Finding a negative Cycle in a directed Graph

Athanasios K. Tsakalidis

A 85/05

March 1985

Fachbereich 10, Angewandte Mathematik und Informatik
Universität des Saarlandes, 6600 Saarbrücken
West Germany

ABSTRACT: We present an algorithm implemented on a pointer machine which can find a negative Cycle in a directed graph in worst case Time $O(n.e)$, where n is the number of nodes and e the number of edges, using only linear Space $O(n+e)$. The best previous result was due to D.Maier [5]. His algorithm is running on a random access machine in worst case Time $O(n(e+n\log n/\log n\log n))$ and uses Space $O(e+n\log n/\log n\log n)$.

1. Introduction

We are given a digraph (V,E) where for each edge e in E we are given a cost $c(e)$ that ranges over the real numbers. The problem is to find if graph G has any negative cycle: a path from a node to itself where the sum of the cost of the edges along the path is negative. If such a cycle exists, we also want to find *one* such cycle.

According to D.Maier [5] this problem has application in the area of two dimensional package placement. Other applications can be found in the area of the minimal Cost-to-Time Ratio problem (see Lawler[3], p. 94) and of the Min-Cost Flow problem (see Papadimitriou and Steiglitz [6], p. 139). A further application is the solution of inequality systems relevant to VLSI layout problems (see Lengauer [4]). In [5] we can find a list of works related to the Negative Cycle problem. The most of them use methods which are modifications of the single-source shortest path problem. These methods exhibit two traits. First they detect the presence of negative cycles, but do not find one. Second, they achieve their worst case behavior in the presence of negative cycles. D.Maier [5] presents a method which doesn't share these shortcomings. His algorithm is running on a random access machine (RAM) (see [1]) in worst case Time $O(n(e+ n\log n/\log\log n))$ and uses Space $O(e + n\log n/\log n\log n)$. We present here an algorithm which improves the time to $O(n.e)$ using linear Space $O(n + e)$ and running on a pointer machine (PM).

A RAM-machine is well defined in Aho,Hopcroft, Ullman [1] and represents the implementations which are based on the using of *arrays* , i.e. in RAMs we can access directly by an adress which is computed by an arithmetic. In contrast to RAMs in a pointer machine we can only access a cell if its adresss is stored in another cell as a pointer, i.e in a pointer machine the memory cosists of a collection of cells and a PM represents the implementations which are based on *records*. Harel and Tarjan [2] have shown that a RAM machine is *pure stronger* than a pointer machine.

We use the same basic ideas of D. Maier [5] but a different auxiliary data structure and thus we can execute much more efficiently some elementary operations used in [4]. The auxiliary data structure is developed according to the results of A. Tsakalidis [7].

In section 2 we give some results proved in [7] and in section 3 the algorithm and its complexity.

2. Preliminaries

We base our analysis on the following theorem, proved by A. Tsakalidis [7].

Theorem 1.

We can perform a sequence of m arbitrary insertions and deletions at given positions in an initially empty tree of bounded degree in Time $O(m)$ and we can decide in Time $O(1)$ about the ancestor-relationship of two given nodes. The Space used is $O(n)$ where n is the number of nodes.

Proof: For the details of the proof we refer the reader to [7]. We give here only the main ideas. First an implementation of a list is given which can perform m arbitrary insertions and deletions in an initially empty list in time $O(m)$ and can answer in time $O(1)$ if a given element x lies on the left or on the right of another given element y. Second we represent a speciel traversal of the tree by a list K implemented as above where each element x is represented by two elements x_1 and x_2 in list K. Then a question about the ancestor-relationship between x and y can be answered by two questions on the topology between x_1, y_1 and x_2, y_2 ∎

The bounded degree of the tree structure is necessary for the linear time complexity in the case that the deletions are explicitly executed. Another alternative is to perform deletions by *marking* the nodes as deleted. Thus we get the following :

Theorem 2.

Performing only insertions or both, where deletions are implicitly performed by marking the nodes deleted, Theorem 1 is valid for an arbitrary tree structure also in the case of a pointer machine.

Proof: We have exactly to follow the construction of the lists used in Theorem 1 in [7]. There we can see that elements deleted by marking them don't influence the topology of the list elements and all questions required can be answered correctly. ∎

3. The Algorithm and its Complexity

We consider the case that we are given a source node s in V, from which all other nodes of digraph (V,E) are reachable. The method used here is based on the following ideas:

1) We use a breadth-first search strategy from s up and every time we are looking for paths with lesser cost to the nodes visited. It means that at each stage i, we look for paths of length i from the source s to the nodes visited at stage $i-1$ or less such that these paths are less costly than paths discovered so far.

2) During this process we change dynamically a tree T_g of arbitrary degree which is first initialized by s as root and at each stage i the path from s to node x corresponds to the least costly path found until the stage i in the graph from s to x. The tree T_g has to be changed every time nodes are first reached or less costly paths are found.

3) An existing negative cycle can be detected if we generate a less costly path from s to some node u passing through u earlier on the path. This fact can be detected in tree T_g by testing every time if we have to connect a node x with another node y which is ancestor of x in T_g.

4) We have to implement T_g according to the implementation of Theorem 2 (see [7]) and thus the necessary dynamic operations can be executed efficiently and every time a question about the ancestor-relationship of two given nodes in tree T_g can be answered in *constant* time.

Next we give the algorithm CYCLE and the data structure used. We don't use any array because we aim an implementation by a pointer machine and thus we connect the elements used only by pointers. We use the following Informations:

I.1) For every node v in V we keep a LIST[v] of all nodes u such that (v,u) in E. The node v is the *head* of LIST[v].

I.2) Every node u in LIST[v] is equipped with a pointer which points to the origin node u which is declared as the head of LIST[u].

I.3) Every node v is equipped with a pointer which points to the position of v in the tree T_g.

I.4) Every node x in T_g is equipped with two pointers which point to the elements x_1 and x_2 in the list K which is developed according to Theorem 2.

In the algorithm CYCLE below, the vector DIST holds the current lowest costs for paths from s to the nodes of V. We use also two sets U and U'. Set U holds the active nodes for the current iteration, i.e. all the nodes v at the stage i such that all elements of LIST[v] have to be visited and tested if they can decrease their distance from s. In the case that such a visited node u from LIST[v] decreases its distance from s it will be stored in the set U' in order to be handled as active node during the next iteration. At this point we have to make the following restriction: if nodes x and y belong into U and by handling x we have to consider the edge (x,y) then we store y in U' and delete y from U. This restriction enables us to keep as active nodes those nodes which have the same depth from s in the tree T_g.

For each node u which is inserted into U' we have to change its ancestor-relationship in the tree T_g, i.e. eventually u has to be inserted as a new son of a node v in T_g if the edge (v,u) has decreased DIST[u]. In this case we have to consider only the node u ignoring the subtree T_u of T_g which is rooted at u. All other nodes in T_u can be visited again if they produce less costly paths, because u will be considered in the set U during the next iteration.

In the following algorithm CYCLE we use a procedure Insert(x,y,T_g) which inserts the node x in the tree T_g as the rightmost son of node y and subsequently executes all necessary operations according to the implementation of T_g given by A. Tsakalidis [7]. Similarly we use the procedure Ancestor(x,y) which returns true iff x is an ancestor of y in tree T_g. This question can be answered in time $O(1)$ according to [7].

algorithm CYCLE

```
(1)  U:=s; DIST[s]:=0;
(2)  for all v≠s do DIST[v]:=∞ od:

(3)  while U≠ ∅
(4)   do U' := ∅
(5)      for each u in U
(6)       do if u has not been added to U'
(7)          then for each v in LIST[u]
(8)                 do if DIST[u] + c(u, v) < DIST[v]
(9)                    then DIST[v]:=DIST[u] + c(u, v)
(10)                        U' := U' ∪ {v}
(11)                        if Ancestor(v,u)=true
                             then construct NEG-CYCLE, a list of
                                  the nodes along the path
                                  in T_g from v to u
(12)                        else Insert(v,u,T_g)
                             fi
                    fi
                 od
          fi
       od
(13)   if U' = ∅ then return ('no negative Cycle')
(14)   U:=U'
     od
```

Theorem 3.

Given a directed graph (V,E) where for each edge d in E we are given a cost $c(d)$ that ranges over the real numbers. We can find in Time $O(n.e)$ a negative Cycle in the digraph, where n is the number of nodes and e the number of edges. Our algorithm is running on a pointer machine and the Space used is $O(n+e)$.

Proof: We use the algorithm CYCLE as it is given above according to the data structure with the Informations I.1) ... I.4). We have only to extend these Informations by I.5) in order to specify if a node u belongs into U'.

I.5) Every head-node u is equipped with a pointer which points to a cell storing boolean values. This cell is set to true if u belongs into U'.

Thus we have to change the Information I.5) at the line (10) if a node v is added to U' and at the line (4) in the case that U' is set to $\emptyset$ we have to change all the respective Informations I.5).

We consider first the total running time of line (12). This time is O(e) according to Theorem 2, since we have to insert at most e nodes into tree T_g. The subsequent operations in the implementation of T_g after inserting a node u and ignoring the subtree T_u can be performed in time $O(1)$, since we have only to connect the elements u_1 and u_2 in the list K (see theorem 1) by a pointer and the space of T_u can be used again.

A question of Ancestor(v,u) at line (11) can be answered in time $O(1)$ according to the implementation of T_g (see [7]) and hence whenever a node v is chosen at line (5) the rest time spent in lines (6) - (14) is $O(outdegree(v))$.

Hence the total worst case running time of the loop (5) - (14) is

$$n.\sum_{v\in V} O(outdegree(v)) = O(n.e)$$

Note that our algorithm stops at the time that a negative cycle is detected. The cost of the statement outside of the loop is clearly $O(n)$. According to the implementation given the Space used is $O(n+e)$ and the algorithm is running on a pointer machine. ∎

References

[1] A.Aho, J.Hopcroft, J.Ullman "Design and Analysis of Computer Algorithms", Addison Wesley (1974).

[2] D.Harel and R.E.Tarjan "Fast Algorithms for finding Nearest Common Ancestors", *SIAM Journal of Computing*, Vol. 13, pp. 338-355 (1984).

[3] E.L.Lawler "Combinatorial Optimization: Networks and Matroids", Holt, Rinehart and Winston (1976)

[4] T.Lengauer "Efficient Algorithms for the Constraint Generation for Intergrated Circuit Layout Compaction", *Proc. of 9-th Workshop on Graphtheoretic Concepts in Computer Science*, Hanser Verlag, pp. 219-230 (1983)

[5] D.Maier "An Efficient Method for Storing Ancestor Information in Trees", *SIAM Journal of Computing*, Vol. 8, No 4, pp. 599-618 (1979).

[6] Ch.Papadimitriou and K.Steiglitz "Combinatorial Optimization: Algorithms and Complexity", Prentice-Hall (1982)

[7] A.K.Tsakalidis "Maintaining Order in Generalized Linked List", *Acta Informatica 21*, pp. 101-112 (1984).