© The Author 2007. Published by Oxford University Press on behalf of The British Computer Society. All rights reserved.
For Permissions, please email: journals.permissions@oxfordjournals.org
Advance Access publication on March 5, 2007 doi:10.1093/comjnl/bxm004

A Linear-Time Constant-Space Algorithm for the Boundary Fill Problem

Vladimir M. Yanovsky*, Israel A. Wagner and Alfred M. Bruckstein
Computer Science Department, Technion IIT, Haifa 32000, Israel
**Corresponding author: volodyan@cs.technion.ac.il*

In this paper, we consider the problem of boundary fill of a 4 or 8-connected region in a graphic device having a color image frame-buffer memory. We provide an algorithm that solves the problem in a time linear in the number of pixels in the region and requiring only constant memory space in addition to the frame-buffer memory itself. We map this problem to a boundary fill problem in a general graph, and solve it using a novel depth first search-based algorithm.

Keywords: Computer, graphics, boundary, fill, algorithm, space, complexity

Received 25 June 2006; revised 2 October 2006

1. INTRODUCTION

Suppose we are given a *seed* pixel located within a region defined by the boundary of some predefined color, called *boundary color*. In the boundary fill problem, the goal is to change the color of all pixels in the region where the seed is located to the *fill color*. Formally, two pixels of non-boundary color (x_1, y_1) and (x_2, y_2) are called *4-neighbors* if $|x_1 - x_2| + |y_1 - y_2| = 1$. Two pixels (x_1, y_1) and (x_2, y_2) are called *4-connected* if there is a path from (x_1, y_1) to (x_2, y_2) such that any two consecutive pixels on the path are 4-neighbors. A 4-connected region is defined as a set of pixels that can be 4-connected to the *seed; 8-connectivity* is defined similarly: two non-boundary pixels (x_1, y_1) and (x_2, y_2) are called *8-neighbors* if $\max\{|x_1 - x_2|, |y_1 - y_2|\} = 1$ and 8-connectivity is defined as a transitive closure of the 8-neighboring relation.

Our algorithm uses as a backbone the well-known Schorr–Waite–Deutsch (SWD) [1] algorithm. The SWD is a graph traversal algorithm, used in the context of garbage collection, that is essentially the depths first search (DFS) using link reversal to eliminate the need to store the backtracking information. However, SWD needs a label indicating whether the node was visited or not. Our algorithm uses the frame buffer itself for book-keeping while filling and $O(1)$ additional memory. The algorithm starts in the *seed* pixel; when it halts, all region pixels are colored with the fill color, while the rest of the pixels remain unchanged.

This problem would be trivial if in each pixel we had memory dedicated for the sole purpose of storing DFS backtracking information. Though our algorithm manipulates frame-buffer memory, it does not require any changes in the memory hardware.

Let us first discuss some issues resulting from the memory limitation we will have to address. Suppose we implement the DFS on the graph defined by the pixels of the region, each region pixel corresponding to a node of the graph and each neighborhood relation – to an edge of the graph. Each possible DFS state of a node (the backtracking information, origin or not, pursued neighbors, etc.) can be encoded using unique corresponding color. The algorithm must be able to tell if the pixel it sees is a new unvisited region pixel or if it had already been visited and its color corresponds to the DFS state information encoded earlier – recall that the region pixels can be of any color, including those that the algorithm uses for the encoding. Moreover, the fill color cannot be used as the sole indication that the corresponding pixel was visited and colored as there could be some pixels in the region that initially were of the fill color, thus misleading the algorithm.

Assuming the degree of each vertex in the graph representation of the region is bounded by a constant, we present a linear time, constant additional space algorithm for traversing and filling a region graph. The algorithm temporarily recolors the nodes of the graph (region pixels). We prove that when the algorithm halts the graph is correctly filled. An implementation of the algorithm in case of eight (16) bits per pixel frame-buffer when the region is 4(8)-connected, respectively, is straightforward, though more careful implementations may be done even more efficiently.

Downloaded from http://comjnl.oxfordjournals.org/ at Technion-Israel Institute of Technology on December 19, 2013

2. RELATED WORK

The boundary-fill problem can be solved by a Tarjan [2] DFS algorithm in a time linear in the number of pixels in the region. This solution, however, may require a stack of size proportional to the number of pixels in the region. To address this drawback, methods to reduce the stack size requirement, such as 'pixel span approaches' were suggested, see, e.g. Hearn *et al.* [3] and Foley *et al.* [4]. In Burtsev *et al.* [5], the authors provide a review of known fill algorithms and present a new boundary fill algorithm that requires $O(x)$ space where x is the horizontal dimension of the box containing the region.

Lieberman [6] provided a DFS-based algorithm for the fill problem in binary (black and white) images; later Shani [7] has shown that in some cases the algorithm of Lieberman [6] fails to produce correct result, corrected the algorithm and improved its efficiency. The algorithm of Shani [7] maintains, in a computer's random access memory (RAM), a data structure holding the boundary of the DFS stack and uses the *blocking* operation – temporarily recoloring some pixels in the boundary color – to prevent the algorithm from revisiting already visited pixels. The time complexity of the algorithm in the worst case is $\Omega(N^2)$, and the space used is $\Omega(N)$, though the author claims that on average the algorithm is expected to perform much better.

Another approach considered so far was to sacrifice running time to obtain stackless solutions. Henrich [8] presents several algorithms that trade-off additional memory for speed. One of them, called *CycleFill*, fills arbitrary black and white images: a white pixel is selected in a region defined by the black boundary, at the end the whole region must be colored black. The algorithm uses only a constant additional space beyond the memory of the frame buffer – it works by preserving the connectivity of the not-yet-filled region. Though it completes the fill in a linear time for simply connected regions, if the region has holes the filling agent must go long distances to break the cycles around the holes looking for a pixel it can recolor without breaking region's connectivity. Henrich [8] did not analyse the time complexity of his algorithm, performing only empirical tests to assess its behavior. However, though regions requiring a number of steps of $\Omega(n^2)$, where n is the number of region pixels, are easy to construct. Related algorithms were later analysed by Wagner and Bruckstein [9] in a multi-agent cooperative cleaning context.

The simultaneous time and space requirements of different algorithms for traversal of undirected graph are as follows. The DFS and breadth-first search can traverse any graph with n vertices and m edges in time $O(n+m)$ and space $O(n)$. A straightforward implementation of each method requires n bits and $n+O(1)$ pointers of auxiliary storage. Tarjan [10] devised methods that need only $2n+m$ bits, of which m are read-only. He saves space by folding the queue or stack required by the search into the graph representation. His method for DFS is a variant of the SWD [1] list-marking algorithm. In order to know the next edge exiting from a node to be traversed without keeping it explicitly on the stack, he reorders the list of exiting edges every time the node is visited. There were many works considering tradeoffs between time and space for traversing a graph by different algorithms, among them DFS. However there is no algorithm for general graph traversal working in linear time and sub-linear space as was proved by Beame [11].

3. THE ALGORITHM

Our description uses a graphical notation where the nodes correspond to pixels, and the edges connect nodes corresponding to the neighboring, either 4- or 8-connected, region pixels. Given a graph and a *seed* node of non-boundary color, the *region* is defined as the set of nodes that are reachable from the *seed* via paths not crossing the boundary-colored nodes. Thus, pixels outside the region do not belong to the graph. The goal is to recolor all nodes in the *fill color*. The boundary color is denoted with BC, whereas the fill color is denoted with FC. 'White color' in the description of the algorithm is used as some color which is neither fill nor boundary. Encoding information in a node is implemented by changing its color. As we shall see, in a subset of the region nodes, the algorithm stores a snapshot encoding the locations of the boundary pixels among their neighbors. This can be done since the original color of the region's pixels is irrelevant and need not be preserved as they would be recolored in the FC anyway. Before we proceed to the formal description, let us provide an overview of the algorithm.

(i) Init: $v = seed$; $v.father = NULL$
(ii) Recolor nodes of FC at distance 2 from v (reachable via non-BC nodes) in any color other than FC or BC.
(iii) Block v by recoloring its neighbors (only non-BC nodes) in BC. The set of the nodes recolored by this step is denoted with *v.blockednodes*.
(iv) Encode the set *v.blockednodes* and v's father, the node from which v was entered, with the help of vth color. This can be done thanks to the fact that we do not need to restore vth color – it will be colored in FC later.
(v) Choose any node *next* adjacent to *v.blockednodes* of color other than FC or BC and not v itself.
(vi) If step (v) failed, i.e. no *next* found, color v and *v.blockednodes* in FC, backtrack to vth father and repeat step (v); halt when backtracked from the seed (after all its neighbors were processed). If succeeded, assign $v = next$ and go to step (ii).

In the pseudocode below we use the following notations.

(i) *Fill*(*List*) – colors the vertices in the *List* in the FC.
(ii) *IsEmpty*(*List*) – returns *true* if the *List* is empty.

Downloaded from http://comjnl.oxfordjournals.org/ at Technion-Israel Institute of Technology on December 19, 2013

(iii) *Pop*(*List*) – removes and returns the head of the *List*.
(iv) *Iterator*(*List*) – returns a ListIterator – iterator on the *List*; it is initialized to point to the head of the *List*.
(v) *Advance*(*ListIterator*) – each call to *Advance* returns the next element in the list; when the iterator reaches the end of the list, the function returns *NULL*.
(vi) *Color*(*SetOfNodes, color*) – sets $v.color := color$ for all $v \in SetOfNodes$.
(vii) *dist*(*u, v*) – returns the distance from *u* to *v* in the graph or ∞ if they are not connected.

Node *v* has the following fields.

(i) *v.color* – the color of *v*.
(ii) *v.father* – the node through which *v* was entered for the first time.
(iii) *v.blockedNodes* – a list of blocked, i.e. temporarily colored in the BC neighbors of *v*. The algorithm iterates through this list using iterator *v.currblock*. Since the nodes represent pixels, the fields *v.father*, *v.blockednodes* and *v.currblock* must be encoded by *v.color*; we will return to this after the algorithm's description.

```
Init: v:=NULL, next:=seed; goto 12
 0 do
 1  if ( v.currblock == NULL)
 2      Fill(v.blockednodes)     %     The
    backtracking case
 3    next := v.father; Fill(v)
 4    v := next
 5  else
 6    Cands={u|dist(u,v.currblock)==1    &&
              u.color != FC && u != v}
 8 if (IsEmpty(Cands)) % All (non-blocked)
   neighbors
 9    Advance(v.currblock) % of v.currblock
   were filled
10  else
11  next:= Pop(Cands) % Choose an unfilled
    neighbor of v.currblock
12  Color({u|dist(u, next)==2}, white)
13  next.father:= v
14  next.blockednodes:=
      {u|dist(u,next)==1 && u.color!=black}
15   next.currblock:= Iterator
     (next.blockednodes)
16   Color(next.blockednodes):= black
17   v:= next
18  while  (v  !=  NULL)  %  Exiting  when
       backtracked from the seed
```

Let us explain the first iteration of the algorithm more informally. In the *Init* line the algorithm initializes the current node *v* to *NULL* and the *next* node visited by the algorithm will be the *seed*.

After this the algorithm recolors all nodes at distance 2 from the *seed* and reachable from it in white color (this could be any color distinct from the FC and BC). This step allows the algorithm later to use the filling color as the indication of already filled nodes – as shall be proven in Lemma 1, if some of the nodes recolored in line (12) were of FC, this was their original color and they were not yet visited by the algorithm. Note that since only region pixels are connected by edges, this step does not recolor pixels outside the region.

In line (13), we set $seed.father = NULL$, this allows us in line (18) to check for the completion; on a frame-buffer this can be implemented by dedicating a bit for this purpose in the colors used for encoding the algorithm's bookkeeping information. In lines (14)–(16), all non-BC neighbors of the *seed* are put in the list *next.blockednodes*, colored in BC and *next.currblock* is initialized to point to the first element of the list. As result all neighbors of the *seed* are now of BC. In line (17), the current vertex, named *v*, is set to be the *seed*.

In line (1) we check if there is still a blocked neighbor of *v* to be pursued. If not, in lines (2)–(4) we fill the neighbors that were blocked in line (16), *v* itself and backtrack to *v*'s father.

In line (6) a set of *Cands* is defined as the set of neighbors of *v.currblock*, not including *v* and the vertices of the FC. In case when all neighbors of *v.currblock* were already pursued and filled, this set is empty and we advance the iterator *v.currblock* in line (9) and go to line (1). Otherwise *next* is assigned a value and we continue to line (12). From here on the algorithm behaves as if node *next* were the seed. An example of the algorithm running on a graph is illustrated in Fig. 1.

The number of colors required to be able to implement the algorithm on a frame-buffer depends on the connectivity model of the region. If the region is 4-connected, for any node there are four possible nodes that can be its father, two bits suffice to encode them. Among the four neighbors of *v* at most three nodes can be blocked in line (4) – the node between *v* and its father is already BC; three bits suffice to encode this set. Two more bits suffice to encode *v.currblock*. Allocating a bit to denote if the node is the *seed*, we get that in case of 4-connected region eight bits per pixel are enough. If the region is 8-connected, similarly we get that in order to encode *v.father*, *v.blockednodes* and *v.currblock* and 'is the *seed* bit' at most $\log_2 16$, 8 (as the number of possibly blocked nodes), $\log_2 8$ and 1 bits, respectively, are needed and the algorithm can be implemented on 16 bits per frame-buffer.

4. CORRECTNESS OF THE ALGORITHM

LEMMA 1 *The algorithm fills all nodes reachable from node* next *and backtracks to v. No node is filled twice.*

Proof. The proof is by induction on the size of the graph reachable from *next*.

Downloaded from http://comjnl.oxfordjournals.org/ at Technion-Israel Institute of Technology on December 19, 2013

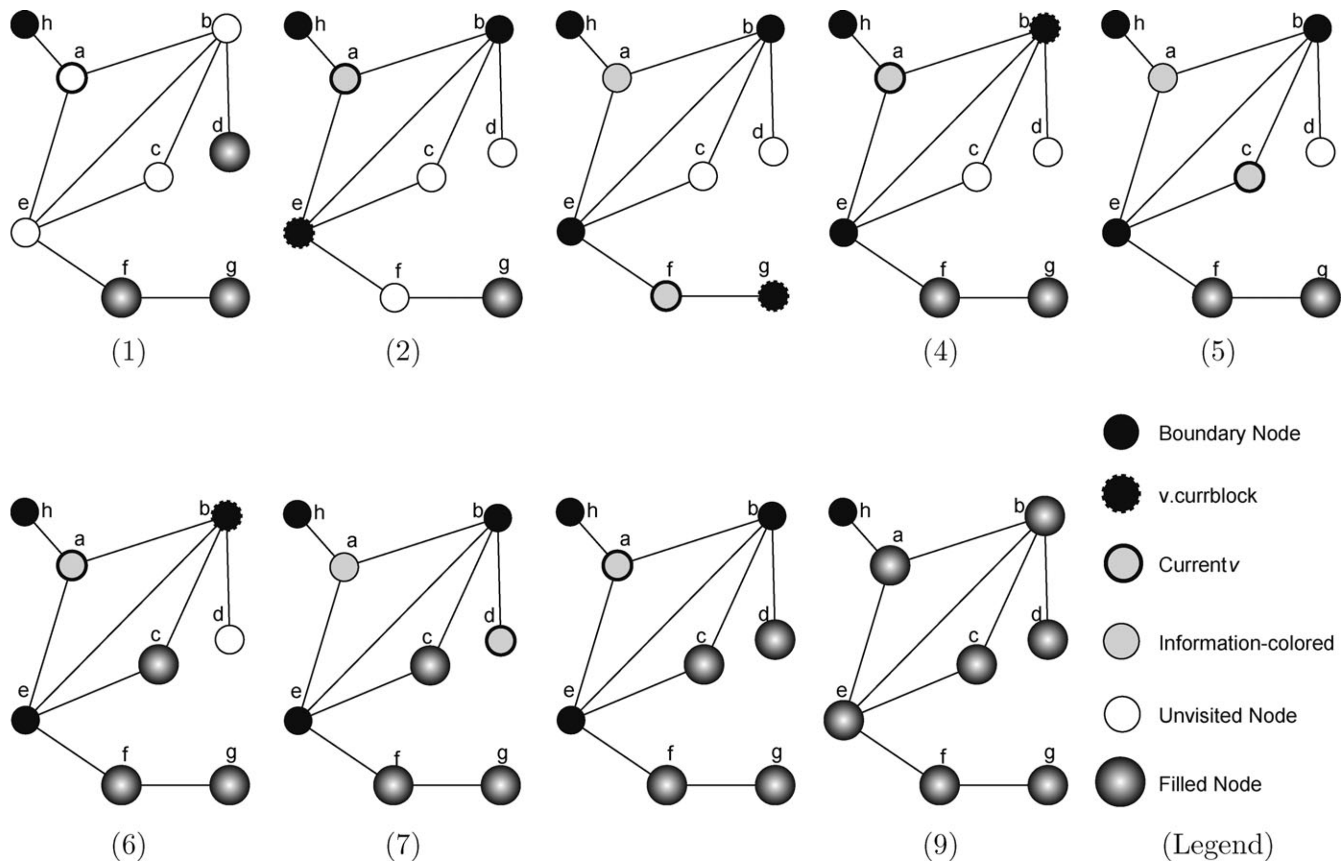

FIGURE 1. Example of the algorithm. (1) The *seed* is set to a. (2) After executing lines (12)–(16) of the algorithm; $a.currblock := e$ and nodes d and f are unfilled. (3) $v := f$; $f.currblock := g$. (4) The set of *Cands* is empty for the blocked node g. As the set *f.blockednodes* contains only g, the algorithm backtracks to a, filling g and f and setting $a.currblock := b$. (5) $v := c$. (6) The set *c.blockednodes* is empty, hence, we backtrack to a filling c. (7) $v := d$. (8) *d.blockednodes* is empty and we backtrack to a. (9) The set *a.blockednodes* and node a are filled; the algorithm halts.

(i) Basis. All neighbors of *next* are of BC. Then the algorithm completes backtracking to v and the lemma holds.

(ii) Inductive assumption: if at most $n-1$ nodes are reachable from *next*, the algorithm backtracks to v after filling all nodes reachable from *next*.

Suppose that n nodes are reachable from *next*.

After recoloring in BC all nodes at distance 1 from *next* and 'unfilling' the set of nodes at distance 2 from it, we call this set *AllCands*, the algorithm is then executed on its subset, called in the sequel *PursuedCands*. By the inductive assumption, after execution on any node in *PursuedCands*, the algorithm fills all nodes reachable from it and backtracks to *next*. Hence, after backtracking from *next* to v, the region nodes at distance 1 from *next* and the nodes reachable from *PursuedCands* are filled.

The set of nodes reachable from *next* are the nodes at distance 1 from it plus the the nodes reachable from *PursuedCands*. Since the nodes in the difference of *AllCands* and *PursuedCands* were not pursued because they were filled, they are reachable from *PursuedCands*. Hence, all the nodes reachable from the not pursued candidates were reachable from the *PursuedCands* and filled upon backtracking from *next*.

If some node was filled twice, it was reachable from two nodes of *PursuedCands*. But after all reachable from the first of these nodes were filled, the second could not get into *PursuedCands*. □

Since $E = \theta(V)$, as an immediate corollary of Lemma 1 we have the following.

Theorem 1

The algorithm correctly fills any graph in time $O(E)$, where E is the number of edges in the graph.

5. SUMMARY

We have shown the first linear-time constant-space algorithm for the boundary-fill problem in raster graphics. Since only $O(1)$ extra memory is required, this algorithm can be implemented in the hardware, using the existing frame-buffers and with minimum communication with the RAM. Though

Downloaded from http://comjnl.oxfordjournals.org/ at Technion-Israel Institute of Technology on December 19, 2013

our description is for a solid (single color) filling, the algorithm can be trivially extended to filling with patterns where the FC depends on a location of the pixel and defined by some function FC(*x, y*, seed), for example, chessboard pattern.

One of the limitations of the algorithm is its requirement for a color frame-buffer – the algorithm is not suitable for black and white (1-bit binary) devices. Another limitation is efficiency overhead of revisiting the same node. Possible extension may include developing a linear-time constant-space algorithm using subregions of connected region pixels, for example, horizontal or vertical scans of region pixels, as basic nodes to improve running time.

REFERENCES

[1] Schorr, H. and Waite, W.M. (1967) An efficient machine-independent procedure for garbage collection in various list sturctures. *Commun. ACM*, **10**, 501–506.

[2] Tarjan, R. (1972) Depth-first search and linear graph algorithms. *SIAM J. Comput.*, **1**, 146–160.

[3] Hearn, D. and Pauline Baker, M. (1997) *Computer Graphics* (2nd edn). Prentice Hall, NJ.

[4] Foley, J., van Dam, A., Feiner, S. and Hughes, J. (1996) *Computer Graphics: Principles and Practice in C* (2nd edn). Addison Wesley, Boston, MA.

[5] Burtsev, S.V. and Kuzmin, Ye.R. (1993) An efficient flood-filling algorithm. *Comput. Graph.*, **17**, pp. 549–561.

[6] Lieberman, H. (1978) How to color in a coloring book, *Proc. SIGGRAPH'78*, Atlanta, GA, August 23–25, pp. 111–116.

[7] Shani, U. (1980) Filling regions in binary raster images: a graph-theoretic approach. *Proc. SIGGRAPH '80*, Seattle, WA, July 14–18, pp.321–327.

[8] Henrich, D. (1994) Space-efficient region filling in raster graphics. *Vis. Comput.*, **10**, 205–215.

[9] Wagner, I.A. and Bruckstein, A.M. (1997) Cooperative cleaners – a study in ant robotics. In Paulraj, A., Roychowdhry, V. and Schaper, C.D. (eds), *Communication, Computation, Control and Signal Processing: A Tribute to Thomas Kailath*. Kluwer Academic Publishers, The Netherlands.

[10] Tarjan, R. (1983) Space-efficient implementations of graph search methods. *ACM Trans. Math. Softw.*, **9**, 326–339.

[11] Beame, P., Borodin, A., Raghavan, P., Ruzzo, W.L. and Tompa, M. (1999) A time-space tradeoff for undirected graph traversal by walking automata. *SIAM J. Comput.*, **28**, 1051–1072.

Downloaded from http://comjnl.oxfordjournals.org/ at Technion-Israel Institute of Technology on December 19, 2013