The Design and Implementation of Modern Online Programming Competitions

Benjamin Spector, *Student, MIT,* and Michael Truell, *Student, MIT,*

September 25, 2018

***Abstract*—This paper presents a framework for the implementation of online programming competitions, including a set of principles for the design of multiplayer games and a practical framework for the construction of the competition environment. The paper presents successful competitions based on these games: the 2016-17 Halite challenge and the 2017-8 Halite II challenge. It also briefly previews new ideas for the September 2018 Halite III challenge.**

***Index Terms*—Programming competitions, AI, AI games, computational games, game design, Halite, Two Sigma, software engineering.**

I. Introduction

ONLINE programming competitions are a popular way for programmers to learn, collaborate, and relax. For schools and universities, they are a useful immersive tool to motivate and educate students. [1] For companies, they are a route to gain publicity and recruit new programmers as they signal a fun, tech-centered culture.[1] In a research context, they serve as de facto benchmarks of algorithm performance [2]–[4].

Much work has been done to formalize the design and implementation of robust, human-playable games. For example, Crawford's famous book from 1980's [5] tackled the topic. Later authors, like Salen, have further contributed to human-playable game design formalization [6].

This paper serves to formalize a set of requirements and characteristics of a good game-based programming competition. This framework is the Effective Programming Competition Framework (EPCoF). For the purposes of this work, a programming competition consists of 1) the game itself, which contestants write programs to play, and 2) the surrounding competition infrastructure. Our EPCoF framework outlines a set of design requirements for these two distinct components of a programming competition

EPCoF is based on the authors' experiences with a series of a successful, turn-based strategy games called Halite. The design and implementation of the Halite games has served to hone the definition of EPCoF and also to demonstrate its utility. Halite launched publicly in the fall of 2016 [2], had more than 1600 finishers, and received hundreds of contributions from an engaged open source development community. A second competition, called Halite II and also developed consistently with EPCoF, launched in October 2017 [3] and received about fourfold the usage of the first competition. The third installment in Halite series is expected to launch in September of 2018.

[1] https://www.youtube.com/watch?v=oxAPXN5UKa8
[2] https://2016.halite.io
[3] https://halite.io

II. Effective Programming Competition Framework

This section presents a framework for the design of a programming competition. This includes both the principles underlying a good programming game, as well as the principles for developing the surrounding programming competition environment.

A. Principles of Programming Game Design

The EPCoF principles for the design of the programming game are as follows:

1) The game should be simple, intuitive, and easy to start to play. Few programmers will spend hours learning complex rules and setting up an environment; the game need to be easy to understand and set up in under ten minutes, regardless of platform or user background.
2) The game should be visually appealing in order to draw in players and minimize the bounce rate from the website. Making games easy to view and understand also makes it more straightforward for competitors to view games and better their bots, which improves the user experience.
3) The logic of the game should allow for a runtime environment that is computationally inexpensive. Because there can be hundreds of thousands to millions of games, games should be quick to run even on lightweight servers.
4) The game should be as difficult as possible to perfectly solve. Little can kill a competition more thoroughly than a player discovering a simple dominant strategy.
5) The game should always have a path forward for users to improve their bot so that they stay engaged. Whether a user is just beginning to set up a local game environment or vying for the leaderboard's diamond slots, they should feel there is always something to be made better about their bot, because users who feel they have reached a dead end are much less likely to continue to participate. Concretely, this means that the game's strategy must be multi-faceted, so that once a player has optimized an aspect of their bot as best as they can, they can come back to it later and turn to the next one. However, it's important that this not be achieved by adding complexity,

such as through additional unit types, which not only make the game less easily understood but also more difficult to balance.

Each of these principles aims to make the game broadly and enduringly appealing.

B. Principles of Programming Competition Construction

The EPCoF principles for the design of the competition environment (the website, global leaderboard system, local game implementation, development tools, etc) are as follows:

1) The competition environment should be beginner-friendly. This should be in conjunction with the game design; just as the game must be simple to learn, the platform in which the game is programmed, debugged, and run must be simple to use.
2) The competition platform must be secure to minimize the possibility of abuse. This is challenging because competitions will typically allow players to upload their bots' executables into a shared environment.
3) The competition platform must easily scale with the number of contestants. This may involve techniques such as ranging from load balancing and automatic scaling of server capacity to systems for dealing with waves of OSS contributions.
4) The competition should provide real-time standings of contestants in order to show players their progress and encourage their continued interest.
5) The competition platform should encourage collaboration and community. Collaboration can take many forms, but due thought needs to be given to forums, social media, and the appropriate use of shared code repositories.
6) The competition must allow users to easily develop and test their bots. Speed of development, turn-around time in debugging, and flexibility of programming language use are all important objectives.

III. The 2016-17 Halite Competition

The inaugural Halite competition initially focused on its specific game. Through various iterations over the course of about 12 months and with successively larger pilot usage, the game itself converged and more attention became focused on having a competition environment that could support thousands of users. Early on, the authors decided to develop Halite in an open source manner hosted on GitHub, a valuable decision that allowed the user community to contribute enormously to the success of the game – particularly in increasing support for the game to about a dozen programming languages. Halite I served as a basis for refining the EPCoF framework, but also became a good case study of its utility.

A. The Halite I Game

1) Rules: Halite I is a turn-based strategy game in which between two and six players participate. Halite I is played on a wrap-around (toroidal) rectangular grid, and each player has the objective of eliminating all other players from the grid.

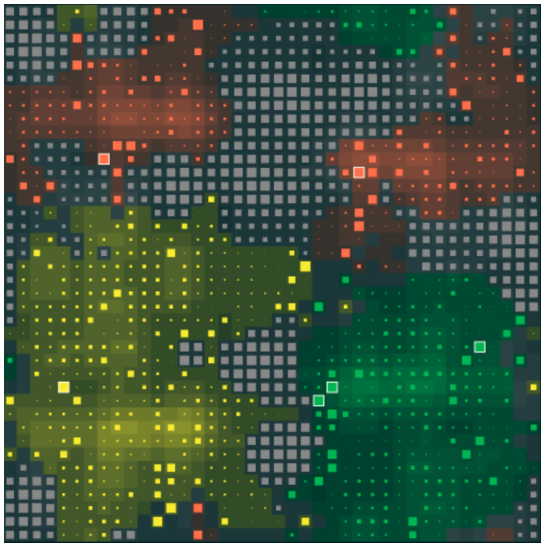


Fig. 1. A snapshot of the game through its visualizer. In this game, the orange player has appreciably less territory or strength than its opponents, but considerably greater production, represented by the brighter regions of the background map.

Each player starts with a single piece owned on the grid, and expands their territory by capturing unowned or enemy-owned sites. With every turn, each player is sent the complete Halite I game state and is expected to return their set of moves for their pieces. The game engine then executes all moves simultaneously. Halite I maps range from 20x20 to 50x50 and usually take several hundred turns to finish.

During a turn, each piece may be moved in one of the four cardinal directions or remain still, and piece additionally has an integer strength associated with it (this strength is capped at 255; any excess strength is wasted). When a piece is ordered to be still, its strength is increased, and when a piece moves, it leaves behind a piece with the same owner and a strength of zero. When two or more pieces from the same player try to occupy the same site, the resultant piece has a strength equal to the sum of their strengths. However, when pieces with different owners move onto the same site or cardinally adjacent sites, the pieces must fight, and each piece loses strength equal to the strength of its opponent. When a player's piece moves onto an unowned site, it will lose an amount of strength equal to the strength of the unowned piece, and take the site if it remains alive. Additionally, each piece does all of its attacks before it takes any damage, meaning it can damage multiple pieces simultaneously. When a piece loses all of its strength, it dies and is removed from the grid.

The last element of Halite I is its maps: every site on the map has associated with it a starting strength and a value called production, which defines how fast owned pieces grow when they are are ordered to remain still. Not all sites on a Halite I map are created equal – a site with a small strength and high production is very valuable, as it costs little to obtain but rapidly produces strength, whereas a site with high strength and low or no production may not ever be worth taking. Halite I maps are randomly generated using a custom algorithm designed to create interesting ones, and no two games are

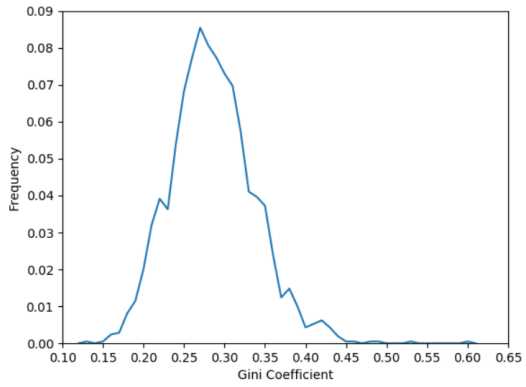


Fig. 2. A plot of the frequency of Gini coefficients (rounded to the nearest 0.01) in a random sample of 2095 Halite games.

ever played on the same map.

2) Analysis: Fundamentally, playing Halite I involves optimizing and balancing four major objectives:

- The bot must effectively expand its territory. This requires choosing the territory towards which the bot will move while balancing the gaining of production as well as the preservation of bot strength. This turns out to be a surprisingly interesting optimization problem.
- The bot must strategically take territory from other bots. A bot must choose how much strength to devote to attacking an enemy. If a bot devotes too little, it will lose on those fronts and quickly be overrun, but if it devotes too much, it will be less effective at expanding elsewhere and will also lose.
- A bot must be tactically effective in combat, maximizing damage inflicted on an opponent, as well as minimizing damage taken from said opponent. Additionally, a bot may want to occasionally sacrifice some additional strength in order to break through a front, because damage done in this way becomes much harder to contain.
- A bot should be efficient in building up its strength. Because pieces only gain strength when stationary, this entails ordering them to be still as often as possible. However, due to the 255 strength cap, the flow of pieces to the borders of the bot can often become clogged if there is too little movement. This may lead to great inefficiencies within the bot.

In addition to optimizing for those core aspects of Halite I, bots ought to deal with other characteristics and emergent behaviors of the game to rise in the rankings. There are too many to list here, so some selected features are summarized below:

Halite's map generator is itself an element of the game and produces a variety of interesting maps, challenging users to build versatile and adaptive bots. One way of characterizing a map is by the distribution of production throughout the map. Some maps have their production fairly evenly distributed throughout the map, whereas others have virtually all of it concentrated in specific regions of the map. An easy way to numerically characterize this is to borrow the Gini coefficient from economics, where a coefficient of 0 corresponds to a perfectly even distribution and a coefficient of 1 represents a single site having all of the production of the map. Figure 2 shows a coefficient distribution centered at a approximately 0.27-0.28, but with considerable variation. This indicates that a Halite bot should be optimized for that class of maps, but still have the versatility to deal with other types as well.

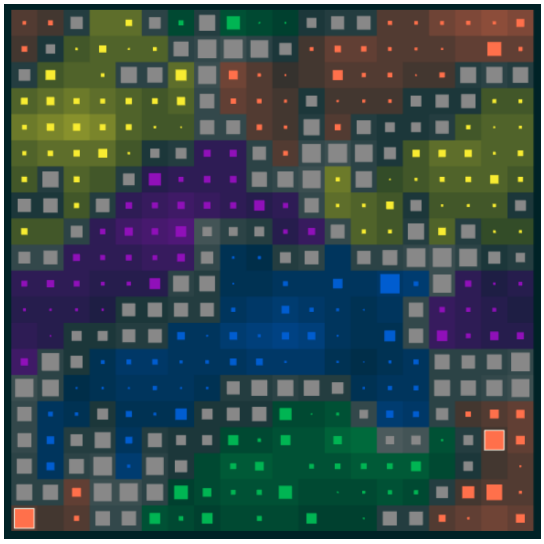


Fig. 3. A snapshot of a game in which all players are obeying the non-aggression pact.

The non-aggression pact (NAP) is an emergent phenomenon that was discovered in the final weeks of the competition and exploded during the last few days. It states that in a multiplayer game, bots should always let an opponent be the first to attack them. When playing only with players not following the NAP, this has essentially no effect because very little happens in Halite I during any single turn. However, when two or more NAP-enabled bots participate in the same match, each is helped because they make peace with each other and do not fight until they are among the last bots remaining (as may be seen in figure 3), meaning they can direct more strength at their other opponents. This turns out to be a highly significant effect, and was responsible for a great deal of volatility in the rankings during the last few days of the competition.

3) EPCoF Compliance: Halite I satisfies the EPCoF requirements outlined above to a great degree.

First, it is a remarkably uncomplicated game. Players just send one simple action to every piece, every turn, and the complexity of the game arises from the interactions of the rules rather than the rules themselves.

Second, Halite I follows good principles for visual design. Because the game is played on a grid, the games are easy to visualize, and make use of bright primary colors to draw in players. Additionally, the continuous nature of players' territories makes it easy to see the tempo of the game, allowing details to be ignored and the big picture to be seen.

Third, Halite I is indeed very fast to compute. The game engine computes the results of each turn in $O(n)$, where n is the number of tiles on the map, and the constant factor is very small, too. Because there were never problems due to the speed of the unoptimized C++ (for it was still very fast),

the code was not further improved, though a user, arjunvis, rewrote and optimized the environment in MATLAB in order to create his bot and reported a simulation speed of 20,000 turns per second.

Fourth, Halite I is essentially unsolvable. Its maps introduce considerable variety to the game, and the branching factor for Halite I may be on the order of 10^{1743} for the largest maps, making any sort of exhaustive tree search unattainable. This is also made more difficult by the fact that it is not straightforward to reliably evaluate the value of a Halite I board for a player. Heuristics, such as the global sum of each player's territory, strength, and production, are relevant but not decisive, because a great number of other factors go into correctly evaluating a board. It may be possible to use evolutionary strategies to generate move-sets as described in Justesen et. al. [7], but to our knowledge no competitors used this strategy because direct policy is much simpler to execute and very effective at playing Halite I– an aspect that made the game more fun to play.

Last, Halite I encourages iterative bot improvement. Because each of the four major objectives of a bot can be achieved in many ways, optimized independently or together, and each objective consists of dozens of sub-problems, there is always a path in sight to a better Halite bot.

For the record, considerable experience was needed to get the game design right. The game designed evolved several times while trying to meet this specific requirement.

B. The Halite I Competition Environment

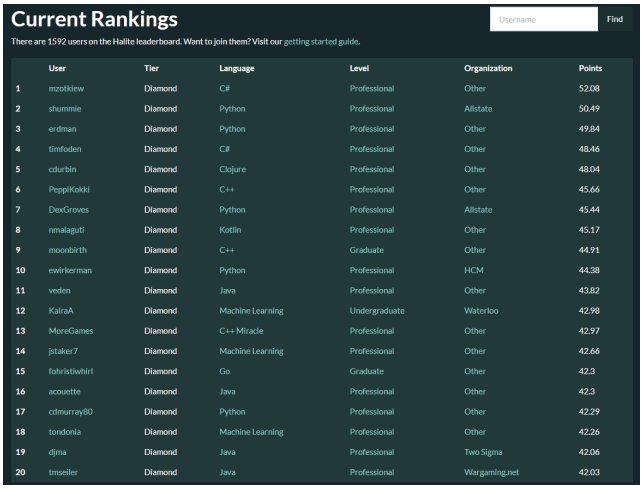
This section discusses the technical and user-oriented aspects of the Halite competition and its compliance with EPCoF.

Users built their bots using the programming language of their choice locally using the Halite game environment software. Once users were happy with their bots, they submitted their source to the competition website. Our game servers would compile their source, run the resulting binary against other bots, and continually publish rankings on a global leaderboard. Replays of all server-side games were made publicly available through the site, and they remain available. Throughout the competition, users could make improvements to their bots and resubmit at their leisure. The winner of the competition was the user who led the leaderboard at the end of the three months.

1) Bot Development: Halite satisfied the first requirement of EPCoF, namely that users can easily build bots for the competition.

Users were provided with a local game binary that would execute Halite game logic and serve as the intermediary between competing bots. Bot programs communicated with this game environment using pipes, allowing the environment to interface with any programming language.

Users were provided with small libraries for Java, C++, and Python to get them started with building a bot. These libraries, called "starter packages," consisted of an implementation of the environment's pipe networking protocol, simple game object representations, and a dummy Halite bot that moved all



	User	Tier	Language	Level	Organization	Points
1	mzotkiew	Diamond	C#	Professional	Other	52.08
2	shummie	Diamond	Python	Professional	Allstate	50.49
3	erdman	Diamond	Python	Professional	Other	49.84
4	timfoden	Diamond	C#	Professional	Other	48.46
5	cdurbin	Diamond	Clojure	Professional	Other	48.04
6	PeppiKokki	Diamond	C++	Professional	Other	45.66
7	DexGroves	Diamond	Python	Professional	Allstate	45.44
8	nmalaguti	Diamond	Kotlin	Professional	Other	45.17
9	moonbirth	Diamond	C++	Graduate	Other	44.91
10	ewirkerman	Diamond	Python	Professional	HCM	44.38
11	veden	Diamond	Java	Professional	Other	43.82
12	KalraA	Diamond	Machine Learning	Undergraduate	Waterloo	42.98
13	MoreGames	Diamond	C++ Miracle	Professional	Other	42.97
14	jstaker7	Diamond	Machine Learning	Professional	Other	42.66
15	fohristiwhirl	Diamond	Go	Graduate	Other	42.3
16	acouette	Diamond	Java	Professional	Other	42.3
17	cdmurray80	Diamond	Python	Professional	Other	42.29
18	tondonia	Diamond	Machine Learning	Professional	Other	42.26
19	djma	Diamond	Java	Professional	Two Sigma	42.06
20	tmseiler	Diamond	Java	Professional	Wargaming.net	42.03

Fig. 4. A snapshot of the final rankings of top Halite participants. The columns can be clicked to sort. Note that Halite II has devoted considerable focus to making it more natural for users to list an organizational affiliation.

of its game pieces randomly (see the above Rules section). The game environment exposed a flexible command line interface, which allowed users to build their own tools for automating parts of the development process. For example, one user created a script to rank his various local bots according to the Trueskill algorithm [8], in the manner of Halite's central leaderboard.

2) Ranking: Throughout the entirety of the Halite competition, users could follow a public ranking showing their bots performance. Users submitted their local bot source code to our competition website, which would then compile their bots and run them against other the bots of other users. The competition platform would publicly display replays of these games and rank entrants according to the Trueskill ranking algorithm [8].

We believe that this leaderboard system was one of the most important features of the Halite competition. It allowed users to gauge their progress in building a bot, pushed users to build more ambitious bots, and served to disseminate bot algorithm ideas (via the associated game visualizations) throughout the Halite user base.

3) Scaling: The competition platform was built to scale well as the user base grew. Namely, as more bots were submitted, the ranking platform had to expand. We organized our ranking servers in a one-to-many architecture. A scalable field of worker servers actually compiled bot source and ran games. A single manager server coordinated these workers – telling them which bots to compile or which games to run – and served as the gatekeeper to a master repository of bot rankings and competition source.

The ranking servers were hosted on AWS EC2. They interfaced with the AWS EC2 API via the boto library to start up new servers every time a constant number of new users joined the service. The system took advantage of AWS's cost-saving spot instance features, as studied by Mao. [9]

4) Security: The Halite competition platform was built to be secure.

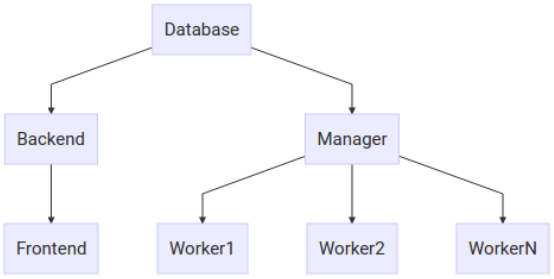


Fig. 5. An rough outline of the server structure of the Halite I competition.

The execution and sandboxing of users' bot code on Halite servers was of primary concern. To accomplish this, Docker containers were used to isolate user-submitted bots from each other and the game environment, limit their CPU and memory resources, and prevent them from accessing the network. Docker was also used to sandbox compilation tasks. VMs and chroot jails were initially considered as the sandboxes. However, both ideas were eventually discounted for their respective latency and insecurity. [10], [11] Docker's security has been studied in detail and was deemed rigorous enough for our uses. [12] The service has in fact been deployed as a code sandbox in production. [13]

To the best of our knowledge, we experienced no issues with security. Players expressed little interest in winning subversively, favoring instead to develop their bots and play legitimately.

5) Community: In order to build a community, the website hosted a Discourse forum. The community built around this forum was vibrant, resulting in thousands of posts and hundreds of thousands of page-views. Other communities have shown similar increases in engagement through the use of forums. [14] Additionally, the community self-started an active Discord channel, in which players chatted and collaborated. Several open voice chats between players were also arranged.

Last, we ran a Halite hackathon for high school students at the Dalton School in New York City. This helped to introduce younger students to the game, and it highlighted the educational aspect of the competition. One high school student team did very well. The educational value has also been demonstrated by others using Halite. For example, the Federal University of Paraná in Brazil recently decided to introduce Halite into one of their courses. The university recognized participation in the competition with the caveat that students would use of certain interfaces and algorithms relevant in the course in question. [15] Numerous other universities are reaching out to the Halite team now about including Halite in their curricula.

The end result of this was a community that was very active and cared deeply about Halite. Users submitted their own starter packages, revised game documentation, provided additional tutorials, and did extra analysis of the website and the game. All of this contributed immensely to the end user experience.

6) Beginner-Friendliness: Halite had a number of features that made it very beginner-friendly.

First, Halite single sign-on (SSO) was integrated with GitHub. This meant that any potential contestant could sign up in seconds with a one-click option, encouraging participation of players new to Halite.

Second, the documentation for the game was clear, concise, and informative. There were tutorials, in both written and visual form, which took users through the process of setting up the environment, building a basic bot, and submitting it.

Third, the centralized leaderboard helped beginners track their progress. Tiers allowed beginners to make goals (there was much discussion of how to become "diamond-tier"), encouraging them by showing both how far a bot can go, as well as the path for getting there.

Last, by providing starter packages, we simplified the process of writing a bot for the user by eliminating boilerplate code and additionally provided users with a stable bot they could immediately submit to get onto the leaderboard without even touching a single line of code. The official Python, Java, and C++ provided from the start of the competition accounted for >80% of all eventual users, and Python by itself was used by over half of all users. Nonetheless, users contributed numerous packages for other languages, resulting in packages for 11 additional languages by the end of the competition.

IV. The 2017-2018 Halite II Competition

The principles outlined in this work were also applied to Halite's second year, Halite II, which became even more successful than its predecessor. Halite II finished with more than 5800 users from 116 countries. Halite II featured a new game designed from these same EPCoF principles, and added both a continuous world and a concrete setting to the game, outer space. Additionally, much of the infrastructure was overhauled for Halite II, improving the end user experience dramatically. These changes can be largely credited to David Li, Harikrishna Menon, and Jaques Clapauch of Two Sigma.[4]

A. The Halite II Game

Halite II is also a turned-based strategy game, in which 2 to 4 players participate. Each player controls a fleet of ships, and has the objective of taking over all planets or becoming the last player remaining. Players control their ships' velocity, and ships can also dock at planets to produce more ships. Ships engage in combat when opposing ships become within range of each other. More details on Halite II are available in The Design Principles Underlying Halite II. [5] Even more detailed rules are available in the Halite II Detailed Rules. [6]

The Halite II game had several advantages and disadvantages in comparison to the original Halite game in terms of EPCoF compliance. The game is certainly more intuitive and visually appealing (see figure 6) – the graphics were

[4]Li was also a Masters Student at Cornell.

[5]https://halite.io/learn-programming-challenge/other-resources/design-principles-underlying-halite-II

[6]https://halite.io/learn-programming-challenge/basic-game-rules/game-rules-deep-dive

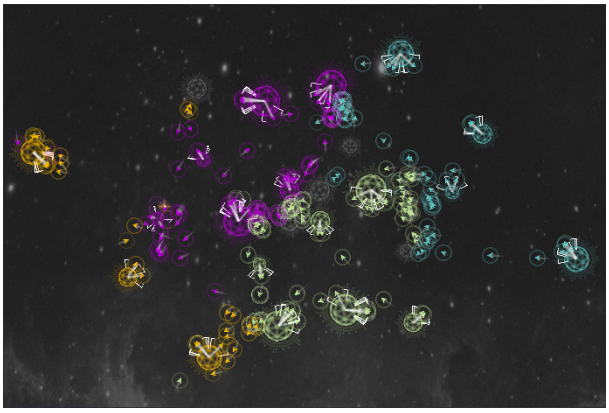


Fig. 6. A snapshot of the Halite II game through its visualizer.

designed by an outside firm with this in mind. Additionally, by leveraging player's intuitive knowledge of physics and the world around them, the game may be easier for new players to grasp its core ideas versus the more abstract Halite I.

However, the game is more computationally expensive and also more difficult for users to program than the Halite I game. The additional computational complexity has less practical importance on the server-side, since the amount of time that it requires is still far less than the compute resources devoted to player's bots, but it does increase the difficulty for players to write their own bots, especially for those using machine learning. More importantly, the use of a continuous space and naive physics makes writing a basic bot more difficult. With Halite I, the starter bot could be drastically improved by changing just one or two lines of code, and the starter code was minimal and simple. To mitigate this, Halite II provided a much more comprehensive but complex set of starter packages in order to mitigate the effect of the continuous world on the user experience, but the authors believe this may have made it more difficult for new programmers to become fully engaged.

A few members of the community commented that the combat/planet capture nature of the Halite II game may have made the game more appealing to men than to women. While it is difficult to determine for sure the exact proportion of users who were men and women, the data implies that both Halite versions were significantly more used by men. As mentioned, the EPCoF principles aim to make games broadly and enduringly appealing, and thus the Halite III team will attempt to make the next version more appealing to all potential participant.

B. The Halite II Competition Construction

The Halite II Competition featured several improvements from the original competition framework.

First, the leaderboard was significantly improved. The introduction of leagues and improvements on affiliations gives player a tighter sense of community, and context around themselves. This change was in part motivated by survey data collected after the first Halite competition. Consider, for example, the path of an inexperienced high school student in Halite. Under the previous system, they would likely only see their global ranking, or perhaps their global ranking among high school students. If the student came in near the bottom of those rankings, they could potentially be discouraged and stop participating and learning. By introducing more local affiliations, the student could compare within their school instead, allowing them to compete with friends and creating a much more motivating experience. Halite had 35 leagues, and one can imagine even finer granularity in future competitions.

Halite II also switched from AWS to Google Cloud for its cloud hosting. The new hosting infrastructure provided more automated server-side scaling. In keeping with Halite's educational mission, Google also donated cloud resources so that users could access GPUs and go much deeper with machine learning than they would have otherwise been able. This allowed the Halite team to satisfy a request for GPUs that had been made by the Halite I community.

Finally, the organizers of Halite II worked hard to promote a sense of community. Numerous Halite hackathons were run. These mostly took the form of education sessions, but circa 20 top players attended a in-person, post-season, one-day hackathon/competition, where the Halite II rules were slightly changed for fun; this was well very well-received. Additionally, the creation of videos by Youtuber sentdex made tens of thousands of potential users aware of Halite.[7] Dozens of post-mortems from top players have been viewed thousands of times, suggesting that players did indeed think the game was interesting and were committed to the competition.

V. Future Work

There are several things that could be improved on in future work.

First, Halite I and II were primarily targeted at college-aged users and older. Of Halite II's users (for which the data is more reliable), 45% were professionals, 43% university or college students, and 12% high-school students. It is also known, though, that games are an effective teaching tool for even younger students. [1], [16], [17] The requirement of text-based languages coupled with the complexity of the games used makes it more difficult for high-school students to participate and nearly impossible for younger students, but if these two issues could be addressed, by further reducing the complexity of the games used or allowing for visual-based programming, such competitions could become an effective teaching tool for young students. The team further plans to integrate an online code editor, which will make the competition more accessible to underprivileged students, many of whom only have access to Chromebooks.

Additionally, it would be productive to engage in making these competitions more inclusive, especially towards women. There are many possibilities, ranging from adjusting the theme of the game to promoting more collaborative work, and, as mentioned, the Halite III team is focusing on these. [18]

Last, it is also interesting to consider what the role of machine learning and AI should play in designing these competitions. As Halite was originally intended to be directly programmed by humans, it was designed to be possible but

[7] https://www.youtube.com/watch?v=QjAu5lJo4zs

challenging to play with machine learning. With the role of AI increasing in the world, though, perhaps future competitions should place a bit more thought onto this topic first. Should modern online programming competitions be designed so that AI is a natural strategy, or should AI just be another tool in the toolbox, with the games designed to be as interesting as possible independent of AI? Halite III is taking the former path, but the authors do not expect that machine learning will be a requisite for (or necessarily even a component of) the best bots. Given the open source nature of the Halite games, the authors are hoping to make the games accessible via the interfaces of the OpenAI Gym, to provide the machine learning community with new multi-agent challenges. [19]

VI. Conclusion

This paper details the construction and administration the Halite Challenge, a successful online programming competition. We've presented our framework for building online programming competitions, both from a general perspective as well as a specific case study. In the future, we hope that others will use, modify, and revise these guidelines in the process of building similar competitions.

Acknowledgments

We would like to thank Emily Malloy, Jacques Clapauch, Matthew Adereth, Herbert Wang, and Arnaud Sahuguet for their invaluable help in running the competition. We'd also like to thank Trammell Hudson and Steve Heller for their mentorship, Alfred Spector for his endless advice, and Two Sigma Investments in its entirety for sponsoring, participating in, and wholeheartedly embracing the concept behind Halite. We thank the Ants AI Challenge, which inspired both the Halite game and competition. We also appreciate Nikki Ho-Shing's editorial comments. Last, we thank our competitors for building a fun, supportive, and intelligent community. Special thanks goes to our community contributors, most notably Brian Haskin, Travis Erdman, and Nick Malaguti.

We encourage the reader to participate in Halite III, which will launch in September 2018.

References

[1] S. Leutenegger and J. Edgington, "A games first approach to teaching introductory programming," in *ACM SIGCSE Bulletin*, vol. 39, pp. 115–118, ACM, 2007.

[2] J. Togelius, S. Karakovskiy, and R. Baumgarten, "The 2009 mario AI competition," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pp. 1–8, IEEE, 2010.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI gym," *arXiv preprint arXiv:1606.01540*, 2016.

[4] S. Lee and J. Togelius, "Showdown ai competition," in *Computational Intelligence and Games (CIG), 2017 IEEE Conference on*, pp. 191–198, IEEE, 2017.

[5] C. Crawford, *The Art of Computer Game Design*. McGraw-Hill/Osborne Media, 2011.

[6] K. Salen and E. Zimmerman, *Rules of play: Game design fundamentals*. MIT press, 2004.

[7] N. Justesen, T. Mahlmann, and J. Togelius, "Online evolution for multi-action adversarial games," in *European Conference on the Applications of Evolutionary Computation*, pp. 590–603, Springer, 2016.

[8] R. Herbrich, T. Minka, and T. Graepel, "Trueskill(tm): A bayesian skill rating system," in *Advances in Neural Information Processing Systems 20*, pp. 569–576, MIT Press, 2007.

[9] M. Mao and M. Humphrey, "Auto-scaling to minimize cost and meet application deadlines in cloud workflows," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 49, ACM, 2011.

[10] J. S. Reuben, "A survey on virtual machine security," *Helsinki University of Technology*, vol. 2, no. 36, 2007.

[11] V. Prevelakis and D. Spinellis, "Sandboxing applications.," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 119–126, 2001.

[12] T. Combe, A. Martin, and R. D. Pietro, "To docker or not to docker: A security perspective," *IEEE Cloud Computing*, vol. 3, pp. 54–62, Sept 2016.

[13] A. Memon, "Remote interview's compile box." https://github.com/remoteinterview/compilebox, 2016.

[14] L. F. Pendry and J. Salvatore, "Individual and social benefits of online discussion forums," *Computers in Human Behavior*, vol. 50, pp. 211–220, 2015.

[15] "C3SL challenge 2 - halite.io." https://challenge.c3sl.ufpr.br/o-jogo/, October 2017.

[16] M. Prensky, "Digital game-based learning," *Computers in Entertainment (CIE)*, vol. 1, no. 1, pp. 21–21, 2003.

[17] O. Shabalina, P. Vorobkalov, A. Kataev, and A. Tarasenko, "Educational games for learning programming languages," 2008.

[18] K. Subrahmanyam and P. M. Greenfield, "Computer games for girls: What makes them play," *From Barbie to Mortal Kombat: gender and computer games*, pp. 46–71, 1998.

[19] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[20] J. Togelius, "How to run a successful game-based ai competition," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 1, pp. 95–100, 2016.

[21] N. Malaguti, "Halite tournament manager." https://github.com/nmalaguti/halite-bots/tournament, 2017.

Benjamin Spector is an undergraduate student at the Massachusetts Institute of Technology. He co-founded and developed the Halite competition, for which he was a co-winner of the ACM/CSTA Cutler-Bell Prize in High School Computing. In addition to interning at Two Sigma, Ben has worked under Serge Belongie at Cornell Tech on sample-efficient deep reinforcement learning, and published a workshop paper at NIPS 2017.

Michael Truell is an undergraduate student at the Massachusetts Institute of Technology. He co-founded and developed the Halite competition, for which he was a co-winner of the ACM/CSTA Cutler-Bell Prize in High School Computing. In addition to interning at Two Sigma, Michael has worked as an RSI/Department of Defense scholar under Dr. Vikash Mansinghka and Feras Saad at the MIT Probabilistic Computing Group on Bayesian nonparametric inference systems.