



HAL
open science

Optimal Mining: Maximizing Bitcoin Miners' Revenues

Mohsen Alambardar, Amir Goharshady, Mohammad Reza Hooshmandasl, Ali Shakiba

► **To cite this version:**

Mohsen Alambardar, Amir Goharshady, Mohammad Reza Hooshmandasl, Ali Shakiba. Optimal Mining: Maximizing Bitcoin Miners' Revenues. 2021. hal-03232783v1

HAL Id: hal-03232783

<https://hal.science/hal-03232783v1>

Preprint submitted on 22 May 2021 (v1), last revised 10 Oct 2021 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Optimal Mining*

Maximizing Bitcoin Miners' Revenues

Mohsen Alambardar
m.alambardar@sci.ui.ac.ir
University of Isfahan
Isfahan, Iran

Mohammad Reza Hooshmandasl
hooshmandasl@uma.ac.ir
University of Mohaghegh Ardabili
Ardabil, Iran

Amir Goharshady
goharshady@cse.ust.hk
The Hong Kong University of Science and Technology
Hong Kong, China

Ali Shakiba
ali.shakiba@vru.ac.ir
Vali-e-Asr University of Rafsanjan
Rafsanjan, Iran

Abstract

Following the Bitcoin model, many modern blockchains reward their miners in two ways: (i) a base reward for each block that is mined, and (ii) the transaction fees of those transactions that are included in the mined block. The base reward is fixed by the respective blockchain's protocol and is not under the miner's control. Hence, for a miner who wishes to maximize earnings, the fundamental problem is to form a valid block with maximal total transaction fees and then try to mine it. Moreover, in many protocols, including Bitcoin itself, the base reward halves at predetermined intervals, hence increasing the importance of maximizing transaction fees and mining an optimal block. This problem is further complicated by the fact that transactions can be prerequisites of each other or have conflicts (in case of double-spending). In this work, we consider the problem of forming an optimal block, i.e. a valid block with maximal total transaction fees, given a set of unmined transactions.

On the theoretical side, we first formally model our problem as an extension of *KNAPSACK* and then show that, unlike classical *KNAPSACK*, our problem is strongly NP-hard. We also show a hardness-of-approximation result. As such, there is no hope in solving it efficiently for general instances. However, we observe that its real-world instances are quite sparse, i.e. the transactions have very few dependencies and conflicts. Using this fact, and exploiting two well-known graph sparsity parameters, namely treewidth and pathwidth, we present exact linear-time parameterized algorithms that are applicable to the real-world instances and obtain optimal results.

We also provide an experimental evaluation demonstrating that our approach vastly outperforms the current Bitcoin miners in practice, obtaining a significant per-block average increase of 13.4% in transaction fee revenues.

1 Introduction

Mining. In blockchain ecosystems, *mining* is the process of adding new blocks of transactions to the public ledger (the blockchain). This terminology is usually applied to proof-of-work blockchains such as Bitcoin [39] and is sometimes even used to refer solely to the process of solving a hashcash puzzle. Blockchains that are not based on proof-of-work sometimes prefer other terms such as *farming* [20]. For the purposes of this paper, we consider the widest definition of mining that is not restricted to a specific consensus protocol such as proof-of-work, and distinguish between the two natural phases of mining:

1. In the first phase, the miner has to gather new transactions and form a valid block.
2. In the second phase, the miner should perform actions that allow her to add the new block to the chain. For example, in Bitcoin she has to solve a hashcash puzzle [39], while in typical proof-of-stake protocols she has to win a specific type of lottery [27, 32, 33].

A significant amount of research and development has been devoted to optimizing the second phase. For Bitcoin alone, there are already several generations of mining hardware [7, 23, 44], from GPU mining, to FPGA, to dedicated ASICs and trusted hardware frameworks [49]. Moreover, miners often collaborate in what is known as a mining *pool*, which has also been widely studied in the literature [18, 25, 35, 36, 38, 47, 48, 50]. In contrast, we focus on the orthogonal task of performing the first phase efficiently and optimally.

Mining Rewards. In order to incentivize miners to take part in mining, especially performing the often costly proof-of-work in the second phase, blockchain protocols reward them in two ways:

- *Base reward:* The miner is rewarded a predetermined amount for each block that she successfully adds to the blockchain. This reward is not under the miner's control and is instead fixed by the underlying protocol.

*Authors are listed alphabetically.

In Bitcoin, it is currently 6.25 BTC and halves at pre-determined intervals [39]. This is also how new units of currency are created. Some cryptocurrencies, such as Ethereum, have a more complex method in which miners who solve the second phase puzzle but whose block does not eventually get added to the chain are also rewarded [2].

- **Transaction fees:** Each transaction has a specific fee that is paid to the miner who includes this transaction in her block and adds it to the chain [39]. The transaction fees are set by the user who creates the transaction. A miner can decide which transactions to include in her block based on their fees. Indeed, it is well-known that transactions with small fees are often added to the chain with considerable delay or not at all.

Focus and Motivation. In this work, we consider a miner’s point-of-view, and focus on the problem of creating a block of transactions in the first phase of mining such that the total amount of gathered transaction fees are maximized. Note that the base reward is not under the miner’s control and hence her only tool for maximizing her profits is to create an optimal block with maximal transaction fees. Moreover, as base rewards halve in cryptocurrencies such as Bitcoin [39], transaction fees form an ever-increasing percentage of miner revenue. By the year 2140, the base reward in Bitcoin becomes 0 and transaction fees will be the only source of compensation for miners [39].

The task of forming an optimal block is complicated by several factors, which we now shortly review:

Block Size Limit. Every transaction has a known size. On the other hand, blockchain protocols enforce an upper-bound on the size of mined blocks. In Bitcoin, the bound is 1,000,000 bytes [37]¹. The maximum block size has a direct impact on the scalability of a cryptocurrency, and has been at the heart of the debate that led to forks such as Bitcoin Cash, which increased the block size limit to 8MB and then to 32MB [31, 34]. There are also cryptocurrencies that advocate for larger blocks, and even a total abandonment of block size limits, such as Bitcoin SV [6]. Block size limits complicate the task of forming an optimal block by forcing the miner to choose which transactions to include and which to ignore.

Dependencies. Transactions have dependencies among themselves. For example, in Bitcoin, if a transaction Tx_2 uses an output of a transaction Tx_1 as one of its inputs, then Tx_1 must appear in the chain before Tx_2 . This can be achieved either by putting Tx_1 in an earlier block, or in the same block as Tx_2 but in an earlier position. Such a dependency will not create any additional constraints for the miner if Tx_1 is already on

¹SegWit [37] affects neither our problem, nor the algorithms we propose. To apply our algorithms to Bitcoin transactions utilizing SegWit, one should simply discard the witness part when computing the size of a transaction. As such, we use vbytes and bytes interchangeably.

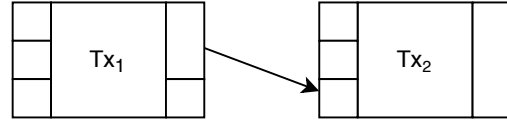


Figure 1. Tx_2 uses the first output of Tx_1 as its third input. Hence, Tx_2 depends on Tx_1 and must appear after it in the chain. In practice, Tx_2 has a pointer to Tx_1 but for demonstration purposes, we found it more convenient to show this as an arrow that models the flow of value from Tx_1 ’s output to Tx_2 ’s input.

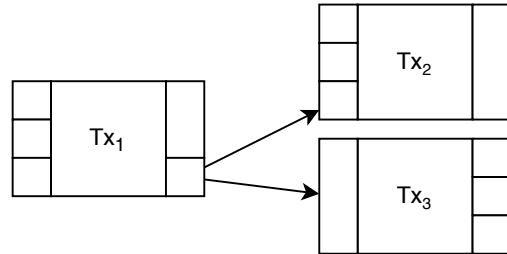


Figure 2. Tx_2 and Tx_3 both use the same funds (the second output of Tx_1). This is considered double-spending and is not allowed. Hence, Tx_2 and Tx_3 are in conflict and a miner cannot include both of them in her block.

the chain. However, if both Tx_1 and Tx_2 are new transactions that are not already put on the chain, then the miner cannot include Tx_2 in her block while leaving Tx_1 out.

Conflicts. It is possible for a pair of otherwise-valid transactions to be in conflict with each other. For example, in Bitcoin, one can create two transactions that double-spend the same output coin, leading to a situation where only one of them can be added to the consensus chain. In such cases, if one of the transactions is already on the blockchain, then the other transaction would be ignored. If none of the transactions are already mined, then the miner has to choose which transaction to put in her block, but she cannot choose both.

Inefficiency of Heuristics. It is easy to see that in the presence of the requirements above, many of the common heuristics used by miners can become infinitely bad on adversarial instances. For example, a miner that ignores all transactions that are involved in a double-spending risks not only losing their transaction fees but also the fees of transactions depending on them. Similarly, a miner that ignores low-fee transactions risks losing potentially high-fee transactions that depend on them. Moreover, it is noteworthy that the default Bitcoin implementation does not address the problem of forming an optimal block, and chooses transactions based on a “priority” formula that is meant to ensure that every transaction is eventually put into a block, instead of

aiming at maximizing the miner’s revenue [1]. As such, this approach has been largely abandoned by the miners [1].

Our Contribution. In this work, we consider the problem of forming an optimal block, i.e. one that maximizes the total transaction fees while respecting the requirements above. Our contributions are as follows:

- We first formally model the problem as what we call a *Dependency-Conflict Knapsack* (DCK) instance.
- We show that, unlike classical KNAPSACK, DCK is *strongly* NP-hard, hence ruling out the existence of pseudo-polynomial algorithms, i.e. algorithms depending polynomially on the block size limit, unless P=NP.
- We provide a hardness-of-approximation result, showing that there exists a constant $\epsilon > 0$ such that it is NP-hard to approximate the reward of the optimal block within a factor of $1 - \epsilon$. We also show that $\epsilon > 0.12$.
- Based on the observation that real-world instances of the problem are quite sparse, i.e. they have few conflicts and dependencies, we consider two graph sparsity parameters, namely treewidth and pathwidth, and show that for instances in which the dependency-conflict graph has constant treewidth, DCK is solvable in $O(n \cdot k^2)$, whereas in constant-pathwidth instances it is solvable in $O(n \cdot k)$. Here, n is the number of new transactions (also known as mempool size) and k is the block size limit. These pseudo-polynomial algorithms form *optimal* blocks.
- We provide real-world experimental results over Bitcoin, showing that the constant pathwidth assumption holds in practice and that our approach leads to significantly more profitable blocks and beats real-world miners by 13.4%.

Related Works. Surprisingly, the problem of forming an optimal block to mine is quite understudied. It is well-known that the problem is NP-hard. To the best of our knowledge, the earliest mention of this fact is in a blog post by Joseph Bonneau back in 2014 [13]. However, we show that it is also *strongly* NP-hard, and hard-to-approximate. Moreover, we provide the first positive theoretical results, i.e. pseudo-polynomial algorithms parameterized by treewidth and pathwidth that obtain an optimal block. We also provide significant practical improvements (See our experiments in Section 6). To the best of our knowledge, parameterized algorithms have not been previously studied in the context of blockchain, except for [14] which considers treewidth as a parameter for static analysis of smart contracts.

2 Preliminaries

In this section, we provide a short overview of the notions of treewidth and pathwidth, which we will later exploit in order to obtain efficient algorithms for optimal mining. Treewidth [43] is a widely-used graph parameter. Intuitively,

it models the degree to which a graph resembles a tree. Only trees and forests have a treewidth of 1. Similarly, pathwidth [42] is a measure of path-likeness of a graph. Many problems that are NP-hard on general graphs admit efficient solutions when restricted to instances with small treewidth or pathwidth [19, 21, 28, 29]. Even problems that are not NP-hard can often be solved more efficiently when parameterized by the treewidth/pathwidth [5, 15–17, 26]. We now provide more formal definitions:

Tree Decompositions and Path Decompositions [21, 42, 43]. Let $G = (V, E)$ be a graph with vertex set V and edge set E . A *tree decomposition* of G is a tree (T, E_T) such that:

- Each node $b \in T$ of the tree decomposition has an associated set $V_b \subseteq V$ of vertices of G . To avoid confusion, we reserve the word “vertex” for vertices of G and use the word “bag” to refer to the nodes of T . Moreover, we define E_b as the set of edges whose both endpoints are in V_b .
- Each vertex $v \in V$ appears in at least one bag. More formally, $\bigcup_{b \in T} V_b = V$.
- Each edge $e \in E$ appears in at least one bag. In other words, $\bigcup_{b \in T} E_b = E$.
- Each vertex $v \in V$ appears in a *connected* subtree of T . In other words, for all $b_1, b_2, b_3 \in T$, if b_3 is on the unique path from b_1 to b_2 , then $V_{b_1} \cap V_{b_2} \subseteq V_{b_3}$.

A tree decomposition is called a *path decomposition* if (T, E_T) is a simple path.

Treewidth and Pathwidth [21, 42, 43]. The width of a tree decomposition is defined as the size of its largest bag minus 1. The *treewidth* (resp. *pathwidth*) of a graph G is the smallest width among all of its tree decompositions (resp. path decompositions).

Example 2.1. Figure 3 shows an example graph, together with a tree decomposition and a path decomposition.

Nice Decompositions. Consider a tree decomposition (T, E_T) of a graph G , in which a bag $r \in T$ is chosen as root. The tree decomposition (T, E_T) is called *nice* if it satisfies the following conditions:

- If a bag $l \in T$ is a leaf, then $V_l = \emptyset$.
- If a bag $b \in T$ is not a leaf, then b is in one of the following forms:
 - *Introduce Bag:* The bag b has a single child b' and there is a vertex $v \in V_b$ such that $V_{b'} = V_b \setminus \{v\}$. In this case, we say that b *introduces* v .
 - *Forget Bag:* The bag b has a single child b' and there is a vertex $v' \notin V_b$ such that $V_{b'} = V_b \cup \{v'\}$. In this case, we say that b *forgets* v' .
 - *Join Bag:* The bag b has exactly two children, b_1 and b_2 , and we have $V_b = V_{b_1} = V_{b_2}$.

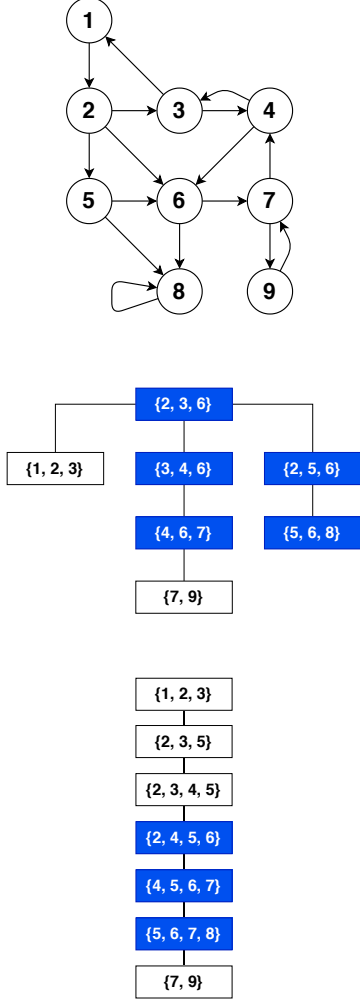


Figure 3. A graph G (top), a tree decomposition of G with width 2 (middle), and a path decomposition of G with width 3 (bottom). In each case, the connected subtree containing the vertex 6 is shown in blue.

A nice path decomposition is defined similarly, except that there can be no join bags in a path decomposition. It is easy to see that any tree decomposition or path decomposition can be turned nice in linear time. See [21] for details. Nice decompositions are useful because they allow one to perform dynamic programming on arbitrary trees in essentially the same manner as on trees or paths. This is exactly what our algorithm in Section 5 does. See [9] for more examples of this type of dynamic programming.

Sparsity. Treewidth and pathwidth are graph sparsity parameters, in the sense that a graph with n vertices and treewidth t can have at most $O(n \cdot t)$ edges. Moreover, many well-studied families of graphs, such as cacti, series-parallel graphs, outerplanar graphs, and control-flow graphs of programs, have constant treewidth [11, 46].

Fixed-parameter Tractability [21]. Given an instance with n vertices and a graph parameter r as input, we say that a graph decision problem is *Fixed-Parameter Tractable* (FPT) with respect to r , if there exists an algorithm that solves it in $O(n^c \cdot f(r))$, where c is a fixed constant not depending on either n or r , and f is an arbitrary computable function. This definition, which is standard in parameterized complexity, captures the requirement that the problem can be solved in polynomial time when the parameter r is small. Moreover, the degree of the polynomial is independent of r . In the sequel, we will obtain FPT algorithms with respect to treewidth and pathwidth.

Computing Treewidth and Pathwidth. In general, computing the treewidth or pathwidth of an arbitrary input graph are NP-hard problems [3, 41]. However, the problems are FPT when parameterized by the treewidth (resp. pathwidth) itself. Indeed, [10, 12] provide *linear-time* FPT algorithms for computing treewidth and pathwidth. Moreover, there are efficient tools and libraries, such as [24, 45] that compute treewidth/pathwidth. As such, in our decomposition-based algorithms, we assume without loss of generality that a *nice* tree decomposition (resp. path decomposition) with $O(n \cdot t)$ bags is given as part of the input.

3 Dependency-Conflict Knapsack

In this section, we formalize our optimal mining task as a variant of the KNAPSACK problem, called DEPENDENCY-CONFLICT KNAPSACK (DCK).

Instances. A DCK instance is a tuple $I = (n, k, \Sigma, V, W, C, D)$, in which:

- n and k are positive integers. Intuitively, n is the number of items and k is the capacity of our knapsack.
- $\Sigma = \{\sigma_1, \dots, \sigma_n\}$ is a set of n items.
- $V, W : \Sigma \rightarrow \mathbb{N} \cup \{0\}$ are functions that assign a *value* and a *weight* to every item. For brevity, we denote $V(\sigma_i)$ by v_i and $W(\sigma_i)$ by w_i .
- $C, D \subseteq \Sigma \times \Sigma$, respectively called the *conflict* and *dependency* relations, are relations on Σ such that:
 - (i) C is symmetric.
 - (ii) The transitive closure of D is anti-symmetric.

Informally, $\sigma_i C \sigma_j$ signifies that the two elements σ_i and σ_j are in conflict, i.e. we cannot put both of them into the knapsack. It is clear that the conflict relation should be symmetric. Similarly, $\sigma_i D \sigma_\ell$ means that σ_ℓ is a prerequisite of σ_i , i.e. if we put σ_i in the knapsack, we have to put σ_ℓ , too. In (ii), we are assuming that there are no cyclic dependencies. We now formalize the problem:

The DCK Problem. Given an instance $I = (n, k, \Sigma, V, W, C, D)$ as above and a positive integer α , the DCK(I, α) problem asks whether there exists a subset $\Sigma^* \subseteq \Sigma$ of items, such that:

- (a) $\sum_{\sigma^* \in \Sigma^*} W(\sigma^*) \leq k$, i.e. the items in Σ^* must fit in a knapsack of size k .
- (b) For every $\sigma_i^*, \sigma_j^* \in \Sigma^*$, we have $(\sigma_i^*, \sigma_j^*) \notin C$.
- (c) For every $\sigma_i, \sigma_j \in \Sigma$, if $\sigma_i D \sigma_j$ and $\sigma_i \in \Sigma^*$, then we also have $\sigma_j \in \Sigma^*$.
- (d) $\sum_{\sigma^* \in \Sigma^*} V(\sigma^*) \geq \alpha$, i.e. the total value of items in Σ^* is at least α .

The maximization variant of the DCK problem, MAXDCK, asks for a Σ^* that maximizes the sum $\sum_{\sigma^* \in \Sigma^*} V(\sigma^*)$.

It is easy to see the correspondence between the DCK problem and the problem of forming a block. The knapsack size k serves as the block size limit, while each of the n items represents a valid new transaction. By this, we mean a transaction that is not already included in the chain, and passes other validity checks (such as providing the right signatures). If a transaction σ double-spends a coin that was spent in another transaction σ' and σ' is already on the chain, then σ is considered to be invalid. However, if σ' is also a new transaction, then they are both considered valid, but in conflict. The weight w_i represents the size of transaction σ_i and the value v_i represents its transaction fee, which will be paid to the miner if she includes it in her block (knapsack). The relation C models conflicts between transactions, i.e. if σ_i and σ_j are transactions that are double-spending the same output, then we have $\sigma_i C \sigma_j$. Similarly, D models dependencies. Condition (ii) makes sure that we do not have cyclic dependencies. In the real-world, if a set of transactions have cyclic dependencies, they are all invalid, and can be removed by a simple preprocessing. Using this correspondence, the DCK problem formalizes the question of whether one can form a valid block with a total transaction fee of at least α , whereas MAXDCK asks for the maximum possible amount of transaction fees among all valid blocks.

DCG. Given a DCK instance $I = (n, k, \Sigma, V, W, C, D)$, its *Dependency-Conflict Graph* (DCG) is a graph $G = (\Sigma, E)$, in which each item serves as a vertex, and there are two types of edges in E :

- *Undirected Conflict Edges:* There is an undirected edge $\{\sigma_i, \sigma_j\}$ for each $\sigma_i C \sigma_j$.
- *Directed Dependency Edges:* There is a directed edge (σ_i, σ_j) for each $\sigma_i D \sigma_j$.

4 Hardness Results

In this section, we provide a simple reduction showing that DCK is strongly NP-hard. This is in contrast to classical KNAPSACK, which has a simple pseudo-polynomial dynamic programming algorithm and is only weakly NP-hard. Moreover, we show that MAXDCK is hard-to-approximate within a constant factor unless P=NP, and hence does not admit a PTAS. This is again in contrast to classical KNAPSACK, which

admits an FPTAS. Additionally, our reduction rules out efficient parameterized algorithms based on several common graph parameters, such as degree and diameter, on the DCG.

The Reduction. Our reduction is from 3-SAT. Given a 3-SAT formula φ with \bar{m} clauses over \bar{n} boolean variables, we construct the following DCK instance $I_\varphi = (n, k, \Sigma, V, W, C, D)$:

- $n = 3 \cdot \bar{m} + 2 \cdot \bar{n}$
- $k = \bar{m} + \bar{n}$
- For each variable x appearing in φ , we add two items σ_x and $\sigma_{\neg x}$ to Σ . We set $W(\sigma_x) = W(\sigma_{\neg x}) = 1$ and $V(\sigma_x) = V(\sigma_{\neg x}) = 0$. Moreover, $(\sigma_x, \sigma_{\neg x}), (\sigma_{\neg x}, \sigma_x) \in C$, i.e. the two items are in conflict.
- For each clause $c = (\ell_1 \vee \ell_2 \vee \ell_3)$ of φ in which each literal ℓ_i is either a boolean variable or its negation, we add three items $\sigma_{c,1}, \sigma_{c,2}$, and $\sigma_{c,3}$ to Σ . We set $W(\sigma_{c,i}) = V(\sigma_{c,i}) = 1$. Moreover, we have $(\sigma_{c,i}, \sigma_{c,j}) \in C$ for every $i \neq j$, i.e. every pair of the three elements are in conflict. Additionally, we set $\sigma_{c,i} D \sigma_{\ell_i}$, i.e. the i -th element of c depends on the element corresponding to ℓ_i .

Example 4.1. Figure 4 illustrates our reduction.

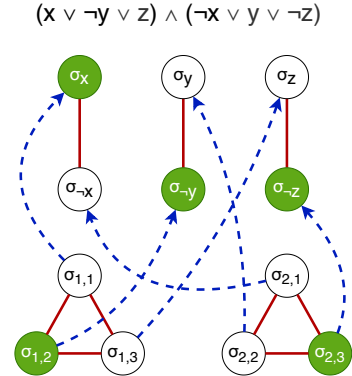


Figure 4. A 3-SAT formula (top), and its reduction to a DCK instance. Red edges denote conflict and blue directed edges denote dependency. For example, $\sigma_{1,1}$ depends on σ_x . In the DCK instance, we have $k = 5$. Moreover, every element has unit weight. The variable elements and their negations have value zero, whereas $\sigma_{i,j}$'s have unit value. The items put into the knapsack in one optimal solution are shown in green. Note that this yields a total value of 2, which proves satisfiability. This is achieved by letting $x = 1$ and $y = z = 0$, which corresponds to satisfying the second literal of the first clause $(\sigma_{1,2})$, and the third literal of the second clause $(\sigma_{2,3})$.

It is easy to verify that I has all the requirements for a DCK instance as defined in Section 3. Note that every solution to I can pick at most one of σ_x and $\sigma_{\neg x}$ for every variable x . Similarly, for each clause c , it can take at most one of the items corresponding to c , and can take $\sigma_{c,i}$ only

if it also takes the item corresponding to the i -th disjunct of c . Also, note that every item has unit weight, and only the items corresponding to clauses have a unit value, while all other items are worthless. Given this discussion, it is easy to see that φ is satisfiable iff $\text{DCK}(I_\varphi, \bar{m}) = 1$. Moreover, $\text{MAXDCK}(I_\varphi) = \text{MAX3-SAT}(\varphi)$. Hence, we have the following theorems:

Theorem 4.2 (Strong NP-hardness). *DCK is strongly NP-hard.*

Proof. As shown above, for every 3-SAT formula φ , we have $3\text{-SAT}(\varphi) \Leftrightarrow \text{DCK}(I_\varphi, \bar{m})$. It is well-known that 3-SAT is NP-hard. Moreover, the reduction above keeps n and k polynomial in terms of $|\varphi|$. Hence, DCK is strongly NP-hard. \square

Theorem 4.3 (Inapproximability). *There exists $\epsilon > 0$, such that it is NP-hard to approximate MAXDCK within a factor of $1 - \epsilon$.*

Proof. A well-known corollary of the PCP theorem [4] is that such an ϵ exists for the MAX3-SAT problem. The theorem follows from the fact that we have $\text{MAXDCK}(I_\varphi) = \text{MAX3-SAT}(\varphi)$ in our reduction. Indeed, it is hard to approximate MAX3-SAT within a ratio of $\frac{7}{8} + \epsilon$ for any $\epsilon > 0$ [30]. Using our reduction, this result also applies to MAXDCK. \square

5 Efficient Decomposition-based Algorithms

In this section, we provide efficient FPT algorithms for the DCK problem with respect to the treewidth and pathwidth of its DCG. For a DCG with treewidth t , our approach leads to an $O(n \cdot k^2 \cdot 2^t \cdot t^3)$ -time algorithm. Moreover, for a DCG with pathwidth p , it obtains a runtime of $O(n \cdot k \cdot 2^p \cdot p^3)$.

Setup and Notation. Let $I = (n, k, \Sigma, V, W, C, D)$ be a DCK instance given as input together with a nice tree decomposition (T, E_T) of its DCG $G = (\Sigma, E)$. Moreover, as justified in Section 2, we assume that the tree decomposition has width t and $O(n \cdot t)$ bags. Recall that for a bag $b \in T$, we denote its associated set of items by $\Sigma_b \subseteq \Sigma$, and define $E_b \subseteq E$ as the set of edges whose both endpoints are in Σ_b . Additionally, as our tree decomposition is nice, we have a distinguished root bag r , and every bag is either a leaf, an introduce bag, a forget bag, or a join bag. For a bag $b \in T$, we denote by T_b^\downarrow the subtree of T consisting of b and its descendants. Similarly, we define G_b^\downarrow as the part of G that corresponds to T_b^\downarrow , i.e. $G_b^\downarrow = \left(\bigcup_{b' \in T_b^\downarrow} \Sigma_{b'}, \bigcup_{b' \in T_b^\downarrow} E_{b'} \right)$.

Example 5.1. Figure 5 shows a DCG G (left) and a nice tree decomposition of G (right). We will use this figure as our running example. Suppose that every vertex i has a value of i and unit weight. Moreover, let $k = 3$. In this example, we have $\Sigma_{b_2} = \{1, 2, 3\}$ and $E_{b_2} = \{(1, 2), \{2, 3\}\}$. Moreover, $T_{b_1}^\downarrow$ is the part of the tree that contains b_1, b_2, b_3, b_4 , and b_5 , and

the graph $G_{b_1}^\downarrow$ contains vertices 1, 2, 3 and edges (1, 2) and {2, 3}.

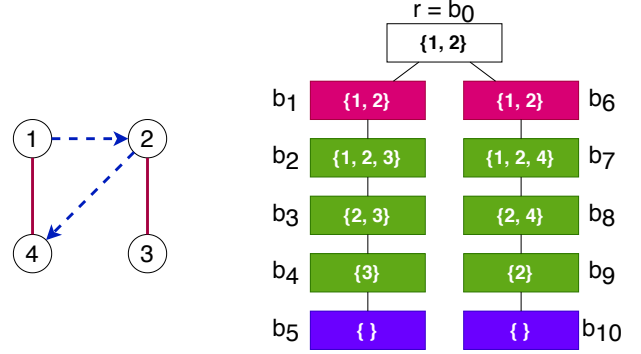


Figure 5. A DCG G (left) and a nice tree decomposition of G (right). Leaf bags are shown in blue, introduce bags in green, and forget bags in red. The only join bag is the root.

Main Idea. Our algorithm is a bottom-up dynamic programming on the tree decomposition. At every bag $b \in T$, for every subset $M \subseteq \Sigma_b$, and every non-negative integer $\kappa \leq k$, we define a variable $dp[b, M, \kappa]$ and initialize it to $-\infty$. The goal is to compute values for each such variable such that the following invariant is satisfied: $dp[b, M, \kappa]$ is the maximum total value of items that can be placed in a knapsack of size κ , such that:

- Every item comes from G_b^\downarrow ,
- All dependency and conflict relations in G_b^\downarrow are respected, and
- The set of items chosen from Σ_b is exactly M .

Having this in mind, we now show how one can compute the values for dp variables. Our algorithm processes the bags of the tree decomposition in a *bottom-up* order and performs the following calculations:

Computing Values at Leaves. Given that our tree decomposition is nice, for every leaf bag $l \in T$, we have $\Sigma_l = \emptyset$. Hence, the algorithm sets $dp[l, \emptyset, \kappa] = 0$ for every κ .

Example 5.2. In the instance of Example 5.1 (Figure 5), the algorithm computes $dp[b_5, \emptyset, \kappa] = dp[b_{10}, \emptyset, \kappa] = 0$ for all $0 \leq \kappa \leq 3$.

Computing Values at Introduce Bags. Let $b \in T$ be an introduce bag. Also, let b' be its child and σ be the item/vertex that is introduced in b . When computing $dp[b, M, \kappa]$, the algorithm first checks whether M violates any dependency/conflict relations within E_b . If so, it sets $dp[b, M, \kappa] = -\infty$. Similarly, if the sum of weights of items in M exceeds κ , it sets $dp[b, M, \kappa] = -\infty$. Otherwise, it sets:

$$dp[b, M, \kappa] = \begin{cases} dp[b', M \setminus \{\sigma\}, \kappa - W(\sigma)] + V(\sigma) & \sigma \in M \\ dp[b', M, \kappa] & \sigma \notin M \end{cases}$$

It is straightforward to see why this works. The argument is similar to classical 0-1 KNAPSACK. If $\sigma \in M$, then we should put σ in the knapsack, leading to a value of $V(\sigma)$, and leaving us with $k - W(\sigma)$ more room to fill with items from $G_{b'}^\downarrow$. If $\sigma \notin M$, there is no gain in value and no loss in space, and the knapsack should be filled using $G_{b'}^\downarrow$. In both cases, we of course have to respect all the conflicts and dependencies in $G_{b'}^\downarrow$, too. This is modeled by $dp[b', \cdot, \cdot]$.

Example 5.3. In the instance of Example 5.1 (Figure 5), the algorithm sets $dp[b_2, \{1, 3\}, \kappa] = -\infty$ for every κ . This is because 1 depends on 2, which is not included, hence the requirement of the dependency edge (1, 2) is violated. Similarly, we have $dp[b_7, \{1, 2, 4\}, 2] = \emptyset$, because there is not enough capacity in a knapsack of size 2 for 3 unit-size elements. Now consider bag b_2 , which introduces vertex 1. The algorithm computes $dp[b_2, \{2, 3\}, 3] = dp[b_3, \{2, 3\}, 3] = 3$ and $dp[b_2, \{1, 2\}, 2] = dp[b_3, \{2\}, 1] + V(1) = 3$.

Computing Values at Forget Bags. If $b \in T$ is a forget bag with a single child b' , then by definition, we have $G_b^\downarrow = G_{b'}^\downarrow$. Suppose that b forgets σ . Then, the algorithm computes dp values at b as follows:

$$dp[b, M, \kappa] = \max\{dp[b', M, \kappa], dp[b', M \cup \{\sigma\}, \kappa]\}.$$

This is because G_b^\downarrow and $G_{b'}^\downarrow$ enforce the same dependency and conflict requirements. Moreover, if the set of items put in the knapsack has intersection M with Σ_b , then its intersection with $\Sigma_{b'} = \Sigma_b \cup \{\sigma\}$ is either M or $M \cup \{\sigma\}$.

Example 5.4. In the instance of Example 5.1 (Figure 5), bag b_6 forgets vertex 4. The algorithm computes $dp[b_6, \{2\}, 3] = \max\{dp[b_7, \{2\}, 3], dp[b_7, \{2, 4\}, 3]\} = \max\{2, 6\} = 6$. Similarly, we have:

$$\begin{aligned} dp[b_1, \{1, 2\}, 2] &= \max\{dp[b_2, \{1, 2\}, 2], dp[b_2, \{1, 2, 3\}, 2]\} \\ &= \max\{3, 3\} = 3. \end{aligned}$$

Now consider the case of computing $dp[b_6, \{1, 2\}, 3]$. In a valid solution, it is impossible to take both 1 and 2, because 2 depends on 4 and 4 is in conflict with 1. Nevertheless, this does not violate any local restrictions at b_6 . Note that $E_{b_6} = \{(1, 2)\}$ and choosing the set $\{1, 2\}$ satisfies the requirement. Hence, the algorithm computes $dp[b_6, \{1, 2\}, 3] = \max\{dp[b_7, \{1, 2\}, 3], dp[b_7, \{1, 2, 4\}, 3]\}$. However, these values are both $-\infty$. $dp[b_7, \{1, 2\}, 3] = -\infty$ because the local dependency requirement (2, 4) is not met at b_7 . Similarly, $dp[b_7, \{1, 2, 4\}, 3] = -\infty$ because the local conflict requirement between 1 and 4 at b_7 is not met. Hence, we will get $dp[b_6, \{1, 2\}, 3] = -\infty$.

Computing Values at Join Bags. Let $b \in T$ be a join bag with children b_1 and b_2 . By definition, we have $\Sigma_b = \Sigma_{b_1} = \Sigma_{b_2}$ and $G_b^\downarrow = G_{b_1}^\downarrow \cup G_{b_2}^\downarrow$. Note that because T is a tree decomposition, every vertex appears in a connected subtree.

Hence, all common vertices of $G_{b_1}^\downarrow$ and $G_{b_2}^\downarrow$ are in Σ_b . To compute $dp[b, M, \kappa]$, the algorithm first checks whether any dependency or conflict requirements in E_b are violated by M . If so, it sets $dp[b, M, \kappa] = -\infty$. Otherwise, it computes $V(M) = \sum_{m \in M} V(m)$, i.e. the total value of items in M , and $W(M) = \sum_{m \in M} W(m)$, i.e. the total weight of items in M . If $W(M) > \kappa$, it sets $dp[b, M, \kappa] = -\infty$. Otherwise, it computes:

$$dp[b, M, \kappa] = \max_{i=W(M)}^{\kappa} (dp[b_1, M, i] + dp[b_2, M, \kappa + W(M) - i] - V(M)).$$

We now explain why this is correct. Suppose that we want to fill a knapsack of size κ with items from $G_b^\downarrow = G_{b_1}^\downarrow \cup G_{b_2}^\downarrow$. We first decide how much of the capacity in the knapsack should be assigned to items from $G_{b_1}^\downarrow$ and denote it by i . This cannot be less than $W(M)$ as we know that we have to put M in the knapsack. After putting M and the other items from $G_{b_1}^\downarrow$, we have $\kappa - i$ remaining capacity for other elements from $G_{b_2}^\downarrow$. However, given that M also appears in $G_{b_2}^\downarrow$, this is equivalent to filling a knapsack of size $\kappa + W(M) - i$ using items in $G_{b_2}^\downarrow$ in which we are forced to take M . The final $-V(M)$ in the formula is to avoid double-counting the value of items in M , which were counted in both dp variables.

Example 5.5. In the instance of Example 5.1 (Figure 5), the only join bag is the root $r = b_0$. The algorithm computes

$$\begin{aligned} dp[r, \{2\}, 2] &= \max\{dp[b_1, \{2\}, 1] + dp[b_6, \{2\}, 2], dp[b_1, \{2\}, 2] \\ &\quad + dp[b_6, \{2\}, 1]\} - V(2) \\ &= \max\{2 + 6, 6 + 2\} - 2 = 6. \end{aligned}$$

Intuitively, we want to fill a knapsack of size 2 and we know that vertex 2 must be present in the knapsack and 1 must be absent. We consider two cases: either we allocate capacity 1 to the subgraph $G_{b_1}^\downarrow$ and capacity 2 to $G_{b_6}^\downarrow$ (The vertex 2 itself has a weight of 1 is counted in both capacities), or vice versa. We can read the maximum values from dp variables computed in b_1 and b_6 . However, as vertex 2 was included in both sides, we have to deduct its value at the end.

Note that in the steps above, the values of dp variables are computed correctly. Specifically, at each bag b , we first check that the local dependency/conflict requirements at b are satisfied. If they are not, we set the $dp[b, \cdot, \cdot]$ to $-\infty$. Hence, a bottom-up inductive argument shows that all $dp[b, \cdot, \cdot]$ values respect the dependency and conflict requirements of the edges of G_b^\downarrow .

Computing the Final Answer. Finally, the algorithm computes the answer to the MAXDCK problem as follows:

$$\max_{M \subseteq \Sigma_r} dp[r, M, k].$$

Recall that r is the root bag, and hence $G_r^\downarrow = G$. So every solution of $dp[r, M, k]$ respects all dependency and conflict

relations in G . Also, note that obtaining the actual contents of our knapsack is a matter of following dp values that lead to the optimal solution in each formula above, just as in the classical 0-1 KNAPSACK.

Example 5.6. In our running example, the final solution is:

$$\begin{aligned} & \max\{dp[r, \emptyset, 3], dp[r, \{1\}, 3], dp[r, \{2\}, 3], dp[r, \{1, 2\}, 3]\} \\ & = \max\{7, -\infty, 6, -\infty\} = 7, \end{aligned}$$

which can be achieved by taking items 3 and 4.

Given the algorithm and discussion above, we have the following theorems:

Theorem 5.7. *Given a DCK instance $I = (n, k, \Sigma, V, W, C, D)$, and a nice tree decomposition of its DCG with width t and $O(n \cdot t)$ bags, our algorithm above solves MAXDCK in time $O(n \cdot k^2 \cdot 2^t \cdot t^3)$.*

Proof. We define $O(2^t \cdot k)$ dp variables at each bag. Given that there are $O(n \cdot t)$ bags, the total number of dp variables is $O(n \cdot k \cdot t \cdot 2^t)$. Computing each dp variable takes $O(t^2)$, i.e. for checking the satisfaction of local constraints in the current bag, except when we are handling join bags, where it takes $O(t^2 + k)$ due to taking the maximum of k elements. This leads to the desired bound of $O(n \cdot k^2 \cdot 2^t \cdot t^3)$ for the whole runtime. \square

Theorem 5.8. *Given a DCK instance $I = (n, k, \Sigma, V, W, C, D)$, and a nice path decomposition of its DCG with width p and $O(n \cdot p)$ bags, our algorithm above solves MAXDCK in time $O(n \cdot k \cdot 2^p \cdot p^3)$.*

Proof. This is exactly similar to Theorem 5.7, except that a nice path decomposition has no join nodes, and hence the algorithm is faster by a significant factor of k . \square

Note that if the treewidth or pathwidth are fixed (small constants), the theorems above lead to pseudo-polynomial algorithms with runtimes $O(n \cdot k^2)$ and $O(n \cdot k)$, respectively. Especially, the latter bound matches the runtime of the classical dynamic programming algorithm for 0-1 KNAPSACK. As we will see in the next section, this is exactly what happens in practice.

Parallelization. The $dp[b, \dots]$ computations performed by our algorithm in every bag are independent of each other and parallelizable. Specifically, when solving instances with a knapsack of capacity k , if we have θ threads and $\theta < k \cdot 2^t$, then the algorithm can be perfectly parallelized. In real-world use-cases, we often have $k \geq 10^6$. So, for all practical purposes, our algorithms' parallel runtimes are $O\left(\frac{n \cdot k^2}{\theta}\right)$ and $O\left(\frac{n \cdot k}{\theta}\right)$ for instances with bounded treewidth and pathwidth, respectively.

6 Implementation and Experimental Results

Implementation. We implemented our algorithm in C++ and used OpenMP [22] for parallelization. We relied on the codebase of the Explora Block Explorer [8] to collect information about the Bitcoin blockchain. This includes details of the transactions in each block and the mempool (transactions that are published but not yet mined). We computed tree and path decompositions using SageMath [45].

Central Hypothesis. We consider DCK instances that model the problem of obtaining optimal blocks (wrt transaction fees) in the Bitcoin blockchain. Our central hypothesis is that the DCGs (Dependency-Conflict Graphs) of these instances have bounded treewidth/pathwidth. In other words, we are creating a graph in which we put a vertex for every transaction and put edges between two transactions if either they are in conflict or one is a dependency of the other. We hypothesize that such a graph would be sparse and have a tree-like/path-like structure. As such, we expect these graphs to have small treewidth/pathwidth. This is intuitively justified by the fact that double-spending is relatively rare and creates very few conflict edges. On the other hand, the dependence between transactions is often in the form of a directed acyclic graph and has a tree-like structure.

Benchmarks and Baseline. We considered blocks number 681734 to 681935 in the Bitcoin blockchain. These blocks correspond to more than a day (almost 27 hours) of activity. More specifically, they were mined between 3rd May 2021 - 15:51 UTC and 4th May 2021 - 19:10 UTC. When solving for block i , we considered the mempool right after block $i - 1$ or 5 minutes before block i was added (whichever were the latest). We used this mempool as the set Σ of all possible transactions. There is a simple reason behind this choice: the mempool is continuously evolving as new transactions are broadcast. As such, a miner who is intent on mining the optimal block should constantly run our algorithm on new mempools. As we will see, each run of our algorithm takes roughly 3 minutes on our machine. To ensure that we are not obtaining any unfair advantage, we set the interval to 5 minutes. We then ran our algorithm to obtain an optimal block. Finally, we compared the total transaction fees obtained by our solution with those earned by the miner of the actual block i on the blockchain.

Experimental Setting. All results were obtained on a machine with 4 Intel Xeon E7-4850 v3 processors (2.20GHz, 14 cores, 28 threads, 35 MB Cache), running Ubuntu 20.04 LTS with 160GB of RAM and a total of 112 threads. Note that this is an extremely modest configuration in comparison with the computation power that the miners routinely use for proof-of-work. Moreover, as mentioned above, our algorithm can

be perfectly parallelized and will therefore use much less time when run by the real-world miners.

Results. The results are shown in Figure 6 and Tables 1–3. We now discuss them in more detail:

- **Widths.** The tables report the pathwidth of every instance (**PW**). This demonstrates that our central hypothesis holds in the real world and the widths are small. Moreover, given that the pathwidth is at most 3 and the capacity is $k = 10^6$ in Bitcoin, our pathwidth-based algorithm is much more promising than the treewidth-based variant, i.e. $O(n \cdot k)$ vs $O(n \cdot k^2)$.
- **Transaction Fee Revenues.** Figure 6 and Tables 1–3 also show the amount of transaction fees obtained by our algorithm vs the amount earned by the miners on chain. Based on these, our approach obtains a maximum per-block improvement of **259 percent** in transaction fee revenues, which is huge. Moreover, the average per-block improvement is a whopping **13.4 percent**. In absolute terms, our algorithm obtains between -0.029 and 0.776 BTC more fees than the miners in each block. If we sum this over all blocks, we get total improvements of **5.539 BTC**, which was equal to roughly **325,800 USD** at the time².
- **Runtimes.** The tables also report the runtime of our algorithm for finding the optimal blocks. Our runtimes range from 114s to 277s, and the average runtime is **175s**. Note that in Bitcoin, a new block is mined roughly every 10 minutes. So, even with our modest computational resources, we are able to find the optimal block in time. Given that the miners have access to much more computational power (that they use for proof-of-work), obtaining the optimal block using our algorithm will have a negligible effect on computation costs, while significantly increasing revenue.

Discussion. We now discuss several aspects of our results, as well as their limits of applicability to other blockchains and threats to their validity in the future.

- **Close Results.** Despite the considerable overall improvement in transaction fee revenues, there are a sizable number of blocks (52 out of 202) for which the transaction fees obtained by our algorithm are very close to those of the miners ($\pm 1\%$). We believe this is evidence that the miners are already using various relatively successful heuristics for maximizing their revenue. However, as the overall results demonstrate, such heuristics are not always effective and lead to a

²Note that this is the sum of savings over each individual block. However, it is not necessarily the exact amount of increase in the miners' revenue if they use our algorithm. Changing the mined block will also change the current mempool. Moreover, many users form their transactions based on the current state of the blockchain. As such, computing the exact total change in revenue is impossible.

much lower-than-optimal return in the long run. In contrast, our algorithm is able to form an optimal block and obtain the highest possible revenue.

- **Lower Results vs Optimality.** In some cases, our algorithm's reported transaction fee revenue is slightly lower than what was obtained by the miners on the blockchain. This seems to contradict the optimality of our algorithm, which was proven in Section 5, but is actually caused by an entirely orthogonal reason: In these instances, the miners had access to transactions which were not in our mempool. Given the distributed nature of the Bitcoin blockchain, its low connectivity [40], and our limited networking resources (a single node in Rafsanjan, Kerman Province, Iran), it was inevitable that we miss some transactions. Moreover, we ran our algorithm in 5-minute intervals. Hence, when forming block i , we missed transactions that were announced shortly before this block was mined. In contrast, it is well-known that miners typically deploy several nodes in different continents, ensuring that they have a much more reliable connection. Moreover, they have considerably larger computational power and can run the algorithm in shorter intervals. In spite of our limited resources, as Tables 1–3 and Figure 6 demonstrate, we were able to obtain significantly higher transaction fee revenues overall.
- **Extension to other (non-Bitcoin) Blockchains.** Our algorithms are directly applicable to any blockchain with static transaction fees, i.e. blockchains in which the exact fee is known at the time the transaction is broadcast. Extending these methods to blockchains with dynamic transaction fees, such as Ethereum, is a challenging and interesting direction of future work. It is also noteworthy that our algorithms do not depend on the consensus mechanism and can be applied to blockchains that do not use proof-of-work.
- **Threats to Validity.** The main threat to the validity of our approach is if our central hypothesis (low width) does not hold. This hypothesis can be violated by the users, who are the originators of transactions and whose actions ultimately define the conflicts and dependencies. For example, if the network suddenly receives a huge number of double-spending attacks, then the DCG will no longer be sparse/low-width. As shown in Section 4, the problem is strongly NP-hard and hard-to-approximate without this assumption. However, as demonstrated by our experimental results, our assumption currently holds in Bitcoin. Another threat is posed if the blocks are added to the chain in extremely small timeframes. In Bitcoin, a new block is mined roughly every 10 minutes. As shown by our experimental results, this is enough time for us to run our algorithms and obtain optimal blocks. Given that our algorithms are perfectly parallelizable,

shorter times between mined blocks would only translate to a need for more computation power. However, our algorithms also rely on tree/path decompositions which are obtained from external non-parallel tools. If the rate of addition of new blocks is extremely fast (e.g. one block per second), we might not be able to compute the decompositions in time.

7 Conclusion

In this work, we considered the problem of forming an optimal block, i.e. a block that yields maximal transaction fee revenue, from the viewpoint of a miner. We formalized it as an extension of the KNAPSACK problem with dependencies and conflicts. We then showed that it is strongly NP-hard and hard-to-approximate within a factor of $\frac{7}{8} + \epsilon$ unless P=NP. Then, we exploited the fact that real-world instances of the problem have sparse underlying dependency-conflict graphs and obtained efficient algorithms parameterized by the treewidth and pathwidth of this graph. Finally, we provided experimental results demonstrating that our approach significantly outperforms real-world miners, obtaining improvements of up to 259 percent per block (average improvement: 13.4 percent). In the 27-hour window of our experiment, this led to an improvement of 5.539 BTC / 325,800 USD in absolute terms. Given that our approach is efficient and parallelizable, it provides the miners with a cost-effective and simple solution to dramatically increase their transaction fee revenues.

Block	\Sigma	E	PW	T	Our Fee	Miner's Fee	\Delta
681734	32852	4322	2	202	1.64384456	1.64942663	-0.34%
681735	31337	3928	2	193	1.36386228	1.36925564	-0.39%
681736	34303	4962	3	220	1.49203669	1.45670239	+2.43%
681737	36821	5653	3	177	1.42085126	1.40673804	+1.00%
681738	34493	5183	2	183	0.95526813	0.94908231	+0.65%
681739	33639	4812	2	194	0.73592618	0.72156298	+1.99%
681740	33850	4738	2	190	0.87610756	0.86174104	+1.67%
681741	33789	4965	3	123	1.24114721	1.21886940	+1.83%
681742	37416	5875	3	125	1.52396150	1.48609066	+2.55%
681743	39858	7250	3	176	1.65437604	1.61597732	+2.38%
681744	41973	7967	3	128	1.51750749	1.49366215	+1.60%
681745	43291	8687	3	195	1.43887144	1.41367160	+1.78%
681746	40951	7994	3	159	1.20973668	1.18488390	+2.10%
681747	39466	7515	3	141	1.10934511	1.09153173	+1.63%
681748	39123	7392	3	188	1.15656696	1.13553911	+1.85%
681749	38113	7214	3	118	1.09677982	1.06994853	+2.51%
681750	36515	6788	3	169	0.86701302	0.84502231	+2.60%
681751	36550	6794	3	132	0.50364127	0.49441269	+1.87%
681752	36661	6550	2	182	1.48995020	1.45118859	+2.67%
681753	36113	6555	3	152	1.08737204	1.06602783	+2.00%
681754	34164	6344	3	122	0.54606148	0.53454122	+2.16%
681755	33792	6209	3	149	0.99764178	0.99079588	+0.69%
681756	32005	5925	2	163	0.68265387	0.67000081	+1.89%
681757	30593	5587	2	121	0.64262189	0.62817897	+2.30%
681758	29659	5153	3	151	1.40618720	1.39134636	+1.07%
681759	30025	5230	3	139	0.92160374	0.90653020	+1.66%
681760	30507	5301	3	146	0.79691323	0.77947123	+2.24%
681761	30308	5195	3	158	0.84170137	0.82064635	+2.57%
681762	30626	5089	3	117	1.03641038	1.01285348	+2.33%
681763	35409	6306	3	174	1.62126813	1.58237716	+2.46%
681764	35769	6472	3	160	1.41243719	1.37652773	+2.61%
681765	33535	6039	3	121	0.84725600	0.82926047	+2.17%
681766	31299	5567	3	120	0.43660905	0.43107542	+1.28%
681767	31319	5402	3	166	1.02576384	1.00472613	+2.09%
681768	29472	5193	3	175	0.40257931	0.39617466	+1.62%
681769	28193	4620	3	162	0.38398402	0.38524465	-0.33%
681770	26461	3727	2	170	0.24132183	0.22053945	+9.42%
681771	25532	3139	2	153	0.39829964	0.31831705	+25.13%
681772	26248	3247	2	159	1.44575882	1.46178535	-1.10%
681773	26864	3398	2	162	1.47124550	0.69572765	+111.47%
681774	24733	3008	2	141	0.42943216	0.43013572	-0.16%
681775	24796	3011	2	140	0.45939623	0.12792332	+259.12%
681776	23142	2746	2	126	0.37246376	0.36564703	+1.86%
681777	22973	2786	2	128	0.79666808	0.73404594	+8.53%
681778	21402	2591	2	120	0.18574904	0.16262119	+14.22%
681779	21443	2593	2	120	0.25666972	0.13265033	+93.49%
681780	22127	2757	3	169	0.34191520	0.34546023	-1.03%
681781	23238	2816	2	131	0.77277192	0.76596395	+0.89%
681782	21336	2558	2	120	0.35848927	0.35196518	+1.85%
681783	20978	2537	2	119	0.25916462	0.23825538	+8.78%
681784	20584	2512	2	116	0.12759041	0.11706145	+8.99%
681785	22647	2778	2	127	0.77748036	0.77849247	-0.13%
681786	20548	2524	2	117	0.16653468	0.15830595	+5.20%
681787	21647	2687	2	123	0.82741649	0.82965631	-0.27%
681788	21385	2631	2	121	0.45095308	0.44175300	+2.08%
681789	20433	2504	2	118	0.26225203	0.25475921	+2.94%
681790	20169	2483	2	115	0.07722830	0.05665851	+36.30%
681791	20418	2463	2	117	0.23601696	0.22984366	+2.69%
681792	21053	2570	2	120	0.55817715	0.55249285	+1.03%
681793	20152	2459	2	114	0.14900975	0.11448200	+30.16%
681794	20136	2473	2	114	0.14462324	0.06667075	+116.92%
681795	20206	2307	2	115	0.37178710	0.35032445	+6.13%
681796	19806	2217	2	140	0.33822887	0.33057839	+2.31%
681797	20913	2413	2	124	1.44319780	1.46291810	-1.35%
681798	25021	3255	2	159	1.53952924	1.55797209	-1.18%
681799	23955	3080	2	156	1.16094207	1.16811242	-0.61%
681800	22997	2848	2	221	0.68890355	0.67315840	+2.34%
681801	21354	2517	2	204	0.32007874	0.31977864	+0.09%
681802	19488	2208	2	186	0.12761411	0.11148755	+14.46%
681803	18646	2033	2	178	0.35288418	0.34142871	+3.36%

Table 1. Experimental Results for Blocks 681734–681803. |\Sigma| is the mempool size, |E| is the number of DCG edges, PW is its pathwidth, and T is our runtime in seconds. The next two columns show the amounts of transaction fees (in BTC) obtained by our algorithm and the miners. The final column is the improvement percentage obtained by our method.

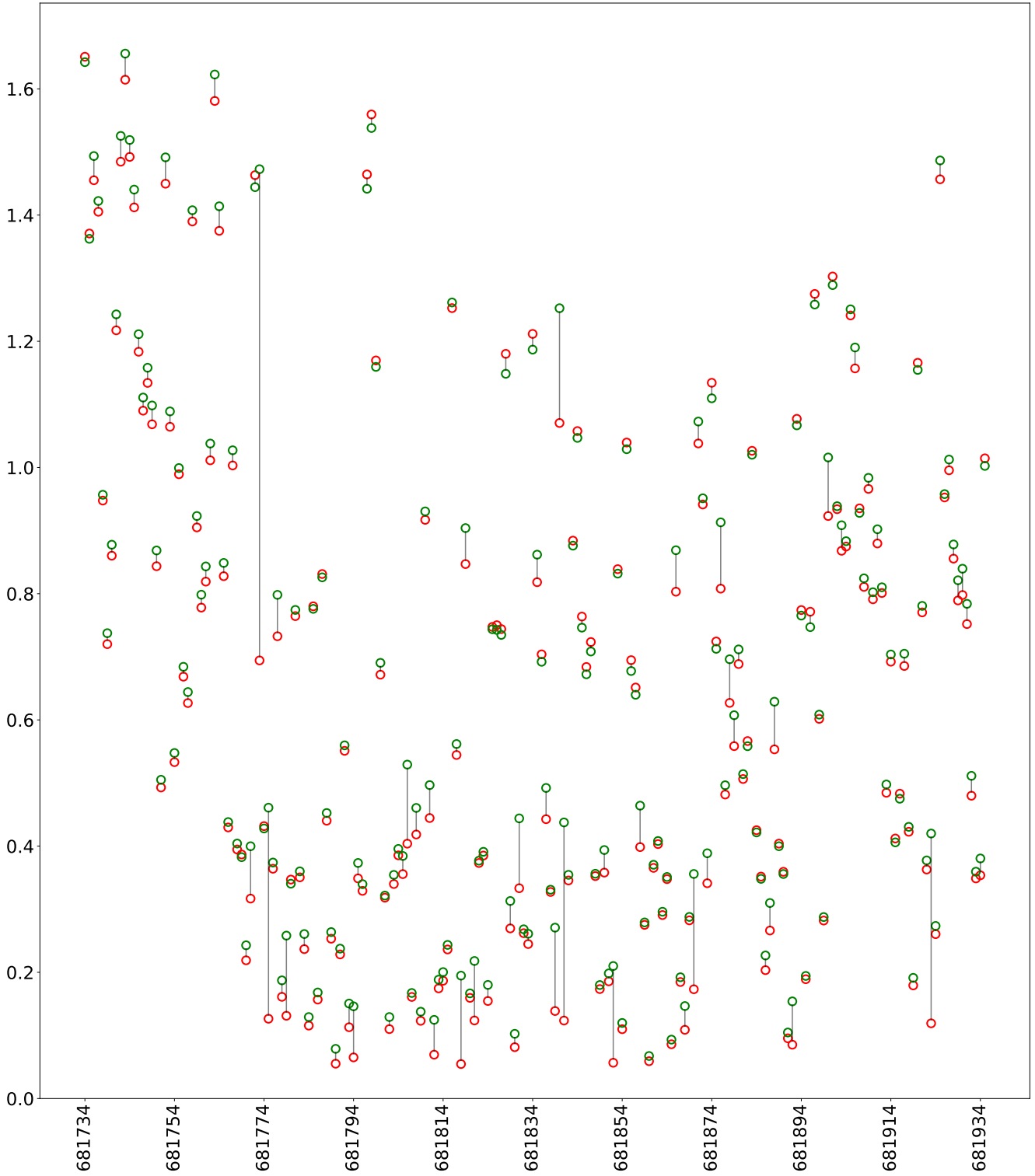


Figure 6. Comparison of fees obtained by our approach (green) and the real-world Bitcoin miners (red). The x axis is the block number and the y axis is the transaction fee revenue in BTC. To increase readability, points corresponding to the same block are connected by a line.

References

- [1] Miner fees. In: Bitcoin Wiki (2020), https://en.bitcoin.it/wiki/Miner_fees#Priority_transactions
- [2] Antonopoulos, A.M., Wood, G.: *Mastering ethereum: building smart contracts and dapps*. O'reilly Media (2018)
- [3] Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k -tree. *SIAM Journal on Algebraic Discrete Methods* **8**(2), 277–284 (1987)
- [4] Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and the hardness of approximation problems. *Journal of the ACM* **45**(3), 501–555 (1998)
- [5] Asadi, A., Chatterjee, K., Goharshady, A.K., Mohammadi, K., Pavlogiannis, A.: Faster algorithms for quantitative analysis of mcs and mdps with small treewidth. In: ATVA. pp. 253–270 (2020)
- [6] Bazán-Palomino, W.: How are bitcoin forks related to bitcoin? *Finance Research Letters* p. 101723 (2020)
- [7] Bhaskar, N.D., Chuen, D.L.K.: Bitcoin mining technology. In: *Handbook of digital currency*, pp. 45–65. Elsevier (2015)
- [8] Blockstream Corporation Inc: *Esplora block explorer*. <https://github.com/Blockstream/esplora> (2021)
- [9] Bodlaender, H.L.: Dynamic programming on graphs with bounded treewidth. In: ICALP. pp. 105–118 (1988)
- [10] Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing* **25**(6), 1305–1317 (1996)
- [11] Bodlaender, H.L.: A partial k -arboretum of graphs with bounded treewidth. *Theoretical computer science* **209**(1-2), 1–45 (1998)
- [12] Bodlaender, H.L., Kloks, T.: Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms* **21**(2), 358–402 (1996)
- [13] Bonneau, J.: Bitcoin mining is NP-hard. Tech. rep. (2014), <https://freedom-to-tinker.com/2014/10/27/bitcoin-mining-is-np-hard/>
- [14] Chatterjee, K., Goharshady, A.K., Goharshady, E.K.: The treewidth of smart contracts. In: SAC. pp. 400–408 (2019)
- [15] Chatterjee, K., Goharshady, A.K., Goyal, P., Ibsen-Jensen, R., Pavlogiannis, A.: Faster algorithms for dynamic algebraic queries in basic rsmns with constant treewidth. *ACM Trans. Program. Lang. Syst.* **41**(4), 23:1–23:46 (2019)
- [16] Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In: POPL. pp. 733–747 (2016)
- [17] Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Pavlogiannis, A.: Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In: ESOP. pp. 112–140 (2020)
- [18] Chatterjee, K., Goharshady, A.K., Ibsen-Jensen, R., Velner, Y.: Ergodic mean-payoff games for the analysis of attacks in crypto-currencies. In: CONCUR (2018)
- [19] Chatterjee, K., Goharshady, A.K., Okati, N., Pavlogiannis, A.: Efficient parameterized algorithms for data packing. In: POPL. pp. 53:1–53:28 (2019)
- [20] Cohen, B., Pietrzak, K.: *The chia network blockchain* (2019)
- [21] Cygan, M., Fomin, F.V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S.: *Parameterized algorithms*. Springer (2015)
- [22] Dagum, L., Menon, R.: OpenMP: An industry-standard API for shared-memory programming. *IEEE Computing in Science and Engineering* **5**(1), 46–55 (1998)
- [23] Dev, J.A.: Bitcoin mining acceleration and performance quantification. In: CCECE. pp. 1–6. IEEE (2014)
- [24] van Dijk, T., van den Heuvel, J.P., Slob, W.: *Computing treewidth with LibTW*. Tech. rep., University of Utrecht (2006)
- [25] Eyal, I., Sirer, E.G.: Majority is not enough: bitcoin mining is vulnerable. *Commun. ACM* **61**(7), 95–102 (2018)
- [26] Fomin, F.V., Lokshtanov, D., Saurabh, S., Pilipczuk, M., Wrochna, M.: Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Transactions on Algorithms (TALG)* **14**(3), 1–45 (2018)
- [27] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., Zeldovich, N.: Algorand: Scaling byzantine agreements for cryptocurrencies. In: SOSP. pp. 51–68 (2017)
- [28] Goharshady, A.K.: *Parameterized and Algebro-geometric Advances in Static Program Analysis*. Ph.D. thesis, Institute of Science and Technology Austria (2020)
- [29] Goharshady, A.K., Mohammadi, F.: An efficient algorithm for computing network reliability in small treewidth. *Reliab. Eng. Syst. Saf.* **193**, 106665 (2020)
- [30] Håstad, J.: Some optimal inapproximability results. *Journal of the ACM* **48**(4), 798–859 (2001)
- [31] Javarone, M.A., Wright, C.S.: From bitcoin to bitcoin cash: a network analysis. In: *Workshop on Cryptocurrencies and Blockchains for Distributed Systems*. pp. 77–81 (2018)
- [32] Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: CRYPTO. pp. 357–388 (2017)
- [33] King, S., Nadal, S.: Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. Tech. rep. (2012)
- [34] Kwon, Y., Kim, H., Shin, J., Kim, Y.: Bitcoin vs. bitcoin cash: Coexistence or downfall of bitcoin cash? In: SP. pp. 935–951. IEEE (2019)
- [35] Laszka, A., Johnson, B., Grossklags, J.: When bitcoin mining pools run dry. In: FC. pp. 63–77. Springer (2015)
- [36] Lewenberg, Y., Bachrach, Y., Sompolinsky, Y., Zohar, A., Rosenschein, J.S.: Bitcoin mining pools: A cooperative game theoretic analysis. In: AAMAS. pp. 919–927 (2015)
- [37] Lombrozo, E., Lau, J., Wuille, P.: Segregated witness (consensus layer). Bitcoin Core Develop. Team, Tech. Rep. BIP **141** (2015)
- [38] McCorry, P., Hicks, A., Meiklejohn, S.: Smart contracts for bribing miners. In: FC. vol. 10958, pp. 3–18 (2018)
- [39] Nakamoto, S.: *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. (2008)
- [40] Naumenko, G., Maxwell, G., Wuille, P., Fedorova, A., Beschastnikh, I.: Erlay: Efficient transaction relay for bitcoin. In: CCS. pp. 817–831 (2019)
- [41] Ohtsuki, T., Mori, H., Kuh, E., Kashiwabara, T., Fujisawa, T.: One-dimensional logic gate assignment and interval graphs. *IEEE Transactions on Circuits and Systems* **26**(9), 675–684 (1979)
- [42] Robertson, N., Seymour, P.D.: Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B* **35**(1), 39–61 (1983)
- [43] Robertson, N., Seymour, P.D.: Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B* **36**(1), 49–64 (1984)
- [44] Taylor, M.B.: The evolution of bitcoin hardware. *Computer* **50**(9), 58–66 (2017)
- [45] The Sage Developers: *SageMath, the Sage Mathematics Software System* (2020)
- [46] Thorup, M.: All structured programs have small tree width and good register allocation. *Information and Computation* **142**(2), 159–181 (1998)
- [47] Velner, Y., Teutsch, J., Luu, L.: Smart contracts make bitcoin mining pools vulnerable. In: FC. pp. 298–316. Springer (2017)
- [48] Wang, C., Chu, X., Qin, Y.: Measurement and analysis of the bitcoin networks: A view from mining pools. In: BIGCOM. pp. 180–188. IEEE (2020)
- [49] Zhang, F., Eyal, I., Escrava, R., Juels, A., van Renesse, R.: REM: resource-efficient mining for blockchains. In: Kirda, E., Ristenpart, T. (eds.) *USENIX Security*. pp. 1427–1444 (2017)
- [50] Zur, R.B., Eyal, I., Tamar, A.: Efficient MDP analysis for selfish-mining in blockchains. In: AFT. pp. 113–131 (2020)