Journal of Combinatorial Optimization (2025) 49:15
https://doi.org/10.1007/s10878-024-01249-0

Optimal blocks for maximizing the transaction fee revenue of Bitcoin miners

Mohsen Alambardar Meybodi[1,2] · **Amir Goharshady**[3] · **Mohammad Reza Hooshmandasl**[4] · **Ali Shakiba**[5]

Accepted: 12 December 2024

Abstract

In this work, we consider a combinatorial optimization problem with direct applications in blockchain mining, namely finding the most lucrative blocks for Bitcoin miners, and propose optimal algorithmic solutions. Our experiments show that our algorithms increase the miners' revenues by more than a million dollars per month. 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

✉ Amir Goharshady
goharshady@cse.ust.hk

Mohsen Alambardar Meybodi
m.alambardar@sci.ui.ac.ir

Mohammad Reza Hooshmandasl
hooshmandasl@uma.ac.ir

Ali Shakiba
ali.shakiba@unsw.edu.au

1 Department of Applied Mathematics and Computer Science, Faculty of Mathematics and Statistics, University of Isfahan, Isfahan 81746-73441, Iran

2 School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran 19395-5746, Iran

3 Departments of Computer Science and Mathematics, Hong Kong University of Science and Technology, Clear Water Bay, New Territories, Hong Kong SAR, China

4 Department of Computer Science, University of Mohaghegh Ardabili, Ardabil, Iran

5 University of New South Wales, Sydney, Australia

Published online: 19 December 2024

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 three well-known graph sparsity parameters, namely treedepth, treewidth and pathwidth, we present exact linear-time parameterized algorithms that are applicable to the real-world instances and obtain optimal results. On the practical side, we provide an extensive experimental evaluation demonstrating that our approach vastly outperforms the current Bitcoin miners in practice, obtaining a significant per-block average increase of 11.34 percent in transaction fee revenues which amounts to almost one million dollars per month.

Keywords Discrete optimization · Combinatorial algorithms · Blockchain

Mathematics Subject Classification 05C75 · 05C83 · 05C85 · 68R01 · 68R05 · 68R10 · 68W40

1 Introduction

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 Nakamoto (2008) 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* (Cohen and Pietrzak 2019) or *validating* (Ethereum Foundation 2021). 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 (Nakamoto 2008), while in typical proof-of-stake protocols she has to win a specific type of lottery (Kiayias et al. 2017; King and Nadal 2012; Gilad et al. 2017).

A significant amount of research and development has been devoted to studying and optimizing the second phase. For Bitcoin alone, there are already several generations of mining hardware (Dev 2014; Taylor 2017; Bhaskar and Chuen 2015), from GPU mining, to FPGA, to dedicated ASICs and trusted hardware frameworks (Zhang et al. 2017). There mining industry is huge and consumes a significant amount of electricity in this phase. Indeed, the cost of becoming a miner is dominated by the electricity usage (Stepanova et al. 2024). Moreover, miners often collaborate in what is known as a mining *pool*, which has also been widely studied in the literature (Lewenberg et al. 2015; Chatterjee et al. 2018; Laszka et al. 2015; Velner et al. 2017; Wang et al. 2020; Zur et al. 2020; Eyal and Sirer 2018; McCorry et al. 2018). Another direction of research focuses on game-theoretic aspects of mining and its effects on other economic problems in the blockchain ecosystem (An et al. 2024; Mikhaylov 2023; Mutalimov et al. 2021; An et al. 2020). 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. In Bitcoin, it is currently 3.125 BTC and halves at predetermined intervals (Nakamoto 2008). 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 (Antonopoulos and Wood 2018).
– *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 (Nakamoto 2008). 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 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 is maximized. The task of forming an optimal block is complicated by several factors, as explained below.

Block Size Limit Every transaction has a known size and blockchain protocols enforce an upper-bound on the size of mined blocks. In Bitcoin, the bound is 1,000,000 bytes (Lombrozo et al. 2015).[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 (Javarone and Wright 2018; Kwon et al. 2019). There are also cryptocurrencies that advocate for larger blocks, and even a total abandonment of block size limits, such as Bitcoin SV (Bazán-Palomino 2020). 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, i.e. if Tx_2 spends funds that were obtained in Tx_1 as shown in Fig. 1, 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

[1] SegWit Lombrozo et al. (2015) 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.

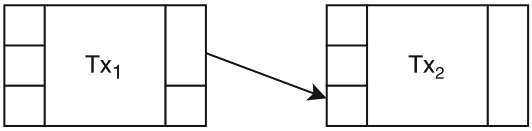

Fig. 1 Tx_2 uses the first output of Tx_1 as its third input. In other words, it spends funds that are obtained and valid only after Tx_1 is added to the blockchain. Hence, Tx_2 depends on Tx_1 and must appear after it in the chain. In practice, Tx_2 has a pointer to Tx_1 but for demonstration purposes, we found it more convenient to show this as an arrow that models the flow of monetary value from Tx_1's output to Tx_2's input

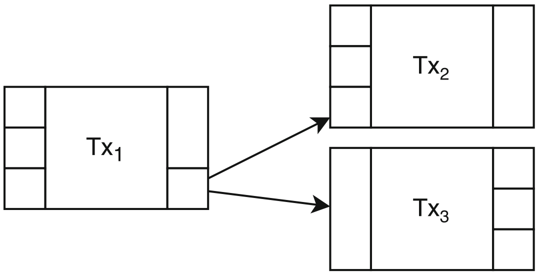

Fig. 2 Tx_2 and Tx_3 both use the same funds (the second output of Tx_1). In other words, the user who received funds in Tx_1 is trying to spend the same coin twice, once in Tx_2 and once in Tx_3. This is considered double-spending and is not allowed. Hence, Tx_2 and Tx_3 are in conflict and a miner cannot include both of them in her block. Note that the miner can choose not to include any of the two transactions

create any additional constraints for the miner if Tx_1 is already on 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, as in Fig. 2, 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 (Miner fees 2020). As such, this approach has been largely abandoned by the miners (Miner fees 2020).

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 over an underlying graph, called the *Dependency-Conflict Graph* (DCG). This is a graph with one vertex per transaction and edges that model dependency and conflict relations.
- ***Negative Results.*** We establish the following hardness results:
  - 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 $\epsilon > 0.12$.
- ***Research Hypothesis.*** We hypothesize that the real-world instances of the DCK problem which arise in Bitcoin mining have sparse graphs. More specifically, we hypothesize that the dependency-conflict graphs in these instances have bounded treewidth, pathwidth and treedepth. We have experimentally verified this hypothesis (Sect. 7).
- ***Positive Results.*** Based on the hypothesis above, we present the following positive results, leading to efficient algorithms:
  - On instances with bounded pathwidth or treedepth, the DCK problem is solvable in $O(n \cdot k)$, whereas it is solvable in $O(n \cdot k^2)$ for instances with bounded treewidth. Here, n is the number of new transactions (also known as mempool size) and k is the block size limit. These pseudo-polynomial algorithms output *optimal* blocks.
  - While it is well-known that treedepth is a stronger parameter than pathwidth, which is in turn a stronger parameter than treewidth, we reach a peculiar observation: the middle parameter, i.e. pathwidth, is actually the best choice for our problem. Pathwidth is preferable to the other two parameters because (i) real-world instances of the problem have relatively larger treedepth than pathwidth, and (ii) the treewidth-based algorithm has a quadratic dependence on the size of the block, whereas the dependence is linear for pathwidth/treedepth.
  - Finally, 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 11.34 percent. This translates to a revenue increase of more than one million dollars per month.

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 (Bonneau 2014). Parameterized algorithms have not been previously studied in the context of blockchain, except for Chatterjee et al. (2019) 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 treedepth, treewidth and pathwidth, which we will later exploit in order to obtain efficient algorithms for optimal mining. Treewidth (Robertson and Seymour 1984) 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 (Robertson and Seymour 1983) is a measure of path-likeness of a graph and treedepth measures the degree to which a graph resembles a star (Aspvall and Heggernes 1994; Nešetřil and De Mendez 2012). Many problems that are NP-hard on general graphs admit efficient solutions when restricted to instances with small treewidth, treedepth or pathwidth (Cygan et al. 2015; Goharshady and Mohammadi 2020; Goharshady 2020; Chatterjee et al. 2019). Even problems that are not NP-hard can often be solved more efficiently when parameterized by these parameters (Asadi et al. 2020; Chatterjee et al. 2020, 2019, 2016; Fomin et al. 2018). We now provide more formal definitions:

Depth Decompositions Nešetřil and De Mendez (2006) Given a connected graph $G = (V, E)$ with vertex set V and edge set E, a *depth decomposition* or *treedepth decomposition* of G is a rooted tree (V, E_T), with the same vertex set as the original graph, which satisfies the following additional constraint:

– For every edge $(u, v) \in E$ of the original graph G, either u is an ancestor of v in the tree or v is an ancestor of u.

Intuitively, a depth decomposition defines a partial order $\prec$ on the vertices of G in which every vertex u is assumed to be smaller than its parent p_u, i.e. $u \prec p_u$, and thus the root is the largest element. In order for the tree to be a valid depth decomposition, every pair (u, v) of vertices that are connected to each other by an edge in the original graph G have to be comparable in $\prec$. If G is not connected, then the depth decomposition will be a forest, consisting of one tree for each connected component of G.

Example 1 Figure 3 shows an example graph together with a depth decomposition.

Treedepth Nešetřil and De Mendez (2006), Nešetřil and De Mendez (2012), Aspvall and Heggernes (1994) The *treedepth* of G is the smallest possible depth among all depth decompositions of G. For example, Fig. 3 shows a depth decomposition of G in which the tree has a depth of 4. So, the treedepth of G is at most 4.

Tree Decompositions and Path Decompositions Robertson and Seymour (1983), Robertson and Seymour (1984), Cygan et al. (2015) 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 $b \in T$ 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.

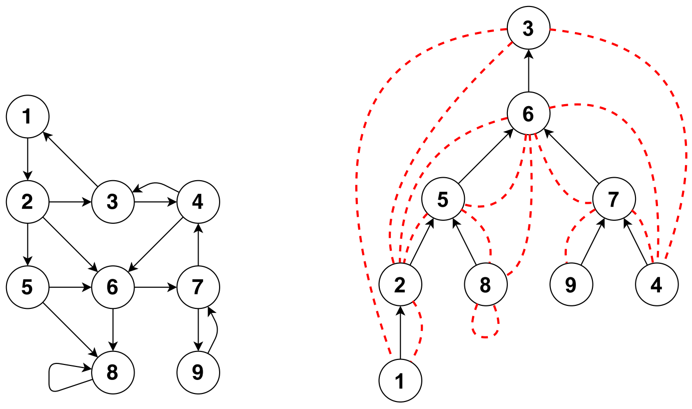

Fig. 3 A graph G (left) and a depth decomposition of G (right). Edges of G are shown by red dotted lines. Note that every edge of the original graph G becomes an ancestor–descendant edge in the decomposition

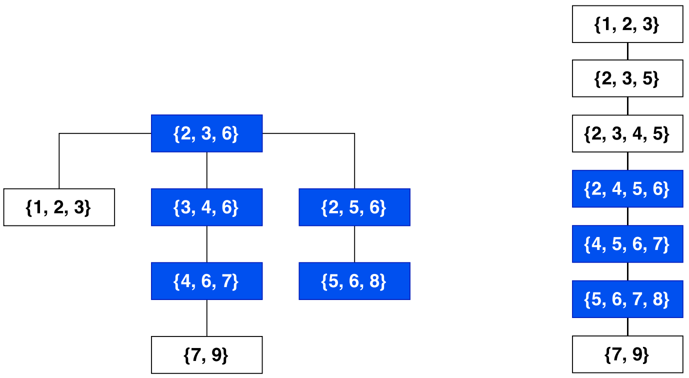

Fig. 4 A tree decomposition of the graph G of Fig. 3 with width 2 (left), and a path decomposition of G with width 3 (right). In each case, the connected subtree containing the vertex 6 is shown in blue. Note that the bags of each decomposition cover every vertex and every edge of the original graph (Color figure online)

- 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 path.
Treewidth and Pathwidth Robertson and Seymour (1983), Robertson and Seymour (1984), Cygan et al. (2015) 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 Figure 4 shows a tree decomposition and a path decomposition for the same graph as in Fig. 3.

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 $V_{b'} = V_b \backslash \{v\}$. 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 $V_b = V_{b_1} = V_{b_2}$.

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 Cygan et al. (2015) 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 Sect. 6 does. See Bodlaender (1988), Cai and Goharshady (2024), Goharshady et al. (2024), Conrado et al. (2024), Conrado et al. (2023), Conrado et al. (2023), Conrado et al. (2023), Ahmadi et al. (2022) for more examples of this type of dynamic programming.

Sparsity and Comparison of Parameters Treedepth, treewidth and pathwidth are graph sparsity parameters, in the sense that a graph with n vertices and treewidth / treedepth / pathwidth t can have at most $O(n \cdot t)$ edges. Thus, the number of edges in a graph with bounded treewidth / treedepth / pathwidth is at most linear in the number of vertices. Moreover, many well-studied families of graphs, such as cacti, series–parallel graphs, outerplanar graphs, and control-flow and call graphs of programs, have constant treewidth (Bodlaender 1998; Thorup 1998; Cai et al. 2025; Goharshady and Zaher 2023; Conrado et al. 2023). It is also well-known that treedepth is a stronger parameter than pathwidth, which is in turn stronger than treewidth (Nešetřil and De Mendez 2006, 2012; Aspvall and Heggernes 1994). More precisely, for any graph G with n vertices, we have:

$$\text{treewidth}(G) \leq \text{pathwidth}(G) \leq \text{treedepth}(G) \leq \text{treewidth}(G) \cdot O(\lg n).$$

Thus, any problem that is solvable in polynomial time for graphs of bounded treewidth is also solvable in polynomial time for bounded pathwidth, and ditto for pathwidth vs treedepth.

Fixed-parameter Tractability Cygan et al. (2015) 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 the three sparsity parameters.

Computing the Sparsity Parameters Computing the treedepth, treewidth or pathwidth of an arbitrary input graph are NP-hard problems (Arnborg et al. 1987; Ohtsuki et al. 1979; Nešetřil and De Mendez 2006). However, the problems are FPT when parameterized by the parameter itself. Indeed, Bodlaender (1996), Bodlaender and Kloks (1996) provide *linear-time* FPT algorithms for computing treewidth and pathwidth and similar algorithms for treedepth are provided in Nešetřil and De Mendez (2006), Aspvall and Heggernes (1994). Moreover, there are efficient tools and libraries, such as The Sage Developers (2020), van Dijk et al. (2006) that compute these parameters. 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. Similarly, we assume that a depth decomposition of optimal depth is part of the input, too.

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.
- $V, W : \Sigma \to \mathbb{N} \cup \{0\}$ 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. This is because any set of items with cyclic dependencies can be merged into a single item. 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.

Independence of C and D In the real-world scenario of forming a block, two transactions have a conflict if and only if they both use the same output. In other words, any pair of conflicting transactions should have a common dependency. However, note that this common dependency might already be added to the blockchain. In our modeling as a DCK instance, each item corresponds to a *new* transaction that is not yet added to the chain. As such, we do not need to posit extra requirements on the relation between C and D. Indeed, for any given *DCK* instance, it is easy to come up with a block formation problem that exactly corresponds to it.

DCG Given $I = (n, k, \Sigma, V, W, C, D)$, 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 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.

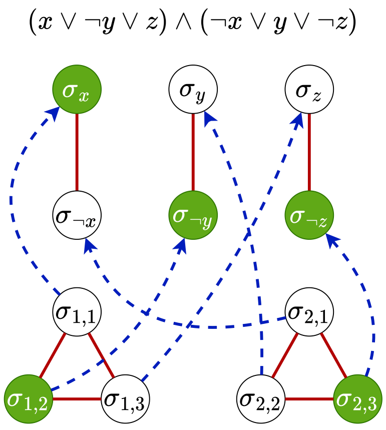

Fig. 5 A 3- SAT formula (top), and its reduction to a DCK instance. Red edges denote conflict and blue edges dependency. For example, $\sigma_{1,1}$ depends on σ_x. In the DCK instance, we have $k = 5$. The variable elements and their negations have value zero, whereas $\sigma_{i,j}$'s have unit value. 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 $y = z = 0$ (Color figure online)

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

- $n = 3 \cdot \overline{m} + 2 \cdot \overline{n}$
- $k = \overline{m} + \overline{n}$
- 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 3 Figure 5 illustrates our reduction.

It is easy to verify that I has all the requirements for a DCK instance as defined in Sect. 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 $\text{DCK}(I_\varphi, \overline{m}) = 1$. Moreover, $\text{MaxDCK}(I_\varphi) = \text{Max3- SAT}(\varphi)$. Hence, we have the following theorems:

Theorem 1 (Strong NP-hardness) DCK *is strongly NP-hard. In other words, it is NP-hard even if the input instance size is defined to be $n + k$.*

Proof As shown above, for every 3- SAT formula φ, we have 3- SAT(φ) $\Leftrightarrow$ DCK($I_\varphi, \overline{m}$). 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 2 (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 (Arora et al. 1998) is that such an ϵ exists for the MAX3- SAT problem. The theorem follows from the fact that we have MAXDCK(I_φ) $=$ MAX3- SAT(φ) in our reduction. Indeed, it is hard to approximate MAX3- SAT within a ratio of $\frac{7}{8} + \epsilon$ for any $\epsilon > 0$ (Håstad 2001). Using our reduction, this result also applies to MAXDCK. $\square$

5 An efficient algorithm based on treedepth

We now provide an efficient linear-time FPT algorithm for the DCK problem. We use the treedepth d of the DCG as our parameter.

Setup and Notation Let $I = (n, k, \Sigma, V, W, C, D)$ be a DCK instance which is given to us as input together with an optimal depth decomposition $T = (V, E_T)$ of its DCG $G = (\Sigma, E)$. The main property of a depth decomposition is that any vertex $v \in V$ can have G-edges only to vertices that are either its ancestors or its descendants in the tree T. Specifically, note that if v is a leaf in T, then it can have G-edges to its ancestors only. We use this property to define certain subgraphs of G that would later enable a left-to-right dynamic programming algorithm for MAXDCK. For every vertex $v \in V$, let $A(v)$ be the set of all ancestors of V in the tree T, including v itself. Of course, we have $|A(v)| \leq d + 1$. Suppose that we run a pre-order traversal of the tree T and label the vertices from left to right as $a_1, a_2, \ldots, a_n$. We define the i-th canonical induced subgraph of G as:

$$\langle G, i\rangle := G\left[\{a_1, a_2, \ldots, a_i\}\right].$$

In other words, $\langle G, i\rangle$ contains the first i vertices of G from left to right, i.e. in pre-order. Similarly, we define canonical subtrees:

$$\langle T, i\rangle := T\left[\{a_1, a_2, \ldots, a_i\}\right].$$

Note that $\langle T, i\rangle$ is a valid depth decomposition for $\langle G, i\rangle$.

Example 4 Figure 6 shows a DCG G and a depth decomposition of G. Conflict edges are shown in red and dependencies in blue. We have labeled the vertices based on a pre-order traversal. The canonical graph $\langle G, 2\rangle$ will contain the vertices $\{a_1, a_2\} = \{2, 3\}$ and the conflict edge between them.

Main Idea Our algorithm is a left-to-right dynamic programming which computes answers for the subproblems defined by the canonical subgraphs $\langle G, i\rangle$. More specifically, for every $1 \leq i \leq n$, every knapsack capacity $\kappa \leq k$, and every subset

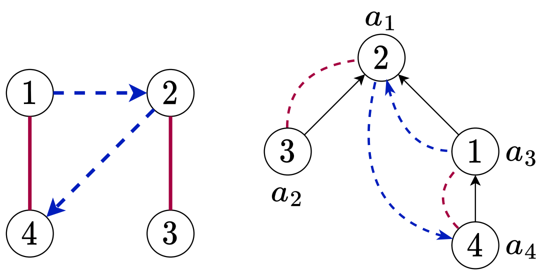

Fig. 6 A DCG G (left) and a depth decomposition T of G with depth 2 (right)

$M \subseteq A(a_i)$, we define a dynamic programming variable $dp[i, M, \kappa]$. Our goal is to compute values for these variables such that at the end of the algorithm we have the following desirable invariant:

– Every $dp[i, M, \kappa]$ contains the maximum possible value that can fit in a knapsack of size κ assuming that we can only take a subset of the first i items, i.e. $\{a_1, a_2, \dots, a_i\}$, into the knapsack and have to respect only the dependency and conflict edges in the canonical subgraph $\langle G, i\rangle$.

Intuition The intuition behind the way we defined our dynamic programming variables is that $\langle G, i\rangle$ has exactly one vertex more than $\langle G, i-1\rangle$ and that vertex is a_i. Moreover, a_i is guaranteed to be a leaf in $\langle G, i\rangle$. Thus, it can only have dependencies and conflicts with its ancestors $A(a_i)$. So, if we know exactly which subset M of the ancestors are supposed to go into the knapsack, then we can fill in the rest of our knapsack using information computed at $\langle G, i-1\rangle$. We are now ready to provide the details of how the dp values are computed.
Computations at $\langle G, 1\rangle$ The canonical subgraph $\langle G, 1\rangle$ has a single vertex a_1. Thus, we have:

$$\begin{aligned} dp[1, \emptyset, \kappa] &= 0 \quad \text{for all } \kappa, \\ dp[1, \{a_1\}, \kappa] &= V(a_1) \quad \text{for all } \kappa \geq W(a_1), \text{ and} \\ dp[1, \{a_1\}, \kappa] &= -\infty \quad \text{for all } \kappa < W(a_1) \end{aligned}$$

In other words, if a_1 fits in the knapsack, we will get its value. Note that the second parameter in our dp, i.e. M, is the exact set of ancestors of a_1 that are included in the knapsack. Thus, if M includes a_1 but a_1 does not fit in κ, we have an impossible situation and the dp value should be set to $-\infty$ to model this.

Example 5 In Fig. 6, suppose every vertex has a value equal to its number and unit weight, e.g. $V(a_1) = 2$ and $V(a_3) = 1$. We have $dp[1, \emptyset, 2] = 0$, $dp[1, \{a_1\}, 0] = -\infty$, and $dp[1, \{a_1\}, 1] = 2$.

Computations at $\langle G, i\rangle$ For every $i > 1$, to compute $dp[i, M, \kappa]$, our algorithm first checks whether the constraints on the new vertex a_i are satisfied. More specifically, it checks that M conforms with all the conflict and dependency requirements modeled by edges connecting a_i to its ancestors. If not, it sets $dp[i, M, \kappa] = -\infty$. Note that this check can easily be performed using the information we have in our dynamic programming parameters since a_i can only have edges to $A(a_i)$ in $\langle G, i\rangle$ due to its status as a leaf and M tells us exactly which subset of $A(a_i)$ is in our knapsack. Additionally, we check that the sum of the weights of the items included in M is at most κ. If $\Sigma_{v\in M} W(v) > \kappa$, then we set $dp[i, M, \kappa] = -\infty$.

If these checks pass, we are sure that only the constraints in $\langle G, i-1\rangle$ can be violated. Moreover, M tells us whether a_i is itself going to be in the knapsack (if $a_i \in M$) or not (if $a_i \notin M$). Let $M' \subseteq A(a_{i-1})$ be a subset of ancestors of a_{i-1}. We say M' is compatible with M and write $M' \risingdotseq M$ if they agree on every vertex, i.e. if for every $v \in A(a_i) \cap A(a_{i-1})$ that is a common ancestor of a_i and a_{i-1}, we have $v \in M \Leftrightarrow v \in M'$. Since we know all elements in M have to be in the knapsack, our algorithm simply tries every M' that is compatible with m and computes:

$$dp[i, M, \kappa] = \begin{cases} \max_{M' \risingdotseq M} dp[i-1, M', \kappa - W(a_i)] + V(a_i) & a_i \in M \\ \max_{M' \risingdotseq M} dp[i-1, M', \kappa] & a_i \notin M \end{cases}.$$

This step is exactly like standard 0- 1 Knapsack. In the first case, a_i is in our knapsack, so the capacity decreases by its weight $W(a_i)$ and the total value increase by $V(a_i)$. In the second case, a_i is excluded from the knapsack.

Example 6 Continuing with our running example of Fig. 6, let us compute some *dp* values. Our algorithm computes these values from left to right, but in this example we do not follow this order for better illustration.

- We have $dp[3, \{a_3\}, 3] = -\infty$ since $a_1 = 2$ is a prerequisite of $a_3 = 1$ and this condition is violated by $M = \{a_3\}$.
- Similarly, we have $dp[4, \{a_4, a_3, a_1\}, 10] = 0$ since a_4 and a_3 are in conflict and are both included in M.
- Now consider $dp[3, \emptyset, 2]$. Here, $M = \emptyset$ and $a_{i-1} = a_2 = 3$. There are two values of $M' \subseteq A(a_{i-1})$ that are compatible with M and they are $\emptyset$ and $\{a_2\}$. We thus have
  $dp[3, \emptyset, 2] = \max\{dp[2, \emptyset, 2], dp[2, \{a_2\}, 2]\} = \max\{0, V(a_2)\} = 3$.
- Finally, let's consider $dp[4, \{a_4\}, 3]$. In this case, $M = \{a_4\}$ satisfies all the conditions imposed by edges connected to vertex $a_4 = 4$. Moreover, $A(a_3) = \{a_3, a_1\}$ and the only $M' \subseteq A(a_3)$ that is compatible with M is $\emptyset$. Thus, we have $dp[4, \{a_4\}, 3] = 4 + dp[3, \emptyset, 2] = 4 + 3 = 7$.

Computing the Final Answer Since we have $\langle G, n\rangle = G$, the final answer to our MaxDCK instance is simply the largest value computed at $\langle G, n\rangle$. Thus, our algorithm returns

$$\max_{M \subseteq A(a_n)} dp[n, M, k].$$

The optimal combination of items that should be put into the knapsack can be reconstructed easily by keeping a pointer to the *dp* value that led to the maximum result in each of our computations above. This is a standard process in dynamic programming and exactly the same as 0- 1 Knapsack.

Example 7 In the previous example, if we have $k = 3$, i.e. we are trying to fill a knapsack of capacity 3, then the largest *dp* value at a_4 would be $dp[4, \{a_4\}, 3]$, which we have already computed as 7. The optimal solution is to take the vertices $\{a_2, a_4\} = \{3, 4\}$ into the knapsack.

Based on the algorithm and discussion above, we have the following theorem:

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

Proof We have already argued for the correctness of our algorithm. To analyze its runtime, note that we have $O(n \cdot k \cdot 2^d)$ dynamic programming variables since i can range from 1 to n, κ is always bounded by k and M is a subset of ancestors of a_i, which can have at most $d + 1$ ancestors, counting itself. To compute each $dp[i, M, \kappa]$, we first have to check the edges going out of a_i, but there are at most d of these. Then, we have to take the maximum over all $M' \subseteq A(a_{i-1})$ that are compatible with M. We argue that each M' is included in the max computation for at most two different values of M. If a_i is a child of a_{i-1} in T, then $A(a_i) = A(a_{i-1}) \cup \{a_i\}$. Thus, there are only two M values that can be compatible with any given M', namely M' itself and $M' \cup \{a_i\}$. Otherwise, let p_i be the parent of a_i. By pre-order construction, we know that a_{i-1} is a descendant of p_i. Thus, $A(a_i) \cap A(a_{i-1}) = A(p_i)$. This means every M' uniquely determines whether each vertex in $A(p_i)$ should be in the knapsack or not. Thus, the only two M values that are compatible with M' are $M' \cap A(p_i)$ and $M' \cap A(p_i) \cup \{a_i\}$. Hence, our total runtime is dominated by the checks rather than max computations and is $O(n \cdot k \cdot d \cdot 2^d)$. □

6 Efficient algorithms based on treewidth and pathwidth

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 $O(n \cdot k \cdot 2^p \cdot p^3)$. Note that the latter is faster by a factor of k, so if k is large and both t and p are small, the pathwidth-based algorithm would be much faster in practice.
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 Sect. 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 $G_b^{\downarrow}$

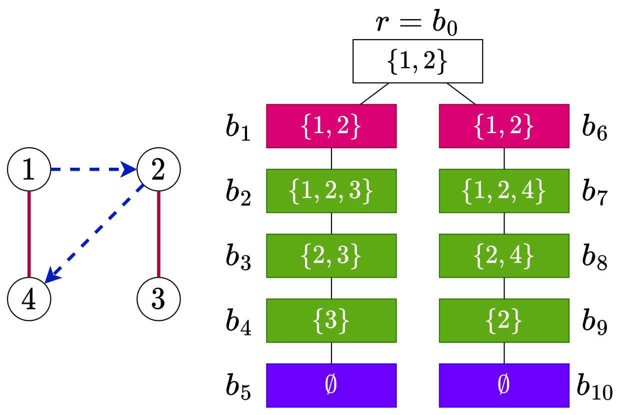

Fig. 7 A DCG G (left) and a nice tree decomposition of G (right). Leaf bags are shown in blue, introduce bags in green, and forget bags in red. The only join bag is the root. We will use this graph as our running example (Color figure online)

as the part of G that corresponds to $T_b^{\downarrow}$. More formally,

$$G_b^{\downarrow} := \left(\bigcup_{b' \in T_b^{\downarrow}} \Sigma_{b'}, \bigcup_{b' \in T_b^{\downarrow}} E_{b'} \right).$$

Example 8 Figure 7 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}.

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 9 In the instance of Example 8 (Fig. 7), the algorithm computes $dp[b_5, \emptyset, \kappa] = dp[b_{10}, \emptyset, \kappa] = 0$ for all $0 \le \kappa \le 3$.

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 $G^{\downarrow}_{b'}$. If $\sigma \notin M$, there is no gain in value and no loss in space, and the knapsack should be filled using $G^{\downarrow}_{b'}$. In both cases, we of course have to respect all the conflicts and dependencies in $G^{\downarrow}_{b'}$, too. This is modeled by $dp[b', \cdot, \cdot]$.

Example 10 In the instance of Example 8 (Fig. 7), 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 $G^{\downarrow}_b = G^{\downarrow}_{b'}$. 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 $G^{\downarrow}_b$ and $G^{\downarrow}_{b'}$ 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 11 In the instance of Example 8 (Fig. 7), 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:

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

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 $dp[b_6, \{1,2\}, 3]$ as

$$\max\{dp[b_7, \{1,2\}, 3], dp[b_7, \{1,2,4\}, 3]\}.$$

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 $V(M) = \Sigma_{m \in M} V(m)$, i.e. the total value of items in M, and $W(M) = \Sigma_{m \in M} W(m)$, i.e. the total weight of items in M. If $W(M) > \kappa$, it sets $dp[b, M, \kappa] = -\infty$. Otherwise, it computes $dp[b, M, \kappa]$ to be equal to

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

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 12 In the instance of Example 8 (Fig. 7), the only join bag is the root $r = b_0$. The algorithm computes $dp[r, \{2\}, 2] = \max\{dp[b_1, \{2\}, 1] + dp[b_6, \{2\}, 2], dp[b_1, \{2\}, 2] + dp[b_6, \{2\}, 1]\} - V(2) = \max\{2+6, 6+2\} - 2 = 6$.

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 $G_{b_2}^{\downarrow}$ (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 subtract 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 13 In our example, the solution is:

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

which can be achieved by taking items 3 and 4.

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

Theorem 4 *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. □

Theorem 5 *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 4, except that a nice path decomposition has no join nodes, and hence the algorithm is faster by a significant factor of k. □

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 $dp[b, ., .]$ 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

$$O\left(\frac{n \cdot k^2}{\theta}\right) \text{ and } O\left(\frac{n \cdot k}{\theta}\right)$$

for instances with bounded treewidth and pathwidth, respectively.

7 Implementation and experimental results

We implemented our algorithm in **C++** and used `OpenMP` (Dagum and Menon 1998) for parallelization. We used the Esplora Block Explorer (Blockstream Corporation Inc 2021) to collect information about the Bitcoin blockchain. This includes details of the transactions in each block and the mempool, i.e. transactions that are published but not yet mined. We computed the decompositions using `SageMath` (The Sage Developers 2020).

Machine All results were obtained on a machine with an Intel Xeon 6226R (2.9 GHz, 16 Cores, 22 MB Cache), running Ubuntu 20.04 LTS with 135 GB of RAM and a total of 32 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.

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 sparsity parameters, namely treewidth, pathwidth and treedepth. 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/star-like structure. 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. We might normally have a chain of dependencies, which would create a path-like structure, or multiple spendings of the outputs of the same transaction which will create a star-like dependency structrue.

Benchmarks We ran two real-time experiments on live Bitcoin data. In the first experiment, we considered blocks number 681,734 to 681,935 in the Bitcoin Blockchain, whereas the second experiment considered blocks 747,429 to 752,075. The 202 blocks in the first experiment correspond to almost 27 h of activity, whereas the 4,646 blocks in the second experiment capture just over a month. We updated our mempool every 5 min using live Bitcoin data as provided by Blockstream Corporation Inc (2021). 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 less than 5 min on our machine. To ensure that we are not obtaining any unfair advantage, we set the interval to 5 min. We used this live mempool as the set Σ of all possible items. We then ran our algorithm to obtain an optimal block.

Baseline We compared the total transaction fees obtained by our solution with transaction fees earned by the miner of the actual block on the Blockchain.

Note that real-world miners could have a better view of the network and hence form a block using a larger set of transactions. This can happen, for example, when a miner is withholding information about a particular lucrative transaction. Hence, a direct comparison of the final transaction fees puts us in a relative disadvantage. Despite this, we manage to beat the real-world miners convincingly.

Results of the First Experiment The results of our first experiment are shown in Fig. 8. We now discuss them in more detail:

- **Widths.** We observed that the pathwidth of the instances occurring during this 27-hour window was at most 3. Given that 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 8 shows the amount of transaction fees obtained by our algorithm (the pathwidth variant) 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 time.[2]
- **Runtimes.** Our runtimes ranged from 114 s to 277 s per block, and the average runtime was **175 s**. Note that in Bitcoin, a new block is mined roughly every 10 min. 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.

Results of the Second Experiment The purpose of our second experiment was to consider a longer timeframe of a month instead of a day and confirm whether the improvements remain qualitatively similar. We now discuss various findings of our second experiment:

- **Widths and Depths.** Figure 9 provides a bar chart of the number of instances with each particular treedepth, treewidth and pathwidth. This demonstrates that our central hypothesis holds in the real world and the widths and depths are small.

[2] Note 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.

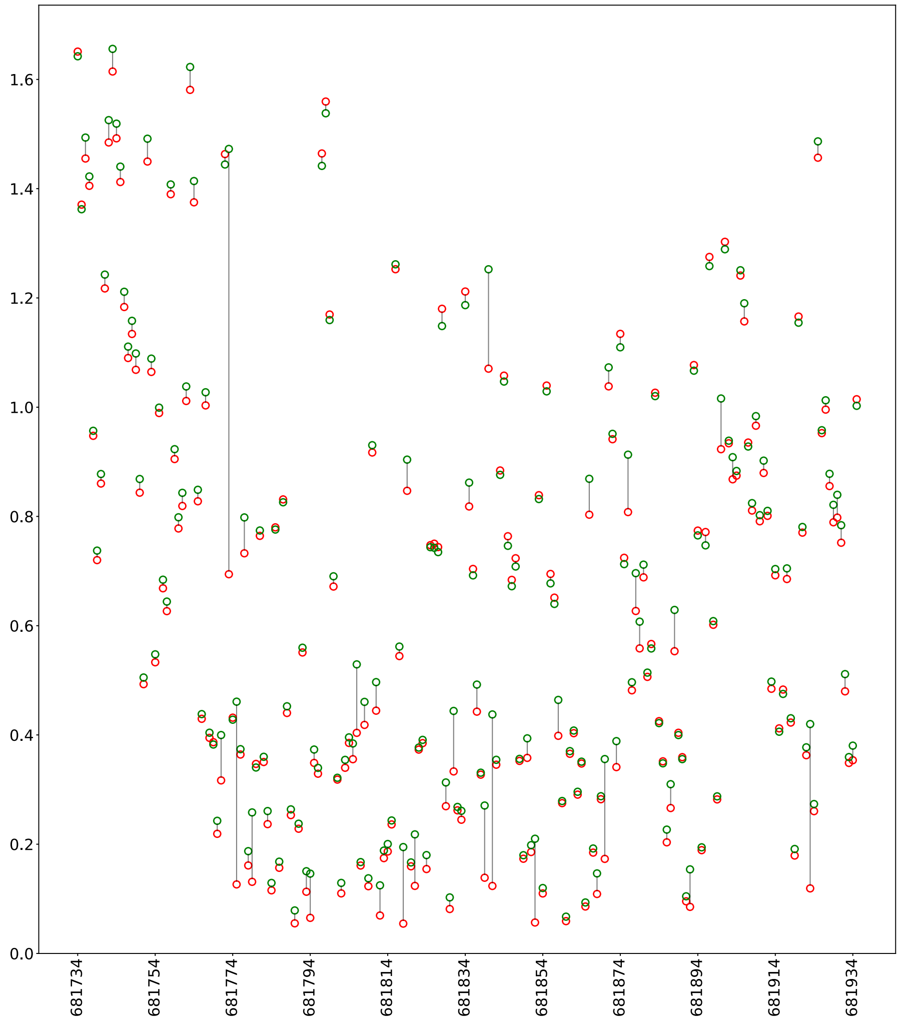

Fig. 8 Comparison of fees obtained by our approach (green) and the real-world Bitcoin miners (red) in our first experiment. The x axis is the block number and the y axis is the transaction fee revenue in BTC. To increase readability, points corresponding to the same block are connected by a line segment (Color figure online)

However, we observe that the treedepth can grow to as large as 10, whereas the pathwidth is at most 4 in all but one of the instances. Moreover, the treedepth is consistently larger than the pathwidth. Thus, given the exponential dependence of our runtime on the depth/width, our pathwidth-based algorithm is preferable to the treedepth-based variant. On the other hand, we observe that the distributions of pathwidth and treewidth have little qualitative difference. Given that the pathwidth is at most 6 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 $O(n \cdot$

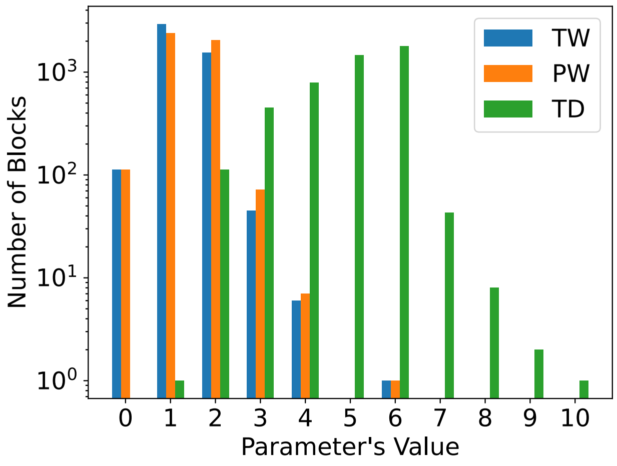

Fig. 9 Distribution of the parameters over the MAXDCK instances in our experimental results

k^2). Thus, surprisingly, the middle parameter (pathwidth) is the most practical parameter. All runtimes and results reported below are using our pathwidth-based algorithm.

- **Transaction Fee Revenues.** In the almost one-month period of our experiment, the miners earned a total of 373.0596 BTC in transaction fees. In contrast, our approach obtained 415.3654 BTC. Thus, using our algorithms, the miners could increase their total revenue by **42.3058 BTC** which is an **11.34 percent increase** and amounts to an astonishing improvement of **1,012,843 USD** in just one month. Our approach obtains a maximum per-block improvement of **216 percent** in transaction fee revenues. The average per-block improvement is **13.8 percent**. On the other hand, the median improvement is **1.16 percent**, showing that the heursitics used by the miners are working well in practice and many of the real-world miners' blocks were actually close to optimal. This being said, the difference between heuristics and the optimal transaction fees obtained by our method is quite significant in practice and accounts for more than a million dollars per month.
- **Runtimes.** Our runtimes ranged from 0.1s to 173.1s, and the average runtime was **56.45s**.

8 Concluding remarks

We now discuss the potential limits of applicability of our results to other blockchains and threats to their validity in the future.

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, i.e. sparsity, 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. As shown in Sect. 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 min. 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 the various types of graph 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.

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 almost a million dollars per month. 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.

9 Extended abstract

A 7-page extended abstract of this work appeared in Meybodi et al. (2022) (IEEE International Conference on Blockchain). In comparison to Meybodi et al. (2022), the current version adds the following new content: the entirety of Sects. 4 and 5, two new sparsity parameters, a more general algorithm in Sect. 6, NP-hardness and hardness-of-approximation results, several new figures, significantly more extensive experiments over a longer timeframe, arguments for the correctness of the algorithms, and detailed runtime analyses. Following theoretical computer science norms, authors are ordered alphabetically.

Funding Open access funding provided by Hong Kong University of Science and Technology The research was partially supported by the IPM Grant CS1401-4-201, an Ethereum Foundation Research Grant, the Hong Kong Research Grants Council ECS Project Number 26208122 and the HKUST-Kaisa Joint Research Institute Project Grant HKJRI3A-055.

Data Availability The datasets generated and/or analyzed during the current study will be made publicly available on Zenodo. The underlying code will also be published as free and open-source software.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

Ahmadi A, Daliri M, Goharshady AK, Pavlogiannis A (2022) Efficient approximations for cache-conscious data placement. In: PLDI, pp 857–871

An J, Mikhaylov A, Chang T (2024) Relationship between the popularity of a platform and the price of NFT assets. Financ Res Lett 61:105057

An J, Mikhaylov A, Jung S-U (2020) The strategy of South Korea in the global oil market. Energies 13:2491

Antonopoulos AM, Wood G (2018) Mastering ethereum: building smart contracts and dapps. O'reilly Media

Arnborg S, Corneil DG, Proskurowski A (1987) Complexity of finding embeddings in a k-tree. SIAM J Algebr Discret Methods 8:277–284

Arora S, Lund C, Motwani R, Sudan M, Szegedy M (1998) Proof verification and the hardness of approximation problems. J ACM 45:501–555

Asadi A, Chatterjee K, Goharshady AK, Mohammadi K, Pavlogiannis A (2020) Faster algorithms for quantitative analysis of MCs and MDPs with small treewidth. ATVA 2020:253–270

Aspvall B, Heggernes P (1994) Finding minimum height elimination trees for interval graphs in polynomial time. BIT Numer Math 34:484–509

Bazán-Palomino W (2020) How are Bitcoin forks related to Bitcoin?. Financ Res Lett 101723

Bhaskar ND, Chuen DLK (2015) Bitcoin mining technology. In: Handbook of digital currency. Elsevier, pp 45–65

Blockstream Corporation Inc, Esplora block explorer (2021). https://github.com/Blockstream/esplora

Bodlaender HL (1988) Dynamic programming on graphs with bounded treewidth. In: ICALP, pp 105–118

Bodlaender HL (1996) A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J Comput 25:1305–1317

Bodlaender HL (1998) A partial k-arboretum of graphs with bounded treewidth. Theor Comput Sci 209:1–45

Bodlaender HL, Kloks T (1996) Efficient and constructive algorithms for the pathwidth and treewidth of graphs. J Algorithms 21:358–402

Bonneau J (2014) Bitcoin mining is NP-hard, tech. report. https://freedom-to-tinker.com/2014/10/27/bitcoin-mining-is-np-hard/

Cai X, Goharshady AK (2024) Faster lifetime-optimal speculative partial redundancy elimination for goto-free programs. In: SETTA

Cai X, Goharshady AK, Hitarth S, Lam CK (2025) Faster register allocation via grammatical decompositions of control-flow graphs. In: ASPLOS

Chatterjee K, Goharshady AK, Goharshady EK (2019) The treewidth of smart contracts. In: SAC, pp 400–408

Chatterjee K, Goharshady AK, Goyal P, Ibsen-Jensen R, Pavlogiannis A (2019) Faster algorithms for dynamic algebraic queries in basic RSMs with constant treewidth. ACM Trans Program Lang Syst 41:23:1-23:46

Chatterjee K, Goharshady AK, Ibsen-Jensen R, Pavlogiannis A (2016) Algorithms for algebraic path properties in concurrent systems of constant treewidth components. In: POPL, pp 733–747

Chatterjee K, Goharshady AK, Ibsen-Jensen R, Pavlogiannis A (2020) Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In: ESOP, pp 112–140
Chatterjee K, Goharshady AK, Ibsen-Jensen R, Velner Y (2018) Ergodic mean-payoff games for the analysis of attacks in crypto-currencies. In: CONCUR
Chatterjee K, Goharshady AK, Okati N, Pavlogiannis A (2019) Efficient parameterized algorithms for data packing. In: POPL, pp 53:1–53:28
Cohen B, Pietrzak K (2019) The Chia network blockchain. https://www.chia.net/assets/ChiaGreenPaper.pdf
Conrado GK, Goharshady AK, Hudec P, Li P, Motwani HJ (2024) Faster treewidth-based approximations for Wiener index. In: SEA
Conrado GK, Goharshady AK, Kochekov K, Tsai YC, Zaher AK (2023) Exploiting the sparseness of control-flow and call graphs for efficient and on-demand algebraic program analysis. Proc ACM Program Lang 7:1993–2022
Conrado GK, Goharshady AK, Lam CK (2023) The bounded pathwidth of control-flow graphs. Proc ACM Program Lang 7:292–317
Cygan M, Fomin FV, Kowalik Ł, Lokshtanov D, Marx D, Pilipczuk M, Pilipczuk M, Saurabh S (2015) Parameterized algorithms. Springer
Dagum L, Menon R (1998) OpenMP: an industry-standard API for shared-memory programming. IEEE Comput Sci Eng 5:46–55
Dev JA (2014) Bitcoin mining acceleration and performance quantification. In: CCECE, pp 1–6
Ethereum Foundation (2021) Stake your ETH to become an Ethereum validator. https://ethereum.org/en/eth2/staking/
Eyal I, Sirer EG (2018) Majority is not enough: Bitcoin mining is vulnerable. Commun ACM 61:95–102
Fomin FV, Lokshtanov D, Saurabh S, Pilipczuk M, Wrochna M (2018) Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. ACM Trans Algorithms (TALG) 14:1–45
Gilad Y, Hemo R, Micali S, Vlachos G, Zeldovich N (2017) Algorand: scaling byzantine agreements for cryptocurrencies. In: SOSP, pp 51–68
Goharshady AK (2020) Parameterized and Algebro-geometric Advances in Static Program Analysis, PhD thesis, Institute of Science and Technology Austria
Goharshady AK, Lam CK, Parreaux L (2024) Fast and optimal extraction for sparse equality graphs. Proc ACM Program Lang 8:2551–2577
Goharshady AK, Mohammadi F (2020) An efficient algorithm for computing network reliability in small treewidth. Reliab Eng Syst Saf 193:106665
Goharshady AK, Zaher AK (2023) Efficient interprocedural data-flow analysis using treedepth and treewidth. In: VMCAI, vol 3881, pp 177–202
Håstad J (2001) Some optimal inapproximability results. J ACM 48:798–859
Javarone MA, Wright CS (2018) From Bitcoin to Bitcoin cash: a network analysis. In: Workshop on Cryptocurrencies and Blockchains for Distributed Systems, pp 77–81
Kiayias A, Russell A, David B, Oliynykov R (2017) Ouroboros: a provably secure proof-of-stake blockchain protocol. In: CRYPTO, pp 357–388
King S, Nadal S (2012) Ppcoin: peer-to-peer crypto-currency with proof-of-stake, tech. report
Kwon Y, Kim H, Shin J, Kim Y (2019) Bitcoin vs. Bitcoin cash: Coexistence or downfall of Bitcoin cash?. In: SP, pp 935–951
Laszka A, Johnson B, Grossklags J (2015) When Bitcoin mining pools run dry. In: FC, pp 63–77
Lewenberg Y, Bachrach Y, Sompolinsky Y, Zohar A, Rosenschein JS (2015) Bitcoin mining pools: a cooperative game theoretic analysis. In: AAMAS, pp 919–927
Lombrozo E, Lau J, Wuille P (2015) Segregated witness (consensus layer), Bitcoin Core Develop. Team, Tech. Rep. BIP, p 141
McCorry P, Hicks A, Meiklejohn S (2018) Smart contracts for bribing miners. In: FC, vol 10958, pp 3–18
Meybodi MA, Goharshady AK, Hooshmandasl MR, Shakiba A (2022) Optimal mining: Maximizing bitcoin miners' revenues from transaction fees. In: Blockchain. IEEE, pp 266–273
Mikhaylov A (2023) Understanding the risks associated with wallets, depository services, trading, lending, and borrowing in the crypto space. J Infrastruct Policy Dev 7:2223
Miner Fees (2020) In Bitcoin Wiki . https://en.bitcoin.it/wiki/Miner_fees#Priority_transactions
Mutalimov V, Kovaleva I, Mikhaylov A, Stepanova D (2021) Assessing regional growth of small business in Russia. Entrep Bus Econ Rev 9:119–133
Nakamoto S (2008) Bitcoin: A peer-to-peer electronic cash system, tech. report

Nešetřil J, De Mendez PO (2006) Tree-depth, subgraph coloring and homomorphism bounds. Eur J Comb 27:1022–1041

Nešetřil J, De Mendez PO (2012) Sparsity: graphs, structures, and algorithms, vol 28. Springer

Ohtsuki T, Mori H, Kuh E, Kashiwabara T, Fujisawa T (1979) One-dimensional logic gate assignment and interval graphs. IEEE Trans Circuits Syst 26:675–684

Robertson N, Seymour PD (1983) Graph minors. i. excluding a forest. J Comb Theory Ser B 35:39–61

Robertson N, Seymour PD (1984) Graph minors. iii. planar tree-width. J Comb Theory Ser B 36:49–64

Stepanova D, Yousif N, Karlibaeva R, Mikhaylov A (2024) Current analysis of cryptocurrency mining industry. J Infrastruct Policy Dev 8:4803

Taylor MB (2017) The evolution of Bitcoin hardware. Computer 50:58–66

The Sage Developers (2020) SageMath, the Sage Mathematics Software System

Thorup M (1998) All structured programs have small tree width and good register allocation. Inf Comput 142:159–181

van Dijk T, van den Heuvel J-P, Slob W (2006) Computing treewidth with LibTW, tech. report, University of Utrecht

Velner Y, Teutsch J, Luu L (2017) Smart contracts make Bitcoin mining pools vulnerable. In: FC, pp 298–316

Wang C, Chu X, Qin Y (2020) Measurement and analysis of the Bitcoin networks: A view from mining pools. In: BIGCOM, pp 180–188

Zhang F, Eyal I, Escriva R, Juels A, van Renesse R (2017) REM: resource-efficient mining for blockchains. In: USENIX Security, pp 1427–1444

Zur RB, Eyal I, Tamar A (2020) Efficient MDP analysis for selfish-mining in blockchains. In: AFT, pp 113–131