Engineering DFS-Based Graph Algorithms

Kurt Mehlhorn[*] Stefan Näher[†] and Peter Sanders[‡]

October 1, 2007

Abstract

Depth-first search (DFS) is the basis for many efficient graph algorithms. We introduce general techniques for efficient implementations of DFS-based graph algorithms and exemplify them on three algorithms for computing strongly connected components. The techniques lead to speed-ups by a factor of two to three compared to the implementations provided by LEDA and BOOST. We have obtained similar speed-ups for biconnected components algorithms. We also compare the graph data types of LEDA and BOOST.

1 Introduction

Depth-first search (DFS) is the basis for many efficient graph algorithms. We introduce general techniques for the efficient implementation of DFS-based graph algorithms and exemplify them on three algorithms for strongly connected components of digraphs. The techniques lead to speed-ups by a factor of two to three compared to the implementations provided by LEDA [MN99, LED] and BGL (BOOST Graph Library) [Boo], see Figure 1. The techniques apply to all DFS-based graph algorithms. We have already applied them to biconnected components algorithms and obtained similar speed-ups.

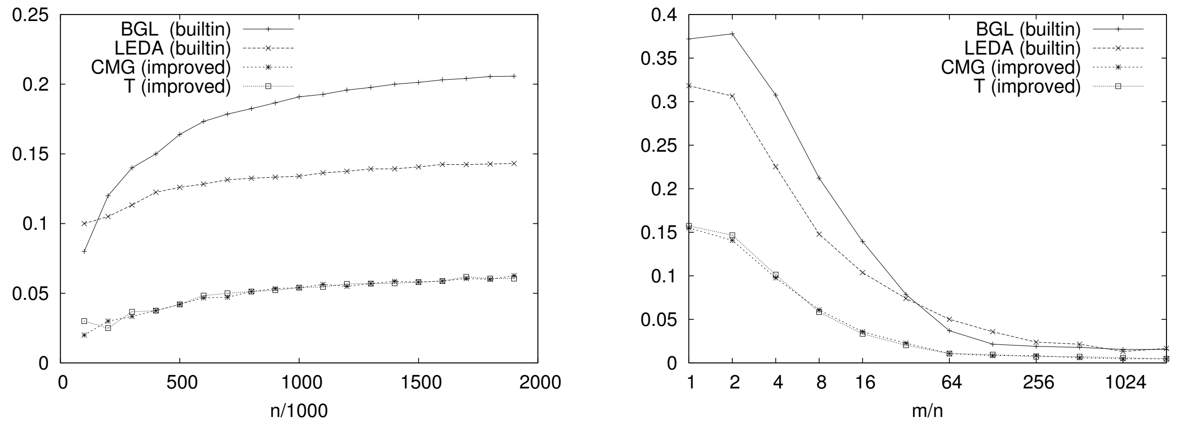


Figure 1: The running time per edge of four different SCC-implementations (in μs): the built-in implementations of BGL and LEDA and the optimized T- and CGM-programs. The figure on the left shows the times for different n and fixed edge density $m/n = 10$ and the figure on the right shows the times for fixed $m = 2^{23}$ and different values of m/n. The running times of the optimized versions of T and CGM are essentially identical.

A *strongly connected component* (*SCC*) of a directed graph (digraph) $G = (V, E)$ is a maximal subset of nodes C with the property that there is a directed path between any two nodes in C. Computing SCCs

[*]MPI Informatik, Saarbrücken, Germany
[†]Universität Trier, Germany, naeher@uni-trier.de.
[‡]Universität Karlsruhe (TH), Germany, sanders@ira.uka.de. Partially supported by DFG grant SA 933/3-1.

means to label the nodes such that any two nodes belong to the same SCC iff they have the same label. There are three different linear time algorithms for this problem, all based on depth-first search: Tarjan's algorithm [Tar72], the Cheriyan-Mehlhorn-Gabow[1] algorithm [CM96, Gab00], and the Kosaraju-Sharir algorithm [Sha81]. We refer to these algorithms as the T-, CMG-, and KS-algorithms, respectively. KS performs two passes of DFS, one on G and one on the reversed graph; the other two algorithms rely on a single DFS with somewhat more complex bookkeeping.

Modern processors are complex computing engines and the actual running time of a program is mainly determined by three factors: (1) the number of instructions executed, (2) efficient use of the processor pipeline, and (3) efficient use of the cache memory. In particular, cache faults carry a high penalty.

We will next sketch two of our speed-up techniques. Graph algorithms assemble information about the input graph during their execution. Part of this information is associated with nodes, e.g., for a node we may record whether DFS has reached it, the DFS-number, the number of the component containing it, and so on. It is natural, to reserve separate storage locations for the different pieces of information. However, this is frequently wasteful and leads to an unneccesarily large memory footprint and hence an unneccesarily large number of cache faults. The different node labels are typically relevant at different times during execution and hence can be combined into a single location. For example, node label zero may indicate that DFS has not reached a node yet, a negative node label may indicate that the node has been reached and record the (negated) DFS-number, and a positive node label may indicate that the node is completed and has been assigned to an SCC. Another example is that a particular label is only relevant while the DFS-call for the node is active. Then the information is best stored on the recursion stack.

DFS only needs access to the edges leaving a node. Therefore, a very simple graph representation suffices. All edges leaving a node a stored in a common array. The arrays for different nodes may be combined into a single array. The static graph types of BGL and LEDA use this representation. On this representation, scanning the graph may incur up to $2n + m/B$ cache faults, one for each call, one for each return, and one for each block of B edges. Here B is the size of a cache line measured in number of edges that fit into a cache line. We give an alternative implementation where the number of cache faults lies between $n + m/B$ and $n + 3m/B$. The edges emanating from a node are pushed onto a stack of edges when the node is first encountered by DFS. Then the next edge to be explored is always on the top of the stack and there is no cache fault when control returns from a recursive call and resumes the scan of an adjacency array.

Besides the technical contribution, our paper should also be useful teaching material for a course in algorithms or algorithm engineering, as it shows how careful analysis of even a simple program can lead to significant improvements in running time. The forthcoming textbook [MS07] contains implementation notes that hint towards efficient implementations. In fact, the current paper grew out of the implementation notes for the chapter on graph traversal.

The structure of this paper is as follows. In Section 2 we review the algorithms and in Section 3 we discuss the improvement techniques. Section 4 gives implementation details and Section 5 describes and discusses the experiments. Finally, Section 6 offers a short conclusion. The Appendix shows some sources.

2 The Basic Algorithms

All SCC algorithms discussed in this paper are based on DFS. They take a directed graph G as input and compute a node array $compNum$ of integers indicating the component number of each vertex. Figure 2 gives a generic version of DFS from which the different SCC algorithms can be instantiated by defining the operations $Init$, $TreeEdge$, $NonTreeEdge$, $FinishTreeEdge$, and $FinishNode$. Furthermore, all algorithms use a *DFS-numbering* of the nodes that gives the order in which nodes are visited. We will write $u \prec v$ if u has been visited before v. Sometimes they also need to know whether a node v is *finished*, i.e., whether the call $dfs(v)$ has been completed.

The algorithm of Kosaraju-Sharir: First, use $DFS(G)$ to compute the order in which nodes are finished (*finishing times*). Second, mark all nodes as unvisited, traverse them in order of decreasing finishing time,

[1]This algorithm was first described by Cheriyan-Mehlhorn [CM96] and later rediscovered by Gabow.

```
procedure $DFS(G)$
  mark all nodes unvisited;
  $Init$;
  forall $v \in V$ do
    if v is unvisited then
      $dfs(v)$;
    fi;
  od;
```

```
procedure $dfs(v)$
  mark $v$ visited;
  $TreeEdge$(v);
  forall $w \in V$ with $(v, w) \in E$ do
    if $w$ is unvisited then
      $dfs(w)$;
      $FinishTreeEdge(v, w)$
    else
      $NonTreeEdge(v, w)$;
    fi;
  od;
  $FinishNode(v)$;
```

Figure 2: Depth-first search of a directed graph $G = (V, E)$

and call $dfs(v)$ on the reverse graph G_{rev} for each still unvisited node v. Each call will visit exactly a strongly connected component of G.

```
$Init$:
  $sccCount$ := 0;
  $open$ := empty stack;

$TreeEdge(v)$:
  $lowPoint[v]$ := $v$;
  $open$.push($v$);

$FinishTreeEdge(u, v)$:
  if $lowPoint[v] \prec lowPoint[u]$ then
    $lowPoint[u]$ := $lowPoint[v]$;
  fi;
```

```
$NonTreeEdge(u, v)$:
  if $v$ open and $v \prec lowPoint[u]$ then
    $lowPoint[u]$ := $v$;
  fi;

$FinishNode(v)$:
  if $lowPoint[v]$ = $v$ then
    repeat $u$ := $open$.pop();
        $compNum[u]$ := $sccCount$;
    until $u$ = $v$;
    $sccCount$++;
  fi;
```

Figure 3: Instantiation of basic DFS operations for Tarjan's algorithm.

The algorithm of Tarjan: Consider the execution of $DFS(G)$ and use $G_c = (V_c, E_c)$ to denote the currently explored subgraph, i.e., V_c comprises the visited nodes and E_c comprises the explored edges. We call an SCC of G_c *open* if it contains an unfinished node (= a node v for which $DFS(v)$ has not finished yet) and *closed* otherwise. We call a node v *open* if v belongs to an open component and *closed* if it belongs to a closed component. For every closed node v the number of its SCC is already known and stored in $compNum[v]$.

The algorithm maintains a stack *open* of all open nodes and stores for every open node v its so-called lowpoint in a node array $lowPoint$ such that $lowPoint[v]$ is the node with the smallest dfs-number among all nodes that are reachable from v by a path of tree edges followed by one non-tree edge including node v itself. It it not difficult to see that a currently completed node v (at the end of $dfs(v)$) is a root of an SCC if and only if $v = lowPoint[v]$. Then v and all nodes with a larger dfs-number can be removed from the stack $open$. They form a new SCC. The basic operations are shown in Figure 3.

The algorithm of Cheriyan-Mehlhorn-Gabow: The algorithm is similar to Tarjan's algorithm. In every component, the node with the smallest DFS-number in the component is called the *representative* or *root* of

```
Init:
  $sccCount$ := 0;
  $roots$ := empty stack;
  $open$ := empty stack;

TreeEdge(v):
  $roots$.push($v$);
  $open$.push($v$);

NonTreeEdge($u, v$):
  if $v$ is open then
      while $v \prec roots$.top() do
          $roots$.pop();
      od;
  fi;

FinishNode($v$):
  if $v$ = $roots$.top() then
      $roots$.pop()
      repeat $u$ := $open$.pop();
              $compNum[u]$ := $sccCount$;
      until   $u = v$;
      $sccCount$++;
  fi;
```

Figure 4: Instantiation of basic DFS operations for Cheriyan-Mehlhorn-Gabow.

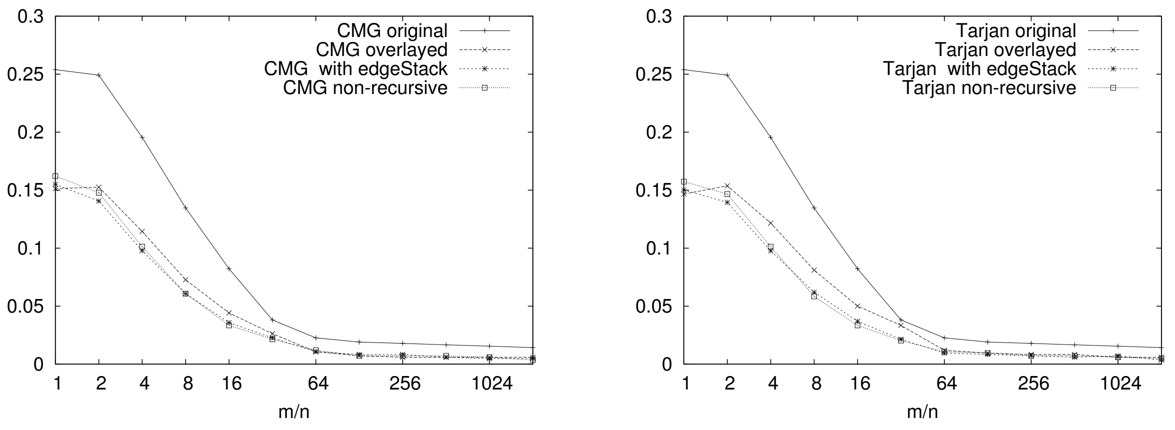


Figure 5: The performance improvements obtained for CMG and T by the optimizations described in Section 3. The time per edge is shown in μs ; m/n varies and $m = 2^{23}$. We turned on one optimization after the other.

the component. The idea of the AGM-algorithm is to maintain the strongly connected components of G_c during the execution of $DFS(G)$.

Open components are represented by two stacks *open*, and *roots* of open nodes stored in order of increasing DFS number. Stack *open* keeps all open nodes and *roots* only roots, i.e, the nodes of *roots* always form a subsequence of the nodes in *open*. With these definitions in place, the basic operations become very simple. They are shown in Figure 4.

Note that the total number of executions of the loops in these operations is $\mathcal{O}(n)$ since each iteration pops data from a stack that experiences at most n push operations. Moreover, the operations on the stacks will be very fast in practice, since stacks have good cache locality.

3 Performance Improvements

We will first describe our improvements for the CMG algorithm, mostly because this is the algorithm for which we implemented them first. Then we discuss modifications needed for the other algorithms. Our main concern will be large graphs where cache faults due to random memory accesses are the main cost factor.

Overlayed Node Data: Besides the graph itself, CMG needs to store and access four kinds of information for each node v: an indication whether v is marked, an indication whether v is open, something like a DFS-number in order to implement '$\prec$', and, for closed nodes, the representative of its component (the output). It seems that previous implementations kept this information separate leading to significant space overhead and many cache faults. The array *compNum* suffices to keep this information. For example, if *NodeIds* are integers in $1..n$, $compNum[v] = 0$ indicates an unmarked node. Negative numbers indicate negated DFS-numbers so that $u \prec v$ iff $compNum[u] > compNum[v]$. This works because '$\prec$' is never applied to closed nodes. Finally, the test $w \in open$ simply becomes $compNum[v] < 0$. Gabow's description [Gab00] of the algorithms already describes some of these optimizations.

Graph Representation: Since the graph is static and only outgoing edges need to be known, a very simple *adjancency array* representation suffices. Nodes are numbered 1 through n. A single edge array $E[1..m]$ stores the target nodes of all edges grouped by source node. A vertex array $V[1..n+1]$ stores the starting position of the edges for each node so that $E[V[v]..V[v+1]-1]$ stores the target nodes of the edges leaving node v ($V[n+1] = m$). This leaves an important choice for storing the node array $compNum$: Store it as a separate array or make he vertices a record with two fields. For sufficiently large n, the latter choice has better cache locality since the adjancency information and the $compNum$ of a node are frequently accessed together.

Make the Common Case Fast: Except for very sparse graphs, the most frequently executed operation is $NonTreeEdge$. This operation needs to know the DFS-number of $roots.top()$. Hence, we redefine $roots$ to store DFS numbers rather than NodeIDs of representatives. The only other access to the content of $roots$ (in $FinishNode$, "$v = roots.top()$?") does not become much more expensive either, since the DFS-number of v can be stored on the recursion stack (i.e., in a local variable).

Reducing Accesses to the Graph: DFS performs a sequential scan of the edges leaving a node v. However, interruptions by recursive calls can lead to additional cache faults when accessing the adjancency list of v. It therefore makes sense to copy the adjacency list to the recursion stack once and for all when it is first accessed. All remaining accesses can then be served from the stack. This is more cache efficient since stacks have very good cache locality independent of the access pattern. A remaining problem is that C++ does not support variable size arrays on the stack. Hence, we use a separate stack for storing the copied adjacency arrays. In a sense, we create a new representation of the graph with a layout tuned for the order in which the nodes are accessed by DFS.

Nonrecursive Implementation: Traditional wisdom tells us that recursive algorithms can be tuned by making them nonrecursive. This is always possible by maintaining a stack explicitly. This can be more efficient because we have more control over content and representation of the stack. However, so far our experiments show little difference. Apparently, modern compilers are good at keeping only things on the stack that are really changing between the recursive calls.

Tarjan's Algorithm: All optimizations above are also applicable to Tarjan's algorithm. An additional optimization which was already discussed by Sedgewick in [Sed92] is as follows: Low points do not have to be stored in a node array since lowpoint data has to be available only for the current node v and the neighbor node w that was just visited by a recursvive call $dfs(w)$ (in $FinishTreeEdge$). It is sufficient to keep the current lowpoint of v in a local variable and to make $dfs(w)$ return the lowpoint of w. This improvement makes Tarjan's algorithm more elegant. In fact, after the optimization, the T- and the CMG-algorithm are quite similar. The latter has an additional stack where the former needs an additional entry on the recursion stack.

Kosaraju-Sharir: Most of the above optimizations are also relevant for this algorithm. The first pass can be simplified to the extent that not even finishing times need to be stored in a node array. Instead, a $FinishNode$ operation simply pushes the NodeId on a stack constituting the output. Computing the reverse graph is in itself not so easy. It is comparable to sorting the edges by their target node. Although cache efficient integer sorting algorithms might help here, the cost is not negligible. Even if the input graph already has reverse edges available, we still end up with two passes rather than one pass. Within the passes, only the cheapest operations (stack accesses,...) can be saved compared to the one-pass algorithms. The bottom line is that Kosaraju-Sharir does not look promising for a high performance implementation. Its main asset is its simplicity.

4 Implementation

For the implementation we used four different kinds of graph data structures: LEDA dynamic and static graphs, BOOST static graphs, and a simple hand coded graph data structure.

In order to use the same code base for LEDA and BGL graphs we used an adapter class that makes the typical LEDA iteration macros and node_array syntax usable for BGL graphs as well. When working with LEDA graphs it is possible to choose whether the compNum array is stored externally as a *node_array* or whether this data should be stored directly in the node objects of the graph by using a *node_slot*. The latter gives much better performance, in particular, when used together with a static graph structure. A detailed description of static graphs and node slots can be found in [NZ02].

For the recursive versions of the code, it is necessary to work with a unlimited system stacksize. On most operating systems the program stacksize is limited by default to a very small quantity. This will most definitely cause stack overflow errors for large inputs. The stacksize can be set by using the limit command on Linux (`limit stacksize unlimited`).

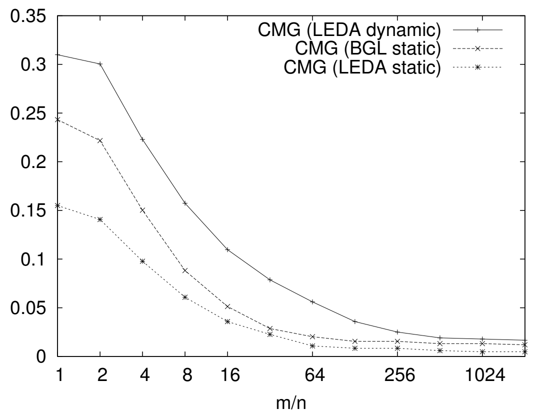
5 Experiments

For our experiments we use random graphs with varying n and m. We performed two sets of experiments, one for varying n and fixed edge density $m/n = 10$ and one for fixed number of edges $m = 2^{23}$ and varying n. For all algorithms under consideration, the number of instructions executed is almost independent of the graph. Also, random graphs are not likely to be easy cases since random edges imply many random memory accesses. In the following discussion we concentrate on large graphs with 2^{23} edges in order to see caching effects clearly. The implementation was done using gcc version 4.1 using LEDA 5.0 and the Boost graph library version 1.34.1. Optimization level was -O2 producing 32 bit code. We report running times on one core of a PC with 2 GByte of memory and a 2.4 GHz Intel processor model Core2-Duo 6600 . We obtained very similar results for AMD Opteron and SUN Sparc processors.

Figure 1 compares the performance of our best implementations for T and CMG with previous implementations. We use the static graph data structure from LEDA and all the optimizations outlined in Section 3. The source codes can be found in the appendix. We see that the tuned versions of T and CMG have essentially the same performance and are significantly faster than the built-in implementations of BGL and LEDA. We also see that the running time per edge decreases with increasing edge density m/n. This is easily understood. The running time of the algorithms depends linearly on n and linearly on m, however the dependence on n comes with a larger factor of proportionality. Note, that tree edges cause recursive calls (but there are only $n - 1$ of them) and that only $n - 1$ non-tree edges can cause the merging of components in CMG. In T we may have $\Theta(m)$ changes of lowpoint values, but these are assignments to local variables and hence almost free. Since the dependency on n comes with a larger factor of proportionality, the running time per edge decreases as a function of m/n for fixed m.

Of course, it is difficult to explain the sources of differences when the underlying algorithm, implementation details, and graph representation are changed at the same time. So let us look at one aspect after the other. Figure 5 shows the effect of our different optimizations applied to CMG and Tarjan's algorithm. In all cases, we use LEDA's static graphs to represent graphs. After showing the performance of the original algorithm, we turn on one optimization after the other: First, the overlayed node data improvement, then the edge stack to reduce accesses to the graph, and finally a non-recursive version with both improvements. The diagrams show that the first improvement reduces the running time of both algorithms by a factor of about two, the edge stack gives another improvement of about 10 percent, whereas the non-recursive implementation only yields a marginal improvement.

Figure 6 shows the effect of different graph representations on the running time of tuned CMG. We see that LEDA's static graph data structure performs best, the BGL implementation comes in second, and LEDA's dynamic graphs come in last. This is to be expected as LEDA's dynamic graphs offer more constant time update operations than BGL's graphs and LEDA's static graphs offer only a small number of update operations. We explain the difference between LEDA's static and dynamic graphs.



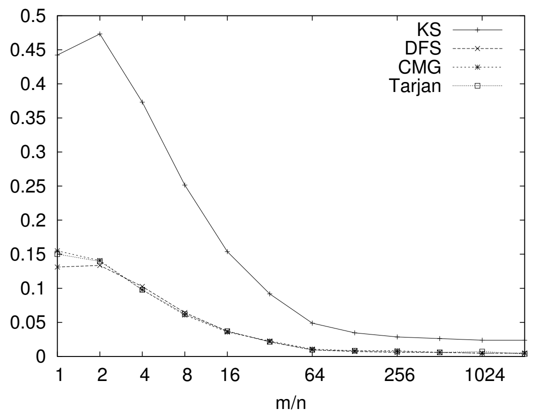


Figure 6: The figure on the left shows running times of the tuned version of CMG-algorithm with BGL static and LEDA static and dynamic graph representation. The figure on the right compares four algorithms: Kosajaru-Sharir, Cheriyan-Mehlhorn-Gabow, Tarjan, and simple DFS labelling

The static graph representation stores all edges in a single array, the dynamic representation uses linked lists for the adjacency lists. The random graph generator ensures that the adjacency list of each vertex is stored in consecutive locations. So the difference in running time is due to the fact the dynamic representation uses substantially more space.

Finally, Figure 6 shows the running time of the tuned versions of all three algorithms together with the running time of a simple DFS-scan. We see that the DFS-scan takes almost as much time as the two one-pass algorithms. A DFS-scan provides a lower bound for the execution time of any DFS-based algorithm. Thus the tuned versions of T and CGM are essentially optimal and KS must take at least twice the time of the other algorithms. In fact, it is worse since the reversed graph must be computed in addition.

6 Conclusion

We have shown that implementation details have an (for us unexpectedly) high impact on the performance of SCC algorithms. It is very likely that this extends to other algorithms based on DFS. In particular, reducing accesses to the graph will work for any DFS-based algorithm like DFS numbering, topological sorting, st-numbering, and others. Overlaying the node data is more interesting for non-trivial algorithms like biconnected components ([Tar72]), triconnected components([HT73]), or planarity testing ([HT74]). Figure 7 shows that our techniques also lead to substantial speed-ups for biconnected component algorithms.

We have not yet looked on the influence of the graph structure on performance. Sibeyn et al. [SAM02] have looked at ways to handle large graph using semiexternal algorithms where node descriptions fit into fast memory but edges only on slow external memory with blocked access. Although the methods used there are too complex to look promising for the cache-main-memory hierarchy, some ideas like reordering nodes or adjacency lists might turn to be relevant also for internal memory algorithms.

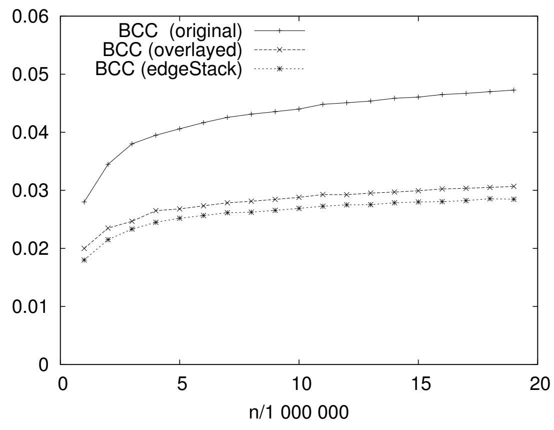
We have tried prefetching instructions in order to hide memory access latencies — so far without success.

References

[Boo] Boost.org. boost C++ Libraries. `www.boost.org`.

[CM96] J. Cheriyan and K. Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15:521–549, 1996.

[Gab00] H. N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3–4):107–114, 2000.



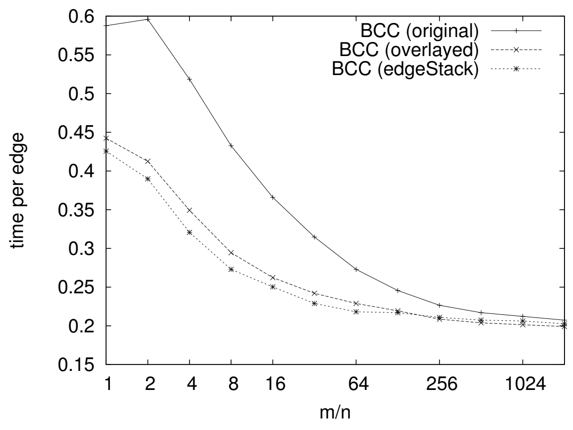


Figure 7: The running time per edge of three different biconnected components implementations (in μs): the original algorithms ([Tar72]) and the effect by applying the overlayed node data and edgeStack improvements one after the other. The figure on the left shows the times for different n and fixed edge density $m/n = 10$ and the figure on the right shows the times for fixed $m = 2^{23}$ and different values of m/n.

[HT73] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[HT74] J. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. of the ACM*, 21(4):549–568, 1974.

[LED] LEDA (Library of Efficient Data Types and Algorithms). `www.algorithmic-solutions.com`.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[MS07] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2007. in preparation.

[NZ02] Stefan Näher and Oliver Zlotowski. Design and implementation of efficient data types for static graphs. *Lecture Notes in Computer Science*, 2461:748–759, 2002.

[SAM02] J. F. Sibeyn, J. Abello, and U. C. Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proceedings of the 14th Annual ACM Symposium on Parallel ALgorithms and Architectures (SPAA-02)*, pages 282–292, New York, August 10–13 2002. ACM Press.

[Sed92] R. Sedgewick. *Algorithms in C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1992.

[Sha81] M. Sharir. A strong-connectivity algorithm and its applications in data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[Tar72] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

Appendix: Selected Sources

Tuned Cheriyan-Mehlhorn-Gabow

```
template <class graph_t>
class STRONG_COMPONENTS_CMG {

int dfsCount;
int sccCount;

template<class compNumArray>
void dfs(const graph_t& G, node v, compNumArray& compNum, b_stack<node>& edgeStack,
                                                          b_stack<node>& open,
                                                          b_stack<int>&  roots)
{ int dfsNum = --dfsCount;
  compNum[v] = dfsNum;
  roots.push(dfsNum);
  open.push(v);

  int sz = edgeStack.size();

  edge e;
  forall_rev_out_edges(e,v) edgeStack.push(G.target(e));

  while (edgeStack.size() > sz)
  { node w = edgeStack.pop();
    int  d = compNum[w];
    if (d >= 0) continue;
    if (d == -1)
      dfs(G,w,compNum,edgeStack,open,roots);
    else
      while (roots.top() < d) roots.pop();
   }

  if (roots.top() == dfsNum)
  { node u;
    do { u = open.pop();
         compNum[u] = sccCount;
        } while (v != u);
    roots.pop();
    sccCount++;
   }
}

public:

template<class compNumArray>
int run(const graph_t& G, compNumArray& compNum)
{ int n = G.number_of_nodes();
  int m = G.number_of_edges();
  b_stack<node> edgeStack(m);
  b_stack<int>  roots(n);
  b_stack<node> open(n);
  dfsCount = -1; sccCount = 0;

  node v;
  forall_nodes(v,G) compNum[v] = -1;

  forall_nodes(v,G)
   if (compNum[v] == -1) dfs(G,v,compNum,edgeStack,open,roots);

  return sccCount;
}
};
```

Tuned Tarjan

```
template <class graph_t>
class  STRONG_COMPONENTS_TARJAN {

int dfsCount;
int sccCount;

template <class compNumArray>
int dfs(const graph_t& G, node v, compNumArray& compnum, b_stack<node>& edgeStack,
                                                         b_stack<node>& open)
{ int dfsNum = dfsCount++;
  int lowPoint = dfsNum;
  compNum[v] = dfsNum;
  open.push(v);

  int sz = edgeStack.size();

  edge e;
  forall_rev_out_edges(e,v) edgeStack.push(G.target(e));

  while (edgeStack.size() > sz)
  { node w = edgeStack.pop();
    int d = compNum[w];
    if (d == -1) d = dfs(G,w,compNum,edgeStack,open);
    if (d < lowPoint) lowPoint = d;
   }

  if (dfsNum == lowPoint)
  { node u;
    do { u = open.pop();
         compNum[u] = sccCount;
       } while (u != v);
    sccCount++;
   }

  return lowPoint;
}

public:

template <class compNumArray>
int run(const graph_t& G, compNumArray& compNum)
{
  int n = G.number_of_nodes();
  int m = G.number_of_edges();

  b_stack<node> edgeStack(m);
  b_stack<node> open(n);

  dfsCount = -(n+1); sccCount = 0;

  node v;
  forall_nodes(v,G) compNum[v] = -1;

  forall_nodes(v,G)
     if (compNum[v] == -1) dfs(G,v,compNum,edgeStack,open);

  return sccCount;
}

};
```