TLDC: Too Long Didn't Count
Entry for the International Competition on Graph Counting Algorithms

Rafael Kiesel
TU Wien
rafael.kiesel@web.de

Markus Hecher
Massachusetts Institute of Technology
United States
hecher@mit.edu

Abstract

We present the new and improved version of Too Long Didn't Count (TLDC), a solver for the length-limited path counting problem in directed and undirected graphs. Our solver is a portfolio of multiple approaches that are chosen based on problem parameters that we can exploit. The different approaches exploit pathwidth, treewidth, maximum path length, and the difference between the longest and shortest possible path, respectively.

1 Introduction

Informally, the one pair length limited path counting problem (PCS) is the following: Given a possibly directed graph G, a source vertex s, a target vertex t, and a maximum length ℓ, count all *simple* paths in G from s to t of length less than ℓ.

PCS is #P-complete [24], and thus, highly challenging to solve efficiently. Nevertheless, PCS is relevant for problems such as network reliability [20]. Therefore, algorithms that are efficient, to the extent possible, are of interest.

In this paper, we briefly describe the techniques that we used in our solver TLDC, submitted to the International Competition on Graph Counting Algorithms 2024 (ICGCA), to solve PCS-instances. A cleaned up version of the source code will be made available at `https://github.com/raki123/TL-DC` in the coming weeks.

TLDC is based on our submission of the same name to ICGCA 2023. While we observed significant speedups compared to the previous edition, the basic workflow remains similar and the improvements are mostly due to refinements of existing ideas.

The basic workflow of TLDC is as follows:

1. Simplification/Preprocessing
2. Parameter computation
3. Choice of strategy from portfolio
4. Search using chosen strategy

In the following, we first go over some preliminaries in Section 2, before going over the different steps of TLDC's workflow in Sections 3.1 to 3.4, respectively, highlighting changes compared to last year. Next, we compare this year's version of TLDC to last year's version empirically, in Section 4. Following up, we discuss the results and the impact of different ideas, in Section 5. Finally, we outline further opportunities to enable fast graph counting, in Section 6.

For a short summary, we recommend Section 5.

2 Preliminaries

We start by introducing some necessary definitions

Definition 1 (Graph, Digraph)**.** *A directed graph $G = (V, A)$ is a set of vertices V combined with a set of arcs $A \subseteq V \times V \setminus \{(v, v) \mid v \in V\}$. A graph is a directed graph (V, A) such that $(v, w) \in A$ iff $(w, v) \in A$ for all $v, w \in V$. For graphs, we talk about edges $\{v, w\} \in E$ (instead of $(v, w), (w, v) \in A$).*

For a graph $G = (V, E)$ (resp. directed graph $G = (V, A)$), we write $V(G)$ and $E(G)$ (resp. $A(G)$) for V and E (resp. A).

We are interested in simple paths in (directed) graphs.

Definition 2 (Path, Simple Path)**.** *Given a (directed) graph G, a* path *in G is a string $v_0, \ldots, v_n \in V^*$ such that for all $0 \leq i < n$ it holds that $(v_i, v_{i+1}) \in A(G)$. The length of such a path is defined as n, i.e., the number of vertices in it minus one, and it* starts *at v_0,* ends *at v_n, and is a path* from v_0 to v_n*. Such a path is* simple *if $|\{v_0, \ldots, v_n\}| = n + 1$, i.e., there are no repeated vertices.*

Formally, (directed) PCS is defined as the following problem:

Given: A (directed) graph G, terminals $s, t \in V(G), s \neq t$, and a maximum length $\ell \geq 0$.
Compute: The number of simple paths from s to t in G whose length is less or equal to ℓ.

To simplify the rest of the paper, we introduce some notation. For a (di)graph G and $v, v' \in V(G)$, we define

$$\begin{aligned}
N^+(v) &= \{w \in V(G) \mid (v, w) \in A(G)\} \\
N^-(v) &= \{w \in V(G) \mid (w, v) \in A(G)\} \\
N(v) &= N^+(v) \cup N^-(v) \\
G \setminus v &= (V(G) \setminus \{V\}, A(G) \setminus (\{(v, w) \in A(G)\} \cup \{(w, v) \in A(G)\}) \\
G \setminus (v, v') &= (V(G), A(G) \setminus \{(v, v')\}) \\
SP(v, w, G) &= \min_{(v, v_1, \ldots, v_{n-1}, w) \text{ is a path in } G} n
\end{aligned}$$

3 TLDC's Workflow

3.1 Preparation

Before we start solving PCS instances, we first try to simplify them as much as possible, to reduce unnecessary overhead. Here, we assume that instances are well formed, meaning that $s \neq t$ and $SP(s, t, G) \leq \ell$. Otherwise, we know the result is zero.

Given an instance (G, s, t, ℓ), we apply the following reductions:

Reduction 1 (Isolated Vertices)**.** *If $v \in V(G)$ such that one of $|N(v)| \leq 1$, $N^-(v) = \emptyset$, or $N^+(v) = \emptyset$ holds, replace (G, s, t, ℓ) by $(G \setminus v, s', t', \ell')$, where s' is the unique neighbor of s, if $v = s$, and $s' = s$, otherwise, t' is the unique neighbor of t, if $v = t$ and $t' = t$, otherwise, and $\ell' = \ell - 1$, if $v = s$ or $v = t$, and $\ell' = \ell$, otherwise.*

Intuitively, we can never use isolated vertices (apart from start and goal) in a path, since we may visit them but can never leave them or the other way around. For start and goal, there is only one option for the vertex in the path next to them, so we can use this vertex as the new start or goal and decrease the maximum length by one.

Reduction 2 (Unreachable Vertices)**.** *If $v \in V(G)$ such that $SP(s, v, G) + SP(v, t, G) > \ell$, replace (G, s, t, ℓ) by $(G \setminus v, s, t, \ell)$.*

Unreachable vertices can never occur in a path from s to t of length less or equal to ℓ.

Reduction 3 (Unusable Arcs/Edges)**.** *If $(v, w) \in A(G)$ such that $SP(s, v, G \setminus w)+1+SP(w, t, G \setminus v) > \ell$, replace (G, s, t, ℓ) by $(G \setminus (v, w), s, t, \ell)$.*

If we cannot use an arc in a path from s to t of length less or equal to ℓ, we can remove it without changing the result of the instance. For undirected graphs, we only apply this reduction if we would remove both (v, w) and (w, v). Practically, we only perform this reduction on (di)graphs with less than 10000 vertices, since computing the distance in the (di)graph with one vertex removed is too expensive to do for each vertex.[1]

Reduction 4 (Start-Goal Arcs/Edges)**.** *If $(s, t) \in A(G)$, replace (G, s, t, ℓ) by $(G \setminus (s, t) \setminus (t, s), s, t, \ell)$ and add one to the solution.*

Arcs/Edges directly from start to goal can be removed, reducing the solution exactly by one.

Reduction 5 (Forwarder Vertices)**.** *If $|V(G)| - 1 \leq \ell$, $v \in V(G)$, $v \neq s$, $v \neq t$,*

$$A^* = \{(x, y) \mid (x, v), (v, y) \in A(G), x \neq y\}$$

such that $A^* \cap A(G) = \emptyset$, and one of $|N(v)| = 2$, $|N^-(v)| = 1$ or $N^+(v) = 1$, then replace (G, s, t, ℓ) by (G', s, t, ℓ), where

$$V(G') = V(G) \setminus \{v\}$$

and

$$A(G') = A(G \setminus v) \cup A^*.$$

Intuitively, if path includes a forwarder vertex, neither as the start nor as the goal vertex, then at least one of its adjacent vertices in the path is predetermined. Thus, we can remove the forwarder vertex, and add the "forwarded" connections.

There are two things to be careful about here. Firstly, this clearly changes the length of some paths, since the forwarder vertex is not used anymore. Therefore, we can only apply this reduction if we are not length limited. Secondly, if one of the "forwarded" connections already exists, we reduce the number of paths, because some previously different paths are now represented by the same path.

Both of these problems can be alleviated by assigning arcs/edges a weight, that denotes how many paths of a certain length the arc/edge represents. In fact, we do exactly that (and lift most other reductions to (di)graphs with such arc/edge weights), if the chosen search strategy allows.

Reduction 6 (Dominators/Cut Vertices)**.** *If $v, w \in V(G)$ such that every path from s to w visits v, replace (G, s, t, ℓ) by $(G \setminus (w, v), s, t, \ell)$. If every path from w to t visits v, replace (G, s, t, ℓ) by $(G \setminus (v, w), s, t, \ell)$.*

In this case, we know in which order the vertices can occur on a path from s to t, if they occur. Therefore, we can remove arcs that point into the wrong direction since any path from s to t they occur on could not be simple, as it would have to use the v twice.

This reduction can help us in two ways. Firstly, it can happen that v occurs on any path from s and to t, with other end w. In this case, w cannot occur on any simple path from s to t, and we can remove it. This is the only way we use this reduction for undirected graphs. Secondly, it can remove the arcs in one direction thus decreasing the instance size. For directed graphs, this is also used.

Practically, this reduction is the one with the largest differences between directed and undirected graphs. Namely, detecting cut vertices (a.k.a. articulation points) can be detected easily

[1]It is likely that there is a more efficient way to check whether this reduction can be applied though.

in undirected graphs via a simple modification of the connected component algorithm for undirected graphs [9]. For directed graphs on the other hand, efficient algorithms to find dominators are much more complicated. There are nevertheless algorithms that can solve the problem in almost linear time [16, 6]. We would like to thank Georgiadis et al. for providing us with their implementations from [6, 7], which we use in our solver to compute dominators.

Reduction 7 (Position Determined Vertices)**.** *If $v \in V(G)$ such that $SP(s,v,G) + SP(v,t,G) > \ell - k$, for a small number k, replace (G,s,t,ℓ) by $(G \setminus v, s, t, \ell)$ and increase the solution by the number of paths that use v.*

Clearly, if we count all paths from s to t that visit v, we can remove v from G and solve the remaining problem independently on a smaller graph. Unfortunately, counting all paths that visit v may be as hard as counting all paths. Therefore, we only apply this reduction, when the minimum distance of a path that uses v is close to ℓ. In this case, we assume that the number of vertices that can be on a path including v is much smaller than $|V(G)|$, and counting becomes significantly easier. We perform this reduction only for $k = 2$. To count the paths, we use our DFS-based solver since it performs well, when the difference between $SP(s,t,G)$ and ℓ is low.

3.2 Search Strategies

We briefly go over the search strategies that we used. Since none of them are completely novel (i.e., were present already at last years edition of ICGCA), we do not explain them in detail but go over some details that we consider to be especially relevant for performance or are non-standard. For all approaches we have a directed and undirected version. However, they only differ by taking into account whether the input graph is directed or not.

3.2.1 Pathwidth Search

Frontier based search, where the edge order is generated in such a manner that the path width is small, is probably the most typical approach to path counting problems [12, 14, 13].

There are a few ways in which our implementation differs from the standard algorithm. Firstly, we added pruning based on the ℓ and the current length of the partial paths. This stayed the same compared to last year.

Secondly, our algorithm is multithreaded, which is the case in some other implementations but not all. We noticed that in last years edition the speedup we gained from parallelism was not as large as one might expect. We identified the insertions into the cache as the bottleneck here. This caused problems because (1) we locked every time we inserted into a cache and (2) all threads had to wait, when the cache had to be increased in size and rehashed.

We addressed (1) and partially (2) by having k caches (currently 1 per thread) and inserting partial solutions, whose hash value modulo k was i, into the i-th cache and locking only this particular cache. To further address (2), we additionally reserved enough space in the next caches for twice the amount of entries that we had in the current caches. This guarantees that resizing and rehashing never occurs.

Last year, we additionally applied a form of unit propagation, where we only cached partial solutions if the next edge could both be taken and skipped. Since we noticed that this was only helpful in the old solver due to the cache insertion bottleneck, we removed this feature in the current version.

One new feature in the current version is that we assign vertices an index between 0 and k, where k is the pathwidth, *globally* instead of per arc/edge to be eliminated. This comes at the cost of always having $k+1$ entries in the partial solution array, however, this is negligible since the hard parts of the search are expected to have $k+1$ entries anyway. On the upside it allows us to only replace the entry for completely processed vertices by the no-edge token, instead of iterating over the whole partial solution array, to adjust the partial solution to be correctly indexed.

Last years TLDC also used caching modulo isomorphisms based on NAUTY [18]. While this is still supported by this years version (although untested), we chose not to include it in this years version since (1) there was no benefit on the public instances and (2) it is very hard to properly include this strategy in a portfolio approach.

3.2.2 Treewidth Search

Also frontier based search with joins, i.e., based on an edge ordering generated from a branch- or tree decomposition, are neither new to ICGCA nor to research in general [25][2]. We apply the same techniques that we apply for pathwidth search here.

Additionally, this years version has significantly improved performance since we fixed a performance bug. Last year, after merging two partial path arrays we first multiplied their solution counts before checking whether we can actually use the result. This is clearly unnecessary and can be very expensive when the instance is length limited. This years edition first performs the check and then multiplies the solution counts.

3.2.3 DFS Search

To the best of our knowledge, we first introduced what we refer to as DFS Search in last years edition of ICGCA. We recall the basic idea.

Starting from s, we traverse all outgoing arcs/edges, mark s as visited, and decrement ℓ. Then, given the new start s' we run a shortest path algorithm, and mark all vertices v such that $SP(s, v, G) + SP(v, t, G \setminus s) > \ell$ [3] as visited since we can never visit them anymore. Furthermore, we mark all vertices that would be removed by Reduction 6 as visited. For the directed graphs, we treat the graph as undirected here. Then, we put the obtained subinstance into a cache combining counts on cache hits. This is repeated until we processed all subinstances.

Compared to last year, we made the following changes: On the one hand, we removed the optimization of counting all remaining paths if we could treat the rest of the graph as a directed acyclic graph because the distance to the goal was equal to ℓ. On a lot of instances this apparently actually hurt our performance instead of benefiting it.

On the other hand, we do not use NAUTY anymore to compute a canonical representation of the remaining graph[4]. Instead, we draw inspiration from [19] and only partially canonize the remaining graph by swapping visited flags of twin vertices such that the vertices with the smallest label are visited first. Here, two vertices $v, w \in V(G)$ are twins for our purposes, if $N^-(v) \setminus \{w\} = N^-(w) \setminus \{v\}$, $N^+(v) \setminus \{w\} = N^+(w) \setminus \{v\}$ and $(v, w) \in A(G)$ iff $(w, v) \in A(G)$. On instances with many twins, this performs much better than complete canonization using NAUTY.

The latter change additionally enables use to include DFS search into our portfolio in a much easier manner, by computing the relevant graph parameters not on the original graph but the one, where all twins are collapsed into one vertex.

3.2.4 Meet in the Middle (MITM) Search

MITM search was introduced by Ohba in the solver *diodrm* [19] submitted to last years edition of ICGCA. We refer to [19] for an explanation of MITM search and assume that the reader is familiar with the related terminology.

While DFS search and MITM search are good in similar situations, we noticed that instances with high average degree can be solved much faster using MITM search, if the length limit is sufficiently low. Therefore, we incorporated MITM search also into TLDC.

[2] Maybe the directed version is new but it does not differ significantly.

[3] We could also use $SP(s, v, G \setminus s)$.

[4] This is still supported but untested.

Our implementation differs in two aspects from Ohba's. Firstly, we do not use Zobrist hashing, whereas *diodrm* does. This is not an intentional choice but due to a lack of time, and hurts our performance.

Secondly, for MITM search given a path from start to middle *diodrm*'s implementation increments a counter for each subset of the path. However, this is only necessary for those subsets that can occur on a path from middle to goal. As a minor optimization we therefore only collect those vertices v on our path from start s to middle m such that $SP(m, v, G) + SP(v, t, G) \leq \lceil \ell/2 \rceil$. All other vertices surely cannot occur on a path from m to t.

3.3 Parameter Approximation

We have two groups of problem parameters that we approximate. Firstly, we have general parameters that we can exploit independently of the chosen strategy. Secondly, we have solver specific parameters that we use, on the one hand, to choose the strategy we apply to the problem, and, on the other hand, sometimes explicitly exploit during the search.

3.3.1 General Parameters

We consider two general parameters.

Firstly, the maximum length ℓ^* of any path from s to t. This helps us since, if ℓ^* is less or equal to ℓ, we do not need to track the lengths of partial paths during the search, which can introduce a significant overhead. We note that this was not our idea but an optimization of *diodrm* [19] that we adopted too.

Since computing or even upper-bounding the maximum length ℓ^* is hard, our current implementation uses $|V(G)| - 1$ as an upper-bound for ℓ^*. We investigated different possible strategies of obtaining better upper-bounds, e.g. using SAT- or MILP-solvers, and a specialized solver [2], however, even the specialized solver took significantly longer than the search itself took.

Secondly, we noticed that using arbitrary size integers from GMP [8] to store partial results introduces a significant performance hit compared to using fixed width integers, even though GMP is highly optimized. Therefore, it is desirable to use fixed width integers instead. However, we clearly cannot just choose some fixed integer type and expect that the number of paths that we need to count will be less. Instead, we compute an upper-bound on the number of paths and choose the smallest integer type that can store the upper-bound. Here, we choose from 64-bit, 128-bit, 256-bit, 512-bit, 1024-bit or arbitrary size GMP integers.

For the upper-bound, we use the following lemmas.

Lemma 3 (Upper-bound Directed)**.** *Let (G, s, t, ℓ) be a directed* PCS *instance, and $v_1, \ldots, v_{\ell-2} \in V(G)$, the $\ell - 2$ vertices with the highest out-degree, in descending order. Then the number of paths from s to t in G is less or equal to*

$$|N^+(s)| \cdot |N^-(t)| \cdot \sum_{j=0}^{\ell-2} \prod_{i=1}^{j} |N^+(v_i)|.$$

Lemma 4 (Upper-bound Directed)**.** *Let (G, s, t, ℓ) be an undirected* PCS *instance, and $v_1, \ldots, v_{\ell-2} \in V(G)$, the $\ell - 2$ vertices with the highest degree, in descending order. Then the number of paths from s to t in G is less or equal to*

$$|N(s)| \cdot |N(t)| \cdot \sum_{j=0}^{\ell-2} \prod_{i=1}^{j} (|N(v_i)| - 1).$$

Proof.][Proof sketches of Lemmas 3 and 4] The proofs of both lemmas are rather similar. The basic idea is the following: The first arc is always one from s and the last arc is always one to t, which is why we multiply by the first two factors.

Strategy	Path-FBS	Tree-FBS	DFS	MITM
Parameters	Pathwidth	Treewidth, Join-size	$\ell - SP(s,t,G)$, Avg. degree	MITM-width

Table 1: Relevant parameters for each strategy

Next, we sum over all remaining path lengths j from 0 to $\ell - 2$. To obtain a path from s to t of length $j+2$, we then choose j arcs/edges in G one after the other (assuming we already chose the first and last arc). For each arc/edge, we can choose any outgoing arc from the previous vertex, or in the case of undirected graph, any edge apart from the one that we previously took.

Since we take j arcs/edges, we make at most j choices. In the worst case, these choices are made at the vertices with the highest j (out-)degrees. □

This provides us with upper-bounds that are good enough to avoid the usage of GMP-integers in a lot of cases.

3.3.2 Strategy Parameters

For the different strategies, the parameters in Table 1 are relevant. Here, Join-size refers to the maximal sum of sizes of child bags of a join node in a tree decomposition, and MITM-width is an approximation of the logarithm of the number of steps MITM search would take on the given instance.

For each of the (non-trivial) parameters, we describe our way of obtaining an approximation for them.

Pathwidth For pathwidth, we tried a wide range of approaches, ranging from heuristics in the literature, over SAT-based approaches, to local improvement strategies. While the SAT-based approaches were able to provide the best approximations we deemed them too time consuming for use in ICGCA. Instead, the final implementation uses the heuristic H1 from [10]. More specifically, if the instance was not solved by preprocessing, we use all available threads each with a timeout of 15 seconds to compute path decompositions. Each threads generates path decompositions by repeatedly choosing one of the next vertices recommended by H1 uniformly at random.

We note that the main benefit of the (weighted) forwarder preprocessing lies here since it (1) may reduce the pathwidth of the instance and (2) leads to smaller graphs and thus faster decomposition generation, allowing us to explore more possibilities.

Treewidth Here, a wide range of implementations is available, due to the PACE challenge [4], which considered the problem of computing treewidth exactly and heuristically. We tested flow-cutter [21], htd [1], and Tamaki et al.'s solver [22], all with a timeout of 15 seconds. While all solvers had some benefits and advantages the solver by Tamaki et al. is by far the best for usage with TLDC. It performs excellently on small instances and is able to provide decompositions, whose width is often times by two lower than the width of those computed by flow-cutter and htd. While it failed to produce any decompositions for some of the larger graphs, whereas flow-cutter and sometimes also htd managed, the corresponding graphs had such large width anyway that a treewidth based search was doomed to fail miserably anyway.

We therefore use Tamaki et al.'s solver with a timeout of 15 seconds to compute a tree decomposition, unless the instance was already solved by preprocessing.

We also considered the possibility of using branch decompositions, which can have strictly smaller widths than the best tree decomposition. For this we tested the implementation of [17], however, the resulting widths were not better than those obtained for treewidth using Tamaki et al.'s solver. The code is nevertheless included.

MITM-width The runtime of MITM search depends (in our experience) mostly on two parameters: the number of vertices in the paths from the middle to start/goal that need to be considered, when storing partial results, and the number of paths. These values are hard to predict, without actually running the algorithm. Therefore, we run MITM search until it finds one million paths from some middle to the start and compute the average number of vertices that needed to be consider for storing the partial results. Then, we approximate the percentage of paths we found after having processed the first million paths. These two values are then combined to obtain what we call MITM-width. The exact formula can be found in Section 3.4.

We added multiple fail safes to this computation, since sometimes already finding the first million paths took prohibitively long.

3.4 Portfolio

To combine the different search strategies, we have two main options. Either, we run them in parallel/sequentially or we choose one strategy and use our whole budget on it. We chose the latter.

A possible downside of this approach is that we may choose the wrong search strategy for an instance that could have been solved in seconds using a different strategy. The upside is that if we were to *always* choose the correct strategy, we would have around four times the budget to solve the instance.

Always choosing the correct strategy is too ambitious. Nevertheless, or rather therefore, we put a lot of effort into finding a way to choose a good strategy based on the parameters computed in Section 3.3.

To tune our parameters, we used all public instances from this year as well as the PCS instances from last year.

Our initial idea was to predict, based on the parameters, whether an given instance will finish using a selected strategy. Consider for this Figure 1. Intuitively, we need a way to separate the light green circles, which correspond to the instances that we do not solve, from the dark blue circles, which represent the instances that we easily solve. However, as we can see there is no smooth color gradient. This means for instances that fall into the areas of green-blue color we are likely to have many wrong predictions.

Luckily, we do not need to know whether an instances finishes with a selected strategy but we only need to be able to predict whether a select strategy performs significantly better than the other strategies. For this, we intuitively want to find parameters $p_{tw}, p_{pw}, p_{dfs}, p_{mitm}$ for treewidth, pathwidth, DFS and MITM search, respectively, such that a parameter is the lowest iff the corresponding strategy is the fastest or at least not significantly worse than the fastest strategy. We build these parameters based on those that we described in the last section.

The precise parameters are as follows:

- $p_{pw} = k$, where $k - 1$ is the pathwidth,
- $p_{tw} = k + {}^{\max(k-js,0)}/_{3}$, where $k - 1$ is the treewidth and js is the join size,
- $p_{dfs} = {}^{1}/_{6}(\ell - SP(s,t,G)) \cdot \Delta$, where Δ is the average degree (counting undirected edges twice),
- $p_{mitm} = 39 \max(ps, 0.1) \cdot (\log(10^6 \max(pd, 2 \cdot 10^{-6})))^{-1}$, where ps is the average number of vertices to cache for paths from s to m, we found, and pd is the estimated ratio of paths from s to m we found (usually 10^6) out of all paths from s to m. The maxima avoid errors at border cases. So if $pd = 0.1$, we found 10 percent of the paths and need to invest ten times the effort to find all of them. The logarithm makes it so that the runtime is exponential in the parameter, as with all the other parameters.

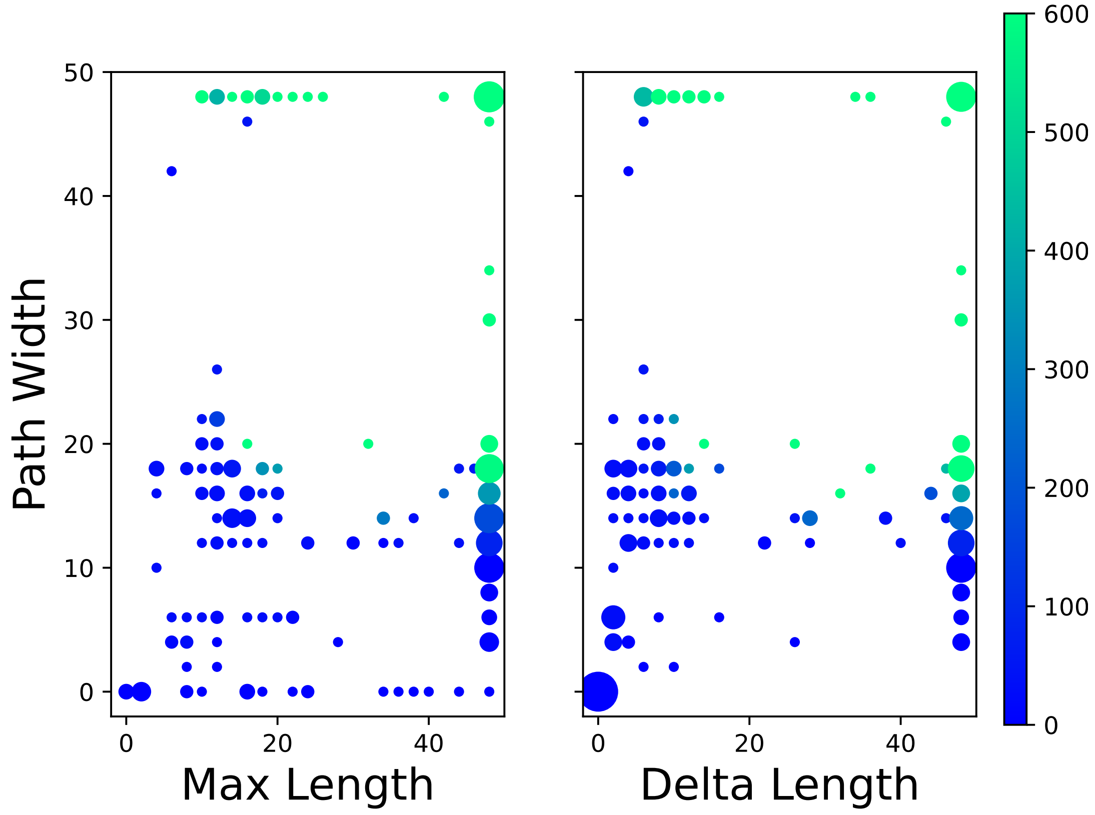

Figure 1: The average runtime of pathwidth search of instances, denoted by color, grouped by ℓ (left) or $\ell - SP(s,t,G)$ (right) and the computed pathwidth upper-bound. Circle size shows the number of instances in a given group.

For the results, consider Figures 2 to 4. In each figure, we try to find a new parameter p_S such that the instances that can be solved significantly faster with search S than with the already considered searches S' have $p_S < p_{S'}$. The figures show two plots, one for the instances which are solved faster (by at least 30 seconds) using the previous searches (left) and one for the instances which are solved faster (by at least 30 seconds) using the current search. The Y-axis shows the minimum of the old parameters and the X-axes show the parameter we used for the current search. The blue diagonal denotes where the two are equal. The plots then show the average runtime benefit of using the search associated with the plot (i.e. right is current, left is virtual best of the previous) of instances grouped by the parameters on the axes. Intuitively, we found well-comparable parameters if the instances groups on left plot are below the diagonal and the instance groups on the right side are on top of the diagonal. Most importantly, this should be the case for the light green instance groups since these are the ones that have a significant benefit/deficit w.r.t. some search strategy.

Summarizing, we see that our parameters work very well for DFS and MITM search, with no instances being misclassified. For pathwidth and treewidth search in Figure 2 we see that the preferred search is very hard to distinguish. Unfortunately, there some of the instances are misclassified. However, only for one instance[5] this actually means that the search we choose according to the parameters does not solve the instance, whereas the other one would have.

[5]Namely the PCS instance “064.col” from last year.

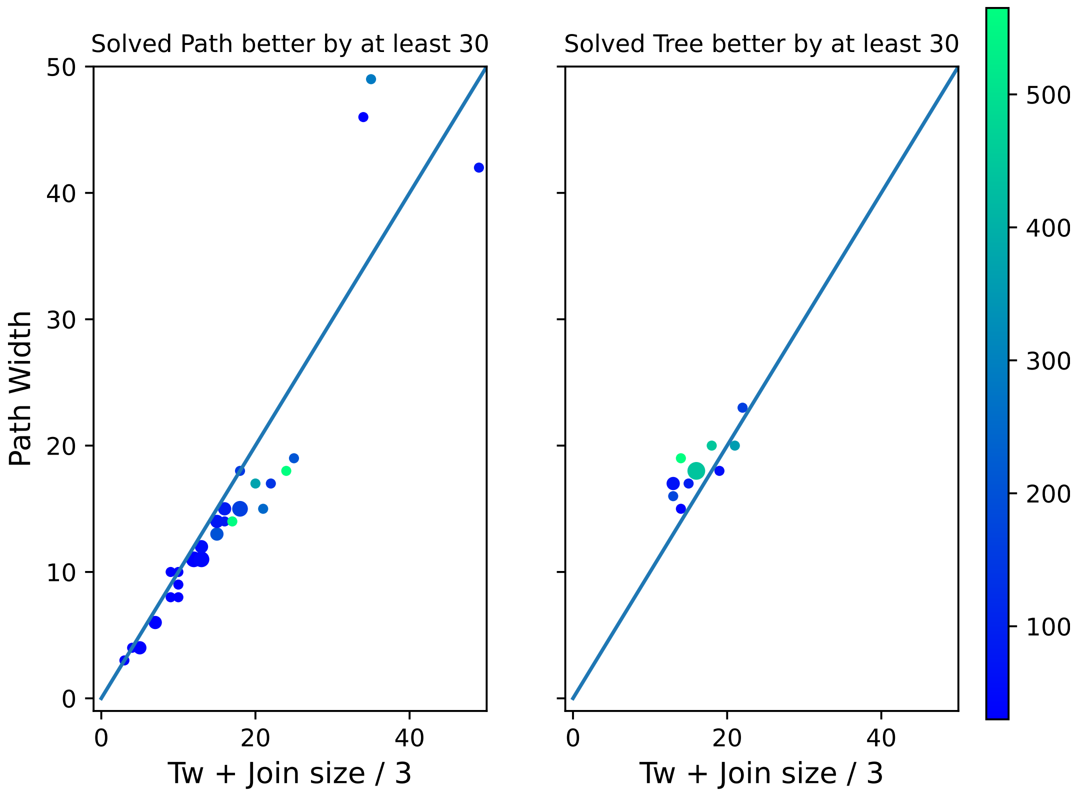

Figure 2: Average runtime *benefit* of using treewidth search (left) or pathwidth search (right) over the other, for instances with a runtime difference of at least 30 seconds.

3.5 General Infrastructure Improvements

Apart from these high level advances, we also incorporated some low level changes that are based more on engineering rather than scientific insights.

Firstly, we incorporated *jemalloc* [5, 11]. *jemalloc* is a scalable concurrent malloc implementation, meaning it handles multithreaded memory allocations far better than the standard malloc implementation. We saw confirmation of this. Simply by adding *jemalloc* our runtime was approximately halved. This effect is likely not observable when running TLDC in a single thread or at least likely much less significant.

Secondly, we replaced "std::unordered_map" as the underlying data structure that we used for all caches by the *ankerl* dense hash map implementation [15]. This also resulted in shorter runtimes by around 20%. This effect may vary from instance to instance but should also be observable in the singlethreaded setting.

4 Empirical Evaluation

We provide a short empircal evaluation of the different search strategies, as well as our final portfolio approach. For this, we use the PCS instances from last year, as well as all instances from this year, including the later removed instances with one path. We note that we always consider the multithreaded setting here, with a timeout of 600 seconds, a memory limit of 64GB, and 12 threads. All results need to be taken with a grain of salt since we made some final changes before

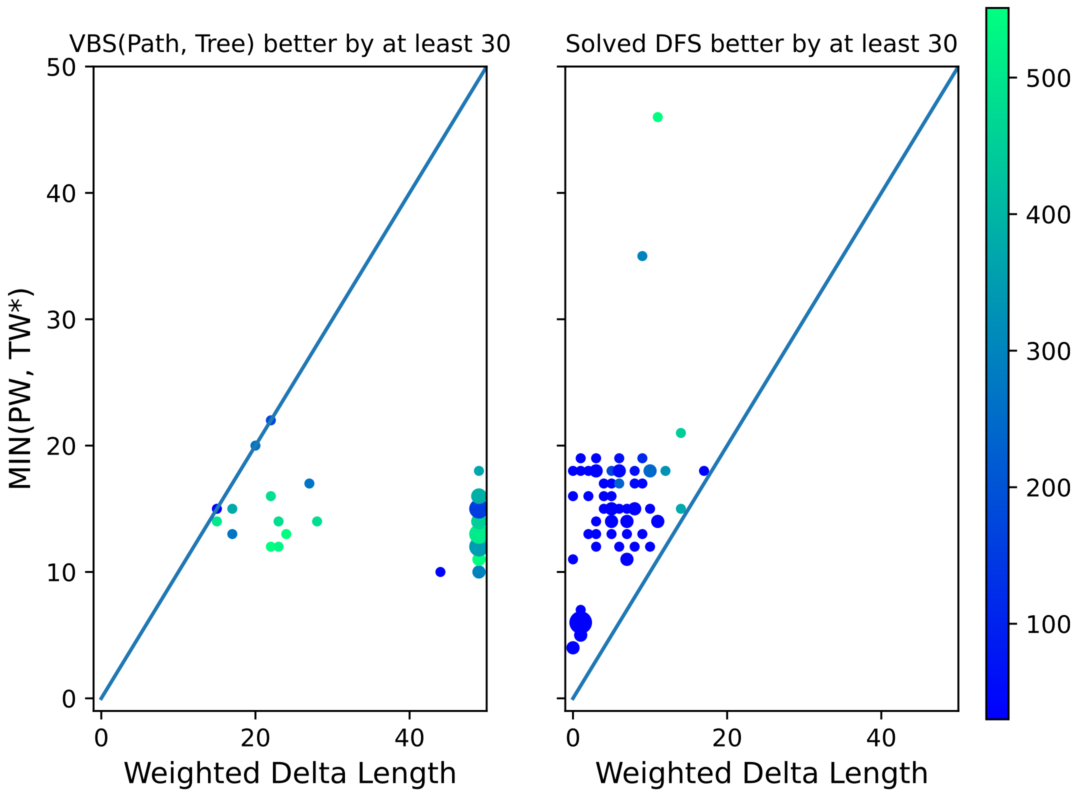

Figure 3: Average runtime *benefit* of using the best of treewidth search and pathwidth search (left) or DFS search (right) over the other, for instances with a runtime difference of at least 30 seconds.

benchmarking the portfolio version that were not yet incorporated into the separate searches before.

The results are in Figure 5. Firstly, we see in the top plot that our solver has significantly improved compared to last year. While last year TLDC solved 91 instances in the competition, this year it solves 96 in the portfolio approach and has the potential to solve 97, when using the virtual best solver. This is mostly due to our improvements to the pathwidth- and treewidth-based search, which both solve more than 91 instances in their own now.

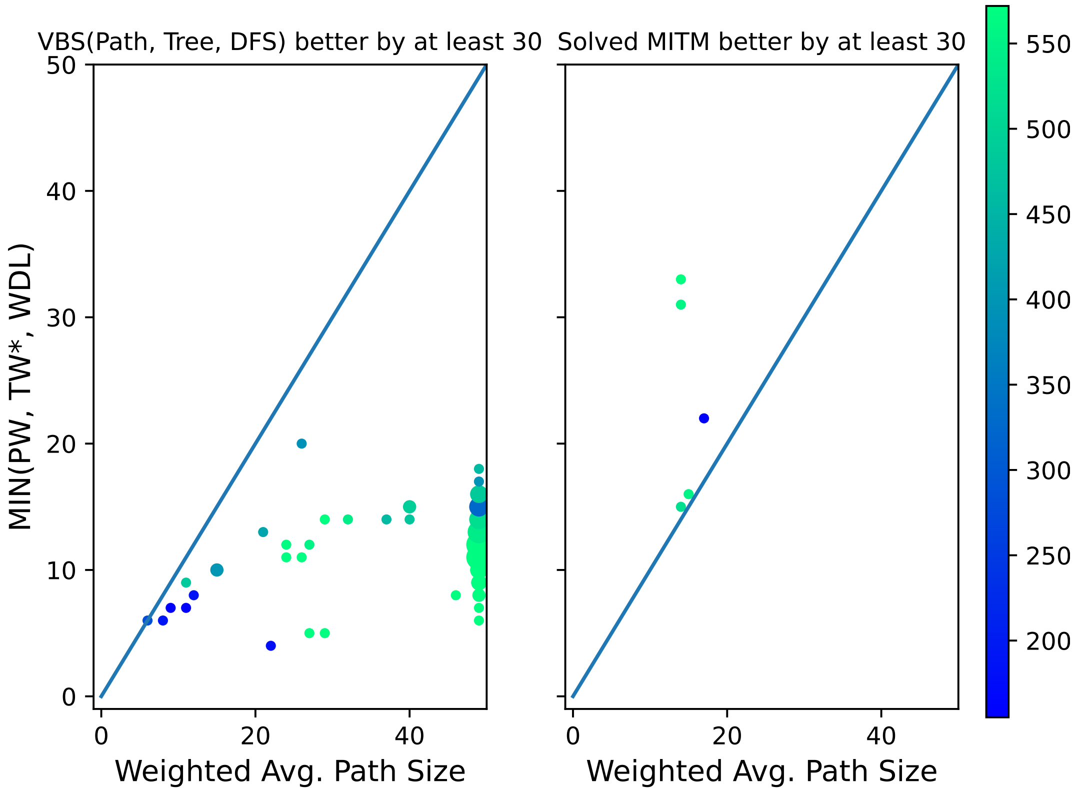

Figure 4: Average runtime *benefit* of using the best of treewidth search, pathwidth search, and DFS search (left) or MITM search (right) over the other, for instances with a runtime difference of at least 30 seconds.

5 Conclusion & Discussion

In short, we see that TLDC improved in multiple areas:

- Better and faster preprocessing,
- General engineering advances, based on better multithreaded allocation and cache data structures,
- General scientific advances, based on different count-types based on an upper-bound on the solution and the maximum length of a path in the graph,
- Minor search strategy specific ideas, like improved caching mechanisms, reconsideration of previous optimizations, etc.,
- The addition of MITM search [19] as a search strategy,
- Better heuristics for width parameters, and
- A portfolio that generalizes better.

Overall, we suspect that the better heuristics for pathwidth and treewidth had the largest benefit. However, also the addition of MITM search was important, since it enabled us to solve four extra directed instances.

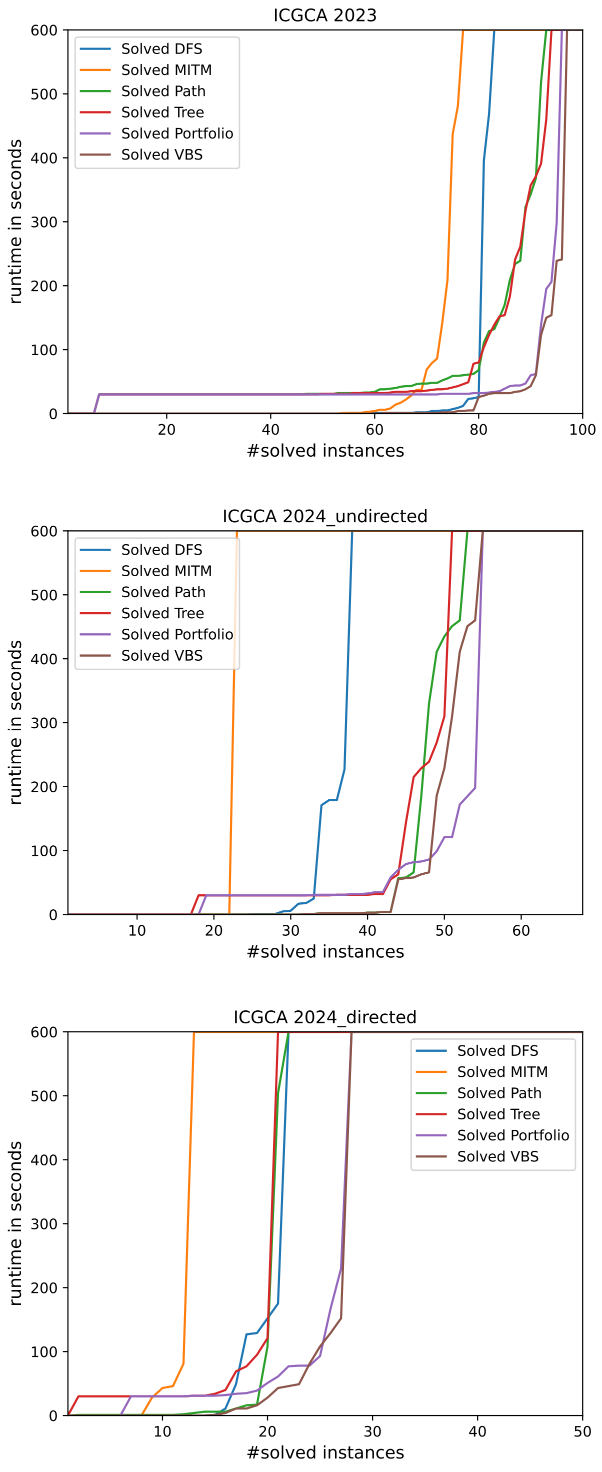

Figure 5: Number of instances solved by the different searches after a certain time. The benchmarks include instances with only a single path that have been removed later.

6 Open Ideas

While TLDC has gotten much faster, there is still a lot of room for improvement. We shortly list some ideas that we found interesting but did not have time to consider in detail.

- Different hashing, specifically Zobrist hashing as in *diodrm* [19]. Our current hashing is specifically good on large chunks of data but except for DFS search, the cache keys are usually small.
- Better heuristics for path-, tree-, and branchwidth can lead to incredible jumps in performance, even if they only decrease the width marginally.
- Exploiting twin vertices for pathwidth search would likely speed up the search both compared to pathwidth search with caching modulo graph isomorphism and normal search, if the number of twins is large. Additionally, it could likely be incorporated into a portfolio much more easily than the search module isomorphism.
- Incorporating Reduction 6 properly in the directed setting maybe even based on a dynamic data structure for the dominators has the potential for much faster DFS search.
- Currently, TLDC does not explicitly exploit strongly connected components in the directed setting. This years (public) instances did not exhibit any need for such a feature, however, other instances may be of a different shape.
- Better upper-bounds for the general parameters can lead to faster search without having to change the search algorithms themselves.
- There is an algorithm for directed path counting [3] that exploits *directed* pathwidth, which may be much smaller than undirected pathwidth. While its theoretical runtime upper-bounds hint at the fact that it will likely be unusable in many cases, there are cases, where it would likely be worth it to have this algorithm in the portfolio.

Last but not least, we found one interesting fact that we want to mention separately. Namely, the number of entries in the cache during frontier based search not only depends on the width of the path-/tree-/branchdecomposition but also on some other indeterminate other things. Specifically, we noticed while comparing TLDC and *diodrm* on specific instances that the edge ordering chosen by *diodrm* resulted in less cache entries than the one chosen by TLDC, even though the width of *diodrm*'s order was higher. Thus, there seems to be more to take into account than only the width. While we currently have no hints at what this may be, it would be very interesting and likely highly beneficial to find out.

References

[1] Michael Abseher, Nysret Musliu, and Stefan Woltran. htd–a free, open-source framework for (customized) tree decompositions and beyond. In *Integration of AI and OR Techniques in Constraint Programming: 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings 14*, pages 376–386. Springer, 2017.

[2] Tomas Balyo, Kai Fieger, Dominik Schreiber, and Christian Schulz. Finding Optimal Longest Paths by Dynamic Programming in Parallel. 2019.

[3] Mateus de Oliveira Oliveira. An algorithmic metatheorem for directed treewidth. *Discret. Appl. Math.*, 204:49–76, 2016. doi: 10.1016/J.DAM.2015.10.020. URL `https://doi.org/10.1016/j.dam.2015.10.020`.

[4] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. The PACE 2017 parameterized algorithms and computational experiments challenge: The second iteration. In Daniel Lokshtanov and Naomi Nishimura, editors, *12th International Symposium on Parameterized and Exact Computation, IPEC 2017, September 6-8, 2017, Vienna, Austria*, volume 89 of *LIPIcs*, pages 30:1–30:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPICS.IPEC.2017.30. URL `https://doi.org/10.4230/LIPIcs.IPEC.2017.30`.

[5] Jason Evans. A scalable concurrent malloc (3) implementation for freebsd. In *Proc. of the bsdcan conference, ottawa, canada*, 2006.

[6] Loukas Georgiadis, Robert Tarjan, and Renato Werneck. Finding dominators in practice. *Journal of Graph Algorithms and Applications*, 10(1):69–94, Jan. 2006. doi: 10.7155/jgaa.00119. URL `https://jgaa.info/index.php/jgaa/article/view/paper119`.

[7] Loukas Georgiadis, Konstantinos Giannis, Giuseppe F. Italiano, Aikaterini Karanasiou, and Luigi Laura. Dynamic Dominators and Low-High Orders in DAGs. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 50:1–50:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-124-5. doi: 10.4230/LIPIcs.ESA.2019.50. URL `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2019.50`.

[8] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.3.0 edition, 2024. `http://gmplib.org/`.

[9] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16(6):372–378, June 1973. ISSN 0001-0782. doi: 10.1145/362248.362272. URL `https://doi.org/10.1145/362248.362272`.

[10] Pallavi Jain, Gur Saran, and Kamal Srivastava. Polynomial time efficient construction heuristics for vertex separation minimization problem. *Electron. Notes Discret. Math.*, 63:353–360, 2017. doi: 10.1016/J.ENDM.2017.11.032. URL `https://doi.org/10.1016/j.endm.2017.11.032`.

[11] jemalloc. jemalloc. `https://github.com/jemalloc/jemalloc`, 2024.

[12] Jun Kawahara, Takeru Inoue, Hiroaki Iwashita, and Shin-ichi Minato. Frontier-based search for enumerating all constrained subgraphs with compressed representation. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 100(9):1773–1784, 2017.

[13] Donald Ervin Knuth. *The art of computer programming*, volume 4A. 2011.

[14] Richard E Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald. Frontier search. *Journal of the ACM (JACM)*, 52(5):715–748, 2005.

[15] Martin Leitner-Ankerl. ankerl::unordered_dense::map, set. `https://github.com/martinus/unordered_dense`, 2024.

[16] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. doi: 10.1145/357062.357071. URL `https://doi.org/10.1145/357062.357071`.

[17] Jing Ma, Susan Margulies, Illya V. Hicks, and Edray Goins. Branch decomposition heuristics for linear matroids. *Discret. Optim.*, 10(2):102–119, 2013. doi: 10.1016/J.DISOPT.2012.11.004. URL `https://doi.org/10.1016/j.disopt.2012.11.004`.

[18] Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of symbolic computation*, 60:94–112, 2014.

[19] Shou Ohba. Submission for international competition on graph counting algorithms. `https://afsa.jp/icgca2023/files/user17/17.pdf`, 2023.

[20] J Scott Provan and Michael O Ball. Computing network reliability in time polynomial in the number of cuts. *Operations research*, 32(3):516–526, 1984.

[21] Ben Strasser. Computing tree decompositions with flowcutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017. URL `http://arxiv.org/abs/1709.08949`.

[22] Hisao Tamaki. Positive-instance driven dynamic programming for treewidth. *J. Comb. Optim.*, 37(4):1283–1311, 2019. doi: 10.1007/S10878-018-0353-Z. URL `https://doi.org/10.1007/s10878-018-0353-z`.

[23] Hisao Tamaki, Hiromu Ohtsuka, Takuto Sato, and Keitaro Makii. Pace2017-tracka. `https://github.com/TCS-Meiji/PACE2017-TrackA`, 2017-.

[24] Leslie G Valiant. The complexity of enumeration and reliability problems. *siam Journal on Computing*, 8(3):410–421, 1979.

[25] Norihito Yasuda, Teruji Sugaya, and Shin-ichi Minato. Fast compilation of s-t paths on a graph for counting and enumeration. In Antti Hyttinen, Joe Suzuki, and Brandon M. Malone, editors, *Proceedings of the 3rd Workshop on Advanced Methodologies for Bayesian Networks, AMBN 2017, Kyoto, Japan, September 20-22, 2017*, volume 73 of *Proceedings of Machine Learning Research*, pages 129–140. PMLR, 2017. URL `http://proceedings.mlr.press/v73/teruji-sugaya17a.html`.