UNIVERSITY OF CALIFORNIA
SANTA CRUZ

MONSTER CARLO: AN MCTS-BASED FRAMEWORK FOR MACHINE PLAYTESTING UNITY GAMES

A thesis submitted in partial satisfaction of the
requirements for the degree of

Master of Science

in

COMPUTATIONAL MEDIA

by

Oleksandra G. Keehl

March 2018

The Thesis of Oleksandra G. Keehl
is approved:

Asst. Professor Adam Smith, Chair

Professor Jim Whitehead

Professor Michael Mateas

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Monster Carlo: An MCTS-based Framework

for Machine Playtesting Unity Games

by

Oleksandra G. Keehl

In this thesis, I describe a Monte Carlo Tree Search (MCTS) powered tool that I created to help assess the impact of various design choices for in-development games built on the Unity[1] platform. MCTS shows promise for playing many games, but the games must be engineered to offer a compatible interface. To circumvent this obstacle, I developed a support library for augmenting Unity games, as well as experiment templates in Jupyter Notebook for running machine playtesting experiments. I also propose ways for designers to use this tool to ask and answer designs questions. To illustrate this, I successfully integrated the library with *It's Alive!*,[2] a game I am currently developing, as well as *2D Roguelike*, an open source tutorial game available from the Unity asset store.[3] The integration took fewer than 100 lines of code (see Appendix B). I demonstrate the tools capability to answer both game design and player modeling questions, as well as provide the results of the system validation experiments.

[1] https://unity3d.com/

[2] https://www.kongregate.com/games/saya1984/its-alive

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

Chapter 1

Introduction

Human playtesting is an important and irreplaceable aspect of game development, however it can be logistically cumbersome and provides for 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 player can, and thus, can cover more ground, collect more statistical data, and, in some cases, even provide guarantees through exhaustive search. As described later in this thesis, automated playtests can be run on design variations and with different playstyles to expose the effects the changes may have on different aspects of the game.

In this thesis, I describe 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 general effects design changes may

[1] https://unity3d.com/

have on the player's experience. I set out to apply Jaffe's Restricted Play balance framework [5] to *It's Alive!* a single player game I am developing. 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 single-player interaction with a *Tetris*-like game. Exhaustive search is not computationally tractable in the general case of *It's Alive!* due to the vast number of possible combinations. In response, I apply Monte Carlo Tree Search (MCTS) to find input sequences that approximately maximize the players score.

The strategy behind Monster Carlo is to bring AI techniques to game designers and programmers in the platforms they are already using, rather than attempting to entice them to develop game specific AI-based game design tools for themselves, following the examples of Jaffe and Zook. In the implementation of Monster Carlo, I made an effort to minimize the impact on the game code changes required for integration. With the same intention, I chose to build the tool inside of Jupyter Notebook, which is already being used for gameplay data analysis[6].

In order to test the tool's versatility, I integrated it with a game of a different style and one I did not develop—*2D Roguelike*, the open source tutorial game available through the Unity asset store.[2] Screenshots of these games may be found in Figure 4.1a and Figure 4.4.

Monster Carlo is intended to answer a variety of design questions: How do monster animation (that is "coming to life") conditions affect the achievable high scores

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

in *It's Alive!*? How does an *It's Alive!* player who collects monsters as soon as possible measure up to a player who waits to do it until the last moment? How does the game dynamic change if we increase both the damage dealt by the zombies and health gained from food pick-ups in *2D Roguelike*? How feasible is a no-backtracking player strategy in *2D Roguleike*?

MCTS has many variations. I experimented with different treatment of terminal branches, such as never replaying a terminal branch, thus increasing the number of states explored; or treating a terminal branch no differently, as the classic MCTS does. I also tried out 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 a human player can get. I also added an optimization technique of saving the entire playtrace of each playthrough with a new best score [1]. All of these game-agnostic variations are available through parameters of Monster Carlo.

There are also different treatments for the the way actions are represented in a tree. I experimented with a flat list of all available actions at each state, which resulted in a wide tree; and subdividing the actions by type, resulting in a deeper, narrower tree, as the algorithm would first have to choose between action types and then between the available actions of the type. Because the concept of an action is game specific, it is up to the developers to define the action hierarchy.

In general, using MCTS for playtesting requires the games to be engineered to be compatible with it. The Monster Carlo support library makes it easy to hook

[3]Upper Confidence Bound-1 applied to Trees

into the Unity engine update cycle and makes this engineering easy. I also created data analysis templates for experiments that take the form of comparing optimized scores between variants.

Monster Carlo is a framework for use in Unity. In contrast to Brandon Drenikow's work on Interactive Visualization of Gameplay Experiences [3], it focuses on searching for new gameplay traces rather than visualizing ones resulting from past human play.

This thesis makes the following contributions:

- Definition of a framework for machine playtesting Unity games where instances of the game execute rollouts within the paradigm of MCTS.
- The complementing C# and Python support libraries for adapting a game to support machine playtesting and running experiments.
- A report on initial experiments that validate the framework and answer design questions about an in-development game.

Chapter 2

Related Work

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

2.1 Design Inquiry with Restricted Play

The Restricted Play concept was introduced by Alexander Jaffe and his coauthors in 2012 [5] and was applied to a two-player, perfect-information game *Monsters Divided*. In their evaluation tool, the authors were able to calculate the optimal strategies for each type of restricted behavior. Due to the size of the game (five cards per player), it was possible to apply exhaustive search to the entire game tree, foreseeing every possible playthrough. In their Future Work section, the authors state that MCTS can be a promising alternative for the games whose complexity makes exhaustive search impractical. Further, because of MCTSs agnosticism to games features, it can be used

without modification on different restricted players and game design variants. In this thesis, I use 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.

In Ludocore (prior to Jaffe's work), a logical game engine for modeling video games [9], Smith et al. use similar concepts for analyzing games created within the Ludocore framework. In particular, I borrow from Ludocore the idea of asking design questions by imposing restrictions on the player behavior, and then asking what this constrained player could achieve as variations in the design are considered. The main difference with my work is that Monster Carlo is meant for integration with Unity games, whereas for Ludocore, games had to be encoded in a specially-designed logic programming language which Ludocore's back-end analysis engine could understand. By contrast, games analyzed by Monster Carlo can, for example, 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.

Zook et al. follow up on Jaffe's suggestion to use MCTS for analyzing large games. They reported experiments with the board game *Scrabble* and the *Magic The Gathering* inspired card game *Cardonomicon*[10]. 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, or against other players of the same skill rank. 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 their games. Similarly, even though *Its*

Alive! is a nondeterministic game (as in *Tetris*, the player does not know which piece will be randomly dropped next), I use the determinization strategy as well by simply fixing the games random seed value.

2.2 MCTS

Monte Carlo Tree Search (MCTS) is a heuristic search algorithm, which selectively explores the tree of possible moves [1]. It assess the potential of each move by averaging the scores of simulated random playouts from the current point in the game until the end (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* [4], *Tetris*[2], *Scrabble* [10] and other games. Most recently, it was used in the creation of AlphaGo Zero[8], the latest world champion in *Go*. In the rest of this thesis, I 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 players 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.[1]. I also used this technique, as it is sufficient to answer the design questions Monster Carlo is intended to answer. One of the main differences in my application is that the games described above all had a win/lose condition (even *Tetris*, as it was played competitively), while in the games described in this thesis focus on the highest score attained.

Cai et al. [2] applied a combination of MCTS and a state hash database to create an artificial *Tetris* player on par with Colin Fahey's player[2]—the benchmark at that time. This paper had many useful insights. For example, the introduction of tree pruning dramatically increased the player's performance against the benchmark, even though it doubled the simulation time and thus halved the number of rollouts. They also assigned large negative values to the branches that lead to terminal states in game to discourage further exploration in that area. They experimented with both deterministic and non-deterministic piece sequences, though it was mainly done for the purpose of assessing the usefulness of the state hash table. While some lessons learned could be applied to *It's Alive!* artificial players, due to some fundamental differences between the two games, I could not make efficient use of others. For example, they calculated the number of game states as approximately 10^{60} based on each cell being either occupied or empty. In *It's Alive!*, each cell can have 77 different states—18 different pieces in four orientations, or empty, which leads to approximately 10^{75} possible field states, which

[1] In AlphaGo Zero, where the goal was creating a competent player, the move sequence was not fixed
[2] http://www.colinfahey.com/tetris/

would likely significantly reduce the effectiveness of the state hashing. Further, because in *It's Alive!* the game ends after five monsters are collected, the terminal state is part of the win condition, so blindly assigning negative values to the terminal tree branches would not work in our case.

2.3 Environments that support MCTS

The Video Game Definition Language(VGDL)[7] is a representational language for modeling videogame mechanics and level designs. The General Videogame Artificial Intelligence(GVG-AI)[3] 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[4] 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 of memory data, rather than the simplified game state abstraction used in VGDL. A growing number of environments, including classic games,

[3]http://www.gvgai.net
[4]https://gym.openai.com/

are available for AI experimentation. A related effort, OpenAI Universe,[5] 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 and then clicking elsewhere to end the turn).

All 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 screen shot pixel data and virtual machine states for state representation and low-level keyboard and mouse events as actions. Monster Carlo does not represent game state beyond the sequence of moves needed to reach it, and a way of asking the framework to make a discrete micro-decisions which assemble into high-level actions (more on this in Chapter 3).

2.4 Unity

Unity is a powerful 3D game engine available for free for private use, which makes it popular with indie developers. Unity does not directly support integration with

[5] https://blog.openai.com/universe

MCTS. In order to change this, it is necessary for the game to somehow 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 the 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, 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.

Chapter 3

System Design

The Monster Carlo framework consists of four major parts (Figure 3.1). The integration modifications to the game and specifications for the design experiment (the green parts on Figure 3.1) have to be written by the game designer, while everything else is provided through the Monster Carlo tool.

While MCTS is used as the main AI method for the experiments performed in this paper, the algorithm can be substituted without adjusting the game or the design experiment notebooks.

3.1 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 a copy of the game, and the game must be compiled with the C# support library (de-

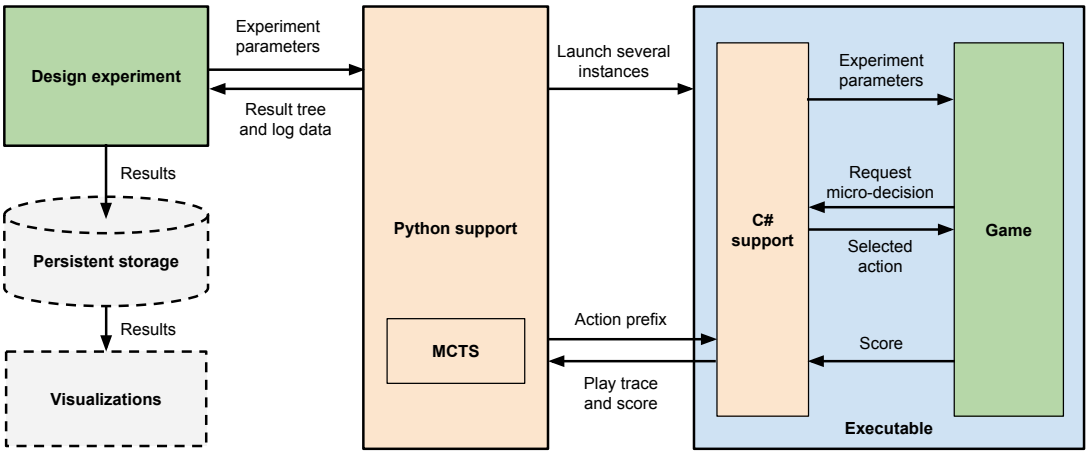


Figure 3.1: Overall architecture of the Monster Carlo framework.

scribed later). This process factory function can be used to pass experiment-specific configuration data to the game, for example, asking it to start in a certain mode optimized for analysis; or arrange for the execution of the game to happen on a different machine, such as on a remote cluster of machines rather than the users personal workstation. Finally, the experiment itself consists of a *for* loop running a desired number of experiments with specified parameters. The experiment results are returned as an object, which can be saved in a pickle file at the end of each experiment and later used for analysis and visualization. I used matplotlib[1] to visualize the results. All the experiment result graphics in this thesis were obtained through this method.

3.2 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

[1]https://matplotlib.org/

tree and any additional data the developer chose to keep track of along the way. This output object can be trivially modified to keep track of various metrics, like the time it takes to perform a rollout, or some kind of game-specific data. In my experiments, I kept track of the growth of the highest seen score over the rollouts, but I could, for example, have as easily kept track of the number of monsters collected during a playthrough. This information can be stored in designated lists outside of the proper tree structure.

The tool supports running multiple instances of the game to significantly speed up the search (see Section 5.2 for the results). The tool takes in arguments for the number of rollouts to perform, and game specific settings, which are passed through to the game instances in the form of environment variables. The optional arguments 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 print out traces along the way.

3.3 C# support module

The support module is a C# file that has to be added to the game project. This module takes in the environment arguments at the start of the experiment and communicates with the Python module through NetworkStream on the port specified in the arguments. The desired game variant is also specified through the arguments. The module receives the most promising path prefix from the Python module at the beginning of each playthrough. Each time a decision must be made in game, the game

tells the module how many legal moves are available, and the module makes a selection 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 makes choices randomly, with support for weighted selection. A custom heuristic can be added in the game by additionally providing an integer array to use as a reference for weighted choice. When the game is over, it provides the final score (in contrast to simply the win/lose result) to the module, which in turn sends the full playtrace, the final score, and any other information the designer may deem important, back to the Python module.

3.4 Modifications to the game

At a bare minimum, the designer must implement micro-decisions and scoring. For this, the game needs to be able to determine legal moves at each step, request the support module for an integer 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, the random seed needs to be reset to the same value for each playthrough for the duration of the experiment. I also recommend creating a headless mode for the game, as it can significantly speed up the playthroughs on some platforms.

The game needs to be able to tell whether it was launched as part of the experiment to replace the user's input with decision requests to the C# module. Launching

the game in the experiment mode can also include skipping menu screens and disabling smooth movements. To optimize the experiment running, I 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 playthrough.

If the designer wishes to conduct Jaffe-style restricted play experiments, they will have to implement the player models (which may remove or limit some available actions or micro-decisions 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. The designer may also choose to implement flat or factored action choices (more on this in section 5.1).

Chapter 4

Experiments

This chapter describes example experiments I conducted using the Monster Carlo framework. They show how to use the restricted play methodology to ask design questions for two games: my in-development game *Its Alive!*, and *2D Roguelike*, a tutorial game provided by Unity that was not implemented with Monster Carlo in mind.

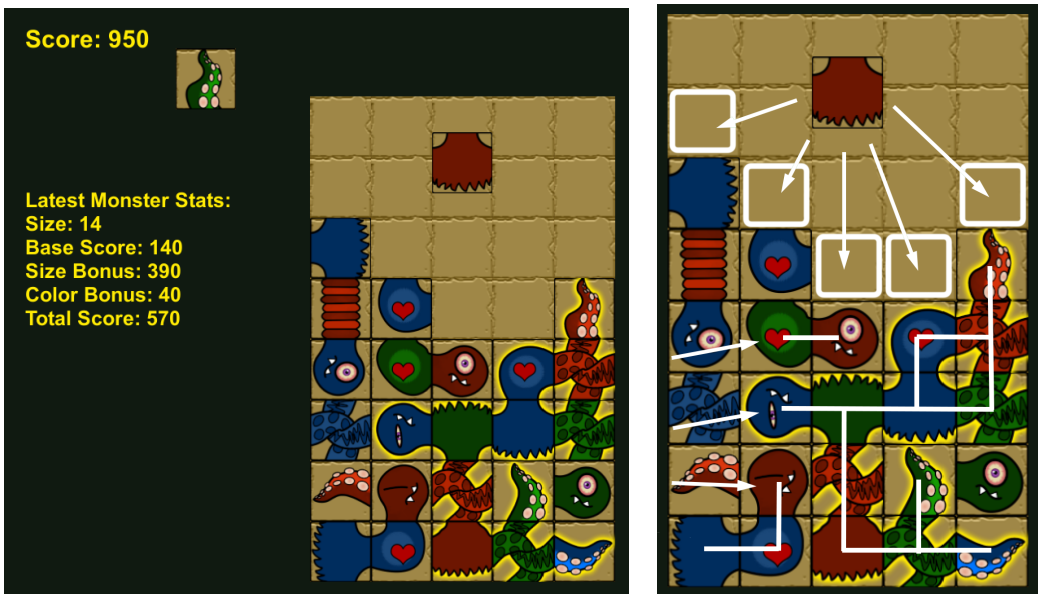
4.1 *It's Alive!*

It's Alive! (see Figure 4.1) is a *Tetris*-style game where the player controls the position and orientation of pieces falling from the top of the board. The game is lost 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 players of this game must form arrangements of pieces that represent grotesquely wiggling 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 for human players 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 actions—rotate, move left, move right, quick-land, cycle highlighter, collect monster. Some of those actions, however, could be repeated indefinitely without affecting the game state, but meaninglessly expanding the scope of the search for Monster Carlo to perform. To avoid this, for the purpose of the experiments, the actions were defined as those that result in 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 has to choose a column and position for landing through micro-decisions. Similarly, cycling the highlighter is not considered an action, instead, deleting any of the living monsters in the current state is considered a legal action, regardless of the highlighter position (Figure 4.1b). With this new definition of action in mind, the player has 20 or more possible actions at every state. That is five possible columns times four landing orientations, plus one action per living monster. On a

5x7 playfield, this makes exhaustive search computationally intractable due to the vast number of possible combinations.

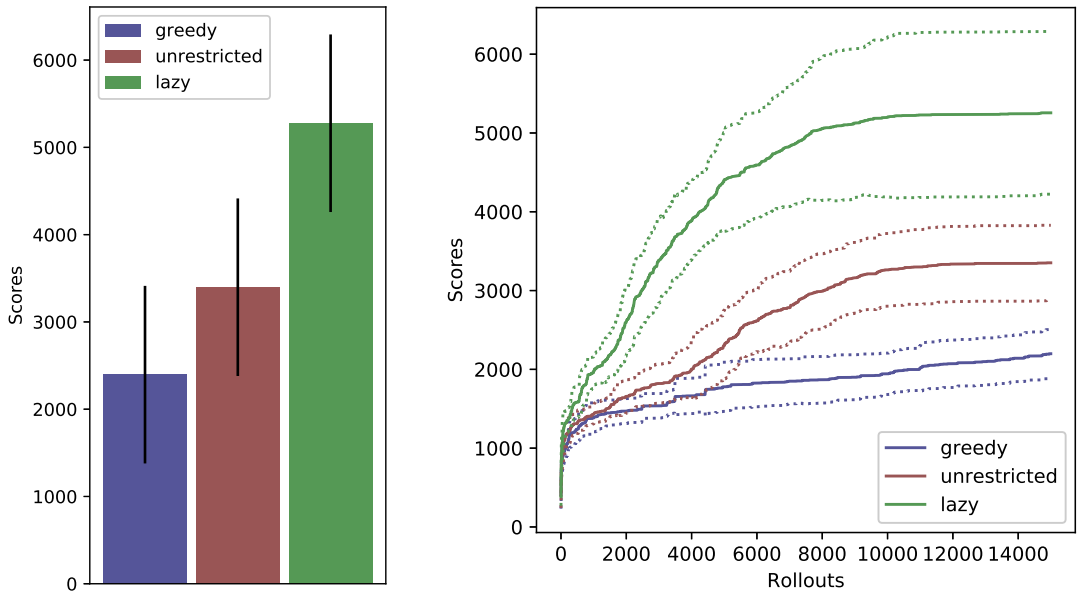


(a) (b)

Figure 4.1: (a) *It's Alive!* screenshot. (b) Possible actions in this state include landing the falling piece in one of the five columns in one of four orientation, or collecting one of the three living monsters.

4.1.1 Playstyle experiments

Like *Tetris*, *It's Alive!* has many quick game over states resulting from piling pieces predominantly in the same column and reaching the ceiling while most of the playfield is still empty. To prevent Monster Carlo from wasting time on these dead-end scenarios, I 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. I 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



(a) (b)

Figure 4.2: (a) The mean and variance of the highest score seen at the end of 30,000 rollouts for 20 independent replicates run with *greedy*, *unrestricted* and *lazy* players in *It's Alive!*. (b) Mean and variance of the highest scores seen across the 20 independent replicates for each playstyle over the first 15,000 rollouts.

required to express this more focused analysis.

I 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 (if I chose column 3): Which orientation should I land the piece in column 3? (more on this in the 5.1 Flat vs. Factored Actions section in the Framework Validation chapter).

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

As is evident from Figure 4.2, the *lazy* player did the 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 according to the Mann-Whitney U test.

Several conclusions can be drawn from these results. For one, deciding when to collect a monster is a meaningful choice for the player. Second, while technically nothing prevented the *unrestricted* player from achieving the same results as the *lazy* player, being presented with an opportunity to collect the monster at every step it is alive misleads the search into local maxima and slows down the 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 in order to gain deeper insight into the impact of playstyle difference that the tool discovered.

4.1.2 Design variants

I considered three design variants. The *regular* design follows the rules outlined above. The *mon-ochrome-ster* design considers two pieces within a monster connected

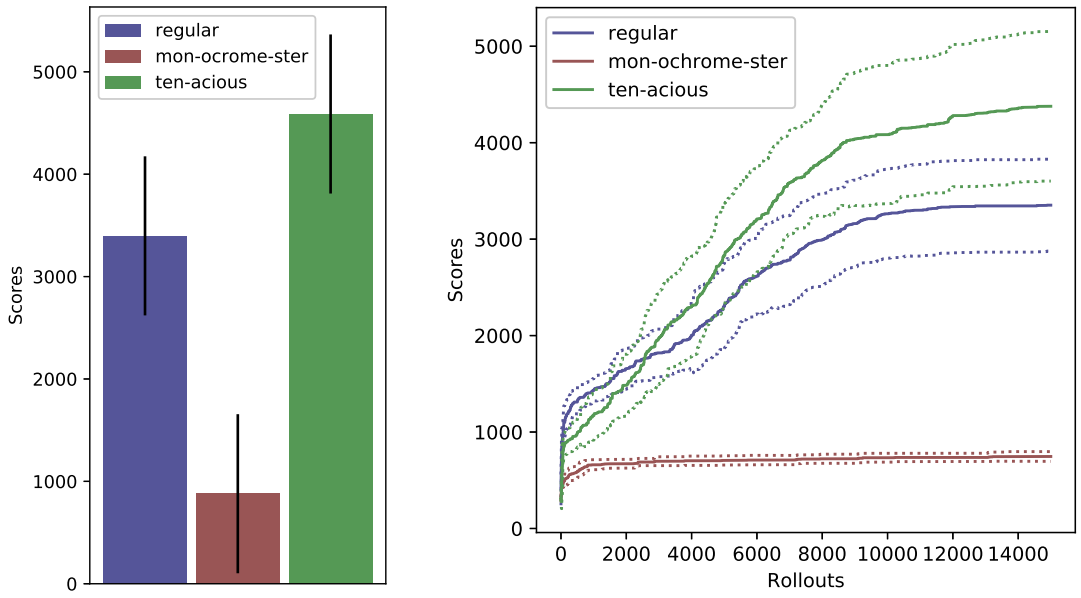


Figure 4.3: (a) The mean and variance of the highest score seen at the end of 30,000 rollouts across 20 independent replicates run on *regular*, *mon-ochrome-ster* and *ten-acious* designs in *It's Alive!*. (b) Mean and variance of the highest scores seen for each design variant across 20 independent replicates over the first 15,000 rollouts.

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.

Several conclusions can be drawn from the results of experiments (Figure 4.3). It is apparent that the *mon-ochrome-ster* mode is much harder than the other two, and affords for a lower maximum score. Counter-intuitively, we see that *ten-acious* design variant, which places a restriction on the player and, therefore, makes for a harder game, led to higher scores than those Monster Carlo achieved in the *regular* design. Both experiments were ran with the same random seed, and 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 4.3b, shows that the *regular* design

scores are higher initially, but are quickly overtaken by those seen in *ten-acious*. I believe this is due to the fact that the restrictions in *ten-acious* prevented the search from lingering in the local maxima created by collecting monsters of smaller sizes. 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.

4.2 *2D Roguelike*

Note that because I am the developer of both Monster Carlo and *It's Alive!*, it is possible that I have over-specialized the framework for analysis of games very much like *It's Alive!*. In this section, I consider the integration effort and results from experiments with a game that I did not make myself, and which I did not consider during the primary development of the Monster Carlo framework.

2D Roguelike (Figure 4.4) is an open source official tutorial game for the Unity game engine.[1] 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

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

Figure 4.4: Screenshot of *2D Roguelike*, the open source official tutorial game for the Unity game engine.

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 doesn't actually move.

4.2.1 Playstyle experiments

For *2D Roguelike* experiments, the actions were factored 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, I 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 higher scores with the statistical significance of p-value = 3.3e-08 according to the Mann-Whitney U test.

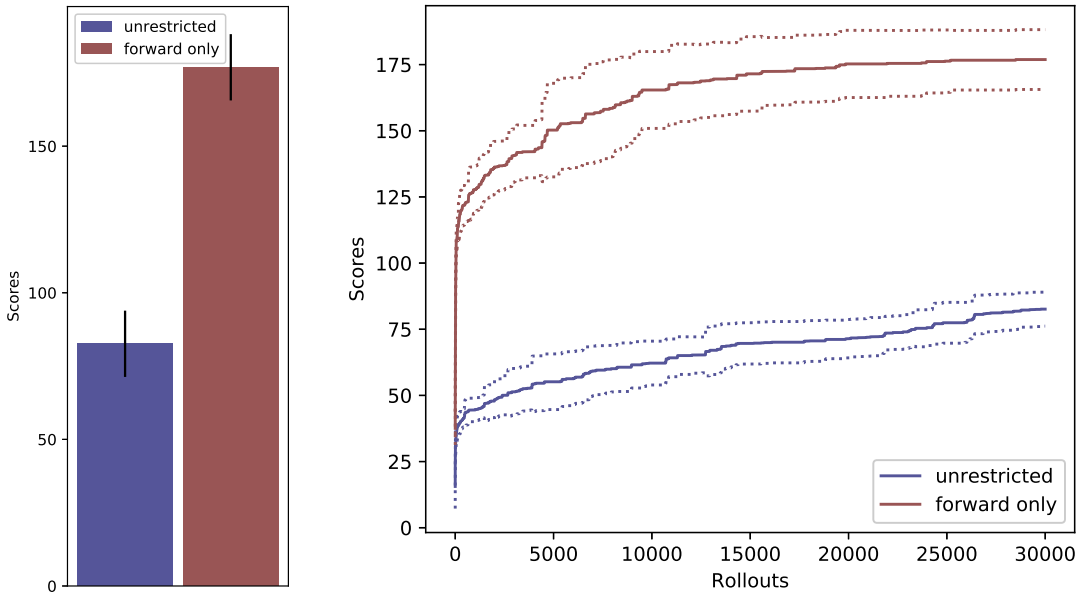


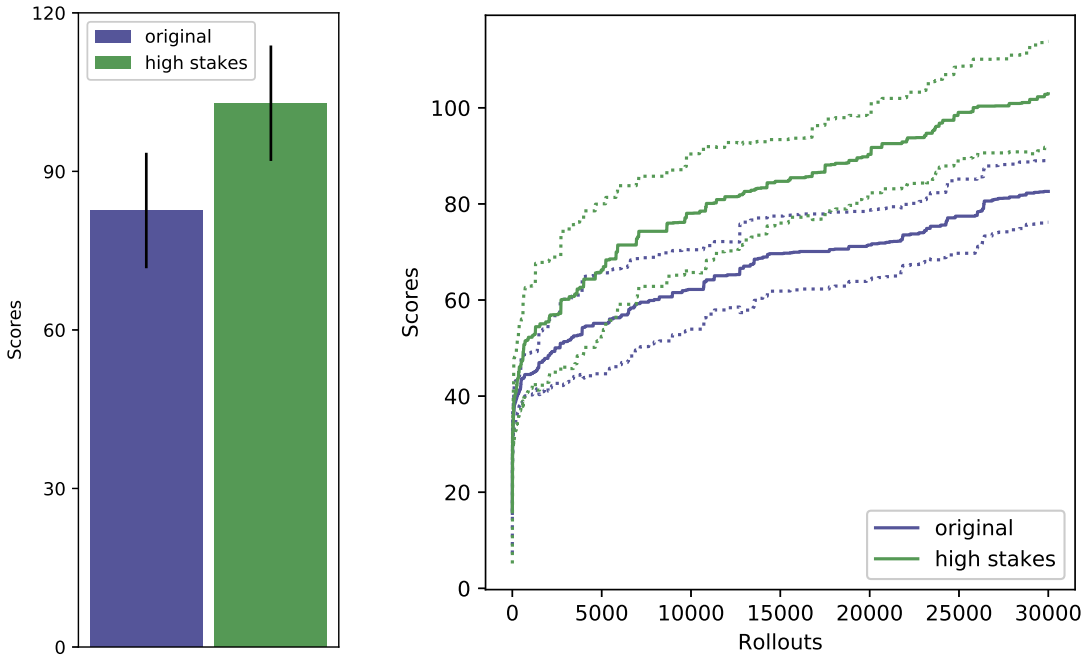
Figure 4.5: (a) The mean and variance of the highest score seen at the end of 30,000 rollouts for 20 independent replicates run with an *unrestricted* and *forward-only* player models in *2D Roguelike*. (b) Mean and variance of the highest scores seen for each design player model across 20 independent replicates over the 30,000 rollouts.

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.

As we can see from Figure 4.5b, while the scores for the *forward-only* player have mostly flattened out, the *unrestricted* player scores are steadily increasing. Given enough time, I 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.

4.2.2 Design variants



(a) (b)

Figure 4.6: (a) The mean and variance of the highest score seen at the end of 30,000 rollouts across 20 independent replicates for experiments run on *default* and *high stakes* design variants of *2D Roguelike*. (b) Mean and variance of the highest scores seen across 20 independent replicates for each design variant over the 30,000 rollouts.

I 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 (Figure 4.6) from this *high stakes* design variant were statistically significantly higher (p-value = 4.8e-07 according to the Mann-Whitney U test). From this, one could

conclude that the *high stakes* variant of the game is easier to play.

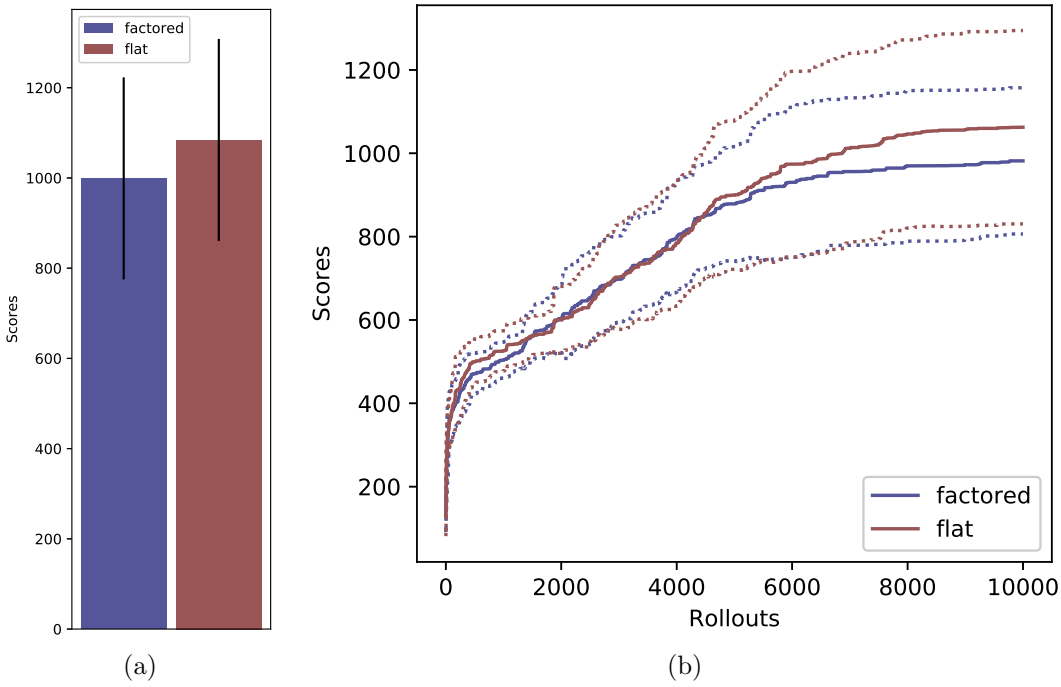
Chapter 5

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 36 in the worst case scenario (not collecting a single monster). This leads to longer rollouts and slower depth-wise exploration rate. For the framework validation experiments I reasoned that having a field of a smaller size would allow me to run experiments faster while still demonstrating relative differences between performance of Monster Carlo with different parameters. I built a smaller version of *It's Alive!* with a 3x5 play field 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.

5.1 Flat vs. Factored Actions



(a) (b)

Figure 5.1: (a) The mean and variance of the highest score seen at the end of 30,000 rollouts acros 20 replicates for experiments run with factored and flat action representations. (b) The mean and variance of the highest seen scores seen across 20 independent replicates for the flat and factored action representations over the first 10,000 rollouts.

In the context of these experiments, an action is defined as a combination of inputs that results in a game state change. For example, in *It's Alive!* the player may move the falling piece left and right or rotate it any number of times, but the field state only changes once a piece has landed. With this in mind, landing a piece in a position x with orientation y is a single action. Likewise, if more than one monster is alive, the player may cycle the highlight between them indefinitely, but the field state only changes once one of them is collected. In this case, a separate collect action exists for

each of the monsters. Since the width of the playfield in this smaller *It's Alive!* is three, and each piece has four orientations, there are always 3x4 = 12 landing actions, and sometimes one or two collection actions. This resulted in a wide and shallow search tree. In addition, since during rollouts actions were selected with equal probability, a monster collection action was five to ten times less likely to be chosen. I chose to factor actions into collection and landing type. I further subdivided the land actions by column. The algorithm would first have to choose if it wanted to land a piece or collect a monster (if one or more were alive). If landing, the algorithm would next have to choose a column, and then an orientation. If collecting, it would choose one from a list of living monsters. This made for a narrower and deeper search tree and made monster collection more likely to occur.

I ran two sets of 20 experiments with 30,000 rollouts each, one with flat actions and one with factored actions. I hypothesised that factoring the actions would lead to higher scores. However, the difference between scores achieved in the two experiments was not statistically significant (p-value = 0.8 according to the Mann-Whitney U test).

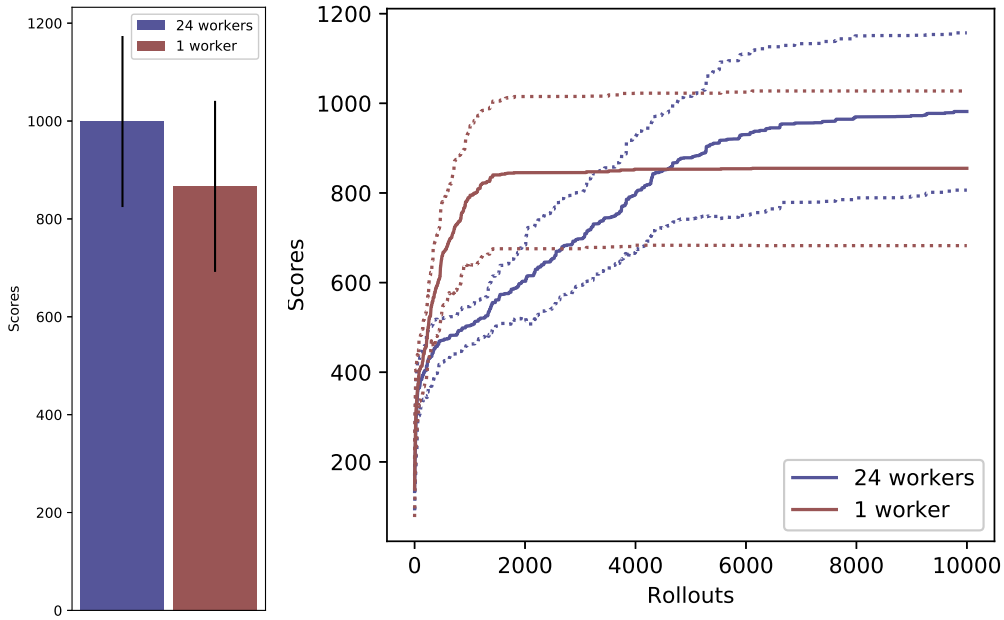
There is an at least partial explanation for this. As described in 4.1 *It's Alive!*, a heuristic was added to prevent the player from making a move leading to a game over state if another move is available. This was done to eliminate many quick death states resulting from blocks being piled up on one column with the majority of the field remaining empty. Because points in *It's Alive!* are awarded both for landing a block and collecting monsters, another trap for MCTS manifested. Monster Carlo would figure out that filling up the screen led to a high score and then would fiddle with rearranging the

last few blocks it placed. With flat actions, the tool rarely elected to collect monsters. Adding factoring, which led to a 50% chance of a monster being collected if at least one was alive, resulted in significantly better scores.

With the addition of the heuristic, the player always collected any living monsters at the end of the game, which largely negated the advantage of factored actions in this case. This same phenomenon was observed in the playstyle experiments for *It's Alive!*(Figure 4.2), where the *lazy* player, who was restricted from collecting monsters until the last moment, outperformed the other versions, demonstrating that being able to collect monsters earlier in the game didn't help MCTS achieve higher scores faster.

5.2 Parallel vs. Single Thread

The classic MCTS updates the tree after each rollout and uses the updated tree to make the decision about 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 even though the results from the other parallel workers are not yet known. I wanted to see whether there was a large drop in the tool's effectiveness at the cost of the speed that parallelization provided. I ran two sets of 20 experiments with 30,000 rollouts. The first set used 24 parallel workers, one for each CPU on the server machine I used, and the second set used a single worker. The results of the high scores achieved are in Figure 5.2. The experiment with 24 workers achieved higher scores with statistical significance of p-value = 0.02 according to the Mann-Whitney

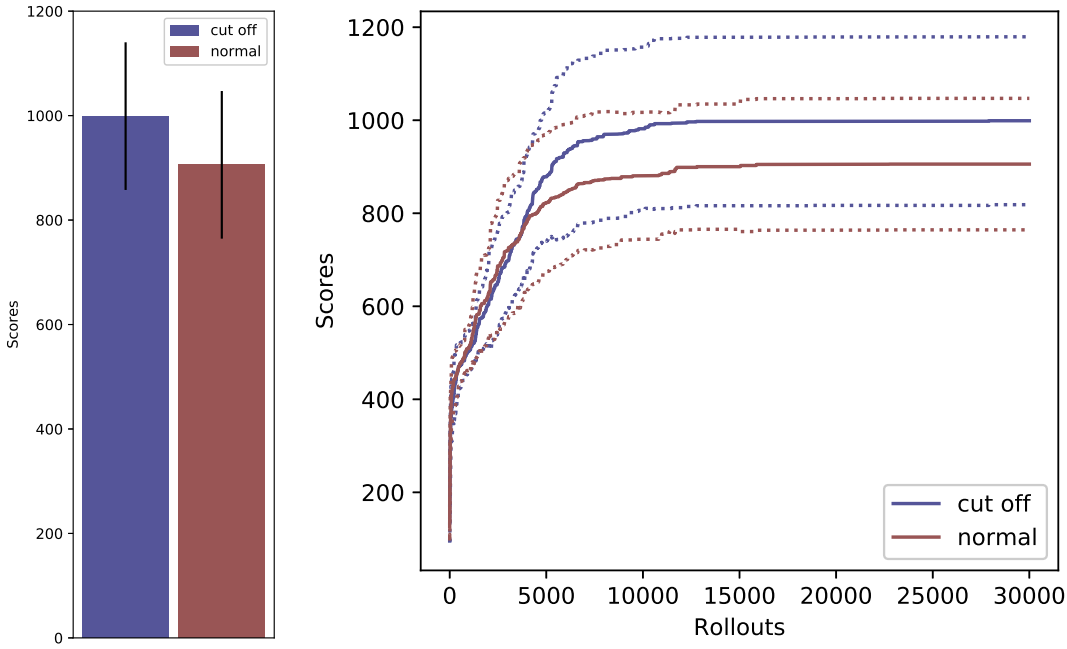


(a) (b)

Figure 5.2: (a) The mean and variance of the highest score seen at the end of 30,000 rollouts across 20 independent replicates for experiments run with a single worker and 24 workers in parallel. (b) The mean and variance of the highest seen scores over the first 10,000 rollouts for the 20 independent replicates ran with 24 or a single worker.

U test. However, the real significance is in the duration of the two experiments. The experiment with a single worker lasted eight hours and 20 minutes, and the experiment with 24 workers took approximately 24 minutes to finish, which is approximately 20 times faster. It is worth noting that initially, the parallel experiment took more rollouts to get to the same scores as the single-thread experiment (Figure 5.2b). This leads me to extrapolate that parallel workers have a destabilizing effect on MCTS. While this initially leads to lower scores, it also makes it less likely for the search to get stuck in a local maximum.

5.3 Terminal Branch Treatment



(a) (b)

Figure 5.3: (a) The mean and variance of the highest score seen for at the end of 30,000 rollouts across 20 independent replicates for experiments run with cutting off terminal branches, or allowing unrestricted repeat explorations. (b) The mean and variance of the highest seen scores over the first 10,000 rollouts for the 20 independent replicates ran with each terminal setting.

I noticed that MCTS tended to get stuck in local maxima, sometimes exploring the same branch over and over again, though much better paths were available. I 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.

I ran two sets of 20 experiments, with 30,000 rollouts each, one set 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

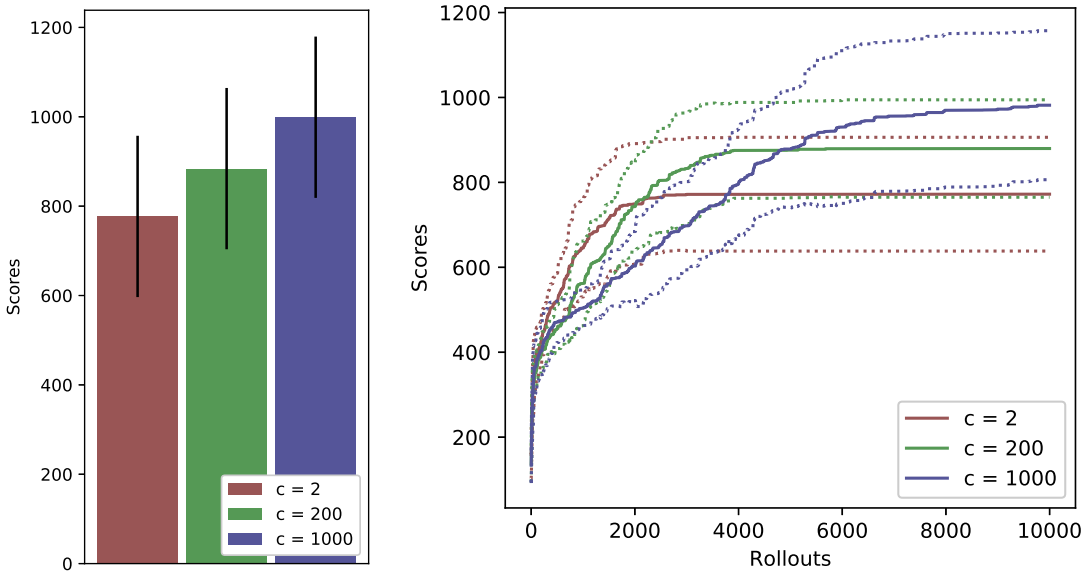
selection.

The results for these experiments (Figure 5.3) showed no statistically significant difference between the highest scores achieved (p-value = 0.06 according to the Mann-Whitney U test) and number of nodes explored (p-value = 0.1). I hypothesize that this is largely due to the fact that the depth of my test game was too great, and so the terminal branch treatment didn't come into play to a significant degree.

However, not revisiting terminal branches allows for exhaustive search on smaller fields. Earlier in this project, I ran tests on *It's Alive!* on a 2x3 grid instead of the regular 5x7. I calculated the maximum score to be 250 points and was surprised to find 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, I was able to verify the fix by running the same test and verifying that no branches scored higher than 250 points.

5.4 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, I 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. I obtained this score by manually playing the game



(a) (b)

Figure 5.4: (a) The mean and variance of the highest score seen at the end of 30,000 rollouts across 20 independent replicates for experiments run with UCT constant set to 2, 200 and 1000. (b) The mean and variance of the highest seen scores over the first 10,000 rollouts for 20 independent replicates with each c value.

with the same random seed.

I ran three sets of 20 experiments with respective UCT constant values set to 2, 200 and 1000. The results in Figure 5.4 demonstrate that Monster Carlo did best with $c = 1000$, which was closer to the expected value of around 1400 points. The results were statistically significant with p-value = 0.0002 according to the Mann-Whitney U test for comparison of $c = 2$ and $c = 1000$, and p-value = 0.01 for $c = 200$ and $c = 1000$. Drawing a parallel with the parallelization experiment, we can see that the higher c experiment increased its score slower over the number of rollouts (Figure 5.4b). However, while the experiment with a lower c got stuck in a local maximum fairly early

on, the scores corresponding to the higher c continued growing, due to the search's higher emphasis on exploration.

5.5 Experiment Speedup Techniques

If the MCTS rollouts were happening at the game's normal speed, each of the aforementioned experiments would take days to complete. I employed a number of speedup techniques in order to be able to run the experiments in a timely manner. The results are summarized in Table 5.1. This translated into an approximately 1600x speedup over the game's base rollout speed (10x for increasing framerate, replacing smooth movement with instant jumps and disabling artificial delays; times 20 for running with 24 workers; times 8 for running on a server-class machine).

The efforts put into experiment speedup should be balanced with the consideration of how many experiments will be performed and how much trust will be put into their results.

Modification	Speed	Work Required	Pros	Cons
`Set Unity's max framerate, eplace smooth movement with instant jumps and disable delays.`	10X	One line to change the application's frame rate. Replace calls to a smooth movement function with a call to instant move. Set artificial delays to zeroes.	Little extra work	Only effective for games with smooth movements and artificial delays
`Run with n parallel workers`	about nX	Changing one parameter in experiment settings	Much faster results, little extra work.	Small drop in search effectiveness. See Parallel vs. Single
`Run on a server-class machine`	8X	Creating separate Linux builds, copying binaries to the server and results back	Much faster	Can only run game in headless mode.
`Run in batchmode / nographics`	Mac: 0.5X PC: 4X	Adding a parameter in the experiment settings	No extra work	Inconsistent results across platforms

Table 5.1: Summary of the experiment speed-up techniques.

Chapter 6

Conclusion

In this thesis I 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.

I 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.

I implemented the concept of factored actions and demonstrated how in some cases they resulted in improvement in search efficiency.

I integrated Monster Carlo with two games: my own in-development game *Its Alive!* and an official tutorial game for the Unity game engine, *2D Roguelike*. The integration with *2D Roguelike* necessitated fewer than 100 lines of code.

I presented results of several experiments I ran on both games, exploring restricted player models and design variations.

Here are some of the things I learned in addition to those described previously.

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. However, larger changes, for example 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 experiments 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, before I discovered most of the MCTS optimizations, I attempted to run experiments on an even larger *It's Alive!* playfield of 6x8. I experimented with an unrestricted player model, and one that couldn't rotate the pieces. I also experimented with lowering the number of different monster colors. The size of the field resulted in a very wide tree that never had a chance to explore very deeply and resulted in chaotic and generally 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 or less stable scores in the same number of rollouts. For example, 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, because 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 was capable of providing information about design variants.

A lot of work remains to be done along the lines of 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 policy networks that can be applied to states that have rarely or even never-yet been explored before. I 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 program. However, even with its current shortcomings, Monster Carlo is capable of providing usable data about the game. With the convenient experimental setup in Jupyter Notebook, my hope is that it can be easily adopted, at least by the makers of the single player discrete state puzzlers, which are a fairly popular genre in mobile games.

Bibliography

[1] 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 Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[2] Zhongjie Cai, Dapeng Zhang, and Bernhard Nebel. Playing tetris using bandit-based monte-carlo planning. In *In Proceedings of AISB 2011 Symposium: AI and Games (AISB 2011)*, 2011.

[3] Brandon Drenikow and Pejman Mirza-Babaei. Vixen: Interactive visualization of gameplay experiences. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 3:1–3:10, New York, NY, USA, 2017. ACM.

[4] Sylvain Gelly and David Silver. Achieving master level play in 9x9 computer go. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1537–1540. AAAI Press, 2008.

[5] Alexander Jaffe, Alex Miller, Erik Andersen, Yun-En Liu, Anna Karlin, and Zoran

Popović. Evaluating competitive game balance with restricted play. In *Proc. of the Eighth AAAI Conf. on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'12, pages 26–31, 2012.

[6] Steve Martinelli. Starcraft ll replay analysis with jupyter notebooks. [Online]. Available at https://github.com/IBM/starcraft2-replay-analysis.

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

[8] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354 EP –, Oct 2017. Article.

[9] A. M. Smith, M. J. Nelson, and M. Mateas. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 91–98, Aug 2010.

[10] Alexander Zook, Brent Harrison, and Mark O Riedl. Monte-carlo tree search for simulation-based strategy analysis. In *Proceedings of the 10th Conference on the Foundations of Digital Games*, 2015.

Appendix A

Example experiment

This example experiment (taken from a Jupyter notebook) launches 8 parallel copies of the *It's Alive!* game (a macOS executable) to run 20 independent replicates of an experiment. In each experiment, 30000 samples are drawn. In a progress callback, a single character is printed to show activity. In the end, the results of all of the experiments are serialized into a pickle for later analysis.

```
import python_module
import subprocess
import random
import os
import json
import pickle
import time

def make_factory(settings):
    def create_game_process(addr, port, nonce):
        env = os.environ.copy()
        env['MC_ADDR'] = addr
        env['MC_PORT'] = str(port)
        env['MC_NONCE'] = nonce
        env['MC_EXP_SETTINGS'] = settings
        name = "./Alive.app/Contents/MacOS/Alive"
```

```
        return subprocess.Popen([name],env=env)
    return create_game_process

def on_progress(tree):
    print(".",end='')

results = {}
for experiment in range(20):
    result = python_module.run(
        make_factory("unrestricted"),
        num_samples=30000, #number of rollouts
        num_workers=8, #number of parallel workers
        callback=on_progress, #this will print a dot
        UCT_constant = 1000,
        terminal_treatment = "CUT_OFF")
    results[experiment] = result
    #the final tree from the MCTS is saved in a pickle file
    file_name = "alive_" + str(experiment) + ".pickle"
    with open(file_name, "wb") as f:
        pickle.dump(result,f)
```

Appendix B

Integration Efforts for *2D Roguelike*

Monster Carlo's C# library, MC_support.cs, was added under Assets/Scripts, and added as a component to the GameManager prefab. For simplicity, I added a namualPlay switch to the GameManager prefab to differentiate between the builds meant for machine testing or human play. Checks for manualPlay were added where appropriate.

In the Awake function of the GameManager.cs I added a call to mcSupport.Connect() and retrieved the designVariant. For this integration, I added options for flat or factored actions. I also set the Applications runInBackground to true, the targetFrameRate to -1 and the QualitySettings.vSyncCount to 0.

I came up with a custom reward function for the game, as the number of days survived wasn't granular enough to guide the search. The reward consists of the sum of the amount of food the player has at the end of each day plus the distance traveled toward the exit on the day of death. I added helper properties and functions to keep

track of this score.

I added a method for resetting the game. This is called when the GameOver condition is triggered. The final reward score is sent to the MC_support and the game is reset, so it can be played again.

The user input is handled inside Player.cs, but MC_support will be calling the shots on this one. At any point, the player can go in one of the four directions. If I want to simply choose one at random, i.e. flat actions approach, I can call MC_support.choose(4) to make a choice for me. However, I can break this up to make MCTS more effective. This is the factored actions approach. First the algorithms makes a choice on whether to move toward or away from the Exit. Intuitively, one can guess that moving toward the Exit more often than not is a good strategy. With this in mind, I call MC_support.choose(2, new Int[2] {1,2}). This will result in MC_support making a weighted choice using SoftMax and favoring 1 over 0. Next, the algorithm selects whether to go vertical or horizontal by calling MC_support.choose(2).

In order to play many games in as short a time as possible, I disabled the games various deliberate time delays or set them to zero. This can be done in the scripts or on the prefabs. In the MovingObject script, a modification is made to instantly move the object to the new position instead of performing a smooth move coroutine. The exact changes I made to the game to integrate it with Monster Carlo are provided below. The changes amount to fewer than 100 lines of code, including support for flat and factored actions.

Add the MC_support script to the project's Assets/Scripts folder.

Changes to GameManager.cs

Add the following properties:

```
public bool manualPlay = true;
public MC_support mcSupport;
public const int FLAT_ACTIONS = 0;
public const int FACTORED_ACTIONS = 1;
public int designVariant = -1;
private bool needFoodReset = false;
public int rewardScore = 0;
private int realRewardScore = 0;
```

Adding the following in the Awake() method:

```
if (!manualPlay) {
     if (mcSupport.IsDriverPresent) {
           mcSupport.Connect();
           designVariant = GetPlayerModel(mcSupport.DesignVariant);
     }
     Application.runInBackground = true;
     Application.targetFrameRate = -1;
     QualitySettings.vSyncCount = 0;
}
```

New helper methods:

```
public void AddReward(int points) {
     realRewardScore += points;
}

private int GetPlayerModel(string variant) {
     switch (variant) {
         case "factored_actions":
             return FACTORED_ACTIONS;
         default:
              return FLAT_ACTIONS;
     }
}

private int GetScore() {
    return instance.realRewardScore;
```

```
}

private void ResetGame() {
    boardScript.ready = false;
    level = 0;
    instance.realRewardScore = 0;
    needFoodReset = true;
    SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex,
        LoadSceneMode.Single);
}
```

At the end of the InitGame() method, add:

```
if (needFoodReset) {
     playerFoodPoints = 100;
     needFoodReset = false;
     instance.rewardScore = instance.realRewardScore;
}
```

Add the following at the beginning of the GameOver() method:

```
if (!manualPlay) {
   mcSupport.SupplyOutcome(GetScore());
   ResetGame();
}
else <the rest of the original GameOver() code>
```

Changes to Player.cs

Add the following properties:

```
private int fieldXcoord = 1;
private int fieldYcoord = 1;
```

Below is the factored actions implementation. In the Update() method, under the line

#if UNITTY_STANDALONE... Add the following:

```
if (!GameManager.instance.manualPlay) {
    try {
        if (GameManager.instance.designVariant
```

```
            == GameManager.FLAT_ACTIONS) {
            switch(GameManager.instance.mcSupport.Select(4)) {
                case 0: //left
                    horizontal = -1;
                    break;
                case 1: //right
                    horizontal = 1;
                    break;
                case 2: //down
                    vertical = -1;
                    break;
                default: //up
                    vertical = 1;
                    break;
            }
        }
        else if (GameManager.instance.designVariant
                 == GameManager.FACTORED_ACTIONS) {
            switch(GameManager.instance.mcSupport.Select(2,
                                        new int[2] {1,2}))  {
                case 0: //moving left or down
                    horizontal *= -1;
                    vertical *= -1;
                    break;
                default: //moving right or up
                    break;
            }
            switch(GameManager.instance.mcSupport.Select(2)) {
                case 0: //moving horizontally
                    horizontal = 1;
                    break;
                default: //moving vertically
                    vertical = 1;
                    break;
            }
        }
    }
    catch (MC_support.ExperimentFinishedException e) {
        Application.Quit();
    }
}
else {<everything before #elif>}
```

At the end of the Restart() method, add

```
fieldXcoord = 1;
fieldYcoord = 1;

In CheckIfGameOver (), add the following above
GameManager.instance.GameOver():

GameManager.instance.AddReward(GetDistance());
```

Add this helper method, which returns distance traveled toward the exit—a good thing.

```
private int GetDistance() {
    int distance = fieldXcoord + fieldYcoord;
    return distance;
}
```

Changes to MovingObject.cs

Inside the Move() method, replace the if(hit.transform == null){...} contents with the following:

```
if(hit.transform == null) {
   if (GameManager.instance.manualPlay)
        StartCoroutine(SmoothMovement(end));
   else
         rb2D.MovePosition(end);
    return true;
}
```

In Prefabs:

Set the Enemy1 and Enemy2 MoveTime (in the Enemy script component) to 0.

Add the MC_support script as a component to the GameManager prefab. Drag and drop the MC_support script into the MC Support slot in the GameManager script

component. Set Manual Play to false. Set Level Start Delay, Turn Delay and Reward Score to 0.

In the Player prefab, Player script component, set Move Time and Restart Level Delay to 0.

Optional: mute the audio sources in the SoundManager prefab.