BizMC: A Lua Framework for running Monte Carlo Tree Search in the BizHawk emulator

William R. Armstrong[1], Markus Eger[1,*]

[1]Cal Poly Pomona, Pomona, CA, USA

Abstract

Applying existing or new AI techniques to games often involves either utilizing or creating an open-source implementation of the game to access its internals, or using the graphical output as an AI agent's input. In this paper, we explore reverse-engineering as a third approach. We describe how an open-source emulator for arcade and console games can be extended to instrument games available only as binaries, and made available for use in AI research. While still requiring some effort for each game, we aim to provide a consistent interface across multiple games that can then be used by AI agents. We show preliminary results of applying this method to three games from different eras and released for different consoles, that are all supported by the underlying emulator.

Keywords

Monte Carlo Tree Search, Console Games, Reverse Engineering

1. Introduction

Classic arcade and console games have long been a staple for game AI benchmarks. While some recent trends move towards playing these games by using the contents of the screen as the input for the AI agent, "like a human player would play", being able to use the games' internal data representation directly is still of interest in the development of new approaches to AI, as well as to test the applicability of existing approaches to new domains. Additionally, while many games present the relevant game state on a single screen, more complex games, such as strategy games, may require navigating different views or menus, or controlling a viewport into the game world. Having direct access to the underlying data may enable development of AI techniques without having to also learn this UI navigation. One AI approach that has successfully been applied to a variety of games, including several strategy games, is Monte Carlo Tree Search (MCTS) and its variations. However, in order to do so, one typically needs to have the target domain in a suitable form, which includes the ability to perform a forward simulation. Existing arcade and console games often do not exist in such a form, and may not be available in source code form, making such an adaptation challenging.

In our work, we present a framework that addresses both sides of this challenge: First, we provide a structured method to instrument existing game code that may only exist in binary form. While this still requires manual effort to disassemble the code and navigate its structure, we provide some guidance on how to ease these efforts, and expose the games in a common structure. Second, we show how this exposed structure can be used as an interface to apply Monte Carlo Tree Search to a variety of these games. While our experiments focus on MCTS and strategy games, our approach enables applying other AI techniques to these same games and compare to this baseline, or to use MCTS for other games running on the same emulator in the future.

2. Background

Arcade and console games draw a lot of attention from the AI research community, because they provide a wide variety of challenges, while being simple enough to be studied in isolation, with Pac-Man perhaps being the most widely studied one [1]. While games are useful to develop, study and evaluate AI techniques [2], care must be taken to not over-specialize [3]. A recent development has therefore been to provide a variety of games in a single environment, such as the Arcade Learning Environment (ALE) [4]. ALE, in particular, utilizes an emulator for the Atari 2600 and provides access to hundreds of Atari games as a development environment for AI techniques. However, game state observations in ALE are only available in two "flavors": Either the agent gets access to the screen (with some optional object recognition/classification), or they get access to the entire contents of the RAM, i.e. the raw bit representation of the game state. While the RAM on the Atari 2600 is only 128 bytes/1024 bits in size, making this state rather compact, this approach would not scale well to more modern consoles. The Nintendo Entertainment System (NES), in comparison, already had 2kB of RAM,

Workshop on Experimental AI in Games

*Corresponding author.
wrarmstrong@cpp.edu (W. R. Armstrong); meger@cpp.edu (M. Eger)

while the Game Boy Advance (GBA) uses a more complex architecture consisting of 32kB of RAM plus another 96 kB of video RAM. When trying to apply AI techniques to the game state directly, these sizes become prohibitive, and require an additional layer that extracts the relevant information from within this sea of bytes. Unfortunately, this requires significant effort. Our approach aims to fill this gap and provide means to create this layer. As an alternative to using an existing implementation of a game, approaches have been proposed to describe the games of interest in a dedicated language, such as the Video Game Description Language (VGDL, Schaul 5). By having the game description in a common format, an AI agent can access the internal game state via the VGDL interpreter. This, of course, requires porting each domain of interest to VGDL, which may not support all necessary features, and requires significant manual effort. There has been interest in generating VGDL rules [6], and it may be feasible to use a similar approach for game porting, but this remains an open problem. Our approach addresses this gap by providing a structured way to instrument existing console games in an open source emulator. On one hand this enables the use of significantly more complex games for game AI research, including games released on the NES and the GBA. On the other hand, while the process is not entirely automated, it requires significantly less effort than a complete rewrite or reverse engineering of the game, by focusing on the aspects that are most relevant to the AI agents.

One notable family of algorithms for game AI agents are tree search approaches, which have been applied to games almost since the beginnings of modern AI research, with one of the earliest AI programs ever written being Samuel's checkers agent [7]. One of the challenges tree search approaches encounter is that the tree grows exponentially with the number of available actions. Samuel used a variant of what is called alpha-beta-pruning to remove branches from the tree that are not relevant, but even with such pruning computing "pure" game trees is not feasible for most games. Abramson combined a Monte Carlo approach with game tree search, resulting in what is now known as Monte Carlo Tree Search (MCTS), which we will describe in more detail below [8]. This approach has become highly successful, leading to the development of AI agents for games ranging from Backgammon [9] to Go [10], often augmented by other techniques such as Neural Networks. MCTS is so ubiquitous that many game AI frameworks provide it as a baseline, including aforementioned ALE, or systems like TAG [11]. Our approach, similarly, uses MCTS to demonstrate how our instrumentation of the games can be used to develop AI agents and provide a baseline for future work.

BizHawk [12] is an open-source, multi-target emulator that supports over 20 consoles including the Atari 2600, the NES, the GBA, and the Nintendo 64. It supports scripting in Lua, which we utilized to implement our instrumentation code. It has also been used in academic research, with agents learning to play games using the pixels on the screen [13, 14]. As noted above, the goal of our approach is to provide access to the actual game state, in order to facilitate development of AI methods that do not work off of pixel-input. In particular, BizHawk originates in, and is popular with, the Tool-Assisted Speedrun community, and our work was motivated in part to find better strategies for complex games that could be used in a speed run.

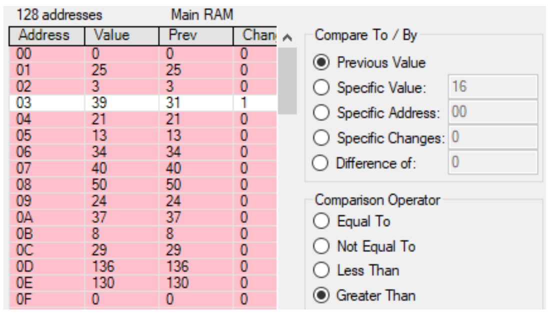
3. Approach

Our approach consists of two parts: First, we need to instrument a target game to access its internal game state. Second, we use this game state representation to perform a Monte Carlo Tree Search to find the best possible option. The approach is designed modular, so that adding a new game only consists of extracting its game state. While this process is not automated, we will describe several aides we provide and utilize in order to speed up some of the more tedious tasks.

3.1. Game Modules

A BizMC game module requires the following function definitions:

- **perform**(string action): instructs the game to perform a specific action given its string description. These string names are descriptive and encode specific parameters to the action. For example, perform("Move_1_2_3") may describe moving Unit #1 to coordinate position 2, 3.
- **expand**(node N): reads the current memory state of BizHawk to determine all of the actions that can be performed, then populates a list of child nodes associated with these actions.
- **rollout**(): performs a rollout strategy for the MCTS which plays the game until some end state has been reached. For most of the modules developed, the rollout function simply selects random actions, but more sophisticated behavior may be defined.
- **score**(): returns a numerical valuation of how "good" the current game state is. For example, if the goal is to find a battle strategy that acquires the most gold, then this function would simply retrieve that value from memory. If the goal was instead to complete the battle in the shortest amount of time, it would return a value based on the emulator's current frame count.

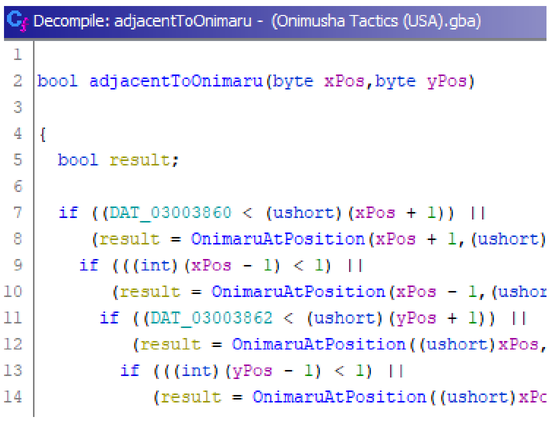

128 addresses Main RAM

Address	Value	Prev	Chan
00	0	0	0
01	25	25	0
02	3	3	0
03	39	31	1
04	21	21	0
05	13	13	0
06	34	34	0
07	40	40	0
08	50	50	0
09	24	24	0
0A	37	37	0
0B	8	8	0
0C	29	29	0
0D	136	136	0
0E	130	130	0
0F	0	0	0

Compare To / By
[✓ Previous Value]
Specific Value: 16
Specific Address: 00
Specific Changes: 0
Difference of: 0

Comparison Operator
Equal To
Not Equal To
Less Than
[✓ Greater Than]

Figure 1: A screenshot of the *BizHawk* RAM Search window


Decompile: adjacentToOnimaru - (Onimusha Tactics (USA).gba)

```

bool adjacentToOnimaru(byte xPos,byte yPos)

{
  bool result;

  if ((DAT_03003860 < (ushort)(xPos + 1)) ||
     (result = OnimaruAtPosition(xPos + 1,(ushort)
    if (((int)(xPos - 1) < 1) ||
       (result = OnimaruAtPosition(xPos - 1,(ushor
      if ((DAT_03003862 < (ushort)(yPos + 1)) ||
         (result = OnimaruAtPosition((ushort)xPos,
        if (((int)(yPos - 1) < 1) ||
           (result = OnimaruAtPosition((ushort)xPc
```

Figure 2: A screenshot of the *Ghidra* Decompiler used to examine a function in *Onimusha Tactics*

In addition, it is necessary to define when the game has reached an end-state (either success or failure). BizHawk allows for assigning callback functions to a particular breakpoint such that they are called when a particular game instruction is called. By locating instructions that are associated with such a state - for example, an instruction to play a specific sound effect when the player character has died, or conversely, an instruction to trigger the ending cutscene if the player has won - the callback function will set an end flag visible by the MCTS.

In order to define these functions, it is necessary to reverse-engineer relevant memory and code addresses from the game. This is typically the most time-consuming part of developing modules and requires the most specialized knowledge. There are several utilities which ease in the process of locating these addresses:

- **RAM Search**: This is useful for locating a specific memory address by filtering RAM values based on specific criteria, as shown in figure 1. For example, locating the address for the player's x-value can be done by moving the character right, then filtering for which values have increased since the last scan.
- **Debugger**: This allows for locating relevant segments of code by setting breakpoints, which allow execution to pause when a specific memory address is written to or read.
- **Hex Editor**: Allows for direct manipulation of in-game memory values. This is another useful tool for identifying memory addresses through trial-and-error, as the effects of these changes will be immediately noticeable.
- **Code-Data Logger**: Outputs a full listing of instructions executed during a time interval to an external file, as wall as all register values and CPU flags. This helps in identifying the entry points of in-game functions and for identifying every single time a particular memory address was read or written to during that time.

In addition to BizHawk, the built-in debuggers for the emulators *fceux* and *Stella* were used to assist in reverse-engineering for the NES and Atari 2600, respectively. *Ghidra*, an open source reverse-engineering tool shown in figure 2, [15] also proved to be extremely useful in static code analysis for developing our modules, as it has a powerful decompiler which converts bytecode back to C-like source code, which is extremely useful for comprehending control flow. Custom loaders have been written for platforms like Game Boy Advance [16] which will properly label memory regions and set a correct entry point when a GBA ROM file is loaded.

3.2. Monte Carlo Tree Search

Monte Carlo Tree Search is a search algorithm which determines the best action to be taken in a game by iteratively building and updating a tree of game states. Each iteration consists of a) selecting moves, b) expanding nodes, c) rollout (simulating the remainder of the game), and d) back-propagation (updating the tree with node values). These roughly correspond to the four required functions of a BizMC module: *perform* creates the appropriate game state after a move has been selected, *expand* generates the child nodes (possible moves) that can be performed from this new game state, *rollout* defines the logic for determining moves in the rollout phase, and *score* generates a value to be propagated back up the tree.

Specifically, BizMC uses a variant, the MCTS with Partial Expansion. Unlike the standard MCTS, this algorithm allows for the search to be performed without needing to fully expand every node as it is encountered. This variant is preferred for games with very high branching factors (average number of possible moves per turn), as is the case in most of the strategy games we developed modules for. Each time a new node is created in the tree, a *savestate* is created in BizHawk and saved in that node: this is a "snapshot" of the gamestate at that point. Whenever BizMC performs an action associated with a child of that node, the savestate is loaded, restoring the gamestate in BizHawk.

In selection, the value of a potential child is calculated for children that have been visited before as:

$$\texttt{node.val} + \sqrt{2} \cdot \sqrt{2 \log \left(\frac{\texttt{parent.visits}}{\texttt{child.visits}} \right)},$$

and for children that have never been visited as:

$$\frac{\texttt{parent.val}}{2} + \sqrt{2} \cdot \sqrt{2 \log \left(\frac{\texttt{parent.visits}}{\text{\# of child nodes}} \right)},$$

Pseudocode for BizMC's implementation of the algorithm is as follows:

```
for $i = 1, n$ do
    curNode = root
    while curNode has a savestate do
        if curNode has not been expanded then
            load curNode's savestate in BizHawk
            populate curNode's list of child nodes
        end if
        curNode = select(curNode)
    end while
    if the game has not yet ended then
        load curNode's parent state
        perform the action associated with curNode
        create savestate and assign to curNode
    end if
    if the game has ended then
        mark curNode as terminal
    else
        perform rollout phase until game has ended
    end if
    result = game.score()
    curNode.update(result)
end for
return best node
```

4. Results

While adding more games and refining our approach is still an ongoing effort, we have already implemented modules for three different games: Escape from the Mindmaster (1982) for the Atari 2600, Romance of the Three Kingdoms II (1991) for the NES, and Onimusha Tactics (2003) for the Game Boy Advance. Note that we split the module for Romance of the Three Kingdoms II into two parts, one for the “Recruit”-part of the game and one for the “Battle”-part, as they have significantly different actions and objectives.

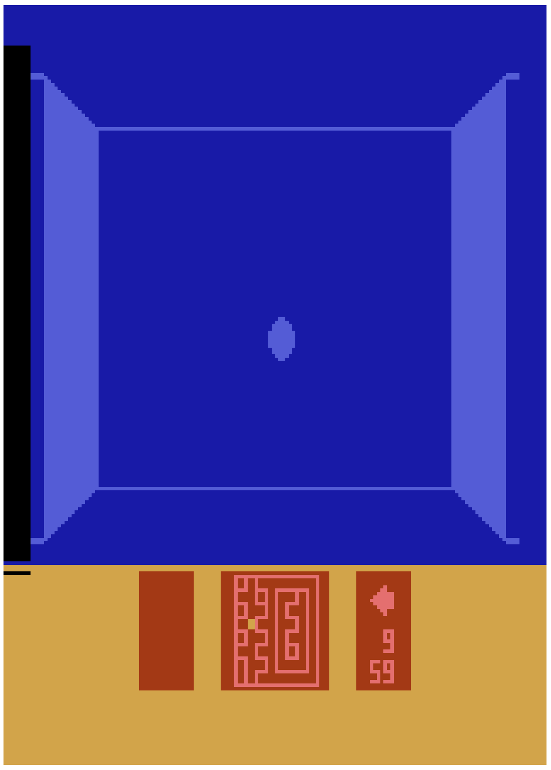


Figure 3: A screenshot from *Escape from the Mindmaster* (1982) for the Atari 2600. Note the map of the “maze“ in the lower part of the screen.

4.1. Escape from the Mindmaster

In each stage of this simple 3D maze game, shown in figure 3, four uniquely-shaped objects are to be returned to their identically-shaped holes before exiting the stage; the locations of both the objects and their holes are randomized at the start of each stage: additionally, in each stage there is a monster whose behavior is similarly randomized, and colliding with it results in a game loss. As only one object can be carried at a time, the moves possible at each node include grabbing an object and placing it in its corresponding hole as a combined action, with the optimal movement from one location to another determined using a pathfinding algorithm. As movement is in three dimensions, the player character must take time to turn left or right where necessary: they may travel backwards if it is faster to do so, with the caveat that doors cannot be traversed in reverse. Because the monster may cross our paths as we attempt to move across the map,

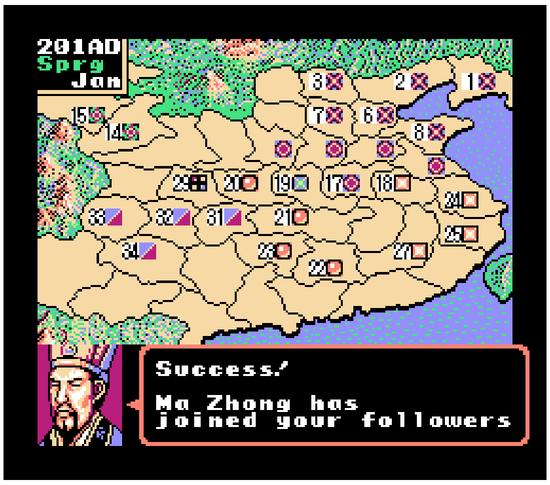


Figure 4: A screenshot from *Romance of the Three Kingdoms II* (1991) for the Nintendo Entertainment System, showing the recruitment part of the game

an additional action, **Wait**, is generated, which waits one frame: this allows us to occasionally manipulate the monster's movement out of our way. BizMC was easily able to find the fastest order of objects to place in each stage, rarely needing to use the Wait action more than a few frames per stage in its optimal solution to a level.

4.2. Romance of the Three Kingdoms II: Recruit

In this historical strategy game, the player may have each of their Generals in the active province perform one action per turn: one such action is to attempt to recruit a General from an enemy nation to their side using one of four possible methods (Special, Horse, Gold or Letter), with Special being a unique method available only to their Leader. Each of these four methods uses a complex formula to determine the percent chance of this action succeeding, based on the statistics of both the Acting and Targeted Generals. The game uses a pseudo-random number generator: An initial seed value for the Random Number Generator (RNG) is set based on which frame the player starts the game, and its internal state only advances when it is polled; thus, in a given game state, we can tell which Recruit actions will succeed based on whether or not the values generated by the corresponding formulas are greater than the current RNG value mod 100. The string name of each Recruit action generated for each node will have the format ACTING-TARGET-METHOD, and only actions that we predict will succeed need to be generated. However, not all Recruit actions will actually succeed; as the acting general travels to the target general's province, there is a certain chance that they will be captured in each intermediate province they visit. Thus, not only do some Recruit actions fail (and are thus rejected), but the number of times that the RNG value advances after a Recruit action varies as it is dependent on the number of provinces visited. This means that there are frequently cases where there are no viable Generals to be recruited: for these cases, we also provide another action, **Wait**, which performs a menu action that causes the RNG value to advance without any other effect on the game state. This allows BizMC to manipulate the RNG value in order to achieve more difficult recruitments.

In this module, our objective function is the combined relative strength of all of the recruited Generals. For this purpose, we define the value of a general as the sum of three of its crucial statistics: Intelligence, War, and Charm. The process of performing each Action in this module is normally time-consuming, as it requires selecting through several consecutive menus: thus, a shortcut is used which bypasses normal game function and executes the desired function immediately. This game is unusual in that it runs a virtual machine atop the native 6502 architecture: by directly altering game memory in BizHawk, we are able to alter the VM's program counter and stack values, effectively calling the function with the appropriate parameters. Using this method, BizMC was able to quickly find a sequence of Recruit actions resulting in a set of extremely valuable General acquisitions. As a demonstration of the deterministic behavior of the Random Number Generator, several of these sequences were replicated with human input on a physical console: this helps confirm our claim that the shortcut taken by BizMC did not cause the game's functionality to diverge from what would happen had it run "normally". Unlike in some other work [17], we actually embrace this exploitation of the RNG, *if* it is useful in actual game play, and/or in a speedrun.

4.3. Romance of the Three Kingdoms II: Battle

The game's battle sequences take place on a hexagonal grid occupied by both enemy and allied Generals, as shown in figure 5. As an invading army, the goal is to win in 30 in-game days, either by defeating all of the enemy Generals or by occupying (moving onto) their capital. Each General has a life total and is defeated when that total reaches zero. The set of moves that can be performed by each General on each day are to **Wait**, **Move** to an adjacent tile, **Attack** an adjacent general, **Recruit** an enemy general (similar to the previous module), and **Fire**, which attempts to set a fire which may spread across the map in a random fashion, influenced by wind direction. Waiting passes the turn, but is sometimes

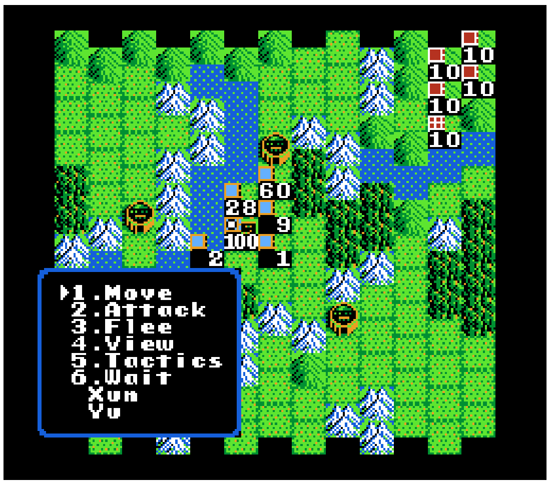


Figure 5: A screenshot from *Romance of the Three Kingdoms II* (1991) for the Nintendo Entertainment System, showing the battle-part of the game

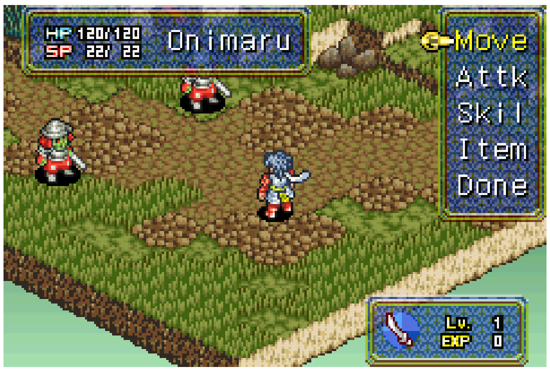


Figure 6: A screenshot from *Onimusha Tactics* (2003) for the Game Boy Advance.

necessary because it builds up Stamina necessary to cross certain terrain types.

The battle module was tested on several battlefields with varying army sizes, and BizMC was generally able to locate a solution when the invading army had a significant strength advantage over the defending army, taking a direct path to the capital and defeating the enemy Commander through direct attacks. In many cases, it made use of the Recruit option to find opportunities to steal enemy Generals, balancing power further in its favor. When the strength difference was less substantial, however, BizMC still managed to win some battles through a less orthodox method. Normally, capturing the enemy capital to end the battle prematurely is difficult because the enemy commander will remain fortified in it until defeated; however, if a fire spreads onto the capital, the commander will be forced to move to survive. This gives the player a brief opportunity to move their own General onto the capital just as the fire has passed. Because the fire's movement is heavily unpredictable, engineering such a situation is impractical in normal human play. However, like in the Recruit module, the Battle module had methods of controlling the random number generator to its favor. Actions like Fire and Recruit advance the RNG value even if they are not successful: in battles where BizMC won through this method, it can be seen performing these seemingly pointless actions as a way of manipulating RNG behind the scenes.

This module also utilized a unique rollout function that makes use of AI logic already written into the game. During each battle, the game alternates between the attacking and defending army, and branches into different code depending on whether the given army is player or CPU controlled. By using Lua to manipulate control flow, it is possible to allow the game to have AI logic take over for the player's army during the rollout phase by temporarily overriding a branch instruction. While the AI is not particularly strong in this game, it is an obvious improvement over random execution, and having the army controlled in this way since AI units execute their actions quickly without needing to traverse through menus.

4.4. Onimusha Tactics

In this strategy RPG, shown in figure 6, the goal of each stage is to defeat all enemies on the screen or defeat a given Boss enemy. The movement is over a 2D plane, but tiles vary in height to give the impression of 3D space. The exact tiles a unit can move to or attack is dependent on the geography: a unit cannot move directly to a tile of a significantly different height, and units with ranged attacks (e.g. bows) have the best attack range at high elevations, but any solid objects in the way will block their attack.

If a unit is not currently selected, the only available action is **Select Unit** (selects a unit who has not used up their turn); otherwise, the main menu actions for the current unit are **Move**, **Attack** and **Special Move**, each of which generates multiple possible actions with string names ACTIONTYPE-X-Y, specifying the target coordinates for the action. Special Moves are actions unique to a specific character which have a special attack range, but use up ability points and thus can only be used a few times; these were generally not useful for the first few stages of the game BizMC was tested on. One more action, **Issen**, randomly triggers and replaces **Special Move** on the menu: by selecting this, the next time the unit is attacked they will preemptively respond with a powerful counterattack.

Because of the large search space, it was necessary to put limits in place as to which actions units took to prevent behavior unlikely to improve the game state, namely running into a corner and avoiding engaging the enemy. Thus, behavior is governed as follows: if an enemy unit is able to be attacked this turn, the unit must choose an attack that does so, moving into position first if necessary; otherwise, they must move into a position that places them as close to an enemy unit as possible. Predictably, BizMC did well on stages where there were only a few units to control but struggled with larger party sizes, even when units' actions were restricted in this manner. Two different objectives were tested: fastest completion, and most experience gained. Since experience and levels carry over between stages, we reasoned that perhaps it was a better goal to choose a set of actions that made the units collectively stronger in preparation for the next stage. As each attack and special move grants players experience, this changed the behavior of units slightly. With the speed objective, the game often manipulated the random Issen counterattack to quickly kill enemy units, but this resulted in less net experience gained. In the experience objective, enemies were instead taken down by a series of weaker attacks, which collectively gained a larger amount of experience. In addition, some units have special moves which grant temporary stat boosts to other units or heal them. While these were not useful in the speed objective, they were used several times in the experience objective as a way to gain additional experience.

5. Conclusion and Future Work

While BizMC provides a useful interface for testing AI techniques on arbitrary games, adding additional games still requires significant reverse-engineering effort. While we have already developed some methods of easing this process, we are still working on creating better tools for finding the appropriate RAM and ROM addresses necessary for its modules to function. As our focus has been the reverse-engineering process itself, Monte Carlo Tree Search was used as a reasonable baseline to test with, and – as we show above – performs well on the games we have investigated so far. Nevertheless, by exposing a common interface to the games, we want to enable the application of other techniques in the future as well.

References

[1] P. Rohlfshagen, J. Liu, D. Perez-Liebana, S. M. Lucas, Pac-man conquers academia: Two decades of research using a classic arcade game, IEEE Transactions on Games 10 (2017) 233–256.

[2] J. Togelius, Ai researchers, video games are your friends!, in: International Joint Conference on Computational Intelligence, Springer, 2015, pp. 3–18.

[3] S. Whiteson, B. Tanner, M. E. Taylor, P. Stone, Protecting against evaluation overfitting in empirical reinforcement learning, in: 2011 IEEE symposium on adaptive dynamic programming and reinforcement learning (ADPRL), IEEE, 2011, pp. 120–127.

[4] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, The arcade learning environment: An evaluation platform for general agents, Journal of Artificial Intelligence Research 47 (2013) 253–279.

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

[6] T. S. Nielsen, G. A. Barros, J. Togelius, M. J. Nelson, Towards generating arcade game rules with vgdl, in: 2015 IEEE Conference on Computational Intelligence and Games (CIG), IEEE, 2015, pp. 185–192.

[7] A. L. Samuel, Some studies in machine learning using the game of checkers. ii—recent progress, IBM Journal of research and development 11 (1967) 601–617.

[8] B. D. Abramson, The expected-outcome model of two-player games, Ph.D. thesis, Columbia University, 1987.

[9] F. Van Lishout, G. Chaslot, J. W. Uiterwijk, Monte-carlo tree search in backgammon, in: Computer Games Workshop, 2007.

[10] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Mastering the game of go with deep neural networks and tree search, nature 529 (2016) 484–489.

[11] R. D. Gaina, M. Balla, A. Dockhorn, R. Montoliu, D. Pérez-Liébana, Tag: A tabletop games framework, in: AIIDE Workshops, 2020.

[12] TASVideos, BizHawk, https://github.com/TASVideos/BizHawk, 2017. Accessed: 2022-07-25.

[13] S. Pham, K. Zhang, T. Phan, J. Ding, C. Dancy, Playing snes games with neuroevolution of augmenting topologies, in: Proceedings of the AAAI Conference on Artificial Intelligence, volume 32, 2018.

[14] H. Ho, V. Ramesh, E. T. Montano, Neuralkart: A real-time mario kart 64 AI, 2017.

[15] NSA, Ghidra, https://github.com/NationalSecurityAgency/ghidra, 2019. Accessed: 2022-08-10.

[16] pudii, gba-ghidra-loader, https://github.com/pudii/gba-ghidra-loader, 2020. Accessed: 2022-08-10.

[17] J. Clark, D. Amodei, Faulty reward functions in the wild, Internet: https://blog. openai. com/faulty-reward-functions (2016).