Monster Carlo: an MCTS-based Framework for Machine Playtesting Unity Games

Oleksandra Keehl
Design Reasoning Lab
Department of Computational Media
University of California, Santa Cruz
okeehl@ucsc.edu

Adam M. Smith
Design Reasoning Lab
Department of Computational Media
University of California, Santa Cruz
amsmith@ucsc.edu

Abstract—**We describe a Monte Carlo Tree Search (MCTS) powered tool for assessing the impact of various design choices for in-development games built on the Unity platform. MCTS shows promise for playing many games, but the games must be engineered to offer a compatible interface. To circumvent this obstacle, we developed a support library for augmenting Unity games, and Python templates for running machine playtesting experiments. We also propose ways for designers to use this tool to ask and answer designs questions. To illustrate this, we integrated the library with *It's Alive!*, a game in development by the authors, and *2D Roguelike*, an open source game from the Unity asset store. We demonstrate the tool's ability to answer both game design and player modeling questions; and provide the results of system validation experiments.**

I. Introduction

Human playtesting is an irreplaceable aspect of game development. It can also be logistically cumbersome and creates a significant bottleneck in the design cycle: design, build, test, learn, and redesign. One of the main arguments for machine playtesting is that a simulator can play through games orders of magnitude faster than a human. Thus, it can cover more ground, collect more data, and, in some cases, provide guarantees through exhaustive search.

We present Monster Carlo, a framework for machine playtesting Unity[1] games. Tools based on this framework can gather data on different design variants and playstyles in order to detect imbalance and other effects design changes may have on a player's experience. We set out to apply Jaffe's Restricted Play balance framework [1], to *It's Alive!*, a game in-development by the first author (Figure 2). Contrasting with Jaffe's work, which examined the win–lose outcome of competitive two-player card games, *It's Alive!* emphasizes maximizing the score in a single-player *Tetris*-like game. Exhaustive search is not tractable in the general case of *It's Alive!*, due to the vast number of reachable states. In response, we apply Monte Carlo Tree Search (MCTS) to find input sequences that approximately maximize the player's score.

Monster Carlo aims to bring AI techniques to game developers in the platforms they are already using. In the implementation of our framework, we made an effort to minimize the game code changes required for integration. We also provide result visualization templates in Jupyter Notebook, a tool often used for gameplay data analysis [2].

[1]https://unity3d.com/

To test the tool's versatility, we integrated it with a game of a different style and one we did not develop: *2D Roguelike* (Figure 5), the open source game from the Unity asset store.[2]

Monster Carlo is meant to answer a variety of design questions: In *It's Alive!*, how do monster "come to life" conditions affect the achievable high scores? How does a player who collects monsters right away compare to a player who waits until the last moment? In *2D Roguelike*, how does the game dynamic change if we increase both the damage dealt by the zombies and health gained from food pick-ups? How feasible is a no-backtracking player strategy?

MCTS has many variations. We experimented with never re-visiting fully explored branches of the search tree, aiming to increase the number of states explored. We tested different values for the tunable exploration constant in the UCT[3] algorithm, finding the search performs better if the constant is closer to an average score of a human player. We also added an optimization technique of saving the entire playtrace of each playthrough with a new best score [3]. All of these game-agnostic variations are available through parameters of Monster Carlo.

In general, using MCTS for machine playtesting requires the games to be engineered to be compatible with it. The Monster Carlo support library is designed to hook into the Unity engine update cycle and makes this engineering easy. We also created data analysis templates for experiments that take the form of comparing optimized scores between variants.

This paper makes the following contributions:

- A framework for machine playtesting Unity games where instances of the game execute rollouts for MCTS.
- First work to use MCTS at the level of a whole game platform rather than a specific game.
- C# and Python support libraries for adapting a game to support machine playtesting and running experiments.[4]
- Initial experiments that validate the framework and answer design questions about an in-development game.

[2]https://www.assetstore.unity3d.com/en/#!/content/29825
[3]Upper Confidence Bound-1 applied to Trees
[4]https://github.com/saya1984/Monster-Carlo

II. Related Work

In this section, we review the work related to three topics relevant to Monster Carlo: design inquiry with restricted play, MCTS, and frameworks designed to support MCTS.

A. Design Inquiry with Restricted Play

Ludocore is a logical game engine for modeling video games [4]. Smith et al. imposed restrictions on the player behavior in order to analyze games created within the Ludocore framework. From Ludocore, we borrow the idea of asking design questions by restricting our player models and observing how these constrained players perform as variations in the design are considered. In Ludocore, games had to be encoded in a specially-designed logic programming language which Ludocore's back-end analysis engine could understand. By contrast, Monster Carlo is meant for integration with Unity games. Such games can dynamically allocate memory, hand-off simulation to a physics library, or perform other computations that would be tedious to model in a purely symbolic framework.

The Restricted Play concept of asking game balance questions by preventing or forcing a player to do certain actions in game was introduced by Jaffe et al. [1] and was applied to a two-player, perfect-information game *Monsters Divided*. In their evaluation tool, the authors calculated the optimal strategies for each type of restricted behavior. The game size (five cards per player) allowed for exhaustive search through the entire game tree, foreseeing every possible playthrough. In their Future Work section, the authors state that MCTS is a promising alternative for games whose complexity makes exhaustive search impractical. Because of MCTS's agnosticism to a game's features, it can be used without modification on different restricted players and game design variants. In this paper, we extend Jaffe's restricted play idea, combining it with MCTS and applying it to a new class of games: single player, discrete state games with a larger space of states.

Zook et al. [5] follow up on Jaffe's suggestion to use MCTS for analyzing large games. They experimented with *Scrabble* and the *Magic: The Gathering* inspired card game *Cardonomicon*. They used restricted play to simulate player skill levels to see what trends and strategies emerge when players of different skills are pitted against each other. Although *Scrabble* and *Cardonomicon* are naturally imperfect-information games (one cannot be sure which tiles or cards the opponent has until they are played), Zook et al. work with determinized, perfect-information variations of these games (explained further in section II-B). *It's Alive!* is also a nondeterministic game (the player does not know which piece will be randomly dropped next), and we use the determinization strategy by fixing the game's random seed value.

Holmgrd et al. [6] used genetic algorithms to evolve custom evaluation functions for use with MCTS to simulate different playstyles in the game MiniDungeons 2. They used the evolved personas to assess feasibility of each playstyle in different level layouts. This approach can be combined with Monster Carlo, as it allows the user to implement custom evaluation functions for use within simulations and final score computation.

B. MCTS

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm which selectively explores the tree of possible moves [3]. It assesses the potential of each move by averaging the scores of simulated random playouts from the current point in the game until a terminal state. Although MCTS denotes a broad family of algorithms, the most common, UCT, has a single tunable parameter: the balance of advancing the more promising branches of the tree with exploring the paths less traveled. MCTS is a relatively simple algorithm and can be used with a multitude of decision problems without the need for game-specific heuristics. It has been successfully applied to *Go* [7], *Tetris* [8], *Scrabble* [5] and other games. Most recently, it was used in the creation of AlphaZero [9], the latest world champion in *Go*. In the rest of this paper, we use the general term MCTS to refer to the specific instance of UCT in the Monster Carlo framework.

MCTS relies on building a tree of the possible moves in each state. In games with a random element, such as *Tetris* or *Scrabble*, the search was performed using a predetermined sequence of pieces. In the determinized version of the game, the automated player's goal is simply to find the single best sequence of moves that maximizes the final score. Without determinization, the much more difficult goal is to devise a policy that maximizes the expected score averaged over all possible random elements in the game. We used determinization, as it is sufficient to answer the design questions Monster Carlo is intended to answer. One of the main differences in our application of MCTS is that the games described above all had a win/lose condition (even *Tetris*, as it was played competitively), while the games described in this paper focus on the highest score attained.

C. Environments that support MCTS

The Video Game Definition Language (VGDL) [10] is a representational language for modeling videogame mechanics and level designs. The General Videogame Artificial Intelligence (GVG-AI)[5] project provides an interpreter for VGDL games which exposes an MCTS-compatible forward model. Although VGDL has been used to model games inspired by many different kinds of pre-existing videogames, it cannot integrate with the original implementations of any of these games. Like Ludocore, GVG-AI tools can only understand games expressed in a specialized language. By contrast, Unity games can make unrestricted use of the general purpose C# programming language used for Monster Carlo.

OpenAI Gym[6] is a testbed for AI. It includes environments which provide state information, pixel data and rewards in response to an agent's action. It can integrate with commercial ROM implementations of many Atari games and can have algorithms learn to play directly from pixel or memory data,

[5]http://www.gvgai.net
[6]https://gym.openai.com/

rather than the simplified game state abstraction used in VGDL. A growing number of environments are available for AI experimentation. A related effort, OpenAI Universe,[7] aims to allow integration with an even wider array of gameplay-like activities, even including a mock travel arrangement task based on interaction with complex websites. Although these frameworks allow MCTS-style algorithms to play a very wide variety of games, they force interaction with the game at the lowest level of interaction common to all of them: reading pixels or memory bytes and injecting keyboard and mouse actions. By contrast, Monster Carlo is intended to give designers control of the level of abstraction used by MCTS including high-level game actions (e.g. directly playing a card rather than clicking somewhere to select a card). This is important for making Monster Carlo's analysis useful for game developers rather than AI researchers.

All of these systems deal with the representations of game states and actions differently. VGDL uses a data structure for tracking the abstract state of the game and a list of interactions between game objects as action representation. OpenAI Gym uses screenshot pixel data and memory contents for state representation and low-level keyboard and mouse events as actions. Monster Carlo does not represent a game state beyond the sequence of moves needed to reach it, and a way of asking the framework to make discrete micro-decisions which assemble into high-level actions (further described in III).

D. Unity and Unity Machine Learning Agents (ML-Agents)

Unity is a powerful game engine available for free for private use, which makes it popular with independent developers. Unity does not directly support integration with MCTS. To change this, it is necessary for the game to communicate what actions are possible at any moment and provide a way for some AI system to select and apply one of those actions. Additionally, there needs to be a way of communicating a score to be optimized to the AI system. As the level of granularity used to model player choices and the notion of score to be optimized are specific to the game being designed (and even specific to certain design questions being asked of that game), these cannot be provided directly at the level of the Unity platform. In response, Monster Carlo aims to offer the designer a minimal-effort way of expressing game-specific concerns on top of the Unity platform.

Unity ML-Agents[8] is a plugin meant to enable using games and simulations to train agents via various machine learning methods. While both Unity ML-Agents and Monster Carlo involve running many thousands of simulated play traces, Monster Carlo focuses on summarizing those trace for immediate review by designers rather than producing a trained behavioral policy as a side effect. As a result, Monster Carlo has very few parameters to adjust and does not require defining a neural network architecture or other policy representation.

[7]https://blog.openai.com/universe
[8]https://github.com/Unity-Technologies/ml-agents

III. System Design

The Monster Carlo framework consists of four major parts (Figure 1). The integration modifications to the game and specifications for the design experiment (the green parts on Figure 1) have to be written by the game designer, while everything else is provided through the Monster Carlo tool.

We used MCTS as the main search algorithm for our experiments, but the algorithm can be changed without adjusting the game or the design experiment notebooks.

A. Experiment setup and result visualization

The user-facing element of the Monster Carlo framework is a sample Jupyter Notebook for running the experiments and visualizing the results. This includes the game process factory. The user must provide a function that can be called to start an instance of the game, and the game must be compiled with the C# support module (see III-C). The customized process factory can be used to pass experiment-specific configuration data to the game. For example, one can configure it to start in a certain mode optimized for analysis, or arrange for the execution of the game to happen on a remote cluster of machines. The experiment results are returned as an object, which can be saved in a file at the end of each experiment and later used for analysis and visualization. We used matplotlib[9] to visualize the results. All the experiment result graphics in this paper were obtained through this method.

B. Python support module

This module contains the implementation of the MCTS algorithm in Python. Upon the termination of the experiment, it returns an object which contains the search tree and any additional data. This output object can be trivially modified to keep track of additional metrics. In our experiments, we kept track of the growth of the highest seen score over the rollouts, but we could, for example, have as easily kept track of the number of monsters collected during a playthrough. In a narrative oriented game, we might track the fraction of dialog content seen or tally which endings where reached.

The tool supports running multiple instances of the game to significantly speed up the search (see Section V-A for the results). It takes as an argument the number of rollouts and additional optional arguments that include the UCT constant value, the number of parallel workers, terminal branch treatment, saving of the best path option, and a callback function, which can be used to implement custom logging. These are passed to the game instances via environment variables.

C. C# support module

The C# module must be added to the game project. The module takes in the environment arguments at the start of the experiment and communicates with the Python module through a TCP socket. It receives the most promising path prefix determined by the MCTS algorithm from the Python module at the beginning of each playthrough. Each time a

[9]https://matplotlib.org

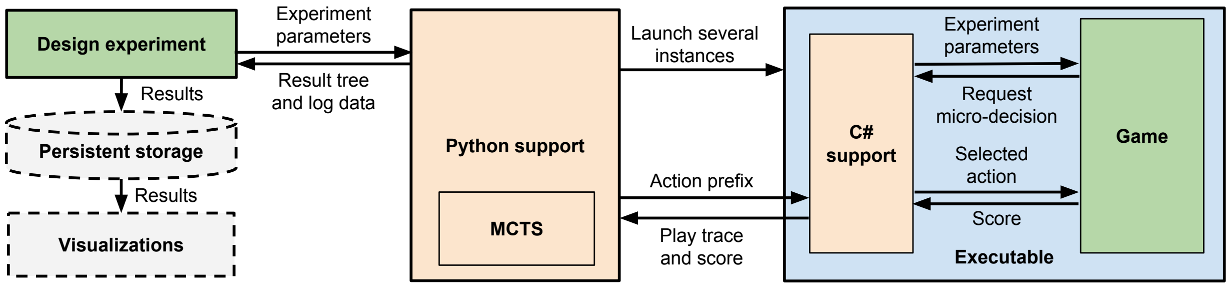

Fig. 1. High level architecture of the Monster Carlo framework. The specification of the design experiment (in Python) and the game code itself (in C#) are project-specific, while the other elements are provided by the framework.

decision must be made in game, the game tells the module how many legal moves are available, and the module makes a choice without needing to know what those moves are. If there are pre-determined moves in the path prefix, the module feeds those back to the game one at a time. When the end of the path is reached, the module continues by making random choices. A custom heuristic can be optionally be expressed in C# by providing an array of action selection weights to be consulted during the rollout phase. When the play session is over, it reports the final score to this module, which sends the full action path, the final score, and any other information the designer specified, back to the Python module.

D. Modifications to the game

The designer must implement micro-decisions and scoring: the game must determine legal moves at each step, request an index of a move to take, and apply that action. When the game reaches a terminal state, it must provide the score to the support module. If random elements are present, each playthrough needs to use the same random seed. We also recommend creating a headless, no-graphics mode for the game, as it can significantly speed up the playthroughs on some platforms.

The game also needs an experiment mode to be able to replace the user's input with decision requests to the C# module. Launching the game in the experiment mode can include skipping menu screens and disabling smooth movements. To optimize the running time, we recommend adding an ability to reset the game after the terminal state is reached, so that the application doesn't have to be re-launched for each fresh playthrough.

Additionally, if the designer wishes to conduct Jaffe-style restricted play experiments, they will have to implement the player models (which may limit available actions before the C# module is queried for a choice). They can also implement a way to switch between the game design variations. The space of design variants considered can be as flexible as the user wants, as long as they can specify those variants in Python and communicate them to the game's executable.

IV. Experiments

This section describes example experiments we conducted using the Monster Carlo framework. They show how to use the restricted play methodology to ask design questions for two games: *It's Alive!* and *2D Roguelike*.

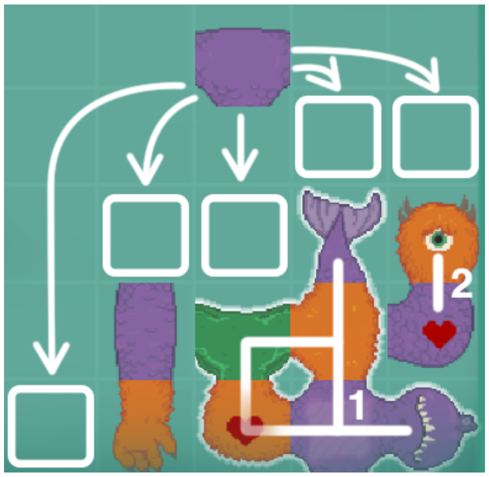

Fig. 2. Possible actions in this state include landing the falling piece in one of the five columns in one of four orientations, or collecting one of the two living monsters.

A. It's Alive!

It's Alive! (see Figure 2) is a *Tetris*-style game where the player controls the position and orientation of pieces falling from the top of the board. Player loses if the pieces pile up to the very top of the board. Rather than trying to make simple horizontal lines of pieces as in *Tetris*, the player must form arrangements of pieces that represent monsters. A monster comes to life when it minimally contains a head piece and a heart piece. At this point, the player may choose to collect it to free up space, or continue building it up. Bonus points are awarded based on the size and color coordination of each monster. If there are several moving monsters on screen, the player can choose which one to collect by shifting the highlight from one monster to another. The player aims for the highest score by animating and collecting five monsters.

The player actions consist of rotating the falling block, moving it left or right, or quick-landing it. The player can also cycle the highlighter through living monsters or delete the currently highlighted monster. Thus, at any point, she has four to six possible keyboard-level actions: rotate, move

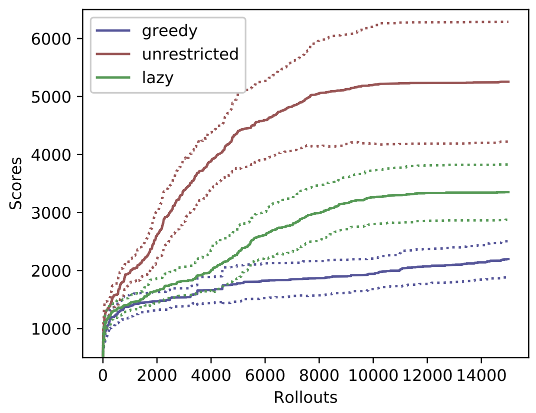

Fig. 3. Highest score achieved for *greedy*, *unrestricted* and *lazy* player models.

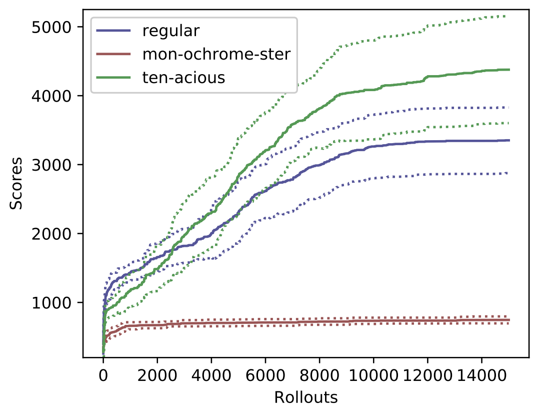

Fig. 4. Highest score for *regular*, *mon-ochrome-ster* and *ten-acious* designs.

left, move right, quick-land, cycle highlighter, collect monster. Some of those actions could be repeated indefinitely without affecting the game state, meaninglessly expanding the scope of the search for Monster Carlo to perform. To avoid this, we use micro-decisions to model only those choices that resulted in meaningful state change. The OpenAI Gym would have forced a mode of interaction at the level of keyboard inputs, whereas Monster Carlo allows the flexibility to focus the analysis on the level of details the designer cares about. Instead of ability to move the block left or right any number of times, the artificial player simply chooses whether to collect one of the living monsters or to land the piece in any orientation in an open column. Similarly, cycling the highlighter is not considered an action, instead, collecting any of the living monsters in the current state is considered a legal action, regardless of the highlighter position (Figure 2). With this new definition of action in mind, the player has 20 or more possible actions at every moment. That is five possible columns times four landing orientations, plus one collect action per living monster. On a 5x7 playfield, this makes exhaustive search computationally intractable due to the vast number of possible combinations.

1) Playstyle experiments: Like *Tetris*, *It's Alive!* has many quick game-over states resulting from piling pieces in the same column and reaching the ceiling while most of the playfield is still empty. To prevent Monster Carlo from wasting time exploring these dead-end scenarios, we prevented all player models from placing a piece that would end the game if a non-game-ending move was possible, such as placing a piece somewhere else or collecting a monster. We did this by excluding the game-ending moves from the list of available actions within the game. No changes to the Monster Carlo framework were required to express this more focused analysis.

We used factored actions for most of *It's Alive!* experiments. Each turn, the player makes a sequence of micro-decisions. First: Should I land the current piece or collect a monster? Next, if I chose to land a piece: Which column should I land it in? Finally, in which orientation should I land the piece?

We experimented with three player styles. The *greedy* player collects the monsters as soon as they came alive. The *lazy* player collects a monster only if the game would otherwise end. The *unrestricted* player is free to collect at any point.

Figure 3 shows that the *lazy* player did best, while the *greedy* player performed the worst. The p-values designating statistical significance of the difference between the scores of each pair of the results ranges from 3.4e-08 to 4.5e-07.[10]

These results show that deciding when to collect a monster is a meaningful choice for the player. Notably, while technically nothing prevented the *unrestricted* player from achieving the same results as the *lazy* player, presenting it with an opportunity to collect the living monster at every step slows down the search progress. This is a reminder that all results from MCTS are approximations computed within a fixed computational budget, so they cannot be trusted with the same level of certainty as in the exhaustive search results in Jaffe's original Restricted Play work. Nevertheless, large score gaps can provide a signal that a designer should look deeper into the specific playtraces found by MCTS that illustrate specific styles of play in action. For this reason, it is important that Monster Carlo returns the resulting tree, not just the aggregate statistics. The user may decide to replay the highest scoring play trace in a mode with more detailed analytics turned on to gain deeper insight into the impact of playstyle difference that the tool discovered.

2) Design variants: We considered three game design variants. The *regular* design follows the rules outlined above. The *mon-ochrome-ster* design considers two pieces within a monster connected only if they are of the same color. In the third variant, *ten-acious*, the monster only comes to life if it consists of at least ten pieces.

The results (Figure 4) show that the *mon-ochrome-ster* mode is much harder than the other two, and affords for a lower maximum score. Counter-intuitively, the *ten-acious* design variant,

[10]Here and in all other experiments, statistical significance is judged according to the single-sided Mann-Whitney U test applied to the highest score achieved within the rollout limit. Each experiment involves 20 independent replicates of each condition.

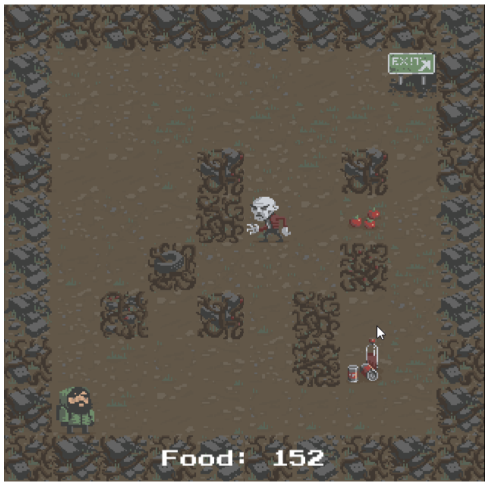

Fig. 5. Screenshot of Unity tutorial game *2D Roguelike*.

which places a restriction on the player and thus, makes for a harder game, led to higher scores than those Monster Carlo achieved in the *regular* design. Both experiments were run with the same random seed, so nothing prevented the *regular* design player from building monsters of ten blocks or more. The progression of the highest score seen across the rollouts in Figure 4, shows that the *regular* design scores are higher initially, but are quickly overtaken by those seen in *ten-acious*. We believe this is caused by the restrictions in *ten-acious*, which prevented the search exploring the frequent collection of smaller monsters. As before, Monster Carlo does not replace the user's judgment of game design alternatives, but it can gather specific evidence that helps the user make that judgment for themselves.

B. 2D Roguelike

Note that because we are the developers of both Monster Carlo and *It's Alive!*, it is possible that we have over-specialized the framework for analysis of games very much like *It's Alive!*. In this section, we consider the integration effort and results from experiments with a game that we did not make ourselves, nor considered during the primary development of the Monster Carlo framework.

2D Roguelike (Figure 5) is an open source official tutorial game for the Unity game engine.[11] It is grid and turn based: zombies get to take a step for every two steps the player takes. The player starts at the lower left corner of the field and the goal is to reach the exit in the upper right corner, signifying he has survived another day. The game is over when the player runs out of food points and the final score is the number of days the player has survived. One food point is lost for every move and several are lost in case of zombie attacks. The food points can be replenished by picking up food items. The levels are laid out randomly. The number of zombies is a function of the number of days survived, gradually increasing. At any point, the player may choose to go up, down, left or right. Each of these actions results in a state change, as the food points go down even if the player attempts to walk through a wall and does not actually move.

1) Playstyle experiments: For *2D Roguelike* we factored the actions into a choice of moving toward or away from the exit, and then deciding whether the move is lateral or vertical. For the first player, as a simple heuristic, we used Monster Carlo's capability for weighted choice to make the player more likely to move toward the exit in the rollout phase of MCTS. The second player was restricted to only move toward the goal. After 30,000 rollouts, the *forward-only* player achieved statistically significantly higher scores ($p = 3.3e\text{-}08$).

Due to the game mechanics, while the *forward-only* player has a short-term advantage of a powerful heuristic, it would eventually come to a hard limit, as it is impossible to pass some levels without backtracking to avoid the zombies. In this game, while the player can break through inner walls, it is impossible to kill the zombies. If the player runs into one and cannot back away, it will eventually kill him. Given enough time, we believe the *unrestricted* player would outperform the *forward-only* player. However, this would take too long to be practically feasible for playtesting. Another option would be to increase the bias with which the *unrestricted* player would select the forward motion vs. backtracking. This would help get more realistic scores faster without imposing the forward-only restriction.

2) Design variants: We compared the game's default configuration with one where both the damage dealt by the zombies and food gained from pick-ups were increased by 50 percent. The results from this *high stakes* design variant were statistically significantly higher ($p = 4.8e\text{-}07$). From this, one could conclude that the *high stakes* variant of the game is easier to play for any given score threshold.

V. Framework Validation

The original design of *It's Alive!* has a 5x7 playfield, which makes for a large search space with the average branching factor of 20 and depth of at least 36 (if no monsters are collected). This leads to longer rollouts and slower depth-wise exploration rate. For the framework validation experiments we reasoned that having a smaller playfield would allow us to run experiments faster while still demonstrating relative differences between performance of Monster Carlo with different parameters. We built a smaller *It's Alive!* with a 3x5 playfield and only three monsters required for the win. A typical human player score for this game is 1200-1400 points. Unless stated otherwise, the experiments were run with an unrestricted player, 24 parallel workers, factored actions, cut-off terminal branch setting, and the exploration parameter in the UCT algorithm set to 1000.

A. Parallel vs. Single Thread

The classic MCTS updates the tree after each rollout and uses the updated tree to chose the next move. With instances of the game running in parallel, the tree is updated each time a playthrough is completed, and the next move is selected with

[11] https://www.assetstore.unity3d.com/en/#!/content/29825

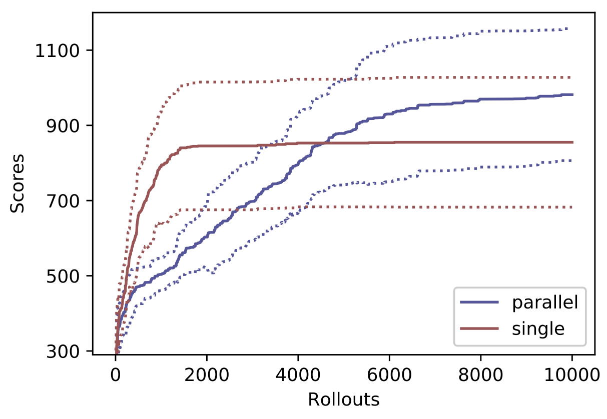

Fig. 6. Highest score achieved for a single worker and 24 workers in parallel on a 3x5 board (20 replicates). The rollouts completed approximately 20 times faster in the parallel case, better results in less wall-clock time.

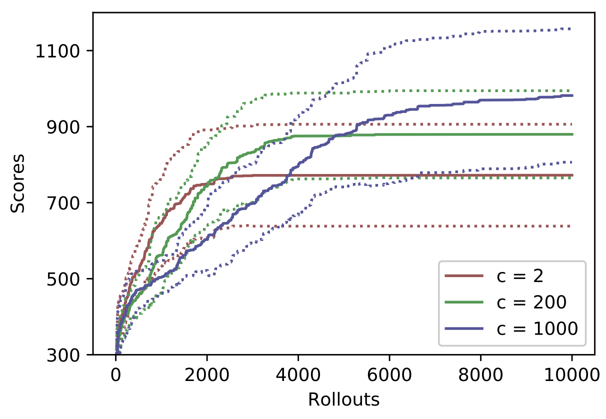

Fig. 7. Highest score achieved with different values of the UCT exploration constant (20 replicates)

the results from several other in-flight rollouts still unknown. This is similar to the tree parallelization with global mutex approach described by Chaslot et al. [11]. We wanted to see if there was a large drop in the tool's effectiveness at the cost of the speed. We ran two experiments with 30,000 rollouts. The first set used 24 parallel workers, and the second used a single worker. The parallel experiment achieved higher scores (Figure 6) with statistical significance (p = 0.02). However, the bigger difference is in the duration. The single-thread experiment lasted eight hours and 20 minutes while the parallel experiment took about 24 minutes. Initially, the parallel experiment took more rollouts to get to the same scores as the single-thread experiment. This leads us to extrapolate that parallel workers have a diversifying effect on MCTS. While this initially leads to lower scores, it also makes it less likely for the search to get bogged down in unproductive territory.

B. Terminal Branch Treatment

We noticed that MCTS tended to explore the same branch and get stuck in local maxima, though much better paths were available. We tried two ways around it. One was to increase the UCT constant, traditionally set to 2. The other way was to prevent the tree from revisiting branches marked as terminal.

We ran two experiments, with 30,000 rollouts each, one with no special treatment of terminal nodes and branches, and the other that would mark fully explored sections of the tree as terminal and ignore them during the optimal path selection. The results for these experiments showed no statistically significant difference between the highest scores achieved or the number of nodes explored. We hypothesize that this is largely due to the fact that the depth of our test game was too great, and so the terminal branch treatment did not come into play to a significant degree.

Notably, not revisiting terminal branches allows for exhaustive search on smaller fields. Earlier in this project, we ran tests on *It's Alive!* with a 2x3 grid. We expected the maximum score to be 250 points but found that one of the branches achieved 290 points. This led to the discovery of a bug that only manifested if the monster pieces were positioned in one specific way. After fixing the bug, we were able use Monster Carlo to exhaustively verify the fix by running the same test with cut-off terminal branches and observe that no branches scored higher than 250 points.

C. UCT Constant

The UCT exploration constant (c) regulates how much MCTS focuses on exploring the most rewarding paths vs. exploring new areas. Because MCTS is usually applied to games with a win/lose outcome and the reward values ranging from 0 to 1, we hypothesized that when applied to a game where the reward value is the range of possible high scores, the UCT constant should be closer to a score you would expect from a moderately proficient player. We obtained this score by manually playing the game with the same random seed.

We ran three experiments with respective UCT constant values set to 2, 200 and 1000. The results in Figure 7 demonstrate that Monster Carlo did best with $c = 1000$, which was closer to the expected score of 1300. The results were statistically significant for comparison of $c = 2$ and $c = 1000$ (p = 0.0002), and $c = 200$ and $c = 1000$ (p = 0.01). While the experiment with a lower c got slowed in local maxima fairly early on, the scores corresponding to the higher c continued growing due to the search's higher emphasis on exploration.

D. Experiment Speedup Techniques

If the MCTS rollouts happened at the game's normal play speed, each of the aforementioned experiments would take days to complete. Therefore, we employed a number of speedup techniques. Game-side changes included setting Unity's framerate to maximum value, replacing smooth movements with instant jumps and disabling all artificial delays (such as waiting half a second between accepting player inputs). This increased the experiment run speed by a factor of ten. Game-agnostic changes consisted of running parallel search with 24 workers (speedup by a factor of 20) and running on a server-class machine without graphics (speedup by a factor of eight). The combination of all these allowed us to run experiments at approximately 1600 times faster than the original. Distributing the rollouts across several server-class machines would allow even greater speedups.

VI. Conclusion

We presented Monster Carlo, an MCTS-based tool that can be integrated with the Unity game engine and be used to perform machine playtesting of in-development games; conducted a number of framework validation experiments, which showed merit in adjusting the UCT constant, using parallel processing when performing rollouts, and applying special treatment to terminal nodes and branches. Monster Carlo was integrated with two games: our in-development game *It's Alive!* and an official Unity tutorial game, *2D Roguelike*. The integration with *2D Roguelike* required fewer than 100 lines of code. We also presented results of several experiments run on both games, exploring restricted player models and design variations.

Obtaining reasonable results from MCTS on a complex game takes time, but so does making meaningful changes. Some modifications, like restricted player models or limited variety of pieces, can be added to a game fairly quickly. Larger changes, like introducing a new type of block to the game or adding heuristics to a player model, usually take much longer. With this in mind, even if a set of experimental replicates takes over an hour to run, it can be considered an acceptable turn-around time, as the results will likely be in before the next model is ready for testing. Additionally, the independent runs of MCTS are extremely parallelization-friendly.

Having a reference score helps with setting an appropriate UCT constant value to guide MCTS toward better results. A reference score can be provided by the game designer, or someone familiar with the game, who can play one or two games to provide a baseline score. This score can be helpful for setting the UCT constant, as well as interpreting the MCTS results: if its best scores are much lower than what a casual player can get, it indicates that MCTS needs tuning.

A severely limited player model can still provide information. In early stages of this project, we ran experiments on an even larger *It's Alive!* playfield of 6x8. We used an unrestricted player model, and one that did not rotate the pieces. We experimented with lowering the number of different monster colors. The size of the field led to a very wide tree that never had a chance to explore very deeply and resulted in chaotic and mostly very low scores for the unrestricted player. However, the non-rotating player, whose actions were limited by a factor of four on every step, was capable of reaching more stable scores in the same number of rollouts. The scores made it evident that the no-rotation player got significantly higher scores when fewer monster colors were present (from an average of 1600 to an average of 2100). This is an obvious example, since having fewer colors means it is more likely to get two blocks of the same color next to each other. However, it showed that even a severely limited player model is capable of providing information about design variants.

Much work remains to be done around Monster Carlo, as the scores it achieves in a reasonable time still fall short of human results. In the current setup, the search algorithm has no representation of game state beyond the action sequence, so it cannot transfer experience gained down one sequence of moves to another if they differ by even a single move. Reinforcement learning algorithms such as those used in AlphaZero can distill knowledge gained during MCTS rollouts into value-estimation and rollout-policy networks that can be applied to states that have never been explored before. We believe that borrowing some ideas from frameworks like OpenAI Gym (such as representing game state with universal data structures like screenshot pixel arrays or memory byte arrays) could help a generic search algorithm learn a much better default action policy than even the human user could provide. However, even with its current shortcomings, Monster Carlo is capable of providing usable feedback. With the convenient experimental setup in Jupyter Notebook, our hope is that new kinds of machine playtesting experiments can by invented and executed easily. Nelson et al. [12] and Isaksen [13] suggest several strategies for understanding game artifacts, some of which could be implemented using Monster Carlo.

References

[1] A. Jaffe, A. Miller, E. Andersen, Y.-E. Liu, A. Karlin, and Z. Popović, "Evaluating competitive game balance with restricted play," in *Proc. of the Eighth AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment*, ser. AIIDE'12, 2012, pp. 26–31.

[2] S. Martinelli, "Starcraft ll replay analysis with jupyter notebooks." [Online]. Available: https://github.com/IBM/starcraft2-replay-analysis

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Trans. on Comp. Intel. and AI in Games*, vol. 4, no. 1, pp. 1–43, March 2012.

[4] A. M. Smith, M. J. Nelson, and M. Mateas, "Ludocore: A logical game engine for modeling videogames," in *Proc. of the 2010 IEEE Conf. on Computational Intelligence and Games*, Aug 2010, pp. 91–98.

[5] A. Zook, B. Harrison, and M. O. Riedl, "Monte-carlo tree search for simulation-based strategy analysis," in *Proceedings of the 10th Conference on the Foundations of Digital Games*, 2015.

[6] C. Holmgrd, M. Green, A. Liapis, and J. Togelius, "Automated playtesting with procedural personas with evolved heuristics," pp. 1–1, 02 2018.

[7] S. Gelly and D. Silver, "Achieving master level play in 9×9 computer go," in *Proc. of the 23rd AAAI Conf. on Artificial Intelligence*, 2008.

[8] C. Zhongjie, "Playing tetris using bandit-based monte-carlo planning," in *AISB 2011: AI and Games*, 2011.

[9] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, 10 2017.

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

[11] G. M. B. Chaslot, M. H. Winands, and H. J. van Den Herik, "Parallel monte-carlo tree search," in *Proc. of International Conference on Computers and Games*, September 2008, pp. 60–71.

[12] M. Nelson, "Game metrics without players: Strategies for understanding game artifacts," 2011. [Online]. Available: https://aaai.org/ocs/index.php/AIIDE/AIIDE11WS/paper/view/4114

[13] A. Isaksen, D. Wallace, A. Finkelstein, and A. Nealen, "Simulating strategy and dexterity for puzzle games," in *IEEE Conference on Computational Intelligence and Games*, Aug. 2017.