INTEGERS: ELECTRONIC JOURNAL OF COMBINATORIAL NUMBER THEORY 8 (2008), #G07

PROOF TREES FOR WEAK ACHIEVEMENT GAMES

Nándor Sieben
Department of Mathematics and Statistics, Northern Arizona University, Flagstaff AZ 86011-5717, USA
nandor.sieben@nau.edu

Received: 11/29/07, Revised: 9/6/08, Accepted: 9/9/08, Published: 10/7/08

Abstract

Proof number search and *threat-space search* are successful techniques for finding winning strategies in achievement games such as go-moku. A version of proof number search can be used effectively to analyze weak achievement games. In this version it is sufficient to consider the defensive moves that are involved in the maker's strategy after null moves. The result of this restricted search is a proof tree. The proof tree can be translated into the usual proof sequence of winning situations used to present winning strategies for weak achievement games. Using this procedure, a proof sequence can be found for a handicap one strategy for weakly achieving Snaky.

1. Introduction

A *polyomino* is a finite set of cells of the infinite chessboard that is connected through edges [7]. Congruent polyominoes are considered to be the same, that is, a polyomino can be freely translated, rotated and reflected. In a *polyomino weak achievement game* [6, 9], two players alternately mark a previously unmarked cell using their own colors. The first player (the *maker*) tries to mark a goal polyomino while the second player (the *breaker*) tries to prevent the maker from achieving his goal. If the maker has a strategy for achieving a polyomino then the polyomino is called a *winner*, otherwise it is called a *loser*. For all but one polyomino, it is known whether it is a winner or a loser. The known winners are subsets of L, Y or Z shown in Figure 1.1. Proof sequences describing winning strategies can be found for example in [5]. Pairing strategies for losers can be found in [13].

The only undecided polyomino is Snaky shown in Figure 1.1. Snaky is believed [6, 4] to be a winner but no winning strategy is known. A pairing strategy is the main tool to show that a polyomino is a loser. We know from [13] that no pairing strategy exists for Snaky. We

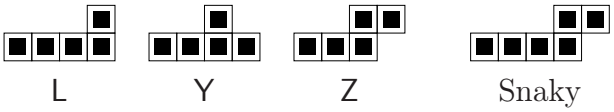


Figure 1.1: Every known winner is a subset of polyomino L, Y or Z. Snaky is undecided.

know from [12] that Snaky is an *edge-to-edge loser*, that is, the maker cannot be successful if he always marks next to his earlier marks. This suggests that finding a possible winning strategy is hard.

A winning strategy with *handicap* k is a strategy for the maker that allows k additional marks for the maker in his first turn. A handicap 2 winning strategy for Snaky was found in [11]. Two descriptions of a handicap 1 winning strategy are found in [8, 15] but these descriptions do not contain the usual proof sequence of situations as described in Section 3. Translating these descriptions to proof sequences seems difficult. We present a proof sequence for a handicap 1 strategy in Appendix B.

A proof sequence can be found by hand but this is very difficult if the goal polyomino has more than a few cells. The main purpose of this paper is to describe a procedure that can be used effectively in a computer program to find a proof sequence. The main tool is a proof tree defined in Section 2. First we describe how to construct a proof sequence from a proof tree. This has several steps. In Section 4, we convert the proof tree into a set of situations. In Section 5, we use a dependency digraph to analyze the connections between the situations. With the analysis we can greatly reduce the number of situations needed by the proof sequence. The details of this simplification process is found in Section 6.

We find proof trees using proof number search and threat search. Proof number search and threat-space search are designed for finding the game-theoretical value in game trees. They have been used to solve Connect-Four, Qubic, and Go-Moku [2, 3, 16, 1]. We adapt these techniques for weak achievement games. The main difference is that the breaker is only trying to prevent the maker from achieving his goal polyomino, she is not trying to achieve the goal polyomino on her own. So during the game tree analysis, it is sufficient to consider those moves by the breaker which are later played by the maker. These moves can be found by repeatedly using null moves for the breaker until a terminal position is found.

2. Proof Trees

To show that an animal is a winner, we can use a *proof tree* as shown in Figure 2.1. It is a partial game tree with special properties as described below. The moves of the maker are represented by solid arrows, the regular moves of the breaker are represented by dashed arrows. We also consider *null moves* by the breaker, these are represented by dotted arrows. A null move, when the breaker does not mark any cell, does not happen in real game play. We still allow them in the proof tree. The meaning of a null move is that the actual move

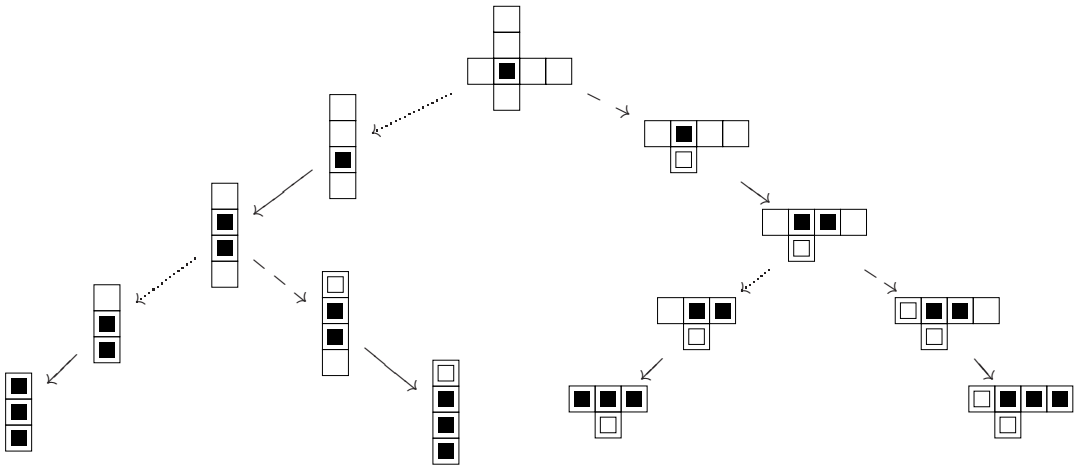


Figure 2.1: A proof tree for the size 3 skinny animal. The full squares are the marks of the maker; the empty squares are the marks of the breaker. The empty cells belong to the territory of the position. Solid arrows represent the moves of the maker. Dotted arrows represent the null moves of the breaker. Dashed arrows represent the required other moves of the breaker.

of the breaker has no effect on the maker's strategy.

The vertices of a proof tree are *positions* of the game. A position P contains the marks $M(P)$ of the maker (full squares), the marks $B(P)$ of the breaker (empty squares). A leaf vertex must be a position won by the maker, that is, the marks of the maker must contain a polyomino congruent to the goal polyomino.

A position created after a mark of the maker is called a *maker position*, the other positions are called *breaker positions*. A proof tree shows one adequate move for the maker after each possible defensive move of the breaker. So a maker position can have several outgoing arrows, but a breaker position only needs a single (solid) outgoing arrow. We say that position Q is a *daughter* of position P if there is an arrow in the proof tree from P to Q and that Q is *reachable* from P if there is a directed path from P to Q. The set of daughters of P is denoted by $D(P)$.

The *territory* $T(P)$ of a position P is the set of cells marked by the maker after position P. More precisely,

$$T(P) := \bigcup \{M(Q) \setminus M(P) \mid Q \text{ is reachable from } P\}.$$

If P is a leaf vertex then $T(P)$ is of course empty. The territory cells are shown as empty cells on Figure 2.1.

There is an infinite number of possible defensive moves but not all of them are sensible. The proof tree selects the required ones using the territories. We require that the intersection

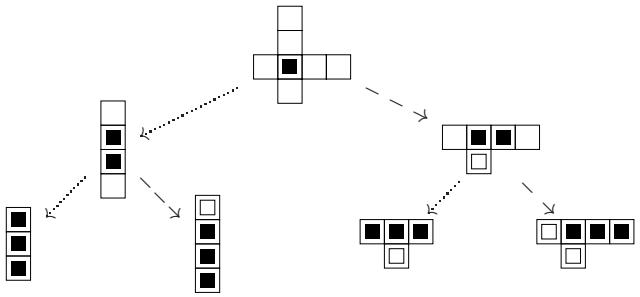


Figure 2.2: A condensed version of the proof tree of Figure 2.1. The lack of solid lines is a reminder of the condensation. The territory cells of the top position tell us that after the first null move of the breaker, the maker marked the cell above his first mark.

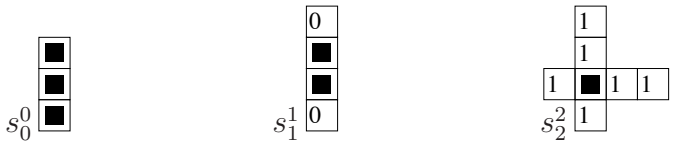


Figure 3.1: A proof sequence for the proof tree of Figure 2.2.

of the territories of the daughters of a maker position is empty, that is

$$\bigcap\{T(Q) \mid Q \in D(P)\} = \emptyset$$

for all maker position P. This condition guarantees that the proof tree considers all the necessary moves for the breaker. The maker can win from any position since the breaker can never mark a cell that ruins all possible winning lines for the maker.

Proof trees are quite large. We can condense them without losing any information by deleting the breaker positions as shown in Figure 2.2. The only drawback is that checking the territory requirements is harder. This is not a problem since every breaker position can be easily recovered from its only daughter by replacing the new maker mark by a territory cell.

3. Proof Sequence of Situations

Even a condensed proof tree becomes too large very quickly as the goal polyomino grows. Figure 4.1 shows a partial condensed proof tree for the polyomino Z. Instead of the proof tree a strategy can be captured by a *proof sequence* $(s_0, \ldots, s_n)$ of *situations* [5, 14, 18]. A situation $s_i = (C_{s_i}, N_{s_i})$ is an ordered pair of disjoint sets of cells. We think of the *core* C_{s_i} as a set of cells marked by the maker and the *neighborhood* N_{s_i} as a set of cells not marked by the breaker. A situation is the part of the playing board that is important for the maker. A situation does not contain any of the breaker's marks. Those marks are not

important as long as the situation contains enough empty cells in the neighborhood. Just like polyominoes, congruent situations are considered to be the same. If s is a situation then we define $s \setminus x := (C_s \setminus \{x\}, N_s \setminus \{x\})$.

In the situations of a proof sequence, it is always the breaker who is about to mark a cell. The game progresses from s_n towards s_0. We require that C_{s_0} is the goal polyomino and $N_{s_0} = \emptyset$. This means that the maker already won by marking the cells in C_{s_0} and there is no need for any free cells on the board in N_{s_0}. For each $i \in \{1, \ldots, n\}$ we also require that if the breaker marks a cell in N_{s_i} then the maker can mark another cell of N_{s_i} and reach a position s_j closer to his goal, that is, satisfying $j < i$. More precisely, for all $x \in N_{s_i}$ there must be an $\tilde{x} \in N_{s_i} \setminus \{x\}$ and a $j \in \{0, \ldots, i-1\}$ such that

$$C_{s_j} \subseteq C_{s_i} \cup \{\tilde{x}\} \text{ and } N_{s_j} \subseteq C_{s_i} \cup N_{s_i} \setminus \{x\}.$$

This relationship between s_i and s_j is denoted by $s_i \vdash_x s_j$.

Figure 3.1 shows a proof sequence that captures the winning strategy of the proof tree of Figure 2.2. In the figure the situations are denoted by s_i^d where the upper index d is the number of required additional marks by the maker until the goal polyomino is reached, assuming optimal defense from the breaker. The numbers inside the neighborhood cells denote the index j of the situation that can be reached if the breaker marks the given cell. Note that the maker can win in 3 turns while the number of cells in the goal polyomino is also 3. We call such a strategy *economical*.

The situations of the proof sequence of Figure 3.1 are simply built from the positions of Figure 2.2. Given a position P the corresponding situation is $(M(P), T(P))$. This works well if the strategy is economical. If the strategy is not economical then we can construct simpler situations.

4. From Proof Tree to Situations

The proof tree of Figure 4.1 is not economical. The maker sometimes needs to mark 6 cells even though the goal polyomino Z only has 5. It is clear that in these positions some of the marks of the maker are not essential to build a corresponding situation. To decide what is important we introduce the notion of *essence* of a position.

If P is a leaf position then $M(P)$ must contain a copy of the goal polyomino. An essence $E(P)$ of P can be chosen to be one of these copies. If P is not a leaf position then the essence of P is the union of the essences of the leaf positions that can be reached from P, that is,

$$E(P) := \bigcup \{E(Q) \mid Q \text{ is a leaf reachable from } P\}.$$

A situation $s(P)$ corresponding to position P is defined as

$$s(P) := (M(P) \cap E(P), T(P) \cap E(P)).$$

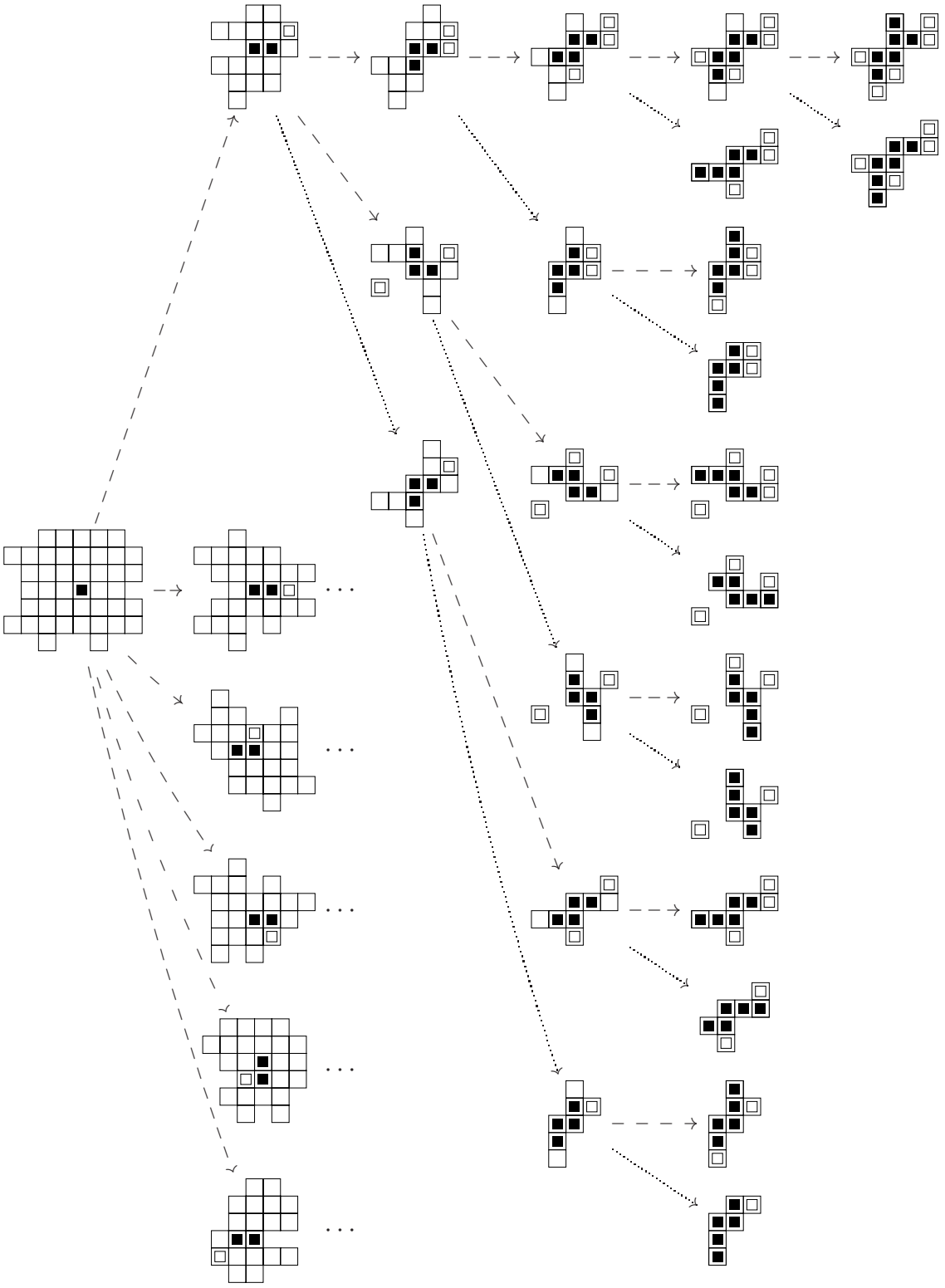


Figure 4.1: A partial condensed proof tree for the polyomino Z. The figure is rotated by 90°. Note that this winning strategy is not economical.

Note that if P is a leaf position then $s(P) = (G, \emptyset)$ where G is the goal polyomino. Since the essence of a leaf position is not unique, we can get several situations from a position. This is not a drawback. If we have more situations then it is easier to find a short proof sequence.

Applying this procedure to the maker positions of the proof tree of Figure 4.1, we can build a 31-element set $\mathcal{S} = \{s(P) \mid P \text{ is a maker position}\}$. The situations in $\mathcal{S}$ could be ordered into a proof sequence, but we do not need all of them.

5. Dependency Digraph of Situations

Let $\mathcal{S}$ be the set of situations gotten from the maker positions of a proof tree using the procedure described in Section 4. Our goal is to use as few situations of $\mathcal{S}$ as possible to build a proof sequence. To analyze the connections between the elements of S, we build a *dependency digraph* D.

Let $\mathcal{T} := \{s \setminus x \mid s \in S \text{ and } x \in N_s\}$ be the set of situations with one neighborhood cell deleted. Note that $\mathcal{T}$ is the set of situations that can be reached from an element of $\mathcal{S}$ applying a mark by the breaker. The vertex set of the dependency digraph is $V_D := \mathcal{S} \cup \mathcal{T}$.

To define the arrow set, we need to partition $\mathcal{S}$ into levels $\{\mathcal{L}_0, \ldots, \mathcal{L}_m\}$. The levels are chosen recursively. We let $\mathcal{L}_0 := \{(G, \emptyset)\}$ where G is the goal polyomino. Given $\mathcal{L}_0, \ldots, \mathcal{L}_i$, let $\mathcal{K}_i := \cup_{j=0}^{i} \mathcal{L}_j$ and define

$$\mathcal{L}_{i+1} := \{s \in \mathcal{S} \setminus \mathcal{K}_i \mid (\forall x \in N_s)(\exists t \in \mathcal{K}_i)\, s \vdash_x t\}.$$

The arrow set of the dependency digraph is

$$E_D := \{(s, s \setminus x) \mid s \in S, x \in N_s\} \cup \{(s \setminus x, t) \mid 0 \le i < m, s \in \mathcal{L}_{i+1}, t \in \mathcal{K}_i \text{ and } s \vdash_x t\}.$$

The dependency digraph shows the possible routes for the maker to win after each defensive move of the breaker.

Figure 5.1 shows the dependency digraph of the situations of the proof sequence of Figure 3.1. In this digraph each vertex in $\mathcal{T}$ has a single outgoing arrow, so the maker never has a choice. This implies that the proof sequence cannot be simplified.

Figure 5.2 shows a partial dependency digraph of the situations gotten from the proof tree shown in Figure 4.1. Note that there are two ways to continue towards the goal polyomino from vertex $s_2^2 \setminus x$. If we delete situation r^1, the maker still can win from $s_2^2 \setminus x$ by picking s_0^0 instead of r^1. So situation s_2^2 does not depend on situation r^1. It is possible that other situations depend on r^1 so we cannot delete r^1 without looking at the whole dependency digraph.

Even if we can delete a situation, it might not be the best choice to do so. If we delete a situation, we cut some of the possible routes towards the goal polyomino, so other situations

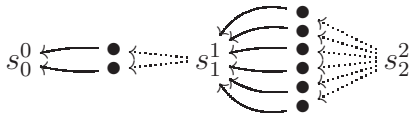


Figure 5.1: A dependency digraph of the situations gotten from the proof sequence of Figure 3.1. The vertices in $\mathcal{T}$ are denoted by bullets.

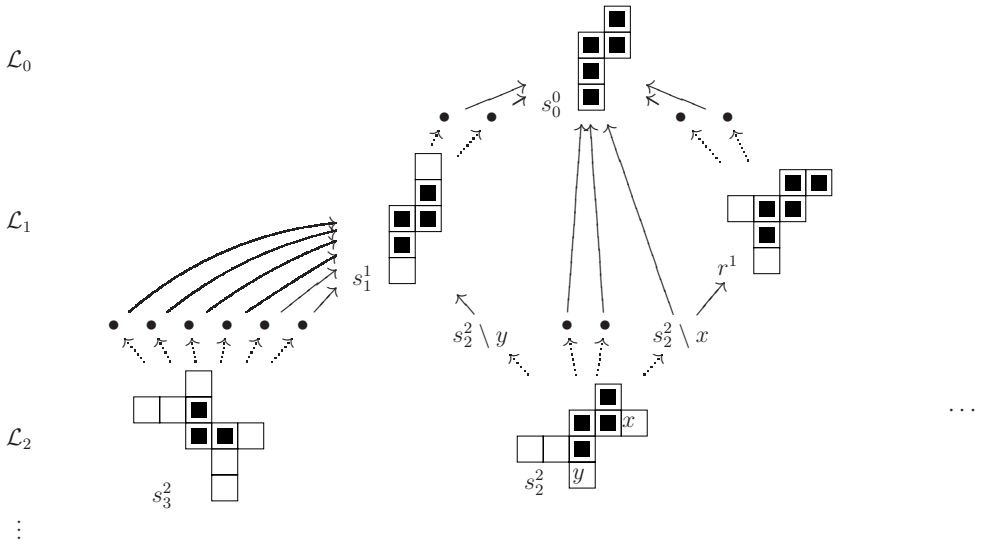


Figure 5.2: Partial dependency digraph of the situations gotten from the proof tree of Figure 4.1. Levels $\mathcal{L}_0$ and $\mathcal{L}_1$ are complete.

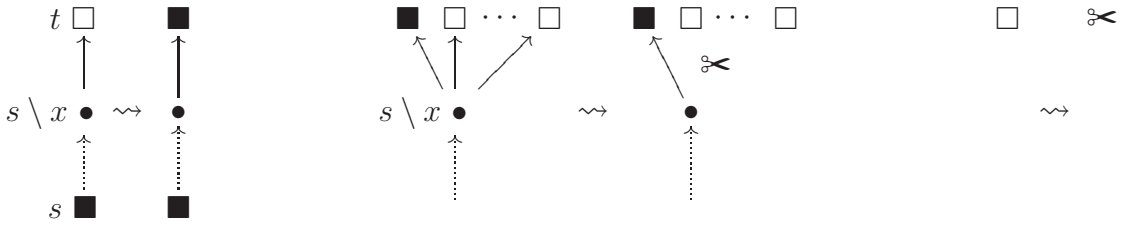
could become indispensable. Our goal is to find the largest set of situations that can be deleted together.

6. Simplification of the Dependency Digraph

Let $\mathcal{S}$, $\mathcal{T}$, $\{\mathcal{L}_0, \ldots, \mathcal{L}_m\}$ and D be defined as in the previous section. We know that $\mathcal{L}_0 = \{(G, \emptyset)\} = \{s_0^0\}$ where G is the goal polyomino. We also know that $\mathcal{L}_m = \{s_n^m\}$ is also a singleton set containing the initial situation s_n^m. The most important property of the dependency digraph is that every directed path starting at s_n^m can be continued until it reaches s_0^0. Our goal is to delete as many situations from $\mathcal{S}$ as possible while keeping this property alive. Of course if we delete a situation from $\mathcal{S}$ then all of its daughters can be deleted from $\mathcal{T}$.

Some of the situations in $\mathcal{S}$ are indispensable, we collect these situations in the set $\mathcal{I}$. Here are the simplification rules we use to build $\mathcal{I}$ and to delete some vertices or edges.

1. We have $s_0^0, s_n^m \in \mathcal{I}$. The starting and goal positions are clearly indispensable.



rule (2) rule (3) rule (4)

Figure 6.1: Schematic rules for simplifying a dependency digraph of situations. The black squares represent the indispensable situations. White squares represent undecided situations. The bullets represent the situations in $\mathcal{T}$.

2. Let $s \in \mathcal{I}$ and $s \setminus x \in \mathcal{T}$. If t is the only element of $\mathcal{S}$ with $(s \setminus x, t) \in E_D$ then $t \in \mathcal{I}$. If s is indispensable and the maker has only one possible answer t for a defensive move x of the breaker then t must also be indispensable.

3. If $t \in \mathcal{I}$ and $(s \setminus x, t), (s \setminus x, t_1), \ldots, (s \setminus x, t_k) \in E_D$ then $(s \setminus x, t_1), \ldots, (s \setminus x, t_k)$ should be deleted. If the maker can continue the game through an already indispensable situation t then he should do so and avoid any other choices.

4. If a situation in $\mathcal{S}$ has no incoming arrows then the situation should be deleted, since clearly no other situation is dependent on it.

Figure 6.1 shows a schematic representation of these rules.

The use of the simplification rules help each other, so we use them while any of them is applicable. If the digraph still has a vertex $s \setminus x \in \mathcal{T}$ such that s is indispensable and $s \setminus x$ has more than one outgoing arrows then further simplifications are possible. Then we find one such $s \setminus x$ with the smallest number of outgoing arrows $(s \setminus x, t_1), \ldots, (s \setminus x, t_n)$ and we make one of $t_1, \ldots, t_n$ indispensable artificially and apply the simplification rules again. At this point we have to use a backtracking algorithm to find the smallest dependency digraph that still has the path continuation property.

There is still another possibility for simplification. If $(s, s \setminus x), (s \setminus x, t_1), (s \setminus x, t_2) \in E_D$ and we delete t_2 from the digraph then it possible that the situation s itself can be reduced. It is possible that a core cell or a neighborhood cell of s was needed to guarantee that situation t_2 is reachable. Since t_2 is no longer an option, these cells could be deleted form s. In theory we could check for this possibility every time we delete a situation and recalculate the dependency digraph. This would make the backtracking quite a bit more complicated and also much slower. To avoid this difficulty we only check for this possibility after the backtracking part is done.

After these simplification processes, the remaining situations in $\mathcal{S}$ can be ordered into a proof sequence. Figure 6.2 shows the result of the algorithm applied to the proof tree of

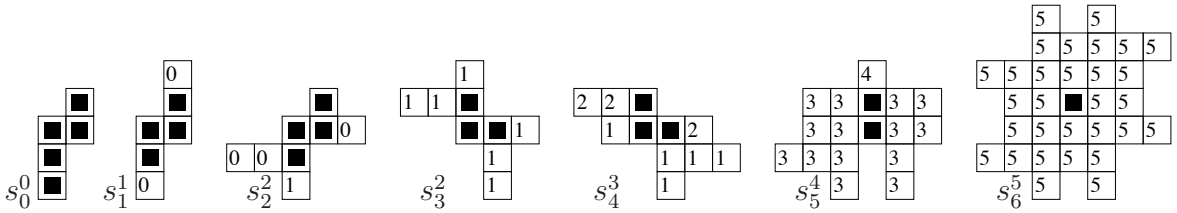


Figure 6.2: A proof sequence for Z.

Figure 4.1. Every situation in this proof sequence can be gotten from the positions shown on the partial proof tree of Figure 4.1. Note that the starting position s_6^5 of the proof sequence is a reduced version of the situation that corresponds to the initial position of Figure 4.1.

7. Proof Number Search

To find a proof tree we use a standard proof number search combined with a variation of threat-space search [1]. Proof number search is an algorithm to evaluate an *AND/OR tree* containing AND-nodes and OR-nodes. A node can have three possible values: *true* (1), *unknown* ($\frac{1}{2}$) or *false* (0). The value of node P is denoted by $v(P)$. If the values of the leaf nodes are known, then the value of an internal node is determined by the values of its children using the following truth tables:

$\vee$	1	$\frac{1}{2}$	0
1	1	1	1
$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{2}$
0	1	$\frac{1}{2}$	0

$\wedge$	1	$\frac{1}{2}$	0
1	1	$\frac{1}{2}$	0
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0
0	0	0	0

Note that $\vee$ takes the maximum and $\wedge$ takes the minimum of the arguments. This is why OR-nodes and AND-nodes are also called MAX-nodes and MIN-nodes.

Our AND/OR trees correspond to game trees. The AND-nodes correspond to maker position and the OR-nodes to breaker positions. This means that at an AND-node the breaker can try several defensive moves. At an OR-node, the maker only needs to find one good move. The value of a node is true if the maker wins from that position. We make the value of a node false if the node seems hopeless for the maker. To determine the value of the root node, the search expands as many leaf nodes with unknown value as needed.

The order of expansion has a great effect on the number of required expansions. We use *proof* and *disproof numbers* to pick the *most proving* leaf node to expand. The proof number is roughly the minimum number of nodes we need to evaluate to conclude that the value of the node is true. The disproof number is roughly the minimum number of nodes we need to

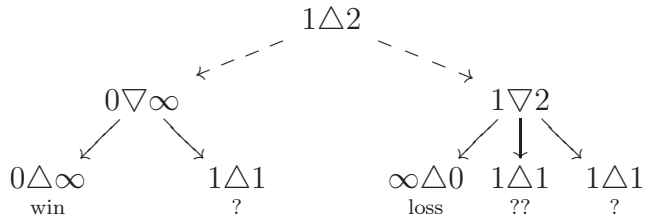


Figure 7.1: An AND/OR tree with proof and disproof numbers. The AND-nodes are represented by $\triangle$, the OR-nodes by $\bigtriangledown$. Unknown leaf nodes are denoted by a question mark. The most proving node is denoted by two question marks. The proof numbers are on the left, the disproof numbers are on the right.

evaluate to conclude that the value of the node is false. If P is a leaf node then the proof number $p(P)$ and disproof number $d(P)$ of P is defined by

$$p(P) := \begin{cases} 0 & \text{if } v(P) = 1 \\ \infty & \text{if } v(P) = 0 \\ i(P) & \text{if } v(P) = \frac{1}{2} \end{cases}, \qquad d(P) := \begin{cases} \infty & \text{if } v(P) = 1 \\ 0 & \text{if } v(P) = 0 \\ i(P) & \text{if } v(P) = \frac{1}{2} \end{cases}.$$

In the simplest implementation $i(P) = 1$. If $i(P)$ is defined to be the depth of P then expanding deeper nodes is more expensive and the search tree becomes shallower. If the OR-node $\check{P}$ and the AND-node $\hat{P}$ are interior nodes then

$$p(\check{P}) := \min\{p(Q) \mid Q \in D(\check{P})\}, \qquad d(\check{P}) := \sum_{Q \in D(\check{P})} d(Q),$$

$$p(\hat{P}) := \sum_{Q \in D(\hat{P})} p(Q), \qquad d(\hat{P}) := \min\{d(Q) \mid Q \in D(\hat{P})\}.$$

The most proving node can be found by starting at the root node and following the first daughter with the smallest proof number at OR-nodes, and with the smallest disproof number at AND-nodes. Figure 7.1 shows an example.

After expanding the most proving node we update the proof and disproof numbers. After this update we delete all nodes with a 0 disproof number.

8. Threat Sequences

Let s be a winning situation and let P be a breaker position, that is, a position when the maker is on the move. If the maker can mark a cell and create a copy of s then he can win from that position so the game is decided. Now suppose that the maker can mark a cell x and almost create a position (C, N) that is almost the same as s except that one core cell y missing, that is, $(C \cup \{y\}, N) = s$. Then the outcome is not yet decided but he created a *threat* for which the breaker has to respond. To avoid this threat the breaker needs to mark

y or a cell of N. Using the terminology of [1], cell x is a *gain cell* and the cells of $N \cup \{y\}$ are *cost cells* of the threat.

A *threat sequence* is a sequence $(P_0, P_1, \ldots, P_n)$ of breaker positions such that P_{i+1} can be gotten from P_i by marking the gain cell of the threat with the maker's color and marking the cost cells with the breaker's color. The threat sequence is called *winning* if the maker can achieve an actual winning situation from P_n. Figure 8.1 shows two winning threat sequences to achieve Z. It is clear that if there is a winning threat sequence starting at P_0, the maker can win from position P_0 by marking the gain cells.

It is possible that the maker can win from a position even though there is no winning threat sequence. Still it is useful to search for threat sequences since this is much faster than a full game tree analysis. The search also helps deciding what moves should be considered during the proof number search.

In our implementation we only consider *dependent threat sequences*. A threat sequence is dependent if for each $i \geq 1$ the threat used to create P_{i+1} would not exist without the gain cell of the threat used to create P_i. This means that we do not want to experiment with using available threats in different orders. We only want to follow newly created threats.

To create a threat sequence we need to find threats and threats need winning situations. We do not have a proof sequence, that is why we are interested in threat sequences in the first place. So we need to create winning situations. It is possible to use only the goal polyomino. This is, in fact, what we did to find the proof tree of Figure 4.1. For more difficult games we can create winning positions by hand as in [18]. Another possibility is to use the winning situations of a proof sequence for a handicap strategy.

Threat sequences simulate the tactical thinking of human players. Humans often find winning lines in the game by disregarding the difficulties caused by the marks of the opponent. This is successful because tactics are very important in achievement games. The long term strategy that comes from experience and intuition of human players are simulated by the proof number search.

9. Node Expansion

During the expansion of nodes in the proof number search we need to create possible moves. The procedure depends on the type of the node.

If we expand an OR-node $\check{P}$ then we need to select our best candidates for the maker to mark. Of course if $\check{P}$ has a copy of the goal polyomino marked by the maker then we can assign $v(\check{P}) := 1$ and there is no need for further expansion. Otherwise we check whether the depth of the node is beyond our limit. If it is then our search in this direction is hopeless and we assign $v(\check{P}) := 0$. If we did not reach the depth limit then we search for a winning

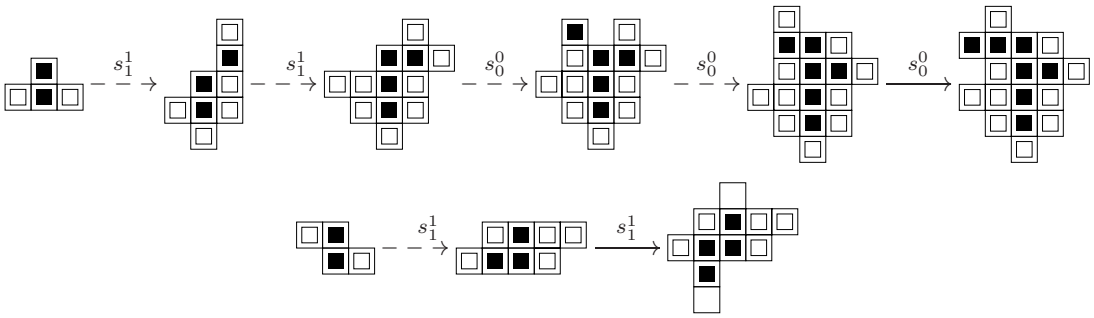


Figure 8.1: Two winning threat sequences to achieve Z. In each position the maker is on the move. The labels on the arrows show the name of the situation of Figure 6.2 used to create the threat. Dashed arrows lead to defended threats while solid arrows lead to winning situations.

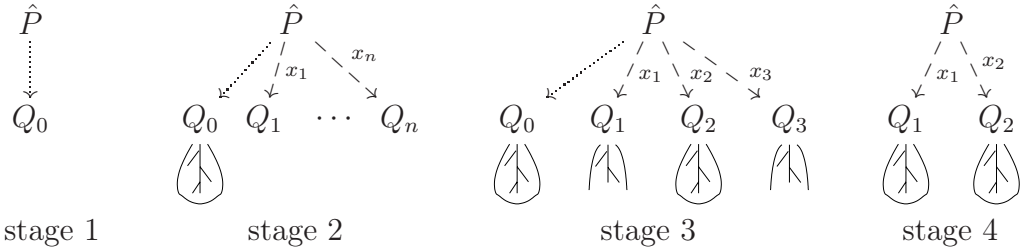


Figure 9.1: Four stages of the life of an AND-node. A label on a dashed arrow shows the cell marked by the maker for his move. There are no labels on the dotted arrows since those represent null moves. In stage 3, $T(Q_0) \cap T(Q_2) = \{x_1, x_3\}$.

threat sequence. If we can find one then the maker should mark the gain cell of the first position of the winning threat sequence, so $\check{P}$ has only one daughter after the expansion. If we cannot find a winning threat sequence or the search for it reaches our depth limit then we need to use some heuristics to select a few promising moves. During this selection we use an evaluation function that measures the potential of a move to create new threats in the future and we pick the moves with the highest values. The evaluation function finds the possible placements of the winning situations on the board with as few missing core cells as possible. We try to avoid picking moves that create immediate threats since those moves are already analyzed by the search for a winning threat sequence. We also try to avoid moves that do not create threat moves after a null move of the breaker.

If we expand an AND-node $\hat{P}$ then our main concern is to satisfy the territory intersection property of proof trees. When we expand $\hat{P}$ we simply create a null move for the breaker. This is shown as stage 1 on Figure 9.1. The resulting position is denoted by Q_0. If Q_0 is expanded further and evaluates to true, then we need to create additional daughters $\{Q_1, \ldots, Q_n\}$ for $\hat{P}$. If $T(Q_0) = \{x_1, \ldots, x_n\}$ then we let $M(Q_i) := M(\hat{P}) \cup \{x_i\}$ and $B(Q_i) := B(\hat{P})$. This is stage 2. Whenever an additional daughter evaluates to true we can erase some of the daughters to satisfy

$$\{x_i \mid v(Q_i) = \tfrac{1}{2}\} = \bigcap \{T(Q_i) \mid v(Q_i) = 1\}.$$

This is stage 3. At the end we reach stage 4 when $\bigcap\{T(Q_i) \mid v(Q_i) = 1\} = \emptyset$ so the value of $\hat{P}$ becomes true and we do not need to consider any more daughters. It is possible that the intersection of the territories is empty even if we delete some of the daughters. Node Q_0 can be deleted frequently this way because the other daughters require more sophisticated play from the maker and so their territories are likely smaller. Of course it is possible that a daughter evaluates to false which makes the value of $\hat{P}$ false as well. Then node $\hat{P}$ is deleted from the tree.

10. The Art of Finding Proof Trees

If the goal polyomino is relatively simple then the procedure we described works automatically even if the initial set of situations used in the threat sequence search and the move selection contains only the goal polyomino or a few more situations that can be easily created by hand. For more complicated goals this is not so easy because the search takes too long. We need to watch how the search progresses and adjust some parameters like the depth limits, the number of considered moves for the maker, the evaluation function to select promising moves. Occasionally cutting a few branches that are clearly hopeless by hand speeds up the search significantly.

We need to be more sophisticated with the initial set of situations as well. First we use only a few initial situations and a starting board position that has a large handicap number. This starting board position could contain for example a couple of cells marked by the maker. The proof number search finishes quickly and the proof sequence gotten from the resulting proof tree becomes the new initial set of situations for the next search.

To gain something from this next search we need to make the starting board position harder to win. This can be done by adding a few cells marked by the breaker. The closer these cells are to the action the harder it is to finish the proof number search and the more useful the resulting proof sequence. Another way to make the starting board position harder is to decrease the handicap, that is, to add fewer cells marked by the maker.

It would be possible to automate this process by extending the initial set of situations with new winning situations discovered during the proof number search. Every time the value of and AND-node becomes true, we could add the corresponding situation $s(P)$ to the initial set of situations. Of course this burden of knowledge slows down the move selection and the threat sequence search. Implementing this would perhaps help to settle the fate of Snaky.

References

[1] L. V. Allis, H. J. van den Herik, and M. P. H. Huntjens. Go-moku and threat-space search. (Preprint) http://citeseer.ist.psu.edu/170657.html.

[2] L. Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Computer Science Department Rijksuniversiteit Limburg, 1994.

[3] L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik. Proof-number search. *Artif. Intell.*, 66(1):91–124, 1994.

[4] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning ways for your mathematical plays. Vol. 2. 2nd ed.* Natick, MA: A K Peters. xvii, 277–473 , 2003.

[5] Jens-P. Bode and Heiko Harborth. Hexagonal polyomino achievement. *Discrete Math.*, 212(1–2):5–18, 2000. Graph theory (Dörnfeld, 1997).

[6] Martin Gardner. Mathematical games. *Sci. Amer.*, 240:18–26, 1979.

[7] Solomon G. Golomb. *Polyominoes: Puzzles, Patterns, Problem and Packings*. Princeton University Press, 1965.

[8] Immanuel Halupczok and Jan-Christoph Schlage-Puchta. Achieving snaky. *Integers*, 7:G02, 28 pp. (electronic), 2007.

[9] Frank Harary. Achievement and avoidance games on finite configurations. *J. Recreational Math.*, 16(3):182–187, 1983/84.

[10] Frank Harary. Is Snaky a winner? *Geombinatorics*, 2(4):79–82, 1993.

[11] Frank Harary, Heiko Harborth, and Markus Seemann. Handicap achievement for polyominoes. In *Proceedings of the Thirty-first Southeastern International Conference on Combinatorics, Graph Theory and Computing (Boca Raton, FL, 2000)*, volume 145, pages 65–80, 2000.

[12] Heiko Harborth and Markus Seemann. Snaky is an edge-to-edge looser. *Geombinatorics*, 5(4):132–136, 1996.

[13] Heiko Harborth and Markus Seemann. Snaky is a paving winner. *Bull. Inst. Combin. Appl.*, 19:71–78, 1997.

[14] Heiko Harborth and Markus Seemann. Handicap achievement for squares. *J. Combin. Math. Combin. Comput.*, 46:47–52, 2003. 15th MCCCC (Las Vegas, NV, 2001).

[15] Hiro Ito and Hiromitsu Miyagawa. Snaky is a winner with one handicap. *8th Hellenic European Conference on Computer Mathematics and its Applications*, 2007.

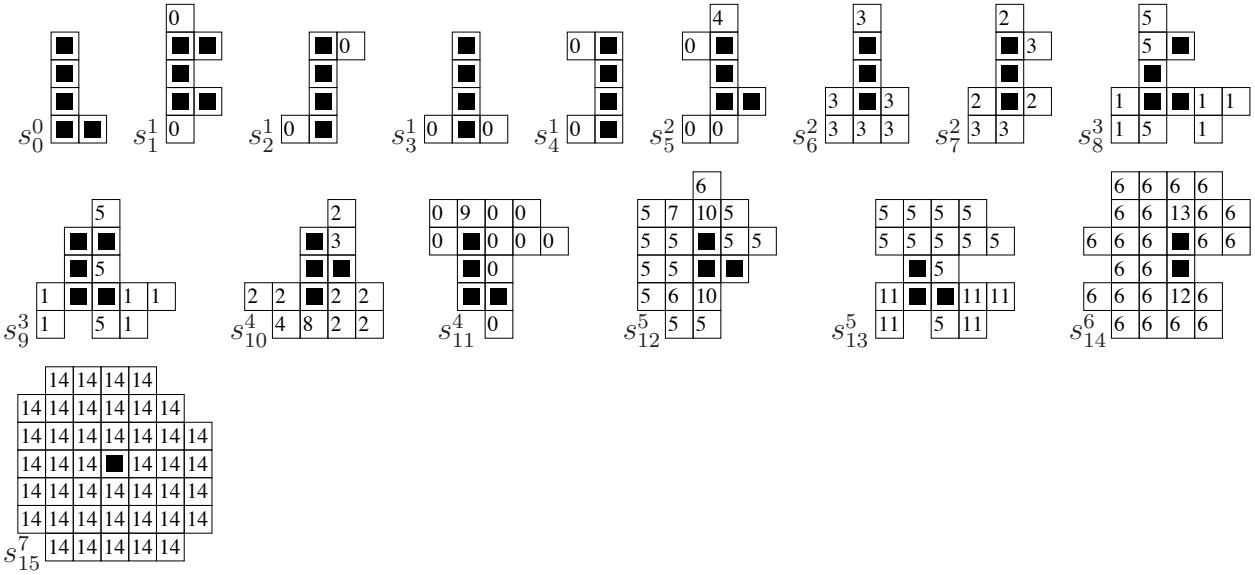
[16] M. P. H. Huntjens L. V. Alus, H. J. van den Herik. Go-moku solved by new search techniques. *Computational Intelligence*, 12(1):7–23, 1996.

[17] Nándor Sieben. Snaky is a 41-dimensional winner. *Integers*, 4:G5, 6 pp. (electronic), 2004.

[18] Nándor Sieben and Elaina Deabay. Polyomino weak achievement games on 3-dimensional rectangular boards. *Discrete Mathematics*, 290:61–78, 2005.

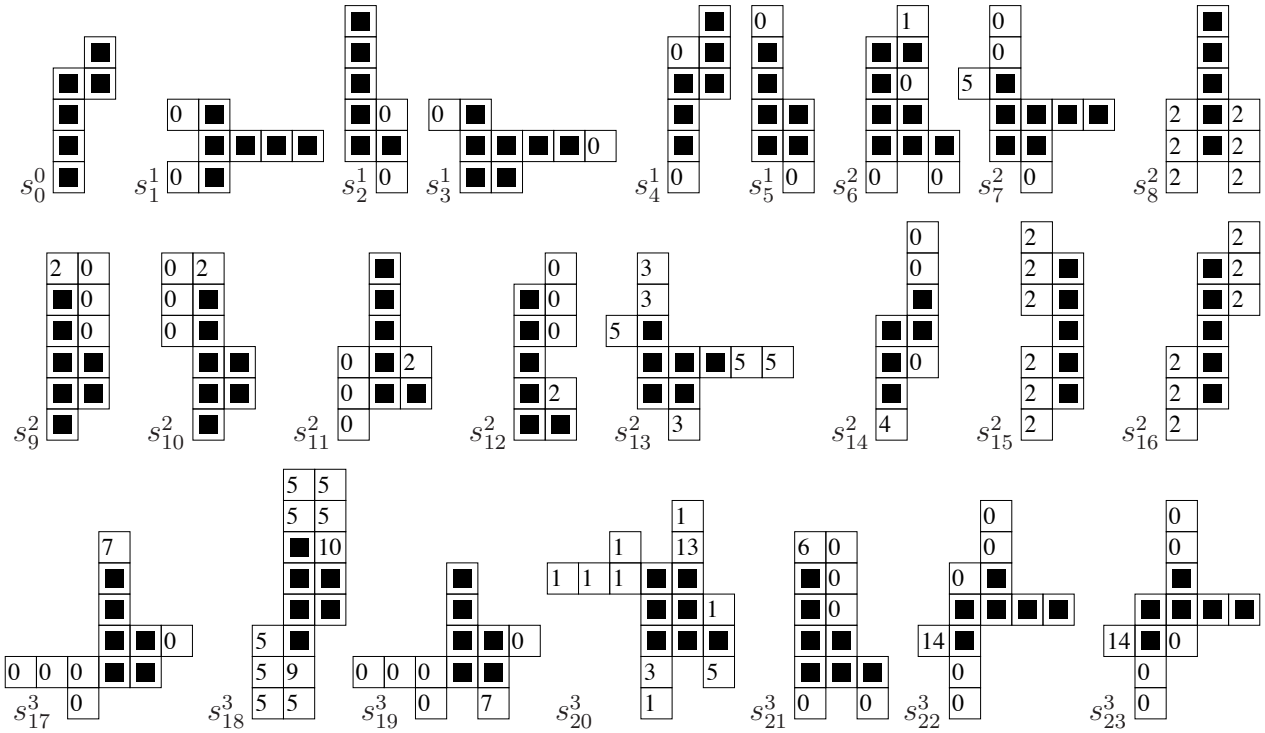
A. A Proof Sequence for L

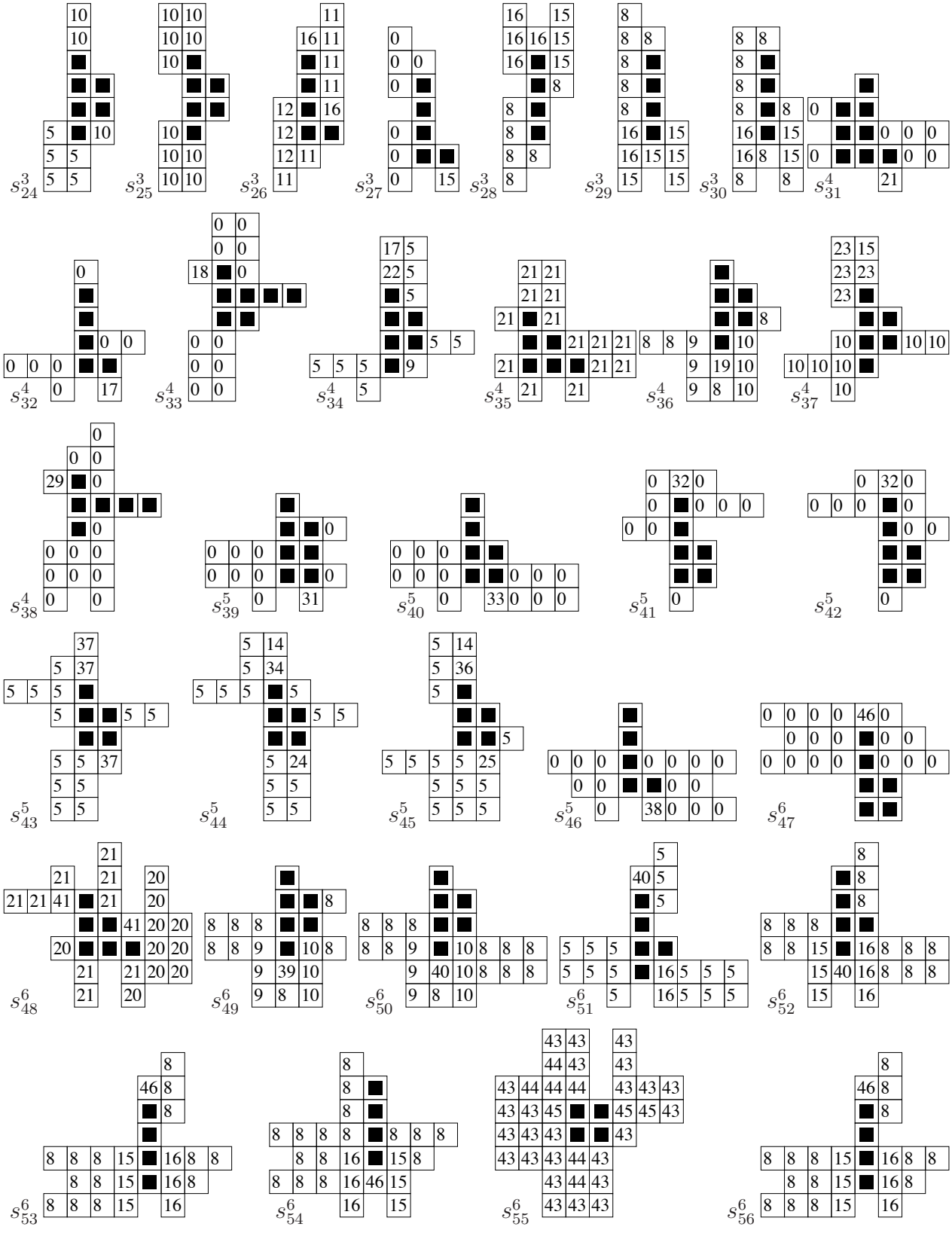
The winning strategy for L published in [5] contains 33 situations. We managed to cut the number of situations in the proof sequence into half. The initial set of winning situations contained s_0^0, s_2^1, s_3^1, s_4^1 and s_6^2.

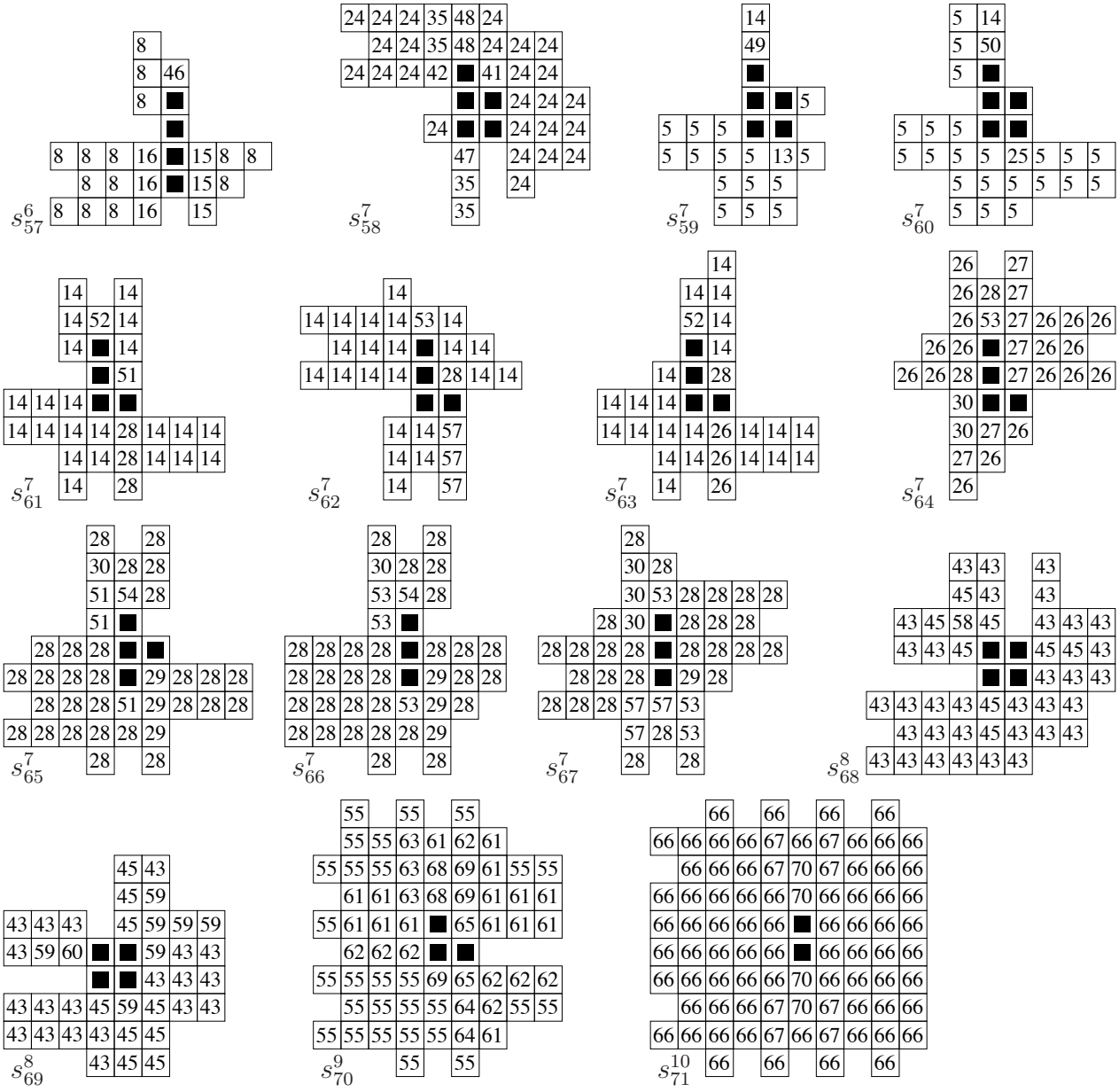


B. A Handicap 1 Proof Sequence for Snaky

Snaky has a long history [10, 14, 12, 13, 17, 8, 15]. It is believed to be a winner. Our procedure found a handicap 1 proof sequence. For initial set of winning situations we used the handicap two proof sequence of [11]. The last situation in our proof sequence is s_{73}^{10}, so using this strategy the maker can win in 11 turns.




s^3_{24}
s^3_{25}
s^3_{26}
s^3_{27}
s^3_{28}
s^3_{29}
s^3_{30}
s^4_{31}
s^4_{32}
s^4_{33}
s^4_{34}
s^4_{35}
s^4_{36}
s^4_{37}
s^4_{38}
s^5_{39}
s^5_{40}
s^5_{41}
s^5_{42}
s^5_{43}
s^5_{44}
s^5_{45}
s^5_{46}
s^6_{47}
s^6_{48}
s^6_{49}
s^6_{50}
s^6_{51}
s^6_{52}
s^6_{53}
s^6_{54}
s^6_{55}
s^6_{56}


s^6_{57}
s^7_{58}
s^7_{59}
s^7_{60}
s^7_{61}
s^7_{62}
s^7_{63}
s^7_{64}
s^7_{65}
s^7_{66}
s^7_{67}
s^8_{68}
s^8_{69}
s^9_{70}
s^{10}_{71}