Electronic Colloquium on Computational Complexity, Revision 1 of Report No. 45 (2019)

On the Fine-grained Complexity of Least Weight Subsequence in Multitrees and Bounded Treewidth DAGs

Jiawei Gao
University of California, San Diego
jiawei@cs.ucsd.edu

Abstract

This paper introduces a new technique that generalizes previously known fine-grained reductions from linear structures to graphs. Least Weight Subsequence (LWS) [30] is a class of highly sequential optimization problems with form $F(j) = \min_{i<j}[F(i) + c_{i,j}]$. They can be solved in quadratic time using dynamic programming, but it is not known whether these problems can be solved faster than $n^{2-o(1)}$ time. Surprisingly, each such problem is subquadratic time reducible to a highly parallel, non-dynamic programming problem [36]. In other words, if a "static" problem is faster than quadratic time, so is an LWS problem. For many instances of LWS, the sequential versions are equivalent to their static versions by subquadratic time reductions. The previous result applies to LWS on linear structures, and this paper extends this result to LWS on paths in sparse graphs, the Least Weight Subpath (LWSP) problems. When the graph is a multitree (i.e. a DAG where any pair of vertices can have at most one path) or when the graph is a DAG whose underlying undirected graph has constant treewidth, we show that LWSP on this graph is still subquadratically reducible to their corresponding static problems. For many instances, the graph versions are still equivalent to their static versions.

Moreover, this paper shows that if we can decide a property of form $\exists x \exists y P(x, y)$ in subquadratic time, where P is a quickly checkable property on a pair of elements, then on these classes of graphs, we can also in subquadratic time decide whether there exists a pair x, y in the transitive closure of the graph that also satisfy $P(x, y)$.

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

Keywords and phrases fine-grained complexity, dynamic programming, graph reachability

Digital Object Identifier 10.4230/LIPIcs.IPEC.2019.15

Funding Work supported by a Simons Investigator Award from the Simons Foundation.

Acknowledgements I sincerely thank Russell Impagliazzo for his guidance and advice on this paper. I would like to thank Marco Carmosino and Jessica Sorrell for helpful comments. Also I would like to thank the anonymous reviewers for comments on an earlier version of this paper.

1 Introduction

1.1 Extending one-dimensional dynamic programming to graphs

Least Weight Subsequence (LWS) [30] is a type of dynamic programming problems: select a set of elements from a linearly ordered set so that the total cost incurred by the adjacent pairs of selected elements is optimized. It is defined as follows: Given elements $x_0, \ldots, x_n$, and an $n \times n$ matrix C of costs $c_{i,j}$ for all pairs of indices $i < j$, compute F on all elements, defined by

$$F(j) = \begin{cases} 0, \text{ for } j = 1 \\ \min_{0 \le i < j}[F(i) + c_{i,j}], \text{ for } j = 2, \ldots, n \end{cases}$$

ISSN 1433-8092

$F(j)$ is the optimal cost value from the first element up to the j-th element. We use the notation LWS_C to define the LWS problem with cost matrix C. The Airplane Refueling problem [30] is a well known example of LWS: Given the locations of airports on a line, find a subset of the airports for an airplane to add fuel, that minimizes the total cost. The cost of flying from the i-th to the j-th airport without stopping is defined by $c_{i,j}$. Other LWS examples include finding a longest chain satisfying a certain property, such as Longest Increasing Subsequence [25] and Longest Subset Chain [36]; breaking a linear structure into blocks, such as Pretty Printing [34]; variations of Subset Sum such as special versions of the Coin Change problem and the Knapsack problem[36]. These problems have $O(n^2)$ time algorithms using dynamic programming, and in many special cases it can be improved: when the cost satisfies the quadrangle inequality or some other properties, there are near linear time algorithms [50, 46, 26]. But for the general LWS, it is not known whether these problems can be solved faster than $n^{2-o(1)}$ time.

A general approach to understanding the fine-grained complexity of these problems was initiated in [36]. Many LWS problems have succinct representations of $c_{i,j}$. Usually C is defined implicitly by the data associated to each element, and the size of the data on each element is relatively small compared to n. Taking problems defined in [36] as examples, in LowRankLWS, $c_{i,j} = \langle \mu_i, \sigma_j \rangle$, where μ_i and σ_j are boolean vectors of length $d \ll n$ associated to each element that are given by the input. The ChainLWS problem has costs $c_1, \ldots, c_n$ defined by a boolean relation P so that $c_{i,j}$ equals c_j if $P(i,j)$ is true, and ∞ otherwise. P is computable by data associated to element i and element j. (For example, in LongestSubsetChain, $P(i,j)$ is true iff set S_i is contained in set S_j, where S_i and S_j are sets associated to elements i and j respectively.) So the goal of the problem becomes finding a longest chain of elements so that adjacent elements that are to be selected satisfy property P. When C can be represented succinctly, we can ask whether there exist subquadratic time algorithms for these problems, or try to find subquadratic time reductions between problems. [36] showed that in many LWS_C problems where C can be succinctly described in the input, the problem is subquadratic time reducible to a corresponding problem, which is called a $\mathsf{StaticLWS}_C$ problem. The problem $\mathsf{StaticLWS}_C$ is: given elements $x_1, \ldots, x_n$, a cost matrix C, and values $F(i)$ on all $i \in \{1, \ldots, n/2\}$, compute $F(j) = \min_{i \in \{n/2+1,\ldots,n\}}[F(i) + c_{i,j}]$ for all $j \in \{n+1, \ldots, 2n\}$. It is a parallel, batch version (with many values of j rather than a single one) of the LWS update rule applied sequentially one index at a time in the standard DP algorithm. The reduction from LWS_C to $\mathsf{StaticLWS}_C$ implies that a highly sequential problem can be reducible to a highly parallel one. If a $\mathsf{StaticLWS}_C$ problem can be solved faster than quadratic time, so can the corresponding LWS_C problem. Apart from one-directional reductions from general LWS_C to $\mathsf{StaticLWS}_C$, [36] also proved subquadratic time equivalence between some concrete problems (LowRankLWS is equivalent to MinInnerProduct, NestedBoxes is equivalent to VectorDomination, LongestSubsetChain is equivalent to OrthogonalVectors, and ChainLWS, which is a generalization of NestedBoxes and LongestSubsetChain, is equivalent to Selection, a generalization of VectorDomination and OrthogonalVectors).

Some of the LWS problems can be naturally extended from lines to graphs. For example, on a road map, we wish to find a path for a vehicle, along which we wish to find a sequence of cities where the vehicle can rest and add fuel so that the total cost is minimized. The cost of traveling between cities x and y without stopping is defined by cost $c_{x,y}$. Connections between cities could be a general graph, not just a line. Works about algorithms for special LWS problems on special classes of graphs include [11, 43, 24, 38].

Using a similar approach as [36], this paper extends the Least Weight Subsequence problems to the Least Weight Subpath (LWSP_C) problem whose objective is to find a least weight subsequence on a path of a given DAG $G = (V, E)$. Let there be a set V_0 containing vertices that can be the starting point of a subsequence in a path. The optimum value on each vertex is defined by:

$$F(v) = \begin{cases} \min(0, \min_{u \rightsquigarrow v}[F(u) + c_{u,v}]), & \text{for } v \in V_0 \\ \min_{u \rightsquigarrow v}[F(u) + c_{u,v}], & \text{for } v \notin v_0 \end{cases}$$

where $u \rightsquigarrow v$ means v is reachable from u. The goal of LWSP_C is to compute $F(v)$ for all vertices $v \in V$. Examples of LWSP_C problems will be given in Appendix B. LWSP_C can be solved in time $O(|V| \cdot |E|)$ by doing reversed depth/breadth first search from each vertex, and update the F value on the vertex accordingly. It is not known whether it has faster algorithms, even for Longest Increasing Subsequence, which is an LWS_C instance solvable in $O(n \log n)$ time on linear structures. If C is succinctly describable in similar ways as LowRankLWS, NestedBoxes, SubsetChain or ChainLWS, we wish to study if there are subquadratic time algorithms or subquadratic time reductions between problems.

For the cost matrix C, we consider that every vertex has some additional data so that $c_{x,y}$ can be computed by the data contained in x and y. Let the size of additional data associated to each vertex v be its weighted size $w(v)$. The weight of a vertex can be defined in different ways according to the problems. For example, in LowRankLWS, the weighted size of an element can be defined as the dimension of its associated vector; and in SubsetChain, the weighted size of an element is the size of its corresponding subset. We use $m = |E|$ as the number of graph edges. Let n be the number of vertices. We study the case where the graph is sparse, i.e. $m = n^{1+o(1)}$. Let the total weighted size of all vertices be N. For LWS_C and other problems without graphs, we use N as the input size. For LWSP_C and other problems on graphs, we use $M = \max(m, N)$ as the size of the input.

In this paper we will see that if we can improve the algorithm for $\mathsf{StaticLWS}_C$ to $N^{2-o(1)}$, then on some classes of graphs we can solve LWSP_C faster than $M^{2-o(1)}$ time.

1.2 Fine-grained complexity preliminaries

Fine-grained complexity studies the exact-time reductions between problems, and the completeness of problems in classes under exact-time reductions. These reductions have established conditional lower bounds for many interesting problems. The Orthogonal Vectors problem (OV) is a well-studied problem solvable in quadratic time. If the *Strong Exponential Time Hypothesis (SETH)* [31, 32] is true, then OV does not have truly subquadratic time algorithms [47]. The problem OV is defined as follows: Given n boolean vectors of dimension $d = \omega(\log n)$, and decide whether there is a pair of vectors whose inner product is zero. The best algorithm is in time $n^{2-\Omega(1/\log(d/\log n))}$ [7, 23]. The *Moderate-dimension OV conjecture (MDOVC)* states that for all $\epsilon > 0$, there are no $O(n^{2-\epsilon}\mathsf{poly}(d))$ time algorithms that solve OV with vector dimension d. If this conjecture is true, then many interesting problems would get lower bounds, including dynamic programming problems such as Longest Common Subsequence [2, 20], Edit Distance [14, 5], Fréchet distance [18, 21, 22], Local Alignment [9], CFG Parsing and RNA Folding [1], Regular Expression Matching [15, 19] , and also many graph problems [42, 8, 16]. There are also conditional hardness results about graph problems based on the hardness of All Pair Shortest Path [49, 4, 10, 39] and 3SUM [6, 35].

The *fine-grained reduction* was introduced in [49], which can preserve polynomial saving factors in the running time between problems. The statements for fine-grained complexity are usually like this: if there is some $\epsilon_2 > 0$ such that problem Π_2 of input size n is in $\mathsf{TIME}((T_2(n))^{1-\epsilon_2})$, then problem Π_1 of input size n is in $\mathsf{TIME}((T_1(n))^{1-\epsilon_1})$ for some ϵ_1. If T_1 and T_2 are both $O(n^2)$ then this reduction is called a subquadratic reduction. Furthermore, the *exact-complexity reduction* is a more strict version that can preserve sub-polynomial savings factors between problems. We use $(\Pi_1, T_1(n)) \leq_{\text{EC}} (\Pi_2, T_2(n))$ to denote that there is a reduction from problem Π_1 to problem Π_2 so that if problem Π_2 is in $\mathsf{TIME}(T_2(n))$, then problem Π_1 is in $\mathsf{TIME}(T_1(n))$.

1.3 Introducing reachability to first-order model checking

Similar to extending LWS_C to paths in graphs, introducing transitive closure to first-order logic also which makes parallel problems become sequential. The first-order property (or first-order model checking) problem is to decide whether an input structure satisfies a fixed first-order logic formula φ. Although model checking for input formulas is PSPACE-complete [44, 45], when φ is fixed by the problem, it is solvable in polynomial time. We consider the class of problems where each problem is the model checking for a fixed formula φ. The sparse version of OV [27] is one of these problems, defined by the formula $\exists u \exists v \forall i \in [d](\neg One(u,i) \vee (\neg One(v,i)))$, where relation $One(u,i)$ is true iff the i-th coordinate of vector u is one.

If φ has k quantifiers ($k \geq 2$), then on input structures of n elements and m tuples of relations, it can be solved in time $O(n^{k-2}m)$ [28]. On dense graphs where $k \geq 9$, it can be solved in time $O(n^{k-3+\omega})$, where ω is the matrix multiplication exponent [48]. Here we study the case where the input structure is sparse, i.e. $m = n^{1+o(1)}$, and ask whether a three-quantifier first-order formula can be model checked in time faster than $m^{2-o(1)}$. The *first-order property conjecture (FOPC)* states that there exists integer $k \geq 2$, so that first-order model checking for $(k+1)$-quantifier formulas cannot be solved in time $O(m^{k-\epsilon})$ for any $\epsilon > 0$. This conjecture is equivalent to MDOVC, since OV is proven to be a complete problem in the class of first-order model checking problems; in other words, any model checking problem of 3 quantifier formulas on sparse graphs is subquadratic time reducible to OV [28]. This means from improved algorithms for OV we can get improved algorithms for first-order model checking.

The first-order property problems are highly parallelizable. If we introduce the transitive closure (TC) operation on the relations, then these problems will become sequential. The transitive closure of a binary relation E can be considered as the reachability relation by edges of E in a graph. In a sparse structure, the TC of a relation may be dense. So it can be considered as a dense relation succinctly described in the input. In finite model theory, adding transitive closure significantly adds to the expressive power of first-order logic (First discovered by Fagin in 1974 according to [37], and then re-discovered by [12].) In fine-grained complexity, adding arbitrary transitive closure operations on the formulas strictly increases the hardness of the model checking problem. More precisely, [27] shows that SETH on constant depth circuits, which is a weaker conjecture than the SETH (which concerns k-CNF-SAT), implies the model checking for two-quantifier first-order formulas with transitive closure operations cannot be solved in time $O(m^{2-\epsilon})$ for any $\epsilon > 0$. This means this problem may stay hard even if the SETH on k-CNF-SAT is refuted.

However, we will see that for a class of three-quantifier formulas with transitive closure, model checking is no harder than OV under subquadratic time reductions.

We define problem $\mathsf{Selection}_P$ to be the decision problem for whether an input structure satisfies $(\exists x \in X)(\exists y \in Y)P(x,y)$. $P(x,y)$ is a fixed property specified by the problem that can be decided in time $O(w(x)+w(y))$, where weighted size $w(x)$ is the size of additional data on element x. For example, OV is $\mathsf{Selection}_P$ where $P(x,y)$ iff x and y are a pair of orthogonal vectors. In this case $w(x)$ is defined as the length of vector x. (If we work on the sparse version of OV, the weighted size $w(x)$ is defined by the Hamming weight of x.)

On a directed graph $G = (V, E)$, we define Path_P to be the problem of deciding whether $(\exists x \in V)(\exists y \in V)[\mathrm{TC}_E(x, y) \wedge P(x, y)]$, where TC_E is the transitive closure of relation E and $P(x, y)$ is a property on x, y fixed by the problem. That is, whether there exist two vertices x,y not only satisfying property P but also y is reachable from x by edges in E. We will give an example of Path_P in Appendix B. Also, we define $\mathsf{ListPath}_P$ to be the problem of listing all $x \in V$ such that $(\exists y \in V)[\mathrm{TC}_E(x, y) \wedge P(x, y)]$.

Considering the model checking problems, we let PathFO_3 and $\mathit{ListPathFO}_3$ denote the class of Path_P and $\mathsf{ListPath}_P$ such that P is of form $\exists z\psi(x, y, z)$ or $\forall z\psi(x, y, z)$, where ψ is a quantifier-free formula in first-order logic. Later we will see that problems in PathFO_3 and $\mathit{ListPathFO}_3$ are no harder than OV. In these model checking problems, the weighted size of an element is the number of tuples in the input structure that the element is contained in.

Trivially, $\mathsf{Selection}_P$ on input size (N_1, N_2) can be decided in time $O(N_1 N_2)$, where N_1 is the total weighted size of elements in X, and N_2 is the total weighted size of elements in Y. Path_P and $\mathsf{ListPath}_P$ on input size M and total vertex weighted size N are solvable time $O(MN)$ by depth/breadth first search from each vertex, where M is defined to be the maximum of N and the number of edges m. This paper will show that on some graphs, if $\mathsf{Selection}_P$ is in truly subquadratic time, so is Path_P and $\mathsf{ListPath}_P$. Interestingly, by applying the same reduction techniques from Path_P to $\mathsf{Selection}_P$, we can get a similar reduction from a dynamic programming problem on a graph to a static problem.

1.4 Main results

This paper works on two classes of graphs, both having some similarities to trees. The first class is where the graph G is a multitree. A *multitree* is a directed acyclic graph where the set of vertices reachable from any vertex form a tree. Or equivalently a DAG is a multitree if and only if on all pairs of vertices u, v, there is at most one path from u to v. In different contexts, multitrees are also called *strongly unambiguous graphs*, *mangroves* or *diamond-free posets* [29]. These graphs can be used to model computational paths in nondeterministic algorithms where there is at most one path connecting any two states [13]. The butterfly network, which is a widely-used model of the network topology in parallel computing, is an example of multitrees. We also work on multitrees of strongly connected component, which is a graph that when each strongly connected components are replaced by a single vertex, the graph becomes a multitree.

The second class of graphs is when we treat G as undirected by replacing all directed edges by undirected edges, the underlying graph has constant treewidth. *Treewidth* [40, 41] is an important parameter of graphs that describes how similar they are to trees. [1] On these classes of graphs, we have the following theorems.

▶ **Theorem 1** (Reductions between decision problems.). *Let $t(M) \geq 2^{\Omega(\sqrt{\log M})}$, and let the graph $G = (V, E)$ satisfy one of the following conditions:*

- *G is a multitree, or*
- *G is a multitree of strongly connected components, or*
- *The underlying undirected graph of G has constant treewidth,*

then, the following statements are true:

[1] Here we consider the undirected treewidth, where both the graph and the decomposition tree are undirected. It is different from *directed treewidth* defined for directed graphs by [33].

- *If* $\mathsf{Selection}_P$ *is in time* $N_1N_2/t(\min(N_1, N_2))$*, then* Path_P *is in time* $M^2/t(\mathsf{poly}M)$.[2]
- *If* Path_P *is in time* $M^2/t(M)$*, then* $\mathsf{ListPath}_P$ *is in time* $M^2/t(\mathsf{poly}M)$.
- *When* $P(x, y)$ *is of form* $\exists z\psi(x, y, z)$ *or* $\forall z\psi(x, y, z)$ *where* ψ *is a quantifier-free first-order formula,* $\mathsf{Selection}_P$ *is in time* $N_1N_2/t(\min(N_1, N_2))$ *iff* Path_P *is in time* $M^2/t(\mathsf{poly}M)$ *iff* $\mathsf{ListPath}_P$ *is in time* $M^2/t(\mathsf{poly}M)$.

This theorem implies that OV is hard for classes $PathFO_3$ and $ListPathFO_3$. By the improved algorithm for OV [7, 23], we get improved algorithms for $PathFO_3$ and $ListPathFO_3$:

► **Corollary 2** (Improved algorithms.). *Let the graph* G *be a multitree, or multitree of strongly connected components, or a DAG whose underlying undirected graph has constant treewidth. Then* $PathFO_3$ *and* $ListPathFO_3$ *are in time* $M^2/2^{\Omega(\sqrt{\log M})}$.

Next, we consider the dynamic programming problems. If the cost matrix C in LWSP_C is succinctly describable, we get the following reduction from LWSP_C to $\mathsf{StaticLWS}_C$.

► **Theorem 3** (Reductions between optimization problems.). *On a multitree graph, or a DAG whose underlying undirected graph has constant treewidth, let* $t(N) \geq 2^{\Omega(\sqrt{\log N})}$*, then,*

1. *if* $\mathsf{StaticLWS}_C$ *of input size* N *is in time* $N^2/t(N)$*, then* LWSP_C *on input size* M *is in time* $M^2/t(\mathsf{poly}(M))$.
2. *if* LWSP_C *is in time* $M^2/t(M)$*, then* LWS_C *is in time* $N^2/t(\mathsf{poly}(N))$.

If there is a reduction from a concrete $\mathsf{StaticLWS}_C$ problem to its corresponding LWS_C problem (e.g. there are reductions from MinInnerProduct to LowRankLWS, from VectorDomination to NestedBoxes and from OV to LongestSubsetChain [36]), then the corresponding LWS_C, $\mathsf{StaticLWS}_C$ and LWSP_C problems are subquadratic-time equivalent. From the algorithm for OV [23] and SparseOV [28], we get improved algorithm for problem LongestSubsetChain:

► **Corollary 4** (Improved algorithm). *On a multitree or a DAG whose underlying undirected graph has constant treewidth,* LongestSubsetChain *is in time* $M^2/2^{\Omega(\sqrt{\log M})}$.

The reduction uses a technique that decomposes multitrees into sub-structures where it is easy to decide whether vertices are reachable. So we also get reachability oracles using subquadratic space, that can answer reachability queries in sublinear time.

► **Theorem 5** (Reachability oracle). *On a multitree of strongly connected components, there exists a reachability oracle with subquadratic preprocessing time and space that has sublinear query time. On a multitree, the preprocessing time and space is* $O(m^{5/3})$*, and the query time is* $O(m^{2/3})$.

1.5 Organization

In Section 2 we prove the first part of Theorem 1, by reduction from Path_P to $\mathsf{Selection}_P$ on multitrees. The case for bounded treewidth DAGs will be presented in Appendix D. Section 3 proves Theorem 3 by presenting a reduction from LWSP_C to $\mathsf{StaticLWS}_C$, and the proof of correctness will be left to Appendix E. Section 4 discusses about open problems. Appendix A lists the definitions of problems, and Appendix B shows some concrete problems

[2] This reduction also applies to optimization versions of these two problems. Let Path_F be a problem to compute $\min_{x,y\in V, x\rightsquigarrow y} F(x, y)$ and $\mathsf{Selection}_F$ be a problem to compute $\min_{x\in X, y\in Y} F(x, y)$, where F is a function on x, y, instead of a boolean property. Then the same technique gives us a reduction from Path_F to $\mathsf{Selection}_F$.

as examples. Appendix C gives a weighted version of Lemma 7. Appendix F proves the second part of Theorem 1 by reduction from $\mathsf{ListPath}_P$ to Path_P. Appendix G proves the last part of Theorem 1, the subquadratic equivalence of $\mathsf{Selection}_P$, Path_P and $\mathsf{ListPath}_P$ when P is a first-order property. Appendix H talks about the reachability oracle for multitrees.

2 From sequential problems to parallel problems, on multitrees

We will prove the first part of Theorem 1 by showing that if $t(M) \geq 2^{\Omega(\sqrt{\log M})}$, then $(\mathsf{Path}_P, M^2/t(\mathsf{poly} M)) \leq_{\mathrm{EC}} (\mathsf{Selection}_P, N_1N_2/t(\min(N_1, N_2)))$. This section gives the reduction for multitrees and multitrees of strongly connected components. For constant treewidth graphs, the reduction will be shown in Appendix D.

2.1 The recursive algorithm

The algorithm uses a divide-and-conquer strategy. We will consider each strongly connected component as a single vertex, whose weighted size equals the total weighted size of the component. In the following algorithm, whenever querying $\mathsf{Selection}_P$ or exhaustively enumerating pairs of reachable vertices and testing P on them, we can extract all the vertices from a strongly connected component. Thus we will be working on a multitree, instead of a multitree of strongly connected components. Testing P on a pair of vertices (or strongly connected components) of total weighted sizes N_1, N_2 is in time $O(N_1N_2)$.

Let $\mathsf{CutPath}_P$ be a variation of Path_P. It is the property testing problem for $(\exists x \in S)(\exists y \in T)[TC_E(x, y) \wedge \varphi(x, y)]$, where (S, T) is a cut in the graph, such that all the edges between S and T are directed from S to T. $\mathsf{CutPath}_P$ on input size M and total vertex weighted size N can be solved in time $O(MN)$ if $P(x, y)$ is decidable in time $O(w(x) + w(y))$: start from each vertex and do depth/breadth first search, and on each pair of reachable vertices decide if P is satisfied.

▶ **Lemma 6.** *For $t(M) \geq 2^{\Omega(\sqrt{\log M})}$, if $\mathsf{Selection}_P(N, N)$ is in time $N^2/t(N)$ and $\mathsf{CutPath}_P(M)$ is in time $M^2/t(M)$, then $\mathsf{Path}_P(M)$ is in time $M^2/t(\mathsf{poly}(M))$.*

Proof. Let γ be a constant satisfying $0 < \gamma \leq 1/4$. Let $T_\Pi(M)$ be the running time of problem Π on a structure of total weighted size M. We show that there exists a constant c where $0 < c < 1$ so that if $T_{\mathsf{Path}_P}(M')$ is at most $M'^2/t(M'^c)$ for all $M' < M$, then $T_{\mathsf{Path}_P}(M) \leq M^2/t(M^c)$. We run the recursive algorithm as shown in Algorithm 1. The intuition is to divide the graph into a cut S, T, recursively compute Path_P on S and T, and deal with paths from S to T.

It would be good if the difference of total weighted sizes between S and T is at most M^γ. Otherwise, it means by the topological order, there is a vertex of weighted size at least M^γ in the middle, adding it to either S or T would make the size difference between S and T exceed M^γ. In this case, we use letter x to denote the vertex. We will deal with x separately. We temporarily set aside the time of recursively running $\mathsf{Selection}_P$ on x (when x is shrunk from a strongly connected component) in all the recursive calls, and consider the rest of the running time.

Let M_S and M_T be the sizes of sets S and T respectively. Without loss of generality,

Algorithm 1: $\mathsf{Path}_P(G)$ on a DAG

// Reducing Path_P to $\mathsf{Selection}_P$ and $\mathsf{CutPath}_P$

if G has only one vertex **then return** false.
Let M be the weighted size of the problem.
Topological sort all vertices.
Keep adding vertices to S by topological order, until the total weighted size of S exceeds $M/2$. Let the rest of vertices be T.
if $|S| - |T| > M^\gamma$ **then**
  Let x be the last vertex added to S. Remove x from S.
Run Path_P on the subgraph induced by S.
Run $\mathsf{CutPath}_P(S, T)$.
if x exists **then**
  Run $\mathsf{CutPath}_P(S, x)$.
  If x is originally a strongly connected component, run $\mathsf{Selection}_P$ on it.
  Run $\mathsf{CutPath}_P(x, T)$
Run Path_P on the subgraph induced by T.
if any one of the above three calls returns true **then return** true.

assume $M_S \geq M_T$, and let $\Delta = M_S - M_T$, which is at most M^γ. Then we have

$$\begin{aligned}
T_{\mathsf{Path}_P}(M) &= T_{\mathsf{Path}_P}(M_S) + T_{\mathsf{Path}_P}(M_T) + 3T_{\mathsf{CutPath}_P}(M) + O(M) \\
&= T_{\mathsf{Path}_P}(M_T + \Delta) + T_{\mathsf{Path}_P}(M_T) + 3T_{\mathsf{CutPath}_P}(M) + O(M) \\
&\leq 2T_{\mathsf{Path}_P}(M/2 + \Delta) + 3T_{\mathsf{CutPath}_P}(M) + O(M) \\
&= 2(M/2 + \Delta)^2/t((M/2 + \Delta)^c) + 3M^2/t(M) + O(M).
\end{aligned}$$

Because $t(M) < M$ and is monotonically growing, The term $3M^2/t(M) + O(M)$ is bounded by $4M^2/t(M) \leq 16(M/2)^2/t(M) \leq 16(M/2 + \Delta)^2/t((M/2 + \Delta)^c)$. Thus the above formula is bounded $18(M/2 + \Delta)^2/t((M/2 + \Delta)^c)$. By picking small enough constant γ and c, this sum is less than $M^2/t(M^c)$.

For the time of running $\mathsf{Selection}_P$ on x where x is originally a strongly connected component, we consider all recursive calls of Path_P. Let the size of each such x be M_i. The total time would be $\sum_i M_i^2/t(M_i) < (\sum_i M_i^2)/t(M^\gamma)$. Because $\sum_i M_i \leq M$, the sum is at most $M^2/t(M^\gamma)$, a value subquadratic to M, with M being the input size of the outermost call of Path_P. ◀

2.2 A special case that can be exhaustively searched

The following lemma shows that if no vertex has both a lot of ancestors and a lot of descendants, then the total number of reachable pairs of vertices is subquadratic to m. This lemma holds for any DAG, not just for multitrees. We will use this lemma in the next subsection to show that in a subgraph where all vertices have few ancestors and descendants, we can test property P on all pairs of reachable vertices by brute force. Actually, we will use a weighted version of this lemma, which will be proved in Appendix C.

▶ **Lemma 7.** *If in a DAG $G = (V, E)$ of m edges, every vertex has either at most n_1 ancestors or at most n_2 descendants, then there are at most $(m \cdot n_1 \cdot n_2)$ pairs of vertices s, t such that s can reach t.*

In a DAG $G = (V, E)$ of m edges, let S, T be two disjoint sets of vertices where edges between S and T only direct from S to T. If every vertex has either at most n_1 ancestors in S or at most n_2 descendants in T, then there are at most $(m \cdot n_1 \cdot n_2)$ pairs of vertices $s \in S$ and $t \in T$ such that s can reach t.

Proof. We define the ancestors of an edge $e \in E$ to be the ancestors (or ancestors in S) of its incoming vertex, and its descendants to be the descendants (or descendants in T) of its outgoing vertex. Let the number of its ancestors and descendants be denoted by $anc(e)$ and $des(e)$ respectively.

For each edge e, it belongs to exactly one of the following three types:

Type A: If $anc(e) \leq n_1$ but $des(e) > n_2$, then let $count(e)$ be $anc(e)$.

Type B: If $des(e) \leq n_2$ but $anc(e) > n_1$, then let $count(e)$ be $des(e)$.

Type C: If $anc(e) \leq n_1$ and $des(e) \leq n_2$, then let $count(e)$ be $anc(e) \cdot des(e)$.

$\sum_{e \in E} count(e) \leq m \cdot n_1 \cdot n_2$ because the *count* value on each edge is bounded by $n_1 \cdot n_2$. We will prove that this value upper bounds the number of reachable pairs of vertices.

For each pair of reachable vertices (u, v) (or (u, v) s.t. $u \in S$ and $v \in T$), let $(e_1, \ldots, e_p)$ be the path from u to v. Along the path, *anc* does not decrease, and *dec* does not increase. A path belongs to exactly one of the following three types:

Type a: Along the path $anc(e_1) \leq anc(e_2) \leq \cdots \leq anc(e_p) \leq n_1$, and $des(e_1) \geq des(e_2) \geq \cdots \geq des(e_p) > n_2$. That is, all the edges are Type A.

Type b: Along the path $des(e_p) \leq des(e_{p-1}) \leq \cdots \leq des(e_1) \leq n_2$, and $anc(e_p) \geq anc(e_{p-1}) \geq \cdots \geq anc(e_1) > n_1$. That is, all the edges are Type B.

Type c: Along the path there is some edge e_i so that $anc(e_i) \leq n_1$ and $des(e_i) \leq n_2$. That is, it has at least one Type C edge.

There will not be other cases, for otherwise if a Type A edge directly connects to a Type B edge without a Type C edge in the middle, then the vertex joining these two edges would have more than n_1 ancestors and more than n_2 descendants.

If a path from u to v is Type a, then its last edge e_p is Type A. If it is Type b, then its first edge e_1 is Type B. If it is Type c, then there is some edge e_i in the path that is Type C. This means:

1. For each Type A edge e, $count(e)$ is at least the number of all Type a pairs (u, v) whose path has e as its last edge.
2. For each Type B edge e, $count(e)$ is at least the number of all Type b pairs (u, v) whose path has e as its first edge.
3. For each Type C edge e, $count(e)$ is at least the number of all Type c pairs (u, v) whose path contains e.

Therefore each path is counted at least once by the $count(e)$ of some edge e. ◀

2.3 Subroutine: reachability across a cut

Now we will show the reduction from $\mathsf{CutPath}_P$ to $\mathsf{Selection}_P$. The high level idea of $\mathsf{CutPath}_P$ is that we think of the reachability relation on $S \times T$ as an $|S| \times |T|$ boolean matrix whose one-entries correspond to reachable pairs of vertices. If we could partition the matrix into all-one combinatorial rectangles, then we can decide all entries within these rectangles by a query to $\mathsf{Selection}_P$, because in the same rectangle, all pairs are reachable.

▷ Claim 8. Consider the reachability matrix of on sets S and T. Let M_S and M_T be the sizes of S and T. If there is a way to partition the matrix into non-overlapping combinatorial rectangles $(S_1, T_1), \ldots, (S_k, T_k)$ of sizes $(r_1, c_1), \ldots, (r_k, c_k)$, and if there is some t so that

Algorithm 2: $\mathsf{CutPath}_P(S,T)$ on a multitree

Compute the total weighted size of ancestors $anc(v)$ and descendants $des(v)$ for all vertices.
Insert all vertices with at least M^α ancestors and M^α descendants into linked list L.
while there exists a vertex $v \in L$ **do**
  // we call v a *pivot vertex*
  Let A be the set of ancestors of v in S.
  Let B be the set of descendants of v in T.
  Add v to A if $v \in S$, otherwise add v to B.
  Run $\mathsf{Selection}_P$ on (A,B). If it returns true then **return** true.
  for each $a \in A$ **do**
    let $des(a) = des(a) - |B|$.
    if $des(a) < M^\alpha$ and $a \in L$ **then** remove a from L.
  for each $b \in B$ **do**
    let $anc(b) = anc(b) - |A|$.
    if $anc(b) < M^\alpha$ and $b \in L$ **then** remove b from L.
  Remove v from the graph.
for each edge (s,t) crossing the cut(S,T) **do**
  Let A be the set of ancestors of s (including s) in S.
  Let B be the set of descendants of t (including t) in T.
  On all pairs of vertices (a,b) where $a \in A, b \in B$, check property P. If P is true on any pair of (a,b) then **return** true.

computing each subproblem of size (r_i, c_i) takes time $r_i \cdot c_i / t(\min(r_i, c_i))$, and all $r_i \geq \ell$, and all $c_i \geq \ell$ for a threshold value ℓ, then all the computation takes total time $O(M_S \cdot M_T / t(\ell))$.

Proof. Let the minimum of all r_i be r_{min} and the minimum of all c_i be c_{min}. Then the factor of time saved for computing each combinatorial rectangle is at least $t(\min(r_{min}, c_{min}))$, greater than $t(\ell)$. So the time spent on all rectangles is at most $O((\sum_{i=1}^t c_i)(\sum_{i=1}^t r_i)/t(\ell))$, also we have $(\sum_{i=1}^t c_i)(\sum_{i=1}^t r_i) \leq M_S \cdot M_T$ because the rectangles are contained inside the matrix of size $M_S \cdot M_T$ and they do not overlap. So the total time is $O(M_S \cdot M_T / t(\ell))$. ◀

The algorithm $\mathsf{CutPath}_P(S,T)$ is shown in Algorithm 2. It tries to cover the one-entries of the reachability matrix by combinatorial rectangles as many as possible. Finally, for the one-entries not covered, we go through them by exhaustive search, which takes less than quadratic time.

In the beginning, we can compute the total weighted size of ancestors (or descendants) of all vertices in the DAG in $O(M)$ time by going through all vertices by topological order (or reversed topological order).

In each query to $\mathsf{Selection}_P(A,B)$, all vertices in A can reach all vertices in B, because they all go through v. For any pair of reachable vertices $s \in S, t \in T$, if they go through any pivot vertex, then the pair is queried to $\mathsf{Selection}_P$. Otherwise it is left to the end, and checked by exhaustive search on all pairs of reachable vertices.

The calls to $\mathsf{Selection}_P$ correspond to non-overlapping all-one combinatorial rectangles in the reachability matrix. This is because the graph G is a multitree. For each call to $\mathsf{Selection}_P$, the rectangle size is at least $M^\alpha \times M^\alpha$. Thus the total time for all the $\mathsf{Selection}_P$ calls is $O(M^2/t(M^\alpha))$ by Claim 8.

Each time we remove a pivot vertex v, there will be no more paths from set A to set B, for otherwise there would be two distinct paths connecting the same pair of vertices. Thus, removing a v decreases the total number of weighted-pairs[3] of reachable vertices by at least $M^\alpha \times M^\alpha$. There are $M \times M$ weighted-pairs of vertices, so the total weight (and thus the total number) of pivot vertices like v is at most $(M \times M)/(M^\alpha \times M^\alpha) = M^{2-2\alpha}$.

Each time we find a pivot vertex v, we update the total weighted size of descendants for all its ancestors, and update the total weighted size of ancestors for all its descendants. Because it has at least M^α ancestors and M^α descendants, the value decrease on each affected vertex is at least M^α. So each vertex has decreased its ancestors/descendants values for at most $M/M^\alpha = M^{1-\alpha}$ times. In other words, each vertex can be an ancestor/descendant of at most $M^{1-\alpha}$ pivot vertices. The total time to deal with all ancestors/descendants of all pivot vertices in the while loop is in $O(M \cdot M^{1-\alpha}) = O(M^{2-\alpha})$.

Finally, after the while loop, there are no vertices with both more than M^α ancestors and M^α descendants. In this case, by a weighted version of Lemma 7 (See Appendix C), the number of weighted-pairs of reachable vertices is bounded by $M \cdot M^\alpha \cdot M^\alpha = M^{1+2\alpha}$. So the total time to deal with these paths is $O(M^{1+2\alpha})$.

Thus the total running time is $O(M^2/t(M^\alpha) + M^{2-\alpha} + M^{1+2\alpha})$. By choosing α and γ to be appropriate constants, we get subquadratic running time.

If $t(M) = M^\epsilon$, then by choosing $\alpha = 1/(2+\epsilon)$, we get running time $M^{2-\epsilon/(2+\epsilon)}$.

3 Application to Least Weight Subpath

In this section we will prove Theorem 3. The reduction from LWSP_C to $\mathsf{StaticLWS}_C$ uses the same structure as the reduction from Path_P to $\mathsf{Selection}_P$ in the proof of Theorem 1 shown in Section 2. Because in LWSP we only consider DAGs, there are no strongly connected components in the graph.

Process $\mathsf{LWSP}_C(G, F_0)$ computes values of F on initial values F_0 defined on all vertices of G. On a given LWSP_C problem, we will reduce it to an asymmetric variation of $\mathsf{StaticLWS}_C$. Process $\mathsf{StaticLWS}_C(A, B, F_A)$ computes all the values of function F_B defined on domain B, given all the values of F_A defined on domain A, such that $F_B(b) = \min_{a \in A}[F_A(s) + c_{a,b}]$. Let N_A and N_B be the total weighted size of A and B respectively. It is easy to see that if $\mathsf{StaticLWS}_C$ on $|N_A| = |N_B|$ is in time $N_A^2/t(N_A)$, then $\mathsf{StaticLWS}_C$ on general A, B is in time $O(N_A \cdot N_B/t(\min(N_A, N_B)))$.

We also define process $\mathsf{CutLWSP}_C(S, T, F_S)$, which computes all the values of F_T defined on domain T, given all the values of F_S on domain S, where $F_T(t) = \min_{s \in S, s \rightsquigarrow t}[F_S(s) + c_{s,t}]$.

The reduction algorithm is adapted from the reduction from Path_P to $\mathsf{Selection}_P$. LWSP_C is analogous to Path_P, $\mathsf{StaticLWS}_C$ is analogous to $\mathsf{Selection}_P$, and $\mathsf{CutLWSP}_C$ is analogous to $\mathsf{CutPath}_P$. In Path_P, we divide the graph into two halves, recursively call Path_P on the subgraphs, and use $\mathsf{CutPath}_P$ to deal with paths from one side of the graph to the other side. Similarly in LWSP_C, we divide the graph into two halves, recursively compute function F on the source side of the graph, then based on these values we call $\mathsf{CutPath}_P$ to compute the initial values of function F on the sink side of the graph, and finally we recursively call LWSP_C on the sink side of the graph. In $\mathsf{CutPath}_P$, we first identify large all-one rectangles in the reachability matrix, and then use $\mathsf{Selection}_P$ to solve them, and finally we go through all reachable pairs of vertices that are not covered by these rectangles. Similarly, in LWSP_C,

[3] The number of weighted-pairs is defined to be the sum of $w(u) \cdot w(v)$ for all pairs of reachable vertices $u \rightsquigarrow v$.

Algorithm 3: $\mathsf{LWSP}_C(G=(V,E,V_0),F_0)$ on a DAG

if G has only one vertex v **then**
  if $v \in V_0$ **then**
    return $\min(0, F_0(v))$.
  return F_0 on v.
Let M be the weighted size of the problem.
Topological sort all vertices.
Keep adding vertices to S by topological order, until the total weighted size of S exceeds $M/2$. Let the rest of vertices be T.
if $|S|-|T| > M^\gamma$ **then**
  Let x be the last vertex added to S. Remove x from S.
Compute F on domain S, by $F \leftarrow \mathsf{LWSP}_C(G_S, F_0)$, where G_S is the subgraph of G induced by S.
Let $F_T \leftarrow \mathsf{CutLWSP}_C(S,T,F)$.
For each vertex $t \in T$, let $F_0(t) \leftarrow \min(F_0(t), F_T(t))$.
if x exists **then**
  Compute $F_x \leftarrow \mathsf{CutLWSP}_C(S,x,F)$ for vertex x.
  Compute F on vertex x by $F(x) \leftarrow \min(F_0(x), F_x(x))$.
  Let $F'_T \leftarrow \mathsf{CutLWSP}_C(x,T,F)$.
  For each vertex $t \in T$, let $F_0(t) \leftarrow \min(F_0(t), F'_T(t))$.
Compute F on domain T, by $F \leftarrow \mathsf{LWSP}_C(G_T, F_0)$, where G_T is the subgraph of G induced by T.
return F on domain V.

we will use the similar method to identify large all-one rectangles in the reachability matrix and use $\mathsf{StaticLWS}_C$ to solve them, and finally we go through all reachable pairs of vertices and update F on each of them.

The algorithm LWSP_C is similar as Path_P (Algorithm 1), and is defined in Algorithm 3. Initially, we let $F(v) \leftarrow 0$ for all $v \in V_0$, and let $F(v) \leftarrow +\infty$ for all $v \notin V_0$. We run $\mathsf{LWSP}_C(G, F_0)$ on the whole graph.

The algorithm $\mathsf{CutLWSP}_C(S,T,F_S)$ is adapted from $\mathsf{CutPath}_P$ (Algorithm 2), with the following changes:

1. In the beginning, $F_T(t)$ is initialized to ∞ for all $t \in T$.
2. Each query to $\mathsf{Selection}_P(A,B)$ in $\mathsf{CutPath}_P$ is replaced by
   a. Compute F_B on domain B by $\mathsf{StaticLWS}_C(A,B,F_S)$.
   b. For each vertex b in B, let $F_T(b)$ be the minimum of the original $F_T(b)$ and $F_B(b)$.
3. Whenever processing a pair of vertices s,t such that s is can reach t in either the preprocessing phase or the final exhaustive search phase, we let $F_T(t) \leftarrow F_S(s)+c_{s,t}$ if $F_S(s)+c_{s,t} < F_T(t)$.
4. In the end, the process returns F_T, the target function on domain T.

The proof of correctness will be shown in Appendix E. The time complexity of this reduction algorithm follows from the argument of Section 2.

4 Open problems

One open problem is to study Path_P and LWSP_C on general DAGs. Also, we would like to consider the case where the graph is not sparse, where we can use $O(MN)$ as the baseline time complexity instead of $O(M^2)$.

It would also be desirable to study the fine-grained complexity of the DAG versions of other quadratic time solvable dynamic programming problems, e.g. the Longest Common Subsequence problem.

References

1 Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann. Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve. In *Foundations of Computer Science (FOCS), 2017 IEEE 58th Annual Symposium on*, pages 192–203. IEEE, 2017.

2 Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. Tight hardness results for lcs and other sequence similarity measures. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 59–78. IEEE, 2015.

3 Amir Abboud, Karl Bringmann, Danny Hermelin, and Dvir Shabtay. SETH-based lower bounds for subset sum and bicriteria path. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 41–57. SIAM, 2019.

4 Amir Abboud, Fabrizio Grandoni, and Virginia Vassilevska Williams. Subcubic equivalences between graph centrality problems, APSP and diameter. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 1681–1697. SIAM, 2014.

5 Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 375–388. ACM, 2016.

6 Amir Abboud and Kevin Lewi. Exact weight subgraphs and the k-sum conjecture. In *International Colloquium on Automata, Languages, and Programming*, pages 1–12. Springer, 2013.

7 Amir Abboud, Ryan Williams, and Huacheng Yu. More applications of the polynomial method to algorithm design. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 218–230. SIAM, 2015.

8 Amir Abboud, Virginia Vassilevska Williams, and Joshua Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete Algorithms*, pages 377–391. SIAM, 2016.

9 Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *International Colloquium on Automata, Languages, and Programming*, pages 39–51. Springer, 2014.

10 Udit Agarwal and Vijaya Ramachandran. Fine-grained complexity for sparse graphs. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 239–252, New York, NY, USA, 2018. ACM. `doi:10.1145/3188745.3188888`.

11 Alok Aggarwal, Baruch Schieber, and Takeshi Tokuyama. Finding a minimum-weight k-link path in graphs with the concave monge property and applications. *Discrete & Computational Geometry*, 12(3):263–280, 1994.

12 Alfred V Aho and Jeffrey D Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 110–119. ACM, 1979.

13 Eric Allender and Klaus-Jörn Lange. $StUSPACE(\log n) \subseteq DSPACE(\log^2 n/\log\log n)$. In *International Symposium on Algorithms and Computation*, pages 193–202. Springer, 1996.

14 Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 51–58. ACM, 2015.

15 Arturs Backurs and Piotr Indyk. Which regular expression patterns are hard to match? In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 457–466. IEEE, 2016.

16 Arturs Backurs, Liam Roditty, Gilad Segal, Virginia Vassilevska Williams, and Nicole Wein. Towards tight approximation bounds for graph diameter and eccentricities. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 267–280. ACM, 2018.

17 Hans L Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

18 Karl Bringmann. Why walking the dog takes time: Fréchet distance has no strongly subquadratic algorithms unless SETH fails. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 661–670. IEEE, 2014.

19 Karl Bringmann, Allan Grønlund, and Kasper Green Larsen. A dichotomy for regular expression membership testing. In *Foundations of Computer Science (FOCS), 2017 IEEE 58th Annual Symposium on*, pages 307–318. IEEE, 2017.

20 Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 79–97. IEEE, 2015.

21 Karl Bringmann and Marvin Künnemann. Improved approximation for Fréchet distance on c-packed curves matching conditional lower bounds. *International Journal of Computational Geometry & Applications*, 27(01n02):85–119, 2017.

22 Karl Bringmann and Wolfgang Mulzer. Approximability of the discrete Fréchet distance. *Journal of Computational Geometry*, 7(2):46–76, 2015.

23 Timothy M Chan and Ryan Williams. Deterministic APSP, Orthogonal Vectors, and More: Quickly derandomizing Razborov-Smolensky. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1246–1255. SIAM, 2016.

24 S.C. Chen, J.Y. Wu, G.S. Huang, and R.C.T. Lee. Finding a longest increasing subsequence on a galled tree. In *the 28th Workshop on Combinatorial Mathematics and Computation Theory, Penghu, Taiwan*, 2011.

25 Michael L Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

26 Zvi Galil and Kunsoo Park. A linear-time algorithm for concave one-dimensional dynamic programming. 1989.

27 Jiawei Gao and Russell Impagliazzo. The fine-grained complexity of strengthenings of first-order logic. *Electronic Colloquium on Computational Complexity (ECCC)*, 26:9, 2019. URL: `https://eccc.weizmann.ac.il/report/2019/009`.

28 Jiawei Gao, Russell Impagliazzo, Antonina Kolokolova, and Ryan Williams. Completeness for first-order properties on sparse structures with algorithmic applications. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 2162–2181, 2017.

29 Jerrold R Griggs, Wei-Tian Li, and Linyuan Lu. Diamond-free families. *Journal of Combinatorial Theory, Series A*, 119(2):310–322, 2012.

30 Daniel S Hirschberg and Lawrence L Larmore. The least weight subsequence problem. *SIAM Journal on Computing*, 16(4):628–638, 1987.

31 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

32 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

33 Thor Johnson, Neil Robertson, Paul D Seymour, and Robin Thomas. Directed tree-width. *Journal of Combinatorial Theory, Series B*, 82(1):138–154, 2001.

34 Donald E Knuth and Michael F Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981.

35 Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1272–1287. SIAM, 2016.

36 Marvin Künnemann, Ramamohan Paturi, and Stefan Schneider. On the Fine-Grained Complexity of One-Dimensional Dynamic Programming. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, volume 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:15, 2017.

37 Leonid Libkin. *Elements of finite model theory*. Springer Science & Business Media, 2013.

38 Guan-Yu Lin, Jia jie Liu, and Yue-Li Wang. Finding a longest increasing subsequence from the paths in a complete bipartite graph. 2012.

39 Andrea Lincoln, Virginia Vassilevska Williams, and Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1236–1252. Society for Industrial and Applied Mathematics, 2018.

40 Neil Robertson and Paul D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

41 Neil Robertson and P.D Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309 – 322, 1986. `doi:https://doi.org/10.1016/0196-6774(86)90023-4`.

42 Liam Roditty and Virginia Vassilevska Williams. Fast approximation algorithms for the diameter and radius of sparse graphs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 515–524. ACM, 2013.

43 Baruch Schieber. Computing a minimum weightk-link path in graphs with the concave monge property. *Journal of Algorithms*, 29(2):204–222, 1998.

44 Larry Joseph Stockmeyer. *The complexity of decision problems in automata theory and logic*. PhD thesis, Massachusetts Institute of Technology, 1974.

45 Moshe Y Vardi. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146. ACM, 1982.

46 Robert Wilber. The concave least-weight subsequence problem revisited. *Journal of Algorithms*, 9(3):418–425, 1988.

47 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2):357–365, 2005.

48 Ryan Williams. Faster decision of first-order graph properties. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 80. ACM, 2014.

49 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 645–654. IEEE, 2010.

50 F Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 429–435. ACM, 1980.

A List of problem definitions and class definitions

Here we list the main problems studied in this paper.

LWS$_C$: Given elements $x_1, \dots, x_n$ and value $F(0) = 0$, compute $F(j) = \min_{0 \leq i < j}[F(i) + c_{i,j}]$ for all $j \in \{1, \dots, n\}$.

StaticLWS$_C$: Given elements $x_1, \dots, x_{2n}$ and values of $F(i)$ on all $i \in \{1, \dots, n\}$, compute $F(j) = \min_{i \in \{1,\dots,n\}}[F(i) + c_{i,j}]$ for all $j \in \{n+1, \dots, 2n\}$.

LWSP$_C$: Given graph $G = (V, E)$ and starting vertex set $V_0 \subseteq V$, compute on each $v \in V$, the value of $F(v)$, where

$$F(v) = \begin{cases} \min(0, \min_{u \rightsquigarrow v}[F(u) + c_{u,v}]), \text{ for } v \in V_0 \\ \min_{u \rightsquigarrow v}[F(u) + c_{u,v}], \text{ for } v \notin v_0 \end{cases}$$

CutLWSP$_C$: On DAG G with a cut (S, T) where edges are only directed from S to T, given the values of function F_S on S, for all $t \in T$ compute $F_T(t) = \min_{s \in S, s \rightsquigarrow t}[F_S(s) + c_{s,t}]$.

Selection$_P$: On two sets X, Y, decide whether $(\exists x \in X)(\exists y \in Y)P(x, y)$.

Path$_P$: On graph $G = (V, E)$, decide whether $(\exists x \in V)(\exists y \in V)[\mathrm{TC}_E(x, y) \wedge P(x, y)]$.

ListPath$_P$: On graph $G = (V, E)$, for all $x \in V$, decide whether $(\exists y \in V)[\mathrm{TC}_E(x, y) \wedge P(x, y)]$.

CutPath$_P$: On graph $G = (V, E)$ with cut (S, T) where edges only direct from S to T, decide whether $(\exists x \in S)(\exists y \in T)[\mathrm{TC}_E(x, y) \wedge P(x, y)]$.

PathFO$_3$: class of Path$_P$ problems such that P is of form $\exists z\psi(x, y, z)$ or $\forall z\psi(x, y, z)$, where ψ is a quantifier-free logical formula.

ListPathFO$_3$: class of ListPath$_P$ problems such that P is of form $\exists z\psi(x, y, z)$ or $\forall z\psi(x, y, z)$, where ψ is a quantifier-free logical formula.

B Problem examples

We give a list of problems that can be considered as instances of LWSP$_C$ or Path$_P$.

Trip Planning (LWSP version of Airplane Refueling)

On a DAG where vertices represent cities and edges are roads, we wish to find a path for a vehicle, along which we wish to find a sequence of cities where the vehicle can rest and add fuel so that the cost is minimized. The cost of traveling between cities x and y is defined by cost $c_{x,y}$. $c_{x,y}$ can be defined in multiple ways, e.g. $c_{x,y}$ is $cost(y)$ if $dist(x, y) \leq M$ and ∞ otherwise. $dist(x, y)$ is the distance between x, y that can be computed by the positions of x, y. M is the maximal distance the vehicle can travel without resting. $cost(y)$ is the cost for resting at position y.

Longest Subset Chain on graphs (LWSP version of Longest Subset Chain)

On a DAG where each vertex corresponds to a set, we want to find a longest chain in a path of the graph such that each set is a subset of its successor. Here $c_{x,y}$ is -1 if S_x is a subset of S_y, and ∞ otherwise.

Multi-currency Coin Change (LWSP version of Coin Change)

Consider there are two different currencies, so there are two sets of coins. We need to find a way to get value V_1 for currency #1 and value V_2 for currency #2, so that the total weight of coins is minimized. Each pair of values $v_1 \in \{0, \ldots, V_1\}$ and $v_2 \in \{0, \ldots, V_2\}$ can be considered as a vertex. We connect vertex (v_1, v_2) to (v_1', v_2') iff $v_1' = v_1 + 1$ or $v_2' = v_2' + 1$. The whole graph is a grid, and we wish to find a subsequence of a path from $(0, 0)$ to (V_1, V_2) so that the cost is minimized. The cost is defined by $C_{(v_1,v_2),(v_1',v_2)} = w_{1,v_1'-v_1}$ and $C_{(v_1,v_2),(v_1,v_2')} = w_{2,v_2'-v_2}$, where $w_{i,j}$ is the weight of a coin of value j from currency #i.

Pretty Printing with alternative expressions (LWSP version of Pretty Printing)

The Pretty Printing problem is to break a paragraph into lines, so that each line have roughly the same length. If a line is too long or too short, then there is some cost depending on the line length. The goal of the problem is to minimize the cost.

For some text, it is hard to print prettily. For example, if there are long formulas in the text, then sometimes its line gets too wide, but if we move the formula into the next line, the original line has too few words. One solution for this issue is to use alternate wording for the sentence, to rephrase a part of a sentence to its synonym. These sentences have different lengths, and formulas in some of them will be displayed better than others. These different ways can be considered as different paths in a graph, and we wish to find one sentence that has the minimal Pretty Printing cost.

A Path_P instance

Say we have a set of words, and we want to find a word chain (a chain of words so that the last letter of the previous word is the same as the first letter of the next word) so that the first word and the last word satisfy some properties, e.g. they do not have similar meanings, they have the same length, they don't have the same letters on the same positions, etc. Each word corresponds to a vertex in the graph. For words that can be consecutive in a word chain, we add an edge to the words.

C Weighted version of Lemma 7

▶ **Lemma 9.** *If in a vertex-weighted DAG $G = (V, E)$ of m edges, every vertex has either ancestors of total weight at most n or descendants of total weight at most n, then there are at most $(m \cdot n^2)$ weighted-pairs of vertices (s, t) such that s can reach t.*

In a vertex-weighted DAG $G = (V, E)$ of m edges, let S, T be two disjoint sets of vertices where edges between S and T only direct from S to T. If every vertex has either ancestors in S of total weight at most n or descendants in T of total weight at most n, then there are at most $(m \cdot n^2)$ weighted-pairs of vertices $s \in S$ and $t \in T$ such that s can reach t.

Let $w(v)$ be the weight of vertex v. The number of weighted-pairs is defined to be the sum of $w(u) \cdot w(v)$ for all pairs of reachable vertices $u \rightsquigarrow v$.

Proof. We define the ancestors of an edge $e \in E$ to be the ancestors (or ancestors in S) of its incoming vertex, and its descendants to be the descendants (or descendants in T) of its outgoing vertex. Let the total weight of its ancestors and descendants be denoted by $anc(e)$ and $des(e)$ respectively.

For each edge $e = (v_1, v_2)$, it belongs to exactly one of the following three types:

Type A: If $anc(e) \le n_1$ but $des(e) > n_2$, then let $count(e)$ be $anc(e) \cdot w(v_2)$.

Type B: If $des(e) \le n_2$ but $anc(e) > n_1$, then let $count(e)$ be $w(v_1) \cdot des(e)$.

Type C: If $anc(e) \le n_1$ and $des(e) \le n_2$, then let $count(e)$ be $anc(e) \cdot des(e)$.

$\sum_{e \in E} count(e) \le m \cdot n_1 \cdot n_2$ because the *count* value on each edge is bounded by $n_1 \cdot n_2$. We will prove that this value upper bounds the number of weighted-pairs of reachable vertices.

For each pair of reachable vertices (u, v) (or (u, v) s.t. $u \in S$ and $v \in T$), let $(e_1, \ldots, e_p)$ be the path from u to v. Along the path, *anc* does not decrease, and *dec* does not increase. A path belongs to exactly one of the following three types:

Type a: Along the path $anc(e_1) \le anc(e_2) \le \cdots \le anc(e_p) \le n_1$, and $des(e_1) \ge des(e_2) \ge \cdots \ge des(e_p) > n_2$. That is, all the edges are Type A.

Type b: Along the path $des(e_p) \le des(e_{p-1}) \le \cdots \le des(e_1) \le n_2$, and $anc(e_p) \ge anc(e_{p-1}) \ge \cdots \ge anc(e_1) > n_1$. That is, all the edges are Type B.

Type c: Along the path there is some edge e_i so that $anc(e_i) \le n_1$ and $des(e_i) \le n_2$. That is, it has at least one Type C edge.

There will not be other cases, for otherwise if a Type A edge directly connects to a Type B edge without a Type C edge in the middle, then the vertex joining these two edges would have more than n_1 ancestors and more than n_2 descendants.

If a path from u to v is Type a, then its last edge e_p is Type A. If it is Type b, then its first edge e_1 is Type B. If it is Type c, then there is some edge e_i in the path that is Type C. This means:

1. For each Type A edge e, $count(e)$ is at least the weight product $w(u) \cdot w(v)$ of all Type a pairs (u, v) whose path has e as its last edge.
2. For each Type B edge e, $count(e)$ is at least the weight product $w(u) \cdot w(v)$ of all Type b pairs (u, v) whose path has e as its first edge.
3. For each Type C edge e, $count(e)$ is at least the weight product $w(u) \cdot w(v)$ of all Type c pairs (u, v) whose path contains e.

Therefore the weight product of the endpoints of each path is counted at least once by the $count(e)$ of some edge e. ◀

D CutPath$_P$ for bounded-treewidth DAGs

We prove the first part of Theorem 1 on DAGs whose underlying undirected graphs have constant treewidth. The algorithm Path$_P$ for constant treewidth graphs is the same as the one for multitrees. In this section we will show the reduction algorithm CutPath$_P$ for constant treewidth graphs on a cut (S, T).

Let $\mathcal{T}$ be the decomposition tree of a graph G. Recall that by the definition of tree decomposition, each node z of the tree corresponds to a set $\mathcal{B}(z)$ which is a subset of vertices of G. Because the treewidth is constant, each set $\mathcal{B}(z)$ has a constant number of vertices. Every vertex of G appears in at least one set of a tree node. Also, for every edge of G, there is at least one tree node whose set contains both its endpoints. And if a vertex v appears both in $\mathcal{B}(z_1)$ and $\mathcal{B}(z_2)$, then along the path from z_1 to z_2, v must appear in all the sets of the tree nodes. Here we consider the decomposition tree as rooted, where all edges are directed from the root to leaves.

We use a similar reduction idea as Section 2.3. In the decomposition tree, each time we find a node z to split the tree into two connected components. We first deal with all the paths that go through the vertices in $\mathcal{B}(z)$. Any other path in the graph must be completely contained in one of the connected components we have created. In the end, all connected components are so small that we can go through all pairs of reachable vertices by exhaustive search. The algorithm is defined in Algorithm 4.

The following claim uses a $1/3 - 2/3$ trick on trees:

▷ Claim 10. In a vertex-weighted rooted tree of total weight n, we can find a connected subgraph of weighted size between $(1/3)n$ and $(2/3)n$ in $O(n)$ time.

Proof. For each node z in the tree, we will compute the weighted size of the subtree rooted at z, denoted by $f(z)$. We compute $f(z)$ from the leaves up to the root, by a reversed topological order. If z is a leaf then let $size(z) \leftarrow w(z)$ where $w(z)$ is the weight of z.

On each parent node p, we initially let $f(p) \leftarrow w(p)$, and then for each child c_i of p, add the value $f(c_i)$ to $f(c_i)$. If before we add the $f(c_i)$ of certain child c_i to $f(p)$, $f(p) < (1/3)n$, and after we add $f(c_i)$ to $f(p)$, $f(p) \geq (1/3)n$, then there are two cases:

If $f(p) \leq (2/3)n$, then the subgraph formed by p and its subtrees $c_1, \ldots, c_i$ is the connected subgraph we want.

If $f(p) > (2/3)n$, then it must be $f(c_i) \geq (2/3)n - (1/3)n = (1/3)n$. That is, the subtree rooted at c_i has weighted size between $(1/3)n$ and $(2/3)n$. But then we should have already returned the subtree rooted c_i instead. So this case would not happen.

Algorithm 4: $\mathsf{CutPath}_P(S, T)$ on constant treewidth DAG

Compute $\mathcal{T}$, the tree decomposition of the underlying undirected graph.
for each z in $\mathcal{T}$ **do**
  Let $size(z)$ be the number of nodes of $\mathcal{T}$.
while there exists a tree node z in $\mathcal{T}$ so that there is a connected subgraph of $\mathcal{T}$ rooted at z with weighted size between $(1/3)size(z)$ and $(2/3)size(z)$ **do**
  `//` z `can be found in time` $O(size(z))$ `by Claim 10.`
  for each $v \in \mathcal{B}(z)$ **do**
    `// Deal with all paths going through` v`.`
    Let A be the set of ancestors of v in S.
    Let B be the set of descendants of v in T.
    Add v to A if $v \in S$, otherwise add v to B.
    if both A and B have at least M^α vertices **then**
      Run $\mathsf{Selection}_P$ on (A, B). If it returns true then **return** true.
    else
      Exhaustively check P on all pairs of $a \in A$ and $b \in B$. If P is true on any (a, b) then **return** true.
    Remove v from the graph, and from the sets of all the tree nodes.
  Remove z from $\mathcal{T}$.
  for each tree node z' who was originally in the same connected component with z **do**
    Update $size(z')$ to be the new size of the connected component z' is in.
for each edge (s, t) crossing the cut(S, T), **do**
  Let A be the set of ancestors of s (including s) in S.
  Let B be the set of descendants of t (including t) in T.
  On all pairs of vertices (a, b) where $a \in A, b \in B$, check property P. If P is true on any pair of (a, b) then **return** true.

After we have added the sizes of all the children of p to $f(p)$, we have finished computing $f(p)$. If $f(p)$ is still less than $1/3$, we will continue to let the next vertex by the reversed topological order be the current parent. ◀

Next we will analyze the reduction algorithm. First, if a the treewidth of a graph is constant, then the corresponding decomposition tree can be computed in linear time [17].

Unlike multitrees, here the calls to $\mathsf{Selection}_P$ are not non-overlapping rectangles: different v from the same $\mathcal{B}(z)$ may share the same ancestors or descendants. However, each time after removing a z, the connected components of the decomposition tree correspond to non-overlapping rectangles in the reachability matrix, and will not overlap with the rectangles corresponding to the ancestors and descendants for any $v \in \mathcal{B}(z)$. Thus, the overlapping only happens when dealing with the ancestors and descendants of different v from the same $\mathcal{B}(z)$, and these $\mathsf{Selection}_P$ rectangles will not overlap with other $\mathsf{Selection}_P$ rectangles after z is removed. Because in each non-overlapping rectangle corresponding to a connected component, we only computed the $\mathsf{Selection}_P$ for $|\mathcal{B}(z)|$ times, which is a constant. So by Claim 8, the total time spent on all the calls to $\mathsf{Selection}_P$ is still $O(M^2/t(M^\alpha))$.

When we remove all vertices $v \in \mathcal{B}(z)$, the graph vertices from sets of different connected components of the decomposition tree are not reachable to each other. Because any path from one connected component to another must go through some vertex in $\mathcal{B}(z)$.

Unlike multitree graphs, this time some vertex v in $\mathcal{B}(z)$ may have fewer than M^α ancestors or descendants. If so, then we do exhaustive search on the sets of v's ancestors and descendants, since calling $\mathsf{Selection}_P$ will not save time. Each time we find a v, the connected component of the decomposition tree that v belongs to loses at least $(1/3)\mathit{size}(v)$ of its vertices, thus each vertex can be the ancestor/descendants of at most $O(\log_{3/2} M)$ such v's. There are at most M vertices in the graph, each of which can take part in at most M^α such paths going through each such v. So the total time is $O(M \cdot \log_{3/2} M \cdot M^\alpha) = O(M^{1+\alpha} \cdot \log_{3/2} M)$.

Also, because each vertex can be the ancestor/descendants of at most $O(\log_{3/2} M)$ such v's, the total time for updating size for all of them is also bounded by $O(M \cdot \log_{3/2} M)$.

In the end, each remaining vertex has $O(M^\alpha)$ ancestors and $O(M^\alpha)$ descendants. The total running time for the exhaustive search is $O(M \cdot M^\alpha \cdot M^\alpha) = O(M^{1+2\alpha})$ by Lemma 7.

The overall running time is $O(M^2/t(M^\alpha) + M^{1+\alpha} \cdot \log_{3/2} M + M^{1+2\alpha})$. By choosing α and γ to be appropriate small constants, we get subquadratic running time.

E Correctness of the LWSP_C algorithm

For the correctness proof, we consider the case where there is no x between S and T. The case where there is an x is similar.

Correctness of $\mathsf{CutLWSP}_C$.

The correctness of $\mathsf{CutLWSP}_C$ follows from the correctness of $\mathsf{CutPath}_P$. We claim that after running $\mathsf{CutLWSP}_C(S, T, F_S)$, for any vertex $t \in T$, there is $F_T(t) = \min_{s \in S, s \rightsquigarrow t}[F_S(s) + c_{s,t}]$. Because for any pair $s \in S$, $t \in T$, such that s reachable to t, they are either processed in a query to $\mathsf{StaticLWS}_C(A, B)$ where $s \in A, t \in B$, or computed separately thus $F_T(t) \leftarrow \min(F_T(t), F(s) + c_{s,t})$.

Correctness of LWSP_C.

The LWSP_C algorithm has the following facts:

1. Whenever a process LWSP_C on domain $V_1 \subseteq V$ returns, the values of F on V_1 are fixed and will not be changed henceforth.
2. Whenever there is an edge from u to v, then the value of F on u is always fixed before the value on v. So the final values of function F on all vertices are fixed by topological order.
3. Each time we call LWSP_C on a subset of vertices $V_1 \subseteq V$, the F values on all ancestors of any vertex in V_1 that are not in V_1 have been fixed by some previous calls to LWSP_C.

Assume that when we call LWSP_C on subgraph with cut (S, T), initially there is

$$F_0(v) = \begin{cases} \min_{u \in R(v) \setminus (S \cup T), u \rightsquigarrow v}[F(u) + c_{u,v}], & \text{if } v \notin V_0 \\ \min(0, \min_{u \in R(v) \setminus (S \cup T), u \rightsquigarrow v}[F(u) + c_{u,v}]), & \text{if } v \in V_0 \end{cases} \tag{1}$$

where $R(v)$ is the set of vertices that can reach v. Then, if $\mathsf{LWSP}_C(S, F_0)$ is correct, after running $\mathsf{LWSP}_C(S, F_0)$, for any $s \in S \backslash V_0$, there is $F(s) = \min_{u \in R(s) \backslash T, u \rightsquigarrow s}[F(u) + c_{u,s}]$. And after running $\mathsf{CutLWSP}_C(S, T, F)$, we have $F_T(t) = \min_{s \in S, s \rightsquigarrow t}[F(s) + c_{s,t}]$. Then after taking $F_0(t) = \min(F_0(t), F_T(t))$ on all t, for any $t \in T \backslash V_0$, we get $F_0(t) = \min_{u \in R(t) \backslash T, u \rightsquigarrow t}[F(u) + c_{u,t}]$. Similarly for any $t \in T \cap V_0$, $F_0(t)$ gets the the minimum of this value and 0. Therefore, on each call of $\mathsf{LWSP}_C(V_1, F_0)$ on a subset $V_1 \subset V$ with initial values F_0, F_0 keeps the invariant in formula (1).

F From listing problems to decision problems

In this section we prove the second part of Theorem 1, that $\mathsf{ListPath}_P$ is reducible to Path_P.

Consider a star graph, which is a graph with its vertex set partitioned in X, Y and another single vertex c. Every $x \in X$ is connected to c, and c is connected to every $y \in Y$. Let problem FindX_P be the following problem: on a star graph, find an $x \in X$ satisfying $(\exists y \in Y)P(x, y)$. We will prove that $\mathsf{ListPath}_P$ is reducible to FindX_P and FindX_P is reducible to Path_P.

▶ **Lemma 11.** *Let* $t(M) \geq 2^{\Omega(\sqrt{\log M})}$. $(\mathsf{ListPath}_P, M^2/(t(\mathsf{poly} M))) \leq_{EC} (\mathsf{FindX}_P, M^2/t(M)))$

Proof. We use a grouping reduction technique similar as the trick in [49] and [8].

We modify the algorithm for Path_P in Section 2 to get the algorithm for $\mathsf{ListPath}_P$. That is, we divide the graph into two subgraphs and call $\mathsf{ListPath}_P$ recursively in a similar wa as Path_P. Path_P needs to call $\mathsf{Selection}_P$ as queries, and in the counterpart of $\mathsf{ListPath}_P$ we will call FindX_P as queries.

Whenever we need to call $\mathsf{Selection}_P(X, Y)$, we partition X and Y into groups of weighted size at most $\sqrt{M}$. Thus there are $O((|X|/\sqrt{M}) \times (|Y|/\sqrt{M}))$ groups. For each pair of group X_i, Y_j, we construct a star graph and call FindX_P on it. The star graph is constructed as follows: Connect every $x \in X_i$ to a dummy vertex c, and connect c to every $y \in Y_j$. Thus if there exist some satisfying x in X_i, FindX_P will find a satisfying x.

Every time a satisfying vertex x in X_i is found by FindX_P, we mark it and add it into the list of satisfying x, and then call the FindX_P on the same star again with x removed from the graph. We keep calling FindX_P on this graph, ignoring all marked vertices, until either all elements in X_i are marked and removed, or FindX_P cannot find a satisfying x.

Because there are at most M vertices that can be listed, there are at most M calls to FindX_P that returns a satisfying x. Each call has instance size $\sqrt{M}$. The running time is $O(M \cdot (\sqrt{M})^2/t(\sqrt{M}))$. The total time spent on the rest of the algorithm is the same as the running time of Path_P. ◀

▶ **Lemma 12.** *Let* $t(M) \geq 2^{\Omega(\sqrt{\log M})}$. $(\mathsf{FindX}_P, M^2/(t(\mathsf{poly} M))) \leq_{EC} (\mathsf{Path}_P, M^2/t(M)))$

Proof. First, we pick an arbitrary element $x_1 \in X$, and construct a graph by letting x_1 connect to all y in Y. Then we call Path_P on this graph. If it returns yes, then we return x_1.

Otherwise, on the star graph we will replace the center vertex c by x_1, remove the original x_1, and call Path_P on this graph. After each call to Path_P, if it returns yes, we divide X in two halves and call Path_P again. Using binary search and shrinking the size of X by half each time, we will finally find a satisfying x. ◀

Lemmas 11 and 12 imply the reduction $(\mathsf{ListPath}_P, M^2/(t(\mathsf{poly} M))) \leq_{\mathrm{EC}} (\mathsf{Path}_P, M^2/t(M)))$ for $t(M) \geq 2^{\Omega(\sqrt{\log M})}$.

G From parallel problems to sequential problems

We prove the third part of Theorem 1, the other direction of the reduction. The reduction from Path_P to $\mathsf{ListPath}_P$ is straightforward.

To reduce from $\mathsf{Selection}_P$ to Path_P, we can construct a graph with dummy vertex c in the middle, such that each x in set X is connected to c, and c is connected to every y in set Y. If P is expressible by first-order logic, then we will let c act like one of the y's when computing $R(x, c)$, and act like of the x's when computing predicates on $P(c, y)$. Let x_1 be an arbitrary element in X, and y_1 be an arbitrary element in Y. We create c by merging x_1 and y_1 into a single element. c has all the relations x_1 and y_1 have. Thus, on any $x \in X, x \neq x_1$, the value of $P(x, c)$ is the same as $P(x, y_1)$. Symmetrically on any $y \in Y, y \neq y_1$, the value of $P(c, y)$ is the same as $P(x_1, y)$. Therefore, there exists x, y such that $P(x, y)$ is true iff $\mathsf{Selection}_P$ on this graph returns true.

In general, if we are allowed to define another property P' such that $P'(x, y) \leftarrow (P(x, y) \wedge (x \neq c) \wedge (y \neq c))$, we have a reduction from $\mathsf{Selection}_P$ to $\mathsf{Path}_{P'}$.

H Reachability Oracle

This section presents a proof of Theorem 5. A reachability oracle on a graph takes in a pair of vertices u, v in the graph, and answers whether v is reachable from u. A naive approach is to use $O(n^2)$ space to store the reachability of all pairs of vertices. By adapting the Path_P algorithm on multitrees, we get sublinear time reachability oracles for multitrees using subquadratic space and subquadratic preprocessing time. If the graph is a multitree of strongly connected components, we can first treat each strongly connected component as a single vertex, whose weighted size is the total weighted size of all vertices in the component.

The reachability oracle for multitrees can be adapted directly from the Path_P algorithm. In the recursion tree of calling Path_P, we take down the final subproblem that each vertex belongs to, and when querying a pair of vertex, we find the Path_P instance corresponding to the least common ancestor of the two final subproblems corresponding to these vertices, and consider the $\mathsf{CutPath}_P$ process called by this Path_P instance.

Next we modify the $\mathsf{CutPath}_P$ algorithm. Among all pivot vertices, we call the ones who have no other pivot vertices as descendants "sink pivot vertices". After computing the number of ancestors and descendants for all vertices, we can decide if a vertex is a sink in time linear to the degree of the vertex.

We create another graph G'. Similar to $\mathsf{CutPath}_P$, we keep finding pivot vertices who have at least M^α ancestors and M^α descendants in the remaining graph, and then remove them. Whenever finding a pivot vertex v, we create edges from all its ancestors to v, and from v to all its descendants in G'.

Thus, when querying a pair of vertices a, b, if a can reach b, there are three cases:

- Case 1: b is a pivot vertex. Then there is an edge from a to b in G'.
- Case 2: The path from a to b goes through at least a pivot vertex. In this case, it must go through a sink pivot vertex. We decide if there is a sink pivot vertex v adjacent to a in G' who is also adjacent to b. Each vertex can be an ancestors of at most M/M^α sink pivot vertices, because each sink pivot vertex has more than M^α descendants, and different sink pivot vertices have disjoint set of descendants. So this checking can be done in time M/M^α.
- Case 3: The path from a to b does not go through any pivot vertex. Then we can do a DFS starting from a that traverses through at most M^α of a's descendants in the remaining graph G to find b. The time taken is $O(M^\alpha)$.

Thus, the query time is $O(M/M^\alpha + M^\alpha)$, which is sublinear to M.

If the graph is a multitree, not a multitree of strongly connected components, then every vertex has unit weighted size. In this case, the modified $\mathsf{CutPath}_P$ process runs in time $O(m^2/m^\alpha + m^{2-\alpha} + m^{1+2\alpha})$, because now we do not use $\mathsf{Selection}_P$ thus the function $t(m)$ is $O(m)$, and there are no large-size vertices thus we can pick $\gamma = 0$. If we choose $\alpha = 1/3$, then the running time is $O(m^{5/3})$. The modified Path_P algorithm has running time satisfying the recursion $T(m) = 2T(m/2) + O(m^{5/3})$, which is $O(m^{5/3})$. So the preprocessing time and space is $O(m^{5/3})$, and the query time is $O(m^{2/3})$.

IPEC 2019

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il