Finding a Minimum Circuit in a Graph

Alon Itai
Technion, Israel Institute of Technology, Haifa, Israel
Michael Rodeh
IBM Israel Sceintific Center, Haifa, Israel

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. An $O(n^2)$ algorithm is presented to find an almost minimum circuit. The straightforward 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 presented. 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, recent results of Bloniarz, Fisher and Meyer are used to obtain an algorithm with $O(n^2 \log n)$ average behavior.

1. Introduction

In this paper we discuss finding short circuits in graphs and digraphs. We assume that the reader is familiar with the standard definitions of graph theory [Liu68]. Let $G=(V,E)$ be a graph with n vertices and e edges. The length of a path (circuit) is the number of its edges. We assume that the vertices are numbered and will not distinguish between a vertex and its number. A minimum circuit is a circuit whose length is minimum. 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 an $O(n^2)$ average time algorithm is developed. 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 a triangle 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) Checking 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 [Str69] (all logarithms are taken to base 2).

Two methods are described for finding directed circuits (dicircuits) in digraphs. The first requires $O(ne)$ worst case and $O(n^2 \log n)$ on the average (we use the results of [BFM76]).

The second uses $\log n$ Boolean matrix multiplications (i.e. $O(n^{\log 7}\log n)$ time).

We use three representations of labelled graphs:

(i) The adjacency lists: $A(v)$ is the set of vertices adjacent to v.

(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 labelling 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: $(M)_{uv}=1$ 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 ([AHU74] p.71, Ex. 2.12).

2. Finding and Almost Minimum Circuit

Let $G=(V,E)$ be an undirected labelled 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 G is circuit free 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 the algorithm *FRONT* is presented. Given a vertex $v\in V$ this algorithm finds a lower bound on the length of the minimum circuit through v. The by-products of the algorithm are used in the sequel. *FRONT* conducts a partial breadth first search (BFS) from v level-by-level. 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 on 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,k,level);
    begin for u∈V do level(u):=nil;
      level(v):=0; enqueue(v);
      while the queue is not empty do
      begin comment if the connected component of
            v contains a circuit then the queue is
            never empty at this point;
        u:=dequeue;
        for w∈A(u) do
        begin if level(w)=nil then
          begin level(w):=level(u)+1;
            enqueue(w) end
1.        else if level(u)≤level(w) then
          begin k(v):=level(u);
            return end
        end
      end;
      comment the connected component of v is
      circuit free;
2.    k(v):=∞
    end
```

FRONT builds a partial BFS tree. When a non-tree 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. The space requirements consist of 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 requires $O(e)$

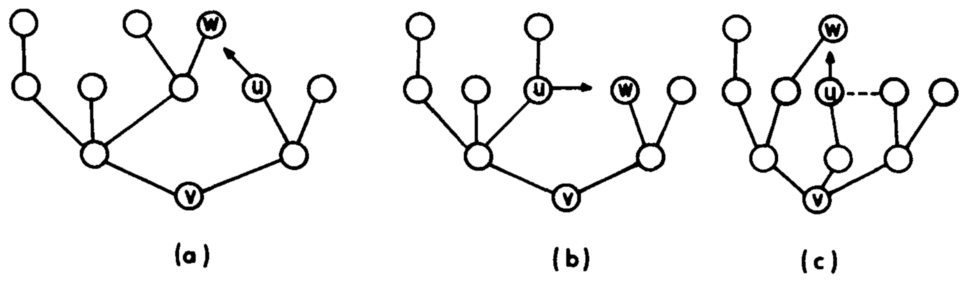


Figure 1

time. In the next section we present a scanning method 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.

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 Figure 1(a); $k(v) = lmc/2-1$. If lmc is odd the algorithm stops as in Figure 1(b) or Figure 1(c); $k(v) = (lmc-1)/2$. In either case we have

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

Since $k(v) \ge kmin$,

$$2kmin + 1 \le lmc.$$

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 Figure 1(a) 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 is bounded by $O(n)$ provided that we record only the minimum value of k and a vertex x for which it was obtained. Then an almost minimum circuit may be found in $O(n)$time by applying a procedure similar to *FRONT* to x.

3. Finding a Minimum Circuit

We have shown how to find a minimum circuit for the special case in which its length is known a priori to be even. In this section by-products of *FRONT* are used 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. 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 $level(u) = level(w)$ 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 $level(x)=level(y)=level(u)$. The vertex x must be either a vertex still in the queue or u itself. Thus, when $FRONT(v)$ terminates, define

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

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 non-zero elements. The procedure *EDGE* below, when applied to $F(v)$ searches for an edge (x,y) such that both x and y belong to $F(v)$.

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.) The edge (x,y) 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 it is needed later, it can either be copied before use or reconstructed using a stack to undo all the destructive operations. The latter solution is preferred since it enables a sublinear algorithm. 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 labelled graphs with a given ud vector. Observe that the class of all labelled graphs is a disjoint union of all the G_{ud} classes.

Let P be a probability measure on labelled graphs, such that any two graphs in G_{ud} are equiprobable. The following probability measures satisfy this requirement [ES74]:

(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 test $w \in S$ (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 one of the $n-u-(l_1+l_2)$ remaining vertices, with equal probabilities. Since $w>u$, if $w \in S$ then it may be any one of the vertices of $S \cap \{u+1,\dots,n\}$. The probability that $w \in S$ is therefore:

$$\frac{|S \cap \{u+1,\dots,n\}|}{n-u-(l_1+l_2)} \geq \frac{n^{1/3}}{n} = n^{-2/3} .$$

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 $lmc:=2 \cdot kmin+1$;
             return end
7.     end;
       $lmc:=2 \cdot kmin+2$;
       v:=any vertex for which k is minimum
    end
```

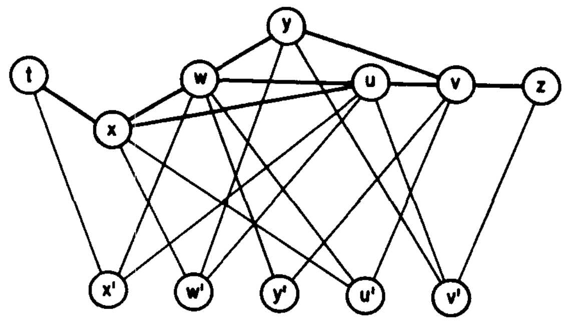
Theorem 1: The average execution time of *MIN_CIRCUIT* is bounded by $O(n^2)$.

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, $e_S \leq n$ 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, the average execution time of *MIN_CIRCUIT* is bounded by $O(n^2)$. Q.E.D.

The minimum circuit itself may easily be found in additional $O(n)$ time by applying *FRONT* to v. If *lmc* is odd the edge a is used to close the circuit.

4. A Reduction to Finding Triangles

Now we 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



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

Figure 2

only be doubled. Thereby, an upper bound for the complexity of the problem is obtained.

To this end we construct the graph $G' = (V' \cup V, E' \cup E)$. V' consists of a copy of those vertices of G for which k is minimum. Let v' denote the vertex corresponding to v. E' contains all the edges between each new vertex v' and the vertices in $F(v)$. Figure 2 contains an example of an auxiliary graph G'. The original graph G appears in boldface.

Lemma 3: G' contains a triangle through v' if and only if v is contained in a minimum circuit of 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 $lmc \geq 2kmin + 1$ 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 x is contained in a minimum circuit of G.

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 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

(i) 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 iff there exists a non-tree 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 non-tree edge (y,x) since $(father(y),x) = (z,x) \in E$. Q.E.D.

For each non-tree edge (x,y) we can check whether $(father(x),y) \in E$ in constant time using the adjacency matrix. Consequently, $O(e)$ time is required to check whether any tree edge belongs to a triangle.

Henceforth, a connected component is *trivial* if it consists of a single isolated vertex. We may now describe the procedure *TREE* :

```
procedure TREE;
  1. Find a rooted spanning tree for each non-
     trivial connected component of G;
  2. If any tree edge is contained in a triangle
     then stop (a triangle has been found);
  3. Delete the tree edges from G.
```

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

```
procedure TRIANGLE;
    Repeat TREE until all edges of G are deleted
    or a triangle is found.
```

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, $e \le 3n-6$ and we delete $n-1$ edges; $n-1 \ge e/3$. 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 i-th iteration is at most $(2/3)^{i-1}e$. The work in the i-th stage is proportional to the number of the remaining edges. Therefore, the total work is proportional to $\sum_{i=1}^{\infty} e(2/3)^{i-1} = 3e = O(n)$. 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. In the course of the execution of *TRIANGLE* the value of c increases. Initially $c = 1$. At first we estimate the time required by *TRIANGLE* while $c \le n-e^{1/2}$. Then we estimate the time while $c > n-e^{1/2}$:

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

At each application of *TREE* $n-c \ge n-(n-e^{1/2}) = e^{1/2}$ edges are deleted. 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 $n-c < n-(n-e^{1/2}) = e^{1/2}$. Since each iteration of *TREE* decreases the degree of each non-isolated 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_{nn} (the full bipartite graph with $2n$ vertices) the algorithm may take $O(e^{3/2})$ time while $c \le n-e^{1/2}$. For the graph obtained by adding $3m^2$ vertices all connected to a single vertex of K_{mm}, $O(e^{3/2})$ time is required while $c > n-e^{1/2}$.

(ii) Search by vertices.

G contains a triangle if there exists a vertex v and an edge a between two vertices u and 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 at most $O(n^{5/3})$ time 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 the algorithm requires additional $O(e)$ time.

(iii) Matrix multiplication

Let M be the adjacency matrix (i.e. $(M)_{uv}=1$ if and only if $(u,v)\in E$). Let M^2 be the Boolean multiplication of M with itself. $(M^2)_{uv}=1$ if and only if there exists a vertex w such that $(M)_{uw}=(M)_{wv}=1$ (i.e. $(u,w),(w,v)\in E$). If also $(M)_{uv}=1$, then (u,v,w) forms a triangle. Let $B=M^2$ and M (and denotes element-by-element logical and). $(B)_{uv}=1$ if and only if a triangle passes through the edge (u,v). Using Strassen's algorithm [Str69] 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 Section 4 yields an $O(n^{\log 7})$ algorithm for finding a minimum circuit.

6. Finding a Minimum Dicircuit

In the sequel digraphs, dicircuits and dipaths denote directed graphs, circuits and paths respectively. Given a digraph $G=(V,E)$ with no self loops, we wish to find a minimum dicircuit in it. The techniques for (undirected) graphs described in the previous sections are not applicable. However, a minimum dicircuit may be found by n applications of the procedure *DICIRCUIT* described below. The worst case behavior of this method is $O(ne)$ but on the average it requires $O(n^2 \log n)$ time. An alternative method is also presented. It uses Boolean matrix multiplication and requires $O(n^{\log 7} \log n)$ time.

(i) The procedure *DICIRCUIT*

$DICIRCUIT(v,k)$ finds a shortest dicircuit among those which pass through v. The procedure conducts a directed BFS from v. Whenever a new vertex w is encountered, *DICIRCUIT* checks whether $(w,v)\in E$. If so, a shortest dicircuit through v has been discovered and the process terminates. Otherwise, w is enqueued. Consequently there exists no edge from a vertex in the queue to v. The queue has the same role as in *FRONT*; $level(u)$ denotes the length of the shortest dipath from v to u if one has been found and nil otherwise; *scan* denotes the number of scanned vertices.

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

The procedure may terminate at three points in the program:

(a) Line 1: The queue has become empty, In this case there exists no dicircuit through v. Hence, $k(v) = \underline{\text{nil}}$.

(b) Line 2: Since $(u,w),(w,v) \in E$, a shortest dicircuit through v has been closed. Its length is $k(v)$.

(c) Line 3: All the vertices have been reached. No edge enters v. Thus, v is not contained in any dicircuit and $k(v) = \underline{\text{nil}}$.

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

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

Proof: It suffices to bound the average time needed to reach all the vertices.

Procedure R of [BFM76] also scans a digraph until all vertices are 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 the same behavior. Q.E.D.

DICIRCUIT can easily be modified to also find the shortest dicircuit through v itself.

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

(ii) Binary search using matrix multiplication. Let $lmdc$ be the length of a minimum dicircuit in G; M-the adjacency matrix; D_j-the matrix of dipaths of length less than or equal to j. $(D_j)_{uv} = 1$ if and only if there exists a dipath of length $1 \le l \le j$ from u to v. The matrix D_j has a non-zero diagonal if and only if $lmdc \le j$. (I.e. $lmdc$ is the smallest j for which D_j contains a non-zero element on its diagonal.) Moreover, the diagonal of D_n is zero if and only if G is acyclic.

We compute D_j by the following method:

$$D_1 := M$$

$$D_{2l} := D_l^2 \ \underline{\text{or}}\ M$$

($\underline{\text{or}}$ is an element-by-element logical or). $(D_l^2)_{uv} = 1$ if and only if there exists a dipath of length $2 \le m \le 2l$. The $\underline{\text{or}}$ operation adds the dipaths of length 1.

Thus, for i:=1,2,... we compute D_{2^i} until the diagonal is non-zero. (If for $i > \lceil \log n \rceil$ the diagonal of D_{2^i} is still zero, then G is acyclic. Therefore, we may terminate if i becomes equal to $\lceil \log n \rceil + 1$.) If G contains a dicircuit a non-zero diagonal is found when $i = \lceil \log lmdc \rceil$. The value of $lmdc$ is found by a binary search on j in the region $2^{i-1} < j \le 2^i$: First we compute

$$D_{(2^{i-1}+2^i)/2} = D_{2^{i-2}+2^{i-1}} = D_{2^{i-2}} D_{2^{i-1}} \ \underline{\text{or}}\ M.$$

If the diagonal is zero the search is continued in the region $2^{i-1} + 2^{i-2} < j \le 2^i$. Otherwise, we continue in $2^{i-1} < j \le 2^{i-1} + 2^{i-2}$.

The process requires $2\log lmdc$ matrix multiplications. Therefore,

$$O(n^{\log 7}\log lmdc) = O(n^{\log 7}\log n) \quad \text{time.}$$

The space requirements are $O(n^2\log lmdc) = O(n^2\log n)$ since $\log lmdc$ matrices have to be stored.

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

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 by 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. They 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 than or equal to $O(n^2)$ entails 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 an $O(n^3)$ worst case behavior. Better worst case performance can be achieved by using Boolean matrix multiplication.

Acknowledgment

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

REFERENCES

[AHU74] A.V.Aho, J.E.Hopcroft and J.D. Ullman, "The Design and Analysis of Computer Algorithms", Addison Wesely, 1974.

[BFM76] 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, S. Michelson and R. Milner (eds.),July 1976.

[ES74] P. Erdös and J. Spencer, "Probabilistic Methods in Combinatorics", Academic Press, 1974.

[Liu68] C.L.Liu, "Introduction to Combinatorial Mathematics", McGraw-Hill, 1968.

[Str69] "Gaussian Elimination is not Optimal", Numerische Mathematik 13, 354-356.