Theoretical Computer Science 815 (2020) 79–94

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On solving the 7,7,5-game and the 8,8,5-game

Wei-Yuan Hsu [a], Chu-Ling Ko [a], Jr-Chang Chen [b], Ting-Han Wei [a], Chu-Hsuan Hsueh [a,c], I-Chen Wu [a,*]

[a] *Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan*
[b] *Department of Computer Science and Information Engineering, National Taipei University, New Taipei City, Taiwan*
[c] *School of Information Science, Japan Advanced Institute of Science and Technology, Nomi, Ishikawa, Japan*

A R T I C L E   I N F O

Article history:
Received 25 August 2019
Received in revised form 2 February 2020
Accepted 17 February 2020
Available online 21 February 2020
Communicated by A. Fink

Keywords:
k-in-a-row games
mnk-games
Positional games
Pairing strategy
Relevancy-zone

A B S T R A C T

An *mnk*-game is a kind of *k*-in-a-row game played on an $m \times n$ board, where two players alternatively mark empty squares with their own colors and the first player who gets *k*-consecutive marks (horizontally, vertically, or diagonally) wins. In this paper, we present an AND/OR search tree algorithm specifically for proving *mnk*-games. We first propose three novel methods to reduce the branching factor of AND/OR search trees. We also propose a new method to find pairing strategies, which further accelerate the proof of *mnk*-games. The combined methods drastically speed up the proof for the 7,7,5-game, which is solved in 2.5 seconds. Moreover, this paper is the first to solve the 8,8,5-game, which is proven as a draw within 17.4 hours.

1. Introduction

The family of *mnk*-games, also known as *k-in-a-row* games and classified as a type of positional game [5][6], is defined as follows. Two players, Black and White, alternately mark with their own color an empty square on a board, say a chess board, and Black plays first. The player who first gets *k*-consecutive *marks* (horizontally, vertically, or diagonally) *wins* the game. A game ends in a *draw* if no players reach *k*-consecutive marks. The terminology *mnk*-game specifies that the board size is $m \times n$, and *k* denotes the winning condition as described previously [17][21]. Well-known examples of games in this family include tic-tac-toe (the 3,3,3-game), and free style Gomoku (the 15,15,5-game).

In the past, many *mnk*-games have been solved [24]. The 15,15,5-game was solved as a win for Black using tree search in 1994 [1]. Despite being a smaller game, the 7,7,5-game was solved in 2013 [8] by decomposing it into 10 separate sub-games, for which the total computation time took around 5 days. The theoretical value of the 8,8,5-game has yet to be proven. This is somewhat counterintuitive, since games tend to be more difficult to solve as the state space increases. The most likely reason for the 7,7,5-game to be more difficult to solve is that it is a game that results in a draw. One property of *mnk*-games is that as the board size increases, Black's advantage increases [1]. With a bigger board, there are more viable strategies for Black to win, which can narrow the search down significantly through pruning. This may not be possible for smaller games such as the 7,7,5-game and the 8,8,5-game, which we both prove to be draws in this paper.

* Corresponding author.
E-mail addresses: in13_13@aigames.nctu.edu.tw (W.-Y. Hsu), southra@aigames.nctu.edu.tw (C.-L. Ko), jcchen@mail.ntpu.edu.tw (J.-C. Chen), ting@aigames.nctu.edu.tw (T.-H. Wei), hsuehch@aigames.nctu.edu.tw, hsuehch@jaist.ac.jp (C.-H. Hsueh), icwu@aigames.nctu.edu.tw (I-C. Wu).

https://doi.org/10.1016/j.tcs.2020.02.023
0304-3975/© 2020 Elsevier B.V. All rights reserved.

In this paper, we investigate the conditions leading to draw positions in *mnk*-games, and propose a series of techniques to solve *mnk*-games as draws. Using the combined algorithm, our program can shorten the solving time for the 7,7,5-game from several days to 2.5 seconds, and can also successfully solve the 8,8,5-game as a draw within 17.4 hours.

The remainder of this paper is organized as follows. In Section 2, we provide the necessary background knowledge on positional games, solved *mnk*-games, pairing strategies, and potential weights. In Section 3, we present an improvement on the AND/OR tree algorithm specifically for proving *mnk*-games. In Section 4, we propose three reduction techniques that include partial pairing, vertex domination, and Breaker r-zones (defined in Section 4). In Section 5, we propose the techniques for finding pairing strategies. In Section 6, we provide the experiment results and discussions. In Section 7, we conclude this paper. This paper is extended from the preliminary version [14] as follows. First, propose Breaker r-zones. Second, significantly improve the description of our algorithm by also adding some more proofs for those properties in Section 4 and the details in Section 5. Third, further improve the solving time which was about 1 minute for the 7,7,5-game and 3 days for 8,8,5-game in the preliminary version.

2. Background

This section reviews background knowledge, including positional games in Subsection 2.1, solved *mnk*-games in Subsection 2.2, pairing strategies in Subsection 2.3, and potential weights in Subsection 2.4.

2.1. Positional games

We start defining positional games formally by first introducing the hypergraph. A *hypergraph* is a tuple (V, E) where V is a set of *vertices* and E is a set of *hyperedges*, each of which is a non-empty subset of V. A *positional game* [5][6] is then defined as a two-player game played on a given hypergraph, where the two players alternately mark p and q unmarked vertices in V with their own color, where p and q are the number of vertices each player has to mark in his/her turn. In this paper, (V, E) is also used to denote a specific game position for simplicity of discussion.

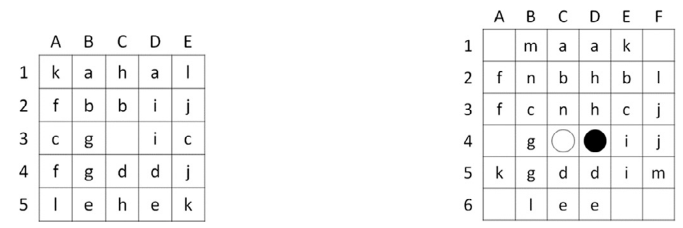
A positional game is called a *strong positional game* (or a *Maker-Maker positional game*) in [5][6][11], in the case when a player wins by being the first to *complete* a hyperedge, i.e. he/she marks all vertices on a hyperedge with his/her own color. The marked/unmarked status is an attribute of a vertex $v \in V$. If all vertices in V are marked and no player wins, the game ends in a draw. The family of *mnk*-games is an example of a strong positional game. Another perspective called *Maker-Breaker* is introduced for the following reasons. *Strategy stealing* provides a weak solution to play symmetric strong positional games (i.e. $p = q$), where the result indicates the second player cannot win [9][10][13][25][26]. That is, the best the second player can do is to force the first player to draw. In a Maker-Breaker positional game, there is no draw and two players win under different conditions. The first player, Maker, wins by completing a hyperedge as is the case in strong positional games. The second player, Breaker, wins by *blocking* all hyperedges, i.e. Breaker marks at least one vertex on each hyperedge to prevent Maker from completing the hyperedge. For simplicity, the game ends immediately when either Maker completes one hyperedge or all hyperedges are blocked. Note that Breaker does not win by completing any hyperedges. A win for the second player in Maker-Breaker is equivalent to a draw in Maker-Maker in a symmetric positional game.

In the context of *mnk*-games, V is the set of all squares on an $m \times n$ board, E is the set of all hyperedges composed by horizontal, vertical and diagonal k-consecutive squares on the board, and both p and q are 1. For simplicity of discussion in the rest of this paper, we adopt the Maker-Breaker variant and try to solve *mnk*-games by proving a win for Breaker. Since Black (the first player) has an initiative advantage, as a Maker-Breaker game, Black will be the Maker and White the Breaker. More specifically, if we prove that Breaker wins a Maker-Breaker game, the game corresponds to either a draw or a win for White (the second player) under the strong positional game rules, as is the original *mnk*-game. Furthermore, since *mnk*-games are symmetric positional games, a win for Breaker implies a draw in the original *mnk*-game.

2.2. Solved mnk-games

Allis et al. used many methods including threat space search and proof-number search on top of the well-known AND/OR tree algorithm to solve the 15,15,5-game (referred to as Gomoku in Allis' work), which was proven to be winning for Black [1][2]. In particular, threat space search is a game search technique which minimizes the game tree search space through branch reduction. The main idea of threat space search on the 15,15,5-game is the concept of *threats*. A threat is a situation where the opponent is forced to block in a specific square to avoid losing immediately; i.e. whenever a threat exists, the branching factor for the opponent is reduced to 1. Threat space search can reduce the game tree search space significantly for strong positional games, but since it is non-trivial to define threats for the Maker-Breaker variant, the technique cannot be directly applied to prove draws. Proof number search is a technique for further reducing the search space used in the tree search [3][18].

In the context of draw games, Berlekamp et al. [7] stated that according to an observation of C.Y. Lee, the second player can force a tie for the 5,5,4-game. This result implies that the 5,5,5-game is a draw. Berlekamp et al. also mentioned that the 5,5,5-game can be directly solved with a specific pairing strategy [13][22][23]. The 6,6,5-game was proven to be a draw by Uiterwijk and van den Herik [21]. In the same paper, the concept of pairing strategies was also mentioned. The 7,7,5-game was solved as a draw by Chen [8], using a combination of depth-first proof-number search [3][15], and a number



	A	B	C	D	E
1	k	a	h	a	l
2	f	b	b	i	j
3	c	g		i	c
4	f	g	d	d	j
5	l	e	h	e	k

(a) A pairing strategy for the initial position of the 5,5,5-game.

	A	B	C	D	E	F
1		m	a	a	k	
2	f	n	b	h	b	l
3	f	c	n	h	c	j
4		g	○	●	i	j
5	k	g	d	d	i	m
6		l	e	e		

(b) A pairing strategy for a middle position of the 6,6,5-game.

Fig. 1. Examples of pairing strategies. It is Maker's turn to move.

of *mnk*-game heuristics for move ordering and branch pruning. These include prioritizing moves near marked vertices and avoiding marking vertices that do not form 5-consecutive marks for either player. Chen's method took about 5 days by using 10 threads on a machine equipped with a CPU @ 3.10 GHz.

While threat space search cannot be applied directly to the Maker-Breaker game, its core concept of reducing the branching factor to accelerate tree search is useful. We propose three methods for Maker-Breaker branch reduction in Section 4. The pairing strategy is also a key component of our method. The pairing strategy in general is covered in Subsection 2.3, and our algorithm for finding specific pairings is described in full in Section 5.

2.3. Pairing strategy

Pairing strategies [13] were used to prove a win for Breaker in the 5,5,5-game [7], the 6,6,5-game [21] and the ∞,∞,9-game [12]. Finding pairing strategies for arbitrary positional games was shown to be NP-hard [11]. In a pairing strategy, some unmarked vertices are assigned into a set of disjoint pairs such that each hyperedge contains at least one pair. If Maker plays at any vertex of one of these pairs, Breaker can prevent Maker from completing any hyperedge by responding immediately at the other vertex in the same pair. A pairing strategy for the initial position of the 5,5,5-game is illustrated in Fig. 1(a), where vertices with the same lower-case letters are assigned into a pair. Note that pairing strategies are not unique for a position.

To be more general, pairing strategies are extended to *middle games* where some vertices are already marked. If a hyperedge e contains a vertex marked by Breaker, the theoretical value of the game $(V, E - e)$ is the same as that of (V, E). Such hyperedges are said to be *redundant*. For example, in Fig. 1(b), the hyperedges passing the white mark at C4 are redundant, including A4-E4, B4-F4, C1-C5, C2-C6, A2-E6, F1-B5 and E2-A6. Since the redundant hyperedges can be removed from the position, no pairs are needed for these hyperedges. Given a position, unmarked vertices are assigned into disjoint pairs such that each hyperedge either contains at least one pair or is redundant. Not all unmarked vertices need to be part of a pair. Note that hyperedges can share the same pair in a pairing strategy. For example, the pair (C1, D1) (denoted as the pair 'a') is shared by the two hyperedges, A1-E1 and B1-F1, in Fig. 1(b). An algorithm and heuristics for finding pairing strategies are described in full in Section 5.

2.4. Potential weights

Beck in [4] introduced *potential weight functions* for solving Maker-Breaker positional games. A win for Breaker can be proven if the potential value of a game obtained through the potential weight functions is lower than some specific threshold. The potential weight function used in this paper for *mnk*-games is presented as follows. Given an *mnk*-game (V, E), let e be a hyperedge in E, and $M(e)$ be the number of vertices marked by Maker on e. The *potential weight* of e is given by the following potential weight function w.

$$w(e) = \begin{cases} 0, & \text{if there exists at least one Breaker's mark in } e. \\ 2^{M(e)}, & \text{otherwise.} \end{cases} \tag{1}$$

The *potential value* P of a game is defined as the summation of the weights of all hyperedges.

$$P = \sum_{e \in E} w(e). \tag{2}$$

Assume that Maker wins by marking all vertices on some hyperedge e. Then, $w(e) = 2^k$ and thus $P \geq 2^k$. Let $E(v) \subseteq E$ be the set of all hyperedges that contain the vertex $v \in V$. The potential weight of v is defined as

$$w(v) = \sum_{e \in E(v)} w(e). \tag{3}$$

Let P_t denote P right after the t-th move (indicating the specific time frame after the t-th move and before the $(t+1)$-st move). Namely, P_0 is the initial potential value, P_{2n-1} is the potential value right after Maker's n-th move, and P_{2n} is the potential value right after Breaker's n-th move for all positive integers n. We state the following property, as given by Beck [4], to analyze k-in-a-row games.

Property 1. *During the course of a Maker-Breaker k-in-a-row game, if $P_{2n-1} < 2^k$, then Breaker wins right after the $(2n-1)$-st move [4].*[1]

Property 1 allows an alternative to pairing strategies to prove that Breaker wins. For example, Fig. 1(a) can also be solved by Property 1 as follows. After Maker marks C3 in Fig. 1(a), the potential value P_1 is 16 in total, 8 for 8 hyperedges not passing C3, each with a weight of 1, and 8 for 4 hyperedges passing C3, each with a weight of 2. Similarly, we calculate the potential values for all other first moves by Maker, each of which yields a value less than 16. Thus, $P_1 < 2^5 = 32$ for all first moves by Maker, indicating that Breaker wins.

3. Improving AND/OR search in *mnk*-games

In this section, we present a new AND/OR search algorithm for proving Breaker's win in Maker-Breaker *mnk*-games based on pairing strategies as described in Subsection 2.3, potential weights as described in Subsection 2.4, and three techniques aimed at pruning the search tree, which we will describe in detail in Section 4. The AND/OR search algorithm features problem reduction, branch pruning, move ordering, and early return.

Algorithm 1 illustrates AND/OR search on a position (V, E).[2] Without loss of generality, let us assume first that all of the wins and losses are from Breaker's viewpoint. Therefore, positions that Breaker is going to move are OR-nodes and those Maker is going to move are AND-nodes. To prove Breaker's win, at least one move at an OR-node and all moves at an AND-node should be proven as Breaker's win. For simplicity of presenting algorithms, in the remainder of the paper, only *wins* (corresponding to the player to move) are considered.

First (lines 1-2), we remove some vertices and hyperedges that will not change the game outcome to reduce the game size (see Section 4). Second (lines 3-4), we generate a move list including all unmarked vertices on the position, and calculate the potential weight for each of these vertices as defined in Subsection 2.4. The move list is then sorted by potential weights. Vertices with higher potential weights will be prioritized. Third (lines 5-11), we check if there is an immediate win for the player to move in order to avoid unnecessary search. More specifically, when there exists a vertex such that a hyperedge is completed during Maker's turn or a pairing strategy is found in Breaker's turn, the winning player is returned immediately. An algorithm that identifies specific pairing strategies will be described in detail in Section 5. Fourth (lines 12-18), we iteratively mark a vertex in the move list. In each iteration, a recursive AND/OR search for the opponent is called. If the search results in the player's win, the result is returned. Lastly (line 19), after the move list is exhaustively checked, the opponent wins.

The four features in our AND/OR search algorithm are briefly mentioned below, and will be discussed in detail in the following sections.

Problem Reduction. *mnk*-games can be reduced by removing vertices that belong to partial pairing sets (Subsection 4.1) as well as redundant vertices and hyperedges (Subsection 2.3). This corresponds to lines 1-2 of Algorithm 1.

Branch Pruning. According to Theorem 2 (Subsection 4.2), a player does not need to mark dominated vertices or vertices that do not belong to the relevancy-zones (r-zones, described in Subsection 4.3). Thus, we can use these techniques to prune branches, which in turn speed up the search. This corresponds to line 13 of Algorithm 1.

Move Ordering. Although potential weights are originally used to prove Breaker's wins, we use them to order moves during search. The potential weight of each unmarked vertex is evaluated according to Formula (3) (Subsection 2.4), which forms the basis of how the unmarked vertices are sorted, in decreasing order. That is, the unmarked vertex with the highest potential weight is considered as the most promising one for the player to move. This corresponds to lines 3-4 of Algorithm 1.

Early Return. According to the definition of Maker-Breaker games, a Breaker's win can only be claimed after all vertices in the position are checked, which consumes a lot of search time. One trick is that if a pairing strategy is found during the search of an *mnk*-game, then Breaker can be considered to be winning. This is the same idea as the example illustrated in Fig. 1(b). We attempt to find pairing strategies in line 8 of Algorithm 1.

[1] The proof is omitted here, but readers are encouraged to read Beck's work [4] which contains the proof of the equivalent of Property 1 for positional games.

[2] Algorithm 1 gives a general concept for both Maker and Breaker. The implementation details are different for Maker and Breaker, each of which are described in Algorithms 4 and 5 respectively, in Appendix A.

Algorithm 1 AND/OR search.

```
procedure AND_OR_SEARCH ($V$, $E$)
    // Remove some vertices and hyperedges from consideration.
 1.   $(V_{ignored}, E_{ignored})$ ←Find the ignorable parts of the game $(V, E)$
 2.   $(V_{reduced}, E_{reduced}) \leftarrow (V - V_{ignored}, E - E_{ignored})$
    // Generate all legal moves and sort them.
 3.   $move\_list$ ← all unmarked vertices in $V_{reduced}$
 4.   Sort $move\_list$ by potential weights
    // Check the player's immediate win.
 5.   for each $v$ in $move\_list$ do
 6.     if the player is Maker and Maker completes a hyperedge after $v$ is marked then
 7.         return Maker's win
 8.     if the player is Breaker and Breaker has a pairing strategy after $v$ is marked then
 9.         return Breaker's win
10.     end for
11.   end if
    // Visit all moves and do recursive call. Some moves can be safely skipped.
12.   for each $v$ in $move\_list$ do
13.     if $v$ can be skipped then continue
14.     $(V', E') \leftarrow (V, E)$
15.     mark $v$ in $V'$ with player's color
16.     $result$ ← AND_OR_SEARCH ($V'$, $E'$)
17.     if $result$ is the player's win then return the player's win
18.   end for
    // If there is no winning move for the player, return the opponent's win.
19.   return the opponent's win.
end procedure
```

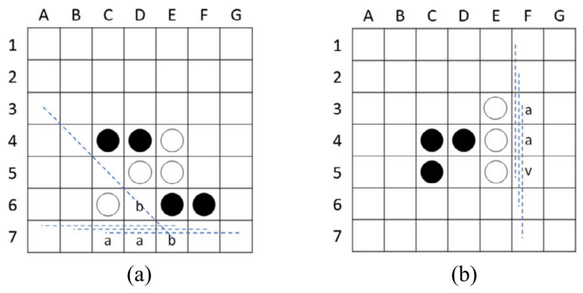


Fig. 2. Examples of partial pairing in the 7,7,5-game.

4. Reduction techniques

In this section, we propose three techniques used to reduce the search space and speed up the proof, including partial pairing in Subsection 4.1, vertex domination in Subsection 4.2, and the Breaker r-zone in Subsection 4.3. All three techniques are used to prune the AND/OR search algorithm in Section 3.

4.1. Partial pairing

We propose the partial pairing method to efficiently remove vertices and hyperedges from consideration in a Maker-Breaker game (V, E) without changing its theoretical value. For simplicity of discussion, we assume that redundant hyperedges are already removed from E.

A set of unmarked vertices $V_{pair} \subseteq V$ is a *partial pairing set* if some vertices in V_{pair} can be assigned into disjoint pairs, called *partial pairs*, such that all hyperedges containing a vertex in V_{pair} contains at least one partial pair. Note that there may be vertices in V_{pair} not assigned to any partial pairs. Let E_{pair} be the set of all the hyperedges each of which contains at least one partial pair composed by V_{pair}. For example, in Fig. 2(a), V_{pair} contains the vertices C7, D7, D6 and E7, and the corresponding E_{pair} is the set of the four hyperedges depicted by dotted lines.

Theorem 1. *Given a position (V, E), for a partial pairing set V_{pair} and the corresponding E_{pair}, the theoretical value of (V, E) is the same as that of the reduced position $\left(V - V_{pair}, E - E_{pair}\right)$.*

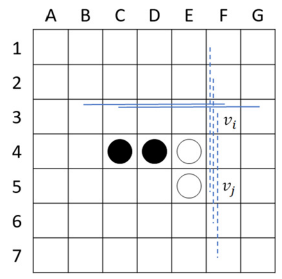


Fig. 3. Vertex F3 dominates F5 in the 7,7,5-game.

Proof. Assume that Maker has a winning strategy to win $\left(V - V_{pair}, E - E_{pair}\right)$. Then, it is quite straightforward that Maker wins in (V, E) by following the winning strategy, since the Breaker cannot mark vertices in V_{pair} to stop Maker from completing a hyperedge in $E - E_{pair}$.

Conversely, assume that Breaker has a winning strategy to win $\left(V - V_{pair}, E - E_{pair}\right)$. Then, there exists a strategy s for Breaker to block all hyperedges in $E - E_{pair}$. Breaker can win (V, E) by blocking all hyperedges in E_{pair} as follows. In the case that Maker marks one vertex in $V - V_{pair}$, then Breaker follows strategy s to block hyperedges in $E - E_{pair}$. In the case that Maker marks one vertex in a partial pair, then Breaker marks the other, which ensures all hyperedges in E_{pair} to be blocked. In the case that Maker marks one vertex in V_{pair}, but not in any partial pairs, Breaker can simply mark an arbitrary vertex since it has been ensured that all hyperedges can be blocked by the above two cases. □

For a vertex v that is not contained in any hyperedge, we can simply remove it for the following reason. Since there is no hyperedge containing v, the set $\{v\}$ can be viewed as a partial pairing set V_{pair} without assigning any partial pairs and the corresponding $E_{pair} = \emptyset$. Thus, $(\{v\}, \emptyset)$ can be removed from the position according to Theorem 1. Such a vertex is said to be *redundant*. Illustrated by Fig. 2(b), let $V_{pair} = \{F3, F4\}$ and the corresponding E_{pair} are depicted by the three dotted lines (F1-F5, F2-F6, and F3-F7). After removing (V_{pair}, E_{pair}), the vertex F5 can be removed since no hyperedge contains it.

4.2. Vertex domination

We propose a method to safely ignore some vertices for the next move without changing the theoretical value of a position (V, E). Let v_i and v_j be unmarked vertices. We say v_i *dominates* v_j if $E(v_i) \supseteq E\left(v_j\right)$ holds, where $E(v) \subseteq E$ is the set of all hyperedges containing vertex v. We call v_i a *dominating vertex* and v_j a *dominated vertex*. In Fig. 3, the vertex F3 dominates F5 since E (F3), i.e. the set of hyperedges depicted by both the solid and dotted lines, contains E (F5), the set of hyperedges depicted by the dotted lines. Intuitively, marking F3 is better than marking F5 for both Maker and Breaker. In this example, from Theorem 2 (see below), if a player can win by marking F5, then he/she can also win by marking F3. Therefore, we only need to consider whether marking F3 results in a win.

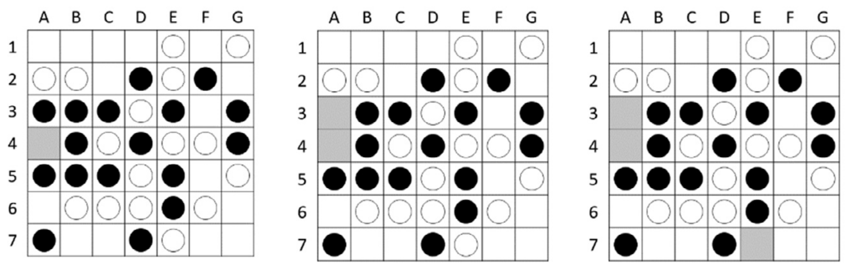
Theorem 2. *Assume that vertex $v_i \in V$ dominates $v_j \in V$ in a position (V, E). If a player wins by marking v_j as the next move, then he/she can win by marking v_i as the next move as well.*

Proof. We prove the equivalent statement, "If a player loses by marking v_i as the next move, then he/she also loses by marking v_j as the next move."

Assume that Maker has a winning strategy s_i to win after Breaker marks the dominating vertex v_i. If Breaker chooses to mark the dominated v_j instead, Maker can find a winning strategy s_j by first copying s_i, but replacing all instances of v_j to be v_i and vice-versa in this new strategy s_j. That is, wherever v_j is marked by Maker in s_i, Maker will mark v_i instead when following s_j; wherever v_i is played by Breaker in s_j, Maker can refer to his/her response for v_j in s_i and mark that instead. From the definition of domination, when Maker reaches a winning position in s_i, Maker also wins in s_j as follows. Suppose Maker completed $e_{winning}$ in s_i. Since the dominating vertex v_i was marked by Breaker in s_i, $e_{winning}$ does not contain v_i. By definition of vertex domination, $e_{winning}$ does not contain v_j neither. Therefore, all the vertices on $e_{winning}$ are also marked by Maker in s_j, which means Maker wins the position.

Similarly, assume that Breaker has a winning strategy s_i to win after Maker marks the dominating vertex v_i. If Maker chooses to mark the dominated v_j instead, Breaker can find a winning strategy s_j by similarly copying s_i, then replacing all instances of v_j to be v_i and vice versa. When Breaker reaches a winning position in s_i, Breaker also win in s_j as follows. In the case that v_j was marked by Breaker in s_i, v_i will be marked by Breaker following s_j. Since v_i dominates v_j, all hyperedges blocked by v_j in s_i will also be blocked by v_i in s_j. All other hyperedges not involving v_i or v_j will be blocked the same way as s_i when following s_j. □

Theorem 2 implies that a player can ignore dominated vertices for the next move. A special case exists when vertex v_i dominates vertex v_j at the same time when v_j also dominates v_i in (V, E). In this case, the two vertices are said



(a) Breaker's turn. Breaker wins after marking A4.

(b) Maker's turn. Breaker wins after responding on A3 or A4.

(c) Breaker's turn. Breaker wins.

Fig. 4. Three examples of Breaker r-zones for the 7,7,5-game. The Breaker r-zone consists of gray squares.

to be *mutually dominated*. Two mutually dominated vertices form a partial pair. Thus, the vertices and the corresponding hyperedges can then be removed according to Theorem 1. For example, in Fig. 2(b), the vertices F3, F4 and F5 are mutually dominated, since $E\,(\text{F3})$, $E\,(\text{F4})$ and $E\,(\text{F5})$ are the same, as depicted by the dotted lines.

4.3. Breaker R-zone

We now present the concept of a relevancy-zone, or r-zone [20][25][27], and how the r-zone can be used with pairing strategies, partial pairing, and vertex domination to reduce the branching factor of search trees when proving Breaker victories.

A. Definition of Breaker R-zone

Given a game (V, E), an r-zone is a calculated area in which one must play in order to not lose within the next move. Once an r-zone is calculated, we can effectively remove all vertices outside of the r-zone from consideration and therefore reduce the branching factor. Since the r-zone was proposed for proving first player wins in strong positional games, we can say that previous efforts [25][27] have been formalized for Maker only. We present here a variation of the r-zone in the context of the Maker-Breaker game, and more specifically for Breaker, which we will refer to as the Breaker r-zone in the remainder of this paper. If there exists a solution tree [16][19] for Breaker to ensure that all hyperedges in all leaf nodes are blocked, we say that there exists a winning strategy for Breaker. In contrast to a Maker r-zone, the Breaker r-zone is defined by Breaker's winning strategy as follows.

Definition. Assume that Breaker has a winning strategy s. The *Breaker r-zone* R of s is defined as

$$R\,(s) = \{v\,|\,\text{vertex } v \text{ that might be marked by Breaker in } s\}.$$

For example, to win in Fig. 4(a), a trivial strategy s_1 for Breaker is to mark A4 so that all hyperedges are blocked. Since A4 is the only vertex Breaker marks in s_1, $R\,(s_1) = \{\text{A4}\}$. Fig. 4(b) is an example for the Breaker r-zone when it is Maker's turn. To win the position, Breaker's strategy should be able to counter Maker's every move choice. A winning strategy s_2 for Breaker is described as follows. If Maker marks A3, Breaker wins by responding on A4; otherwise, Breaker wins by responding on A3. Since A3 and A4 will be marked by Breaker in s_2, $R\,(s_2) = \{\text{A3}, \text{A4}\}$. Fig. 4(c) is the position exactly one move before Fig. 4(b). One winning strategy s_3 for Breaker in Fig. 4(c) is to mark E7 and then block the remaining hyperedges by marking either A3 or A4. In this case, the Breaker r-zone $R(s_3)$ is {A3, A4, E7}. Note that by choosing different winning strategies, there may be different Breaker r-zones that lead to Breaker winning for the same position. Another winning strategy for Breaker in Fig. 4(c) is to mark C7 and then block the remaining hyperedges by marking either A4 or A6. Then the corresponding Breaker r-zone is {C7, A4, A6}.

We present the following properties for the Breaker r-zone. Let us consider different situations from Maker's perspective, i.e. what outcomes will occur if Maker plays outside of the r-zone.

Property 2. *Given a position p where it is Maker's turn to move, and that Breaker wins by strategys after Maker marks a vertex v on p, Breaker can still win following the same strategy if Maker chooses to mark another vertex $v' \notin R(s)$ on p instead.*

Intuitively, $R(s)$ can be viewed as the most critical vertices that must be considered by both players. Property 2 stands since Maker plays outside of the Breaker r-zone, so Breaker is still free to follow s, thus ensuring all hyperedges will be blocked as the game proceeds.

Property 2 reduces the search branches when we prove Breaker's win since after we find a winning strategy for one move, Maker's other moves (unmarked vertices) not in the Breaker r-zone also result in Breaker's win. Hence, we do not

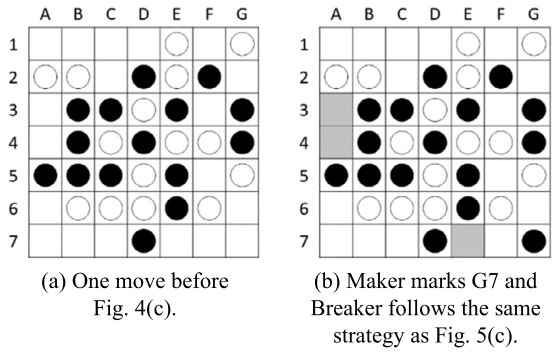

(a) One move before Fig. 4(c).
(b) Maker marks G7 and Breaker follows the same strategy as Fig. 5(c).

Fig. 5. Examples for Property 2.

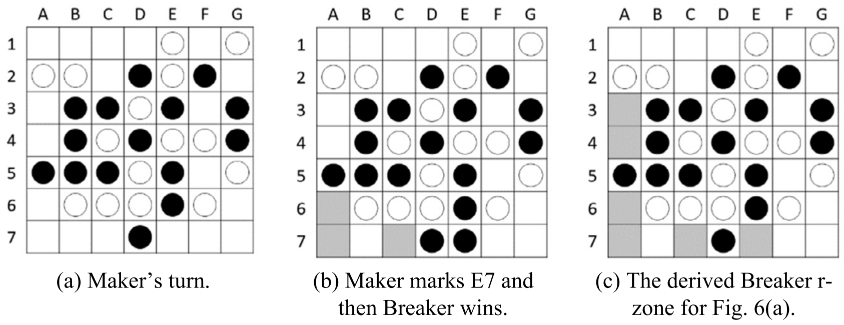

(a) Maker's turn.
(b) Maker marks E7 and then Breaker wins.
(c) The derived Breaker r-zone for Fig. 6(a).

Fig. 6. An example for Property 3.

need to investigate these moves and can cut them off. For example, Breaker wins Fig. 4(c) with a strategy s_3 where the Breaker r-zone $R(s_3)$ is {A3, A4, E7}. When backtracking to the position one move before Fig. 4(c), as shown in Fig. 5(a), we do not consider Maker marking any vertex $v' \notin \{A3, A4, E7\}$, since if Maker marks any vertex $v' \notin \{A3, A4, E7\}$, say G7 in Fig. 5(b), Breaker can win following the same strategy s_3.

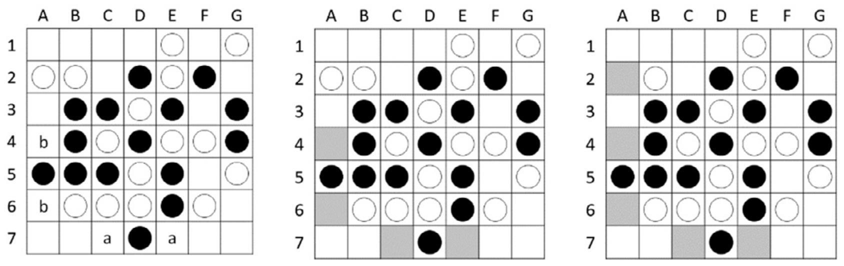
Property 2 allows us to prune moves when one Breaker r-zone is found. On the other hand, the following Property 3 will allow us to terminate search and propagate a Breaker r-zone upward when several Breaker r-zones are found.

Property 3. *Given a position p where it is Maker's turn to move. Assume that there are u unmarked vertices on p, and Breaker can win by strategys_i after Maker marks v_i on p for $i = 1, 2, \ldots, t$ $(t \le u)$. If $\cap_{i=1}^{i=t} R(s_i) = \emptyset$, then there exists a winning strategy s for Breaker such that $R(s) = \cup_{i=1}^{i=t} R(s_i)$.*

Proof. For any unmarked vertex $v_j \notin \{v_1, v_2, \ldots, v_t\}$ on p, we have $v_j \notin \cap_{i=1}^{i=t} R(s_i)$, since $\cap_{i=1}^{i=t} R(s_i) = \emptyset$. That is, $v_j \notin R(s_k)$ for some $k \in \{1, 2, \ldots, t\}$. From Property 2, if Maker marks v_j, then Breaker wins by following s_k. Therefore, Breaker must have s_j for v_j such that $R(s_j) = R(s_k) \subseteq \cup_{i=1}^{i=t} R(s_i)$. By definition, $R(s) = \cup_{i=1}^{i=u} R(s_i)$. Since $R(s_j) \subseteq \cup_{i=1}^{i=t} R(s_i)$ for $t < j \le u$, Breaker can win p with$s$ such that $R(s) = \cup_{i=1}^{i=u} R(s_i) = \cup_{i=1}^{i=t} R(s_i)$. Thus, Property 3 is proven. $\square$

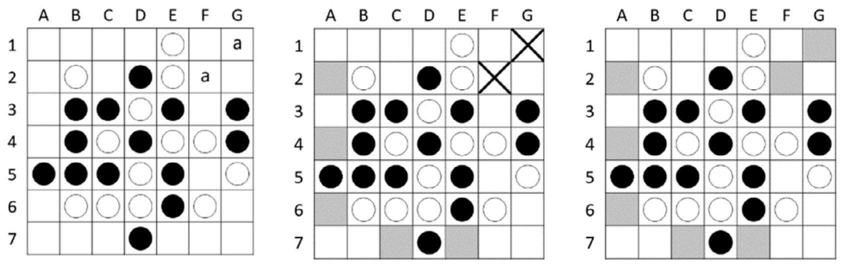
Fig. 6 is an example for Property 3. To prove Breaker wins Fig. 6(a) by AND/OR search, we need to investigate each move for Maker. However, Property 3 allows us to solve Fig. 6(a) without explicitly investigating all legal moves. Instead, only two moves are investigated as follows. First, we consider Maker marks A7 in Fig. 6(a) and reaches the position in Fig. 4(c), where the strategy is s_3 and the Breaker r-zone is $R(s_3) = \{A3, A4, E7\}$. Second, we consider that Maker marks E7 in Fig. 6(a) and reaches the position in Fig. 6(b). Breaker has a winning strategy s_4, to mark C7 and then block the remaining hyperedges by marking either A6 or A7, where $R(s_4) = \{A6, A7, C7\}$. Since $R(s_3) \cap R(s_4) = \emptyset$, no matter which move Maker takes in Fig. 6(a), Breaker wins by following either s_3 or s_4. The Breaker r-zone for Fig. 6(a) is then derived as $R(s_3) \cup R(s_4) = \{A3, A4, E7, A6, A7, C7\}$, as shown in Fig. 6(c).

Now that we have the definition and basic properties of the Breaker r-zone, we can use it to reduce the search space of the Maker-Breaker game. In the following subsections, we propose methods of deriving Breaker r-zones based on pairing strategies, partial pairing, and vertex domination.



(a) Maker's turn. Breaker wins since there exists a pairing strategy. (b) The Breaker r-zone derived from the pairing strategy. (c) One move before Fig. 7(a). The Breaker r-zone is derived from Fig. 7(b).

Fig. 7. An example for deriving Breaker r-zones from a pairing strategy.



(a) Breaker's turn. There exist a partial pairing set. (b) The position reduced from Fig. 8(a). (c) The Breaker R-zone derived by Theorem 4.

Fig. 8. An example for deriving a Breaker r-zone based on partial pairing.

B. Breaker R-zone and pairing strategies

In Subsection 2.3, pairing strategies are used to prove winning positions for Breaker. In this subsection, we present a method to derive Breaker r-zones based on pairing strategies.

Theorem 3. *If Breaker wins a position with a pairing strategy s, the Breaker r-zone of s is* $R(s) = \{v \mid v \text{ is an unmarked vertex assigned to a pair in } s\}$.

Proof. Since s is a winning strategy, Breaker only needs to mark on vertices assigned to a pair in s as explained as follows. In the case that Maker marks some vertex in a pair, Breaker responds by marking the other in the same pair. In all other cases, Breaker can simply mark any unmarked vertex in s, which ensures that each pair of vertices must contain at least one Breaker's mark so that all hyperedges are blocked. □

For example, Fig. 6(a) can also be solved by the paring strategy as shown in Fig. 7(a). Breaker wins by assigning A4 and A6 as a pair and assigns C7 and E7 as the other pair. According to Theorem 3, we derive the Breaker r-zone $\{A4, A6, C7, E7\}$ without investigating the succeeding positions. Theorem 3 allows us to propagate pruning upward in the search tree when using pairing strategies as a terminal condition. For example, a Breaker r-zone for the position in Fig. 7(c) can therefore be derived from Fig. 7(b).

C. Breaker R-zone and partial pairing

In Subsection 4.1, partial pairing is used to reduce the game size while maintaining the same theoretical value. In this subsection, we propose Theorem 4 to derive Breaker r-zones from reduced games, which will in turn allow us to propagate pruning upwards for the corresponding unreduced games.

Theorem 4. *For a partial pairing set* V_{pair} *in* (V, E) *and the corresponding* E_{pair}*, if Breaker has a winning strategy* $s_{reduced}$ *for the reduced game* $\left(V - V_{pair}, E - E_{pair}\right)$*, then Breaker has a winning strategy s for* (V, E) *such that* $R(s) = R(s_{reduced}) \cup V_{pair}$.

Proof. From the proof of Theorem 1, Breaker can construct a winning strategy s by following $s_{reduced}$ and also responding according to partial pairings in V_{pair}. Therefore, by definition, $R(s) = R(s_{reduced}) \cup V_{pair}$. □

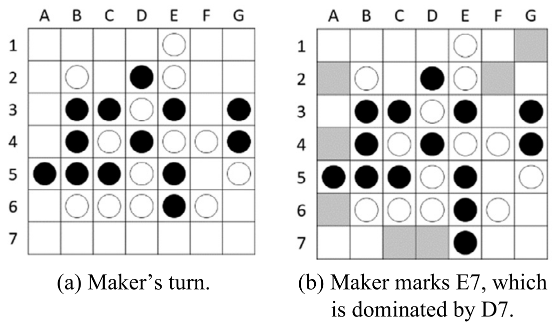


Fig. 9. An example for deriving a Breaker r-zone from dominated moves.

For example, in the game (V, E) shown in Fig. 8(a), there exists a partial pairing set $V_{pair} = \{\text{F2, G1}\}$ and $E_{pair} = \{\text{C5-G1}\}$. The reduced game $(V - V_{pair}, E - E_{pair})$ is shown in Fig. 8. For the purpose of proving Breaker's win, removing the hyperedge C5-G1 in Fig. 8(b) has the same effect as Breaker's mark on G1 blocking C5-G1 in Fig. 7(c). Therefore, the position in Fig. 8(b) can be solved by the same strategy used to solve the position in Fig. 7(c), from which we then obtain $R(s_{reduced}) = \{\text{A2, A4, A6, C7, E7}\}$. According to Theorem 4, we derive that Breaker wins the position in Fig. 8(a) with $R(s_5) = R(s_{reduced}) \cup V_{pair} = \{\text{A2, A4, A6, C7, E7, F2, G1}\}$ as shown in Fig. 8(c).

D. Breaker R-zone and vertex domination

In Subsection 4.2, vertex domination is used to show how moves can supersede each other. In this subsection, we discuss deriving Breaker r-zones when considering vertex domination. Note that normally, as new game states are visited during the search, Breaker r-zone components can be propagated upwards to form larger r-zones. The following theorem allows us to derive Breaker r-zones from dominated moves without having to visit those dominated branches.

Theorem 5. *Given a position (V, E) where it is Maker's turn to move, assume that vertex $v_i \in V$ dominates $v_j \in V$. If Breaker wins by strategy s_i after Maker marks v_i, then Breaker can win by strategy s_j after Maker marks v_j, where $R(s_j)$ is derived by the following formula.*

$$R(s_j) = \begin{cases} R(s_i) \cup \{v_i\} - \{v_j\}, & \text{if } v_j \in R(s_i), \\ R(s_i), & \text{if } v_j \notin R(s_i). \end{cases}$$

Proof. From the proof of Theorem 2, Breaker can construct the winning strategy s_j by copying s_i, then replacing all instances of v_i and v_j. Thus, we obtain $R(s_j) = R(s_i) \cup \{v_i\} - \{v_j\}$ for the case that $v_j \in R(s_i)$. In the case that $v_j \notin R(s_i)$, since v_j is outside the r-zone, Breaker can win by following s_i. Thus, we obtain $R(s_j) = R(s_i)$ for the case that $v_j \notin R(s_i)$. □

We can use Theorem 5 to find the Breaker r-zone for a position where a dominated vertex is marked. For example, Fig. 9(a) is the position that is exactly one move before the position in Fig. 8(c) and Fig. 9(b). The vertex D7 dominates E7 and G7 in Fig. 9(a). If Maker marks D7, the game reaches the position in Fig. 8(c), and Breaker wins with $R(s_5) = \{\text{A2, A4, A6, C7, E7, F2, G1}\}$. According to Theorem 5, we can derive $R(s_6)$ for E7 and $R(s_7)$ for G7 as follows. Since $\text{E7} \in R(s_5)$, we obtain $R(s_6) = R(s_5) \cup \{\text{D7}\} - \{\text{E7}\} = \{\text{A2, A4, A6, C 7, D7, F2, G1}\}$, as shown in Fig. 9(b). Since $\text{G7} \notin R(s_5)$, we obtain $R(s_7) = R(s_5)$.

5. Techniques for finding pairing strategies

In Section 3 and Subsection 4.3.B, we describe how the existence of a pairing strategy for a given position indicates that Breaker has won. In this section, we propose an algorithm (see Algorithm 2) to identify pairing strategies, if any, for a given position (V, E), following the description in Subsection 2.3. The algorithm returns the set of pairs when a pairing strategy is successfully found, or an empty set otherwise. During the pair-assignment process, we focus on unmarked vertices that are still not assigned into pairs and hyperedges that do not contain any pair yet. These vertices and hyperedges are called *unassigned vertices* and *free hyperedges*, respectively, for simplicity of discussion. First, all unmarked vertices are unassigned vertices, and all hyperedges that are not yet blocked are free hyperedges. Second, we call a recursive function to assign vertices into pairs (see Algorithm 3). After the recursive function returns, Algorithm 2 returns the result, which consists of the search's *success* or *failure*, and the set of pairs that form the pairing strategy.

Algorithm 2 Find a pairing strategy.

```
procedure FIND_A _PAIRING_STRATEGY($V,\ E$)
 1.  $V_{unassigned} \leftarrow \{v \mid$ unmarked vertex $v \in V\}$
 2.  $E_{free} \leftarrow \{e \mid e \in E$ and $e$ is not blocked$\}$
 3.  $(result,\ \mathcal{P}) \leftarrow$ RECURSIVELY_ASSIGN_PAIRS($V_{unassigned}, E_{free}$)
 4.  return $(result, \mathcal{P})$
end procedure
```

Algorithm 3 Assign pairs.

```
procedure RECURSIVELY_ASSIGN_PAIRS($V_{unassigned}, E_{free}$)
 1.  $(V_{unassigned}, E_{free},\ \mathcal{P}_{greedy}) \leftarrow$ GREEDY_ASSIGN_PAIRS($V_{unassigned}, E_{free}$)
     // Terminal conditions
 2.  if $E_{free}$ is empty set then return (success, $\mathcal{P}_{greedy}$)
 3.  for each $e \in E_{free}$ do
 4.      if $e$ contains less than two vertices $\in V_{unassigned}$ then return (fail, $\emptyset$)
 5.  end for
     // Recursive call
 6.  $e_{highest} \leftarrow$ the hyperedge in $E_{free}$ with the highest hyperedge score
 7.  loop at most $N$ times:
 8.      $(v_i,\ v_j) \leftarrow$ two unassigned vertices on $e_{highest}$ with the highest pair score
 9.      $V_{recur} \leftarrow V_{free} - \{v_i, v_j\}$
10.      $E_{recur} \leftarrow E_{free} - (E_{free}(v_i) \cap E_{free}(v_j))$
11.      $(result,\ \mathcal{P}_{recur}) \leftarrow$ RECURSIVELY_ASSIGN_PAIRS($V_{recur}, E_{recur}$)
12.      if $result$ is success then return (success, $\mathcal{P}_{greedy} \cup \{(v_1, v_2)\} \cup \mathcal{P}_{recur}$)
13.      else forbid choosing $(v_i,\ v_j)$ on $e_{highest}$ for the following iterations
14.  end loop
15.  return (fail, $\emptyset$)
end procedure
```

Algorithm 3 is a recursive function to find a pair for each free hyperedge by assigning unassigned vertices into pairs. First, we assign new pairs using two greedy rules below.[3] Second, two terminal conditions are checked as follows. If each of all the free hyperedges contain a pair, then the assignment is finished, and the function returns success and the set of pairs, $\mathcal{P}_{greedy}$. If there exists a free hyperedge containing less than two unassigned vertices, then the function returns fail and an empty set. Finally, we assign the remaining vertices into pairs by recursive calls and backtracking.

Let $E_{free}(v)$ denote the set of free hyperedges that contain the vertex v. The following two greedy rules provide the best assignments for current unassigned vertices and free hyperedges.

Greedy rule 1. *If a free hyperedge contains exactly two unassigned vertices, assign them into a pair.*

This is the only pair assignment for the hyperedge such that it contains pairs.

Greedy rule 2. *For unmarked vertices v_i and v_j if $E_{free}(v_i) = E_{free}(v_j)$, assign the two vertices into a pair.*

This rule gives the best assignment for finding a pairing strategy. In other words, if this assignment fails to find a valid pairing strategy, no other assignment will succeed. We illustrate this by showing the equivalent statement, "if a pairing strategy is found by assigning v_i and v_j into separate pairs, we can also guarantee that a pairing strategy can be found where v_i and v_j belong in the same pair." For example, E_{free} (A2) = E_{free} (A3) in Fig. 10(a), where E_{free} (A2) and E_{free} (A3) are represented by the solid lines (A1-A5, A2-A6). A pairing strategy can be found, shown in Fig. 10(b), where vertices A1 and A2 are assigned as a pair (labeled as 'x'), and vertices A3 and A6 are assigned as another pair (labeled as 'y'). Since E_{free} (A2) = E_{free}(A3), we can alternatively assign A2 and A3 as a pair, shown in Fig. 10 (c) as 'f'. The assignments for A1 and A6 can be removed. Consequently, the two free hyperedges, A1-A5 and A2-A6, still contain a pair, comprising a valid pairing strategy.

We design two heuristics, called the *hyperedge score* and the *pair score*, to find the most promising assignment. A hyperedge score for a hyperedge e is the difference between k (in *mnk*-games) and the number of unassigned vertices in e. A pair score for a pair of vertices, v_1 and v_2, is $\left|E_{free}(v_1) \cap E_{free}(v_2)\right| \times 2 - \left|(E_{free}(v_1) \cup E_{free}(v_2)) - (E_{free}(v_1) \cap E_{free}(v_2))\right|$. The pair score aims to maximize the number of free hyperedges that will contain a pair after the assignment, while minimizing the number of free hyperedges that pass either v_1 or v_2.

For the recursive call, we find the hyperedge $e_{highest}$ with the highest hyperedge score, and repeat the following process. We assign two unassigned vertices, v_i and v_j, on $e_{highest}$ with the highest pair score as a pair. We treat the remaining unassigned vertices and free hyperedges as a sub-problem, and find a pairing strategy for them by recursively calling

[3] The implementation details are shown in Algorithm 9 in Appendix C.

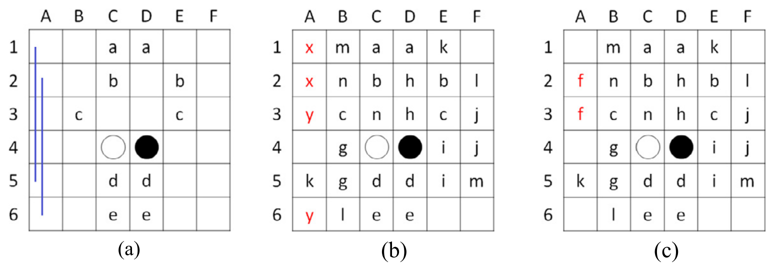


Fig. 10. An example for Greedy rule 2.

Table 1
The performance of combinations of the three pruning methods. The symbol "×" denotes that the method is used.

Partial Pairing	**×**	×	×	×				
Vertex Domination	**×**		×		×	×		
Breaker r-zone	**×**	×			×		×	
Time (second)	7457.3	31441.1	33855.6	>72 h	>72 h	>72 h	>72 h	>72 h

Table 2
The time to solve the 7,7,5-game with different settings of N.

N	0	1	**2**	3	4	5
Time (second)	1870.0	4.1	**2.5**	9.4	16.7	29.0

Algorithm 3. If the recursive call finds a pairing strategy, Algorithm 3 returns success and the pairing strategy as well as the pairs assigned so far. Otherwise, we use backtracking and assign another pair with the next highest score. The above process repeats at most C_2^b times, where b is the number of unassigned vertices on $e_{highest}$. If no pairing strategy is found, then Algorithm 3 returns fail and an empty set. In practice, we repeat at most N times ($N \leq C_2^k$), where N is a *hyper-parameter* that is properly chosen to obtain a balance between time cost and the chance to find a pairing strategy.

6. Experiment results

To solve the 7,7,5-game and the 8,8,5-game, we design a program that implements the AND/OR tree search described in Section 3. Our program stores repeated positions in a transposition table. In Subsection 6.1, we test the speedups of solving the 7,7,5-game by using the three reduction techniques for pruning described in Section 4. In Subsection 6.2, we solve the 8,8,5-game by additionally computing pairing strategies following the algorithms described in Section 5. The experiments were performed with one thread on the machine equipped with Intel(R) Xeon(R) Gold 6154 CPU @ 3.00 GHz.

6.1. Pruning methods

In this subsection, we apply the three pruning methods: partial pairing, vertex domination, and Breaker r-zones, to solve the 7,7,5-game, and compare their speedups. Note that we do not compute pairing strategies as terminal conditions of the search tree in these experiments. Table 1 shows the time to solve the 7,7,5-game using different combinations of the pruning methods. For cases where the 7,7,5-game is not solved within 72 hours, the program is terminated manually.

Chen et al. solved the 7,7,5-game as a draw in 5 days using 10 threads on a machine equipped with CPU @ 3.10 GHz [8]. Our experiment results show that the three pruning methods dramatically speed up the proof for the 7,7,5-game. The game cannot be solved within 72 hours when any one method alone is applied. With two pruning methods, the game can be solved when partial pairing is combined with Breaker r-zones or vertex domination, with running time of 8.73 hours (31441.1 seconds) and 9.40 hours (33855.6 seconds), respectively. When all methods are used, the solving time is 2.07 hours (7457.3 seconds), reaching a time reduction of a factor of about four compared with the cases in which only two methods are used.

6.2. Solving with pairing strategies

The 8,8,5-game is more complex than the 7,7,5-game, and had been left unsolved in past literature. Under our experiment setup, it is not possible to solve the 8,8,5-game within 72 hours even when combining all of three pruning methods. In this subsection, we compute pairing strategies and use them as terminal conditions to speed up the tree search, leading to the successful solution of the 8,8,5-game as a draw. To the best of our knowledge, this paper is the first to solve the 8,8,5-game.

Table 3
The time to solve the 8,8,5-game with different settings of N.

N	1	**2**	3
Time (second)	>72 h	**62634.3**	>72 h

The algorithms for finding a pairing strategy are given in Section 5. First, we tune the hyper-parameter N in Algorithm 3 for the 7,7,5-game. Table 2 shows the solving time of the 7,7,5-game with different settings of N.

As shown in Table 1, the solving time is 7457 seconds when pairing strategies are not used. When only the greedy rules are used, that is, $N = 0$, the solving time is 31.17 minutes (1870.0 seconds), reaching about a 4-fold speedup. Next, we also use the two heuristics in the recursive call portion of Algorithm 3. We test N between 1 and 5, where the best setting is $N = 2$ with a solving time of 2.5 seconds, reaching a speedup of 748 compared with $N = 0$.

Second, we use the best setting of $N = 2$ in the 7,7,5-game to successfully solve the 8,8,5-game in 17.40 hours (62634.3 seconds). Other settings of $N = 1$ and $N = 3$ did not lead to solving the game within 72 hours, as shown in Table 3.

7. Conclusion

To the best of our knowledge, this paper is the first to solve the 8,8,5-game as a draw. In this paper, we present an improvement on the AND/OR tree algorithm specifically for proving *mnk*-games. We propose three reduction techniques and an algorithm for finding pairing strategies. We also use potential weights to order moves in the search to speed up the proof. The program implementing these methods proves a draw for the 7,7,5-game in about 2.5 seconds and prove a draw for the 8,8,5-game within 17.4 hours. We expect that it is likely to challenge *mnk*-games with larger dimensions based on the methods described in this paper, together with some other methods, like proof-number search or depth-first proof-number search [15].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to thank the Ministry of Science and Technology of Taiwan for financial support of this research under contract numbers MOST 108-2634-F-009-011, 109-2634-F-009-019, 108-2634-F-259-001, 109-2634-F-259-001 (through Pervasive Artificial Intelligence Research (PAIR) Labs, Taiwan), 108-2221-E-305-008-MY3 and 108-2221-E-003-011-MY3, and National Taipei University under the contract number 109-NTPU_ORDA-F-002. We are grateful to the National Center for High-performance Computing for computer time and facilities.

Appendix A

In this Appendix, we describe the implementation details of the AND/OR tree. In Subsection 4.3, Breaker r-zone is propagated upward during the search to prune branches. We realize this procedure by making the AND/OR search function return both the game result and the Breaker r-zone when the outcome of a position is found. The Breaker r-zone can be any container that properly represents a set of vertices. Note that the Breaker r-zone is an empty set when Maker wins.

Algorithm 4 illustrates the AND/OR search on a position (V, E) where it is Maker's turn to move. First, we find a partial pairing set (see Algorithm 7 in Appendix B) and any redundant vertices and hyperedges (see Algorithm 6) to reduce the game size. Second, we generate a move list in which dominated vertices are removed, then sort the list by potential weight. Vertices with higher potential weights are searched earlier. Third, we check the move list. If there exists a vertex marked by Maker such that a hyperedge is complete, Maker's win is returned immediately. Fourth, we iteratively mark a vertex in the move list. In each iteration, a recursive AND/OR search for Breaker is called (see Algorithm 5). If the search results in Maker's win, the result and an empty set is returned. Otherwise, we assume a Breaker r-zone $R_{subtree}$ is generated by the recursive call. The Breaker r-zone is augmented with $R_{subtree}$ and the vertices that need to checked is reduced by $R_{subtree}$ according to Property 2. After the entire move list is checked without a winning result for Maker, the result is Breaker's win. Fifth, we augment the Breaker r-zone with the dominating vertices and partial pairs according to Theorem 5 and Theorem 4 respectively. Breaker's win and the augmented Breaker r-zone are returned.

Algorithm 5 illustrates AND/OR search on a position (V, E) where it is Breaker's turn to move. Since Maker's mark does not generate any redundant vertices nor redundant hyperedges, we do not reduce the game before Breaker's move. We directly generate a sorted move list in which dominated vertices are removed, and check if there is an immediate win for Breaker. If there exists a vertex marked by Breaker such that a pairing strategy is detected (see Algorithm 2), the Breaker r-zone is derived according to Theorem 3. Then, we iteratively mark a vertex in the move list. In each iteration, a recursive AND/OR search for Maker is called (see Algorithm 4). If the result is Breaker's win, the result and the Breaker r-zone is returned. After the entire move list is exhausted, the result of Maker's win and an empty set are returned.

Algorithm 4 AND/OR search (Maker's part).

```
procedure MAKER_SEARCH($V$, $E$)
        // Remove some vertices and hyperedges from consideration.
  1.    $(V_{pair}, E_{pair}) \leftarrow$ FIND_PARTIAL_PAIRING_SET($V$, $E$)
  2.    $(V, E) \leftarrow (V - V_{pair}, E - E_{pair})$
  3.    $(V_{redundant}, E_{redundant}) \leftarrow$ FIND_REDUNDANCIES($V$, $E$)
  4.    $(V, E) \leftarrow (V - V_{redundant}, E - E_{redundant})$
        // Generate all legal moves and sort them.
  5.    move_list ← all unmarked vertices in $V$
  6.    dominated_list ← all dominated vertices in move_list          // see Subsection 4.2
  7.    remove vertices in dominated_list from move_list
  8.    Sort move_list by potential weights                           // see Subsection 2.4
        // Check the player's immediate win.
  9.    for each $v$ in move_list do
 10.      if Maker completes a hyperedge after $v$ is marked then
 11.        return (Maker_Win, $\emptyset$)
 12.    end for
        // Visit all moves and do recursive call. Some moves can be safely skipped.
 13.    $(R, V_{tocheck}) \leftarrow (\emptyset, V)$
 14.    for each $v$ in move_list do
 15.      if $v \notin V_{tocheck}$ then continue
 16.      $(V', E') \leftarrow (V, E)$
 17.      mark $v$ in $V'$ with Maker's color
 18.      $(result, R_{subtree}) \leftarrow$ BREAKER_SEARCH($V'$, $E'$)
 19.      if result is Maker_Win then return (Maker_$Win$, $\emptyset$)
 20.      $(R, V_{tocheck}) \leftarrow (R \cup R_{subtree}, V_{tocheck} \cap R_{subtree})$
 21.    end for
 22.    for each $v$ in dominated_list do                              // Theorem 5
 23.      if $v \notin V_{tocheck}$ then continue
 24.      $v_{dominating} \leftarrow$ the vertex that dominates $v$
 25.      $R \leftarrow R \cup \{v_{dominating}\}$
 26.    end for
 27.    $R \leftarrow R \cup V_{pair}$                                      // Theorem 4
        // If there is no winning move for the player, return the opponent's win.
 28.    return (Breaker_$Win$, $R$)
end procedure
```

Algorithm 5 AND/OR search (Breaker's part).

```
procedure BREAKER_SEARCH($V$, $E$)
        // Generate all legal moves and sort them.
  1.    move_list ← all unmarked $v$ in $V$
  2.    remove all dominated vertices from move_list
  3.    sort move_list by potential weights
        // Check the player's immediate win.
  4.    for each $v$ in move_list do
  5.      $(V', E') \leftarrow (V, E)$
  6.      mark $v$ in $V'$ with Breaker's color
  7.      if all hyperedges in $E'$ are blocked then return (Breaker_$Win$, $\{v\}$)
  8.      $(result, \mathcal{P}) \leftarrow$ FIND_A _PAIRING_STRATEGY ($V'$, $E'$)
  9.      if result is success then
 10.        $R \leftarrow \{v\} \cup \{$all vertices in $\mathcal{P}\}$
 11.        return (Breaker_$Win$, $R$)
 12.      end if
 13.    end for
        // Visit all moves and do recursive call. Some moves can be safely skipped.
 14.    for each $v$ in move_list do
 15.      $(V', E') \leftarrow (V, E)$
 16.      mark $v$ in $V'$ with Breaker's color
 17.      $(result, R) \leftarrow$ MAKER_SEARCH($V'$, $E'$)
 18.      if result is Breaker_Win then return (Breaker_$Win$, $\{v\} \cup R$)
 19.    end for
        // If there is no winning move for the player, return the opponent's win.
 20.    return (Maker_$Win$, $\emptyset$)
end procedure
```

Algorithm 6 Find redundant vertices and hyperedges.

```
procedure FIND_REDUNDANCIES($V$, $E$)
  1.  $E_{redundant} \leftarrow \emptyset$
  2.  for each $e$ in $E$ do
  3.    if any vertex on $e$ is marked by Breaker then
  4.      $E_{redundant} \leftarrow E_{redundant} \cup \{e\}$
  5.  end for
  6.  $V_{redundant} \leftarrow \emptyset$
  7.  for each $v$ in $V$ do
  8.    if $v$ does not belong to any hyperedge in $E - E_{redundant}$ then
  9.      $V_{redundant} \leftarrow V_{redundant} \cup \{v\}$
 10.  end for
 11.  return $(V_{redundant}, E_{redundant})$
end procedure
```

Algorithm 6 identifies redundant hyperedges and vertices, which are mentioned in Subsection 2.3 and 4.1, respectively, by checking all vertices and hyperedges of a given position.

Appendix B

In this Appendix, we describe the implementation details to find a partial pairing set. According to Theorem 1, partial pairing can be used to reduce Maker-Breaker games. We propose Algorithm 7 to find a partial pairing set in a given game (V, E). To make our algorithm feasible for middle games, we remove all hyperedges that contain Breaker's marks from E. Then, we try to find simple cases of partial pairing sets based on mutually dominating vertices. Finally, we collect redundant vertices from the reduced game.

Algorithm 7 Find a partial pairing set.

```
procedure FIND_PARTIAL_PAIRING_SET($V$, $E$)
  1.  $E \leftarrow E - \{e |$ hyperedge $e$ that contains Breaker's marks$\}$
  2.  $(V_{pair}, E_{pair}) \leftarrow$ FIND_MUTUAL_DOMINATION $(V, E)$
  3.  $V_{pair} \leftarrow V_{pair} \cup \{v |$ redundant vertex $v$ in $(V - V_{pair}, E - E_{pair})\}$
  4.  return $(V_{pair}, E_{pair})$
end procedure
```

In Algorithm 8, we repeatedly find mutually dominated vertices in a game (V, E). Once two mutually dominating vertices are found, they are collected as a partial pair. Then, the vertices and the corresponding hyperedges are removed from the game. The above process repeats until none is found. All the collected vertices are returned as one partial pairing set.

Algorithm 8 Find mutual domination.

```
procedure FIND_ MUTUAL_DOMINATION $(V, E)$
  1.  $(V_{pair}, E_{pair}) \leftarrow (\emptyset, \emptyset)$
  2.  repeat
  3.    for each $v_i \neq v_j$ in $V$ do
  4.      if $E(v_i)$ equals to $E(v_j)$ then // $v_i$ and $v_j$ is mutually dominated
  5.        $(V_{pair}, E_{pair}) \leftarrow (V_{pair} \cup \{v_i, v_j\}, E_{pair} \cup E(v_i))$
  6.        $(V, E) \leftarrow (V - \{v_i, v_j\}, E - E(v_i))$
  7.      end if
  8.    end for
  9.  until no mutually dominated vertices exist
 10.  return $(V_{pair}, E_{pair})$
end procedure
```

Appendix C

In this Appendix, we show the implementation details of the Greedy Rules described in Section 5 in Algorithm 9.

Algorithm 9 Greedy assign pairs.

```
procedure GREEDY_ASSIGN_PAIRS($V_{unassigned}$, $E_{free}$)
 1.    $\mathcal{P} \leftarrow \emptyset$
       // Greedy Rule 1
 2.    while $e \in E_{free}$ that contains only two unassigned vertices $v_1, v_2$ exist do
 3.      $\mathcal{P} \leftarrow \mathcal{P} \cup \{(v_1, v_2)\}$
 4.      $V_{unassigned} \leftarrow V_{unassigned} - \{v_1, v_2\}$
 5.    end while
       // Greedy Rule 2
 6.    while there exists $v_i \neq v_j$ in $V_{unassigned}$
            such that $E_{free}(v_i)$ equals to $E_{free}(v_j)$ do
       // assign $e_i$ and $v_j$ as a pair
 7.      $\mathcal{P} \leftarrow \mathcal{P} \cup \{(v_i, v_j)\}$
 8.      $V_{unassigned} \leftarrow V_{unassigned} - \{v_i, v_j\}$
 9.      $E_{free} \leftarrow E_{free} - E_{free}(v_i)$
10.    end while
11.    return ($V_{unassigned}$, $E_{free}$, $\mathcal{P}$)
end procedure
```

References

[1] L.V. Allis, Searching for Solutions in Games and Artificial Intelligence, Ph.D. thesis, Univ. Limburg, Maastricht, the Netherlands, 1994.
[2] L.V. Allis, H.J. van den Herik, M.P.H. Huntjens, Go-Moku solved by new search techniques, Comput. Intell. 12 (1996) 7–23.
[3] L.V. Allis, M. van der Meulen, H.J. van den Herik, Proof-number search, Artif. Intell. 66 (1) (1994) 91–124.
[4] J. Beck, On positional games, J. Comb. Theory, Ser. A 30 (1981) 117–133.
[5] J. Beck, Positional games, Comb. Probab. Comput. 14 (5–6) (2005) 649–696.
[6] J. Beck, Combinatorial Games: Tic-Tac-Toe Theory, Cambridge University Press, 2008.
[7] E.R. Berlekamp, J.H. Conway, R.K. Guy, Winning Ways for Your Mathematical Plays, Games in Particular, vol. 2, Academic Press, London, ISBN 0-12-091102-7, 1982.
[8] C.-Y. Chen, The Research of New Rules for Gomoku and the Results of Solving Gomoku with 5x5, 6x6, 7x7 Board Size, Master Thesis, National Taiwan Normal University, Taiwan, 2013 (in Chinese).
[9] S.-H. Chiang, I-C. Wu, P.-H. Lin, Drawn k-in-a-row games, Theor. Comput. Sci. 412 (35) (2011) 4558–4569.
[10] S.-H. Chiang, I-C. Wu, P.-H. Lin, On drawn k-in-a-row games, in: The 12th Advances in Computer Games Conference (ACG12), Pamplon, Spain, May 2009.
[11] A. Csernenszky, R. Martin, A. Pluhár, On the complexity of chooser-picker positional games, Integers 2012 (2011) 427–444.
[12] L. Győrffy, G. Makay, A. Pluhár, Pairing strategies for the 9-in-a-row game, Ars Math. Contemp. 16 (1) (2019) 97–109.
[13] A.W. Hales, R.I. Jewett, Regularity and positional games, Trans. Am. Math. Soc. 106 (1963) 222–229.
[14] W.-Y. Hsu, C.-L. Ko, C.-H. Hsueh, I-C. Wu, Solving 7,7,5-game and 8,8,5-game, in: International Conference on Computers and Games (CG2018), New Taipei City, Taiwan, July 2018.
[15] A. Nagai, Df-pn algorithm for searching AND/OR trees and its applications, Ph.D. thesis, Department of Information Science, University of Tokyo, 2002.
[16] W. Pijls, Arie de Bruin, Game tree algorithms and solution trees, Theor. Comput. Sci. 252 (2001) 197–215.
[17] A. Pluhar, The accelerated k-in-a-row game, Theor. Comput. Sci. 271 (1–2) (2002) 865–875.
[18] M. Schijf, L.V. Allis, J.W.H.M. Uiterwijk, Proof-number search and transpositions, ICGA J. 17 (2) (1994) 63–74.
[19] G. Stockman, A minimax algorithm better than alpha-beta? Artif. Intell. 12 (2) (1979) 179–196.
[20] T. Thomsen, Lambda-search in game trees – with application to go, in: International Conferences on Computers and Games (*CG*2000), in: Lecture Notes in Computer Science, vol. 2063, 2001, pp. 19–38.
[21] J.W.H.M. Uiterwijk, H.J. van den Herik, The advantage of the initiative, Inf. Sci. 122 (1) (2000) 43–58.
[22] J.W.H.M. Uiterwijk, Set matching: an enhancement of the Hales-Jewett pairing strategy, in: Advances in Computer Games (ACG 2017), in: Lecture Notes in Computer Science, vol. 10664, 2017, pp. 38–50.
[23] J.W.H.M. Uiterwijk, Set matching with applications: an enhancement of the Hales–Jewett pairing strategy, ICGA J. 40 (3) (2018) 129–148.
[24] H.J. van den Herik, J.W.H.M. Uiterwijk, J.V. Rijswijck, Games solved: now and in the future, Artif. Intell. 134 (2002) 277–311.
[25] I-C. Wu, D.-Y. Huang, A new family of k-in-a-row games, in: The 11th Advances in Computer Games Conference (ACG'11), Taipei, Taiwan, September 2005.
[26] I-C. Wu, D.-Y. Huang, H.-C. Chang, Connect6, ICGA J. 28 (4) (2005) 235–242.
[27] I-C. Wu, P.-H. Lin, Relevance-zone-oriented proof search for Connect6, IEEE Trans. Comput. Intell. AI Games 2 (3) (2010) 191–207.