STRATEGA - A General Strategy Games Framework

Alexander Dockhorn, Jorge Hurtado-Grueso, Dominik Jeurissen, Diego Perez-Liebana
School of Electronic Engineering and Computer Science
Queen Mary University of London, UK

Abstract

Strategy games are complex environments often used in AI-research to evaluate new algorithms. Despite the commonalities of most strategy games, often research is focused on one game only, which may lead to bias or overfitting to a particular environment. In this paper, we motivate and present STRATEGA - a general strategy games framework for playing n-player turn-based and real-time strategy games. The platform currently implements turn-based games, which can be configured via YAML-files. It exposes an API with access to a forward model to facilitate research on statistical forward planning agents. The framework and agents can log information during games for analysing and debugging algorithms. We also present some sample rule-based agents, as well as search-based agents like Monte Carlo Tree Search and Rolling Horizon Evolution, and quantitatively analyse their performance to demonstrate the use of the framework. Results, although purely illustrative, show the known problems that traditional search-based agents have when dealing with high branching factors in these games. STRATEGA can be downloaded at:

https://github.research.its.qmul.ac.uk/eecsgameai/Stratega

1 Introduction

Since Michael Buro motivated AI research in strategy games (Buro 2003), multiple games and frameworks have been proposed and used by investigators in the field. These games, although different in themes, rules and goals, share certain characteristics that make them interesting for Game AI research. Most of the work done in this area pertains to the sub-genre of real-time strategy games (and, in particular, to Starcraft (Ontanón, Synnaeve, and others 2013)), but it is possible to find abundant research in other real-time and turn-based strategy games in the literature (see Section 2).

The complexity of these problems is often addressed by incorporating game domain knowledge into the agents, either by providing the AI with programmed game specific information or training it with game replays. Despite the contributions made this way can be significant, and in some cases it is possible to transfer algorithms and architectures from one game to another, we believe the time is right to introduce general AI into this domain. The objective of this paper is to present a new framework for general AI research in strategy games, which tackles the fundamental problems of this type of domains without focusing on an individual game at a time: resource management, decision making under uncertainty, spatial and temporal reasoning, competition and collaboration in multiple-player settings, partial observability, large action spaces and opponent modelling, among others.

In a similar way to General Game Playing (GGP) (Genesereth, Love, and Pell 2005) and General Video Game Playing (Perez-Liebana, Lucas, and others 2019), this paper proposes a general multi-agent, multi-action strategy games framework for AI research. Section 3 presents the vision for this framework, while Section 4 describes the current implementation. Our main interest when proposing this framework is to foster research into the complexity of the action decision process without a dependency on a concrete strategy game. The following list summarises the main characteristics of this platform:

- Games and levels are defined via text files in YAML format. These files include definitions for games (rules, duration, winning conditions, terrain types and effects), units (skills, actions per turn, movement and combat features) and their actions (strength, range and targets).
- STRATEGA incorporates a Forward Model (FM) that permits rolling any game state forward by supplying an action during the agent's thinking time. The FM is used by the Statistical Forward Planning (SFP) within the framework: One Step Look Ahead, Monte Carlo Tree Search (MCTS) and Rolling Horizon Evolutionary Algorithm (RHEA).
- A common API for all agents that provides access to the FM and a copy of the current game state, which supplies information about the state of the game and entities. This copy of the state can be modified by the agent, so what-if scenarios can be built for planning. This is particularly relevant to tackle partial observability in RTS games.
- The framework includes functionality for profiling the agent's decision-making process, in particular regarding FM usage. This facilitates analysis of the impact, in the execution of the game, of methods that use these models

(such as the ones included in the benchmark) by providing the footprint in time and memory of FM usage.

- Functionality for game and agent logging, in order to understand the complexity of the action-decision process faced by the agents and easily analyse experimental results. This information includes data at the game end (outcome and duration), turn (score, leading player, state size, actions executed) and action (action space) levels.

The aim of this framework is not only to provide a general benchmark for research on game playing AI performance on strategy games, but also to shed some light on how decisions are made and the complexity of the games. A common API for games and agents allows to build new scenarios and to compare different AI approaches in them. In fact, the second contribution of this paper is to showcase the use of the current framework. To this end, Section 5 presents baseline results for the agents and games already implemented in this benchmark.

2 Related Work

STRATEGA incorporates many features of well-known strategy games and GGP frameworks, described in this section.

2.1 Strategy Games

There is a relevant proliferation of multi-action and multi-unit games in the literature of games research in the last couple of decades, ranging from situational and tactical environments to fully-fledged strategy games. An example of the former is HeroAIcademy (Justesen, Mahlmann, and others 2017), where the authors present a turn-based game where each player controls a series of different units in a small board. Each turn, players distribute 5 action points across these units with the objective of destroying the opponent's base. The authors used this framework to introduce Online Evolutionary Planning, outperforming tree search methods with more efficient management of the game's large branching factor.

Later on, (Justesen et al. 2019) introduced the Fantasy Football AI (FFAI) framework and its accompanying Bot Bowl competition. This is a fully-observable, stochastic, turn-based game with a grid-based game board. Due to a large number of actions per unit and the possibility to move each unit several times per turn, the branching factor is enormous, reportedly the largest in the literature of turn-based board games. Its gym interface provides access to several environments each offering a vector-based state observation. While those environments differ in the size of the game-board, the rules of the underlying game cannot be adjusted.

Recently, (Perez-Liebana et al. 2020b) provided an open-source implementation of the multi-player turn-based strategy and award-winning game The Battle of Polytopia (Midjiwan AB 2016). In this game, players need to deal with resource management and production, technology trees, terrain types, partial observability and control multiple units of different types. The action space is very large, with averages of more than 50 possible actions per move, and an estimated branching factor per state of 10^{15}. The framework includes support for SFP agents, including MCTS and RHEA, which in baseline experiments seem to be at a similar level to rule-based agents, but inferior to a human level of play.

The most complete turn-based strategy games framework to date is arguably Freeciv (Prochaska and others 1996), inspired by Sid Meier's *Civilization* series (Firaxis 1995 2020). It incorporates most of the complexities and dynamics of the original game, allowing the interactions between potentially hundreds of players. Due to its complexity, most researchers have used it to tackle certain aspects of strategy games, like level exploration and city placement (Jones and Goel 2004) (Arnold, Horvat, and Sacks 2004).

Regarding real-time strategy games, microRTS (Ontañón et al. 2018) is a framework and competition developed to foster research in this genre, which generally has a high entry threshold due to the complexity of the game to be learned (e.g. Starcraft using BWAPI(Team 2020)). In comparison to other frameworks, players can issue commands at the same time and each action takes a fixed time to complete. The framework implements various unit and building types that act on the player's command. The framework supports both fully and partially observable states. Recently, AlphaStar (Vinyals et al. 2019) shows the great proficiency of deep supervised and reinforcement learning (RL) methods in Starcraft II. With the use of an important amount of computational resources, their system is able to beat professional human players consistently by learning first from human replays and then training multiple versions of their agent in the so-called AlphaStar League. This example show that even for complex RTS games it is possible to develop agents of high proficiency which so far remain limited to play a single game.

2.2 General Game Playing (GGP)

As previously mentioned, the goal of STRATEGA is to support research on general strategy game playing, which forms an interesting sub-domain of general game playing. GGP has already been supported by numerous frameworks that focus on its different aspects. The GGP framework (Genesereth, Love, and Pell 2005) was introduced to study general board-games and its game description language motivated the development of the video game description language (Schaul 2013) and its accompanying General Video Game AI (GVGAI) (Perez-Liebana, Liu, and others 2019) framework. GVGAI focuses on 2D tile-based arcade-like video games and supports a small number of 2D physics-based games. In a similar fashion, the Arcade Learning Environment (ALE) (Bellemare et al. 2013) provides access to Atari 2600 games, offering multiple ways in which game states can be perceived and is tightly interconnected with Open AI Gym (Brockman et al. 2016).

Different styles of defining games have been presented by the Ludii (Piette et al. 2019) and the Regular Boardgames (RBG) (Kowalski et al. 2019) frameworks. While the former uses high-level game-related concepts (*ludemes*) for game definitions, the latter uses regular expressions. Both permit defining turn-based games, but currently seem to lack methods for implementing real-time games. Finally, (Tian et al. 2017) propose the Extensive, Lightweight and Flexible (ELF) platform that allows the execution of Atari, Board and Real-time Strategy games. In particular, ELF incorporates MiniRTS, a fast RTS game with a similar scope to microRTS.

2.3 General Strategy Games

Some initial attempts have been made to provide platforms to host multiple RTS games. In one of the most recent works, (Andersen, Goodwin, and Granmo 2018) signified the need for an RTS framework that can adjust the complexity of its games and presented Deep RTS. This platform focused on providing a research benchmark for (deep) Reinforcement Learning methods, supported games of different complexity, ranging from low (such as those in microRTS) to high (as Starcraft II). A similar but more flexible framework is Stratagus (Ponsen et al. 2005), a platform that shares some characteristics with our proposal. Different strategy games can be configured via text files and LUA scripts can be used to modify some game dynamics. Some statistics are also gathered for all games, such as units killed and lost, and a common API is provided for agents.

Our general strategy games platform goes beyond these proposals in a two-fold manner. First, from the perspective of the agents, we provide forward model functionality to enable the use of statistical forward planning agents. Secondly, from the games perspective, our platform provides higher customisation of the game mechanics, allowing the specification of game goals, terrain features, unit and action types, complemented with agent and game logging functionality. Furthermore, the STRATEGA framework makes use of higher level concepts to ease development and customisation of strategy games. While GPP frameworks may be able to produce strategy games of similar complexity, they can require extensive effort to encode the games we are looking at.

3 Platform for General Strategy Games

STRATEGA currently implements n-player turn-based strategy games, where games use a 2D-tile representation of level and units (Perez-Liebana et al. 2020a). During a single turn, a player can issue one or more actions to each unit. While the standard game-play lets all players fight until all but one has lost all its units, the game's rules can be modified to implement custom winning conditions. At the game start, every player receives a set of units, which can be moved along the grid and attack other units to reduce their health. Furthermore, units can get assigned special abilities and differ in their range, health points, damage and other variables.

The framework is written in C++. It can run headless or with a graphical user interface (GUI) that provides information on the current game state, run-time statistics, and allows human players to play the game. The GUI, the game engine and game playing agents are separated in multiple threads to maximise the framework's efficiency and the developer's control over the agents. Figure 2 shows a screenshot of the current state of the framework. Isometric assets are included in the platform to depict different types of units and terrains, which can also be assigned via YAML configuration files. Figure 1 shows the overall structure of the framework.

3.1 Creating Games

The definition of all game components such as units, abilities, game modifiers, levels and tiles is done through YAML-

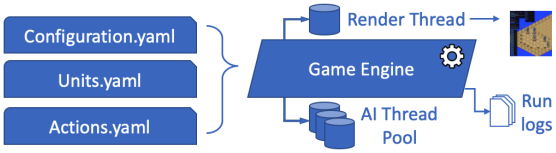

Figure 1: Overall structure of the framework.

Figure 2: Exemplary game state of STRATEGA and its GUI.

files. The excerpt of the **Action YAML-file** shown in Figure 3 shows the definition of an *Attack* action. Several properties can be configured and even new properties can be added to the framework. In the example, the attack action has a range of 6 tiles, damage of 10 (that can affect friendly units) and establishes if and how much score the attacker and attacked players receive when the action is executed.

The unit definition in the **Units YAML-file** shown in Figure 3 follows a similar pattern, allowing for hierarchically structured unit types. In our example, the *LongRangeUnit* is an extension of the *BasicUnit* type, inheriting the properties from its base. The entry defines basic properties for the unit: range of vision, movement, base attack damage, health and also the *Path* to its graphical asset. The actions available for this unit, as defined in the units YAML file, are indicated under the *Actions* heading. Two more attributes indicate the number of actions that the unit can execute during a turn (*NumberActionExecutePerTurn*) and if they can be executed more than once. Finally, *CanBeMoreThanOne* determines if this unit can be instantiated multiple times or is unique.

The **Configuration YAML-file** defines how a specific instance of a game should be created and played. The game rules section defines how many turns a game can have and if players or agents have a limited time budget to execute their turns. Game modifiers include (but are not limited to) turning on/off the fog of war or changing winning conditions. This eases the customisation by quickly modifying the game without changing and compiling its code, allowing to reuse the units and actions in different games. The Configuration YAML-file also specifies the list of $N \geq 2$ players and the level to play in. This level is formed by the initial distribution of the tiles and the definition of each terrain type. Figure 3

```
Actions:
 - Attack:
   Value: 10
   Range: 6
   GiveRewardAttacker: true
   AttackerReward: 2
   GiveRewardAttacked: false
   AttackedReward: -1
   CanExecutedToFriends: true
```

```
Units:
 - BasicUnit:
   NumberActionExecutePerTurn: 1
   CanRepeatSameAction: false
   Types:
    - LongRangeUnit:
      RangeVision: 6
      RangeMovement: 4
      RangeAction: 5
      AttackDamage: 70
      Health: 100
      Path: LongRange.png
      Actions:
       - Move
       - Attack
      CanBeMoreThanOne: false
```

```
Game Configuration:
  Game Rules:
    TimeForEachTurn: 10
    NumberOfMaxRounds: 100
  Players:
   - MCTS Player
   - RHEA Player
  Level: >
    XXXXXXXXXXXXX
    XX.........XX
    X.....T.....X
    XX.........XX
    XXXXXXXXXXXXX
  LevelNodes:
   - Trap:
     Character: T
     Walkable: true
     Cover: false
     HealthEffect: true
     EffectEntry: true
     KillUnit: false
```

Figure 3: Excerpts of YAML files that define the game. From top to bottom: actions, units, and general rules.

includes as an example the definition of a trap tile, which indicates that is walkable (units can enter the tile), does not offer any cover, deals 50 points of damage to the unit as soon as it enters the tile, but does not kill the unit automatically.

3.2 Agent interface

We provide an API for agent development and offer access to several sample agents (see Section 4.2). Agents must define a constructor for initialising the player and a method to indicate an action to execute in the current game tick. This method receives a copy of the game state, which can be modified (i.e. to incorporate game objects into the non-visible part of the level due to fog of war) and rolled forward supplying actions. This access to an FM facilitates the implementation of SFP agents. Agents have also access to the properties of the game state (positions of units, terrain tiles, game turn, score, etc.) and a pathfinding feature to determine shortest paths.

On each turn, the game requests an action from the player to be executed on the game. The agent receives information about all possible actions that can be executed in the current game state, for all the units present in the board. The agents can choose to return one of these actions, or a special action that indicates that the agent does not want to execute any more actions during the present turn. The game requests an action from the player as long as i) the agent does not return an *EndTurn* action; ii) there are still actions available for the player; and iii) the turn budget, if specified in the YAML configuration files, is not consumed.

3.3 Debugging and Logging

One of the most important aspects of this framework is the capability of analysing and logging game states and executions. Figure 2 shows live debug information by means of interactive floating windows. This information includes **game data** (current turn, number of actions executed by a player in a turn, frames per second, score and leading player), **profiling** (size - in bytes - of the game state and time - in microseconds - needed to copy the game state, advance the forward model by one action and the time taken by agents to provide a move to play) and **action** and **unit** information, which indicates the size of the action space in the current game state and the accumulated action space during the present turn. The interface also allows us to obtain more information and execute those actions from the list on the floating window, as well as obtaining information from the units in the game.

Once the game is over, a log file is also written in YAML format, including per-turn information on decision-making time, score, action space and actions executed, number of units, player rankings and specific game information. The framework includes simple scripts to analyse this data and produce logging plots as the ones shown below in Figure 4.

4 A Turn-based Strategy Framework

This section describes the implementation of the agents and 3 turn-based strategy games defined currently in the platform.

4.1 Games: Kings, Healers and Pushers

In **Kings**, players receive a king unit and a random set of additional units. Their task is to keep their king alive at all costs while trying to defeat the opponent's king. Similar to chess, losing other units does not determine the end of the game but effectively reduces the flexibility of a player. Four types of units are defined in this game mode, archer, warrior, healer, and the king. While the warrior moves slowly and deals high damage, the archer moves quickly, has long-range attacks, but its damage is reduced. In addition to its movement speed, the archer can also see further than any other unit in the game. The healer can restore other units'

health points. At last, the king can only move one square at a time but deals the highest damage. All units can move and attack once in the same turn. The game is played on a map with different types of terrains, each type provides a different cover-percentage for reducing incoming damage. Additionally, the map contains traps, which kill a unit upon entering. The map is covered in fog-of-war, with each unit revealing parts of the map based on its vision radius.

In the game **Healers**, both players have access to warriors and healers. The healer can move faster than the warrior but cannot attack. In comparison to Kings, healers and warriors have higher starting health points The twist in this game is that all units receive damage at the end of each turn. The goal of the players is to keep their units alive, while they can attack the opponent's units. The last player with units left wins. The map contains plains and mountains, this time with no tile providing coverage. Mountain act as a non-walkable obstacle and fog-of-war is disabled on this game.

The game **Pushers** is fundamentally different from the way the other games are played. Only 1 unit type is available, the *Pusher*. They cannot attack other units but can push them one tile back once per turn, to make the other player's units fall into the traps in the level. The agent's winning condition remains the same (survive the longest), but the game focuses on tactical movement instead of aggressive unit actions.

4.2 Agents

This section describes the different agents implemented in the framework and a heuristic used to evaluate game states.

Strength Difference Heuristic (SDH) SDH is a heuristic to estimate the relative strength of a unit, which estimates it as a linear sum of the unit's attributes (maximum health, attack damage, or movement range) divided by the maximum value among all available unit types. If a unit cannot execute an action, the corresponding attribute is 0. Note this heuristic will not change during a game, dynamic attributes like a unit's current health are not considered in the strength-estimation.

To estimate the value of a state, we compute the difference between the strength of the current player's units and the opponent's units. Additionally, a unit's strength is multiplied with the percentage of remaining health to encourage attacks.

Rule-based Combat (RBC) Agent This agent focuses on combat-oriented games like **Kings** or **Healers**. Its strategy is to focus all attacks on a single enemy unit while keeping its units out of danger. Every time the agent has to make a decision, it first targets an enemy unit. It then tests for each friendly unit if it can attack an opponent, heal an ally, or move closer to the target. Once a valid action has been found, the agent will execute it and repeat the process until no actions are left. The target is chosen based on an isolation-score. A unit's allies contribute negatively to the isolation score, while its enemies contribute positively. The contribution is equal to the unit's strength divided by the turns it takes for it to reach the unit. To find a target, the agent searches for an enemy with the highest isolation score. When attacking or healing a unit, the agent prioritises units with high-strength.

Rule-based Push (RBP) Agent The Push-Agent is highly specialised for games like **Pushers**. The agent's strategy is to push opponents into a direction that is closest to a death trap. For each unit, the agent computes the shortest paths from the unit to the adjacent tiles of the opponent's units. Each path then gets assigned a score equal to the length of the path, plus an estimate of how long it would take to kill the opponent from the tile. Starting from the path with the lowest score, the agent checks if following the path for one turn would result in a position that endangers the unit. If the path is not dangerous it is assigned to the unit, otherwise, the agent will try the next path. Once a unit was assigned a path, it will either push the target opponent or follow the path. Once a unit has moved or pushed, the agent will restart the process until no unit can act safely any more.

One Step Look Ahead (OSLA) Agent The OSLA agent uses the game's forward model to predict the upcoming state for each of the available actions. Resulting states are rated according to the SDH heuristic function. A high positive (resp. negative) score will be used in case the agent won (lost) the game after applying an action. Finally, the agent selects the action which yields the highest score.

Monte Carlo Tree Search (MCTS) Agent Over time, many variants of MCTS have been proposed for various problem domains (Browne et al. 2012). For the framework, we implemented a basic version of MCTS using the Upper Confidence Bounds (UCB) (Auer, Cesa-Bianchi, and Fischer 2002) selection criterion. The MCTS agent uses a tree node structure to facilitate a search through the game's state space. Each node stores an associated game state, a list of available actions, and a pointer to one child node per action. The tree is initialised by creating a root node using the provided game state. During each iteration of the search, the agent first selects a node, then expands it by another child node, further simulates a rollout, and ends with backpropagating its value on the path to the root node. A node is selected for expansion by step-wise going down the tree until a tree node which has not been fully expanded yet has been found. Each step, the child node with the highest UCB value is selected. The new child-node is generated by applying the associated action to the selected node's game state. During the tree policy we do not consider opponent turns, instead we skip them to avoid the non-determinism of their action selection. The new child node's value is determined by applying random actions until the end of the game or a predetermined depth. Its value is backpropagated through the visited nodes of the tree until the root. The search ends in case a maximum number of forward model calls has been reached. Finally, we return the root's child node with the highest visit count.

Rolling Horizon Evolutionary Algorithm (RHEA) Agent The Rolling Horizon Evolutionary Algorithm searches for an optimal action sequence with a fixed length (the horizon) (Perez-Liebana, Samothrakis, and others 2013). Therefore, it first generates a pool of candidate action sequences which is then continuously modified by an evolutionary algorithm. Each individual is created by step-wise selecting an action and applying it to the current game state. Afterwards,

the individual's value is determined using a provided heuristic. Similarly, to the MCTS agent, the RHEA agent skips the opponent's turn during rollouts since they introduced too much non-determinism in the evaluation of an action sequence. At the beginning of each iteration, tournament selection is applied to select the best individuals among a random subset of individuals. The generated pool is modified by mutation and crossover operators. During mutation, we iterate over an individuals action list and randomly choose to replace an action with a random one. Remaining actions are checked if they would still be feasible according to the given game state and, if not, replaced by a random feasible action. During crossover of two individuals, we randomly select which parent provides the next action. If the action is not applicable, it is replaced by a random feasible action. Resulting individuals are reevaluated and added to the next population. RHEA keeps iterating until a maximal number of forward model calls has been reached. Thereafter, the first action of the best-rated individual is returned.

5 Experimental Showcase

We tested the performance of the sample agents by running a round-robin tournament for each of the three games. We ran 50 games per match-up between the rule-based, RHEA, MCTS, and OSLA agents. During these 50 games, we have randomised 25 initial game states which were played twice, each player alternating their starting positions. The search-based agents were configured to use a budget of 2000 forward model calls (number of times the state is rolled forward) per selected action. For the RHEA agent, we used a population size of 1, individuals of length 5, and a mutation rate of 0.1. The MCTS agent was configured to use a rollout length of 3 and an exploration constant of $\sqrt{2}$. Both SFP agents skip the opponent's turn and only optimise the player's action sequence. OSLA, MCTS and RHEA use the Strength Difference Heuristic to evaluate game states. Games are run for a maximum of 30 turns, ending in a tie if no winner has been declared when reaching this number.

Table 1 summarises our results, reporting each agent's win rate per opponent and across all games. Results show that the RBC agent is very proficient in playing the game-modes Kings (avg. win-rate = 0.92) and Healers (0.82). While MCTS and RHEA agents were able to beat the OSLA agent, they were no match against the RBC agent. In contrast, the RBP agent did perform quite well against OSLA (1.00) and RHEA (0.74) but lost against the MCTS (0.46) agent.

The good performance of both rule-based agents shows that there is much room for improvement in terms of the performance of search-based agents. A great starting point to understand their problems is to analyse the game's complexity. Figure 4 shows two plots with the average size of the action-space over time and the number of actions executed per turn of 50 MCTS vs RHEA games in Kings. Both agents start with an average of 150 actions per move and execute between 5 and 6 moves per turn. The large fluctuation of the action-space size can be explained with the number of units that are still active in the agent's turn. After the unit count has been reduced, the size of both action spaces gets

Agents	RBC	OSLA	MCTS	RHEA	Average
	Kings				
RBC	—	1.00	0.86	0.90	0.92
RHEA	0.10	0.98	0.60	—	0.56
MCTS	0.14	0.92	—	0.12	0.39
OSLA	0.00	—	0.02	0.00	0.01
	Healers				
RBC	—	0.98	0.82	0.66	0.82
RHEA	0.34	1.00	0.70	—	0.68
MCTS	0.16	0.94	—	0.26	0.45
OSLA	0.02	—	0.06	0.00	0.03
	Pushers				
RBP	—	1.00	0.46	0.74	0.73
MCTS	0.54	1.00	—	0.30	0.61
RHEA	0.26	0.94	0.40	—	0.53
OSLA	0.00	—	0.00	0.00	0.00

Table 1: Winning rate by row player against column agent. Players sorted, per game, by overall higher winning average.

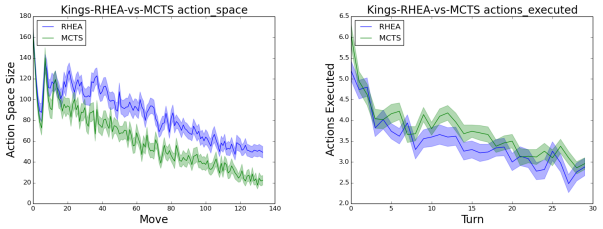

Figure 4: Logging: MCTS vs RHEA games in Kings.

gradually reduced, although RHEA's is always a bit higher. On the other hand, the number of actions executed per turn, although it also decreases with the game, is higher in RHEA. This shows that RHEA's higher winning rate is correlated with a more precise action selection that maintains a larger action space through the game.

6 Opportunities and Future Work

The goal of this framework is to allow research in the many different facets of Game AI research in strategical and tactical games, either turn-based or real-time. These include games that require a complex decision-making process, from multi-unit management to resource gathering, technology trees and long-term planning. Our aim is to provide a framework for i) search (showcased in this paper with SFP agents) and reinforcement learning agents; and ii) research in game and level generation, and automatic game tuning, which is made possible due to the definition of rules, mechanics, units and action via YAML files. The framework implemented in C++ aims to provide a much required high execution speed and interfaces for different programming languages for the implementation of agents and generators.

The current state of STRATEGA is fully functional for tactical turn-based games and SFP agents, and provides logging capabilities to analyse game results, as shown in this paper. It has been, however, developed with a road-map in mind

to incorporate extra games and logging features. Regarding the former, we plan to incorporate aspects of tactical role-playing games (object pick-ups, inventories, buff/debuffs etc.), technology and cultural trees, and resource and economy management, both for turn-based and real-time games. Regarding the logging features, the API will be enhanced so agents can log aspects of the internal representation of their decision-making process, following the example laid out in (Volz, Ashlock, and Colton 2015). This will provide a deeper insight into this task and also facilitate research into the explainability of the agent's decision-making process.

Finally, the agent's API and the highly customisable games allow tackling research on strategy games from a general game playing perspective, which is exemplified here by testing several agents in three different games implemented within the framework. Our intention is to propose this platform as a new competition benchmark in the near future.

Acknowledgements

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

References

Andersen, P.-A.; Goodwin, M.; and Granmo, O.-C. 2018. Deep RTS: a Game Environment for Deep Reinforcement Learning in Real-time Strategy Games. In *2018 IEEE conference on computational intelligence and games (CIG)*, 1–8.

Arnold, F.; Horvat, B.; and Sacks, A. 2004. Freeciv learner: a machine learning project utilizing genetic algorithms. *Georgia Institute of Technology, Atlanta*.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2/3):235–256.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: an evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47(1):253–279.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; et al. 2016. Openai gym.

Browne, C. B.; Powley, E.; Whitehouse, D.; et al. 2012. A Survey of Monte Carlo Tree Search Methods. *Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Buro, M. 2003. Real-time Strategy Games: A new AI Research Challenge. In *IJCAI*, volume 2003, 1534–1535.

Firaxis. 1995 – 2020. *Civilization*.

Genesereth, M.; Love, N.; and Pell, B. 2005. General Game Playing: Overview of the AAAI Competition. *AI magazine* 26(2):62–62.

Jones, J., and Goel, A. 2004. Hierarchical judgement composition: Revisiting the structural credit assignment problem. In *Proceedings of the AAAI Workshop on Challenges in Game AI, San Jose, CA, USA*, 67–71.

Justesen, N.; Uth, L. M.; Jakobsen, C.; et al. 2019. Blood Bowl: A new Board Game Challenge and Competition for AI. In *2019 Conference on Games*, 1–8. IEEE.

Justesen, N.; Mahlmann, T.; et al. 2017. Playing multiaction adversarial games: Online evolutionary planning versus tree search. *IEEE Transactions on Games* 10(3):281–291.

Kowalski, J.; Mika, M.; Sutowicz, J.; and Szykuła, M. 2019. Regular Boardgames. *Proceedings of the AAAI Conference on Artificial Intelligence* 33:1699–1706.

Midjiwan AB. 2016. *The Battle of Polytopia*.

Ontañón, S.; Barriga, N. A.; Silva, C. R.; Moraes, R. O.; and Lelis, L. H. S. 2018. The first microRTS artificial intelligence competition. *AI Magazine* 39(1):75–83.

Ontanón, S.; Synnaeve, G.; et al. 2013. A survey of real-time strategy game AI research and competition in StarCraft. *Trans. on CI and AI in games* 5(4):293–311.

Perez-Liebana, D.; Dockhorn, A.; Hurtado-Grueso, J.; and Jeurissen, D. 2020a. The Design Of "Stratega": A General Strategy Games Framework. *arXiv preprint arXiv:2009.05643*.

Perez-Liebana, D.; Hsu, Y.-J.; Emmanouilidis, S.; Khaleque, B.; and Gaina, R. D. 2020b. Tribes: A New Turn-Based Strategy Game for AI Research. In *2020 AAAI Advancement for the Artificial Intelligence in Digital Entertainment*, 1–8.

Perez-Liebana, D.; Liu, J.; et al. 2019. General Video Game AI: A Multitrack Framework for Evaluating Agents, Games, and Content Generation Algorithms. *IEEE Transactions on Games* 11(3):195–214.

Perez-Liebana, D.; Lucas, S. M.; et al. 2019. *General Video Game Artificial Intelligence*, volume 3. Morgan & Claypool Publishers. https://gaigresearch.github.io/gvgaibook/.

Perez-Liebana, D.; Samothrakis, S.; et al. 2013. Rolling horizon evolution versus tree search for navigation in single-player real-time games. In *Proceedings of GECCO*, 351–358.

Piette, E.; Soemers, D. J.; Stephenson, M.; Sironi, C. F.; Winands, M. H.; and Browne, C. 2019. Ludii–The Ludemic General Game System. *arXiv preprint arXiv:1905.05013*.

Ponsen, M. J.; Lee-Urban, S.; Muñoz-Avila, H.; Aha, D. W.; and Molineaux, M. 2005. Stratagus: An open-source game engine for research in real-time strategy games. *Reasoning, Representation, and Learning in Computer Games* 78.

Prochaska, C., et al. 1996. FreeCiv. http://www.freeciv.org/.

Schaul, T. 2013. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 1–8.

Team, B. D. 2020. The Brood War API (BWAPI) 4.2.0. https://github.com/bwapi/bwapi.

Tian, Y.; Gong, Q.; Shang, W.; Wu, Y.; and Zitnick, C. L. 2017. ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games. In *Advances in Neural Information Processing Systems*, 2659–2669.

Vinyals, O.; Babuschkin, I.; Chung, J.; Mathieu, M.; Jaderberg, M.; et al. 2019. Alphastar: Mastering the Real-time Strategy Game Starcraft II. *DeepMind blog* 2.

Volz, V.; Ashlock, D.; and Colton, S. 2015. 4.18 Gameplay Evaluation Measures. *Dagstuhl Seminar on AI and CI in Games: AI-Driven Game Design* 122.