On-line Graph Algorithms for Incremental Compilation

Alberto Marchetti-Spaccamela* Umberto Nanni†

Hans Rohnert‡

TR-92-056

August 1992

(preliminary version)

Abstract

Compilers usually construct various data structures which often vary only slightly from compilation run to compilation run. This paper gives various solutions to the problems of quickly updating these data structures instead of building them from scratch each time. All problems we found can be reduced to graph problems. Specifically, we give algorithms for updating data structures for the problems of topological order, loop detection, and reachability from the start routine.

*Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy. This report was written while the author was visiting ICSI. The author's email address is: marchetti@vxscaq.infn.it.

†Dipartimento di Matematica Pura e Applicata, Università di L'Aquila, L'Aquila, Italy. This report was written while the author was visiting ICSI. The author's email address is: nanni@vxscaq.infn.it.

‡This report was written while the author was on leave from Siemens AG, Munich, Germany. The author's email address is: rohnert@ztivax.zfe.siemens.de.

1 Introduction

One way to speed up compilation times is to incorporate mechanisms for incremental change of important data structures into the compiler. In fact the first compilation run of a program is generally followed by several other compilation runs each time with a slightly changed program. Therefore it is desirable that data structures built during the first compilation run are maintained dynamically without building them from scratch during every subsequent run. Namely, after having made these data structures persistent on disk during the first compilation run, subsequent compilation runs read in these data structures again, modify them as much as necessary because of changes in the input program text, and write them back to disk storage. The data structures we have in mind can be modelled by directed graphs such as the call graph and, notably in object-oriented programs, the inheritance graph. Also, some type of consistency checks may be reduced to graph problems.

Specifically, we consider the following problems in directed graphs: loop detection, topological order and reachability from a start or "main" vertex. These problems are considered in their static and dynamic solutions: the dynamic operations considered are insertions and/or deletions of an edge/vertex of the graph. The accent here is on "practical" approaches, leading to algorithms and data structures that are simple and easy to implement.

We will often give two or even more solutions to a problem and the best solution for the problem at hand depends on its size and some more specific requirements. In particolar, different solutions for the same problem will have different space/time requirements. The consequent tradeoff's must be resolved on the basis of the available resources and the size of the input. In fact the size of the graphs are sometimes fairly small or of medium size (the inheritance graph is seldom really large), but they can reach a considerable size (the call graph of large systems may have many thousands of vertices). Another issue is the tradeoff between the costs for updating and querying the data structures. In fact we will see that some of the solutions will have a high cost for updating and a small cost for querying; in this case the choice of the best solution will be influenced by the relative frequency of these operations.

Finally, another intriguing issue, while dealing with incremental algorithms, is the evaluation of time performances. The time cost of a dynamic algorithm may be expressed in terms of *worst case* cost (i.e. an upper bound to the time spent for a single operation performed on the data structure), or *amortized* cost [11], which provides the average time spent per operation over the worst possible *sequence* of operations. A possible way to evaluate such a cost is to bound the worst case *total*

cost of any sequence of operations and then dividing it by the length of the sequence. Note that the amortized analysis does not use any probabilistic assumption on the distribution of the possible operations.

The rest of the paper is organized as follows. In section 2 we introduce the terminology and we give preliminary definitions. In sections 3 and 4 we will consider the problems of performing loop detection and topological sort in a graph that can change under sequence of insert and delete operations. Finally in sections 4 and 5 we will consider the problems of reachability from a start vertex and of maintaining transitive closure.

Since one of the purposes of this document is to serve as a collection of optimization hints for compiler writers, some of the presented solutions and techniques are already known in the literature [3, 4, 6, 7, 2] and are described here just for the sake of completeness. We present original contributions from the algorithmic point of view in Section4, with the dynamic maintenance of topological ordering; in Section6, with a parallel implementation of an algorithm for the dynamic transitive closure.

Furthermore some of the previously known techniques (or similar algorithms matching existing bounds) have been proposed with the target of a practical and simple implementation.

2 Preliminaries

In the following we assume the standard graph theoretical terminology as contained for example in [1, 10]. A *directed graph* $G = (V, E)$ (also referred to as a *digraph*) consists of a finite set V of *vertices* and a finite set of ordered pairs $E \subseteq V \times V$ of *edges*. A *path* from v_1 to v_k is a sequence $\langle v_1, v_2, \ldots v_k \rangle$ of vertices such that any adjacent pair (v_h, v_{h+1}) is an edge in E. A *cycle* is a nonempty path from a vertex to itself. A digraph with no cycles is called a *directed acyclic graph* (in short *dag*). A vertex y is said to be *reachable* from a vertex x if there exists a path from x to y. We will denote as $Reach(v)$ the set of vertices reachable from a vertex v (including v itself).

If $G = (V, E)$ is a digraph, the digraph $G^+ = (V, E^+)$ such that an edge from $(x, y) \in E^+$ if and only if there exists a nonempty path from x to y in G, is called the *transitive closure* of G.

In this paper we assume that graph are represented by storing the edges in adjacency lists, that is using space $O(n + m)$.

We consider several problems on directed acyclic graphs both in the static and the dynamic version. In a static problem we find a solution S for a given problem for a specified graph G. If the

graph is updated to a new graph G', the old solution S is no longer valid and we must recompute the new solution S'. In a typical dynamic situation, we want to maintain a data structure under a sequence of two kinds of operations: *query* and *update*. The update operation will modify the graph by inserting or deleting an edge or a vertex. The query operation requires to find an answer to the particular problem. The trivial approach to this kind of problems consists in using the static algorithm either when a query has to be answered, or when an update occurs.

In this paper we study algorithms and data structures that allow to obtain the new solution S' without recomputing it from scratch. The time and space requirements to maintain a solution to a given problem depend on the kind of allowed updates, for example whether we can only add edges to the graph, or also delete edges, or if we consider a fully dynamic situation in which insertions and deletions are arbitrarily performed. In the former cases we can often gain advantage by the natural monotonicity of the solution.

In the rest of the paper we will concentrate our attention on the insertions and deletions of edge because this is the most interesting problem. In fact we will see that the insertion/deletion of an isolated vertex (i. e. a vertex with no ingoing or outgoing edges) does not pose any problem. Therefore the deletion of a vertex can be performed by executing a sequence of edge deletions followed by an update operation that deletes a vertex with noedges. The case of vertex insertion can be dealt analougously. Furthermore we do not consider the problem of of updating the representation of a graph after insertion and deletion of edges, but only the updates to the structures handled by our algorithms.

3 Loop Detection

The first problem that we consider is *loop detection*, that consists in finding whether a given graph is acyclic or not. An example where it is necessary to check a graph for the existence of loops, is the inheritance graph of programs written in object-oriented languages, which has to be a directed acyclic graph. The introduction and the deletion of new classes and new inheritance relationships between classes give raise to the problem of repeatedly checking the acyclicity of a dynamically changing graph.

Given a graph $G = (V, E)$ with vertex set V and edge set E it is well-known how to test G for cycles: Simply run a topological sort [6]. If the algorithm terminates and assigns a topological number to every vertex the graph is acyclic. If it aborts prematurely because in the remaining graph there are no more vertices with no ingoing edges the graph contains cycles. The time complexity of

this procedure is $O(n+m)$ where $n = |V|$ and $m = |E|$. The space complexity is also $O(n+m)$, just for storing G.

Dynamic Loop Detection

Given an acyclic graph $G = (V, E)$ we deal now with the problem of performing an intermixed sequence of operations of the followind kinds:

a) updates to the structure of the graph, consisting in *insertion* or *deletion* of edges;

b) *queries* concerning the existence of loops;

c) *test* whether the insertion of an edge (u, v) would introduce loops.

Observe that the deletion of an edge e from $G = (V, E)$ is trivial since this operation can not introduce cycles. Furthermore insertions or deletions of isolated vertices require just to handle entries in the tables representing the graph. Therefore, in the sequel of this section, we concentrate on the operation of inserting an edge $e = (v, w)$, with $v, w \in V$.

Trivial solutions to dynamic loop detection consist in applying the $O(m+n)$ algorithm to compute a topological sort of the vertex set either when a query or test is requested, or when an update occurs.

Another simple solution, usually better than performing a topological sorting, consists in performing a depth first search from vertex w. If it reaches v then adding e introduces a cycle into G, otherwise is does not. The time complexity for $dfs(w)$ is $O(|G_{Reach(w)}|)$ where $Reach(w)$ is the set of vertices reachable from w and $G_{Reach(w)}$ is the subgraph induced by $Reach(w)$. In the worst case this means $O(m)$. The space complexity of $dfs(w)$ is $O(n)$ for storing a marker array and the call stack of the recursive *dfs* procedure.

As far as a really dynamic solution for the loop detection problem is concerned, it can be reconducted to the dynamic maintenance of a topological sorting, that is considered in section 4: as shown there, it will require $O(m \cdot \min\{n, k\})$ total time to perform any sequence of k consecutive edge insertions (that leave the graph acyclic) followed by a sequence of h edge deletions. If we start from an empty graph the insertion of m edges requires $O(n)$ amortized time per update, which is a considerable improvement over the $O(m)$ static algorithm, especially for long sequences of insertion in a dense graph. In the worst case a single insert operation may require $O(m)$ time. This solution is space efficient since it requires $O(n)$ additional space beside the space necessary for storing the graph.

Using the above solution, to perform a *test(u,v)* operation (testing whether a new possible edge (u, w) leaves the graph acyclic or not) requires $O(m)$ worst case time (by using dfs, for example), since the cost can not be amortized over a sequence of operations.

It is possible to improve the time requirements of the test operation at the expenses of space requirements. In fact testing in costant time is indeed possible if we maintain the transitive closure G^+ of the graph G. In fact the insertion of the edge (u, w) leaves the graph acyclic if and only if there is no connection from w to u in G^+, and this can be checked in constant time. This solution requires more space to be maintained, since we have to store the whole $n \times n$ matrices describing $G+$. This leads to $O(n^2) \geq O(m)$ space for a simple matrix implementation, that can be reduced for sparse graph storing only the meaningful (non-empty) items of G^+ by using, for example, perfect hashing [2].

Nevertheless the maintenance of transitive closure might be preferred in some context, and it is considered in section 6.

4 Maintaining Topological Order

Sometimes it is sufficient a partial information about the ordering among the vertices of a directed acyclic graph. A *topological order* among the vertices of a dag with n vertices maps each vertex v to an integer $ord(v)$ between 1 and n which fullfills the following property: for each $(x, y) \in E$ we have $ord(x) < ord(y)$. As shown in the previous section of this paper, finding (or maintaining) a topological order may provide information about the acyclicity of the graph. A description of the simple static solution of the topological sort can be found, for example, in [1, 5].

In this section we consider the problem of maintaining a topological order of a directed acyclic graph. We assume that the topological order is stored in array A and we show how to maintain A under insertions and deletions of edges. Afterwards we briefly discuss an alternative implementation using doubly linked lists.

Let $G = (V, E)$ be a directed acyclic graph and let A be an array representing a topological order of the vertices of G in the obvious way: $A[i] = v$ if $ord(v) = i$ (i. e. vertex v is the i-th vertex in the topological order). We say that vertex x follows vertex y in ord if $ord(x) > ord(y)$; equivalently we also say that y precedes x in ord.

Deleting an edge does not pose a problem for maintaining the topological order of a directed graph, and so there is nothing to do in this case. Edge insertion is computationally more complicated; the trivial way to perform the insertion is to first check whether the new edge does not introduce a

cycle and then to recompute the topological order from scratch. In the worst case the running time is $O(m)$.

In the following we propose an algorithm that avoids the recomputation of the topological order from scratch. In the worst case the algorithm has the same time complexity of the procedure that computes a new topological order form scratch. However we claim that the procedure is often faster than recomputing the full topological order from scratch again. To substantiate this claim we will show that the total time spent for performing a sequence of m edge insertions is $O(mn)$, thus improving on the $O(m^2)$ bound obtained by performing m topological orderings of the graph. Equivalently the cost of an insertion amortized over a sequence of $O(m)$ insertions is only $O(n)$ (see [11] for examples of amortized analysis). The space requirements are favourable: the algorithm uses $O(n+m)$ space for the representation of the graph and $O(n)$ space to store the topological order; $O(n)$ additional space is required during the execution of the update operation.

Given a graph G let *ord* be a topological ordering of its vertices and let $G' = (V, E \cup \{(u,v)\})$ where $(u,v) \notin E$. If u precedes v, i.e., $ord(u) < ord(v)$, there is nothing to do since the new graph complies to the existing topological order. On the other side if u follows v in the topological order, then the new edge introduces a new path from u to v and, hence the topological order needs to be modified. The proposed algorithm finds a topological order ord' that is obtained from *ord*.

Lemma 1 *Given a graph $G = (V,E)$ and a topological order ord of its vertices, let $G' = (V, E \cup \{(u,v)\}$ be the graph obtained from G inserting edge (u,v). If G' is acyclic, then there is a topological order ord' of G such that $ord'(w) = ord(w)$ for all w such that either $ord(w) < ord(v)$ or $ord(w) > ord(u)$.*

Proof Without loss of generality we can assume that u follows v in *ord* (otherwise the lemma is trivially true since *ord* is a topological order of G').

Let w be a vertex that precedes v (follows u) in the topological order *ord*. To prove the lemma it is sufficient to show that the set of predecessors (successors) of w is not modified by the insertion of the new edge.

If w is a vertex such that $ord(w) < ord(v)$ then there is no path from v to w in G. Hence the insertion of the edge (u,v) does not introduce new paths from any vertex of the graph to w and, therefore, the set of successors of w will not be modified by the inserted edge.

Analougously, if $ord(w) > ord(u)$ then there is no path in G from w to u and to vertices that precede u in the topological order *ord*. Hence the insertion of the edge (u,v) does not introduce new

paths from w to any other vertex of the graph and we can conclude that if G' is acyclic then there must be a topological ordering ord' such that $ord'(w) = ord(w)$. □

As a consequence of the lemma it is sufficient to focus on vertices w such that $ord(v) \leq ord(w) \leq ord(u)$; in fact these are the only vertices that may eventually have a different position in ord and ord'. In order to obtain ord' it is necessary to modify the position of these vertices in such a way that v and all vertices in $Reach(v)$ follow u and all vertices in $Reach_{\text{INV}}(v)$ in the new topological order. These modifications must be done without violating any order relationship in the graph before the insertion of edge (u, v). In order to perform this task we start a depth first search *dfs* starting from v. If $ord(u) > ord(v)$ then v and all its successors, i.e., all vertices in $Reach(v)$, have to be moved after of u in the new topological order (if they are not yet already after u). Those vertices which were in the topological order between v and u (inclusive u) and are not successors of v, have to be shifted ahead while preserving their relative ordering to make room for v and its successors. The details are as follows.

The algorithm consists of two stages. In the first stage it detects whether the new edge introduces a cycle or not and marks those vertices that in the new topological ordering must follow u. The second stage is performed only if a cycle has not been detected and computes the new arrangement.

The first stage consists of a *dfs* starting from v to explore $Reach(v)$. The individual *dfs* invocations stop when hitting a sink with no outgoing edges or when hitting a vertex x with $ord(x) > ord(u)$. Such a vertex x is already after u in the topological order, therefore, by lemma 1, x and its successors do not have to be considered since they already are after u (and therefore we do not need to modify their positions in the topological order for the insertion of edge (u, v)). During the depth first search the algorithm marks the components of the array A that correspond to visited vertices that in the topological order precede u. The set of marked vertices includes v and corresponds to the set that must follow u in the new topological ordering ord'. If the depth first search hits vertex u, then edge (u, v) introduces a loop and the procedure stops, since there is no way to define a topological order.

If a cycle has not been detected, the second stage of the algorithm updates the array A. The new order ord' is obtained from ord in such a way that u and all its successors will follow v without modifying their relative order.

The second stage is implemented by scanning the array A that stores the topological ordering from $A[ord(v)]$ to $A[ord(u)]$. We use a counter *shift* initialized at 0. When the algorithm considers component i of the array A it distinguishes between two possibilities. If $A[i]$ has been marked during

the first stage then the counter *shift* is increased by one. The corresponding vertex, $A[i]$, is one of the vertices to be moved after u and is inserted into an ordered list.

If $A[i]$ has not been marked during the first stage then the algorithm moves ahead in the topological order the corresponding vertex by *shift* many places decreasing its entry in by *shift* (i.e. $A[i-shift] := A[i]$). When the algorithm considers the component relative to vertex u (i.e. $A[ord(u)]$) then the counter *shift* is equal to the number of marked vertices during the first stage. Therefore moving ahead vertex u there is room in the array for placing the marked vertices. These vertices are inserted into the array preserving their relative ordering (which was consistent with a full topological ordering of the graph before the insertion of (u, v)).

Notice that if we maintain the topological order using a doubly linked list A, then it is not necessary additional space (beside the queue used by *dfs*). In fact, during stage 1, the visited vertices can be eliminated from the original list A and enqueued to form a new list L. Stage 2 consists in inserting the new list L within A just after the vertex u.

As far as the time complexity of the insertion algorithm we first observe that the running time of the second stage is proportional to the number of vertices with *ord* numbers between $ord(v)$ and $ord(u)$; therefore, the worst case cost of the second stage is $O(n)$. On the other side the time complexity of the first stage is proportional to the total degree of the successors of v to be moved after u plus; this in the worst case can be $O(m)$. Therefore $O(m)$ is the worst case complexity of the procedure for maintaining the topological order under edge insertions.

We now show that the total time spent for performing any sequence of k edge insertions is $O(m \cdot \min\{n, k\})$ where m is the total number of edges in the graph after the insertions have been performed.

Theorem 2 *The total running time necessary for performing a sequence of m dfs edge insertions starting from a graph with no edges is $O(mn)$.*

Proof Since the running time of the second stage is $O(n)$ then, in order to prove the theorem it is sufficient to show that the running time of the first stage over a sequence of m edge insertions is $O(mn)$. In order to prove this claim it is sufficient to show that, during any sequence of m edge insertions, each edge of the graph is scanned at most n times.

Each time edge (x, y) is visited by the *dfs* starting in v due to the insertion of edge (u, v), it will be labelled u. Hence an edge (x, y) is *labelled* u if there exists a vertex v such that before inserting

```
1.  Procedure INS((u,v) : edge);
2.  begin
        Stage 1
3.          Perform a DFS search starting from v
            without continuing the search from a vertex w
            such that ord(w) > ord(u)
            and
            by marking components of A that correspond to
            visited vertices such that ord(w) > ord(u);
4.      if vertex u has been visited
5.          then HALT {a cycle has been detected}
6.          else
                Stage 2
7.              begin
8.                  shift := 0;
9.                  let L be an empty list;
10.                 for i := A[ord(v)] to A[ord(u)]
11.                     do if A[i] has been marked during stage 1
12.                         then begin
13.                                  insert A[i] at the end of L;
14.                                  shift := shift + 1
15.                                  end
16.                         else A[i - shift] := A[i];
17.                 insert vertices belonging to the list L in the array A
                    using components from A[i - shift + 1] to A[i]
18.             end
19. end;
```

Figure 1: Maintaining Topological sort while inserting edges

edge (u, v):

$$x \in Reach(v) \text{ and } ord(u) \geq ord(x) \quad (1)$$

Since each time an edge is scanned it gets a label, it follows that the total number of labels assigned is proportional to the running time required for the m *dfs* searchs.

We now show that for any graph G and any initial topological ordering, each edge (x, y) is labelled u at most once during any sequence of edge insertions. Clearly this latter claim implies the theorem.

Assume that (x, y) is labelled u during the insertion of edge (u, v); we show that any successive insertion of an edge (u, v') will not label again vertex u. In fact, if edge (x, y) is labeled u during the insertion of (u, v), then there is a path from u to x and thereafter $ord(u) < ord(x)$. This implies that any successive edge insertion (u, w) cannot label u edge (x, y) anymore because this would violate condition 1. □

Note that the above theorem is not true if edge deletions occur during the sequence of insertions. This is due to the fact that we cannot bound anymore the number of times that an edge is scanned, as shown in the following example.

Let $G_i = (V_i, E_i)$, for $i = 1, 2$, be a dag and $s_i, p_i \in V_i$ be two vertices such that for any $x \in V_i$, we have $x \in Reach(s_i)$, and $p_i \in Reach(x)$. Let now be $G = (V_1 \cup V_2, E_1 \cup E_2)$, and consider the following sequence of updates of G that inserts and deletes edges (p_1, s_2) and (p_2, s_1) alternatively:

$$insert(p_1, s_2),\ delete(p_1, s_2),\ insert(p_2, s_1),\ delete(p_2, s_1),\ insert(p_1, s_2),\ delete(p_1, s_2),\ \ldots$$

It is easy to see that the above sequence maintains the graph acyclic and that the proposed algorithm for edge insertions searches the whole graph for any two successive *insert* operations.

This example also shows the intrinsic hardness of this problem. In fact if a topological order has to be stored in an array or, equivalently, if the complexity is computed in terms of the number of elements which change their position, a sequence of q updates alternating insertion and deletion of edges may require $\Omega(qn)$ operations.

5 Maintaining Reachability from Start Vertex

Another question often checked in compilers is whether a specific routine is reachable via routine calls from the start routine M or which routines overall are reachable from M. The same problem arises in garbage collectors which try to identify data blocks no more reachable by a chain of pointers

from one or at least one of several starting pointers. This problem will be referred as the *single source reachability* or, in short, *reachability.*

A generalization of this problem is the *transitive closure* of a directed graph, that is the problem of maintaining the reachability between any pair of vertices in the graph. This is considered in section 6.

The evident static solution to the reachability problem is to perform a graph search such as *dfs* starting in start vertex s and to check which vertices are reached. The time complexity of this approach is obviously $O(|G_{Reach(s)}|) \leq O(|G|)$.

Dynamic Reachability

A very efficient dynamic solution for this problem is possible in case of either sequences of edges insertion, or sequences of deletions in a dag. The solution proposed here is similar to the one proposed in [7], and has the same time and space complexity of [3, 4, 9].

The performances of the algorithms are the following:

- $O(m)$ time for any sequence of edge insertions, that is costant amortized time per edge insertion starting from an empty graph;
- $O(m)$ time for any sequence of edge deletions, that is costant amortized time per edge deletion finishing with the empty graph;
- constant time to test whether a given vertex v is reachable from the "main" vertex M.

The extra space required by the algorithm presented here is $O(n)$, namely a counter $C[i]$ for any vertex $i \in V$. The procedures to perform insertion and deletion of an edge (i, j) are pseudocoded in Figures 2 and 3 respectively.

The counter $C[h]$, relative to the vertex h, is equal to the number of edges entering in h and coming from a vertex which is in $Reach(M)$:

$$C[h] = |\{(x, h) \mid x \in Reach(M)\}|$$

The graph is supposed to be stored by using adjacency lists for each vertex. In the algorithms the adjacency list of the vertex h is referred to as `OUT-LIST`$[h]$.

The initialization required before using the algorithms and starting from the empty graph is the following:

$$C[M] = 1$$

```
1.  Procedure INS(i, j : vertex);
2.  begin
3.      if C[i] > 0
4.          then begin
5.                  Set-queue (Q, {j})
6.                  while Q not empty
7.                      do begin
8.                              dequeue (Q, h)
9.                              increment C[h]
10.                             if C[h] = 1
11.                                 then for each (h, y) ∈ OUT-LIST[h]
12.                                              do enqueue(Q, y)
13.                         end;
14.             end;
15. end;
```

Figure 2: Maintaining $Reach(M)$ while inserting edges

```
1.  Procedure DEL(i, j : vertex);
2.  begin
3.      if C[i] > 0
4.          then begin
5.                  Set-queue (Q, {j})
6.                  while Q not empty
7.                      do begin
8.                              dequeue (Q, h)
9.                              decrement C[h]
10.                             if C[h] = 0
11.                                 then for each (h, y) ∈ OUT-LIST[h]
12.                                              do enqueue(Q, y)
13.                         end;
14.             end;
15. end;
```

Figure 3: Maintaining $Reach(M)$ while deleting edges

$$C[h] = 0 \quad \text{for any} \quad h \neq M.$$

The algorithms are to be used in order to update the counters $C[h]$ when an edge insertion or deletion occurs.

The behavior of algorithm INS is the following. When an edge (i, j) is inserted, if the vertex i is not reachable from M, no further work is required (line 3). Otherwise the vertex j is inserted in the queue Q (line 5). Thereafter a vertex y is enqueued (lines 11, 12) if and only if there exist an edge (h, y) such that h has become reachable from M due to the insertion of the new edge (i, j) (lines 9, 10). On the other side, any time a vertex h is dequeued (line 8), the counter $C[h]$ is increased by one (line 9). The correctness of the algorithm can be proved by induction on the number of the iteration of the *while* loop.

As far as the time complexity is concerned, we observe that for any enqueued vertex we perform a costant number of operations. Since any time a vertex is enqueued its counter is increased by one, the time spent by the procedure in a sequence of edge insertions is equal to the total sum of the counters, and this is bounded by the number of edges in the graph. The additional space required, as noted before, is $O(n)$.

The procedure DEL is simmetrical to the previous one and its correctness and time complexity are not discussed. Though, it is worth to notice that procedure INS retains its validity while inserting edges in any directed graph, while DEL can be used only in acyclic graphs.

Efficient deletion of edges in a cyclic directed graph is indeed a hard problem that still does not have a practical solution. The problem is addressed in [9].

More details about the proof of correctness and complexity of the proposed algorithms can be found in [7].

6 Dynamic Transitive Closure

In this section we briefly describe the implementation of a simple and fast algorithm to maintain the transitive closure G^+ of a graph G during insertion and deletion of edges. The graph G^+ has an edge (x, y) if and only if there is a path from x to y in G. This problem may be interesting per se and furthermore, as remarked in section 3, maintaining explicitly the transitive closure of a graph, allows us to test in constant time whether the insertion of a given edge (i, j) would leave the graph acyclic. If this is a dominant operation in a given appication, it might be worthly to spend some additional space for the implementation of this algorithms.

It is possible a straightforward generalization of algorithms and data structures described in the previous section. In the following we present the algorithm also in a simple parallel version which has the advantage to be perfectly suited to be easily adapted on any number of processors from 1 to n, where n is the number of vertices in G.

The main data structure presented here requires additional space $O(n^2)$ to store the $n \times n$ matrix of counters, which can be reduced to $O(|E^+|)$, that is to the number of edges in G^+ by representing only the non-zero values of the matrix. It is worth notice, for the parallel implementation, that any processor will have access to a separate set of elements of this matrix (namely a row). So there is no need of memory sharing nor interprocessor communication.

For what concerning the time required to update the data structures during insertions and deletions the sequential implementation achieves these bounds (the complexity of the parallel implementation is discussed below):

- $O(nm)$ worst case total time for any sequence of edge insertions, that is $O(n)$ per insertion starting from the initial graph;
- $O(nm)$ worst case total time for any sequence of edge deletions, that is $O(n)$ per deletion in a sequence that leaves the graph empty.
- testing whether there is a path from any vertex x to any vertex y (testing whether a new edge (y, x) would introduce a cycle in the graph) requires constant time (a look up in a table).

The pseudocode of the two procedures for insertion and deletion of an edge (i, j) is given in Figures 4 and 5 respectively. From now on C will denote a matrix such that, for any $x, y \in V$, $C[x, y]$ is equal to the number of edges entering in y and coming from a vertex which is reachable from x (including the vertex x itself):

$$C[x, y] = |\{(z, y) \mid z \in Reach(x)\}|$$

Procedure `PAR-INS`, handling the insertion of an edge, uses a procedure `CLOS` requiring two parameters: the first one is the vertex k whose closure has to be propagated, and the second one is the first vertex to be inserted in the queue (namely the head of the new inserted edge).

If the n iterations of the **pardo** in line 3 are implemented on n different processors, any processor P_k it is in charge to update the set $Reach(k)$, i.e. to increment all the counters $C[k, x]$ for any x.

```
1.  Procedure PAR-INS(i, j : vertex);
2.  begin
3.      for each {vertex} k ∈ N pardo
4.          if C[k, i] > 0 then CLOS(k, j);
5.  end;

1.  Procedure CLOS(k, j : vertex);
2.  begin
3.      Set-queue (Q_k, {j});
4.      while Q_k not empty
5.          do begin
6.                  dequeue (Q_k, h);
7.                  increment C[k, h];
8.                  if C[k, h] = 1
9.                  then for each (h, y) ∈ OUT-LIST[h]
10.                         do enqueue(Q_k, y)
11.             end;
12. end;
```

Figure 4: Parallel transitive closure: insertion of an edge.

```
1.  Procedure PAR-DEL(i, j : vertex);
2.  begin
3.      for each {vertex} k ∈ N pardo
4.          if C[k, i] > 0 then DECLOS(k, j);
5.  end;

1.  Procedure DECLOS(k, j : vertex);
2.  begin
3.      Set-queue (Q_k, {j});
4.      while Q_k not empty
5.          do begin
6.                  dequeue (Q_k, h);
7.                  decrement C[k, h];
8.                  if C[k, h] = 0
9.                  then for each (h, y) ∈ OUT-LIST[h]
10.                         do enqueue(Q_k, y)
11.             end;
12. end;
```

Figure 5: Parallel transitive closure: deletion of an edge.

Using p processors, any of them would be in charge to update the closure of $\lceil n/p \rceil$ vertices. With $p = 1$ we get the sequential implementation.

Using n $[k]$ processors the work load for any of them in a sequence of edge insertions or edge deletions is $O(m)$ $[O(m \cdot \lceil n/k \rceil)]$. But, as far as the time bound per update is concerned, for any edge insertion we have to wait for the last processor which could take $O(m)$ time, but also in the worst case the time for any single update is always bounded by $O(|G_{Reach(j)}|)$.

7 Conclusions

In this paper we have studied the problem of dynamically maintaining data structures that are built during compiling runs. The proposed solutions show that in several cases it is possible to maintain these data structures under a sequence of update operations without building them form scratch, thus speeding successive compilation runs.

The proposed data structures and algorithms are simple and easy to implement. We have seen that for some problems several solutions are possible that have different time/memory requirements. The choice of the particular solution will be based on the available resources and their cost.

Many more questions need to be further investigated. As an example in the context of object-oriented programs it is interesting to find the vertex that is the least common ancestor of two vertices x and y (e.g., two vertices in the inheritance dag) or if they have common descendents. Static solution for these problems are known but efficiently dynamic solutions still need further investigation.

8 Acknowledgements

The above problems were originally suggested by Stephen Omohundro in the context of defining the new version 1.0 of the object-oriented language Sather and designing its compiler [8].

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] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. Technical Report Bericht Nr. 77, Fachbereich Mathematik-Informatik, Universität-Gesamthochschule Paderborn, 4790 Paderborn, Germany, January 1991, also presented at FOCS '88.

[3] G. F. Italiano. Amortized efficiency of a path retrieval data structure. *Theoret. Comput. Sci.*, 48:273–281, 1986.

[4] G. F. Italiano. Finding paths and deleting edges in directed acyclic graphs. *Inform. Process. Lett.*, 28:5–11, 1988.

[5] J. A. McHugh. *Algorithmic Graph Theory.* Prentice Hall, 1990.

[6] K. Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms.* Springer-Verlag, Berlin Heidelberg New York Tokyo, 1984.

[7] U. Nanni and P. Terrevoli. A fully dynamic data structure for path expressions on dags. *R.A.I.R.O. Theorical Informatics and Applications*, 1992. to appear.

[8] S. M. Omohundro, C. Lim, and J. Bilmes. The sather language compiler/debugger implementation. Technical Report TR-92-017, International Computer Science Institute, Berkeley, Ca., 1992.

[9] J. A. La Poutré and J. van Leeuwen. Maintenance of transitive closure and transitive reduction of graphs. In *Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, 314, pages 106–120. Springer-Verlag, 1988.

[10] R. E. Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* SIAM, 1983.

[11] R. E. Tarjan. Amortized computational complexity. *SIAM J. Alg. Disc. Meth.*, 6:306–318, 1985.