A Heuristic Strong Connectivity Algorithm for Large Graphs

Adan Cosgaya-Lozano* and Norbert Zeh**

Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada
{acosgaya,nzeh}@cs.dal.ca

Abstract. We present a contraction-based algorithm for computing the strongly connected components of large graphs. While the worst-case complexity of the algorithm can be terrible (essentially the cost of running a DFS-based internal-memory algorithm on the entire graph), our experiments confirm that the algorithm performs remarkably well in practice. The strongest competitor is the algorithm by Sibeyn et al. [17], which is based on a semi-external DFS algorithm developed in the same paper. Our algorithm substantially outperforms the algorithm of [17] on most of the graphs used in our experiments and never performs worse. It thus demonstrates that graph contraction, which is the most important technique for solving connectivity problems on *undirected* graphs I/O-efficiently, can be used to solve such problems also on *directed* graphs, at least as a heuristic.

1 Introduction

Driven by the availability of massive amounts of data in a wide range of application areas, tremendous efforts have been made over the last two decades to develop algorithms that can process data sets beyond the size of a computer's main memory efficiently. Traditional algorithms perform poorly on such inputs, as most of these algorithms exhibit little or no access locality and cause a disk access for almost every computation step, which results in a slow-down by a factor of about 10^6 compared to processing the data in memory. *I/O-efficient algorithms*, on the other hand, are designed to access data sequentially or in large blocks, in order to reduce the number of disk accesses to the point where massive data sets can be processed efficiently.

In the algorithms community, much work has focused on developing provably I/O-efficient algorithms for a wide range of fundamental problems, particularly for geometric and graph problems. See [4, 19] for surveys. For graph problems,

* Supported by scholarships funded by the National Council for Science and Technology of Mexico, the Natural Sciences and Engineering Research Council of Canada, and Dalhousie University.

** Supported in part by the Natural Sciences and Engineering Research Council of Canada, the Canadian Foundation for Innovation, and the Canada Research Chairs programme.

much progress has been made on *undirected* graphs and special graph classes. In contrast, no provably efficient algorithms are known for general *directed* graphs. This lack of theoretical results motivates the study of *heuristic* techniques for processing directed graphs I/O-efficiently. The most successful effort so far is the depth-first search (DFS) algorithm by Sibeyn et al. [17], which is a *semi-external* algorithm; that is, it can process the edges of the graph I/O-efficiently if the vertices fit in memory. Since DFS is a central building block used in many classical graph algorithms, the algorithm of [17] provides a general tool for solving problems on directed graphs efficiently if the vertices fit in memory. If, on the other hand, the size of the vertex set exceeds the memory size, the performance of the algorithm deteriorates to that of an internal-memory DFS algorithm. Sibeyn et al. demonstrated the effectiveness of their approach in the semi-external case by using it to compute the strongly connected components of a variety of directed graphs. A directed graph is *strongly connected* if, for every vertex pair (x, y), there exists a directed path from x to y. The *strongly connected components* (SCC's) of a graph are its maximal strongly connected subgraphs (SCSG's).

While Sibeyn et al. used strong connectivity merely as an example to demonstrate the efficiency of their DFS procedure, we propose a heuristic specifically for computing SCC's in this paper. The aim is (a) to achieve a better performance than [17] on graphs whose vertex sets fit in memory and (b) to process graphs whose vertex sets do not fit in memory, which, our experiments confirm, the algorithm of [17] cannot do in a reasonable amount of time. Our algorithm achieves both goals on a variety of input graphs, outperforming the semi-external algorithm by a factor of 2–4 on most of the test graphs whose vertices fit in memory, and being able to efficiently process graphs well beyond the reach of the semi-external algorithm.

Our algorithm is based on graph contraction: it identifies and contracts strongly connected subgraphs until the graph fits in memory, and then computes the SCC's in internal memory. Thus, given its good performance, our algorithm demonstrates that, at least as a heuristic, graph contraction is useful for solving connectivity problems on directed graphs. This is interesting because this technique is the most important tool for solving connectivity problems on *undirected* graphs I/O-efficiently, both in theory and in practice.

The remainder of this paper is organized as follows. Section 2 reviews previous work on implementing I/O-efficient graph algorithms; Section 3 describes the algorithm; Section 4 discusses implementation details; Section 5 discusses experimental results; and Section 6 offers concluding remarks.

2 Previous Work

While much theoretical work has focused on developing I/O-efficient graph algorithms, much less is known about their practical efficiency. The main reason is their algorithmic complexity. Most of these algorithms build on a number of widely used primitives—list ranking, Euler tour construction, etc.—in addi-

tion to internal-memory algorithms that are used to process the parts of the graph loaded into memory. No good implementations of these primitives are publicly available, which makes implementing any I/O-efficient graph algorithm a formidable task, as it requires the implementation not only of the actual algorithm but also of a number of more elementary, yet non-trivial, building blocks.

In spite of these challenges, a number of experimental results have been obtained for *undirected* graphs. Dementiev et al. [10] provided a carefully engineered implementation of a minimum spanning tree (MST) algorithm based on ideas from [18]. Their algorithm is theoretically inferior to the MST algorithms of [1, 5, 8] but performs extremely well in practice. Ajwani et al. [2, 3] provided implementations of the undirected breadth-first search algorithm by Mehlhorn and Meyer [14] and obtained excellent results on a wide range of graph classes. The semi-external DFS algorithm by Sibeyn et al. [17] seems to be the only work that focused specifically on solving fundamental problems on directed graphs.

Other related work includes a large body of work on preprocessing large graphs, particularly road networks, for fast shortest path queries. The most recent results in this area include [6, 12, 13, 15].

3 A Contraction-Based Strong Connectivity Algorithm

This section describes a simple contraction-based SCC algorithm referred to as EM-SCC throughout this paper. Section 4 discusses its implementation.

The algorithm consists of two phases: a *preprocessing phase* and a *contraction phase*. The contraction phase looks for SCSG's in the input graph G and contracts each into a single vertex, thereby reducing the size of G without altering its connectivity. This process continues until the graph fits in memory, at which point the algorithm loads it into memory and computes its SCC's using an internal-memory algorithm. In this sense, EM-SCC resembles the connectivity algorithm for undirected graphs by Chiang et al. [8]. In the undirected case, however, the graph is guaranteed to fit in memory after a logarithmic number of contraction steps, while, in the directed case, the algorithm succeeds only if each round finds sufficiently many and large SCSG's to contract.

The contraction phase searches for SCSG's by loading memory-sized subgraphs of G into memory and computing their SCC's. The preprocessing phase tries to group the vertices and edges of G so that the chance of finding non-trivial SCC's in these subgraphs is maximized.

Next we discuss these two phases in detail. Throughout this discussion, we assume the input graph is connected. It is not hard, however, to extend the algorithm to disconnected graphs with little or no impact on its performance.

3.1 Preprocessing Phase

The preprocessing phase of EM-SCC is conceptually simple. It arranges the vertices of G in a list V_0 in the order of their first occurrences along an Euler tour of a spanning tree T of G. It stores the edges in a list E_0, which is the

concatenation of "one-sided" adjacency lists of the vertices in V_0 arranged in the same order as the corresponding vertices in V_0. The adjacency lists are one-sided in the sense that an edge xy is stored in the adjacency list E_x of x if $x > y$, and in E_y otherwise; vertices are compared by their positions in V_0.

The contraction phase discussed in Section 3.2 below sweeps the two lists V_0 and E_0 in tandem and processes maximal groups of consecutive vertices in V_0 that induce memory-sized subgraphs of G. Intuitively, the ordering of the vertices in V_0 produced by the preprocessing phase should ensure that the processed subgraphs are connected or have few connected components (in the undirected sense). Assuming sufficiently random edge directions and sufficiently many non-tree edges, this should lead to non-trivial SCC's in the processed subgraphs.

To compute lists V_0 and E_0, the algorithm has to compute the tree T, its Euler tour, and a ranking of the Euler tour. To compute the spanning tree, we use the MST algorithm by Dementiev et al. [10] (setting all edge weights to 1). Sorting and scanning the edge set of T suffices to compute an Euler tour of T. To rank this tour, we use the list ranking algorithm by Sibeyn [18].

Give the ranked tour, the algorithm finds the first occurrence of every vertex of G in the tour by sorting and scanning the node list of the tour, numbers the vertices of G in the order of these occurrences, and places them into V_0 in order. The edge list E_0 is constructed by sorting and scanning the edges of G three times: twice to label each edge with the numbers of its endpoints, and once more to arrange the edges in the order described above.

Before trying the simple preprocessing strategy discussed here, we experimented with a more sophisticated hierarchical clustering approach, which clustered vertices based on their degrees. The contraction phase then considered (contracted versions of) clusters of increasing size and decreasing density in its search for SCSG's. The intuition was that, assuming the edge directions are sufficiently random, dense graphs are more likely to contain large SCSG's, so that this degree clustering approach should lead to a rapid reduction of the graph size early on in the contraction phase. The cost of computing this clustering, however, was prohibitive, and the speed-up of the contraction phase compared to the simple preprocessing described here was insignificant.

3.2 Contraction Phase

The contraction phase of EM-SCC proceeds in *rounds*. Each round produces a more compressed version of G from the previous version by identifying SCSG's and contracting them. Let $G = G_0, G_1, \ldots, G_r$ be the sequence of graphs this produces; that is, round i produces graph G_i from graph G_{i-1}. The algorithm represents each graph G_i using two lists V_i and E_i whose structure is identical to that of V_0 and E_0 described in the previous section.

The ith round partitions V_{i-1} into subsets $V'_1, V'_2, \ldots, V'_k$ of consecutive vertices such that the graphs $G'_j = G_{i-1}[V'_j]$ they induce fit in memory. The algorithm loads these graphs into memory, one at a time, and identifies and contracts their SCC's. In more detail, the ith round scans V_{i-1} and E_{i-1} in tandem, collecting the vertices and edges in the current graph G'_j in memory. Let x be the

first vertex in V_{i-1} that belongs to G'_j, and let n_j and m_j respectively be the numbers of vertices and edges currently in G'_j. To decide whether to include the next vertex y in V_{i-1} in G'_j, the algorithm scans E_y and counts the edges whose lower endpoints belong to G'_j, that is, are no less than x; let m_y be their number.

If $n_j + 1$ vertices and $m_j + m_y$ edges fit in memory, the algorithm includes y in G'_j and partitions the edges in E_y into two groups: those with lower endpoints no less than x and those with lower endpoints less than x. It loads the former into memory (thereby adding them to G'_j) and appends the latter to an initially empty edge list E''_i to be processed at the end of this round. Then the algorithm proceeds to the next vertex in V_{i-1}.

If adding m_y edges to G'_j would make it exceed the memory size, the algorithm declares vertex y to be the first vertex of G'_{j+1} and appends its entire adjacency list to E''_i. Then it computes the SCC's of G'_j in memory, contracts them, and eliminates parallel edges that result from these contractions. At the end, the vertices in G'_j are labelled with ID's of their SCC's, that is, with the ID's of their corresponding super-vertices in G_i. The algorithm writes this mapping information back to V_{i-1} and appends the sorted list of super-vertices to V_i. The edges of the contracted version of G'_j are appended to an initially empty edge list E'_i. This finishes the processing of G'_j, and the algorithm starts to construct G'_{j+1} with y as its first vertex.

The ith round ends after the last vertex in V_{i-1} has been consumed. At this point, the algorithm discards the edge list E_{i-1}, but not V_{i-1}, as the information stored in V_{i-1} is necessary to compute the final component labelling of the vertices of G. If the algorithm numbers the vertices of G_i in increasing order as it produces them, V_i already contains the sorted vertex list of G_i. To produce E_i, the endpoints of all edges in E''_i have to be replaced with their corresponding super-vertices in G_i. Since the edges in E''_i are already sorted by their upper endpoints in G_{i-1}, a single scan of V_{i-1} and E''_i suffices to replace those endpoints. To replace the lower endpoints, the algorithm sorts the edges in E''_i by these endpoints and scans V_{i-1} and E''_i again. Finally, it concatenates the resulting list with E'_i, and sorts the concatenation primarily by upper endpoints (in V_i) and secondarily by lower endpoints. A single scan now suffices to eliminate duplicates from this list, which produces the edge list E_i of G_i.

3.3 Postprocessing

Let G_r be graph produced by the last round of the contraction phase; that is, G_r fits in memory. Then the algorithm loads G_r into memory and labels every vertex in V_r with the SCC containing it. What remains to be done is to copy these labels back to the vertices in G. This is done by iteratively copying these labels from V_i to V_{i-1}, for $i = r, r-1, \ldots, 1$.

To copy the labels from V_i to V_{i-1}, the algorithm sorts the vertices in V_{i-1} by their corresponding super-vertices in V_i. Now every vertex in V_{i-1} can be labelled with the label of its corresponding vertex in V_i using a single scan of the two sorted lists. Finally, the algorithm returns the vertices in V_{i-1} to their original order, in preparation for the next iteration.

4 Implementation Details

We implemented algorithm EM-SCC in C++ using the STXXL library [9], which provides I/O-efficient counterparts of the C++ STL containers and algorithms. In particular, we used STXXL vectors to store the vertex and edge lists of graphs, the STXXL sorting procedure to perform all sorting steps in the algorithm, and the STXXL priority queue implementation in the list ranking step of the preprocessing phase. The rest of this section discusses the most important implementation choices made in the different parts of the algorithm.

Graph representation. As already discussed, each graph G_i is represented by a vertex list V_i and an edge list E_i. In our implementation, every vertex in V_i was represented using two integers, one being its own ID, the other one the ID of the corresponding super-vertex in G_{i+1}.

Edges were represented as pairs of vertex ID's, that is, using two integers. The only exception was the addition of an extra integer to represent the edge weight up to and including the MST computation. This could have been avoided by modifying the MST implementation to compute an arbitrary spanning tree of an unweighted graph. We did not do this, as the MST computation did not account for a major part of the running time of our algorithm.

MST algorithm. We used the MST algorithm of [10] to compute the spanning tree T in the preprocessing phase. The implementation was available from [16]. That algorithm is a sweeping algorithm, which iteratively removes vertices by contracting the lightest edge incident to each processed vertex. This strategy can be implemented using an external priority queue or using an I/O-efficient bucket structure. The default implementation uses a bucket structure, as it results in slightly better performance; we had no reason to change this.

Euler tour. To compute the Euler tour of T, we used the standard strategy. We created two copies xy and yx of each spanning tree edge xy and sorted the resulting edge list by their first vertices. Then we scanned the sorted edge list and, for each pair of consecutive edges, xy_1 and xy_2, incident to the same vertex x, we made edge xy_2 the successor of edge y_1x in the Euler tour. This was easily implemented by storing the edges in an STXXL vector and using the STXXL sorting algorithm to implement the sorting step.

List ranking. The list ranking algorithm of [18] is a sweeping algorithm similar to the MST algorithm of [10]. The *down-sweep* removes vertices one by one from the list until only one vertex remains. For each removed vertex v, its two incident edges are replaced with a weighted edge between v's neighbours; the weight equals the length of the sublist between these two neighbours. The *up-sweep* re-inserts the removed vertices in the opposite order and computes the rank of each vertex v from the rank of one of the two vertices that became adjacent as a result of the removal of v in the down-sweep.

As discussed in [18], this algorithm can be implemented using a bucket structure, similar to the one used in the MST algorithm, to pass information between vertices in the two sweeps. An alternative implementation uses a priority queue and two stacks. Since our focus was not on engineering an optimal list ranking algorithm, we opted for the easier implementation using a priority queue.

Internal-memory SCC algorithm. We used the one-pass SCC algorithm described in [11] to compute the SCC's of graphs loaded into memory. The implementation of this algorithm requires two stacks to keep track of partially identified SCC's. In order to maximize the amount of memory available for processing each graph G'_j, we implemented them using STXXL stacks. This limited the memory footprint of the stacks to 4 pages.

Internal-memory graph representation. To maximize the size of the subgraphs that can be processed in internal memory in each round of EM-SCC, we used a fairly compact graph representation in internal memory, consisting of two arrays: an edge array and a vertex array. The edge array contained the concatenation of adjacency lists of the vertices. Since the SCC algorithm only needed access to the out-edges of each vertex, only those edges were stored in the adjacency lists. When accessing an adjacency list, it was known to which vertex this adjacency list belonged. Hence, the tail vertex of every edge did not have to be stored explicitly. This allowed us to represent every edge using a single integer storing the head vertex of the edge.

We represented every vertex using a two-integer record in the vertex array. The first integer represented the SCC containing this vertex (once identified), the other the index of the first edge in its adjacency list in the edge array. Vertex ID's did not have to be stored explicitly, as a consecutive numbering of the vertices allowed us to use the position of a vertex in the vertex array as its ID.

Since this representation stores edges in a different order than on disk, it was necessary to sort the edges by their tails to construct the internal-memory representation of a graph G'_j from its external one. This required the use of an initial edge representation using both its endpoints during the construction of the internal-memory graph representation. Once the edges were arranged in the right order, we dropped their tail endpoints, thus halving the memory requirements of the representation. Since the ability of our algorithm to identify SCC's improves with the size of the subgraphs it can process in memory, we decided to process subgraphs that occupied all of the available main memory (minus some buffer blocks for caching used by the STXXL vectors) using the compact representation. As a result, the initial sorting step required to construct this representation used the STXXL external sorting algorithm to sort up to $2M$ data, where M denotes the memory size.

5 Experimental Results

This section discusses our experimental results, comparing the performance of EM-SCC with that of the semi-external SCC algorithm by Sibeyn et al. (called SE-SCC here). First we describe our test environment and the data sets used in our experiments. Then we discuss the results of our experiments.

5.1 Environment and Settings

All experiments were run on a PC with a 3GHz Pentium-4 processor, 1GB of RAM, and one 500GB 7200RPM IDE disk using the XFS file system. The

operating system was Fedora Core 6 Linux with a vanilla 2.6.20 Linux kernel. The code was compiled using g++ 4.1.2 and optimization level –O3. All of our timing results refer to wall clock times in minutes.

Since STXXL allows the specification of the block size for data transfers between disk and memory, we experimented with different block sizes between 256KB and 8MB. A block size of 2MB resulted in the best performance, since EM-SCC accesses data in a mostly sequential fashion. This block size was used throughout our experiments. Two additional parameters control the amount of memory allocated to the LRU pager used by STXXL vectors to cache accessed blocks. The first parameter is the page size as a multiple of the block size. Data is swapped one page at a time. The other parameter is the number of pages to be cached. We set both parameters to 2, as the mostly sequential data accesses of EM-SCC did not benefit substantially from a bigger cache, but this would have left less memory for the graphs to be processed in memory.[1]

5.2 Data Sets

We tested both algorithms on synthetic graphs and real web graphs. The synthetic graphs were generated using the same data generator used by Sibeyn et al. [17]. The web graphs were produced by real web crawls of the .uk domain, the .it domain, and from data produced by a more global crawl using the Stanford WebBase crawler. They were obtained from `http://webgraph.dsi.unimi.it/`, and their characteristics are shown as part of Table 1. Next we give an overview of the types of synthetic graphs used in our experiments.

Random: These graphs were generated according to the $G_{n,m}$ model; that is, m edges were generated, choosing each edge endpoint uniformly at random from a set of n vertices.

Cycle: The vertices were evenly spaced on a ring, and every vertex had out-edges to its $d = m/n$ nearest neighbours.

Geometric 1D: The vertices were evenly spaced on a ring of length n. Edges were generated by choosing their tails uniformly at random. If u was chosen as the tail of an edge, vertex v was chosen to be the head of this edge with probability proportional to α^d, where $\alpha < 1$ and d is the distance between u and v. In our experiments, we chose $\alpha = 0.9$.

Geometric 2D: The vertices were placed on a $\sqrt{n} \times \sqrt{n}$ grid wrapped around at the edges to form a torus. Edges were generated as for geometric 1D graphs, but d was chosen to be the Manhattan distance between u and v in the grid. Here we chose $\alpha = 0.8$.

[1] Using a single disk, a block size of 2MB and a page size of two blocks is equivalent to using a block size of 4MB and a page size of one block. We chose the former option because we also tested our algorithms using two disks, in which case the blocks of each page can be assigned to different disks. Using two disks, our algorithm experienced a speed-up of about 30%. Since the semi-external algorithm wasn't able to take advantage of multiple disks, we do not discuss the timings using two disks in detail here.

Out-star: Given a star degree s, this graph was generated in $\lfloor m/s \rfloor$ rounds. In each round, a tail vertex and s head vertices were chosen uniformly at random. Then edges were added from the tail to the chosen head vertices. We chose $s = 1000$ in our experiments.

In-out-star: This construction was similar to the out-star construction, but half of the rounds directed the generated edges towards the centre of the star. Again, we chose $s = 1000$.

Simple web: This construction started with a small complete subgraph and added new vertices by connecting them to the current graph. Afterwards, a small fraction (5% in our case) of random edges were added.

5.3 EM-SCC vs. SE-SCC

Table 1 shows the running times of EM-SCC and SE-SCC on different synthetic inputs and on the three web graphs. For the synthetic graphs with 2^{25} vertices, EM-SCC outperformed SE-SCC by a factor between 2 and 4. The only exception were random graphs and geometric 2D graphs, where SE-SCC took only slightly longer than EM-SCC. For the two smaller web graphs, EM-SCC outperformed SE-SCC by a factor between 3 and 4. As can be observed, the performance of SE-SCC depends strongly on the structure of the input graph, whereas (surprisingly) the performance of EM-SCC is much more immune to these variations. Sibeyn et al. characterized geometric 1D graphs as being among the hardest inputs for their algorithm, and geometric 2D and random graphs as being among the easiest inputs. This is in line with our observations. On the other hand, cycle graphs were mentioned as easy inputs in [17], while this was the synthetic input that took SE-SCC the longest to process in our experiments.

The remaining inputs had at least 2^{26} vertices and were beyond the reach of SE-SCC on our hardware, as the vertex set no longer fit in memory. (See [7] for a discussion of the graph representation used by SE-SCC and approximate vertex numbers it can process without using virtual memory.) We ran SE-SCC on the smallest of these graphs (with 2^{26} vertices and 2^{29} edges), using virtual memory, and terminated each of these test runs after 12h without SE-SCC having produced any result. Since the performance of SE-SCC on the semi-external instances of random and geometric 2D graphs was comparable to that of EM-SCC, we expected that SE-SCC would have the least difficulties to process larger instances of these graph classes, and we let the experiments on these inputs run for 24h. Again, SE-SCC did not finish within this amount of time.

In contrast, EM-SCC was able to process most of the test graphs in under two hours, while none took more than 2 1/2 hours. The exceptions were the out-star graphs and the sparsest of the in-out-star and simple web graphs. The next section discusses possible reasons why EM-SCC could not process these inputs, which sheds some light on its limitations.

5.4 The Effect of Graph Structure

The ability of EM-SCC to process certain graphs is limited by the available amount of main memory. The input graph needs to have few enough SCC's to

Cycle						Geometric 1D					
n	m	m/n	EM	SE	SCC's	n	m_r	m/n	EM	SE	SCC's
2^{25}	2^{29}	16	58	208	1	2^{25}	2^{29}	13.2	51	161	11
2^{26}	2^{29}	8	71	—[1]	1	2^{26}	2^{29}	7.2	65	—[1]	45084
2^{27}	2^{29}	4	94	—	1	2^{27}	2^{29}	3.8	90	—	5.2m
2^{26}	2^{30}	16	120	—	1	2^{26}	2^{30}	13.2	103	—	17
Geometric 2D						In-out-star					
n	m_r	m/n	EM	SE	SCC's	n	m	m/n	EM	SE	SCC's
2^{25}	2^{29}	15.6	58	62	7175	2^{25}	2^{29}	16	63	141	22490
2^{26}	2^{29}	7.9	70	—[2]	45060	2^{26}	2^{29}	8	79	—[1]	2.6m
2^{27}	2^{29}	4.0	91	—	5.2m	2^{27}	2^{29}	4	—	—	—
2^{26}	2^{30}	15.6	117	—	18	2^{26}	2^{30}	16	134	—	44800
Out-star						Simple web					
n	m	m/n	EM	SE	SCC's	n	m	m/n	EM	SE	SCC's
2^{25}	2^{29}	16	65	109	33m	2^{25}	2^{29}	16	63	113	1.6m
2^{26}	2^{29}	8	—[3]	—[1]	—	2^{26}	2^{29}	8	86	—[1]	10.6m
2^{27}	2^{29}	4	—[3]	—	—	2^{27}	2^{29}	4	—[3]	—	—
2^{26}	2^{30}	16	—[3]	—	—	2^{26}	2^{30}	16	133	—	3.2m
Random						Real web graphs					
n	m	m/n	EM	SE	SCC's	n	m	m/n	EM	SE	SCC's
2^{25}	2^{29}	16	61	63	12	18.5m	298.1m	16.1	29	104	3.8m
2^{26}	2^{29}	8	77	—[2]	45173	41.3m	1,150.7m	27.9	116	517	6.7m
2^{27}	2^{29}	4	109	—	5.2m	118.1m	1,019.9m	8.6	124	—[1]	38.5m
2^{26}	2^{30}	16	133	—	17						
2^{27}	2^{30}	8	159	—	90279						
2^{28}	2^{30}	4	345	—	10.4m						

Table 1. Experimental results on synthetic data and real web graphs. Dashes indicate inputs that could not be processed by the algorithm. For geometric 1D and 2D graphs, m_r denotes the number of edges requested to be generated. Since the data generator filters duplicate edges for these two graph types, the actual number of edges, m, is less than m_r. The ratio m/n in the table reflects this. Notes: (1) experiment terminated after 12h; (2) experiment terminated after 24h; (3) no further compression after a small number of initial contraction rounds, but graph still beyond memory size.

fit in memory, and the SCC's have to be composed of short enough cycles for EM-SCC to find them as part of the memory-sized subgraphs it processes. The inability of EM-SCC to process all but one of the out-star graphs nor the sparsest of the in-out-star and simple web graphs reflects these limitations.

Since EM-SCC was not able to process these graphs, we can of course only extrapolate from the properties of the graphs in these classes the algorithm *was* able to process. The smallest simple web graph had about 1.6m SCC's, and the smallest out-star graph had about 33m SCC's. Compared to at most a few thousand SCC's in the smallest cycle, geometric 1D and 2D, and random

graphs, these graphs have significantly more SCC's. For the bigger and sparser inputs, we suspect that the number of SCC's exploded, preventing EM-SCC from compressing the graph down to memory size.

The smallest in-out-star graph had about 22,000 SCC's, which is more than for cycle, geometric 1D and 2D, and random graphs, but significantly less than for out-star and simple web graphs. Therefore, there are two possible explanations for the inability of EM-SCC to process the sparsest in-out-star graph: either the lower density of the graph again resulted in an explosion of the number of SCC's, or the SCC's consisted of very long cycles, which EM-SCC was not able to find using the amount of main memory available on our test machine.

Another interesting observation we made in our experiments was the lack of a smooth transition between graphs EM-SCC could process efficiently and graphs it could not process at all. More precisely, all the graphs it was able to process required one or two contraction rounds, followed by a final round computing the SCC's in internal memory. On the other hand, for all inputs the algorithm was not able to process, it took only a few contraction rounds to reach a stage where no further contraction took place. For the out-star and in-out-star graphs, contraction stopped after at most 4 rounds. It is possible that the algorithm had found all SCC's at that point, but there simply were too many. For the simple web graph of density 4, it took 18 contraction rounds to reduce the graph by only 33%, and subsequent rounds achieved no further contraction. We suspect that more main memory would have helped in this case to identify and contract SCC's consisting of long cycles.

6 Conclusions

We have presented a contraction-based heuristic algorithm, EM-SCC, for computing the strongly connected components of large graphs. Our algorithm demonstrates that graph contraction is a useful tool for computing SCC's of large graphs, as it was able to process a wide range of input graphs faster than the currently best algorithm by Sibeyn et al., and it was able to process graphs whose vertex sets did not fit in memory.

The main limitation of EM-SCC is that it relies on the graph to have relatively few SCC's, consisting of relatively short cycles. This limitation seems impossible to overcome using graph contraction alone.

An interesting strategy that might speed up EM-SCC on inputs it *can* process is the use of pipelining to pass data between successive contraction rounds without writing this data to disk.

References

1. J. Abello, A. L. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.
2. D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external-memory BFS algorithms. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 601–610, 2006.

3. D. Ajwani, U. Meyer, and V. Osipov. Improved external memory BFS implementation. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 2007.
4. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–357. Kluwer Academic Publishers, 2002.
5. L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004.
6. R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In *Proceedings of the 7th International Workshop on Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer-Verlag, 2008.
7. A. Beckmann. Parallelizing semi-external depth first search. Master's thesis, Martin-Luther-Universität, Halle, Germany, October 2005.
8. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.
9. R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2007.
10. R. Dementiev, P. Sanders, D. Schultes, and J. F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *Proceedings of IFIP TCS: 3rd International Conference on Theoretical Computer Science*, pages 195–208, 2004.
11. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
12. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contration hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Workshop on Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer-Verlag, 2008.
13. A. V. Goldberg and R. Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*, pages 26–40, 2005.
14. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer-Verlag, 2002.
15. P. Sanders, D. Schultes, and C. Vetter. Mobile route planning. In *Proceedings of the 16th European Symposium on Algorithms*, volume 5193 of *Lecture Notes in Computer Science*, pages 732–743. Springer-Verlag, 2008.
16. D. Schultes. External memory minimum spanning trees. http://algo2.iti.uni-karlsruhe.de/schultes/emmst, 2003.
17. J. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth-first search on directed graphs. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 282–292, 2002.
18. J. F. Sibeyn. External connected components. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 468–479. Springer-Verlag, 2004.
19. J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.