Hive Is PSPACE-Hard

Daniël I. Andel
Utrecht University, The Netherlands

Benjamin G. Rin
Utrecht University, The Netherlands

Abstract

Hive is an abstract strategy game played on a table with hexagonal pieces. First published in 2001, it was and continues to be highly popular among both casual and competitive players. In this paper, we show that for a suitably generalized version of the game, the computational problem of determining whether a given player in an arbitrary position has a winning strategy is PSPACE-hard. We do this by reduction from Formula Game, which is first reduced to an intermediate problem we call Formula Game Geography, after which the latter is reduced to our decision problem.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases Computational complexity, Combinatorial games, Hive, PSPACE-hardness, Formula Game, Generalized Geography

Digital Object Identifier 10.4230/LIPIcs.FUN.2026.3

Related Version *ArXiv Version*: `https://arxiv.org/abs/2506.03492`

Acknowledgements We would like to thank Rosalie Iemhoff and Robert Barish for their helpful conversations and comments. We also wish to thank the anonymous referees for their valuable feedback.

1 Introduction

Hive is a popular strategy game designed by John Yianni and published in 2001. Like classic games such as chess, checkers, and Go, it is a two-player, perfect-information, deterministic, turn-based tabletop game with at most one winner. In this paper, we consider the computational complexity of the decision problem Hive, which asks whether a given player has a forced winning strategy from a given position in a generalized version of Hive. Our main result is that this decision problem is PSPACE-hard. We prove this by reduction from Formula Game, first by reducing the latter to a variant of Generalized Geography that we call Formula Game Geography (FGG), then in turn reducing FGG to Hive.[1]

While the standard version of Hive has only 22 pieces and a small set of rules, it is nevertheless a deeply complex and strategic game. Still, it has only a finite number of possible game positions, making it possible in principle to construct a large look-up table which unveils a winning strategy. Therefore, as with most board games, investigating its asymptotic computational complexity requires us to generalize the game in some natural way. Typically this is done by generalizing the board to dimensions $n \times n$ (cf. [2, 3, 5, 6, 8, 9, 11]). However, Hive does not have a board; the playing field is determined by the placement of pieces. Hence, we generalize Hive to *n-Hive* by stipulating simply that each player has an equal set of n pieces to play with, rather than 11. We make no further adjustments to the rules (Section 1.1).

[1] The proof presented in this paper originates from work done for the bachelor thesis of the alphabetically first author, written under supervision of the second [1].

We note, though, that our generalization of Hive to n-Hive allows for each player to have multiple Queens. In Hive, the Queen is a piece that serves a role similar to that of the King in chess, so this generalization is arguably not as natural as one permitting only one Queen. By contrast, complexity bounds for $n \times n$ chess found in [2] and [11] work with generalizations of chess with only one King per player. We return to this consideration in Section 4.

Hereafter, we may refer to n-Hive as "Hive" for simplicity. Our main theorem is then stated as follows.

▶ **Theorem** (Main theorem). *The problem* Hive *of determining whether a given player has a winning strategy in an arbitrary Hive position is PSPACE-hard.*

Analogous results have been found over the last several decades for various other strategy games. In the 1980s, generalized versions of classic games such as Go [5], Gobang [6], Hex [7], chess [2, 11], and checkers [9] were proved to be PSPACE-hard. Since then, the complexity of numerous other games has been determined. Recent examples of PSPACE-hard problems include generalizations of the video game Lemmings [12], the ancient Hawaiian board game Konane [3], and the 2003 board game Arimaa [8]. Some of the known hard games have been found to be PSPACE-complete, others are known to be EXPTIME-complete, and others still have no known upper complexity bound. The determining factor in whether a two-player game's complexity lies within PSPACE is typically the presence or absence of a polynomial bound on the potential length of games (see, e.g., [4, Ch. 6]). In the case of Hive, no such bound is known to exist. Accordingly, we consider it likely that Hive is not in PSPACE.

To prove that Hive is PSPACE-hard, we begin with the classic Formula Game (Section 2.1.1), which reduces to the problem Formula Game Geography (FGG) we define in Definition 12. Briefly, FGG is Generalized Geography played on a directed graph that simulates a Formula Game instance, with some added constraints on the degrees of nodes. It is straightforwardly seen to be PSPACE-complete (Corollary 13). Our proof for Hive then follows the same broad approach as that of Lichtenstein and Sipser's well-known proof that $(n \times n)$ Go is PSPACE-hard [5]. However, we find that implementing suitable gadgets in Hive is nontrivial, both locally in terms of each gadget's internal construction and globally with respect to the interconnections between gadgets.

First, we must account for Hive's unusual geometry. Unlike Go, chess, and most other classic games, Hive does not have pieces interacting on a square grid at convenient 90° angles. Instead, Hive pieces are hexagonal and always placed in such a way that every piece borders at least one other. While this is not prohibitively limiting, it does call for careful measures in gadget design, as well as the construction of special gadgets with names such as *60° direction changer* and *120° direction changer*.

Second, with respect to more global positional features, we encounter challenges arising from our use of an important game rule called the One Hive rule. This rule states that the set of all pieces in the position – collectively called the *Hive* – must always be connected. Our reduction takes repeated advantage of this rule to make many pieces immobile, arranging them so that they would illegally separate the Hive if moved. While this idea is locally fruitful in enabling us to limit the players' options, it also demands special care in our handling of gadget connections, in order to ensure that the position remains legal under the One Hive rule while also keeping the desired pieces immobile. (See Section 3.8.)

We note that our proof does not provide any nontrivial upper bound on Hive's complexity. As remarked above, we consider it unlikely that Hive is in PSPACE. Since the rules do not contain an analogue of the 50-move rule in chess to limit potential game length, we conjecture that Hive is in fact EXPTIME-complete. However, we leave the resolution of this conjecture as a matter for future research (see Section 4).

1.1 Game rules

The official rules of Hive can be found in [13]. The game is played on a flat surface by two players, *Black* and *White*. Both players have eleven hex-shaped tile pieces at their disposal: three Ants, three Grasshoppers, two Beetles, two Spiders, and a Queen Bee. Each piece moves differently. The goal of the game is to surround the opponent's Queen Bee (hereafter usually "Queen"). Players take turns either placing a new piece on the table or moving one that is already in play. If a player is not able to place or move a piece, the player passes. The game ends when a Queen Bee is surrounded by other pieces, upon which the player whose Queen is surrounded loses the game.[2] If both players would lose simultaneously, the game is a draw. In the proof we only consider positions in which all pieces are on the table. Hence we omit the rules for placing pieces. We first discuss the pieces and their movement rules. Then we will explain the One Hive rule and Freedom To Move rule.

1.1.1 Queen Bee

The Queen Bee, or Queen, moves one space per turn. See Figure 1 for an example.

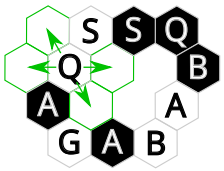


Figure 1 The white Queen can move to any one of the four indicated spaces.

1.1.2 Beetle

Like the Queen Bee, the Beetle moves one space per turn. Additionally, the Beetle can, unlike any other piece, move on top of another piece (either color). A piece with a Beetle on top of it is unable to move. Once on top of the Hive, the Beetle can move from tile to tile or drop off again. See Figure 2.

Figure 2 The Beetle can move to one of the two open spaces or climb on top of the Ant.

1.1.3 Ant

The Ant can move any number of spaces to any place it can reach, provided it follows the One Hive rule and Freedom To Move rule (Sections 1.1.6 and 1.1.7). See Figure 3.

[2] This wording implies that in a game of n-Hive with multiple Queens per player, a player loses when at least one of their Queens is surrounded. One could alternatively define a variant in which a player loses only when *all* their Queens are surrounded. Or, similarly, one could stipulate that a Queen is removed from the game once it is surrounded, and the goal of the game is then to remove all opposing Queens. (This suggestion is due to an anonymous referee.) While we do not treat such variants in this paper, we expect that they, too, are computationally hard, though they require a different reduction.

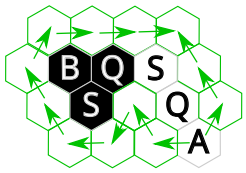


Figure 3 The Ant can move to any one of the indicated spaces this turn.

1.1.4 Grasshopper

The Grasshopper jumps over pieces. It starts its move by jumping on top of an adjacent piece, then keeps moving in the same direction until it comes off the Hive again. See Figure 4.

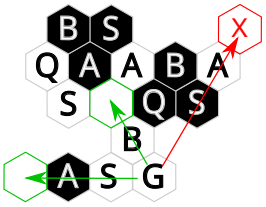


Figure 4 The Grasshopper can jump to one of the two indicated spaces. The space marked X is unavailable, as there is a blank spot between.

1.1.5 Spider

The Spider moves exactly three spaces per turn. During this movement it cannot go back and forth between two spaces. An example is in Figure 5.

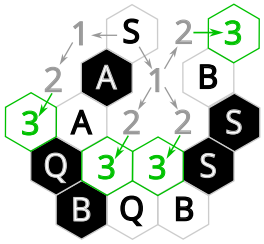


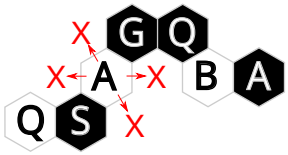
Figure 5 The white Spider can move to one of the four spaces marked 3. By the One Hive rule (see Section 1.1.6), it may not start its movement by going directly to the space on its right marked 2 and continue from there, as the Spider would lose touch with the Hive while in transit.

1.1.6 One Hive rule

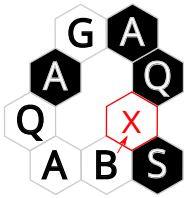
The pieces in play must at all times be connected. They can be seen to form one (big) Hive. Even *during* a turn – that is to say, while a piece is in transit sliding toward its destination – the Hive may not be disconnected and no piece may be left stranded. See Figures 5 and 6 for examples of movements forbidden under the One Hive rule.

1.1.7 Freedom To Move

The pieces move in a sliding fashion. So, if a piece is (almost) surrounded and cannot slide physically out of its spot without displacing other pieces, it is not allowed to move. Consider the white Queen in Figure 2. Recall that this piece is hex-shaped; it is not just the letter "Q"



(a) The white Ant is not able to move, since this would split the Hive in two.



(b) The Beetle may not move to the space marked X, as this would leave the Spider stranded while the Beetle is in transit.

Figure 6 Examples of movement restrictions under the One Hive rule.

written on a hex-shaped location. Since the entire hex is one piece, it is physically stuck and unable to slide out. See Figure 7 for further examples. Note, however, that Grasshoppers and Beetles provide exceptions to the Freedom To Move rule, as they are able to jump over and climb on top of the Hive, respectively.

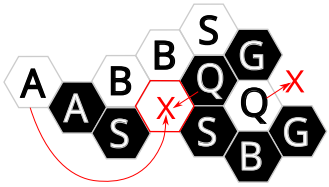


Figure 7 The black Queen is unable to move, as it cannot physically slide *out of* its space. The same holds for the white Queen. The white Ant is unable to move to the space marked X, as it cannot physically slide *into* that space.

1.2 Proof tools

Here we discuss some strategic concepts and other tools used within the proof.

1. **Win Threats** The most obvious strategic concept is that of a win threat. A piece threatening a win-in-one must be stopped. See Figure 8 for an example of a win threat being countered by locking down the dangerous piece using the One Hive rule (our next proof tool).
2. **Locking down pieces using the One Hive rule** In our proof it is useful to have a player immobilize an opposing piece by moving another piece away. Figure 8 shows an example in which Black is to move and must immobilize a dangerous white Grasshopper. In general, we call a piece *locked down,* or simply *locked,* when it is either unable to move (by the game rules) or not rational to move (because the piece's owner would lose the game soon after moving it).
3. **Freeing a piece** In our proof, it will be useful to construct positions containing large numbers of locked pieces, to keep the position simple and control the flow of the game. A piece that is locked down may sometimes become unlocked (or *free*) if the circumstances nearby change. When a locked-down piece has no prospect of ever becoming free in the future, we call it a *dead* piece (see below).
   All pieces in the proof begin locked down except for one. The position will be arranged so that moving one piece will free another, which in turn frees the next piece while locking the previous one, and so on. Such arrangements can result in a forced sequence, or at

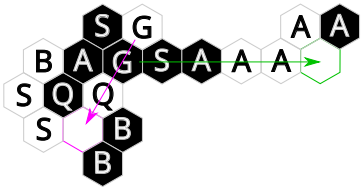


Figure 8 White's threat here is the win-in-one by the Grasshopper; moving it to the indicated space would complete the surrounding of the black Queen. Black can immobilize this white Grasshopper by moving the black Grasshopper away. Then the white Grasshopper is the only link between the left and right sides of the Hive, making it no longer allowed to move. As an aside, recall from Section 1.1.4 that even if it were still allowed to move, it would need the black Grasshopper in its original space anyway in order to jump over and reach the winning space.

least a sequence with a very limited number of choices. The freeing of a piece is usually done by giving the Hive a secondary path of connection so that the piece in question no longer needs to stay still just to keep the position legal under the One Hive rule. Examples occur in every gadget throughout the proof. In other cases, a piece is trapped by the Freedom to Move rule, until one of the pieces surrounding it is moved away.

4. **Selfmate hex** A hex is a *selfmate* hex if a piece entering would complete the surrounding of a Queen of its own color. With careful placement of selfmate hexes in our construction, we can restrict the movement of some pieces to help control the flow of play. Selfmate hexes are especially useful to restrict pieces whose available spaces are few and far apart, such as the Spider and Grasshopper. See Figure 9 for an example with a Grasshopper.

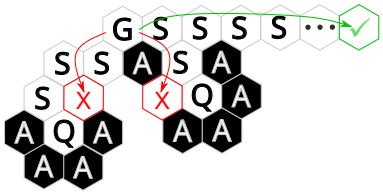


Figure 9 A position with two selfmate hexes for White, marked X. The Grasshopper cannot safely jump to these, as doing so surrounds a white Queen, losing immediately.

5. **Beetle tower** The Beetle is the one piece that can climb on top of the Hive, even on top of another Beetle that is already on top of the Hive. In fact, there is no limit to how high the Beetle can climb. It is thus possible to build a Beetle tower as high as desired, with any type of piece on the bottom.
6. **Dead piece** As mentioned, a dead piece is one that is either illegal or fatal to move. There are multiple ways to construct a dead piece, all with benefits and drawbacks. The list below is not exhaustive, but these are the dead pieces used in the proof.
   a. *Dead interior piece:* a Spider or Ant that is completely or almost completely surrounded, placed so it will remain this way until the end of the game. The color of this dead piece is irrelevant. Such pieces are labeled **DI** in diagrams.
   b. *Dead exterior piece:* a white Beetle on top of six black Beetles, with a white Queen on the bottom. (There is also a version with colors reversed.) This dead piece is usable in places where the top Beetle cannot meaningfully affect the game within one space's reach. If the white Beetle moves, the black Beetle underneath can immediately climb on top of it and lock it down. As moving the white Beetle made no immediate threat, Black can now surround the helpless white Queen underneath in at most five moves. Such pieces are labeled **DE** in diagrams, colored with the top Beetle's color.

To implement this dead piece, we must take special care in our proof construction to ensure that the white Beetle's movement indeed achieves nothing strategically relevant, such as freeing a dangerous piece that can win the game faster than the black Beetles can. If there would be any place where this condition fails, in principle it is possible to implement other dead piece designs – for example, a white Beetle covering a black Ant that, if uncovered, could immediately reach a white Queen to surround.

c. *Special dead piece:* a Beetle on top of three enemy Beetles and an enemy Queen below. Inherently, it is not necessarily dead, but in the context where it is used, the top Beetle is strategically unable to move. Such pieces, labeled **DS**, are discussed further in Section 3.2.

7. **Chains of locked-down pieces** Any number of pieces can be immobilized under the One Hive rule by placing them in a long chain with a dead exterior piece at the end.

2 Proof preliminaries

The material in this section is mostly standard (see, e.g., [10, Ch. 8]).

2.1 Quantified Boolean Formulas

▶ **Definition 1.** *The set* QBF (fully quantified Boolean formulas) *is the set of all syntactic strings* $Q_1 v_1 \dots Q_n v_n \psi$*, where* $Q_i \in \{\forall, \exists\}$*, each* v_i *is a Boolean variable, and* ψ *is a Boolean formula in conjunctive normal form containing only variables from* $\{v_1, \dots, v_n\}$*.*

▶ **Example 2.** The formula $\exists v_1 \forall v_2 \exists v_3 \forall v_4((v_1 \vee \neg v_2 \vee v_3) \wedge (v_2 \vee \neg v_4))$ is in QBF.

▶ **Definition 3.** *The truth-value of any* $Q_1 v_1 \dots Q_n v_n \psi \in$ QBF *is defined recursively by:*
$\exists v_1\ Q_2 v_2 \dots Q_n v_n \psi$ *is true if* $Q_2 v_2 \dots Q_n v_n \psi$ *is true for some possible truth-value for* v_1*.*
$\forall v_1\ Q_2 v_2 \dots Q_n v_n \psi$ *is true if* $Q_2 v_2 \dots Q_n v_n \psi$ *is true for each possible truth-value for* v_1*.*

▶ **Definition 4.** TQBF $= \{\langle \varphi \rangle \mid \varphi$ *is a true fully quantified Boolean formula*$\}$*.*

▶ **Theorem 5.** TQBF *is PSPACE-complete.*

2.1.1 Formula Game

Formula Game is a two-player game played between a so-called $\exists$-player and $\forall$-player with a given formula $Q_1 v_1 \dots Q_n v_n \psi \in$ QBF. Players take turns choosing truth-values for the variables $v_1, \dots, v_n$ in order, with the type of the i^{th} quantifier Q_i determining which player chooses the value of v_i. After v_n is valuated, the $\exists$-player wins if the formula is true; otherwise, the $\forall$-player wins. Alternatively, since ψ is in conjunctive normal form, we could equivalently continue the game for two more turns: one in which the $\forall$-player chooses a clause c, and one in which the $\exists$-player chooses a literal l in c. The $\exists$-player then wins if and only if l is true. In what follows, we assume this alternate version of the game.

▶ **Theorem 6.** *The problem* Formula Game *of determining whether the* $\exists$*-player has a winning Formula Game strategy for a given formula is PSPACE-complete.*

▶ Remark 7. It is straightforward to see that every formula in QBF is equivalent to one with the form $\exists v_1 \forall v_2 \exists v_3 \forall v_4 \dots \exists v_n \psi$ – that is, a formula whose quantifiers alternate and whose first and last are both $\exists$. We can easily construct it by adding quantifiers where needed that bind extra variables not occurring in ψ. We therefore have the following result.

▶ **Theorem 8.** Formula Game *and* TQBF *remain PSPACE-complete even when restricted to formulas of the form* $\exists v_1 \forall v_2 \exists v_3 \forall v_4 \dots \exists v_n \psi$*.*

2.1.2 Generalized Geography

▶ **Definition 9.** Generalized Geography *is a two-player game played on the nodes of a directed graph. A designated* start *node begins the game marked. Players then alternate turns by marking an unmarked node that has an incoming edge from the last marked node. A player to move with no unmarked nodes available (all available choices are already marked) loses the game.* Generalized Geography (GG) *is the corresponding decision problem.*

The proof of the theorem below is very well known (see, e.g., [10, Sec. 8.3]), but we present it here anyway as it will aid with the exposition of our main proof.

▶ **Theorem 10.** GG *is PSPACE-complete.*

Proof. We omit the proof that GG is in PSPACE. We give a polynomial-time reduction from Formula Game played on formulas with the form $\exists v_1 \forall v_2 \exists v_3 \forall v_4 \ldots \exists v_n \varphi$. Players 1 and 2 from the GG game respectively mimic the $\exists$- and $\forall$-players. We construct the formula's quantifier prefix by a chain of diamond gadgets (see Figure 10). Each diamond represents the choice that the $\exists$-player, in case of an existential quantifier, or $\forall$-player, in case of a universal quantifier, must make. The assignment of truth values for variables is now simulated as described in Figure 10.

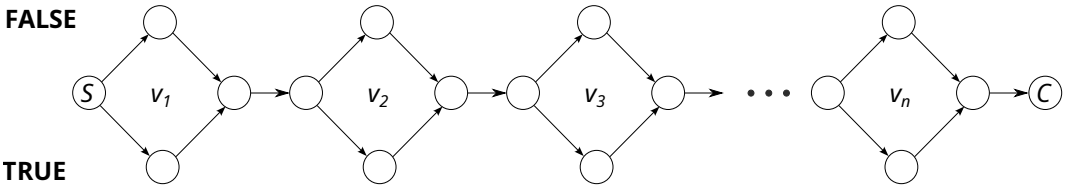


Figure 10 Beginning at the designated start node S, Player 1 has the first choice: marking the lower node sets v_1 to `true`, while marking the upper node sets v_1 to `false`. Both of these nodes have one outgoing edge, going to the right-hand node of the first diamond. Player 2 marks this node, after which Player 1 marks the first node of the next diamond. Player 2 can now choose to set v_2 to `true` by marking the lower node, or to `false` by marking the upper node. Play continues this way until Player 1 finally chooses for v_n (since the last quantifier is $\exists$).

After play has gone through the last diamond, we arrive at node C, with Player 2 to move. In the Formula Game being simulated, all variables have been assigned a value by now and the players take the two final turns. To simulate these, we need two help gadgets: a *clause chooser* gadget and *literal chooser* gadget. We construct the *clause chooser* gadget by representing each clause with a node. Node C is given an outgoing edge to each of these nodes. Player 2 can now choose to continue play in any one of the clauses.

We construct the *literal chooser* gadget by first adding a new node for each literal. We then add edges from each clause node to the nodes representing its literals. We also add an edge from each one of these literal nodes to the node corresponding to it within the diamond structures (i.e., the edges from nodes for literals v_i and $\neg v_i$ respectively go to the lower and upper nodes in the diamond of v_i). In response to Player 2's choice of clause made in the *clause chooser* gadget, Player 1 chooses a literal l from the chosen clause and marks its node.

This node has only one outgoing edge, leading to a diamond-structure node that was previously marked in the game if and only if one of the players selected it during the earlier variable assignment simulation. If it is indeed already marked (the variable's assigned value makes the chosen literal true), Player 2 now has to mark it again and loses. If it is not already marked, Player 2 can mark it and force Player 1 to lose, as the only available node from here is the already marked node on the right side of the diamond.

So, with this construction, Player 1 has a winning strategy in the GG game if and only if the ∃-player has a winning strategy in the Formula Game. ◀

▶ **Theorem 11** (S. Reisch [7]). Generalized Geography *on graphs with maximum degree 3, maximum indegree 2 and maximum outdegree 2 is PSPACE-complete.*

Proof. We can replace all nodes with indegree 3 by an equivalent chain of nodes with indegree 2 or less (see Figure 11). The treatment of nodes with outdegree 3 can be seen by reversing all arrows in the diagram. Indegree or outdegree above 3 can be handled similarly with a longer chain. The final case of indegree and outdegree exactly 2 (making total degree 4) is similarly straightforward. ◀

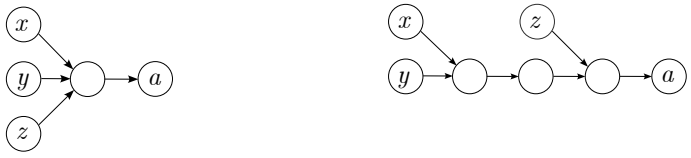


Figure 11 A node with indegree 3 (left) is replaced by an equivalent chain of nodes with indegree 2 or less (right).

Note that in PSPACE-hardness proofs of many other games, such as Go [5] and Hex [7], the reduction is done from a version of GG that is also assumed to be planar. For our proof, however, this assumption is not necessary.[3]

We now define the problem of Formula Game Geography, which is easily seen to be PSPACE-complete. In Section 3 we reduce this to Hive, establishing our main result.

▶ **Definition 12.** Formula Game Geography (FGG) *is* GG *played on a graph that is constructed by first applying the reduction from Theorem 10 to a* Formula Game *instance with the form specified in Theorem 8, then running the output of that through the reduction from Theorem 11. That is,* FGG *is* GG *played on a graph that simulates a* Formula Game *instance with the form* $\exists v_1 \forall v_2 \exists v_3 \forall v_4 \ldots \exists v_n \psi$ *and satisfies the degree properties of Theorem 11.*

▶ **Corollary 13.** FGG *is PSPACE-complete.*

3 Main proof

We now set out to reduce FGG to Hive. We begin with an overview of the proof structure and general idea. Then we introduce and explain the individual gadgets used in the proof. In Section 3.8, we describe how the gadgets are connected together. Section 3.9 details how we handle crossings. Finally, in Section 3.10 we make a few miscellaneous observations, including that the reduction can be carried out in polynomial time and that the constructed position is legally reachable under Hive rules.

[3] It has been well known since [7, Fig. 8a] that GG is PSPACE-complete even when the given graph is assumed to be planar. For many games, this assumption is indispensable in a hardness proof, as it is not possible to let two structures cross without interfering with each other. For that reason, proofs for such games require a way to deal with any crossing edges in GG. Planarity ensures that these crossings simply never occur. Hive has a planar nature as well, except for the fact that the Grasshopper (and technically, the Beetle) can travel over structures. We use this to our advantage to overcome the issue of crossing edges. See Section 3.9 for details.

3.1 Proof structure and general idea

Given is an FGG instance, which by definition corresponds to some quantified Boolean formula $\varphi = \exists v_1 \forall v_2 \exists v_3 \forall v_4 \ldots \exists v_n \psi$. That is, we are given a graph consisting of components that correspond to components of φ – i.e., to quantifiers, variables, clauses, etc. It will therefore be convenient at times to identify the graph components (for example, a diamond structure of the type seen in the proof of Theorem 10) with the corresponding components of φ (in this example, a quantified variable). In this way, our proof will somewhat resemble a proof by reduction directly from Formula Game, though we also take advantage of the degree properties provided in Theorem 11.

From the given graph that represents φ, we will construct a Hive position from which the game will proceed through a series of four stages, corresponding to the stages of play seen in the reduction from Formula Game to GG (Theorem 10). Black will represent the $\exists$-player (Player 1) and White will represent the $\forall$-player (Player 2).

First, in the *quantifier* stage, the players simulate taking turns assigning truth-values to the variables. Next, the $\forall$-player chooses a clause in the *clause chooser* stage. Then, in the *literal chooser* stage, the $\exists$-player chooses a literal l from that clause. Finally, the *tester* stage consists of a sequence of forced moves that simulate evaluating the chosen literal's assigned value. Black will win if and only if this literal l was originally assigned `true`.

A main feature in the design of our simulation is that all pieces are locked down at all times, except for one. So the player whose turn it is can only move one piece. Depending on the situation, the piece can move to just one space or to a choice between two spaces.

We regulate the game flow mostly by locking and unlocking pieces through repeated use of the One Hive rule. That this works in the case of any single gadget can be easily verified. However, it is not trivial that pieces that would be locked by the One Hive rule in individual gadgets remain locked when the gadgets are connected together. The details of this matter are treated in Section 3.8.

▶ **Theorem 14** (Main theorem). *Hive is PSPACE-hard.*

Proof. We reduce FGG to Hive. To simulate the components of an FGG graph, we construct gadgets for the following: choice of truth-value at a quantifier, $\forall$-player's choice of clause, $\exists$-player's choice of literal, truth-value testing of the chosen literal, and joining two edges. The first and fourth are handled by a single gadget, the *quantifier/tester*. The second and third are handled respectively by the *clause chooser* and *literal chooser* gadgets. The last is handled by the *join*. For connecting these gadgets together, we also have the following structures: *turn switcher*, *direction changer*, and *gap*. All constructions can be rotated in 60° increments when needed.

Before we present the gadgets, recall that dead exterior pieces must always be placed such that the top Beetle has no opportunity to create short-term threats in one step (such as by freeing a dangerous Ant that can win immediately). By inspection, it is straightforward to confirm that this condition is clearly met within each gadget. After Sections 3.8-3.9, it is also straightforward to verify that this condition holds when gadgets are connected together.

3.2 Quantifier/tester

This gadget is the heart of the reduction. It simulates both the choice a player has to make when assigning a value to a variable and the test that checks whether the variable's chosen value makes the later chosen literal true, which decides who wins. Figure 12 depicts the gadget for existentially quantified variables $v_1, v_3, v_5, \ldots, v_n$. For universally quantified

variables $v_2, v_4, \ldots, v_{n-1}$, the construction is the same with colors reversed. The gadget's main piece is the Spider S, which begins locked by the Freedom to Move rule but soon becomes freed to move either down or up. The choice of where it moves determines whether the simulated variable is assigned `true` or `false`.

All pieces in the diagram begin locked, except Grasshopper 1, which is located externally and is not part of this gadget. In the case of variables $v_2, \ldots, v_n$, Grasshopper 1 enters from the previous *quantifier/tester gadget.* In the case of v_1, Grasshopper 1 stands alone. When the simulation starts, this latter Grasshopper is the only free piece. In general, we use hexes with "…" to denote a lengthy chain of dead exterior pieces, with whatever is on the far side being understood as separate from the structure in the figure.

We now describe the flow of play in the *quantifier* stage. Without loss of generality, consider an existentially quantified variable v_i. Black Grasshopper 1 comes in from outside to land on space 1. This frees the white Grasshopper 2, which can only jump away to space 2 (the alternative is selfmate). The central black Spider S is now freed, as desired, but the white Grasshopper 2 is currently threatening to surround a black Queen. Since White's last move freed black Grasshopper 3, Black moves it to space 3, locking White's Grasshopper and preventing the win. This then frees white Spider 3, which moves to space 4 (elsewhere is selfmate), simultaneously locking Grasshopper 3 under the One Hive rule. Now it is Black's turn, with the black Spider S free to move. Note that white Spider 3 will never return to its previous space in the future, because black Grasshopper 3 would then become able to surround a white Queen.

Every piece is now locked down except S, whose movement will now determine the truth-value of v_i. This includes white Grasshoppers 6 and 7, which are part of another structure. Later, after the *quantifier, clause chooser,* and *literal chooser* stages have been completed, it is possible that white Grasshopper 6 or 7 will become unlocked and enter the present gadget in the *tester* stage. As we will see in Claim 16, this will occur if and only if the literal l chosen during the *literal chooser* stage is either v_i or $\neg v_i$, respectively.

Since the black Spider S is the only currently movable piece, the ∃-player chooses either to move S down (assigning v_i `true`) or up (assigning it `false`). Without loss of generality, suppose it moves down. This frees white Spider 1, which has no choice but to move to space 5 (the alternative is selfmate). This then frees black Grasshopper 4.

We claim now that Grasshopper 4 can only jump away to the next structure. If it jumps left instead, Black loses. To see this, recall that the special dead piece **DS** consists of a white Beetle on top of three black Beetles, with a black Queen underneath. This tower of pieces has two roles. In the first place, naturally, it serves as a dead piece. This is because the top white Beetle can make no valuable move in one turn, and if it ever would move, the black Beetle under it would be able to climb on top of it, leaving the next black Beetle free to find a white Queen in the next few moves to surround without any recourse. In the second place, the tower ensures that black Grasshopper 4 cannot jump over white Spider 1, lest the top white Beetle suddenly step left and surround the black Queen inside the tower. So the Grasshopper must move southwest to the next structure, unlocking whatever is there.

Having described the flow of play in the *quantifier* stage, we skip the *clause chooser* and *literal chooser* stages for the time being and proceed directly to the *tester* stage. Later we discuss the intermediate stages.

▶ **Lemma 15.** *If a literal $l \in \{v_i, \neg v_i\}$ is the one chosen during the* literal chooser *stage, the ∃-player (Black) has a winning strategy in the* tester *stage if and only if l is true under the variable assignment that was chosen during the* quantifier *stage.*

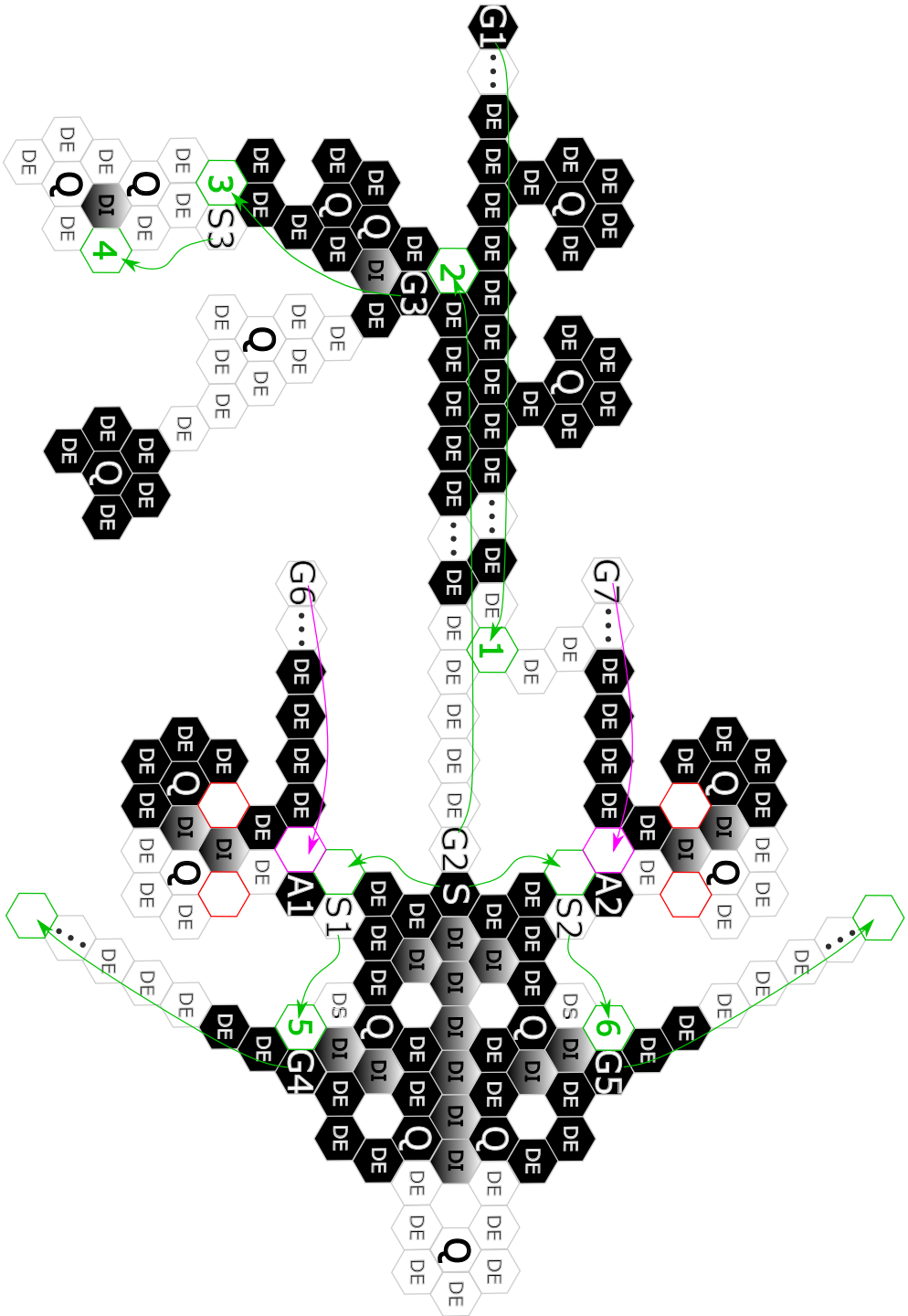


Figure 12 A *quantifier/tester* gadget for ∃-quantifiers. This gadget plays two roles in the FGG simulation. First, it simulates the choice given to the ∃-player for the quantification of one variable $v_i \in \{v_1, v_3, v_5, \dots v_n\}$. Later on, after the *literal chooser* stage has passed and the ∃-player has chosen a literal from the clause selected by the ∀-player, if that literal happens to be either v_i or $\neg v_i$ then this gadget simulates the *tester* that checks whether the assignment satisfies the chosen literal. Otherwise, the test is performed by a different *tester* gadget, corresponding to the appropriate variable. The ∃-player then wins if and only if the chosen literal is indeed satisfied (see Lemma 15).

Proof. The proof of this lemma relies on the following claim.

▷ Claim 16. In the *tester* stage, a Grasshopper enters into the lower (respectively, upper) *tester* entrance for a given ∃-quantified variable v_i if and only if l is v_i (respectively, $\neg v_i$). On the other hand, a Grasshopper enters into the lower (respectively, upper) *tester* entrance for a given ∀-quantified variable v_i if and only if l is $\neg v_i$ (respectively, v_i).

We verify this claim later, in Section 3.10. For now, assume the claim is true and consider first the case in which variable v_i was quantified existentially. Suppose the chosen literal l is indeed either v_i or $\neg v_i$, so white Grasshopper 6 or 7, respectively, has therefore entered the *tester*. We now show that the ∃-player (Black) has a winning strategy if and only if l is true under the value assigned earlier to v_i in the *quantifier* stage. There are four subcases to consider.

1. Suppose v_i was assigned `true` at the *quantifier* stage and the ∃-player chose $l = v_i$ at the *literal chooser* stage. So Spider 1 is on space 5 and the black Grasshopper 4 is out of the picture. As the *tester* stage begins, white Grasshopper 6 enters the gadget on the indicated space (by Claim 16), freeing black Ant 1 to surround the nearby white Queen by moving two hexes down. Thus the ∃-player has a winning strategy.
2. Suppose v_i was assigned `true` at the *quantifier* stage and the ∃-player chose $l = \neg v_1$ at the *literal chooser* stage. In this case, Spider 1 and Grasshopper 4 are placed as in Case 1, but white Grasshopper 7 enters the gadget rather than 6. When it does, no piece is freed. So Black has no way to stop White from winning on the next turn by jumping Grasshopper 7 northwest, or just punishing Black for moving one of the dead pieces. Thus the ∀-player has a winning strategy, as desired.
3. The case in which v_i was assigned `false` and $\neg v_i$ was chosen is symmetric to Case 1.
4. The case in which v_i was assigned `false` and v_i was chosen is symmetric to Case 2.

We now consider universally quantified variables. Recall that here the colors of pieces are reversed in the gadget. So the Queens being surrounded in the four subcases are of the opposite color, reversing the outcomes. However, Claim 16 provides that the Grasshopper arriving in the *tester* stage enters through the opposite entrance from before, reversing the outcomes again. So the four subcases again produce the correct winner. ◀

3.3 Join

The *join* gadget simulates an FGG node with indegree 2 and outdegree 1, joining two other structures together. We require these after each *quantifier* gadget in order to combine the two outgoing Grasshopper paths into one, which then feeds into the next *quantifier* (or, in the case of the final *quantifier,* the first *clause chooser* gadget). We also need *join* gadgets when a literal appears in more than one clause, so that its multiple occurrences can be combined and fed into the appropriate *tester*. This is because only one Grasshopper can enter a *tester* gadget in the *tester* stage, so if there are multiple Grasshopper representatives of the same literal coming from different clauses in ψ, their paths must be joined beforehand.

Figure 13 depicts the *join* gadget. To start, either Grasshopper 1, coming from another structure northwest, or Grasshopper 2, coming in from southwest, lands on the corresponding space marked 1. This frees the black Grasshopper, which can only jump safely to space 2 in the east. This then frees Grasshopper 3, which continues further east to the next structure.

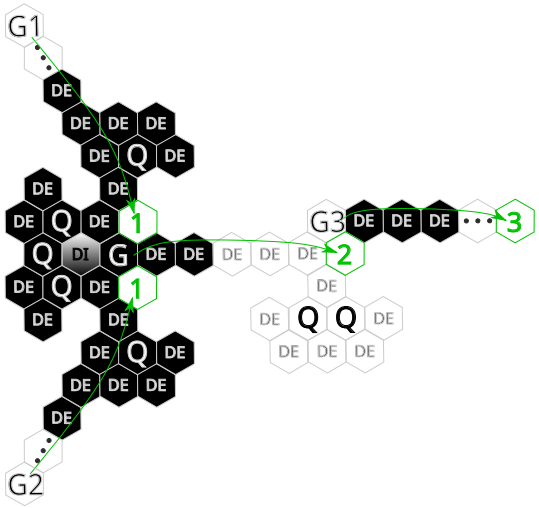


Figure 13 A white-to-black *join* gadget. (Black-to-white *joins* are identical with colors reversed.)

3.4 Turn switcher

This gadget's only function is to switch turns. We have it available for whenever a black Grasshopper is exiting one structure while a white Grasshopper is needed to enter the next, or vice versa. This occurs, e.g., between two *clause choosers* (see Section 3.5). Figure 14 depicts the black-to-white version. Here, the incoming black Grasshopper lands on space 1. This frees the white Grasshopper, which has no safe choice but to continue the same direction.

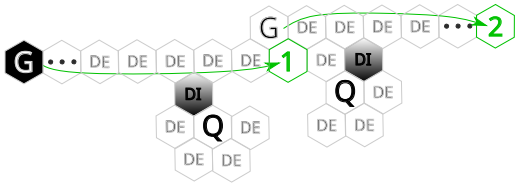


Figure 14 A white-to-black *turn switcher*. (The black-to-white are identical with colors reversed.)

3.5 Clause/literal chooser

The *clause chooser* gadget, shown in Figure 15, functions as an FGG node with indegree 1 and outdegree 2. The black Grasshopper enters from the left and lands on space 1. This frees the white Grasshopper, which can jump to one of the two spaces marked 2. This represents White's choice between two clauses. The *literal chooser* gadget, constructed the same with colors reversed, represents the similar choice of the ∃-player (Black) between two literals.

Chaining several *clause choosers* together in the manner described in Theorem 11, with *turn switchers* in between to change back colors, effectuates giving White a choice over many clauses. Likewise for Black and literals in the next stage. If formula ψ has $q \in \mathbb{N}$ clauses and $r \in \mathbb{N}$ literals, the reduction requires $q - 1$ *clause chooser* and $r - 1$ *literal chooser* gadgets.

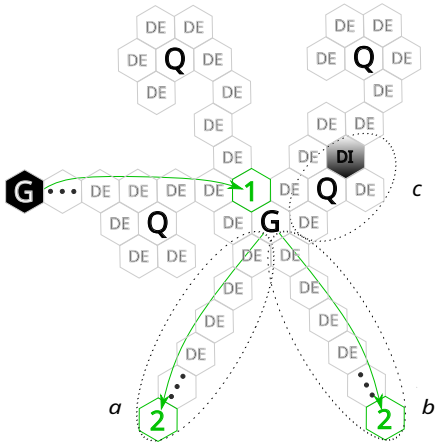


Figure 15 Combined diagram for *clause chooser, literal chooser, and direction changer* gadgets. The four pieces in group c are not needed in a *clause chooser*. Reversing all piece colors of a *clause chooser* yields a *literal chooser*. Removing group a from the diagram yields a black-to-white 60° clockwise *direction changer*. For this, group c is necessary because its absence would make the white Grasshopper's jump illegal under the One Hive rule. Finally, a black-to-white 120° clockwise *direction changer* is constructed as a *clause chooser* without group b (here, again, c is not needed).

3.6 Direction changers

These gadgets are functionally *turn switchers* that change the direction of the game's flow of play by 60 or 120 degrees. Figure 15 shows either a 60° or 120° black-to-white clockwise *direction changer*, depending on which piece groups are included. Piece colors can naturally be reversed to produce white-to-black *direction changers*. Play proceeds as in the *clause* and *literal choosers*, except with no choice to make for the outgoing Grasshopper's direction.

Direction changers are needed in many places, such as between the *quantifier* gadgets, whose outgoing Grasshoppers always move northwest or southwest and must be redirected so they can enter a *join* (see Figure 16).

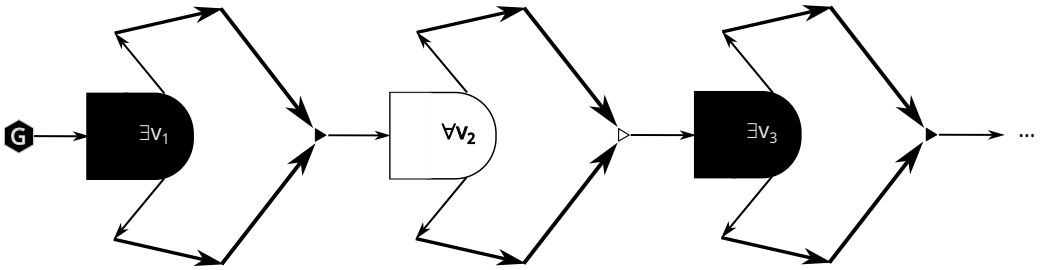


Figure 16 Sketch of *quantifier* gadgets connected together. G represents the starting Grasshopper. The arrows directed northwest and southwest out of the *quantifiers* represent the chains of dead pieces over which the outgoing Grasshoppers may jump. The points where one arrow touches another contain 60° or 120° *direction changers*. The small triangles between *quantifiers* denote *join* gadgets.

3.7 Gap

To ensure that all pieces locked by the One Hive rule are truly locked, it is necessary at some places to disconnect two structures by including a *gap*. There are three types of places where we need a *gap*. First, a *gap* is required in one of the two paths between two *quantifier*

gadgets, to ensure that the Hive is connected in only one place. Second, we need a *gap* wherever a *literal chooser* (or a *join* that has connected two *literal choosers*) would meet a *tester* gadget. Third, a *gap* is employed whenever a crossing occurs. The first two uses of *gaps* are discussed in Section 3.8. The third is treated in Section 3.9.

In Figure 17, white Grasshopper 1 comes in from the previous structure to land on space 1. This frees the black Spider, which can pass over the gap by following the arrow and move to space 2 (the alternative is selfmate). This move frees white Grasshopper 2, which jumps away to the next structure (moving elsewhere loses; note that jumping over the Spider would land on a space surrounding a **DE** beetle tower that contains a white Queen).

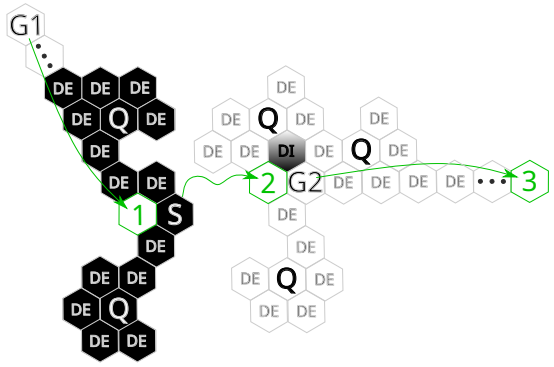


Figure 17 A white *gap* gadget to disconnect parts of the Hive. Colors may be reversed.

3.8 Connecting the gadgets

In this section, we show how the gadgets are connected to simulate an FGG graph, while ensuring that every gadget is linked to every other gadget by exactly one path of adjacent pieces. There must be at least one path because of the One Hive rule, and there must be at most one path to make sure pieces purposely locked by the One Hive rule are indeed locked.

When the gadgets are connected, the full board configuration of the FGG simulation can be seen as beginning with a long horizontal line of *quantifier/tester* gadgets – see Figure 16. Play flows from west to east. A structure connects to another through a single chain of dead pieces over which an outgoing Grasshopper may jump. Each of the *quantifier* gadgets is linked to *direction changers* that eventually feed into a *join*. A *gap* is also included among the *direction changers* in the lower of the two paths coming from each *quantifier*. The *join* connects to the next *quantifier* gadget, except for the *join* coming from the rightmost one. This last *join* sends its output Grasshopper into a tree of further structures that extend above and below the horizontal line. The tree consists of the *clauser chooser* gadgets, arranged as described in Section 3.5 (based on the construction from Theorem 11), which then further branch into *literal chooser* gadgets that are themselves arranged the same way. That is, the *clause choosers* together form a chain of binary choices that collectively simulate a single choice of one clause among many, each of which in turn branches into a chain of literal choosers that do the same for literals in that clause. So, for each combination of clause c_i and literal $l_j \in c_i$, there is one path. At the end of each of these paths, a *gap* is inserted.

The leaves coming out of the *literal choosers* feed into paths heading back west via *direction changers*, each leading toward the appropriate *quantifier/tester* for the literal in question. Paths from *literal choosers* corresponding to the same literal in different clauses

are connected through *join* gadgets before reaching the destination *tester*. The set of *join* gadgets that feed into any given *tester* entrance form a sort of branching tree themselves, if viewed in the direction opposite to the flow of play.

When connecting the path from a literal l to the corresponding *tester*, if $l \in \{v_i, \neg v_i\}$ for $i \in \{1, 3, 5, \ldots, n\}$ (recall that this means the variable v_i is existentially quantified), we connect it via the lower (true) Grasshopper entrance if and only if $l = v_i$. If $l \in \{v_i, \neg v_i\}$ for $i \in \{2, 4, 6, \ldots, n-1\}$ (that is, the variable is universally quantified), we do the reverse, connecting instead via the lower (true) entrance if and only if $l = \neg v_i$. This is to ensure that Claim 16 holds.

With the construction described in this section, we see that gadgets are connected in such a way as to satisfy the One Hive rule at all times, while ensuring that pieces intended to be locked down by the rule remain duly locked. The only consideration not yet covered is the possible intersection of two paths, which we treat in Section 3.9.

3.9 Crossings

Recall from fn. 3 that we do not assume planarity of the input GG graph for our reduction. In this section, we present a way to deal with crossings in Hive directly.

When enough space is taken between the gadgets, no crossings occur in the *clause chooser* and *literal chooser* areas. In the area after the incorporated *gaps*, however, when the literals from the clauses are directed to the corresponding *testers*, crossings may occur for some formulas. Recall that some of the paths out of *literal choosers* combine through *joins* before they are directed to the corresponding *tester*. In this process, too, crossings might occur.

To ensure that these crossings cause no problems, first we describe how the crossings themselves are implemented. Then, we show why they do not free any pieces we need to keep locked under the One Hive rule.

The crossing itself is easily constructed. At the point of intersection, each of the two paths is a chain of dead pieces over which a Grasshopper can jump from one gadget into the next. So, to implement the crossing we can simply allow the two chains to cross each other, with a single piece at the intersection serving as a common element to both chains. The Grasshoppers of both chains are still able to jump over their respective chains in the same manner as without a crossing.

To ensure pieces in the Hive locked by the One Hive rule remain locked, we simply add a *gap* for each crossing, placed in one of the crossing paths between the point of intersection and the *tester* gadget. We now argue that this approach suffices.

▶ **Lemma 17.** *Even with crossings, all gadgets still connect via exactly one path.*

Proof. Let A be a path to a *tester* t_a from one of the incorporated *gaps*. Call this *gap* g_a. Let B be a path to a *tester* t_b from another *gap* g_b. Note that all *testers* are connected along the horizontal line of *quantifier/tester* gadgets, so they are a single structure S for the purposes of the One Hive rule.

Suppose A and B cross. Call the shared hex c. By our construction, one of the paths (without loss, path A) is constructed as normal between the crossing point c and the *tester* t_a, while the other, B, has an additional *gap* between c and t_b. Were it not for this added *gap*, the reduction would break, as many pieces throughout the Hive would then be unhampered by the One Hive rule – namely, those between c and t_a, those between c and t_b, and a large number of pieces within gadgets between t_a and t_b (inclusive) in the horizontal line. This is because anything connected to the Hive through these pieces would have a secondary path of connection on top of the regular one, thanks to the connection between t_a and t_b through c.

However, because of the added *gap* in B between c and t_b, we find the following. First, each of the pieces in the fragment of B running between this gap and t_b has only one path connecting it to the Hive – namely, via the fragment itself. Second, each piece in B between c and the gap has only one path as well – namely, via the fragment of A from c to t_a. The latter fragment is also the only connecting path to S for pieces between g_a and c and for pieces between g_b and c. All other gadgets are connected to the Hive as though A and B never crossed.

Since all gadgets connect by exactly one path, the lemma is proved. ◀

3.10 Completing the proof

Having laid out the reduction, we now make a few final observations. First, we are now finally equipped to verify Claim 16. By inspection of the constructed position as a whole, we see that a Grasshopper indeed eventually enters an existential *quantifier/tester* gadget on its northern (respectively, southern) side if and only if the corresponding literal $\neg v_i$ (respectively, v_i) is the literal l chosen in the *literal chooser* stage. We similarly see the reverse for the universal case.

Next, we observe that the starting position of our FGG simulation is legally reachable from the starting state of the game. Recall that Hive always begins with an empty playing field. Our players will take turns placing pieces and moving them, starting with the first *quantifier* and working their way through all the gadgets. Many pieces can be placed exactly where they need to end up in the position and never moved. Others can be placed nearby and then moved as needed. Given the pieces' versatility, it is easy to see how the starting position of the simulation can be reached. The Beetles are useful pieces to assist another piece in reaching its spot (for instance, by temporarily making an extra connection to free a piece locked down by the One Hive rule), as they can reach every possible space and are always able to return to their designated location. Any excess pieces in the players' supply are placed on the board and moved into a chain, attached to a gadget lying on the border of the Hive in such a way that it does not influence the game, with a dead exterior piece at the end of the chain.

Finally, we note that the size of the construction is polynomial relative to the size of the FGG graph. This is straightforward, as each gadget consists of a constant number of pieces, and each quantifier, clause, and literal requires a limited number of gadgets to be simulated.

This concludes the proof of the main theorem. ◀

4 Conclusion

We've shown that the winning strategy decision problem for Hive generalized to arbitrary numbers of pieces is PSPACE-hard by reduction from Formula Game, with the aid of Formula Game Geography, a variant of Generalized Geography played on graphs that simulate Formula Game instances and have maximum indegree 2, outdegree 2, and total degree 3. Our proof has followed the broad reduction strategy of [5], but with peculiarities owing to the hexagonal geometry of Hive and unique challenges imposed by the One Hive rule. At the same time, the One Hive rule has also been a great advantage in constructing the gadgets, by enabling us to prevent the majority of pieces from moving and control the flow of the game.

The obvious recommendation for future research is to determine the exact upper and lower bounds of Hive's complexity. As mentioned in Section 1, Hive has no correlate of the 50-move rule in chess; in fact, outside of the simultaneous surrounding of Queens, Hive has

no official way for games to end in a draw other than by player agreement. Accordingly, we have conjectured that Hive is EXPTIME-complete, a proof of which we hope will be found in future work.

Another recommendation, as mentioned in Section 1, is to consider a generalization of Hive in which the players have only one Queen each. This would align the generalized game more closely to the spirit of the original game, much as [2] and [11] maintain one King per player in generalized chess. Our reduction makes essential use of the presence of multiple Queens by employing them to create selfmate hexes, which we use to prevent pieces from traveling in certain directions. Without this resource, an alternative may be to create "indirect" selfmate hexes by placing Ants in such a way that they are locked down by the One Hive rule until an enemy piece moves to the critical hex (what would have been a selfmate hex in our current reduction). That Ant would then become free to travel any distance and, if the construction is successful, reach the enemy Queen to surround it. Another approach is to make use of repeated game-winning threats to force play, as in [5], rather than relying on the One Hive rule and selfmate hexes.

Along similar lines, as noted in fn. 2, even the assumption of multiple Queens allows for more than one possible generalization of the standard game. We expect that a variant of n-Hive in which a player must surround all enemy Queens to win, rather than just one, is at least as hard as other variants. However, the gadget constructions employed in the present paper are not suited to show this, given their reliance on selfmate hexes.

Finally, we mention that as an addition to the basic game, the publishers of Hive have released three other pieces in expansions between 2007 and 2013: the Ladybug, Mosquito, and Pillbug. In principle it could be investigated whether the inclusion of these additional pieces brings any increase to the game's computational complexity. We conjecture that it does not, as the complexity of Hive appears to lie to a greater extent in the geometry of the game and its abstract rules, such as the One Hive rule, rather than the exact movement of the pieces. Nevertheless, it cannot currently be ruled out.

More interestingly, perhaps, we wonder whether a hardness proof can be carried out for Hive with a *smaller* piece set than the standard one. While some pieces seem indispensable – for example, Beetles – the Ants are seldom used and appear replaceable in our reduction, and we further ask whether a version of Hive without access to Spiders might also prove still to be computationally hard.

References

1 Daniël Andel. On the complexity of Hive. Bachelor's thesis, Utrecht University, 2020.

2 Aviezri S. Fraenkel and David Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n. *J. Comb. Theory, Ser. A*, 31(2):199–214, 1981. doi:10.1016/0097-3165(81)90016-9.

3 Robert Hearn. Amazons, Konane, and Cross Purposes are PSPACE-complete. *Games of No Chance 3*, 56, March 2009. doi:10.1017/CBO9780511807251.015.

4 Robert A. Hearn and Erik D. Demaine. *Games, puzzles and computation.* A K Peters, 2009.

5 David Lichtenstein and Michael Sipser. Go is polynomial-space hard. *Journal of the ACM*, 27(2):393–401, April 1980. doi:10.1145/322186.322201.

6 Stefan Reisch. Gobang ist PSPACE-vollständig. *Acta Informatica*, 13:59–66, 1980. doi:10.1007/BF00288536.

7 Stefan Reisch. Hex ist PSPACE-vollständig. *Acta Informatica*, 15:167–191, 1981. doi:10.1007/BF00288964.

8 Benjamin G. Rin and Atze Schipper. Arimaa is PSPACE-Hard. In Andrei Z. Broder and Tami Tamir, editors, *12th International Conference on Fun with Algorithms, FUN 2024, June 4-8, 2024, Island of La Maddalena, Sardinia, Italy*, volume 291 of *LIPIcs*, pages 27:1–27:24. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.FUN.2024.27.

9 J. M. Robson. N by N checkers is Exptime complete. *SIAM Journal on Computing*, 13(2):252–267, 1984. doi:10.1137/0213018.

10 Michael Sipser. *Introduction to the Theory of Computation*. CENGAGE Learning, 3 edition, 2013. International ed.

11 James A. Storer. On the complexity of chess. *Journal of Computer and System Sciences*, 27(1):77–100, 1983. doi:10.1016/0022-0000(83)90030-2.

12 Giovanni Viglietta. Lemmings is PSPACE-complete. *Theor. Comput. Sci.*, 586:120–134, 2015. doi:10.1016/J.TCS.2015.01.055.

13 John Yianni. Hive: a game buzzing with possibilities. https://www.gen42.com/wp-content/uploads/Hive-rules.pdf, 2010. Accessed: 2026-03-03.