CIS-TR-80-4

DIRECTED MAXIMAL- CUT PROBLEMS

by

Arthur Farley*
Andrzej Proskurowski*

Abstract

A maximal-cut problem for a directed graph can be defined analogically to the undirected case. The latter problem is known to be NP-complete. Depending on the definition of the value of a cut, the resulting problem for general directed graphs is either efficiently solvable or NP-complete. We consider the latter variant of the problem and show an algorithm solving it in linear time for directed acyclic graphs. Breaking the cycles of a general graph and applying the same algorithm to the resulting forest gives an efficient solution algorithm for some classes of undirected graphs.

* Department of Computer and Information Science, University of Oregon, Eugene, OR 97403

1. Introduction

The maximal-cut problem for an undirected, weighted graph is that of finding a partition of vertices of the graph into two sets which maximizes the sum of the weights of edges connecting the two partitions. The problem is known to be NP-complete in the general case [2]. One can state directed counterparts of the maximal-cut problem, differentiated by the definition of the value of a directed cut. We state the general directed maximal-cut problem as the following decision problem:

> Directed Maximal-Cut: Given a directed graph $G=\langle V,A\rangle$, a weight function $w: A \to Z^+$ and an integer k, is there a partition of V into sets L and R such that the value of the cut exceeds k.

In one problem, DMC1, the value of a directed cut (L,R) is defined to equal the sum of weights of arcs oriented from L to R minus the sum of weights oriented from R to L. DMC1 is efficiently solvable.

Theorem 1.1 [1] There is an algorithm solving DMC1 for an arbitrary directed graph in time proportional to the number of arcs in the graph.

Another definition of the value of a directed cut yields an NP-complete problem. For DMC2, we define the value of a directed cut (L,R) to equal the sum of weights of arcs oriented from L to R. DMC2 is hard.

Theorem 1.2 [1] DMC2 is NP-complete.

In this paper we present an efficient algorithm which determines a partition (L,R) yielding maximal cut value (according to our second definition of cut value) for an arbitrary directed tree, thus solving DMC2. In what follows we will refer to such a cut as "the solution to the cut problem."

2. Directed paths

We will introduce the concepts underlying our solution for trees by first considering the problem for unidirectional paths. By unidirectional path we will understand a directed graph $G=(V,A)$ such that $V=\{v_0,v_1,\ldots,v_n\}$ and $A=\{\langle v_i,v_{i+1}\rangle \mid 0\le i<n\}$. Algorithm 2.1 which solves the cut problem for such graphs, traverses the path from v_0 through v_n placing vertices into sets L or R depending upon the weights of the encountered, incident arcs. With each vertex v, we associate two sets of vertices P(v) and Q(v) and two corresponding arc weight sums p(v) and q(v). As vertex v is visited by Algorithm 2.1, P(v) and Q(v) constitute a partition of vertices which have been visited but have yet to be placed into L or R. Furthermore, no arc connects two members of P(v) or Q(v), and v is in P(v). The value of p(v) is equal to the sum of arc weights which would be added to the value of the cut if every vertex of P(v) is placed in L and every vertex of Q(v) is placed in R. The value of q(v) is equal to the sum of arc weights which would be added to the value of the cut if the above assignments were reversed.

Algorithm 2.1 Solves the cut problem for unidirectional path.

Input: A path $v_0,\ldots,v_n$ with arcs $\langle v_i, v_{i+1}\rangle$ weighted w_i, $0\le i<n$.

Output: A partition (L,R) of vertices solving the cut problem with the value C.

Method: Instead of writing $P(v_i)$ we use P[i], etc.

```
(*Step 1.  Initialize*)
   for i:=0 to n do begin P[i]:={v_i}; Q[i]:=Φ; p[i]:=0; q[i]:=0 end;
   L:=Φ; R:=Φ; C:=0;
(*Step 2.  Traverse the path*)
   for i:=0 to n-1 do
      if p[i]≥q[i]
        then begin L:=L∪P[i]; R:=R∪Q(i]; C:=C+p[i];
                   q[i+1]:=w_i end
```

```
        else begin P[i+1]:=P[i+1]∪Q[i];
                   Q[i+1]:=Q[i+1]∪P[i];
                   p[i+1]:=p[i+1]+q[i];
                   q[i+1]:=q[i+1]+p[i]
                   end;
  (*Step 3.  Process last vertex*)
     if q[n]≥p[n] then begin L:=L∪Q[n];
                             R:=R∪P[n];
                             C:=C+q[n] end
                  else begin L:=L∪P[n];
                             R:=R∪Q[n];
                             C:=C+p[n] end;
  (*End of Algorithm 2.1*)
```

By induction using an appropriate loop invariant, it can be shown that Algorithm 2.1 guarantees that the value of P[i],Q[i],p[i], and q[i] have the meaning defined above when v_i is about to be processed. Surely, if the value of p[i] is greater than or equal to q[i], we can do no better than place P[i] in L and Q[i] in R. This will allow the algorithm to maximize the value obtained from arcs up to v while maintaining the possibility of acquiring w_i as well. The value of q[i+1] is modified to reflect the possible contribution from the arc $\langle v_i, v_{i+1} \rangle$ if v_{i+1} were placed in R. On the other hand, if the value of q[i] is greater than p[i], we cannot yet make a decision as the arc $\langle v_i, v_{i+1} \rangle$ may have a high value but will not be obtainable if v_i is placed in R. Figure 1 shows the decision tree considered by Algorithm 2.1 for an initial segment of a unidirectional path. If p[i] ever proves to be greater than (or equal to) the value of q[i], then the algorithm returns to the root of the decision tree, with the next vertex appearing as v_0. When v_n is visited, the decision made there is correct, and the algorithm terminates.

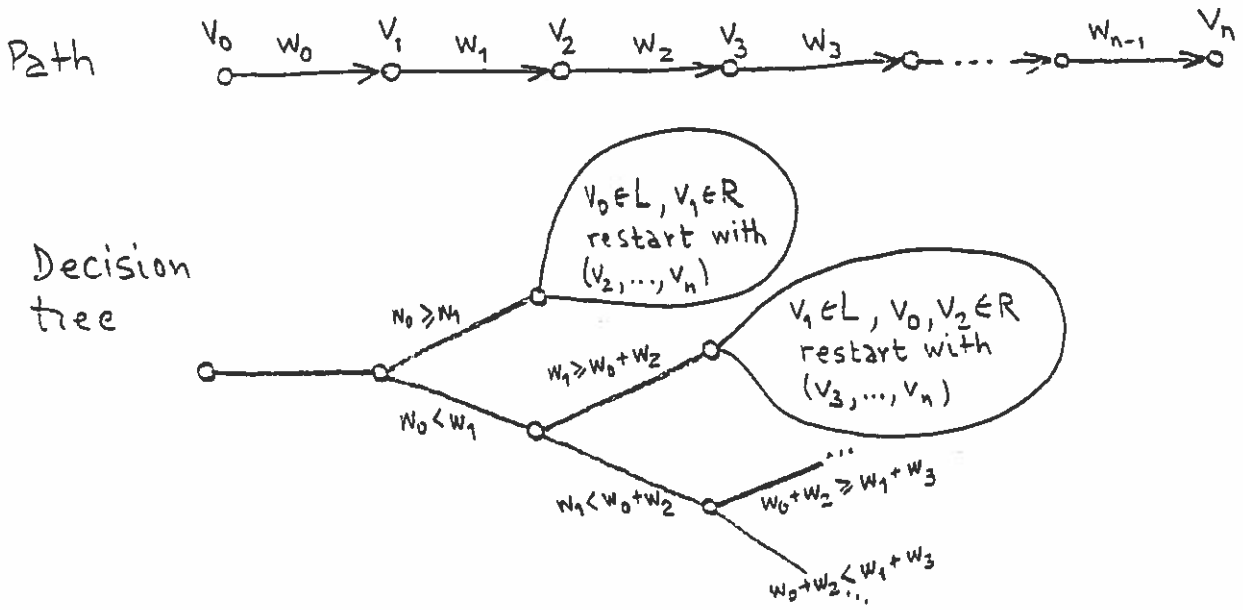

Figure 1 A unidirectional path $v_0 \ldots v_n$ and the corresponding decision tree.

When we assume that the path is no longer unidirectional, having arcs $\langle v_{i+1}, v_i \rangle$ as well as $\langle v_j, v_{j+1} \rangle$, for $0 \leq i \neq j < n$, a new situation can occur. It could become the case that $q[i] \geq p[i]$, in which case Q[i] can be put in L and P[i] in R. We maximize the contribution of edges up to v_i and yet are able to possibly acquire the weight of arc $\langle v_{i+1}, v_i \rangle$. Both old and new situations must be dealt with by our solution algorithm for directed trees, to be presented in the next section.

3. Directed Trees

The ideas introduced in our discussion of Algorithm 2.1 can be applied to design a linear-time, solution algorithm for trees. One main difference lies in the fact that we must consider contributions from several "explored" subtrees when pruning a current leaf vertex. This calls for an appropriate merging of

sets of considered vertices when processing what is now the root of the explored subtrees.

Another difference concerns the information that must be considered when making local decisions which are globally optimal. In a path, the "current vertex" separates the explored from the unexplored in the sense of complete information necessary to make a correct placement of the vertices. In a rooted tree, the separation is obtained by additionally considering the arc between the current vertex and its father. This frees the decision for the subtree rooted at the current vertex from undue influence of other subtrees of the father vertex. There are no such subtrees in a path and therefore the decision about the current vertex could be made when pruning the next vertex (i.e., the father).

Let us assume that a weighted, directed tree T is given by its recursive representation ([4], also "father array" in [3]). In this representation, each vertex v (except for the root r) has a unique arc assigned; namely, the arc between v and its father. Each such arc is directed either <u>toward</u>, or away <u>from</u> v; this together with the weight of the arc is the total information given about T at v. For the sake of uniformity we add to T an arc of weight 0 directed from the root to a dummy vertex.

<u>Algorithm 3.1</u> Solves the directed maximal-cut problem for trees.

Input: Recursive representation of a weighted, directed tree T of n vertices. For each vertex v_i there is its father f[i] (where $f[i]=v_j, j<1$), weight w[i], and direction d[i] of the arc between v_i and f[i].

Output: Partition (L,R) of the set of vertices of T maximizing the value of a directed cut (DMC2), and its value, C.

Method: With each vertex of T, v_i, associate the sets P[i] and Q[i] and the corresponding values p[i] and q[i]

```
(*Step 1.  Initialize*)
   for i:=1 to n do
   begin P[i]:={i}; Q[i];=Φ; p[i]:=0,q[i]:=0 end;
   L:=Φ; R:=Φ; C:=0;
(*Step 2.  Prune the leaf vertices*)
   for i:=n downto 1 do
   begin case d[i] of
(*2.1*)  from: if p[i]≥q[i]
(*2.1.1*)      then (*take p_i*) begin L:=L∪P[i]; R:=R∪Q[i];    C:=C+p[i];
(*2.1.1.2*)                      q[f[i]]:=q[f[i]]+w[i] end
(*2.1.2)       else (*cross the edge*)
                 begin p[i]:=p[i]+w[i];
                   if q[i]≥p[i]
(*2.1.2.1*)        then (*take q[i]*) begin L:L∪Q[i]; R:=R∪P[i]; C:=C+q[i] end
                   else (*defer decision*) update-parent(i)
                   end; (*of the arc away from v_i*)
(*2.2*)  toward:  if q[i]≥p[i]
(*2.2.1*)      then (*take q_i*) begin L:=L∪Q[i]; R:=R∪P[i]; C:=C+q[i];
                                       p[f[i]]:=p[f[i]]+w[i] end
(*2.2.2*)      else (*cross the edge*)
                 begin q[i]:=q[i]+w[i];
                 if p[i]≥q[i]
                 then (*take p_i*)
                 begin L:=L∪P[i]; R:=R∪Q[i]; C:=C+p[i] end
                 else (*defer decision*) update-parent(i)
                 end (*end of the arc toward v_i*)
         end; (*of processing the vertex v_i*)
```

The procedure update-parent is defined as follows:

```
procedure update-parent(i);
(*modifies the values associated with the father f_i of vertex v_i
when no decision could have been made while considering v_i*)
begin let j be such that v_j=f[i];
      P[j]:=P[j]∪Q[i]; Q[j]:=Q[j]∪P[i];
      p[j]:=p[j]+q[i]; q[j]:=q[j]+p[i] end;
```

In a given directed tree T, placing the leaves (vertices of degree one) into sets R or L is a simple matter. Every leaf that is a source should be in L and every leaf that is a sink in R. Only this placement assures that no pendant arc is excluded from being considered to contribute to the value of the cut. Of course, further consideration may force the other end-vertex of a pendant arc to be placed in the "wrong" set (L for a sink and R for a source). An arc is called pendant if it connects a leaf vertex with its only neighbor, a pendant vertex. We will show that Algorithm 3.1 places the leaves correctly and also that it passes the information about the pendant arcs onto the unprocessed part of T.

Lemma 3.2 For a given T, Algorithm 3.1 places leaves of T into the appropriate sets and passes the information about pendant arcs to pendant vertices of T.

Proof 3.2 Whenever v_i in Step 2.1 is a leaf-vertex of T, the condition $p[i] \geq q[i]$ is fulfilled because of the initialization step ($p[i]=0$ and $q[i]=0$). Therefore, if v_i is a source (d_i=from), Step 2.1.1.1 will be executed placing v_i in L and setting its father's q[j] to w[i]. No other value will be changed as $Q[i]=\Phi$ and $p[i]=0$. Assignment of w[i] to q[j] assures that the weight of the arc will be considered if v_j is to be placed in R. If v_i is a sink, similar assignments will

be made in Step 2.2.1.1. □

We now proceed to prune internal vertices of T (but leaves of the partial tree of unexplored vertices). We will show that if no decision about placement of vertices of a subtree S of T (except for its leaves) has been made, then the highest value of the cut for this subtree is attained if its vertices are placed consistently with the natural bipartition of S.

Every tree T=(V,A) is a bipartite graph (P,Q;A) such that P∪Q=V. We call the partition (P,Q) of V the natural bipartition of T. By the procedure update-parent of Algorithm 3.1, (P(x),Q(x)) is the natural bipartition of the subtree T_x of T rooted at x and containing only vertices not placed prior to pruning of x in the execution of the algorithm, with x∈P(x).

Lemma 3.3 Let a subtree S of a weighted, rooted directed tree T=(V,A) contain only vertices processed by Algorithm 3.1 and not placed prior to pruning of its root. Then a solution to the undirected maximum-cut problem for T preserves the natural bipartition (P,Q) of S, i.e., P⊆L & Q⊆R or vice versa.

Proof 3.3 By mathematical induction. If S consists of only one vertex, then the hypothesis is obviously true. Therefore, let us assume that it is true for all subtrees with less than k>1 vertices and consider a subtree S with k vertices. By contradiction, assume that (L,R) is a solution to the cut problem for T in which T_x is a smallest subtree of S (rooted in x) such that x and its father y in S are in the same set (L or R). The value of the cut (L,R) for T is the sum of a cut value V for T_x and weights of some arcs outside of T_x. By the inductive hypothesis and minimality of T_x, V results from a placement decision consistent with the natural bipartition of T_x. If <x,y>∈A then,

according to Step 2.1.1 of the algorithm and our assumption that no decision had been made when processing x, $p(x)<q(x)$. The decision has to be deferred also after considering the weight $w<x,y>$ (in Step 2.1.2 of the algorithm), and thus $q(x)<p(x) + w<x,y>$. If $x,y \in L$ then $V=p(x)$ but placing x and other vertices of P(x) into R would result in $V=q(x)$ and thus a greater cut value for T. If $x,y \in R$ then $V=q(x)$. Placing x (and other vertices of P(x)) into L would yield $V=p(x)$ and add $w<x,y>$ to the value of the cut, again increasing its value. Similar arguments (supported by Steps 2.2.1 and 2.2.2 of Algorithm 3.1) apply when $<y,x> \in A$. Thus, the proposed (L,R) cannot yield the maximum value of the cut. This contradicts our assumption and proves the Lemma. □

We have proved so far that as long as no decision has been made about placing of the vertices of T into sets L and R, Algorithm 3.1 carries along sufficient information about the natural bipartition of explored subtrees to make the correct local decision. Now we will prove that this local decision is also correct in the global sense.

Lemma 3.4 Let a subtree S of a weighted, rooted directed tree T=(V,A) contain only vertices processed by Algorithm 3.1 and placed during pruning of its root. Then a solution to the directed maximal-cut problem for T preserves this placement. The modification of parameter values of the father of the root of S allows determination of a global solution.

Proof 3.4 By Lemma 3.2, this is true if the subtree T consists of only one vertex, the leaf-vertex of the pruned tree. Assuming that the hypothesis is true for subtrees of less than k>1 vertices, consider

S with k vertices. The placement of vertices of S decided upon while pruning its root x, is consistent with the natural bipartition of S (as determined by the procedure update-parent). By Lemma 3.3, one of the two consistent placements is globally correct. By arguments similar to that used to prove Lemma 3.3, we show that the decision made at x maximizes the contribution of S to the global cut value. Assume that $\langle x,y\rangle \in A$, where y is the father of x in T. Then the decision is made either in Step 2.1.1, or in Step 2.1.2.1. If $p(x) \geq q(x)$ then placing P(x) in L will give the largest value of the cut. In this case, q(y) is incremented by $w\langle x,y\rangle$, which would be added to the value of the cut in the future if y were to be placed in R. If $p(x) < q(x)$ but $q(x) \geq p(x) + w\langle x,y\rangle$, then placing Q(x) in L and P(x) in R gives the best contribution of S to the value of the cut. The future placement of y is irrelevant, and, as such, there is no change in p(y) or q(y). If $\langle y,x\rangle \in A$, then a similar argument shows that decision made at x must be globally valid. □

The above lemmas and an efficient implementation of the algorithm imply the following theorem.

Theorem 3.5 Algorithm 3.1 computes a solution to the DMC2 problem in O(n) time for an arbitrary directed tree.

An efficient implementation of Algorithm 3.1 could represent the sets P(v) and Q(v) for $v \in T$ as linear lists allowing constant time concatenation and insertion operations.

4. General Directed Graphs

The above algorithm for directed trees is almost directly applicable to other directed, acyclic graphs (DAG's). In such graphs, sink- and source-vertices determine "tree components", for which solutions of the directed

maximal-cut problem (computed by Algorithm 3.1) contribute directly to the global solution for the DAG in question.

To deal with directed graphs containing strong (directed) cycles we introduce the notion of leaf constraints, which force leaf vertices of a tree into sets L or R. The presence of constraints would not change the operation of Algorithm 3.1. If a leaf is constrained, appropriate initialization of the values p and q associated with the vertex can force the desired placement. For example, a source leaf x will be forced into set R by presetting $p(x)$ to $-\infty$ and $q(x)$ to 0.

Assume we are given a directed graph G and a set of vertices B whose removal breaks all weak cycles of G. For each cycle-breaker $v \in B$, break G at v, making copies (or "clones") of v on each arc incident to v. As such, each clone of v is a leaf vertex in the resulting forest F. Figure 2 demonstrates this operation on a given directed graph.

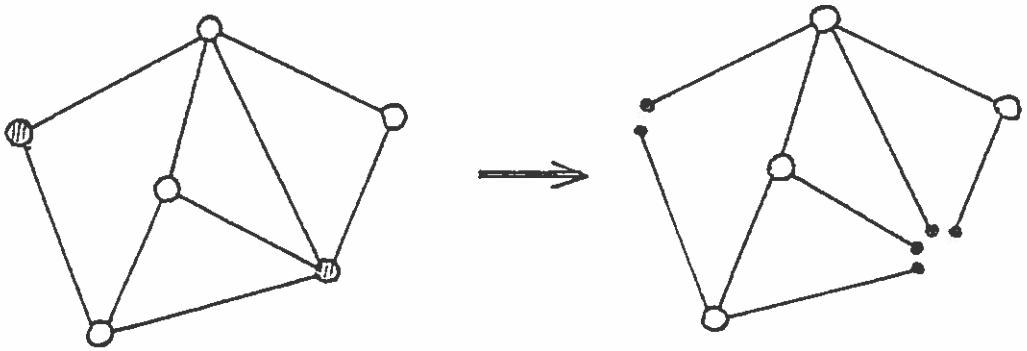

Figure 2 Cloning the cycle-breakers

To compute the value of a maximal directed cut of the original digraph G, we apply Algorithm 3.1 to F, while forcing all clones of a given cycle breaker v into the same partition. By repeated application of that linear time algorithm under all possible combinations of assignments for vertices of B, we obtain $2^{|B|}$ maximal cut values. The maximum among these with the corresponding assignments of vertices to sets L and R solves the directed maximal-cut

problem for graph G.

The complexity of our solution is obviously $O(n2^{|B|})$, where B is a minimum cycle breaking set of vertices of G. When the size of B is at most logarithmic in the number of vertices of G, the directed maximal-cut problem is solved in polynomial actually, (quadratic) time. A constant bound on $|B|$ yields a linear time solution. Thus, for the class of unicyclic graphs, selecting one vertex on the unique cycle and twice applying Algorithm 3.1 to the resultant forest is sufficient to determine a directed maximal-cut. Figure 3 illustrates the creation of a forest from a unicyclic graph.

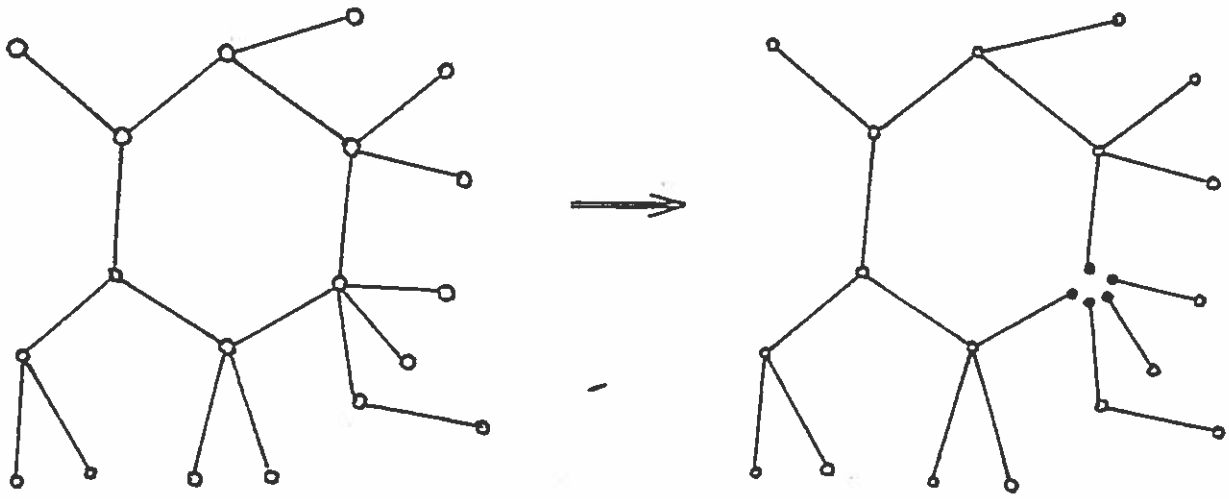

Figure 3 Breaking the cycle of a unicyclic graph.

Unfortunately, there exist classes of graphs having minimum cycle breaking sets which are linear in size with the number of vertices. Figure 4 presents an instance of the class of triangle-based cacti, each of which has a linear number of cycle-braking vertices. Another example of a class with large size of any cycle-breaking set is that of complete graphs, each having $|V|-2$ cycle breakers. The problem of determining a minimum cycle-breaking set for an arbitrary graph is NP-complete, being a hereditary property [2].

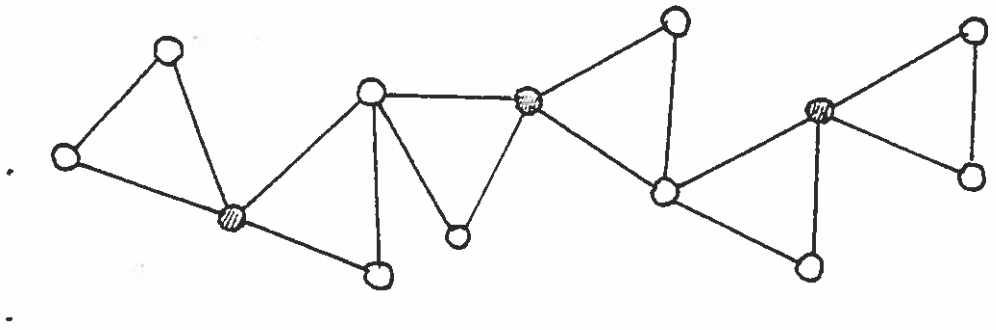

Figure 4 A graph with large cycle-breaking set.

5. Conclusion

In this paper we have considered a particular variant of the directed maximal-cut problem. The problem is known to be NP-complete in general. We first present an efficient algorithm, solving the problem for directed paths. We then generalize the path algorithm, proving correct a solution for directed trees. We discuss how the tree algorithm can be used to efficiently solve the problem for certain, more general classes of directed graphs.

Acknowledgement

The authors are indebted to teachings of Professor Moshe Rubinstein that persuaded them to draw the decision tree in Figure 1.

References

[1] A. Farley and A. Proskurowski: Two directed maximal-cut problems; to appear in Proc. of HSU Conference on Graph Theory, Combinatorics and Computing, Utilitas Mathematica.

[2] M.R. Garey and D.S. Johnson: Computers and Intractability, W.H. Freeman and Co., 1979.

[3] D.E. Knuth: Art of Computer Programming, vol. I, Addis on-Wesley, 1973.