Hexasort – the Complexity of Stacking Colors on Graphs

Linus Klocker
TU Wien, Austria

Simon Dominik Fink
Department of Algorithms and Complexity, TU Wien, Austria

Abstract

Many popular puzzle and matching games have been analyzed through the lens of computational complexity. Prominent examples include Sudoku [13], Candy Crush [7], and Flood-It [4]. A common theme among these widely played games is that their generalized decision versions are NP-hard, which is often thought of as a source of their inherent difficulty and addictive appeal to human players. In this paper, we study a popular single-player stacking game commonly known as Hexasort. The game can be modelled as placing colored stacks onto the vertices of a graph, where adjacent stacks of the same color merge and vanish according to deterministic rules. We prove that Hexasort is NP-hard, even when restricted to single-color stacks and progressively more constrained classes of graphs, culminating in strong NP-hardness on trees of either bounded height or degree. Towards fixed-parameter tractable algorithms, we identify settings in which the problem becomes polynomial-time solvable and present dynamic programming algorithms.

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

Keywords and phrases Hexasort, offline color stacking on graphs, NP-complete, polynomial-time solvable, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.FUN.2026.26

Related Version *Full Version*: `https://arxiv.org/abs/2603.01244` [10]

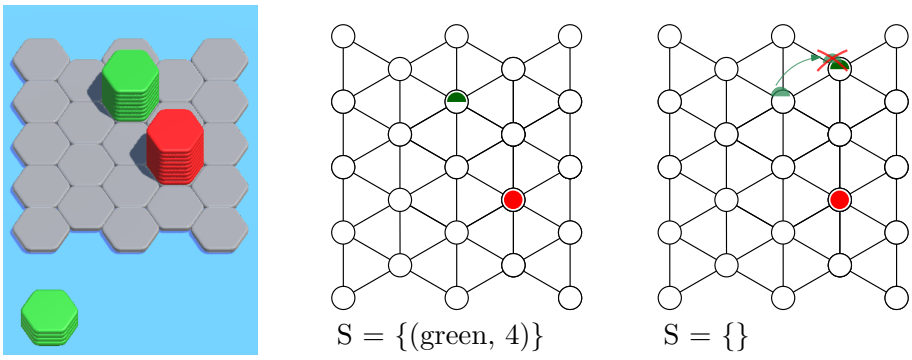
Funding Vienna Science and Technology Fund (WWTF) grant [10.47379/ICT22029] and Austrian Science Fund (FWF) grant [10.55776/Y1329]

1 Introduction

We study the computational complexity of the single-player stacking game Hexasort. There exist many different browser-based or mobile phone app implementations,[1] often with various variations to the mechanics. Common to all is that they are played on a hexagonal game board, where a given sequence of stacks of colored hexagons need to be placed onto the empty cells. Whenever the topmost hexagons of adjacent stacks have the same color, all hexagons of said color move from one stack onto the other; see Figure 1. If this results in sufficiently many same-colored hexagons on top of each other, all of them vanish. The goal of the game is to place the full input sequence without running out of empty cells.

In our model of the game, we generalize the game board to an arbitrary graph, where the colored stacks are placed onto its vertices. We restrict our consideration to monochromatic stacks and define that whenever a stack is placed adjacent to stacks of the same color, the old stacks completely move onto the new one. We will not consider any further complications to the mechanics such as initially non-empty game boards and blockers that can only be removed after having performed a certain amount of merges, although we will later show

[1] See e.g. hexasort.io, games-by-sam.com/bee-sort-by-sam, or spielaffe.de/Spiel/Hexa-Sort-3D.



(a) Original game screenshot. **(b)** Graph abstraction. **(c)** Result after placing ◓.

Figure 1 Example of a FITTING HEXASORT game with its graph representation (not showing stack heights). After placing ◓, the green stacks merge, reach t and vanish.

that similar mechanics can be achieved through gadgets. An instance (G, S, t) of HEXASORT thus consists of a graph $G = (V, E)$, a *threshold* $t \in \mathbb{N}^+$, and a sequence S of *stacks* having colors C, where each stack $s = (c, h) \in S$ has a *color* $c \in C$ and a *height* $h \in \mathbb{N}^+$.

A *configuration* of the game assigns to each vertex either a stack or nothing, that is, it is a function $\delta : V(G) \to C \times \mathbb{N}^+ \cup \ \bot$ where $\delta(v) = \bot$ if vertex v is currently *empty*, and otherwise $\delta(v) = (c, h)$ if v hosts a stack of color c and height h. Initially, we have $\delta(v) = \bot$ for all $v \in V$. To place the next stack $s_i = (c, h)$ from S, the player chooses an empty vertex x. Let $N_c[x]$ denote the non-empty neighbors of x that have color c and let h' be the sum of their heights. We obtain a new configuration δ' from δ by setting $\delta'(u) = \bot$ for all $u \in N_c[x]$ and updating $\delta'(x)$ depending on the accumulated height as follows. If $h + h' \geq t$, we set $\delta'(x) = \bot$, otherwise we set $\delta'(x) = (c, h + h')$.

The central algorithmic question of the problem EMPTY HEXASORT is whether all stacks in S can be placed on the graph in their given order such that all vertices are empty again after S is exhausted. The problem variant FITTING HEXASORT relaxes this to only placing all stacks in S in order, without the requirement for all vertices to be empty afterwards. Note that any instance of EMPTY HEXASORT can be turned into an equivalent one of the FITTING variant by appending $|V|$ stacks to S, all having new and distinct colors as well as a height smaller than t. Conversely, any yes-instance of EMPTY HEXASORT is trivially also a yes-instance for the FITTING variant.

This paper explores the boundary between tractable and intractable instances of FITTING and EMPTY HEXASORT. In Section 2, we show that EMPTY HEXASORT is NP-hard, even under severe structural restrictions. In particular, the problem is weakly NP-hard with one color on graphs consisting of two independent edges, and strongly NP-hard on trees of either bounded height or degree. Towards fixed-parameter tractable algorithms, in Section 3 we identify settings in which both variants become polynomial-time solvable and present a dynamic programming approach with running time exponential in the size of the input graph.

1.1 Related Work

Several games with mechanics modeling partial aspects of HEXASORT, such as placing objects on graphs, monochromatic components, as well as merging and elimination thresholds, have been investigated. To the best of our knowledge, the combination of these mechanics and

especially the computational complexity of Hexasort has not been studied previously. As we will see, the combination of non-monotone development of game states, threshold-based elimination that discards any excess, together with having to eliminate all stacks in Empty Hexasort poses an especially interesting challenge.

A recent example of a game involving the placement of objects on a graph is Matching Match Puzzle, analyzed by Iburi et al. [8]. In this puzzle, the player is given a colored graph and a set of matchsticks, each having a color assigned to both of its endpoints. The task is to place the matchsticks onto the edges of the graph such that, at every vertex, all incident matchstick endpoints agree on a single color consistent with the given graph coloring. The problem is NP-hard even on spiders and, interestingly, this result is established via a reduction from 3-Partition, similar to the reduction used in Section 2.2.

One well-studied example of a game relying on monochromatic components is the game Flood-It. A survey by Fellows et al. [4] summarizes complexity results for two variants known as Flood-It and Free-Flood-It. Both are played on graphs with colored vertices. In Flood-It, a fixed pivot vertex is repeatedly recolored, causing its whole monochromatic connected component to change color as well. In Free Flood-It, the vertex whose monochromatic component shall be recolored can be freely chosen. In both cases, the objective is to make the entire board monochromatic using as few moves as possible. Flood-It is already NP-hard on boards of size $3 \times n$ with four colors.

A game that combines merging and vanishing mechanics is Clickomania. In Clickomania, the player is presented a grid of colored blocks. A move consists of selecting a monochromatic connected component of at least two blocks, which removes them from the board. Blocks above the removed blocks fall down, and empty columns vanish. The objective is to clear the board completely or minimize the number of remaining blocks. Clickomania remains NP-hard even when restricted to only two colors and two columns [1].

Outside the algorithmic study of games, the merging and threshold-based elimination of stacks in Hexasort exhibits parallels with the behavior of physical water droplets on digital microfluidics platforms like OpenDrop [2]. Here, a grid of electrodes is used to manipulate discrete fluid droplets. By activating specific electrodes, droplets can be moved, mixed, and merged on the surface. Due to surface tension and physical constraints, a single electrode tile can only support a specific volume of liquid; exceeding this volume makes the droplet unstable. In this analogy, stack height corresponds to droplet volume, and merging occurs based on proximity. In addition to medical and chemical applications, the device has also been gamified using colored droplets [12].

1.2 Preliminaries

We distinguish between sets (unordered collections, denoted by $\{\dots\}$) and sequences (ordered lists, denoted by $\langle\dots\rangle$). For two sequences $A = \langle a_1, \dots, a_n\rangle$ and $B = \langle b_1, \dots, b_m\rangle$, we define their *concatenation*, denoted as $A+B$, as the new sequence formed by appending the elements of B to the end of A, so $A + B = \langle a_1, \dots, a_n, b_1, \dots, b_m\rangle$.

We use the term *three-merge* to denote a move that places the current stack such that it has exactly two adjacent non-empty vertices of the same color. For a color c, we denote $sum(c, S)$ as the sum of heights of color c currently remaining in S.

We will assume that $h \leq t$ for all $(c, h) \in S$, as any stacks exceeding t can equivalently be capped to have height t. Observe that no two stacks that are adjacent on G can have the same color in a configuration δ. Finally, in Empty Hexasort, we assume that for any $c \in C$, we have $\text{sum}(c, S) \geq t$, as the instance is trivially negative otherwise.

2 NP-Hardness

Our first proof that Empty Hexasort is NP-hard is via a reduction from the weakly NP-complete problem Partition to Empty Hexasort with a single color on graphs consisting of only two independent edges in Section 2.1. We afterwards extend this hardness result to connected graphs with nine vertices. We further show strong NP-hardness via a reduction from 3-Partition to Empty Hexasort on trees of bounded height or degree in Section 2.2. Thereby, we show hardness even when either the graph size and number of colors are small constants, or when all numeric inputs are polynomially bounded in the input size.

The figures in this section employ a visual encoding for colored stacks to ensure accessibility:  , , , and depict red, blue, green, and black stacks, respectively. Note that the exact stack heights are usually not relevant for the figures and thus not shown.

2.1 Reduction from Partition

In the Partition problem, we are given a multiset P of positive integers and must decide whether P can be partitioned into two disjoint subsets P_1 and $P_2 = P \setminus P_1$ such that $\sum P_1 = \sum P_2$. The Partition problem is a well-known special case of the Subset Sum problem and is (weakly) NP-hard [6]. We restrict our attention to Partition instances where (1) the sum of elements $\sum P$ is even, (2) no element $p \in P$ satisfies $p \geq \sum P/2$, and (3) the desired partition (if it exists) is not formed by the sum of a prefix of P. Note that instances that violate one of these criteria can be trivially decided in polynomial time.

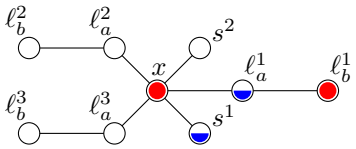
Partition can be reduced to Empty Hexasort as follows. Let $P = \{p_1, p_2, \ldots, p_n\}$ be an arbitrary instance of Partition. We construct a corresponding instance $I_H = (G, S, t)$ of Empty Hexasort with threshold $t = \sum P/2$ where the graph $G = (V, E)$ has four vertices $V = \{v_1, v_2, v_3, v_4\}$ and two edges $E = \{v_1v_2, v_3v_4\}$. The input sequence of stacks S is defined as $S = \langle (c, p_1), (c, p_2), \ldots, (c, p_n) \rangle$, where c is the only color.

▶ **Theorem 1.** *Empty Hexasort on two independent edges and a single color is weakly NP-hard.*

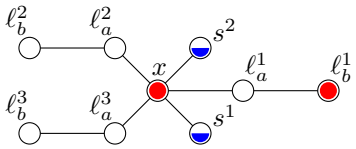
Proof. In our above construction, it is easy to see that any solution to P can be translated to a solution of the derived I_H by alternatingly placing the stacks of the first partition on v_1 and v_2 while placing those of the second on v_3 and v_4. As each partition sums up to exactly $\sum P/2 = t$, the stacks vanish exactly once the last element of the partition is placed.

For the converse direction, we first show that placing all stacks of I_H exclusively on the endpoints of one of the two edges cannot empty the graph. Recall that we excluded the trivial case of a prefix of P forming the desired partition. Thus, placing all stacks on one edge will at some point form a stack that exceeds t by a value of at least 1 and consequently vanishes. Afterwards, the remaining stacks sum up to height at most $t - 1$ and thereby cannot vanish. Thus, the stacks of I_H need to be distributed onto the two independent edges. Assuming one edge receives stacks exceeding t leaves less than t for the other edge. ◀

Note that above approach could easily be extended to connected graphs if we allowed a given initial configuration with some already-placed differently-colored stack keeping both edges separate until the end. Instead, we construct a gadget to adequately restrict the placement of stacks on a more involved but connected graph structure. The goal is to argue that a given sequence of stacks can only be placed in such a way that two specific, non-adjacent paths of length two remain available for processing the Partition input sequence. We adapt our above construction to obtain an instance $I_H^* = (G', S', t)$ where G' is a connected *spider*



(a) After placing $S_{initial}, S_{reservation}$ and S the next red stack in $S_{connector}$ cannot three-merge the red stacks, violating Lemma 2.



(b) The admissible configuration of Lemma 4.

Figure 2 Labeled graph G' with red and blue stacks placed. The configuration depicted in (a) cannot lead to an empty graph, while (b) can.

graph, that is a tree with only one vertex of degree greater than two. Our specific spider graph G' has nine vertices: one central vertex x with degree five, two *short arms* consisting of the vertices s^1, s^2, respectively, and three *long arms* of length two; see Figure 2. For long arm $i \in \{1, 2, 3\}$, let ℓ_a^i be the neighbor of x while ℓ_b^i is the leaf.

We will now use the four colors: c_{black} (for the PARTITION input in S), and c_{red}, c_{blue}, c_{green} (for the *gadget stacks*). Define the sequence of the initial gadget stacks as $S_{initial} = \langle (c_{\text{red}}, t-1), (c_{\text{red}}, t-1), (c_{\text{blue}}, t-1), (c_{\text{blue}}, t-1) \rangle$. Define the reservation gadget stacks as $S_{reservation} = \langle (c_{\text{green}}, \lfloor \frac{t}{2} \rfloor + 1), (c_{\text{green}}, \lfloor \frac{t}{2} \rfloor + 1), (c_{\text{green}}, \lceil \frac{t}{2} \rceil - 1), (c_{\text{green}}, \lceil \frac{t}{2} \rceil - 1) \rangle$. Define the sequence of the connector gadget stacks $S_{connector}$ which frees the gadget of the initial stacks as $S_{connector} = \langle (c_{\text{red}}, t-1), (c_{\text{blue}}, t-1) \rangle$. We construct the full input sequence as $S' = S_{initial} + S_{reservation} + S + S_{connector}$. The idea behind our construction is that the gadget stacks (non-c_{black}) must vanish to empty the graph, but block most of the graph while the c_{black} stacks of S are processed, such that they need to be placed on the four endpoints of exactly two independent edges. To formally show this, we need the following lemmas, which will also be useful for later settings.

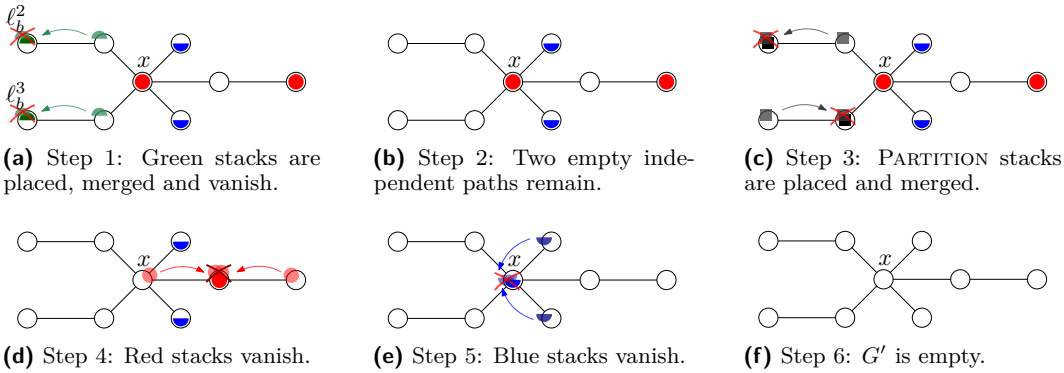
▶ **Lemma 2** (Forced three-merge). *Let c be a color of an* EMPTY HEXASORT *instance $I = (G, S, t)$ such that S contains exactly three stacks $(c, h_1), (c, h_2), (c, h_3)$ of color c in the given order and where $h_i < t$ for $i \in 1, 2, 3$ while $h_1 + h_2 \geq t$. In any solution to I, the first and second stack need to be placed such that they are non-adjacent but have a common neighbor that must be empty when the third stack is placed.*

Proof. If the first two stacks are placed on adjacent vertices, they merge immediately and vanish, leaving the third stack unremoved at the end. If instead they are placed more than one vertex apart, they cannot later form a three-merge. Hence, exactly one vertex must separate them, onto which the third stack is placed to remove all of them in one go. ◀

Thus, two colors to which Lemma 2 applies cannot alternate on a path as shown in Figure 2a.

▶ **Lemma 3.** *Let c be a color of an* EMPTY HEXASORT *instance $I = (G, S, t)$ such that S contains exactly four stacks $(c, h_\ell), (c, h_\ell), (c, h_s), (c, h_s)$ of color c in the given order and where $h_s < \frac{t}{2} < h_\ell$ such that $h_s + h_\ell \geq t$. In any solution to I, these stacks can vanish either (a) via two independent merges of an h_ℓ-stack with an h_s-stack, or (b) by merging all four stacks in one move. In case (a), the stacks are placed on the endpoints of two independent edges, in case (b) on a star of degree 3.*

Proof. Every color-c stack is smaller than t, so no stack can vanish on its own. Any merge involving exactly three color-c stacks would cause precisely those three stacks to vanish, leaving a single stack that can never vanish. Merging the first two stacks (of height $2h_\ell > t$)



(a) Step 1: Green stacks are placed, merged and vanish.

(b) Step 2: Two empty independent paths remain.

(c) Step 3: PARTITION stacks are placed and merged.

(d) Step 4: Red stacks vanish.

(e) Step 5: Blue stacks vanish.

(f) Step 6: G' is empty.

Figure 3 Spider graph stack placement progression leading to an empty graph.

would cause them to vanish, leaving the latter two stacks of total height $2h_s < t$ that cannot vanish. In contrast, two stacks of different height sum to at least t and thus vanish. Thus, only the latter two h_s stacks could be merged without vanishing, although this leaves no chance of merging them with the former h_ℓ ones to make them vanish. Altogether, this shows the color-c stacks need to vanish in either (a) two pairs of two, or (b) all four in one go. In case (a), we first need to place the first two stacks without them merging, and then the third stack so that it only merges with one of them, necessitating the first edge. Finally placing the last stack adjacent to the leftover h_ℓ stack requires the second, independent edge. For case (b), the only way of merging all four stacks in one move is placing them on a star of degree 3, with the last stack placed on the center. ◀

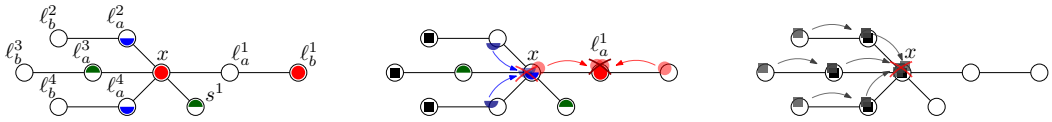
Together, this allows us to show the following lemma; see also Figure 2b.

▶ **Lemma 4.** *After placing $S_{initial} + S_{reservation}$ in $I_H^* = (G', S', t)$, any solution needs to have the c_{red} stacks on x and ℓ_b^1, the c_{blue} stacks on s^1, s^2 and all other vertices empty.*

Proof. First, suppose that vertex x remains empty after placing $S_{initial}$, i.e., the first two c_{red} and two c_{blue} stacks. Lemma 2 now implies that four neighbors of x must be occupied by the c_{red} and c_{blue} stacks, while all leaf vertices are empty. However, occupying four neighbors of x leaves no space for the c_{green} stacks, inevitably violating Lemma 3. Hence, the central vertex x must host a stack of color either c_{red} or c_{blue}, which means that we only need to consider case (a) of Lemma 3.

After placing $S_{initial}$ vertex x holds the stack $(c_a, t-1)$ where $c_a \in \{c_{\text{red}}, c_{\text{blue}}\}$. From Lemma 2 it follows that the second c_a stack of $S_{initial}$ must be placed at ℓ_b^i for some $i \in \{1, 2, 3\}$, say w.l.o.g. ℓ_b^1. We now show that the two other stacks of $S_{initial}$, call their color c_b, must be placed on s^1 and s^2. First, if a c_b stack is placed on any vertex of ℓ^2 or ℓ^3, there only remains one path of length two, which violates Lemma 3. Second, if a c_b stack is placed on ℓ_a^1, between the two existing c_a stacks, this violates Lemma 2, as ℓ_a^1 must remain empty for the c_a three-merge. Thus, only s^1 and s^2 remain.

Finally, we show that $c_a = c_{\text{red}}$ and $c_b = c_{\text{blue}}$. Recall that Lemma 2 especially means that the common neighbor of the first two stacks needs to be empty when placing the third on it. As the third c_{red} stack appears before the third c_{blue} stack in $S_{connector}$, the common neighbor of the two previous c_{red} stacks needs to be empty at that point before placing $S_{connector}$. This is only the case when x and ℓ_b^1 are c_{red} while s^1 and s^2 are c_{blue}. Removing the c_{red} stacks then frees up x as common neighbor of the c_{blue} stacks. ◀



(a) The admissible configuration of Lemma 6.

(b) Removing the gadget stacks ●, ◒ and finally ◓.

(c) Pulling the ■ stacks to merge and vanish on x.

Figure 4 The spider gadget used in our 3-Partition reduction.

Figure 3 visualizes how the two edges that are empty after Lemma 4 can be used to solve the partition instance encoded in S and then clear the graph via $S_{connector}$. In the full version [10], we show that any solution must make these steps, thereby obtaining the following theorem.

▶ **Theorem 5.** *Empty Hexasort on spider graphs with nine vertices and four colors is weakly NP-hard.*

2.2 Reduction from 3-Partition

Note that by reducing from Partition in the last section, we rely on large numbers and therefore can only show weak NP-hardness. In this section, we now want to show strong NP-hardness, that is hardness even when t (and thereby also all stack heights) is bounded by a polynomial in the input size [5]. For this, we use a reduction from the strongly NP-hard problem 3-Partition, where we are given a multiset A of $3m$ positive integers and a bound $B \in Z^+$. The question is whether A can be partitioned into m triplets $A_1, A_2, \ldots, A_m$ such that each triplet sums up to exactly B. As noted by Garey et al. [6, p. 224] the problem is NP-complete in the strong sense, and we can assume that every input value $a \in A$ is restricted such that $B/4 < a < B/2$ and $\sum_{a \in A} a = mB$.

For a reduction from 3-Partition to Empty Hexasort, we will adapt our prior gadget to now leave exactly three distinct, isolated vertices available for the placement of the three elements of a single triplet. Using m disjoint copies of this gadget, we can thereby model the m triplets in a 3-Partition solution. More specifically, our reduction works as follows. A gadget consists of one central vertex x of degree five, four arms of length two with vertices ℓ_a^i, l_b^i for $i \in \{1, 2, 3, 4\}$, and one arm of length one, s^1; see Figure 4.

Given a 3-Partition instance (A, B) we obtain an Empty Hexasort instance $I_H = (G, S, t)$ where graph G has $m = \frac{|A|}{3}$ disjoint copies of the gadget graph and the threshold is $t = B + 4$. For S, we use $3m + 1$ colors: c_{black} for the triplet elements, and three gadget colors c^j_{red}, c^j_{blue}, c^j_{green} for $j \in \{1, \ldots, m\}$. The input sequence is constructed as follows.

- The initial gadget stack sequence $S^j_{initial} = \langle (c^j_{\text{red}}, t-1), (c^j_{\text{red}}, t-1), (c^j_{\text{blue}}, t-1), (c^j_{\text{blue}}, t-1), (c^j_{\text{green}}, t-1), (c^j_{\text{green}}, t-1) \rangle$ forces the isolation of three vertices.
- The sequence $S_A = \langle (c_{\text{black}}, a_1), \ldots, (c_{\text{black}}, a_{3m}) \rangle$ represents $A = \{a_1, \ldots, a_{3m}\}$.
- The connector gadget stack sequence $S^j_{connector} = \langle (c^j_{\text{red}}, t-1), (c^j_{\text{blue}}, t-1), (c^j_{\text{green}}, t-1) \rangle$ frees each gadget of the (initial) gadget stacks.
- The triplet connector sequence $S^j_{final} = \langle (c_{black}, 1), (c_{black}, 1), (c_{black}, 1), (c_{black}, 1) \rangle$ draws the placed triplet, causing all its stacks to merge and vanish.

We construct the full input sequence as $S = S_{initial} + S_A + S_{connector} + S_{final}$, where $S_{initial}$, $S_{connector}$ and S_{final} are obtained by concatenating all $S^j_{initial}$, $S^j_{connector}$, and S^j_{final}, respectively, for all $j \in \{1, \ldots, m\}$. Observe that Lemma 2 still applies to all c^j_{blue}, c^j_{red}, and c^j_{green}. Analogously to Lemma 4, we again show that $S_{initial}$ forces a certain configuration; see Figure 4a and the full version [10].

▶ **Lemma 6.** *After placing $S_{initial}$ in I_H, any solution has, for each gadget, the vertices ℓ_a^i and all $\ell_b^{j\neq i}$ empty for some $i \in \{1,2,3,4\}$. Furthermore, the two neighbors of ℓ_a^i have the same color c, and each gadget consists of exactly three colors, where c appears first in S_{final}.*

Proof sketch. First observe that no color can be split onto multiple disconnected gadgets due to Lemma 2. Using a case analysis together with the pigeonhole principle, one can then show that every gadget contains exactly six stacks, which then need to use exactly three colors. The positions of empty vertices then follow similar to Lemma 4. ◀

We will again assume without loss of generality that ℓ_a^1 as well as ℓ_b^2, l_b^3, l_b^4 are always the empty vertices. As each gadget received exactly three colors, where each color needs to be put onto a single gadget, the exact position of a color within $S_{initial}$ and S_{final} is irrelevant, as long as c appears first in S_{final}. We assume w.l.o.g. that the j-th gadget received colors c^j_{red}, c^j_{blue}, c^j_{green} such that its x is c^j_{red}, while s^1 and ℓ_a^2 are c^j_{green}; see Figure 4a. We can now show the following; see [10] for the full proof.

▶ **Theorem 7.** *Empty Hexasort on the disjoint union of spider graphs with ten vertices each is strongly NP-hard.*

Proof sketch. We focus on showing that any solution to I_H implies a solution to (A, B), continuing where Lemma 6 left off. Here, as ℓ_a^1 cannot be used by S_A due to Lemma 2, exactly the three vertices ℓ_b^2, l_b^3, and ℓ_b^4 remain available for placing S_A. It is easy to see that, again due to pigeonhole principle, exactly three stacks from S_A must be placed on them. Next, the stacks of $S_{connector}$ are placed. Since for each gadget j now only ℓ_a^1 is empty, the corresponding c^j_{red} connector stack is forced onto this vertex, merging the red triplet. The c^j_{blue} and c^j_{green} connector stacks are then forced to three-merge on x, see Figure 4b. This leaves the gadget empty except for ℓ_b^2, l_b^3, and ℓ_b^4 containing stacks from S_A. Emptying the gadget then requires at least four c_{black} height-1 stacks from S_{final} to pull the triplet stacks onto x; see Figure 4c. By the pigeon hole principle, each gadget can receive at most four such stacks, contributing a value of 4 to the required height of $t = B + 4$. Thus, each gadget can only be emptied if its three stacks from S_A sum up to at least (or rather exactly) B, implying a 3-Partition solution to (A, B). Our reduction can be performed in polynomial time and maintains the numerical values and especially their bounds from (A, B), thereby showing strong NP-hardness. ◀

We again want to extend our result to connected graphs. For this, we will connect the s^1 vertices of all gadgets in a tree-like manner: Let T be a tree on m vertices $t_1, \ldots, t_m$. For the Empty Hexasort instance $I_H^* = (G', S, t)$, the connected graph G' is obtained by combining the m gadgets of G from above with T: for all $j \in \{1, \ldots, m\}$, identify the s^1 vertex of the j-th gadget in G with the tree node t_j in T. Observe that, if T has bounded height or degree, so does G'. It remains to show that Lemma 6 still applies, which in this case hinges on stacks of the same color not being distributed over multiple gadgets.

▶ **Lemma 8.** *After placing $S_{initial}$ in I_H^*, any solution needs to have all stacks of exactly three colors on each gadget.*

Proof sketch. Note that due to Lemma 2, the only vertices that can host a (non-black) color that is split over multiple gadgets are the x and s^1 ones. The same lemma means that placing a stack on ℓ_b^i forces a same-colored stack on x, which is only possible once, and that ℓ_a^i remains empty. Thus, no gadget can have 7 or more stacks, which via the pigeon hole principle implies that all gadgets have exactly 6 stacks. From this, we can conclude that, if either of x and s^1 hosts a color that only appears once on the respective gadget, so does the other with a different color that only appears once on the gadget.

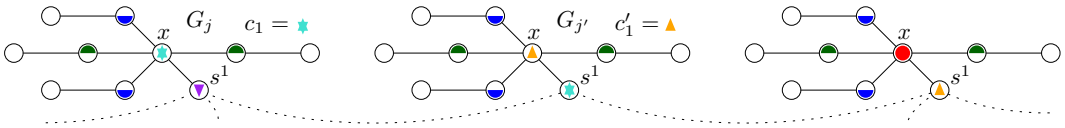


Figure 5 A connected graph G' with stacks placed across multiple gadgets excluded by Lemma 8.

We will now take an arbitrary such pair, remove it from the graph, and show that the smaller remainder graph must always contain another such pair, which contradicts G' being a finite acyclic graph. Consider a x vertex of gadget j that has a color c_1 that only appears once on the gadget. Due to Lemma 2, the second stack of c_1 needs to be on the s^1 vertex of gadget j' such that t_j and $t_{j'}$ are adjacent in T; see Figure 5. As noted above, vertex x of gadget j' needs to have a different color c_1' that only appears once on that gadget. Especially, the s^1 vertex of gadget j cannot have color c_1' as otherwise it would alternate with c_1, violating Lemma 2. We remove gadget j, and continue by considering the pair of color c_1' on the smaller, connected graph that is the connected component of the remaining gadget j'. This yields a contradiction as this argument can continue indefinitely while our graph is finite. ◀

The remainder of Lemma 6 and thereby the following theorem now follows; see the full version [10] for an explicit proof.

▶ **Theorem 9.** *Empty Hexasort on trees of bounded height or degree is strongly NP-hard.*

3 Algorithms

In this section we investigate algorithmic approaches towards solving Hexasort. Note that a trivial brute-force approach to decide an instance of Fitting or Empty Hexasort runs in $O(|V|^{|S|})$ time: We try for each stack in the sequence S each vertex on which it could be placed and then check whether the final configuration is admissible. We develop a dynamic programming approach to Hexasort by instead systematically exploring the space of reachable board configurations, yielding running time exponential in the graph size.

▶ **Theorem 10.** *Fitting and Empty Hexasort can be decided in time* $\mathcal{O}(|S| \cdot (|C| \cdot t)^{|V|})$.

Proof. The core of our approach is a Boolean table, whose entry $\mathrm{DP}[i, \delta]$ is true if and only if the board configuration δ is attainable after placing the first i stacks of S. We initialize the dynamic program by setting only $\mathrm{DP}[0, \delta_0] = \text{true}$ for the unique empty configuration δ_0 where $\delta_0(v) = \perp$ for all $v \in V$. The recurrence relation processes the sequence step-by-step: for each index i from 1 to m and every configuration δ such that $\mathrm{DP}[i-1, \delta] = \text{true}$, we consider every possible valid move. A move consists of placing the current stack $s_i = (c, h)$ on any *empty* vertex and sets $\mathrm{DP}[i, \delta'] = \text{true}$ for the resulting configuration δ'. An Empty Hexasort instance is positive if and only if $\mathrm{DP}[m, \delta_0] = \text{true}$ upon exhaustion of the sequence S. A Fitting Hexasort instance is positive if $\mathrm{DP}[m, \delta_0] = \text{true}$ holds for any δ.

Regarding the running time, observe that there are $\mathcal{O}((|C| \cdot t)^{|V|})$ configurations as each of the $|V|$ vertices has $|C| \cdot (t-1) + 1$ possible states. Since the algorithm iterates through the sequence and considers up to $|V|$ placement options for each reachable state, the total time complexity is $\mathcal{O}(|S| \cdot (|C| \cdot (t-1) + 1)^{|V|} \cdot |V|) = \mathcal{O}(|S| \cdot (|C| \cdot t)^{|V|})$. ◀

While this algorithm becomes slower the larger the graph becomes, we will now show that once the graph becomes large enough to provide dedicated space for each color, the problem becomes trivially positive. For this, we need separate approaches for both problem variants.

▶ **Lemma 11.** *Any* Fitting Hexasort *instance where G contains a matching of size at least $|C|$ is positive.*

Proof. For each color $c \in C$, we reserve an edge from the matching. When a stack (c, h) arrives, it is placed alternatingly on one of the two endpoint vertices. Thereby, for each color at any point in time, at least one free vertex is available. Thus, all stacks can be placed. ◀

▶ **Lemma 12.** *Any* Fitting Hexasort *instance where G contains a vertex of degree $|C| \cdot t$ is positive.*

Proof. If G contains a vertex x of degree at least $|C| \cdot t$, we can place the stacks from $|S|$ on any of its neighbors, only placing a stack directly on x whenever its color has already t stacks adjacent, thereby eliminating all stacks of this color. As no color ever has more than t stacks on the graph and any stack placed on x immediately vanishes (the merge of its t neighbors surely exceeds size t), all stacks can be placed and the instance is thus always positive. ◀

▶ **Observation 13.** *Any* Empty Hexasort *instance where the stacks of some color c sum up to less than t when ignoring all height-t stacks is negative if the last color-c stack has height less than t.*

▶ **Lemma 14.** *Any* Empty Hexasort *instance, that is not trivially negative by Observation 13, and where G contains $|C|$ non-overlapping spiders, each with two length-two legs and one length-one leg, is positive.*

Proof. For each color $c \in C$, we reserve a degree-3 spider with center x adjacent to the first vertices of the three legs s, $\ell_a^1\ell_b^1$, $\ell_a^2\ell_b^2$. Let $z = (c, h_z)$ be the last color-c stack in S. If $h_z = t$ or $sum(c, S) - h_z < t$, place all color-c stacks alternatingly on any two adjacent vertices of the spider. Otherwise, let X be an arbitrary minimal set of color-c stacks, excluding any height-t stack, such that $\Sigma X \geq t$. Note that by assuming that Observation 13 does not make the instance trivially negative, such set must exist. Now let $y = (c, h_y)$ be the first stack of X in S, which we place on s. Place all other stacks of X, except for z, alternatingly on ℓ_a^1 and ℓ_b^1 such that the last stack of X is placed on ℓ_a^1. Place all other stacks of color c, except for z, alternatingly on ℓ_a^2 and ℓ_b^2 such that the last of them is on ℓ_a^2. Place the last stack z on the center x. By minimality of X, whether or not $z \in X$, at this point s contains y and ℓ_a^1 a stack that reaches height at least t when merged with y (or with z and y). When z is placed on x, it merges ℓ_a^1 and s with any remaining stack on ℓ_a^2, which makes color c vanish. Consequently, all stacks in S can be placed while emptying the graph in the end. ◀

For Fitting Hexasort, combining these results now allows us to show fixed-parameter tractability when parametrizing by the number of colors and the threshold value.

▶ **Theorem 15.** *Fitting Hexasort can be decided in time $\mathcal{O}(f(|C|, t) \cdot |S|^2)$ for some function f.*

Proof. First note that if G is small enough such that $|V|$ is bounded by $|C|$, the instance can be solved in $\mathcal{O}((|C| \cdot t)^{|C|} \cdot |S|)$ by Theorem 10. If G is instead large and contains a matching of size at least $|C|$, the instance is always positive by Lemma 11. The existence of such matching can be checked in polynomial time; see [3] for a survey of algorithms. Also, by Lemma 12, all instances with a vertex of degree at least $|C| \cdot t$ are always positive. It remains to treat the case where G is large but has bounded degree and does not contain a large enough matching. Equivalently, the latter means that G has a vertex cover X of a size bounded by $2 \cdot |C|$ [9, 11].

Let V^0 denote the set of isolated vertices. It is easy to see that the instance is always positive if $|V^0| \geq |S|$ – place every stack on a single vertex. Thus G has a bounded vertex cover X, no high-degree vertex, and a bounded number of isolated vertices V^0. This implies that, disregarding the isolated vertices V^0, graph G has a size bounded by $|C|^2 \cdot t$. This is because each of the at most $2 \cdot |C|$ vertices of X can have at most $|C| \cdot t$ neighbors outside X, and any non-isolated vertex needs to either be in X or have a neighbor in X. The running time of the dynamic program from Theorem 10 can be factored as $\mathcal{O}(|S| \cdot (|C| \cdot t)^{|V \setminus V^0|} \cdot (|C| \cdot t)^{|V^0|})$. As isolated vertices are structurally identical and cannot affect each other, we do not need to track their specific configurations. Instead, we simply count the number of non-empty isolated vertices and thus simplify the exponential factor $((|C| \cdot t)^{|V^0|})$ with the linear factor $|V^0|$. This yields a running time of $\mathcal{O}(|S| \cdot (|C| \cdot t)^{|V \setminus V^0|} \cdot |V^0|)$. Since $|V \setminus V^0|$ is bounded by $|C|^2 \cdot t$ and $|V^0|$ is bounded by $|S|$ the complexity simplifies to $\mathcal{O}((|C| \cdot t)^{|C|^2 \cdot t} \cdot |S|^2)$ satisfying our bound in this final case. ◀

4 Conclusion and Future Work

In this work, we studied the computational complexity of the popular game Hexasort in the Empty and Fitting variants. In Section 2 we presented multiple NP-hardness reductions, starting with a reduction from the Partition problem to Empty Hexasort with one color on two independent paths of size two. Reducing Empty to Fitting Hexasort by appending four stacks with new colors to such instance, this result also applies to the latter variant when allowing five colors. We further used 3-Partition to show that Empty Hexasort is strongly NP-hard, even when restricted to trees of bounded height or degree. This demonstrates that the problem remains intractable even under severe structural restrictions on the graph, together with either a constant number of colors or numbers that are polynomially-bounded by the input size.

At the same time in Section 3, we identified several classes of instances that admit efficient algorithms. We introduced a dynamic programming algorithm that solves both variants on arbitrary graphs in time $\mathcal{O}\big(|S| \cdot (|C| \cdot t)^{|V|}\big)$. Note that this immediately implies tractability on constant-size graphs when the threshold t is polynomially-bounded by the input size. We also showed that instances with sufficiently large graphs, which allow each color to be solved independently on its own exclusive part of the graph, are always tractable. Based on this, we were able to show that Fitting Hexasort is fixed-parameter tractable when parametrizing by the number of colors and the threshold value.

For Fitting Hexasort we thus obtain results for bounding any two of the graph size $|V|$, the number of colors $|C|$ and the threshold t. While the problem is (weakly) NP-hard for constant $|V|$ and $|C|$, it becomes tractable for either constant $|V|$ and polynomially-bounded t or for constant $|C|$ and t. We leave the complexity of the case where t is bounded by a constant as open question. Furthermore, it would be nice to extend our results to show also strong NP-hardness for constant $|C|$. Similarly, our results for Empty Hexasort are tight in the sense that, when bounding both the graph size by a constant as well as the elimination threshold by a polynomial in the input size, we obtain a polynomial-time algorithm. In contrast, only applying either of the bounds yields NP-hardness. We leave as an open question whether Empty Hexasort becomes tractable when bounding both the threshold and the number of colors, e.g. by using Lemma 14 as we did for the Fitting case. The main challenge here is arguing that any graph containing only a few copies of this more involved structure needs to have a simple structure itself.

Regarding the Fitting Hexasort instances found in playable versions of the game, their hexagonal grid is usually large enough w.r.t. the number of colors to allow for a sufficiently large matching. What instead complicates these instances is that stacks here are also allowed to contain hexagons of multiple different colors, which is a further direction in which our results could be extended.

References

1 Aviv Adler, Erik D. Demaine, Adam Hesterberg, Quanquan Liu, and Mikhail Rudoy. Clickomania is hard even with two colors and columns. In *The Mathematics of Various Entertaining Subjects (MOVES 2015)*, volume 2, pages 325–363. Princeton University Press, 2017. doi:10.23943/princeton/9780691171920.003.0018.

2 Mirela Alistar and Urs Gaudenz. Opendrop: An integrated do-it-yourself platform for personal use of biochips. *Bioengineering*, 4(2), 2017. doi:10.3390/bioengineering4020045.

3 Ran Duan and Seth Pettie. Linear-time approximation for maximum weight matching. *J. ACM*, 61(1), January 2014. doi:10.1145/2529989.

4 Michael R. Fellows, Frances A. Rosamond, Maise Dantas da Silva, and Uéverton S. Souza. *A Survey on the Complexity of Flood-Filling Games*, pages 357–376. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-319-98355-4_20.

5 Michael R. Garey and David S. Johnson. "Strong" NP-completeness results: Motivation, examples, and implications. *Journal of the ACM (JACM)*, 25(3):499–508, 1978. doi:10.1145/322077.322090.

6 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979. See Appendix A3, problem SP12 (PARTITION), p. 223.

7 Luciano Gualà, Stefano Leucci, and Emanuele Natale. Bejeweled, candy crush and other match-three games are (NP-)hard. In *Proceedings of the 2014 IEEE Conference on Computational Intelligence and Games (CIG 2014)*, pages 1–8. IEEE, 2014. doi:10.1109/CIG.2014.6932866.

8 Yuki Iburi and Ryuhei Uehara. Computational complexity of matching match puzzle. In Andrei Z. Broder and Tami Tamir, editors, *Proceedings of the 12th International Conference on Fun with Algorithms (FUN 2024)*, volume 291 of *LIPIcs*, pages 17:1–17:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.FUN.2024.17.

9 Hugo Jacob, William Lochet, and Christophe Paul. On a tree-based variant of bandwidth and forbidding simple topological minors, 2025. doi:10.48550/arXiv.2502.11674.

10 Linus Klocker and Simon D. Fink. Hexasort – the complexity of stacking colors on graphs, 2026. arXiv:2603.01244.

11 Robert Sasak. Comparing 17 graph parameters. Master's thesis, The University of Bergen, 2010.

12 Steve Mould. Opendrop - electrowetting device demonstration [video]. https://www.youtube.com/watch?v=rf-efIZI_Dg, 2015.

13 Takayuki Yato and Takahiro Seta. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E86-A(5):1052–1060, 2003. URL: https://globals.ieice.org/en_transactions/fundamentals/10.1587/e86-a_5_1052/_p.