Linear-time recognition of bipartite graphs plus two edges

Peter Damaschke

Computing Sciences, Chalmers University, 41296 Göteborg, Sweden

Received 9 April 1999; received in revised form 19 February 2002; accepted 4 March 2002

Abstract

Cai and Schieber (1997) proved that bipartite graphs plus one edge can be recognized in linear time. We extend their result to bipartite graphs plus two edges. Our algorithm works on a depth-first-search spanning tree. This gives, as a byproduct, also a simplified solution to the one-edge case. It is a notoriously open question whether recognizing bipartite graphs plus \( k \) edges is a fixed-parameter tractable problem. The present result might support the affirmative conjecture.

© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Algorithms and data structures; Parametric complexity; Recognizing graph properties; Tree computations

1. Introduction

The concept of parametric complexity provides another perspective to NP-hard optimization problems besides the theory of approximation algorithms. While approximation algorithms have to compute suboptimal solutions within polynomial time \( p(n) \) (with input size \( n \)), a problem is called fixed-parameter tractable (FPT) if one can decide in time \( O(f(k)p(n)) \) the existence of a solution with value \( k \) or less. Here \( f \) is a fixed but arbitrary function.

For some problems it is notoriously open whether they are FPT, among them the recognition of bipartite graphs with at most \( k \) additional edges. As long as this question remains unsolved, the best we can do is to get some insight for small \( k \). (Remark: The optimization version is known as MAX-CUT: Find a bipartition of the vertex set with a maximum number of edges connecting both parts.)

E-mail address: ptr@cs.chalmers.se (P. Damaschke).
Consider graphs with \( n \) vertices and \( m \) edges. The \( O(n + m) \) time recognition of bipartite graphs is folklore [1]. A naive algorithm to recognize bipartite graphs plus \( k \) edges would require \( O((n + m)m^k) \) time. However, Cai and Schieber [3] devised an \( O(n + m) \) time algorithm for case \( k = 1 \), which reduces the general time bound to \( O((n+m)m^{k-1}) \). Here we make a further step of progress and show that case \( k = 2 \) still can be solved in \( O(n + m) \) time, thus reducing the exponent to \( k - 2 \). Unlike [3], we simplify things by utilizing a depth-first-search tree rather than an arbitrary spanning tree. Our result, although far from being general, may give some evidence in support of the conjecture that the problem is FPT, because non-FPT problems typically have complexity \( O(n^k) \). It would be nice to extend the result to larger \( k \), hopefully without further blow-up of case distinctions. (Already a simplified proof of our present result would be valuable.)

Our problem is equivalent to the problem of computing a hitting set of \( k \) edges for the family of all odd cycles. For \( k = 1 \), this family of hitting sets is just the intersection of odd cycles. However \( k = 2 \) requires already a much more complicated characterization. Other cycle intersection problems in graphs have been studied in [5].

1.1. More motivation and related work

As mentioned in [3], MAX-CUT is a case of constrained MAX-SAT where one seeks almost satisfying assignments in special Boolean formulae [9]. In applications like diagnosis or pattern recognition, one wants to find an unknown object (represented by Boolean values of attributes) which agrees to certain rules (clauses) that reflect the knowledge about this object. But since real situations are often vague, one may tolerate violation of a few rules, such that fast algorithms for recognizing properties subject to \( k \) violations are of interest, even for small fixed \( k \).

Several recognition problems for graphs properties after adding/removing \( k \) edges are FPT. If the property is closed under the minor relation then, due to the Robertson–Seymour Theory, it can be tested in linear time for any fixed \( k \). However the resulting algorithms may be tremendously complicated and impractical, such that elementary algorithms are still of interest. An example is the undirected \( k \)-feedback edge set problem (forests plus \( k \) edges) [4]. However note that bipartiteness is not closed under minors, since edge contraction may produce odd cycles.

Another general theorem from [2] says that a \( k \)-edge modification problem is FPT if the graph property is characterized by finitely many forbidden induced subgraphs. This approach also fails for our problem, since bipartite graphs are characterized by exclusion of odd cycles of any length. FPT results for interval graphs and chordal graphs minus \( k \) edges can be found in [7]. Applications include DNA mapping and solving sparse systems of linear equations (well-known as the minimum fill-in problem).

A related but different aim of research is to get polynomial algorithms for finding modifications of problem instances so as to reach a specified property, where the amount of modification must be bounded by some function of the optimum \( k \) and the input size, see [8,11].
2. Prerequisites

We presume that the reader is familiar with the usual graph-theoretic notion and with depth-first-search (DFS), otherwise we refer to [1] or other algorithm textbooks. Here we only recall some basic facts.

DFS splits the edge set of a connected graph $G$ into tree edges (forming a DFS tree) and back edges. The produced DFS tree depends on the start vertex (root) and the (previously fixed) ordering of edges incident to any vertex in $G$. The following obvious property of DFS will be useful: If $e$ is a back edge in a DFS tree of $G$ then applying DFS to $G\setminus e$, starting from the same root, produces the same DFS tree, provided that the same edge orderings are used.

A rooted ordered tree is a tree with a distinguished vertex, the root, and a fixed ordering of the children of each inner vertex. We suppress the attributes and simply speak of a tree. A rooted path is any subpath of a path from the root to some leaf. Whenever we speak of a path $P$, we consider it as an edge set, whereas the set of vertices participating in $P$ is denoted $V(P)$. The same convention applies to subtrees. Consequently, $G\setminus P$ denotes $G$ without the edges of $P$; the vertices are not removed. A branching vertex of a tree is a vertex of degree $\geq 2$.

If there is a rooted path $P$ from vertex $u$ to vertex $v$ then we say that $u$ is an ancestor of $v$, or $u$ is above $v$, or $v$ is below $u$. Furthermore, $u$ and $v$ is called the top and the bottom, respectively, of $P$. Sometimes we denote the path $P$ simply by $uv$. For notational convenience we also allow that $P=\emptyset$ but $|V(P)|=1$, that is, $u=v$. Vertices $u,v$ are called comparable if $u$ is above $v$ or vice versa. Otherwise $u,v$ are incomparable, and $u$ is either to the left of $v$ or to the right of $v$ in the tree. (This notion should be self-explanatory.) The uniquely determined lowest common ancestor (LCA) of two vertices $u,v$ is an ancestor $w$ of $u,v$ such that no other vertex below $w$ is an ancestor of both $u$ and $v$.

Rooted paths have the well-known Helly property: If all members of a family of rooted paths have pairwise non-empty intersections then the intersection of all members of the family is non-empty. (In contrast, this is not true for arbitrary paths in a tree.) Tree paths $P$ considered as vertex sets $V(P)$ have the Helly property, too.

3. Recognizing bipartite graphs plus two edges

In this section we prove the main result:

Theorem 1. Bipartite graphs with two additional edges can be recognized in $O(n+m)$ time.

We may restrict attention to connected graphs, as the case of disconnected graphs can be easily reduced to this case.

Our algorithm starts with performing DFS on the given graph $G$. We paint the vertices of the DFS tree properly with 2 colors and temporarily ignore the back edges. This can be done in $O(n+m)$ time. An edge is called good if its endpoints have received different colors, otherwise the edge is bad. Note that all tree edges are good. A
bad path is a path of tree edges whose endpoints are joined by a bad edge. Similarly, a good path is a path of tree edges whose endpoints are joined by a good back edge. Clearly, bad and good paths are rooted paths in the DFS tree. Let $B$ be the intersection of bad paths and $J$ the union of good paths. For any bad edge $e$ we denote by $B_e$ the intersection of all bad paths, except the bad path defined by $e$. Similarly, for a good back edge $e$ we denote by $J_e$ the union of all good paths, except the good path defined by $e$.

Note that a cycle in $G$ is odd if and only if it contains an odd number of bad edges. We build upon the following result which may be derived from Proposition 3.2(4) and Theorem 3.3(1) of [3]. However we give our own, simplified proof.

**Lemma 2.** Suppose that we have at least two bad edges. Then the intersection of all odd cycles in $G$ equals $B \setminus J$. Consequently, a non-bipartite graph $G$ is a bipartite graph plus one edge if and only if $B \setminus J \neq \emptyset$.

**Proof.** Let $e$ belong to every odd cycle. Then $e$ is a tree edge in $B$. If $e$ also belongs to some good path then we can obviously form a cycle in $G \setminus e$ including exactly one bad edge. Hence $e \not\in J$. Conversely, take any $e \in B \setminus J$. Deletion of $e$ splits the DFS tree of $G$ in two components joined by every bad edge but by none of the good edges. Hence every cycle in $G \setminus e$ has an even number of bad edges. Thus all odd cycles in $G$ include $e$. □

We call a pair of edges $(e, f)$ from $G$ a suitable pair if removing $e, f$ yields a bipartite graph. Now we start an extensive case distinction. To provide some top-down view of the algorithm, we suppose that several basic computations in trees can be done efficiently and refer to the technical lemmas which are deferred to Section 4. For first reading, just look ahead to these lemmas and inspect their proofs later.

If DFS produced at most two bad edges then this is a suitable pair, and we are done.

So assume in the following the existence of three or more bad edges.

In this case it is useless to delete exactly two bad edges $e, f$, since at least one odd cycle formed by some bad edge and its corresponding bad path would remain.

However, one of the edges to be removed, say $e$, may be bad. In this case we seek a bad edge $e$ whose deletion leaves a bipartite graph plus one edge. Note that $G \setminus e$ is connected, and that applying DFS to $G \setminus e$ would produce the same DFS tree. (Remember the remark in Section 2.) Thus, by Lemma 2, $e$ is a bad edge such that $B_e \setminus J \neq \emptyset$. To find such an edge $e$ it suffices to have a tree edge $t \notin J$ that belongs to all bad paths but one. Note that this means $t \in B_e$ for some $e$.

If $B = \emptyset$ then, by the Helly property, there exist two bad edges whose bad paths are disjoint, and $e$ must be one of them. The two disjoint bad paths can be found in $O(n + m)$ time using Lemma 7. Hence we have reduced this case to two applications of Lemma 2: We have to test two graphs whether they are bipartite plus one edge. If $B \neq \emptyset$, we compute $D = \bigcup_e B_e$ and finally $D \setminus J$, with help of Corollary 6. So we have settled the case of bad edges that have to be deleted.

In the following, both edges $e$ and $f$ to be deleted are good edges.
Let at least one of them, say $e$, be a good back edge. Then we seek a back edge $e$ whose deletion leaves a bipartite graph plus one edge. We conclude similarly as above: Applying DFS to $G \setminus e$ produces the same DFS tree. Thus, by Lemma 2, $e$ is a good edge such that some $t \in B \setminus J_e$ exists. Since $t \in B$, $t$ must be a tree edge, and $t \notin J_e$ means that $t$ is not in any good path, possibly except the good path of $e$. Thus, once we have a tree edge $t \in B$ appearing in at most one good path then we may choose $e$ to be the corresponding good back edge. We find such a edge $t$ in $O(n+m)$ time, see Corollary 6 again. Thus we have completely resolved the case that $e$ (or $f$) is a back edge.

Therefore, in the remainder of the section, both $e$ and $f$ are supposed to be tree edges.

Deletion of any two tree edges $e, f$ splits the DFS tree into three components $U, V, W$. The term triangle refers to a triple of back edges joining $U$ and $V$, $V$ and $W$, $U$ and $W$, respectively.

**Lemma 3.** $(e, f)$ is suitable if and only if:

1. neither component contains a bad edge,
2. any two components are not joined by both a bad and a good edge,
3. there is no triangle of bad edges, and
4. there is no triangle of one bad edge and two good edges.

To see this, just remember that odd cycles are exactly those containing an odd number of bad edges, and verify that the negations of cases (1)–(4) cover all possibilities of odd cycles. We omit the details. The figures depict the forbidden configurations. Single and double lines are bad and good edges, respectively.

![Forbidden Configurations]

Case (1) is equivalent to the condition that $e, f$ hit all bad paths. First we characterize these hitting pairs $e, f$, afterwards we shall invoke the other conditions, such that only the suitable pairs remain. We emphasize that we need a succinct representation of the pairs $e, f$ that satisfy (1), since there may exist $O(m^2)$ of them.

First of all, there is an obvious special case: If $B \neq \emptyset$ then each pair of some $e \in B$ and an arbitrary tree edge $f$ fulfills (1).

It remains to collect all pairs $e, f \notin B$ satisfying (1). Clearly, both $e$ and $f$ must belong to bad paths. We distinguish two principal cases: $e$ is above $f$ (or vice versa) in the DFS tree, or $e, f$ are incomparable (with respect to the ancestor relation of tree edges).

1. $e, f$ are incomparable.
   1.1. If there exist two disjoint bad paths $P, Q$ then, clearly, $e \in P$ and $f \in Q$ (or vice versa). If the highest edges of $P$ and $Q$ are incomparable then no other bad path can
meet both $P$ and $Q$, hence $e$ and $f$ must hit every bad path which meets $P$ and $Q$, respectively. So this case is easily reduced to two computations of path intersections. W.l.o.g. let the highest edge of $P$ be above the highest edge of $Q$, as shown in the figure. Then, since $e, f$ are incomparable, $e$ is restricted to the lower subpath $P'$ of $P$ which is not above $Q$. No other bad path can meet both $Q$ and $P'$, hence we are in the previous case, with $P'$ in the role of $P$. The top of $P'$ is found by a single LCA computation.

1.2. If all bad paths pairwise intersect then the Helly property implies $B \neq \emptyset$. Obviously $e$ and $f$ are below the bottom $v$ of $B$. No bad path ends already at $v$, otherwise (1) would be violated. Let $T$ be the tree formed by the union of bad paths. If $T$ is merely a path then there exist no incomparable $e, f \in B$. If $v$ has more than two children in $T$ then no incomparable pair $e, f$ can hit all bad paths. If $v$ has exactly two children then we can split the family of bad paths in two subfamilies, according to the included edge immediately below $v$. Then $e$ can be any edge on the rooted path from $v$ to the bottom of the intersection of one subfamily, and a similar statement applies to $f$. Moreover, $e$ and $f$ can be chosen independently, and any such pair $e, f$ hits all bad paths. We can decide in $O(n + m)$ time whether our input belongs to this case, and then compute the two mentioned path intersections using Corollary 6.
We resume that in Case 1 all pairs $e, f$ satisfying (1) are represented by two disjoint paths with incomparable highest edges, such that $e$ is on one path, $f$ is on the other path, and $e, f$ can be chosen independently.

2. $e$ is above $f$.

Here we have to introduce some further notion. Let $X, Y$ be rooted paths such that $X$ is entirely above $Y$. (That means, the bottom of $X$ is above the top of $Y$.) Moreover let $Z \supseteq X \cup Y$ be some fixed rooted path. We may represent the set $X \times Y$ of ordered pairs of edges from $X, Y$ by an axis-parallel rectangle in the plane as follows: Both the $x$- and $y$-axis represent $Z$, with a unit length segment devoted to every edge. By this, vertices on $Z$ are represented by points on the axes. Our rectangle is, by definition, vertically bounded by the bottom and top of $X$ on the $x$-axis, and horizontally bounded by the bottom and top of $Y$ on the $y$-axis. A pair $(e, f) \in X \times Y$ is a unit square in this rectangle. Next, let $R$ be some rooted path intersecting $Z$. We represent $R$ by the point whose $x$- and $y$-coordinate is the top and the bottom, respectively, of $R \cap Z$.

Each point $o$ in the plane defines, in a natural way, four quadrants with origin $o$ which are suggestively called the NW-, NE-, SW-, and SE-quadrant. A cone is a union of two opposite quadrants with the same origin, either NW and SE, or SW and NE.

2.1. If there exist two disjoint bad paths $P, Q$ then w.l.o.g. we have $e \in P$ and $f \in Q$. Let $Q^+$ be the extension of $Q$ up to the root, and $P' = P \cup Q^+$. Since $e$ is above $f$, it follows $e \in P'$. Again, $e$ must hit every bad path that meets $P'$ but not $Q$, and similarly, $f$ must hit every bad path that meets $Q$ but not $P'$. Let $X \subseteq P'$ and $Y \subseteq Q$ be the set of all $e$ and $f$, respectively, that satisfy these conditions. However not all pairs in $X \times Y$ are suitable in general: There may exist further bad paths $R$ which meet both $P'$ and $Q$, and each of them must contain at least one of $e \in X$ and $f \in Y$. We represent $X \times Y$ as a rectangle and every bad path $R$ as a point, as explained above. We have $R \cup \{e, f\}$ iff $e$ is above the top and $f$ is below the bottom of $R \cap Z$. 
that is, iff the square representing \((e, f)\) is to the NW of the point representing \(R\). Thus the region of pairs satisfying (1) is exactly \(X \times Y\) minus all these NW-quadrants. An \(O(m)\) space representation of this region can be computed in \(O(n + m)\) time by Lemma 9.

2.2. If all bad paths pairwise intersect then \(B \neq \emptyset\) (Helly property). Let \(A\) denote the path from the root to the top of \(B\). Recall that \(e, f \in B\), but every bad path contains \(e\) or \(f\), and \(e\) is above \(f\). If \(e\) were below \(B\) then, by this constellation, \(e\) would obviously belong to every bad path, which contradicts the definition of \(B\). Thus \(e \in A\).

Moreover it follows that \(f\) must be in the intersection \(F\) of those bad paths which do not exceed the top of \(B\). One can compute \(F\) in \(O(n + m)\) time, due to Corollary 6.

Since \(F\) is entirely below \(A\), we run into the same situation as in 2.1, where \(A\) and \(F\) take on the roles of \(P'\) and \(Q\), respectively. Continue as in 2.1.

We summarize the preceding discussion in

**Lemma 4.** The family \(I\) of pairs of tree edges \((e, f)\) that satisfy (1), i.e. that hit all bad paths, can be partitioned into the following subfamilies:

- **I\(_1\):** one edge belongs to \(B\), the other one is arbitrary,
- **I\(_2\):** \(= X' \times Y'\) where \(X', Y'\) are disjoint rooted paths with incomparable highest edges, and every edge in \(X' \cup Y'\) belongs to some bad path but not to \(B\),
- **I\(_3\):** \(=(X \times Y) \setminus Qu\) where \(X, Y\) are disjoint rooted paths, \(X\) is entirely above \(Y\), and \(Qu\) is the union of at most \(m\) NW-quadrants in the rectangle representing \(X \times Y\).

We remark again that \(e, f \notin B\) here.

All these representations are computable in \(O(n + m)\) time and can be stored in \(O(1)\) space or, in case of \(I\(_3\), in \(O(m)\) space.

In the following we assume \(B \subseteq J\) (otherwise, Lemma 2 ensures that \(G\) is bipartite plus one edge, and we are done). Remember that we have to characterize those members of \(I\) satisfying also (2)–(4). Consider any pair \((e, f)\)\(\in I\). Once more, we have to distinguish some cases, according to the relative positions of \(e, f\).

1. Let \(e, f\) be incomparable. No triangle (as defined prior to Lemma 3) can appear in this case, hence (3) and (4) are true. It remains to characterize the \((e, f)\) satisfying (2). Since \(e, f\) are incomparable, (2) is equivalent to the following condition: Neither \(e\) nor \(f\) belongs to both a bad and a good path. Since \(B \subseteq J\), no pair from \(I\(_1\) is suitable. For \((e, f)\in I\(_2\) we get \(e \in X' \setminus J\) and \(f \in Y' \setminus J\). (Recall that \(e\) and \(f\), respectively, belongs to some bad path, hence we must insist that neither \(e\) nor \(f\) is in \(J\).) These set differences can be computed in \(O(n + m)\) time using Corollary 6.

2. Let \(e\) be above \(f\). Let \(U, V, W\) denote the component of \(G \setminus \{e, f\}\) containing the vertices above \(e\), between \(e\) and \(f\), and below \(f\), respectively. (In general, these components contain, of course, further vertices.) We continue with the easier subcase.

2.1. If \((e, f)\in I\(_3\) then \(e, f \notin B\). Hence there exists a bad edge joining \(U\) and \(V\), and a bad edge joining \(V\) and \(W\). Therefore, any bad path including both \(e\) and \(f\) would violate (3). Hence we must exclude from \(X \times Y\) the SE-quadrant of every bad path.
which intersects both \( X \) and \( Y \). Moreover, due to (2), any good path must contain either none or both of \( e, f \). Therefore we also have to exclude the following sets: the SW-NE-cone of every good path \( R \) which intersects both \( X \) and \( Y \), and the members of \( X \) (\( Y \)) being in some good path which intersects \( X \) (\( Y \)) only. Conversely, note that the remaining pairs satisfy (2)–(4). The set differences can be computed using Lemma 9.

2.2. Consider \((e, f) \in I_1\). If \( e, f \in B \) then \( B \subseteq J \) implies existence of either a good edge joining \( U \) and \( W \), or two good edges joining \( U \) and \( V \), \( V \) and \( W \). This would obviously violate (2) or (4). Thus exactly one of \( e, f \) is in \( B \).

2.2.1. Suppose \( f \in B \). Since \( e = \notin B \), not all the bad paths can have their top vertices in \( U \), hence there must be some bad edge joining \( V \) and \( W \). By \( B \subseteq J \), some good edge leaves \( W \), but due to (2), it cannot end in \( V \). Hence some good edge joins \( U \) and \( W \). Applying (2) again we find that all bad edges join \( V \) and \( W \). This means, \( e \) is not contained in any bad path. Furthermore it follows from (4) that also none of the good edges joins \( U \) and \( V \). Conversely, if all bad edges join \( V \) and \( W \) and all good edges outside the components only join \( U \) and \( W \) then (2)–(4) are obviously true. Reformulation yields the following characterization of suitable pairs of the type considered here: \( f \in B \), every good path contains none or both of \( e, f \), and no bad path contains \( e \). To compute the set of these suitable pairs, let \( X \) be the rooted path above \( B \) and \( Y = B \), and proceed with \( X \times Y \) similarly as in case 2.1. Here we omit the straightforward details.

2.2.2. Let \( e \in B \). Due to the symmetry in Lemma 3 we just turn the condition from 2.1.1 upside down and get the characterization of suitable pairs: \( e \in B \), every good path contains none or both of \( e, f \), and no bad path contains \( f \). However this case is not symmetric to 2.2.1 with respect to the DFS tree, thus we have to conclude in a different way: Applying Lemma 10 to \( B \) and the family of good paths, we can determine the suitable pairs of this type in \( O(n+m) \) time. Note that excluding the edges \( f \) contained in bad paths is a simple step, as we can determine the union of bad paths in \( O(n+m) \) time by Corollary 6.

\( G \) is bipartite plus two edges if and only if we find a suitable pair in one of all these cases, i.e. if one of the sets computed above is non-empty. Thus our case distinction establishes an \( O(n+m) \) time recognition algorithm.

Admittedly this case distinction is tiresome. A more “economical” proof would be highly welcome.

4. Implementing the basic operations

In the following we list the computational primitives which are used in our algorithm and prove their time bounds. Some of them may also be of independent interest.

We emphasize that all our routines manage without the tool of multiple LCA computations in its full generality: It is known that \( p \) LCA computations (given \( p \) pairs of vertices, find their LCAs) can be done in \( O(n+p) \) time [6,10]. Usage of this result would slightly simplify some of our proofs, but on the negative side it would
complicate the algorithm, since we would have to invoke an algorithm for multiple LCA computations, which is rather complicated. Therefore it is more appropriate to use elementary tools only: We will need ancestor tests (given \( u \) and \( v \), decide whether \( u \) is an ancestor of \( v \)), and \( p \) LCA computations done in \( O(n+p) \) time if one operand is fixed. We shall see that these routines are quite easy to implement.

When considering a tree, we will always implicitly assume that the preorder and postorder numbers are at hand, that is, DFS has already been executed in an \( O(n+m) \) time preprocessing phase. DFS assigns a preorder number and a postorder number to every vertex of \( G \). Using these precomputed numbers we can easily decide the relative position (above, below, left, right) of any two given \( u \) and \( v \) in \( O(1) \) time. This is called a comparison of vertices.

In the following, rooted paths are given by their endpoints \( u, v \) and denoted \( uv \). An ordered list of vertices of path \( uv \) can be obtained in \( O(n) \) time: Follow the tree edges upwards from \( v \) to \( u \). Using the reverse list, we can traverse a specified ordered path also in downwards direction, after \( O(n) \) time preprocessing.

**Lemma 5.** Given a family \( F \) of \( p \) rooted paths (not necessarily distinct) in a tree of \( n \) vertices, we can compute, in \( O(n+p) \) time, for every tree edge \( t \) the number of paths in \( F \) which include \( t \).

**Proof.** Let \( r \) be the root. Replace each path \( uv \) in \( F \) by \( rv \). In a second run, replace each path \( uv \) in \( F \) by \( ru \). If we can compute the desired numbers with respect to these two path families, then subtracting the results for every tree edge \( t \) gives the final result. Thus we may restrict attention to the case that all given paths start at \( r \). But then the number of paths containing an edge \( t \) is simply the number of bottom vertices \( v \) below \( t \). Hence we get all these numbers by straightforward additions along the tree. We need \( O(p) \) time to count, for each vertex \( v \), the number of paths having bottom \( v \), followed by \( O(n) \) additions.

We remark that the time bound refers to the uniform cost measure, however, as we apply the lemma with \( p < n^2 \), all appearing numbers have \( O(\log n) \) bits only.

**Corollary 6.** Given two families of at most \( p \) rooted paths, we can compute the following objects in \( O(n+p) \) time: the intersection \( B \) of all paths of one family, the union \( J \) of all paths of the other family, \( B \setminus J \), the tree edges hitting exactly \( k \) paths of one family, and the tree edges hitting all but \( k \) paths of one family.

**Lemma 7.** If a family \( F \) of \( p \) rooted paths has an empty intersection then we find two disjoint paths in \( F \) in \( O(n+p) \) time.

**Proof.** Consider the members of \( F \) in arbitrary order, and let \( B_k \) be the intersection of the first \( k \) paths. \( B_k \) is itself a path. For any \( k \), assume that \( B_k \) is already computed. (This assumption is trivial for \( k = 1 \).) Starting from the bottom of \( B_k \), go upwards in the tree until some ancestor of the bottom of the \( (k+1) \)th path \( P_{k+1} \) is encountered.
(This can be tested in $O(1)$ time for each visited vertex, by comparison.) In any such event, update the bottom and top of $B_{k+1}$ in $O(1)$ time: The new bottom is the vertex where $P_{k+1}$ entered $B_k$, the new top is the top of either $B_k$ or $P_{k+1}$. Then continue with $k := k + 1$. If the top of $B_k$ is reached before we met an ancestor then $B_{k+1} = \emptyset$, and we can stop. By our assumption and the Helly property, $P_{k+1}$ is disjoint to some of the previous paths, and we find one by exhaustive search and vertex comparisons; details are obvious. □

The next lemma implies in particular that $q$ LCA computations with one fixed operand can be done in $O(n + q)$ time, without using an algorithm for the general off-line LCA problem [6,10].

**Lemma 8.** Let $B$ be a fixed rooted path of length $n(B)$. For any path $Q$ from a family of $q$ rooted paths, let $l(Q)$ be the bottom of $Q$, and x($Q$) the bottom of $B \cap Q$ (if the intersection is non-empty). If the $l(Q)$ are available in two lists, sorted by their preorder and postorder numbers, respectively (which can be done in $O(n + q)$ time) then we can compute all x($Q$) in another $O(n(B) + q)$ steps.

**Proof.** Note that x($Q$) is the LCA of $l(B)$ and $l(Q)$, whenever $B$ and $Q$ intersect.

First we throw away the paths $Q$ with $B \cap Q = \emptyset$. Every such $Q$ is recognized in $O(1)$ time by comparisons of top and bottom vertices of $B$ and $Q$.

We partition the set of remaining paths $Q$ into two subsets: one with $l(Q)$ to the left of $l(B)$, and a second one including all other $Q$. For any $Q, Q'$ from the former set we see the following implication:

\[ \text{preorder}(l(Q)) \leq \text{preorder}(l(Q')) \Rightarrow \text{preorder}(x(Q)) \leq \text{preorder}(x(Q')), \]  

in other words, $x(Q)$ is an ancestor of $x(Q')$.

(This is obvious because $l(Q)$ lies to the left of or above $l(Q')$.)

For $Q, Q'$ from the latter set we similarly have:

\[ \text{postorder}(l(Q)) \leq \text{postorder}(l(Q')) \Rightarrow \text{postorder}(x(Q)) \leq \text{postorder}(x(Q')), \]  

in other words, $x(Q')$ is an ancestor of $x(Q)$.

(Note that $l(Q)$ lies to the left of or below $l(Q')$.)

By these monotonicity properties, we can sort the paths in both sets by increasing $x(Q)$, without explicitly computing the $x(Q)$ before. Just use the lists of preorder or postorder numbers.

So consider a list of paths $Q$ with increasing (but yet unknown) $x(Q)$. Traverse $B$ in upwards direction and test whether the actual vertex $y$ is an ancestor of $l(Q)$ of the first $Q$ in our list. In that case we know that $x(Q) = y$, and we can remove $Q$ from the list, and so on. In this way we determine all $x(Q)$ in $O(n(B) + q)$ time. □

**Lemma 9.** Let us be given a rectangle $X \times Y$. Let $Qu$ be the union of $p$ quadrants and $Co$ the union of $p$ cones. Then we can compute, in $O(n + p)$ time, an $O(p)$ space representation of sets $(X \times Y) \backslash Qu$, $(X \times Y) \backslash Co$, and $(X \times Y) \backslash (Qu \cup Co)$, respectively.
We omit the straightforward proofs. We just remark that any such set is bounded by at most four monotone staircase curves.

**Lemma 10.** Let us be given a rooted path $B$ and a family $F$ of $p$ rooted paths intersecting $B$. Suppose that we are interested in the family of pairs of tree edges $e, f$ such that $e \in B$, $f \notin B$, $e$ is above $f$, and every path in $F$ contains either none or both of $e, f$. Then we can compute, in $O(n+p)$ time, an $O(p)$ space representation of this family.

**Proof.** For a rooted path $P$, let $t(P)$ and $x(P)$ denote the top and the bottom, respectively, of $B \cap P$. For notational convenience, we identify vertices with their postorder numbers in the following.

Consider any $x \in V(B)$. Let $F_x$ denote the set of all $P \in F$ such that $x(P) = x$, these are the $P$ that “enter” $B$ in vertex $x$. We call a path $P \in F_x$ with maximum $t(P)$ among all paths in $F_x$ the candidate path. (Ties are broken arbitrarily.) Now consider any $Q \in F_x$. Clearly $t(Q) \leq t(P)$, in other words, $t(P)$ is above $t(Q)$. No edge in $Q \setminus P$ can be a candidate for $e$: Note that $e \in Q \cap B$ implies $e \in P$ and thus $f \in P$, contradiction.

Hence we have for every vertex $x \in V(B)$: The candidate path in $F_x$ contains all candidates for edges $f$ that belong to any path in $F_x$. (This explains the naming.) The family of candidate paths $P \in F$, one from every nonempty $F_x$, is computable in $O(n+p)$ time: Lemma 8 shows how to compute all the $x(P)$ in $O(n+p)$ time, and taking the maximum is a trivial operation.

Moreover, an edge of $B$ belonging to any $P, Q \in F$ such that $x(P) \neq x(Q)$ is obviously not a candidate for $e$. In other words, for every candidate edge $e$ there exists a unique vertex $x(e) \in V(B)$ below $e$ such that all $P \in F$, $P \ni e$ satisfy $x(P) = x(e)$. Since we know the $x(P)$ and $t(P)$ (from $O(n+p)$ time computation), we can also compute all $x(e)$ and mark the non-candidate edges in $O(n+p)$ time: Use bucketsort and a downwards scan of $B$; straightforward details are omitted. Note that several $e$ may have the same $x(e)$.

Consider a fixed candidate path $P \in F_x$ as selected above, and the set of candidate edges $e$ such that $x(e) = x$. For any path $Q \in F_x$ let $y(Q)$ be the bottom of $P \cap Q$. By
Lemma 8 we can compute all the $y(Q)$ in $O(n(P\setminus B) + |F_x|)$ time. Since the candidate paths are disjoint outside $B$, this is a total of $O(n+p)$ time for all candidate paths. (The sorted lists containing the preorder and postorder numbers of the $l(Q)$ as required by Lemma 8 can be constructed beforehand in totally $O(n+p)$ time.)

Now define $X(P)$ to be the set of all candidate edges $e$ with $x(e)=x$, and let $Y(P)=P\setminus B$. Then the pairs $(e, f)$ satisfying the desired condition are the members of $(X(P) \times Y(P)) \setminus \Co(P)$, where $\Co(P)$ is the union of all SW–NE-cones defined by paths $Q \in F_x$: To see this, recall that $f$ must belong to the candidate path in $F_{x(e)}$, hence $(e, f) \in X(P) \times Y(P)$, and that we have to exclude those $(e, f)$ such that some $Q$ contains exactly one of $e$ and $f$. But these pairs form a cone. Now apply Lemma 9 to obtain an $O(|F_x|)$ representation of $(X(P) \times Y(P)) \setminus \Co(P)$.

Finally, we have to consider all candidate paths $P$ and to compute the union of all $(X(P) \times Y(P)) \setminus \Co(P)$. Since all the $X(P)$ and $Y(P)$ are mutually disjoint and each $Q \in F$ is assigned to at most one candidate path (with $x(P)=x(Q)$), we get the asserted complexity bounds. □

References