Log-space Algorithms for Paths and Matchings in k-trees

Bireswar Das, Samir Datta, Prajakta Nimbhorkar

Symposium on Theoretical Aspects of Computer Science 2010 (Nancy, France), pp. 215-226
www.stacs-conf.org

LOG-SPACE ALGORITHMS FOR PATHS AND MATCHINGS IN k-TREES

BIRESWAR DAS [1] AND SAMIR DATTA [2] AND PRAJAKTA NIMBHORKAR [1]

[1] The Institute of Mathematical Sciences
Chennai, India
E-mail address: {bireswar,prajakta}@imsc.res.in

[2] Chennai Mathematical Institute, Chennai, India
E-mail address: sdatta@cmi.ac.in

Abstract. Reachability and shortest path problems are NL-complete for general graphs. They are known to be in L for graphs of tree-width 2 [14]. However, for graphs of tree-width larger than 2, no bound better than NL is known. In this paper, we improve these bounds for k-trees, where k is a constant. In particular, the main results of our paper are log-space algorithms for reachability in directed k-trees, and for computation of shortest and longest paths in directed acyclic k-trees.

Besides the path problems mentioned above, we consider the problem of deciding whether a k-tree has a perfect macthing (decision version), and if so, finding a perfect matching (search version), and prove that these problems are L-complete. These problems are known to be in P and in RNC for general graphs, and in SPL for planar bipartite graphs [8].

Our results settle the complexity of these problems for the class of k-trees. The results are also applicable for bounded tree-width graphs, when a tree-decomposition is given as input. The technique central to our algorithms is a careful implementation of divide-and-conquer approach in log-space, along with some ideas from [14] and [19].

1. Introduction

Reingold's striking result [21], showed that undirected reachability is in L, thus collapsing the class SL to L. On the other hand, directed reachability, which happens to be NL-complete is another similar sounding problem for which there is only partial progress to report. A result of Allender and Reinhardt, [22] hints at a partial collapse of NL by showing that directed reachability is in the formally smaller class UL, although, *non-uniformly*.

In the absence of better constructive upper bounds it is natural to consider natural restrictions on graphs which allow us to improve the upper bounds on reachability and related problems. Typical examples of this approach are [1],[23], where the complexity of various versions of planar and somewhat non-planar (in the sense of excluding only a K_5 or only a $K_{3,3}$ minor) are considered. In the same spirit, but using different techniques, [14]

1998 ACM Subject Classification: Computational Complexity.
Key words and phrases: k-trees, reachability, matching, log-space.

considers reachability and related questions in series-parallel graphs and places all of these in L. They leave open the question of complexity of such problems in bounded tree-width graphs. Series-parallel graphs have tree-width two and happen to be planar. But higher tree widths graphs are highly non-planar. In fact, any k-tree for $k > 4$ contains both K_5 and $K_{3,3}$.

We resolve the open questions posed in [14] and show a matching L lower bound to complete the characterization of reachability problems in k-trees. Thus one of the main results of our paper is the following:

Theorem 1.1. *The following problems are* L*-complete:*

1. *Computing reachability between two vertices in directed k-trees,*
2. *Computing shortest and longest paths in directed acyclic k-trees.*

In this paper, we also consider the perfect matching problem. The parallel complexity of perfect matching problems is a long standing open problem where the best known algorithms use randomness as a resource [20],[15]. Even in the planar case, the search problem for perfect matchings is known to be in NC for bipartite graphs only [8].

We prove a complete characterization for the decision and search versions of the perfect matching problem for k-trees. This improves significantly upon previous best known upper bound of LogCFL for bounded tree-width graphs. Thus another main result of our paper is:

Theorem 1.2. *Deciding whether a k-tree has a perfect matching, and if so, finding a perfect matching is* L*-complete.*

Our primary technique is a careful use of divide-and-conquer to enable the algorithm to run in L. However, for the distance computation we need to import a constructive version of tree separation from [19] where it is stated in the context of Visibly Pushdown Automata (VPAs). We believe that porting this technique for use in general log-space computation is an important contribution of this paper.

At this point, we must mention an important caveat. All our log-space results hold directly only for k-trees and not for partial k-trees which are also equivalent to tree-width k graphs. The reason being that a tree decomposition for partial k-trees is apparently more difficult to construct (best known upper bound is LogCFL[24]) as opposed to k-trees (for which it can be done in L [17]). Having mentioned that it is important to observe that if we are given the tree decomposition of a partial k-tree, we can do the rest of computation in L.

The rest of the paper is organized as follows: Section 2 gives the necessary background. Section 3 contains log-space algorithms for reachability in directed k-paths and k-trees. Section 4 contains log-space algorithms for shortest and longest path in directed acyclic k-paths and k-trees. Section 5 contains log-space algorithms for perfect matching problems in a k-tree.

2. Preliminaries

We define k-trees and a subclass of k-trees known as k-paths here, and also describe a suitable representation for the graphs in these two classes. This representation is used in our algorithms in the rest of the paper. All the definitions given here are applicable to both directed as well as undirected graphs. For directed graphs, the directions of the

edges can be ignored while defining k-trees and k-paths and while computing their suitable representations.

The class of graphs known as k-trees is defined as (cf. [12]):

Definition 2.1. The class of *k-trees* is inductively defined as follows.

- A clique with k vertices (k-clique for short) is a k-tree.
- Given a k-tree G' with n vertices, a k-tree G with $n+1$ vertices can be constructed by picking a k-clique X (called the *support*)in G' and then joining a new vertex v to each vertex u in X. Thus, $V(G) = V(G') \cup \{v\}$, $E(G) = E(G') \cup \{\{u, v\} \mid u \in X\}$.

A *partial k-tree* is a subgraph of a k-tree. The class of partial k-trees coincides with the class of graphs that have tree-width at most k. k-trees are recognizable in log-space [2] but partial k-trees are not known to be recognizable in log-space. In literature, several different representations of k-trees have been considered [10, 2, 17]. We use the following representation given by Köbler and Kuhnert [17]:

Definition 2.2. Let $G = (V, E)$ be a k-tree. The tree representation $T(G)$ of G is defined by

$$V(T(G)) = \{M \subseteq V \mid M \text{ is a } k\text{-clique or a } (k+1)\text{-clique}\},$$
$$E(T(G)) = \{\{M_1, M_2\} \subseteq V \mid M_1 \subsetneq M_2\}$$

In [17], it is proved that $T(G)$ is a tree and can be computed in log-space. In the rest of the paper, we use G in place of $T(G)$. Thus, by *a k-tree G*, we always mean that G is in fact represented as $T(G)$. The term *vertices in G* refers to the vertices in the original graph, whereas *a node in G* and *a clique in G* refer to the nodes of $T(G)$. Partial k-trees also have a tree-decomposition similar to that of k-trees, which is also not known to be log-space computable.

k-paths is a sub-class of k-trees (e.g. see [11]). The recursive definition of k-paths is similar to that of k-trees. However, a new vertex can be added only to a particular clique called the *current clique*. After addition of a vertex, the current clique may remain the same, or may change by dropping a vertex and adding the new vertex in the current clique. We consider the following representation of k-paths, which is based on the recursive definition of k-paths, and is known to be computable in log-space [2]:

Given a k-path $G = (V, E)$, for $i = 1, \cdots, m$, let X_i be the current cliques at the ith stage of the recursive construction of the k-path. Let $V_1 = \cup_i X_i$ and $V_2 = V \setminus V_1$. We call the vertices in V_2 as *spikes*. The following facts are easy to see:

1. No two spikes have an edge between them.
2. Each spike is connected to all the vertices of exactly one of the X_i's.
3. X_i and X_{i+1} share exactly $k-1$ vertices

The representation of G consists of a graph $G' = (V', E')$ where $V' = \{X_1, \ldots, X_m\} \cup V_2$ and $E' = \{(X_i, X_{i+1}) \mid 1 \leq i < m\} \cup \{(X, v) | X \text{ is a clique in } \in V', v \in V_2 \text{ has a neighbour in } X\}$.

3. Reachability

We give log-space algorithms to compute reachability in k-paths and in k-trees. Although the graphs considered in this section are directed, when we refer to any of the definitions or decompositions in Section 2, we consider the underlying undirected graph.

3.1. Reachability in k-paths

Without loss of generality, we can assume that s and t are vertices in some k-cliques X_i and X_j, and not spikes. If s (t) is a spike, then it has at most k out-neighbors (resp. in-neighbors) and we can take one of the out-neighbors (resp. in-neighbors) as the new source s' and new sink t' and check reachability. As there are only k^2 such pairs, we can cycle through all of them in log-space. The algorithm is based on the observation that a simple s to t path ρ can pass through any clique at most k times. We use a divide- and-conquer approach similar to that used in Savitch's algorithm (which shows that directed reachability can be computed in $DSPACE(\log^2 n)$). The main steps involved in the algorithm are as follows:

1. Preprocessing step: Make the cliques disjoint by labeling different copies of each vertex with different labels and introducing appropriate edges. Compute reachabilities within each clique including its spikes, and *remove the spikes*. Number the cliques $X_1, \ldots, X_m$ left to right.

2. Now assume that s and t are in cliques X_i and X_j respectively. Note that $i = j$ is also possible, but without loss of generality, we can assume $i < j$. This is because, if $i = j$, we can make another copy X_i' of X_i, join the copies of the same vertex by bidirectional edges to preserve reachabilities, and choose the copy of s from X_i and that of t from X_i'.

3. Divide the k-path into three parts P_1, P_2 and P_3 where P_1 consists of cliques $X_1, \ldots, X_i$, P_2 consists of $X_i, \ldots, X_j$, and P_3 consists of $X_j, \ldots, X_m$. Note that X_i (X_j) appears in both P_1 and P_2 (P_2 and P_3 respectively). Now compute reachabilities of all pairs of vertices in X_i (X_j) when the graph is restricted to P_1 (respectively P_3). Then the reachability of t from s within P_2 is computed, using the previously computed reachabilities within P_1 and P_3.

Each of these steps can be done by a log-space transducer. The details are given below.

Preprocessing: Although adjacent k-cliques in a k-path decomposition share $k-1$ vertices, we perform a preprocessing step, where we give distinct labels to each copy of a vertex. As all the copies of a vertex form a (connected) sub-path in the k-path decomposition, we join two copies of a vertex appearing in two adjacent cliques by bidirectional edges. It can be seen that this preserves reachabilities. Any copy of s and t can be taken as the new s and t. Another preprocessing step involves removing the spikes maintaining reachabilities between all pairs of vertices, and computing reachabilities within each k-clique. Both of these preprocessing steps can be done by a log-space transducer. The proof appears in the full version of the paper.

The Algorithm: We describe an algorithm to compute pairwise reachabilities in X_i and X_j in P_1 and P_3 respectively, and also s-t reachability in P_2 using these previously computed pairwise reachabilities. Algorithm 1 describes this reachability routine. The routine gets as input two vertices u and v, and two indices i and j. It determines whether v is reachable from u in the sub-path $P = (X_i, \ldots, X_j)$. This input is given in such a way that u and v always lie in X_i or X_j. Consider the case when both u and v are in X_i (or both in X_j). Let l be the center of P. Then a path from u to v either lies entirely in the sub-path $P' = (X_i, \ldots, X_l)$ or it crosses X_l at most k times. Thus if $X_l = \{v_1, \ldots, v_k\}$ then for $\{v_{i_1}, \cdots, v_{i_r}\} \subseteq X_l$ we need to check reachabilities between u and say v_{i_1} in P', then between v_{i_1} and v_{i_2} in $P'' = (X_l, \ldots, X_j)$ and so on, and finally between v_{i_r} and v in P'. It suffices to check all the r-tuples in X_l, where $0 \leq r \leq k$. The case when $u \in X_i$ and $v \in X_j$ (and vice versa) is analogous. In Algorithm 1, we present only one case where

$u, v \in X_i$. Other three cases are analogous. Thus at each recursive call, the length of the sub-path under consideration is halved, and $O(\log m)$ iterations suffice. The algorithm can

Algorithm 1 Procedure IsReach(u, v, i, j)

```
Input: Pre-processed k-path decomposition of graph G, clique indices i, j, vertex labels
  u, v ∈ X_i. {Other three cases are analogous.}
Decide: Whether v is reachable from u in sub-path P = (X_i, ..., X_j).
if j − i = 1 then
   Compute the reachability directly, as the sub-path has only 2k vertices.
   Return the result.
end if
l = (j+i)/2
if u, v ∈ X_i then
   if IsReach(u, v, i, l) then
      Return 1;
   else
      for q = 1 to k do
         v_0 ← u, v_{q+1} ← v
         for all q-tuples (v_1, ..., v_q) of vertices in X_l do
            if ⋀_{x=0, x even}^{q+1} IsReach(v_x,v_{x+1},i,l) ∧ ⋀_{x=1, x odd}^{q+1} IsReach(v_x,v_{x+1},l,j) then
               Return 1;
            end if
         end for
      end for
   end if
end if
```

be implemented in log-space. The correctness and complexity analysis of the algorithm appears in the full version.

3.2. Reachability in k-trees

Given a directed k-tree G in its tree decomposition and two vertices s and t in G, we describe a log-space algorithm that checks whether t is reachable from s. This algorithm uses Algorithm 1 as a subroutine and involves the following steps: The complexity analysis is given in Lemma 3.1.

1. **Preprocessing:** Like k-paths, assign distinct labels to the copies of each vertex u in different cliques. Introduce a bidirectional edge between the copies of u in all the adjacent pairs of cliques. As reachabilities are maintained during this process, any copy of s and t can be taken as the new s and t. Let X_i and X_j be the cliques containing s and t respectively.

2. **The Procedure:** After this preprocessing, we have a tree T with its nodes as disjoint k-cliques of vertices of G, and s and t are contained in cliques X_i and X_j. Compute the unique undirected path ρ between X_i and X_j in T in log-space. Each node on ρ has two of its neighbors on ρ, except X_i and X_j, which have one neighbor each. An s to t path has to cross each clique in ρ, and additionally, it can pass through the subtrees attached to

each node X_l on ρ. Hence for each node X_l on ρ, we pre-compute the pairwise reachabilities among the k vertices contained in X_l when the k-tree is restricted to the subtree rooted at X_l. We define the subtree rooted at X_l as the subtree consisting of X_l and those nodes which can be reached from X_l without going through any node on ρ. Note that once this is done for each node X_l on ρ, we are left with ρ. As ρ is a k-path, we can use Algorithm 1 in Section 3.1 to compute reachabilities within ρ.

3. **Computing reachabilities within the subtree rooted at X_l:** We do this inductively. If the subtree rooted at X_l contains only one node X_l, we have only k vertices, and their pairwise reachabilities within X_l can be computed in $O(k \log k)$ space. We recursively find the reachabilities within the subtrees rooted at each of the children of X_l. Let the size of the subtree rooted at X_l be N. At most one of the children of X_l can have a subtree of size larger than $\frac{N}{2}$. Let X_a be such a child. Recursively compute the pairwise reachabilities for each pair of vertices in X_a within the subtree rooted at X_a. The reachabilities are represented as a $k \times k$ boolean matrix referred to as the *reachability matrix* M for the vertices in X_a, when the graph is confined to the subtree rooted at X_a. M is then used to compute the pairwise reachabilities of vertices in X_l, when the graph is confined to X_l and the subtree rooted at X_a. This gives a new matrix M' of size k^2. It is stored on stack while computing the reachability matrix M'' for another child X_b of X_l. The matrix M' is updated using M'', so that it represents reachabilities between each pair of vertices in X_l when the graph is confined to X_l and the subtrees rooted at X_a and X_b. This process is continued till all the children of X_l are processed. The matrix M' at this stage reflects the pairwise reachabilities between vertices of X_l, when the graph is confined to the subtree rooted at X_l. Note that the storage required while making a recursive call is only the current reachability matrix M'. Recall that M' contains the pairwise reachabilitities among the vertices in X_l in the subgraph corresponding to X_l and the subtrees rooted at those children of X_l which are processed so far. We give the complexity analysis in the full version.

Lemma 3.1. *The procedure described above can be implemented in log-space.*

Hardness for L: L-hardness of reachability in k-trees follows from L-hardness of the problem of path ordering (proved to be SL-hard in [9], and is L-hard due to SL=L result of [21]). We give the details in the full version.

4. Shortest and Longest Paths

We show that the shortest and longest paths in weighted directed acyclic k-trees can be computed in log-space, when the weights are positive and are given in unary. Throughout this section, the terms k-path and k-tree always refer to directed acyclic k-paths and k-trees respectively, with integer weights on edges and we here onwards omit the specification *weighted directed acyclic*. We use the following (weighted) form of the result from [18]: The proof is exactly similar to that in [18] and we omit it here.

Theorem 4.1 (See[18], Theorem 9)**.** *Let $\mathcal{C}$ be any subclass of weighted directed acyclic graphs closed under vertex deletions. There is a function f, computable in log-space with oracle access to* Reach($\mathcal{C}$)*, that reduces* Distance($\mathcal{C}$) *to* Long-Path($\mathcal{C}$) *and* Long-Path($\mathcal{C}$) *to* Distance($\mathcal{C}$)*, where* Reach($\mathcal{C}$)*,* Distance($\mathcal{C}$)*, and* Long-Path($\mathcal{C}$) *are the problems of deciding reachability, computing distance and longest path respectively for graphs in $\mathcal{C}$.*

We use this theorem to reduce the shortest path problem in k-trees to the longest path problem, and then compute the longest (that is, maximum weight) s to t path. The reduction involves changing the weights of the edges such that the shortest path becomes the longest path and vice versa. This gives a directed acyclic k-tree with positive integer weights on edges given in unary. The class of k-trees is not closed under vertex deletions. However, once a tree decomposition of a k-tree is computed, deleting vertices from the cliques leaves some cliques of size smaller than k, which does not affect the working of the algorithm.

We show that the maximum weight of an s to t path can be computed in log-space using a technique which uses ideas from [14]. The algorithm to compute maximum weight s to t path in k-trees uses the algorithm for computing maximum weight path in k-paths as subroutine. Therefore we first describe the algorithm for k-paths in Section 4.1

4.1. Maximum Weight Path in Directed Acyclic k-paths

Let G be a directed acyclic k-path and s and t be two designated vertices in G. The computation of maximum weight of an s to t path is done in five stages, described below in detail. The main idea is to obtain a log-depth circuit by a suitable modification of Algorithm 1, and to transform this circuit to an arithmetic formula over integers, whose value is used to compute the maximum weight of an s to t path in G.

Computing the maximum weight s to t path in G involves the following steps:

(1) **Construct a log-depth formula from Algorithm 1:** Modify Algorithm 1 so that it outputs a circuit $\mathcal{C}$ that has nodes corresponding to the recursive calls made in Line 15 and the tuples considered in the **for** loop in Line 14. A node q in $\mathcal{C}$ that corresponds to a recursive call IsReach(u, v, i, j) has children $q_1, \cdots, q_N$, which correspond to the tuples considered in that recursive call (**for**-loop on Line 12 of Algorithm 1). We refer to q as a *call-node* and $q_1, \ldots, q_N$ as *tuple-nodes*. A tuple-node q' corresponding to a tuple $(v_1, \ldots, v_N)$ has call-nodes $q'_1, \ldots, q'_N$ as its children, which correspond to the recursive calls made while considering the tuple $(v_1, \ldots, v_N)$ (Line 15 of Algorithm 1). The leaves of $\mathcal{C}$ are those recursive calls which satisfy the **if** condition on Line 3 of Algorithm 1, thus they are always call-nodes. As the depth of the recursion in Algorithm 1 is $O(\log n)$, the circuit $\mathcal{C}$ also has $O(\log n)$ depth. Hence it can be converted to a formula $\mathcal{F}$ by only a polynomial factor blow-up in its size. The maximum number of children of a node is $O(k^k)$ and hence the size of $\mathcal{F}$ is bounded by $O(k^{k \log n})$, which is polynomial in n for constant k.

(2) **Prune the boolean formula:** The internal call-nodes of $\mathcal{F}$ are replaced by $\vee$ gates and tuple-nodes are replaced by $\wedge$ gates. The leaves of $\mathcal{F}$ are replaced by 0 or 1 depending on whether the corresponding recursive call returned 0 or 1 in the **if** block on Line 3 of Algorithm 1. It can be seen that a sub-formula of $\mathcal{F}$ rooted at a call-node evaluates to 1 if and only if the corresponding recursive call returns 1 in Algorithm 1. Similarly, the sub-formula rooted at a tuple-node evaluates to 1 if and only if the conjunction corresponding to it (on Line 15 of Algorithm 1) evaluates to 1. Now, we evaluate the sub-formula rooted at each node of $\mathcal{F}$. Note that a node that evaluates to 0 does not contribute to any path from s to t, and hence its subtree can be safely removed.

(3) **Transformation into a $\{+, max\}$-tree:** The new, pruned formula obtained in Step 2 is then relabeled: Each $\wedge$ label is replaced with a $+$ label and each $\vee$ label with a max label. Each leaf corresponds to calls of the form $IsReach(u, v, i, i+1)$. It is labeled with the length of the maximum weight u to v path confined within cliques i and $i+1$, which can be computed in $O(1)$ space. This weight is strictly positive, since the 0-weight leaves are removed in Step 2. Further, all the weights are in unary. Thus we now have a $\{+, max\}$-tree T with positive, unary weights on its leaves. It is easy to see that the value of the $\{+, max\}$-tree T is the maximum weight of any s to t path in G.

(4) **Transformation into a $\{+, \times\}$-tree:** The evaluation problem on the $\{+, max\}$-tree T obtained in Step 3 is then reduced to the evaluation problem on a $\{+, \times\}$-tree T' whose leaves are labeled with positive integer weights coded in binary. This reduction works in log-space and is similar to that of [14]. The reduction involves replacing a $+$-node of T with a $\times$-node, and a max-node with a $+$ node. The weight w of a leaf is replaced with r^{mw}, where r is the smallest power of 2 such that $r \geq n$, and m is the sum of the weights of all the leaves of T plus one. The correctness of the reduction follows from a similar result in [14], and we omit the proof here.

(5) **Evaluation of the $\{+, \times\}$ tree:** This can be done in log-space due to [5, 3, 7, 13]. The value of T is $v = \lfloor \frac{log_r v'}{m} \rfloor$.

4.2. Maximum Weight Path in Directed Acyclic k-trees

Given a directed acyclic k-tree (in its tree-decomposition) G, two vertices s and t in G, and weights on the edges of G, encoded in unary, we show how to compute the maximum weight of an s to t path in G. Unlike the case of k-paths, the reachability algorithm for k-trees given in Section 3.2 can not be used to get a log-depth circuit since the recursion depth of the algorithm is same as the depth of the k-tree. Therefore we need to find another way of recursively dividing the k-tree into smaller and smaller subtrees, as we did for k-paths in Sections 3.1 and 4.1. This is based on the technique used in the following result of [19]:

Lemma 4.2. *(***Lemma** 6 **of** [19]**, also see** [4]*) Let M be a visibly pushdown automaton accepting well-matched strings over an alphabet Δ. Given an input string x, checking whether $x \in L(M)$ can be done in log-space.*

Using Lemma 4.2, we can compute *a set of recursive separators for a tree* defined below:

Definition 4.3. Given a rooted tree T, separators of T are two nodes a and b of T such that

1. The subtrees rooted at a and b respectively are disjoint,

2. T is split into subtrees T_1, T_2, T_3 where T_1 consists of a, some (or possibly all) of the children of a, and subtrees rooted at them, T_2 is defined similarly for b, and T_3 consists of the rest of the tree along with a copy of a and b each.

3. Each of T_1, T_2, T_3 consists of at most a $\frac{3}{4}$ fraction of the leaves of T.

This process is done recursively for T_1, T_2, T_3, until the number of leaves in the subtrees is two. Such a subtree is in fact a path. A *set of recursive separators of T* consists of the separators of T and of all the subtrees obtained in the recursive process.

The following lemma gives the procedure to compute a set of recursive separators of a tree T:

Lemma 4.4. *Given a tree T, the set of recursive separators of T can be computed in log-space.*

Proof. The algorithm of [19] deals with well-matched strings. An example of a well-matched string is a balanced parentheses expression, which is a string over $\{(,)\}$. In [19], a log-space algorithm is given for membership testing in those languages which are subsets of well-matched strings and are accepted by visibly pushdown automata. We restrict ourselves to balanced parentheses expressions. To check whether a string on parentheses is in the language, the algorithm of [19] recursively partitions the string into three disjoint substrings, such that each of the parts forms a balanced parentheses expression, and length of each part is at most $\frac{3}{4}$th of the length of the original string. To use this algorithm, we order the children of each node of T in a specific way, label the leaves with parentheses '$($' and '$)$' such that the leaves of the subtree rooted at any internal node form a string on balanced parentheses. We add dummy leaves if needed. The steps are as follows:

1. By adding dummy leaves, ensure that each internal node has an even number of children which are leaves, and there are at least two such children.

2. Arrange the children of each node from left to right such that the non-leaves are consecutive, and they have an equal number of leaves to the left and to the right.

3. For each internal node, label the left half of its leaf-children with '(' and the right ones by ')'. This ensures that the leaves of the subtree rooted at each internal node form a balanced parentheses expression. Conversely, leaves which form a balanced parentheses expression are consecutive leaves in the subtree rooted at an internal node.

The leaves of T now form a balanced parentheses expression, and we run the algorithm of [19] on this string. The recursive splitting of the string into smaller substrings corresponds to the recursive splitting of T at some internal nodes, which satisfies Definition 4.3. This is ensured by the way the leaves are labeled. Each balanced parentheses expression corresponds to either a subtree rooted at an internal node or the subtrees rooted at some of the children of an internal node.

The subtrees obtained by splitting a tree have at most $\frac{3}{4}$th of the number of leaves in the tree. Thus at each stage of recursion, the number of leaves in the subtrees is reduced by a constant fraction. Moreover, the algorithm of [19] can output all the substrings formed at each stage of recursion in log-space. As a substring completely specifies a subtree of T, our procedure outputs the set of recursive separators for T in log-space. ∎

Once an algorithm to compute the set of recursive separators for k-trees is known, a reachability routine similar to Algorithm 1 can be designed in a straight forward way. We give the details in the full version. From the reachability routine, the computation of maximum weight path follows from the steps 1 to 5 described in Section 4.1.

4.3. Distance Computation in Undirected k-trees

We give a simple log-space algorithm for computing the shortest path between two given vertices in an undirected k-tree. We use the decomposition of [16], where a k-tree is decomposed into layers. We use the following properties of the decomposition:

1. Layer 0 is a k-clique. Each vertex in layer $i > 0$ has exactly k neighbors in layers $j < i$. Further, these neighbors of i which are in layers lower than that of i form a k-clique.

2. No two vertices in the same layer share an edge.

This decomposition is log-space computable [17]. Moreover, given two vertices s and t, it is always possible to find a decomposition in which t lies in layer 0. This can also be done in log-space. If both s and t are in layer 0, then there is an edge between s and t, which is the shortest path from s to t. Therefore assume that s lies in a layer $r > 0$. The following claim leads to a simple algorithm. The proof appears in the full version.

Claim 4.5. 1. The shortest s to t path never passes through two vertices u and v such that $layer(u) < layer(v)$. 2. There is a shortest path from s to t passing through the neighbor of s in the lowest layer.

This claim suggests a simple algorithm which can be implemented in log-space: Start from s and choose the next vertex from the lowest possible layer, at each step till we reach layer 0.

5. Perfect Matching in k-trees

Hardness for L: To show that the decision version of perfect matching is hard for L, we show that the problem of path ordering, can be reduced to the perfect matching problem for k-trees. We give the proof in the full version:

Lemma 5.1. *Determining whether a k-tree has a perfect matching is* L*-hard.*

L upper bounds: We describe a log-space algorithm to decide whether a k-tree has a perfect matching and, if so, output a perfect matching. The algorithm is inspired by an $O(n^3)$ algorithm [6] for computing the matching polynomial in series-parallel graphs. The idea is to exploit the fact that k-trees have a tree decomposition of bounded width, so that any perfect matching of the entire k-tree induces a partial matching on any subtree which leaves at most constantly many vertices unmatched. Thus we generalize the problem to that of determining, for each set, S, of constantly many vertices in the root of the subtree, whether there is a matching of the subtree that leaves exactly the vertices in S unmatched. Now we "recursively" solve the generalized problem and for this purpose we need to maintain a bit-vector indexed by the sets S which is still of bounded length. The algorithm composes the bit-vectors of the children of a node to yield the bit-vector for the node. The bit-vector, which we refer to as *matching vector*, is defined as follows:

Definition 5.2. Let G be a k-tree with tree-decomposition T. T has alternate levels of k-cliques and $k+1$-cliques. Root T arbitrarily at a k-clique. Let s be a node in T that shares vertices $\{u_1, \ldots, u_k\}$ with its parent. Further, let H be the subgraph of G corresponding to the subtree of T rooted at s. The *matching vector* for s is a vector $\vec{v}_H = (v_H^{(S_1)}, \ldots, v_H^{(S_{2^k})})$ of dimension 2^k, where $S_1, \ldots, S_{2^k}$ are all the distinct subsets of $\{u_1, \ldots, u_k\}$, and $v_H^{(S_i)} = 1$ if H has a matching in which all the vertices of H matched, except those in S_i, $v_H^{(S_i)} = 0$ if there is no such matching.

It can be seen that G has a perfect matching if and only if $v_G^{(\emptyset)} = 1$.We show how to compute $\vec{v}_G$ in L, and also show how to construct a perfect matching in G, if one exists. We prove Part 1 of the following theorem. For a proof of part 2, we refer to the full version.

Theorem 5.3. *1. The problem of deciding whether a k-tree has a perfect matching is in* L.
2. Finding a perfect matchings in a k-tree is in FL.

Proof. (**of** 1) We compute the matching vector for the root by recursively computing the matching vectors of each of its children. For a leaf node in the tree-decomposition, the matching vector can be computed in a brute-force way. At an internal node s, the matching vector is computed from the matching vectors of its children, which we describe here:

Case 1**:** s **is a** k**-node** Let s has vertices $V_s = \{u_1, \ldots, u_k\}$. Recall that a k-node shares all its vertices with all its neighbors. Let the children of s in T be $s_1, \ldots, s_r$. Let the subgraph corresponding to the subtree rooted at s be H and those at its children be $H_1, \ldots, H_r$. In order to determine $v_H^{(S)}$, we need to know if there is a matching in H that leaves exactly the vertices in S unmatched. This holds if and only if the vertices in S are not matched in any of the H_j's, and each vertex in $V_s \setminus S$ is matched in exactly one of the H_j's. In other words, we need to determine if there is a partition $T_1, T_2, ..., T_r$ of $V_s \setminus S$, such that H_j has a matching in which precisely $V_s \setminus T_j$ is unmatched. That is, $v_{H_j}^{(V_s \setminus T_j)} = 1$ for all $1 \leq j \leq r$. More formally,

$$v_H^{(S)} = \bigvee_{\substack{T_1,\ldots,T_r \subseteq V_s \setminus S: \\ \forall j \neq j' \in [r] T_j \cap T_{j'} = \emptyset: \\ \cup_{j \in [r]} T_j = V_s \setminus S}} \bigwedge_{j \in [r]} v_{H_j}^{(V_s \setminus T_j)} = \bigvee_{\emptyset = U_0 \subseteq \ldots \subseteq U_r = V_s \setminus S} \bigwedge_{j \in [r]} v_{H_j}^{(V_s \setminus (U_j \setminus U_{j-1}))} \tag{5.1}$$

where, the second equality follows by defining $U_0 = \emptyset$ and $U_i = \cup_{j \in [i]} T_j$ for $i \in [r]$. The size of the above DNF formula depends on r which is not a constant hence the straightforward implementation of the above computation would not be in L. However, consider a conjunct in the big disjunction in the second line above. The j^{th} factor of this conjunct depends only on U_j and U_{j-1}, each of which can be represented by a constant number ($= 2^k$) of bits. Thus, we can iteratively extend U_{j-1} in all possible ways to U_j and use the bit indexed by $V_s \setminus (U_j \setminus U_{j-1})$ in the vector for the child. How to obtain the vector of the child within a log-space bound is detailed in the full version.

Case 2**:** s **is a** $k+1$ **node** The procedure is slightly more complex in this case. Let s have vertices $\{u_1, \ldots, u_{k+1}\}$. Let the subgraph corresponding to the subtree rooted at s be H. Let $s_1, \ldots, s_r$ be the children of s, with corresponding subgraphs $H_1, \ldots, H_r$. Note that s may share a different subset of k vertices with each of its children and with its parent. Let the vertices s shares with its parent be $\{u_1, \ldots, u_k\}$. Then its matching vector is indexed by the subsets of $\{u_1, \ldots, u_k\}$, and moreover, u_{k+1} should always be matched in H. To compute $\vec{v}_H$, we first extend the matching vectors of each of its children and make a 2^{k+1} dimensional vector $\vec{w}_H$. The matching vector $\vec{v}_{H_j}$ of a child s_j of s is extended to the new vector $\vec{w}_{H_j}$ as follows: Let s_j contain $\{u_1, \ldots, u_k\}$. We consider an entry $v_{H_j}^{(S)}$ of $\vec{v}_{H_j}$. The vector $\vec{w}_{H_j}$ has two entries corresponding to it.

$$w_{H_j}^{(S \cup \{u_{k+1}\})} = v_{H_j}^{(S)}, \qquad w_{H_j}^{(S)} = \bigvee_{\substack{p \in [k], u_p \notin S, \\ (u_{k+1}, v_p) \in E}} u_{H_j}^{(S \cup \{u_p\})}$$

These new vectors of each of the children can be composed similar to that in the previous case to get $\vec{w}_H$. To get $\vec{v}_H$, we remove the 2^k entries from $\vec{w}_H$ which are indexed on subsets containing u_{k+1}. This vector is passed on to the parent of s. The complexity analysis, and a proof of (2) appears in the full version. ∎

References

[1] Eric Allender, David Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45, 2009.

[2] V. Arvind, B. Das, and J. Köbler. The Space Complexity of k-Tree Isomorphism. In *In Proceedings of ISAAC*, 2007.

[3] Michael Ben-or and Richard Cleve. Computing algebraic formulas using a constant number of registers. *SIAM J. Comput.*, 21(1):54–58, 1992.

[4] Burchard von Braunmühl and Rutger Verbeek. Input driven languages are recognized in log n space. In *Selected papers of the international conference on "foundations of computation theory" on Topics in the theory of computation*, pages 1–19, 1985.

[5] S. Buss, S. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM J. Comput.*, 21(4):755–780, 1992.

[6] N. Chandrasekharan and S. Hannenhalli. Efficient algorithms for computing matching and chromatic polynomials on series-parallel graphs. *Computing and Information Proceedings, (ICCI 92)*, 1992.

[7] A. Chiu, G. Davida, and B. Litow. Division in logspace-uniform NC^1. *Theoretical Informatics and Applications*, 35, 2001.

[8] Samir Datta, Raghav Kulkarni, and Sambuddha Roy. Deterministically isolating a perfect matching in bipartite planar graphs. In *STACS 2008*, volume 1 of *Leibniz International Proceedings in Informatics*, 2008.

[9] Kousha Etessami. Counting quantifiers, successor relations, and logarithmic space. *J. Comput. Syst. Sci.*, 54(3):400–411, 1997.

[10] J. G. Del Greco, C. N. Sekharan, and R. Sridhar. Fast parallel reordering and isomorphism testing of k-trees. *Algorithmica*, 32(1):61–72, 2002.

[11] A. Gupta, N. Nishimura, A. Proskurowski, and P. Ragde. Embeddings of k -connected graphs of pathwidth k. *Discrete Applied Mathematics*, 145(2):242–265, 2005.

[12] F. Harary and E. M. Palmer. On acyclic simplicial complexes. *Mathematica*, 15, 1968.

[13] William Hesse, Eric Allender, and David A. Mix Barrington. Uniform constant-depth threshold circuits for division and iterated multiplication. *JCSS*, 65(4), 2002.

[14] Andreas Jakoby and Till Tantau. Logspace algorithms for computing shortest and longest paths in series-parallel graphs. In *Proceedings of 27th FSTTCS, LNCS 4855*, 2007.

[15] Richard M. Karp, Eli Upfal, and Avi Wigderson. Constructing a perfect matching is in random NC. *Combinatorica*, 6(1):35–48, 1986.

[16] M. M. Klawe, D. G. Corneil, and A. Proskurowski. Isomorphism testing in hookup classes. *SIAM Journal on Algebraic and Discrete Methods*, 3(2):260–274, 1982.

[17] Johannes Köbler and Sebastian Kuhnert. The isomorphism problem for $\boldsymbol{k}$-trees is complete for logspace. *ECCC*, (TR09-053), 2009.

[18] Nutan Limaye, Meena Mahajan, and Prajakta Nimbhorkar. Longest paths in planar dags in unambiguous log-space. In *Computing: Australasian Theory Symposium (CATS)*, 2009.

[19] Nutan Limaye, Meena Mahajan, and B. V. Raghavendra Rao. Arithmetizing classes around $\mathrm{NC}^{\mathbf{1}}$ and L. In *STACS*, 2007.

[20] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[21] Omer Reingold. Undirected $\boldsymbol{st}$-connectivity in logspace. In *Proc. 37th STOC*, 2005.

[22] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. In *IEEE Symposium on Foundations of Computer Science*, pages 244–253, 1997.

[23] Thomas Thierauf and Fabian Wagner. Reachability in $\boldsymbol{K_{3,3}}$-free graphs and $\boldsymbol{K_5}$-free graphs is in unambiguous log-space. In *FCT*, 2009.

[24] Egon Wanke. Bounded tree-width and LOGCFL. *J. Algorithms*, 16(3):470–491, 1994.