SIAM J. COMPUT.
Vol. 7, No. 4, November 1978

FINDING A MINIMUM CIRCUIT IN A GRAPH*

ALON ITAI† AND MICHAEL RODEH‡

Abstract. Finding minimum circuits in graphs and digraphs is discussed. An almost minimum circuit is a circuit which may have only one edge more than the minimum. To find an almost minimum circuit an $O(n^2)$ algorithm is presented. A direct algorithm for finding a minimum circuit has an $O(ne)$ behavior. It is refined to yield an $O(n^2)$ average time algorithm. An alternative method is to reduce the problem of finding a minimum circuit to that of finding a triangle in an auxiliary graph. Three methods for finding a triangle in a graph are given. The first has an $O(e^{3/2})$ worst case bound ($O(n)$ for planar graphs); the second takes $O(n^{5/3})$ time on the average; the third has an $O(n^{\log 7})$ worst case behavior. For digraphs, results of Bloniarz, Fisher and Meyer are used to obtain an algorithm with $O(n^2 \log n)$ average behavior.

Key words. graph, digraph, triangle, circuit, shortest path, matrix multiplication, analysis of an algorithm, computational complexity, worst-case, average-case, random graph

1. Introduction. In this paper we discuss finding short circuits in graphs and digraphs. The problem of digraphs arose when we tried to define the distance between two perfect matchings in a bipartite graph [4]. We assume that the reader is familiar with the standard definitions of graph theory [9]. Let $G=(V, E)$ be a graph with n vertices and e edges. In this paper the edges of a path (circuit) are all distinct. The length of a path (circuit) is the number of its edges. We assume that the vertices are numbered and we shall not distinguish between a vertex and its number. A minimum circuit is a circuit whose length is minimum. Harary [6] defines the girth of a graph to be the length of its minimum circuit. Several theorems relate to this notion [5], [7]. An almost minimum circuit is a circuit whose length is greater than that of a minimum circuit by at most one. We present an $O(n^2)$ algorithm for finding an almost minimum circuit. To find a minimum circuit we develop an $O(n^2)$ average time algorithm. The straightforward algorithm for finding a minimum circuit has an $O(ne)$ behavior. We also show an $O(n^2)$ reduction from the problem of finding a minimum circuit to that of finding a triangle (a circuit of length 3). Three methods for finding triangles are presented:

(i) Using rooted trees. The algorithm takes $O(e^{3/2})$ time in the worst case and $O(n)$ for planar graphs.

(ii) Check directly whether an edge is contained in a triangle. $O(ne)$ worst case and $O(n^{5/3})$ average time.

(iii) By Boolean matrix multiplication, in $O(n^{\log 7})$ time [10] (all logarithms are taken to base 2).

Algorithms for finding a shortest path in digraphs can be adapted to finding a minimum directed circuit (dicircuit). In particular, Friedman's $O(n^3(\log \log n/\log n)^{1/3})$ algorithm for weighted graphs [8] and directed breadth first search. The latter requires $O(ne)$ time in the worst case. However, it is proven, using the methods of [2], that on the average $O(n^2 \log n)$ time suffices. Using Boolean matrix multiplication we show how to find a shortest dicircuit in at most $O(n^{\log 7} \log n)$ time.

We use three representations of labeled graphs:

(i) The adjacency lists: $A(v)$ is the set of vertices adjacent to v. In this paper it is assumed that all graphs are given in this representation.

* Received by the editors November 24, 1976, and in revised form on October 18, 1977.

† Computer Science Department, Technion, Israel Institute of Technology, Haifa, Israel.
‡ IBM, Israel Scientific Center, Haifa, Israel.

(ii) The upper adjacency vectors: $UA(v)$ is a sorted vector which contains those vertices $w > v$ adjacent to v. This representation depends on the labeling of the vertices. Each edge is represented in exactly one vector. The vectors may be obtained from the adjacency lists in $O(e)$ time (using bucket sort).

(iii) The adjacency matrix: $(M)_{u,v} = 1$ if and only if u and v are connected by an edge. The adjacency matrix may be constructed from the adjacency lists in $O(e)$ time [1, p. 71, Ex. 2.12], even though this representation requires $O(n^2)$ space. Hence n^2 is a lower bound to the space requirements of all algorithms which use this representation.

2. Finding an almost minimum circuit. Let $G = (V, E)$ be an undirected graph with n vertices and e edges which has neither parallel edges nor self loops. Let *lmc* denote the length of a minimum circuit (if none exists then $lmc = \infty$). A circuit is *an almost minimum circuit* if its length is less than or equal to $lmc + 1$. We present an $O(n^2)$ algorithm for finding an almost minimum circuit.

First we present the algorithm *FRONT*. Given a vertex $v \in V$ this algorithm finds a lower bound for the length of the shortest circuit through v. The algorithm assigns values to two global variables: the vector k of length n and the $n \times n$ matrix *level*. These values are used in the sequel. *FRONT*(v) conducts a partial breadth first search (BFS) from v. When defined the value of $level(v, u)$ is the level of u in the search. If the connected component which contains v is circuit-free then the algorithm terminates with $k(v) = \infty$. Otherwise, it stops when the first circuit is closed; this circuit does not necessarily pass through v; $k(v)$ is defined to be the last level from which the search was conducted; $2k(v)+1$ is a lower bound for the length of the minimum circuit through v.

The algorithm *FRONT* uses a first-in, first-out queue which is initially empty. The queue operations are *enqueue*(u) which inserts u at the rear of the queue, and *dequeue* which removes and takes the value of the first element of the queue.

```
   procedure FRONT($v$);
   begin for $u \in V$ do $level(v, u)$ := nil;
     enqueue($v$); $level(v, v)$ := 0;
     while the queue is not empty do
     begin comment if the graph contains a circuit in the connected component of $v$
                   then the queue is never empty at this point;
       $u$ := dequeue;
       for $w \in A(u)$ do
           begin if $level(v, w)$ = nil
                   then begin $level(v, w)$ := $level(v, u)+1$;
                              enqueue($w$) end
1.                 else if $level(v, u) \leqq level(v, w)$
                        then begin $k(v)$ := $level(v, u)$;
                                   return end
           end
     end;
     comment the connected component of $v$ is circuit-free;
2.   $k(v) := \infty$
   end
```

FRONT builds a partial BFS tree. When a nontree edge is encountered (line 1) the algorithm terminates. Otherwise $k(v) = \infty$ (line 2). Each tree edge is scanned at most twice. Thus the algorithm takes $O(n)$ time. In the queue each vertex may appear at most once. Therefore, the algorithm requires $O(n)$ space for local variables, the vector *level* of length n, and the queue, in which each vertex can appear at most once. Hence, the algorithm requires $O(n)$ space in addition to the input. Observe that a minimum circuit through v could be found by scanning all the edges. In the worst case this takes $O(e)$ time. In the next section we present a method of scanning which takes $O(n)$ time on the average.

Let us apply *FRONT* to every vertex $v \in V$, and let *kmin* be the minimum value of $k(v)$.

LEMMA 1. *Let x be a vertex for which $k(x) = kmin < \infty$, then x is contained in an almost minimum circuit.*

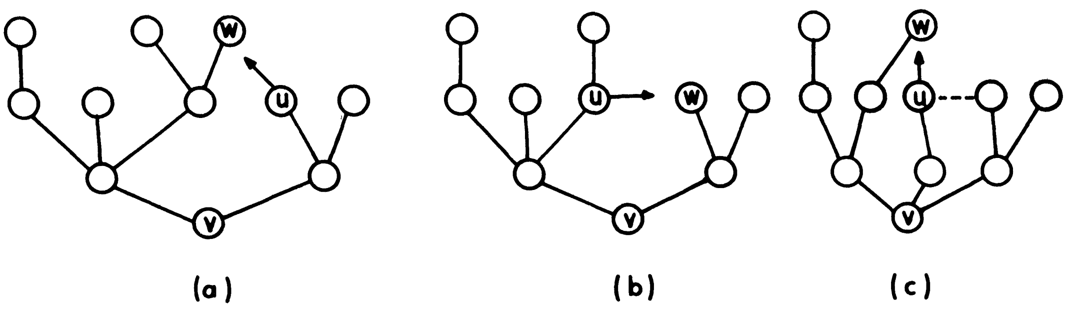


FIG. 1

Proof. Let v be a vertex which belongs to a minimum circuit C. If *lmc* is even, *FRONT*(v) stops when encountering a vertex w as in Fig. 1a; $k(v) = lmc/2 - 1$. If *lmc* is odd the algorithm stops as in Fig. 1b or Fig. 1c; $k(v) = (lmc - 1)/2$.

$$2k(v)+1 \leqq lmc \leqq 2k(v)+2.$$

Since $k(v) \geqq kmin$, $2kmin + 1 \leqq lmc$. The circuit found when applying *FRONT* to x is not longer than $2kmin + 2$. Therefore, it is not longer than $lmc + 1$ and is an almost minimum circuit. This circuit contains x, since otherwise its length would have been at most $2(kmin - 1) + 2 = 2kmin < lmc$, a contradiction. Q.E.D.

Note that if *lmc* is even then for a vertex x on a minimum circuit the algorithm stops as in Fig. 1a and finds a minimum circuit. In particular, in bipartite graphs the length of all circuits is even and the algorithm finds a minimum circuit.

Since *FRONT* is applied n times at most $O(n^2)$ time is required to find an almost minimum circuit. If the algorithm is applied to the full bipartite graph to which we add zero or more edges the algorithm might find only circuits of length four, even though the graph may contain triangles. In this case the algorithm requires $O(n^2)$ time, hence the bound is tight for the algorithm.

The space requirements can be lowered to $O(n)$. As noted, the queue requires only $O(n)$ space. The matrix *level* can be replaced by a vector in which for all v *level*(v, u) share the same location. The algorithm for finding an almost minimum circuit can be optimized by keeping the value of *kmin* and terminating *FRONT*(v) whenever *level*$(v, w) = kmin$.

3. Finding a minimum circuit. We have shown how to find a minimum circuit for the special case in which the length is known *a priori* to be even. In this section we use by-products of *FRONT* to develop an $O(n^2)$ average time algorithm to find a minimum circuit for the general case.

Assume that *FRONT* has been applied to a vertex v for which k is minimum and let us look at the values of *level*. If the connected component of v is circuit-free then the entire graph is circuit-free. Otherwise, a circuit is detected. Using the notation of *FRONT*, this circuit passes through u and w. If $level(v, u) = level(v, w)$ then the circuit is odd and thus minimum. Otherwise, the circuit is even and may not be minimum. It remains to check for the existence of an edge (x, y) such that $level(v, x) = level(v, y) = level(v, u)$. The vertex x must be either a vertex still in the queue or u itself. Thus, when $FRONT(v)$ terminates, define

$$F(v) = \{u\} \cup \{x \mid x \in V, x \text{ is in the queue}, level(v, x) = level(v, u)\}.$$

In $O(n)$ time we may sort $F(v)$ (bucket sort) and prepare a bit vector representing $F(v)$ and a linked list of its nonzero elements. The procedure *EDGE* below, when applied to $F(v)$ searches for an edge (x, y) in $F(v)$.

Let S be an ordered list of distinct vertices with the additional property that membership can be determined in constant time. (Observe that $F(v)$ satisfies these requirements.) $(x, y) \in E$ is an *S-edge* if $x, y \in S$. $EDGE(S)$ searches for vertices $u < w$ such that (u, w) is an S-edge. First it searches (lines 1–4) for (u, w) such that u is not among the last $n^{1/3}$ vertices of S. If unsuccessful, it searches exhaustively for an edge, the endpoints of which belong to the last $n^{1/3}$ portion of S (lines 5–6). If both searches fail then there exists no S-edge.

EDGE uses *UA* in a destructive mode. Since needed later, it can either be copied before use or reconstructed using a stack to undo all destructive operations. The latter solution is preferred since it enables a sublinear algorithm ($o(n)$). However, the details are omitted.

```
   procedure EDGE(S);
1. begin for $i := 1$ step 1 until $|S| - n^{1/3}$ do
     begin $u := S(i)$;
          while $UA(u)$ is not empty do
2.        begin choose at random a vertex $w$ in $UA(u)$;
3.             if $w \in S$ then return $((u, w))$;
               delete $w$ from $UA(u)$
          end
4.   end;
5.   for $i := \max(1, |S| - n^{1/3} + 1)$ step 1 until $|S|$ do
     begin $u := S(i)$;
          for $j := i + 1$ step 1 until $|S|$ do
          begin $w := S(j)$;
               if $(u, w) \in E$ then return $((u, w))$
          end
     end;
6.   return(nil)
   end
```

EDGE may require $O(n^2)$ time. However, its average behavior is better.

Let *ud* be *the upper degree vector* ($ud(v) = |UA(v)|$) and G_{ud} be the class of all labeled graphs with a given *ud* vector. Observe that the class of all labeled graphs is a disjoint union of all the G_{ud} classes.

Let P be a probability measure on labeled graphs, such that any two graphs in G_{ud} are equiprobable. The following probability measures are special cases of P [3]:

(i) The existence of each edge is an independent random variable with equal probabilities.

(ii) All graphs with a given number of edges are equiprobable.

For $S \subseteq V$, let E_S be a subset of $S \times (V-S)$ and e_S the cardinality of E_S.

LEMMA 2. *Let* $G_{E_S} = \{G = (V, E) | E \supseteq E_S\}$. *Then the average behavior of EDGE on* G_{E_S} *is bounded by* $O(e_S + n^{2/3})$.

Proof. If (u, w) belongs to E_S then the check $w \in S$ (line 3) necessarily fails. *EDGE* might waste at most $O(e_S)$ time on such edges. Therefore, it suffices to prove that the other edges require $O(n^{2/3})$ time on the average.

Using the linked list representation of S and the adjacency matrix, lines 5–6 require at most $O(n^{2/3})$ time. Thus, it remains to show that lines 1–4 require $O(n^{2/3})$ average time.

Under P, all graphs in $G_{E_S} \cap G_{ud}$ are equiprobable. We now wish to estimate the probability that an edge (u, w) chosen at random in line 2 is an S-edge. By assumption (u, w) does not belong to E_S. Let there be l_1 edges in $UA(u) \cap E_S$. Denote by l_2 the number of edges in $UA(u) - E_S$ checked before (u, w). The vertex w may be any of $n - u - (l_1 + l_2)$ remaining vertices, with equal probabilities. Since $w > u$, if $w \in S$ then it can be any one of the vertices of $S \cap \{u+1, \cdots, n\}$. The probability that $w \in S$ is therefore:

$$\frac{|S \cap \{u+1, \cdots, n\}|}{n - u - (l_1 + l_2)} \geqq \frac{n^{1/3}}{n}.$$

By decreasing the probability of success, the average number of trials until the first success increases. Hence, the average execution time of lines 1–4 is bounded by

$$O\left(\sum_{i=1}^{\infty} i(1 - n^{-2/3})^{i-1} n^{-2/3}\right) = O(n^{2/3}). \qquad \text{Q.E.D.}$$

The following procedure *MIN_CIRCUIT* finds a minimum circuit of length *lmc*. If *lmc* is finite the circuit passes through v. If *lmc* is odd then the circuit also passes through the edge a.

```
   procedure MIN_CIRCUIT(lmc, v, a);
1. begin for $v \in V$ do FRONT(v);
2.   find kmin;
     if $kmin = \infty$ then begin lmc := $\infty$;
                                return end;
3.   for $v \in V$ and $k(v) = kmin$ do
4.   begin find F(v);
           prepare a representation of F(v) as a sorted linked list;
5.         prepare a bit vector representation of F(v);
6.         a := EDGE(F(v));
           if $a \neq$ nil then begin lmc := 2kmin + 1;
                                return end
7.   end;
     lmc := 2kmin + 2;
     v := any vertex for which k is minimum
   end
```

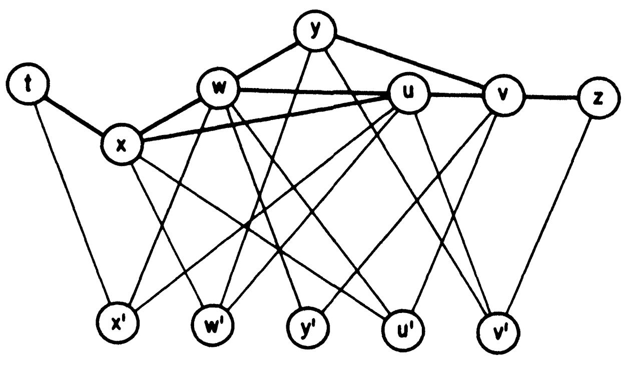
THEOREM 1. *The average execution time of MIN_CIRCUIT is bounded by* $O(n^2)$.

Proof. Line 1 requires at most $O(n^2)$ time; line 2, $O(n)$ time. In each iteration, lines 4–5 require $O(n)$ time. In line 6 *EDGE* is called with $S = F(v)$ and E_S is the set of edges incident with S which were scanned by *FRONT*(v). Hence, $e_S \leqq n$ and each iteration of line 6 costs $O(e_S + n^{2/3}) = O(n)$ time on the average. Since the loop (lines 4–7) may be executed at most n times, *MIN_CIRCUIT* requires $O(n^2)$ time on the average. (The average of a sum is equal to the sum of the averages.) Q.E.D.

Steps 1 and 2 of *MIN_CIRCUIT* can be done in one pass as explained in the end of the previous section. The space requirements can be lowered to $O(n)$ since we can keep a vector instead of the matrix *level*. In step 4, the values of *level*$(v, \cdot)$ are required to find $F(v)$, however *FRONT*(v) can be called again to obtain these values. This optimization increases the running time by at most a constant factor, while decreasing the space by a factor of n.

4. A reduction to finding triangles. Now we turn to show a reduction of the problem of finding a minimum circuit to that of finding a triangle in an auxiliary graph. A disadvantage of this method is that the number of edges might grow considerably. However, the number of vertices may only be doubled. Thereby, an upper bound for the complexity of the problem is found.

To this end we construct the graph $G' = (V' \cup V, E')$. V' consists of a copy of those vertices of G for which k is minimum. A vertex v' (v' denotes the vertex corresponding to v) is connected by an edge to all the vertices in $F(v)$. Fig. 2 contains an example of an auxiliary graph G'. The original graph G appears in boldface.



v	$F(v)$	$k(v)$
t	w, u	2
x	t, w, u	1
w	x, y, u	1
y	w, v	1
u	x, w, v	1
v	y, u, z	1
z	y, u	2

FIG. 2

LEMMA 3. *G' contains a triangle through v' if and only if v is contained in a minimum circuit in G and lmc is odd (i.e. $lmc = 2kmin + 1$).*

Proof. Let G' contain a triangle (v', x, y). By the construction, v' is connected only to vertices of $F(v)$. Therefore, $x, y \in F(v) \subseteq V$. The vertices x and y are at distance *kmin* from v. *FRONT* traces minimum length paths v–x, v–y. The length of these paths is *kmin* and they are vertex disjoint (i.e. they intersect only at v), because an additional intersection would entail a shorter circuit. (v', x, y) is a triangle in G' and $x, y \in V$. Thus, (x, y) belongs to E. This edge and the two paths form a circuit of length $2kmin + 1$. Since $lmc \geqq 2kmin + 1$ the circuit is minimum.

In the other direction, assume lmc is odd and a minimum circuit passes through v. Therefore, $lmc = 2kmin + 1$, $k(v) = kmin$ and $v' \in V'$. Let C be a minimum circuit through v. There are exactly two vertices x, y in C whose distance from v is $kmin = \lfloor lmc/2 \rfloor$. Thus, $x, y \in F(v)$ and $(x, y) \in E'$. Therefore, (v', x, y) is a triangle in G'. Q.E.D.

COROLLARY. *If a triangle in G' passes through a vertex $x \in V$ then there exists a minimum circuit of G through x.*

Proof. If the triangle consists solely of vertices of V then the triangle is contained in G and is a minimum circuit (because parallel edges and self loops have been excluded). If the triangle contains a vertex of V' then this follows from the proof of Lemma 3. Q.E.D.

Finding a triangle in G' provides us with an edge $(x, y) \in E$ which is contained in a minimum circuit of G. The circuit itself may be found in $O(n)$ time by an algorithm similar to *FRONT*.

5. Algorithms for finding triangles.

We study several algorithms for finding triangles.

5.1. Search by rooted spanning trees.

Let T be a rooted spanning tree of a connected graph. Using the following lemma we may construct an algorithm to check whether the graph contains a triangle.

LEMMA 4. *There exists a triangle which contains a tree edge if and only if there exists a nontree edge (x, y) for which $(father(x), y) \in E$.* (Every edge is checked in both directions.)

Proof. If $(father(x), y) \in E$ then obviously $(x, y, father(x))$ is a triangle.

In the other direction, assume that (x, y, z) is a triangle and (x, z) is a tree edge (without loss of generality $x = father(z)$). Two cases arise: If $(z, y) \notin T$ then the condition is met for this edge since $(father(z), y) = (x, y) \in E$. Otherwise, $(z, y) \in T$. In this case $z = father(y)$ (each vertex has at most one father). The condition is met for the nontree edge (y, z) since $(father(y), x) = (z, x) \in E$. Q.E.D.

For each nontree edge (x, y) we can check whether $(father(x), y) \in E$ in constant time using the adjacency matrix. Consequently, in time $O(e)$ we may check whether there exists a tree edge which belongs to a triangle.

Let us call a connected component *trivial* if it is an isolated vertex. We may now describe the procedure *TREE*:

```
procedure TREE;
  1. Find a rooted spanning tree for each nontrivial connected component of $G$;
  2. If any tree edge is contained in a triangle the algorithm terminates;
  3. Delete the tree edges from $G$.
```

Each iteration of *TREE* requires at most $O(e)$ time.

```
procedure TRIANGLE;
  Repeat TREE until all edges of $G$ are deleted.
```

THEOREM 2. *For planar graphs TRIANGLE requires at most $O(n)$ time.*

Proof. *TRIANGLE* deletes edges from the graph. We first show that each iteration of *TREE* deletes at least a third of the remaining edges. At first $e \leqq 3n - 6$ and we delete $n - 1$ edges; $(n - 1) \geqq e/3$. At subsequent iterations a third of the edges of each connected component are deleted. Therefore, a third of the remaining edges are deleted. Consequently, the number of edges at the ith iteration is at most $(\frac{2}{3})^{i-1}e$. The work in the ith stage is proportional to the number of remaining edges. Therefore, the total work is proportional to $\sum_{i=1}^{\infty} e(\frac{2}{3})^{i-1} = 3e = O(n)$. Q.E.D.

THEOREM 3. *For any graph TRIANGLE requires at most $O(e^{3/2})$ time.*

Proof. Let c denote the number of connected components. During the execution of *TRIANGLE* the value of c increases. Initially $c = 1$. At first we estimate the time required by *TRIANGLE* while $c \leqq n - e^{1/2}$. Then we estimate the time while $c > n - e^{1/2}$:

(a) $$c \leqq n - e^{1/2}.$$

Each iteration of *TREE* causes the deletion of $n - c \geqq n - (n - e^{1/2}) = e^{1/2}$ edges. Since there are e edges there may be at most $e/e^{1/2} = e^{1/2}$ such iterations.

(b) $$c > n - e^{1/2}.$$

The degree of each vertex is at most $n - c \leqq n - (n - e^{1/2}) = e^{1/2}$. Since each iteration of *TREE* decreases the degree of each nonisolated vertex, there may be at most $e^{1/2}$ such iterations.

Therefore, we have at most $2e^{1/2}$ iterations in the entire process. Each iteration takes $O(e)$ time. Thus, *TRIANGLE* takes $O(e^{1/2})O(e) = O(e^{3/2})$ time. Q.E.D.

For $K_{n,n}$ (the full bipartite graph with $2n$ vertices) the algorithm may take $O(e^{3/2})$ time while $c \leqq n - e^{1/2}$. For the graph obtained by adding m vertices all connected to a single vertex of $K_{m,m}$ $O(e^{3/2})$ time is required while $c > n - e^{1/2}$.

5.2. Search by vertices. G contains a triangle if there exists a vertex v and an edge a between two vertices $u, w (u < w)$ of $UA(v)$.

```
procedure VERTEX;
for $v \in V$ do
  begin $a := EDGE(UA(v))$;
        if $a \neq$ nil then return($v$)
  end
```

EDGE requires that $UA(v)$ be represented by an ordered linked list; moreover, membership in $UA(v)$ can be determined in constant time using the adjacency matrix.

THEOREM 4. *VERTEX finds a triangle in $O(n^{5/3})$ on the average.*

Proof. The proof is based on Lemma 2. When calling $EDGE(UA(v))$, E_S is empty. Therefore, $EDGE(UA(v))$ requires at most $O(n^{2/3})$ time on the average. The result follows since *EDGE* is called at most n times. Q.E.D.

Note, that if the upper adjacency vectors or the adjacency matrix has to be prepared then by the note in the Introduction, the algorithm requires additional $O(e)$ time. In any case, $O(n^2)$ space is required.

5.3. Matrix multiplication. Let M be the adjacency matrix (i.e. $(M)_{u,v} = 1$ if and only if $(u, v) \in E$). Let M^2 be the Boolean multiplication of M with itself. $(M^2)_{u,v} = 1$ if and only if there exists a vertex w such that $(M)_{u,w} = (M)_{w,v} = 1$ (i.e. $(u, w), (w, v) \in E$). If also $(M)_{u,v} = 1$, then (u, v, w) forms a triangle. Let $B = M^2$ **and** M (**and** denotes element-by-element logical and). $(B)_{u,v} = 1$ if and only if a triangle passes through the edge (u, v). Using Strassen's algorithm [10] we may multiply Boolean matrices in $O(n^{\log 7})$ time, thus obtaining an $O(n^{\log 7})$ algorithm.

Combining this algorithm with the reduction of § 4 we obtain an algorithm for finding a minimum circuit that takes at most $O(n^{\log 7})$ time.

6. Finding a minimum dicircuit. In the sequel digraphs, dicircuits and dipaths denote directed graphs, circuits and paths respectively.

The techniques for (undirected) graphs described in the previous sections are not applicable to the problem of finding minimum dicircuits in digraphs. Dicircuits may be found by n applications of the procedure *DICIRCUIT* described below. This method has worst case behavior $O(ne)$ but $O(n^2 \log n)$ on the average. Another method using Boolean matrix multiplication requires $O(n^{\log 7} \log n)$ time.

6.1. The procedure *DICIRCUIT*. *DICIRCUIT*(v) finds a shortest dicircuit through v. We conduct a directed BFS from v. The queue has the same role as in *FRONT*; $level(v, u)$ denotes the length of the shortest dipath from v to u if one exists and **nil** otherwise; *scan* denotes the number of scanned vertices.

```
   procedure DICIRCUIT(v);
   begin for $u \in V$ do $level(v, u) := $ nil;
         enqueue(v); $level(v, v) := 0$; $scan := 1$;
         while $scan < n$ do
         begin if queue is empty then begin
                                      $k(v) :=$ nil;
1.                                    return end;
               $u :=$ dequeue;
               for $w \in A(u)$ do
                  if $w = v$ then begin $k(v) := level(v, u) + 1$;
2.                               return end
                  else if $level(v, w) = $ nil then
3.                        begin $level(v, w) := level(v, u) + 1$;
                              enqueue(w);
                              $scan := scan + 1$ end

         end;
         enqueue(u);
4.       while queue is not empty do
               begin $u :=$ dequeue;
                     if $(u, v) \in E$ then begin $k(v) := level(v, u) + 1$;
5.                                             return end
               end;
6.       $k(v) :=$ nil
   end
```

The procedure may terminate at four points in the program:

(a) Line 1. The queue has become empty. In this case there is no dicircuit through v, so we return with $k(v) = $ **nil**.

(b) Line 2. We have returned to vertex v. In this case we have closed a shortest dicircuit through v whose length is $k(v)$.

(c) Line 5. We have reached all the vertices. In this case we look for the first vertex in the queue which closes a dicircuit. This is done via the adjacency matrix. The vertices of the queue are ordered by nondecreasing value of *level*. Therefore, a dicircuit closed in this stage is indeed a shortest dicircuit through v.

(d) Line 6. There is no edge from the scanned vertices to v. Therefore, there is no dicircuit through v, so we return with $k(v) = $ **nil**.

Even though *DICIRCUIT* may require $O(e + n)$ time, the average performance is somewhat better.

THEOREM 5. *Suppose P is a probability measure on labeled digraphs with n vertices such that digraphs with the same outdegrees are equiprobable. Then DICIRCUIT takes $O(n \log n)$ time on the average.*

Proof. *DICIRCUIT* takes most time if it scans all vertices. We may consider only the time needed to reach all vertices since the additional time (lines 4–5) is $O(n)$. Procedure R of [2] also scans a digraph until all vertices have been reached. The main difference is that R uses a stack while *DICIRCUIT* uses a queue. However, R does not take advantage of any property of the stack not shared by a queue. R is proven to take $O(n \log n)$ time on the average. Thus *DICIRCUIT* has an $O(n \log n)$ average behavior too. Q.E.D.

A shortest dicircuit through v can be found by inserting $father(w) := u$ after line 3. The dicircuit is found by backtracking from the vertex u which closed the dicircuit (lines 2 and 5).

By applying *DICIRCUIT* to all vertices of the digraph a shortest dicircuit may be found in $O(n^2 \log n)$ average time.

6.2. Binary search using matrix multiplication. Let *lmdc* be the length of the minimum dicircuit in G; M the adjacency matrix; D_j the matrix of dipaths of length less than or equal to j. ($(D_j)_{u,v} = 1$ if and only if there exists a dipath of length $1 \leqq l \leqq j$ from u to v.) The matrix D_j has a nonzero element on the main diagonal if and only if $lmdc \leqq j$. (i.e. *lmdc* is the smallest j for which D_j contains a nonzero element on its main diagonal.)

Let $j = i + k$ then $D_j = (D_iD_k)$ **or** M where D_iD_k is Boolean matrix multiplication and **or** is an element-by-element logical or, since $(D_iD_k)_{u,v} = 1$ if and only if there exists a dipath of length l, $2 \leqq l \leqq i + k$. The **or** operation adds the dipaths of length 1.

We compute D_j by the following method:

$$D_1 = M$$

$$D_{2l} = D_l^2 \textbf{ or } M.$$

We compute D_{2^i} until there is a nonzero element on the main diagonal. This happens when $i = \lceil \log lmdc \rceil$. The value of *lmdc* is found by a binary search on j in the region $2^{i-1} < j \leqq 2^i$: First we compute $D_{(2^{i-1}+2^i)/2} = D_{2^{i-2}+2^{i-1}} = D_{2^{i-2}}D_{2^{i-1}}$ **or** M. If the diagonal is all zeros we continue the search in the region $2^{i-1} + 2^{i-2} < j \leqq 2^i$. Otherwise, we continue in $2^{i-1} < j \leqq 2^{i-1} + 2^{i-2}$.

The process requires $2 \log lmdc$ matrix multiplications (i.e. $O(n^{\log 7} \log lmdc) = O(n^{\log 7} \log n)$ time).

The space requirements are $O(n^2 \log lmdc) = O(n^2 \log n)$ since we store $\log lmdc$ matrices.

The minimum dicircuit itself may be found in additional $O(e)$ time by a directed BFS from a vertex v for which $(D_{lmdc})_{v,v} = 1$.

7. Conclusions. Using *FRONT* we have an $O(n^2)$ reduction from the problem of finding a minimum circuit to that of finding a triangle. We have shown a method to find a triangle in $O(n^{5/3})$ average time. However, this itself does not yield an $O(n^2)$ average time algorithm to find a minimum circuit since the graphs obtained by the reduction might have a special structure and do not necessarily satisfy the probabilistic assumptions which led to the $O(n^{5/3})$ average time bound. Fortunately, we can solve the problem directly in $O(n^2)$ time on the average. However, any algorithm which finds a triangle in time greater or equal to $O(n^2)$ implies an algorithm to find a minimum circuit within the same time bound. Consequently, finding triangles by Boolean matrix multiplication leads to an $O(n^{\log 7})$ worst case algorithm to find a minimum circuit.

We have seen several algorithms for finding a triangle. *TRIANGLE* is efficient for sparse graphs (especially for planar graphs). *VERTEX* appears better on the average but has $O(n^3)$ worst case behavior. Better worst case performance can be achieved by using Boolean matrix multiplication.

A related problem is finding a minimum weighted circuit in a weighted graph. It is unclear to us whether our methods can be modified to answer this problem too.

Acknowledgment. The authors wish to thank Shmuel Katz for making valuable suggestions.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] P. A. Bloniarz, M. J. Fisher and A. R. Meyer, *A note on the average time to compute transitive closures*, Proc. of the 3rd Int. Colloquium on Automata, Languages and Programming (July 1976), S. Michelson and R. Milner, eds.

[3] P. Erdos and J. Spencer, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[4] A. Itai and M. Rodeh, *Some matching problems*, Proc. of the 4th Int. Colloquium on Automata, Languages and Programming (July 1977), A. Salomaa, ed.

[5] P. Erdös, *Graph theory and probability* II, Canad. J. Math., 13 (1961), pp. 346–352.

[6] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[7] L. Lovász, *On chromatic number of finite set systems*, Acta Math. Acad. Sci. Hungar., 19 (1968), pp. 59–67.

[8] M. L. Friedman, *New bounds on the complexity of the shortest path problem*, this Journal, 5 (1976), pp. 83–89.

[9] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

[10] V. Strassen, *Gaussian elimination is not optimal*, Numer. Math., 13, pp. 354–356.