K-Best Solutions of MSO Problems on Tree-Decomposable Graphs*

David Eppstein[†1] **and Denis Kurz**[‡2]

1 **Computer Science Department, UC Irvine, USA**
eppstein@uci.edu
2 **Department of Computer Science, TU Dortmund, Germany**
denis.kurz@tu-dortmund.de

Abstract

We show that, for any graph optimization problem in which the feasible solutions can be expressed by a formula in monadic second-order logic describing sets of vertices or edges and in which the goal is to minimize the sum of the weights in the selected sets, we can find the k best solution values for n-vertex graphs of bounded treewidth in time $\mathcal{O}(n+k\log n)$. In particular, this applies to finding the k shortest simple paths between given vertices in directed graphs of bounded treewidth, giving an exponential speedup in the per-path cost over previous algorithms.

1998 ACM Subject Classification G.2.2 Graph Theory

Keywords and phrases graph algorithm, k-best, monadic second-order logic, treewidth

Digital Object Identifier 10.4230/LIPIcs.IPEC.2017.16

1 Introduction

Finding multiple alternative routes between two vertices in a network, formalized as the *k shortest paths problem*, has many applications including biological sequence alignment, metabolic pathway reconstruction, hypothesis generation in natural language processing, computer network routing, and vehicle routing [15]. It can be solved in constant time per path, after near-linear preprocessing time [14]. However, for graphs with cycles, its paths may have repeated vertices, which are often undesirable. A variant of the problem, the *k shortest simple paths problem*, disallows these repetitions, but for the past 45 years there have been no asymptotic improvements to an algorithm of Yen, which takes quadratic time per path [23]. Indeed, Vassilevska Williams and Williams show that, at least for $k = 2$, no substantial improvement to this algorithm is likely [22]. Even for undirected graphs, known algorithms for k shortest simple paths take at least linear time per path [21], far from the constant time per path for non-simple paths.

This situation suggests studying parameterized classes of graphs for which faster k-best optimization algorithms are possible. In this paper we provide a first result of this type, showing that the lengths of the k shortest simple paths can be found in logarithmic time per path (exponentially faster than the polynomial per-path time of previous algorithms) for graphs of bounded treewidth. Our results are based on general algorithmic metatheorems

* A full version of the paper is available at https://arxiv.org/abs/1703.02784.
† The research of David Eppstein was supported in part by NSF grants CCF-1228639, CCF-1618301, and CCF-1616248.
‡ This research was performed in part during a visit of Denis Kurz to UC Irvine, supported by DFG GRK 1855 (DOTS).

and in particular *Courcelle's theorem* according to which decision and optimization problems expressible in the monadic second-order logic of graphs (MSO) can be solved in linear time on graphs of bounded treewidth. MSO is a form of logic in which the variables of a formula represent vertices, edges, sets of vertices, and sets of edges of a graph, one can test set membership and vertex–edge incidence, and variables can be existentially or universally quantified. For instance the property that an edge set P represents a simple path from s to t can be expressed in MSO as a formula

$$\begin{aligned}
&(\forall v,e,f,g)\Big[\big(I(v,e)\wedge e\in P\wedge I(v,f)\wedge f\in P\wedge I(v,g)\wedge g\in P\big)\Rightarrow\big(e=f\vee e=g\vee f=g\big)\Big]\\
&\quad\wedge(\exists e)\Big[I(s,e)\wedge e\in P\wedge(\forall f)\Big[\big(I(s,f)\wedge f\in P\big)\Rightarrow e=f\big)\Big]\Big]\\
&\quad\wedge(\exists e)\Big[I(t,e)\wedge e\in P\wedge(\forall f)\Big[\big(I(t,f)\wedge f\in P\big)\Rightarrow e=f\big)\Big]\Big]\\
&\quad\wedge(\forall S)\Big[\neg(\exists v,e)\big[v\in S\wedge e\in P\wedge I(v,e)\big]\vee\neg(\exists v,e)\big[\neg(v\in S)\wedge e\in P\wedge I(v,e)\big]\vee\\
&\qquad\qquad(\exists v,w,e)\big[v\in S\wedge\neg(w\in S)\wedge I(v,e)\wedge I(w,e)\big]\Big]
\end{aligned}$$

(where s, t, v and w are vertex variables, e, f, and g are edge variables, S is a vertex-set variable, and I is the vertex–edge incidence predicate). This formula expresses the constraints that each vertex is incident to at most two edges of P, s and t are each incident to exactly one edge of P, and every partition of the vertices that is not crossed by P has P only on one of its two sides.

Courcelle's theorem translates MSO formulas to tree automata, allowing graphs of bounded treewidth that satisfy the formula to be recognized in linear time by a bottom-up dynamic programming algorithm that executes the tree automaton on a tree-decomposition of the graph. Extensions of this method also solve optimization problems for MSO predicates (formulas with one unbound set-variable) that seek the minimum weight vertex or edge set obeying the predicate [3, 10]. For instance, it can find shortest simple paths on bounded-treewidth graphs with negative edges and negative cycles, an NP-hard problem on arbitrary graphs.

In this paper, we show that, for any MSO predicate, the weights of the k minimum-weight sets satisfying the predicate can be found on graphs of bounded treewidth in logarithmic time per set. In particular, using the formula given above, we can find the k shortest simple paths in logarithmic time per path. Other previously-studied graph optimization problems to which our method applies (with an exponential per-solution speedup) include finding the k smallest spanning trees [16], the k best matchings [8], and (with a doubly exponential speedup) the k best solutions to the traveling salesperson problem [11]. Although the example formula above describes simple paths in undirected graphs, our method applies as well to directed graphs whose underlying undirected graph has bounded treewidth.

To prove this we use a special tree-decomposition with bounded width, logarithmic depth, and bounded degree. We translate the dynamic program for finding the minimum weight set satisfying an MSO predicate into a *fully persistent dynamic graph algorithm* for the same optimization problem, one that can report the minimum weight solution after modifying the given graph by changing the weights of some of its edges or vertices. We apply this method to find the *second-best* (rather than best) solution, and to detect a feature of the graph (a vertex or edge) at which the best and second-best solution differ. By branching on this feature we can recursively decompose the original problem into a hierarchy of subproblems whose second-best solutions (together with the global best solution) include all of the k best

solutions to the input problem. To find the k best solutions, we perform a best-first search of this hierarchy.

For space reasons, we defer some proofs to the full version, online at arXiv:1703.02784.

2 Preliminaries

In this section, we establish most of our notation and definitions, and review algorithmic components from previous research that we will use to establish our results.

We denote by $[i]$ the set $\{1, \ldots, i\}$ for $i \in \mathbb{N}$. The cardinality of a set M is denoted by $|M|$, and its power set by 2^M. For a function $f : M \to N$, we write $f(M') = \{f(a) \mid a \in M'\}$ for $M' \subseteq M$. For an n-element sequence S and $i \in [n]$, S_i denotes the i-th element of S.

Our algorithms assume a RAM model of computation in which addition and comparison operations on input weights or sums of weights can be performed in constant time per operation.

2.1 Binary heap of subproblems

We adopt the following technique for finding the k best solutions to a combinatorial optimization problem [20, 15], which was first used by Gabow for the k smallest spanning trees [19]. We assume that the problem's solutions can be represented as sets of edges or vertices in a weighted graph, and that the goal is to minimize total weight. We form *subproblems* by forcing certain edges or vertices to be included in the solution and preventing other edges or vertices from being included; these constraints can be simulated without changing the graph by changing some weights to large positive and negative numbers. We say that a subproblem is *feasible* if it has a solution consistent with its constraints. If S is any subproblem, then we may consider two solutions to S, its *best solution* (the one with minimum weight, subject to the constraints) and its *second-best solution* (the one that differs from the best solution and otherwise has the minimum possible weight), with ties broken in any consistent way. We say that an edge or vertex is a *pivot feature* if it is present in the best solution but absent in the second-best solution, or vice versa. If a subproblem has only one solution, we say that it is *uniquely solvable*.

We then form a binary tree of feasible subproblems, as follows. The root of the tree is the subproblem with no constraints (the one whose solutions are all solutions to the given problem on the whole graph G). Then, for each subproblem S in the tree that is not uniquely solvable, the two children of S are determined by choosing (arbitrarily) a pivot feature of S, constraining that pivot feature to be included in the solutions for one child, and constraining the same pivot feature to be excluded from the solutions for the other child. These two children of S have solution sets that partition the solutions of S into two nonempty subsets, one containing the best solution and the other containing the second-best solution. A uniquely solvable subproblem in this tree of subproblems forms a leaf, with no children.

Each solution of the input problem appears as second-best in exactly one subproblem in this tree, except for the global best solution which is never second-best anywhere. The tree is ordered as a binary min-heap: each subproblem's second-best solution is better than the second-best solutions of its children. Therefore, the k-best solutions of the input problem can be listed by finding the best solution and then using a best-first search in the tree of subproblems to find the $k - 1$ subproblems whose second-best solutions have the smallest values. This search creates an active set consisting of the tree root, and then (for $k - 1$ iterations) uses a priority queue to find the smallest second-best solution value in

the active set, outputs that solution, and replaces that node in the active set by its two children. Alternatively, the added time from the priority queue can be avoided by using a heap-selection algorithm of Frederickson [18] which runs in $O(k)$ time, plus the time for $\mathcal{O}(k)$ evaluations of subproblems.

Using this method, the main remaining task is to find the second-best solution values and pivot features of each subproblem in this tree of subproblems, as efficiently as possible.

2.2 Path-copying persistence

We will develop a data structure that allows us to add a new constraint to a subproblem, forming one of its child subproblems, and efficiently compute the new second-best solution value of the new child subproblem. However, without additional techniques such a data structure would allow us to follow only a single branch of the tree of subproblems. We use ideas from *persistent data structures*, following Sarnak et al. [12], to extend these data structures to ones that let us explore multiple branches of the tree of subproblems concurrently.

An *ephemeral data structure* is one that has only a single version, which is changed by certain *update* operations and accessed but not changed by additional *query* operations. In the corresponding *fully persistent data structure*, each operation takes an additional argument, the *version* of the data structure, and operates on that version. Persistent queries return the result of the query on that version, and do not change it. Persistent updates create and return a new version of the data structure, in which the given change has been made to the version given as an argument. The previous version is left intact, allowing future updates and queries to it.

Path copying is a technique introduced by Sarnak et al. for making any tree-based ephemeral data structure (such as a binary heap or binary search tree) fully persistent. The underlying ephemeral data structure must form a tree of nodes, with each node pointing to its children but without pointers to parents or other non-child nodes. Every operation in the ephemeral data structure should be performed by starting at the tree root, following child pointers to find additional nodes reachable by paths from the root, and then (in case of a query) collecting information from those nodes or (for updates) changing or replacing some of the reached nodes.

To make such a data structure fully persistent, we represent each version of the data structure by the root of its tree. Persistent queries follow the same algorithm as ephemeral queries, starting from the root node representing the desired version. When an ephemeral update would change or replace some nodes, the persistent structure creates new nodes for all of these changed or replaced nodes and all of their ancestors, without changing to the existing nodes. In this way, the space and time of each persistent update are proportional to the time for an ephemeral update.

Path-copying persistence was used in the k-shortest paths algorithm of Eppstein [14], to construct certain persistent heap structures that represent sets of *detours*. Here, we apply the same technique in a different way, to make persistent an ephemeral data structure for second-best solutions. We will associate a version of the second-best solution data structure with each subproblem in the binary tree of subproblems described in the previous section. Then, when we expand a subproblem (finding its pivot feature and using that feature to define two new child subproblems) the data structure versions of the two child subproblems can be found by applying two different persistent updates to the version of their parent.

2.3 Shallow tree decompositions

The data structure to which we will apply the path-copying persistence technique will be based on a tree decomposition of the given graph. However, in order to make the path-copying efficient, we need to use a special kind of tree decomposition, one with low depth.

A *tree decomposition* of a graph $G = (V, E)$ is a pair $(T = (U, F), bag)$, where T is a tree, $bag : U \to 2^V$ maps tree nodes to subsets of V ("bags"), each vertex belongs to a nonempty collection of bags that induce a connected subtree of T, and for each edge at least one bag contains both edge endpoints. The *width* of a tree decomposition is one less than the size $\max\{|bag(u)| \mid u \in U\}$ of its largest bag. The *treewidth* of G is the smallest w such that G has a tree decomposition of width w. For any fixed w, one can recognize the graphs of treewidth at most w and compute a tree decomposition of optimal width for these graphs, in linear time [6].

We define the *depth* of a tree decomposition to be the longest distance from a leaf to a root node chosen to minimize this distance. A *shallow tree decomposition* of a graph G of bounded treewidth w is a tree decomposition with width $\mathcal{O}(w)$ and depth $\mathcal{O}(\log |G|)$ whose tree is binary. Shallow tree decompositions always exist [5], and can be constructed by a parallel algorithm whose sequential version takes linear time [7].

2.4 Hypergraph algebra

We adopt much of the following notation from Courcelle and Mosbah [10].

Let A be a ranked alphabet consisting of *edge labels*, and let $\tau : A \to \mathbb{N}$ map labels to their *orders*. A *hypergraph* $G = (V, E, lab, vert, src)$ *of order* r consists of a set of vertices V and a set of hyperedges E, an edge labeling function $lab : E \to A$, a function $vert : E \to V^*$ that maps edges to node sequences, and a sequence src of r source nodes. The *order* of a hyperedge $e \in E$ is the length $|vert(e)|$ of its vertex sequence, and must match the order of its label: $\tau(lab(e)) = |vert(e)|$. A graph of order r is a hypergraph of order r with $\tau(lab(E)) = \{2\}$, so every hyperedge has order 2. Hyperedges of a graph may be called *edges*.

We define, for an edge label alphabet A, a hypergraph algebra with a possibly infinite set of hypergraph operators and the following finite set of constants. The constant $\mathbf{0}$ denotes the empty hypergraph of order 0. The constant $\mathbf{1}$ denotes the hypergraph of order 1 with a single source vertex and no hyperedges. For each $a \in A$, the constant $\mathbf{a}$ denotes the hypergraph of order $\tau(a)$ with node set $\{v_1, \ldots, v_{\tau(a)}\}$, a single hyperedge e with $lab(e) = a$ and $vert(e) = (v_i)_{i \in [\tau(a)]}$, and $src = vert(e)$.

Let G be a hypergraph of order r and let G' be a hypergraph of order r'. The hypergraph algebra has the following operators. The $(r + r')$-order hypergraph $G \oplus_{r,r'} G'$ consists of the disjoint union of the vertex and edge sets of G and G', and the concatenation of their source sequences. For each $i, j \in [r]$, $\theta_{i,j,r}(G)$ is the hypergraph of order r obtained from G by replacing every occurence of src_j with src_i in the source sequence of G and in every vertex sequence of a hyperedge of G. This is equivalent to fusing src_j into src_i. For a mapping $\alpha : [p] \to [r]$, $\sigma_\alpha(G)$ is the hypergraph of order p obtained from G by replacing its r-element source sequence src with the p-element sequence src', with $src'_i = src_{\alpha(i)}$. This infinite family of operators can generate any hypergraph. For each family of hypergraphs L that only contains hypergraphs of bounded treewidth and bounded order, a finite subset of these operators generates a superset of L [2, 4, 9]. We denote by $\mathcal{G}_w$ a finite hypergraph algebra as above that generates all r-order hypergraphs over a fixed label set of treewidth at most w and $r < R$ for some fixed but arbitrary R.

Let A be an alphabet of edge labels, and let $G = (V, E, lab, vert, src)$ be a hypergraph over A. A formula in *counting monadic second-order logic* (*CMS formula*) is a formula in monadic second-order logic, extended by predicates $\mathbf{Card}_{m,p}(X)$ for $m, p \in \mathbb{N}$, with $X \models \mathbf{Card}_{m,p}(X)$ iff $|X| \equiv m \mod p$, and by incidence predicates $edg_a(e, v_1, \ldots, v_{\tau(a)})$ for $a \in A$, with $(G, e, v_1, \ldots, v_{\tau(a)}) \models edg_a(e, v_1, \ldots, v_{\tau(a)})$ iff $lab(e) = a$ and $vert(e) = (v_1, \ldots, v_{\tau(a)})$. The finite set $\Phi^{h,q}_{A,R}(\mathcal{W})$ consists of exactly one representative of every class of equivalent CMS formulas for hypergraphs of order at most R over edge labels A on variables $\mathcal{W}$ whose depth of nested quantification is at most h and $p < q$ for all subformulas of the form $\mathbf{Card}_{m,p}(X)$. Since h, q, A and $\mathcal{W}$ are fixed in most contexts, we use the short form $\Phi_R = \Phi^{h,q}_{A,R}(\mathcal{W})$. We require variable alphabets $\mathcal{W}$ that include constants for all the source nodes of the hypergraphs, i.e., $\{\mathbf{s}_i \mid i \in [R]\} \subset \mathcal{W}$ and $(G, v) \models (v = \mathbf{s}_i)$ iff $v = src_i$. Every other variable X in $\mathcal{W}$ is assumed to be either a vertex set variable, denoted as $type(X) = V$, or a hyperedge set variable, denoted as $type(X) = E$. This can be enforced by only considering formulas that include subformulas $\forall x : x \in X \Rightarrow x \in V$ if X is supposed to model a set of vertices, and $\forall x : x \in X \Rightarrow x \in E$ otherwise. Variables representing single elements can be emulated by a subformula $\exists x : \forall y : x \in X \wedge (y \in X \Rightarrow x = y)$.

Our algorithms work on parse trees of hypergraphs with respect to some fixed hypergraph algebra $\mathcal{G}_w$. A *parse tree* $T = (U, F)$ is a directed rooted tree, with edges being directed away from the root r. Leaves of T are associated with a constant of $\mathcal{G}_w$; inner nodes are associated with an operator of $\mathcal{G}_w$. The hypergraph represented by T is the hypergraph constant associated with r if r is a leaf. Otherwise, T represents the hypergraph obtained by applying the operator associated with r to the hypergraphs represented by the parse subtrees rooted in the children of r. For $u \in U$, the hypergraph represented by the parse subtree of T rooted in u is denoted by $G(u)$. Sometimes it is convenient to have exactly two child nodes for each inner node of the parse tree. We can think of σ_α operators as having a second operand that is always the empty hypergraph. Likewise, $\theta_{i,j,r}$ operators can be viewed as having a second operand that is always the one-vertex hypergraph. These modified operators can even be derived from the original ones. Instead of σ_α, we can use $\sigma'_\alpha = \sigma_\alpha \circ \oplus_{0,r}(\mathbf{0})$. The composition $\theta'_{i,j,r} = \theta_{i,j,r} \circ \sigma_\alpha \circ \theta_{i,r+1,r+1} \circ \oplus_{1,r}(\mathbf{1})$ with $\sigma : [r] \to [r+1]$, $\sigma(i) = i$, can replace $\theta_{i,j,r}$. All operators of this derived algebra are binary. We call a respective parse tree a *full parse tree*. The *proper* child of a θ or σ node is the one that does not represent the one-vertex or empty hypergraph, respectively.

Let G be a hypergraph, and $T = (U, F)$ a parse tree of G. We consider combinatorial problems on G that can be characterized by a CMS formula $\varphi \in \Phi_R$. Let n be the number of free variables of φ, and let $X = (X_i)_{i \in [n]}$ be the free variables themselves. For example, we need $n = 1$ edge set to describe a (simple) path, or $n = c - 1$ node sets to describe a c-coloring of a graph. An *assignment* maps every X_i to a subset of $type(X_i)$. A *satisfying assignment* is an assignment f such that $(f(X), G) \models \varphi$. A *solution* is the sequence $(f(X_i))_{i \in [n]}$ for some assignment f. Two solutions S, S', are considered distinct if there is a $j \in [n]$ with $S_j \neq S'_j$. For $S_1 \neq S_2$, (S_1, S_2) is also distinct from (S_2, S_1) regardless of φ, even if φ is a formula on two free variables, and symmetric on these free variables. For a parse tree node $u \in U$, $S(u)$ denotes the *subsolution of S at u*, which is obtained from S be removing every graph feature from every set S_i that is not present in $G(u)$. A solution is *feasible* if the corresponding assignment is satisfying. We denote by $\boldsymbol{sat}(G, \varphi)$ the set of feasible solutions, i.e., $Y \in \boldsymbol{sat}(G, \varphi) \Leftrightarrow (Y, G) \models \varphi$.

Sets of solutions, and $\boldsymbol{sat}(G, \varphi)$ in particular, are sets of n-tuples of sets. We write $M \sqcup N$ for the disjoint union of sets M and N (undefined if $M \cap N \neq \emptyset$). Let X, Y be solutions, and A, B sets of solutions. We say that X and Y *interfere* if there are $i, j \in [n]$ such that

$X_i \cap Y_j \neq \emptyset$. By extension, A and B interfere if some $X \in A$ interferes with some $Y \in B$. If A and B do not interfere, $A \uplus B$ denotes the set of all combinations of each $X \in A$ and $Y \in B$, i.e.,

$$A \uplus B = \{(X_i \sqcup Y_i)_{i\in[n]} \mid X \in A, Y \in B\}.$$

Note that the above operators are not defined for every combination of operands. A *semi-homomorphism* from $\langle \mathcal{S}, \uplus, \sqcup, \emptyset, \emptyset\rangle$ to some evaluation structure $\langle \mathcal{R}, \oplus, \otimes, \mathbf{0}, \mathbf{1}\rangle$ is a mapping $f : \mathcal{S} \to \mathcal{R}$ that acts like a homomorphism where applicable, i.e., $f(\emptyset) = \mathbf{0}$, $f(\emptyset) = \mathbf{1}$, $f(A \uplus B) = f(A) \oplus f(B)$ if A and B do not interfere, and $f(A \sqcup B) = f(A) \otimes f(B)$ if $A \cap B = \emptyset$.

Values of solution sets are expressed in terms of *evaluation structures*. An evaluation structure is an algebra $\langle \mathcal{R}, \oplus, \otimes, \mathbf{0}, \mathbf{1}\rangle$ such that $\langle \mathcal{R}, \oplus, \mathbf{0}\rangle$ and $\langle \mathcal{R}, \otimes, \mathbf{1}\rangle$ are monoids. An *evaluation* v is a function that maps hypergraphs generated by $\mathcal{G}_w$ to an (ordered) evaluation structure $\mathcal{R}$. An evaluation v is an *MS-evaluation* if there exists a semi-homomorphism h and a CMS formula $\varphi \in \Phi_R$ such that $v(G) = h(\boldsymbol{sat}(G, \varphi))$ for every hypergraph G.

A *linear CMS minimization problem* P consists of a formula $\varphi = \varphi(P) \in \Phi_R$ with n free variables X that characterizes feasible solutions, and an *instance* of the problem consists of an r-order hypergraph G and a sequence c of n cost functions with $c_i : \mathit{type}(X_i) \to \mathbb{R}$. The value $c(Y)$ of a solution Y for P is defined as $\sum_{i\in[n]} \sum_{y\in Y_i} c_i(y)$. An optimal solution of P is a feasible solution Y^* such that for each feasible solution Y' of P, we have $c(Y^*) \leq c(Y')$. Our task is to find the value of an optimal feasible solution of P, or equivalently, to compute the value of $v(G) = \min\{c(Y) \mid Y \in \boldsymbol{sat}(G, \varphi)\}$. This evaluation can be expressed as $v(G) = h(\boldsymbol{sat}(G, \varphi))$ with $h(A) = \min\{c(Y) \mid Y \in A\}$, and is thus an MS-evaluation. Note that h is a semi-homomorphism, and the corresponding evaluation structure is $\langle \mathbb{R} \cup \{\infty\}, +, \min, 0, \infty\rangle$. The class of all linear CMS minimization problems is called LinCMS. Let $m = |\boldsymbol{sat}(G, \varphi)|$, and $\boldsymbol{sat}(G, \varphi) = \{Y^1, \ldots, Y^m\}$. Let $\Pi = \Pi(G, \varphi) \subseteq S_n([m])$ be the set of permutations such that $(c(Y^{\pi(i)}))_{i\in[m]}$ is nondecreasing for $\pi \in \Pi$. For a problem $P \in \text{LinCMS}$, k-*val*(P) is again a problem that gets the same input as P itself, plus some $k \in \mathbb{N}$. It asks for the sequence $(c(Y^{\pi(i)}))_{i\in[k']}$ for some $\pi \in \Pi$, where $k' = \min\{k, m\}$. The problem k-*sol*(P) gets the same input as k-val(P), but asks for $(Y^{\pi(i)})_{i\in[k']}$ for any $\pi \in \Pi$. Depending on π, different outputs for k-sol(P) are possible, but there is only one valid output for k-val(P). Also note that 1-val(P) is equivalent to P itself. For simplicity, we assume $k \leq m$ for the remainder of this article.

3 The second-best solution

In the following section, we describe how to solve 2-sol(P) in linear time for each P in LinCMS. In a later section we generalize these results to an arbitrary (constant) number of solutions, and to LOGSPACE and PRAM models of computation.

A hypergraph G generated by $\mathcal{G}_w$, and a formula $\varphi \in \Phi_R$, $\boldsymbol{sat}(G, \varphi)$ can be evaluated by a tree automaton on a parse tree of G [10, 17]. Our notion of full parse trees enables us to unify Lemmas 2.4 to 2.6 from Courcelle and Mosbah [10] as follows.

▶ **Lemma 1.** *Let $\varphi \in \Phi_R$, and let T be a full parse tree rooted in r that represents a hypergraph G. If r is not a leaf, it has two child nodes u_1, u_2 representing $G_1 = G(u_1)$, $G_2 = G(u_2)$, respectively, and there exist $l \in \mathbb{N}$ and $\psi_1^k, \psi_2^k \in \Phi_R$ for each $k \in [l]$ such that*

$$\boldsymbol{sat}(G, \varphi) = \bigsqcup \left\{ \boldsymbol{sat}(G_1, \psi_1^k) \uplus \boldsymbol{sat}(G_2, \psi_2^k) \mid k \in [l] \right\}, \tag{1}$$

where l only depends on the hypergraph operator associated with r.

In the situation of Lemma 1, we call ψ_i^k a *child formula* of φ with respect to the corresponding operator. For $k \in [l]$, we call (ψ_1^k, ψ_2^k) a *fitting pair of child formulas*. A solution $S \in \boldsymbol{sat}(G, \varphi)$ has a unique fitting pair (ψ_1, ψ_2) of child formulas with $S \in (\boldsymbol{sat}(G_1, \psi_1) \sqcup \boldsymbol{sat}(G_2, \psi_2))$, which follows from the fact that all unions in Equation (1) are disjoint. All child formulas ψ with respect to a $\theta_{i,j,m}$ node u can be chosen such that $src_i \notin S_k$ for any $S \in \boldsymbol{sat}(G(u'), \psi)$, where u' is the proper child of u. Further, Courcelle and Mosbah demonstrated that every MS-evaluation v can be computed by a similar tree automaton. The running time required to compute v on the entire hypergraph G is $\mathcal{O}(|G| \cdot \mu)$, where μ is the time required to compute $f(A) \oplus f(B)$ and $f(A) \otimes f(B)$ for valid combinations A, B. In the uniform cost model, the operators of the evaluation structure $\langle \mathbb{R} \cup \{\infty\}, +, \min, 0, \infty \rangle$, addition and selecting the smaller of two real numbers, require $\mathcal{O}(1)$ time. Problems in LinCMS can therefore be solved in linear time. Applying the semi-homomorphism h with $v(G) = h(\boldsymbol{sat}(G, \varphi))$ to Equation (1) yields

$$v(G) = h(\boldsymbol{sat}(G, \varphi)) = \min \left\{ h(\boldsymbol{sat}(G_1, \psi_1^k)) + h(\boldsymbol{sat}(G_2, \psi_2^k)) \mid k \in [l] \right\}. \tag{2}$$

A different linear-time approach for this special case had been proposed earlier by Arnborg, Lagergren and Seese [3]. The basic algorithm of Courcelle and Mosbah computes $h(\boldsymbol{sat}(G(u), \psi))$ for every parse tree node u in a bottom-up manner, and every $\psi \in \Phi_R$. Conceptually, we perform a depth-first search on T, starting at its root r. Every time we finish a node v, we *evaluate* it, i.e., we compute the evaluation $h(\boldsymbol{sat}(G(v), \phi))$ for every formula $\psi \in \Phi_R$ based on the child formulas of ψ by Equation (2). Since the number of formulas and the number of child formulas per formula are fixed, the overall running time is linear in the size of T.

Courcelle and Mosbah also propose an improved algorithm, called the *CM algorithm* in this article, that determines in a top-down preprocessing phase the set of formulas that are reachable from φ at r via the child formula relation. We call these the *relevant formulas* of a parse tree node u.

Let x be a graph feature that is part of at least one solution for φ at r, and let v_1, v_2 be the child nodes of r. If r is an $\oplus$ or σ node, the graph features of $G(v_1)$, $G(v_2)$ are disjoint, so exactly one of them contains x. If r is a θ node that fuses src_j into src_i, let v_2 be the the child node such that $G(v_2)$ only consists of src_i. As can be seen from the proof of Lemma 2.5 in [10], subformulas of φ can be chosen such that src_i is not part of any solution at v_1. Taken together, x is part of some solutions at either v_1 or v_2, but not both. Applying the argument iteratively to this child node, there is exactly one leaf of the full parse tree where some solutions contain x. We denote this leaf by $u(x)$.

Let u be a parse tree node, ψ_1, ψ_2 two formulas relevant for u. The CM algorithm only computes $h(G(u), \psi_1)$ and $h(G(u), \psi_2)$, which correspond to the values of optimal solutions on $G(u)$ for the problems characterized by ψ_1 and ψ_2, respectively. We do not have any information about the solutions themselves besides their values. In particular, we do not know for any hypergraph feature if it appears in one of those solutions. Since we do not require optimal solutions to be unique, we also do not know if they represent the same solution even in the case $h(\boldsymbol{sat}(G(u), \psi_1)) = h(\boldsymbol{sat}(G(u), \psi_2))$.

In the next section, we need to test if optimal solutions for ψ_1 and ψ_2 are the same by only looking at their evaluations. To do so, we establish for each parse tree node u a mapping from the relevant formulas of u to solution IDs in $[|\Phi_R|]$, with the following discriminating property. Each relevant formula ψ of u is mapped to an optimal solution for ψ on $G(u)$ such that two formulas are assigned the same solution if and only if they are assigned the same solution ID. Solution IDs can be computed along with solution values while keeping the linear time bound:

▶ **Lemma 2.** *Given solution IDs for all relevant formulas of all child nodes of a parse tree node u, solution IDs for u can be computed in constant time.*

Proof. Deferred to the full version. ◀

The compression function can be stored with u, and a mapping from solution ID to solution at leaf nodes, requiring only constant space. Even if $\boldsymbol{sat}(G(u), \psi)$ contains multiple optimal solutions, it is now possible to refer to *the optimal solution*, which is the one defined recursively in terms of matching solution IDs. This particular optimal solution can be found by a simple depth-first search based algorithm in linear time, starting at φ at the root of the parse tree. For each (u, ψ), we find a fitting pair of child formulas ρ_1, ρ_2 for child nodes v_1, v_2 of u, respectively, such that the assigned solution IDs match, process (v_1, ρ_1) and (v_2, ρ_2) independently, and output the associated subsolution at leaf nodes.

To solve 2-val(P), we adapt the evaluation structure. Instead of values in $\mathbb{R} \cup \{\infty\}$, we use pairs $(x, y) \in \mathcal{R} = (\mathbb{R} \cup \{\infty\})^2$, where x and y represents the values of an optimal and second-best solution. We define two new binary operators $+_2$ and $\min_2$ over $\mathcal{R}$, with $(x_1, y_1) +_2 (x_2, y_2) = (x_1 + x_2, \min(x_1 + y_2, x_1 + y_1))$ and $\min_2((x_1, y_1), (x_2, y_2)) = (a, b)$, where a, b are the smallest and second-smallest element of the multiset $\{x_1, y_1, x_2, y_2\}$, respectively.

▶ **Lemma 3.** *Let P be a LinCMS problem characterized by the formula $\varphi \in \Phi_R$ and cost functions c. Given a parse tree T of a hypergraph G, the CM algorithm solves 2-val(P) in linear time when used in conjunction with the evaluation structure $\langle \mathcal{R}, +_2, \mathit{min}_2, (0, 0), (\infty, \infty) \rangle$.*

Proof. For each leaf u of T, we solve 2-val(P) on $G(u)$ directly. Let $S^1 \in \boldsymbol{sat}(G, \varphi)$ be optimal. The mapping $v(G) = (c(S_1), \min(c(\boldsymbol{sat}(G, \varphi) \setminus \{S^1\})))$ can be written as $h(\boldsymbol{sat}(G, \varphi))$ with $h(A \uplus B) = h(A) \min_2 h(B)$ and $h(A \sqcup B) = h(A) +_2 h(B)$. The operators $\min_2$ and $+_2$ can be evaluated in constant time, resulting in linear total time using the CM algorithm. ◀

Solution IDs can be trivially generalized to the new evaluation structure, allowing us to refer to *the* optimal and second-best solution, and to reconstruct these two solutions in linear time.

▶ **Corollary 4.** *Let P be a LinCMS problem, and let $w \in \mathbb{N}$ be fixed. Given a parse tree T of a hypergraph G with bounded treewidth, we can solve 2-sol(P) on G in time $\mathcal{O}(|G|)$.*

4 Dynamizing the second-best solution

In this section, we introduce the evaluation tree data structure that stores intermediate results of the algorithm from the previous section. The data structure allows for a trivial query for the values of the two best solutions in constant time, and a query for the solutions themselves in linear time. We demonstrate how to perform a query for a pivot feature, and how to update an evaluation tree to match the two subproblems with respect to this pivot feature as described in Section 2.1. Both operations start in the root of the given tree, enabling us to use the path persistence technique described in Section 2.2.

An *evaluation tree* is a tree with the same structure as the parse tree. We store with every node of the evaluation tree the result of all evaluations of the CM algorithm as described in Section 3, as well as all solution ID information. In addition, we also store the solution sets $\boldsymbol{sat}(G(u), \psi)$ themselves for each leaf node u. We first describe the query and update procedures for problems characterized by CMS formulas with exactly one free variable. For this purpose, we identify solutions (S_1) with their first set S_1. For the sake of simplicity, we also identify nodes of the parse tree with their twins in the evaluation tree of the current subproblem. We assume that the hypergraph G is represented as a full parse tree T.

Let $\varphi \in \Phi_R$ be a CMS formula that characterizes a subproblem P, and let x be a pivot feature of P. If x is a source vertex of G, the two subproblems of P can again be characterized by formulas in Φ_R, namely $\varphi \wedge (x \in X_1)$ and $\varphi \wedge (x \notin X_1)$, respectively. However, if x is a hyperedge or a vertex that is not a source vertex of G, there are no such formulas in general. Therefore, not all subproblems can be characterized by a CMS formula, and we have to generalize the mapping $\boldsymbol{sat}$ to cover those subproblems. We define the set $\boldsymbol{sat}_P(G(u), \psi)$ to contain all solutions in $\boldsymbol{sat}(G(u), \psi)$ that satisfy the constraints imposed by the binary subproblem tree. Note that $h(\boldsymbol{sat}_P(G, \varphi))$ is the solution of 2-val(P).

Let S^1 and S^2 be the optimal and second-best solution of P, respectively. To find a pivot feature x for P, we maintain a current parse tree node u_j, two formulas ψ_j^1, ψ_j^2, and the invariant that the subsolutions $S^1(u_j)$, $S^2(u_j)$ of S^1, S^2 at u_j are in $\boldsymbol{sat}_P(G(u_j), \psi_j^1)$, $\boldsymbol{sat}_P(G(u_j), \psi_j^2)$, respectively, and that $S^1(u_j) \neq S^2(u_j)$. In the j-th iteration, we choose u_{j+1} to be a child node of u_j in the parse tree. Initially, u_1 is the root of T, and $\psi_1^1 = \psi_1^2 = \varphi$. The invariant holds trivially because of $h(\boldsymbol{sat}_P(G(u_1), \varphi)) = h(\boldsymbol{sat}_P(G, \varphi))$, and because the optimal solution differs from the second-best one by definition.

If u_j is a leaf, we construct the fixed-size subsolutions $S^1(u)$ and $S^2(u)$ explicitly. If the invariant holds, these solutions are distinct, and we can find a pivot feature in constant time. Otherwise, u_j is an inner node with children v_1 and v_2. Let ρ^1 (ρ^2) be a fitting pair of child formulas for ψ_j^1 (ψ_j^2) for which solution IDs match. If the invariant holds for j, the subsolutions $S^1(u_j)$ and $S^2(u_j)$ differ, so their subsolutions at v_1 ($S^1(v_1)$ and $S^2(v_1)$) or those at v_2 have to differ as well. We choose u_{j+1}, ψ_{j+1}^1 and ψ_{j+1}^2 accordingly. The invariant for $j+1$ then holds by construction.

▶ **Lemma 5.** *Given an evaluation tree of depth d for a subproblem P, the above algorithm finds a pivot feature for P in time $\mathcal{O}(d)$.*

Proof. Deferred to the full version. ◀

Next, we describe the update process to transform the evaluation tree for P into the evaluation tree for one of its two subproblems. The process is symmetric for both subproblems, so we only describe it for the subproblem P' that requires solutions to contain the pivot feature.

First, we execute the query algorithm to find a pivot feature x, and push each node it visits to a stack. The pivot query algorithm can only terminate at leaf nodes, so the last node u that is pushed to the stack is a leaf. Recall that for each leaf node u and each relevant formula ψ, we store $\boldsymbol{sat}_P(G(u), \psi)$ explicitly. We enumerate this solution set and remove every solution that does not contain x to obtain $\mathcal{S}(\psi) = \boldsymbol{sat}_{P'}(G(u), \psi)$. To re-evaluate ψ, we apply h to $\mathcal{S}(\psi)$.

Any other node u on the stack is an inner node of the full parse tree, and an ancestor of $u(x)$. Re-evaluation works the same as the original evaluation in the CM algorithm, by evaluating Equation (2) with operators $\min_2$, $+_2$ instead of min, + as in Lemma 3.

▶ **Lemma 6.** *Given an evaluation tree of depth d for a subproblem P, the above algorithm updates the evaluation tree to match a subproblem P' of P in time $\mathcal{O}(d)$.*

Proof. Deferred to the full version. ◀

Let S be the optimal or second-best solution of P, whichever remains feasible in P'. We do not require optimal solutions for subproblems to be unique, so we might find two optimal solutions for P' that both differ from S. Alternatively, S might turn up as the second-best solution for P'. Although these results would be valid outcomes of the update procedure, they

would break the k-best algorithm of Section 2.1: If we lose track of S by choosing two other solutions as optimal and second-best, we cannot detect whether we find S in a subproblem of P' again, leading to (the value of) S being output twice. This happens because solution IDs are not suitable to efficiently check two solutions with respect to different evaluation trees for equality. Therefore, we need to make sure that S is the optimal solution of P' as encoded by the solution IDs for P and P'. Fortunately, the update procedure can trivially enforce this property.

Now consider a problem characterized by N free variables, with $N > 1$. We may consider N to be constant, as it only depends on the problem definition. Recall that solutions S and S' are distinct when there is a discriminating index i with $S_i \neq S'_i$. The pivot query algorithm has to take this into account. For inner nodes, it relies only on solution IDs to choose a successor of the current parse tree node. Leaf nodes still have a fixed number of feasible solutions for a fixed-size hypergraph. The same arguments as above yield running time $\mathcal{O}(d)$. Similarly, we only have to adapt the processing of nodes for the update algorithm. For $N = 1$, we always imposed the new subproblem constraint on the first and only set of a solution, because the discriminating index was always the same. Now, we have to take into account the discriminating index as determined during the pivot query phase. Using the same arguments, updates can still be performed in time $\mathcal{O}(d)$. We now state our main result.

▶ **Theorem 7.** *Let P be a LinCMS problem. Then computing the values of the k best solutions for P on a graph G requires $\mathcal{O}(|G| + k \log |G|)$ time and space.*

Proof. In linear time, we compute a shallow tree decomposition, transform it into a parse tree T of depth $\mathcal{O}(\log |G|)$, and apply the CM algorithm to T by Lemma 3. With the results, we initialize a binary subproblem tree. We perform a best-first search to find the k best subproblems with respect to their second-best solution as described in Section 2.1, which requires us to solve $\mathcal{O}(k)$ subproblems. Using the persistent tree technique from Section 2.2 in conjunction with the update operation above, we need $\mathcal{O}(d)$ additional time and space per subproblem. ◀

5 Fixed numbers of solutions

We generalize here our algorithm for the second best solution to any fixed number of solutions, and to low-space and parallel complexity classes. Similar results could be obtained by expressing the k-best solution problem (for a constant k) in MSO with k set variables, all constrained to be solutions, to be distinct, and with minimum total weight; however, this would give a significantly worse dependence on k than our solution and would require additional complication to recover the differences between the solutions.

For fixed $k \in \mathbb{N}$, operators $\min_k$ and $+_k$ can be defined analogously to $\min_2$ and $+_2$, respectively. Evaluating $\min_k(x, y)$ and $x +_k y$ for k-tuples $x, y \in \mathbb{R}^k$ still requires constant time. Hence, the CM algorithm with the evaluation structure $\langle (\mathbb{R} \cup \infty)^k, +_k, \min_k, (0, \ldots, 0), (\infty, \ldots, \infty) \rangle$ solves k-val(P) for any problem P in LinCMS. Let T be a parse tree with depth d, and let $\leq_{\text{post}}$ be a postordering of T. To solve k-val(P), we evaluate the nodes of T according to $\leq_{\text{post}}$. As soon as a parse tree node has been evaluated, the evaluations of its children can be dropped. Thus, the number of stored evaluations never exceeds $d + 1$.

▶ **Theorem 8.** *Let P be a LinCMS problem, and let $k, w \in \mathbb{N}$ be fixed. Given a graph G with treewidth w, the problem k-val(P) on G can be solved using logarithmic memory space.*

Proof. A shallow tree decomposition T can be computed with logarithmic memory space [13]. We use a depth-first search from an arbitrarily chosen root node to process the bags of T in

postorder, processing each bag by replacing it with its fixed-size portion of the parse tree, which is then evaluated bottom-up. The evaluations of all parse tree nodes corresponding to a bag require constant space, and we store $\mathcal{O}(\log |G|)$ of them at a time. ◀

In the PRAM model (see e.g. [1]), the CM algorithm can be parallelized as follows.

▶ **Theorem 9.** *Let P be a LinCMS problem, and let $k, w \in \mathbb{N}$ be fixed. In the EREW PRAM model, given a shallow tree decomposition T of graph G with treewidth w, the problem k-val(P) on G can be solved in time $\mathcal{O}(\log |G|)$ by $\mathcal{O}(|G|)$ processors.*

Proof. We allocate one processor $p(u)$ for each node u of T, which computes the portion of the parse tree corresponding to its bag and evaluates all nodes of that portion. Processor $p(u)$ waits until all processors $p(v)$ of child nodes v of u have finished. The processor of the root node of T therefore waits $\mathcal{O}(\log |G|)$ time. Only processor $p(u)$ writes solutions for node u, and only the parent u' of u reads them, according to the EREW model. ◀

Finally, using the algorithm of Bodlaender [5] on $\mathcal{O}(|G|^{3w+4})$ processors to compute a shallow tree decomposition, we obtain the following.

▶ **Corollary 10.** *Let P be a LinCMS problem, and let $k, w \in \mathbb{N}$ be fixed. In the CRCW PRAM model, given a graph G with treewidth w, the problem k-val(P) on G can be solved in time $\mathcal{O}(\log |G|)$ by $\mathcal{O}(|G|^{3w+4})$ processors.*

References

1 Selim G. Akl. *Design and analysis of parallel algorithms.* Prentice Hall, 1989.

2 Stefan Arnborg, Bruno Courcelle, Andrzej Proskurowski, and Detlef Seese. An algebraic theory of graph reduction. *J. ACM*, 40(5):1134–1164, 1993. doi:10.1145/174147.169807.

3 Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, 1991. doi:10.1016/0196-6774(91)90006-K.

4 Michel Bauderon and Bruno Courcelle. Graph expressions and graph rewritings. *Mathematical Systems Theory*, 20(2-3):83–127, 1987. doi:10.1007/BF01692060.

5 Hans L. Bodlaender. Nc-algorithms for graphs with small treewidth. In Jan van Leeuwen, editor, *Graph-Theoretic Concepts in Computer Science, 14th International Workshop, WG '88, Amsterdam, The Netherlands, June 15-17, 1988, Proceedings*, volume 344 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1988. doi:10.1007/3-540-50728-0_32.

6 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996. doi:10.1137/S0097539793251219.

7 Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. *SIAM J. Comput.*, 27(6):1725–1746, 1998. doi:10.1137/S0097539795289859.

8 Chandra R. Chegireddy and Horst W. Hamacher. Algorithms for finding k-best perfect matchings. *Discrete Applied Mathematics*, 18(2):155–165, 1987. doi:10.1016/0166-218X(87)90017-5.

9 Bruno Courcelle. Graph rewriting: An algebraic and logic approach. In *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 193–242. Elsevier, 1990.

10 Bruno Courcelle and Mohamed Mosbah. Monadic second-order evaluations on tree-decomposable graphs. *Theor. Comput. Sci.*, 109(1&2):49–82, 1993. doi:10.1016/0304-3975(93)90064-Z.

11 Edo S. Van der Poort, Marek Libura, Gerard Sierksma, and Jack A. A. van der Veen. Solving the k-best traveling salesman problem. *Computers & OR*, 26(4):409–425, 1999. doi:10.1016/S0305-0548(98)00070-7.

12 James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989. doi:10.1016/0022-0000(89)90034-2.

13 Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of bodlaender and courcelle. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 143–152. IEEE Computer Society, 2010. doi:10.1109/FOCS.2010.21.

14 David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1998. doi:10.1137/S0097539795290477.

15 David Eppstein. K-best enumeration. *Bull. EATCS*, 115, 2015. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/322.

16 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997. doi:10.1145/265910.265914.

17 S. Feferman and R. L. Vaught. The first order properties of products of algebraic systems. *Fund. Math.*, 47:57–103, 1959.

18 Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993. doi:10.1006/inco.1993.1030.

19 Harold N. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Comput.*, 6(1):139–150, 1977. doi:10.1137/0206011.

20 H. W. Hamacher and M. Queyranne. K best solutions to combinatorial optimization problems. *Ann. Oper. Res.*, 4(1-4):123–143, 1985. doi:10.1007/BF02022039.

21 Naoki Katoh, Toshihide Ibaraki, and Hisashi Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982. doi:10.1002/net.3230120406.

22 Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, October 23-26, 2010, Las Vegas, Nevada, USA*, pages 645–654. IEEE Computer Society, 2010. doi:10.1109/FOCS.2010.67.

23 Jin Y. Yen. Finding the K shortest loopless paths in a network. *Manag. Sci.*, 17(11):712–716, 1971. URL: http://www.jstor.org/stable/2629312.