Utrecht University

Bachelor Thesis

On the complexity of Hive

Author:
Daniël ANDEL

Supervisor:
Dr. Benjamin RIN

Artificial Intelligence
Faculty of Humanities

May, 2020

Abstract

It is shown that for an arbitrary position of a Hive game where both players have the same set of N pieces it is PSPACE-hard to determine whether one of the players has a winning strategy. The proof is done by reducing the known PSPACE-complete set of true quantified boolean formulas to a game concerning these formulas, then to the game generalised geography, then to a version of that game with the restriction of having only nodes with maximum degree 3, and finally to generalised Hive. This thesis includes a short introduction to the subject of computational complexity.

Contents

I really think that Hive represents a major achievement in gaming. If we look at the pinnacle of games in its Genre, it would be fair to say that nothing has come close to games like Chess and Go for Hundreds of years.

– Neil Thompson, board game expert with over 500 reviews on games on boardgamegeek.com

1 Introduction

The ancient games of chess and GO have been the subject of countless academic studies. In the early eighties, generalised versions of these games were proved to be PSPACE-hard (GO) [5] and PSPACE-complete (chess) [11]. Since then, the complexity of numerous other games has been determined. Recent examples of PSPACE-complete problems are generalisations of the video game Lemmings [12] and the ancient Hawaiian board game Konane [4]. This thesis presents a proof that the problem of determining whether one of the players has a winning strategy in an arbitrary N-Hive position is PSPACE-hard. Hive is a popular tabletop game, designed by John Yianni and published by Gen42 Games in 2001. Whilst containing only 22 pieces and having a small set of rules, it is a very complex and abstract game according to board game enthusiasts. However, Hive has a finite number of possible game positions. Hence it is in principle possible to construct a large look-up table which unveils a winning strategy. In the interest of investigating the asymptotic computational complexity it is necessary to generalise the game. With board games, this is usually done by generalising the board to an NxN board. Hive does not have a board; the playing field is determined by the placement of pieces. We generalise Hive to N-Hive by stating that each player has N pieces to play with, rather than 11. We make no further adjustments to the rules. To preserve the spirit of the game, we demand that both players have the same set of pieces. From here on we use the word Hive for N-Hive, and when the regular game is meant we state this.

In this chapter we will give an introduction to the field of computational complexity theory, a general description of the proof strategy, the rules of regular Hive and some strategic concepts for constructing the proof. The next chapter contains the actual proof that Hive is PSPACE-hard. Finally, this thesis is concluded by recommendations for future research.

1.1 Introduction

In complexity theory, the central question is:

What makes some problems computationally hard and others easy?

Despite over 40 years of intensive research, we still do not know the answer to this question. However, researchers did develop a system to distinguish between computational problems on the matter of difficulty. This elegent scheme for classifying problems according to their computational difficulty is one of the biggest achievements in complexity theory [9, p. 2]. Some of the more important classes are polynomial time (P), nondeterministic polynomial time (NP), exponential time (EXPTIME), polynomial space (PSPACE) and nondeterministic polynomial space (NPSPACE). The class P, for instance, contains all decision problems that can be solved in polynomial time. In this context, solved means that there exists an algorithm that can determine the solution of every instance of the problem in polynomial time. Polynomial-time solvability is regarded the standard for being considered "feasibly" or "efficiently" solvable (known as the Cobham-Edmonds thesis) [1, 2]. One of the most important unanswered questions in mathematics and theoretical computer science is whether all problems in NP are also in P, also known as the P versus NP problem. For *space* complexity, this question has been answered. In 1970, Walter J. Savitch showed that every nondeterministic algorithm that runs in $f(n)$ space can be converted into a deterministic algorithm that runs in maximum $f^2(n)$ space [7]. The square of a polynomial is still a polynomial. Therefore, all problems in NPSPACE are also in PSPACE.

So far, we know that the complexity classes mentioned before can be placed in the following order:

$$\text{P} \subseteq \text{NP} \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME},$$

It could be that one or more of these inclusions are actually equalities, as it is the case with PSPACE and NPSPACE. A simulation like the one from Savitch's theorem would be required to prove the merging of two classes. However, most researchers think that these inclusions are proper. A consequence of the time hierarchy theorem [3], which is beyond the scope of this thesis, is the fact that P $\subset$ EXPTIME. Hence, at least one of these inclusions has to be proper. Up until now we are not able to prove which.

A *complete* problem is seen as one of the most difficult problems in its class. It is most difficult because the other problems in the class can be reduced to it. For a problem to be PSPACE-complete, it has to be both PSPACE-hard and in PSPACE. A problem is PSPACE-hard if every problem in PSPACE is polynomial time reducible to it. As such, it is sufficient to show that a PSPACE-complete problem is polynomial time reducible to it. The reduction has to be polynomial time and not polynomial space because the reduction must be *easy* relative to the complexity of the problems in the class. If the reduction is not easy to compute, an easy solution to the problem would not necessarily provide an easy solution to the problems reducing to it [9, p. 337-338].

1.2 General description of proof strategy

We will prove that Hive is PSPACE-hard by giving a reduction from a PSPACE-complete set, true quantified boolean formulas (TQBF), to the problem of deciding whether one of the players has a winning strategy in an arbitrary N-Hive position. The proof proceeds in a similar manner as the proofs for other PSPACE-hard problems, namely via a series of steps. We use the GO paper as a blueprint [5]. First we reduce TQBF to the TQBF game, then to a game called generalised geography (GG), then to a version with bipartite graphs with maximum degree 3, and finally to Hive.

In the final reduction to Hive we make repeated use of the One Hive rule to force moves. The simulation is designed in such a way that the player whose turn it is has only one reasonable move, starting with the first Quantifier gadget and maintaining the flow throughout the whole formula.

1.3 Rules of regular Hive

The game is played on a flat surface by two players, *Black* and *White*. Both players have 11 hex-shaped pieces at their disposal: 3 Ants, 3 Spiders, 2 Beetles, 2 Grasshoppers and a Queen Bee. Just like chess pieces, they all behave in their own way. The goal of the game is to surround the opponent's Queen Bee. During a turn, a player can either place a new piece on the table or move one that is already in play. Turns alternate between the two players. If a player is not able to place or move a piece, he or she passes. The game ends when one Queen Bee is surrounded by other pieces. The player whose Queen is surrounded loses the game, unless the piece that surrounded it also completes the surrounding of the other player's Queen Bee. In that case the game is drawn. In the proof we only consider positions where all pieces are on the table. Hence, the rules for placing pieces are left out here. First, the pieces and their movement are discussed. Then, the One Hive rule is explained and thereafter the Freedom To Move rule.

1.3.1 Queen Bee

The Queen Bee moves one space per turn. See figure 1 for an example.

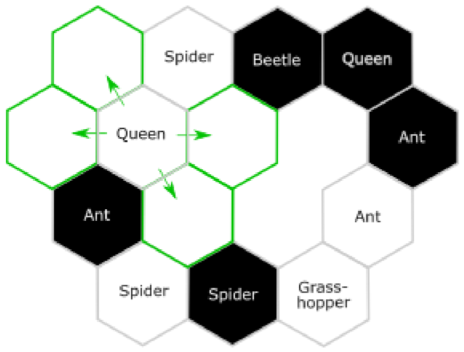

Figure 1: The white Queen can move to any one of the four green spaces

1.3.2 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. 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 for an example.

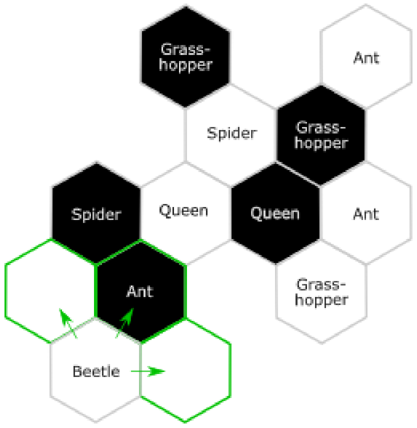

Figure 2: The white Beetle can move to one of the two green spaces, or climb on top of the black Ant

1.3.3 Ant

In a turn, the Ant can move to any place it can get to, whilst taking into account the One Hive rule and the Freedom To Move rule (as described in 1.3.6 and 1.3.7). See figure 3 for an example.

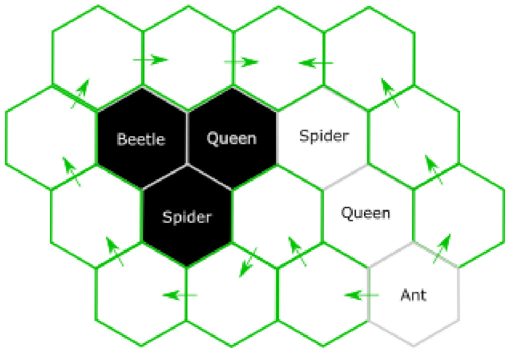

Figure 3: The white Ant can move to any one of the green spaces this turn

1.3.4 Spider

The Spider moves exactly three spaces per turn. During this movement it cannot go back and forth between two spaces. See figure 4 for an example.

1.3.5 Grasshopper

The Grasshopper moves in a jumping manner. It starts its move by jumping on top of a piece that it is adjacent to, and keeps moving in the same direction until it comes off the Hive again. This is where the Grasshopper lands, and where it finishes its move. For an example see figure 5.

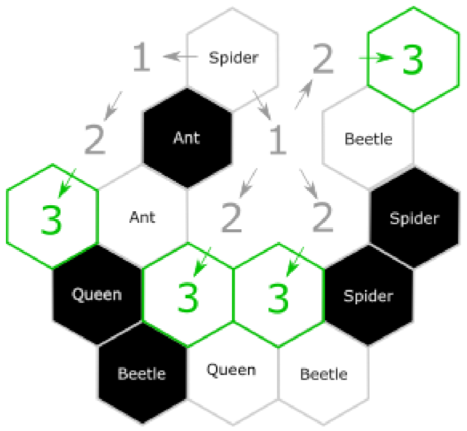

Figure 4: The white Spider can move to one of the four green spaces marked with a 3. It is not allowed to start its movement by going to the space on the right marked with a 2 and move on from there, as the Spider would lose touch with the Hive *during this movement*

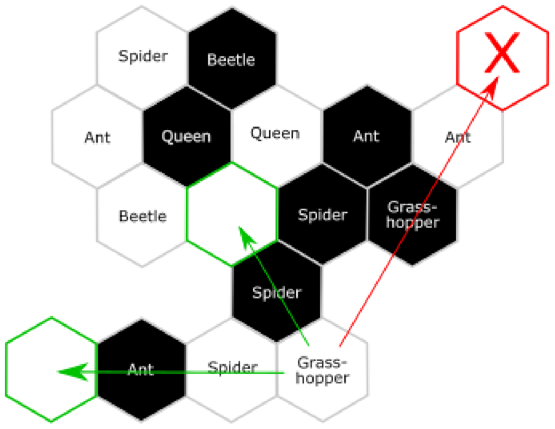

Figure 5: The white Grasshopper can jump to one of the two green spaces

1.3.6 One Hive rule

The pieces in play must at all times be linked to each other. They can be seen to form one (big) hive. Even *during* a turn, the Hive may not be broken in two and no piece may be left stranded. See figure 6 and 7 for two examples of a piece that is not allowed to move because of the One Hive rule.

1.3.7 Freedom To Move

The pieces move in a sliding fashion. So, if a piece is (almost) surrounded and is not able to physically slide out of its spot, it is not allowed to move. Recall that the pieces are hex-shaped. It is not just the word "Queen" written on a hex-shaped location. The entire hex is one piece, and that is why it is stuck and unable to move. The Grasshopper and the Beetle form exceptions to the Freedom To Move rule, as they are able to jump over and climb on top of the Hive respectively. See figure 8 for some examples.

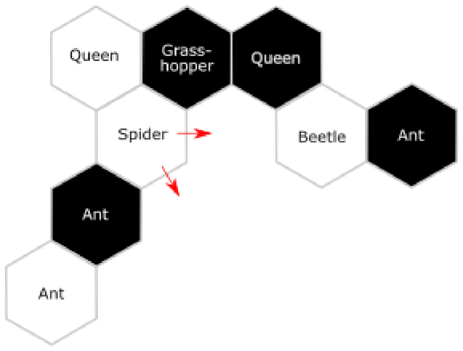

Figure 6: The white Spider is not allowed to move, because that would split the Hive in two

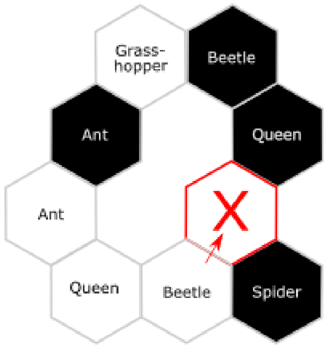

Figure 7: The white Beetle is not allowed to move to the space marked with an X, because that would leave the black Spider stranded *during the turn*

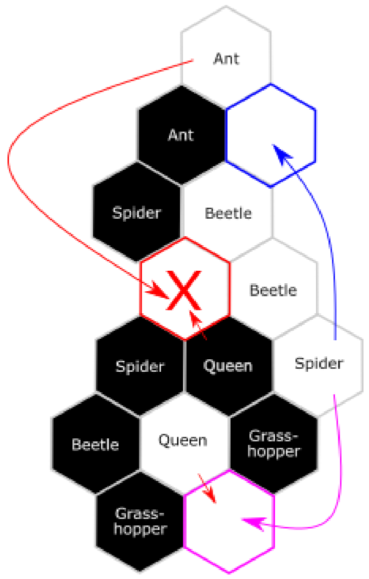

Figure 8: The black Queen is not allowed to move, as it cannot physically slide *out* of its space. The same goes for the white Queen. The white Ant is not allowed to move to the red space, as it cannot physically slide *into* that space. The pink space is a suicide hex, as the white Spider would trap its own Queen there. The blue space is an indirect suicide hex, since occupying it would free up the dangerous black ant

1.4 Strategic concepts

In this subsection the strategic concepts used for the gadgets are discussed. By strategic concepts we thus mean structures that are useful in the proof, not strategic concepts while playing the game. Nevertheless, some can be used in a regular game. Examples are the trapping (section 1.4.3) or freeing (section 1.4.4) of pieces using the One Hive rule.

1.4.1 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 9 for an example of a win threat being countered through trapping the dangerous piece by making use of the One Hive rule.

1.4.2 Suicide hexes

Restricting the movement of a piece by surrounding it by suicide hexes is a useful strategy in constructing the proof. A suicide hex is a space where a piece would complete the surrounding of a Queen of the same colour. It is especially easy to implement with a piece that has the characteristic of only having a few spaces it can move to, and those spaces also being far apart. Such is the case with the Spider and the Grasshopper. See figure 8 for an example with a Spider.

1.4.3 Trap pieces using One Hive rule

To keep the flow of the game going, it is convenient to trap an incoming piece by moving another piece away. See figure 9 for an example of this. The Grasshopper that is being trapped has just moved into its space.

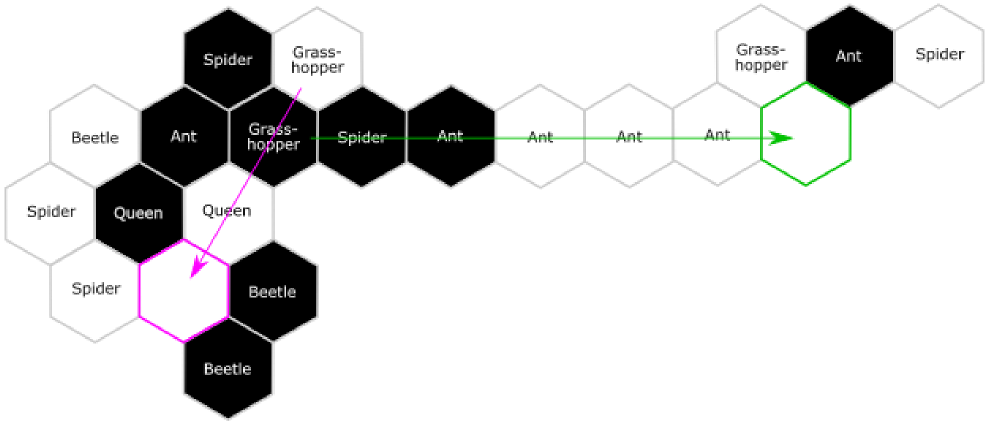

Figure 9: White's threat here is the win-in-one by the Grasshopper (moving it to the purple space would complete the surrounding of the black Queen). Black can trap this white Grasshopper by moving his own Grasshopper away. Then the white Grasshopper is the only link between the left and right-hand side of the Hive, and not allowed to move anymore. Furthermore, even if it was still allowed to move, it would need the black Grasshopper to be in its old spot to jump over it in order to get to the pink space.

1.4.4 Free a piece

Since in the proof all pieces are locked down except for the first one, the moving of that piece should free the next one and so on. The freeing of pieces is usually done by exploiting the One Hive rule. See figure 10 for an example.

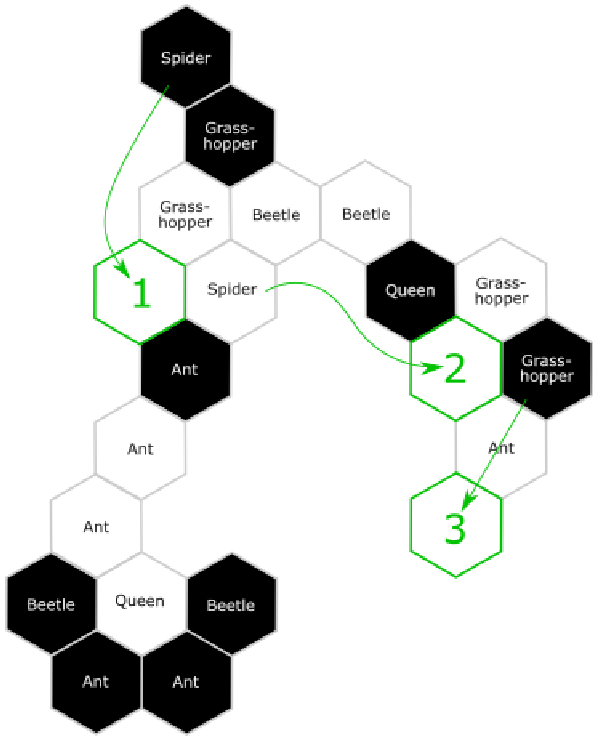

Figure 10: Moving the black Spider frees up the white Spider. Moving that piece then frees up the black Grasshopper. If all other pieces were to be locked, it is clear that this leads to a forced flow.

1.4.5 Beetle tower

The Beetle is the only 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 limitation on how high the Beetle can climb. It is thus possible to build a Beetle tower as high as one wishes, with another piece on the bottom if desired.

1.4.6 Dead pieces

In our proof, it will be useful to construct positions containing some pieces that either cannot move (by the game rules) or are not rational to move (because the piece's owner would lose the game soon after moving it). We call these dead pieces. There are multiple ways to construct a dead piece, all with their own benefits and disadvantages. The list below is not exhaustive, but these are the dead pieces we use in the proof.

1. A Spider or Ant that is (almost) completely surrounded. It must keep being not able to slide out of its spot throughout the course of the whole simulation. The colour of this dead piece is irrelevant.

2. A white Beetle on top of 6 black Beetles, with a white Queen on the bottom. This dead piece (or rather, tower of pieces) is useful in places where the top Beetle cannot affect the game in any meaningful way within one space reach. Once the white Beetle moves, the black Beetle underneath can immediately climb on top of it and block it permanently. As the white Beetle's move did not accomplish anything, black can now in at most 5 moves surround the helpless white Queen underneath. In our proof construction, we must be especially careful with this type of dead piece in areas where there is only one space between the dead piece and another part of the Hive that should not be connected directly to this part. The top Beetle making this direct connection is problematic for our proof if it frees up a dangerous piece (which can complete the surrounding of an opponent's Queen in one or two steps) of the same colour, whilst not freeing up a dangerous piece of the other colour, as the punishment for moving the top Beetle then comes a couple of turns later and the freeing of such a dangerous piece can be materialised the next turn.

3. A white Beetle on top of a Black ant. This type of dead piece can be a solution in areas where the previous one cannot be used. Moving the Beetle frees up the black Ant underneath. The Ant should then be able to reach a space where it can complete the surrounding of a white Queen.

Naturally, the colours in type 2 and 3 can be reversed to obtain a dead piece of opposite colour. In the figures, dead pieces of type 1 are called "DEAD inside", and dead pieces of type 2 and 3 are called "DEAD edge".

1.4.7 Chain of pieces

Any number of pieces can be immobilised by placing them in a long chain with a dead piece type 2 at the end. The first piece of the chain is attached to the Hive at a spot where it does not interfere with the structure. None of the pieces in the chain is allowed to move, as this would leave the rest of the chain unattached to the Hive.

2 Proof

2.1 TQBF

The problem we start the proof off with concerns quantified Boolean formulas, specifically the set TQBF of true fully quantified Boolean formulas. When each variable in a formula is in the scope of a quantifier, the formula is said to be fully quantified. A fully quantified formula is always either true or false. Note that the order of the quantifiers is important. A formula is in conjunctive normal form if it is a conjunction of clauses, where each clause is a disjunction of literals. In this context a literal is either a variable or the negation of a variable.

Definition 1. The set QBF of fully quantified Boolean formulas is $\{Q_1v_1...Q_nv_n \cdot F(v_1, ..., v_n) \mid Q_i \in \{\forall, \exists\}$, where v_i is a Boolean variable and F is a Boolean formula in conjunctive normal form containing the variables $v_1, ..., v_n\}$.

An example of a formula $F \in QBF$ is $\exists v_1 \forall v_2 \exists v_3 \forall v_4 (v_1 \vee \neg v_2 \vee v_3) \wedge (v_2 \vee \neg v_4)$

Definition 2. $TQBF = \{\langle\phi\rangle \mid \phi$ is a true fully quantified Boolean formula$\}$ [9, p. 338-339].

Theorem 1. *TQBF is PSPACE-complete* [10].

2.2 Preliminary reductions

2.2.1 Formula-Game

We now introduce a game in which two players oppose each other in a dispute over a formula. The players are called the $\exists$-player and the $\forall$-player, but they are also addressed as the existential and universal player respectively. The formula they get presented is of the form $Q_1v_1...Q_nv_n \cdot F(v_1, ..., v_n)$, where $Q_i \in \{\forall, \exists\}$, v_i is a Boolean variable and F is a Boolean formula in conjunctive normal form containing the variables $v_1, ..., v_n$. The $Q_1v_1...Q_nv_n$ part determines the flow of the game. The first quantifier prescribes which player starts. If it is an $\exists$, it is the turn of the existential player, and if it is a $\forall$, it is the universal player's turn. The same holds for later turns. In a turn, the player chooses a value, 0 or 1, for the variable succeeding the quantifier. After the first quantifier, play proceeds from left to right. Note that it is possible that a player has multiple turns in a row. The goal of the existential player is to make the formula $F(v_1, ..., v_n)$ `true`, and the goal of the universal player is to make it `false`. After all the variables $v_1, ..., v_n$ have been assigned a value, it is time to fill them in in the formula F. If F is now `true`, the existential player wins. If it is `false`, the universal player wins.

Definition 3. *Formula-Game* $= \{\langle\phi\rangle \mid$ the $\exists$-player has a winning strategy in the formula game when played with $\phi\}$.

Theorem 2. *Formula-Game is PSPACE-complete.*

It is not difficult to see that Formula-Game is PSPACE-complete. It is namely exactly the same as TQBF.

Proof. (Adapted from [9, p. 341-343]) Suppose we have a QBF-formula ϕ that looks like $\exists v_1 \forall v_2 \exists v_3 \forall v_4 ... \exists v_n \cdot F$. The formula ϕ is `true` if there exists a value for v_1 such that for all values of v_2 there exists a value for v_3 such that for all values of v_4, et cetera, F is `true`. Similarly, the $\exists$-player has a winning strategy in the formula game when played with ϕ if she can find an assignment for v_1 such that for all settings of v_2 she can find an assignment for v_3 such that for all settings of v_4, and so on, F is `true`. The same reasoning applies when the quantifiers are in a different order. Hence, a formula $\phi \in$ *Formula-Game* exactly when $\phi \in TQBF$ and with that Theorem 2 follows directly from Theorem 1 . □

For the purpose of determining the computational complexity of a two player perfect information game, it is in our opinion favourable to describe the game as *the problem of deciding whether one of the players has a winning strategy in an arbitrary [Game] position.* Another option is to describe the game as a set: [*Game*] $= \{x \mid$ player 1 has a winning strategy in game [Game] in position x$\}$. However, this definition does not give weight to the fact that it is in the spirit of such games that both players have equal chances.

The Formula-Game can be described as the problem of determining whether one of the players has a winning strategy in an arbitrary QBF-formula. The only difference with the set-based definition of the game is that in the latter a check whether the formula at hand is a QBF-formula is necessary. This technicality is left to the reader.

Theorem 3. *The problem of determining whether one of the players has a winning strategy in the formula game played with an arbitrary QBF-formula is PSPACE-complete.*

2.2.2 Generalised geography

Geography is a game, usually played by children, in which players take turns naming cities from all over the world. There are two simple rules: the city you name must start with the same letter as the last city ended with, and a city can only be named once. If a player cannot come up with a city, or names a city that has been named before, he or she loses. Play starts by the first player naming a city.

This game can be modeled by a directed graph. Every city has a corresponding node. There is an edge from one node to another if, according to the game rules, the city corresponding to the second can be named after the city corresponding to the first. Play starts by the first player marking a node. Players now take turns marking a node that has an incoming edge coming from the last marked node. If a player marks a node that has already been marked, he or she loses.

Without the references to cities, and with the additions of it being played by two players and a designated start node, we have the game we are using in this proof.

Definition 4. Generalised geography (GG) is a two player game played on the nodes of a directed graph. The first player starts by marking a designated start node. Players then alternate turns by marking a node that has an incoming edge from the last marked node. If a player marks a node that has been marked before, he or she loses the game.

Theorem 4. *GG is PSPACE-complete. [8]*

Proof. (Adapted from [9, p. 343-348]) To prove that GG is PSPACE-complete, it suffices to show that GG $\in$ PSPACE, and that it is PSPACE-hard.

To prove that GG $\in$ PSPACE, we give a recursive algorithm M that decides GG in polynomial space. If G_0 is the directed graph and s the designated start node, call $M(\langle G_0, s, 0\rangle)$, where $M(\langle G, u, p\rangle)$ is defined below for any given graph G, start node u, and $p \in \{0, 1\}$. When this returns 0, player 1 wins. When this returns 1, player 2 wins.

$M(\langle G, u, p\rangle)$

1. If u has outdegree 0, return p.
2. Else remove u and all edges containing u from G to form G_{new}.
3. For all nodes v that had an incoming edge from u in the original G, recursively call $M(\langle G_{new}, v, |p-1|\rangle)$.
4. If one of these returns $|p-1|$, the other player has a winning strategy in this path, so return $|p-1|$. Else return p.

This algorithm clearly decides GG. The only space required by this algorithm is for the recursion stack. The depth of the recursion is at most the number of nodes. At each level we add one node to the stack. Hence, this algorithm even runs in linear space.

To prove that GG is PSPACE-hard, we give a polynomial time reduction from the formula game problem of theorem 3. Player 1 from the GG game mimics the $\forall$-player, and player 2 the $\exists$-player. We construct the quantifier part of the formula by a chain of diamond gadgets. Each diamond represents the choice that the universal player, in case of a universal quantifier, or the existential player, in case of an existential quantifier, has to make. Suppose we have a QBF-formula ϕ that looks like $\exists v_1 \forall v_2 \exists v_3 \forall v_4 ... \exists v_n \cdot F$. The quantifier part of the formula is now simulated as described in figure 11. Later we will separately consider the case wherein the quantifier types are not alternating.

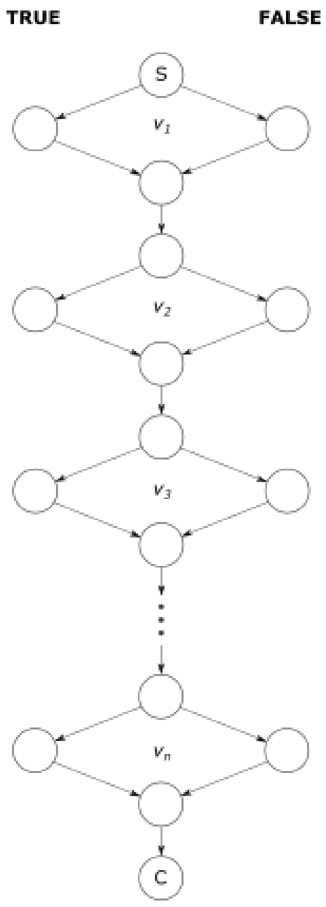

Figure 11: Play starts by player 1 marking the designated start node. Player 2 has the first choice: marking the left-hand node sets v_1 to `true`, and marking the right-hand node sets v_1 to `false`. Both these nodes have one outgoing edge, going to the bottom node of the first diamond. Player 1 marks this node, after which player 2 marks the first node of the next diamond. Player 1 can now choose to set v_2 to `true` by marking the left-hand node, or to `false` by marking the right-hand node. And so on.

After play has gone through the last diamond we arrive at node c. In the formula game all variables have been assigned a value by now, and the game would be finished at this point. If ϕ is `true` under these settings, the existential player wins, and the universal player wins if it is `false`. To make sure player 2 has a winning strategy when the existential player would have won, and player 1 has a winning strategy when the universal player would have won, we need two help gadgets: a *clause chooser* gadget and a *literal chooser* gadget. The universal player has won the formula game if there is a clause in which none of the literals is `true` under the chosen assignment. We construct the *clause chooser* gadget by representing each clause by a node. Node c has an outgoing edge to each of these nodes. Player 1 can now choose to continue play in one of the clauses. The existential player has won the game if every clause has at least one literal that is `true` under the chosen assignment. To construct the *literal chooser* gadget, we let every clause node have outgoing edges to nodes representing the literals the clause contains. Player 2 can now choose which literal from the clause to continue play with. Every node representing a literal has an outgoing edge to the node representing the same literal in the diamond structures. If play traversed through this node in the diamond structure, which means this literal was marked as `true`, player 1 now has to mark it again and loses. If it is still unmarked, player 1 can mark it and player 2 has no choice but to mark the marked node on the bottom of the diamond. So, with this construction player 1 has a winning strategy in the GG game when the universal player has a winning strategy in the formula game, and player 2 has a winning strategy in the GG game when the existential player has a winning strategy in the formula game. If the quantifiers are not alternating as in the example of figure 11, we can adjust the construction by including dummy nodes where needed. Such a dummy node is placed at the point where we need a turn switch without further interfering with the game. It has one incoming edge, coming from the node above, and one outgoing edge, going to the node below. □

2.2.3 Maximum degree 3

In GG, a node can have any number of incoming or outgoing edges. This is hard to simulate in a game like Hive. To keep things simple, we demand that each node has at most two incoming and one outgoing, or one incoming and two outgoing edges.

Theorem 5. *GG is PSPACE-complete even when played on graphs with maximum degree 3, maximum indegree 2 and maximum outdegree 2.*

Proof. (Adapted from [6]) To accomplish these properties, we can make simple modifications to the nodes that are not consistent with the theorem. They are treated as is explained in figure 12 and figure 13. □

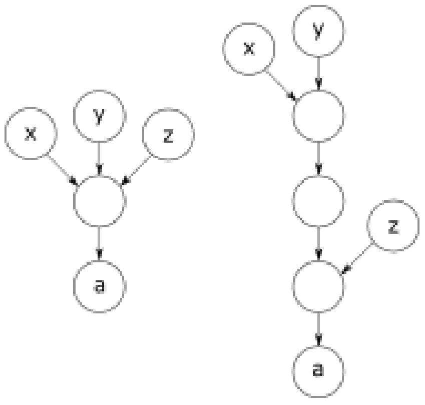

Figure 12: A node with indegree greater than 2 is replaced by a chain of nodes. On the left-hand side is a node with indegree 3. This would be replaced by the chain on the right-hand side. Naturally, greater indegree results in a longer chain.

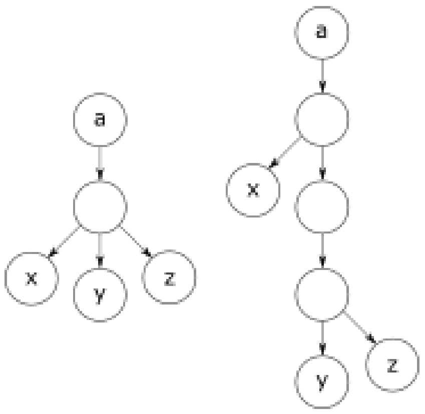

Figure 13: A node with outdegree greater than 2 is treated in a similar manner, and is replaced by a slightly different chain of nodes. On the left-hand side is a node with outdegree 3. This would be replaced by the chain on the right-hand side. Naturally, greater outdegree results in a longer chain.

Note that in the proofs for PSPACE-hardness of other games, like GO [5] and Hex [6], the reduction is done from a version of GG that is also bipartite and planar. For our proof these properties are not necessary. The bipartite attribute ensures that both players play on their own set of nodes. This is particularly important at the point where an edge from the *literal chooser* gadget goes to the corresponding node in the diamond of a universal quantifier. The act of marking that node (again) is what we call the check-back. In the structures of the proofs for these other games it is vital that the existential player, who is doing the check-back, was also the one who possibly marked this exact node in the diamond. This is because of how this check-back is designed in the proofs for these games. In our proof it is irrelevant whether it was a ∀-diamond or ∃-diamond, because they are simulated by different gadgets (for which the check-back works in both cases). This can be observed in the subsection where the *quantifier and tester* gadget (which includes the simulation of the check-back) is explained (2.3.2). In most board games it is not possible to let two structures cross without interfering with each other. For that reason, in proofs for other games there had to be found a solution for crossing edges in GG. Planarity ensures that these crossings do not occur. Hive has a planar nature as well, except for the fact that the Grasshopper can jump over structures. We use this to our advantage to overcome planarity. See section 2.3.10 for the details.

2.3 Final reduction to Hive

First the general idea of the reduction is discussed. After that the separate gadgets are introduced and explained. Then we discuss how the gadgets are put together. The proof is concluded by some necessary remarks on the bigger picture of the simulation.

2.3.1 General idea

The idea is that we are still simulating the Formula-Game problem from theorem 3, but via the graph problem representation of generalised geography and with the restriction that it is played on graphs with maximum degree 3.

From the starting position until the surrounding of a Queen, the game will proceed through a series of stages. First, in the *quantifier* stage, the players will take turns assigning values to the variables. Next, the ∀-player will choose a clause in the *clause chooser* stage. Then, in the *literal chooser* stage, the ∃-player will choose a literal from that clause. Finally, the ∃-player will do the check-back in the *tester* stage.

Our main strategy in designing this simulation is to at all times have all pieces locked down except for one. Locked down means that the piece is either legally not allowed to move, or if it is, moving it would result in an unavoidable loss. 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 the player has the choice between two spaces.

The locking down of most pieces and regulating the flow of the game is done by making repeated use of the One Hive rule. That this is the case in a single gadget can be easily verified. However, the fact that the pieces that are locked by the One Hive rule in the separate gadgets are still locked when the gadgets are put together to accomplish a simulation, is not trivial. The details on this are discussed in section 2.3.9.

All structures can be rotated in steps of 60 degrees when needed.

Theorem 6. *Hive is PSPACE-hard.*

Proof. To simulate the gadgets from the game in theorem 5, we need the following structures: *∃-player's choice at a quantifier*, *∀-player's choice at a quantifier*, *tester*, *join*, *∀-player's choice for the clause*, and *∃-player's choice for the literal.* To be able to put these gadgets together, we also need the following structures: *Turn switcher*, *direction changers* and a *gap.*

2.3.2 Quantifier and tester

This gadget is the heart of the reduction. It simulates both the choice a player has to make at a quantifier, assigning a value to the variable, and the tester that checks whether the variable was assigned `true` or `false` and decides who wins. In the GG-representation of Formula-Game the diamonds look the same for both players. In our proof, we have to make different structures for the ∃-quantifier and the ∀-quantifier. The choice at the *quantifier* stage has to directly influence the *tester* stage. This is done by the placement of the Spiders (the gadget's main piece, whose choice of movement direction will determine whether the simulated variable is assigned `true` or `false`), but because of the reversed colours the *tester* for the ∀-structure has to be designed in a different way than the ∃-structure. Furthermore, we have to find a way to temporarily lock the first Spider of each *quantifier*, only to unlock it at the right time. The structure that simulates the very first quantifier of the formula should not contain this Spider-lock. Summarising, this gadget appears in four different forms. The two possible structures simulating the first quantifier are simpler and are discussed first (figure 14 for the ∃-quantifier and figure 17 for the ∀-quantifier).

When the first quantifier of the formula is ∃, the following structure is used to simulate it. Black (the existential player) has the first turn of the game and the only piece that is not locked down is the black Spider. Note that white's Grasshoppers 1 and 2 are still locked during the *quantifier* phase of the game, and are played to get to the *tester* part after the *quantifier*, *clause* and *literal chooser* stages have all been completed.

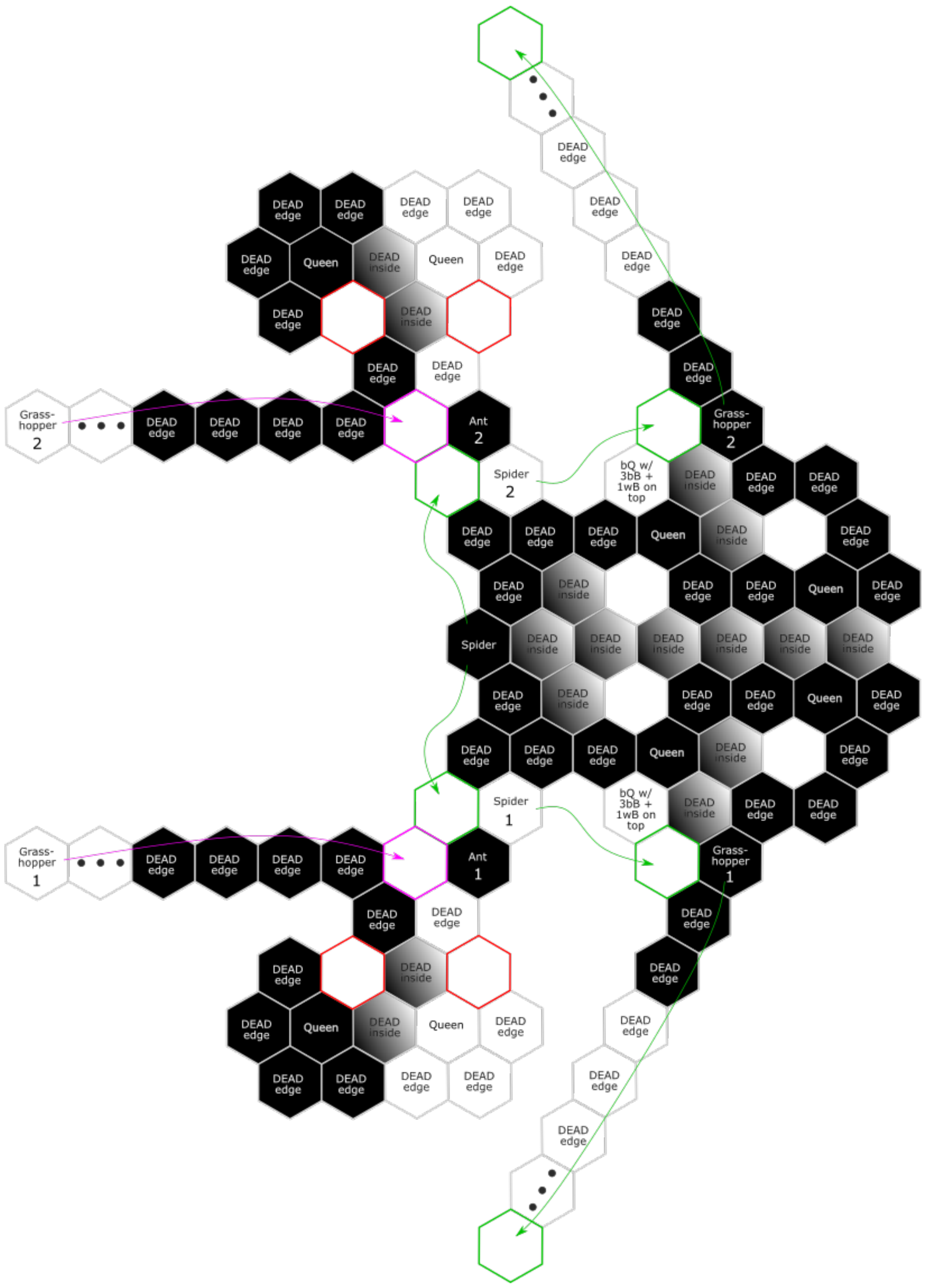

Figure 14: This structure simulates both the choice for the existential player at the first quantifier if it is an ∃ and the *tester* that later checks whether the variable bound by the quantifier was assigned `true` or `false`. The bottom half of this structure represents `true`, and the upper half represents `false`. The black Spider is the only piece that can move. The existential player chooses to move the Spider down (assigning the variable `true`) or up (assigning the variable `false`). In the case of moving it down, it frees up white Spider 1, which has no choice but to move to the right (the red space is suicide). This then frees up black's Grasshopper 1, which can only jump away to the next structure. The special dead piece "bQ w/ 3bB + 1wB on top" is explained on the next page. Moving the black Spider up results in a similar sequence of moves with white's Spider 2 and black's Grasshopper 2. Whichever one of the black Grasshoppers moves on to the next structure will cause the next gadget to become unlocked. Now for the *tester* part. During the *tester* stage, when the existential player chooses literal v_{true} white Grasshopper 1 is coming in, and if v_{false} is chosen white Grasshopper 2 is coming in. A simple case analysis shows that the existential player has a winning strategy iff he or she chooses the same literal in the *tester* phase as the literal that was chosen in the *quantifier* phase (see below).

Special dead piece "bQ w/ 3bB + 1wB on top" consists of a black Queen on the bottom, with 3 black Beetles and 1 white Beetle on top. This tower of pieces has two functions. In the first place, naturally, it is a dead piece. The top piece (the white Beetle) can make no valuable move in one turn. After it would move, the black Beetle that now is on top of the tower would be able to jump on top of the white Beetle, leaving the next black Beetle free to, without restraint, find a white Queen to surround in the next moves. In the second place, the tower ensures that black Grasshopper 1 cannot jump over white Spider 1 to land on the space next to it.

The following case analysis for the *tester* shows that for the first quantifier, in case of an ∃, the existential player has a winning strategy iff he or she chooses the literal that was chosen in the *quantifier* phase. There are four cases:

`true true` The variable was assigned `true` at the *quantifier* stage and the existential player chooses the literal v_{true}.
Before the white Grasshopper 1 is coming in we have the following close-up of the position from figure 14, with the rest of the pieces locked down (figure 15). It shows that the existential player indeed has a winning strategy when he or she chooses the literal v_{true} after v was assigned `true` in the *quantifier* phase.

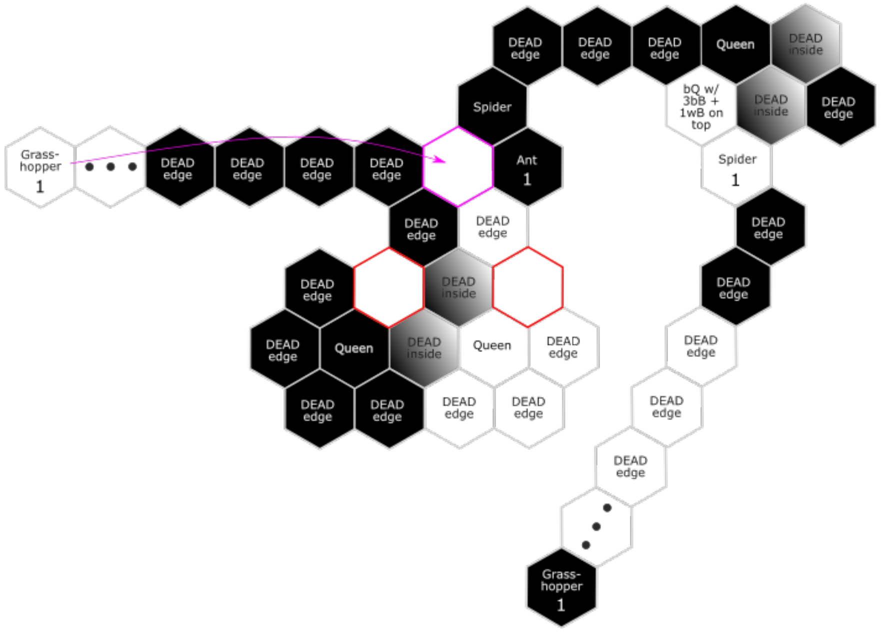

Figure 15: This is a close-up of the position from figure 14 (the variable was assigned `true`) right before the white Grasshopper 1 is coming in. White Grasshopper 1 comes in from the left to land on the purple space. This frees up black Ant 1, which can now complete the surrounding of the white Queen by moving to the right red space. This means that the existential player has a winning strategy.

`true false` The variable was assigned `true` at the *quantifier* stage and the existential player chooses the literal v_{false}. Before the white Grasshopper 2 is coming in we have the following close-up of the position from figure 14, with the rest of the pieces locked down (figure 16). It shows that the existential player does not have a winning strategy when he or she chooses the literal v_{false} after v was assigned `true` in the *quantifier* phase. In fact, the universal player has a winning strategy.

`false true` This case is symmetric to the `true false` case.

`false false` This case is symmetric to the `true true` case.

Concluding, for the first variable of the formula, if it is bound by an ∃, it holds that the existential player has a winning strategy iff he or she chooses the literal that was chosen in the *quantifier* phase. We will generalise this to all variables when we discuss the other three forms of the gadget.

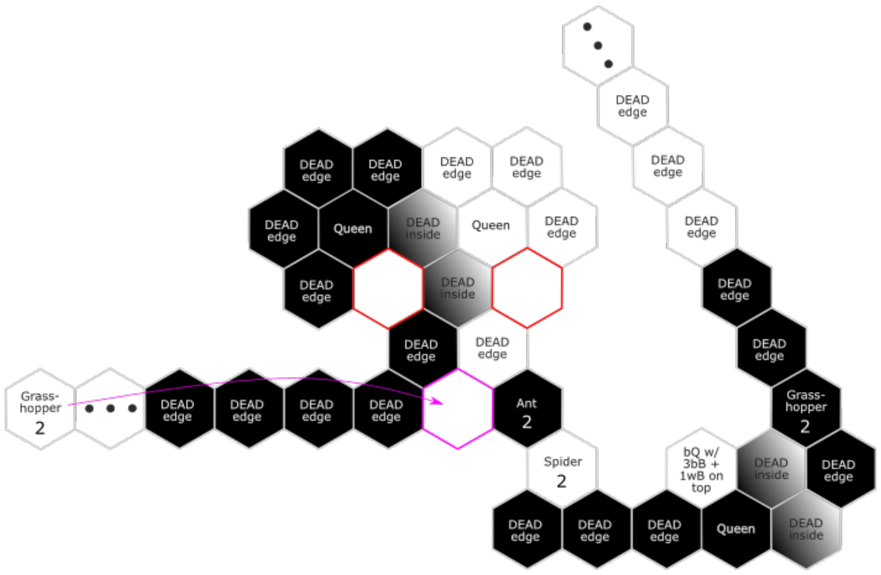

Figure 16: This is a close-up of the position from figure 14 (the variable was assigned `true`) right before the white Grasshopper 2 is coming in. White Grasshopper 2 comes in from the left to land on the purple space. This does not free up any piece. In fact, black's pieces are in total lock-down, and white can win on his next turn by moving white Grasshopper 2 to the left red space and thereby completing the surrounding of the black Queen, or punish black for moving one of his dead pieces. This means that the universal player has a winning strategy.

Next, we look at the structure that simulates the first quantifier of the formula if it is a $\forall$. It is quite similar to the previous structure. The colours are reversed for the *quantifier* part. For the *tester* part the colours cannot simply be reversed, because the existential player should still have a winning strategy iff he or she chooses the literal that was chosen in the *quantifier* phase. Hence, the *tester* part is designed in a different way than in the $\exists$-structure.

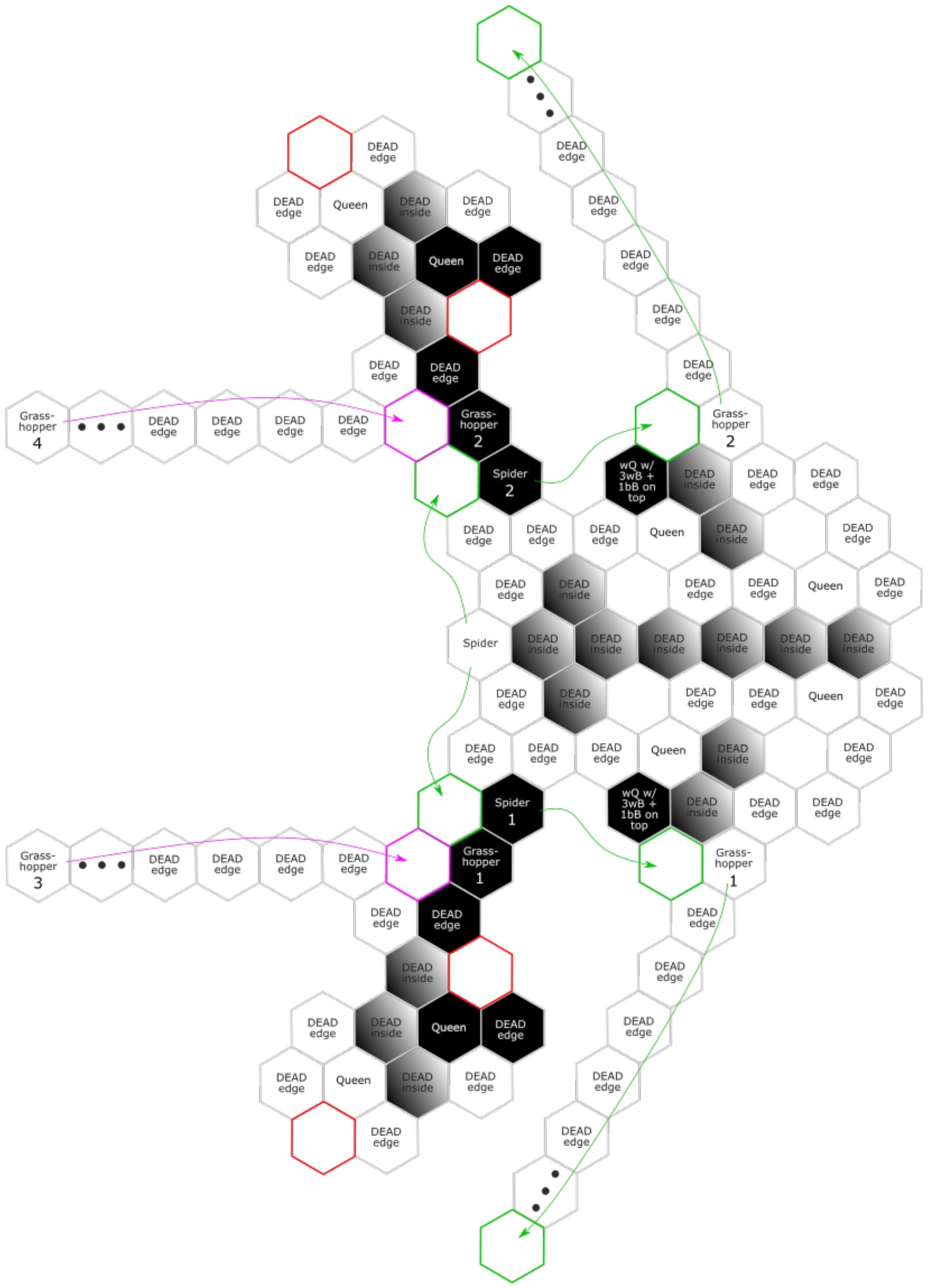

Figure 17: This structure simulates both the choice for the universal player at the first quantifier if it is a ∀ and the *tester* that checks whether the variable bound by the quantifier was assigned `true` or `false`. The play through the *quantifier* part is analogous to figure 14, but with reversed colours. Here it is a white Spider that can move, allowing the universal player the choice of assigning the variable `true` (moving the Spider down) or `false` (moving it up). For the *tester* part there are again four possibilities. These are discussed on the next page.

The following case analysis shows that for the first quantifier, in case of a ∀, the existential player has a winning strategy iff he or she chooses the literal that was chosen in the *quantifier* phase. This time the figures are left out. There are four cases:

`true true` The variable was assigned `true` at the *quantifier* stage and the existential player chooses the literal v_{true}. White Grasshopper 3 is coming in to land on the purple space. This frees up black Grasshopper 1, which can now complete the surrounding of the bottom-left white Queen by jumping to the red space next to it. This means that the existential player has a winning strategy.

`true false` The variable was assigned `true` at the *quantifier* stage and the existential player chooses the literal v_{false}. White Grasshopper 4 is coming in to land on the purple space. This does not free up any piece. In fact, black's pieces are in total lock-down, and white can win on his next turn by moving white Grasshopper 4 to red space two spaces away on the top-right from it and thereby completing the surrounding of the black Queen, or punish black for moving one of his dead pieces. This means that the universal player has a winning strategy.

`false true` Similar to the previous case.

`false false` Similar to the `true true` case.

Summing up, we showed that for the first variable of the formula the existential player has a winning strategy iff he or she chooses the literal that was chosen in the *quantifier* phase.

For the ∃ and ∀-structures that simulate the quantifiers in a formula that are not the first one, we need to lock the first Spider, while being able to unlock it at the right time. We introduce the ∀-structure first, as it is simpler.

Proceedings of play for figure 18: Black Grasshopper 3 is coming in from the previous structure to land on green space 1. This causes a double connection between the *quantifier and tester* part of this structure and the structure that black Grasshopper 3 came from. It must be a black piece making this double connection, because it frees up black Spider 2 and black Grasshopper 2, which would now be able to complete the surrounding of the white Queen on the red space. White Grasshopper 5 is now also free to move and it can only jump away to green space 2. This frees up the white Spider, which is what we wanted to achieve. However, it is black's turn. White's last move also freed up black Grasshopper 4, which jumps to green space 3 and prevents the white Grasshopper to surround the black Queen in the corner. It now is white's turn, with all pieces locked down except for the Spider. Play will now proceed in exactly the same manner as in the previously discussed ∀-structure (figure 17). Moreover, the extra pieces in this structure do not affect the *tester* part from figure 17, so we do not have to repeat the case analysis. Following up on this we have that for variables bound by a ∀, the existential player has a winning strategy iff he or she chooses a literal that was chosen in the *quantifier* phase.

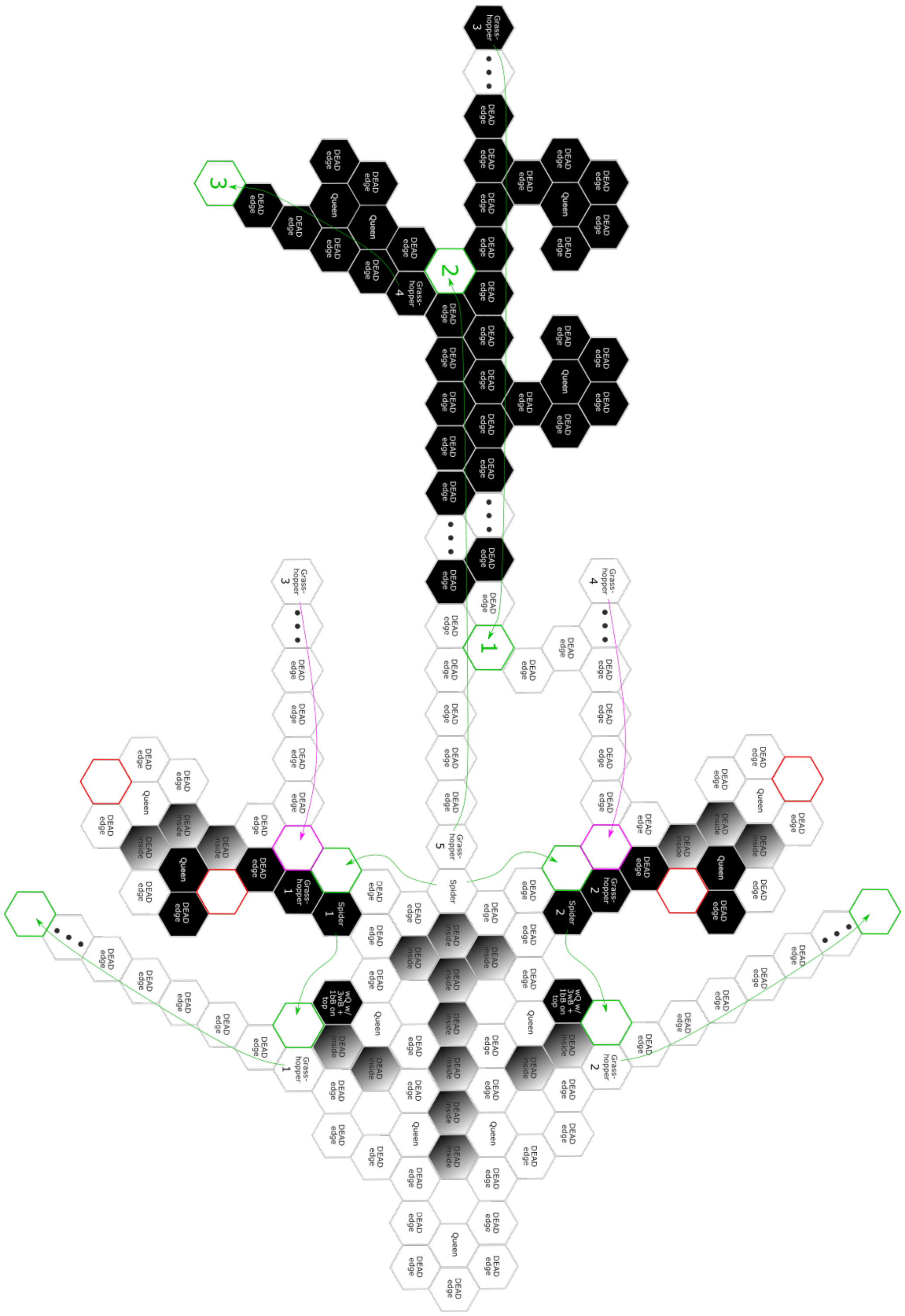

Figure 18: This structure simulates the variables bounded by a ∀, as well as the *tester* that checks whether the variable was assigned `true` or `false`. See previous page for the proceedings of play.

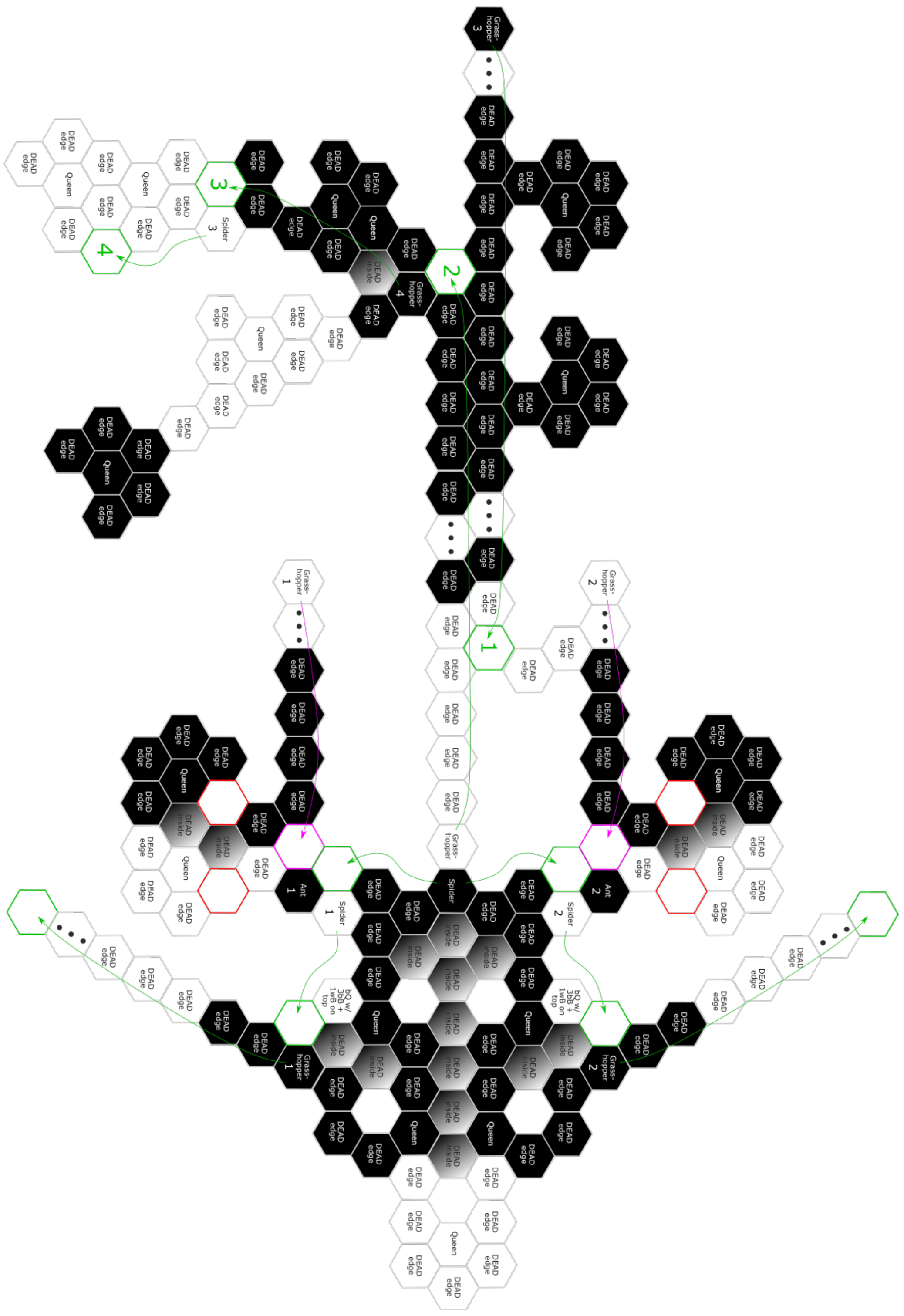

Figure 19: This structure simulates the variables bounded by an ∃, as well as the *tester* that checks whether the variable was assigned `true` or `false`. See next page for the proceedings of play.

Proceedings of play for figure 19: Black Grasshopper 3 is coming in from the previous structure to land on green space 1. This frees up the white Grasshopper, which can only jump away to green space 2. This frees up the black Spider, which is what we wanted to achieve. However, the white Grasshopper would be able to complete the surrounding of a black Queen on its next turn (or, if designed in a different way, would be able to move which would dismantle the reduction), so black must ensure that this Grasshopper cannot move. White's last move freed up black Grasshopper 4, which jumps to green space 3. This then frees up white Spider 3, which moves to green space 4. Note that white Spider 3 will never move back to its previous space because black Grasshopper 4 would then be able to surround a white Queen. Now it is black's turn, with the black Spider free to move. Play will now proceed in exactly the same manner as in the previously discussed ∃-structure (figure 14). Moreover, the extra pieces in this structure do not affect the *tester* part from figure 14, so we do not have to repeat the case analysis. Following up on this we have that for variables bound by a ∃, the existential player has a winning strategy iff he or she chooses a literal that was chosen in the *quantifier* phase. Combining this with the result for variables bound by a ∀, we have that the existential player has a winning strategy iff he or she chooses a literal that was chosen in the *quantifier* phase.

2.3.3 Join

This gadget is required after each *quantifier* gadget, and when a literal appears in more than one clause. There can only be one Grasshopper coming in to do the check-back in the *tester* gadget. So, if there are multiple edges going to the *tester* they need to be joined beforehand.

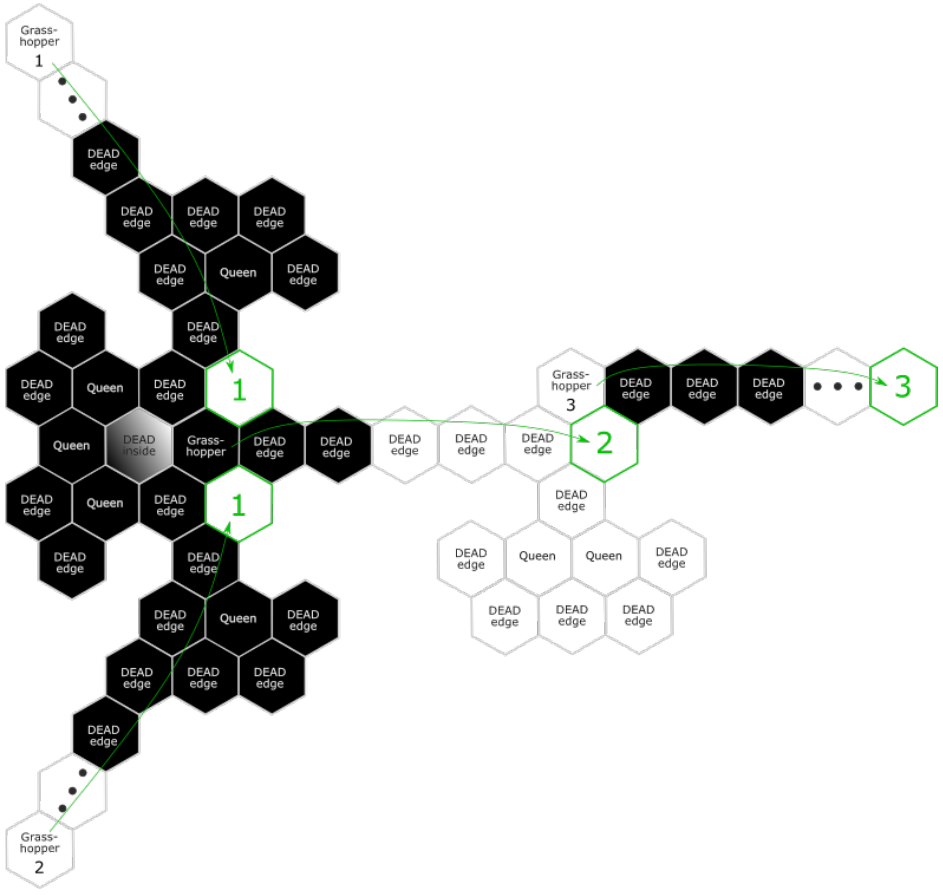

Figure 20: This gadget joins two other structures together. Either Grasshopper 1, coming from the structure above, or Grasshopper 2, coming in from the structure below, lands on the corresponding space marked with a 1. This frees up the black Grasshopper, which can only jump to the space marked with a 2. This then frees up Grasshopper 3, which continues to the next structure on the right. Naturally, the colours can be reversed when two structures with black Grasshoppers coming out need to be joined.

2.3.4 Clause/literal chooser

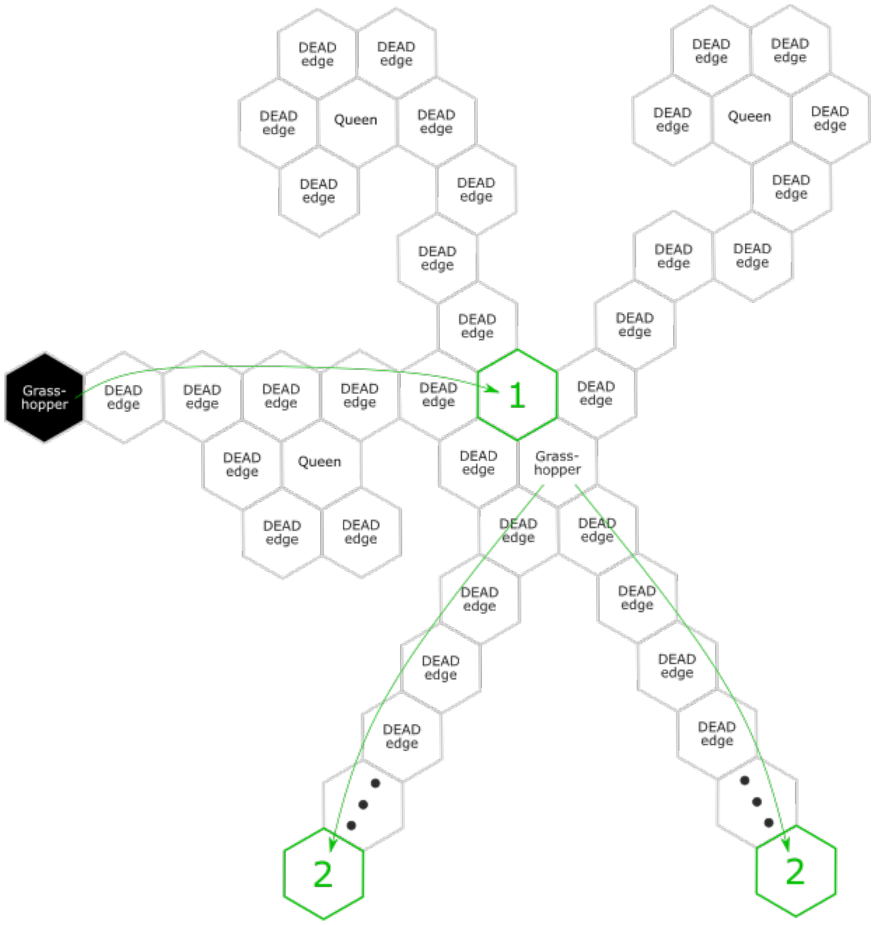

Figure 21: This gadget simulates the split from the proof for PSPACE-hardness of GG that is used when the universal player chooses a clause, and when the existential player chooses a literal. The black Grasshopper is coming in from the left and lands on the space marked with a 1. This frees up the white Grasshopper, which can jump to one of the two spaces marked with a 2. This represents the choice between two clauses. By reversing the colours, we obtain the *literal chooser* gadget.

2.3.5 Turn switcher

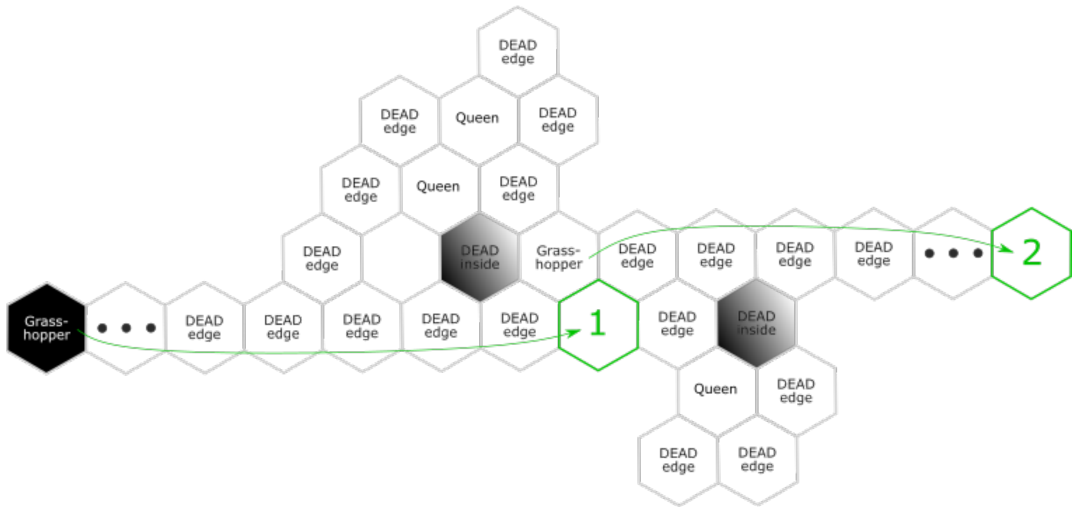

Figure 22: This gadget's only purpose is to switch turns. There is a black Grasshopper coming out of the previous structure, but we need a white Grasshopper coming in in the next structure to then give black a choice there. The black Grasshopper coming in from the left lands on the space marked with a 1. This frees up the white Grasshopper, which has no choice but to jump to the structure on the right. Naturally, the colours can be reversed to obtain a turn switch from black to white.

2.3.6 60 degrees direction changer

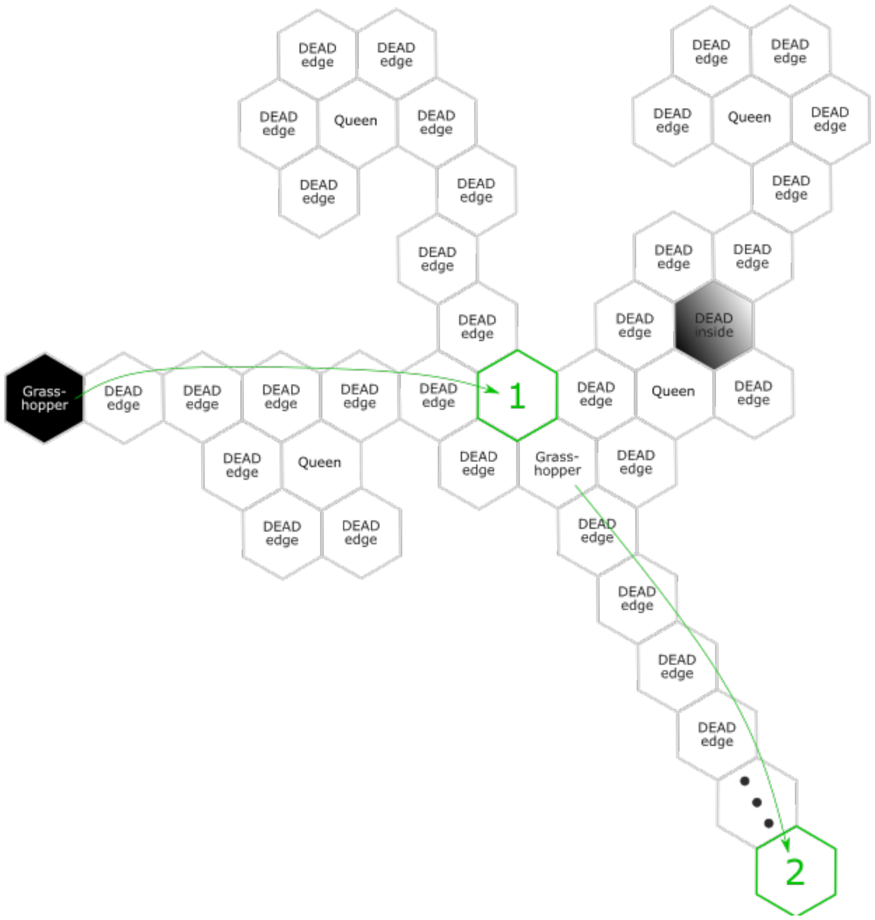

Figure 23: This gadget enables us to make a turn of 60 degrees. The black Grasshopper comes in from the left to land on the space marked with a 1. This frees up the white Grasshopper, which can only jump to the space marked with a 2. Naturally, the colours can be reversed to obtain a *direction changer* with a white Grasshopper coming in from the previous structure. Furthermore, the structure can be flipped vertically to obtain an anticlockwise 60 degrees direction change.

2.3.7 120 degrees direction changer

One could think that this gadget is redundant, as two *60 degrees direction changers* combined with a *turn switcher* would achieve exactly the same thing. However, with that construction a problem would arise at the point where a Grasshopper is making its jump into the *tester* gadget. The structures simulating the quantifiers of a formula are aligned horizontally, from left to right. The Grasshoppers coming in to the *tester* parts are also coming in from left to right. Hence, we need a direction change to get those Grasshoppers to jump from left to right. For structures that simulate quantifiers that are not the first quantifier, the *60 degrees direction changer* does not fit at this point. The part with the "DEAD inside" (see figure 23) overlaps the part that connects this *quantifier* with the previous one. You can verify this by examining figures 18 and 19 and imagining where the *direction changers* are incorporated. The *120 degrees direction changer* gadget, on the other hand, does fit at this point. The part with the almost surrounded Queen that is aligned with the outgoing Grasshopper sticks out only one space. This leaves us with a two space gap for the the bottom half of the *quantifier*, and a one space gap for the upper half. The attentive reader will realise that we need to investigate the situation for the upper half, as white dead pieces of type 2 (see section 1.4.6) in the *120 degrees direction changer* are separated only by a one space gap from black dead pieces type 2 in the *quantifier*. We examine the ∀-structure of figure 18 (the ∃-structure case is similar). There are two possibilities: play has already gone through the *quantifier*, or it has not. In the first case, no pieces that pose an immediate threat (so called dangerous pieces) are freed when the connection is made. In the second case, the loop that arises after making the connection contains black Grasshopper 2, which can complete the surrounding of the white Queen in one move by jumping to the red space. Hence, white will clearly not make the connection. If black makes the connection, white has to respond in a swift manner. Moving white Grasshopper 5 to green space 2 prevents black Grasshopper 2 from moving again. Black now has no immediate win threats, and has to move black Grasshopper 4 to green space 3 to prevent white's threat, after which white can use the Beetles that were freed up by the movement of the black Beetle to win in the coming turns.

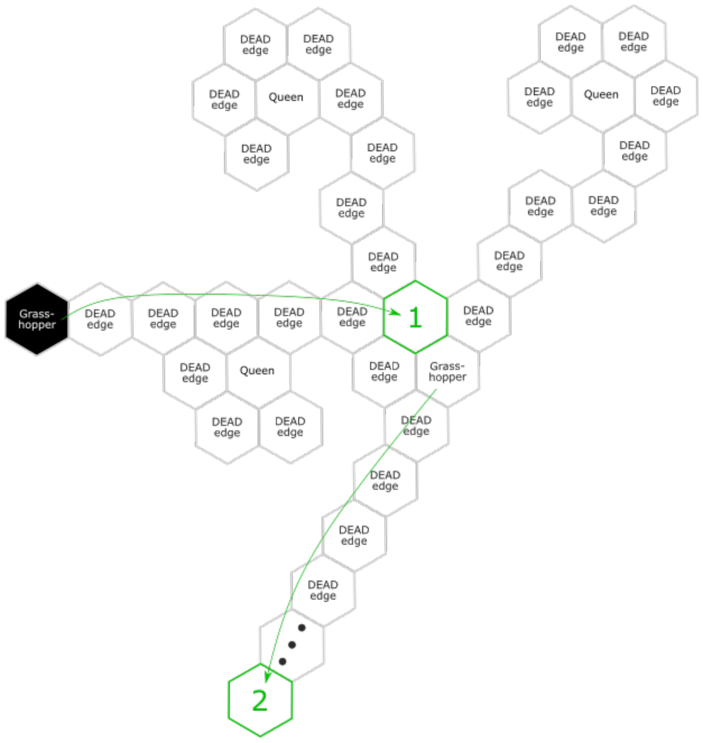

Figure 24: This gadget enables us to make a turn of 120 degrees. The black Grasshopper comes in from the left to land on the space marked with a 1. This frees up the white Grasshopper, which can only jump to the space marked with a 2. Naturally, the colours can be reversed to obtain a *direction changer* with a white Grasshopper coming in from the previous structure. Furthermore, the structure can be flipped vertically to obtain an anticlockwise 120 degrees direction change.

2.3.8 Gap

To ensure that all pieces trapped by the One Hive rule are really trapped, it is necesarry to at some places include a gap between two structures. 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. Secondly, we need a gap between the *literal chooser* gadgets and the *tester* gadgets. For the details on these two types, see the next section (2.3.9). Lastly, the *gap* gadget is used whenever a crossing occurs (see section 2.3.10).

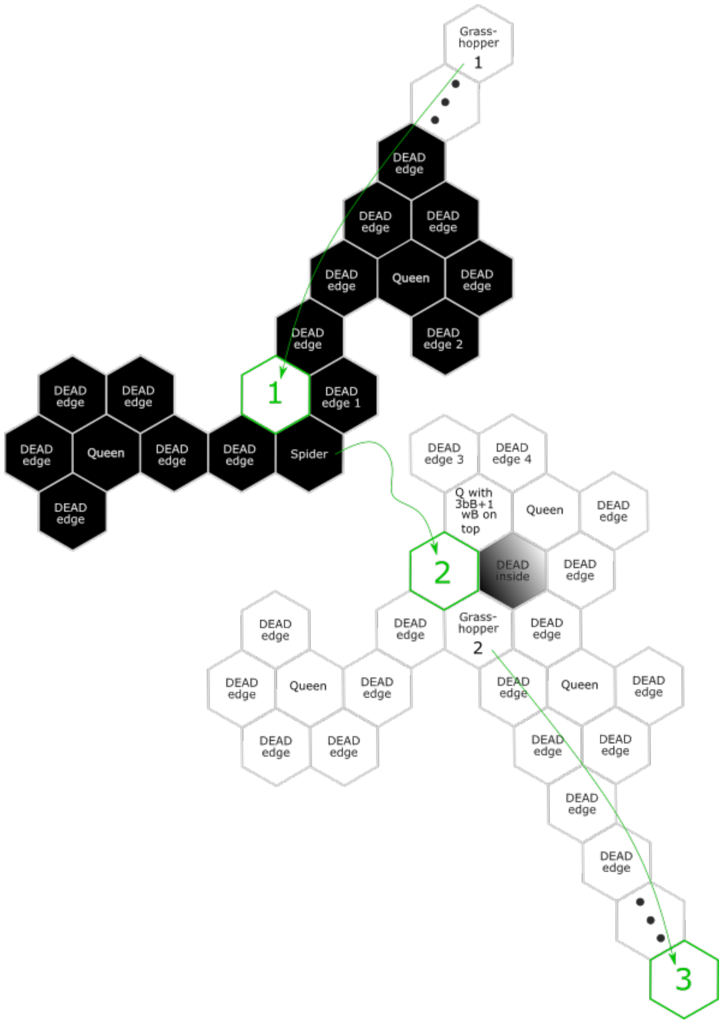

Figure 25: This gadget enables us to establish a gap where we need it. White Grasshopper 1 comes in from the previous structure to land on the space marked with a 1. This frees up the black Spider, which can pass over the gap by following the green arrow and move to the space marked with a 2 (moving left would be suicide). This move frees up white Grasshopper 2, which jumps away to the next structure.

2.3.9 Putting the gadgets together

In this section we show how the gadgets can be put together to simulate a formula, while taking in account that every gadget must be linked to any other gadget by exactly one path. There must be at least one path because otherwise it would be the case that two pieces are not connected, which contradicts the One Hive rule. There must be at most one path because otherwise the pieces that are purposely trapped using the One Hive rule would no longer be trapped. When we consider the gadgets that are linked with each other, the full board configuration of the simulation of a formula can be seen as a long horizontal line, with one branching tree at the end point, and possibly multiple branching trees on either side (top and bottom) of the line. In this analogy, the *quantifiers* are the line. Play will flow from left to right, with the individual *quantifier* gadgets linked solely via one way. The *clause choosers*, and after that *literal choosers*, following the last *quantifier* will then start to form a branching tree. Looking in the opposite direction as the flow of play, the *join* gadgets originating from a *tester* representing a literal that is used in multiple clauses form a branching tree as well.

We achieve this setup by putting the gadgets together as follows:
Recall that the first *quantifier* gadget has two outgoing chains of pieces, one pointing to the top-left and one pointing to the bottom-left. *Direction changers* are used to get them aligned again with the line (from left to right). The bottom one has a *gap* incorporated. Next, their direction is pointed towards the middle to join them with the *join* gadget, after which the next *quantifier* gadget (this time with an initially trapped spider) is inserted. And so on for the rest of the *quantifier* gadgets. Note that the *quantifiers* are linked only via one way, because of the *gaps* that are incorporated in the bottom half. After the last *quantifier* the universal player must choose a clause. This is done by a series of *clause choosers*, branching out with every extra clause after the second one (see figure 13). Then, for each clause, we have a similar branching out for the literals (performed by *literal chooser* gadgets). At this point, there is one path for each possible combination of clause and literal. At the end of each path, a *gap* is inserted. Then the paths that need to go to the same *tester* are joined by *join* gadgets (see figure 12). Finally, every (potentially joined) path is directed to the corresponding *tester* gadget.

2.3.10 Crossings

When enough space is taken between the gadgets, no crossings occur in the *clause chooser* and *literal chooser* area. In the area after the incorporated *gaps*, however, when the literals from the clauses are directed to the corresponding *testers*, crossings are unavoidable for some formulas. Note that some of the paths are joined before they are directed to the corresponding *tester*. In this process crossings might also be unavoidable. In this section we present a way to deal with these crossings.

Lemma 7. *Even when crossings occur, all gadgets are still linked via exactly one path.*

Proof. The crossing itself is easily implemented. At the point of the crossing, each of the two paths is a chain of dead pieces over which a Grasshopper will jump from one gadget into the next one. So, to implement the crossing we can let the two chains cross each other. At the point of the crossing, one piece from one of the two chains is left out, as it would be in the same spot as a piece from the other chain. The Grasshoppers of both chains are still able to jump over their respective chain in exactly the same manner as without a crossing.

Let a be the path of one of the two paths from the aformentioned incorporated *gap* to the *tester*. Let b be the path of the other one between its respective *gap* and *tester*. We divide a in two parts. Let a_1 be the part between *tester* and the crossing, and a_2 the part between the crossing and the *gap*. We do the same for b. There now are two paths from the *quantifier* area to the crossing, namely a_1 and b_1. This means that all pieces that were previously locked down by the One Hive rule in a_1, b_1 and the *quantifier* gadgets between (and including) the two *testers* are free to move, which would ruin the proof. This is prevented by including a *gap* in b_1, close to the crossing. b_1 is now separated in two parts: b_{1a} (from *tester* to newly included *gap*) and b_{1b} (from newly included *gap* to the crossing). In the situation where there was no crossing between the two paths, a and b were two separate paths from different *testers* to different *gaps*. In the new situation, there are two separate paths as well. One consists of a_1, which at the crossing branches out into b_{1b}, a_2 and b_2. The other one is b_{1a}. This method is carried out for all crossing paths, ensuring the One Hive rule still applies in all gadgets. □

2.3.11 Remarks

To complete the proof we need to address three things: the starting position of the simulation of a formula, the amount of pieces that is required in a simulation and the duration of a simulation.

First we point out that the starting position of a simulation is reachable. The players take turns putting pieces on the playing field and moving them, starting with the first *quantifier* and working their way through all the gadgets. Given the fact that the pieces can be put into the game near the place where they need to end up, and the fact that they are quite versatile, it is easy to imagine that the starting position of the simulation can be reached. The players only need to ensure they do not incidentally trap a Queen in the process. The Beetles are useful pieces to assist another piece in reaching its spot (for instance by temporarily making an extra connection such that a piece trapped by the One Hive rule can move), as they can reach every possible space and are always able to move back to their designated space. As both players have the same set of pieces, it is very likely that some pieces are not used after the full board configuration of the simulation of a formula is reached. The remaining pieces are placed on the board and moved into a chain, attached to a gadget that lies on the border of the Hive in such a way that it does not influence the game, with a beetle tower at the end of the chain.

Next we note that a polynomial amount of pieces (relative to the length of the formula) is sufficient to simulate the formula. This is straightforward, as every gadget consists of a fixed amount of pieces, and every distinct variable, clause and literal requires a fixed amount of gadgets to be simulated by.

Lastly we point out that a simulation can be carried out in a polynomial amount of time (compared to the length of the formula). To reach the starting position of the simulation of a formula takes more time than the actual playthrough of the simulation that follows (which requires a linear amount of moves). However, it is evident that the starting position can be reached in polynomial time.

This concludes the proof of theorem 6. □

3 Conclusion

We showed that generalised Hive is PSPACE-hard by reducing the PSPACE-complete set TQBF of true quantified boolean formulas, via a series of steps, to the problem of deciding whether one of the two players has a winning strategy in an arbitrary N-Hive position. Using these in-between steps is a well known technique in proving the complexity of generalised versions of board games [4–6, 11] and video games [12]. Each of these games has its own characteristics. The main contribution of this proof is that it gives insight in the concept that underlies the One Hive rule, as we mainly relied on making repeated use of the One Hive rule in designing the different gadgets (as well as in the broad configuration of the simulation of a formula).

The obvious recommendation for future research is to determine whether N-Hive is in PSPACE, and if that does not succeed, in EXPTIME. Together with this proof, the proof for N-Hive being in PSPACE would prove PSPACE-completeness.

It might be more in the spirit of the regular Hive game to include generalised versions that have one Queen each. The analogous choice for the King was also made in NxN chess [11]. The Queen Bee in Hive is similar to the King in chess. In both games the goal is to mate your opponent's main piece: the Queen Bee or King. Hence, we think that everyone agrees that you would need at least one main piece for each player in any admissable generalisation. A suggestion for a proof for PSPACE-hardness of N-Hive with (at least) one Queen each could be to change the proof strategy. In this proof we made repeated use of the One Hive rule to lock most pieces down, and some pieces were locked down because they could only move to a suicide hex as described in section 1.4.2. With only one Queen each, these suicide hexes are not at the mathematician's disposal anymore (alright, you can have one for each player). Instead, a suicide hex could be mimicked by a space that would free an Ant when occupied. This Ant would then be able to trap the opponent's one Queen in only one move (or more, depending on the proof strategy). It could be convenient to abandon the idea of forced moves by locking every piece down except for one using the One Hive rule and suicide hexes. In the proofs for other games, GO for example, the idea of immediate win threats is used as the main strategy for the flow of the game. That approach might be easier.

As addition to the basic game, Hive has three expansion pieces: the Ladybug, Mosquito and Pillbug. It could be investigated whether adding these expansion pieces would change the complexity of the game. Adding the expansion pieces would obviously not make the game less complex, but we think that it would neither increase complexity. These pieces are merely interbreedings of other pieces, and the complexity of Hive lies to a greater extend in the abstract rules (like the One Hive rule) than in the exact movement of the pieces.

References

[1] A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the International Congress for Logic, Methodology and Philosophy of Science*, pages 24–30, 1964.

[2] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[3] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 5 1965.

[4] R. A. Hearn. Amazons, konane, and cross purposes are pspace-complete. *Games of No Chance 3*, 56:287–306, 2 2009.

[5] D. Lichtenstein and M. Sipser. Go is polynominal-space hard. *Journal of the Association for Computing Machinery*, 27(2):393–401, 4 1980.

[6] S. Reisch. Hex is pspace-complete. *Acta Informatica*, 15(2):167–191, 6 1981.

[7] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 4 1970.

[8] T. J. Schaefer. On the complexity of some two-person perfect-information games. *Journal of Computer and System Sciences*, 16(2):185–225, 4 1978.

[9] M. Sipser. *Introduction to the Theory of Computation.* CENGAGE Learning, 3 edition, 2013. International ed.

[10] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 1–9, Austin, Texas, USA, 4 1973.

[11] J. A. Storer. On the complexity of chess. *Journal of Computer and System Sciences*, 27(1):77–100, 8 1983.

[12] G. Viglietta. Lemmings is pspace-complete. *Theoretical Computer Science*, 586:120–134, 6 2015.