Practical Minimum Path Cover

Manuel Cáceres
Department of Computer Science, University of Helsinki, Finland
Department of Computer Science, Aalto University, Finland

Brendan Mumey
School of Computer Science, Montana State University, Bozeman, MT, USA

Santeri Toivonen
Department of Computer Science, University of Helsinki, Finland

Alexandru I. Tomescu
Department of Computer Science, University of Helsinki, Finland

Abstract

Computing a minimum path cover (MPC) of a directed acyclic graph (DAG) is a fundamental problem with a myriad of applications, including reachability. Although it is known how to solve the problem by a simple reduction to minimum flow, recent theoretical advances exploit this idea to obtain algorithms parameterized by the number of paths of an MPC, known as the *width*. These results obtain fast [Mäkinen et al., TALG 2019] and even linear time [Cáceres et al., SODA 2022] algorithms in the small-width regime.

In this paper, we present the first publicly available high-performance implementation of state-of-the-art MPC algorithms, including the parameterized approaches. Our experiments on random DAGs show that parameterized algorithms are orders-of-magnitude faster on dense graphs. Additionally, we present new fast pre-processing heuristics based on transitive edge sparsification. We show that our heuristics improve MPC-solvers by orders of magnitude.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Network flows

Keywords and phrases minimum path cover, directed acyclic graph, maximum flow, parameterized algorithms, edge sparsification, algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2024.3

Related Version *Previous Version*: `https://arxiv.org/abs/2308.08960`

Supplementary Material *Software*: `https://github.com/algbio/PerformanceMPC`

Funding This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO), from the Research Council of Finland grants No. 322595, 352821, 346968, 358744, and has been supported by Helsinki Institute for Information Technology HIIT, and US National Science Foundation grants DBI-1759522 and DBI-2309902.

1 Introduction

1.1 Motivation

A minimum path cover $\mathcal{P}$ (MPC) of a directed acyclic graph (DAG) $G = (V, E)$ is a minimum-sized set of paths covering V, that is, every vertex of V is present in at least one path of $\mathcal{P}$. Dilworth [19] proved that the number of paths in such a set, namely the width k, equals the maximum number of pairwise non-reachable[1] vertices. See Figure 1 for an illustration

[1] A vertex u reaches a vertex v if there is a path from u to v.

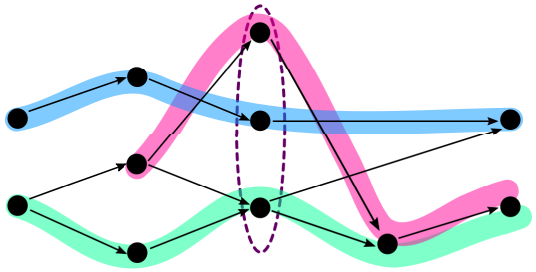

Figure 1 A DAG, an MPC shown as highlighted paths, and a maximum-sized set of non-reachable vertices in a dashed oval. The width of this graph is $k = 3$.

of these concepts. Later, Fulkerson [25] showed that the problem of finding an MPC is polynomially solvable with a reduction to maximum matching in a bipartite graph encoding the reachability relation between the vertices.

Computing an MPC has applications in many areas of computer science such as bioinformatics [39, 15, 22, 47, 10], scheduling [14, 17, 5, 49], computational logic [4, 26], distributed computing [46, 28], evolutionary computation [30], programming languages [34], databases [29], cryptography [38], and program testing [40]. Moreover, MPCs also encode the reachability between the vertices, as formally shown by the constant-time reachability index of Jagadish [29] as well as by the transitive closure algorithm of Simon [44]. As such, MPCs are fundamental objects in the problem of reachability and the applications therein.

The results of Dilworth and Fulkerson were developed in the context of partially ordered sets (posets) where the input object corresponds to a transitive DAG. The problem was later defined on general DAGs (as in this manuscript) and solved by a simple and elegant reduction to minimum flow [40], the *folklore reduction*. In this reduction, a minimum flow is computed on a different graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which is then decomposed to obtain the corresponding MPC. Hence, an MPC can be found in time $O(T_{MF}(\mathcal{G}) + ||\mathcal{P}||)$, where T_{MF} is the time to compute a maximum flow[2] and $||\mathcal{P}||$ is the total length of the paths in the computed MPC[3].

On the one hand, by using the recent breakthrough result on flows [11, 48], we can compute an MPC in almost-optimal $O(|E|^{1+o(1)} + ||\mathcal{P}||)$-time. Although this is an impressive theoretical discovery, state-of-the-art flow algorithms rely on complex convex optimization techniques, and are far from being competitive in practice against current high-performance flow solvers (see e.g. [3] for progress in this line of research).

On the other hand, recent efforts further study the minimum flow reduction and develop algorithms parameterized by the width k, obtaining running times of $O(k(|V| + |E|) \log |V|)$ [23, 34, 39] and the first parameterized linear time algorithm running in time $O(k^3|V| + |E|)$ [8] and later improved to $O(k^2|V| + |E|)$ [6]. Although these approaches are beaten in the large-width regime, they have practical potential as 1) they are simple combinatorial approaches, which also facilitates their implementation, 2) the expected width of random DAGs is known to be upper-bounded by $O(\frac{\log(\rho \cdot |V|)}{\rho})$ [2, 44], where $\rho = \frac{|E|}{\binom{|V|}{2}}$ is the density of the DAG, which was recently confirmed experimentally [35] and 3) the width in several applications has been observed to be even smaller [37, 9, 43].

[2] In this reduction, the minimum flow problem can be reduced to maximum flow as we will explain later.
[3] The MPC can be decomposed from the flow in time $O(||\mathcal{P}|| + |E|)$ as we will explain later. Also note that $||\mathcal{P}|| = O(k|V|)$ is a simple bound.

1.2 Contributions

We present the first open source implementations of different state-of-the-art MPC-solvers:

- The folklore reduction, which is compatible with all maximum flow and minimum cost flow solvers from the LEMON library [18] as well as our own implementations of classical maximum flow algorithms.
- The $O(k(|V| + |E|) \log |V|)$-time algorithm [23, 34, 39], which is also compatible with all the flow solvers from the previous point.
- The parameterized linear time algorithms $O(k^3|V| + |E|)$ [8], $O(k^2|V| + |E|)$ [7].

Our experiments on random DAGs show that the parameterized approaches are orders-of-magnitude faster than the folklore reduction on the fastest flow-solvers. In fact, our implementations of the parameterized approaches are able to compute MPCs on graphs with more than 10^8 edges in less than 2 minutes. In particular, the parameterized linear time algorithms shine on dense and small-width instances and outperform all its competitors, running in less than 5 seconds. We also present new fast pre-processing heuristics based on the concept of transitive sparsification [29, 44]. By removing transitive edges, our heuristics reduce the running time of solvers by up to an order-of-magnitude.

The rest of the paper is organized as follows. Section 2 explains the algorithms in our implementations. Section 3 shows our proposed pre-processing heuristics based on transitive sparsification. Section 4 presents our experimental setup and results.

2 Flow-based MPC algorithms

All state-of-the-art MPC-algorithms are based on a simple and elegant reduction to minimum flow. Analogous to the maximum flow problem, in minimum flow [13] we are given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a source $s \in \mathcal{V}$ and a sink $t \in \mathcal{V}$, and *demands* on the edges $d : \mathcal{E} \to \mathbb{N}_0$. The goal is to compute an st-flow (or just flow) $f^* : \mathcal{E} \to \mathbb{N}_0$ of minimum size $|f^*|$ (net flow exiting s), which satisfies *flow conservation* (the flow entering and exiting a non-source nor sink vertex is the same) and respects the demands ($f^*(e) \geq d(e)$ for all edges). For a more formal definition of these concepts we refer to [1].

2.1 The Folklore reduction

The reduction from MPC to minimum flow has been discovered and re-discovered many times in the literature (see e.g. [8]), but it can be attributed to its first public appearance in the paper of Ntafos and Hakimi [40]. Given a DAG $G = (V, E)$, we build its *flow reduction* as the pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $d : \mathcal{E} \to \mathbb{N}_0$, where $\mathcal{V}$ contains two copies v^{in}, v^{out} of each vertex $v \in V$ connected by an edge with demand $d(v^{in}, v^{out}) = 1$, every other edge in $\mathcal{E}$ has demand 0. The set $\mathcal{V}$ also contains a global source s connected to every v^{in} and a global sink t connected from every v^{out}. Finally, $\mathcal{E}$ replicates E by having an edge (u^{out}, v^{in}) for every edge $(u, v) \in E$. Note that $|\mathcal{V}| = O(|V|), |\mathcal{E}| = O(|V| + |E|)$. A flow f corresponds to a path cover $\mathcal{P}$ of G with $|f|$ paths. See Figure 2 for an illustration of these concepts.

Each of these paths can be obtained by decomposing one unit of flow at a time from f. The decomposition can be naively performed with $|f|$ graph searches in $O(|f|(|V| + |E|))$ time. We implement a faster algorithm running in time $O(||\mathcal{P}|| + |E|)$ (see Section 2.4).

As every path cover can be interpreted as a flow, an MPC corresponds to a minimum flow f^* in this network. Minimum flow on the flow reduction can be reduced to maximum flow by also providing an initial flow f, that is, a path cover. For every edge $e \in \mathcal{E}$, we place it in the maximum flow instance only if $f(e) > d(e)$, in which case we define its capacity to

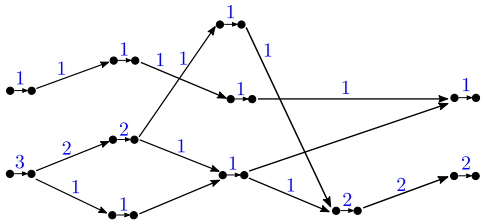

Figure 2 A feasible flow of size 4 in the flow reduction of the graph shown in Figure 1. Flow values are shown on top of the corresponding edges (0 if not present). Vertices s and t are not shown for simplicity. A decomposition of this flow produces a path cover with 4 paths, i.e. not minimum.

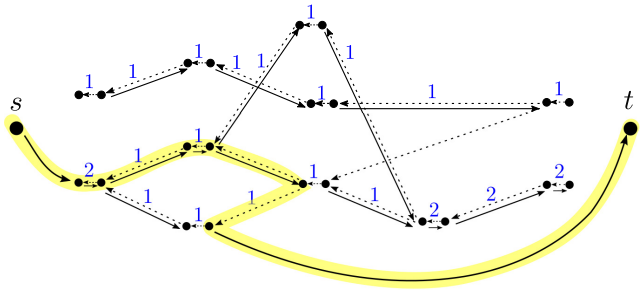

Figure 3 Residual graph of the flow in Figure 2. Direct edges are shown as solid arrows, while reverse edges as dashed arrows. A residual path is highlighted. From s and to t only the path edges are shown. Flow values on the edges are the result of using the residual path.

be $c(e) = f(e) - d(e)$. Moreover, for every edge $(u, v) \in \mathcal{E}$, we place its reverse edge (v, u) in the maximum flow instance with capacity $c(u, v) = |f|$. It can be shown [39] that if f' is a maximum flow of this instance, then $f^* = f' - f$ is a minimum flow of $\mathcal{G}, d$.[4]

A more direct interpretation of the minimum flow problem [8] defines the residual graph $\mathcal{R}(\mathcal{G}, f)$, by placing every *reverse edge* (used to increase the flow in the opposite direction) and placing *direct edges* whenever $f(e) > d(e)$ (used to decrease the flow). An st-path in $\mathcal{R}(\mathcal{G}, f)$ (*residual path*) can then be used to decrease the flow size by one unit. See Figure 3.

In both interpretations of the problem, a minimum flow of the reduction can be obtained in time $O(|f|(|V|+|E|))$ by a simple Ford-Fulkerson approach [24], which finds $O(|f|)$ residual paths. Since there is always a path cover that uses $|V|$ paths to cover every vertex (one path per vertex, the *naive* solution), the previous approach runs in $O(|V|(|V| + |E|))$ time.

2.2 Greedy solution

Felsner et al. [23] proposed a greedy heuristic to compute a chain decomposition of a poset. They iteratively extract the longest chain of elements from the remaining poset. They proved, with analogous arguments to those of the greedy set cover logarithmic approximation [12], that the number of chains extracted is bounded by $O(k \log |V|)$. Later, Kowaluk et al. [34] showed that the same principle can be applied to general DAGs by finding the path covering the most uncovered vertices. They showed that these paths can be found in a DAG by a reduction to shortest path, which was later simplified by Mäkinen et al. [39] with a dynamic programming solution. As such, computing the greedy solution and using it in the flow reduction derives a $O(k(|V| + |E|) \log |V|)$-time algorithm for MPC.

[4] Flow in reverse edges is interpreted as negative flow in the opposite direction.

2.3 Parameterized linear time algorithms

Recently, Cáceres et al. [8] proposed a slightly different method to compute an MPC using the flow reduction. In their method, the vertices are processed one by one in topological order [31, 45][5] and an MPC of the (graph induced by the) already processed vertices is computed at each step. More specifically, if $\mathcal{P}$ is an MPC of the first vertices in topological order and v is the next vertex to process, the solution $\mathcal{T} = \mathcal{P} \cup \{(v)\}$ is used as initial solution of the flow reduction. A nice property of this incremental framework is that the size of an MPC of the current iteration is either $|\mathcal{T}| = |\mathcal{P}| + 1$ or $|\mathcal{P}|$, and thus only one traversal of the residual suffices to compute it. Note that this a simplification of the framework. In practice, the algorithms maintain a flow f^* on the flow reduction of the first vertices (representing the MPC $\mathcal{P}$). When the next vertex v is processed, the flow reduction is updated by adding vertices v^{in} and v^{out}, and their corresponding edges (see Section 2.1). The flow f^* is also updated to $f^*(s, v^{in}) = f^*(v^{in}, v^{out}) = f^*(v^{out}, t) = 1$ (representing $\mathcal{T}$), which is then used to look for a residual path, possibly modifying f^* again.

By considering simple graph traversals of the residual, this approach runs in time $O(|V|(|V| + |E|))$. However, Cáceres et al. [8] combine *transitive sparsification* of edges with a special *layered traversal* of the residual to obtain a linear dependency in the number of edges and a factor k^3 dependency in the number of vertices for a total running time of $O(k^3|V| + |E|)$.

Layered traversal. The algorithm assigns a level $\ell : \mathcal{V} \to \{0, \ldots, |\mathcal{P}|\}$ to every vertex in the flow reduction. The level assignment maintains the property that paths in the residual graph are sequences of vertices with non-increasing levels, which allows to perform the traversal for the search of a residual path in a layered[6] manner. The residual graph is traversed from the highest reachable layer until the lowest reachable layer (or until a residual path is found). To perform this layered traversal, the algorithm uses $|\mathcal{P}| + 1$ FIFO queues (one per layer), each of which performs a BFS from the highest-reachable layer down to the lowest-reachable layer. Algorithm 4 in the Appendix shows the corresponding pseudocode of the traversal.

After the layered traversal finishes, the algorithm updates the flow and level assignment to maintain the algorithm's invariants [8]. The flow is only updated if the traversal finds a residual path, in such a case, the flow in direct edges of the path is decreased by one and the flow in (the reverse of) reverse edges is increased by one, which decreases the total flow by one. As for the level assignment, if the lowest visited level is l, then all visited vertices change their level to l while the level of v^{out} is set to $l + 1$. Additionally, if there is no flow from layer l exiting directly to t, then the algorithm performs a *merge* of layer l. For an explanation of the merge procedure, we refer to the original publication [8] and our code.

Transitive sparsification. An edge (u, v) is *transitive* if there exists another path from u to v. Transitive edges can be safely removed from the DAG when computing an MPC, as removing these edges preserves the reachability relation between the vertices and hence, by Dilworth's theorem, the width. A *transitive sparsification* is both a spanning subgraph with the same reachability relation as G as well as the process to obtain such a subgraph. The $O(k^3|V| + |E|)$-time algorithm sparsifies the number of incoming edges to each vertex to $O(k)$. The authors use a simple idea first proposed by Jagadish [29] and Simon [44] on

[5] As topological sorting algorithms run in linear time, we assume that such order is given as input. In our implementation we use the faster DFS-based algorithm of Tarjan [45].

[6] A *layer* is a maximal set of vertices with the same level.

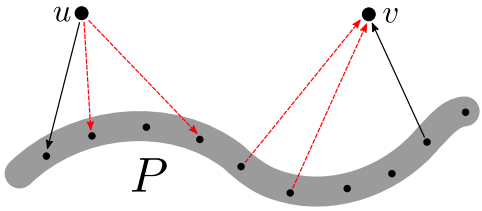

Figure 4 A path P is highlighted (path edges were removed by simplicity). The figure shows (dashed arrows) the transitive edges incoming to a vertex v as well as outgoing from a vertex u.

posets[7]: if several incoming edges to v come from the same path P, then all these edges, except maybe the last, are transitive. Conversely, if several outgoing edges from u go to the same path P, then all these edges, except maybe the first, are transitive. See Figure 4. The algorithm of Cáceres et al. [8] uses this idea and the MPC $\mathcal{P}$ from the previous iteration to sparsify the edges to the current vertex v to at most $|\mathcal{P}| \leq k$. To perform this sparsification efficiently, the algorithm requires that every vertex stores the id of one path that contains such vertex, which is achieved by maintaining a path decomposition of the minimum flow f^*, that is $\mathcal{P}$.

The $O(k^2|V| + |E|)$-time algorithm. The same authors later improved the running time of their algorithm [7] by shaving a k-factor from the dependency on the number of vertices. They noted that is not necessary to maintain the ids of all paths containing a vertex during the algorithm, but that it suffices to maintain only one of those ids, and the MPC can be retrieved at the end by performing only one decomposition. To achieve this, they identified a set of *antichain vertices* separating consecutive layers: as these vertices form an antichain, each of those must be covered by a different path, and in fact the algorithm covers each of these vertices with exactly one path (we refer to the original publication [7] for details). As such, it suffices that every vertex points back to (one of) the corresponding antichain vertex on its layer, these pointers are called *back links* (*bl* in the pseudocode for short). As opposed to an entire decomposition, back links can be maintained by only decomposing the vertices in layer l (lowest visited level), and then fixing the back links of vertices in lower layers (in constant time per vertex of level $> l$). Algorithm 1 implements the ideas in [8, 6].

2.4 The decomposition algorithm

As mentioned earlier, the last step of all MPC-solvers, as well as intermediate steps of the parameterized linear time algorithms, require to decompose the flow f^* into an MPC. A naive solution extracts one path at a time in total $O(|f^*|(|V| + |E|))$ running time. We instead implement an algorithm that runs in time $O(||\mathcal{P}|| + |E|)$[8]. Such an improvement was first described by Kogan and Parter [32]. Here we use the version of Cáceres [6]. The algorithm first removes all 0-flow edges in time $O(|E|)$ and then processes the vertices in topological order. When processing vertex v, it iterates through each in-neighbor u and places v after u in $f^*(u, v)$ different paths[9]. The total running time is $\sum_{v \in V,(u,v) \in E} f^*(u, v) = O(||\mathcal{P}||)$.

[7] While Jagadish presented the idea on transitive DAGs, Simon worked on general DAGs but using a path decomposition/partition instead of a path cover.
[8] Note that $||\mathcal{P}|| = O(|f^*| \cdot |V|)$, and thus this approach removes a factor $|f^*|$ from $|E|$.
[9] This can be done in constant time per path by iterating through the list of paths of u.

Algorithm 1 One iteration of the $O(k^2|V| + |E|)$-time algorithm. The pseudocode shows the steps of the algorithm when processing vertex v. Function id returns the path id stored in an antichain vertex. Function bl returns the corresponding antichain vertex (back link). Notation ${}_uP_w$ indicates the decomposed path connecting antichain vertices u and w. For more details we refer to the original publication [7] and our code.

```
   // Add v to flow reduction
 1 V ← V ∪ {v^in, v^out}
 2 E ← E ∪ {(s, v^in), (v^in, v^out), (v^out, t)}
   // Create initial solution T
 3 f*(s, v^in), f*(v^in, v^out), f*(v^out, t) ← 1
   // Sparsify edges incoming to v
 4 sur ← (⊥)^|f*|
 5 for u ∈ N⁻(v) in G do
 6 |   sur[id(bl(u))] ← max(sur[id(bl(u))], u)
 7 for u ∈ sur, u ≠ ⊥ do
   |   // Add (u, v) to flow reduction
 8 |   E ← E ∪ {(u^out, v^in)}, f*(u^out, v^in) ← 0
   // Layered traversal with Algorithm 4
 9 D, S ← layeredTraversal(G, f*, ℓ, v)
10 l ← min{ℓ(u) | u ∈ S}
11 if D ≠ ∅ then     // Update flow values
12
13 |   for (u, w) ∈ D do
14 |   |   if (u, w) ∈ E then
   |   |     f*(u, w) ← f*(u, w) − 1
15 |   |   else f*(w, u) ← f*(w, u) + 1
16 else // Update path and backlink info
17
18 |   id(v) ← |f*|, bl(v) ← v
   // Update levels
19 ℓ(v^out) ← l + 1
20 ℓ(u) ← l, for u ∈ S
   // Decompose vertices in layer l
21 for Decomposed path ₙP_w do
   |   // Update links and path ids
22 |   id(w) ← id(u)
23 |   for x ∈ ₙP_w do
24 |   |   if x ≠ u then bl(x) ← u
25 |   |   nl(x) ← w
   // Fix backlinks in higher layers
26 for u ∈ V, ℓ(u^in) > l ∨ ℓ(u^out) > l in top.
    order do
27 |   if ℓ(bl(u)^out) ≠ ℓ(u^in) then
   |     bl(u) ← nl(bl(u))
28 |   if ℓ(u^in) ≠ ℓ(u^out) then
   |     id(u) ← id(bl(u))
29 if No flow from layer l to t then
   |   // Merge of layer l
```

3 Fast pre-processing sparsification heuristics

In this section, we present two transitive sparsification heuristics. Recall that a transitive sparsification removes transitive edges, making the input graph sparser. These heuristics are intended to be used as pre-processing steps of MPC-solvers to speed up their computation. As such, it is very important that they run fast compared to the MPC-solver. In fact, we ensure that their worst-case running time is upper-bounded by the running time of state-of-the-art solvers. Both of our heuristics use paths to sparsify the incoming/outgoing edges to/from a vertex as done by the parameterized linear time algorithms.

DFS sparsification. Our first sparsification heuristic uses the root-to-leaf paths of a DFS-spanning tree. A first naive implementation of this idea processes each of these paths to sparsify the incoming edges. However, this approach runs in time proportional to the total length of the root-to-leaf paths, which can be $\Omega(|V|^2)$. Instead, our simple solution runs in time $O(|V| + |E|)$ as it is implemented on top of a normal recursive DFS traversal. Algorithm 2 shows the corresponding pseudocode.

The main idea behind this algorithm is to use the DFS recursion itself as DFS paths. For this, it stores the preorder of each vertex visited (in `dfs_pre`) as well as the maximum preorder value observed of an in-neighbor (in `last_reach`). When processing an edge (v, w)

Algorithm 2 DFS sparsification heuristic.

```
1  E' ← ∅
2  S ← ∅
3  dfs_pre[v] ← 0 for v ∈ V
4  next_pre ← 1
5  last_reach[v] ← 0 for v ∈ V
6  for v ∈ V in top. order do
7  |  if v ∉ S then dfsSp(v)
8  return G' = (V, E')
9  Function dfsSp(v):
10 |  S ← S ∪ {v}
11 |  dfs_pre[v] ← next_pre
12 |  next_pre ← next_pre + 1
13 |  for w ∈ N+(v) in top. order do
14 |  |  if w ∉ S then
15 |  |  |  dfsSp(w)
16 |  |  if last_reach[w] < dfs_pre[v]
   |  |   then
17 |  |  |  E' ← E' ∪ {(v, w)}
18 |  |  |  last_reach[w] ← dfs_pre[v]
```

Algorithm 3 Greedy sparsification heuristic.

```
1  E' ← E
2  P ← ∅
3  while P is not a path cover do
4  |  P* ← path of (V, E') with most
   |   uncovered
5  |  P ← P ∪ {P*}
6  |  R ← ∅
7  |  for v ∈ P* do
8  |  |  for u ∈ N−(v) in (V, E') do
9  |  |  |  if u ∈ R ∧ (u, v) ∉ P then
10 |  |  |  |  E' ← E' \ {(u, v)}
11 |  |  |  R ← R ∪ {u}
12 return G' = (V, E'), P
```

(after traversing w), if the observed preorder value of an in-neighbor of w is bigger than the preorder of v ($\texttt{last\_reach}[w] > \texttt{dfs\_pre}[v]$), then the edge is transitive and it is not added to the sparsification, as there is a vertex further down the DFS-tree also with an edge to w (the one with preorder value $\texttt{last\_reach}[w]$). Conversely, among all vertices in a DFS root-to-leaf path with an edge to w, the only edge that is not sparsified is the one with the largest preorder value, that is, the one closest to the leaf.

Greedy sparsification. Our second sparsification heuristic also outputs the greedy initial solution explained in Section 2.2. It uses the $O(k \log |V|)$ paths from the greedy solution to sparsify outgoing edges. As such, this heuristic sparsifies the edges to $|E'| = O(k|V| \log |V|)$. Since this algorithm computes the greedy solution, its worst-case running time is also $O(k(|V| + |E|))$. However, we implemented a non-trivial practical improvement where each extracted path is immediately used to sparsify, and thus the following paths are extracted from a sparser graph. Algorithm 3 shows the corresponding pseudocode. The main novelty of this approach is that greedy paths are computed at the same time that the DAG is being sparsified, resulting in a faster initial solution computation. Note that the algorithm does not sparsify an edge if this is present in the greedy path cover, however, there are at most $||\mathcal{P}|| = O(k|V| \log |V|)$ such edges.

4 Experiments and Results

4.1 Implementations

Our code was written in C++ and it can be found at `https://github.com/algbio/PerformanceMPC` under the GNU General Public License v3.0. The code is compatible with all maximum flow and minimum cost flow solvers from the LEMON library [18], which

are known to be the fastest publicly available flow solvers. For an input, we provide the best times among the solvers in the library as `lemon`. For a cleaner and fairer comparison, we re-implemented the following well-known maximum flow solvers:

- `DFS`: Ford-Fulkerson [24] approach that finds residual paths using depth-first search.
- `BFS`: Edmonds-Karp algorithm [21], which finds residual paths using breath-first search.
- `Blocking`: Dinitz' algorithm [20], which uses *blocking flows*.

All solvers can start from one of the following initial solutions (path covers):

- `naive`: $|V|$ paths, each covering a different vertex.
- `greedy`: $O(k \log |V|)$ paths based on greedy set cover [23, 34, 39].

After running the flow solver all our implementations run the same $O(||\mathcal{P}|| + |E|)$-time decomposition routine to obtain the corresponding MPC $\mathcal{P}$ (see Section 2.4).

Our code also implements the parameterized linear time algorithms:

- `k3`: The first parameterized linear time algorithm running in time $O(k^3|V| + |E|)$ [8].
- `k2`: A later improvement over `k3` running in time $O(k^2|V| + |E|)$ [7].

All these implementations constitute the state-of-the-art algorithms for MPC. To the best of our knowledge there are no other publicly available fast MPC-solver's implementations. Most publicly available MPC-solvers use the slower reduction to bipartite maximum matching, and thus also need to compute the transitive closure[10]. Mäkinen et al. [39] were the first to implement the greedy-based approach, which was later improved by Ma et al. [37] using Dinitz' algorithm for finding residual paths: these implementations correspond to our `DFS greedy` and `Blocking greedy`, respectively. Finally, for all our MPC-solvers we also implemented our two sparsification heuristics from Section 3: `dfs-sp` and `greedy-sp`.

4.2 Setup

The experiments ran on an isolated Intel(R) Xeon(R) CPU E5-2670 @ 2.6 GHz with 64GB of RAM, running Almalinux 8.4 (64bit, kernel 4.18.0). The code was compiled using `gcc` version 8.5.0 with optimization flag `-O3`. We measure `user` time using the `sys/resource.h` Unix library. We report the average value of 10 repetitions of each experiment. We used a timeout of 10 minutes for each experiment.

4.3 Datasets

Random DAG. For a fixed value of N and M, we generate a random DAG with N vertices and M edges. The generation procedure first fixes a topological order of the N vertices. Then, it generates M different pairs of vertices and interprets them as edges directed according to the topological order. We vary $N \in 10,000 \times \{1, 2, \ldots, 5\}$. For space constraints reasons, we show the results for $N = 50,000$ and vary $M \in \{32,768 = 2^{15}, 2^{16}, \ldots, 2^{27} = 134,217,128\}$ to observe the behavior at different densities (other values of N can be found in the Appendix). This dataset corresponds to the random DAG model proposed by Barak and Erdös [2]. We use this dataset to compare general performance. Table 1 in the Appendix shows the width of DAGs in this dataset. Note that the width decreases with the number of edges of the `Random DAG`. Indeed, the expected width of a `Random DAG` of parameters N and M, is upper bounded by $O(\frac{\log(\rho \cdot N)}{\rho})$ [44], where $\rho = \frac{M}{\binom{N}{2}}$ is the density of the DAG. Moreover, Kritikakis and Tollis [35] recently showed that experimentally the width is proportional to $1/\rho$.

[10] The densest spanning supergraph having the original graph as a transitive sparsification.

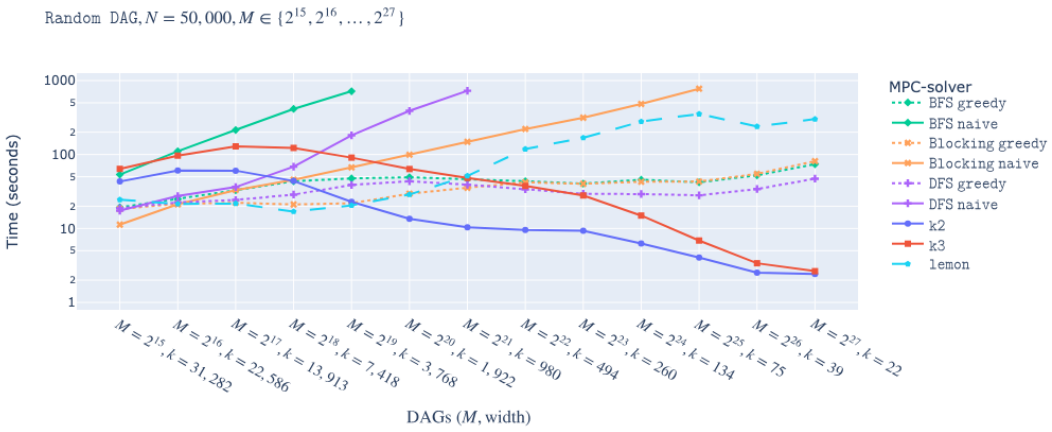

Figure 5 Running time of MPC-solvers in `Random DAG`. Note the log-scale in the y-axis.

Path Partition. For a fixed value of N, M and K, we generate the previously described `Random DAG` with N vertices and M edges. Then, we divide the N vertices into K parts by placing each vertex on a uniformly random chosen part. Finally, we add the corresponding $N - K$ edges (in topological order) so that each part is a path in the DAG. As such, the graph's width is at most K. We fix $N = 50,000$ and vary $M \in \{32,768 = 2^{15}, 2^{16}, \ldots, 2^{27} = 134,217,128\}$ as before and , $K = \{21 = \lfloor 2^1 \cdot \ln N \rfloor, \ldots, \lfloor 2^4 \cdot \ln N \rfloor = 173\}$, we only show the results for $K = 173$ due to space constraints (other values of K can be found in the Appendix). We use this dataset to study the performance on small-width instances. Table 2 in the Appendix shows the number of edges and width of DAGs in this dataset. We note that our `Path Partition` dataset is equivalent to the "Path-Based Model" of Lionakis et al. [36] and a generalization of the "Random arcs k-path Model" of Paavilainen [42].

Transitive Closure. For a fixed value of N and M, we generate a `Random DAG` with N vertices and M edges. Then, we compute its transitive closure. We fix $N = 10,000$ and vary $M \in \{8,192 = 2^{13}, \ldots, 2^{23} = 8,388,608\}$. We use this dataset to study the performance on posets and the behavior of transitive sparsification heuristics. Table 2 in the Appendix shows the number of edges and width of DAGs in this dataset. Note that the width distribution of `Random DAG` is not affected as adding transitive edges does not change the width.

4.4 Results

Figure 5 shows the running time of the MPC-solvers on the `Random DAG` dataset (for other values of N see Figure 9 in the Appendix). Solvers starting from a `naive` solution are depicted with a solid line joining the corresponding data points[11]. The maximum flow-based solvers show a polynomial dependency in the number of edges of the input graph, with `Blocking naive` being the fastest (as predicted by theory as it uses a faster flow algorithm) followed by `DFS naive` and then by `BFS naive`. These results suggest that 1) the more complex `Blocking` algorithm pays off, as each step significantly reduces the path cover size, and that 2) although `BFS` ensures a polynomial running time for maximum flow, in the case

[11] We consider `k3` and `k2` "to start from a naive solution" since, at each step, these consider the next vertex as a single path.

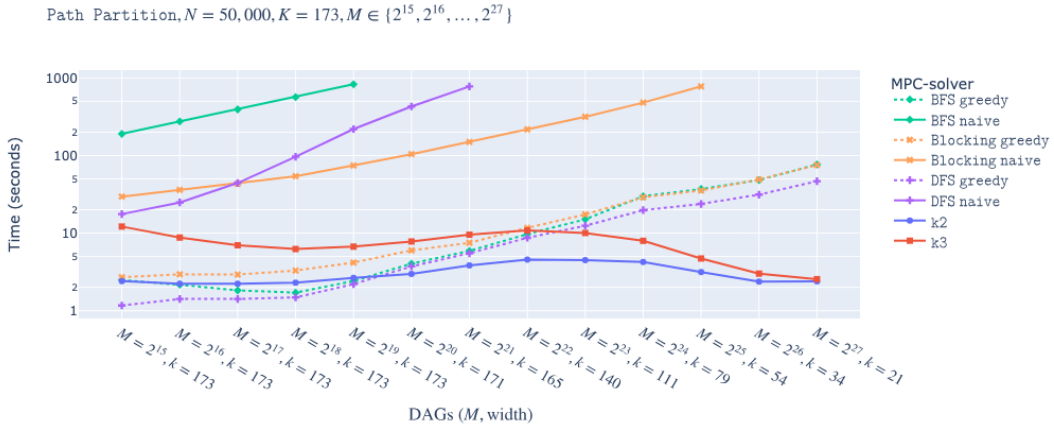

Figure 6 Running time of MPC-solvers in `Path Partition`. Note the log-scale in the y-axis.

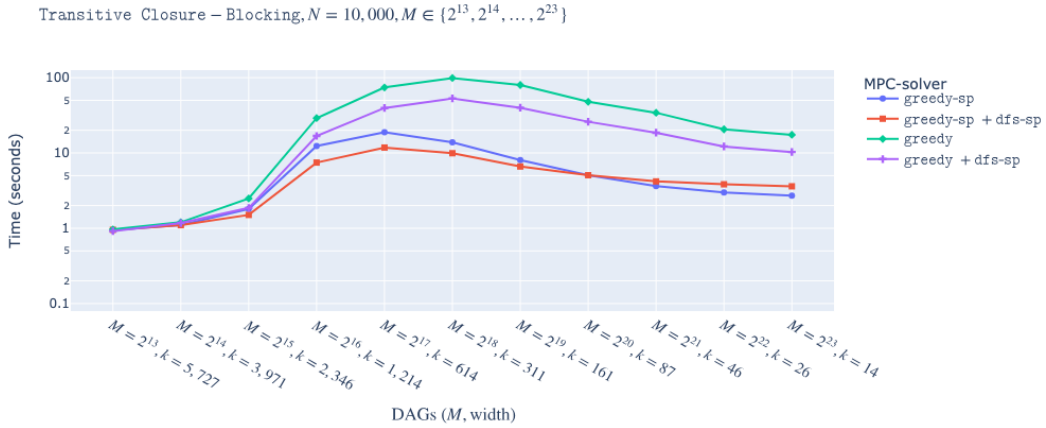

Figure 7 Running time of solver `Blocking greedy` in dataset `Transitive Closure` with different combinations of pre-processings `dfs-sp` and `greedy-sp`. Note the log-scale in the y-axis.

of MPC this is unnecessary as $k \leq |V|$ and `DFS` performs better in practice as residual paths are quickly found. Moreover, these solvers run out of time (> 10 mins.) after $|E| = 2^{24}, 2^{20}$ and 2^{18}, respectively. Solvers starting from a `greedy` solution are shown with a dotted line. The solvers show a much faster running time, which stands below the 2 mins. irrespective of the number of edges. As such, on dense graphs, these approaches are orders-of-magnitude faster that their `naive` counterparts. In this case, the difference between the different solvers is subtle, as substantially less residual paths must be found to transform the $O(\log |V|)$-approximation to an MPC, and indeed `DFS` beats the machinery of `Blocking` from $|E| \geq 2^{22}$ and $k \leq 494$. The running time of `k3` decreases with the number of edges. This is explained by the linear dependency on the number of edges: these algorithms process the edges, in constant-time, only during the initial edge sparsification, whereas vertices are charged with all the remaining machinery of the approach. This amounts to $O(k^3)$ and $O(k^2)$ per vertex, respectively, which is known to decrease with increasing density [44, 35]. Both solvers run on less than 2 mins. on every graph, and outperform the maximum flow-based solvers on dense graphs, from $|E| \geq 2^{20}$ in the case of `k2` and from $|E| \geq 2^{23}$ in the case of `k3`, being almost

two orders of magnitude faster on the densest instance ($|E| = 2^{27}$). Finally, it is worth mentioning that `k2` outperforms `k3` on every instance tested, which shows that the more complex routines of the `k2` algorithm, as implemented in this work, manage to effectively shave a factor k from the running time. In practical terms, the time saved by avoiding the full decomposition is larger that the time required to perform these savings. Finally, the best of `lemon` dominates the `naive` versions of our solvers but it is dominated in the dense regime, by both the `greedy` versions and the parameterized linear time algorithms. In the sparse regime `lemon` corresponds to the implementations `Preflow` [27] and `CapacityScaling` [21], whereas in the dense regime is dominated by `CostScaling` [41] as well as `NetworkSimplex` [16].

Figure 6 shows the running time of the solvers on the `Path Partition` dataset (for other values of K see Figure 10 in the Appendix). The behavior is very similar to the `Random DAG` dataset. In this case, `DFS` beats `Blocking` in sparse and very sparse graphs in `greedy` and `naive`, respectively. For all graphs in the dataset, the solvers `k3` and `k2` run in no more than 15 secs. and 5 secs., respectively.

Pre-processing. We exclude `k3` and `k2` from this comparison as using a pre-processing edge sparsification is counterproductive (perform internal sparsification). For space constraints we only show the effect of pre-processing on one MPC-solver.

Figure 7 shows the running time of `Blocking greedy` on the `Transitive closure` dataset and different combinations of pre-processings `dfs-sp` and `greedy-sp` (for other MPC-solvers see Figure 8 in the Appendix). We note that for $M \geq 2^{19}$ the number of edges in the corresponding graphs is larger than in the densest instance of the previous datasets, as such we call these graphs *dense*. On dense graphs, `dfs-sp` roughly decreases the running time in half, while `greedy-sp` reduces the running time by one order-of-magnitude. When using both heuristics `greedy-sp` and `dfs-sp`, we perceive a combined positive effect until $M \leq 2^{19}$. For denser graphs, performing both sparsifications does not pay off as `greedy-sp` is able to sparsify more edges (recall that `greedy-sp` sparsifies the edges to $|E'| = O(k|V|\log|V|)$), but it does not affect the running time significantly either. On non-dense graphs ($M < 2^{19}$), applying both sparsifications dominates and it is up to 4 times faster than plain `greedy`.

Even though our sparsification heuristics show a clear improvement on dense graphs, these improved versions are still outperformed by the parameterized approaches.

5 Conclusions and Future Work

We presented the first high-performance implementation of state-of-the-art MPC algorithms and showed that approaches parameterized by the width dominate the practical performance landscape on different kinds of random graphs. In particular, the parameterized linear time algorithms [8, 7] shine on small-width instances, being orders-of-magnitude faster. Recent works [33, 6] circumvent the $\Omega(k|V|)$ lower bound[12] by computing a minimum chain cover instead. In practice, it is interesting to test if these algorithms are effectively faster than our MPC-solvers or if these ideas can be used to improve the performance of our implementations. We also presented two new pre-processing fast heuristics based on transitive sparsification and showed how they improve the running time by an order-of-magnitude.

An important application of MPC is reachability. In particular, it is known (see e.g. [35]) how to compute a constant-time reachability index of size $O(k|V|)$ in time $O(k|E'|)$, where $|E'|$ is the number of edges in the sparsest transitive sparsification, also known as transitive

[12] There are instances with $||\mathcal{P}|| = \Omega(k|V|)$ [6].

reduction. This result directly derives parameterized linear time solutions for the problems of constant-time reachability, transitive closure and transitive reduction, which can be implemented and compared against state-of-the-art solutions for those problems.

Finally, one algorithm we have not implemented is the $O(k^2|V|\log|V|+|E|)$-time approach of Cáceres et al. [8, Theorem 1.1] as this was later outperformed by the $O(k^2|V|+|E|)$-time algorithm [7]. However, this divide-and-conquer approach is simple to parallelize [8, Theorem 1.2] and thus it could outperform our implementations when run on multiple processors.

References

1 Ravindra K Ahujia, Thomas L Magnanti, and James B Orlin. Network flows: Theory, algorithms and applications. *New Jersey: Prentice-Hall*, 1993.

2 Amnon B Barak and Paul Erdös. On the maximal number of strongly independent vertices in a random acyclic directed graph. *SIAM Journal on Algebraic Discrete Methods*, 5(4):508–514, 1984.

3 Ruben Becker, Maximilian Fickert, and Andreas Karrenbauer. A novel dual ascent algorithm for solving the min-cost flow problem. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 151–159. SIAM, 2016.

4 Simone Bova, Robert Ganian, and Stefan Szeider. Model checking existential logic on partially ordered sets. *ACM Transactions on Computational Logic*, 17(2):1–35, 2015.

5 Stefan Bunte and Natalia Kliewer. An overview on vehicle scheduling models. *Public Transport*, 1(4):299–317, 2009.

6 Manuel Cáceres. Minimum chain cover in almost linear time. In *Proceeding of the 50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*, volume 261, pages 31:1–31:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

7 Manuel Caceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. Minimum path cover in parameterized linear time. *arXiv preprint arXiv:2211.09659*, 2022.

8 Manuel Cáceres, Massimo Cairo, Brendan Mumey, Romeo Rizzi, and Alexandru I Tomescu. Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time. In *Proceedings of the 33rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, pages 359–376. SIAM, 2022.

9 Ghanshyam Chandra and Chirag Jain. Sequence to graph alignment using gap-sensitive co-linear chaining. In *Proceedings of the 27th Annual International Conference on Research in Computational Molecular Biology (RECOMB 2023)*, pages 58–73. Springer, 2023.

10 Zheng Chang, Guojun Li, Juntao Liu, Yu Zhang, Cody Ashby, Deli Liu, Carole L Cramer, and Xiuzhen Huang. Bridger: a new framework for de novo transcriptome assembly using RNA-seq data. *Genome Biology*, 16(1):1–10, 2015.

11 Li Chen, Rasmus Kyng, Yang P Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *Proceedings of the 63rd IEEE Annual Symposium on Foundations of Computer Science (FOCS 2022)*, pages 612–623. IEEE, 2022.

12 Vasek Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of operations research*, 4(3):233–235, 1979.

13 Eleonor Ciurea and Laura Ciupala. Sequential and parallel algorithms for minimum flows. *Journal of Applied Mathematics and Computing*, 15(1-2):53–75, 2004.

14 Charles J Colbourn and William R Pulleyblank. Minimizing setups in ordered sets of fixed width. *Order*, 1(3):225–229, 1985.

15 Nicola Cotumaccio and Nicola Prezza. On indexing and compressing finite automata. In *Proceedings of the 32nd ACM-SIAM Symposium on Discrete Algorithms (SODA 2021)*, pages 2585–2599. SIAM, 2021.

16 George Dantzig. *Linear programming and extensions*. Princeton university press, 1963.

17 Jacques Desrosiers, Yvan Dumas, Marius M Solomon, and François Soumis. Time constrained routing and scheduling. *Handbooks in Operations Research and Management Science*, 8:35–139, 1995.

18 Balázs Dezső, Alpár Jüttner, and Péter Kovács. LEMON–an open source C++ graph template library. *Electronic Notes in Theoretical Computer Science*, 264(5):23–45, 2011.

19 Robert P Dilworth. A decomposition theorem for partially ordered sets. *Classic Papers in Combinatorics*, pages 139–144, 1987.

20 Yefim Dinitz. Dinitz' algorithm: The original version and even's version. In *Theoretical Computer Science: Essays in Memory of Shimon Even*, pages 218–240. Springer, 2006.

21 Jack Edmonds and Richard M Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

22 Nicholas Eriksson, Lior Pachter, Yumi Mitsuya, Soo-Yon Rhee, Chunlin Wang, Baback Gharizadeh, Mostafa Ronaghi, Robert W Shafer, and Niko Beerenwinkel. Viral population estimation using pyrosequencing. *PLoS Computational Biology*, 4(5):e1000074, 2008.

23 Stefan Felsner, Vijay Raghavan, and Jeremy Spinrad. Recognition algorithms for orders of small width and graphs of small Dilworth number. *Order*, 20(4):351–364, 2003.

24 Lester Randolph Ford and Delbert R Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

25 Delbert R Fulkerson. Note on Dilworth's decomposition theorem for partially ordered sets. *Proceedings of the American Mathematical Society*, 7(4):701–702, 1956.

26 Jakub Gajarskỳ, Petr Hlinenỳ, Daniel Lokshtanov, Jan Obdralek, Sebastian Ordyniak, MS Ramanujan, and Saket Saurabh. FO model checking on posets of bounded width. In *Proceedings of the 56th IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS 2015)*, pages 963–974. IEEE, 2015.

27 Andrew V Goldberg and Robert E Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

28 Selma Ikiz and Vijay K Garg. Efficient incremental optimal chain partition of distributed program traces. In *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006)*, pages 18–18. IEEE, 2006.

29 H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Transactions on Database Systems*, 15(4):558–598, 1990.

30 Wojciech Jaśkowski and Krzysztof Krawiec. Formal analysis, hardness, and algorithms for extracting internal structure of test-based problems. *Evolutionary Computation*, 19(4):639–671, 2011.

31 Arthur B Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

32 Shimon Kogan and Merav Parter. Beating matrix multiplication for $n^{1/3}$-directed shortcuts. In *49th International Colloquium on Automata, Languages, and Programming (ICALP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022. Full version available at `https://www.weizmann.ac.il/math/parter/sites/math.parter/files/uploads/main-lipics-full-version_3.pdf`.

33 Shimon Kogan and Merav Parter. Faster and unified algorithms for diameter reducing shortcuts and minimum chain covers. In *Proceedings of the 34th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2023)*, pages 212–239. SIAM, 2023.

34 Mirosław Kowaluk, Andrzej Lingas, and Johannes Nowak. A path cover technique for LCAs in dags. In *Scandinavian Workshop on Algorithm Theory*, pages 222–233. Springer, 2008.

35 Giorgos Kritikakis and Ioannis G Tollis. Fast reachability using DAG decomposition. In *Proceedings of the 21st International Symposium on Experimental Algorithms (SEA 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

36 Panagiotis Lionakis, Giacomo Ortali, and Ioannis G Tollis. Constant-time reachability in DAGs using multidimensional dominance drawings. *SN Computer Science*, 2(4):320, 2021.

37 Jun Ma, Manuel Cáceres, Leena Salmela, Veli Mäkinen, and Alexandru I Tomescu. Chaining for Accurate Alignment of Erroneous Long Reads to Acyclic Variation Graphs. *Bioinformatics*, page btad460, July 2023. doi:10.1093/bioinformatics/btad460.

38 Stephen J MacKinnon, Peter D Taylor, Henk Meijer, and Selim G. Akl. An optimal algorithm for assigning cryptographic keys to control access in a hierarchy. *IEEE Transactions on Computers*, 34(09):797–802, 1985.

39 Veli Mäkinen, Alexandru I Tomescu, Anna Kuosmanen, Topi Paavilainen, Travis Gagie, and Rayan Chikhi. Sparse dynamic programming on DAGs with small width. *ACM Transactions on Algorithms*, 15(2):1–21, 2019.

40 Simeon C Ntafos and S Louis Hakimi. On path cover problems in digraphs and applications to program testing. *IEEE Transactions on Software Engineering*, 5(5):520–529, 1979.

41 James B Orlin. A polynomial time primal network simplex algorithm for minimum cost flows. *Mathematical Programming*, 78:109–129, 1997.

42 Topi Paavilainen. Faster algorithms for minimum path cover by graph decomposition. Master's thesis, Helsingin yliopisto, 2018.

43 Jyotshna Rajput, Ghanshyam Chandra, and Chirag Jain. Co-linear chaining on pangenome graphs. In *Proceedings of the 23rd International Workshop on Algorithms in Bioinformatics (WABI 2023)*, volume 273, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

44 Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1-3):325–346, 1988.

45 Robert E Tarjan. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2):171–185, 1976.

46 Alexander I Tomlinson and Vijay K Garg. Monitoring functions on global states of distributed programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, 1997.

47 Cole Trapnell, Brian A Williams, Geo Pertea, Ali Mortazavi, Gordon Kwan, Marijke J Van Baren, Steven L Salzberg, Barbara J Wold, and Lior Pachter. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*, 28(5):511, 2010.

48 Jan Van Den Brand, Li Chen, Richard Peng, Rasmus Kyng, Yang P Liu, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *Proceedings of the 64th IEEE Annual Symposium on Foundations of Computer Science (FOCS 2023)*, pages 503–514. IEEE, 2023.

49 Xianyuan Zhan, Xinwu Qian, and Satish V Ukkusuri. A graph-based approach to measuring the efficiency of an urban taxi service system. *IEEE Transactions on Intelligent Transportation Systems*, 17(9):2479–2489, 2016.

A Extra Algorithms and Figures

Algorithm 4 Layered traversal for the search of a residual path in $\mathcal{R}(\mathcal{G}, f)$. The algorithm returns a residual path (if one is found) as well as the set of visited vertices during the traversal.

```
1  Function layeredTraversal(G, f, ℓ, v):
2      S ← {v^in}                                          // Visited vertices
       // For each j ∈ {0, ..., |f|}
3      Q_j ← {u^out | (u, v) ∈ E ∧ ℓ(u^out) = j}
4      for j ← |f| down to 0 do
5          while Q_j ≠ ∅ do
6              Remove u from the front of Q_j
7              S ← S ∪ {u}
8              for v ∈ N⁺(u) in R(G, f) do
9                  if v = t then
10                     return Residual path D, S
11                 if v ∉ S then
12                     Add v to the back of Q_ℓ(v)
13     return ∅, S
```

B Additional Experimental Data and Results

Table 1 Width k for dataset `Random DAG` ($N = 50,000$) and different values of parameter M.

$M = \lvert E \rvert$	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}	2^{23}	2^{24}	2^{25}	2^{26}	2^{27}
Width k	31,282	22,586	13,913	7,418	3,768	1,922	980	494	260	134	75	39	22

Table 2 Number of edges $|E|$ and width k for datasets `Path Partition` and `Transitive Closure` and different values of parameter M.

M	`Path Partition` $N = 50,000, K = 173$		`Transitive Closure` $N = 10,000$	
	$\|E\|$	k	$\|E\|$	k
2^{13}	—	—	15,103	5,752
2^{14}	—	—	66,725	3,971
2^{15}	82,595	173	930,242	2,346
2^{16}	115,363	173	11,609,355	1,214
2^{17}	180,899	173	28,629,499	614
2^{18}	311,971	173	39,559,611	311
2^{19}	574,115	173	45,288,626	161
2^{20}	1,098,403	171	47,910,591	87
2^{21}	2,146,979	165	49,121,619	46
2^{22}	4,244,131	140	49,647,821	26
2^{23}	8,438,435	111	49,869,260	14
2^{24}	16,827,043	79	—	—
2^{25}	33,604,259	54	—	—
2^{26}	67,158,691	34	—	—
2^{27}	134,267,555	21	—	—

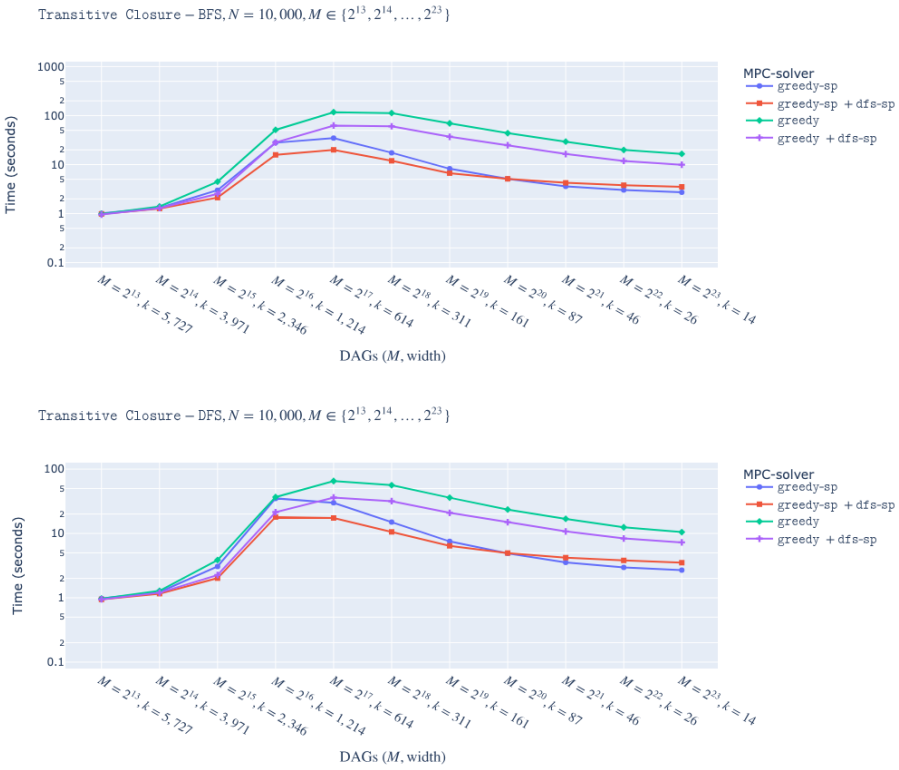

Figure 8 Running time of other solvers in dataset `Transitive Closure` with different combinations of pre-processings `dfs-sp` and `greedy-sp`.

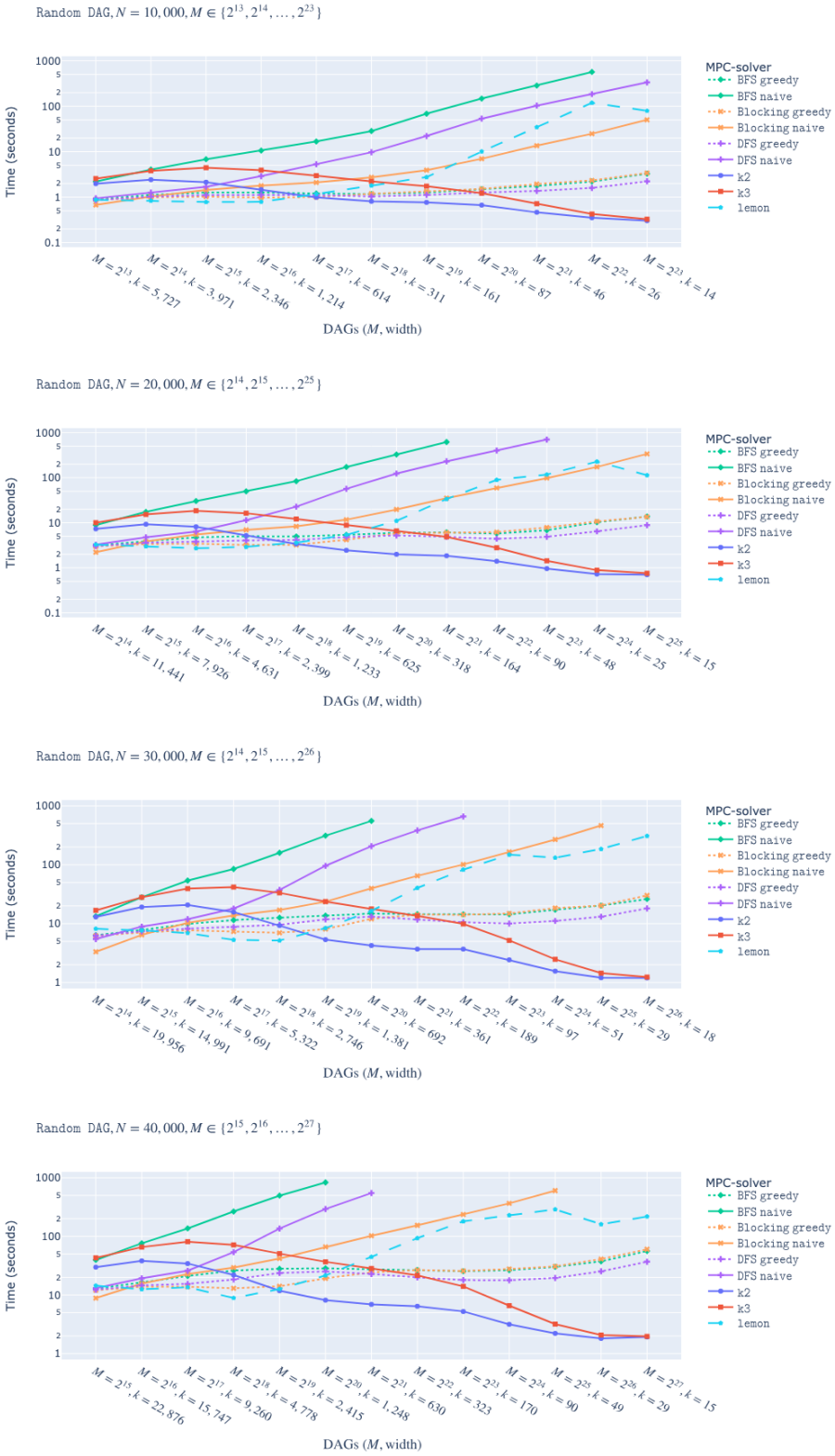

Figure 9 Running time of MPC-solvers in dataset `Random DAG`, for different N.

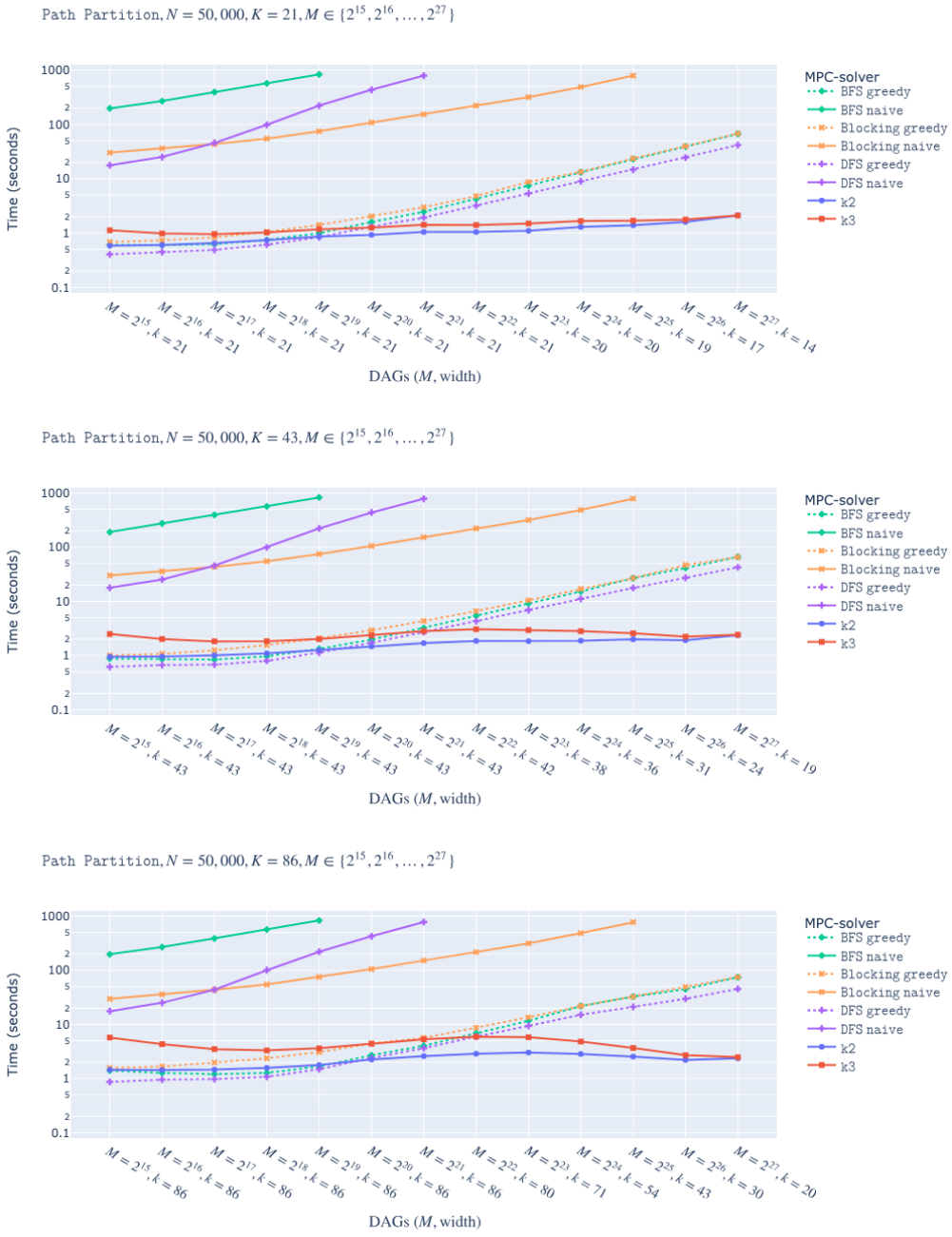

Figure 10 Running time of MPC-solvers in dataset `Path Partition`, for different K.