An Algorithmic Design and Implementation of Outer-Open Gomoku

Chih-Hung Chen, Shun-Shii Lin, and Yen-Chi Chen
Dept. Computer Science and Information Engineering
National Taiwan Normal University
Taipei, Taiwan, R.O.C.
e-mail: honhonzone@gmail.com, linss@csie.ntnu.edu.tw, zxkyjimmy@gmail.com

***Abstract*—The Outer-Open Gomoku is a new Gomoku game with three goals: "maintaining connect five", "simple rule" and "fairness". This paper introduces the design idea and the implementation of our Outer-Open Gomoku program OOGiveMeFive. It takes advantage of the modern instruction set architecture, inverse bitboard, reducing instruction numbers and search space, etc., to obtain a good performance of searching more than 56 million nodes per second using a typical personal computer. Our program won 6 gold medals among 8 recent computer games competitions.**

Keywords-outer-open gomoku; gomoku; bitboard; threat-space search; relevance-zone

I. INTRODUCTION

Gomoku has a very long history; it is a two-player zero-sum game of perfect information. This game is usually played on a board with 15×15 intersections. Black and White players place individual stone alternately on an empty intersection. The winner is the player who first makes five (or more) of his stones in a row horizontally, vertically, or diagonally.

Free Style Gomoku is also called Gobang, Five in a Row, or Connect5. Its board size is roughly the same as that of Go and its search space complexity is very large. But the free-style Gomoku (without any restriction on Black) is known to favor Black over White, hence it is realized as an unfair game long ago. As the advances in computer algorithms, in 1994, Allis used "threat space search" [1] and "proof number search" [2-3] to prove that Free Style Gomoku is a Black win game. Hence, there were lots of new game rules proposed to against this situation, such as Renju, Swap2. However, the game rules were becoming more and more complex such that most players cannot remember all the rules. To the best of our knowledge, this strange phenomenon didn't appear in other games. In 2001, Wagner and Virag proved that the new rule "Renju" [4] was also unfair-it also let black win. Therefore, Free Style Gomoku and Renju could not become a competition item in most human or computer tournaments.

Aiming at the serious difficulty of Gomoku, the game of Outer-Open Gomoku was proposed to hope to give Gomoku a newborn chance. We will introduce this new game in the following section.

II. OUTER-OPEN GOMOKU

Outer-Open Gomoku was proposed by Professor Lin, Shun-Shii in 2011 and was formally presented at the 2012 Conference on Technologies and Applications of Artificial Intelligence [5]. Lin's rule has three goals: "maintaining connect five", "simple rule" and "fairness", to try to rescue the fate of Gomoku. The rule only restricts Black to play at the two outer rows (i.e. row 1, 2, 14, or 15) or columns (i.e. column A, B, N, or O) of the board for the first move. From the second move, there are no prohibited moves for both sides. The game is a draw if the board is filled and no horizontal, vertical or diagonal line of 5 stones has been created.

The game of Gomoku played with Lin's rule is called Outer-Open Gomoku. The Black player has 104 choices (one of the two outer rows) for his first move, and then both players will be free to place their stones anywhere. By this rule, it reduces the favor to Black player and becomes more intuitive than those rules which need switch the roles between Black and White.

The game Outer-Open Gomoku was firstly played in formal in a big tournament at TCGA (Taiwan Computer Game Association) 2014. In the same year, it was also held as an event of the computer tournament at TAAI (Technologies and Applications of Artificial Intelligence) 2014. In 2015, Outer-Open Gomoku was included as part of the ICGA (Computer Olympiad) 2015. After that, Outer-Open Gomoku was held at TCGA, ICGA, TAAI with 3~6 participating programs from Taiwan, China, and USA every year. There are 8 Outer-Open Gomoku tournaments over recent years, and Black players won 70 games within these 132 matches. It seems like a balance game to Black and White players. The statistics of the results over the years are shown in Fig. 1 in which our program-OOGiveMeFive won 6 gold medals among 8 recent computer games competitions.

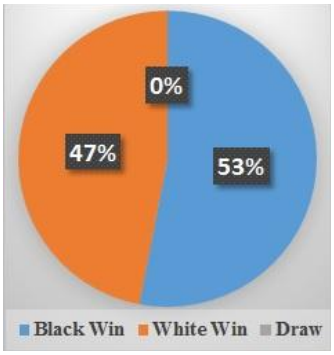

Figure 1. Statistics of the results for past competitions.

978-1-5386-0539-4/17/$31.00 ©2017 IEEE

III. OOGIVEMEFIVE'S ARCHITECTURE

A. Pattern Definition

We refer to the characters of Gomoku [1], Renju [4], and Connect6 [6] to define the patterns by the threat-based idea. The definitions of patterns are introduced as follows:

1) Multi-threat: Assume the defender cannot win the game. The attacker is said to have Multi-threat on a line if the defender needs to place more than one stone to avoid the attacker winning the game in the next move (See Fig. 2 a, b, and b' for Black player).

2) Single-threat: Assume the defender cannot win the game. The attacker is said to have Single-threat on a line if the defender needs to place just one stone to avoid the attacker winning the game in the next move (See Fig. 2 c and c' for Black player).

3) Live3-threat: The attacker is said to have Live3-threat on a line if the attacker can place one more stone to become Multi-threat (See Fig. 2 d and d' for Black player).

4) Dead3-threat: The attacker is said to have Dead3-threat on a line if the attacker can place one more stone to become Single-threat (See Fig. 2 e and e' for Black player).

5) Live2-threat: The attacker is said to have Live2-threat on a line if the attacker can place two more stones to become Multi-threat (See Fig. 2 f and f' for Black player).

6) Dead2-threat: The attacker is said to have Dead2-threat on a line if the attacker can place two more stones to become Single-threat (See Fig. 2 g and g' for Black player).

7) Live1-threat: The attacker is said to have Live1-threat on a line if the attacker can place three more stones to become Multi-threat (See Fig. 2 h for Black player).

8) Dead1-threat: The attacker is said to have Dead1-threat on a line if the attacker can place three more stones to become Single-threat (See Fig. 2 i for Black player).

9) Dead-point: The attacker is said to have Dead-point on a line if the attacker cannot generate threat anyway (See Fig. 2 j).

B. Data Structure and Codec

In order to implement a high-performance program, we adopt bitboard[7-8] as the data structure. A bitboard is commonly a 32-bit or 64-bit representation of the board, but there are 225 positions (15×15) in Outer-Open Gomoku. So we need 4 64-bit unsigned integers for keeping the stone information of one player. For row-major ordering, the pattern of vertical or diagonal direction will cross more than one 64-bit unsigned integer. With the goal of efficiency to maintain the bitboards, we take advantage of the modern instruction set architecture, such as Advanced Vector Extensions 2 (AVX2), Bit Manipulation Instructions 2 (BMI2), to enhance the performance in our program.

For the codec of bitboard corresponding to the game board, we take bit-alignment design to faster fetch the pattern and clean the representation. We add two walls on the left and bottom of the game board, as shown in Fig. 3, which makes data alignment (power of two) on a row/column [9]. It is easy to transform between the position and the coordinate of the board by using bitwise operations in place of other instructions which need more clock cycles (e.g. division and modulo operation). If we need to rotate the bitboard for implementing a transposition table, we can swap the high bits with the low bits or calculate 2's complement of the high bits. For example, the position 40 is represented as $(0010\ 1000)_2$, the relevant operations are as follows:

- Column (40) = $(0010\ 1000)_2$ & $(0000\ 1111)_2$ = 8 (column H)
- Row (40) = $(0010\ 1000)_2$ >> 4 = 2 (row 2)
- $R_{y=x}(40) = (1000\ 0010)_2 = 130$ (reflect over y=x)
- $R_{x\text{-axis}}(40) = (1110\ 1000)_2 = 232$ (reflect over x-axis)
- $R_{y=-x}(40) = (1000\ 1110)_2 = 142$ (reflect over y=-x)

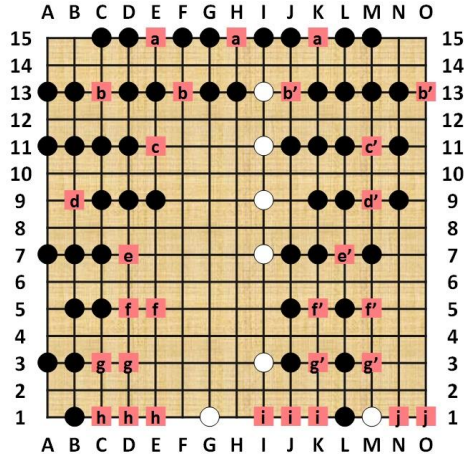

Figure 2. The threat-based patterns for Outer-Open Gomoku.

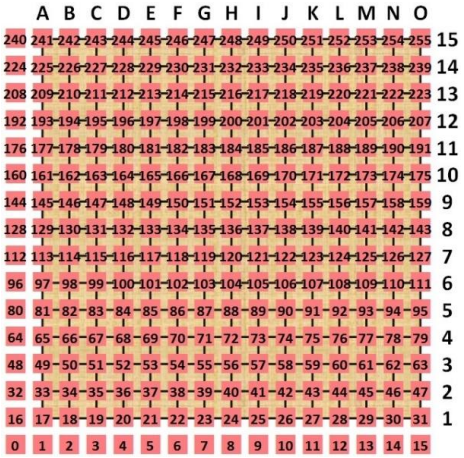

Figure 3. The codec of bitboard.

C. Bitboard Representation

In the typical bitboard representation, a bit is set to 1 if the corresponding position of the board is occupied. It requires two bitboards, Black and White, to represent the information of the game. But unlike normal bitboard design, we set the bit as 0 if the corresponding position of the board is occupied. For example, the bits are initially set to $(1111\ 1111\ 1111\ 1110)_2$ for the 2nd row of the Black bitboard. It

will be changed to $(1111\ 1110\ 1111\ 1110)_2$ when the Black player places a stone on position 40 (H2).

Fig. 4 illustrates the inverse bitboard representation in practice. The partial bits of 5th row are $(1111\ 1\ 0000)_2$ and $(1011\ 1\ 1110)_2$ in the Black and White bitboards respectively. These bitboards are maintained by incremental updating. The pair (Black(position), White(position)) is $(00)_2$ if this position is a Wall. The pairs $(01)_2$, $(10)_2$, and $(11)_2$ indicate a Black stone, White stone, and Empty respectively.

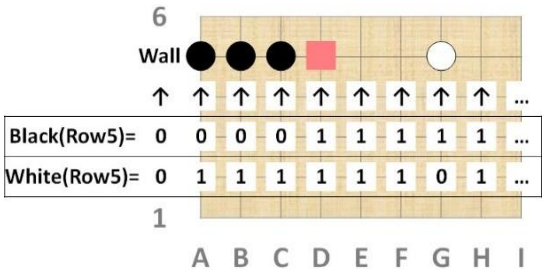

Figure 4. Inverse bitboard representation in practice.

D. Move Generation

The state-space complexity of the Outer-Open Gomoku is really high (roughly 10^{105}). So the performance of move generation is very important because it will be executed multitudinous times during the game tree search. For the bitboard move generation, it can use bitwise operation to *and* its 2's complement to find the least significant 1 bit (LS1B) quickly. We take Fig. 4 as an example and compare the operation of the inverse bitboard with traditional bitboard. TABLE I shows that fewer bitboards and fewer operations are needed for the inverse bitboard. It means using inverse bitboard can reduce the executing time and storage space.

TABLE I. COMPARISON WITH THESE TWO BITBOARDS

Operation	Traditional Bitboard	Inverse bitboard
Black	000001111	111110000
White	010000001	101111110
Occupied	Black | White = 010001111	Not needed
Empty	~ Occupied = 101110000	Black & White = 101110000
LS1B	Empty & (-Empty) = 10000	Empty & (-Empty) = 10000

E. Pattern Fetching

Consider a single line only, a stone will affect the four nearest positions on its left side and also the right side, and therefore the length of the pattern is nine positions. Unlike extracting pattern from positions step by step, we can gain good performance for pattern extraction by using the table lookup technique to obtain a pattern immediately. It is known that the central bit of a pattern is the position we want to evaluate and is an Empty position. So we can ignore this central bit to reduce the table size. We use pext instruction (parallel bits extract) to extract bits of a pattern except the central bit. For example, assume we want to evaluate D5 in Fig. 4, the result of pext operation is shown below.

- //Instr. Destination, Source, Mask
- pext Index_B, 111110000, 111101111
- pext Index_W, 101111110, 111101111
- //Combine Index, Index = 1111 0000 1011 1110
- Index = (Index_B << 8) | Index_W

The pattern status is saved as an element with 16 bits, each bit indicates whether the corresponding pattern is occupied. For this design, a higher bit has a higher priority, the order of patterns is listed in TABLE II. By the way we use sliding window accumulator in place of Live1-threat and Dead1-threat (bit 3-0) in order to offset the bias occurred by the pattern. It implies that higher density has higher scores, as illustrated in Fig. 5.

TABLE II. BIT STATUS IN SCORE TABLE

Bit	Status
15	Multi-threat of the attacker
14	Multi-threat of the defender
13	Single-threat of the attacker
12	Single-threat of the defender
11	Live3-threat of the attacker
10	Live3-threat of the defender
9	Dead3-threat of the attacker
8	Dead3-threat of the defender
7	Live2-threat of the attacker
6	Live2-threat of the defender
5	Dead2-threat of the attacker
4	Dead2-threat of the defender
3 - 0	Sliding window accumulation

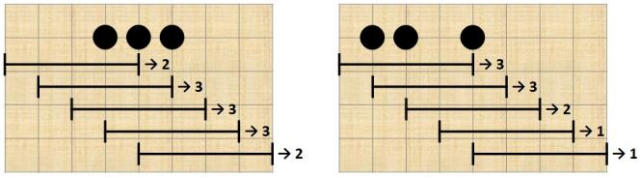

Figure 5. The sliding window accumulator.

The 16-bit index is generated by shifting the Black as the higher bits and combining the White as the lower bits. There are 2^{16} elements in our score table, each element has 2 bytes. Thus the table size is $2^{16} \times 2$ bytes = 128K bytes. This score table is tiny enough that can be loaded into L2 cache in our computer. It makes the operations of table lookup very fast when the program is executed.

To continue the example in Fig. 4, we show how the sliding window accumulator (SWA) works in practice. Status(Index) = $(0010000000011001)_2$, the most significant 1 bit (MS1B, bit 13) means there is a Single-threat for the attacker, the MS2B (bit 4) indicates there is a Dead2-threat for the defender, and the least 4 bits get 9 points (4+3 points for attack, and 2 points for defend in Fig. 6), which are accumulated as:

$$P(x) = \begin{cases} 1, & \text{if } x = \text{Color} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$L(x) = \begin{cases} 1, & \text{if } x = \text{Color or Empty} \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

$$SWA = \sum_{Color=Attack}^{Defend} \sum_{S.W.=1}^{5} \sum_{i=1}^{5} 1(L(x)=5)(P(x)+1) \quad (3)$$

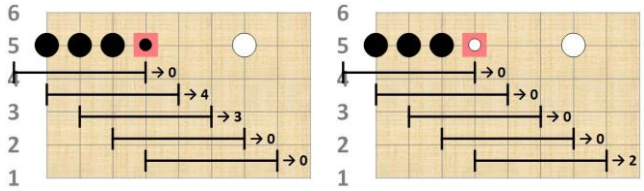

Figure 6. Sliding window accumulator works in practice.

F. Reduce Search Space

Because of less restriction in Outer-Open Gomoku, the number of legal moves is pretty large. So it desires to reduce the search space by some heuristics. We use M mask (the shape of a ✳ defined as Fig. 7) to mask out those probable moves and ignore most of moves that are far away from fighting field. The set M is a finite union of sets M_1, M_2, ..., M_n, defined as:

$$M=\bigcup_{i=1}^{n} M_i = M_1 \cup M_2 \cup \ldots \cup M_n \quad (4)$$

where M_i means the mask of the i-th move.

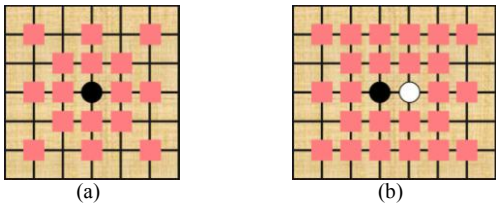

Figure 7. M mask: (a) update by the 1st move, (b) update by the 2nd move.

The set M can be represented by a bitboard and maintained with incremental updating step by step. Fig. 7 (a) illustrates how the M mask reduces the search space from 224 to 16 positions labeled as red squares. Although the M mask can decrease lots of search space, but it is still too enormous. We will reduce the search space again in searching stage, and will describe that in the following section.

G. Search Algorithm

In this work, we use alpha-beta pruning as the major framework for search stage, and integrate threat-space search [1-3], null move heuristic, and relevance-zone-oriented proof search [10] into alpha-beta pruning. The threat is usually the key feature in connecting games, so as Outer-Open Gomoku. Consider a single line only, the Single-threat and the Live3-threat are the important features to start a threat-space search, and the terminal condition is to find a Multi-threat defined in Section 3.A. During the game tree search, it will switch on threat-space search when a Single-threat or a Live3-threat is found in alpha-beta search. If we cannot start the threat-space search or find a winning path in alpha-beta level, the null move heuristic will be used to lead a threat. We take Fig. 8 (a) as an example to describe how to find a winning strategy by using the null move, the threat-space search, and the relevance-zone-oriented proof search. Since the 5th move played by Black on D8 cannot find a winning path, so we try to explore C7 by Black then a null move occurred (the 6th move by White is passed). Next, we check M mask to sieve out those positions which have the Live3-threats by Black (red squares a~h in Fig. 8 (b)).

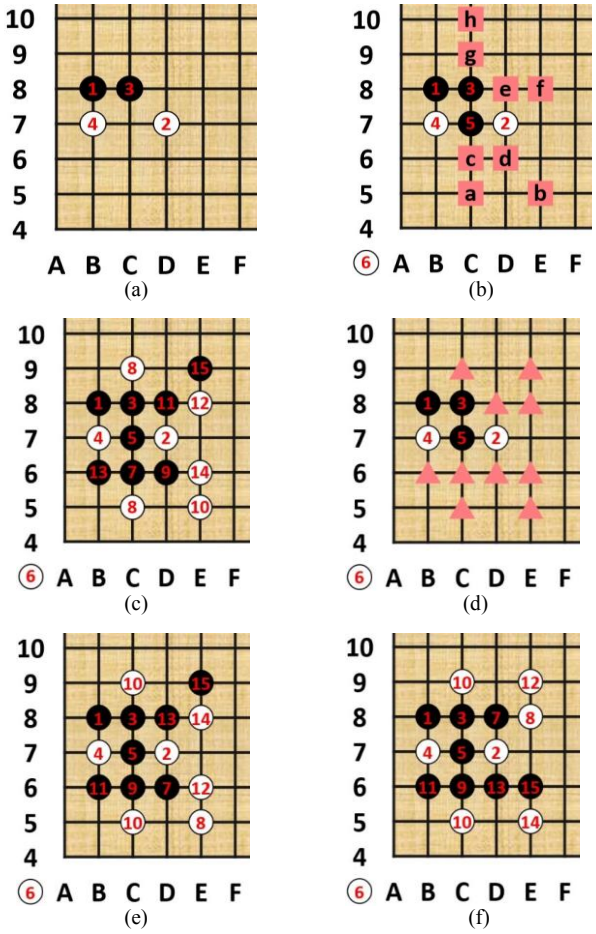

Figure 8. Relevance-zone in practice: (a) Giving board, (b) Live3-threat positions, (c) VCLT started from C6, (d) Relevance-zone by C6, (e) VCLT started from D6, (f) VCLT started from D8.

Those positions a~h will be tested with the threat-space search and the strongest defense by White one by one. The term "strongest defense" denotes the defender will place his stones on the positions where the attacker can generate a threat. In other words, the defender could place one or two stones for a single move. Therefore, the attacker must win the game when he can still find a winning path against the strongest defense; on the other hand, it doesn't mean the attacker has no change to win the game if the attacker cannot find a winning path against the strongest defense. Because the defender may place two stones for a move, the attacker may loss some opportunity to exactly win the game during the game tree search. Although the strongest defense may cause inexact result of game tree search, but it can quickly test whether a winning path is existed. It always

explores only one branch for the defender who places two stones at the same time to merge two sub-trees together. This trick narrows down the most profitable avenues of search.

In Fig. 8 (b), the positions a and b by Black cannot lead a winning path, but Black places on the position c could win the game with the strategy redefined as victory by continuous Live3-threat (VCLT, see Fig. 8 (c)) [6], [11]. The other key winning strategy for Outer-Open Gomoku is redefined as victory by continuous Single-threat (VCST) [6], [11].

When a winning path is found, the relevance-zone is formed with those triangle positions as shown in Fig. 8 (d). It is well-known to us that White must place the 6th move on one of the triangle positions, or Black will win the game by replaying the same sequence. So White must check all the triangle positions in relevance-zone to see whether Black still win the game by a threat-space search. If all the triangle positions are loss for White, Black was proved to win the game by playing at C6.

In Fig. 8 (b), if the position c (C6) is not proved to win for Black, then the position d will be checked, and so on. In this example, the positions c, d, and e have a winning strategy by VCLT as shown in Fig. 8 (c), (e), and (f) respectively, and with the same relevance-zone (see Fig. 8 (d)). We can see those relevance-zones are highly overlapping. A better way to reduce these redundancies is to use the transposition table to save the information of positions previously searched. Another way is to union all the relevance-zones first (i.e. $Z = Z_1 \cup Z_2 \cup \ldots \cup Z_n$), then check all the positions in Z. Thus the positions which are overlapping would not be checked again.

H. Parallel Search

In our design, we consider the possibility of parallel search because the evaluation task could be divided into multi-thread processing. When the program is estimating a position, the non-parallel programming will evaluate horizontal, vertical, and diagonal lines sequentially, then accumulate the scores from four directions. So it would spend more time in searching because there are many idle threads (see Fig. 9). And the speedup would approximate to the number of threads used.

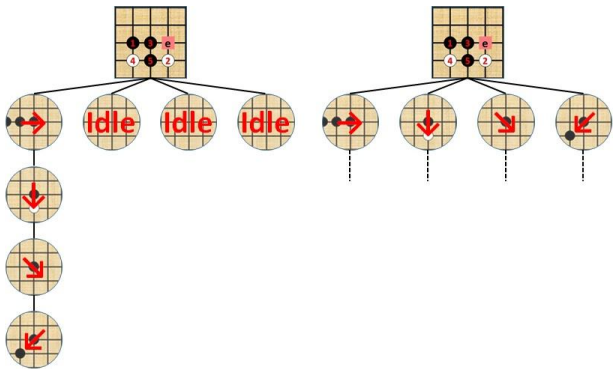

Figure 9. Non-parallel vs. parallel programming in search stage.

IV. Conclusions and Future Works

We take advantage of modern instruction set architecture, such as AVX2, BMI2, and also optimize the bitboard design, make good use of bitwise operations, reduce search space, parallel search, etc., to enhance the performance of our program. By implementing everything above, we get a satisfactory number: 56,636,596. This result shows our program can search more than 56 million nodes per second by using a typical personal computer with a standard CPU (Intel i7-5820, 3.30 GHz) and 32GB RAM. Although we reach a good performance, it was stall accused of weakness in evaluation function.

With the development of Artificial Intelligence, recently the deep learning techniques have been brought into focus. We hope to copy the success of Go (AlphaGo), No-Limit Hold'em (Libratus and DeepStack), and Chess (Giraffe) to build a value network instead of our current evaluation function to strengthen the power of our program in the near future.

Acknowledgment

This research was supported by the Ministry of Science and Technology(R.O.C.) under grants MOST 104-2221-E-003-009-MY2 and MOST 105-2218-E-259-001.

References

[1] L. V. Allis, H. J. van den Herik, and M. P. H. Huntjens, "Go-Moku Solved by New Search Techniques," Computational Intelligence, vol. 12, 1996, pp. 7-23.

[2] L.V. Allis, M. van der Meulen, and H.J. van den Herik, "Proof-Number Search," Artificial Intelligence, vol. 66 (1), 1994, pp. 91-124.

[3] L.V. Allis, "Searching for Solutions in Games and Artificial Intelligence," Ph.D. Thesis, University of Limburg, 1994.

[4] J. Wágner and I. Virág, "Solving Renju," ICGA Journal, vol. 24 (1), 2001, pp. 30-34.

[5] S.-S. Lin and C.-Y. Chen, "How to Rescue Gomoku? The Introduction of Lin's New Rule (in Chinese)," The Conference on Technologies and Applications of Artificial Intelligence (TAAI 2012), 2012.

[6] I-C. Wu, D.-Y. Huang, and H.-C. Chang, "Connect6," ICGA Journal, vol. 28 (4), 2005, pp. 235-242.

[7] B. Boskovic, S. Greiner, J. Brest, and V. Zumer, "The Representation of Chess Game," 27th International Conference on Information Technology Interfaces, 2005.

[8] E. A. Heinz, "How DarkThought Plays Chess," ICCA Journal, vol. 20 (3), 1997, pp. 166-176.

[9] C.-H. Chen and S.-S. Lin, "The Design and the Implementation of Outer-Open Gomoku Program-OOGiveMeFive (in Chinese)," The Conference on Technologies and Applications of Artificial Intelligence (TAAI 2014), 2014.

[10] I-C. Wu and P.-H. Lin, "Relevance-Zone-Oriented Proof Search for Connect6," the IEEE Transactions on Computational Intelligence and AI in Games, vol. 2(3), 2010, pp. 191-207.

[11] I-C. Wu and D.-Y. Huang, "A New Family of k-in-a-row Games," the 11th Advances in Computer Games Conference (ACG'11), Taipei, Taiwan, September 2005.