error and *SIZE_LIMIT* the maximum node group size. In the beginning, the abstraction ϕ is initialized by mapping states to themselves. Within the computation budget (line 4), an MCTS iteration is run with the forward model cost c_{FM} and the current tree depth L returned (line 5). For every B iteration, the state abstraction is updated by calling the *ConstructAbstraction* function (Algorithm 2), after which the forward model call counter n_{FM} and the MCTS iteration counter n_{MCTS} are updated.

Algorithm 1 Elastic MCTS

1: **Require:** $N_{FM}, B, \eta_R, \eta_T, K,$ *SIZE_LIMIT*
2: **Initialize:** $n_{FM} = 0, n_{MCTS} = 0$
3: $\phi := s \rightarrow \hat{s}, \hat{s} = \{s\}$ # Initialize the abstraction
4: **while** $n_{FM} < N_{FM}$ **do**
5: $c_{FM}, L = MCTSIteration(\phi)$
6: **if** $n_{MCTS} > \alpha_{ES}$ **then**
7: $\phi := s \rightarrow \hat{s}, \hat{s} = \{s\}$
8: **else if** $n_{MCTS} \% B == 0$ **then**
9: $\phi = ConstructAbstraction(\phi, \eta_R, \eta_T, L,$ *SIZE_LIMIT*$)$
10: $n_{FM} = n_{FM} + c_{FM}$
11: $n_{MCTS} = n_{MCTS} + 1$

We next introduce the *ConstructAbstraction* function. Algorithm 2 iterates from the bottom of the tree to the root node, layer by layer (line 2). For each layer, all nodes that are not added to the abstraction are iterated (line 3). For a candidate node s_1, the algorithm iterates through all same-layer abstract nodes to consider accepting s_1 (line 5). Specifically, SCSA limits the maximal abstract node size. Therefore, if an abstract node is found exceeds the limit, this abstract node is skipped (line 6-7). Otherwise, the approximate errors between s_1 and each state from the abstract node is calculated (line 10-11). A node is added into an abstract node (line 14-15) only if the similarities between s_1 and all ground nodes from the abstract node are below the thresholds (line 9-13). If a node is finally find not added into any abstract node, a new abstract node is created (line 17-18).

VI. Experiments

Baselines: We implement 5 baseline agents to evaluate the performance of the SCSA agent. They are *Rule-based* , *MCTS*, *MCTS*$_u$, *RG MCTS*$_u$ and *Elastic MCTS*$_u$. Details about each agent are listed below:

1) *Rule-based* : Stratega platform has implemented a Rule-based agent for each game. We here briefly introduce their implementation. The Rule-based agent for *KTK* prioritizes attacking isolated enemy units and healing strong ally units. For each enemy unit, an isolation score is calculated considering its nearby ally units and enemy units. At each round, the Rule-based agent controls its units to i) approach the enemy units with the highest isolation score and attack them; and ii) approach an ally unit to heal it. The *PTA* Rule-based agent controls *pushers* to approach the nearest enemy unit and push it towards the nearest hole. In *TK*, the Rule-based agent first researches *Mining*, which is necessary for spawning

Algorithm 2 ConstructAbstraction

Require: $\phi, \eta_R, \eta_T, L,$ *SIZE_LIMIT*
for $l = L$ to 1 **do**
  for all node s_1 in depth l that is not grouped **do**
    $s_1\_in\_\phi =$ false
    for all abstract node $\hat{s}$ in ϕ **do**
      if $|\hat{s}| >$ *SIZE_LIMIT* ***then***
        break
      s_1_in_$\hat{s}$ = true
      for all node s_2 in $\hat{s}$ **do**
        $\epsilon_R = \max_a |R(s_1, a) - R(s_2, a)|$
        $\epsilon_T = \sum_{s'} |T(s'|s_1, a) - T(s'|s_2, a)|$
        if $\epsilon_R > \eta_R$ **or** $\epsilon_T > \eta_T$ **then**
          s_1_in_$\hat{s}$ = false, **break**
      if s_1_in_$\hat{s}$ == true **then**
        Add s_1 in abstract node $\hat{s}$
        $s_1\_in\_\phi = true$
    if $s_1\_in\_\phi ==$ false **then**
      $\phi(s_1) = \{s_1\}$ # Create a new abstract node

workers. Once the research is finished, a *worker* is spawned and is assigned the task of collecting gold from the nearest gold vein. These gold are used to spawn *warriors*. Once the number of *warriors* reaches 2, these *warriors* are sent to attack the enemy *king*. Whenever its *warriors* died, new *warriors* would be spawned if they had enough gold resources.

2) *MCTS*: An MCTS agent without using state abstraction.
3) *MCTS$_u$*: An MCTS agent with unit ordering.
4) *RG MCTS$_u$* : An MCTS$_u$ agent with randomized state abstraction. Each new tree node either joins an existing node group (with the probability $\frac{1}{N+1}$ for each group, supposing there are already N node groups) or creates a new node group with itself as the only *member node* (with the probability $\frac{1}{N+1}$).
5) *Elastic MCTS$_u$*: MCTS$_u$ with state abstraction based on approximate MDP homomorphism and *early stop* [19].
6) *SCSA*: An MCTS$_u$ agent with approximate MDP homomorphism abstraction. Each abstract node has a size limit defined by *SIZE_LIMIT*.

Heuristic functions: The same as typical MCTS in games, we utilize heuristic functions to evaluate states reached by the MCTS roll-out. The heuristic functions are game-specific. In each game, all agents except for the Rule-based agent share the same heuristic function. Below, we introduce the implementation details of the heuristic functions for each game.

In all games, the scores of states where the player wins, loses and draws the game are 1, -1 and 0, respectively. For all the other states, the heuristic function returns a score between 0 and 1. The *KTK* heuristic function returns a score of $R = 1 - \frac{d \cdot h}{D_{\text{ktk}} \cdot H}$, where the d is the sum of the distance from each ally unit to the enemy king, D_{ktk} is the maximum value of d, h is the health points of the enemy king and H is the maximum value of h. The strategy is controlling the units to approach the enemy king and try to search for a state that leads to victory.

For *PTA*, the score of a state is a sum of three parts. The first one is $0.2 \times \frac{\sum_u \min_{u'} dis(u,u')}{D_{\text{pta}}}$, where the u is an ally unit, u' is an enemy unit, $dis(\cdot,\cdot)$ returns the Euclidean distance between the two units. The second part is $0.4 \times \frac{|U_t|}{|U_0|}$, where $|U_t|$ is the number of alive ally units at time step t and U_0 is the number of the units in the beginning. The last part is $0.4 \times \frac{|U'_0| - |U'_t|}{|U'_0|}$, where $|U'_t|$, $|U'_0|$ are the number of enemy units at time step t, respectively.

In *TK*, the state score is calculated according to finishing a series of tasks. Finishing *Mining* research returns 0.2, having $worker$ alive returns 0.1 and having units that have action *attack* returns 0.1. Other scores include $0.1\times$ the distance of ally workers to its nearest gold vein, $0.2\times$ collected gold, $0.3\times$ the distance between all ally units and enemy units. The normalized score lands in $[0, 1]$.

TABLE I: Agent parameters for Section VI-B experiment

Agents	C	K	α_{ABS}	η_R	η_T	*SIZE_LIMIT*
Kill The King (KTK)						
MCTS	0.1	10	/	/	/	/
MCTS$_u$	1.0	10	/	/	/	/
RG MCTS$_u$	0.1	10	8	/	/	/
Elastic MCTS$_u$	0.1	10	10	0.05	1.0	/
SCSA	0.1	10	/	0.05	1.0	2
Push Them All (PTA)						
MCTS	10	10	/	/	/	/
MCTS$_u$	10	20	/	/	/	/
RG MCTS$_u$	0.1	10	4	/	/	/
Elastic MCTS$_u$	10	10	8	1.0	1.0	/
SCSA	10	10	/	1.0	1.0	2
Two Kingdoms (TK)						
MCTS	0.1	20	/	/	/	/
MCTS$_u$	1.0	20	/	/	/	/
RG MCTS$_u$	0.1	10	8	/	/	/
Elastic MCTS$_u$	1.0	20	6	0.05	1.0	/
SCSA	1.0	20	/	0.05	1.0	2

A. Agent Parameter Optimisation with NTBEA

As each agent has different optimal parameters for each game, we apply the N-Tuple Bandit Evolutionary Algorithm (NTBEA) [25] to automatically optimize agent parameters in different games. The NTBEA uses an N-Tuple system to break down the combinatorial space of the parameters. NTBEA has its own parameters, an exploration factor, the number of neighbours and the number of iterations. Following Xu et al. [19], these values are set to 2, 50 and 50, respectively. Next, we introduce the parameter space for each game-playing agent.

The parameters for MCTS and MCTS$_u$ are the exploration factor $C \in \{0.1, 1, 10, 100\}$ and rollout length $K \in \{10, 20, 40\}$. RG MCTS$_u$ has C, K and an *early stop* threshold $\alpha_{ES} \in \{4 \times B, 8 \times B, 10 \times B, 12 \times B\}$, where the $B = 20$ is a constant batch size. Elastic MCTS$_u$ has C, K, α_{ES} and approximate errors for reward function and transition function $\eta_R \in \{0.0, 0.04, 0.1, 0.3, 0.5, 1.0\}, \eta_T \in$

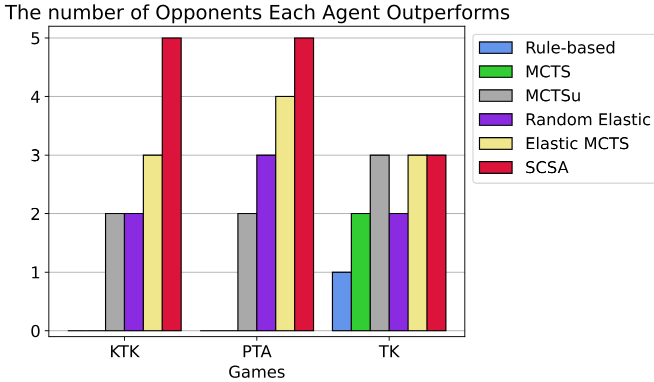


Fig. 4: The number of opponents that the agent outperforms.

$\{0.0, 0.5, 1.0, 1.5, 2.0\}$. For SCSA, we use the same parameters as Elastic MCTS$_u$. SCSA does not require the early stop threshold but requires a *SIZE_LIMIT*, which is linearly searched and set to 2 in the first group of experiments. The hyper-parameters tuned by NTBEA are shown in Table I.

Except for parameters, the budgets for search-based agents vary in different games. We set this budget based on the competitive performance of MCTS$_u$ playing against the corresponding Rule-based agent. In *KTK* and *PTA* the budget is set to $10,000$ number of forward model calls. In *TK*, the budget is $5,000$ forward model calls.

B. Performance on multi-unit-grid-based games

To evaluate the general performance of SCSA agents, we ran an experiment with agents playing against each other in a two-player manner. For each game, we set up 50 initial unit positions (randomly sampled). For each initial unit position, 2 evaluations are made by switching sides. Each evaluation is made with 5 random seeds, resulting in 500.

The general performance of each agent is shown in Figure 4. In KTK and PTA, SCSA outperforms all its opponents. In the more complex *TK* game, SCSA shows a competitive performance to Elastic MCTS$_u$.

The detailed win rates for each agent pair are shown in Tables II, III and IV. In *KTK*, the SCSA agent outperforms all its opponent agents in a significant gap. MCTS shows a weaker performance than the rule-based agent. MCTS$_u$ shows a stable and better performance than the MCTS. Elastic MCTS$_u$ outperforms both RG MCTS$_u$ and MCTS$_u$.

In *PTA*, the overall win rates are higher than *KTK*. In this game, the SCSA agent also outperforms all its opponents. Elastic MCTS$_u$ shows a strong performance in that it beats all agents except for the SCSA agent. MCTS$_u$ outperforms MCTS by a large margin. The MCTS agent is still weaker than the rule-based agent.

In *TK*, the Rule-based agent outperforms the MCTS significantly while other agents outperform Rule-based with large margins. In this complex game, MCTS$_u$, Elastic MCTS$_u$ and SCSA are showing close performances.

TABLE II: Win rates with standard errors for games *Kill The King*

Agent 1	Agent 2	Agent 1	Agent 2
1 King, 1 Archer, 1 Warrior and 1 Healer			
MCTS	Rule-based	$47.2(1.9)\%$	$52.8(1.9)\%$
MCTS$_u$	Rule-based	$\mathbf{62.2(1.1)}\%$	$37.6(1.3)\%$
RG MCTS$_u$	Rule-based	$\mathbf{63.4(1.0)}\%$	$36.6(1.0)\%$
Elastic MCTS$_u$	Rule-based	$\mathbf{54.2(1.4)}\%$	$44.8(1.6)\%$
SCSA (ours)	Rule-based	$\mathbf{55.6(0.6)}\%$	$44.4(0.6)\%$
MCTS$_u$	MCTS	$\mathbf{58.0(1.2)}\%$	$41.4(1.2)\%$
RG MCTS$_u$	MCTS	$\mathbf{61.6(0.9)}\%$	$38.4(0.9)\%$
Elastic MCTS$_u$	MCTS	$\mathbf{61.4(1.2)}\%$	$36.6(1.2)\%$
SCSA (ours)	MCTS	$\mathbf{58.2(1.9)}\%$	$30.2(1.7)\%$
RG MCTS$_u$	MCTS$_u$	$49.0(0.9)\%$	$51.0(0.9)\%$
Elastic MCTS$_u$	MCTS$_u$	$\mathbf{61.2(1.4)}\%$	$38.8(1.4)\%$
SCSA (ours)	MCTS$_u$	$\mathbf{53.8(1.7)}\%$	$38.8(1.7)\%$
Elastic MCTS$_u$	RG MCTS$_u$	$50.2(1.0)\%$	$49.8(1.0)\%$
SCSA (ours)	RG MCTS$_u$	$\mathbf{52.4(1.4)}\%$	$47.4(1.3)\%$
SCSA (ours)	Elastic MCTS$_u$	$\mathbf{49.2(2.1)}\%$	$38.8(1.9)\%$

In conclusion, these experiments verify the performance improvement brought by the unit ordering and Elastic MCTS. It also evaluates the performance of the SCSA agent, confirming its outstanding performance in all three games. In addition, it shows good performance of SCSA agents in different games can be achieved by the same value for *SIZE_LIMIT*. In this experiment, we show that 2 is an appropriate value for *SIZE_LIMIT*. Comparing different games, the SCSA agent performs less strongly in the more complex *TK* game, indicating a potential issue of scalability.

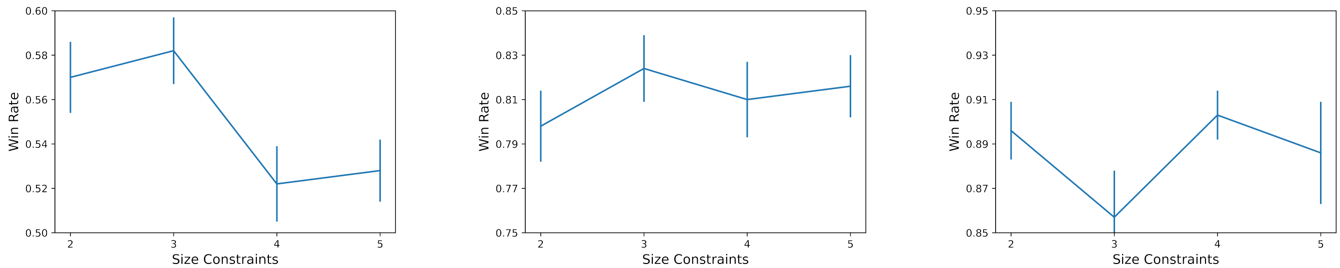
C. Influence of abstract state size

To better investigate the influence of different values for *SIZE_LIMIT*, we assigned different values from 2 to 5 and run the agent pair of SCSA - Rule-based agent. The same as in Section VI-B, 500 games are run for each value of *SIZE_LIMIT*. Figure 5a-5c shows the win rates with standard errors in three games. We observe the optimal *SIZE_LIMIT* values for *KTK* and *PTA* is 3 but *TK* has its optimal values at 2 and 4. We also observed that larger *SIZE_LIMIT* values (e.g. a value of 5) can degrade the performance, which reveals the trade-off between the tree size and the performance. With a larger *SIZE_LIMIT*, more groups are aggregated together therefore the tree size becomes smaller. However, a tree that is too small might cause performance degradation.

We used a value of 2 for all games and the agents showed satisfactory performance. Compared to different α_{ES} values are required in each domain (See Table V in Xu et al. [19]), the *SIZE_LIMIT* is less sensitive across domains.

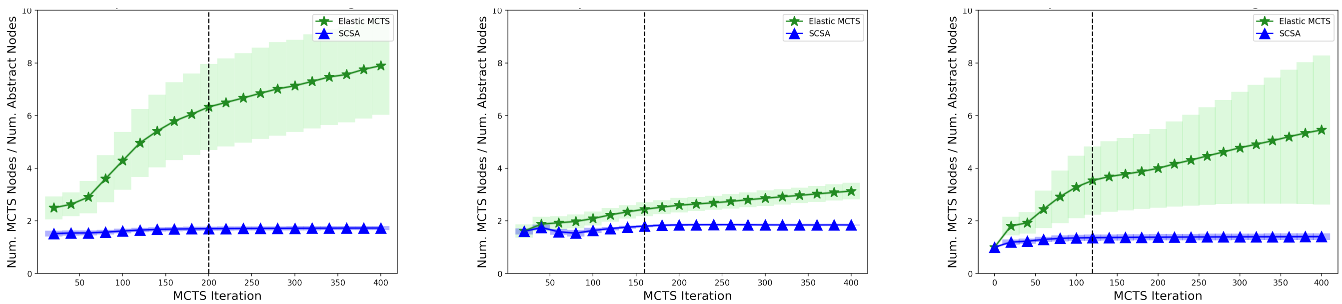
D. Compression Rate

To compare the influence of different state abstractions on tree size, we visualize compression rate at different MCTS iterations (see Figure 6a-6c). The compression rate is defined as the number of tree nodes dividing the number of abstract nodes. We can see that the *SIZE_LIMIT* has constrained



(a) Kill The King (*KTK*) (b) Push Them All (*PTA*) (c) Two Kingdoms (*TK*)

Fig. 5: Performance of the SCSA agent with different values for its size constraint. Results of the SCSA agent playing against the corresponding Rule-based agents are visualized, including win rates and standard errors.



(a) Kill The King (*KTK*) (b) Push Them All (*PTA*) (c) Two Kingdoms (*TK*)

Fig. 6: Compression rates for each tested game including standard errors from 10 game plays. The vertical line indicates the iteration when the abstraction is split into the original node for the Elastic MCTS$_u$.

TABLE III: Win rates with standard errors for games *Push Them All*

Agent 1	Agent 2	Agent 1	Agent 2
MCTS	Rule-based	48.8(1.6)%	51.2(1.6)%
MCTS$_u$	Rule-based	**69.0(1.1)**%	30.8(1.2)%
RG MCTS$_u$	Rule-based	**74.0(2.3)**%	26.0(2.3)%
Elastic MCTS$_u$	Rule-based	**81.8(0.9)**%	18.0(1.1)%
SCSA (ours)	Rule-based	**79.8(1.6)**%	20.2(1.6)%
MCTS$_u$	MCTS	**64.4(1.1)**%	33.6(1.4)%
RG MCTS$_u$	MCTS	**86.2(0.7)**%	12.0(0.7)%
Elastic MCTS$_u$	MCTS	**85.4(1.5)**%	13.0(1.7)%
SCSA (ours)	MCTS	**86.0(1.4)**%	12.8(0.9)%
RG MCTS$_u$	MCTS$_u$	**73.4(2.0)**%	25.6(1.9)%
Elastic MCTS$_u$	MCTS$_u$	**80.2(1.0)**%	18.0(1.0)%
SCSA (ours)	MCTS$_u$	**77.2(1.8)**%	22.0(2.0)%
Elastic MCTS$_u$	RG MCTS$_u$	**62.4(1.4)**%	35.8(1.3)%
SCSA (ours)	RG MCTS$_u$	**58.8(1.8)**%	40.4(2.0)%
SCSA (ours)	Elastic MCTS$_u$	**52.0(1.6)**%	46.8(1.9)%

TABLE IV: Win rates with standard errors for games *Two Kingdoms*

Agent 1	Agent 2	Agent 1	Agent 2
MCTS	Rule-based	12.6(0.5)%	**81.2(1.2)**%
MCTS$_u$	Rule-based	**90.8(1.7)**%	7.6(1.6)%
RG MCTS$_u$	Rule-based	**88.0(1.1)**%	11.8(1.2)%
Elastic MCTS$_u$	Rule-based	**89.2(1.4)**%	9.6(1.3)%
SCSA (ours)	Rule-based	**88.8(1.0)**%	9.8(1.3)%
MCTS$_u$	MCTS	**96.0(0.4)**%	4.0(0.4)%
RG MCTS$_u$	MCTS	**89.6(1.2)**%	10.4(1.2)%
Elastic MCTS$_u$	MCTS	**96.0(0.5)**%	4.0(0.5)%
SCSA (ours)	MCTS	**94.0(0.6)**%	6.0(0.6)%
RG MCTS$_u$	MCTS$_u$	43.6(1.6)%	**56.2(1.6)**%
Elastic MCTS$_u$	MCTS$_u$	52.2(2.1)%	47.6(1.9)%
SCSA (ours)	MCTS$_u$	49.0(1.8)%	50.8(1.8)%
Elastic MCTS$_u$	RG MCTS$_u$	**62.4(1.0)**%	37.6(1.0)%
SCSA (ours)	RG MCTS$_u$	**57.0(1.6)**%	43.0(1.6)%
SCSA (ours)	Elastic MCTS$_u$	51.4(2.7)%	48.6(2.7)%

the compression rate by limiting the maximal size of each abstract node. The overall compression rates of Elastic MCTS$_u$ are higher than SCSA and differ in different games. The vertical lines in Figure 6a-6c indicate the iteration when Elastic MCTS$_u$ drops state abstraction.

We observe from the plots that Elastic MCTS$_u$ without early stop can obtain a higher compression rate but this degrades the performance (See [19]). The SCSA agent outperforms Elastic MCTS$_u$ in two of the three games and it has lower compression rates. These observations reveal a trade-off between the abstracted tree size and the agent performance.

VII. Conclusion and Future Work

Automatic state abstraction has recently been applied to MCTS to address large search spaces in strategy game-playing. However, the lack of data results in state abstraction of unstable quality. We propose the novel SCSA to control the abstraction quality. Compared to the previous *early stop* approach, our method has a much smaller range for its hyperparameter. The empirical results on 3 strategy games of different complexity present the effectiveness of SCSA on strategy game playing.

The SCSA outperforms baselines in two games but not in the complex TK game, indicating a potential shortcoming of scalability. A possible solution is to combine state abstraction with pruning. We plan to further investigate the scalability of the SCSA agent in our future work.

Limitation We analyze the tree size under different state abstraction size constraints, revealing a trade-off between memory usage and agent performance.

References

[1] Oriol Vinyals, Igor Babuschkin, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[2] Christopher Berner, Greg Brockman, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[3] Hengyuan Hu, Adam Lerer, Alex Peysakhovich, and Jakob Foerster. "other-play" for zero-shot coordination. In *ICML*, pages 4399–4410, 2020.

[4] Brandon Cui, Hengyuan Hu, Luis Pineda, and Jakob Foerster. K-level reasoning for zero-shot coordination in hanabi. *Advances in NeurIPS*, 34:8215–8228, 2021.

[5] Chao Yu, Akash Velu, et al. The surprising effectiveness of PPO in cooperative multi-agent games. In *Advances in NeurIPS Datasets and Benchmarks Track*, pages 24611–24624, 2022.

[6] Hengyuan Hu, Adam Lerer, Brandon Cui, Luis Pineda, Noam Brown, and Jakob Foerster. Off-belief learning. In *ICML*, pages 4369–4379, 2021.

[7] DJ Strouse, Kevin McKee, Matt Botvinick, Edward Hughes, and Richard Everett. Collaborating with humans without human data. *Advances in NeurIPS*, pages 14502–14515, 2021.

[8] Anton Bakhtin, Noam Brown, et al. Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, pages 1067–1704, 2022.

[9] John Levine, Clare Bates Congdon, et al. General video game playing. In *Artificial and Computational Intelligence in Games*, volume 6, pages 77–83. 2013.

[10] Diego Perez-Liebana, Spyridon Samothrakis, et al. General video game ai: Competition, challenges and opportunities. In *AAAI Conference on Artificial Intelligence*, pages 4335–4337, 2016.

[11] Chiara F Sironi, Jialin Liu, et al. Self-adaptive mcts for general video game playing. In *International Conference on the Applications of Evolutionary Computation*, pages 358–375, 2018.

[12] Abdessamed Ouessai, Mohammed Salem, and Antonio M Mora. Improving the performance of mcts-based μrts agents through move pruning. In *IEEE CoG*, pages 708–715, 2020.

[13] Abdessamed Ouessai, Mohammed Salem, and Antonio Miguel Mora. Parametric action pre-selection for mcts in real-time strategy games. In *CoSECivi*, pages 104–115, 2020.

[14] Nan Jiang, Satinder Singh, and Richard Lewis. Improving uct planning via approximate homomorphisms. In *International conference on AAMAS*, pages 1289–1296, 2014.

[15] Jesse Hostetler, Alan Fern, and Thomas Dietterich. Sample-based tree search with fixed and adaptive state abstractions. *JAIR*, 60:717–777, 2017.

[16] Michael Chung, Michael Buro, and Jonathan Schaeffer. Monte carlo planning in rts games. In *CIG*, pages 117–124, 2005.

[17] Gabriel Synnaeve and Pierre Bessiere. A bayesian tactician. In *ECAI Workshops*, pages 1–13, 2012.

[18] Alexander Dockhorn et al. Game state and action abstracting monte carlo tree search for general strategy game-playing. In *IEEE CoG*, pages 1–8, 2021.

[19] Linjie Xu, Alexander Dockhorn, and Diego Perez-Liebana. Elastic monte carlo tree search. *IEEE Transactions on Games*, 15(4):527–537, 2023.

[20] Ankit Anand, Aditya Grover, Mausam Mausam, and Parag Singla. Asap-uct: Abstraction of state-action pairs in uct. In *IJCAI*, page 1509–1515, 2015.

[21] Ankit Anand, Ritesh Noothigattu, Parag Singla, et al. Oga-uct: On-the-go abstractions in uct. In *ICAPS*, pages 29–37, 2016.

[22] Hendrik Baier and Michael Kaisers. Guiding multiplayer mcts by focusing on yourself. In *IEEE CoG*, pages 550–557, 2020.

[23] Samuel Sokota, Caleb Y Ho, Zaheen Ahmad, and J Zico Kolter. Monte carlo tree search with iteratively refining state abstractions. *Advances in NeurIPS*, pages 18698–18709, 2021.

[24] Alberto Uriarte and Santiago Ontanón. Game-tree search over high-level game states in rts games. In *AIIDE Conference*, pages 73–79, 2014.

[25] Simon M Lucas, Jialin Liu, and Diego Perez-Liebana. The n-tuple bandit evolutionary algorithm for game agent optimisation. In *IEEE Congress on Evolutionary Computation*, pages 1–9, 2018.

[26] Alexander Dockhorn, Jorge Hurtado Grueso, et al. Stratega: A general strategy games framework. In *AIIDE Workshops*, pages 1–7, 2020.

[27] Cameron B Browne, Edward Powley, et al. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[28] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, pages 282–293, 2006.

[29] Balaraman Ravindran and Andrew G Barto. Approximate homomorphisms: A framework for non-exact minimization in markov decision processes. pages 1–10, 2004.