# The Complexity of Searching a Graph

N. MEGIDDO

*Tel-Aviv University, Tel-Aviv, Israel*

S. L. HAKIMI

*Northwestern University, Evanston, Illinois*

M. R. GAREY AND D. S. JOHNSON

*AT&T Bell Laboratories, Murray Hill, New Jersey*

AND

C. H. PAPADIMITRIOU

*Massachusetts Institute of Technology, Cambridge, Massachusetts, and National Technical University of Athens, Athens, Greece*

Abstract. T. Parsons originally proposed and studied the following pursuit-evasion problem on graphs: Members of a team of searchers traverse the edges of a graph $G$ in pursuit of a fugitive, who moves along the edges of the graph with complete knowledge of the locations of the pursuers. What is the smallest number $s(G)$ of searchers that will suffice for guaranteeing capture of the fugitive? It is shown that determining whether $s(G) \leq K$, for a given integer $K$, is NP-complete for general graphs but can be solved in linear time for trees. We also provide a structural characterization of those graphs $G$ with $s(G) \leq K$ for $K = 1, 2, 3$.

Categories and Subject Descriptors: F.2.2 **[Analysis of Algorithms and Problem Complexity]**: Nonnumerical Algorithms and Problems—*computations on discrete structures*; *sorting and searching*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: NP-completeness, pursuit and evasion

## 1. *Introduction*

Let $G = (V, E)$ be a connected undirected graph. Imagine that this graph represents a system of *tunnels* in which a *fugitive* is hidden. Members of a team of $s$ *searchers* traverse the edges of the graph seeking to capture the fugitive, while the latter

This paper is based on "The Complexity of Searching a Graph" by N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, and C. H. Papadimitriou, appearing in *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. IEEE, New York, 1981, pp. 376–385. © 1981 IEEE.

Authors' present addresses: N. Megiddo, Dept, K53, The IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099; S. L. Hakimi, Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA 95616; M. R. Garey, Room 2D-152, AT&T Bell Laboratories, Murray Hill, NJ 07974; D. S. Johnson, Room 2D-150, AT&T Bell Laboratories, Murray Hill, NJ 07974; and C. H. Papadimitriou, Department of Computer Science, University of California, San Diego, La Jolla, CA 92093.

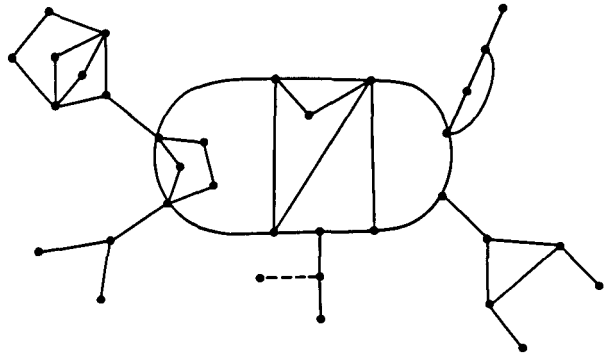

FIG. 1. Graph $G$ with $s(G) = 3$, increasing to 4 when dotted edge is added.

moves around the edges of the graph with unbounded speed, cunning, and luck, trying to avoid them. Alternatively, we can think of a *diffused* fugitive (say, dust or toxic gas), in which case the team is, in effect, *clearing* the graph. What is the smallest number $s$ for which there exists a successful searching strategy, that is, one that is guaranteed to capture the fugitive?

This problem, well known in the combinatorics community, was first suggested by Parsons [11, 12], who introduced it as a nondiscrete problem, with both searchers and fugitive allowed to move continuously. However, as Parsons observed, it is not difficult to show that the problem is equivalent to a discrete one. In fact, we can restate the problem in a manner that is similar in style, if not detail, to many of the pebbling problems that have been studied of late (e.g., see [7, 13]). The basic operations are (1) placing a searcher (pebble) on a vertex, (2) removing a searcher from a vertex, and (3) moving a searcher from one vertex to another along an edge. There is no limit on the number of searchers a vertex can hold, although a capacity of two will always suffice.

Initially, all the edges of the graph are *contaminated*, that is, capable of harboring a fugitive. An edge is *cleared* by placing a searcher at one end (as a *guard*) and moving a second searcher along the edge itself from the guarded endpoint to the other endpoint. If the guarded endpoint is such that all other edges incident on it are already clear, then we can dispense with the second searcher and clear the edge simply by moving the endpoint's guard along the edge to the other endpoint. A clear edge remains clear so long as every path from it to a contaminated edge is blocked by at least one guard. A clear edge is *recontaminated* if ever an unguarded path to a contaminated edge comes into existence owing to the moving of a searcher. The entire graph has been cleared, that is, successfully searched, once all of its edges are simultaneously clear.

A *search strategy* is a sequence of pebbling operations that will clear an initially contaminated graph. The *search number* $s(G)$ for a graph $G$ is the minimum number of searchers for which a search strategy exists. The calculation of $s(G)$, given $G$, is a very tricky algorithmic problem. For example, it is probably not immediately obvious that for the graph $G$ in Figure 1, $s(G) = 3$, and that, if the dotted edge is added, the search number increases to 4.

In this paper we present both complexity results and efficient special-case algorithms for this problem, along with some characterization results for graphs with small search number. In Section 2 we show that determining $s(G)$ for an arbitrary graph $G$ is indeed difficult by proving that the problem: Given $G$

and an integer $K$, is $s(G) \leq K$? is NP-complete. In Section 3 we study the search number problem for the special case of trees and show that for such graphs the search number can be determined in linear time and a search plan using the minimum number of searchers can be found in time $O(n \log n)$. In Section 4 we present results characterizing the structure of graphs with search number $K$ for $K = 1, 2, 3$. Finally, in Section 5, we mention some open questions and some recent results for several related problems, including some interesting ties between the search number problem and the problem of laying out a graph on a line to minimize its "cutwidth," which arises in connection with VLSI circuit layout.

## 2. *Complexity Questions*

It is not difficult to see that the general question: Given $G$ and $K$, can $G$ be cleared with $K$ or fewer searchers? is in PSPACE. With other pebbling problems, such as the ones discussed in [7] and [13], this is the best one can hope for, because of the possibility of "recomputation," that is, using more time, but fewer pebbles, by repeating parts of the pebbling process. The analog of this in the graph searching problem would be "recontamination," allowing previously cleared edges to become contaminated again in order to save searchers. In a preliminary version of this paper [10], we noted that we had been unable to find any graphs that would require more searchers if recontamination were disallowed and raised as an open problem the question: Can recontamination help? This question has now been resolved by LaPaugh [6], who has shown that, in our pebbling model for graph searching, recontamination can be disallowed without changing the search number for any graph; that is, there always exists a search plan for $G$ with $s(G)$ searchers that does not involve recontamination of any edges.

It follows from this result that the decision problem belongs to NP. One need only guess the sequence in which the edges are cleared, from which it is straightforward to check whether or not that sequence can be achieved using $K$ or fewer searchers. We now show that the problem is in fact NP-complete.

THEOREM 1. *The question: Given G and K, can G be cleared with K searchers? is NP-complete.*

PROOF. We prove this by providing a transformation from the following known NP-complete problem [2]:

MIN-CUT INTO EQUAL-SIZED SUBSETS
INSTANCE: Graph $G = (V, E)$ with $|V|$ even, positive integer $K$.
QUESTION: Is there a partition of $V$ into two subsets $V_1$ and $V_2$ with $|V_1| = |V_2| = \frac{1}{2}|V|$ such that $|\{\{u, v\} \in E: u \in V_1, v \in V_2\}| \leq K$?

For the proof, we need several lemmas about searching complete graphs. Let $K_n$ denote the complete graph with $n$ vertices. It was observed by Parsons that, for $n \geq 4$, $s(K_n) = n$. We need a somewhat different fact. At any point during the search of a graph, call a vertex *cleared* if all edges incident upon it are clear.

LEMMA 1. *Suppose that at some step t during a search of $K_M$, $M \geq 4$, the first vertex becomes cleared. Then there must have been at least $M - 1$ searchers on $K_M$ during this step.*

PROOF. Suppose that the vertex $v$ was cleared by clearing the edge $\{u, v\}$ during step $t$. The other $M - 2$ vertices all have both clear and contaminated edges incident upon them at this point, and therefore they must contain at least one

searcher each. Furthermore, the edge $\{u, v\}$ must have been traversed by some other searcher during this step. The lemma follows. □

LEMMA 2. *Suppose that the graph $G$ contains $m$ vertex-disjoint copies of $K_M$ (for some $M \geq 4$). Then, in the process of searching $G$, there must be for each $k$, $1 \leq k \leq m$, a point at which $k$ cliques have at least one cleared vertex (with respect only to internal clique edges), $m - k$ have none, and there is one clique with a cleared vertex that contains $M - 1$ or more searchers.*

PROOF. This follows directly from Lemma 1 and the fact that at any step at most one clique can go from zero to one (or more) such cleared vertex (vertices). □

LEMMA 3. *Suppose that $G$ contains $m$ vertex-disjoint copies of $K_M$ and that at some step during the search of $G$, there is a set $C_1$ of cliques, each containing one or more cleared vertices (again with respect only to internal clique edges), and a set $C_2$ of cliques containing no such cleared vertices. Then, if $\{u, v\}$ is an edge of $G$ such that $u$ belongs to a clique in $C_1$ and $v$ belongs to a clique in $C_2$, either $u$ or $v$ must contain a searcher.*

PROOF. If $\{u, v\}$ is clear, then $v$ has both clear and contaminated edges incident upon it and hence must contain a searcher. If $\{u, v\}$ is contaminated, then since $u$ has at least one clear edge incident upon it, $u$ must contain a searcher. □

We now proceed with the proof of Theorem 1. Let $G = (V, E)$ and $K > 0$ constitute a given instance of MIN-CUT INTO EQUAL-SIZED SUBSETS. We construct a corresponding instance of GRAPH SEARCHING as follows:

Let $n = |V|$, let $d$ be the maximum vertex degree in $G$, let $N = 6(d + K)$, and let $M = n(n + 2) \cdot N$. The graph to be searched consists of the following parts:

(i) for each vertex $v_i \in V$, an $M$-clique $C_i$;
(ii) an additional "special" $M$-clique $C_A$;
(iii) between each pair $C_i$, $C_j$ of $M$-cliques, $nN$ edges

$$\text{plus} \begin{cases} N \text{ additional edges} & \text{if either } i \text{ or } j \text{ is } A, \\ 3 \text{ additional edges} & \text{if } \{v_i, v_j\} \in E. \end{cases}$$

The edges in (iii) are added in such a way that no clique vertex is involved in more than one outside edge, which can be done since $M = n(n + 2) \cdot N \geq n^2N + N + 3d$.

Call the resulting graph $H = (U, F)$. The search number $s$ to be tested is given by

$$s = (M + 1) + \left(\frac{n}{2}\right)^2 nN + 3K.$$

We now show that $s(H) \leq s$ if and only if $G$ has the desired cut into equal-sized subsets.

First, suppose the desired cut exists for $G$, that is, there is a partition of $V$ into $V_1$ and $V_2$ with $|V_1| = |V_2|$ such that $K' \leq K$ edges join vertices in $V_1$ to vertices in $V_2$. We show how $H$ can be cleared with at most $s$ searchers.

First relabel the vertices (and corresponding cliques) so that $V_1 = \{v_1, v_2, \ldots, v_{n/2}\}$ and $V_2 = \{v_{(n/2)+1}, \ldots, v_n\}$. We clear the cliques in the order

$$C_1, C_2, \ldots, C_{n/2}, C_A, C_{(n/2)+1}, \ldots, C_n.$$

To clear the clique $C_i$ in its turn, place a searcher on each of its vertices that currently does not contain a searcher. Then use an $(M + 1)$th searcher to clear all the edges internal to $C_i$. Finally, clear each edge from $C_i$ to a later clique by moving the searcher from the edge's endpoint in $C_i$ along the edge to its endpoint in the later clique, where the searcher will be left as a guard until that later clique is cleared.

It is easy to see that the maximum number of searchers for this procedure must occur at some time when one of the cliques is being cleared and has $M + 1$ searchers itself. If the clique being cleared is not $C_A$, the total number of searchers is at most

$$(M + 1) + \left(\frac{n}{2} - 1\right)\left(\frac{n}{2} + 1\right)nN + \left(\frac{n}{2} - 1\right)(N + 3d)$$

$$\leq s - 3K - nN + \left(\frac{n}{2}\right)(N + N) < s.$$

If the clique being cleared is $C_A$, then the number of searchers is

$$(M + 1) + \left(\frac{n}{2}\right)^2 nN + 3K' \leq (M + 1) + \left(\frac{n}{2}\right)^2 nN + 3K = s.$$

Thus $s$ searchers suffice.

Now, suppose $H$ can be cleared using $s$ searchers. Consider a step in the search process at which $(n/2) + 1$ of the cliques have at least one cleared vertex (with respect only to the internal clique edges), $n/2$ do not, and $M - 1$ searchers are on a clique with a cleared vertex. Such a step exists by Lemma 2.

Suppose the clique with the $M - 1$ searchers is not $C_A$. Then by Lemma 3 the total number of searchers in use at that time must be at least

$$(M - 1) + \left(\frac{n}{2}\right)^2 nN + \frac{n}{2} N = s + \frac{n}{2} N - (3K + 2) > s,$$

a contradiction. Thus the clique with $M - 1$ searchers must be $C_A$, and the total number of searchers on endpoints of edges not corresponding to edges in $G$ must be (by Lemma 3) at least

$$(M - 1) + \left(\frac{n}{2}\right)^2 nN = s - (3K + 2).$$

If $V_1 = \{v_i: C_i$ contains a cleared vertex$\}$ and $V_2 = \{v_i: C_i$ does not contain a cleared vertex$\}$, we must have $|V_1| = |V_2| = n/2$, and the number of edges in $G$ joining a vertex in $V_1$ to a vertex $V_2$ is at most $\lfloor (3K + 2)/3 \rfloor \leq K$. Hence the desired cut exists for $G$.

We conclude that $H$ can be searched with $s$ searchers if and only if $G$ has a cut into equal-sized subsets with $K$ or fewer edges. Since $H$ and $s$ can be constructed easily in polynomial time from $G$ and $K$, we have a polynomial transformation and the theorem is proved. □

## 3. *The Special Case of Trees*

It follows from results of Parsons in [11] that the graph-searching problem, when restricted to trees, is in both NP and co-NP. (He gives a recursive forbidden subgraph characterization of those trees $T$ with $s(T) > k$, for each $k$, and it is

possible to argue, using his basic lemma (even without LaPaugh's result), that recontamination cannot help in the case of trees.)

We show that the tree-searching problem is in P and that the search number of a tree can in fact be found in linear time. The algorithms we use do not involve the standard dynamic programming tricks that one so often sees for problems on trees but use instead a rather intricate application of recursion. Our basic algorithm runs in time $O(n \log n)$, but it can be sped up to time $O(n)$ by making the stack explicit and including pointers for making shortcuts. The algorithm also can be modified to keep track of sufficient additional information so that a search plan, rather than just the search number, can be computed in time $O(n \log n)$. We do not know whether this can be reduced to time $O(n)$.

The basic idea behind the algorithm is given by a normal form theorem for optimal search procedures on trees. Given a tree $T = (V, E)$ and a vertex $v \in V$, we say that a subtree $T'$ of $T$ is a *branch at* $v$ if $v$ has degree 1 in $T'$ and $T'$ is a maximal subtree having this property. Parsons [11] proved the following lemma:

PARSONS' LEMMA. *For any tree $T$ and integer $k \geq 1$, $s(T) \geq k + 1$ if and only if $T$ has a vertex $v$ at which there are three or more branches that have search number $k$ or more.*

Notice that this result implies that a tree with search number $k$ must have at least $3^{k-1}$ edges, so the search number of an $n$ node tree always satisfies $s(T) \leq 1 + \log_3(n - 1)$.

Moreover, this result leads us to the key concept needed for our normal form theorem, the concept of the "avenue" of a tree. Intuitively, the avenue of a tree $T$ is a path $v_1, v_2, \ldots, v_r$ of two or more vertices such that $T$ can be cleared using $s(T)$ searchers by placing a searcher on $v_1$ and subsequently moving it along the avenue to $v_2, v_3, \ldots, v_r$, pausing long enough at each vertex $v_i$ along the path so that the nonavenue branches at $v_i$ can be cleared (one at a time) using the remaining $s(T) - 1$ searchers. Formally, a path $v_1, v_2, \ldots, v_r$ of two or more vertices is an *avenue* for $T$ if $v_1$ and $v_r$ each have exactly one branch with search number $s(T) = s$ (containing $v_2$ and $v_{r-1}$, respectively) and for every $j$, $2 \leq j \leq r - 1$, $v_j$ has exactly two branches with search number $s$ (containing $v_{j-1}$ and $v_{j+1}$, respectively). It is not hard to see that this definition implies that the avenue can be used inductively to search $T$ with $s(T)$ searchers in the manner indicated above. Our main structural result for trees is the following:

LEMMA 4. *If $s(T) = s$, then either (i) $T$ has a vertex $v$ such that all branches at $v$ have search number smaller than $s$, or (ii) $T$ has a unique avenue.*

PROOF. Suppose that (i) fails to hold for $T$, that is, that every vertex in $T$ has at least one branch with search number $s$. Consider the set $A$ of all edges $\{u, v\}$ of $T$ with the property that both the branch at $u$ containing $\{u, v\}$ and the branch at $v$ containing $\{u, v\}$ have search number $s$. Notice that any branch in $T$ that contains an edge $\{u, v\}$ of $A$ must have search number $s$, since it must entirely contain either the branch at $u$ containing $\{u, v\}$ or the branch at $v$ containing $\{u, v\}$, and adding edges and vertices to a graph cannot reduce its search number. By definition, an avenue can consist only of edges from $A$. Moreover, an avenue cannot exclude any of the edges in $A$, since the vertex on the avenue closest to that excluded edge would then have an additional branch with search number $s$. Thus, if an avenue exists, it must be $A$. The desired result will follow by showing that $A$ itself is an avenue.

First, we show that $A$ is nonempty and hence involves at least two vertices. To see this, for each vertex $v \in V$ let $e(v)$ denote some edge incident on $v$ such that the branch at $v$ containing $e(v)$ has search number $s$. Such an $e(v)$ must exist for each $v$ by our assumption above that (i) does not hold. Furthermore, since there are only $|V| - 1$ edges in $T$, there must exist two vertices $u$ and $v$ for which $e(u) = e(v) = e$. The edge $e$ therefore belongs to $A$.

We next show that $A$ is a path. It contains no more than two edges incident upon any single vertex, since the existence of a vertex with three such incident edges would imply $s(T) \geq s + 1$ by Parsons' Lemma. It remains to show that $A$ forms a *connected* subgraph of $T$. Suppose it were not connected, and let $e$ be any edge on a path in $T$ joining two disconnected components of $A$, that is, a path consisting entirely of edges from $T - A$. For each endpoint of $e$, the branch at the endpoint that includes $e$ must contain an edge from $A$ and hence must have search number $s$. But this implies that $e$ must belong to $A$, contradicting the choice of $e$. Therefore, no such paths exist in $T$ and $A$ must be connected. Thus $A$ is a path.

Finally, we need to show that no vertex $v$ in the path $A$ has more than the required number of branches with search number $s$. For this it suffices to show that every branch at $v$ that involves no edges from the avenue $A$ has search number less than $s$. Let $u$ be the vertex adjacent to $v$ in such a branch at $v$. Since $\{u, v\}$ is not in $A$, and since the branch at $u$ containing $\{u, v\}$ has search number $s$ (it contains an edge in $A$, in fact all of $A$), it follows immediately from the definition of $A$ that the branch at $v$ containing $\{u, v\}$ does not have search number $s$. Therefore, $A$ is an avenue for $T$ and, indeed, is the *unique* avenue for $T$. □

We note that the "or" of the theorem statement is exclusive, since a vertex of the form required for (i) cannot be on the avenue, and, if it were off the avenue, the branch at that vertex that included the avenue would have search number $s$, contradicting the requirement that all its branches have search number less than $s$.

We call a vertex $v$ in a tree $T$ a *hub* of $T$ if all branches at $v$ have search number less than $s(T)$, and we regard such a hub as an avenue of length zero, consisting of a single vertex. We note that in this case the avenue need not be unique. Indeed, in a minimal tree with search number $s$, every vertex of degree greater than 1 is a hub.

For purposes of exposition, it is convenient to assume in the remainder of this section that every tree under construction has a specified vertex called the *root*. It is clear later how such an assumption arises naturally from the way in which we decompose trees to compute search numbers. (The choice of root will not affect the search number of a tree.)

From Lemma 4, we can divide all rooted trees into four *types* (see Figure 2), depending on the location of the root:

(1) *Type H.* The root coincides with a *hub* (avenue of length zero) of $T$.
(2) *Type E.* The root is on a branch at one of the *endpoints* $v_1$ or $v_r$ of the avenue of $T$ (possibly it *is* one of the endpoints, but in this case we require that $v_1 \neq v_r$, that is, that the avenue have nonzero length).
(3) *Type I.* The root is one of the *interior* nodes $v_2, \ldots, v_{r-1}$ of the avenue for $T$.
(4) *Type M.* The root is on a (*middle*) branch with search number less than $s(T)$ at an interior node of the avenue, but not on the avenue itself.

For a tree of type $M$, we call the branch on which the root lies the *M-tree* of $T$. Notice that the *M-tree* of $T$ always has a search number less than that for $T$.

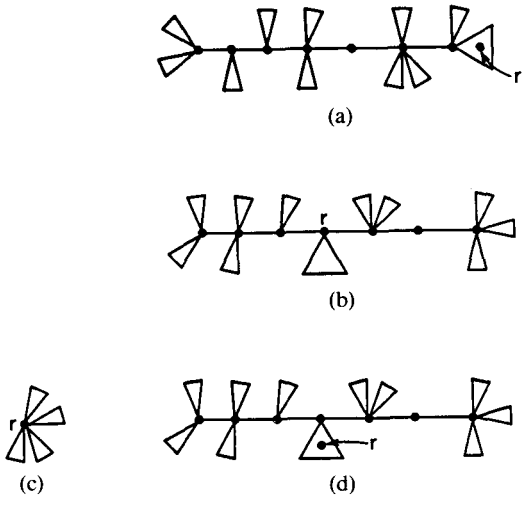


FIG. 2. Four types of rooted trees. (a) Type *E*. (b) Type *I*. (c) Type *H*. (d) Type *M*.

Our algorithm *compute-info* works by recursively computing an information record *info*($T$) = [*type*, *s*, *M-info*] associated with the tree $T$ (whose root has been chosen arbitrarily). The three fields are defined as follows:

(a) *Type*. One of *H*, *E*, *I*, *M*, the type of $T$.
(b) *s*. The search number for $T$.
(c) *M-info*. If the type is *H*, *E*, or *I*, then *M-info* is *nil*. Otherwise, *M-info* is the info-record for the *M-tree* of $T$.

The computation of *info*($T$) for a rooted tree $T$ proceeds as follows, based on the degree of the root $r$ of $T$:

(1) If $r$ has degree 1, there are two cases. If $T$ is just an edge, then *info*($T$) is the record [*E*, 1, *nil*]. Otherwise, we let $T'$ be obtained from $T$ by moving its root from $r$ to the single neighbor of $r$, recursively compute *info*($T'$), and transform this into *info*($T$) using a special routine called *re-root*.
(2) If $r$ has degree 2 or more, we then split $T$ into two trees $T_1$ and $T_2$, both rooted at $r$ and having no other nodes in common (subject to this requirement and the requirement that each contain at least one edge, the two trees can be chosen arbitrarily). We then recursively compute *info*($T_1$) and *info*($T_2$), and we compute *info*($T$) from these using the special routine *merge*, which is the heart of our algorithm.

The routine *re-root* is simple, and we explain it first. Suppose *info*($T'$) = [*type'*, *s'*, *M-info'*]. Then *info*($T$) = [*type*, *s*, *M-info*] is determined as follows: The search number $s$ is the same as $s'$ (since $T$ and $T'$ are identical trees, differing only in the choice of root). The value of *type* (and *M-info* when *type* = *M*) is as follows:

(1) If $T'$ is type *E*, then *type* = *E*.
(2) If $T'$ is type *H*, then *type* = *E*, since a leaf cannot be a hub.
(3) If $T'$ is type *I*, and $s' = 1$ (i.e., $T'$ is a path), then *type* = *E*. If $T'$ is type *I* and $s' \neq 1$, then *type* = *M* and *M-info* = [*E*, 1, *nil*].
(4) If $T'$ is type *M*, then *type* = *M* and *M-info* = *re-root*(*M-info'*).

It is easy to verify from this description that *re-root* works as required:

LEMMA 5. *For any tree $T$ that is not just a single edge, the algorithm re-root correctly computes info($T$) from info($T'$), where $T$ is rooted at a leaf $r$ and where $T'$ is identical to $T$ except for being rooted at the neighbor of $r$.*

We now come to the routine *merge*, which can be described in terms of the following cases:

We are merging two trees $T_1$ and $T_2$ with info-records [*type1*, *s1*, *M-info1*] and [*type2*, *s2*, *M-info2*], respectively, to form the info-record [*type*, *s*, *M-info*] for $T$. Without loss of generality, we may assume that $s1 \geq s2$.

If $s1 = s2$, we have the following five cases:

*Case* 1. If *type1* = *type2* = *H*, then the info-record for $T$ is [*H*, *s1*, *nil*].

*Case* 2. If *type1* = *H*, and *type2* = *E*, or vice versa, then the info-record for $T$ is [*E*, *s1*, *nil*].

*Case* 3. If *type1* = *type2* = *E*, then the info-record for $T$ is [*I*, *s1*, *nil*].

*Case* 4. If *type1* = *I* and *type2* = *H*, or vice versa, then the info-record for $T$ is [*I*, *s1*, *nil*].

*Case* 5. (At least one of $T_1$, $T_2$ is type *M*, both are type *I*, or one is type *I* and the other is type *E*.) The info-record for $T$ is [*H*, *s1* + 1, *nil*].

If $s1 > s2$, we have the following two cases:

*Case* 6. If *type1* = *H*, *E*, or *I*, then the info-record for $T$ is [*type1*, *s1*, *nil*].

*Case* 7. (Note that *type1* = *M* and $s1 > s2$.) Call *merge* on the two info-records *M-info1* and [*type2*, *s2*, *M-info2*], with the result [*type'*, *s'*, *M-info'*]. If $s' < s1$, then the info-record for $T$ is [*M*, *s1*, [*type'*, *s'*, *M-info'*]]. If $s' = s1$ (we cannot have $s' > s1$), then the info-record for $T$ is [*H*, *s1* + 1, *nil*].

LEMMA 6. *The algorithm merge correctly computes info($T$) from info($T_1$) and info($T_2$), where $T$ is a tree rooted at $r$, and $T_1$ and $T_2$ are trees rooted at $r$, otherwise disjoint, whose union is $T$.*

PROOF. The proof is by case analysis, corresponding to the cases in the description of the algorithm.

*Case* 1. $s1 = s2$, *type1* = *type2* = *H* (Figure 3a). In this case, both trees have the root as hub, and their union is a tree in which all branches at the root have search number less than $s1 = s2$. Since the search number cannot decrease, it stays at *s1* and the new tree has the root as a hub.

*Case* 2. $s1 = s2$, *type1* = *H*, *type2* = *E* (Figure 3b). The new tree can be searched with $s1 = s2$ searchers by marching along the avenue of $T_2$ and down the branch containing $r$ until $r$ is reached. $T$ has an avenue of length at least one, with $r$ as an endpoint, and hence it is type *E*.

*Case* 3. $s1 = s2$, *type1* = *type2* = *E* (Figure 3c). The avenue of $T$ is the shortest path that contains both the avenues of $T_1$ and $T_2$, and this avenue can be used to search $T$ with *s1* searchers. Since $r$ is one of the interior points on the avenue, $T$ is type *I*.

*Case* 4. $s1 = s2$, *type1* = *I*, *type2* = *H* (Figure 3d). The root of $T_1$ is a node on the interior of its avenue, and after combination with $T_2$ the off-avenue branches

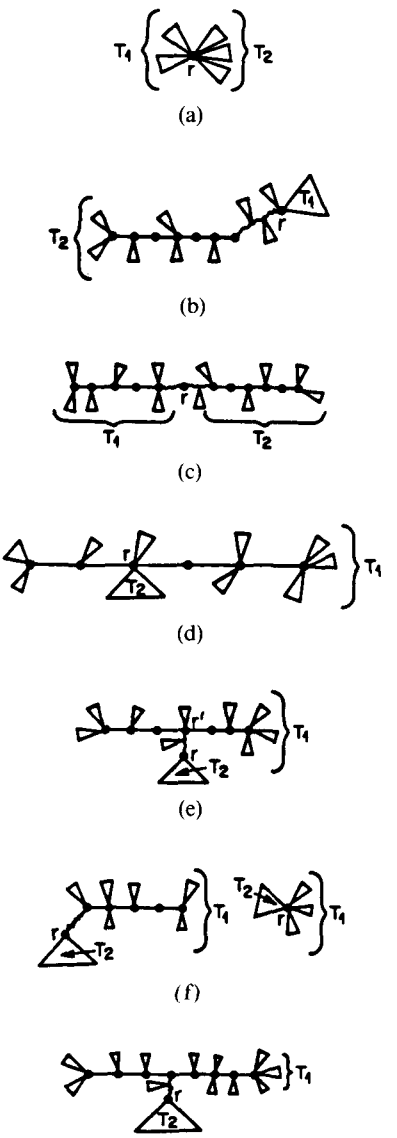


FIG. 3. Cases of procedure *merge*. (a) $T_1$, $T_2$ Type $H \rightarrow T$ Type $H$. (b) $T_1$ Type $H$, $T_2$ Type $E \rightarrow T$ Type $E$. (c) $T_1$, $T_2$ Type $E \rightarrow T$ Type $I$. (d) $T_1$ Type $I$, $T_2$ Type $H \rightarrow T$ Type $I$. (e) $T_1$ Type $M$ or $I \rightarrow T$ Type $H$. (f) $T_1$ Type $H$, $E$, or $I \rightarrow T$ Type $H$, $E$, or $I$. (g) $T_1$ Type $M \rightarrow T$ Type $M$ or $H$.

at that root will all continue to have a search number less than $s1 = s2$. Thus, $T$ has the same info-record as $T_1$, that is $[I, s1, nil]$.

*Case* 5. $s1 = s2$, $type1 = I$, $type2 = E$ or $I$, or $type1 = M$ (Figure 3e). Let $r'$ be the root of $T_1$ if $type1 = I$, or the node on the avenue of $T_1$ closest to the root of $T_1$ if $type1 = M$. It is easy to see that $r'$ is a node with three branches having search number $s1$, so $s1 + 1$ searchers are needed for $T$. Since $s1 + 1$ searchers are obviously also sufficient and since the root of $T$ has only branches with search number less than $s1 + 1$, $T$ is of type $H$ and has search number $s1 + 1$.

*Case* 6. $s1 > s2$, $type1$ is $H$, $E$, or $I$ (Figure 3f). If $type1$ is $H$, then the root continues to have only branches with search number less than $s1$ in $T$, so $T$ has search number $s1$ and is of type $H$. If $type1$ is $E$, then $T$ can be searched with $s1$ searchers by marching along the avenue of $T_1$ and down the branch containing $r$

until $r$ is reached. Thus $T$ has an avenue of length at least one, with $r$ on a branch at an endpoint (possibly it *is* the endpoint), so $T$ is of type $E$. If *type1* = $I$, the argument of Case 4 applies, so $T$ is of type $I$ and has search number $s1$.

*Case* 7. $s1 > s2$ and *type1* = $M$ (Figure 3g). The search number for $T$ depends on the search number $s(T')$ of the union of the *M-tree* for $T_1$ with $T_2$. The search number for $T'$ cannot be greater than $s1$, since the *M-tree* has search number less than $s1$, and so does $T_2$. If $s' = s(T') < s1$, then the avenue of $T$ is exactly the same as the avenue of $T_1$, the same search number $s1$ suffices, and $T'$ is now the *M-tree* of $T$. If $s' = s1$, then the head of the branch corresponding to $T'$ in $T$ has three branches with search number $s1$, so the search number of $T$ is at least $s1 + 1$. Moreover, the root $r$ has only branches with search number less than $s1 + 1$, so $s1 + 1$ searchers suffice and $T$ is type $H$. □

THEOREM 2. *The algorithm compute-info correctly computes info(T) for any rooted tree T with n nodes in $O(n \log n)$ time.*

PROOF. The correctness of the algorithm follows immediately from the preceding two lemmas. For the time bound, let $S(T)$ denote the time required to compute the search number (info-record) for $T$. Then we have

$$S(T) \leq R(T) + S(T_1) + S(T_2) + M(T_1, T_2),$$

where $R(T)$ is the cost of calling *re-root* at most once, $T_1$ and $T_2$ are the trees that $T$ is split into, and $M(T_1, T_2)$ is the cost of executing *merge* for these two trees. Since the recursive calls to *re-root* have arguments with strictly decreasing search numbers, we immediately have that $R(T)$ is $O(s(T))$. To estimate $M(T_1, T_2)$, let us consider it as a function $M(s_1, s_2)$, of the search numbers of the two subtrees. In Cases 1–6, $M(s_1, s_2)$ is simply a constant. In Case 7, however, there is a recursive call of *merge* in which $s_1$ is decreased by at least 1. Hence,

$$M(s_1, s_2) \leq M(s_1 - 1, s_2) + O(1),$$

and the procedure terminates after at most $s_1 \leq s(T)$ such calls. Therefore,

$$S(T) \leq S(T_1) + S(T_2) + O(s(T)).$$

It follows that $S(T) = O(n \cdot s(T))$ and, since $s(T) = O(\log n)$, $S(T) = O(n \log n)$. □

In order to reduce this time bound to $O(n)$, we need to avoid (or shortcut) the recursive calls of *re-root* so that $R(T)$ will be a constant, and we similarly need to shortcut the recursive calls to *merge* that occur in Case 7. By a careful use of pointers, we can indeed avoid much of the time that is spent traversing up and down the implicit *M-info* stack and hence obtain the following:

THEOREM 3. *The search number for a tree can be computed in linear time.*

PROOF. A tree of type $M$ heads a chain of *M-trees* having strictly decreasing search numbers described by its *M-info* field. The recursive calls of *re-root* and *merge* descend along this chain. In our improved implementation, we represent this chain explicitly by a doubly linked list with several additional pointers that will allow us to make shortcuts. Specifically, we replace the *M-info* field for a tree $T$ with the follwing pointers:

(1) a pointer to the info-record for the *M-tree* of $T$, labeled $t$ (for "tight") if the search number of the *M-tree* is exactly one less than that of $T$, and otherwise labeled $l$ (for "loose");

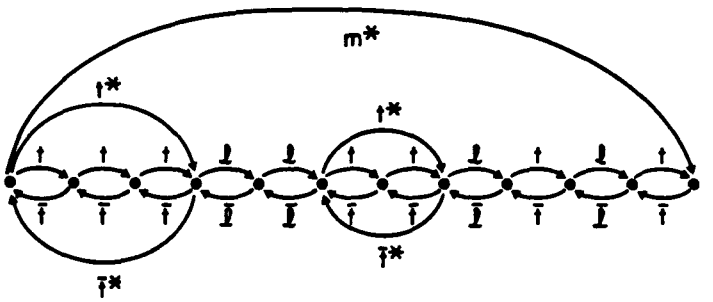


FIG. 4. An *M*-tree chain with pointers.

(2) the *reverse* of this pointer, labeled $\bar{t}$ or $\bar{l}$, depending on whether the original pointer was labeled $t$ or $l$;
(3) the *closures* of the pointers labeled $t$ and $\bar{t}$ (these are pointers $t^*$ and $\bar{t}^*$ pointing to the endpoints of the longest chains of $t$ or $\bar{t}$ pointers, respectively, starting at $T$ and are undefined for trees that are strictly in the interior of such a chain);
(4) a pointer $m^*$ to the endpoint of the *M-info* chain starting at $T$, that is, to the info-record reached from $T$ by following $t$ and $l$ pointers until a tree with *nil M-tree* pointer is encountered.

These pointers are illustrated in Figure 4. It is easy to see how such pointers can be maintained within the algorithm without changing its asymptotic time complexity.

It is now immediate that *re-root* can be performed in constant time. The recursive calls to *re-root* simply travel down the *M-tree* chain to its end, where in constant time the algorithm modifies the info-record for that last tree appropriately. Since the end of the *M-tree* chain can be be accessed directly in constant time via the $m^*$ pointer, the entire procedure can be performed in constant time.

We now come to the procedure *merge*. The first six cases of *merge* continue to require only constant time. In Case 7, the algorithm recursively calls *merge* on the info-records for $T_2$ and the *M-tree* of $T_1$. Repeated recursive calls then travel down the *M-tree* chain for $T_1$ until either it ends or a tree with search number *s2* or less is encountered. If the chain includes a tree with search number less than *s2*, but none with search number equal to *s2*, then recursive calls to *merge* may continue but will now travel down the *M-tree* chain for $T_2$; otherwise, there will be no further recursive calls to *merge*.

We can provide a shortcut for this procedure by examining the info-record for the tree $T^*$ pointed at by the $t^*$ pointer of $T_1$ and by using our other pointers appropriately. (If $T_1$ does not have a $t^*$ pointer, we define $T^* = T_1$.) We have two cases:

*Case* 1. $s(T^*) \leq s(T_2)$. In this case, by definition of the $t$ and $t^*$ pointers, there is some tree $T_3$ in the *M-tree* chain for $T_1$, between $T_1$ and $T^*$, having $s(T_3) = s(T_2)$. In the original algorithm, the recursive calls to *merge* end after merging $T_3$ and $T_2$. Since $s(T_3) = s(T_2)$, we also know that this last merge must be in one of Cases 1–5. We consider separately two possibilities: (a) the last merge is in one of Cases 1–4 and (b) the last merge is in Case 5. To see that we can determine which of these two holds in constant time, observe that whenever (a) holds, $T_3$ cannot be of type $M$, and hence we must have $T_3 = T^*$. Thus we can determine whether (a) holds by first following the $t^*$ pointer for $T_1$ to $T^*$, then

testing whether $s(T^*) = s(T_2)$, and finally checking that $T^*$ and $T_2$ satisfy the criteria for one of Cases 1–4.

If (a) holds, we know from Cases 1–4 of *merge* that the result of merging $T_2$ and $T_3 = T^*$ is an info-record of the form [*type''*, *s2*, *nil*], and we can compute that info-record in constant time. As the recursion is unwound, the first subcase of Case 7 will apply at each level, so the final result of merging $T_1$ and $T_2$ will be an info-record with type $M$, search number *s1*, and *M-info* chain identical to that for $T_1$ down to $T_3 = T^*$, with $T_3 = T^*$ replaced by [*type''*, *s2*, *nil*]. Thus we can make this change in constant time without explicitly performing the recursion.

If (b) holds, we know that the result of merging $T_3$ and $T_2$ will be the info-record [$H$, *s2* + 1, *nil*]. Hence, as the recursion is unwound, the second subcase of Case 7 will apply at each level; so the final result of merging $T_1$ and $T_2$ will be the info-record [$H$, *s1* + 1, *nil*]. Again, this can be computed in constant time without explicitly performing the recursion (and without explicitly finding $T_3$). Therefore, it follows that *merge* can be performed in constant time whenever $s(T^*) \leq s(T_2)$.

*Case* 2. $s(T^*) > s(T_2)$. In this case, we first check the endpoint of the *M-tree* chain from $T_1$ using the pointer $m^*$. If its search number is greater than that for $T_2$, then we know that the original algorithm would continue the recursive calls to *merge* down to that point, with the last call being in Case 6 of *merge*. This last merge simply results in the info-record of the tree with the larger search number, namely, the one on the end of the chain for $T_1$, and hence the outcome of merging $T_1$ and $T_2$ is just the info-record for $T_1$.

Otherwise, we locate two special trees $T_3$ and $\bar{T}_3$ on the *M-tree* chain for $T_1$. The tree $T_3$ is the first on the chain to have $s(T_3) \leq s(T_2)$. The tree $\bar{T}_3$ is the last on the chain to have $s(\bar{T}_3) > s(T_2)$ and that does not have an incoming $t$ pointer. These trees can be found on $O(s(T_2))$ time by following first the $m^*$ pointer from $T_1$ and then backing up the chain using the $\bar{T}$ and $\bar{t}^*$ pointers. Moreover, it is easy to see that the recursive calls to *merge* will continue down to $T_3$ and that the *M-tree* chain for $T$ (the result of merging $T_1$ and $T_2$) will be identical to that for $T_1$ on the portion preceding $\bar{T}_3$. Thus we need only call *merge* recursively on $T_3$ and $T_2$ and modify the portion of the chain from $\bar{T}_3$ on accordingly, much as was done in Case 1. If the result of merging $T_3$ and $T_2$ has search number less than that for the predecessor of $T_3$ on the chain, then the resulting info-record (along with its chain) simply replaces the portion of the chain from $T_3$ on. Otherwise, the portion from $\bar{T}_3$ on is replaced by the single info-record having type $H$, search number one greater than that for $\bar{T}_3$, and *nil M-tree* chain. A constant amount of time may also be needed to repair pointers.

We now combine the results of Cases 1 and 2 to improve our timing analysis from Theorem 2. We have just shown how to implement *merge* in time $M(s(T_1), s(T_2))$ satisfying

$$M(s(T_1), s(T_2)) \leq M(s(T_2), s(T_2)) + O(s(T_2)),$$

since the tree $T_3$ in our last case has $s(T_3) \leq s(T_2)$. By the recurrence

$$M(s(T_1), s(T_2)) \leq M(s(T_1) - 1, s(T_2)) + O(1)$$

shown in the proof of Theorem 2 (which is easily extended to the new and faster version of the algorithm being analyzed here), we can now conclude that

$$M(s(T_1), s(T_2)) \leq O(S(T_2)),$$

where $s(T_2) \leq s(T_1)$. Consequently,

$$\begin{aligned} S(T) &\leq S(T_1) + S(T_2) + O(\min\{s(T_1), s(T_2)\}) \\ &\leq S(T_1) + S(T_2) + O(\log(\min\{|T_1|, |T_2|\})). \end{aligned}$$

This yields $S(T) = O(|T|)$, as claimed. (It is well known that the recurrence defined by $f(1) = f(2) = 1$ and, for $n \geq 3$,

$$f(n) = \max_{1<i<n} \{\lceil \log_2(\min\{i, n+1-i\}) \rceil + f(i) + f(n+1-i)\}$$

satisfies $f(n) = O(n)$. An easy way to verify this is to prove that, for $n \geq 2$, $f(n) \leq 3n - 4 - \lceil \log_2 n \rceil$ by a straightforward induction.) □

In addition, a slight modification of our algorithm allows us to keep track of enough information to compute a *search plan* for $T$ that uses $s(T)$ searchers, although we need to apply the algorithm repeatedly in such a way that a total of $O(n \log n)$-time is needed to find the complete plan.

We do this by augmenting each info-record to include an additional entry *path*, which gives the endpoints of a path in the tree that is guaranteed to contain the avenue. The routine *re-root* has no effect on *path*. The routine *merge* updates the *path* as follows, where we let $path(T_1) = [a_1, b_1]$ and $path(T_2) = [a_2, b_2]$:

*Case* 1. $path(T) = [r, r]$, where $r$ denotes the root.

*Case* 2. Assume $type1 = H$. $path(T) = [r, b_2]$, where $b_2$ is the endpoint of $path(T_2)$ farther from the root.

*Case* 3. $path(T) = [a_1, b_2]$, where $a_1$ and $b_2$ are the endpoints of $path(T_1)$ and $path(T_2)$ farthest from the root.

*Case* 4. $path(T) = [a_1, b_1]$.

*Case* 5. $path(T) = [r, r]$.

*Case* 6. $path(T) = [a_1, b_1]$ if $type1 = H$ or $I$. Otherwise, $path(T) = [a_1, r]$, where $a_1$ is the endpoint of $path(T_1)$ farther from the root.

*Case* 7. $path(T) = [r, r]$ if $s' = s1$, and, otherwise, $path(T) = [a_1, b_1]$.

The correctness of the computation of *path* follows immediately from the arguments given in the proof of Lemma 6. Note that our linear-time algorithm for computing search number can be modified so that it also computes *path* information while still running in linear time.

In order to construct a search plan for $T$ using the resulting value of $path(T)$, we proceed as follows: Let $path(T) = [a, b]$, and let $a = u_1, u_2, \ldots, u_p = b$ denote the path in $T$ joining $a$ and $b$. The edges of $T$ that are not in the path are partitioned into branches $T_1, T_2, \ldots, T_m$, each of which contains exactly one vertex on the path and has search number at most $s(T) - 1$, since the path contains the avenue of $T$. A search plan using $s(T)$ searchers is obtained by placing one searcher on $u_1$ clearing all branches at $u_1$ by using the remaining searchers, moving the single searcher along the edge from $u_1$ to $u_2$, clearing the branches at $u_2$ by using the remaining searchers, and so on. Thus the search plan can be constructed simply by inserting search plans for each of $T_1, T_2, \ldots, T_m$ in the appropriate places in this "skeletal" search plan for $T$, which describes the movements for exactly one of the searchers. This can be done by recursively calling the search plan construction algorithm, which itself calls the algorithm for constructing *path*. The recursion tree

has depth at most $s(T)$, since the search numbers for the branches always decrease. The total number of nodes in the subproblems at each level is $O(n)$, because the branches are connected and partition the edges of $T$. It follows that the time to construct the complete search plan is $O(n \log n)$, since *path* is constructed in linear time for each tree, as is the skeletal search plan for the tree. Thus we have

THEOREM 4. *A search plan using $s(T)$ searchers can be constructed for any tree $T$ in time $O(n \log n)$, where $n = |T|$.*

Although we do not know whether this can be reduced to $O(n)$, we observe that, if one requires that searchers always be moved from place to place along edges of the graph (i.e., cannot be removed from the graph completely and later placed elsewhere) and that a search plan must describe each move along a single edge individually, then there exist trees whose search plans require $\Omega(n \log n)$ steps. One such class of trees can be constructed as follows: Let $T_k$ be a tree with search number $k$ having $3^{k-1} + 1$ vertices. Such a tree is easily constructed using Parsons' lemma. Let $T_k^*$ be the tree constructed by taking two copies of $T_k$ and joining them by a path of length $3^k$. The search number of the resulting tree is either $k$ or $k + 1$ (depending on the vertices of the copies of $T_k$ we select to join by the path), and the tree has $O(3^k)$ vertices. However, it is easy to see that at least $k$ searchers must traverse the path from one copy of $T_k$ to the other, and hence a search plan that describes each move along an edge individually must have length at least $k \cdot 3^k$, which is $\Omega(n \log n)$. By attaching a leaf to each vertex of the path, we can obtain a similar example without degree-2 vertices.

## 4. *Characterizing Graphs with Search Number K*

Although the search number of a graph does not seem to correspond directly to any of the standard measures on graphs, it is not difficult to see relationships between search number and connectivity. Our NP-completeness proof related the search number to the minimum cut of a graph into equal-sized subsets. It is also easy to see that any $K$-vertex-connected graph requires at least $K$ searchers. (A graph has vertex connectivity equal to the minimum number of vertices whose deletion will disconnect the graph.) This particular relationship goes only one way, however; it is easy to construct trees with arbitrarily high search number, even though all trees have vertex connectivity one. The problem of characterizing precisely those graphs with search number $K$ or less ($K$ fixed) turns out to be interesting and nontrivial, even for $K$ as small as 2 or 3. In this section we describe our characterization results for $K \leq 3$.

We allow multiple edges and self-loops in all our graphs. The *reduction* of a graph is the graph that results by repeatedly applying the operation of replacing any degree-2 vertex and its two incident edges by a single edge $\{a, b\}$ joining its two neighbors (where possibly $a = b$) until no degree-2 vertices remain. Two graphs are *homeomorphic* if they have the same reduction, and each is said to be a *homeomorph* of the other. Homeomorphism provides a topological notion of "equivalence" for graphs, when viewed as structures to be searched, and homeomorphic graphs clearly have the same search number. Hence, there will be no loss of generality in restricting our characterizations to reduced graphs.

It is also useful to have a topological notion of one graph being "contained" in another. Roughly speaking, we would like to say that a graph $H$ is contained in a graph $G$ if the structure of $H$ is embedded in $G$ in such a way that any plan for searching $G$ with $k$ searchers must also tell us how to search $H$ using

$k$ searchers. For instance, if $H$ is a subgraph of $G$, or of any homeomorph of $G$, then $s(H) \leq s(G)$, and any search plan for $G$ can be easily translated into a search plan for $H$ that needs no additional searchers. However, we can make our notion of containment even stronger than this, while still preserving the intuitive concept we wish to capture, by allowing the additional operation of "contraction." A *contraction* of a graph $G$ is any graph obtained from it by repeatedly applying the operation of choosing an edge, coalescing its endpoints, and then deleting the edge (now a loop). It is easy to see that a contraction $H$ of a graph $G$ can have no greater search number than $G$. Moreover, given any edge-clearing sequence for $G$, its restriction to just those edges that belong to $H$ must yield an edge-clearing sequence for $H$ that requires no additional searchers. Thus, we say that the graph $G$ *contains* the graph $H$ whenever some homeomorph of $G$ has a subgraph contractible to $H$. An equivalent way of saying this is that the graph $G$ contains the graph $H$ if and only if $H$ can be obtained from $G$ by a sequence of operations of the following three types:

(1) Replace any edge $\{u,v\}$ by the two edges $\{u, w\}$ and $\{w, v\}$, where $w$ is a new degree-2 vertex;
(2) Delete any edge or vertex;
(3) Coalesce the endpoints of any edge and delete that edge.

It is immediate that if $G$ contains $H$, then $s(G) \geq s(H)$, since none of these operations can increase the search number. Notice that, although Operation 1 can be used to increase the number of edges and vertices in the graph, it merely transforms the graph to another that is equivalent under homeomorphism and, therefore, that is topologically no more complicated.

It is easy to see that the 1-searchable graphs are just the paths (graphs homeomorphic to an edge). Our characterization result for $K = 2$ is as follows:

THEOREM 5. *For any reduced graph $G$, the following are equivalent*:

(*a*) $s(G) \leq 2$;
(*b*) *$G$ does not contain any of the graphs in Figure* 5;
(*c*) *$G$ consists of a path $a_1, a_2, \ldots, a_r$ in which each consecutive pair $a_i, a_{i+1}$ is joined by either one or two edges, along with an arbitrary number of individual edges and self-loops attached to each $a_i$ (and otherwise disjoint).*

PROOF

(a) $\rightarrow$ (b). By inspection, it is easy to verify that none of the graphs in Figure 5 can be searched by two searchers. Thus, if $G$ were to contain one of these graphs, its search number would necessarily exceed 2.

(b) $\rightarrow$ (c). Assume that (b) holds for $G$. Since $G$ does not contain the graph of Figure 5c, all biconnected components of $G$ must be either edges or cycles. A cycle can have at most two vertices of degree 3 or more, for otherwise $G$ would contain the graph in Figure 5b. Hence, the only biconnected components of $G$ are single edges, self-loops, or pairs of parallel edges. This implies that $G$ must be a tree, except possibly for loops and for tree edges that occur in parallel pairs. We claim that the removal of all degree-1 vertices from $G$ must result in a path, again possibly with loops or path edges that occur in parallel pairs, which is identical to the form given by (c). Suppose not. Then there must be a node in $G$ adjacent to three distinct vertices, other than itself, none of which is a leaf. Each of those vertices must have degree 3 or more in $G$, since $G$ has no vertices of degree 2. But then the containment operations can be applied to $G$ to produce the graph in Figure 5a (notice that the

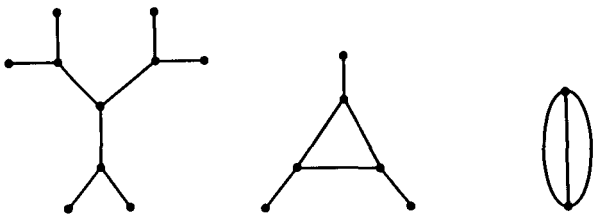


FIG. 5. Forbidden subgraphs for 2-searchable reduced graphs.

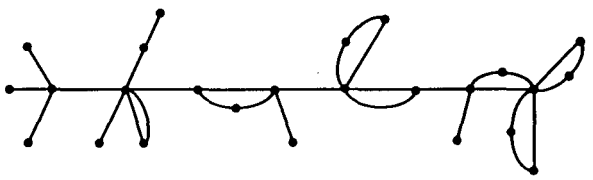

FIG. 6. Typical graph $G$ with $s(G) = 2$.

degree-3 vertices may have incident loops, which can be transformed into pairs of edges using a combination of Operations 1 and 2). This is a contradiction to our assumption that $G$ does not contain this graph, from which it follows that $G$ must have the required form.

(c) $\rightarrow$ (a). The graph $G$ can be searched by two searchers by having them march along the path given by condition (c), clearing the edges and pairs of edges on the path as they move along. At each node of the path, one of them is free to clear the hanging edges and self-loops at that node. □

Figure 6 illustrates a typical 2-searchable graph.

The situation for $K = 3$ is somewhat more complicated, particularly for graphs that are not biconnected. We first characterize the biconnected graphs that have search number three. A biconnected graph $G$ is *outerplanar* if it has a planar embedding in which a single face includes all of its vertices. The edges of that face are called *boundary edges*, and the remaining edges are called *chords*. Fix an outerplanar embedding of $G$, so that the boundary edges and chords are well defined, and consider any simple path $P$ forming part of the boundary of $G$. A chord joining two vertices of $P$ *spans* all the edges of $P$ in the subpath joining its two endpoints. Two such chords are *nested* if there is some edge of $P$ spanned by both of them. We say that two boundary edges of $G$ are *opposing poles* if neither of the two boundary paths joining their endpoints (excluding the two edges themselves) has a pair of nested chords. Whenever such a pair of opposing poles exists for some outerplanar embedding of $G$, we say that $G$ is *bipolar*. The special biconnected graph consisting of a single edge will be declared to be bipolar by default, and its single edge will be regarded as an opposing pole with itself.

For example, the graph $G$ of Figure 7 is bipolar, because the edges $\{v_8, v_9\}$ and $\{v_4, v_5\}$ are opposing poles. Notice that the edges $\{v_1, v_2\}$ and $\{v_6, v_7\}$ are not opposing poles, because the path $v_1, v_{10}, v_9, v_8, v_7$ has a pair of nested chords.

The key to our characterization for $K = 3$ is the following:

LEMMA 7. *For a reduced biconnected graph $G$, the following are equivalent*:

(*a*) $s(G) \leq 3$;
(*b*) *$G$ does not contain any of the graphs in Figure* 8;
(*c*) *$G$ is outerplanar and bipolar.*

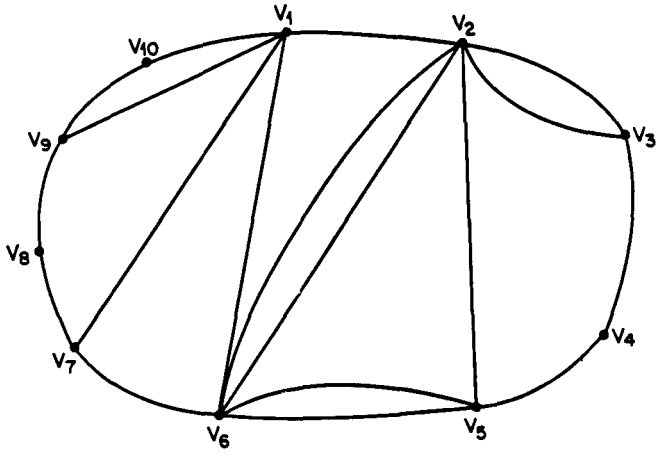


FIG. 7. An outerplanar graph with two poles.

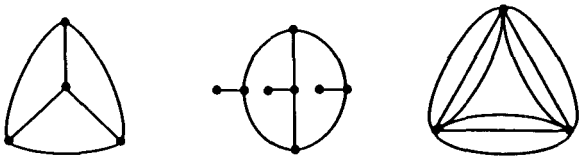


FIG. 8. Forbidden subgraphs for 3-searchable reduced biconnected graphs.

PROOF

(a) → (b). By inspection, it is easy to verify that none of the graphs in Figure 8 can be searched by three searchers. Thus, if $G$ were to contain one of these graphs, its search number would necessarily exceed 3.

(b) → (c). It follows from the standard forbidden subgraph characterization of outerplanar graphs (e.g., see [4]) that a reduced biconnected graph is outerplanar if and only if its does not contain either of the graphs in Figures 8a and b. Since $G$ does not contain either of these graphs by assumption, it is outerplanar. To see that $G$ must also be bipolar, fix an outerplanar embedding of $G$ and consider a maximal-length boundary path $P$ that has no nested pair of chords (conceivably, it might be just a single vertex). If $P$ includes all edges on the boundary of $G$, then we are done, since its two extreme edges must be opposing poles. Similarly, we are also done if $P$ includes all but one edge of the boundary, since then the excluded edge and either extreme edge of $P$ must be opposing poles. Otherwise, let $e$ and $e'$ denote the two boundary edges adjacent to $P$ on either side. Then, by the maximality of $P$, the addition of either $e$ or $e'$ to $P$ would cause it to have a pair of nested chords, with $e$ or $e'$ being a common boundary edge spanned by the nested chords. Hence, the longer path $P'$ formed by adding *both* $e$ and $e'$ to $P$ must include two pairs of nested chords that (by the outerplanarity of $G$) are "disjoint" in the sense that the "longer" chords from the two pairs have disjoint spans. Hence, the boundary path complementary to $P'$ cannot also have a nested pair of chords, or we would immediately have that $G$ contains the graph of Figure 8c. It follows that $e$ and $e'$ are opposing poles for $G$, because neither of the two boundary paths joining their endpoints has a pair of nested chords.

(c) → (a). Let $e$ and $e'$ be a pair of opposing poles for $G$ under some fixed outerplanar embedding. We show how to clear $G$ with three searchers in such a way that $e$ is cleared first and $e'$ last. Note that $G$ consists of the two opposing poles, two boundary paths joining the endpoints of the poles (with some edges in these boundary paths possibly having a single edge in parallel with them), and a collection of "cross-chords," each of which joins a vertex in one of the boundary paths to a vertex in the other boundary path. To clear $G$, we initially place two searchers on the ends of $e$ and use the third searcher to clear $e$ and any cross-chords that are in parallel with it, leaving the two searchers on the ends of $e$ as guards. These two searchers will be used subsequently only for clearing the boundary paths they are on, with the third searcher being used to clear all chords and parallel edges. The clearing process proceeds by repeatedly applying one of the following operations, as appropriate: If either of the two boundary searchers is not on an endpoint of $e'$ and has exactly one incident-contaminated edge (necessarily on the boundary), clear that edge by moving the searcher along it. Otherwise, if either of the two searchers is not on an endpoint of $e'$ and has exactly two incident-contaminated edges, one on the boundary and the other a parallel copy of it, place the third searcher on the same vertex as the first, clear the parallel edge with it, clear the boundary edge with the first searcher, and remove the third searcher from the graph. If neither of the above applies, but one of the boundary searchers is not yet on an endpoint of $e'$, then by outerplanarity we know that there must be a cross-chord joining the two vertices currently occupied by the two boundary searchers. In this case, we simply clear the cross-chord with the third searcher and continue. Finally, if both searchers are on the endpoints of $e'$, we can finish by clearing $e'$ and any parallel cross-chords using the third searcher. It is straightforward to verify that this procedure suffices to clear $G$ using three searchers, as required. □

To complete the characterization for $K = 3$, we must, in addition, specify how the 3-searchable biconnected components can be interconnected and where the components of lesser search number can be attached. We begin by giving several definitions that will be useful in describing the attached components.

Let $H$ be a reduced graph with $s(H) = 2$. By Theorem 5(c), if we remove from $H$ all loops and edges to leaves, the resulting graph must simply be a path, possibly with some edges occurring in parallel pairs. The only way to clear $H$ using two searchers is to first clear all off-path edges at one end of the path, then clear the path edges to the next vertex on the path, and so on, always clearing all off-path edges at a vertex on the path before moving along to the next vertex on the path, and ending by clearing all off-path edges at the other end of the path. Thus, we say that a vertex is an *endpoint* of $H$ if it is either one of the two ends of the path or a leaf of $H$ adjacent to one of the ends. Two endpoints are *opposite endpoints* if (1) they are the opposite ends of the path, (2) they are leaves adjacent to the opposite ends of the path, or (3) one is an end of the path and the other is a leaf adjacent to the opposite end. Any sequence for clearing the edges of $H$ using two searchers must begin by clearing some edge incident on an endpoint of $H$ and must end by clearing an edge incident on an opposite endpoint of $H$.

A graph is called a *pinched graph* if it is 2-searchable or can be obtained from a 2-searchable graph by coalescing two or more of its degree-1 vertices into a single vertex. If a pinched graph is 2-searchable, we can regard any of its vertices as its *coalesced node*; otherwise, the *coalesced node* of a pinched graph is its single coalesced vertex. Note that all pinched graphs are 3-searchable: If we first place

one searcher on the coalesced node, a 2-searcher search strategy for the original (unpinched) graph will suffice to clear all the edges.

Clearly, each biconnected component $B$ of a 3-searchable graph $G$ must itself be 3-searchable. Hence, by Lemma 7, the reduced graph $B'$ of $B$ must be outerplanar and bipolar. However, even if we restrict our attention to reduced graphs $G$, $B$ itself need not be reduced, since it might contain degree-2 vertices at which other biconnected components of $G$ are attached. For this reason, it is convenient to extend the definitions of boundary edges and opposing poles from the reduced graph $B'$ to $B$ itself. Let us say that an edge $e$ of $B$ *reduces* to an edge $e'$ of $B'$ if either $e$ and $e'$ are the same edge or $e$ belongs to a path of $B$ that becomes the edge $e'$ when $B$ is reduced to $B'$. We then say that an edge is a *boundary edge* of $B$ if it reduces to a boundary edge of $B'$, and we say that two boundary edges of $B$ are *opposing poles* for $B$ if they reduce to opposing poles for $B'$. Any two vertices $u$, $v$ that are endpoints of opposing poles $\{u, u'\}$ and $\{v, v'\}$ for $B$ are called *antipodal points* of $B$.

Let $B_1, B_2, \ldots, B_r$ be 3-searchable biconnected components of a connected 3-searchable graph $G$ that are "chained together" in the sense that there exists a sequence $a_0, a_1, \ldots, a_r$ of distinct vertices of $G$ such that, for $1 \leq j \leq r-1$, $a_j$ is an articulation point of $G$ belonging to both $B_j$ and $B_{j+1}$, and such that, for $0 \leq j \leq r-1$, $a_j$ and $a_{j+1}$ are antipodal points for $B_{j+1}$. In this case we say that $C = (a_0, B_1, a_1, B_2, a_2, \ldots, a_{r-1}, B_r, a_r)$ is a *chain* for $G$. A *valid set of opposing poles* for such a chain $C$ is any sequence of edges $\{a_0, x_1\}, \{a_1, y_1\}, \{a_1, x_2\}, \ldots, \{a_r, y_r\}$ such that, for $1 \leq j \leq r$, $\{a_{j-1}, x_j\}$, and $\{a_j, y_j\}$ are opposing poles for $B_j$. We write $V(C)$ to denote the union of all the vertices in components of the chain $C$ and $A(C)$ to denote the corresponding set $\{a_1, a_2, \ldots, a_{r-1}\}$ of articulation points. Given a valid set of opposing poles for $C$, we also use $N(C)$ to denote the corresponding set $\{x_1, y_1, x_2, y_2, \ldots, x_r, y_r\}$ of "neighboring" points.

For any component $B_j$ of the chain $C$ and its specified opposing poles, $\{a_{j-1}, x_j\}$ and $\{a_j, y_j\}$, consider either of the two boundary paths $P$ joining the endpoints of those poles. Let $H_P$ denote the subgraph of $B_j$ consisting of $P$ and all edges of $B_j$ that reduce to chords joining vertices of $P$ when $B_j$ is reduced. We call a vertex of $H_P$ *free* if it is an articulation point of $H_P$, or if it belongs to $A(C) \cup N(C)$. The collection of all free vertices of $C$ is denoted by $F(C)$. The free vertices will turn out to be the only vertices at which other components can be attached to the chain.

We say that a subgraph $H$ of a graph $G$ with chain $C$ *hangs from* the vertex $v$ in $V(C)$ if $v$ is the unique articulation point joining $H$ to the rest of $G$. In this case the subgraph $H$ is also said to be *hanging by* the vertex $v$ of $H$.

Our characterization for $K = 3$ is then the following:

THEOREM 6. *A reduced connected graph $G$ has $s(G) \leq 3$ if and only if $G$ consists of a chain $C = (a_0, B_1, a_1, B_2, \ldots, a_{r-1}, B_r, a_r)$, with valid set of opposing poles $\{a_0, x_1\}, \{a_1, y_1\}, \{a_1, x_2\}, \ldots, \{a_r, y_r\}$, along with components of the following forms hanging from the vertices in $V(C)$:*

(a) *an arbitrary number of edges and self-loops hanging from each vertex in $F(C)$;*
(b) *an arbitrary number of pinched graphs hanging by their coalesced nodes form each vertex in $A(C)$;*
(c) *for $1 \leq j \leq r$, at most one 2-searchable graph hanging by one of its endpoints from each of $x_j$ and $y_j$, or, if $x_j = y_j$, at most two such subgraphs hanging from the single vertex $x_j = y_j$.*

PROOF. We first show how to search a reduced graph $G$ of the above form using three searchers.

The biconnected components in the chain are cleared in order of occurrence, using essentially the same procedure described in the proof of Lemma 7 for each of them. As a matter of convenience, we use the term *trivial subgraph* to refer to self-loops and edges that end at leaves.

In general, before clearing component $B_i$, we shall have cleared all components $B_j$, $1 \leq j < i$, and all subgraphs not in the chain that hang from the vertices of such $B_j$, including any subgraphs hanging from the articulation point $a_{i-1}$. We shall also have a single searcher on $a_{i-1}$ to guard between the cleared and uncleared portions of $G$, with the other two searchers being free. It is easy to achieve this initially, simply by placing a searcher on $a_0$, using one of the remaining two searchers to clear any edges and self-loops that hang from $a_0$ (there can be no pinched graphs by hypothesis). To clear $B_i$ and its corresponding hanging subgraphs, we proceed as follows: If there is a nontrivial 2-searchable subgraph hanging from $x_i$, we first clear that subgraph using the two free searchers, in such a way as to end at $x_i$, leaving a single searcher on $x_i$ as a guard. We then clear any trivial subgraphs hanging from $x_i$ using the remaining searcher and clear the edge $\{a_{i-1}, x_i\}$, also with that searcher. At this point, we proceed to clear the remaining edges of $B_i$, as in the proof of Lemma 7, so as to end with the edge $\{a_i, y_i\}$, clearing trivial subgraphs hanging from vertices in $F(C)$ along the way using the roving searcher. It is easy to check that $F(C)$ has been defined in just such a way that this can be done, that is, that for each such vertex $v$ there is always some time at which $v$ has a searcher on it and the roving searcher is free for clearing the trivial subgraphs hanging from $v$. We end with all of $B_i$ being clear, and a searcher on each of $a_i$ and $y_i$, with the third searcher being free at this point. If there are any trivial subgraphs hanging from $y_i$, we clear them now with the third searcher. We then use that searcher and the searcher on $y_i$ to clear any 2-searchable subgraph hanging from $y_i$, starting from the end that $y_i$ is on. Finally, with one searcher remaining on $a_i$ as a guard, we use the two remaining searchers to clear each of the pinched graphs hanging from $a_i$. This leaves us in the position required for starting to clear $B_{i+1}$, as described earlier, with the entire graph $G$ having been cleared if $i = r$.

It is not difficult to see that this strategy will successfully clear $G$ using three searchers, as claimed, and hence any graph of the stated form must be 3-searchable.

Now we show that any reduced 3-searchable graph must have the specified form. Let $G$ be any reduced 3-searchable graph that is not 2-searchable (a 2-searchable graph is trivially of the required form). Using LaPaugh's result that recontamination is never needed, we know that there exists an edge sequence, with no repetitions, for clearing $G$ using three searchers. Fix a choice of such an edge sequence. For purposes of uniformity, it is convenient to modify $G$ and the selected edge-clearing sequence as follows: Insert a new degree-2 vertex in the middle of the first edge cleared, add three edges joining that new vertex to a second new vertex, and place those three new edges at the head of the clearing sequence, immediately followed by the two edges resulting from subdividing the old first-cleared edge. Make the analogous change with the last edge cleared, using another two new vertices, this time placing the three new edges at the tail of the edge-clearing sequence, immediately preceded by the two edges resulting from subdividing the old last-cleared edge. It is easy to see that the resulting edge-clearing sequence will still suffice for clearing the new graph with three searchers, and that the new graph will have the desired form only if the original did. Thus, there will be no loss of generality in regarding the new graph as the graph $G$ under consideration, with its given edge-clearing sequence as obtained above.

Let $e_1$ and $e_m$ denote the first and last edges cleared according to the current edge-clearing sequence. Let $B_1$ and $B_r$ denote the biconnected components of $G$ containing $e_1$ and $e_m$, respectively, and let $B_1, B_2, \ldots, B_r$ denote the unique "path" of biconnected components in $G$ joining $B_1$ to $B_r$, that is, such that every simple path from $B_1$ to $B_r$ consists of a (possibly empty) path in $B_1$, followed by a nonempty path in $B_2$, followed by a nonempty path in $B_3, \ldots$, followed by a nonempty path in $B_{r-1}$, followed by a (possibly empty) path in $B_r$. The existence and uniqueness of this sequence of biconnected components follow from the way in which biconnected components partition a graph. For $1 \leq i \leq r-1$, let $a_i$ denote the unique articulation point of $G$ common to both $B_i$ and $B_{i+1}$, and let $a_0$ and $a_r$ be the remaining vertices of $B_1$ and $B_r$ (each of these components has only two vertices, by construction). One of our goals is to show that $\{a_0, B_1, a_1, B_2, \ldots, B_r, a_r\}$ is a chain.

A key fact that we use frequently in the remainder of the proof is that, in order to prevent recontamination of $e_1$, every path from $e_1$ to $e_m$ must contain at least one searcher throughout the process of clearing $G$.

The first implication of this pertains to the structure of the subgraph $H$ hanging from any articulation point $a_i$. Since every simple path from $e_1$ to $e_m$ must contain a searcher and since $a_i$ itself is the only vertex of $H$ included in any such paths, it follows that $H$ must be cleared using only two searchers, possibly along with a third searcher fixed as a guard on $a_i$. Thus the graph obtained from $H$ by splitting the vertex $a_i$ into distinct new leaves, one for each edge of $H$ incident on $a_i$, must be 2-searchable, although it may include several separate connected components. By the definition of pinched graph, this simply says that the original subgraph $H$ consists of a collection of pinched graphs, all having coalesced node $a_i$, as claimed in the theorem statement. Hence we may ignore these edges for the remainder of the proof.

For $1 \leq i \leq r$, let $B_i^*$ denote the subgraph of $G$ consisting of $B_i$ and all subgraphs hanging from vertices of $B_i$ other than $a_{i-1}$ and $a_i$. We next show that we can normalize the given edge-clearing sequence so that, for $1 \leq i \leq r$, all edges of $B_i^*$ are cleared before any edges of $B_{i+1}^*$. By construction, this already holds for $i = 1$. Given that it holds for $1, 2, \ldots, i-1$, we show how to modify the edge clearing sequence so that it also holds for $i$. Let $e$ be the first edge in the clearing sequence that belongs to some $B_j^*$ for $j > i$, and let $e'$ be the last edge of $B_i^*$ in the sequence. If $e$ follows $e'$ in the sequence, we are done. Otherwise, note that from the time that $e$ is cleared until the time at which $e'$ is cleared, there must be at least one searcher on some vertex (other than $a_i$) of $B_i$, to guard between the contaminated portion of $B_i$ and the cleared edges of $B_{i-1}$, and there must be at least one searcher on some vertex (other than $a_i$) of some $B_k$ for $k \geq j$, to guard between $e_m$ and the cleared portion of $B_j$. Thus all edges of $B_i^*$ cleared during the interval are cleared using at most two searchers on $B_i^*$, and all edges of components $B_k^*$, for $k > i$, cleared during that interval are cleared using at most two searchers on that part of $G$. It follows immediately that we can modify our edge-clearing sequence, without requiring more than three searchers overall, by moving all edges of $B_i^*$ just before $e$ in the sequence, but keeping them in the same order relative to one another. The fact that three searchers still suffice can be easily seen using the observation that we could leave a guard on the vertex $a_i$ throughout the clearing of $B_i^*$ and throughout the clearing of the other edges that were originally cleared before $e'$, using only the two remaining searchers to clear $B_i^*$ and those other edges. Hence repeated application of this transformation will result in an edge-clearing sequence

for $G$ of the desired form, and there will be no loss of generality in assuming that the sequence we have is of this form.

We can now focus on the edges in each $B_i^*$ individually, since we know that they occur consecutively in the edge-clearing sequence, and we need only show that each of these "extended" components has the form specified in the theorem. We already know that $B_1^*$ and $B_r^*$ have this form, so consider any $B_i^*$ with $1 < i < r$.

By Lemma 7, we know that the reduced graph of $B_i$ must be outerplanar and bipolar. We also know that the clearing of $B_i^*$ must begin with a searcher on the vertex $a_{i-1}$ in order to guard between the cleared edges of $B_{i-1}$ and the contaminated edges of $B_i$. If $B_i^*$ consists of a single edge, or a collection of parallel edges, then it trivially has the required form. So suppose $B_i^*$ has at least three vertices and three edges (since $G$ is reduced, it cannot consists of three vertices and only two edges).

We begin by showing the existence in $B_i^*$ of a neighbor $v$ of $a_{i-1}$ and a (possibly empty) 2-searchable subgraph $H$, hanging by one of its endpoints from $v$, such that the clearing sequence for $B_i^*$ can be transformed to begin by first clearing all the edges of $H$ followed immediately by the edge $\{a_{i-1}, v\}$.

This is easy to see if the first edge of $B_i^*$ that is cleared also belongs to $B_i$, for then we claim that this edge must already join $a_{i-1}$ to some neighboring vertex $v$ (and hence that the subgraph $H$ is empty). If this first cleared edge does *not* have $a_{i-1}$ as an endpoint, then, immediately after the edge is cleared, the three searchers would be on the two ends of that edge and on the vertex $a_{i-1}$. But each of these three vertices must still have two or more incident contaminated edges (since $G$ is reduced), so none of those searchers would be able to move, contradicting our assumption that we have an edge sequence that suffices for clearing $G$ with three searchers.

On the other hand, if there are edges of $B_i^*$ that are cleared before the first edge of $B_i$, then they must all belong to subgraphs hanging from some single vertex $v$ of $B_i$; for, otherwise, the two searchers not on $a_{i-1}$ would be needed as guards on their respective subgraphs until after the first edge of $B_i$ is cleared, preventing us from ever clearing a first edge of $B_i$. Let $H$ be the union of all the subgraphs hanging from $v$ that contain those cleared edges (there may be other subgraphs hanging from $v$ with no edges that are cleared before the first edge of $B_i$). Then an argument like that used above for transforming the edge-clearing sequence to separate the components $B_j^*$ from one another can be used again to transform the sequence for $B_i^*$ so that all edges of $H$ are cleared before the first edge of $B_i$ is cleared. Thus, in our new edge-clearing sequence, an edge of $B_i^*$ is cleared before the first edge of $B_i$ if and only if it belongs to $H$. Moreover, since the subgraph $H$ is cleared using only two searchers, ending with a guard on $v$, $H$ must be a 2-searchable graph with $v$ as an endpoint.

We want to claim that $a_{i-1}$ and $v$ must be adjacent. Suppose not. Then just before the first edge of $B_i$ is cleared we must have a searcher on $a_{i-1}$ and a searcher on $v$, with every vertex of $B_i$ other than these two having at least three incident contaminated edges (because $G$ is reduced). Since $B_i$ is biconnected and we have only one additional searcher, the first edge of $B_i$ that is cleared must be incident on either $a_{i-1}$ or $v$. Suppose this edge is incident on $v$ and that its other endpoint is $u \neq a_{i-1}$. (The case in which it is incident on $a_{i-1}$ instead is handled identically.) Just after $\{u, v\}$ has been cleared, $u$ must still have two incident uncleared edges, so the searcher on it cannot move. The vertex $a_{i-1}$ also has two incident contaminated edges, so the only searcher that might still be able to move is the one on $v$. If at this point $v$ has more than a single incident contaminated edge, none of our

searchers can move, contradicting the assumption that this sequence clears $G$. If $v$ has only a single incident contaminated edge, the searcher on it must next move to clear that edge, ending at a vertex $u'$ differing from both $a_{i-1}$ and $u$. At this point, all three searchers must be on vertices that each have at least two incident contaminated edges, so none of them can move, again giving us a contradiction. Therefore, it must be the case that $v$ is adajcent to $a_{i-1}$. Moreover, since there must be guards on both $a_{i-1}$ and $v$ until the edge $\{a_{i-1}, v\}$ is cleared, we can further transform our clearing sequence for $B_i^*$ so that the edge $\{a_{i-1}, v\}$ is cleared first.

This gives us what we wanted for the initial portion of the clearing sequence for $B_i^*$. A symmetric argument can then be used to show the existence of a neighbor $v'$ of $a_i$ and a (possibly empty) 2-searchable subgraph $H'$ hanging by one of its endpoints from $v'$, such that the clearing sequence for $B_i^*$ can be further transformed to have the property that it *ends* by clearing the single edge $\{a_i, v'\}$ followed immediately by all the edges in $H'$. We explicitly allow $v = v'$ here.

Let $e$ denote the edge $\{a_{i-1}, v\}$ and let $e'$ denote the edge $\{a_i, v'\}$. Suppose that $e$ and $e'$ were not opposing poles for $B_i$. Then one of the two boundary paths of $B_i$ joining $e$ and $e'$ must have a pair of nested chords (or paths that reduce to nested chords when $B_i$ is reduced). From the time that $e$ has been cleared until the time that $e'$ is cleared, there must be at least one searcher on the opposite boundary path to guard between $e$ and $e'$. Thus the boundary path with the pair of nested chords must be cleared entirely using only two searchers, which is impossible because this subgraph of $B_i$ contains the forbidden subgraph of Figure 5c. Therefore, $e$ and $e'$ most be opposing poles for $B_i$, and $a_{i-1}$ and $a_i$ are antipodal points of $B_i$, as required.

Note that we now have all that is required of each $B_i$ in order for $C = (a_0, B_1, a_1, B_2, \ldots, B_r, a_r)$ to be a chain and that a valid set of opposing poles for $C$ consists of the corresponding collection of poles $e, e'$ for the $B_i$.

To complete the proof, we need only show that all edges of $B_i^*$ cleared between the times that $e$ and $e'$ are cleared must either be edges of $B_i$ or trivial subgraphs hanging from vertices in $F(C)$. Clearly, any such edges not in $B_i$ can only be trivial subgraphs, since there have to be two searchers on $B_i$ itself throughout this interval, leaving at most one searcher available for use in clearing those edges. Suppose a trivial subgraph hangs from some vertex $u$ not in $F(C)$. By the definition of $F(C)$, there must exist a pair of vertex disjoint paths joining the opposing poles of $B_i$, neither of which passes through $u$, and each of those paths must contain at least one searcher as a guard throughout the interval during which $B_i$ is being cleared. Thus all edges incident on $u$ must be cleared using only a single searcher, possibly with guards on adjacent vertices. However, since $G$ is reduced, $u$ must have degree at least 3, so the subgraph to be cleared by this single searcher contains a three-edge star, which cannot possibly be cleared by a single searcher. Therefore, trivial subgraphs can hang only from vertices in $F(C)$, and we have shown that $G$ must be of the form given in the theorem statement. □

Theorem 6 also leads naturally to an efficient algorithm for recognizing 3-searchable graphs.

THEOREM 7. *There is a linear-time algorithm for recognizing graphs with search number* 3.

PROOF. An outline of such an algorithm goes as follows: Without loss of generality, we can restrict attention to graphs $G$ that are connected and reduced

(since a graph can be reduced in linear time). We begin by identifying the biconnected components of the given graph $G$ in linear time using the method of [14]. As is well known, the biconnected components of any graph are joined together in a tree structure, with all simple paths joining any fixed pair of biconnected components passing through a unique sequence of the other biconnected components.

By working in from the "leaves" of this tree, we can identify the subgraphs of $G$ that can hang off a chain like that given by Theorem 6. The first of these are the "trivial" subgraphs, individual edges and loops joined to the rest of $G$ at a single articulation point, which we call 1-*pieces*. Next are the 2-searchable subgraphs that can hang from vertices in $V(C)$ and their neighboring vertices in the chain components. Temporarily ignoring any hanging 1-pieces, we note that each of these is just a path composed of single and double edges, attached to the rest of the graph at only a single articulation point that is an endpoint of the path. The maximal such paths, along with their hanging 1-pieces, are called 2-*pieces*. Finally, we have the subgraphs that correspond to non-2-searchable pinched graphs. Each of these consists of a single biconnected component (having at least two edges), along with any hanging 1- and 2-pieces, such that the subgraph is joined to the rest of $G$ at only a single articulation point and such that the subgraph becomes 2-searchable when that articulation point is split into distinct degree-1 vertices for each of its incident edges. We call these 2′-*pieces*. It is not hard to see that the 1-, 2-, and 2′-pieces of $G$ can be identified in linear time.

If all biconnected components of $G$ belong to 1-, 2-, and 2′-pieces, we immediately have from Theorem 6 that $s(G) \leq 3$. Otherwise, call any remaining biconnected components 3-*pieces*. From Theorem 6, we know that each 3-piece must be outerplanar (notice that this is stronger than just saying its reduced graph is outerplanar). It is straightforward to check this property for all the 3-pieces in linear time. Moreover, it is easy to see that the outerplanar embedding of each 3-piece is essentially unique, with the only choices being the relative placement of parallel edges. Thus we can also choose a fixed outerplanar embedding of each 3-piece.

If the 3-pieces of $G$ do not all belong to a single path $P$ in the biconnected component tree, we are done, since $s(G)$ must then exceed 3. Otherwise, it remains for us to verify that they form a chain. Since they are joined together in a path, we only need to determine whether each 3-piece, along with its associated hanging 1-, 2-, and 2′-pieces, has the form required by Theorem 6.

For any 3-piece that is not one of the ends of $P$, its two articulation points in the chain, which must be antipodal points of that component, are determined by where its two neighboring 3-pieces are joined. This immediately allows us to use Theorem 6 and its associated definitions, along with the identified 1-, 2-, and 2′-pieces, to verify that it has the required form. Essentially all we need to do is to try to search it (plus any hanging subgraphs), beginning at one articulation point and ending at the other, following the algorithm given in the proofs of Lemma 7 and Theorem 6. (Given the articulation points, there are at most four possible combinations of opposing poles. It suffices to try each combination separately, although with a little care one can test for the desired form in about the time it takes to try out just one such combination.)

For the 3-pieces on the ends of $P$, if there are two of them, their articulation points interior to the chain are determined by where the rest of the chain is connected. Thus we can check whether each is of the required form by essentially the same method, that is, by attempting to search it (and its hanging subgraphs),

beginning at the corresponding articulation point and following the algorithm from the proofs of Lemma 7 and Theorem 6. Our knowledge of the 1-, 2-, and 2′-pieces provides obvious constraints and makes this easy to do.

Finally, we have the case in which $P$ consists of only a single 3-piece. If that 3-piece has a hanging 2- or 2′-piece, then we know that the vertex from which the 2- or 2′-piece hangs, or one of its two neighbors on the boundary of the 3-piece, must be part of a pair of antipodal points and hence be suitable for starting the search of the 3-piece and its hanging subgraphs. Thus, we can try each of these three possibilities, proceeding exactly as above. If the 3-piece has no hanging 2- or 2′-pieces, we simply choose some vertex on its boundary and proceed around the boundary in one direction until the first time that the path traversed has a nested pair of chords. In order to break that nested pair, some edge spanned by the longer of the pair must belong to a pair of opposing poles. In fact, it is not hard to see that either the first or the last edge spanned by that longer chord must belong to a pair of opposing poles, since the other edges it spans can only break a subset of the nested chords broken by one of these. Therefore, we need only consider, at most, four possibilities, the four endpoints of those two edges, for where a search of the 3-piece might start, and we can check each of them as described above. The original graph $G$ is 3-searchable if and only if we succeed in searching each of the 3-pieces, along with its associated hanging subgraphs, in this way. □

## 5. *Further Results and Open Problems*

Since the results in this paper were originally announced, additional progress on the problem has been made. As mentioned above, LaPaugh [6] has proved that "recontamination" does not help, and hence the problem of determining the search number is in NP. Makedon et al. [9] have shown that the problem remains NP-complete for graphs with a maximum vertex degree of 3, using a clever modification of our proof in which the large cliques are replaced by structures resembling brick walls. (Still open, however, is the complexity of determining $s(G)$ when $G$ is a planar graph.)

Attention has also been drawn to the relation between search number and other measures of graph complexity. The maximum degree-3 case holds particular interest because in this case the search number of $G$ is known to be identical to the cutwidth of $G$. The *cutwidth* of a graph $G$ is the minimum, over all orderings of the vertex set $V(G)$ of $G$, of the maximum for $1 < i \leq |V(G)|$ of the number of edges whose left endpoint is to the left of the $i$th vertex in the ordering and whose right endpoint is equal to or to the right of the $i$th vertex. The cutwidth problem, like the search number problem, is known to be NP-complete for general graphs ([3] and L. Stockmeyer, private communication, 1974 (See [2])), and Yannakakis [15] has recently shown that it can be solved in time $O(n \log n)$ for trees. The tree algorithm has a recursive structure similar to our search number algorithm for trees, although the two algorithms maintain rather different information about the subtrees occurring as subproblems. It is not difficult to see (as first observed to us by I. H. Sudborough) that the search number of a graph cannot exceed its cutwidth. Any cutwidth ordering (of the vertices) induces a search number ordering (of the edges) by increasing order of the left endpoints, breaking ties by right endpoints, and this search order requires a number of searchers that are, at most, equal to the cutwidth. The equality of cutwidth and search number is shown for degree-3 trees in [1] and for general degree-3 graphs in [8] and [9]. For arbitrary graphs, the

cutwidth and search number need not be the same, since the "star" $S_n$ on $n$ vertices has search number 2 and cutwidth $\lfloor n/2 \rfloor$.

For the relation between search number and other measures, such as "topological bandwidth," "pebble demand," and "interval thickness," see [5], [8], and [9].

REFERENCES


1. CHUNG, M., MAKEDON, F., SUDBOROUGH, I. H., AND TURNER, J. Polynomial time algorithms for the min cut problem on degree restricted trees. *SIAM J. Comput. 14* (1985), 158–177.
2. GAREY, M. R., AND JOHNSON, D. S. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, 1979.
3. GAVRIL, F. Some NP-complete problems on graphs. In *Proceedings of the 11th Conference on Information Sciences and Systems.* Johns Hopkins Univ., Baltimore, Md., 1977, pp. 91–95.
4. HARARY, F. *Graph Theory.* Addison-Wesley, Reading, Mass., 1969, pp. 106–107.
5. KIROUSIS, L. M., AND PAPADIMITRIOU, C. H. Searching and pebbling. *Theoret. Comput. Sci. 47* (1986), 205–218.
6. LAPAUGH, A. S. Recontamination does not help. Manuscript, 1982.
7. LENGAUER, T., AND TARJAN, R. E. Asymptotically tight bounds on time–space trade-offs in a pebble game. *J. ACM 29*, 4 (Oct. 1982), 1087–1130.
8. MAKEDON, F. S. Layout problems and their complexity. Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Northwestern Univ., Evanston, Ill., 1982.
9. MAKEDON, F. S., PAPADIMITRIOU, C. H., AND SUDBOROUGH, I. H. Topological bandwidth. *SIAM J. Algebraic Discrete Meth. 6* (1985), 418–444.
10. MEGIDDO, N., HAKIMI, S. L., GAREY, M. R., JOHNSON, D. S., AND PAPADIMITRIOU, C. H. The complexity of searching a graph (preliminary version). In *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science.* IEEE, New York, 1981, pp. 376–385.
11. PARSONS, T. D. Pursuit-evasion in a graph. In *Theory and Applications of Graphs*, Y. Alavi and D. Lick, Eds. Springer-Verlag, Berlin, 1976, pp. 426–441.
12. PARSONS, T. D. The search number of a connected graph. In *Proceedings of the 9th Southeastern Conference on Combinatorics, Graph Theory, and Computing.* Utilitas Mathematica, Winnipeg, Canada, 1978, pp. 549–554.
13. PIPPINGER, N. Pebbling. Rep. No. RC8275. IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., 1980.
14. TARJAN, R. E. Depth-first search and linear graph algorithms. *SIAM J. Comput. 1* (1972), 146–160.
15. YANNAKAKIS, M. A polynomial algorithm for the min-cut linear arrangement of trees. *J. ACM 32*, 4 (Oct. 1985), 950–988.


RECEIVED SEPTEMBER 1983; REVISED SEPTEMBER 1985, MAY 1987; ACCEPTED MAY 1987