Generally Genius: A Generals.io Agent Development and Data Collection Framework

Aaditya Bhatia[*], **Austin Davis**[*], **Soumik Ghosh**[*], **Gita Sukthankar**
Department of Computer Science,
University of Central Florida
4328 Scorpius Street,
Orlando, Florida, 32816-2362 USA
{aaditya.bhatia, austinleedavis, soumikghosh}@knights.ucf.edu, gita.sukthankar@ucf.edu

Abstract

We present an agent development and data collection framework for Generals.io (GIO)–a real-time strategy game with imperfect information in which players attempt to gain control of opponents' starting positions within a 2D grid world. The framework provides event-based communication amongst several modules implemented as microservices, enabling real-time data collection from GIO's streaming data. Its modular design facilitates rapid bot development and testing, while the emphasis on data collection makes it easy to analyze agent performance. We use this framework in a case study of a top-performing GIO agent called Flobot. Our analysis demonstrates that Flobot's performance varies based on its starting position. Based on the analysis performed with our framework, we propose a modification to Flobot's pathfinding algorithm. Statistical tests show that the new algorithm results in a significant reduction in performance variance.

Introduction

Real-Time Strategy (RTS) games provide a complex and challenging environment for AI systems. Traditional choices in this domain, such as StarCraft, exhibit an array of complexities such as game theoretic counter-play, partial observability, long- and short-term planning, real-time decision making, and large action spaces (Ontañón et al. 2013). These games increase the complexity even further by requiring players to manage resources, micromanage units, and understand the complex interplay between multiple species or factions. Taken together, these characteristics impose a steep learning curve for both the players and those desiring to advance AI research.

In an effort to reduce this barrier to entry while retaining the substantial complexity inherent in RTS games, we turned our attention towards *Generals.io* (GIO). GIO is a fast-paced, web-based, real-time strategy game of conquest where players maneuver their armies to conquer their opponents' starting locations. With a reduced action set, GIO can be learned in mere minutes, yet it still elicits strategic complexity akin to that found in other popular RTS games like StarCraft; the depth of strategic possibilities make GIO a promising platform for AI research delivered in a much more accessible setting.

Originally released in 2016, GIO has a large audience boasting 115k monthly visitors during the first quarter of 2023 (SimilarWeb 2023). GIO was adopted by the U.S. Army Captains' Career Course to assist officers in learning principles of warfare such as mass, maneuver, surprise, and economy of force. The game is free to play and is hosted online at *https://generals.io*, where players compete in either a one-on-one or free-for-all format. GIO also supports AI-vs-AI matches on a separate, dedicated bot server (*https://bot.generals.io*). The online format allows players to play in competitive ladders where Elo ratings (Elo 1967) can be used to compare players' relative skill levels.

Still, several factors exacerbate efforts to build a thriving research community around AI development for the game of GIO. First, is the lack of documentation for the bot server's API. Second, is that most previously released agents are either no longer available or use deprecated (or defunct) software packages. Third, there does not currently exist a well-designed end-to-end framework for designing, building, and deploying agents to the GIO bot servers. Finally, the lack of data collection make it difficult for researchers to collect performance statistics for their agents.

Our contributions are the Generally Genius (GG) framework, a comprehensive and user-friendly bot development and data collection framework for GIO. The GG framework can be found on our website[1]. The GG framework uses an event-based architecture and a key-value store database to collect and store game data. This framework provides a straightforward interface for creating, testing, and analyzing AI agents within the game. We designed it with accessibility and flexibility in mind, catering to researchers at all levels of proficiency. Furthermore, we implemented a robust data capture system to facilitate the collection and analysis of gameplay data. This system records in-depth gameplay statistics, enabling researchers to study game dynamics and agent performance in unprecedented detail. The captured data provides invaluable insights into RTS game strategies, AI behavior, and agent learning processes.

[*]These authors contributed equally.

[1]Further details about the GG Framework are available at https://CorsairCoalition.github.io

Background

GIO is played on a 2D grid world with four tile types: starting tiles, impassible mountain tiles, land tiles, and city tiles. Players maneuver their armies throughout the map to wrest control of the starting tiles from their opponents. When a player loses control of their starting tile (represented by a crown icon in Figure 1) all their territory and half their remaining armies are reassigned to their conqueror, and the player is eliminated from the game. The player who conquers all starting tiles wins.

Players maneuver their armies in the grid by submitting commands to a movement queue. One queued movement is executed each game tick (every half second), limiting total movement by a player to at most 2 tiles per second. Players can move their armies along the four cardinal directions (up, down, left, and right) into any tile not obstructed by impassible terrain. When a player moves an army from one tile to another, they must leave behind part of their army to "defend" the originating tile. Thus, movement is only possible if an army has a strength of at least two. As a result of this mechanic, when armies move across the map, they leave a snail-like trail of occupied tiles, typically with only a single unit to defend.

When an attacking army (with strength s_A) moves into territory occupied by a defending army (with strength s_D), then ownership of the tile is determined according to the following rules:

1. If $s_A \geq (s_D + 2)$, the attacker conquers the tile, but loses s_D units in the process. So, $s_A \leftarrow (s_A - (s_D + 1))$
2. Otherwise, $s_D \leftarrow s_D - (s_A - 1)$ and $s_A \leftarrow 1$. In this case, the defender retains control of the tile. As a result, whenever $s_D = (s_A - 1)$, the occupied tile of a defender can have an zero army strength after the attack.

Territorial control is a key concept in GIO. A player's army grows over time based on the type and number of tiles they control. Each city and starting tile owned by a player gains one unit per second (2 game ticks); all other conquered land tiles only gain a single troop at the end of each 25-second round (50 ticks). As such, cities (which start the game with a sizeable strength near 45) become an important source of recruitment in the late game.

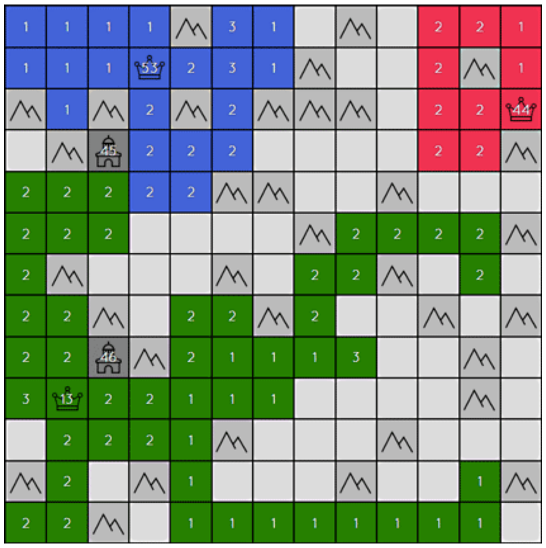

Figure 1: Screenshot of a typical *GIO* game. This particular map is 13×13 tiles wide and consists of three players: blue, red, and green, with starting tiles (indicated by crown icons) at coordinates $(4, 2)$, $(13, 3)$, $(2, 10)$, respectively from the top-left corner. Green player controls the most territory, having sent multiple excursions across the map (evidenced by the long trail of 1's emanating from their base), whereas red player has adopted a turtle strategy.

Strategies

The game mechanics described above gives rise to many complex interactions and strategies that have evolved in the player and bot communities. Viable GIO strategies follow similar principles to those from other strategy games such as chess, but with key differences because GIO is played in real-time under imperfect information. GIO players must nonethless adapt their strategies to suit early, mid, and late game and the variations in map terrain. A screenshot captured during the early-game of a free-for-all match is shown in Figure 1.

In the early game (tick 1-100, i.e., the first 50 seconds on standard speed), players typically focus on territorial expansion to maximize their tile recruitment on tick 50 and tick 100. Their armies often explore in tentacle-like fashion from their starting tile (represented by crowns in Figure 1), spreading their armies to capture adjacent tiles. This tentacle spreading phase typically occurs after a brief 24-tick buildup and aims to maximize the tile-based recruitment earned on turns 25 and 50. If armies are spread in too many directions during this phase, free-for-all players risk revealing their position to multiple opponents (see the Green player in Figure 1). The goal in this phase is to discover a single opponent and begin amassing forces for an attack.

Over-expansion can leave a player exposed during any phase, but it is especially risky during the mid-game. During this phase, players must balance between exploration of the map and exploitation of an opportunity. Projecting forces to far-reaching positions on the map can leave a player's starting tile exposed (cf. green's crown strength vs. its opponents' in Figure 1). Overemphasizing rapid territorial expansion or city conquest (represented as dark-gray cities in Figure 1) requires the player to spend long periods reconstituting forces. Furthermore, when one player expends a significant amount of units to capture a city, it often leaves them in a weakened state, making them vulnerable to attacks from other players. Since only the original attacker bears the burden of paying the unit cost to capture the city, newly captured cities become enticing targets for opportunistic opponents seeking to take advantage of the depleted resources and weakened defenses. Thus, players must balance both the timing and the opportunity cost of capturing cities. Scrimmages are frequent during this phase of the game as players search for their enemy's starting tile. Once an enemy general is exposed, players must concentrate their forces to capture it. Interestingly, having a large army does not necessarily help if the general is poorly defended. Thus, this phase requires a mix of tactical and strategic thinking.

By the late game (roughly any turn beyond 100), army recruitment primarily comes from starting tiles and/or city control, and total army strength can reach into the thousands. This makes control of a map's natural choke points less valuable in this stage than in the mid-game. As in many other grid-based strategy games, control of the map center is critical in the late-game because it enables a player to rapidly reinforce their units and easily access any position on the board. Combat against non-city tiles is typically conducted to clear the fog of war, and although conquering cities can affect unit recruitment rates, the net impact is diminished due the massive size of the armies.

Related Work

Generic Frameworks

Although agent-based machine learning techniques have been possible for many years, advances in RL-based neural architectures and GPU acceleration have contributed to a surge in interest in recent years. As such, many tools have been released to assist the community in designing, developing, testing, and deploying agents in both real-world and simulated, game-based environments. The aim of these tools is varied. For example:

- Dopamine (Castro et al. 2018) from a team of Google employees focuses on fast-prototyping of RL agents.
- TF-Agents (Guadarrama et al. 2018), also from Google employees emphasizes production ready deployment of RL agents.
- RLLib (Moritz et al. 2017) from the Ray project emphasizes ML software scalability for the Python community.
- Keras-RL (Plappert 2016) provides high-level implementations for many popular RL algorithms.
- TRFL (Hessel et al. 2018) (pronounced "truffle") is Google Deepmind's open-source extension to Tensorflow focused on low-level implementations of RL algorithms.
- Horizon (Gauci et al. 2019) is Facebook's PyTorch-based framework for off-policy, model driven RL with deep learning.
- Coach (Caspi et al. 2017) by Intel uses Keras to provide a highly modularized collection of DL and RL algorithms with support for Kubernetes deployment.
- MAgent2 (Zheng et al. 2018) is an engine for high performance multi-agent environments with very large numbers of agents, along with a set of reference environments.

Developers of these frameworks face many trade-offs, but they commonly aim to provide modularity, class hierarchies and abstractions, implementations of popular algorithms, mechanisms for performing data collection and/or data analysis, and standardized environments. These frameworks provide many communities the tools needed to accelerate agent development and testing while also improving repeatability, reliability, scalability, and transparency.

Our framework delivers several of these features to the GIO community: it emphasizes modularity, usability, and data collection.

GIO Agent Development

GIO bot developers face several obstacles. First, is a lack of documentation. An official API and a rudimentary JavaScript library are available to support bot development, but the official documentation does not cover the protocol in sufficient detail. So, bot developers must reverse engineer the communication protocol over the WebSocket connection. Second, many of the bots which have been open-sourced have not been maintained and require significant experience with JavaScript to bring up-to-date. Third, there are very few extant online resources, tools, or tutorials that can be used to bootstrap a development effort. These factors have become barriers to entry for researchers and enthusiasts interested in developing bots for GIO.

Because of the limitations above, only a few AI agents have been released publicly. AlphaGenerals (Sayers and Li 2017) employs behavioural cloning with data augmentation built on a CNN to build a policy that mimics player actions. The GeneralsNet (XBattleFan 2017) bot and the A3C (Du 2017) bot both used a convolutional policy network to play GIO, although A3C used a custom simulated environment during training rather than the official GIO servers. The reigning champion among publicly-available bots is Flobot (Lan 2017), originally released in 2017.

Flobot is a simple, rule-based agent which consistently ranks in the top two on the competitive bot ladder owing to several simple, yet effective strategies: *early-game*, *mid-game*, *spread*, *infiltrate*, and *end-game*.

- The *early-game* strategy handles the first fifty game ticks. It waits 25 game ticks to build a sizeable army on its general, then projects them toward the center of the map to clear fog. This branching behavior occurs twice: once on tick 25 and once on tick 50.
- The *mid-game* strategy focuses on reinforcing Flobot's general by collecting its movable armies on its general tile.
- The *spread* strategy occurs every 25 ticks, when territorial reinforcements are awarded. Flobot uses the extra armies on its border tiles to move forward, slowly pushing back the fog of war.
- The *infiltrate* strategy is activated once an enemy tile is discovered and before the enemy general is revealed. Infiltration finds a path of least resistance to explore the enemy's territory.
- The *end-game* strategy activates once the enemy general is revealed. It computes the army strength required to conquer the enemy general (even accounting for tick-based reinforcements) and collects its armies on Flobot's own general. Once sufficient forces are amassed, it sends them directly toward the enemy general. This strategy is repeated until the enemy general falls.

Methodology

Our goal was to build a framework for GIO that facilitates agent development and evaluation. To that end, we present the GG framework, a collection of services and utilities that enable researchers to develop and analyze AI agents for competitive GIO.

Architecture

We designed a bot framework from ground up using the microservices architecture. It is composed of a set of services that communicate over the Internet using a message broker. While the framework is language agnostic, the majority of the code was written in TypeScript to maximize code reuse and benefit from a rich package library and advanced development tools. We drew our inspiration from Robot Operating System (ROS) (Berger and Wyrobek 2021) and used an entirely event-based architecture for maximum efficiency. Our framework includes the following components:

1. **Core Strategy Module**: This module processes all game data, maintains the state, synchronizes all other modules, coordinates and receives action recommendations, makes decisions of selecting the best action, and instructs the IO module to communicate the game moves with the GIO servers. This module can generate strategic recommendations based on reinforcement learning, heuristics, or simple rule-based play. When it receives a set of recommended moves, it scores them based on the priority and confidence level of the recommender, and selects the action with the highest scores.
2. **Action Generators**: A collection of scripts (turtle, spread, explore, mass, capture, and attack) that receive game state updates and independently generate recommended moves. These scripts resemble some of Flobot's strategies, and their modular design them to serve as a starting point for GIO agent strategy development.
3. **Message Broker**: We chose Redis (Sanfilippo 2009) as a message broker and a database for our framework due to its lightweight implementation and extremely fast response. Redis is an open source, in-memory data store that enables low-latency communication at scale. It provide the communication backbone for all components and enable the event-driven architecture through publisher/subscriber based messaging channels and a built-in database.
4. **IO Module**: A server component that provides a link between the game server and our message broker. Once initialized, it connects to the game server and awaits commands on Redis to join a game. It tracks high-level game state events, such as game started, lost, won, but simply relays detailed state updates without making any game-related decisions.

This distributed architecture allowed us to write language-agnostic modules that interacted through a common message-passing Redis backend. It simplified the debugging process by making it easy to isolate issues, test individual components, and make incremental updates.

Name	Type	Update Frequency
width	int	Once
height	int	Once
size	int	Once
playerIndex	int	Once
usernames	string[]	Once
teams	int[]	Once
ownGeneral	int	Once
enemyGeneral	int	Once
ownQuadrant	int	Once
enemyQuadrant	int	Once
turn	int	Every Turn
armies_map	int[]	Every Turn
terrain_map	int[]	Every Turn
cities_map	int[]	Every Turn
armies_enemy	int[]	Every Turn
armies_self	int[]	Every Turn
land_enemy	int[]	Every Turn
land_self	int[]	Every Turn
moveCount	int[]	Every Turn
ownTiles	int[]	Every Turn
enemyTiles	int[]	Every Turn
discoveredTiles	int[]	Every Turn
maxArmyOnTile	int[]	Every Turn
enemyGeneralLocated	bool	Once
victory	bool	Once
totalMoves	int	Once

Table 1: List of data collected by the GG framework

Data Collection

Our GG framework automates many downstream tasks for rapid development of AI agents including detailed logging, comparison experiments, organizing and searching experiment metadata. All together, these capabilities improve agent development, debugging, reproducibility, collaboration, and model sharing. During execution, our framework collects several summary and detailed metrics listed in Table 1. All data is available in real time and also persisted to the Redis database for later analysis.

Development Tools

We developed the following additional components to support bot development and data analysis:

1. The event-based communication protocol is published on GitHub
2. A playback utility to record game events on demand and play them back on Redis as needed to emulate on-going games and conduct real-time analysis.
3. Jupyter notebooks capture ongoing game events, aggregate data, and facilitate analysis and visualization of the gathered data.

We have containerized the entire environment using docker to bundle our services in one unit, provide a ready-to-go environment for new developers, and reduce the complexity of setting up a distributed environment for development and testing. This is essential for our architecture that involves running multiple components as stand-alone programs that should not be exposed over the Internet without proper security measures. Together, these tools lower the barrier to entry and enable less experienced developers to begin AI agent development.

Evaluation

In this section, we demonstrate the effectiveness of our framework to analyze agent in-game performance. To this end, we present a case study where we analyze the performance of the well-known Flobot AI agent. Based on this analysis, we proposed and implemented modifications to the Flobot decision-making processes. We then evaluate the impact of these modifications on agent performance. The insights driving these modifications were only made possible due to the metrics gathered by the GG framework, and the modification process was greatly accelerated due to GG framework's modular design.

Case Study

In GIO, terrain and player starting locations are randomly generated. Both of these factors can have a major impact on the outcome of the game. For instance, players who start in an area surrounded by many mountain (impassible) features typically struggle to spread their armies in the early game; this often leads to a power differential that cannot be overcome. There are many other variations that can influence a game's outcome (e.g., city placement, choke point defensibly, access to the map center, exposure to multiple opponents).

In this case study, we investigated the 1-on-1 performance of Flobot based on its starting location. Specifically, we subdivided the GIO map into four quadrants, the top left (TL), top right (TR), bottom left (BL), and bottom right (BR). The goal of this experiment was to determine if Flobot favored starting in a particular quadrant. The null hypothesis (H_0) for this experiment was that Flobot is invariant to the starting configuration on the map, i.e., that Flobot would perform equally well regardless of its starting position. The alternative hypothesis (H_a) is that Flobot's performance varies depending on the starting qaudrant.

To test this hypothesis, we competed Flobot against itself in 102 games, recording the quadrant on the map in which Flobots starting tile was placed and relative position of the opponent. Each game took approximately 1-minute to complete, but since these were mirror-matches, we could investigate the data from both sides' perspectives, i.e. this produced 204 outcomes for our sample data set. We compared the win rates of Flobot across these samples based on the starting tile's quadrant. The results are summarized in the Table 2 below.

Table 2 summarizes how Flobot's performance varied across different starting configurations. Specifically, we found that Flobot performed best when it started in the left side of the map, with a win rate of 64% when in the top-left quadrant and a win rate of 67% when starting in the bottom-left quadrant. When Flobot began the game in the right side of the map, the win rates dropped considerably. It achieved

	Left	Right
Top	0.64	0.40
Bottom	0.67	0.29

Table 2: Flobot Win Rate by Starting Quadrant

a 40% win rate from the top-right quadrant and Flobot's win rate was only 29% from the bottom-right quadrant.

We performed a Pearson's chi-square (χ^2) goodness of fit test (Pearson 1900) on the categorical data to evaluate how likely it is that the observed differences arose merely by chance. The test returned $\chi^2 = 21.32$ as the test statistic with $df = 3$, giving us a p-value of 9.02×10^{-5} Thus, we reject the H_0 because the data provides support for the alternative hypothesis that Flobot's performance does depend on its starting quadrant. We hypothesized that the difference in performance across the quadrants was due in part to a bias introduced during pathfinding.

Flobot uses the A-star algorithm with an admissible heuristic. Therefore, the paths Flobot selected were guaranteed to be optimal in terms of movement cost; however, since movement costs are equal for all tiles in GIO, there are many paths that are equally optimal (see Figure 2). We observed that Flobot always added adjacent tiles to its pathfinding algorithm's open queue in the same order, namely {up, right, down, left}. This selection mechanism biases Flobot's path selection toward those paths that had many contiguous moves in the up and right direction as compared to the down and left direction (i.e., P_1 is favored over P_2 in Figure 2).

When starting in the bottom-right quadrant, Flobot's open tile selection algorithm favored moves that explored the top-right quadrant, ignoring the left half of the game map. We theorized that the bias in Flobot's selection process put it at a disadvantage whenever its starting tile was located on the bottom right side of the map.

In order to test our theory, we adjusted the algorithm that

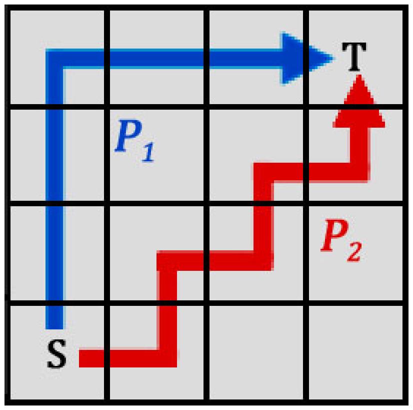

Figure 2: Two *optimal* paths from the starting tile, S, and terminating at the target tile, T. Path P_1 was generated using Flobot's biased tile selection, and path P_2 is an alternative optimal path. Flobot was biased to select P_1, despite the fact that P_2 provides better exploration of the center of the map.

Algorithm 1: *RotatingFlobot* Tile Expansion Sorting

```
GETORDER($x$, $y$, $mapWidth$, $mapHeight$):
    $quadrant \leftarrow 0$
    **if** $y > height$ **then:**
        $quadrant \leftarrow quadrant + 2$
    **if** $y > width$ **then:**
        $quadrant \leftarrow quadrant + 1$
    **switch** $quadrant$ **do:**
        **case** 0:
            **return** {right, down, left, up}
        **case** 1:
            **return** {down, left, up, right}
        **case** 2:
            **return** {left, up, right, down}
        **case** 3:
            **return** {up, right, down, left}
```

populates the open queue of the A-star algorithm so that it cycles the order in which adjacent tiles are added. We call the agent which uses this new unbiased tile selection algorithm *UnbiasedFlobot*.

We played 100 mirror-matches between *UnbiasedFlobot* and *UnbiasedFlobot*. The null hypothesis of this experiment (H_1) is that *UnbiasedFlobot*'s win rate does not depend upon the starting quadrant, with the alternative hypothesis (H_b) being that the starting quadrant does affect the bot's win rate. The results of our second experiment are summarized in Table 3.

	Left	Right
Top	0.46	0.44
Bottom	0.54	0.59

Table 3: *UnbiasedFlobot* Win Rate by Starting Quadrant

Once again, we performed a Pearson's χ^2 goodness of fit test on the data to determine the probability the observed differences in win rates between quadrants could be explained by chance. The test returned $\chi^2 = 3.086$ as the test statistic ($p = 0.3785$) given $df = 3$. So, we fail to reject H_1. Thus, by changing the algorithm which controlled the order in which tiles are added to A-star's open queue, we no longer observed a bias in performance based on the starting tile's quadrant.

Since the original Flobot performed best on the bottom-left quadrant due to its bias toward up-right movement (i.e., toward the opposite corner), we theorized an algorithm which adapts its tile selection priority according to its starting tile quadrant would be able to exploit the fixed bias of Flobot's, especially in the bottom right quadrant (where Flobot is weakest).

To test this theory, we developed a new agent, dubbed *RotatingFlobot*, which analyzes its starting tile's quadrant at the beginning of the game and uses that information to adjust its tile selection prioritization. The pseudocode for this algorithm is given in Algorithm 1 below where x and y represent *RotatingFlobot's* starting tile coordinates.

Agent	**Flobot**	**Unbiased**	**Rotating**
Opponent	Flobot	Unbiased	Flobot
Match Type	Mirrored	Mirrored	1v1
Sample Size	204	200	160
TL win rate	66%	50%	57%
TR win rate	36%	50%	46%
BL win rate	52%	48%	65%
BR win rate	16%	52%	49%
p**-val** (χ^2)	0.00168	0.9929	0.3785
WinRate	44%	50%	53%
p**-val (Binom)**	–	–	0.38

Table 4: Summary of Case Study Settings and Results

To test *RotatingFlobot*'s performance, we competed it against the original Flobot and recorded the results using our framework's data management tools. The two bots competed in 160 games during which we recorded the outcome and starting quadrants for each bot. We performed a Pearson's χ^2 test against the null hypothesis (H_2) that there was no statistical difference in performance across the quadrants. Given the test statistic $\chi^2 = 3.086$ and three degrees of freedom, we failed to reject H_2 ($p = 0.3785$), i.e., we saw no statistical evidence that *RotatingFlobot*'s performance varied across its starting quadrant.

We then evaluated *RotatingFlobot*'s performance against Flobot with regards to win rates. RotatingFlobot won 86 games during the 160 matches vs Flobot (a 53.75% win rate). We performed a binomial test to determine if the proportion of wins by RotatingFlobot deviates from the theoretical expectation given the two agents are evenly matched. Here, our null hypothesis (H_3) is that the proportion of wins for RotatingFlobot are equal to Flobot's. Given $n = 160$ and $k = 86$, our binomial test gave a p-value of 0.385, thus we fail to reject H_3; there is no evidence of a difference in win rates between RotatingFlobot and Flobot.

To be clear, this does not necessarily mean the two bots are equals, only that our tests failed to detect a statistical difference in their win rates. Since our algorithm represents a relatively minor change in comparison to the numerous other factors that could affect a game's outcome, it would likely require a test with much higher statistical power (i.e., many more samples) to detect a difference.

The results of our case study are summarized in Table 4.

Conclusion

In this paper, we introduced the Generally Genius (GG) framework, a comprehensive agent development and data collection framework for Generals.io (GIO). We designed the GG framework from the ground up, leveraging an event-based microservices architecture and drawing inspiration from the Robot Operating System (ROS) (Berger and Wyrobek 2021) inter-module communication. This language-agnostic framework supports bot development in over 100 programming languages using simple API calls. Altogether, these design choices facilitate rapid bot development and testing through reusable action modules, a common protocol for real-time updates, and persistent access to historical game data. By providing a robust platform for agent development and data collection, the GG framework significantly enhances the process of developing and testing AI game agents in GIO.

We gave a detailed description of GIO and characterized the status quo for agent development in GIO. We described the GG frameworks architecture and the improvements it provides over extant development tools for GIO, and the data collection components it includes. We released an implementation of one of the top performing agents in GIO, Flobot, using the framework's modular design.

By leveraging the framework's data collection components, we conducted a case study with an in-depth analysis of Flobots performance versus its board starting position. We identified variations in Flobot's performance across the four quadrants on the game board, and confirmed these variations were not due to random chance via a Pearson's χ^2 goodness of fit test (p-value: 1.68×10^{-3}). We theorized the variation in performance was introduced by a bias in Flobot's implementation of the A-star algorithm used to perform pathfinding. We presented an unbiased selection mechanism and showed that it eliminated this bias. We concluded the case study by presenting a third bot, $RotatingFlobot$, which aims to exploit the bias in the original Flobot's tile selection algorithm. However, statistical tests on win rates could not confirm $RotatingFlobot$ outperformed Flobot in 1-on-1 matches.

References

Berger, E.; and Wyrobek, K. 2021. Robot Operating System. https://ros.org/. Accessed: 2023-05-22.

Caspi, I.; Leibovich, G.; Novik, G.; and Endrawis, S. 2017. Reinforcement Learning Coach. https://doi.org/10.5281/zenodo.1134899. Accessed: 2023-05-26.

Castro, P. S.; Moitra, S.; Gelada, C.; Kumar, S.; and Bellemare, M. G. 2018. Dopamine: A Research Framework for Deep Reinforcement Learning. arXiv:1812.06110.

Du, Y. 2017. Generals A3C. https://github.com/yilundu/generals_a3c. Accessed: 2023-05-26.

Elo, A. E. 1967. The proposed USCF rating system. Its development, theory, and applications. *Chess Life*, 22(8): 242–247.

Gauci, J.; Conti, E.; Liang, Y.; Virochsiri, K.; Chen, Z.; He, Y.; Kaden, Z.; Narayanan, V.; and Ye, X. 2019. Horizon: Facebook's Open Source Applied Reinforcement Learning Platform. *ICML 2019 Workshop RL4RealLife*.

Guadarrama, S.; Korattikara, A.; Ramirez, O.; Castro, P.; Holly, E.; Fishman, S.; Wang, K.; Gonina, E.; Wu, N.; Kokiopoulou, E.; Sbaiz, L.; Smith, J.; Bartók, G.; Berent, J.; Harris, C.; Vanhoucke, V.; and Brevdo, E. 2018. TF-Agents: A library for Reinforcement Learning in TensorFlow. https://github.com/tensorflow/agents. Accessed: 2023-05-26.

Hessel, M.; Martic, M.; de Las Casas, D.; and Barth-Maron, G. 2018. Open sourcing TRFL: a library of reinforcement learning building blocks. https://www.deepmind.com/blog. Accessed: 2023-05-26.

Lan, F. 2017. Flobot. https://github.com/flo-lan/generals.io-Bot. Accessed: 2023-04-12.

Moritz, P.; Nishihara, R.; Wang, S.; Tumanov, A.; Liaw, R.; Liang, E.; Paul, W.; Jordan, M. I.; and Stoica, I. 2017. Ray: A Distributed Framework for Emerging AI Applications. *CoRR*, abs/1712.05889.

Ontañón, S.; Synnaeve, G.; Uriarte, A.; Richoux, F.; Churchill, D.; and Preuss, M. 2013. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5(4): 293–311.

Pearson, K. 1900. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302): 157–175.

Plappert, M. 2016. Keras-RL. https://github.com/keras-rl/keras-rl. Accessed: 2023-05-26.

Sanfilippo, S. 2009. Redis In-memory Data Structure Server. https://redis.io. Accessed: 2023-05-22.

Sayers, T.; and Li, S. 2017. AlphaGenerals: A Generals.io AI Agent. https://github.com/TySayers/generals-io-bot. Accessed: 2023-05-26.

SimilarWeb. 2023. Generals.io traffic analytics & market share. https://www.similarweb.com/website/generals.io. Accessed: 2023-04-12.

XBattleFan. 2017. Generals Net. https://github.com/XBattleFan/generals-net. Accessed: 2023-05-26.

Zheng, L.; Yang, J.; Cai, H.; Zhou, M.; Zhang, W.; Wang, J.; and Yu, Y. 2018. MAgent: A many-agent reinforcement learning platform for artificial collective intelligence. In *Thirty-Second AAAI Conference on Artificial Intelligence*.