Computing 31, 115–139 (1983)

Computing
© by Springer-Verlag 1983

Bandwidth and Pebbling

A. L. Rosenberg*, Durham, and **I. H. Sudborough****, Evanston

Received February 4, 1982; revised May 24, 1982

Abstract — Zusammenfassung

Bandwidth and Pebbling. The main results of this paper establish relationships between the bandwidth of a graph G – which is the minimum over all layouts of G in a line of the maximum distance between images of adjacent vertices of G – and the ease of playing various pebble games on G. Three pebble games on graphs are considered: the well-known computational pebble game, the "progressive" (i.e., no recomputation allowed) version of the computational pebble game, both of which are played on directed acyclic graphs, and the quite different "breadth-first" pebble game, that is played on undirected graphs. We consider two costs of a play of a pebble game: the minimum number of pebbles needed to play the game on the graph G, and the maximum *lifetime* of any pebble in the game, i.e., the maximum number of moves that any pebble spends on the graph. The first set of results of the paper prove that the minimum lifetime cost of a play of either of the second two pebble games on a graph G is precisely the bandwidth of G. The second set of results establish bounds on the pebble demand of all three pebble games in terms of the bandwidth of the graph being pebbled; for instance, the number of pebbles needed to pebble a graph G of bandwidth k is at most $\min(2k^2+k+1,\ 2k\log_2|G|)$; and, in addition, there are bandwidth-k graphs that require $3k-1$ pebbles. The third set of results relate the difficulty of deciding the cost of playing a pebble game on a given input graph G to the bandwidth of G; for instance, the Pebble Demand problem for n-vertex graphs of bandwidth $f(n)$ is in the class NSPACE $(f(n)\log^2 n)$; and the Optimal Lifetime Problem for either of the second two pebble games is NP-complete.

AMS Subject Classifications: 05C99, 68E10, 94C15.
Key words: Bandwidth, graph pebbling.

Bandweite und Markenspiele. Die Hauptergebnisse dieser Arbeit ergeben Beziehungen zwischen der „Bandwidth" eines Graphen G – die das Minimum ist, über alle Projektionen von G auf eine Linie, von dem maximalen Abstand zwischen Bildern benachbarter Knoten von G – und der Leichtigkeit, verschiedene „Pebble Games" auf G zu spielen. Es werden drei Pebble Games auf Graphen betrachtet: das wohlbekannte „computational" Pebble Game, die „progressive" (d. h. keine Wiederberechnung erlaubt) Version des computational Pebble Game, von denen beide auf directed acyclic Graphen gespielt werden, und das ziemlich verschiedene „breadth-first" Pebble Game, das auf undirected Graphen gespielt wird. Wir betrachten zwei verschiedene Kosten für das Pebble Game: die minimale Anzahl von Pebbles, die man braucht, um das Pebble Game auf einem Graphen G zu spielen, und die maximale *Lebensdauer* eines Pebble in einem Spiel, d. h. die maximale Anzahl von Zügen während denen ein Pebble auf dem Graphen verweilt. Die erste Gruppe von Hauptergebnissen in dieser Arbeit zeigt, daß die minimalen Lebensdauer-Kosten eines Spielverlaufs in einem der beiden letzten Pebble Games auf einem

* A portion of this author's research was done while visiting the Department of Computer Science, University of Toronto, Toronto, Canada; a portion was done while with the Mathematical Sciences Department, IBM Research Center, Yorktown Heights, N.Y.; and a portion was supported by NSF Grant MCS-8116522.

** A portion of this author's research was supported by NSF grant MCS-79-08919.

Graphen genau die Bandwidth von G ist. Die zweite Gruppe von Ergebnissen stellt obere Schranken auf für die Anzahl von benötigten Pebbles in Abhängigkeit von der Bandwidth des betrachteten Graphen, z. B. um einen Graphen G mit Bandwidth k zu pebblen, braucht man höchstens $\min(2k^2+k+1, 2k\log_2|G|)$ Pebbles; ferner gibt es Graphen G von Bandwidth k für die man $3k-1$ Pebbles braucht. Die dritte Gruppe von Ergebnissen setzt die Schwierigkeit, die Kosten eines Pebble Game auf einem gegebenen input-Graphen G festzustellen, in Beziehung zur Bandwidth von G, z. B. das „Pebble Demand Problem" für Graphen mit n vertices von Bandwidth $f(n)$ ist in der Klasse NSPACE $(f(n)\log^2 n)$; und das „Optimal Lifetime Problem" ist für jedes der beiden letzten Pebble Games NP-vollständig.

1. Introduction

The purpose of this paper is to demonstrate certain intimate relationships between two seemingly disparate properties of a graph, its *bandwidth* and the ease with which it can be *pebbled.*

The *bandwidth* of an undirected graph G is the minimum, over all *layouts* of G in the line – that is, one-to-one mappings

$$\lambda: \text{Vertices}(G) \to \{1, 2, \dots, |G|\},$$

($|G|$ denoting the number of vertices of G) – of the maximum distance between adjacent vertices of G. The bandwidth of a graph is important because of its relevance to computations involving sparse matrices [1, 2, 17] and because of its representing the simplest instance of a variety of graph-embedding problems (simplest since the target graph is a line) that have applications to routing computations on fixed-interconnection networks of processors [7] and to finding storage representations for data structures [10, 18, 19].

Pebble games for graphs originated in the study of problems concerning register allocation and space requirements for computations [15, 3]. The basic form of such games is the following. One is given a directed acyclic graph G and an endless supply of tokens called "pebbles". One is to play the following one-person game on G: one is allowed to pebble any vertex of G whenever all of that vertex's (directed) predecessors are pebbled; and one is allowed to remove a pebble from a vertex whenever one wishes. The goal of the game is to pebble (and then remove the pebbles from) all of the "sink" vertices of G, i.e., vertices having no successors. The traditional measure of the merit of a play of the pebble game is the number of pebbles one uses in a play of the game – the fewer pebbles the better. Pebble games represent the most successful tool yet discovered for studying register-allocation and space-requirement problems, as well as a variety of other problems related to such basic computational issues as time-space tradeoffs. (See [16] for an excellent discussion of the applications of pebbling arguments as well as a survey of the major results obtained via such arguments.)

There seems on the surface to be little relationship between the notion of the bandwidth of a graph and the ease of playing any pebble game on the graph. And, indeed, it appears that no tight such relationship exists if one restricts attention to the conventional (number of pebbles used) measure of the merit (or "ease") associated with pebble games. (This disclaimer notwithstanding, T. Lengauer [8] has proved a number of results relating ease of "black/white" graph pebbling and a different notion of the width of a graph; and we (Section 3) present results relating the ease of pebbling a graph to the bandwidth of the graph itself and to the bandwidth of some "expansion" of the graph.) But, there is one measure of the merit of a play of a pebble game on a graph, that yields *precisely* the bandwidth of the graph. And the preceding assertion is true for two seemingly unrelated types of pebble games, one of which is a variant of the computational pebble game described above while the other is a marked departure from the form of such games. The measure of merit that characterizes bandwidth in both cases is the *lifetime* of the longest lived pebble in the game, where the lifetime of a pebble p is the number of pebbles that are removed from the graph while p resides on the graph.

The two costs of the computational pebble game, namely, number of pebbles used and maximum pebble lifetime, are quite closely related. The number of pebbles needed to pebble a directed acyclic graph G is within 1 of the lifetime-cost of some "expansion" (a type of homeomorph) of G. Thus the conventional cost of the computational pebble game is also related to the bandwidth problem for graphs, though in an indirect way.

One application of the upper bounds we shall give for the pebble demand for directed acyclic graphs (dags) of limited bandwidth arises from considering the dags as representations of computations. The best result known for the pebble demand of an arbitrary fixed-vertex-degree dag is due to Hopcroft, Paul, and Valiant [6]: any n-vertex dag can be pebbled using $O(n/\log n)$ pebbles. Since the pebbles correspond to memory registers, their number corresponds to the space requirements of the computation represented by the dag. The upper bounds we shall present indicate that the $O(n/\log n)$ bound can be improved when the dags to be pebbled have limited bandwidth. For example, an n-vertex dag of bandwidth $O(n^{1/2})$ can be pebbled with only $O(n^{1/2} \log n)$ pebbles.

The main advantage of having results such as those we are about to present is that they afford one different ways of looking at important problems, and they allow one to bring the results and techniques from one problem area to another. We illustrate this advantage by presenting two results about pebbling that follow from results about bandwidth. The first result establishes the difficulty of finding optimal plays of our pebble games by citing known results about the difficulty of determining the bandwidth of a graph. The second result proves that the lifetime-cost of any play of either of our pebble games on an n-reticulated graph (a weak version of n-superconcentrator that includes all homeomorphs of side-n grids and side-$2n$ pyramids, for example) grows at least as fast as $n^{1/2}$.

The remainder of the paper is organized as follows. Section 2 establishes the "equivalence" of the lifetime cost of the progressive computational pebble game on a graph and the graph's bandwidth; Section 4 does the same for our new "breadth-first" pebble game; Section 3 presents results relating the conventional cost of the computational pebble game with graph bandwidth; and Section 5 presents the two corollaries of the two equivalences.

2. NRA Pebbling and Bandwidth

Our first pebble game, the *NRA* (for *no recomputation allowed*) pebble game is played on a connected directed acyclic graph (*dag*, for short). Its rules are as follows.

1. No vertex of the "input" dag G ever holds more than one pebble.
2. No vertex of G is ever pebbled more than once. (This is the "NRA" clause.)
3. At each move of the game one does one of the following.
   A. One can place a pebble on any "virgin" (i.e., never before pebbled) vertex of G providing that all of that vertex's predecessors contain pebbles.
   B. One can, whenever possible, remove a pebble from a vertex. (Of course, this can be done only after all of the vertex's successors have been pebbled.)
4. The game ends when the last pebble is removed from G.

The NRA pebble game differs from the traditional computational pebble game in its not allowing recomputation, i.e., in Rule 2.

The *lifetime* of a pebble in a play of the NRA pebble game is the number of type B moves during which the pebble stays on the dag, not counting the move on which it is removed. (We assume that a pebble is discarded when it is removed from the dag.)

A *dag-ing* of the undirected graph G is any orienting of the edges of G that results in a directed acyclic graph.

The dag G is *k-NRA-pebbleable*, $k \in N$, if there is a play of the NRA pebble game on G in which no pebble has lifetime exceeding k.

Theorem 1: *The undirected graph G has bandwidth k if and only if some dag-ing of G is k-NRA-pebbleable.*

Proof: Say first that G has bandwidth k, and let λ be a bandwidth-k layout of G in the line. Consider the dag-ing of G defined by:

The edge (v, w) of G is directed from v to w just when $\lambda(v) < \lambda(w)$.

This method of dag-ing G yields a left-to-right ordering of G's vertices. We risk no misunderstanding, therefore, if we refer to the leftmost vertex of G or to the leftmost pebble residing on G.

Let us play the NRA pebble game on this dag as follows. At each move, we remove the leftmost pebble from G if possible; if no removal is possible, we pebble the leftmost virgin vertex of G. Note first that, because of our method of dag-ing G, the described play of the game is a legitimate one (no vertex is pebbled until all of its predecessors contain pebbles, and no vertex is pebbled more than once.) Moreover, since λ is a bandwidth-k layout of G, no pebble can sit on the dag for more than k type B moves. Thus G is k-NRA-pebbleable.

Conversely, say that some dag-ing G' of G is k-NRA-pebbleable, and let us focus on a play of the NRA pebble game on G' in which no pebble has a lifetime exceeding k. Consider the following layout λ of G in the line: for each vertex v of G' (hence, also of G),

$$\lambda(v) = \text{the type } B \text{ move at which } v \text{ is depebbled.}$$

Since every vertex of G' gets pebbled and depebbled precisely once, the mapping λ is well-defined and one-to-one (hence is a layout of G). Since no pebble used in this play of the game resides on G' for more than k type B moves, λ places adjacent vertices of G' at most distance k apart in the line: the source end of an edge must contain a pebble when the target end is pebbled; and neither pebble can sit for more than k type B moves. It follows that G has bandwidth at most k. □

We thus have our first characterization of bandwidth in terms of pebbling.

3. Relationships Between Pebble Demand and Bandwidth

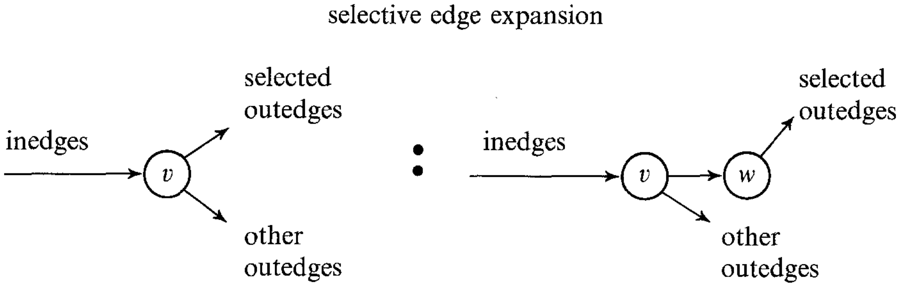
A. An Indirect Relationship

There appears intuitively to be some relationship between our notion of the cost of a play of the pebble game and the conventional, number of pebbles used, cost. We now make explicit an indirect such relationship.

Let G be an n-vertex directed graph (digraph, for short), let v be a vertex of G, and let $e_1, \ldots, e_n$ be some (maybe all) of the edges leaving v in G. The $(e_1, \ldots, e_n)$-*expansion* of G is the $(n+1)$-vertex digraph obtained by adding to G a new vertex w and an edge from v to w, and reallocating the edges incident to v in G so that:

1. all edges into v in G remain edges into v in G';
2. the edges $e_1, \ldots, e_n$ become edges out of w in G';
3. all other edges out of v in G remain edges out of v in G'.

One can obviously view "selective edge expansion" as an operation on a directed graph, defined pictorially by:



The digraph G' is an *expansion* of the digraph G if G' is obtainable from G by a finite sequence of selective edge expansions. The reader can easily extend this notion of graph expansion to undirected graphs.

Say that the dag G has *pebble demand* k if there is a play of the NRA pebble game on G for which no more than k pebbles ever sit on G at any time. Pebble demand is the traditional [3, 7, 15, 16] measure of the cost of a play of the computational pebble game.

Lemma 1: *If the dag G has pebble demand k, then some expansion of G is k-NRA-pebbleable. If some expansion of G is $(k-1)$-NRA-pebbleable, then G has pebble demand at most k.*

Proof: Assume first that some expansion of G is $(k-1)$-NRA-pebbleable. That expansion, then, has pebble demand at most k, for one of the pebbles residing on the graph at any instant must be the last to be removed. But it is easy to verify that the pebble demand of a graph can only be increased (by at most 1) by an expansion. It follows that G has pebble demand at most k.

Assume next that G has pebble demand at most k. We shall construct in stages an expansion of G that is k-NRA-pebbleable.

Stage 1: Replace the endless supply of pebbles used to pebble G by $k+1$ infinite buckets of colored pebbles, with colors $1, 2, \ldots, k+1$. Play the NRA pebble game on G exactly as before, but make sure now that no two pebbles coresident on G at any time have the same color. (This is possible by assumption.)

Stage 2: Modify the play of the NRA pebble game on G by substituting for each move the following procedure:

```
For $1 \leq i \leq k+1$:
    if a pebble with color $i$ would have been placed on the graph or removed from the
    graph at this move in the original game, then do the same thing;
    if a pebble with color $i$ would not have been touched at this move in the original
    game, then:
        if a pebble with color $i$ is not on $G$ at this time, then do nothing;
        if a pebble with color $i$ is on the graph at this time, then *replace* it with another
        pebble of the same color.
```

Stage 3: Proceed as in Stage 2, with one exception: whenever a pebble of color i would be "replaced" in Stage 2, instead perform the following actions. (a) Expand the current graph G: let v be the vertex on which the pebble in question resides. Let $e_1, \ldots, e_n$ be those edges leaving v that go to vertices that have never contained a pebble. Form the $(e_1, \ldots, e_n)$-expansion of G by replacing the vertex v on which the pebble in question resides by the chain $v \rightarrow w$ (as in the definition of expansion) with the pebble still on v. (b) Place a pebble of an unused color on vertex w (since G has pebble demand k, at most k pebbles will be sitting on the current version of G, so the desired unused color will exist). (c) Remove the pebble from v.

Note in this last stage that all successors of v that have already been pebbled remain successors of v, and only never-pebbled successors become successors of w.

Once the modified NRA pebble game is over, one is left with a graph G' which is an expansion of G, and which is k-NRA-pebbleable. We relegate the verification of this last assertion to the Appendix. □

As an immediate consequence of Theorem 1 and Lemma 1, we have:

Theorem 2: *If the dag G has pebble demand k, then the undirected version of some expansion of G has bandwidth at most k. If the undirected graph G has bandwidth k, then some dag-ing of G has pebble demand at most $k+1$.*

Proof: If the dag G has pebble demand k, then by Lemma 1 some expansion of G is k-NRA-pebbleable, and so by Theorem 1 the undirected version of this expansion has bandwidth k. Conversely, if the undirected graph G has bandwidth k, then by Theorem 1, some dag-ing of G is k-NRA-pebbleable, and so, by the beginning of the proof of Lemma 1, this dag-ing of G has pebble demand at most $k+1$. □

Remark: If the rules of the pebble game are modified to allow *sliding* of pebbles – i.e., shifting a pebble from a predecessor of vertex x to vertex x whenever all predecessors of x contain pebbles – then the second assertion of Theorem 2 can be strengthened to:

If the undirected graph G has bandwidth k, then some dag-ing of G has pebble demand at most k.

Proof Sketch: Let λ be a bandwidth-k layout of G. Direct the edge that connects x and y from x to y just when $\lambda(x) < \lambda(y)$. The following pebbling strategy for this dag-ing of G uses k pebbles. (1) Place pebbles on the leftmost k vertices of G. (2) Iteratively, if the leftmost pebbled vertex (call it x) is a predecessor of the leftmost unpebbled vertex (call it y), then slide the pebble from x to y; otherwise, delete the pebble from x and add a pebble to y. Details are left to the reader. □

We turn now to a study of more direct relationships between the pebble demand of a graph and the graph's bandwidth.

B. Direct Relationships

We now transfer our attention from the NRA pebble game to the conventional game (with recomputation allowed). Note that for the NRA game, an upper bound on the bandwidth of a digraph gives no upper bound on NRA pebble demand. For example, the digraph with vertex set $\{1, 2, \ldots, 2n\}$ and edges

$$(i, i+1) \text{ for } 1 \le i \le 2n-1$$

and

$$(i, 2n-i+1) \text{ for } 1 \le i \le n-1$$

(which is the computation graph that arises from a linear recursion) has bandwidth 2 but NRA pebble demand $n+1$.

In the last subsection we showed that an undirected graph with bandwidth k can have its edges directed so that the resulting dag has pebble demand $k+1$ (k if we allow sliding). In fact, the direction chosen for each edge of G (which is dictated by the small-bandwidth layout λ of G) is such that the resulting dag G' is topologically sorted by the layout λ. What happens to pebble demand if we are not allowed to choose a happy set of directions for the edges? In other words, what is the pebble demand of an arbitrary dag of bandwidth k? (The dag $G=(V, E)$ has bandwidth k under the layout λ iff $|\lambda(x) - \lambda(y)| \le k$ for all edges (x, y) of G.) In this subsection, we show that the pebble demand of a bandwidth-k dag is bounded above by:

$$\min(2k^2+k+1,\ 2k \log_2 |G|).$$

This bound follows from two separate observations we shall verify: (1) If the graph G has bandwidth k, then it has pebble demand $2k+1$ in the so-called *black/white* pebble game. It then follows by a result in [11] that G has pebble demand $2k^2+k+1$ in the conventional (black) pebble game. (2) If G has bandwidth k, then it has pebble demand at most $2k\log_2|G|$. This bound is shown by describing a divide-and-conquer algorithm for pebbling a graph. The first result gives the better upper bound on pebble demand for graphs with small bandwidth (less than $\log_2|G|$). For example, if G has 2^{10} vertices and bandwidth 3, then the first result gives the bound 22 and the second yields the bound 60. We now verify each of these observations in turn.

We start with a description of the black/white pebbling game and a strategy for playing the game.

The rules for the black/white pebble game on a dag G are as follows.

1. No vertex of G ever holds more than one pebble.
2. At each move of the game one does one of the following.
   A. One can place a black pebble on any vertex v of G provided that all of v's predecessors contain pebbles.
   B. One can place a white pebble on any vertex of G.
   C. One can remove a black pebble from any vertex of G.
   D. One can remove a white pebble from any vertex provided that all of its predecessors contain pebbles.

Note that when only black pebbles are used, this game is just the conventional computational pebble game (with recomputation allowed). The goal of the black/white pebble game, just as of the conventional (black) pebble game, is to place a pebble on all of the "sink" vertices of G (not necessarily at the same time) and then to remove all of the pebbles from G.

The dag G has *black/white pebble demand* k ($k\in N$) if there is a play of the black/white pebble game on G for which no more than k black/white pebbles sit on G at any time.

Let G be an arbitrary dag of bandwidth k, and let the one-to-one function λ: Vertices$(G)\to\{1,2,\ldots,|G|\}$ be a bandwidth-k layout of G. We describe now an algorithm for pebbling the vertices of G in the black/white pebble game, that never leads to more than $2k+1$ pebbles sitting on the vertices of G at any time. This algorithm will demonstrate that bandwidth-k dags have black/white pebble demand at most $2k+1$.

In fact, our algorithm pebbles the vertices of G in the order given by the layout λ: at any given point in the algorithm, pebbles will sit only on vertices $\lambda^{-1}(i),\ldots,\lambda^{-1}(i+2k)$, for some positive integer i $(1\le i\le|G|-2k)$.

The Algorithm:

1. Put white pebbles on all of the vertices $\lambda^{-1}(1),\ldots,\lambda^{-1}(2k)$; then replace the white pebbles on $\lambda^{-1}(1),\ldots,\lambda^{-1}(k)$ by black pebbles.

```
2. For $i=1$ step 1 until $n-2k$ do
   begin
   A. add a white pebble to $\lambda^{-1}(i+2k)$;
   B. replace the white pebble on $\lambda^{-1}(i+k)$ by a black pebble;
   C. remove the black pebble from $\lambda^{-1}(i)$
   end
3. Replace all the white pebbles by black pebbles and then remove all of the black
   pebbles from the graph.
```

The correctness of the algorithm is easy to establish. In Step 1 all of the predecessors of the vertices $\lambda^{-1}(1), \dots, \lambda^{-1}(k)$ are pebbled; hence the white pebbles can be taken off and replaced by black pebbles. In Step (2 B) all of the predecessors of $\lambda^{-1}(i+k)$ contain pebbles, since the graph has bandwidth k and $\lambda^{-1}(i), \dots, \lambda^{-1}(i+2k)$ contain pebbles. Finally, in Step 3, all of the white pebbles can be replaced by black pebbles: since the graph has bandwidth k, all of the vertices $\lambda^{-1}(n-2k+1), \dots, \lambda^{-1}(n)$ contain pebbles, with white pebbles sitting only on vertices $\lambda^{-1}(n-k+1), \dots, \lambda^{-1}(n)$, and therefore all vertices with white pebbles have all their predecessors pebbled.

Our algorithm pebbles the graph G using at most $2k+1$ black/white pebbles; therefore, G has black/white pebble demand at most $2k+1$. It follows from a result of Meyer auf der Heide [11] that G has black pebble demand at most $2k^2+k+1$. This verifies our first observation:

Lemma 2: *If a dag G has bandwidth k, then it has pebble demand at most* $2k^2+k+1$.

In order to verify our second observation, we now describe a recursive divide-and-conquer algorithm for placing black pebbles on a graph of small bandwidth. The goal is, of course, to use as few pebbles as possible. Let G be a dag of bandwidth k; and let the one-to-one function

$$\lambda\colon \text{Vertices}(G) \to \{1, 2, \dots, |G|\}$$

be a bandwidth-k layout of G. Let

$$B(j,k) = \{v \mid j \le \lambda(v) \le j+k-1\}$$

denote the block of k vertices starting with the vertex $\lambda^{-1}(j)$ under the layout λ. The algorithm works by pebbling all of the vertices in a block $B(j,k)$ at a given time. Note that $B(j,k)$ divides the vertices of G into two sets: the set of vertices assigned to integers less than j, which we say are *to the left* of $B(j,k)$, and the set of vertices assigned to integers greater than $j+k-1$, which we say are *to the right* of $B(j,k)$. This division is of consequence since no vertex to the right of $B(j,k)$ is adjacent to a vertex to the left of $B(j,k)$ (since G has bandwidth k and $B(j,k)$ consists of k consecutive vertices under the layout λ). This observation underlies our ability to use a "divide-and-conquer" strategy to pebble G.

Let v be a vertex in $B(j,k)$ that can receive a pebble by a sequence $s_1, s_2, \dots, s_t$ of pebble-game steps during which no other vertex in $B(j,k)$ receives a new pebble. Let w be a vertex to the right of $B(j,k)$ that receives a pebble during this sequence, say at step s_h. Then the subsequence σ obtained by deleting from $s_1, s_2, \ldots, s_h$ all steps that add a pebble to vertices on the left of $B(j,k)$ forms a valid sequence of pebble game steps that puts pebbles only on vertices to the right of $B(j,k)$ and culminates in a pebble being placed on w. This is so because the addition of pebbles to the left of $B(j,k)$ cannot help add a pebble to w – which is to the right of $B(j,k)$ – unless it first helps add a pebble to a vertex in $B(j,k)$. However, we know that the sequence σ does not add pebbles to $B(j,k)$. Thus by focussing on $B(j,k)$, we have divided the problem of pebbling the graph G into two subproblems: pebbling the vertices to the left of $B(j,k)$ and pebbling the vertices to the right of $B(j,k)$. *If* j is chosen to be roughly in the middle of the portion of the graph currently being pebbled, then we have divided the problem of pebbling the graph G into two roughly half-size problems by considering $B(j,k)$. We now describe a recursive procedure that implements this informal divide-and-conquer pebbling strategy.

Let *PEBBLE* (i,j,m,μ) denote a recursive procedure that pebbles all of the vertices in $B(m,k)$ of a graph G, by a sequence of steps that never adds a pebble to any vertex v of G for which $\lambda(v) < i$ or $\lambda(v) > j+k-1$; i.e., the procedure adds pebbles to $B(m,k)$ by steps that do not add pebbles to the left of $B(i,k)$ or to the right of $B(j,k)$. In order to implement the desired divide-and-conquer strategy, the procedure PEBBLE (i,j,m,μ) will always be invoked with m midway between i and j. When the procedure is invoked, some of the vertices in the blocks $B(i,k)$ and $B(j,k)$ may previously have been pebbled.

A pebble will be called a *p-pebble*, for some $p \geq 1$, if it is placed on the graph G during a call to the procedure PEBBLE whose last argument $\mu = p$, but excluding those times during such a call when the procedure is called (recursively) with the last argument $\mu = p+1$. This last argument μ is included only for convenience; its purpose is simply to differentiate between pebbles placed at different times on the vertices of the graph G.

```
Procedure PEBBLE $(i,j,m,\mu)$
          if $|i-j| \leq 2k$
          then place $\mu$-pebbles on all possible vertices in
               $B(m,k)$ by steps which add no pebbles to
               vertices $v$ such that $\lambda(v) < i$ or $\lambda(v) > j+k$;
          else begin
          place $\mu$-pebbles on all vertices in $B(m,k)$
               that have no predecessor;
               $p \leftarrow (i+m)/2$;
               $r \leftarrow (j+m)/2$;
               flag $\leftarrow 1$;
               while flag $= 1$ do
                    begin
                    flag $\leftarrow 0$;
                    PEBBLE $(i,m,p,\mu+1)$;
                    PEBBLE $(m,j,r,\mu+1)$;
                    PEBBLE $(p,r,m,\mu)$;
```

```
            *if* a $\mu$-pebble is added to $B(m,k)$ in the
            last step *then* flag $\leftarrow 1$;
            remove all $(\mu+1)$-pebbles from $G$
            *end*
    *end*
```

The procedure works as follows. If $|i-j| \leq 2k$, then the vertices in $B(m,k)$ are just pebbled directly. Alternatively, if $|i-j| > 2k$, then PEBBLE(i,j,m,μ) initially adds pebbles to all vertices in $B(m,k)$ that have no predecessors. At that point, values p and r are chosen midway between i and m and midway between m and j, respectively. The algorithm then (1) pebbles all of the vertices in $B(p,k)$ that are possible to pebble by steps that add no pebbles to the left of $B(i,k)$ or to the right of $B(m,k)$, via a recursive call to PEBBLE$(i,m,p,\mu+1)$, and (2) pebbles all of the vertices in $B(r,k)$ that are possible to pebble by steps that add no pebbles to the left of $B(m,k)$ or to the right of $B(j,k)$, via a recursive call to PEBBLE$(m,j,r,\mu+1)$. By design, the number of vertices between $B(i,k)$ and $B(m,k)$ and the number of vertices between $B(m,k)$ and $B(j,k)$ is (about) one-half of the number of vertices between $B(i,k)$ and $B(j,k)$, so the portions of the graph G treated under these two calls to PEBBLE are (about) one-half the size of the portion treated by the previous call. When these calls to PEBBLE have done their task on the blocks $B(p,k)$ and $B(r,k)$, the algorithm then calls the procedure PEBBLE with arguments (p,r,m,μ). This invocation adds to $B(m,k)$ all pebbles that can be added by a sequence of steps that adds no new pebbles to vertices to the left of $B(p,k)$ or to the right of $B(r,k)$. Again, this region of the graph, between $B(p,k)$ and $B(r,k)$, is (about) one-half the size of the region between $B(i,k)$ and $B(j,k)$. Thus, each of the recursive calls to PEBBLE is for a portion of the graph that is (about) one-half the size of the preceding portion. After the new pebbles are placed on $B(m,k)$, all the $(\mu+1)$-pebbles are removed from the graph.

To see that the procedure invocation PEBBLE $(1, n-k, (n-k)/2, 1)$ results in the pebbling of all of the vertices of $B(m,k)$, where $m=(n-k)/2$, we can consider the following inductive argument. Let v be a vertex in $B(m,k)$ that can be pebbled at some point during a play of the pebble game by a sequence of steps that adds no new pebbles to a vertex in $B(m,k)$. (There may be already several pebbles on the vertices of $B(m,k)$, but during this portion of the play, culminating in a pebble being placed on v, no new pebble is added to a vertex in $B(m,k)$.) This pebble can be placed on v because pebbles have been placed on predecessors of v. By our hypothesis that no new pebbles are added to $B(m,k)$, these predecessors of v lie either to the left or to the right of $B(m,k)$. By an inductive assumption based on the size of the graph, we can assume that each call to procedure PEBBLE to pebble a block B in the middle of the portion of the graph to the left or to the right of $B(m,k)$ results in the pebbling of all of the vertices of B that can be pebbled by a sequence of steps that adds pebbles only to the respective regions. Let p be the index of the first vertex in the block to the left of $B(m,k)$ and let r be the index of the first vertex in the block to the right of $B(m,k)$. Then the call to the procedure PEBBLE with arguments (p,r,m,λ) results in vertex v containing a pebble, since each of the predecessors of vertex v are in this region and, by the inductive assumption can be pebbled by the procedure PEBBLE. Thus, every vertex in $B(m,k)$ is eventually pebbled, since as pebbles are added to $B(m,k)$, each vertex can eventually be pebbled without pebbling any other vertex in $B(m,k)$.

Let $P(t)$ denote the maximum over all blocks of t consecutive vertices in the graph G of the number of pebbles needed to pebble the vertices in that block. If $t \leq 2k$, then the algorithm pebbles the vertices in the block directly; therefore, at most $2k$ pebbles are needed. On the other hand, if $t > 2k$, the algorithm pebbles three half-size portions and leaves pebbles on the two blocks $B(p, k)$ and $B(r, k)$ when pebbling the third portion. Thus, a recurrence relation that bounds the value $P(t)$ is given by:

$$\begin{aligned} &P(t) \leq P(t/2) + 2k, && \text{if } t > 2k, \text{ and} \\ &P(t) \leq t, && \text{if } t \leq 2k. \end{aligned}$$

It follows that $P(t) \leq 2k \log_2 t$. The vertices of $B(\lceil n/2 \rceil, k)$ can be pebbled by this algorithm by calling PEBBLE $(1, n, \lceil n/2 \rceil, 1)$. Any other vertex in the graph can be pebbled after all the vertices in $B(\lceil n/2 \rceil, k)$ are pebbled by using the PEBBLE routine to pebble to the left or right of this block. It follows that $2k \log_2 n$ pebbles suffice to pebble any bandwidth-k n-vertex directed acyclic graph. We have thus proved the following.

Lemma 3: *If an n-vertex dag G has bandwidth k, then it has pebble demand at most* $2k \log_2 n$.

Theorem 3: *If the n-vertex directed acyclic graph G has bandwidth k, then it has pebble demand at most* $\min(2k^2 + k + 1,\ 2k \log_2 n)$.

Proof: The theorem follows directly from Lemmas 2 and 3. □

To place these upper bounds in perspective, we remark that we have no example of a graph of bandwidth k that has pebble demand exceeding $3k - 1$. In Figs. 1 and 2 we illustrate graphs that have bandwidths 2 and 3, respectively, and that require 5 and 8 pebbles, respectively. By extending these constructions we can show that $3k - 1$ pebbles is a lower bound on the pebble demand of a graph of bandwidth k. There is a considerable gap between this lower bound and our upper bound.

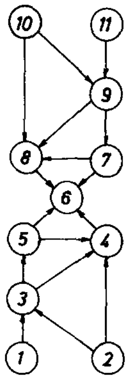


Fig. 1. A graph with bandwidth 2 that requires 5 pebbles

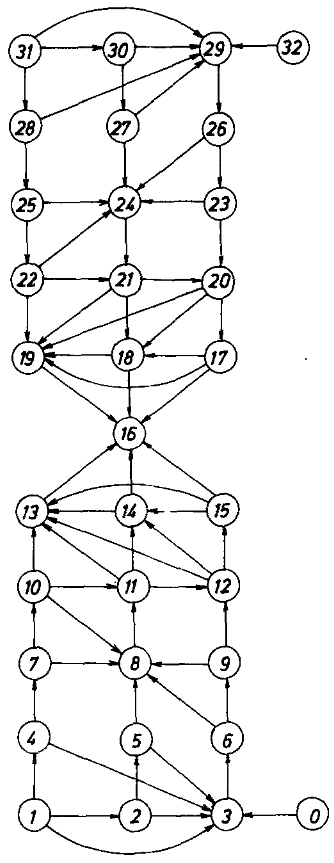


Fig. 2. A graph with bandwidth three that requires 8 pebbles

C. The Difficulty of the Pebble Demand Problem

Since $O(k \log_2 n)$ pebbles suffice to pebble an n-vertex graph of bandwidth k, the problem of deciding whether p pebbles suffice to pebble a graph with bandwidth $f(n)$ can be solved nondeterministically in space $f(n)\log^2 n$: one can "guess" which moves to make in the pebble game and keep track of the positions of the pebbles on the graph inside the worktape space. Since $f(n)\log n$ pebbles are sufficient, and since each pebble's location requires only $\log n$ space to record (in binary notation), it follows that $f(n)\log^2 n$ space is sufficient to solve the Problem. In other words, the *Pebble Demand Problem* restricted to graphs of bandwidth $f(n)$ is in the class NSPACE$(f(n)\log^2 n)$[1].

[1] NSPACE$(s(n))$ is the class of languages accepted by nondeterministic Turing machines that use $O(s(n))$ space on inputs of length n.

The Pebble Demand (PD) Problem:
Input: A directed acyclic graph G and a positive integer k.
Property: A pebble can be placed on a sink-vertex of G using at most k pebbles.

To contrast with this containment, it is known that the unrestricted PD-Problem is complete for PSPACE[2] [5]; and the problem remains PSPACE complete for so-called And/Or graphs [9]. We note that the sliding rule is being used in the pebble games of Section C. It is known that this changes pebble demand by at most one [5].

The And/Or Pebble Demand (PD) Problem:
Input: A directed acyclic graph G, a labelling function *label*: Vertices$(G) \to \{and, or\}$, and a positive integer k.
Property: A pebble can be placed on a sink-vertex of G using at most k pebbles.

The pebble game on an And/Or graph is a natural extension of the pebble game we have considered: if a vertex v of G is labelled *or*, then only one of the predecessors of v needs to contain a pebble in order to place a pebble on v; if a vertex v is labelled *and*, then all predecessors of v must contain a pebble in order to place a pebble on v. Let AND/OR PD$(f(n))$ denote the And/Or PD Problem restricted to And/Or graphs G of bandwidth $f(n)$, where n is the number of vertices in G. That is,

AND/OR PD$(f(n))$
Input: A linear encoding of a directed acyclic graph G, a labelling function *label*: Vertices$(G) \to \{and, or\}$, and a positive integer k.
Property:
(1) The graph G has bandwidth $f(|G|)$, under the layout implicit in the linear input encoding; and
(2) a pebble can be placed on a sink-vertex of G using at most k pebbles.

The algorithm presented previously, to show that $2k \log n$ pebbles suffice to pebble a graph of bandwidth k, also works to show that $2k \log n$ pebbles suffice to pebble an And/Or graph of bandwidth k. Thus, AND/OR PD$(f(n))$ is in NSPACE$(f(n) \log^2 n)$. For a corresponding lower bound we show now that the AND/OR PD$(f(n))$ problem is log space hard for NTISP (poly, $f(n)$)[3].

In order to establish the claimed bound, it will, of course, suffice to reduce a known NTISP (poly, $f(n)$)-complete problem to this pebbling problem. For this purpose we choose the 3-SATISFIABILITY problem restricted to well-formed formulas of bandwidth $f(n)$, denoted by 3-SAT$(f(n))$ [13]:

3-SAT$(f(n))$:
Input: A well-formed formula (wff) $w = C_1 C_2 \dots C_m$ in 3 CNF.

[2] PSPACE is the class of languages accepted by deterministic Turing machines that use space $n^{O(1)}$ on inputs of length n.

[3] NTISP (poly, $f(n)$) is the class of languages accepted by nondeterministic Turing machines that simultaneously use time $n^{O(1)}$ and space $O(f(n))$ on inputs of length n.

Property:

(i) The wff w has bandwidth f: if a positive or negative instance of a variable occurs in clauses C_i and C_j, then $|i-j| \leq f(m)$; and

(ii) the wff w is satisfiable: there is a truth assignment to the variables of w that makes every clause true.

From a wff w, we shall construct an And/Or graph G_w with a goal vertex t and a number of pebbles s such that: w is satisfiable iff t can be pebbled with s pebbles. Furthermore, the And/Or graph G_w will be shown to have a layout λ of bandwidth at most a constant c times the bandwidth of w, where c is a fixed constant independent of the wff w. The reduction from w to an encoding of the And/Or graph G_w under the layout λ can be accomplished in logarithmic space. It will follow that the AND/OR PD problem, when restricted to graphs of bandwidth f, is log space hard for NTISP (poly, $f(n)$).

Theorem 4: *AND/OR PD* $(f(n))$ *is* log *space hard for NTISP* (*poly*, $f(n)$), *for all functions* f *on the natural numbers such that* $f(n) \geq \log n$.

Proof: The reduction we describe is suggested by the earlier reduction from QBF (quantified boolean formulas) to the PEBBLE DEMAND problem described by Gilbert, Lengauer, and Tarjan [5]. The SATISFIABILITY problem is, of course, the restriction of QBF to the special case when all of the quantifiers are existential. The reader should recall that the QBF problem is PSPACE complete and that the 3-SATISFIABILITY problem is NP complete.

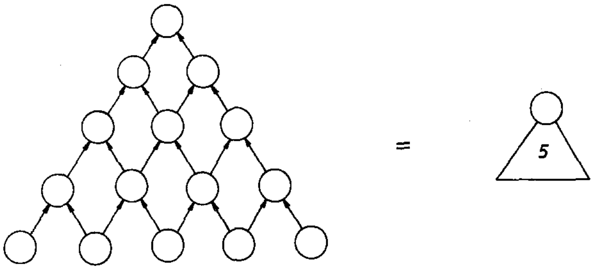


Fig. 3. A 5-Pyramid

An important building block in our construction is the "pyramid" graph shown in Fig. 3, which we abbreviate with a triangle as indicated in the figure. Cook [3] has proved that the sink-vertex (or, *apex*) of a pyramid with k sources (called a k-pyramid) has pebble demand precisely k. This fact makes pyramids useful in pebbling arguments, as is indicated in the following observation from Gilbert, Lengauer, and Tarjan [5]:

> ... A pyramid can be used to lock a pebble on a given vertex for a given time interval. This is done by making the vertex the apex of a pyramid which is so large that in order to repebble the vertex, so many pebbles have to be taken off the graph for use on the pyramid that the results achieved after the vertex was first pebbled are lost ... Note also that if any source of a k-pyramid contains a pebble

that cannot be moved, then the apex can be pebbled with $k-1$ additional pebbles ...

Let $w=C_1 C_2 \ldots C_m$ be a bandwidth-k wff in 3 CNF. Let x be a variable that occurs in w. Let C_i and C_j be, respectively, the first and last clauses of w containing instances of x (either positive or negative). We term the set $[i,j]=_{\text{def}}\{i,i+1,\ldots,j\}$ the *domain* of the variable x, denoted domain (x); and we let first (x) denote the integer i. The variable x is *active* for the clause C_p, if $p \in$ domain (x). There can be at most $3k$ active variables for any clause C_p in w, where $1 \le p \le m$, since w has bandwidth k and w is in 3 CNF. Let $x_1, x_2, \ldots, x_n$ be an enumeration of the variables in w in the following format: the variable y precedes the variable z in the enumeration if either (1) first $(y)<$ first (z) or (2) first $(y)=$ first (z) and the (positive or negative) instance of y in $C_{\text{first}(y)}$ precedes the instance of z in that clause. Then, for any $i \ge 1$, define the *successor* of the variable x_i, denoted SUCCESSOR (x_i), by:

SUCCESSOR $(x_i)=x_j$ where j is the largest number $>i$ such that, for all k $(i<k<j)$, either

(a) domain $(x_i) \cap$ domain $(x_k) \neq \emptyset$, or

(b) for some $p<i$, SUCCESSOR $(x_p)=x_k$.

The And/Or graph G_w constructed has $m+p$ blocks of vertices, where p is the maximum number of variables that are active for any clause in w; we have already observed that p is at most $3k$. In other words, there is one block of vertices for each clause and one for each of the variables that are active in such clauses. The part of a block corresponding to a variable x includes four vertices, as shown in Fig. 4. Two pebbles placed on this subgraph encode the truth value of x and $\bar{x}$ as illustrated in Fig. 4b–d.

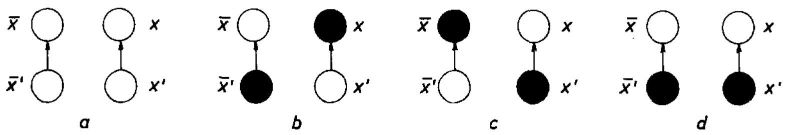


Fig. 4. *a* vertices representing a variable; *b* true configuration; *c* false configuration; *d* double false configuration

Fig. 5 illustrates how subgraphs corresponding to separate variables x, $y=$ SUCCESSOR (x), and $z=$ SUCCESSOR (y) are connected. This block works as follows. Two pebbles are placed on the subgraph corresponding to the variable x to represent a truth assignment. These two pebbles remain stationary while pebbles are being placed on vertices corresponding to the clauses C_p for which x is active. When x is no longer active, the two pebbles are moved upward to the portion of the graph corresponding to the variable y. Since y becomes active only after x stops being active, the truth assignment for x is no longer required at the time the two pebbles are shifted upwards. The role of the vertex D_x is to force at least one of the pebbles placed on the portion of the graph corresponding to variable x to remain on either x'

or $\bar{x}'$ during the time period that x is active: if the pebbles were to be moved so that both x and $\bar{x}$ contain pebbles, then these two pebbles could not be moved further up this block of the graph without repebbling one of the pyramid graphs we shall describe presently. On the other hand, if either $\bar{x}'$ or x' contains a pebble when the time period for the variable x to be active is over, then D_x can be pebbled (since it is an "or" vertex) and the two pebbles can be moved up to $\bar{y}'$ and y'.

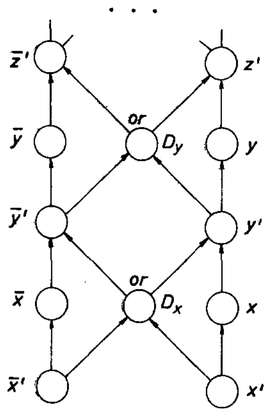


Fig. 5. Portion of block corresponding to variables x, y, and z such that $y = \text{SUCCESSOR}(x)$ and $z = \text{SUCCESSOR}(y)$

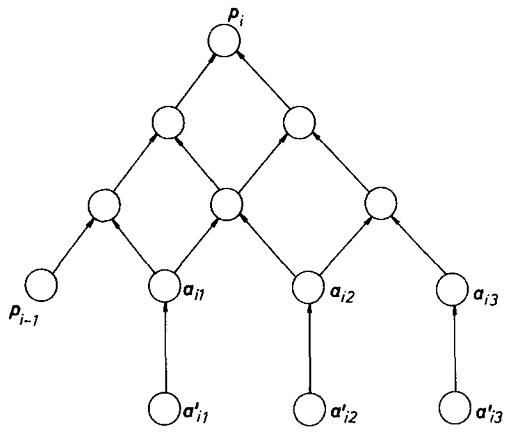


Fig. 6. Block of vertices for clause $a_{i1} + a_{i2} + a_{i3}$. Note that the vertices a_{ik} and a'_{ik} occur among the variable blocks. Vertex p_{i-1} is part of the $i-1$-st clause block; p_0 is the vertex b, which has no predecessors

Fig. 6 illustrates the block of vertices C'_i corresponding to a clause $C_i = (a_{i1} + a_{i2} + a_{i3})$. After some $s-3$ pebbles are allocated to the blocks corresponding to the variables, the remaining three pebbles are available to pebble the clause blocks. For each literal a_{ip}, $1 \le p \le 3$, there is a fixed pebble on vertex a_{ip} if the literal is true, or on vertex a'_{ip} if the literal is false. Thus, if $w = C_1 C_2 \dots C_m$ is satisfied, then the clause pyramids can be pebbled in the order $C'_1, C'_2, \dots, C'_m$; however, if some clause $(a_{i1} + a_{i2} + a_{i3})$ is false, then p_i is the apex of an empty 4-pyramid and cannot be pebbled with three pebbles.

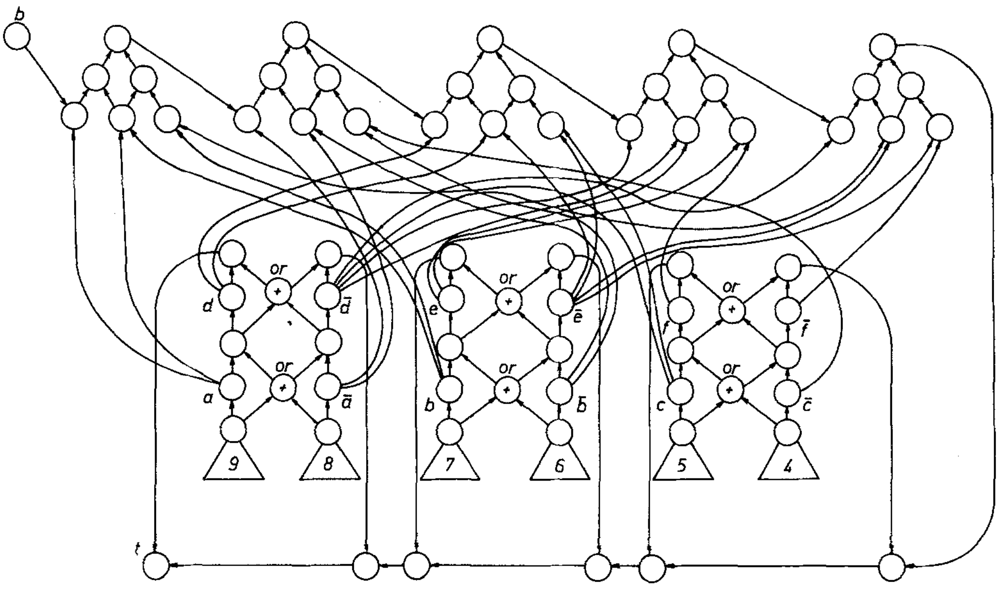


Fig. 7. Graph for $w = (a+b+c)(\bar{a}+\bar{b}+\bar{c})(d+\bar{e}+c)(\bar{d}+e+f)(\bar{d}+\bar{e}+\bar{f})$. Number of pebbles: $2p+3=9$

Fig. 7 illustrates the entire construction. Note that b is a vertex with no predecessors and that t is the goal vertex. Our goal now is to show that there exists a layout λ of the And/Or graph G_w such that the bandwidth is at most ck, where k is the bandwidth of w and c is a fixed constant.

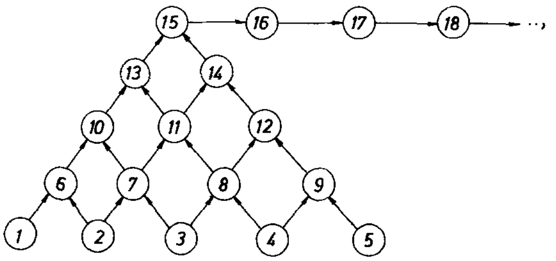


Fig. 8. A layout of a 5-pyramid with bandwidth five

We observe first that a k-pyramid enjoys a layout of bandwidth k: assign the smallest integers to the pyramid's source vertices, the next smallest integers to the source vertices' successors, and so on, level by level; see Fig. 8. It should, perhaps, be noted that five pebbles are necessary and sufficient to place a pebble, not only on the apex of the 5-pyramid, but also on the chain of vertices 16, 17, 18, ... shown in Fig. 8. Thus, a graph consisting of a k-pyramid, a separate $(k-1)$-pyramid, a separate $(k-2)$-pyramid, ... can be laid out with bandwidth k so that consecutive vertices are connected by separate paths to the apexes of the pyramids. This is illustrated for a 5-pyramid, 4-pyramid, and a 3-pyramid in Fig. 9.

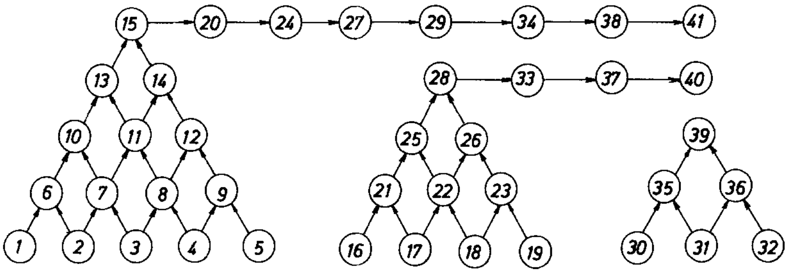


Fig. 9. A layout of a 5-pyramid, a 4-pyramid, and a 3-pyramid with bandwidth five so that vertices connected to the apexes of the pyramids are consecutive vertices

As previously stated, there are at most $3k$ active variables for any clause of a wff w with bandwidth k in 3 CNF. Thus, there are at most $6k$ pyramids necessary for the blocks of vertices corresponding to variables. The largest pyramid is of size $6k+3$. As we have seen, all of the pyramids of size $6k+3, 6k+2, \ldots, 4$ can be laid out with bandwidth $6k+3$ so that there are $6k$ consecutive vertices that are connected to the apexes of these pyramids. Furthermore, we lay out the vertices corresponding to the clauses in the same order as the clauses are presented in the wff w, so that all vertices corresponding to clause C_i receive smaller integers than those corresponding to clause C_{i+1}. There are six vertices in each block corresponding to a clause; cf. Fig. 6. Assign the vertices corresponding to each variable x to integers between those assigned to clauses C_i and C_{i+1}, where $i = \text{first}(x)$. There are five vertices corresponding to a variable x; cf. Fig. 5. Thus, there are at most 21 vertices in the block corresponding to a clause C and each of the blocks corresponding to the variables that occur in that clause. The only edges connecting a vertex corresponding to a variable x and a vertex corresponding to a clause C occur when an instance (positive or negative) of the variable x appears in the clause C. Therefore, since the wff w has bandwidth k, no edge of G_w connects vertices that are laid out at a distance more than $21k$ apart. In particular, a variable-x vertex is assigned an integer between those assigned to clause blocks C'_i and C'_{i+1} and is connected by an edge directed to a clause-C_j vertex only if a positive or negative instance of the variable x occurs in both clauses C_i and C_j. It follows that $|i-j| \leq k$ and, so, since each clause is represented by at most 21 vertices, the edge in the And/Or graph G_w connects vertices assigned integers at most $21k$ apart. □

4. BF Pebbling and Bandwidth

Our second pebble game, the BF (for *breadth first*) pebble game is played on a connected undirected graph. Its rules are as follows.

1. No vertex of the "input" graph G ever holds more than one pebble.
2. No vertex of G is ever pebbled more than once.
3. The moves of the game are as follows.
   Move 0. Place some (at least one) pebbles on the vertices of G.
   Move $i>0$. If no pebbles remain on G, then halt; the game is over. Otherwise, remove some one pebble from G; and place pebbles on some never-before-pebbled vertices of G, *including at least all those neighbors of the just depebbled vertex that have never been pebbled before.*

In a play of the BF pebble game, if a pebble is deposited on a vertex at Move i and is removed from the vertex at Move j, then the *lifetime* of the pebble in that play of the game is the integer $j-i$. (We assume that a pebble is discarded when it is removed from the graph.)

The graph G is *k-BF-pebbleable*, $k \in N$, if there is a play of the BF pebble game on G in which no pebble has lifetime exceeding k.

Theorem 5: *The graph G is k-BF-pebbleable if and only if it has bandwidth at most k.*

Proof: Say first that G has bandwidth k. Then there is a layout

$$\lambda\colon \text{Vertices}(G) \to \{1, 2, \ldots |G|\}$$

such that $|\lambda(v)-\lambda(w)| \leq k$ for all adjacent vertices v and w of G. Consider, therefore, the following BF pebbling strategy for G.

Move 0. Pebble vertices $\lambda^{-1}(1)$, $\lambda^{-1}(2), \ldots, \lambda^{-1}(k)$ of G.
Move $1 \leq i \leq |G|-k$.
(a) Remove the pebble from vertex $\lambda^{-1}(i)$;
(b) place a pebble on vertex $\lambda^{-1}(i+k)$;
Move $|G|-k<i\leq |G|$. Remove the pebble from vertex $\lambda^{-1}(i)$.

Since λ is a bandwidth-k layout of G in the line, all successors v of vertex $\lambda^{-1}(i)$ have $\lambda(v) \leq i+k$. Thus Move i (b) is a legitimate one. A simple induction now proves that the indicated strategy is a *k-BF*-pebbling of G.

Conversely, if the graph G is k-BF-pebbleable, then there is a $(|G|+1)$-move BF pebble game witnessing this fact. With this play of the game in mind, define the injection λ from G into the line by:

$$\lambda(v) = \text{the move at which vertex } v \text{ is depebbled.}$$

That λ is well-defined and one-to-one is a direct consequence of the game rules: every vertex of G is pebbled and depebbled exactly once. That λ is a bandwidth-k layout of G is also easy to see. Note that in a k-BF-pebbling of G, if the first end of an edge of G is pebbled at Move i and depebbled at Move $j \leq i+k$, then it must be that the second end of the edge is pebbled at Move $r \in \{i, \ldots, j\}$ and depebbled at Move $m \leq r+k$; hence, $|m-j| \leq k$. But j is the image of the first end under λ, and m is the image of the second end. Since we have been considering an arbitrary edge of G, it follows that λ is a bandwidth-($\leq k$) layout of G, as was claimed. □

We thus have our second characterization of bandwidth in terms of pebbling.

5. Applications of the Bandwidth-Pebbling Relationships

It is always useful to have a number of different views of an important concept, as each view tends to bring with it various tools and insights. We cite now three simple examples of the benefits of the insights afforded by the results of this paper.

The first corollary establishes the difficulty of finding optimal plays of either the NRA or BF pebble games.

Corollary 1: *Given a graph G and an integer K, the problems of deciding whether or not G is k-BF-pebbleable or k-NRA-pebbleable for some $k<K$ are NP-complete. This remains true even if G is restricted to be a degree-3 free tree. If the integer K is fixed (so that the solutions need not be uniform in K), then the problems can be solved in time $O(|G|^K)$.*

Proof: In the presence of Theorems 1 and 4, the three assertions in the statement of the corollary follow, respectively, from results in [14], [4], and [12, 20] concerning the difficulty of the bandwidth problem. □

Corollary 1 is somewhat interesting in that it shows the complexity of the "lifetime" (L) problem for the NRA pebble game to be quite different from the "pebble demand" (PD) problem. Specifically, both of the games are NP-complete in general (the L result being Corollary 1, and the PD result being proved in [21]). In contrast, if one restricts attention to trees, then the L problem is likely to be harder than the PD problem, since the former problem remains NP-complete while the latter is solvable in polynomial time [16, sect. 8].

The second corollary establishes the inherent difficulty of pebbling reticulated graphs.

The graph G is *n-reticulated* ($n \in N$) if G's vertex-set contains two disjoint n-element subsets A and B such that, given any $k \leq n$ vertices from A and k vertices from B, there is a set of k *edge-disjoint paths* in G connecting the A-vertices with the B-vertices.

Remarks: (1) Any homeomorph of the side-n grid is $(n-1)$-reticulated: one could choose the first elements of the rows and columns, respectively (ignoring their common element), as the sets A and B. (2) By similar reasoning, any homeomorph of the side-n pyramid is $n/2$-reticulated.

Corollary 2: *The lifetime-cost of any play of either the NRA pebble game or the BF pebble game on an n-reticulated graph is $\Omega(n^{1/2})$.*

Proof: The result will follow from Theorems 1 and 4 once we show that the bandwidth of any n-reticulated graph is $\Omega(n^{1/2})$.

Let G be n-reticulated. Consider any layout of G in the line. There must be some point p on the line such that at least $n/2$ elements of the set A lie on one side of p and at least $n/2$ elements of the set B lie on the other side of p. By G's n-reticulation, there are at least $n/2$ edge-disjoint paths crossing p, hence at least this many distinct edges crossing p (where "crossing p" means having one end to the left of p and the other end to the right of p). Simple reasoning verifies that these edges must have at least $(n/2)^{1/2}$ distinct endpoints on one side of p or the other. It follows that the endpoints of at least one of these crossing edges must be distance $\Omega(n^{1/2})$ apart in the line. □

The reader can easily verify (via examples like the side-n pyramid) that Corollary 2 cannot be strengthened in general.

The third corollary establishes a criterion for a graph family to have unbounded bandwidth.

The depth-d *computation tree* is the dag obtained from the depth-d complete binary tree by directing all edges toward the root of the tree. A family of graphs Γ is *deep* if for each integer $k \in N$, some graph G in Γ contains as a subgraph some expansion of a computation tree of depth $d \geq k$.

Corollary 3: *If the bandwidth-$f(n)$ family of graphs Γ is deep, then*

$$\limsup_{n \to \infty} f(n) = \infty .$$

Proof: The result follows from Theorem 3, once one notes that the proof in [15] that depth-d computation trees have pebble demand $\Omega(d)$ holds also for any expansions of these trees. □

It should also be noted that the results in Subsection 3 C indicate the effects of placing a restriction on the bandwidths of graphs in the and/or pebble demand problem. Specifically, we showed that, when restricted to graphs of bandwidth $f(n)$, this problem is in NSPACE $(f(n)\log^2 n)$ and is log space hard for the class NTISP (poly, $f(n)$). This contrasts with the results of Monien and Sudborough [13] which show that several NP-complete problems become log space complete for NTISP (poly, $f(n)$) when restricted to graphs of bandwidth $f(n)$. It remains open whether AND/OR PD$(f(n))$ can be solved in less than $f(n)\log^2 n$ space. One avenue to a resolution of this problem would be a result indicating that $O(f(n))$ pebbles are sufficient to pebble any bandwidth-$f(n)$ and/or graph, for this would yield the corollary that $f(n)\log n$ space is sufficient for AND/OR PD $(f(n))$; but this result has eluded us. Another unresolved question is whether the general pebble demand problem (as opposed to the and/or version) is log space hard for NTISP (poly, $f(n)$) when restricted to bandwidth-$f(n)$ graphs.

APPENDIX: Details of the proof of Lemma 1.

In the proof of Lemma 1, we constructed a graph G' which is an expansion of G, and which we claimed to be k-NRA-pebbleable. We now verify this claim.

Note first that, by keeping track of the process by which G was expanded, one can identify each vertex v in G with a directed chain in G' of the form

$$v \to w_1 \to \ldots \to w_n;$$

vertices preceding v in G (i.e., with edges entering v) correspond to chains preceding v in G', and vertices preceded by v in G correspond to vertices preceded by some vertex in v's chain in G'. Let us color the vertices of G' so that the initial vertices in our chains are blue, the terminal vertices in our chains are red, and all other vertices are white. In a natural and unambiguous way, we can talk about the blue vertex and the red vertex in G' corresponding to a given vertex in G.

Consider the play of the NRA pebble game on G that was used in constructing G'. Let us play the game on G' by taking cues from that play of the game on G, as follows. The play on G begins with the pebbling of some vertex v in G. Let the play on G' begin with the pebbling of the blue vertex corresponding to v in G'. Now, imagine that the game on G is being played with pebbles colored $1, \ldots, \mathrm{k}+1$, as described earlier. As we did then, let us replace each move in the play on G by a polling of the status of the pebbles of the various colors on G, together with new moves reflecting the status; this is essentially a repeat of Stage 2 in the construction of G'. Each move comprises polling the pebbles of colors $1, \ldots, k+1$ (at most k of which will be on the graph, of course). We shall maintain the following inductive condition.

If a pebble of color i is to be placed on (resp., removed from) a vertex v of the graph G at any move, then at the corresponding polling of color i on G', a pebble can be placed on the blue (resp., removed from the red) vertex corresponding to v in G'.

Clearly, the inductive situation holds at the beginning of play. Assume that we are at a given move on G; let us analyze the corresponding move on G'. We poll the various colors. If a pebble of color i is either placed on or removed from G at this move, then by assumption, we can mimic this move on G' using (respectively) a blue or a red vertex. If no pebble of color i would be moved (i.e., placed or removed) on G at this move, and if no pebble of color i resides on G currently, then color i is ignored at this polling. If no pebble of color i would be moved on G at this move, but there is a pebble of color i sitting on vertex v of G, then the pebble residing (by inductive assumption) on the chain in G' corresponding to v is "advanced" along its chain – its successor in the chain is pebbled, and it is depebbled. We must verify (a) that this advance is a legal move and (b) that our inductive hypothesis is maintained in the face of this advance. (The verification of (b) simultaneously verifies the maintenance of the inductive hypothesis when a pebble is placed on a blue vertex.)

Regarding condition (a), recall that when we constructed G' by means of a sequence of selective edge expansions from G, we performed the construction in much the way that we are playing the game on G' now: we played the game on G with colored pebbles, and we expanded G every time a pebble threatened to sit for too long. Well, it was exactly in the situation where we now advance pebble i along its chain that we lengthened the chain (and advanced the pebble) in the construction process; hence we are sure that when we try to advance the pebble, there is a chain-vertex to advance it to. Second, each vertex in a chain has (by definition) a unique predecessor; hence, the placing of the pebble that is the first half of advancing is legal. Third, when G was expanded at this vertex in the course of constructing G', the only outedges that were left incident to the current vertex were those that had been pebbled earlier in the play; hence, in G', all of the successors of the current vertex must have been pebbled at some earlier time in the play (the chain-successor being the last of these to be pebbled) and so it is legal to remove the pebble from this vertex as the second half of advancing. In summary, our advance along the chain was a legal move.

Regarding condition (b), note that when G was expanded at this vertex in the course of constructing G', all outedges of this vertex that led to virgin vertices were taken along as outedges of the vertex that is the chain-extension of the current vertex; and when G was further expanded at the chain-extension vertex, only still-virgin outedges were taken along in the further expansion. This means that the only outedges incident to the chain-extension of the current vertex are outedges leading to vertices that are pebbled while the chain-extension holds its pebble. It follows that advancing this pebble along its chain will not jeopardize any subsequent pebble placements or pebble removals. In other words, the inductive situation is maintained.

We see now that our play of the NRA pebble game on G' is a valid one that uses at most $k+1$ pebbles at any time. Moreover, because of the way we derive our play of the game on G' from our original play on G (i.e., by replacing moves on G by pollings-plus-moves on G'), we are assured that no pebble resides on G' for more than k type B moves. In other words, we have shown that the graph G' is k-NRA-pebbleable. $\square$

Acknowledgments

It is a pleasure to thank B. Monien, A. Sherman, S. Berkowitz, and J. Peters for valuable and stimulating conversations about bandwidth and pebble games, and L. Stockmeyer and an anonymous referee for valuable suggestions that improved the presentation.

References

[1] Aykuz, F. A., Tuku, S.: An automatic node-relabelling scheme for bandwidth minimization of stiffness matrices. Amer. Inst. of Aero. and Astro. J. *6*, 728 – 730 (1968).

[2] Chen, K. Y.: Minimizing the bandwidth of sparse symmetric matrices. Computing *11*, 27 – 30 (1973).

[3] Cook, S. A.: An observation on time-space tradeoffs. J. Comp. Syst. Sci. *9*, 308 – 316 (1974).

[4] Garey, M. R., Graham, R. L., Johnson, D. S., Knuth, D. E.: Complexity results for bandwidth minimization. SIAM J. Appl. Math. *34*, 477 – 495 (1978).

[5] Gilbert, J. R., Lengauer, T., Tarjan, R. E.: The pebbling problem is complete in polynomial space. SIAM J. Comput. *9*, 513 – 524 (1980).

[6] Hopcroft, J. E., Paul, W., Valiant, L. G.: On time versus space. J. Assoc. Comput. Mach. *24*, 332 – 337 (1977).

[7] Kung, H. T., Stevenson, D.: A software technique for reducing the routing time on a parallel computer with a fixed interconnection network. In: High Speed Computer and Algorithm Optimization, pp. 423 – 433. New York: Academic Press 1977.

[8] Lengauer, T.: Relationships between pebble games on directed and undirected graphs. Typescript, 1980.

[9] Lingas, A.: A *P*-space complete problem related to a pebble game. Lecture Notes in Computer Science *62*, pp. 300 – 321. Berlin-Heidelberg-New York: Springer 1978.

[10] Lipton, R. J., Eisenstat, S. C., DeMillo, R. A.: Space and time hierarchies for classes of control structures and data structures. J. ACM *23*, 720 – 732 (1976).

[11] Meyer auf der Heide, F.: A comparison of two variations of a pebble game on graphs. Theor. Comp. Sci. *13*, 315–322 (1981).
[12] Monien, B., Sudborough, I. H.: Bandwidth problems in graphs. Proc. 1980 Allerton Conf. on Communication, Control, and Computing *1980*, 650–659.
[13] Monien, B., Sudborough, I. H.: Bandwidth constrained NP-complete problems. Proc. 1981 ACM Symp. on Theory of Computing, Milwaukee, Wisc., pp. 207–217.
[14] Papadimitriou, Ch. H.: The NP-completeness of the bandwidth minimization problem. Computing *16*, 263–270 (1976).
[15] Paterson, M. S., Hewitt, C. E.: Comparative schematology. Proc. Proj. MAC Conf. on Concurrent Systems and Parallel Computation, 1970, pp. 119–127.
[16] Pippenger, N.: Pebbling. In: Proc. 5th IBM Symp. on Mathematical Foundations of Computer Science, 1980.
[17] Rose, D. J.: A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In: Graph Theory and Computing (Read, R., ed.), pp. 183–217. New York: Academic Press 1972.
[18] Rosenberg, A. L.: Data encodings and their costs. Acta Inform. *9*, 273–292 (1978).
[19] Rosenberg, A. L., Snyder, L.: Bounds on the costs of data encodings. Math. Syst. Th. *12*, 9–39 (1978).
[20] Saxe, J. B.: Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time. Carnegie-Mellon Tech. Rpt. CMU-CS-80-102, 1980.
[21] Sethi, R.: Complete register allocation problems. SIAM J. Comput. *4*, 226–248 (1975).
[22] Sudborough, I. H.: Pebbling and bandwidth. In: Fundamentals of Computation Theory (Lecture Notes in Computer Science, Vol. 117), pp. 373–383. Berlin-Heidelberg-New York: Springer 1981.

A. L. Rosenberg
Department of Computer Science
Duke University
Durham, NC 27706, U.S.A.

I. H. Sudborough
Department of Electrical Engineering
and Computer Science
Northwestern University
Evanston, IL 60201, U.S.A.