Theoretical Computer Science 47 (1986) 205-218
North-Holland

SEARCHING AND PEBBLING

Lefteris M. KIROUSIS

Department of Mathematics, University of Patras, Patras, Greece, and Department of Computer Science, National Technical University, Athens, Greece

Christos H. PAPADIMITRIOU

Department of Computer Science, National Technical University, Athens, Greece, and Department of Computer Science, Stanford University, Stanford, CA 94305, U.S.A.

Communicated by R.M. Karp
Received November 1983
Revised December 1984

Abstract. We relate the search number of an undirected graph G with the minimum and maximum of the progressive pebble demands of the directed acyclic graphs obtained by orienting G. Towards this end, we introduce node-searching, a slight variant of searching, in which an edge is cleared by placing searchers on both of its endpoints. We also show that the minimum number of searchers necessary to node-search a graph equals its vertex separator plus one.

Key words. Searching a graph, pebble games, layout parameters of a graph, vertex separator of a graph.

1. Introduction

This paper is about the connection between two kinds of games played on graphs; pebbling and searching. The first group, the *pebble games*, model sequential computation. Pebble games have been extensively studied in the past (for an overview see [13], and see [2, 15] for some important applications). *Graph-searching* games, on the other hand, model pursuit and evasion in graphs and have been studied rather recently [9].

Pebble games are played on *directed acyclic graphs* (*dags*). At each move of the basic (*black*) pebble game, we either place a pebble on a vertex with no pebble, or delete one from a pebbled vertex. The game starts with all vertices of the graph pebble-free, and ends when the same situation is attained after all vertices have been pebbled at least once. The rules we follow for the black pebble game are:

(i) A pebble can be placed on a vertex only if all the immediate predecessors of that vertex are pebbled. (Thus, nodes with zero in-degree can be pebbled at any time.)

(ii) A pebble can be deleted at any time.

In another version, called the *black and white pebble game*, we allow a second kind of pebbles called *white* pebbles. Such pebbles can be placed at any point on any vertex, but they can be removed only after they *turn black*. A white pebble turns

0304-3975/86/$3.50 © 1986, Elsevier Science Publishers B.V. (North-Holland)

black at the moment when all the immediate predecessors of the vertex on which it lies are pebbled. In this version, a vertex is considered pebbled when it carries a pebble, whatever its colour. (Notice here the 'duality' between the roles of the white and black pebbles.) White pebbles model 'nondeterministic' moves in which the value of a result is conjectured, with checking postponed for some time in the future (when the pebble turns black).

In both versions, the complexity we measure is the *pebble demand* of the graph, i.e., the maximum number of pebbles needed to be simultaneously on the graph to carry out the game. For a dag G, the black (respectively black and white) pebble demand is denoted by $b(G)$ (respectively $bw(G)$).

It is known [1] that computing the black pebble demand of a graph is PSPACE-complete, although this has not been shown for the black and white version. As for relations between the two versions, it is known that the use of white pebbles may reduce the pebble demand by at most a square root factor [10]. Nevertheless, only examples of graphs G for which $bw(G) \leq \frac{1}{2}b(G)$ have been constructed.

A *progressive* version is defined for both deterministic and nondeterministic (i.e., black and white) pebble games. In these versions, each vertex can be pebbled only once. The progressive version of the black pebble game had in fact been proposed before the general game, as a model of register allocation for the computation of arithmetic expressions, where recomputation is not deemed realistic [15]. The progressive black, and black-white pebble demands for a graph G are denoted by $pb(G)$, and $pbw(G)$, respectively. It is known that the restriction of progressiveness may drastically increase the pebble demand. For a study of the corresponding time-space tradeoffs see [6]. The problems of determining the progressive pebble demands of a dag are NP-complete [5, 15].

The second kind of game we consider, the *search game*, was first studied by Parsons [12]. In the original version, which, in this paper, we call *edge-searching*, an undirected graph is considered as a system of *tunnels* in which a swift and cunning fugitive is hidden. The *search-number* of a graph G, denoted in this paper by $es(G)$, is the minimum number of 'searchers' that guarantee the capture of the fugitive. It was shown in [3] that there is always an optimal search strategy (i.e., one that uses $es(G)$ searchers) in which no tunnel is 'searched' twice. In other words, *recontamination* does not help in searching a graph. (Notice here the contrast with the pebble games, in which 'recomputation' is indeed known to help.) This implies that the problem of determining whether the search number of a given graph is less than or equal to a given number is in NP. It was shown in [9] that it is NP-complete, whereas it can be efficiently solved for trees.

Recently, connections between the search number of a graph and other parameters related to the *layout* of the graph on the one-dimensional grid, such as the *cutwidth* and the *topological bandwidth*, were discovered (see [7, 8]). Nevertheless, despite the apparent similarities between the searching and pebble games, no connection between them was known. The reason was, evidently, that the pebble games appeared to be inherently *directed* graph games, whereas the searching games were played

on undirected graphs. Attempts to do searching on directed graphs did not yield interesting problems.

In this paper we relate searching and pebbling. We consider the smallest progressive pebble demand among all *acyclic orientations* (*directives*) of an undirected graph G. We denote this parameter by $\mathrm{mpb}(G)$ for the black pebble game and by $\mathrm{mpbw}(G)$ for the black and white pebble game. We also define a new version of a search game, the *node-search* game. This game is identical to the ordinary search game, except that the fugitive is considered 'captured' if both ends of the edge at which it hides are guarded by searchers (real-life interpretations of this rule are possible!).

Evidently, the two versions of searching cannot differ much. In Section 2, we establish a close connection between node-searching and edge-searching, both in terms of the size of the corresponding numbers, and in terms of the complexity of computing them (more precisely: of deriving a search strategy). We show that an optimal node-searching strategy can be constructed if an optimal edge-searching strategy is given, and vice versa. It follows that the theorem of LaPaugh stating that recontamination does not help in edge-searching carries over to node-searching, and also that the problem of computing the node-search number is NP-complete.

In Section 3, we prove that the node-search number of an undirected graph G, $\mathrm{ns}(G)$, is equal to both $\mathrm{mpb}(G)$ and $\mathrm{mpbw}(G)$. Also, we show that if an undirected graph G is given an acyclic orientation with in-degree at most k, then the progressive black and white pebble demand of the resulting graph is at most $(k+1)\mathrm{ns}(G)$. Notice how the gap between directed and undirected graphs is bridged by minimizing over all directives of an undirected graph. In fact, this minimization equates the two versions of the pebble game known to differ in general.

In Section 4, we study the *vertex separator* of G (see [5]). Consider a permutation of the nodes of G. For each vertex v, other than the last in the permutation, there is a set of edges joining the vertices of index smaller than or equal to the index of v to those of greater index. Consider the cardinality of the left endpoints of this set of edges. The maximum such cardinality over all vertices v as above is by definition the vertex separator over the given permutation. The minimum of this parameter over all permutations is by definition the vertex separator of G, denoted by $\mathrm{vs}(G)$. For relations between this and other layout parameters and searching games see [7, 8]. For example, it had been known that $\mathrm{vs}(G) \leq \mathrm{es}(G) \leq \mathrm{vs}(G)+2$ [16].

We improve this last inequality to the rather unexpected equality $\mathrm{ns}(G) = \mathrm{vs}(G)+1$, thus establishing that two graph parameters defined in very different settings are in fact equal.

2. Two versions of searching

We start by briefly repeating some definitions from [9]. Let $G=(V, E)$ be an undirected graph. A move of a *strategy* S that *clears* (or *searches*) the graph belongs

to one of the following types: (a) *sliding* a searcher along an edge, (b) placing a searcher on a vertex as a *guard*, and (c) deleting a searcher from a vertex.

Initially, all edges are contaminated by a gas (or capable of harbouring a swift fugitive). An edge with at least one guarded endpoint is cleared by sliding along it a searcher, and then placing this searcher on the unguarded endpoint, if there is such a one. If all other edges incident on the first guarded vertex are clear, the guard itself may be used for the sliding. A clear edge remains clear as long as there is no unguarded path (i.e., a path with no searcher on its vertices) leading from it to a contaminated edge. If such a path ever appears due to deletion or sliding of searchers, the edge is said to be recontaminated. The search is complete once there are no contaminated edges. Also, a vertex is said to be clear if it is guarded, or if all edges incident on it are clear. Otherwise, it is contaminated.

The maximum number of searchers simultaneously appearing on the graph at any point of the search strategy S is called the *edge-search number* of S, $\mathrm{es}(G, S)$. An *optimal strategy* is one that makes this parameter take its least possible value. This least value is the *edge-search number* of the graph, denoted by $\mathrm{es}(G)$.

We also define a *progressive* version of edge-searching, i.e., a version in which recontamination is not allowed. The corresponding optimal number of searchers is denoted by $\mathrm{pes}(G)$. The following is a deep result due to Andrea LaPaugh.

Theorem 2.1 (LaPaugh [3]). *For all undirected graphs* G, $\mathrm{pes}(G) = \mathrm{es}(G)$, *that is, recontamination does not help in edge-searching.*

As an immediate corollary to the above theorem we get the following.

Corollary 2.2. *For any graph* G *there is always an optimal edge-searching strategy satisfying the following two conditions*:

(i) *no vertex, nor any edge, is ever visited twice by a searcher*;

(ii) *once all edges incident on a vertex are cleared, any searcher on that vertex is immediately deleted.*

In the sequel we shall consider only strategies of the above type.

We now define another version of searching a graph, called *node-searching*. In node-searching, we do not slide searchers along the edges, but only place them as guards on the vertices. An edge is cleared once both of its endpoints are simultaneously guarded. The remaining rules and notions of node-searching are defined in the same way as in edge-searching. In particular, the least number of searchers required to node-search all edges of the graph G is called the *node-search number of* G, $\mathrm{ns}(G)$, and its progressive version is denoted by $\mathrm{pns}(G)$.

Using at most one more searcher to traverse any edge that is guarded, we can convert any node-searching strategy to an edge-searching one. Also, any edge-searching strategy can be transformed into a node-searching one, using temporarily an extra searcher for the moves where a guard is slid along the last contaminated edge incident on it (this is the only case that edge-searching can save searchers).

G

Fig. 1. $es(G) = 1, ns(G) = 2$.

We have therefore shown that for any graph G, $ns(G) - 1 \leq es(G) \leq ns(G) + 1$. We can easily construct examples showing that all three cases are possible (see Figs. 1, 2 and 3).

It is interesting to note that the rules for node-searching a graph are almost the same as the rules of the *breadth-first pebble game* defined in [14]. In that game though, a different complexity measure is considered (see also [8]).

We shall prove now that recontamination does not help even in node-searching. The idea of the proof is to define an optimal node-searching strategy of G by simulating an optimal edge-searching of a graph G_e obtained by suitably modifying G. The simulation is faithful enough so that the no-recontamination restriction carries over from G_e to G. The result then follows from Theorem 2.1.

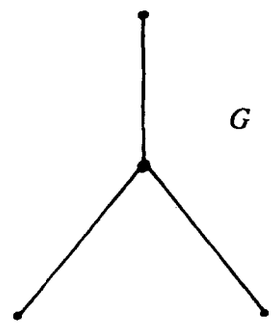
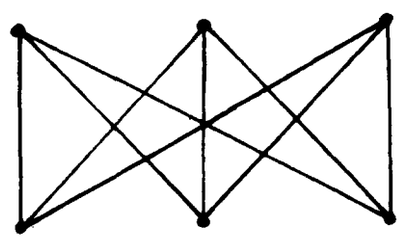


Fig. 2. $es(G) = ns(G) = 2$.

Theorem 2.3. *For all undirected graphs G, $pns(G) = ns(G)$, i.e., recontamination does not help in node-searching.*

Proof. We construct a graph G_e from G by replacing every edge of G by three parallel edges connecting the same pair of vertices (Fig. 4).

It is easy to prove that the edge-search number of a graph does not change if we insert a new vertex of degree two in the middle of an edge. Therefore, by inserting some new degree-two vertices on G_e, we can eliminate multiple edges; nevertheless, for reasons of brevity, we employ multiple edges.



G

Fig. 3. $es(G) = 5, ns(G) = 4$.

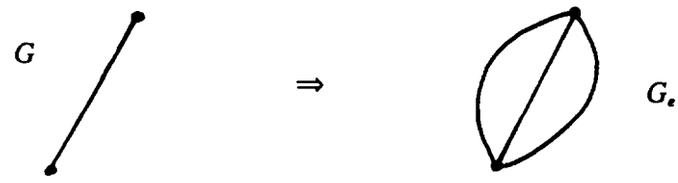


Fig. 4. Obtaining G_e from G.

We shall prove that $\mathrm{ns}(G) = \mathrm{es}(G_e) - 1$.

One direction, namely that $\mathrm{es}(G_e) - 1 \leq \mathrm{ns}(G)$, is almost obvious since by one extra searcher we can carry out all the necessary edge-searching of G_e, simulating a node-searching of G. For the other direction, we carry out a node-searching of G following step by step an optimal edge-searching strategy of G_e of the form described in Corollary 2.2. Observe that in any such strategy there can be no move at which a searcher already lying on a vertex is slid towards a contaminated vertex. This is so, because any two adjacent vertices are joined with more than one (actually three) edges. Therefore, if at the end of a move there appears a searcher on a previously contaminated vertex, this searcher is new on the graph.

The rules for the simulation are the following:

(i) when a searcher is placed on a contaminated vertex of G_e, probably after having slid it along an edge incident on this vertex, a searcher is placed on the corresponding vertex of G;

(ii) when a searcher is removed from a vertex of G_e, probably to be slid along an edge incident on this vertex, the searcher from the corresponding vertex of G is deleted. Notice that if sliding takes place, the second endpoint should be guarded,so, after the sliding, the searcher is deleted from the graph G_e.

It is not hard to see that the above simulation defines a node-searching of G. This is so, because during the clearing of a triple edge in G_e, there is always a moment when both of its endpoints are guarded.

Also, it is immediate that the number of searchers on G is at most $\mathrm{es}(G_e)$.

To prove that this bound can be reduced by one, it is sufficient to show that any move that raises the total number of searchers on G_e to $\mathrm{es}(G_e)$ must be a sliding move along an edge whose both endpoints are guarded. Indeed, after such a move, we do nothing on G.

Consider, towards a contradiction, that the number of searchers on G_e is raised to $\mathrm{es}(G_e)$ when we place a new searcher as in (i). The next move must be a deleting one, and, by Corollary 2.2, we must have completed at this point the clearing of all edges incident on a vertex. But this is impossible since by such a move we can at best clear the first contaminated edge of a triple edge. This completes the proof .□

From Theorem 2.1 we obtain the following corollary.

Corollary 2.4. *There is always an optimal node-searching strategy in which no vertex is visited twice by a searcher, and in which every searcher is deleted immediately after all the edges incident on it have been cleared (ties are broken arbitrarily).*

In the sequel, we shall always consider strategies as in the above corollary.

The next result we prove about node-searching is that it is an NP-complete problem. The fact that node-searching is in NP follows from Theorem 2.3 since a recontamination-free strategy can be exhibited and proved to be a strategy in polynomial time. To prove the NP-hardness, we reduce the problem of edge-searching a graph G to the problem of node-searching a graph G_v obtained by replacing every edge $\{a, b\}$ of G by three edges in series; $\{a, m_1\}$, $\{m_1, m_2\}$, $\{m_2, b\}$ (see Fig. 5). We call the m_1's and m_2's *middle* vertices, and the vertices of G *initial* vertices. (Incidentally, notice the curious duality evident in the constructions of G_e and G_v, apparently reflecting the relation of the corresponding theorems.)

Theorem 2.5. $\mathrm{es}(G) = \mathrm{ns}(G_v) - 1$.

Once we have shown this theorem, the following is immediate.

Corollary 2.6. *The problem of deciding, given a graph G and an integer k, whether* $\mathrm{ns}(G) \leq k$ *is* NP*-complete.*

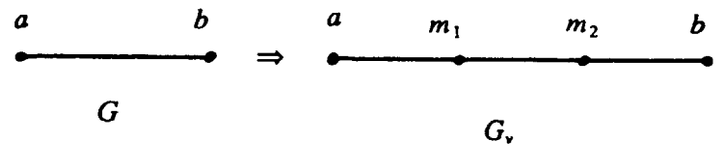


Fig. 5. Obtaining G_v from G.

Proof of Theorem 2.5. It is almost obvious that $\mathrm{ns}(G_v) \leq \mathrm{es}(G) + 1$. Indeed, given any edge-searching strategy of G, we can node-search G_v using at most one more searcher, as follows: every time an edge of G is cleared by a searcher, we clear the corresponding three edges of G_v, using, if necessary, an extra searcher for the sliding.

For the opposite direction, consider an optimal node-searching of G_v of the form described in Corollary 2.4. Without loss of generality, we may also assume that when a searcher is placed on a middle vertex of a triple edge of G_v, at least one of its initial vertices has already accepted a searcher. Indeed, we can always postpone the placement of searchers on middle vertices until this condition is fulfilled. Since no searcher can be deleted from a middle vertex unless the neighbouring initial vertex is not contaminated, this postponement causes no problems.

It is also convenient, for counting purposes, to have a distinct name (tag) associated with every searcher.

We now define an edge-searching of G by simulating a node-searching strategy of G_v as described above. In the simulation, for every searcher we place on G_v we place at most one new searcher on G, with, originally, the same name. The names of the searchers on G may change in the process, but almost always for each one

of them there will correspond one with the same name on G_v. The one exception will be made clear in the sequel.

To be specific, the rules are the following:

(i) For every searcher placed on an initial vertex of G_v, if the corresponding vertex of G is unguarded, place on it a searcher with the same name. Otherwise, simply change the name of its guard to the name of the searcher just used on G_v.

(ii) For every searcher deleted from an initial vertex of G_v, delete the searcher, if any, from the corresponding vertex of G.

(iii) When a searcher is placed on a middle vertex m_1 of a triple edge (a, m_1, m_2, b) of G_v whose second middle vertex m_2 is contaminated, clear the edge $\{a, b\}$ of G, if, of course, it is contaminated. Recall that in this case either a or b has a searcher in G_v, and therefore, if anything is to be done, also in G. If no recontamination is caused, the clearing of $\{a, b\}$ in G is done by sliding one of the searchers on a or b towards the other. Otherwise, a new searcher is used for the sliding. In both cases though, the name of the searcher that is slid is changed to that of the searcher on m_1. Of course, this searcher is left as a guard on the vertex towards which it is slid, only if this vertex is unguarded.

(iv) Do nothing in any other case.

First, let us prove that the above rules define an edge-searching of G. Indeed, when we delete a searcher from a vertex a of G according to rule (ii), every middle vertex in G_v adjacent to a must have accepted a searcher. But then, by (iii), all (single) edges incident on a in G must be clear.

We now prove that the number of searchers used on G is at most $\text{ns}(G_v)$. As indicated, it is enough to show that the names of the searchers on G are correctly manipulated.

Deleting a searcher from an initial vertex of G_v causes no problem since if there is a synonymous searcher on G, it will be on the corresponding vertex, and so it will be deleted.

Suppose that in G_v we are to delete a searcher named s placed, as described in (iii), on a middle vertex m_1. The corresponding searcher in G must be either on a or on b. But a is clear in G_v. So, either there was no reason to place a searcher named s on a in G, or this searcher changed its name when a accepted a searcher in G_v. By a similar argument, the only possibility left is to have s on b in G, and b to be contaminated in G_v. But then, in G_v, m_2 must have a searcher with no synonymous one ever used in G. This searcher will stay on m_2 until b gets a searcher in G_v. But when this happens, the name of s will change. So the searcher on m_2 will account for keeping a searcher on G for a while, without having one with the same name on G_v.

We finally prove that, in G, we actually use $\text{ns}(G_v)-1$ searchers.

Consider the placing of a searcher s by which we raise the number of searchers on G_v to $\text{ns}(G_v)$. The next move must be a deleting move, and, by Corollary 2.4, it is s, or an immediate neighbour of s, that must be deleted. If s is as in (iii), then a searcher from a must be deleted next. But then there is no new searcher used on

G since this searcher on a can be used for the sliding. If s is placed on an initial vertex a and it is s that is to be deleted next, then again on G we place no new searcher. This is so, because all edges incident on a in G have been cleared before placing a. If, finally, the searcher to be removed next lies on a middle vertex m_1 of a triple (a, m_1, m_2, b), then again $\{a, b\}$ has been cleared before the placement of a. So, no new searcher is used in G. Of course, in all cases we could have renaming or sliding of existing searchers.

This shows that at most $\mathrm{ns}(G_v)-1$ searchers are used and completes the proof of the theorem. □

3. Searching and pebbling

Let G be an undirected graph. We define $\Delta(G)$ to be the set of dags whose underlying directed graph equals G, that is, if D is a dag resulting by assigning directions to the edges of G, then $D \in \Delta(G)$. We say that D is a *directive* of G.

For a graph G, define $\mathrm{mpb}(G)$ to be $\min_{D\in\Delta(G)} \mathrm{pb}(D)$, the minimum progressive black pebble demand over all directives of G. Similarly, $\mathrm{mpbw}(G) = \min_{D\in\Delta(G)} \mathrm{pbw}(D)$ for the black and white pebble demand. We shall prove the following theorem.

Theorem 3.1. *For any graph* G, $\mathrm{mpb}(G) = \mathrm{ns}(G) = \mathrm{mpbw}(G)$.

Proof. Obviously, it suffices to prove that $\mathrm{mpb}(G) \leqslant \mathrm{ns}(G) \leqslant \mathrm{mpbw}(G)$ since it is clear that $\mathrm{mpbw}(G) \leqslant \mathrm{mpb}(G)$.

For the leftmost inequality, we consider an optimal, recontamination-free node-searching strategy S of G. We shall define a directive D of G which has pebble demand at most equal to the number of searchers used in S. D is defined orienting the edges of G, that is, defining on each edge a *head* and a *tail*. We consider as the tail of an edge e that endpoint of e which is visited first during S. To show that D has pebble demand no greater than the number of searchers in S, we shall show that if the searchers of S are viewed as black pebbles on D, then S is a (progressive) pebbling of D. For this, it is enough to prove that when we place a searcher on a vertex v, the adjacent vertices that have already accepted a searcher, i.e., the immediate predecessors of v in D, still have a searcher. But this is obvious, since recontamination is not allowed. This proves the leftmost inequality.

For the rightmost inequality, it is enough to prove that if we view the pebbles of a progressive black and white pebbling strategy on any D as (colourless) searchers, we get a node-searching strategy of G. In fact, we shall show that, for any vertex v and for any of its adjacent vertices w, there is an instant when they both simultaneously have a pebble-searcher. Since v accepts a unique pebble, this shows that all edges of G will be eventually cleared (with no recontamination).

Indeed, first suppose that w is a predecessor of v. If the pebble placed on v is black, then, when this placement occurs, w must have a pebble. If, on the other hand, a white pebble is placed on v, then its turning black must take place when w carries a pebble. So, the claim is proved for the first case. Now, if w is a successor of v, we just interchange the roles of v and w in the above argument. This completes the proof of the inequality and the theorem □

Notice how strongly we use in the above proof the hypothesis that each vertex is pebbled at most once. To show that this is necessary, we prove a result about the node-search number of trees. More general theorems along this line of edge-searching can be found in [12]. The proofs, though, are more complicated since it was not known that recontamination does not help.

Proposition 3.2. *The node-search number of a complete ternary tree T is equal to its height plus one.*

Proof. The proof is by induction on the height h of T. The fact that $h+1$ is an upper bound for $\mathrm{ns}(T)$ is almost trivial. For the other direction, let r be the root of T, and s_1, s_2, and s_3 be its immediate successors. Suppose that T_1, T_2, and T_3 are the three complete subtrees rooted on the s_i's. Suppose, towards a contradiction, that there is a strategy S, as in Corollary 2.4, for node-searching T with h searchers. This strategy, when restricted to one of the T_i's, obviously yields a node-searching of it. Therefore, by the inductive hypothesis, there are three distinct moments in S when all h searchers appear on each one of the t_i's, respectively. Say, without loss of generality, that the order of these moments is the same as the order of the indices of the corresponding subtrees. Then, at the second such moment, all h searchers are on T_2 and neither T_1 nor T_3 have any searcher on them. But then, since revisiting a vertex is not allowed, T_1 must be clear and T_3 untouched, a contradiction. □

It is easy to check now that any tree directed *from the root towards the leaves* can be pebbled, allowing repebbling, by just two pebbles. This shows that in Theorem 3.1, the no-repebbling hypothesis is necessary.

So, graph search provides an exact lower bound on the progressive pebble demand of any directive of a graph. Is it also a good *upper* bound? It is easy to see that, for a starshaped graph S with $n+1$ vertices (one at the centre, and the remaining n connected only with the center), $\mathrm{ns}(G)=2$, while $\mathrm{bw}(D)=n+1$, where D is the directive where all edges are directed towards the centre. So it is reasonable to examine only directives with bounded in-degree.

Let k be an integer. Let $\Delta_k(G)$ be the set of all directives of G with in-degree at most k. We shall prove the following theorem.

Theorem 3.3. *For any $D \in \Delta_k(G)$, $\mathrm{pbw}(D)$ (and therefore also $\mathrm{bw}(D)$) is at most equal to $(k+1)\mathrm{ns}(G)$.*

Proof. Let S be an optimal, recontamination-free strategy for node-searching G. We shall describe a progressive black and white pebbling strategy S_p for D using at most $(k+1)\text{ns}(G)$ pebbles. The steps of S_p follow one by one the steps of S according to the follwing rules:

(i) Whenever S places a searcher on a vertex v of G, S_p places a pebble on the same vertex on D. The colour of the pebble is black only if all the immediate predecessors of v in D have a pebble; otherwise, it is white. A white pebble turns black at the moment when all its immediate predecessors are pebbled.

(ii) When S deletes a searcher from a vertex v of $G(D)$, and if, in S, all searchers from the immediate successors of v have been removed, S_p deletes the corresponding pebble from D. Otherwise, the deletion of that pebble is postponed until this condition is satisfied.

We have to prove now that S_p is a pebbling of D, and that it uses at most $(k+1)\text{ns}(G)$ pebbles.

For the first statement, we only need to show that white pebbles are never deleted before they turn black. Since a pebble can only be deleted from a vertex v after the searcher on v has been removed in S, it suffices to prove that, at the instant before a searcher is removed from a vertex v in S, the pebble on v must be black. For each predecessor w of v, since the searcher on v is about to be removed, the edge (w, v) has been cleared and hence, a pebble has been placed on w. Moreover, as the searcher is still on v, the pebble has not been removed from w, proving that, at this instant, every predecessor of v has a pebble and hence, the pebble on v is black.

We now prove that S_p uses at most $(k+1)\text{ns}(G)$ pebbles. At any point of the execution of the strategy S_p, the pebbles on D can be partitioned into the following two classes:

(a) pebbles lying on a vertex v that, at the corresponding point of the implementation of S on G, carries a searcher;

(b) pebbles lying on a vertex v that, at the corresponding point of the implementation of S, does not have a searcher.

Obviously, the cardinality of the class (a) is at most $\text{ns}(G)$. Also, from rule (ii) of S_p, it follows that at least one immediate successor of any vertex v as in class (b) carries a searcher in the sense of S. But the number of those searchers can be at most $\text{ns}(G)$. From the fact now that the in-degree of D is k or less, we conclude that the pebbles of class (b) are at most $k\text{ns}(G)$. Therefore, the total number of pebbles is at most $(k+1)\text{ns}(G)$. □

4. Vertex separator

Let $G=(V, E)$ be an undirected graph. A *layout* of G is a one-to-one mapping $L: V \to \{1, \ldots, |V|\}$. With a layout we associate various interesting parameters. For example, the *bandwidth* of (G, L) is the maximum value the difference $|L(v)-L(w)|$ can take for an edge $\{v, w\}$ of G.

The *cutwidth* of (G, L) is the maximum number of edges crossing over any two consecutive vertices of the layout, i.e., it is

$$\max_{1 \leq i < |V|} |\{\{v, w\} \in E \mid L(v) \leq i \text{ and } L(w) > i\}|.$$

The *vertex separator* of (G, L) is the maximum number of vertices that are the least numbered endpoint of an edge crossing over any two consecutive vertices of L, i.e., it is

$$\max_{1 \leq i < |V|} |\{v \in V \mid L(v) \leq i \text{ and for some } w \in V\ (\{v, w\} \in E \text{ and } L(w) > i)\}|.$$

We consider layouts minimizing each one of the above parameters. The resulting layout-independent parameters of G are called *bandwidth*, *cutwidth* and *vertex separator* of G, respectively. They are denoted by $\text{bdw}(G)$, $\text{ctw}(G)$, and $\text{vs}(G)$, respectively.

A related parameter is the topological bandwidth of G, $\text{tbdw}(G)$, which is the least possible value of $\text{bdw}(G')$, where G' is any graph obtained from G by the insertion of degree-two vertices on the edges of G.

The layout problems have applications in various fields including sparse matrices, VLSI, scheduling, etc. For an overview see [7, 8]. In these papers some interesting relations between the search number and layout parameters are proved. In [7], it is shown that $\text{es}(G) \leq \text{ctw}(G)$, and that equality holds for degree-three graphs. In [8], it is shown that $\text{ns}'(G) \leq \text{tbdw}(G)$, and that equality holds for degree-three graphs. $\text{ns}'(G)$ is a variant of our node-search number. Unfortunately, in both inequalities the related parameters can get arbitrarily far apart if there is no restriction on the degree. The only case where a layout parameter and a search parameter are known to differ by at most one additive constant is described by the inequalities $\text{vs}(G) \leq \text{es}(G) \leq \text{vs}(G) + 2$, due to Turner [16].

We are now going to prove the following theorem.

Theorem 4.1. *For an arbitrary graph* G, $\text{ns}(G) = \text{vs}(G) + 1$.

By this theorem, results related to one of the above parameters, like NP-completeness [5], or polynomial-time dynamic programming algorithms [11], carry over to the other.

Proof of Theorem 4.1. We first prove that $\text{ns}(G) \leq \text{vs}(G) + 1$. Let L be a layout of G for which the vertex separator takes its least possible value. We carry out a node-searching of G following the rules:

(i) a searcher may be placed on a vertex v, only after all vertices w such that $L(w) < L(v)$ have accepted a searcher;

(ii) a searcher is deleted from a vertex immediately after all adjacent vertices have accepted a searcher. Ties are broken arbitrarily.

It is not hard to see that there is a search strategy subject to the above rules. Consider a point during its implementation when the number of searchers on G is maximum, and suppose that the vertices with searchers on them are $u_1, \ldots, u_n$. Let $i = \max\{L(u_1), \ldots, L(u_n)\}$.

According to the rules above, any vertex w for which $L(w) \leq i$ is not contaminated. Let $u_k = L^{-1}(i)$. Then, any vertex from the set $\{u_1, \ldots, u_n\} - \{u_k\}$ must be connected either with u_k or with a vertex w which has not yet accepted a searcher, and for which $L(\omega) > i$, because, otherwise its searcher would have been deleted. But this implies that the vertex separator for this layout is at least $n-1 \geq \text{ns}(G)-1$. From the choice of the layout now, we get that $\text{vs}(G) \geq \text{ns}(G)-1$.

For the opposite direction, consider an optimal, recontamination-free node-searching strategy S for G. Define a layout $L: V \to \{1, \ldots, |V|\}$ so that $L(v) < L(w)$ iff v accepts a searcher before w does. For any $1 \leq i < |V|$, let

$$D_i = \{v \in V \mid L(v) \leq i \text{ and for some } w \in V\ (L(w) > i \text{ and } \{v, w\} \in E)\}.$$

Let i_0 be an index such that D_{i_0} takes its maximum cardinality. Then

$$\text{vs}(G) \leq |D_{i_0}|. \tag{1}$$

Consider that point during the execution of the node-searching strategy S at which exactly the vertices v for which $L(v) \leq i_0$ have accepted a searcher. Let $u_i, \ldots, u_n$ be those among them that at this point carry a searcher. Since the vertices in D_{i_0} are joined to vertices not yet cleared, we conclude that

$$D_{i_0} \subset \{u_1, \ldots, u_n\}. \tag{2}$$

We examine now the next move in the strategy S. If that is a move of placing a new searcher, then, clearly,

$$|\{u_1, \ldots, u_n\}| < \text{ns}(G). \tag{3}$$

From (1), (2), and (3) we conclude that $\text{vs}(G) < \text{ns}(G)$.

Suppose, on the other hand, that the next move of S is a move deleting a searcher from a vertex among the u_i's. If this vertex is u_i, then, since recontamination is not allowed, $u_i \notin D_{i_0}$. Therefore,

$$D_{i_0} \subset \{u_1, \ldots, u_n\} - \{u_i\}. \tag{4}$$

From (1) and (4) we have that $\text{vs}(G) < \text{ns}(G)$. This completes the proof of the theorem. □

Acknowledgment

We would like to thank the Theory Group at the National Technical University of Athens for stimulation, encouragement, and valuable suggestions. Special thanks to George Papageorgiou for his contribution to the proof of Theorem 3.1. We would also like to thank the referee for comments and criticism that led to an improvement of the presentation.

References

[1] J.R. Gilbert, T. Lengauer and R.E. Tarjan, The pebbling problem is complete in polynomial space, *Proc. 11th ACM Symp. on Theory of Computing* (1979) 237-248.
[2] J.E. Hopcroft, W. Paul and L. Valiant, On time versus space, *J. ACM* **24** (1977) 332-337.
[3] A. LaPaugh, Recontamination does not help to search a graph, Tech. Rept., Electrical Engineering and Computer Science Department, Princeton University, 1983.
[4] C.E. Leiserson, Area efficient graph layouts (for VLSI), *Proc. 21st IEEE Symp. on Foundations of Computer Science* (1980) 270-281.
[5] T. Lengauer, Black-white pebbles and graph separation, Tech. Rept., Bell Labs., 1980.
[6] T. Lengauer and R.E. Tarjan, Upper and lower bounds on time-space trade-offs, *Proc. 11th ACM Symp. on Theory of Computing* (1979) 262-277.
[7] F. Makedon, Layout problems and their complexity, Ph.D. Thesis, Northwestern University, Evanston, IL, 1982.
[8] F. Makedon, C.H. Papadimitriou and I.H. Sudborough, Topological bandwidth, in: G. Ausiello and M. Protasi, eds., *Proc. 8th Coll. on Trees in Algebra and Programming*, Lecture Notes in Computer Science **159** (Springer, Berlin, 1983) 315-331.
[9] N. Megiddo, S.L. Hakimi, M.R. Garey, D.S. Johnson and C.H. Papadimitriou, The complexity of searching a graph (preliminary version), *Proc. 22nd IEEE Symp. on Foundations of Computer Science* (1981) 376-381.
[10] F. Meyer auf der Heide, A comparison of two variations of a pebble game on graphs, *Theoret. Comput. Sci.* **13** (1981) 315-322.
[11] B. Monien and I.H. Sudborough, Minimizing edge length and other graph labeling problems, Manuscript, 1983.
[12] T.D. Parsons, Pursuit-evasion in a graph, in: Y. Alani and D. R. Lick, eds., *Theory and Applications of Graphs* (Springer, Berlin, 1976) 426-441.
[13] N. Pippenger, Pebbling, IBM Res. Rept., RC8258, 1980.
[14] A.L. Rosenberg and I.H. Sudborough, Bandwidth and pebbling, *Computing* **31** (1983) 115-139.
[15] R. Sethi, Complete register allocation problems, *SIAM J. Comput.* **4** (1975) 226-248.
[16] J. Turner, Personal communication, 1982.