An Algorithm for the Longest Cycle Problem

E. T. Dixon
TRW Systems Group
Redondo Beach, California

S. E. Goodman
University of Virginia
Charlottesville, Virginia

ABSTRACT

An algorithm is presented which solves the problem of finding the longest simple cycle in an undirected edge-weighted network. The properties of the vector spaces associated with the fundamental cycles of a network are used to develop a suitable algorithm for finding simple cycles, and this is imbedded into a branch and bound scheme that solves the longest cycle problem.

1. INTRODUCTION

The problem of finding the longest simple cycle or path in an edge-weighted network is NP-complete, and as such is in the same class as the traveling salesman and the classic Hamiltonian cycle problems. In fact, it contains both as special cases [1].

Unfortunately, it is a major problem just to recognize a non-Hamiltonian longest cycle as such, and this appears to be at least partially responsible for the total lack of any theory on the subject. It also seems particularly difficult to get hold of either good bounds or good candidate feasible solutions to the longest cycle problem (LCP), in some contrast to the traveling salesman problem. The related longest path problem has an efficient algorithm for the special case of acyclic directed networks, but not much more seems to exist.

Applications of the longest cycle/path problems include routing and scheduling (especially critical path planning), the assessment of the vulnerability of a given network, and optimum cable design.

It is the objective of this paper to develop some theory relating to cycle vector spaces and to use this to present a branch and bound algorithm for the LCP for an arbitrary connected, undirected, network with edge weights.

We start by formulating the LCP as an integer linear program in bounded variables. The ILP model will find the longest cycle through a given node, say node 1, and application to each of the N nodes in the network will get the longest overall cycle.

We define the variables

$$C_{ij} = \begin{cases} 1 \text{ if nodes } i \text{ and } j \text{ are adjacent} \\ 0 \text{ otherwise} \end{cases}$$

$$W_{ij} = \begin{cases} \text{weight of edge } (i,j) \text{ if nodes } i \text{ and } j \text{ are adjacent} \\ 0 \text{ otherwise} \end{cases}$$

$$X_{ij} = \begin{cases} 1 \text{ if edge } (i,j) \text{ is used in the longest cycle} \\ 0 \text{ otherwise} \end{cases}$$

$$F_i = \begin{cases} 1 \text{ if node } i \text{ is used in the longest cycle} \\ 0 \text{ otherwise} \end{cases}$$

The integer programming model is then to

$$\max z = \Sigma X_{ij} W_{ij} \tag{1}$$

subject to the constraints

$$\sum_{i=1}^{N+1} C_{ik} X_{ik} = F_k \qquad k = 2, 3, \ldots, N \tag{2}$$

$$\sum_{j=1}^{N+1} C_{kj} X_{kj} = F_k \qquad k = 2, \ldots, N \tag{3}$$

$$\sum_{j=2}^{N} C_{1j} X_{1j} = \sum_{i=1}^{N} C_{iN+1} X_{iN+1} = 1 \tag{4}$$

$$Y_i - Y_j + NX_{ij} \leq N - 1 \qquad i, j = 1, 2, \ldots, N + 1 \tag{5}$$

$$0 \leq Y_i \leq N + 1, \ Y_i \text{ integer} \tag{6}$$

The node N + 1 is introduced by splitting node 1 into "new" nodes 1 and N + 1, each with the same adjacencies. The constraints (2) - (6) specify a simple path from node 1 to node N + 1. Constraints (5) and (6) introduce dummy integer variables, one Y_i for each node, which do not permit the formation of disjoint "side" cycles. This assures us that all feasible solutions will contain exactly one path from 1 to N + 1, and is a fairly standard trick in formulating combinatorial programs (cf. [2]).

One could now attempt to solve the ILP (1) - (6) using standard integer programming techniques, but the effort is somewhat prohibitive for all but the largest computer packages. For a graph with N nodes, the model has $O(N^2)$ constraints and integer variables. For most medium ILP routines, N = 10 would be stretching things.

We will now try to develop an algorithm for solving the LCP which takes a different approach that is based on the idea of a fundamental cycle set. To this end we first review some basic results (cf. [3] for a more complete discussion and proof of Theorem 1 presented below).

Given an arbitrary connected network G, let T be one of its spanning trees. Since there are exactly N - 1 edges in T, then there are $|E| - N + 1 \equiv K$ edges remaining in the network; so if any of these K edges are added to T, cycles will be formed; and in particular if any one edge is added exactly one cycle will be formed.

Definition 1: A fundamental cycle of a spanning tree T is a cycle obtained by inserting exactly one of the K remaining edges into T.

Definition 2: A fundamental cycle set of a spanning tree T is the set of cycles obtained by individually inserting each of the K remaining edges into T.

The theory we are about to develop will rely on the following definitions.

Definition 3: The set of cycles $C_1, C_2, \ldots, C_K$ will represent the fundamental cycle set of a spanning tree T, where each C_i is a binary $|E|$ - vector such that

$$C_i = (a_1\ a_2\ \ldots\ a_{|E|})$$

where each $a_j = 1$ if edge e_j is included in cycle C_i and $a_j = 0$ otherwise.

Definition 4: The sum or combination (mod 2) or 2 (or more) fundamental cycles C_i and C_j (of some fundamental cycle set) is denoted $C_i \oplus C_j$ such that

$$C_i \oplus C_j = C_i \cup C_j - C_i \cap C_j$$

where the intersections and unions are taken with respect to the binary $|E|$ - vectors.

Definition 5: An arbitrary combination of 2 or more fundamental cycles is called a circ.

The algorithm to follow will depend on the following theorem.

Theorem 1: Every simple cycle C of an undirected network G can be written as a combination of the fundamental cycles of a given fundamental cycle set $\{C_i\}$ of G, i.e. $C = \oplus \Sigma \alpha_i C_i$, where each $\alpha_i = 0$ or 1, and this representation is unique.

This theorem tells us that all the cycles of an undirected graph may be obtained by taking all the combinations of a fundamental cycle set, and then eliminating any duplicates. Although this approach will work, it is a bad one since it, in general, involves much more work than is necessary. We shall develop a more efficient algorithm that avoids some of the combinatorial difficulties.

2. AN ALL-CYCLE ALGORITHM

We shall now develop an algorithm for finding all the cycles of an undirected network utilizing the results of Theorem 1 and the fundamental cycle set. Although there are already existing algorithms (e.g. [4], [5], [6]) for finding the fundamental cycle set of a graph, none produce the cycles as we need them, so we shall first develop our own.

The problem of finding a spanning tree T for a network G is an easy one and may be solved in a variety of ways. But the problem of constructing a fundamental cycle set from T is somewhat more complicated because of the job of actually tracing each cycle through the tree. A number of people in the past have constructed algorithms using various search techniques for doing this, but we shall employ a more direct approach using logical operations.

We first begin to construct a spanning tree, in the manner of Patton [4], by choosing a node for the root and using all its incident edges. We then put the associated adjacent nodes on a "to be examined" (TBE) stack and label each with its unique path to the root by an appropriate $|E|$ - vector (where $a_i = 1$ for each edge used in the path, and $a_i = 0$, otherwise). We now pick the top node off the TBE stack and consider each of its unused edges in the following manner. If the adjacent node has not been examined, nor is on the TBE stack, then we include this edge in the tree. We then place the adjacent node on the TBE stack, and label it with its unique path to the root consisting of this edge plus the path from the top node. If the adjacent node has been examined, or is on the TBE stack, then this edge will form a fundamental cycle, and the cycle must be found. Now since both nodes have been labeled with their paths to the root of the tree (let us denote them by P_i and P_j) we can find the edges involved in the fundamental cycle by the simple logical operation (where P_K corresponds to an $|E|$ - vector representing the edge in question)

$$C_K = (P_i \oplus P_j) \oplus P_K.$$

This is because in an exclusive OR operation the only elements that are eliminated are those that are common to both members, and any such edges would not be part of the associated cycle.

After examining each of the unused edges incident to the top node of the TBE stack, we mark it examined, remove it from the stack and pick the next node. When no more nodes are on the TBE stack the algorithm terminates and all fundamental cycles have been found.

Definition 6: If D_1 and D_2 are 2 circs, then they are said to intersect if $D_1 \cap D_2 \neq \phi$; otherwise they are said to be disjoint.

Theorem 2: Let C be a cycle of a graph G and let $\{C_i\}$ be some fundamental cycle set for G. Partition those cycles for which $\alpha_i = 1$ in the unique representation of C in terms of the $\{C_i\}$ into two nonempty sets σ_1 and σ_2. Let $D_1 = \oplus \sum_{C_i \varepsilon \sigma_1} C_i$, and $D_2 = \oplus \sum_{C_i \varepsilon \sigma_2} C_i$, so that $C = D_1 \oplus D_2$. Then D_1 intersects D_2.

Theorem 3: Let $S = \{S_i\}$ be the set of fundamental cycles that comprise a cycle C of G, i.e. those for which $\alpha_i = 1$ as in Theorem 2. Then the cycles of S can be reordered as $S' = \{S_i'\}$ such that

$$S_1' \oplus S_2' \oplus \ldots \oplus S_i' \text{ intersects } S_{i+1}',\ i = 2, 3, \ldots, |S| - 1.$$

Proof: Let $S_1' = S_1$, $D_1 = S_1$ and $D_2 = S_2 \oplus \ldots \oplus S_{|S|}$. Now since $C = D_1 \oplus D_2$, by Theorem 2 D_1 must intersect D_2. There must be at least one $S_i \in D_2$ such that S_1 intersects S_i. Now let $D_1 = S_1' \oplus S_i$, $S_2' = S_i$ and remove S_i from D_2. The procedure can be repeated to obtain the remainder of the required ordering. Note that any $S_i \in S$ can be used as S_1'.

Theorem 3 will enable us to avoid a large number of combinations that could not lead to a cycle or which would effectively duplicate other combinations.

Let us now consider the following scheme for hunting through the combinations of fundamental cycles. We will use a binary tree structure, as shown in Figure 1, to partition the combinations into better defined subclasses. To the left of the root of the tree will lie all combinations of fundamental cycles which do not involve fundamental cycle C_1, and to the right of the root all combinations that do involve C_1.

The node to the right of the root will then be occupied by the first remaining fundamental cycle that intersects C_1 (here we have assumed that it is C_4). Now looking down from the C_4 node, to the left will lie all combinations involving C_1 and *not* C_4, and to the right all the combinations that involve both C_1 and C_4. The left node will be occupied by the next remaining (if there are any) fundamental cycle (other than C_4) that intersects C_1, and the right node by the next remaining fundamental cycle that intersects $C_1 \oplus C_4$. The remainder of the tree is constructed in a similar fashion.

The idea behind our all-cycle algorithm is to traverse this intersection tree in opposite pre-order fashion (meaning top node, right subtree, then left subtree). We shall now make the algorithm more explicit.

All-Cycle Algorithm

```
Let S = set of all simple cycles
    D = current circ

S1) [initialize] Produce a fundamental cycle set for the
    connected graph. Let S = $\phi$, D = first fundamental cycle,
    S = D and go to S2.
S2) [investigate right subtree] Let $C_i$ be the first avail-
    able fundamental cycle that intersects D. If none, go
    to S4, otherwise let D = $D \oplus C_i$ and go to S3.
S3) [check for a cycle] If D represents a simple cycle, let
    S = S + D. Remove $C_i$ from the available list and go to
    S2.
S4) [investigate left subtree] Remove the last used funda-
    mental cycle from the current circ. If D = 0, go to S6,
    otherwise let $C_i$ be the first available fundamental cycle
    that intersects D. If none, go to S5, otherwise let
    D = $D \oplus C_i$ and go to S3.
S5) [back up in tree] Add the last used fundamental cycle to
    the available list. Go to S4.
S6) [find next subtree] Let D = the first available funda-
    mental cycle. If none, stop, otherwise let S = S + D and
    go to S2.
```

The above algorithm serves to pre-order traverse (in reverse fashion) the intersection tree of all combinations of fundamental cycles. We can now prove the following result.

Theorem 4: The All-Cycle Algorithm produces all the cycles of graph G, without duplications.

Proof: Clearly S contains only cycles. By Theorem 1 and the fact that every node of the intersection tree represents a different combination of fundamental cycles, no cycles are included more than once. Since Theorem 3 guarantees that all cycles can be generated by "continued intersections," and since all such intersections are examined, all the cycles of G are produced.

Closely associated with the all-cycle problem, and similar to Theorem 3, is the following unsettled conjecture.

Conjecture: Let $S = \{S_i\}$, be the set of fundamental cycles that comprise a cycle C of graph G. Then the elements of S can be *reordered as $S' = \{S'_i\}$, such that $S'_1 \oplus S'_2 \oplus \ldots \oplus S'_i$ is a simple cycle and intersects S'_{i+1}, $i = 2, 3, \ldots, |S| - 1$.*

This conjecture is somewhat stronger than Theorem 3 in that it implies that each of the circs in the "continued intersection" must be a simple cycle. If true, this conjecture would be useful in establishing a polynomial bound on the execution time per cycle required by the all-cycle algorithm. While we have yet to find a counterexample for the conjecture, we have found some very rare situations where, once a continued intersection is started that meets the requirements of the conjecture, it cannot be continued with all circs being cycles. In each of these cases however another ordering existed that did meet the requirements.

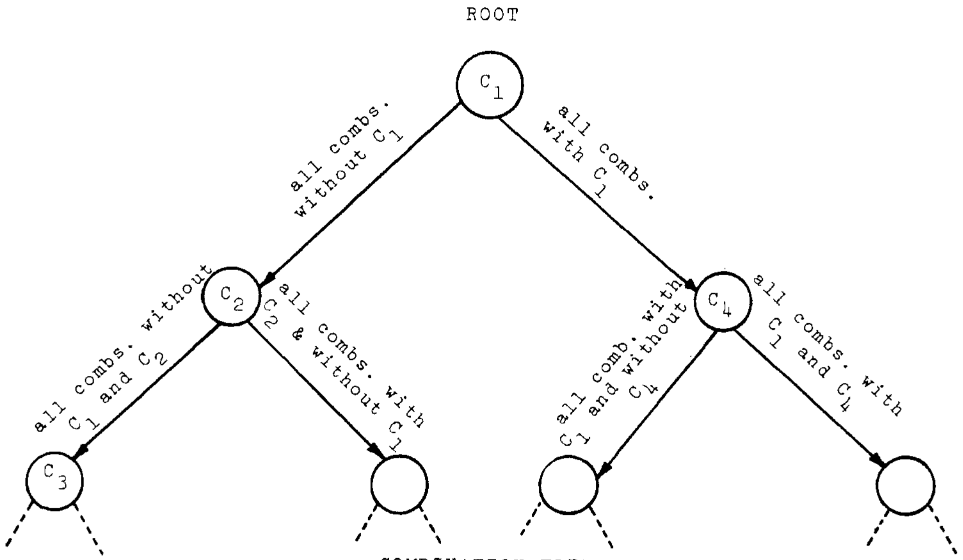
The all-cycle algorithm presented here has been coded in FORTRAN and programmed on a CDC 6400. It appears to perform well against other all-cycle algorithms for fairly low density graphs, i.e. those for which $p \le .2$ where p is the ratio of the number of edges in the graph to the number of edges in the complete graph on the same number of points. For example, the algorithm in [5] took 120 seconds to find all the cycles of 25 12-node cubic undirected graphs on a CDC 6500, while the algorithm of this section took 10 seconds. On the other hand, a modified version of our algorithm that will handle dense, directed graphs is not competitive with [6]; it took us over three times as long to list cycles for a complete directed graph on 9 nodes. On the basis of some of the comments in [6] however, it would appear that we are competitive for low density, undirected graphs. In any case, the algorithm of this section appears to be the one that is best suited to support the LCP algorithm which follows.

3. A LONGEST CYCLE ALGORITHM

The results of the last section may now be used to produce a branch and bound algorithm for the LCP. The algorithm will use the framework of the all-cycle algorithm along with a branch and bound scheme to further aid in fathoming subtrees.

We note that at each node of the intersection tree (of Figure 1) we have a current circ, and a set of remaining available fundamental cycles. If we now calculate the total weight of the different edges in this circ and these available fundamental cycles, we will surely have an upper bound on the length of the longest possible cycle that could result from combinations between them. If we now knew of a cycle whose length was equal to or greater than this bound, we could clearly eliminate this subtree from further consideration.

Since we have a set of fundamental cycles to begin with, we can use the longest of these as our first known "long" cycle. Then if a longer cycle is found, it will become the new longest known cycle. An additional increase in computation speed can also be made by sorting the fundamental cycles from longest to shortest, because, in general, we would expect to produce a longer cycle from longer fundamental cycles.


ROOT
C_1
all combs. without C_1
all combs. with C_1
C_2
C_4
all combs. without C_1 and C_2
all combs. with C_2 & without C_1
all comb. with C_1 and without C_4
all combs. with C_1 and C_4
C_3

COMBINATION TREE

Fig. 1

LCP Algorithm

S1) Produce a fundamental cycle set for graph G. If no cycles are found stop, otherwise sort the cycles from longest to shortest and go to S2.

S2) Execute the All-Cycle Algorithm of Section 2, except add the following operations at each node of the intersection tree.

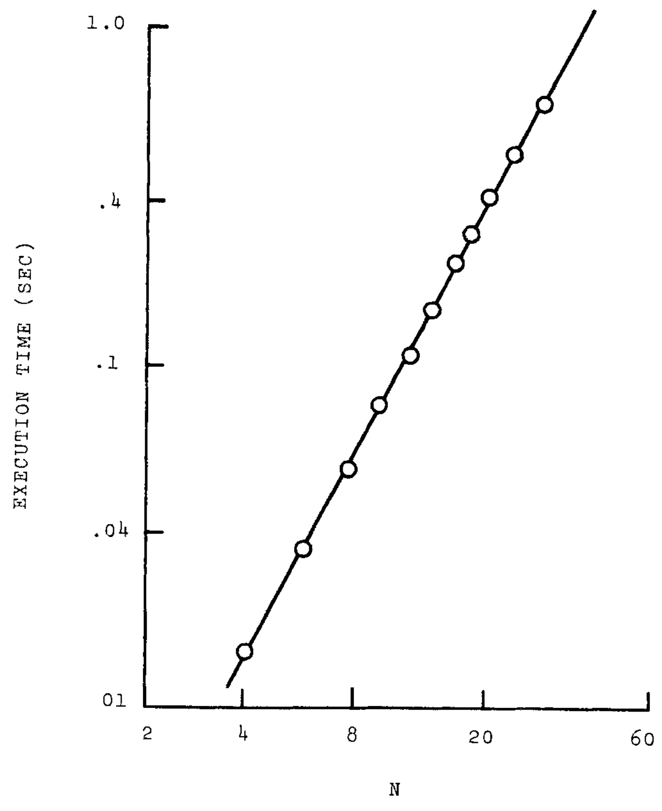
(a) Compute a bound, denoted by LB, on the longest possible cycle obtainable below this node by calculating the total weight of the different edges included in the current circ and the available fundamental cycles.

(b) If LB $\leq$ LC, where LC is the length of the longest known cycle, then this subtree is fathomed. Back up to the previous node.

(c) If this node's associated circ corresponds to a cycle whose length L is greater than LC, make this cycle the longest known cycle, and set LC = L.

The LCP algorithm was coded in FORTRAN for the CDC 6400. Average run times for low density ($p = .1$) unweighted random graphs with $4 \leq N \leq 40$ are shown in Figure 2.* Additional experience indicates that the algorithm is efficient for low density graphs in general, but bogs down rather badly for $p \gtrsim .2$. This behavior is not very surprising and stems from the fact that in high density graphs the complexity of multiple intersections of cycles becomes combinatorially hopeless.

For large graphs, a savings in storage can be made, at the expense of an increase in execution time, by using a predecessor label for each node instead of the full $|E|$ - vector, and then backtracking to recover disjoint segments of unique paths to a root. The authors are grateful to a referee for this suggestion.



LCP ALGORITHM EXECUTION TIME

FOR LOW DENSITY ($p = .1$) RANDOM GRAPHS

Fig. 2

**Graphs with randomly generated edge weights appear to take about 10% longer.*

REFERENCES

1. Hardgrave, W. W. and G. L. Nemhauser, "On the Relation Between the Traveling Salesman and the Longest-Path Problems," *Operations Research*, 10, 1962, pp. 647-657.

2. Hu, T. C., *Integer Programming and Network Flows*, Addison-Wesley, Reading, Massachusetts, 1970.

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

4. Patton, K., "An Algorithm for Finding a Fundamental Set of Cycles of a Graph," *CACM*, 12, 1969, pp. 514-518.

5. Gibbs, N. E., "A Cycle Generation Algorithm for Finite Undirected Linear Graphs," *JACM*, 16, 1969, pp. 564-568.

6. Tarjan, R., "Enumeration of the Elementary Circuits of a Directed Graph," *SIAM J. Computing*, 2, 1973, pp. 211-216.

Paper received June 28, 1974.