Heuristics for Semi-External Depth First Search on Directed Graphs*

Jop F. Sibeyn† James Abello‡ Ulrich Meyer§

Abstract

Computing the strong components of a directed graph is an essential operation for a basic structural analysis of it. This problem can be solved by twice running a depth-first search (DFS). In an external setting, in which no longer all data can be stored in the main memory, the DFS problem is unsolved so far. Assuming that node-related data can be stored internally, semi-external computing, does not make the problem substantially easier. Considering the definite need to analyze very large graphs (web graph, telephone call graphs, ...), we have developed a set of heuristics which together allow to perform semi-external DFS for directed graphs in practice. The heuristics have been applied to graphs with very different graph properties, including "web graphs" as described in the most recent literature (which are not exceptionally hard) and some call graphs from ATT. The program is typically about 100 times faster than the best alternative, a factor that will further increase with future technology developments.

Keywords: Depth First Search, External Memory, Graph Algorithms, Strong Components.

1 Introduction

Depth-first search, *DFS*, is a basic and crucial operation on graphs. On undirected graphs it can be used for computing the biconnected components. On directed graphs DFS is the key routine to computing strongly connected components. DFS can also be used for determining whether a graph is acyclical, and, if yes, to compute a topological sorting. For graphs with n nodes and m edges, sequential DFS can be performed in $\mathcal{O}(n + m)$ time. This algorithm accesses the n adjacency lists of the nodes in an a priori unpredictable order which implies that it exploits random memory access in an essential way. In an *external* setting, where the data do not fit in the main memory, it performs extremely bad. Surprisingly, there are no substantially better external-memory algorithms which means that in practice the above mentioned problems cannot be solved for graphs such as the web graph or the call graphs (in which the nodes represent customers and every directed edge represents a telephone call) of telephone companies.

In this paper we assume that the internal memory can hold $c \cdot n$ data, for some small constant c, but that m is too large. This is called *semi-external* computing. Even in this setting, theory does not provide any practical solution for the DFS problem. Though nothing is proven, it appears that the problem has an intrinsic hardness similar to NP-hardness in the sequential and P-completeness in the parallel domain. It is a common and accepted practice to try to tackle such problems either by approximation algorithms or by heuristics. For DFS an approximation does not appear to make much sense: apparently, an almost correct DFS forest, does not need to lead to almost correct strong components. But, heuristics may be useful as we will show.

1.1 Previous Work

All previous external DFS algorithms follow the basic idea of the sequential approach: a stack is maintained; repeatedly the top element is popped; if it was not visited before, then all the nodes in its adjacency list are pushed on the stack. In an external setting it is not obvious how to maintain the information about the status (visited or not) of the nodes so that one does not need m accesses to the secondary memory. This problem has been solved by Chiang et al. [8] and Kumar and Schwabe [14]. However, their algorithms still uses more than n IOs. There exists a polylog-time parallel algorithm for unordered[1] DFS [1] but it is not suited for conversion into an external memory algorithm along the lines of [8] because it is not work-optimal.

Even though this is much better than m IOs, it is still far too much to be practical (on current systems one hard disk access takes about 10 ms). In a semi-external setting, maintaining the status information is no issue: the array of n bits can be maintained internally. However, fundamentally this does not change

*Partially supported by the Future and Emerging Technologies programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and the DFG grant SA 933/1-1.

†Department of Computing Science, Umeå University, 90187 Umeå, Sweden. URL: http://www.cs.umu.se/~jopsi

‡ATT Labs Research, Florham Park, NJ 07932, USA. URL: http://www.research.att.com/info/abello

§Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany. URL: http://www.uli-meyer.de

[1]Unordered DFS means that the edges of a node may be processed in any order, whereas ordered DFS computes one particular tree according to the order of the edges within the adjacency lists.

anything: accessing the n adjacency lists in the unpredictable order in which the nodes are popped from the stack still requires close to n IOs. This is only acceptable for graphs with on average thousands of edges per node: in that case, for each access to the secondary memory, the program can on average load a sufficiently large number of edges per hard disk access.

Only for special cases there are better results. For outerplanar graphs and graphs with bounded tree width, Maheswari and Zeh [15, 16] have given much faster algorithms, which have an IO complexity of the same order as sorting. In [17] it is shown how external graph algorithms can take advantage of a redundant graph representation and super-linear space. This yields the first DFS algorithm performing $o(n)$ IOs on undirected planar graphs with arbitrary node degrees. Arge et. al. have further improved this result, showing that for planar graphs external DFS can be solved with $\mathcal{O}(\mathit{sort}(n \cdot \log n))$ IOs.

1.2 New Contributions

In this paper we present a heuristic for the semi-external DFS problem, which has been worked out into a program. It has the following strong features: minimal usage of internal memory (4 integers plus 4 booleans per node, of which at most 3 integers plus 3 booleans are accessed in an unstructured way at the same time), acceptable running speed: in practice the program takes between 5 and 100 times longer than the sequential algorithm if the data would fit in the main memory, being at least 50 faster than algorithms that perform n page misses. Together this implies that it can indeed be applied for very large graphs, as is shown by presenting results for several classes of generated graphs with very different graph properties. If the paper gets accepted we will also present our experience with several call graphs of ATT with up to 200 million nodes and 2 billion edges. These experiments are almost completed, results of measurements on subgraphs are reported in the paper. As far as we know this is the first serious attempt to tackle semi-external DFS in practice.

A completely different approach was chosen by Broder et all. [6] in their study of the strong components of the webgraph. They do not apply any (semi-) external techniques, but use a computer with a huge internal memory in combination with data compression. In comparison with our methods, their approach has the advantage that it is much faster. But, even the most clever data compression cannot make a graph arbitrarily small, and not everyone can afford a computer with multigigabyte internal memory. Therefore we believe that semi-external methods are important as one of several possible approaches to tackling problems that are on the edge of being intractable.

The heuristics have become quite sophisticated, but can certainly be improved further. We hope that this is the beginning of a fruitful development similar to what we have seen for exact algorithms for the minimum Steiner tree problem, for which over the years ever better heuristics were able to solve the problem for ever larger graphs [11, 13, 18].

Ideas of the Algorithm. The basic idea of the algorithm is too maintain a tentative forest, which is gradually developing into a DFS forest. This idea is simple and natural, does not easily lead to a good algorithm though. Algorithms of this kind may continue to consider all edges without making (much) progress. In our algorithm these problems are overcome to a large extend by:

- initially constructing a forest with a close to minimal number of trees (which is not easy for directed graphs);
- only replacing an edge in the tentative forest if necessary;
- rearranging the branches of the tentative forest, so that it faster grows deep (as a consequence, from among the many correct DFS forests, our algorithm finds a relatively deep one);
- after considering all edges once, determining as many nodes as possible that have reached their final position in the forest and reducing the set of graph and tree edges accordingly.

A single simple reduction strategy gives effective reduction for graphs whose DFS forest mainly consists of one very long path. We spent considerable effort on developing other, less simple strategies to even obtain effective problem reduction for graphs with a less fortunate structure.

Another issue is the order in which the edges are processed. One possibility, we tried this as well, minimizing the number of edges to consider, is to consider the edges (in pseudo-random order) one-by-one, however this requires elaborate datastructures (such as dynamic trees), which require too much internal memory, and are too inefficient ($\mathcal{O}(\log n)$ per operation), to be practical. Therefore, our algorithm is repeatedly reading batches of n eges, which are processed together with the edges from the tentative forest in an internal DFS operation in $\mathcal{O}(n)$ time.

Achievements. The algorithm has been turned into a C program, which in addition to the computation does testing, checking and analysis. It also has build in routines to generate directed and undirected graphs of 9 different classes (random graphs, one- and two-dimensional geometric graphs, web graphs, star graphs and acyclic graphs), to randomize the edges and to filter out double edges and self loops. All large data structures are maintained as files. All IO is done in a buffered way, with buffers that are so large (1 MB) that seek time is negligible. In this way, the time for IO has been reduced to a non-dominating fraction (about 25%) of the overall running time. No virtual memory is used if the internal memory has space for $16.5 \cdot n$ bytes. If the internal memory has space for at least $12.375 \cdot n$ bytes, the program still runs without major delay, using up to $4.125 \cdot n$ bytes of virtual memory.

The time in seconds is the most important in practice, but we do not use this as our main performance measure because it depends too much on the system. Instead, in the design of our algorithm we focus on minimizing the ratio r of the total number of edges that is scanned in and processed in the internal DFS operations divided by m. r depends on the algorithm and on the graph but not on the system. It even provides a good measure for the time the program will take: for appropriate constants c_1, c_2 and c_3, the total time consumption can be described rather accurately by an expression of the form

$$T_{\text{total}} = c_1 \cdot n + c_2 \cdot m + c_3 \cdot r \cdot m.$$

We are aiming at solving DFS problems efficiently for all graphs, but we particularly also wanted to achieve excellent performance for random graphs. Therefore, the algorithm starts by performing an initial round in which $m_0 = \min\{m, \mathcal{O}((m/n)^{1/3} \cdot n)\}$ edges are processed (the hidden constant is so small, that this is practical). As a result we find a very long path, and, by rearranging the tree, we can guarantee that the nodes on this path are final, which allows to filter out all their ingoing edges and all outgoing edges to other nodes on the path. The remaining problem has size $\mathcal{O}((m/n)^{1/3})$. Thus, the whole problem can be solved by processing $\mathcal{O}(m_0)$ edges and scanning through all edges once. For $m = \omega(n)$, this implies that DFS can be performed in just slightly more than one scan. More generally: the DFS problem is easy for graphs for which the DFS forest mainly consists of a single very long path with an occasional branch of size one or two. It is much harder for graphs whose DFS forest consist of large and very shallow trees. Such forests arise for acyclic graphs in which each node choses its edges uniformly from among all nodes with a smaller index. Even for such graphs our reductions are very effective, but we may nevertheless find values of r around 10 for very large problem instances.

In order to become concrete, we give two results measured on a Pentium III with an 1 GHz processor, 512 MB of main memory and a 7200 rpm hard disk. Solving the DFS problem for a random graph with $n = 2 \cdot 10^7$ and $m = 10 \cdot n$ gives $r = 0.686$ and takes $T = 1093$ seconds. For denser graphs, the time per edge strongly decreases. For an acyclic graph, constructed as mentioned, with the same n and m, we find $r = 7.995$ and $T = 6367$. This should be compared with the about 200,000 seconds all alternatives take (processing about 100 nodes per second). It is interesting to notice that when determining the strong components, the second DFS computation shows much less "worst-case behavior": for the easy graphs it takes slightly more time, while for the hardest instances it takes substantially less.

1.3 Computer Model

For our algorithm the details of the computer model are not important. The only relevant system feature is the distinction between random and batched memory access. The cost $t(b)$ for reading b consecutive integers from a specific location in the secondary memory can namely quite accurately be approximated by

$$t(b) = t_{\text{seek}} + b \cdot t_{\text{transfer}}. \tag{1}$$

The same expression, with possibly a somewhat larger value for t_{transfer}, can be used to estimate the time for *writing* b integers. In current practice, for obtaining coarse estimates, it is convenient to work with $t_{\text{seek}} = 10^{-2}$ and $t_{\text{transfer}} = 10^{-6}$.

(1) implies that, in comparison to the speed of the processor, it is extremely inefficient to randomly access secondary memory, processing only $1/t_{\text{seek}} \simeq 100$ objects per second. However, if objects are accessed in batches of a size B that is so large that $B \cdot t_{\text{transfer}} \gg t_{\text{seek}}$, we can process up to $1/t_{\text{transfer}} \simeq 1,000,000$ objects per second (if the objects are small), which is not so much worse than the speed at which the main memory can supply the processor with data that do not reside in the cache.

1.4 Graph Representation

The nodes are indexed from 0 up to $n-1$. It is assumed that the graph initially resides on file as m pairs of node indices giving the edges. Finally the output, is written away to file as a set of n node pairs, defining the DFS forest. For each tree root u an edge (n, u) is written, n being the smallest number that is not a node index.

Internally, the forest is maintained as adjacency lists which are implemented in arrays, requiring only $n + m'$ integers for a graph with n nodes and m' edges. An array implementation lacks the flexibility of linked lists, but, in our case, saving internal memory is more important than ease of use.

The first node in the adjacency list of a node u is called the leftmost child of u. More generally, we will sometimes refer to the children as if the tree were drawn so that the children appear from left to right in the order in which they appear in the adjacency list of their parent.

1.5 Basic Algorithms

For comparison purposes we present a non-recursive version of the basic internal DFS algorithm.

```
Procedure Conventional_DFS(int n) {
  for (u = 0; u < n; u++)
    free[u] = 1;
  for (j = u = 0; u < n; u++) {
    tail = head = 0;
    stck[head] = u;
    while (tail ≤ head) {
      v = stck[tail];
      tail++;
      if (free[v]) {
        free[v] = 0;
        j++;
        prfx[v] = j;
        for (each neighbor w of v) {
          head++;
          stck[head] = w; } } } } }
```

If this algorithm is run in an external setting, then it must access the hard disk for getting free$[v]$ for each node v popped from the stack, giving a total of close to m accesses. If this algorithm is run in a semi-external setting, then it must access the hard disk for getting the adjacency list for each free node v, giving a total of close to n accesses. Even in this latter case (and no algorithm from the literature performs better), the time is horrible: just for the hard disk accesses, it takes $n \cdot t_{\text{seek}} \simeq n/100$ seconds. Only for very dense graphs this algorithm becomes competitive.

If we consider the development of the time for a floating point operation t_{int} and that of the hard disk access parameters, t_{transfer} and t_{seek}, then we see that since many years, t_{int} has been decreasing most rapidly and t_{seek} most slowly. t_{seek} depends on the rotational speed of the hard disk, which is not easy to increase. t_{int} depends on the clock speed, which has been doubling every 18 months for the past decades. If this development continues, then the relative advantage of algorithms like ours, which perform many internal operations and very few accesses to the external memory, increases. Right now our algorithm is about 100 times faster than the conventional algorithm. In the coming three years this factor will probably double, just as it doubled since three years ago.

2 Algorithm

2.1 Basic Approaches

Edge-by-Edge Approach. There are two possible basic approaches towards algorithms of the type we are considering. The first appears to be most promising:

```
Procedure Edge_By_Edge_Processing(int n, int m) {
  while any change during last m edges do {
    let (u, v) be the next edge in cyclical order;
    if (u, v) is a cross edge {
      let w be the parent of v;
      cut the link (w, v) and add the link (u, v); } } }
```

```
Procedure Semi_External_DFS(int n, int m) {
  initialize an empty tree datastructure;
  for (u = 0; u < n; u++)
    add a link (n, u) to the tree;
  Edge_By_Edge(n, m); }
```

Here we assumed that the n nodes are indexed from 0 up to $n-1$, to which we have added the artificial node n. Initially all nodes are connected to node n, which makes all further operations slightly easier. These links from n will gradually disappear, and finally the children of n are the roots of the trees of the DFS forest. Henceforth we will speak of tree instead of forest, meaning the tree rooted at node n. The correctness of this algorithm is obvious, because a tree is a DFS tree if and only if there are no cross edges. What is unclear is how many edges have to be tested. In the basic form the algorithm is given, this number is far too high, but with the rearrangement idea presented in the next section it becomes acceptable (which means that in practice one has to traverse all edges only a few times).

In an earlier stage we developed an algorithm along these lines for undirected graphs. In that case one should not cut the link (w, v), but a link just below the lowest-common ancestor, *LCA*, of u and v. In that case it is crucial to cut in the branch leading to the undeeper of the nodes u and v.

So, basically this approach is ok, but it is not practical. The problem is that for every individual edge we must test whether it is a cross edge (and for the undirected graphs we must even be able to determine LCAs). In a static structure, this can be done in constant time, after computing pre- and post-order numbers of the tree. However, we are performing updates to our structure all the time. Therefore we must maintain some more sophisticated datastructures, such as dynamic trees [19, 20]. These offer all we want at $\mathcal{O}(\log n)$ time per operation (including the relinkings). In itself this is already more than desirable, but, practically speaking, it is even worse that these datastructures require a substantial amount of internal memory. In our implementation we need 21 integers per node, and it appears that one cannot reduce this by much. Considering our applications (at all times we have the web graph in mind for which $m/n \simeq 12$), assuming that $21 \cdot n$ integers fit into the main memory, often amounts to assuming that $n + m$ integers fit into the main memory, and in that case one can run a slightly modified variant of the conventional DFS algorithm.[2]

[2]When running the conventional DFS algorithm, it is essential that the whole graph, which can be represented with $n + m$ integers, plus one bit per node can be maintained in the main memory. The stack, however, which has

Batched Approach. As an alternative for edge-by-edge processing, the edges can be processed in a batched way as follows:

Procedure Batched_Processing(int n, int m) $\{$
while any change during last m edges **do** $\{$
load the next batch of n edges into the main memory;
add these edges to the adjacency lists of the tree;
search a new DFS tree in the current set of $2 \cdot n$ edges;
update the adjacency structure; $\}$ $\}$

One clear advantage of this approach is that the amortized time per processed edge is constant. Another advantage is that we need no complicated data structures and that all internal data can be fit in $5 \cdot n$ integers of which at most $3 \cdot n$ are accessed in an unstructured way at the same time (this requires very careful memory management).

However, it is not obvious that with this approach the number of processed edges does not become too large. In addition to the rearrangement presented hereafter, the most important idea is to give preference to the existing tree edges when searching for a new DFS tree. This is easily achieved by inserting the newly loaded edges at the end of the adjacency lists of the nodes (assuming that when pushing the children of a node on the stack, this is done in reversed order, so that the first child comes at the top of the stack).

Clearly the algorithm becomes better when loading larger batches of edges: this amortizes the fixed cost of handling the n tree edges over more newly loaded edges; and with larger batch size it becomes more likely that the algorithm rapidly finds deep paths. However, for a graph with n nodes and m' edges, the internal DFS subroutine requires random access to $n + m'$ integers. In our case $m' = n + b$ b is the number of edges in a batch. Taking $b = n$ appears to be a reasonable compromise (particularly also because we need $3 \cdot n$ integers of randomly accessible memory even at other places in the program).

The operation of loading n edges and processing them in an internal DFS operation is called a *phase*. Processing all m edges once will be called a *round*. Actually the algorithm is composed of discrete rounds, each consisting of $\lceil m/n \rceil$ phases, at the end of which the algorithm performs the reduction operations described in Section 2.3 and tests for termination, before starting the next round.

2.2 Rearrangement

The presented basic algorithms will ultimately terminate (because some progress is made every round), but this may take very long. The problem is that the algorithm does not work particularly goal oriented. The idea is to restructure the tree in order to help the algorithm to more rapidly find a DFS tree. More concretely, for every node we sort its adjacency list on some weight attributed to its children. In our case every node has as weight the size of its subtree. Sorting the adjacency lists in decreasing order with respect to this weight function moves nodes at the root of large subtrees to the left.

Of course, a rearrangement of this kind introduces new cross edges, but this is not serious: finally these cross edges might have shown up anyway, and it is better to encounter them as early as possible.

This rearrangement is rather stable in itself. If we have two children v_1 and v_2 of a node u, v_1 coming left of v_2, which are roots to subtrees of sizes s_1 and s_2, respectively, then even if $s_1 = s_2$, the subtree of v_1 tends to grow faster than the subtree of v_2, because we expect that the tree of v_1 has more outgoing cross edges than the tree of v_2 (the difference being the cross edges running between these two subtrees themselves). So, the rearrangement really anticipates the natural development of the tree, and it is unlikely that a rearrangement is undone later on. The rearrangement is therefore not a major source of continuing updates itself.

We only have to be careful not to destroy the structure in parts of the tree of which we already had concluded that it would not change anymore. In Section 2.3 we define the notion of "passive" nodes, and edges leading to them are removed from the set of processed edges. In order to assure in an easy way that passive nodes do not become active again, we only rearrange the children of active nodes.

2.3 Reduction

Effective strategies for either reducing the set of active nodes or edges is often the key to efficient graph algorithms. In our case it is hard to find strategies that are guaranteed to work, but there are several strategies that together work quite well for most graphs.

An example of a trivial reduction strategy, is based on the observation that the nodes on the leftmost path are not going to change place anymore (provided we do not rearrange them): they are no longer *active* but they are *passive*. All edges leading towards passive nodes can be deleted: a passive node does not change its place in the tree anymore, so any ingoing edge must be irrelevant. Outgoing edges are still relevant if they are forward cross edges. For the other three categories of edges (forward tree, backward tree and backward cross) it is easy to check that they will never will become forward cross edges anymore.

For the passive nodes we will maintain as an invariant that a node u can only be declared passive if all the nodes on the

a size of up to $2 \cdot m$ integers, can very well be maintained on the hard disk, because the algorithm accesses it in a structured fashion.

path from *root*$[u]$ to u are passive. This fact will be exploited in the proof that the reduction strategies in the following are correct. Recall that all tree roots are in fact children of the artificial node n. Node n is passive right from the beginning. It is essential that the rearrangement does not rearrange the children of passive nodes.

The reduction has several advantages: the reduction of the number of active nodes reduces the size of the tree that must be maintained internally and thereby leafs more space for loading new edges, always filling up until we have $2 \cdot n$ edges. This makes the search more effective because a larger fraction of the edges is processed at the same time. It also gives rice to a better ratio between overhead (= processing the old nodes) and useful work. Reducing the number of edges is useful because this reduces the amount of work per round, and because more useful information is present in the main memory at the same time.

Unchanged Initial Part. Many DFS trees quite soon have a quite large stable first part. The involved nodes may be declared passive. This idea is implemented by comparing the current DFS tree (given as a list of edges in DFS order) with the DFS tree at the end of the previous round.

Lemma 1 *It is correct to declare all nodes occurring in the unchanged initial edges of the DFS tree as passive.*

Proof: Denote the nodes by $u_0, \ldots, u_k$, for some $k \geq 0$. These nodes have and had the lowest preorder numbers of all nodes in the tree. There cannot be any forward cross edges from the nodes $u_0, \ldots, u_{k-1}$. For a contradiction, assume that there is a cross edge (u_i, v). This would already have been a cross edge during the past round, and it would have caused a change of the tree after the edge leading to u_i. Cross edges outgoing from u_k are irrelevant for anything before u_k. □

Unchanged Final Part. It is tempting to perform a similar reduction for an unchanged last part of the tree. However, this is not always correct: if one node on the path to a deeper lying unchanged last part has been moved into another tree, then all edges that formerly were forward tree edges have become cross edges. However, if a complete tree at the end of the DFS forest is unchanged, then it will never change again and its nodes can be made passive: apparently there are no cross edges leading to this tree, otherwise at least some nodes would have moved out. A generalization of this, is

Lemma 2 *It is correct to declare a node u occurring in the unchanged final edges of the DFS tree as passive if the parent of u is passive.*

Proof: Because node n is passive, all nodes belonging to entire trees appearing in the unchanged final part will become passive. We already concluded that this is correct. The only interesting case is a node u that lies in a tree that is only partially unchanged. Inductively we may assume that it was correct to declare the nodes on the path to u passive. So, these will not move to a position were any node from them to u would become a cross edge. If there would be any other node with smaller preorder number with an edge to u, then it would already have caused a change during the past round. Edges to u from the nodes with higher preorder numbers are not dangerous, because currently they are backward cross edges, and these nodes will not change their positions anymore. □

No Passing Edges. The above strategies have problems if the DFS forest consists of many trees. In the middle many nodes may have reached their final positions and we are not detecting this. Scanning through all edges once, we can discover much more than by only looking at the changes in the DFS trees. For example, assume that u is the root of a tree, and that there is no edge from a tree to the left of it to any node within the tree of u (including u itself) or to a tree to the right of u. In that case is u passive: no edge that currently has preorder number larger than u will ever get a preorder number that is smaller than u, and thus will there never be a cross edge leading to u. This idea is generalized in the following algorithm for determining passive nodes:

Algorithm FIND_PASSIVES

1. For every node u compute the preorder number *preorder*$[u]$.

2. Traverse all edges and determine for every node u the maximum *reached*$[u]$ over all preorder numbers of the nodes to which there is an edge from u.

3. For every node u compute the preorder maximum *max_reached*$[u] = \max_{v \in S_u}\{$*reached*$[v]\}$, where S_u denotes the set of nodes v with *preorder*$[v] \leq$ *preorder*$[u]$.

4. For every node u set *is_ok*$[u]$ = *max_reached*$[v] <$ *preorder*$[u]$, where v is the node with *preorder*$[v]$ = *preorder*$[u] - 1$.

5. Process the nodes in BFS order (for example), and declare a node u passive if its parent is passive and *is_ok*$[u]$.

Notice that it is not correct to make every node u with *is_ok*$[u]$ passive: there might be a link to a node v on the path to u, in that case u moves along with v.

Lemma 3 *Find_Passives is correct.*

Proof: The argument is similar to that in the proof of Lemma 2. Inductively we assume that it was correct to make the nodes on the path to u passive. *is_ok*$[u]$ means that there is no edge from any node with smaller preorder number to any node with preorder number equal or large than that of u. So, none of the nodes after u will ever come to stand before it. That nodes that come before u move is irrelevant for the position of u as long as this does not concern the nodes on the path to u. $\square$

Aggresive vs. Gentle Reduction. The performed reductions never exclude nodes and edges that may still be required for finding a correct DFS tree. This is what we call *gentle* reduction. It is imaginable that it might be more efficient to perform more *aggresive* reductions, in which all nodes and edges are excluded that probably will not be needed anymore. Of course, then we must test at the end wheter there are still cross edges left and possibly start a new phase of reductions until no cross edges are left. Thus, the correctness is not an issue. We tried this idea but we did not find any strategy that performs better than with the three reductions described above, the reason being that for just a few cross edges all edges must be processed again.

2.4 Minimizing Initial Number of Trees

Worst-Case Example. Consider a graph with n nodes positioned on a cycle, with $2 \cdot n$ edges connecting each node to both its direct neighbors. If for this graph we start with an initial forest with two trees, the left-most with node 0 as root and node 1 as only child, and the other tree with node $n-1$ as root and all other nodes with decreasing indices as a chain under it. If we run our algorithm on this, then in every round only a few nodes (the expected number is two) will be transferred from the right tree to the left tree, and the expected number of rounds is $\Omega(n)$.

Exactly this example is no problem because we start by processing at least $2 \cdot n$ edges in a single DFS operation (see Section 2.5), but for similar graphs with more edges, for example the graphs of the built-in class CYCLE (see Section 3.1), that basically connect the nodes to a bidirectional cycle we found that the algorithm really might require $\mathcal{O}(n)$ rounds.

The problem we encounter here, is that several trees that finally will belong to the same tree initially are arranged in reversed order. This problem appears to be limited to graphs which have large subgraphs that resemble bidirectional cycles, and indeed, as far as we know, these are the only classes of graphs which are not treated well with the algorithm described.

Partially Overcoming the Problem. For the bad example above, the problem lies in the existence of several wrongly oriented trees. If we succeed in linking these trees together before we start the algorithm, then there is no problem. For undirected graphs, this would amount to first finding a spanning tree. The analogue for directed graphs is to try to find a forest with as few trees (and thus as many edges) as possible. We do not how to solve this problem efficiently, therefore we just try to fuse as many trees together as possible (a greedy approach).

Let *root*(u) denote the root of the tree to which node u belongs. A first correct idea is that when there is an edge (u, v) with *root*$(u) \neq$ *root*(v) and $v =$ *root*(v), that then v may be linked below u, reducing the number of trees by one.

A slightly further reaching idea is that when there is an edge (u, v) with *root*$(u) \neq$ *root*(v) and v being a node from which there is a path to *root*(v) entirely running through nodes of the tree to which v belongs, that then v may be linked to u. The edges on the path to *root*(v) should be added and all edges on the path from *root*(v) should be removed.

The problem with the latter idea is that it is not easy to determine whether there is such a path to the root semi-externally. Therefore, we perform a heuristic that tries to find most of the nodes from where the tree root may be reached:

Algorithm DETERMINE_TARGETS

1. For all nodes u, compute root$[u]$, giving the root of the tree to which u belongs.

2. For all nodes u, compute dpth$[u]$, giving the depth of node u.

3. Traverse all edges and determine for every node u the node rchd$[u]$ in the same tree which has minimum depth.

4. Construct a new graph G' with one ingoing edge $(\text{rchd}[u], u)$ for every node u. G' constitutes a forest with downward edges. The roots of the original tree are among the roots of G'.

5. For all roots of the original tree, perform a search in G', marking all reached nodes as targets. For every target, the node from which it is reached is saved, this is a link on the path to the root in the real graph.

The thus discovered targets are called primary targets. At the cost of one pass through all edges, one may try to find secondary targets: any node who was an edge to a target in the same tree is a target itself. In the current implementation of the program, Determine_Targets is complemented with one pass to find secondary targets.

The complete heuristic for minimizing the number of trees now repeats the following loop at least once, and as long as this gives a reduction:

Algorithm MINIMIZE_TREE_NUMBER

1. Determine_Targets.

2. Traverse all edges, and when finding an edge (u, v) leading to a target v with *root*$(u) \neq$ *root*(v), link v to u and update the edges in the tree of *root*(v). Set the boolean marking that any node in this tree is a target to false.

Because this operation is quite expensive, it should be performed only when necessary. Therefore we first test whether there are at least two trees of considerable size.

2.5 Optimizations

Many smaller tricks together helped to gain about a factor two in the running time. We mention the most important of these.

Before we have allocated any datastructures, we have some additional space in memory, which we can use for searching through a slightly larger set of edges first. In the current program we assume that there is internal memory for at least $4 \cdot n$ integers, and this allows us to search through $3 \cdot n$ edges with the implemented memory saving variant of the internal-memory DFS algorithm. How much one saves with this combined processing depends on the graph for which it is performed. For random graphs we estimate that processing $x \cdot n$ edges at once is about equally effective as performing $3/2 \cdot x$ phases. For relatively sparse random graphs, this is a noticeable improvement.

The rearrange operations take about 25% of the total time. It appears that one might save on this. For some graphs rearrangement is hardly needed after an initial phase, but for others it must be performed until the end. It turns out that r does not increase when nodes are rearranged after every second phase. Among other things, this guarantees that it is performed at least once per round. The rearrangement involves sorting (stable and in-situ). For longer adjacency lists we apply quick sort, which is optimized for the fact that the frequency with which a given value is occurring decreases with the value.

Another expensive procedure is the reduction. Particularly for relatively sparse graphs this operation may weigh heavy. It consists of two parts: figuring out which nodes can be made passive and the actual edge reduction. The second part is only performed if the number of active nodes has become sufficiently smaller. Determining the nodes that can be made passive involves the operation to determine that roots are final requiring a scan through all edges. This operation is only performed if it was sufficiently effective up till now. Furthermore, the whole reduction procedure is only performed if based on the recent development of the number of active edges, it is deemed likely to lead to a reduction. Together these operations give a substantial reduction for problems with slowly decreasing problem sizes.

3 Performance on Generated Graphs

We cannot give much of a performance guarantee (in principle the algorithm may require up to n rounds). The main argument in favor of our algorithm is that it really works, and thereby can solve problems that no one could solve before. In this section we present the results of extensive experiments for classes of generated graphs with very different structures. In the following section we present a real-life example.

3.1 Graph Classes

The program comes with an built-in input generator for a number of graph classes, which are described in the following. Here we already give some qualitative remarks on the performance of our algorithm for them. In Section 3.2 we work this out in some more detail together with the discussion of the figures. Recall that r denotes the ratio between of total number of edges that are scanned and processed in the internal DFS operations divided by m, the number of edges in the input graph. $g = m/n$ gives the average number of edges per node.

RAND denotes random graphs [5] from the class $\mathcal{G}_{n,m}$ where the $2 \cdot m$ endpoints of the edges are chosen uniformly and independently from among the n nodes. Random graphs are the easiest we know: already for small values of g, we find $r < 1$. The class GEOM-2D is a special variant of a random geometric graphs [10]: the nodes of these graphs are positioned on a two-dimensional torus and the probability that a node gets an edge to another node at Manhattan distance d decreases geometrically with d: for all $d > 0$, it equals $c \cdot \alpha^d$, where c is a normalization constant. It turns out that the graphs from GEOM-2D are among the easiest, almost as easy as random graphs.

The graphs of the class GEOM-1D are constructed analogously to those in GEOM-2D, but here the nodes are positioned on a ring. Again the probability that an edge runs over distance d equals $c \cdot \alpha^d$. A deterministic variant of this is given by the graphs from the class CYCLE, where the nodes are positioned on a ring in which each node has links to all its g nearest neighbors. All indices are randomized again. The graphs from CYCLE are so regular that the algorithm Minimize_Tree_Number described in Section 2.4 is able to figure out the global structure. Thereafter it is just a matter of eliminating shortcuts, which does not take many rounds. By their more irregular structure are graphs from GEOM-1D much harder to conquer. While graphs from GEOM-2D are very easy, so are those from GEOM-1D the hardest we know, and for very large n and specific choices of the parameter α, the algorithm performs unsatisfactory, with values of r exceeding 100. Minimize_Tree_Number succeeds to strongly reduce the number of trees, but does not always reduce it to one. Thereafter we have a situation that is similar to the worst-case example of Section 2.4 (though not entirely as bad). For GEOM-1D it is better to perform a stronger variant of Minimize_Tree_Number but for all other classes of graphs, which appear to be more relevant, this would make the program slower. More subtle tuning might help here.

The generator also offers special random graphs that model the web graph. The recent literature provides many models of this kind. The graph class CF-WEB exactly follows the description of Cooper and Frieze [9]. The idea is that with certain probability one either creates a new node with edges to the old nodes, or to add a number of outgoing edges to an existing node. The probability that a node is selected as source or destination increases with its degree. SIMPLE-WEB denotes a similar graph type, which can be generated more easily: starting with a small complete subgraph, new nodes are connected to those already present. A small fraction of the nodes has one edge leading to a node that is chosen arbitrarily at random (not only leading to those already existing). There is no degree-dependency. ACYC is a class with a similar graph structure. These graphs are generated by choosing all edges to run from nodes with large indices to nodes with smaller indices, choosing first one endpoint uniformly at random and then the other endpoint uniformly from among all remaining possibilities. Afterwards, all indices are reassigned using a random permutation. Graphs from all these classes have intermediate hardness, though those from CF-WEB, which are claimed to model the real web graph most accurately, are somewhat easier. The graphs from CF-WEB and SIMPLE-WEB become really easy if there are sufficiently many (10% is enough) edges from old nodes or nodes with random edges, respectively.

Other built-in graph classes based on random graphs are the following: for some parameter s, chosen by the user, OUT-STAR generates a graph with m edges in m/s phases, where in each phase a randomly selected node u gets outgoing edges to s randomly chosen nodes. IN-OUT-STAR is similar but here, during $m/(2 \cdot s)$ of the phases, the node u gets s ingoing edges. The graphs from both classes are relatively easy, generally with r values below 5. For small g, the graphs from IN-OUT-STAR tend to be somewhat harder than those from OUT-STAR.

These graph classes represent very different types. RAND and Geom-2d are more or less uniform (at least when looking from far) and have a small diameter. Geom-1d and CYCLE are also uniform but have a very large diameter. Information does not travel fast in graphs of this type. CF-WEB, SIMPLE-WEB and ACYC have a structure that mainly or entirely consists of edges pointing down to a sink. These trees are very wide and do not grow very deep. The star graphs are irregular. Their DFS trees are strongly branching, and undeep, with most nodes at the deepest levels. They almost look like BFS trees.

3.2 Experiments

In the following we report on the experimental behavior of our algorithm for the above described classes of graphs. We focus on demonstrating how r develops for varying input parameters rather than testing single very large instances. For the figures discussed in the following we mostly chose n between 10^4 and 10^6, and $g = m/n$ between 10 and 100. All values are the average of at least three experiments.

Figure 1 provides a plot of the ratio r for the input class RAND. r decreases noticeably with g. Interestingly, r appears to be independent of n. In the program, we first perform $\mathcal{O}(g^{1/3})$ phases (the actual formula is of the form $a_1 + a_2 \cdot g^{1/3}$, where a_1 is negative to account for the positive effect of the initial phase in which $3 \cdot n$ edges are processed together). After this, if the development of the forest looks promising, the edges are filtered and for random graphs only $\mathcal{O}(g^{1/3} \cdot n)$ edges remain. The number of edges to process hereafter is small, so the whole DFS search takes one scan through all edges plus $\mathcal{O}(g^{1/3})$ phases. In Figure 2 we show how the number of active nodes $n_{\text{act}} = n - \textit{depth}$, where *depth* is the depth of the forest, and active edges decreases as a function of the number of phases s. The reciprocal development of n_{act} is clearly visible. As in a random graph the edges that are not yet processed are independent of the processed edges, is the probability that any of these survives the filtering exactly $(n_{\text{act}}/n)^2$. In the above estimate that the program needs $\mathcal{O}(g^{1/3})$ phases for random graphs, we assumed that the fraction of the already processed edges that must be processed again is not higher than that of the edges that were processed before. The correctness of this assumption is supported by the other graphs in Figure 2.

In Figure 3 we compare the r-values for our two random web graph classes CP-WEB and SIMPLE-WEB. The IO behavior is quite different from the case of uniform random graphs: first of all, r increases when the graphs become denser. Furthermore, the dependence on n increases with the density of the graph. SIMPLE-WEB requires about twice as much IO as CP-WEB. One should be aware though that in these examples the degree of randomness decreases with g. If we would have fixed the fraction of random edges, then the graphs would have looked more and more like random graphs with increasing g and would have become easier and easier.

Figure 4 depicts how this IO is distributed over the rounds of the algorithm. Furthermore, it demonstrates the dependence between active nodes and scanned edges. The number of active edges decreases faster than the number of active edges, because an edge is thrown out when one of its endpoints in passive. For most graphs this means that if there are $x \cdot n$ active nodes left, that then the number if active edges is reduced to $x^2 \cdot n$ (but for non-uniform graphs the nodes with lower degree tend to disappear first).

In Figure 5 we demonstrate the relation between the number of scan phases and the total scan volume. For the graphs from CF-WEB it turns out that with growing degree the total volume increases much slower than the number of phases. For these graphs the reductions are of crucial importance.

In the following we turn to the class GEOM-1D. The performance strongly depends on a the geometric decrease parameter α. Roughly speaking, the higher the chance that nodes with larger distance are connected (large α) the smaller the total IO. Furthermore, r crucially depends on the average node degree as well. In fact this dependence is not even monotonic. Figure 6 demonstrates our observations for two α values.

Much better performance was observed for the inputs of the GEOM-2D class, even for denser graphs; compare Figure 7. They yield similar performance as pure random graphs. We attribute this to the larger expansion properties of GEOM-2D as compared to GEOM-1D.

4 Performance on Call Graphs

Real practical example might be even more convincing than test on generated graphs, whatever diverse in structure they may be. In this section we report on preliminary experiences of application of the program to call graphs of ATT. Within the coming weeks we are going to run tests on really big graphs with up to 1.5 billion edges.

The graphs we were testing had about 200,000 nodes and 32,000,000 edges. The graphs are very irregular, the maximum outdegree (for a typical example) is 14,464, while the maximum indegree is much higher: 26,151. At the same time there are 74,489 nodes with indegree zero. Though these graphs do not look like random graphs at all, our program had no problem to find DFS forests for them. Both rounds of the computation of the strong components gave r values just over 2. On a Pentium III with a 1 GHz clock frequence, the whole computation of the strong components takes about 3 minutes. Almost all nodes are either conatained in strong components of size 1, or in a single large component of size 105,887.

5 Conclusions and Further Research

In this paper we have presented heuristics that effectively deal with the semi-external DFS problem. Nevertheless, we can hope to become even better. $r < 10$, as we achieve for most graphs, appears to be a good result, but it leaves room for improvement. For some special classes, such as the 1-dimensional geometric graphs, the current algorithm is not very effective, and one may hope that approaches that deal with these graphs even lead to improvements for other classes.

One useful idea might be to rearrange the order of the edges on the file. So far, these edges were processed in a purely cyclical way (only throwing out edges that are not needed anymore). We have tested the effect of sorting this file on the initial node, this turned out to be a bad idea. However, there might be other rearrangements that do lead to improvements. Another interesting idea is to try to parallelize the algorithm. This appears hard, because internal DFS cannot really be parallelized. Still, there is some hope. For example, one might compute DFS trees for disjoint or partially overlapping parts of the graph and after this combine them somehow in order to come close to the final solution.

The main application of DFS is the computation of strong components, and the most important graph of which we would like to know the strong components is the web graph. As people appear not to be willing to share their knowledge about the web, we should try to gather this information ourselves (and make it available).

It is very likely that heuristics are effective also for other semi-external problems. The only fundamental limitation appears to be that the answer must have size $\mathcal{O}(n)$.

References

[1] Aggarwal, A., R.J. Anderson, 'A Random NC Algorithm for Depth First Search,' *Cominatorica*, 8(1), pp. 1–12, 1988.

[2] Aggarwal, A., J.S. Vitter, 'The Input/Output Complexity of Sorting and Related Problems,' *Communications of the ACM*, 31(9), pp. 1116–1127, 1988.

[3] Alon, N., J. H. Spencer, P. Erdős, *The Probabilistic Method*, John Wiley and Sons, 1992.

[4] Arge, L., U. Meyer, L. Toma, N. Zeh 'On External-Memory Planar Depth First Search,' *Proc. 7th Workshop on Algorithms and Data Structures*, pp. 471–482, LNCS 2125, Springer-Verlag, 2001.

[5] Bollobás, B., *Random Graphs*, Academic Press, London, 1985.

[6] Broder, A., R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata and A. Tomkins, J. Wiener, 'Graph Structure in the Web,' *Computer Networks*, Vol. 33, pp. 309–320, June 2000.

[7] Buchsbaum, A.L., M. Goldwasser, S. Venkatasubramanian, J.R. Westbrook, 'On External Memory Graph Traversal,' *Proc. 11th Symposium on Discrete Algorithms*, ACM-SIAM, 2000.

[8] Chiang, Y-J, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter, 'External-Memory Graph Algorithms,' *Proc. 6th Symposium on Discrete Algorithms*, pp. 139–149, ACM-SIAM, 1995.

[9] Cooper, C., A. Frieze, 'A General Model of Undirected Web Graphs,' *Proc. International Colloquium on Automata, Languages and Programming*, LNCS 2161, pp. 500–512, Springer, 2001.

[10] Diaz, J., J. Petit, M. Serna, ‘Random Geometric Problems on $[0,1]^2$,’ *Intern. Workshop on Randomization and Approximation Techniques in Computer Science*, LNCS 1518, pp. 294–306, Springer-Verlag, 1998.

[11] Hwang, F.K., D.S. Richards, P. Winter, *The Steiner Tree Problem*, Annals of Discrete Mathematics, 53, North-Holland, Amsterdam, 1992.

[12] Kapidakis, S., *Average-Case Analysis of Graph-Searching Algorithms*, PhD. Thesis, Department of Computer Science, Princeton University, 1990.

[13] Koch, T., A. Martin, ‘Solving Steiner Tree Problems in Graphs to Optimality,’ *Networks*, 32, pp. 207–232, 1998.

[14] Kumar, V., E. J. Schwabe, ‘Improved Algorithms and Data Structures for Solving Graph Problems in External Memory,’ *Proc. 8th Symposium on Parallel and Distributed Processing*, pp. 169–177, IEEE, 1996.

[15] Maheshwari, A., N. Zeh, ‘External Memory Algorithms for Outerplanar Graphs,’ *Proc. 10th International Symposium on Algorithms and Computation*, LNCS 1741, pp. 307–316, Springer-Verlag, 1999.

[16] Maheshwari, A., N. Zeh ‘I/O-Efficient Algorithms for Graphs of Bounded Treewidth,’ *Proc. 12th Symposium on Discrete Algorithms*, pp. 89–90, ACM-SIAM, 2001.

[17] Meyer, U., ‘External Memory BFS on Undirected Graphs with Bounded Degree,’ *Proc. 12th Symposium on Discrete Algorithms*, pp. 87–88, ACM-SIAM, 2001.

[18] Polzin, T., S. Vahdati Daneshmand, ‘Improved Algorithms for the Steiner Problem in Networks,’ *Discrete Applied Mathematics*, 112, pp. 263–300, 2001.

[19] Sleator, D.D., R.E. Tarjan, ‘A Data Structure for Dynamic Trees,’ *Journal of Computer and System Sciences*, 26(3), pp. 362–391, 1983.

[20] Sleator, D.D., R.E. Tarjan, ‘Self-Adjusting Binary Search Trees,’ *Journal of the ACM*, 32(3), pp. 652–686, 1985.

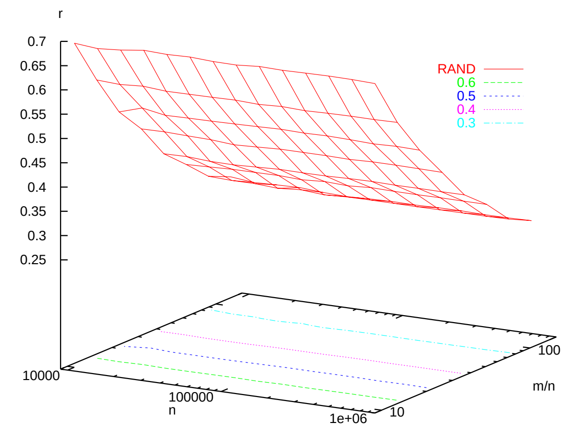

Figure 1: Ratio r for inputs of the class RAND.

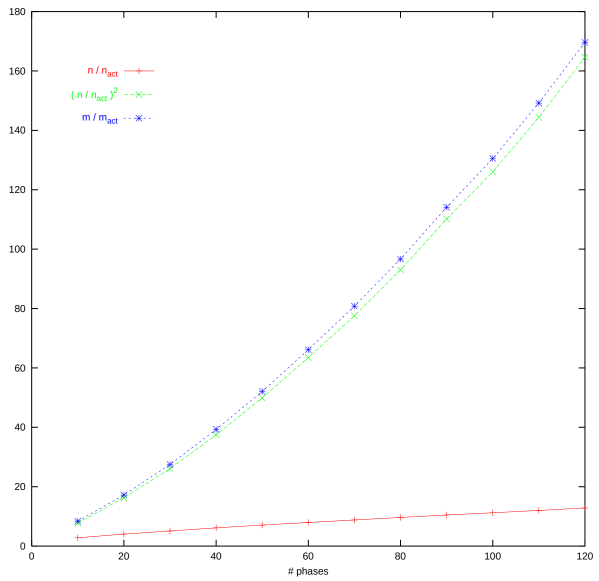

Figure 2: Lowest graph: Development of the number of active nodes n_{act} as a function of the number of performed phases s for random graphs with 10^6 nodes. Upper graphs: Fractions of the edges that survive the filtering. One curve gives $(n_{\text{act}}/n)^2$, the fraction of not-processed edges that survives, while the other curve gives the fraction of surviving processed edges.

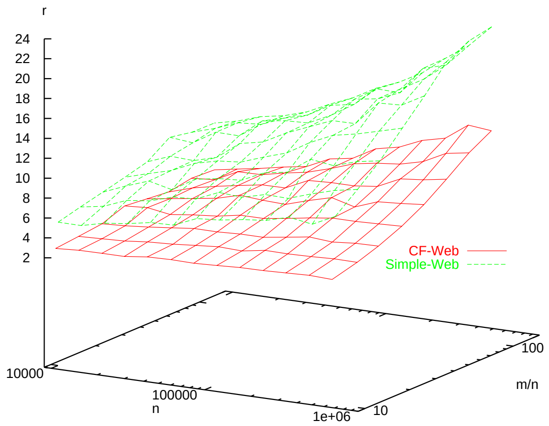

Figure 3: Comparison between the r-values for CF-WEB and SIMPLE-WEB. The graphs of CF-WEB are generated with $2 \cdot m/\sqrt{n}$ random edges. In all graphs of SIMPLE-WEB there are $0.02 \cdot n$ nodes with a random edge.

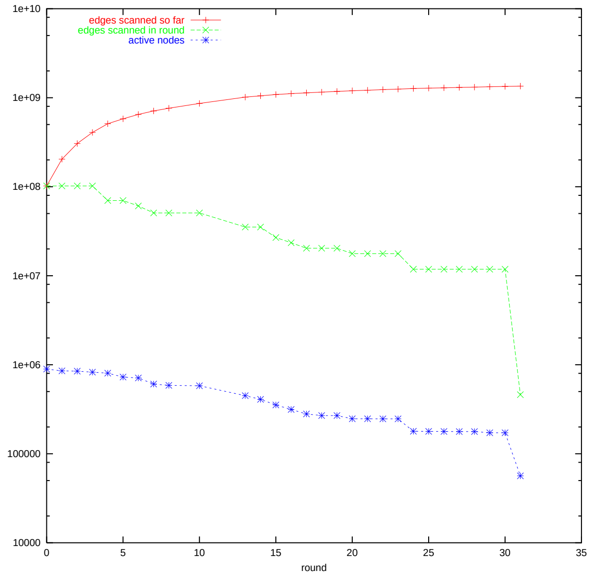

Figure 4: Measurements for a graph from CF-WEB with about $n = 10^6$ nodes and $m = 10^8$ edges. The diagram shows the sum of scanned edges as a function of the number of performed rounds (during a round all remaining edges are processed once); the number of edges scanned in each round; and the number of active nodes. The fewer nodes are active during a round the more edges can be pruned.

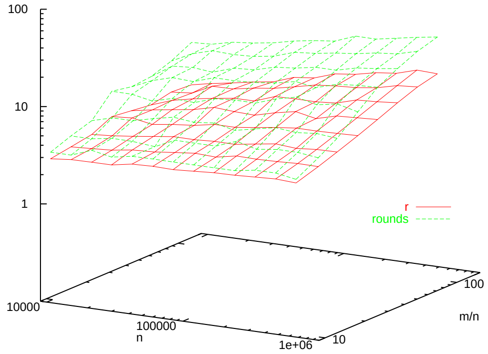

Figure 5: Comparison between the r-value and the number of rounds for CF-WEB. Observe that with growing g the number of rounds grows faster than the r-values.

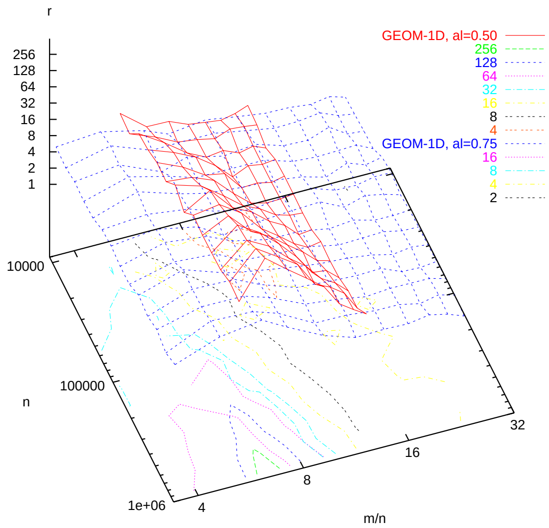

Figure 6: The influence of the parameter α of the GEOM-1D class. For larger α the program performs much better because of the better expansion properties of the graphs.

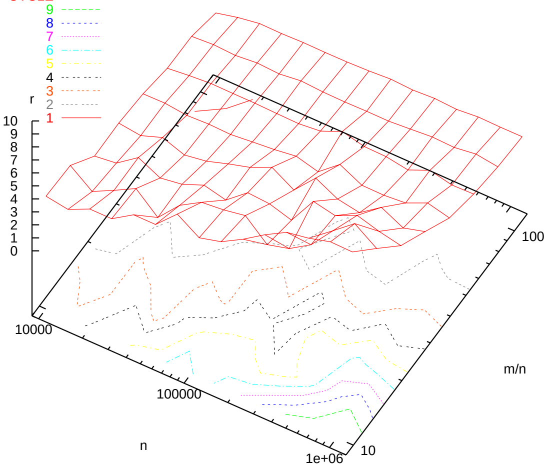

Figure 8: Plot of the r-values for inputs from the CYCLE class.

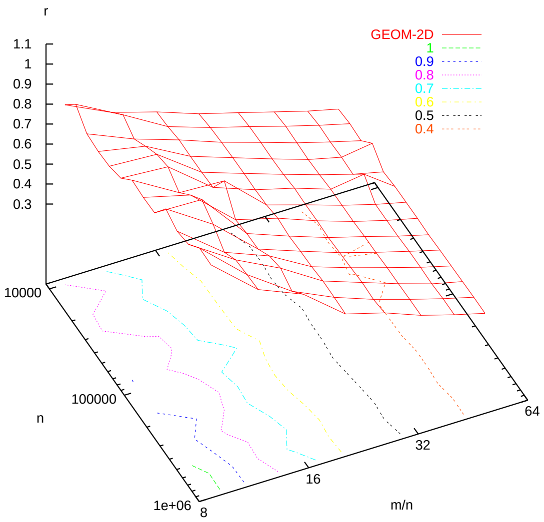

Figure 7: Plot of the r-values for inputs from the GEOM-2D class. In all cases we took $\alpha = 0.5$.

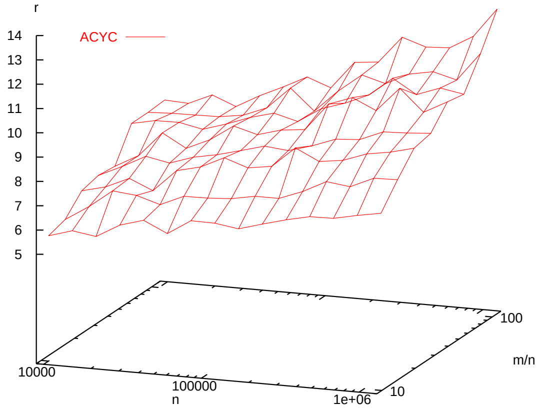

Figure 9: Plot of the r-values for inputs from the ACYC class.

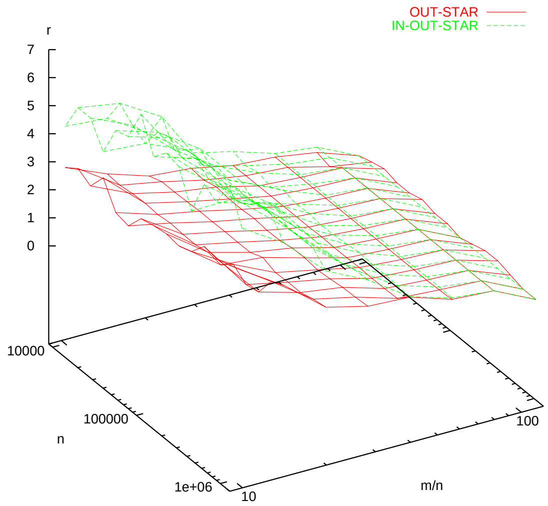

Figure 10: Plots of the r-values for inputs from the OUT-STAR and IN-OUT-STAR classes. All experiments performed with star degree $s = 50$.