Discrete Applied Mathematics 262 (2019) 72–82

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Graph pebbling algorithms and Lemke graphs

Charles A. Cusack [a,b,*], Aaron Green [a], Airat Bekmetjev [b], Mark Powers [a]

[a] Department of Computer Science, Hope College, 27 Graves Place, Holland, MI 49422, United States
[b] Department of Mathematics, Hope College, 27 Graves Place, Holland, MI 49422, United States

ARTICLE INFO

Article history:
Received 20 September 2017
Received in revised form 14 February 2019
Accepted 17 February 2019
Available online 13 March 2019

Keywords:
Graph pebbling
Algorithms
Lemke graph

ABSTRACT

Given a simple, connected graph, a *pebbling configuration* (or just *configuration*) is a function from its vertex set to the nonnegative integers. A *pebbling move* between adjacent vertices removes two pebbles from one vertex and adds one pebble to the other. A vertex r is said to be *reachable* from a configuration if there exists a sequence of pebbling moves that places at least one pebble on r. A configuration is *solvable* if every vertex is reachable. The *pebbling number* $\pi(G)$ of a graph G is the minimum integer such that every configuration of size $\pi(G)$ on G is solvable. A graph G is said to satisfy the *two-pebbling property* if for any configuration with more than $2\pi(G)-q$ pebbles, where q is the number of vertices with pebbles, two pebbles can be moved to any vertex of G. A Lemke graph is a graph that does not satisfy the two-pebbling property. In this paper we present a new algorithm to determine if a vertex is reachable with a given configuration and if a configuration on a graph is solvable. We also discuss straightforward algorithms to compute the pebbling number and to determine whether or not a graph has the two-pebbling property. Finally, we use these algorithms to determine all Lemke graphs on at most 9 vertices, finding many previously unknown Lemke graphs.

1. Introduction

Let $G = (V, E)$ be a connected graph with vertex set V and edge set E. A *pebbling configuration* (or just *configuration*) is a function $C : V(G) \to \mathbb{N} \cup \{0\}$ where $C(v)$ represents the number of pebbles placed on vertex v. The *size* of C, denoted $|C|$, is the sum of the number of pebbles on the vertices of G. A *pebbling move* from a vertex u to an adjacent vertex v, denoted (u, v), removes two pebbles from u and adds one pebble to v. A pebbling move (u, v) is *legal* if u has at least two pebbles on it. A *pebbling sequence* is just a sequence of legal pebbling moves. A vertex is *reachable* if it has at least one pebble assigned to it in the initial configuration C or it can receive a pebble through a sequence of legal pebbling moves. If vertex r is reachable, C is called *r-solvable*. If every vertex in G is reachable, then C is said to be *solvable*. The *pebbling number* of G, $\pi(G)$, is the minimum integer such that every configuration of $\pi(G)$ pebbles on G is solvable.

Denote the Cartesian product of graphs G and H by $G\square H$. The following conjecture, known as Graham's conjecture, has been of much interest since it first appeared in [7].

Conjecture 1. $\pi(G\square H) \leq \pi(G)\pi(H)$.

A graph G is said to satisfy the *two-pebbling property* if for any configuration with more than $2\pi(G)-q$ pebbles, where q is the number of vertices with pebbles, two pebbles can be moved to any vertex of G. A *violating configuration* is a

* Corresponding author at: Department of Computer Science, Hope College, 27 Graves Place, Holland, MI 49422, United States.
E-mail addresses: cusack@hope.edu (C.A. Cusack), bekmetjev@hope.edu (A. Bekmetjev).

https://doi.org/10.1016/j.dam.2019.02.028
0166-218X/

configuration of more than $2\pi(G)-q$ pebbles on q vertices of G such that some vertex is not reachable with two pebbles. A *minimally violating configuration* is a violating configuration on G with exactly $2\pi(G)-q+1$ pebbles on q vertices.

Graphs that do not have the two-pebbling property are called *Lemke graphs*. Although many results exist about when Graham's conjecture holds ([20,23,24] to name just a few), very few results involve Cartesian products of Lemke graphs. Thus, Lemke graphs are important because they are thought to be the most likely counterexamples to Graham's conjecture if it is false. Recently, Gao and Yin [13] showed that Graham's conjecture holds for the Cartesian product of a tree or complete graph with members of a certain family of Lemke graphs [12].

A pebbling sequence is *executable* on a pebbling configuration if the moves can be legally performed in order. A sequence of pebbling moves that accomplishes some goal (e.g. places one pebble on a specified vertex) is *minimal* if removing any moves renders the goal unattainable, even if reordering the moves is allowed. For instance, if a minimal sequence that places a pebble on r contains the move (u, v), then either $r = v$, or it is impossible to reach r without the move (v, w) for some w and that move is impossible without the pebble that came from u. If a pebbling sequence accomplishes some goal, then some minimal sequence does as well. We assume throughout that all pebbling sequences are minimal. In particular, this implies that there are no cycles of pebbling moves [19].

It has previously been shown that problems Reachable (determining if a configuration is r-solvable for a given root r) and Solvable (determining if a configuration is solvable) are both NP-complete [16,19,25], even for graphs with diameter two [11] and for planar graphs [17]. However, polynomial time algorithms have been developed for diameter two graphs with pebbling number $n+1$ (class 1), those with pebbling number n (class 0) that have small connectivity [4], planar graphs with diameter two [17], and outerplanar graphs [17].

Determining if the pebbling number of a graph is at most k (Pebbling-Number) is Π_2^p-complete [19]. The pebbling number of a diameter two graph is either n or $n+1$, and [5,8] imply that determining which it is can be accomplished in polynomial time. An explicit algorithm is given in [4], with a slight improvement given in [14]. Techniques have been developed to compute the pebbling number for small graphs [22] and bounds on the pebbling number [15]. In addition, the pebbling number can be computed in polynomial time for split graphs [1], 2-paths [2], and semi-2-trees [3].

In this paper we present a new algorithm for Solvable and Reachable. We also discuss relatively straightforward algorithms for computing the pebbling number of a graph and determining whether or not a graph has the two-pebbling property, both of which make heavy use of the Solvable and Reachable algorithm. Since all of these problems are NP-complete or harder, it should not be surprising that our algorithms are not polynomial time. However, we demonstrate their utility by finding all Lemke graphs with at most 9 vertices.

2. Solvability and reachability algorithms

Various techniques for determining solvability of a configuration of pebbles on a graph exist, including a relatively simple backtracking algorithm. We implemented one such algorithm that simply makes the next possible move that does not pebble around a cycle, backtracking (by undoing a move) when no more moves are possible from a given configuration and trying the next possible move and continuing. As expected, the performance degrades very quickly as the size of the graph and/or configuration increases. We describe below an algorithm that performs significantly faster on most graphs and configurations of pebbles. But first we describe four algorithms based on simple heuristics—three that try to very quickly determine if a configuration on a graph is solvable and one that tries to determine if it is unsolvable. Each of these can be implemented to solve either Reachable or Solvable with minor changes. Each algorithm takes as input a graph G and configuration C.

2.1. Simple heuristic algorithms

IsSolvableDistance. The first algorithm does the most naive thing possible: For every vertex v with at least 2^k pebbles, mark every vertex in the graph of distance k or less from v as reachable. If all vertices are marked reachable (vertex r is reachable), then the configuration is clearly **solvable** (r-**solvable**). Otherwise the solvability remains **unknown** since there are many possible moves that were not attempted.

IsSolvableShortestPath. The second algorithm uses a shortest-path algorithm and simply makes all possible moves along the shortest-path tree toward the desired root. For Reachable, this can be done using breadth-first search. For Solvable, the Floyd–Warshall algorithm can be used to find the all-pairs shortest paths and moves can be made appropriately for each root. In either case, the algorithm will answer **solvable** (r-**solvable**) if at least one pebble can be moved to each root (to vertex r) and **unknown** otherwise since, as above, not all possible moves were attempted.

IsSolvableShortestPebblePath. The third algorithm is based on the previous algorithm but has one very slight, but important, modification. If there are several shortest paths, one that has pebbles on it is probably more likely to help us get to the root. Thus, this algorithm is almost identical to the standard shortest path algorithm except that the distance from the root to a given node is the number of edges minus the number of vertices with pebbles on them. The only change in the algorithm is that when processing the list of neighbors of a vertex v, the distance of an unprocessed neighbor u is one greater than the distance to v if u has no pebbles and is the same if u has at least one pebble. It has the same possible outputs as the previous algorithm.

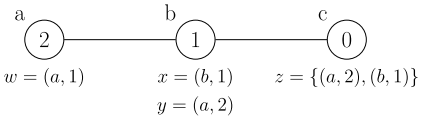


Fig. 1. A graph with 2 pebbles on a and one pebble on b. The pure pebble $(a, 2)$ can be moved to b and combined with the pure pebble $(b, 1)$ to form the composite pebble $\{(a, 2), (b, 1)\}$ which can be placed on c.

IsUnSolvableWeightFunction. The fourth algorithm uses a weight function to determine if a configuration is unsolvable (or not r-solvable). Given a root vertex r, define $w_r(v) = 2^{-dist(v,r)}$ for each vertex v, and for a given configuration C, define $w_r(C) = \sum_{v \in V(G)} C(v)w_r(v)$. Our algorithm is based on the fact that if $w_r(C) < 1$, then C is not r-solvable [6,15,20]. Thus, the algorithm returns **not r-solvable** (**unsolvable**) if $w_r(C) < 1$ ($w_r(C) < 1$ for any vertex r). Otherwise the r-solvability (solvability) is **unknown**.

2.2. Solvability algorithm

The main solvability algorithm, IsSolvable, (see Algorithm 1) executes each of the above simple heuristic algorithms, stopping if the solvability has been established. We assume that each algorithm is implemented to return either **true** or **false** instead of **solvable**, **unsolvable**, or **unknown**, where an answer of **false** is understood to mean **unknown**. If all of these algorithms return **unknown**, then IsSolvableMergePebbles (to be described in the next section) is used.

Algorithm 1 The IsSolvable algorithm. Determines whether or not graph G is solvable under configuration C.

```
function IsSolvable(G, C)
    if IsSolvableDistance(G, C) then return true
    if IsSolvableShortestPath(G, C) then return true
    if IsSolvableShortestPebblePath(G, C) then return true
    if IsUnSolvableWeightFunction(G, C) then return false
    return IsSolvableMergePebbles(G, C)
```

2.3. Merge pebbles algorithm

Before we can describe IsSolvableMergePebbles, we need to develop some terminology and notation. Label each pebble with its *source* — that is, the vertex where it was before any moves were made. A *merged pebble* is a pebble that has kept track of the sources of all the pebbles used to get it to the vertex it currently resides on. The idea is that instead of throwing away pebbles with each move, they are "merged" together (thus the name). A *pure pebble* is a merged pebble whose pebbles all came from the same source. It is represented as a pair (v, m), where v is the source vertex and $m \in \mathbb{Z}^+$, the *mass*, is the number of pebbles from v that were required to get the pebble to the current vertex. A *composite pebble* is a merged pebble that is not a pure pebble. In other words, it has pebbles from at least two sources and is represented by a set of pure pebbles that we refer to as its *signature*. In Fig. 1, the pure pebble $w = (a, 1)$ can be placed on a since a already has pebbles on it (so it takes one pebble from a to place a pebble on a). Likewise, $x = (b, 1)$ can be placed on b. The pure pebble $y = (a, 2)$ can be placed on b because it takes 2 pebbles from a to move a pebble to b. Because x and y are both on b, they can be merged to form the composite pebble $z = \{(a, 2), (b, 1)\}$ which can be placed on c.

If s is a merged pebble, denote the mass from source v by s_v. So in the example above, $z_a = 2$ and $z_b = 1$. The *merger* of s and t, which we denote by $s + t$, is the merged pebble with signature $\{(v, s_v + t_v) \mid v \in V\}$, where the vertices with $s_v + t_v = 0$ are omitted. From our example above, it is evident that $z = x + y$. A merged pebble s is said to be *legal* for a given vertex v if there is a sequence of pebbling moves that can place s on v. So y is legal for b and z is legal for a and c (although as we will see, there is no reason to place z on a). Note that a necessary condition for a pebble s to be legal is that $s_v \leq C(v)$ for all v. Two legal merged pebbles s and t on a vertex v are called *compatible* if $s + t$ is legal for the neighbors of v. Continuing our example, x and y are both legal for b and are compatible since $x + y = z$ is legal for any neighbor of b.

The *merge pebbles algorithm* maintains a list of legal merged pebbles for each vertex. The algorithm has two phases—the *distribute phase* followed by the *merge phase*. Algorithm 2 presents an overview of the algorithm. We will first discuss a simplified version of the algorithm, and then describe two important optimizations.

In the distribute phase we determine, for each vertex s, which other vertices can receive a pebble from s and at what cost. In other words, pure pebbles are distributed to every vertex reachable using only pebbles from each individual vertex. If $C(s) \geq 1$, s can reach itself, so $(s, 1)$ is placed on s; if $C(s) \geq 2$, $(s, 2)$ is placed on all neighbors of s; and in general, if $2^d \leq C(s) \leq 2^{d+1} - 1$, we place $(s, 2^i)$ on every vertex of distance i away from s for $i \in \{0, \ldots, d\}$. At the end of this phase each vertex will have a (possibly empty) list of pure pebbles. From our example, pebbles w, x, and y will be placed during

Algorithm 2 The Merge Pebbles Algorithm. Determines whether or not graph G is solvable under configuration C.

```
function IsSolvableMergePebbles(G, C)
    for s ∈ V(G) do
        P(s) ← ∅                                   ▷ List of merged pebbles on vertex s
    R ← ∅                                          ▷ Covered vertices
    for s ∈ V(G) do                                ▷ Distribute Phase
        if C(s) > 0 then
            distributePurePebbles(G, P, R, s, C(s))
    if |R|= |V(G)| then
        return true
    else
        return doMergers(G, C, P, R)               ▷ Merge Phase
```

the distribute phase, and z will be placed during the distribute phase (which will be described below). Each list has two parts—the *unprocessed* and *processed* parts. Pebbles are always added to the unprocessed part of the list. More details will be given about the two parts of the list when we discuss the merge phase.

A simple shortest-path algorithm based on breadth-first search that is run from each source vertex s that has at least one pebble can be used to implement the distribute phase (see Algorithm 3). At the beginning, we set $m = 1$, a pure pebble $p = (s, m)$ is placed on s, and s is added to the list of reachable vertices. It is important to note that this implementation assumes that all vertices that obtain a merged pebble will each receive a reference to the same pebble. We will see why this is important later. The distance of s is set to 0 ($d(s) = 0$) and s is placed on the queue as usual. For every other vertex u, we set $d(u) = \infty$. The current distance (δ) is set to -1. Variable δ stores the distance from s to the last vertex that was processed. When the value of δ increases, it indicates that the current vertex u is of distance one more than the previous vertex processed, so a pure pebble with double the mass will be needed since twice as many pebbles will be required to reach u.

Algorithm 3 The distribute phase. Distributes the $n > 0$ pure pebbles from source s on graph G, where P is a set of lists of merged pebbles, one list for each vertex, and R is the set of vertices that have received at least one pebble.

```
function distributePurePebbles(G, P, R, s, n)
    m ← 1
    p ← (s, m)
    P(s).addToUnprocessed(p)
    R.Add(s)
    for u ∈ V(G) do d(u) ← ∞
    d(s) ← 0
    δ ← −1
    Q.enqueue(s)
    while not Q.isEmpty do
        u ← Q.dequeue
        if d(u) > δ then
            δ ← d(u)
            m ← m ∗ 2
            p ← (s, m)
            if m > n then return                   ▷ Remaining vertices are too far away
        for each neighbor v of u do
            if d(v) = ∞ then
                d(v) ← d(u) + 1
                Q.enqueue(v)
                P(v).addToUnprocessed(p)
                R.add(v)
```

Whenever a vertex u is removed from the queue, we need to do two things. First, if $d(u) > \delta$, u is one further away from s than the previously visited vertex, so m is doubled, $p = (s, m)$ so that p is a new pure pebble with double the mass, and $\delta = d(u)$. If p is no longer legal (that is, $p_s = m > C(s)$), we stop since all unvisited vertices, including u, are too far from s to receive a pebble. This is true since all vertices of distance k are processed before any vertices of distance $k + 1$. Second, (regardless of whether or not we doubled p), we visit every neighbor v of u. If $d(v) = \infty$, we set $d(v) = d(u) + 1$,

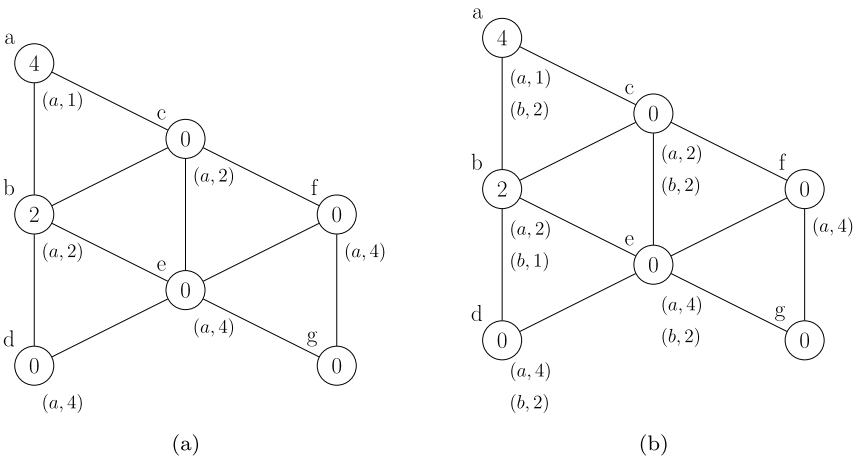


Fig. 2. (a) A graph with pure pebbles distributed from vertex a. (b) After pure pebbles are distributed from vertices a and b.

add v to the queue, add p to the list of pebbles on v, and add v to the list of reachable vertices. If $d(v) < \infty$, nothing is done since we already visited v, so it already has a pure pebble from s that has a smaller mass.

At the end of the distribute phase, we return true if all of the vertices are covered. Otherwise we continue to the merge phase. See Fig. 2 for an example of the result of the distribute phase on a small graph. It should be clear that the distribute phase takes $\Theta(|V|(|V| + |E|))$ time.

The merge phase must complete all of the rest of the possible moves by merging pairs of compatible pebbles and placing them on neighbor vertices. This process is repeated until the desired vertex is reached (for Reachable) or until all vertices have been reached (for Solvable), or until no untried pair remains (at which point we declare the vertex unreachable or the configuration unsolvable).

Now we proceed with a description of the merge phase of the algorithm (see Algorithm 4). We iterate over all of the vertices until we find a vertex v with a non-empty unprocessed list. We process the list as follows. Move the first pebble p from the unprocessed list to the processed list. One by one, merge it with each pebble on the processed list of v (including itself) that it is compatible with, and add the new merged pebble to the unprocessed list of all neighbors of v. Continue with the next unprocessed pebble on v. Once the unprocessed list for v has been exhausted, we continue iterating over the remaining vertices. If any new merged pebbles were placed on any vertices, iterate over the vertex set again. As stated previously, this process continues until either the vertex (or all vertices) are covered or until no new merged pebbles are placed anywhere. Fig. 3 continues the example that began in Fig. 2 by demonstrating the merge phase, including the optimizations that will be described next.

Algorithm 4 The merge phase. G is a graph, C is a configuration of pebbles, P is a set of lists of merged pebbles, one list for each vertex, and R is the set of vertices that have received at least one pebble.

```
function doMergers(G, C, P, R)
    moveMade ← true
    while moveMade do
        moveMade ← false
        for v ∈ V(G) do
            while P(v).hasUnprocessedPebbles do
                p ← P(v).getFirstUnprocessedPebble
                P(v).setProcessed(p)
                for q ∈ P(v).getProcessedPebbles do
                    if p and q are compatible then                ▷ This uses C
                        r ← p + q                                 ▷ Merging pebbles
                        for each neighbor u of v do
                            moveMade ← true
                            P(u).addToUnprocessed(r)
                            R.add(u)
                            if |R|= |V(G)| then return true
    return false
```

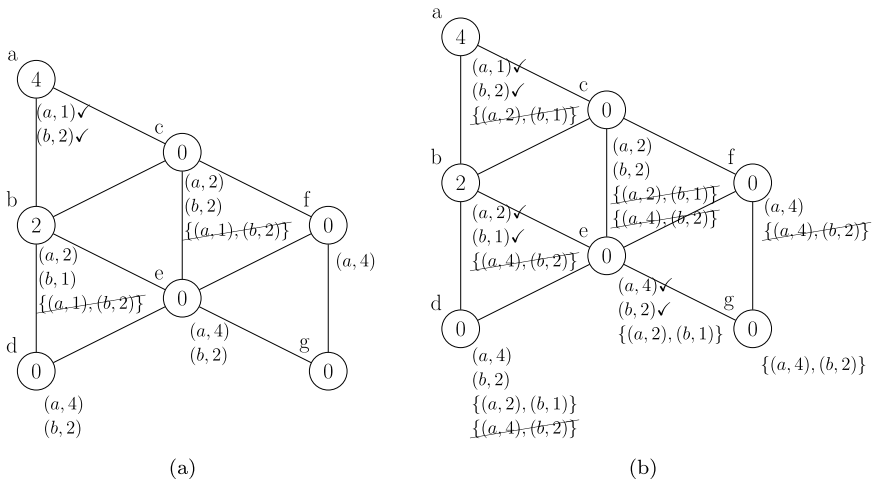


Fig. 3. (a) Merging the two pebbles from a and placing the merged pebble on neighbors. The crossed off ones are "too expensive" (see Lemma 2), and ones with check marks (✓) have been merged with everything. (b) Merging the two pebbles from b and then the two from e and placing on neighbors (skipping c and d). Each vertex has at least one pebble, so the graph is solvable. Note that not all possible mergers were made.

Next, we describe two important optimizations to the algorithm. The first optimization is based on the following variation of the No Cycle Lemma [9,19,20].

Lemma 1. *Let c and d be compatible merged pebbles on some vertex w such that c came from vertex u and d came from vertex v. Then no minimal pebbling sequence will place $c + d$ on u or v.*

For each merged pebble, we can keep track of all of the vertices that this pebble should not be merged to based on Lemma 1 — we will call these the *inadmissible* vertices for the pebble. It is important to note that this is not the list of vertices that this pebble should not be placed on, but the list of vertices that any merged pebble involving this pebble should not be placed.

To implement this optimization, the following changes are necessary. During the distribute phase, each pure pebble p at distance k from its source will have added to its inadmissible list all vertices of distance at most k from the source. This is accomplished by adding v to the inadmissible list of p every time p is placed on a vertex v, and copying the list to the new p whenever it is doubled. Here is where the fact that only one copy of each merge pebble is used is critical. After all vertices of distance k have been processed, the inadmissible list for p will have all vertices of distance at most k on it.

During the merge phase, every time we merge two pebbles s and t from a vertex v, $s+t$ will have on its inadmissible list all of the vertices on the inadmissible lists of s and t. (Note that v will already be on the list of both s and t.) Then, instead of placing $s+t$ on all neighbors of v, it will only be placed on the neighbors of v that are not on the inadmissible list of $s+t$, with each neighbor being added to the inadmissible list. As above, each neighbor receives a reference to the same instance of $s+t$ so that all neighbors appear on the inadmissible list after it has been placed on all neighbors.

For the second optimization, we need further notation. Given merged pebbles s and t, we write $s \leq t$ if for every vertex v, $s_v \leq t_v$, and $s < t$ if $s \leq t$ and $s_u < t_u$ for at least one vertex u. Note that this defines only a partial order on merged pebbles. The following result is the basis of the second optimization.

Lemma 2. *Let c and d be merged pebbles with $c < d$ that are both legal for some vertex u and let M be a pebbling sequence that moves a pebble to some vertex v. If M places d on u, then there exists a smaller pebbling sequence that moves a pebble to v that places c on u instead of placing d on u.*

Proof. Since $c < d$, c involves fewer resources and therefore fewer moves are needed to place c on u than to place d on u. Therefore the set of pebbling moves that places d on u can be replaced with a smaller set of pebbling moves that places c on u, producing the required smaller sequence.

Lemma 2 once again reduces the number of neighbors a merged pebble might be placed on. No changes are needed in the distribute phase of the algorithm. One simple change is needed in the merge phase. When attempting to place a pebble c on a vertex, it is only placed if there is not already a pebble b on that vertex such that $b < c$. In Fig. 3, the pebbles that are crossed off were not actually placed on those vertices based on one of these optimizations.

It remains to be proven that this algorithm solves Reachable and Solvable, and to determine the time complexity. It should be relatively clear that the complexity of both versions of the algorithm will be the same since the worst-case

termination condition is exactly the same. Further, as long as the reached vertices are returned by the algorithm, the SOLVABLE version can be used to solve REACHABLE. Thus, we will focus on the SOLVABLE version of the algorithm for the remainder of the section.

It is clear that if the algorithm places a legal merged pebble on any vertex v, then there is a legal pebbling sequence that places a pebble on v. We need to show that the converse is also true.

Let M be a minimal executable pebbling sequence that places a pebble on some vertex r. Then a unique merged pebble p_M that is legal for r can be constructed. Define $m_w^- = |\{(u, v) \in M \mid u = w\}|$ and $m_w^+ = |\{(u, v) \in M \mid v = w\}|$. That is, m_w^- is the number of moves made from w and m_w^+ is the number of moves made to w. Then $m_u = 2m_u^- - m_u^+$ is the number of pebbles originating from u that are used to execute the pebbling sequence. Note that since M is a minimal executable pebbling sequence, $m_r = -1$ and $m_u \geq 0$ for all other vertices.

If M is empty, then define $p_M = \{(r, 1)\}$. If M is not empty, then define $p_M = \{(u, m_u) \mid u \in V(G) \text{ and } m_u > 0\}$. It should be clear that p_M is legal for r and that it is properly defined in that it has the correct mass from each source vertex. Note that different sequences can produce the same merged pebble. However, since each minimal executable pebbling sequence that places a pebble on r has a unique merged pebble associated with it, we can think of the pebble that is moved to r as the merged pebble p_M.

Theorem 3. *Let M be a minimal executable sequence that places a pebble on r. Then the merge pebbles algorithm places a pebble p on r such that $p \leq p_M$.*

Proof. We will prove this by induction on the number of pebbling moves. If zero moves are required, then r begins with a pebble and the algorithm would place the pebble $(r, 1)$ on r. If one move is required, then it is the move (u, r) for some vertex u. In this case, the distribute phase of the algorithm would place the pure pebble $(u, 2)$ on r.

Assume at least two moves are required to place a pebble on r. A final move was made from some vertex u to r. We can partition M as $M = M_a \cup M_b \cup (u, r)$, where either M_a and M_b are both minimal executable sequences that each places one pebble on u; or M_a is a minimal executable sequence that places one pebble on u, M_b is empty, and u has a pebble.

By induction, the algorithm will place pebbles a and b on u, where $a \leq p_{M_a}$ and $b \leq p_{M_b}$ (where $b = (u, 1) = p_{M_b}$ if M_b is empty). Clearly a and b are compatible, so the algorithm will place $a + b \leq p_{M_a} + p_{M_b} = p_M$ on r.

Due to the nature of the algorithm, determining a tight bound on the complexity of the algorithm is essentially impossible. A crude analysis reveals that an upper bound on the worst-case complexity of the algorithm is $O(|V|^2 + |V| \cdot |E| + |V| \cdot A^2(|V| + D \cdot A \cdot |V|)) = O(|V| \cdot |E| + |V|^2 \cdot A^3 \cdot D)$, where D is the maximum degree of any vertex and A is the maximum number of merged pebbles that are placed on any node. Computing an exact bound on A is tricky. In general it can be exponential, but no worse than $\prod_{v \in V}(C(v) + 1) \leq 2^{|C|}$, and it is usually much smaller. In practice, the algorithm works very well on small graphs (e.g. less than about 16 vertices) and configurations with a relatively small number of pebbles (e.g. less than 32). In almost all cases it significantly outperforms the backtracking algorithm.

3. Pebbling number algorithms

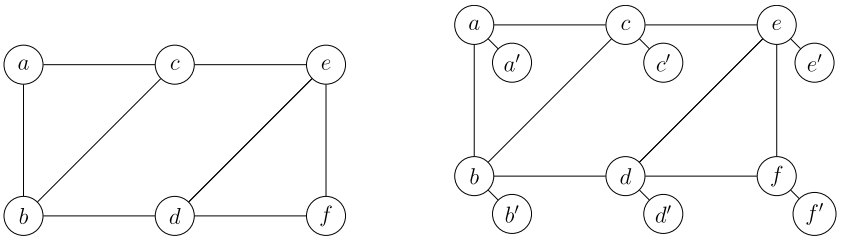
There are two different ways of thinking about computing the pebbling number of a graph. The first is to determine that all configurations of a given size are solvable and that at least one configuration with one fewer pebble is unsolvable. This can be done by setting the number of pebbles to $\max\{|V(G)|, 2^{\text{Diam}(G)}\}$ and then trying every configuration with that many pebbles until you check them all or find an unsolvable configuration. In the latter case, increase the number of pebbles by one and check all configurations with that many pebbles. Continue the process until all configurations of the current size are found to be solvable, at which point the number of pebbles is the pebbling number.

The other approach is to find the maximum number of pebbles that can be arranged on G that results in an unsolvable configuration. The pebbling number of G must be one greater than this number. One way to implement this idea is using a backtracking algorithm to construct the unsolvable configurations on G with the maximum number of pebbles, backtracking when a solvable configuration is found. The algorithm adds pebbles to a configuration until it is solvable, at which point it removes the last pebble and places it on the next vertex and continues. Since it only quits when all unsolvable configurations have been checked, it properly computes the pebbling number. Algorithm 5 provides the pseudocode for this pebbling number algorithm. It assumes that the vertices of G are indexed from 1 to $|V(G)|$ and that initially $C(v) = 0$ for all $v \in V(G)$. After executing PEBBLINGNUMBER(G), the global value of p will be set to $\pi(G)$.

At first glance it might seem that the first approach would be more efficient since it only checks one unsolvable configuration of any given size and focuses most of its effort on configurations with $\pi(G)$ pebbles, whereas the second approach tries many configurations of every size between 1 and $\pi(G)$. However, in practice the latter approach is much more efficient. Since the backtracking algorithm focuses on unsolvable and minimally solvable configurations, it is able to skip checking the solvability of many configurations with $\pi(G)$ pebbles because it already determined that a related configuration with fewer pebbles was solvable. Thus, although it checks the solvability of many more unsolvable configurations, this is offset by the number of solvable ones it can skip. In addition, many of the configurations it checks have fewer pebbles and determining the solvability of those is usually faster.

Algorithm 5 A pebbling number algorithm based on backtracking.

```
global p ← 0
function PebblingNumber(G)
    C ← empty configuration
    PNBacktrack(G, C, 1)
function PNBacktrack(G, C, j)
    if not IsSolvable(G, C) then
        if |C|≥ PN then
            p ← |C|+1
        for i ← j to |V(G)| do
            C(v_i) ← C(v_i) + 1
            PNBacktrack(G, C, i)
            C(v_i) ← C(v_i) − 1
```



(a) G (b) Two-pebbling extension of G.

Fig. 4. A graph and its two-pebbling extension.

4. Two pebbling property algorithms

The *two-pebbling extension* of a graph G is the graph created by adding, for each vertex $v \in V(G)$, a new vertex v' that is adjacent only to v. See Fig. 4 for an example. Call v' the *extended vertex* of v. The following lemma is straightforward.

Lemma 4. *Let G' be the two-pebbling extension of a graph G, C be a configuration of pebbles on G, and C' the configuration C extended to G' by placing zero pebbles on the extended vertices. Then C' is solvable on G' if and only if at least two pebbles can be moved to any vertex of G under C.*

We will describe an algorithm to find all minimally violating configurations on a graph G. Any minimally violating configuration will have $2\pi(G) - q + 1$ pebbles on q vertices. Because q can vary, we will describe an algorithm to find all minimally violating configurations for a given q. We will then use that algorithm for all possible values of q. There are some shortcuts we can use to speed up this process.

The first shortcut is obvious. It should be clear that if $q = 1$ or $q = |V(G)|$, then given any configuration of $2\pi(G)-q+1$ pebbles on q vertices, two pebbles can be moved to any vertex. Thus, we can skip those q values.

The second shortcut is based on the following lemma.

Lemma 5. *Let G be a graph on n vertices with diameter d such that $\pi(G) - n - d + 1 \geq 0$. If $2\pi(G) - n + 2$ pebbles are placed on $n - 1$ vertices of G, then it is possible to place 2 pebbles on any vertex.*

Proof. Let vertex r be the vertex that does not contain a pebble. First we show that 2 pebbles can be placed on r. Since every vertex of G except r has a pebble, at most $d + 1$ pebbles are needed to move one pebble to r. This leaves $2\pi(G) - n + 2 - (d + 1) = \pi(G) + (\pi(G) - n - d + 1) \geq \pi(G)$ pebbles on the graph, so a second pebble can be placed on r.

Next we show that 2 pebbles can be placed on every other vertex. If r is not a cut vertex of G, then $G \setminus \{r\}$ is a connected graph that contains at least one vertex with 2 pebbles (and all of the others have at least one), so 2 pebbles can be placed on any vertex in that component.

If r is a cut vertex of G, then $G \setminus \{r\}$ contains several connected components. If a component contains a vertex with at least 2 pebbles, then 2 pebbles can be placed on any other vertex in this component. To place a second pebble on a vertex in a connected component whose vertices all contain precisely one pebble, first place two pebbles on r (as argued above) and then pebble along the connected component to v. This works because placing two pebbles on r need not involve any of the pebbles in the destination connected component.

If none of those shortcuts allow us to skip the current value of q, then we need to check every possible configuration of $2\pi(G) - q + 1$ pebbles on q vertices to see if any of them violate the two pebbling property. To do this, first generate all possible q-subsets of $V(G)$, starting with $q = 2$. For each of these subsets, we need to distribute $2\pi(G) - q + 1$ pebbles across the vertices in every possible way. Thus, we generate all possible ordered partitions of $2\pi(G) - q + 1$ pebbles into q parts with no part having fewer than one pebble. For each of these partitions, we place the pebbles onto the vertices of the current subset and check if the resulting configuration is a violating configuration. This is done by using a solvability algorithm in combination with Lemma 4. We place the configuration onto the two-pebbling extension of G, where all of the extended vertices begin with no pebbles, and check the configuration for solvability. If it is solvable, then it is not a violating configuration. If it is not solvable, then it is a minimally violating configuration, so we add it to a set of all minimally violating configurations found. After all partitions of pebbles have been exhausted on all subsets of vertices, we increment q and repeat the process. When all values of q have been checked, we are done. Clearly a graph has the two-pebbling property if and only if the set of violating configurations is empty. The complete algorithm is given as Algorithm 6.

Algorithm 6 Two-pebbling property algorithm. Returns all minimally violating configurations of graph G.

```
function TwoPebblingProperty(G)
    A ← ∅                                          ▷ The set of minimally violating configurations
    π(G) ← PebblingNumber(G)
    G′ ← two-pebbling extension of G
    n = |V(G)|
    for q = 2 to n − 2 do
        p ← 2π(G) − q + 1
        FindViolators(G′, n, p, q, A)
    if π(G) − n − diam(G) + 1 < 0 then
        FindViolators(G′, n, p, n − 1, A)
    return A
function FindViolators(G′, n, p, q, A)
    C ← empty configuration on G′
    for each q-subset Q of {1, 2, . . . , n} do
        for each ordered partition P of p into q parts do
            Remove all pebbles from C
            for i ← 1 to q do
                C(Q(i)) ← P(i)
            if not IsSolvable(G′, C) then
                A.add(C)                           ▷ where the last n elements of C are ignored
```

The algorithm can easily be modified to find all violating configurations by running the same algorithm with $2\pi(G) - q + 1 + i$ pebbles for increasing values of i until no unsolvable configurations are found. But there is a better algorithm based on the fact that every violating configuration is a minimally violating configuration with more pebbles on one or more of the same q vertices that have pebbles. Thus, we can iterate over the minimally violating configurations and for each one use a backtracking algorithm that systematically adds pebbles to the q vertices that already have at least one and determine the solvability (still on the two-pebbling extension of G), recording a new violator and continuing if it is still unsolvable, or backtracking if it is solvable.

5. Simple reductions

As the diameter of a graph increases, the pebbling number increases exponentially. This means that computational techniques to determine the pebbling number of a graph or prove that a graph has the two-pebbling property become infeasible fairly quickly. In this section we develop a few results that allow us to avoid such lengthy computations by reducing the problem to a much smaller one.

The first result states that adding a vertex to a graph can no more than double the pebbling number.

Theorem 6. *Let G be a graph, $v \in V(G)$, and G' be the induced subgraph of G on $V(G) \setminus v$. If G' is connected, then $\pi(G) \leq 2\pi(G')$.*

Proof. Consider a configuration of $2\pi(G')$ pebbles on G. If v has no pebbles, then G' has $2\pi(G')$ pebbles and 2 pebbles can be placed anywhere on G' and one can be placed on v so it is solvable.

Table 1
Average CPU time per graph in seconds.

Size	Graphs	Lemke	CA	MP	BT
5	4	0	0.0730	0.1260	0.6115
6	47	0	0.3872	0.7050	29,252.0897
7	469	0	0.2813	3.2356	
8	6 940	22	4.2601	144.3602	
9 (diam 3)	148,226	306	1.6702		
9 (diam 4)	19,306	0	26.0005		
9 (diam 5)	1,804	0	1 352.0392		
9 (diam 6)	169	0	58,511.2535		

If v has $x > 0$ pebbles, then G' has $2\pi(G') - x$ pebbles. Since $x \leq 2\pi(G')$, we can place at least

$$\begin{aligned} 2\pi(G') - x + (x-1)/2 &= 2\pi(G') - (x+1)/2 \\ &\geq 2\pi(G') - (2\pi(G') + 1)/2 \\ &= \pi(G') - 1/2 \end{aligned}$$

pebbles on the vertices of G' so the configuration is solvable.

Theorem 7. *Let G be a graph, $v \in V(G)$, and G' be the induced subgraph of G on $V(G) \setminus v$. If G' is connected, has the two-pebbling property, and $\pi(G) = 2\pi(G')$, then G has the two-pebbling property.*

Proof. Consider a configuration of $4\pi(G') - q + 1$ pebbles on q vertices of G. If v has no pebbles, then G' has $2\pi(G') + (2\pi(G') - q + 1)$ pebbles on q vertices, so every vertex in G' can get 4 pebbles. This implies that v can receive 2 pebbles. If v has one pebble, then G' has $2\pi(G') - 1 + (2\pi(G') - q + 1) \geq (2\pi(G') - (q-1) + 1)$ pebbles on $q - 1$ vertices, so every vertex in G' can get 2 pebbles. Since this includes the neighbors of v, this implies that v can get 2 pebbles.

If v has $x \geq 2$ pebbles, then G' has $4\pi(G') - q + 1 - x$ pebbles, and we can move at least $(x-1)/2$ pebbles to G'. Since $x \leq 4\pi(G') - q + 1 - (q-1) = 4\pi(G') - 2q + 2$, we can get at least

$$\begin{aligned} 4\pi(G') - q + 1 - x + (x-1)/2 &= 4\pi(G') - q + 1 - (x+1)/2 \\ &\geq 4\pi(G') - q + 1 - (4\pi(G') - 2q + 2 + 1)/2 \\ &= 4\pi(G') - q + 1 - 2\pi(G') + q - 3/2 \\ &= 2\pi(G') - 1/2 \end{aligned}$$

pebbles on G' so that every vertex in G' can get at least two pebbles.

6. Results

We ran the two-pebbling property algorithm on all graphs with up to 9 vertices on a machine with two Intel Twelve Core Xeon E5-2650v4 2.2 GHz processors and 64 GB of 2400 MHz Registered ECC DDR4 memory. We used two obvious forms of parallelism. First, we ran the algorithm on 24 graphs at a time, starting a new graph each time a graph finished. Second, the two-pebbling property algorithm ran on separate threads for each q value.

The graphs were obtained from [18]. Since acyclic graphs and graphs with diameter at most two have the two-pebbling property [7,21], the algorithm first determines if a graph has either of these properties, quitting if it does. For graphs with neither property, we determined the pebbling number and then checked for the two-pebbling property, computing the solvability of many configurations of pebbles in the process. In order to assess the performance of our algorithms, we ran the graphs on three different solvability algorithms: the algorithm based on backtracking (BT), the merge pebbles algorithm (MP), and the combined algorithm (CA) that uses the merge pebbles algorithm along with the four heuristic algorithms. We measured both the wall-clock time and total execution time (the CPU time of all 24 processors added together). Although we might expect total execution time to be about 24 times the wall-clock time, this is actually not the case since, for instance, the final 6 graphs might take several hours to finish, leaving 18 CPUs idle. Thus we will use the total execution time. To make the numbers more manageable, Table 1 gives the average execution time per graph in seconds for each of these algorithms. Since the algorithms are never run on acyclic graphs and graphs of diameter one and two, the number of graphs indicated in the table does not include those graphs.

We break the 9-vertex graphs down by diameter since increasing the diameter leads to an exponential increase in the pebbling number and therefore the running time of the algorithms. The ten 9-vertex graphs with diameter 7 that are not acyclic take on the order of weeks to run so they are better handled directly. Four of them are P_8 with an added vertex that is connected to two adjacent vertices of the path. Since these have as a subgraph a spanning tree with the same pebbling number they clearly have the two-pebbling property. The other six are P_8 with an additional vertex that is connected to two vertices in the path that are of distance two from each other and optionally connected to the vertex in

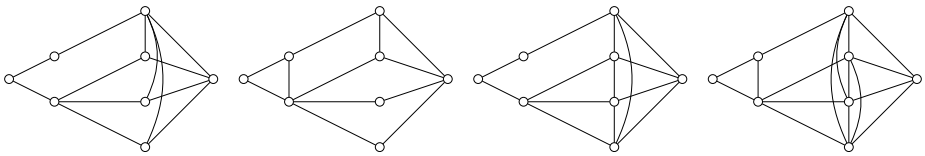


Fig. 5. The original Lemke graph, the two other minimal Lemke graphs, and the maximal Lemke graph on 8 vertices.

between (e.g. one connects the new vertex to p_3 and p_5, and another connects it to p_3, p_4, and p_5). Given this structure, Theorems 6 and 7 imply that they satisfy the two pebbling property.

Note that the backtracking solvability algorithm is so much slower than the other algorithms that it is unreasonable to use it on the graphs of 7 or more vertices. Also notice that the shortcut algorithms are of substantial benefit, so much so that using just the merge pebbles algorithm on the graphs of size 9 was not attempted.

We found all Lemke graphs with 9 or fewer vertices. There are no Lemke graphs with fewer than 8 vertices, 22 with 8 vertices, and 306 with 9 vertices. See [10] for the information needed to construct these graphs. For all of these, we found all of the violating configurations. We will give a more complete analysis of these graphs in a future paper, but we provide some preliminary observations.

All of the Lemke graphs on 8 vertices are a subgraph of a single Lemke graph and a supergraph of one of three Lemke graphs (see Fig. 5). Notice that the original Lemke graph is one of the minimal 8-vertex Lemke graphs. All 22 of the 8-vertex Lemke graphs have the exact same 6 configurations that violate the two-pebbling property, and two have 6 additional violating configurations because they exhibit symmetry in two pairs of vertices. Of the 6 configurations, 5 are minimally violating configurations. All configurations occur with $q = 5$.

The number of violators on the 9-vertex Lemke graphs is 6 (74 graphs), 12 (60 graphs), 29 (158 graphs), 35 (4 graphs), or 58 (10 graphs). All of the violating configurations have $q = 6$. 5358 of the violators have 13 pebbles (the minimum required) and 1108 have 14. So far, all known violators for any Lemke graph have at most $2\pi(G) - q + 2$ pebbles, which is just one more than the minimum number required.

212 of the Lemke graphs on 9 vertices have an 8-vertex Lemke graph as an induced subgraph. A closer analysis of these graphs will certainly lead to further results about Lemke graphs, including a better understanding of what makes these graphs violate the two-pebbling property and how to construct more infinite families of Lemke graphs.

Acknowledgment

This research was supported by the Howard Hughes Medical Institute through the Undergraduate Science Education Program.

References

[1] L. Alcón, M. Gutierrez, G. Hurlbert, Pebbling in Split Graphs, ArXiv e-prints, 2012.
[2] L. Alcón, M. Gutierrez, G. Hurlbert, Pebbling in 2-paths, Electron. Notes Discrete Math. 50 (2015) 145–150, LAGOS'15 VIII Latin-American Algorithms, Graphs and Optimization Symposium.
[3] L. Alcón, M. Gutierrez, G. Hurlbert, Pebbling in semi-2-trees, Discrete Math. 340 (7) (2017) 1467–1480.
[4] A. Bekmetjev, C.A. Cusack, Pebbling algorithms in diameter two graphs, SIAM J. Discret. Math. 23 (2) (2009) 634–646.
[5] A. Blasiak, J. Schmitt, Degree sum conditions in graph pebbling, Australas. J. Combin. 42 (2008) 83–90.
[6] D.P. Bunde, E.W. Chambers, D. Cranston, K. Milans, D.B. West, Pebbling and optimal pebbling in graphs, J. Graph Theory 57 (3) (2008) 215–238.
[7] F.R. Chung, Pebbling in hypercubes, SIAM J. Discrete Math. 2 (4) (1989) 467–472.
[8] T.A. Clarke, R.A. Hochberg, G.H. Hurlbert, Pebbling in diameter two graphs and products of graphs, J. Graph Theory 25 (1997) 119–128.
[9] B. Crull, T. Cundiff, P. Feltman, G.H. Hurlbert, L. Pudwell, Z. Szaniszlo, Z. Tuza, The cover pebbling number of graphs, Discrete Math. 296 (1) (2005) 15–23.
[10] C.A. Cusack, A. Green, A. Bekmetjev, M. Powers, [dataset] http://algoraph.hope.edu/?page=data, 2017.
[11] C.A. Cusack, T. Lewis, D. Simpson, S. Taggart, The complexity of pebbling in diameter two graphs, SIAM J. Discrete Math. 26 (3) (2012) 919–928.
[12] Z.-T. Gao, J.-H. Yin, The proof of a conjecture due to Snevily, Discrete Math. 310 (10) (2010) 1614–1621.
[13] Z.-T. Gao, J.-H. Yin, Lemke graphs and Graham's pebbling conjecture, Discrete Math. 340 (2017) 2318–2332.
[14] D. Herscovici, B. Hester, G. Hurlbert, T-pebbling and extensions, Graphs Combin. 29 (4) (2013) 955–975.
[15] G. Hurlbert, The weight function lemma for graph pebbling, J. Comb. Optim. 34 (2) (2017) 343–361.
[16] G. Hurlbert, H. Kierstead, On the Complexity of Graph Pebbling, unpublished, 2005.
[17] T. Lewis, C.A. Cusack, L. Dion, The complexity of pebbling reachability and solvability in planar and outerplanar graphs, Discrete Appl. Math. 172 (2014) 62–74.
[18] B. McKay, http://users.cecs.anu.edu.au/~bdm/data.
[19] K. Milans, B. Clark, The complexity of graph pebbling, SIAM J. Discret. Math. 20 (3) (2006) 769–798.
[20] D. Moews, Pebbling graphs, J. Combin. Theory Ser. B 55 (2) (1992) 244–252.
[21] L. Pachter, H.S. Snevily, B. Voxman, On pebbling graphs, Congr. Numer. 107 (1995) 65–80.
[22] N. Sieben, A graph pebbling algorithm on weighted graphs, J. Graph Algortihms Appl. 14 (2) (2010) 221–244.
[23] H.S. Snevily, J.A. Foster, The 2-pebbling property and a conjecture of Graham's, Graphs Combin. 16 (2) (2000) 231–244.
[24] S.S. Wang, Pebbling and Graham's conjecture, Discrete Math. 226 (1) (2001) 431–438.
[25] N.G. Watson, The Complexity of Pebbling and Cover Pebbling, ArXiv Mathematics e-prints, 2005.