JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 38, 775-790 (2022)
DOI: 10.6688/JISE.202207_38(4).0005

Monte-Carlo Simulation for Mahjong

JR-CHANG CHEN[1,+], SHIH-CHIEH TANG[2] AND I-CHEN WU[2,3]
[1]*Department of Computer Science and Information Engineering*
National Taipei University
New Taipei City, 23741 Taiwan
[2]*Department of Computer Science*
National Yang Ming Chiao Tung University
Hsinchu, 30050 Taiwan
[3]*Research Center for IT Innovation*
Academia Sinica
Taipei, 11529 Taiwan
E-mail: jcchen@mail.ntpu.edu.tw; fight5566jay@gmail.com; icwu@cs.nctu.edu.tw

Mahjong is a four-player, stochastic, imperfect information game. This paper focuses on the Taiwanese variant of Mahjong, whose complexity is higher than that of Go. We design a strong anytime Monte-Carlo-based Taiwanese Mahjong program. First, we adopt the flat Monte Carlo to calculate the win rates of all afterstates/actions such as discarding each tile. Then, we propose a heuristic method, which we incorporate into flat Monte Carlo to obtain the accurate tile to be discarded. As an anytime algorithm, we can stop simulations and return the current best move at any time. In addition, we prune bad actions to increase accuracy and efficiency. Our program, SIMCAT, won the championship in the Mahjong tournaments in Computer Olympiad 2020 and TAAI 2019/2020.

Keywords: Monte-Carlo simulation, discard-twice method, imperfect information game, Mahjong, progressive pruning

1. INTRODUCTION

Mahjong is a game originated from China, and is popular around the world with an estimation of about six hundred million players [24]. There are many different sets of Mahjong rules, such as the Japanese, Taiwanese, American, Beijing and Hong Kong rules. In Asia, this game does not only provide entertainment for amateurs, but also has many professional player associations and leagues [23]. Mahjong is a four-player, stochastic, imperfect information game. The complexity of Mahjong, estimated to be 4.3×10^{185}, is higher than that of Go [21, 25]. Moreover, there are more than 10^{48} hidden states during the course of a game [12]. Therefore, it is a challenge to design a strong program. In this paper, for simplicity of discussion, we adopt Taiwanese rules whose winning conditions are relatively simple.

Since Mahjong is a multi-player partially-observable imperfect information game, it is difficult to directly apply the techniques from perfect information games, such as alpha-beta search [7], Max^n [14] and Monte-Carlo tree search [6, 11]. Therefore, many programs, including VERYLONGCAT [5] and MAHJONGDAXIA [22], search based on a simplified model [5]. However, these programs do not use anytime search algorithms, and namely cannot recommend the best move to play until the whole search completes. This shortcom-

Received August 24, 2021; revised April 1, 2021; accepted April 27, 2021.
Communicated by Chang-Shing Lee.
[+] Corresponding author.

ing in turn leads to a shallow search and weak playing strength since time control must be considered strictly in tournament settings. Hence, it is better to have an algorithm that can stop the search and return the current best move at any time.

In this paper, we present a Monte-Carlo-based Mahjong program following Taiwanese rules. We use flat Monte Carlo [3] to approximate the win rates of given states as described in Section 3, and use progressive pruning [1, 2, 4] to prune inferior actions for better performance. Then, we design heuristics to enhance the strength in Section 4. In our experiments in Section 5, these methods are analyzed to show the performance. The best version of our programs, named SIMCAT, outperformed the baseline version with a win rate of 66.2% and won the championship in the Mahjong tournaments in Computer Olympiad 2020 and TAAI 2019/2020. The rules are introduced in Section 2, and the concluding remarks are given in Section 6.

2. BACKGROUND

In this section, we review Mahjong in Section 2.1 and the previous works in Section 2.2.

2.1 Mahjong

In this subsection, we only briefly mention the rules relevant to this paper. Detailed information refers to [15, 18].[1]

There are 144 *tiles* in Mahjong, among which 8 tiles are flowers and 136 tiles compose 34 *patterns*, each of which contains four identical tiles. In Fig. 1, the patterns are classified into five suits, which are *wan* (denoted by w), *tong* (or *pin*, denoted by *t*), *sou* (or *tiao*, denoted by *s*), wind and dragon. Each of wan, tong and sou includes nine number patterns, which are 1w to 9w, 1t to 9t, and 1s to 9s. Wind includes east, south, west and north. Dragon includes white, green and red.

A *meld* is a *sequence* (a.k.a. *shun* in Chinese), a *triplet* (a.k.a. *ker* in Chinese) or a *gong*. A sequence consists of three consecutive number tiles of the same suit, such as 1w, 2w and 3w, which are denoted by 123w for simplicity.[2] A triplet (or a gong) consists of three (or four) identical tiles, such as 111w (or 1111w). A *pair* consists of two identical tiles, such as 11w. A *tatsu* consists of two or three tiles and need one more tile to form a meld, such as 13w and 233w. Note that a pair is also a tatsu. A *block* is either a meld or a tatsu.

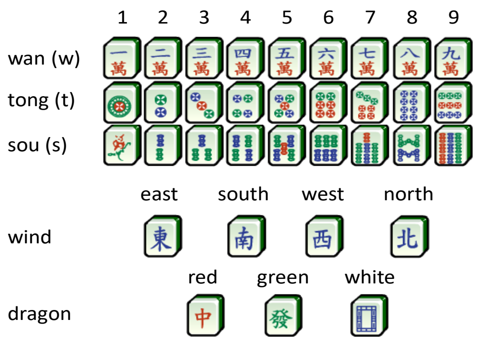


Fig. 1. The 34 patterns in Mahjong.

[1] This paper follows the terminologies and notations in [15].
[2] The simplified notations are used in this paper. For example, 111w represents the three tiles 1w, 1w, and 1w.

In Mahjong, four players around a square table take turns to play counterclockwise. From the view of the *current player* (that is, the game-playing program, denoted by C), the other three players are called the *lower player* (the player to the right), the *opposite player* (the player who is opposite to C) and the *upper player* (the player to the left) by turns, denoted by L, O and U respectively. At the beginning, all tiles are faced down and randomly piled up into the *wall*, and each player takes 16 tiles, called a *hand*, from the wall. Four players take turns to take a legal action (see below). The *winning condition* is satisfied for a player when his/her hand contains five melds and one pair. A player can win a game by picking a tile from the wall, or by taking the tile discarded by other players. When 16 tiles are left in the wall and no player wins, the game ends in a draw.

Legal actions include *pick* (a.k.a. *mo* in Chinese) and *steal* (a.k.a. *bid*). Pick indicates that the player takes a tile from the wall. Steal includes *eat*, *pong* and *gong*. Eat indicates that a player takes the tile discarded by the upper player to form a sequence. Pong indicates that a player takes the tile discarded by other players to form a triplet. Gong indicates that a player picks from the wall or takes the tile discarded by other players to form a gong, and then picks again. After a player picks or steals a tile in case of not winning yet, he/she has to *discard* a tile, maintaining a total of $3n + 1$ tiles, where the integer $0 \leq n \leq 5$. The ith round for a player indicates that it is the ith time he/she discards a tile.

2.2 Previous Works

Max^n is the generalization of minimax which can be applied to multi-player perfect information games [14]. A player in multi-player games makes a move that maximizes his/her return value.

Due to the imperfect information and the complexity in Mahjong, a simplified model was adopted where other players just picked a tile and then discarded it. For Taiwanese rules, VERYLONGCAT [5] used the expectimax tree [10] to compute the win rate of a given hand, and utilized expert knowledge for pruning. A lookup table for querying the *minimum number of tiles to win* (*MTW*) was built in advance. A transposition table was used to accelerate the computation. MAHJONGDAXIA used divide and conquer to decompose a hand, and also used a complicated heuristic function to choose a tile to be discarded [22].

In the past, many researches focused on training. For Japanese rules, Mizukami and Tsuruoka proposed a method that trained models including opponent models by using game records of expert players and decided moves using Monte-Carlo simulation together with these models [16]. Gao used supervised learning to train a convolution neural network [9]. The network without any search reached 2-dan on the well-known Japanese Mahjong online platform, Tenhou.[3] Microsoft made a Japanese Mahjong program, SUPHX, using deep reinforcement learning [12], and reached 10-dan on Tenhou. It required a considerable amount of computational resources for training.

Monte-Carlo sampling [13, 17] is a computational algorithm that uses random simulations to obtain numerical results. For move decision problems, flat Monte Carlo (flatMC) [3] has three stages, which are the generation of all possible states by playing each legal action, the simulation of these states, and the choice of the final action with the highest mean of simulated values. In this paper, we adopt flatMC to compute the win rate of a hand and propose heuristics to decide the move based on simulation results, so that the program

[3] Tenhou is available at http://tenhou.net/

is simple, effective, and suitable for the time constraint in tournaments since it is an anytime algorithm.

3. FLAT MONTE CARLO FOR MAHJONG

We adopt flat Monte Carlo (flatMC) as described in Section 2.2, which consists of three stages, for Mahjong. In this section, we describe the implementation of the simulation stage in detail. To use flatMC to compute win rates, we simply simulate legal actions for state transitions until a game ended. Given a state s, the next state after taking the action a from s is called an *afterstate* s_{af}, similar to the terminology used in [19]. Let S_{after} be the set of all afterstates of s. We simulate each $s_{af} \in S_{after}$ and obtain its win rate.

In Mahjong, we design the function FlatMCMJ(s_{af}, m, n_t) for the above process, where m is the number of rounds for each player in each simulation, and n_t is the total number of simulations. To reduce the complexity, the simulation stage uses an *optimistic* strategy. That is, in the beginning of a simulation, we generate m tiles for each player which he/she will pick from the wall in next m rounds. As we foresee all tiles, we can find an optimal solution to win by discarding useless tiles. Note that in FlatMCMJ, other players do not declare a win even though their hands satisfy the winning condition. Therefore, the result is either a win of the current player or a draw. Two simulation models are proposed below.

3.1 Single-Player Model

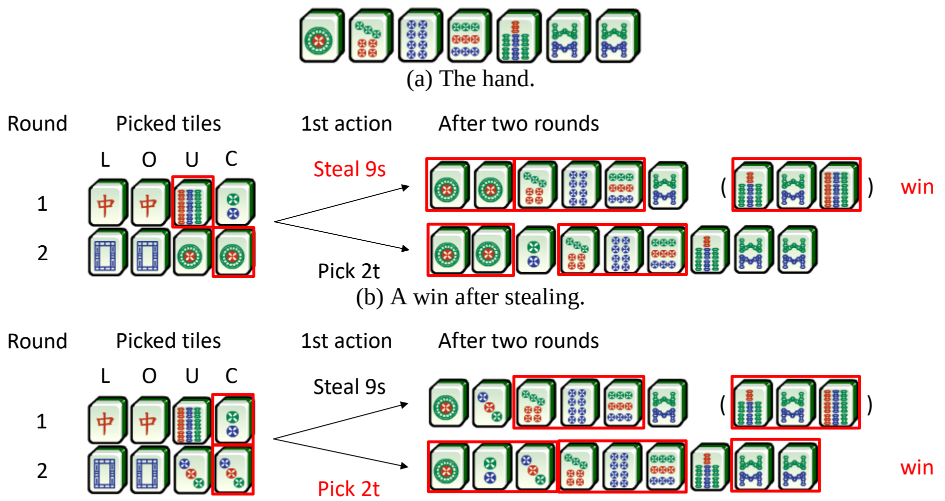
This model only simulates the pick action by the current player, and ignores the tiles discarded by other players. A tile is *hidden* from a player if it is in the wall or is in other players' hands. When a player picks a tile, the probability that the tile belongs to a pattern is calculated by the number of hidden tiles of the pattern divided by the number of total hidden tiles. For each simulation, the player repeatedly takes one tile m times but does not discard any tile. So, his/her hand contains $16 + m$ tiles in the end. The player wins when 17 tiles among the $16 + m$ tiles satisfy the winning condition, and otherwise it is a draw.

The advantage is that the implementation is simple. Since it is not necessary to consider the tiles other players discard, we only need to generate the current player's tiles. A characteristic of FlatMCMJ is to reserve as many blocks as possible, which will be discussed in detail in Section 4.

3.2 Four-Player Model

Based on the single-player model, this model additionally simulates the steal action by the current player. We assume that other players are dummy players, just picking a tile and then discarding it.

During a simulation, due to the uncertainty of Mahjong, when other players discard a tile, the player may win after stealing it or after ignoring it (that is, picking another tile). For example, assume that the hand is 1789t788s in Fig. 2 (a). In next two rounds, a player may win by stealing 9s and then picking 1t in Fig. 2 (b), or not stealing 9s (that is, picking 2t) and then picking 3t in Fig. 2 (c). Therefore, we store and simulate both hands such that we can choose the best solution from these results.



(a) The hand.

(b) A win after stealing.

(c) A win after not stealing.

Fig. 2. Examples to illustrate both stealing and not stealing should be considered. The notations, L, O, U, and C denote the lower player, the opposite player, the upper player, and the game-playing program, respectively.

3.3 Implementation

In Fig. 3, the functions, FlatMCMJ and OneSim, implement the four-player model. In FlatMCMJ, we generate all afterstates s_{af} of a given state s. Then, we simulate for each s_{af} in OneSim and can elaborate its win rate. The number of total simulations n_t is evenly distributed to each s_{af}. The best action is the one with the highest win rate. In OneSim, let S_i and S_{i+1} be the set of current states and the set of afterstates. There are m rounds in one simulation. In each round, all afterstates generated from the current states in S_i become the new set S_{i+1} in the next round. After m rounds, the player wins if there exists a winning strategy in the set of states, and otherwise it is a draw.

In each round, we generate at most seven afterstates for each hand by taking legal actions which are one pick, three pongs and three eats. For example, assume that the hand is 234456w1199t258s, and the four tiles, t_L, t_O, t_U and t_C, picked by the four players are 1t, 9t, 4w and 2s respectively. The seven afterstates generated are one hand that contains the picked tile 2s, three hands that contain triplets, 111t, 999t and 444w, and three hands that contain sequences, 234w, 345w and 456w. After m rounds, at most 7^m hands are generated, and hence the computational cost grows exponentially.

Function OneSim(s_{af}, m)
Input: an initial afterstate s_{af}, the number of total rounds m
Output: win or draw

```
1.   $S_0 \leftarrow \{s_{af}\}$
2.   for $i$ = 0 to $m$ 1 do
3.       $S_{i+1} \leftarrow \varnothing$
```

```
4.          get four tiles $t_L$, $t_O$, $t_U$ and $t_C$, each of which is assigned to the lower,
                opposite, upper and current players respectively
5.          **for each** $s$ in $S_i$ **do**
6.              **while** a pong occurs **do**
7.                  $s' \leftarrow$ remove the two tiles identical to the discarded tile from $s$
8.                  $S_{i+1} \leftarrow S_{i+1} \cup \{s'\}$
9.              **end while**
10.             **if** an eat occurs **then**
11.                 **for** at most three combinations of eat **do**
12.                     $s' \leftarrow$ remove the two tiles related to $t_U$ from $s$
13.                     $S_{i+1} \leftarrow S_{i+1} \cup \{s'\}$
14.                 **end for**
15.             **end if**
16.             $s' \leftarrow$ add $t_C$ to $s$                    /* pick */
17.             $S_{i+1} \leftarrow S_{i+1} \cup \{s'\}$
18.         **end for**
19.     **end for**
20.     **if** there exists a winning condition in $S_m$ **then**
21.         **return** win
22.     **end if**
23.     **return** draw
```

```
**Function**    FlatMCMJ($s$, $m$, $n_t$)
**Input:** an initial state $s$, the number of rounds $m$, the number of total simulations $n_t$
**Output:** the best action
**Local variables:** an action array $A[MAX\_ACTIONS]$,
                     an integer array $winCount[MAX\_ACTIONS]$

1.      $A \leftarrow$ All actions from $s$
2.      Initialize all elements in $winCount$[] to 0
3.      **for** each $a_i$ in $A$ **do**
4.          $s_{af} \leftarrow$ make $a_i$ at $s$                /* $s_{af}$ is an afterstate of $s$ */
5.          **Repeat** $n_t/|A|$ times **do**
6.              $winCount[i] \leftarrow$ accumulate the win count by calling OneSim($s_{af}$, $m$)
7.          **end repeat**
8.      **end for**
9.      **return** the action with the highest win count in $winCount$[]
```

Fig. 3. The algorithm for FlatMCMJ of the four-player model of Mahjong.

4. DISCARD-TWICE METHOD

This section discusses an important issue called *tatsu-breaking* in Mahjong. The winning condition is five melds and one pair, which implies exactly six blocks. If a hand has more than six blocks, we call the hand has *excessive blocks*. Consider the hand ht = 445w1267t1144888999s with seven blocks. The hand will discard a tile from a tatsu eventually because only six blocks are needed to satisfy the winning condition. The breaking operation is so-called to *break a tatsu* in this paper, such as discarding 1t from the tatsu 12t. In this example, FlatMCMJ will discard 4w and the hand will still maintain seven blocks. In reality, we do not know which tatsu is useless before the winning condition is satisfied, but it is a win in FlatMCMJ since the algorithm does not need to know which tiles to be discarded while playing. Consequently, FlatMCMJ may overestimate the simulated win rate in this case. This causes that FlatMCMJ tends to reserve as many blocks as possible.

Therefore, we propose rule-based heuristics to find a reasonable tatsu-breaking option to cope with this problem. We describe a method to calculate the number of blocks of a hand in Section 4.1 and two heuristics in Section 4.2.

4.1 Calculate the Number of Blocks of a Hand

We propose a method that can quickly calculate the number of blocks for a given hand. First, the *minimum number of tiles to win*, called *MTW*, is the least number of tiles we need to pick or steal to satisfy the winning condition for a given hand [5]. Based on the definition, we let the function MTW(c) be the minimum number of tiles that the player has to pick and not discard to form c – 1 meld and one pair. Hence, MTW(6) is the minimum number of tiles to win. For example, for the hand *ht* = 445w1267t1144888999s, MTW(1) = MTW(2) = MTW(3) = 0, MTW(4) = 1, MTW(5) = 2, MTW(6) = 3, MTW(7) = 4 and MTW(8) = 6. The first three are zeros, since there are already one pair and two melds, say 44w, 888s and 999s. For MTW(4), we simply need to add one more tile, say 3t, to form one new meld for the tatsu 12t. Similarly, for MTW(5), MTW(6) and MTW(7), we only need to add one extra, say 8t, 1s and 4s, respectively. However, for MTW(8), we need two tiles, say 67w, to form an extra meld, since it runs out of tatsu after making MTW(7). Apparently, the function is monotonically increasing.

Second, if $\mathrm{MTW}(c) - \mathrm{MTW}(c-1) \geq 2$, it is impossible to find a block besides those included in $\mathrm{MTW}(c-1)$. That is, it needs two tiles to form an extra meld or a pair when computing MTW(c). Thus, there are at most $c-1$ blocks. For example, there are at most seven blocks in *ht* since $\mathrm{MTW}(8) - \mathrm{MTW}(7) = 2$. $\mathrm{MTW}(c) - \mathrm{MTW}(c-1) \leq 1$, the hand has at least one block that is not included in the blocks when computing $\mathrm{MTW}(c-1)$. Thus, there are at least c blocks. For example, there are at least seven blocks in *ht* since MTW(7) – MTW(6) = 1. Therefore, the number of blocks for a given hand is the maximum c such that

$$\mathrm{MTW}(c) - \mathrm{MTW}(c-1) \leq 1. \tag{1}$$

4.2 Rule-Based Heuristics

In this subsection, we propose the *discard-twice* method to cope with excessive blocks. This method considers the first discarded tile t_1 and the next discarded tile t_2. The best t_1 is discarded according to heuristics.

First, given a hand *ht*, we calculate the number of blocks after discarding each tile t_1 in *ht* using the method in Section 4.1. Then, we classify the tiles into two sets, called the *tatsu-breaking set* and the *non-tatsu-breaking set*. If the number of blocks decreases after t_1 is discarded, then t_1 belongs to the tatsu-breaking set. Otherwise, t_1 belongs to the non-tatsu-breaking set. For example, there are five blocks in *ht* = 445w1267t1144s. If 1t is discarded, then the afterstate 445w267t1144s has four blocks since the maximum *c* satisfying Eq. (1) is four. If 4w is discarded, the afterstate 45w1267t1144s has five blocks. Hence, 1t is in the tatsu-breaking set and 4w is in the non-tatsu-breaking set. In *ht*, the eight tiles 1267t1144s are in the tatsu-breaking set, and the three tiles 445w are in the non-tatsu-breaking set.

Second, we consider the effect of breaking a tatsu by consecutively discarding two tiles, t_1 and t_2 in *ht*. We calculate the win rate for each afterstate using FlatMCMJ described in Section 3.3, and obtain the following data.

- The discarded tile t_{1tb} with the best win rate wr_{1tb} in the tatsu-breaking set
- The set of discarded tiles $\{t_{1tb}\} \subseteq ht - t_{1tb}$ with good win rates wr_{2tb} after t_{1tb} is discarded
- The set of discarded tiles $\{t_{1ntb}\}$ with good win rates wr_{1ntb} in the non-tatsu-breaking set
- The discarded tile t_{2ntb} with the best win rate wr_{2ntb}, where $t_{2ntb} \in ht - t$ for each tile $t \in \{t_{1ntb}\}$ that is discarded

Given an action, the mean value *m* and the standard deviation σ are calculated after lots of simulations. The *confidence interval* of the action is between the lower bound $m - r_d \times \sigma$ and the upper bound $m + r_d \times \sigma$, where r_d is a constant ratio. A *good* win rate wr_{2tb} means that the confidence interval of wr_{2tb} overlaps with that of the best win rate among the tiles in $\{t_{2tb}\}$. That is, the upper bound of three standard deviations 3σ of wr_{2tb} is equal to or greater than the lower bound of 3σ of the best in wr_{2tb} assuming $r_d = 3$. A good win rate wr_{1ntb} is defined similarly.

Based on the above data, we propose two rule-based heuristics to decide which tile is eventually discarded and give examples below.

(A) Heuristic 1 (H1): Choose the tile t_1 whose next discarded tile achieves the best win rate.

The idea is to break the tatsu in this round if the player is forced to break a tatsu in the next round. First, we discard two tiles at once for all combinations of two tiles in *ht*. Then, we analyze the win rates of afterstates by FlatMCMJ. Among all possible tile t_2, we choose the one whose next discarded tile t_2 achieves the best win rate. If t_1 is in the tatsu-breaking set, the tatsu is broken.

For example, in Table 1, given the hand *ht* = 445w1267t1144s, the best win rate is 16.7% obtained by discarding 1t and then 2t. Hence, the tile to be discarded in this round is 1t and the tatsu 12t is broken. In contrast, FlatMCMJ discards 4w since it only considers discarding one tile. However, if the player picks a useful tile such as 3w, 6w, 5t, 8t, 1s and 4s, the player must break a tatsu and lead to a lower win rate, which cannot be detected by FlatMCMJ. By applying H1, this situation is considered by discarding two tiles, and the tatsu can be broken.

Table 1. The decision made by H1. The hand is 445w1267t1144s.

	Discard the 1st tile	Discard the 2nd tile
Tatsu-breaking	t_{1tb} = **1t** wr_{1tb} = 18.7%	t_{2tb} = 2t wr_{2tb} = **16.7**%
Non-tatsu-breaking	t_{1ntb} = 4w wr_{1ntb} = 21.2%	t_{2ntb} = 1t wr_{2ntb} = **13.4**%

(B) Heuristic 2 (H2): Choose the good tile t_1 that appears both in the non-tatsu-breaking set in this round and in the tatsu-breaking set in the next round.

Although H1 can break a tatsu, it would be too aggressive sometimes because it is possible to pick or steal a good tile before discarding t_2. Take the hand ht = 145w1267t 1144s as an example. Obviously, a tatsu like 12t usually has a better chance of forming a meld than a single tile like 1w. However, H1 only considers the win rate after discarding t_2. After discarding the single tile 1w, the hand becomes 45w1267t1144s, and all discarding actions will break a tatsu in the next round. That is, no matter which tile is discarded in the first round, a tatsu is forced to be broken after two rounds in this case. So, the hands after two rounds are similar, and their sampled win rates are close by FlatMCMJ. In Table 2, H1 chooses to discard 1t since its wr_{2tb} = 14.0% is slightly better than 13.5%. Therefore, H1 cannot distinguish which one is better between discarding a single tile and breaking a tatsu.

The idea of H2 is to discard the tile which is less likely to form to a meld before breaking the tatsu. After discarding the tile 1t in the tatsu 12t in this round in the tatsu-breaking set, discarding 1w or 2t in the next round may get close win rates. Thus, $\{t_{2tb}\}$ includes 1w. On the other hand, 1w is less likely to form a meld. Discarding 1w often gets a high win rate by FlatMCMJ and thus $\{t_{1ntb}\}$ includes 1w. Hence, H2 chooses the single tile 1w in $\{t_{1ntb}\}\cap\{t_{2tb}\}$ rather than breaks the tatsu 12t.

Table 2. The decision made by H2. The hand is 145w1267t1144s.

	Discard the 1st tile	Discard the 2nd tile
Tatsu-breaking	t_{1tb} = 1t wr_{1tb} = 15.1%	t_{2tb} = **1w** wr_{2tb} = **14.0**%
Non-tatsu-breaking	t_{1ntb} = **1w** wr_{1ntb} = **21.3**%	t_{2ntb} = 1t wr_{2ntb} = **13.5**%

(C) The Combination of FlatMCMJ and Heuristics

The property of FlatMCMJ is that it tends to reserve as many blocks as possible. H1 makes the decision according to the win rates after two tiles are discarded consecutively. H2 discards the tile which is hard to form a meld prior to breaking a tatsu. In this subsection, we design three versions of FlatMCMJ to mix these properties as follows.

- FlatMCMJ$_{\text{original}}$: Choose t_{1tb} if $wr_{1tb} \geq wr_{1ntb}$, and choose a tile by FlatMCMJ otherwise. It indicates that when breaking a tatsu is judged to be good in this round, we do not consider the next round. This version is the same as the original FlatMCMJ.
- FlatMCMJ$_{\text{H1}}$: Choose t_{1tb} if $wr_{1tb} \geq wr_{1ntb}$, and choose a tile by H1 otherwise. This version deals with the problem that FlatMCMJ$_{\text{original}}$ tends to reserve excessive blocks.
- FlatMCMJ$_{\text{H2+H1}}$: Choose t_{1tb} if $wr_{1tb} \geq wr_{1ntb}$, and choose a tile by H2 otherwise. If no tile is chosen by H2, namely $\{t_{1ntb}\}\cap\{t_{2tb}\} = \emptyset$, choose a tile by H1. This version suppresses the tendency of FlatMCMJ$_{\text{H1}}$ towards aggressively breaking a tatsu.

5. EXPERIMENTS

The experiments are done on a desktop computer with an AMD Ryzen 5 2600 6-core processor. Taiwanese rules are adopted. There are two teams, each of which includes the two players sitting on the opposite side and uses the same version of the program. A match includes 384 games according to the Computer Olympiad tournament [20].[4] The team that wins more games is the winner of the match. To avoid the influence of luck, each wall is used in two games, so both teams can play the same hand once.

We compare the two models of FlatMCMJ in Section 5.1. We experiment with progressive pruning to accelerate the computation in Section 5.2. Different numbers of simulations are compared in Section 5.3 The discard-twice method is discussed in Section 5.4. Finally, the best version is compared with the baseline in Section 5.5.

5.1 Comparison of SP and FP Models of FlatMCMJ

We compare the performance of the single-player model (SP) and the four-player model (FP) of FlatMCMJ proposed in Section 3. In Table 3, FP outperforms SP reaching a win rate of 62.1%. The reason is that SP ignores all steals during simulations. The results also show that the steal action is important in Mahjong. We use FlatMCMJ(FP) for the following experiments.

Table 3. Comparison of the single- and four-player models. A total of 781 matches and win rates with 95% confidence.

team	# of win matches	win rate
FlatMCMJ(SP)	228	37.9% ($\pm$0.67%)
FlatMCMJ(FP)	374	62.1% ($\pm$0.67%)

5.2 The Results of Using Progressive Pruning

Progressive pruning (PP) [1, 2, 4, 13] adopts the confidence interval to prune inferior actions/moves during search, and thus facilitates finding out the best action. The process to find the best action includes many iterations. In each iteration, we compute the confidence interval of each action by simulations. Inferior actions are pruned (see below), and if exactly one action is left, the process ends and returns the action immediately. Since the total number of simulations is fixed in the whole process, superior actions will obtain more simulations in the next iteration after inferior actions are pruned.

The confidence interval of an action is $(m - r_d \times \sigma, m + r_d \times \sigma)$, where m, σ and r_d are the mean value, the standard deviation and a constant ratio, respectively, after lots of simulations. Assume that a node has two actions a and b. The mean and standard deviation of a are m_a and σ_a respectively, and those of b are m_b and σ_b respectively. The action a is inferior to b if $m + r_d \times \sigma_a < m_b - r_d \times \sigma$, which indicates the upper bound of a is less than the lower bound of b. Hence, the action a is pruned.

This subsection analyzes the effectiveness of progressive pruning in Mahjong. First, we analyze the pruning rate δ affected by the constant ratio r_d and the number of simulations n of each action. When r_d is set to a smaller value or when σ decreases, the confidence interval becomes smaller, filtering out more actions. In Fig. 4, $r_d = 2$ filters out more actions than $r_d = 3$. The two lines of r_d are close (less than 5%) when $n \geq 9000$. When n increases, σ decreases and thus δ increases. In Fig. 4, δ reaches 79.8% for $r_d = 2$ and 75.0% for $r_d = 3$ when $n = 10000$.

[4] In the Mahjong tournament of the Computer Olympiad, a match includes 384 games; however, 192 games before 2014.

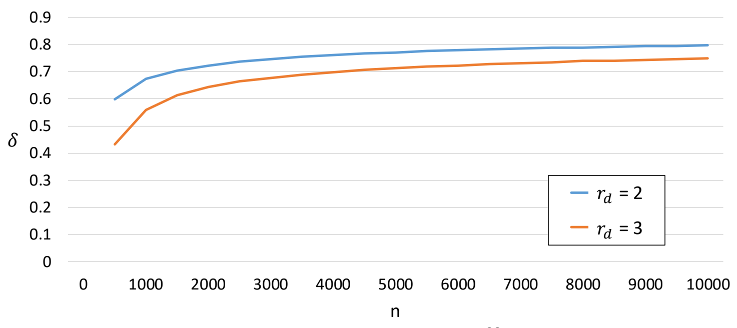


Fig. 4. The pruning rate δ under different n and r_d.

Second, we investigate the percentage δ_e of the best actions that are filtered out. Assume that the best action is obtained by making 15,000,000 simulations for each action of a hand. As shown in Fig. 5, $\delta_e < 0.2\%$ when $n > 300$ for $r_d = 2$, and $\delta_e \approx 0$ for $r_d = 3$. Almost all of the best actions are reserved by progressive pruning. We adopt $r_d = 3$ in the following experiments.

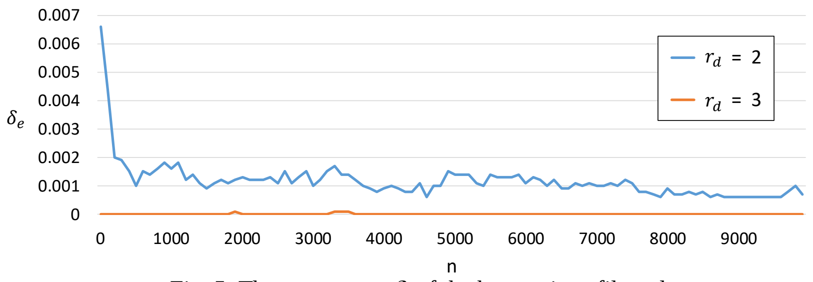


Fig. 5. The percentage δ_e of the best actions filtered out.

Third, we compare the effectiveness of using and not using progressive pruning, denoted by FlatMCMJ(FP) and FlatMCMJ(FP)+PP respectively. In the experiments, the total number of simulations is 10000 for each hand. In FlatMCMJ(FP)+PP, let each action simulates 300 times in the first iteration and 100 times in the following iterations. Table 4 shows that FlatMCMJ(FP)+PP performs slightly better than FlatMCMJ(FP). Since some poor actions are pruned, the saved simulations are applied to good actions.

Table 4. Comparison of using and not using pruning.

team	# of win matches	win rate
FlatMCMJ(FP)	1162	49.7% (±1.70%)
FlatMCMJ(FP)+PP	1177	50.3% (±1.70%)

5.3 Comparison using Different Numbers of Simulations

We compare different numbers of simulations for each hand. Table 5 shows the win rates of FlatMCMJ(FP)+PP using 20000, 40000, 80000 and 200000 simulations against that using 10000 simulations. The results show that more simulations we use, better performance we have. In the following experiments, we adopt 200000 simulations which achieve the best win rate of 53.5%.

Table 5. Comparison between different numbers of simulations (against 10000 simulations).

# of simulations	# of matches	# of win matches	win rate
20,000	8558	4415	51.5% (±0.89%)
40,000	2435	1275	52.3% (±1.67%)
80,000	1604	849	52.9% (±2.05%)
200,000	1188	635	53.5% (±2.38%)

5.4 The Results of the Discard-twice Method

The discard-twice method includes two heuristics, H1 and H2, as mentioned in Section 4. We compare the performance of the three versions, $\text{FlatMCMJ}_{\text{original}}$, $\text{FlatMCMJ}_{\text{H1}}$ and $\text{FlatMCMJ}_{\text{H2+H1}}$, which does not use any heuristics, only uses H1, and uses both H1 and H2, respectively. In this experiment, the four-player model (FP) and progressive pruning (PP) are used in all versions.

In Table 6 (a), $\text{FlatMCMJ(FP)}_{\text{H1}}$+PP weakens the strength. In practice, H1 makes the program tend to break a tatsu even when a single tile exists in a hand. H2 makes up this disadvantage. Hence, $\text{FlatMCMJ(FP)}_{\text{H2+H1}}$+PP is the best, reaching a win rate of 59.5%, as shown in Table 6 (b).

Table 6. Comparison of the two heuristics of the discard-twice method.
(a) The version using no heuristics vs. the version using H1.

team	# of win matches	win rate
$\text{FlatMCMJ(FP)}_{\text{original}}$ +PP	150	53.8% (±4.91%)
$\text{FlatMCMJ(FP)}_{\text{H1}}$ +PP	129	46.2% (±4.91%)

(b) The version using no heuristics vs. the version using both H1 and H2.

team	# of win matches	win rate
$\text{FlatMCMJ(FP)}_{\text{original}}$ +PP	133	40.5% (±4.46%)
$\text{FlatMCMJ(FP)}_{\text{H2+H1}}$ +PP	195	59.5% (±4.46%)

While performing the best, the heuristic H2+H1 incurs little overhead. We use 10000 hands that have excessive blocks as testing data. Each hand is given 200000 simulations in total. Note that PP is not used to make sure all simulations are executed. Table 7 lists the execution time for the three versions, $\text{FlatMCMJ(FP)}_{\text{original}}$, $\text{FlatMCMJ(FP)}_{\text{H1}}$ and $\text{FlatMCMJ(FP)}_{\text{H2+H1}}$. The results show that the three versions consume nearly the same computation cost and that the overhead of heuristics H1 and H2+H1 is very small, about 4%.

Table 7. Computation time for different heuristics with 200000 simulations.

	$\text{FlatMCMJ(FP)}_{\text{original}}$	$\text{FlatMCMJ(FP)}_{\text{H1}}$	$\text{FlatMCMJ(FP)}_{\text{H2+H1}}$
Execution time (sec.)	5.015 (±0.0163)	5.214 (±0.0170)	5.215 (±0.0170)

5.5 Comparison with the Baseline

The final version, FlatMCMJ(FP)$_{H2+H1}$+PP, adopts the four-player model, progressive pruning and the discard-twice method. We compare its performance with that of the baseline version, FlatMCMJ(SP). Both versions simulate 200000 times. As shown in Table 8, the final version outperforms the baseline, reaching a win rate of 66.2%.

Table 8. Comparison of the final version and the baseline.

team	# of win matches	win rate
FlatMCMJ(SP)	214	33.8% (±3.10%)
FlatMCMJ(FP)$_{H2+H1}$ + PP	419	66.2% (±3.10%)

6. CONCLUSIONS

This paper describes the design of our Monte-Carlo-based Mahjong program, SIMCAT. We propose the single-player and four-player models for Mahjong that are used in the simulation stage in flat Monte Carlo. Moreover, we design the discard-twice method that includes two rule-based heuristics.

In the experiments, the version that uses flat Monte Carlo with the four-player model, progressive pruning and the discard-twice method outperforms the baseline that uses the single-player model, reaching a win rate of 66.2%. SIMCAT used the above methods and won the championship in the Mahjong tournaments in Computer Olympiad 2020 and TAAI 2019/2020.

Our work provides the basis for Mahjong programs. Several possible future works can be developed based on our work. First, the discard-twice can be extended to discard *N* (*N* ≥ 3). However, in the case of a given fixed number of simulations for each hand, the average simulation count for each discard-*N* action decreases significantly, resulting in inaccurate win rates for discard-*N* actions. Second, our model can be applied to other Mahjong rules, such as American rules and Hong Kong rules. Third, based on flatMC, the search may choose Monte Carlo Tree Search for further investigation. Fourth, our methods may be merged with deep reinforcement learning, such as AlphaZero [7].

ACKNOWLEDGMENT

This research is partially supported by the Ministry of Science and Technology (MOST) of Taiwan under Grant Number 110-2634-F-009-022, 110-2634-F-259-001 through Pervasive Artificial Intelligence Research (PAIR) Labs, and 108-2221-E-305-008-MY3. The computing resource is partially supported by National Center for High-performance Computing (NCHC) of Taiwan. Thank Fong-Sheng Liao for the discussions that inspired us to design the method.

REFERENCES

1. B. Bouzy, "Move-pruning techniques for Monte-Carlo go," *Advances in Computer Games,* 2005, pp. 104-119.

2. B. Bouzy and B. Helmstetter, "Monte-Carlo go developments," *Advances in Computer Games*, 2004, pp. 159-174.
3. C. B. Browne, *et al.*, "A survey of Monte Carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, 2012, pp. 1-43.
4. G. M. J. B. Chaslot, J. T. Saito, B. Bouzy, J. W. H. M. Uiterwijk, and H. J. van den Herik, "Monte-Carlo strategies for computer go," in *Proceedings of the 18th Belgium-Netherlands Conference on Artificial Intelligence*, 2006, pp. 83-91.
5. L.-K. Chuang and I.-C. Wu, "A study of Mahjong program design," Master's Thesis, Department of Computer Science, National Chiao Tung University, Taiwan, 2015. (in Chinese)
6. R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Proceedings of the 5th International Conference on Computers and Games*, 2006, pp. 72-83.
7. D. Silver, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv Preprint*, 2017, arXiv:1712.01815.
8. D. J. Edwards and T. Hart, "The alpha beta heuristic," Artificial Intelligence Project, RLE and MIT Computation Center, 1963, 5 pages.
9. S. Gao, F. Okuya, Y. Kawahara, and Y. Tsuruoka, "Building a computer Mahjong player via deep convolutional neural networks," *arXiv Preprint*, 2019, arXiv:1906.02146.
10. T. G. Hauk, "Search in trees with chance nodes," Master's Thesis, Department of Computer Science, University of Alberta, Canada, 2004.
11. L. Kocsis and C. Szepesvári, "Bandit based Monte-Carlo Planning," in *Proceedings of the 15th European Conference on Machine Learning*, 2006, pp. 282-293.
12. J. Li, *et al.*, "Suphx: Mastering Mahjong with deep reinforcement learning," *arXiv Preprint*, 2020, arXiv:2003.13590.
13. F.-S. Liao, "Monte-Carlo sampling methods for Computer Mahjong," Master's Thesis, Department of Computer Science, National Chiao Tung University, Taiwan, 2017. (in Chinese)
14. C. A. Luckhardt and K. B. Irani, "An algorithmic solution of N-person games," in *AAAI-86 Proceedings*, Vol. 1, 1986, pp. 158-162.
15. S. D. Millter, *Riichi Mahjong: The Ultimate Guide to the Japanese Game Taking the World by Storm*, lulu.com Publisher, US, 2016.
16. N. Mizukami and Y. Tsuruoka, "Building a computer Mahjong player based on Monte Carlo simulation and opponent models," in *Proceedings of IEEE Conference on Computational Intelligence and Games*, 2015, pp. 275-283.
17. A. Shapiro, "Monte Carlo sampling methods," *Handbooks in Operations Research and Management Science*, Vol. 10, 2003, pp. 353-425.
18. Y.-C. Shan, C.-H. Wei, C.-H. Lin, I-C. Wu, L.-K. Chuang, and S.-J. Tang, "A framework for computer Mahjong competitions," *ICGA Journal*, Vol. 37, 2014, pp. 44-56.
19. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, MA, 2018.
20. W.-J. Tseng, L.-K. Chuang, I-C. Wu, S.-S. Lin, and S.-J. Yen, "Longcat wins mahjong tournament in ICGA 2013," *ICGA Journal*, Vol. 36, 2013, pp. 184-185.
21. H. J. van den Herik, J. W. Uiterwijk, and J. van Rijswijck, "Games solved: Now and in the future," *Artificial Intelligence*, Vol. 134, 2002, pp. 277-311.

22. C.-W. Wu and S.-S. Lin, "The design and implementation of Mahjong program MahJongDaXia," Master's Thesis, Department of Computer Science and Information Engineering, Taiwan, 2016. (in Chinese)
23. Japan Mahjong Professional League, http://www.ma-jan.or.jp/.
24. Mahjong Time, online Mahjong platform, http://mahjongtime.com/mahjong-about-mahjong-time.html/.
25. Math of Mahjong (麻雀の数学), http://www10.plala.or.jp/rascalhp/mjmath.htm/.

Jr-Chang Chen (陳志昌) is an Associate Professor of the Department of Computer Science and Information Engineering at National Taipei University. He received his BS, MS and Ph.D. degrees in Computer Science and Information Engineering from National Taiwan University in 1996, 1998, and 2005 respectively. He served as the Secretary General of Taiwanese Association for Artificial Intelligence in 2015-2017. Dr. Chen's research interests include artificial intelligence and computer games. He is the co-author of the two Chinese chess programs named ELP and CHIMO, the Chinese dark chess program named YAHARI, and the minishogi program named NYANPASS, which won gold medals in the Computer Olympiad tournaments. He served as the general chair of the 10th International Conference on Computers and Games (CG2018).

Shih-Chieh Tang (唐士傑) is currently a Ph.D. candidate in the Department of Computer Science, National Yang Ming Chiao Tung University. His research interests include artificial intelligence, machine learning and computer games. He is the leader of the team developing the Mahjong program, named SIMCAT, which won gold medals in TAAI 2019/2020 and Computer Olympiad 2020.

I-Chen Wu (吳毅成) is currently the Executive Officer of Artificial Intelligence Computing Center at Academia Sinica, a Research Fellow of Research Center for IT Innovation at Academia Sinica, and also a professor of the Department of Computer Science at National Yang Ming Chiao Tung University. He received his BS in Electronic Engineering from National Taiwan University, MS in Computer Science from NTU, and Ph.D. in Computer Science from Carnegie-Mellon University, in 1982, 1984 and 1993, respectively. He serves the Editor-in-Chief of ICGA Journal and an associate editor in the IEEE Transactions on Games. He currently serves as the Vice President of the International Computer Games Association, and the president of the Taiwanese Computer

Games Association; and served as the President of the Taiwanese Association for Artificial Intelligence in 2015-2017.

His research interests include computer games and deep reinforcement learning, and his research achievements include several state-of-the-art game playing programs, winning over 30 gold medals in international tournaments, like Computer Olympiad. He wrote over 150 technical papers, and served as chairs and committee in over 30 academic conferences and organizations, including the general chair of IEEE CIG conference 2015.