ICGA Journal 42 (2020) 14–37
DOI 10.3233/ICG-200144
IOS Press

2048-like games for teaching reinforcement learning

Hung Guei *, Ting-Han Wei * and I-Chen Wu **
Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan

Abstract. 2048-like games are a family of single-player stochastic puzzle games, which consist of sliding numbered-tiles that combine to form tiles with larger numbers. Notable examples of games in this family include Threes!, 2048, and 2584. 2048-like games are highly suitable for educational purposes due to their simplicity and popularity. Numerous machine learning methods have been proposed to play 2048-like games; the application of these techniques can help students gain first-hand experience in implementing machine learning algorithms. This paper proposes a guideline for using 2048-like games for teaching reinforcement learning and computer game algorithms, while also summarizing our experience of using 2584 and Threes! as pedagogical tools that were well received judging by student feedback in two graduate level courses.

Keywords: 2048, Threes!, computer science, reinforcement learning, pedagogy, education

1. INTRODUCTION

2048 is a single-player stochastic puzzle game introduced as a variant of 1024 and Threes! (Cirulli, 2014). 2048 is easy to learn and to play reasonably well, yet mastering the game is far from trivial. Many machine learning methods have been applied to 2048 such as the well-known temporal difference learning, a kind of reinforcement learning, together with n-tuple network (Szubert & Jaśkowski, 2014; Yeh et al., 2017; Jaśkowski, 2017; Matsuzaki, 2019). As a teaching tool, 2048's popularity can increase student engagement, while the existing machine learning methods for it provide a well-established basis to educate from.

2048 can be a good tool for teaching basic programming and simple AI design (Neller, 2015). In a preliminary work of this paper (Guei et al., 2018), we have summarized our experience of using 2584 only as a pedagogical tool for teaching reinforcement learning and computer game algorithms. This paper includes the detailed analysis of the use of two 2048-like games, 2584 and Threes!, and discusses the advantages of using 2048-like games as teaching materials. In addition, we propose a guideline for using 2048-like games as a pedagogical tool for beginners, summarizing the experiences and listing the differences between using 2584 and Threes! in the curricula of 2017 and 2018 for two consecutive classes of graduate level students. The courses are designed as a series of projects with a clear goal to develop a strong AI programs for 2048-like games. Students are required to gradually improve their programs while learning new knowledge. The requirements are not complicated, students can even try their own ideas on their projects. Therefore, the courses not only provide beginners with an intuitive way to learn reinforcement learning and computer games effectively, but also motivate them to challenge higher performances.

From the experiences in the courses, 2048-like games are also shown to be good candidates for beginners to learn both temporal difference learning and computer games for its simplicity and popularity.

*Equal contribution.
**Corresponding author. E-mail: icwu@aigames.nctu.edu.tw.

Due to the simplicity, reinforcement learning methods can be easily applied to the games, without the need for complex implementations or expensive equipment. Due to the popularity, many students have high interests in and personal experience with the game, which may motivate student engagement. The courses were also well received judging by student feedback, with average scores of 4.21 and 4.35 points, of which 1 to 5 point indicate that the students *strongly disagree*, *disagree*, *neutral*, *agree*, or *strongly agree* that the projects were helpful for learning.

This paper is organized as follows. Section 2 introduces 2048-like games and their common properties, followed by a brief review of existing techniques applied to 2048-like games. Section 3 shows how teaching materials were designed with 2048-like games in our courses and summarizes student results and other observations. Section 4 covers some relevant discussions and makes concluding remarks.

2. BACKGROUND

In this section, we will first introduce the family of 2048-like games, then briefly review the relevant computer game algorithms and techniques proposed in recent years.

2.1. Single-player 2048-like games

In this subsection, we will first introduce 2048,[1] 2584,[2] and Threes!,[3] then summarize their similarities and differences.

2048

2048 is a 4 by 4 puzzle that begins with two randomly placed tiles. Tiles on the puzzle are numbered with powers of 2, for example, 2-tiles, 4-tiles, or as an extreme example, 65536-tiles. The objective is to slide the puzzle such that the tiles can merge, forming tiles with larger numbers.

Upon sliding the puzzle, all tiles will move in the chosen direction as far as possible, i.e. each tile will move in the chosen direction until it reaches either the border of the puzzle, or until the adjacent tile in the specified direction has a different value. If the adjacent tile in the chosen direction has the same value, say, both tiles are v-tiles, they will be merged into a single $2v$-tile. The player will receive $2v$ points as the reward in the process. Tiles that are generated through merging in a single action cannot immediately merge with another tile until the next action. A direction is illegal if the puzzle remains unchanged after sliding; illegal actions are unable to be selected.

The environment will immediately generate a new tile after the player makes an action. The new tile can be either a 2-tile or a 4-tile, with probabilities of 0.9 and 0.1 respectively. It will be randomly placed at an empty position on the puzzle. After the new tile has been added, the player can continue to slide the puzzle. This process repeats until there is no legal direction to move. The player wins the game if a 2048-tile is generated, where the final score of the entire game is the accumulation of rewards gained from every sliding action.

Figure 1 is a part of an episode of 2048. This example starts with a puzzle (a) where the player must decide on a direction to slide. For puzzle (a), only up, right, down are legal directions. After the player

[1] Available at https://gabrielecirulli.github.io/2048/.
[2] Available at https://davidagross.github.io/2048/.
[3] Available at http://asherv.com/threes/ and http://threesjs.com/.

slides the puzzle up, two 32-tiles, two 2-tiles, and two 4-tiles are merged, which changes the puzzle to (b). The player receives $64 + 4 + 8 = 76$ points as the reward. The environment then generates a 2-tile as shown as in (c), and the player continues to slide the puzzle left, changing the puzzle to (d). The game continues until there are no legal directions for the player.

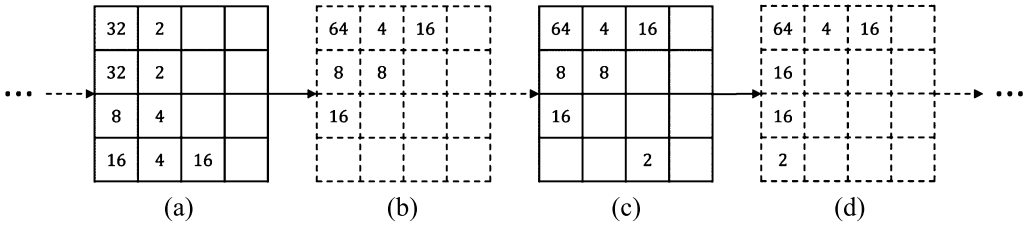


Fig. 1. A part of an episode of 2048.

2584

2584 is a 2048-like game that is different from 2048 only in the tile values and the merging rules. In this game, the tiles are labeled by the numbers in the Fibonacci series. The player wins the game if the 2584-tile is reached in the puzzle. Like 2048, the environment also generates 1st and 2nd tiles (1-tiles and 2-tiles) with probabilities of 0.9 and 0.1. The major difference is that the two tiles whose values are adjacent numbers in the series can be merged into the next larger number in the series. For example, 1-tile + 1-tile = 2-tile; 1-tile + 2-tile = 3-tile; 2-tile + 3-tile = 5-tile; 3-tile + 5-tile = 8-tile.

Threes!

Threes!, developed by Vollmer and Wohlwend, precedes 2048 and 2584 and is often attributed as the first game in the 2048 family of games. A game of Threes! is played on a 4 by 4 puzzle, starting with nine initial tiles that consist of 1-, 2-, or 3-tiles. The sliding distance in Threes! is at most 1, which is different from 2048. The merging rule for tiles is also slightly different in that a 1-tile and a 2-tile can be merged into a 3-tile, and two v-tiles can be merged into a $2v$-tile for all $3 \leqslant v < 6144$. 6144-tiles are the maximum allowable in Threes!, i.e. they cannot be merged any further. Note that tiles can only be merged when the corresponding direction on the puzzle is filled. If there are more than one mergeable tile pair in the same row, only the first pair can be merged.

The environment generates a new tile on the puzzle after every action. The type of the next generated tile is provided as a *hint*, i.e., the player can peek at the next generated tile before taking an action. There are four kinds of hints: 1-*hint*, 2-*hint*, 3-*hint*, and +-*hint*. The first three kinds of hints mean that the next generated tile will be the corresponding tile, i.e., a 1-hint means the next generated tile will be a 1-tile. While a +-hint means the next generated tile will be a bonus tile, i.e. the environment generates a tile with a value greater than 3.

The rules of generating new tiles are much more complicated than those for 2048. The generated tile will be randomly placed at a newly cleared empty space on the opposite side of the last sliding direction. The type of the new tile is controlled by two rules: the *normal rule* and the *bonus rule*. To explain the normal rule, consider that there is a bag of 12 tiles composed of equal amounts of 1-, 2-, and 3-tiles. Every time a new tile is generated, a tile is randomly selected from the bag and removed until the bag is empty, at which point it is refilled. The normal rule only generates 1-, 2-, and 3-tiles. The bonus rule dictates how bonus tiles are generated. When the largest tile on the current puzzle, the $v_{\max}$-tile, is at least a 48-tile, there is a probability of $1/21$ to generate a bonus v_+-tile where $6 \leqslant v_+ \leqslant 1/8 \times v_{\max}$ with equal probability, while normal tiles are generated with a probability

of 20/21 from the bag. For example, if $v_{\max} = 384$, v_+ will be one of 6, 12, 24, or 48 with equal probabilities.

The game ends when the player is no longer able to make any legal actions. The final score is the sum of $3^{\log_2(v/3)+1}$ of all v-tiles with $v \geqslant 3$. The game of Threes! does not define a "win" like 2048 (reaching the 2048-tile); the objective is to simply achieve the maximum possible score. In the context of this paper, however, players win if a 384-tile is generated.

Similarities and differences between 2048, 2584, and Threes!

The objective of all three games is to merge tiles and maximize the score. In 2048, merging only occurs for two tiles of the same value, where the score is the sum of rewards, i.e. the sum of merged tile values. In 2584, merging occurs for tiles with adjacent values in the Fibonacci series, where the score is calculated the same way as 2048. In Threes!, 1-tiles and 2-tiles can be merged into 3-tiles, while two tiles of the same value that is greater and equal to 3 can also be merged. However, the final score of Threes! is more complicated, calculated by summing $3^{\log_2(v/3)+1}$ for all v-tiles where $v \geqslant 3$.

While the reward of actions is straight-forward in 2048 and 2584 (the sum of all merged tiles from the action), the same cannot be said for Threes!, where the reward of actions is not apparent at first. More specifically, the score is calculated for the whole puzzle by tallying all the tiles when the game is over, so rewards for intermediate actions are not obtainable from the environment. To obtain rewards for intermediate actions, we can calculate the difference between the value of the puzzle before and after the action.

The rules for generating new tiles is simple for 2048 and 2584. Both 2048 and 2584 generate only 1$^{\text{st}}$ and 2$^{\text{nd}}$ tiles (2-tile and 4-tile in 2048; 1-tile and 2-tile in 2584). The probabilities of generating the 1$^{\text{st}}$ and 2$^{\text{nd}}$ tiles for these two games are the same, 0.9 for the 1$^{\text{st}}$ tile and 0.1 for the 2$^{\text{nd}}$ tile. For Threes!, generating a new tile is much more complicated, which is dependent on the player's actions and current tiles on the puzzle. When the largest tile of the current puzzle is less than the 48-tile, the environment generates 1-, 2-, and 3-tiles with equal probabilities. Otherwise, the environment may generate a bonus tile with probability 1/21, as described above.

The theorical maximum tile generally depends on the board size, the merging rule, and the type of newly generated tiles. All three games are played on a 4 by 4 puzzle. In 2048, the maximum tile is theoretically the 131072-tile, which is the 17$^{\text{th}}$ tile. However, the chances of seeing 65536-tiles and 131072-tiles are extremely low. In 2584, since adjacent-valued tiles are mergeable, tile indices can easily grow to more than 20, which is significantly larger than 2048 or Threes!. The theorical maximum tile is the 3524578-tile, which is the 32$^{\text{nd}}$ tile. However, as is the case in 2048, large tiles like these are rare. In Threes!, although there is enough space for large value tiles, the maximum tile is limited by the game rules to be the 6144-tile.

2.2. Two-player 2048-like games

To accommodate for a competition at the end of the term, and to also allow students to have hands-on experience with the minimax paradigm and adversarial techniques, we proposed changes to the original games so that it may be treated as a two-player game as follows.

In our course design, we define the term *player* to be the role that slides the puzzle and plays the game; the term *adversary* is defined to be the role of an antagonistic environment who tries to make the game more difficult for the player. In other words, the player tries to maximize the score, while the adversary minimizes the score. Thus, the modified two-player game begins with the adversarial

side. First, the adversary places some tiles on an empty puzzle. Then, the player and the adversary take turns sliding the puzzle or placing a new tile. The game ends when the player is unable to move.

2.3. A generic framework for 2048-like games

All puzzles in 2048-like games can be categorized into two kinds: *states* and *afterstates*. An instance of a 2048-like game begins with a special state called the *initial state*. The player performs an *action*, i.e., sliding the puzzle, to the state, upon which the state will transform into an afterstate. The environment may then make immediately changes to the afterstate, which renders it into another state for the next *time step*. The game continues until reaching a *terminal state*, which is a state for which the player is unable to perform any actions. A short episode of only five puzzles of a generic game is shown in Fig. 2.

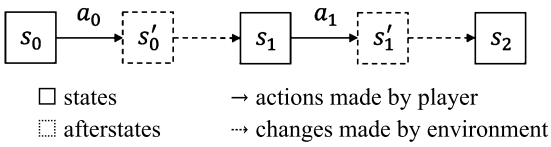


Fig. 2. An episode of a generic 2048-like game.

2.4. Techniques related to 2048-like games

Machine learning and search algorithms for 2048 can be applied to other 2048-like games without major changes. In this subsection, we will briefly introduce some methods which are related to 2048 and 2048-like games.

Tree search

The original 2048 game conforms to the *expectimax paradigm*, i.e., the player tries to maximize the score; and the type and the position of the tile added by the environment are selected randomly. The expectimax search tree is composed of max nodes and expected nodes, which corresponds to states and afterstates. Max nodes select the node corresponding to the maximum value. Expected nodes (also known as chance nodes) calculate the expected value by expanding all possible situations and accumulating the value with their probabilities (Russell & Norvig, 2016).

In the case of a two-player game where the adversary can determine both the type and the position of new tiles, minimax search should be performed instead of expectimax search. The minimax search tree is composed with max nodes and min nodes, corresponding to states and afterstates. The behavior of max nodes is similar to max nodes in the expectimax paradigm. Min nodes minimize the value of player, i.e., with an adversarial 2048 game where the adversary attempts to make the game as difficult as possible for the player, the adversary can determine both the type and the position of the new tiles to minimize player rewards or to force the player to end the game before victory can be achieved. This is referred to in this paper as conforming to the *minimax paradigm*. Conversely, if the type of new tile is decided randomly and the adversary can only determine the position of the new tile, the game conforms to the *expectiminimax paradigm*.

In practice, some optional techniques are often applied with tree search methods. First, with limited time to expand the game tree to leaf nodes to obtain the exact value, heuristic functions or value functions can be very useful (Russell & Norvig, 2016). Second, alpha-beta pruning is usually applied

with the minimax search to save computing time. Third, to avoid redundant search on the same nodes, a transposition table can be built to cache state results.

Reinforcement learning

Reinforcement learning (RL) is a kind of machine learning method which learn how to perform actions with a goal of maximizing the total outcome (Sutton & Barto, 1998). The agent constantly interacts with the environment by performing actions, and the environment responds by presenting the new states and providing corresponding rewards to the actions. More specifically, as shown in Fig. 3, at each time step t, the agent performs an action a_t based on the current environment state s_t. Then, the environment responds the reward r_t as well as the next state s_{t+1}.

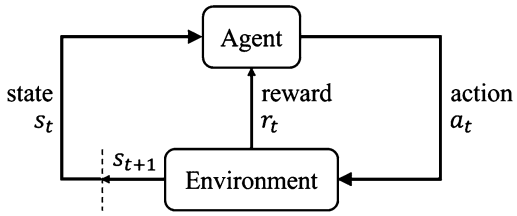


Fig. 3. Reinforcement learning framework.

Temporal difference learning (TD) is a reinforcement learning method (Sutton & Barto, 1998), which was first applied to 2048 by Szubert & Jaśkowski (2014). In Fig. 2, s_0, s_1, s_2 are states, and a_0, a_1 are actions. Thus, Fig. 2 can be redrawn with a time index t as in Fig. 4, which also includes the corresponding rewards, called scores above, and the afterstates, s'_t and s'_{t+1}, representing the states after taking actions and before transiting states by the environment.

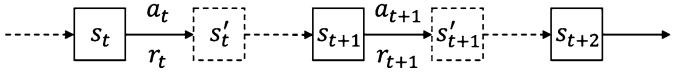


Fig. 4. State transitions with RL framework for generic 2048-like games.

The simplest form, TD(0), updates the value function $\mathrm{V}(s_t)$ with the prediction error $\delta_t = r_t + \gamma \mathrm{V}(s_{t+1}) - \mathrm{V}(s_t)$ from subsequent values through $\mathrm{V}(s_t) \leftarrow \mathrm{V}(s_t) + \alpha\delta_t$, where α is the learning rate, and γ is the discount factor. $\mathrm{V}(s_t)$ can be viewed as the expected return of a given state s_t. Therefore, a policy $\pi(s_t)$ can be derived to be $\pi(s_t) = \mathrm{argmax}(r_t + \mathrm{P}(s'_t, s_{t+1})\mathrm{V}(s_{t+1}))$, where s'_t is the afterstate corresponding to state s_t (as shown in Fig. 4), and $\mathrm{P}(s'_t, s_{t+1})$ is the transition probability function from afterstate s'_t to the next state s_{t+1}.

The general form, TD(λ), updates the value function with all subsequent errors with a trace decay parameter λ. The TD error $\delta_t^\lambda = ((1-\lambda)\sum_{n=1}^{T-t-1}(\lambda^{n-1}(\sum_{k=0}^{n-1}(\gamma^k r_{t+k}) + \gamma^n \mathrm{V}(s_{t+n}))) + \lambda^{T-t-1}\sum_{k=0}^{T-t-1}(\gamma^k r_{t+k})) - \mathrm{V}(s_t)$, where T is the time step of the terminal state (Sutton & Barto, 1998). Higher λ values increase the proportion of prediction error from more distant states and actions. However, in practice, instead of updating every single value with the entire TD error, n-step truncating TD(λ) is often applied, which only updates a value with its subsequent n TD errors. For example, 5-step TD(0.5) and 3-step TD(λ) have been successfully applied to 2048 (Yeh et al., 2017; Jaśkowski, 2017).

The above learning framework is known as the *state learning framework* (TD-state), i.e., the value function stores state values. Another possible implementation is the *afterstate learning framework* (TD-afterstate). In the afterstate learning framework, the simplest TD(0) updates the afterstate value

function $V(s'_t)$ with error $\delta_t = r_{t+1} + \gamma V(s'_{t+1}) - V(s'_t)$, and the policy $\pi(s_t) = \text{argmax}(r_t + V(s'_t))$. When the numbers of training episodes are the same, TD-afterstate is slightly better than TD-state (Szubert & Jaśkowski, 2014). There is another possible training algorithm, Q-learning, which takes a state as input, and outputs the four afterstate values (Sutton & Barto, 1998). However, Q-learning is significantly less efficient than TD learning, and therefore is not considered to be fitting for 2048-like games (Szubert & Jaśkowski, 2014).

Forward update and backward update are both possible when implementing TD learning. Forward update here refers to the scheme where all states are updated in order of an episode from the initial state to the terminal state; backward update simply reverses this order. It has been reported that backward update is slightly better than forward update when the training episodes are the same (Matsuzaki, 2017a). Additionally, backward update is also easier for students to understand and implement (Guei et al., 2018).

Multi-stage temporal difference learning (MS-TD) divides the entire episode into several stages, each with a unique function approximator. It has been applied to 2048, resulting in the first computer program to achieve the 65536-tile (Wu et al., 2014; Yeh et al., 2017). Several other implementations have also been investigated, such as finding more optimal ways to divide stages (Jaśkowski, 2017; Matsuzaki, 2017b).

Temporal coherence learning (TC) is a variant of TD with adaptive learning rates (Beal & Smith, 1999), which has been applied to 2048 by Jaśkowski (2017). The update amount is controlled by the parameter α and the coherence $|\sum \delta| / \sum |\delta|$, where δ is the TD error of each update. The update amount decreases if the TD error δ starts to oscillate between positive and negative values.

Function approximator

The state spaces of 2048-like games are quite large. Take 2048 as an example; the upper bound of the state space is $18^{16} \approx 1.2 \times 10^{20}$ since the theorical maximum tile value is $2^{17} = 131072$. Even when we only consider tiles lesser than 65536-tile, the upper bound is still $16^{16} \approx 1.8 \times 10^{19}$, which is unaffordable for current computers to store the $V(s)$ of the entire state space with a direct mapping table. Therefore, function approximators, which are able to store the value of states more efficiently, are necessary. Function approximators not only save space by approximating functions with specific features, they also improve the training process since multiple states may share common features.

n-tuple networks have become well-known function approximators for 2048-like games since they were first proposed in 2014. An n-tuple network estimates $V(s)$ by extracting some features from the state s and accumulating feature values, where features are usually sub-puzzles in 2048-like games. Sometimes puzzle information is also included into the n-tuple network feature design, such as the maximum tile of the puzzle, or hints in Threes!. The performances of n-tuple networks are highly correlated with feature design. Good configurations are not trivial, and have been investigated as a topic of research (Yeh et al., 2017; Matsuzaki, 2016). Some common settings are shown in Fig. 5. Note that they are basic patterns, where the full configurations contain the eight isomorphisms (rotation and mirror) of these patterns.

Each feature maps to an entry in a lookup table. The value function is therefore $V(s) = \sum_i^m \text{LUT}[\phi_i(s)]$, where LUT is the table of weights, m refers to the total number of features, and ϕ_i is the i^{th} feature. The TD error δ is also split to every feature during the update, $\text{LUT}[\phi_i(s)] \leftarrow \text{LUT}[\phi_i(s)] + (1/m)\alpha\delta$. Take the "Lines" configuration in Fig. 5 as an example, which has 2 base features: ϕ_1 and ϕ_2, and there are 16 features in total: $\phi_{1_1}, \phi_{1_2}, \ldots, \phi_{1_8}, \phi_{2_1}, \phi_{2_2}, \ldots, \phi_{2_8}$. If we

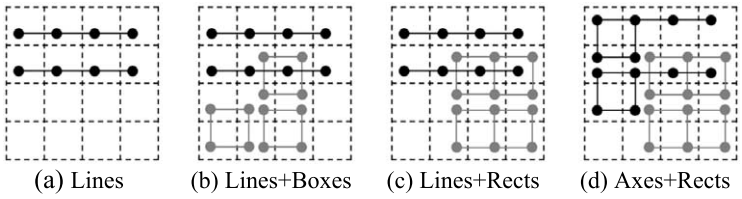


Fig. 5. Common n-tuple network configurations.

only consider up to the 32768-tile, the size of the LUT will be $16^4 \times 2 = 131072$. More specifically, there are two sub lookup tables: LUT_1 and LUT_2, corresponding to each base feature. Features derived from the same base pattern share a lookup table. The value function is therefore $\text{V}(s) = \sum_{i=1}^{8} \text{LUT}_1[\phi_{1_i}(s)] + \sum_{i=1}^{8} \text{LUT}_2[\phi_{2_i}(s)]$. Using Fig. 1 (a) as an example, if we encode the patterns with powers of two, ϕ_{1_i} (Fig. 1 (a)) will be 5100, 4355, 0424, 0000, 0015, 5534, 4240, 0000; and ϕ_{2_i} (Fig. 1 (a)) will be 5100, 2211, 0023, 0004, 0015, 1122, 3200, 4000.

Deep neural networks (DNN) and convolutional neural networks (CNN) can also be used as function approximators for 2048 (Guei et al., 2016; Matsuzaki & Teramura, 2018; Wei, 2019; Kondo & Matsuzaki, 2019; Matsuzaki, 2019). Compared with n-tuple networks, DNNs usually consist of fewer weights overall, but more weights are involved for each output. At the time of writing, strong 2048-like game programs currently use n-tuple networks as the function approximator (Yeh et al., 2017; Jaśkowski, 2017). Using DNNs efficiently for 2048-like games is still an open research topic.

Other improvements

The bitboard is an optional improvement for 2048-like games which can replace array boards when storing states. Bitboards are efficient in both memory usage and calculation time. Take 2048 as an example. Considering tiles lesser than 65536-tile, we can use 4 bits for each tile, and 16 bits for a row of the puzzle. Since the sliding result of states in 2048-like games are deterministic, we can then build a lookup table that pre-calculates the sliding results of every 16-bit row when the program first executes. During runtime, the program can then simply find the sliding results row by row efficiently. Also, the generation of isomorphic states through reflection or transposition can be implemented with bitwise operations.

A 64-bit bitboard (4 bits per tile) is usually enough for 2048 and Threes!. However, at least 5 bits per tile is necessary for 2584. For 2048, although the theorical maximum tile is the 131072-tile, which corresponds to the 17^{th} tile, the 65536-tiles and 131072-tiles are nearly impossible to reach and can be safely ignored in most cases. For Threes!, the maximum tile is limited to the 6144-tile, which is the 14^{th} tile. Considering there are extra pieces of information, such as the hint tile or the player's last action, for every state, a 64-bit bitboard can also be used. For 2584, since the indices can easily grow to more than 20, at least 5 bits per tile is necessary. While the theorical maximum tile is the 32^{nd} tile, this is also nearly impossible to reach and can also be safely ignored in most cases.

3. COURSE DESIGN AND STUDENT RESULTS

Theory of Computer Games is a graduate level course taught by Professor I-Chen Wu at National Chiao Tung University (NCTU), Taiwan. The prerequisites are Algorithms and Data Structures. Prior experience in other courses such as Artificial Intelligence or Machine Learning in advance is helpful, but not required. Though the title of the course involves "theory", we place special emphasis on

implementation in this course. Therefore, students are expected to be moderately proficient at programming.

Students are required to develop a game-playing program as the term project during the semester, which spans about five months. Since 2014, due to the popularity and the simplicity of 2048 and also because of the advent of machine learning, 2048-like games had become a staple application for the term project. In this paper, we will take the courses in 2017 and 2018 as examples, to illustrate how to use 2584 and Threes!, two 2048-like games mentioned above, as pedagogical tools.

The overall program is broken down into six projects. In the series of projects, students are required to develop their program step by step. We will first give a short summary of the projects, then introduce details and results in subsequent subsections.

- Project 1: Learning to use the framework
- Project 2: TD and n-tuple networks
- Project 3: Solving a reduced game by expectimax
- Project 4: Improving the performance of the player
- Project 5: Designing the adversarial environment
- Project 6: Final tournament

3.1. Learning to use the framework

Students have to implement the framework and the environment used for the entire semester's projects in two weeks. We provide the framework[4] for 2048 as the sample code in both C++ and Python. The framework is a demo program for 2048, which contains certain standardized IO functions, such as those that record episodes in a specific format. Students can start their projects using the sample code, modifying it from the original 2048 implementation to the target game.

The target environment needs to be simplified accordingly. We scale the difficulty of the target game so that its environment allows a win rate of about 85%–95% after simple TD training, which we discuss in the next subsection. Table 1 lists modifications for both 2584 and Threes!. In order to make sure that students are familiar with the rules and the framework, they also need to design a simple strategy to play the game. For example, a trivial strategy involves selecting the action that yields the maximum reward.

Table 1
Rule modifications in Project 1

Game	Rule modifications
2584	Nothing is changed.
Threes!	Bag size is set to 3 (original size is 12); no bonus tiles. New tiles can be randomly placed at any empty position on the opposite side of the last sliding direction.

We do not expect students to use machine learning or sophisticated search techniques in this first project. Strict time requirements (1 CPU core; 1 GB memory; 100,000 moves/s for C++ and 1,000 moves/s for Python) are enforced. Students will receive 85% of full credit for this project if they

[4]Project frameworks and specifications are available at following repositories.
(C++) https://github.com/moporgic/2048-Framework/branches/.
(Python) https://github.com/moporgic/2048-Framework-Python/branches/.

implement the correct environment. The remaining 15% is dependent on their program performance. The detailed grading criteria are provided in Table 2, and the student results are summarized in Table 3. Most of the students simply chose to play according to the maximum reward, or in some cases, the agents were designed to always select left and down if possible. Some of them performed a simple two-ply search. Only a few students designed complex heuristic functions using patterns or custom rules.

Table 2
Grading criteria in Project 1

Game	Grading criteria		
	Environment*	Average performance	Maximum tile
2584	85%	15%	Bonus
		Calculated by $\min(\lceil \text{Score}_{\text{AVG}} \div 1000 \rceil, 15)$	Calculated by $\max(\text{MaxTile} - 13, 0)$
Threes!	85%	10%	5%
		Calculated by $\min(\log_2(\text{Score}_{\text{AVG}} \div 3) + 1, 10)$	Calculated by $\max(\text{MaxTile} - 9, 0)$

$\text{Score}_{\text{AVG}}$ is the average score of 1000 testing episodes; MaxTile is the index of the maximum tile in 1000 testing episodes.
*The credit is not counted if the environment implementation is incorrect.

Table 3
Student results in Project 1

Game	Avg. performance*	Avg. max tile
2584 39 submissions 4 were late	10.64 ± 4.11 15 reached full credit (15 points)	$16.64^{\text{th}} \pm 1.29$ 11 reached the 18^{th} tile
Threes! 44 submissions 3 were late	8.41 ± 0.73 3 reached full credit (10 points)	$9.36^{\text{th}} \pm 0.86$ 11 reached the 10^{th} tile

*Calculated by the grading criteria listed in Table 2.

3.2. TD and N-tuple networks

Students need to train a stronger player using temporal difference learning (TD) with n-tuple networks in Project 2. The main purpose is to ensure that students understand the mechanism of TD and n-tuple networks. They are not required to apply advanced TD techniques such as TD(λ) or MS-TD in this project. The whole project should be done within the one-month deadline. The environment of this project follows the previous one, as shown as in Table 1 (above).

There are only minor differences between forward/backward implementations in most training cases. Students are free to choose whichever they prefer, but we recommend backward training since it is easier to understand and debug. Also, we recommend students to apply the afterstate learning framework first since it is easier to implement in terms of taking actions and training.

A clear reward definition is necessary. We suggest students follow the definition of the original games, but they are free to define their own reward functions, such as giving a constant reward for each action taken. Using this simple reward function, the agents are rewarded for surviving as long as possible. Note that Threes! does not actually define rewards in terms of actions, but instead give rewards all at once according to the game over state. Practically, a simple reward can be calculated by taking the difference between the value function outputs of the state before and after an action.

Students can either survey papers for good n-tuple network configurations or design the patterns on their own. However, we recommended students to start their project with the simplest configuration "Lines" as shown in Fig. 5 (above) with only rotations, and apply feature extraction without sharing lookup tables. With this implementation, there would be eight 4-tuple patterns (four rows and four columns) corresponding to eight standalone sub lookup tables. Since this is relatively simple, it is a good starting point for students who are not familiar with n-tuple networks.

In summary, the baseline setting, which is recommended for students, use simple TD(0) with the above n-tuple network configuration "Lines" with only rotations, base learning rate $\alpha = 0.1$, and no learning rate decay. When implemented correctly, the win rate grows rapidly in the early stages of training, and students should see their first "win" in the first 10k training episodes. With this setting, students can reach a win rate of about 85% after 100k training episodes, which should only take at most several hours.

Similar to Project 1, restrictions on memory usage and program speed are given (1 CPU core; 2 GB memory; 50,000 moves/s for C++ and 500 moves/s for Python). We use the win rate of 1000 games as the grade in Project 2, as shown as in Table 4. For Threes!, since the 6144-tile is difficult to obtain, we use the win rate of 384-tile instead.

Table 4
Grading criteria in Project 2

Game	Grading criteria	
	Average performance*	Maximum tile
2584	100%	Bonus
	Calculated by $\lceil \text{WinRate}_{384} \rceil$	Calculated by $\max(\text{MaxTile} - 17, 0)$
Threes!	100%	Bonus
	Calculated by $\lceil \text{WinRate}_{2584} \rceil$	Calculated by $\max(\text{MaxTile} - 9, 0)$

WinRate_{2584} and WinRate_{384} are the win rate in 1000 testing episodes.
MaxTile is the index of the maximum tile in 1000 testing episodes.
*The credit is not counted if the environment implementation is incorrect.

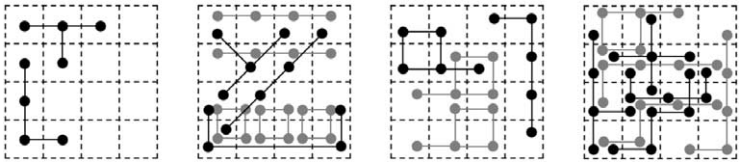
From the result listed in Table 5, most of the students achieved a win rate of over 90% with TD for both 2584 and Threes!. However, the n-tuple network configurations were different. In 2584, one third of the students submitted their work with the simplest "Lines" configuration since reaching the 2584-tile is relatively easy even for simple 4-tuple networks. However, in Threes!, the simplest configuration can only achieve a win rate of about $85\% \sim 90\%$. For better performance, many students applied the efficient 6-tuple "Axes + Rects" configuration.

Table 5
Student results in Project 2

Game	Method		Avg. win rate	Avg. max tile	Avg. n for n-tuple
2584	TD(0): 32	Forward: 3	$96.1\% \pm 4.2\%$	$21.47^{\text{th}} \pm 1.24$	4.6 ± 0.8
34 submissions	TC(0): 2	Backward: 31	10 reached 100%	15 reached the 22^{nd}	# of {4,5,6}-tuple: {20,7,7}
2 were late			3 did not reach 90%		
Threes!	TD(0): 41	Forward: 7	$93.5\% \pm 11.2\%$	$12.91^{\text{th}} \pm 1.07$	5.6 ± 0.8
43 submissions	TC(0): 1	Backward: 36	5 reached 100%	14 reached the 14^{th}	# of {4,5,6}-tuple: {9,1,33}
8 were late	MS-TD(0): 1		9 did not reach 90%		

Another reason that 6-tuple networks were not widely used for 2584 is their high memory usage. For 2584, tile indices can easily grow to more than 20, which is significantly larger than Threes!,

since it is possible to merge the elementary tiles in both ways. Therefore, students need to design compression techniques for their lookup table if they want to use larger networks. As a workaround, some students trained their agent with custom 4-tuple and 5-tuple configurations, as shown in Fig. 6. A student hand-tuned an efficient 5-tuple, illustrated in Fig. 6 (c), with comparable performance to the 6-tuple "Axes + Rects", winning 99% of its games in 2584.



(a) 2 × 4-tuple (b) 7 × 4-tuple (c) 4 × 5-tuple (d) 8 × 5-tuple

Fig. 6. Some custom n-tuple configurations designed by students.

3.3. Solving a reduced game by expectimax

The aim of this project is to familiarize the students with expectimax search, which is necessary in the following projects. Students should use the expectimax algorithm to solve a 2×3 reduced game. Even with the reduced board size, brute-force expansion of the game tree is not feasible, so students are also required to implement a simple transposition table. The time allocated for this project can range between 2 weeks to 1 month, depending on the complexity of implementation.

The environment basically follows the previous projects'. However, since the board size is limited, we need to make some small changes for Threes!: the initial state only contains 1 initial tile here, while the previous environment contains 9 tiles. In Project 3, we define three kinds of puzzles: *states*, *afterstates*, and *unreachablestates*. The solver program is expected to be able to distinguish them. States and afterstates are described in detail above; unreachablestates are puzzles that are unreachable from the legal initial states under legal gameplay. For example, a state filled with six 1-tiles is obviously an unreachable state for 2584 since it is impossible to accumulate that many 1-tiles as they will definitely merge with each other. To distinguish unreachablestates, the simplest way is to start expanding the game tree from the root. After the entire tree is expanded, all unvisited puzzles are unreachablestates.

The expected value can be calculated by performing max operations at max nodes, and performing weighted sum operations at chance nodes. Besides the expected value of the state, it is also possible to calculate the best value and the worst value. The best value is the total return when the environment coincidentally generates the best tile, which leads the player to the highest score under oracle play. The worst value is exactly the opposite. Some examples for Threes! can be found in Table 6.

Student programs are given one minute to expand the entire game tree for the reduced game and to store the results in a transposition table. Their programs are not allowed to use any pre-calculated external database. In normal cases, with proper implementation, the process of calculating the value of the whole 2×3 game tree with the transposition table costs less than one second. Some redundant recalculations may increase the total time to several seconds. However, without the transposition table, expectimax search is unable to solve the problem within the time limit. Note that the expected value of states and afterstates are not necessarily the same; therefore, the transposition table should store their values independently.

The grade for Project 3 is calculated by the percentage of problems solved. Some problems and their corresponding answers are provided as sample inputs and outputs. The sample outputs we provided

Table 6
Examples of state values for Threes!

State						Value		
Type	Puzzle			Hint	Last Action*	Expected	Best	Worst
States	0	1	0	3	N/A	455.97	1092	258
	0	0	2					
Afterstates	3	2	1	2	Up	198.94	837	75
	48	6	0					
States	1	12	3	2	N/A	326.66	1053	81
	1	2	6					
Afterstates	0	0	3	1	Down	310.07	1050	0
	3	6	12					
States	48	3	2	1	N/A	0	0	0
	96	12	6					
Afterstates	96	12	6	3	Right	3	3	3
	0	6	2					
States	0	1	0	1	N/A		(unreachable)	
	0	0	3					
States	96	192	12	3	N/A		(unreachable)	
	0	1	3					

*It is necessary for afterstate values since the position of new tile relates to player's action.

covered all possible boundary cases, so students had the opportunity to correct any problems accordingly before submission. Table 7 lists the student results for this project, where the result for Threes! is worse than those for 2584. There are two potential reasons for this. First, we observed that the students ran into difficulties implementing expectimax search with hints, which was unique to Threes!. Since the framework we provided was designed for 2048, students had to program hint processing on their own. Second, we introduced some changes to Threes! in 2018 to increase the project difficulty, so that we can better discriminate student performance. Although the definitions for the best and worst values are not difficult to understand, more specifications tend to lead to more avenues of error for students.

Table 7
Student results in Project 3

Game	Solver requirement	Avg. % of problems solved
2584	Expected value	$99.9\% \pm 0.5\%$
34 submissions		31 reached 100%
3 were late		3 did not reach 100%
Threes!	Expected value	$92.5\% \pm 15.5\%$
35 submissions	Best value	24 reached 100%
11 were late	Worst value	6 did not reach 90%

3.4. Improving the performance of the player

Project 4 tries to encourage students to improve their player with optional techniques. It is similar to Project 2, but with more stringent environment conditions, which are listed in Table 8. Take 2584 for example. The environment for Project 4 drops 1-tiles and 3-tiles, instead of 1-tiles and 2-tiles. Since 3-tiles are not mergeable with neither 1-tiles nor other 3-tiles, it is more difficult to get a high score. A case in point is that the simple "Lines" 4-tuple design shown in Fig. 5 (above) can only reach a win rate of about 40% under these new conditions.

Table 8
Rule modifications in Project 4

Game	Rule modifications
2584	1-*tiles and* 3-*tiles* are generated with probabilities of 0.75 *and* 0.25, respectively.
Threes!	*Bag size and bonus tiles follow original.* New tiles can be randomly placed at any empty position on the opposite side of the last sliding direction.

Note: Differences between Project 4 and Project 2 are in italic.

To maintain good performance under more difficult environments, students must improve their programs using additional techniques that were covered in class. Also, they were encouraged to read recent papers for 2048 to find ideas to improve their program. Students were given one month to complete Project 4.

The four additional techniques we covered in class are listed as follows. The first suggestion is adding expectimax search, which students used in Project 3. With a one-ply search added, without having to retrain the network, the win rate of the "Lines" configuration can reach 70% win rate. Since expectimax search is costly in computation time, bitboard and lookup tables are also optional improvements to consider. The second technique is to use more complex network structures. For example, replacing 4-tuple networks with 6-tuple networks. To avoid high memory usage and to speed up the training process, students should also implement isomorphism of features, or include some form of compression. The third is to manually lower the learning rate, which is likely to push the performance higher if this was not previously used. The last suggestion is to try an assortment of advanced TD methods, such as using TC to automatically decay the learning rates, or using MS-TD.

We use the same scoring criteria as Project 2, i.e., the final grades are calculated by win rate, where the winning tile is either the 384-tile or the 2584-tile for Threes! or 2584, respectively. However, we relax the computing resource restrictions (1 CPU core; 4GB memory; 10,000 moves/s for C++ and 100 moves/s for Python) since the more sophisticated agent designs require more resources. The student results are summarized in Table 9.

Students were also encouraged to use advanced training techniques such as TC, TD(λ), or MS-TD. However, only a few students adopted these techniques in Project 4. Most students tried to improve their player by using a larger network and incorporating expectimax search. Contrary to previous results in Project 2, students got better performances in Threes! in Project 4. One possible reason is that the rule changes for 2584 is much more difficult, as 3-tiles are unmergeable with neither 1-tiles nor other 3-tiles, and are generated with a high probability of 0.25. Another possible reason for this result is the usage of larger networks. Many students encoded the hint into their network, which greatly improved performance. Since the maximum tile for Threes! is limited to the 6144-tile (the 14^{th} in terms of index), students can also enlarge the network easily.

Table 9
Student results in Project 4

Game	Method		Avg. win rate	Avg. max tile	Avg. n for n-tuple	Avg. depth*
2584	TD(0): 29	Forward: 3	82.0% ± 11.3%	19.52th ± 0.71	5.5 ± 0.7	2.6 ± 0.8
31 submissions	TC(0): 1	Backward: 28	12 reached 90%	16 reached the 20th	# of {4,5,6}-tuple: {4,8,19}	25 applied
0 were late	TC(0.5): 1		12 did not reach 80%			
Threes!	TD(0): 35	Forward: 7	88.8% ± 15.1%	12.25th ± 0.66	6.3 ± 0.7	1.5 ± 0.9
40 submissions	TC(0): 4	Backward: 33	26 reached 90%	13 reached the 13th	# of {4,5,6,7†}-tuple: {2,0,21,17}	10 applied
13 were late	MS-TD(0): 1		4 did not reach 80%			
	MS-TD(0.5): 1					

*Having no additional lookahead corresponds to a depth of 1; an additional 1-ply search corresponds to 3.
†Extra encoding of the network such as hints is counted as one extra tuple.

Due to the computation time limit, adding an extra two-ply expectimax search was nearly impossible with the original framework. Also, since Project 3 for Threes! was relatively difficult, fewer students added expectimax search. The average win rate in Threes! might have been higher if expectimax was applied more often.

3.5. Designing the adversarial environment

The objective of Project 5 is to build an adversary, which is a clear departure from previous projects where students work solely from the perspective of the player. The new rules for the adversary are listed in Table 10. The adversary in two-player Threes! can control both the type and position of a new tile, which conforms to the minimax paradigm. In contrast, the adversary in 2584 can only control the position, while the type is randomly generated, which conforms to an expectiminimax paradigm.

Table 10
Rule modifications for adversary in Project 5

Game	Modifications to tile generation rules		Paradigm
	Type of new tile	Position of new tile	
2584	Generated randomly	Determined by the adversary	Expectiminimax
Threes!	Determined by the adversary	Determined by the adversary	Minimax

One month is given for Project 5. The best practice is to retrain the networks for the player and adversary under the new paradigm. However, it is still possible to reuse the weight table trained for Project 4 since the rules are basically the same. Previously, the player estimates afterstate values by looking up the corresponding weight tables and selecting the action based on the maximum value. For the adversary, one can use the same weight table in the opposite way, i.e. by choosing the minimum value. However, the weight table for Project 4 is designed for a player under expectimax paradigm. the performance may drop slightly due to paradigm mismatch. Also, since the previous implementation usually takes an afterstate as input, it is not ideal for an adversary to access such a weight table since an additional 1-layer search is needed.

Students are encouraged to try simple minimax search before adding alpha-beta pruning. We do not recommend Monte Carlo tree search (MCTS) to students since its performance tends to be weaker than minimax search together with TD value function from our experience. Note that with the expectiminimax paradigm, the adversary is not allowed to determine the type of dropped tiles. Therefore, programs need to handle expected nodes when performing search. However, students may incorrectly bypass expected nodes by pre-generating the sequence of new tiles with the correct probabilities. The

lecturer should place special emphasis on how to design the correct expectiminimax tree to avoid this situation.

Players and adversaries communicate over a network connection. Since communication is very costly, the time restriction is relaxed to 10 moves/s; the hardware resource restriction remains the same as Project 4. We grade the adversary by its win rate, defined as the rate at which the player loses. The grading details are listed in Table 11. Five standard players with different configurations are designed and trained for grading, as shown as in Table 12.

Table 11
Grading criteria in Project 5

Game	Grading criteria	
	Average performance*	Maximum tile
2584	100%	Bonus
	Calculated by $\lceil 100\% - \text{WinRate}_{384} \rceil$	Calculated by $\max(17 - \text{MaxTile}, 0)$
Threes!	100%	Bonus
	Calculated by $\lceil 100\% - \text{WinRate}_{2584} \rceil$	Calculated by $\max(14 - \text{MaxTile}, 0)$

WinRate_{2584} and WinRate_{384} are the win rate of player in 1000 testing episodes.
MaxTile is the index of the maximum tile in 1000 testing episodes.
*The credit is not counted if the environment implementation is incorrect.

Table 12
Standard players for grading in Project 5

No.	n-tuple configuration	Training setting	Extra search
1	Fig. 5 (a) Lines*	1M episodes, with learning rate 0.1 and no decay	N/A
2	Fig. 5 (a) Lines*	1M episodes, with learning rate 0.1 and no decay	Expectimax† 1-ply
3	Fig. 5 (d) Axes + Rects	1M episodes, with learning rate 0.1 and no decay	N/A
4	Fig. 5 (d) Axes + Rects	1M episodes, with learning rate 0.1 and no decay	Expectimax† 1-ply
5	Fig. 5 (d) Axes + Rects	1M episodes, with learning rate 0.1 and no decay	Minimax/expectiminimax 1-ply

Note that only two networks are trained. No.1 and No.2 use the same trained network; No.3, No.4, and No.5 use the other.
*Only with 4 rotation isomorphisms and without shared LUTs. i.e. only 4 rows and 4 columns, each corresponding to a standalone LUT.
†The mismatched paradigm causes the player to play weaker, thereby reducing the difficulty of getting a high grade.

The student results are summarized in Table 13. The variance of adversary performance in Threes! is quite large; half of the students who performed better had an average win rate of $84.3\% \pm 15.9\%$. This was unexpected since the minimax setting should have been easier to handle than the expectiminimax setting for 2584. It is possible that hint processing is much more complex, mistakes were made during network reuse, or minimax search was implemented incorrectly.

Students were told that they could retrain the weight table for an adversary that takes the states as the input, but only a few followed the suggestion. In 2017, most of the students completed search, but only half of them followed up with alpha-beta pruning. Since the time restriction did not allow deep search even with alpha-beta pruning, the advantage of using alpha-beta search was not significant. However, in 2018, not so many students applied search. Also, it is likely that many students implemented the incorrect minimax search for Threes! because the adversary performance is quite low.

Table 13
Student results in Project 5

Game		Method		Avg. win rate*	Avg. n for n-tuple	Avg. depth†
2584	TD(0): 29	Forward: 3	Reuse: n/a‡	87.5% ± 14.7%	5.5 ± 0.7	2.9 ± 0.5
31 submissions	TC(0): 1	Backward: 28	Retrain: n/a‡	18 reached 90%	# of {4,5,6}-tuple: {4,7,20}	29 applied
4 were late	TC(0.5): 1			4 did not reach 80%		
Threes!	TD(0): 31	Forward: 7	Reuse: 31	52.7% ± 35.6%	6.4 ± 0.7	2.4 ± 0.9
37 submissions	TC(0): 4	Backward: 30	Retrain: 6	9 reached 90%	# of {4,5,6,7**}-tuple: {2,0,17,18}	25 applied
1 were late	MS-TD(0): 1			26 did not reach 80%		
	MS-TD(0.5): 1					

*The win rate of adversaries (calculated by 100% − win rate of player).
†The additional 1-layer search of reusing the network is not included.
‡We did not record network reusing statistics for 2584 in 2017.
**Extra encoding of the network such as hints is counted as 1.

3.6. Final tournament

In the final project, all students are required to participate in the tournament and to compete with other classmates. The environment of our final tournament is the same as that of Project 4 and 5. Students do not need to retrain or redesign their player and adversary, but some fine-tuning may help. In 2017 and 2018, we used round-robin scheduling since the number of students is manageable. However, Swiss-system tournament scheduling could be a good choice if the number of students increases for future semesters. Since the final tournament is the most sophisticated, we do not impose hardware limitations, and the time restriction is still 10 moves per second over the internet.

Since only the player receives a score, two games are necessary to determine the result between two students: each student acts as the player and the adversary exactly once, where the student whose player gets the higher score wins. In the past years, we applied a simple strategy to produce the final ranking. The winner receives one point, and the ranking is then determined by the total number of points. It is also possible to apply other rating systems such as the Elo rating, but more games are required if the time limit is not an issue.

The whole tournament took about five hours to run. 30 students participated in the 2584 tournament in 2017, and 40 students in the Threes! tournament in 2018. The top ranked student got 28 points of 29 matches in 2017; the winner for 2018 got 109 points in 117 matches for Threes!. It is worth mentioning that the top ranked programs adopted different strategies. The 1st place 2584 program applied a simple 4-tuple network as in Fig. 6 (a), while the 2nd place program used the custom 5-tuple network as in Fig. 6 (c) with a relatively long training time. However, both the 1st and 2nd place Threes! programs applied complex 7-tuple networks (a 6-tuple in Fig. 5 (d) with extra encodings).

In the 2584 tournament, only a few students did not use search techniques. MCTS was not adapted since it tends to be weaker than minimax search on 2048-like games. Some students did not handle expected nodes under the expectiminimax paradigm correctly. Their programs simply generated a new tile with its corresponding probabilities when performing search, thereby bypassing expected nodes altogether. This tends to result in a minor loss of strength. However, in the Threes! tournament, due to the complexity of search implementation, many students ended up with incorrect search algorithms that had a detrimental effect on performance. As a result, only a portion of students applied search methods.

Another difference between the above two tournaments was the usage of advanced TD methods in 2018. Only a few students applied TC in 2017. However, many advanced TD methods such as n-step

TD, TC, TC(λ), or MS-TD appeared in 2018. This increased the strength of the highest-performing programs and intensified the competition between students.

4. DISCUSSION

In this section, we will discuss our current project design from several perspectives, including how we grade assignments, common mistakes, possible improvements, and also some alternatives of teaching reinforcement learning or computer game algorithms. We will then make a brief summary.

Overall course details and grading

There were 40 and 53 students at the beginning of the course in 2017 and 2018. The course was designed for graduate level students, while undergraduate students were also welcome; of the total number of students, 14 and 25 undergraduate students took the course in 2017 and 2018, respectively. However, there were 6 dropouts and 2 fails in 2017, and 8 dropouts and 2 fails in 2018. The main reason for dropping the course was that they underestimated the heavy work load involved, especially for the term project.

An online forum was hosted to answer student questions. Students were expected to submit their source code, network weight tables, and to answer questions about their implementation. Since both the environment and the agent were handled by the program, it is important to ensure that students do not "cheat", i.e., simplify the environment such so that a higher score can be achieved. One possible solution is to prepare the environment for students and prohibit them from modifying the environment code. This requires a modularized framework, where the source files related to the environment would be replaced by official code during grading. This modularized scheme was used until 2016. Since 2017, students were expected to implement the environment and modify the game rules as the semester progressed, so we had to change to a different grading method.

The grading method used involved using an online judge system to automatically judge and grade student programs. We designed a simple 2048 game recording format, as shown as in Table 14. The student programs are required to output their game records in this format. From the output, we can then judge the correctness of the environment. This was also very helpful for students, since they could debug their program or evaluate its performance with the online judge. The tournament server was made available prior to the tournament. Students could play against one another freely over the internet once they connected their programs to the server. We believe this motivated students to try new techniques during play with classmates.

Table 14
The record of a 2048 game (containing only the first 16 moves)

```
11D1(1)#L(2)61(3)#D[4](3)11(2)#D(3)B1(3)#L[4](3)61#R[8](2)D1(2)#D[4]A1(3)#D31
```

Moves are presented as 2-character.
For an environment move, the first character is the position, and the second character is the value of the new tile. E.g. `11` (place 1-tile at position 1).
For a player move, the first character is '#', and the second character is the sliding direction. E.g. `#L` (slide to left).
The time usage in milliseconds of a move is followed by the move within a pair of parentheses. E.g. `D1(1)` (time usage is 1 ms).
The reward of a move is followed by the move within a pair of square brackets. E.g. `#D[4]` (reward is 4).

From student feedback and our observations, there are several common issues students tend to run into. First, some students were not familiar with the Linux shell and programming. They were not experienced in debugging their programs when encountering fatal errors, e.g., segmentation faults. A crash course on basic Linux commands and debugging with GDB could be offered by the TA to address this. Second, students tend to share the same inquiries. Though there was an online forum for questions, there were many frequently asked questions by email or in person. Several workshops or an online FAQ may be necessary. Third, while we generally provided enough time for students, procrastination was still an issue. Since the projects were designed sequentially, delays in previous projects tend to propagate, which can cause students to drop the course altogether.

Improvements over the past few years

2048-like games have been used as a learning tool for several years. Usually, minor modifications for the term projects take place between semesters. Comparing the term project designs of these past years, there are several differences and improvements. First, there were only 5 projects in 2016. The 2584 environment was provided, where the students did not have to modify the environment and retrain the agent, i.e., Project 4 did not exist in 2016. Second, we did not recommend the use of advanced TD methods such as TC or TD(λ) in 2016. Therefore, the agents trained in 2016 tended to be weaker. Third, we only asked students to calculate the expected value before 2018. After we added the best and the worst values into the project requirements, we could discriminate student performance better. Fourth, we started to provide a Python framework in 2018 since an increasing number of people use Python in this field. Fifth, the 2016 environment followed a fixed order of new tiles: 1-tile, 1-tile, 1-tile, 3-tile. In other words, the game followed a minimax paradigm. Our statistics show that episodes in the 2017 tournament were typically shorter than episodes in 2016. Even more, for the Threes! tournament in 2018, episodes were even shorter. With shorter episodes, we can allow more than one match between each student matchup.

Possible future improvements

However, there are still several ways to improve the course design. First, only the basic TD learning algorithm is covered by these projects. The TD-afterstate algorithm is the only required algorithm in our current projects since it is the most effective method for 2048-like games. A possible further extension may require students to implement not only TD-afterstate, but also TD-state and Q-learning. By comparing these training implementations, students will learn more about the mechanisms of these well-known reinforcement learning algorithms. In addition, only model-free training is covered now, while an extension to model-based training is also possible. We may be able to achieve this by designing a complex black box environment that changes, say, the distribution of generated tiles and the rewards based on time step or current state, then providing students with APIs to interact with the environment. The final grade can be determined by Grade $\propto$ WinRate/EnvSampleCount, where the students are rewarded for better environment sampling efficiency. Another possible extension is to use the 2×3 simplified puzzle to experiment with policy iteration and value iteration.

Second, we would like to address the lack of MCTS in our projects. MCTS is a critical algorithm used in contemporary programs and should be included into our project design in the future. It is not covered in the projects currently since it is not as efficient as expectimax or minimax search with a learned value function for 2048-like games. In the proposed projects, all the mentioned methods were designed as part of a complete program, which progressively improves the agent. Therefore, simply

inserting MCTS into one of the projects would seem out of place since it is not competitive enough for the final tournament. Perhaps MCTS can take advantage of the TD value functions by using it as a heuristic for rollout policies, which we will have to test before incorporating it into the lectures. Third, while many of the modern agents use DNN, especially DCNN as the function approximator, the current state-of-the-art 2048-like agents still use n-tuple networks. Following the reasoning behind the omission of MCTS, DNN is also not included in the projects. However, we should also consider modifying our projects to include DNNs to follow the current trend.

Third, the final tournament could be improved. Network connections between programs and the tournament server leads to time constraints due to network latency. As a result, we were unable to allow multiple matches between each matchup. To increase the number of games and in turn improve evaluation accuracy, it is possible to host the tournament locally, which would involve asking students to submit their programs beforehand. That said, the tournament was designed in its current form because we found that students enjoy the interactivity of seeing one's opponents face to face, while also providing a good opportunity for students to exchange ideas.

Using other games as materials

Throughout the more-than-10 years this course was offered, we have considered using other games such as Go, Amazons, Hex, LOA, Chinese dark chess, NoGo, Gomoku, and Connect6. However, after considering and even using some of them, we arrived at the conclusion that 2048-like games are more suitable for teaching temporal difference learning, and even arguably for reinforcement learning in general.

First, for the two-player games listed above, the opponent is treated as the RL environment (as shown as Fig. 3 above), which introduces additional considerations such as opponent modeling. While it is still teachable, it adds an arguably unnecessary layer of complexity for the students. Since the objectives of the course includes a significant portion of RL concepts, we want to prioritize teaching a simple, cohesive RL framework to the students before they think about concepts such as self-play in the two-player setting, as is the case for Go, Amazons, Hex, LOA, etc. In 2048-like games the environment behaves separately from the agent's actions, and can truly represent situations where the environment and agents are asymmetric, similar to many real-world problems. This opens an opportunity of teaching students about model-free learning much more intuitively, and prepares students to apply their newly acquired knowledge to other fields.

Second, the reward signals are much simpler and more abundant for 2048-like games. For the above two-player games, the "natural" reward that does not require additional human-defined knowledge is simply the outcome of the game (say, 1 for a win and 0 for a loss). The typical, naïve way of approaching this would be to teach Monte-Carlo learning (Sutton & Barto, 1998). While we agree that Monte-Carlo learning is important, we can teach the same concept with 2048, whereas if we use any of the examples above (Go, LOA, Amazons, Hex, etc.), we would have to also teach heuristic evaluations at the same time. In contrast, 2048-like games have rewards built into the game rules, which allows the students to get an intuitive feeling for the RL training progress. Meanwhile, there are also heuristics that can be devised for 2048-like games, so it does not exclude the possibility of teaching hand-tuned evaluation functions.

Third, while the rules for Go are simple, it is a difficult game to master. Even in Asia where Go is a well-known and popular game, few students can play and interpret game records at an adequate level. For the students who have not played Go before, it is unfair to expect them to understand their program's progress. While being a strong player personally is by no means necessary to create strong

programs (that is, after all, the purpose of machine learning), it certainly helps. At the very least familiarity with the game increases students' enjoyment, which we think is an important factor if part of our goal is to engage students and attract them to our research community.

In the end, it is all a matter of choice, and while we believe the same goals can be reached in many different ways, 2048-like games have the advantage of similarity to real-world scenarios, simplicity in terms of teachable components, reward-rich environment, and familiarity to students.

Comparison with alternatives

There are already various courses on reinforcement learning or computer games algorithms that are available. However, most of them focus on either just the former (Lazy Programmer Inc., 2019; Simonini, 2019; University of Alberta, 2019) or the latter (Archibald, 2019; Packt Publishing, 2019; Talaga, 2019). In the following paragraphs, we summarize the characteristics and advantages of these popular alternatives, and also provide a brief distinction between these alternatives and the proposed course.

Courses on reinforcement learning usually cover a wide range of RL methods, such as TD, Q-learning, SARSA, Policy Gradients, A2C, A3C, etc. These courses typically provide students with small tasks, such as implementing video game AI bots, to compare the benefits of these different RL methods. Take some available alternatives as examples. "Artificial Intelligence: Reinforcement Learning in Python" is a complete guide to AI and RL, which allow students to implement 17 different RL algorithms (Lazy Programmer Inc., 2019). "A Free course in Deep Reinforcement Learning from beginner to expert" guide students to implement some most popular deep reinforcement algorithms on various video games (Simonini, 2019). "Reinforcement Learning Specialization" is a collection of 4 RL courses that helps students to master the concepts of RL and various RL algorithms, and understand how to apply AI to solve real-world problems (University of Alberta, 2019).

Courses on computer games algorithms mainly focus on search techniques and their variants, such as minimax search, alpha-beta search, MCTS, etc. They may also include other AI related topics such as decision tree, Bayesian classifiers, evolutionary algorithms, or even basic NN tutorials. For example, "RISK AI Project" uses board game RISK as topic to teach some major search methods and decision tree. It also comes with a final tournament like the proposed courses (Archibald, 2019). "Implementing AI to Play Games" also comes with popular search techniques, besides, it also covers constraint satisfaction problem (CSP) and evolutionary algorithms (Packt Publishing, 2019). "Ultimate Tic Tac Toe AI Implementation" is a complete project that teach students different search strategies by extending a Tic Tac Toe program (Talaga, 2019).

More specifically, these courses focus on their areas of expertise, and they may allow students to learn more about either reinforcement learning or computer game algorithms. They may provide tutorials for various algorithms, but not as complete assignments as the projects listed in this paper. In short, this work aims to present a complete series of assignments that covers both reinforcement learning and computer game techniques, with a coherent goal of guiding students to develop their own strong AI programs of 2048-like games. In addition, unlike other alternatives that use simple topics as assignments, we guide students to follow up on the recent researches of 2048-like games. Students may therefore try new ideas on 2048-like games. Even more, their program performance may be comparable to that of some of the state-of-the-art programs. Takes works for Threes! in 2018 as examples. A student trained an agent by TD in Project 2, its 6144-tile reaching rate was 47.3%. Although its performance did not reach the level of TC (can up to 75% after 10M episodes), it is still a good result since TD-trained agents rarely achieve such a high 6144-tile reaching rate. Another example is that an

awesome TD-trained agent could reach 3072-tile with a probability of 78.3% in Project 4, where the environment was almost equivalent to the real Threes! games. This result even outperformed the best MS-TD-trained agent whose 3072-tile reaching rate was 67.8% (Yeh et al., 2017). These examples indicate that with our lectures, students were not only capable of gasping the key concepts, but also be able to improve current methods.

5. SUMMARY

Due to the popularity and simplicity of 2048, 2048-like games have become our staple application since 2014. From positive student feedback (4.21 and 4.35 points in average), experience sharing from students,[5] and gradual improvement in program strength over years of using 2048-like games as term projects, we think 2048-like games are highly suitable for educational purposes, especially for beginners to learn the well-known temporal difference learning. Simultaneously, the techniques used in student programs may inspire new research. It can be a good pedagogical tool to motivate young minds in joining our field and community.

ACKNOWLEDGEMENTS

This work was partially supported by the Ministry of Science and Technology (MOST) of Taiwan under Grant Number MOST 107-2634-F-009-011, MOST 108-2634-F-009-011, and MOST 109-2634-F-009-019 through Pervasive Artificial Intelligence Research (PAIR) Labs. The computing resources for student assignments were provided by the Computer Center of Department of Computer Science of National Chiao Tung University (NCTU CSCC).

REFERENCES

Archibald, C. (2019). RISK AI Project. Retrieved from http://modelai.gettysburg.edu/2019/risk.

Beal, D.F. & Smith, M.C. (1999). Temporal coherence and prediction decay in TD learning. In *International Joint Conferences on Artificial Intelligence, 1* (pp. 564–569). San Mateo, CA, USA: Morgan Kaufmann.

Cirulli, G. (2014). 2048, success and me. Retrieved from http://gabrielecirulli.com/articles/2048-success-and-me.

Guei, H., Wei, T.-H., Huang, J.-B. & Wu, I.-C. (2016). *An Empirical Study on Applying Deep Reinforcement Learning to the Game 2048, Workshop Neural Networks in Games in the International Conference on Computers and Games*. Leiden, The Netherlands: Springer.

Guei, H., Wei, T.-H. & Wu, I.-C. (2018). Using 2048-like games as a pedagogical tool for reinforcement learning. *ICGA Journal*, *40*(3), 281–293. International Conference on Computers and Games, New Taipei, Taiwan.

[5] Some experiences of students are shared (in Chinese) at following websites.
http://blog.sharknevercries.tw/2018/01/23/2584-AI.
https://junmo1215.github.io/tags.html#2584-fibonacci-ref.

Jaśkowski, W. (2017). Mastering 2048 with delayed temporal coherence learning, multi-stage weight promotion, redundant encoding and carousel shaping. *IEEE Transactions on Computational Intelligence and AI in Games*, *10*, 3–14. doi:10.1109/TCIAIG.2017.2651887.

Kondo, N. & Matsuzaki, K. (2019). Playing game 2048 with deep convolutional neural networks trained by supervised learning. *Journal of Information Processing*, *27*, 340–347. doi:10.2197/ipsjjip.27.340.

Lazy Programmer Inc. (2019). Artificial intelligence: Reinforcement learning in Python. Retrieved from https://www.udemy.com/course/artificial-intelligence-reinforcement-learning-in-python.

Matsuzaki, K. (2016). Systematic selection of N-tuple networks with consideration of interinfluence for game 2048. In *Conference on Technologies and Applications of Artificial Intelligence* (pp. 186–193). Hsinchu, Taiwan: IEEE.

Matsuzaki, K. (2017a). Developing a 2048 player with backward temporal coherence learning and restart. In *Advances in Computer Games* (pp. 176–187). Leiden, The Netherlands: Springer. doi:10.1007/978-3-319-71649-7_15.

Matsuzaki, K. (2017b). Evaluation of multi-staging and weight promotion for game 2048. Retrieved from Kochi University of Technology Academic Resource Repository. http://hdl.handle.net/10173/1564.

Matsuzaki, K. (2019). A further investigation of neural network players for game 2048. In *Advances in Computer Games*, Macao, China.

Matsuzaki, K. & Teramura, M. (2018). Interpreting neural-network players for game 2048. In *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)* (pp. 136–141). IEEE.

Neller, T.W. (2015). Pedagogical possibilities for the 2048 puzzle game. *Journal of Computing Sciences in Colleges*, *30*, 38–46.

Packt Publishing (2019). Implementing AI to play games. Retrieved from https://www.udemy.com/course/implementing-ai-to-play-games.

Russell, S.J. & Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Malaysia: Pearson Education Limited.

Simonini, T. (2019). A free course in deep reinforcement learning from beginner to expert. Retrieved from https://simoninithomas.github.io/Deep_reinforcement_learning_Course.

Sutton, R.S. & Barto, A.G. (1998). *Reinforcement Learning: An Introduction*. Cambridge: MIT Press.

Szubert, M. & Jaśkowski, W. (2014). Temporal difference learning of n-tuple networks for the game 2048. In *Computational Intelligence and Games* (pp. 1–8). Dortmund, Germany: IEEE.

Talaga, P. (2019). Using ultimate tic tac toe to motivate AI game agents. Retrieved from http://modelai.gettysburg.edu/2019/ultimatettt.

University of Alberta (2019). Reinforcement learning specialization. Retrieved from https://www.coursera.org/specializations/reinforcement-learning.

Wei, T.-J. (2019). A deep learning AI for 2048. Retrieved from https://github.com/tjwei/2048-NN.

Wu, I.-C., Yeh, K.-H., Liang, C.-C., Chang, C.-C. & Chiang, H. (2014). Multi-stage temporal difference learning for 2048. In *Conference on Technologies and Applications of Artificial Intelligence* (pp. 366–378). Taipei, Taiwan: Springer.

Yeh, K.-H., Wu, I.-C., Hsueh, C.-H., Chang, C.-C., Liang, C.-C. & Chiang, H. (2017). Multistage temporal difference learning for 2048-like games. *IEEE Transactions on Computational Intelligence and AI in Games*, *9*, 369–380. doi:10.1109/TCIAIG.2016.2593710.