Listing Dense Subgraphs in Small Memory

Patricio Pinto, Nataly Cruces and Cecilia Hernández
Department of Computer Science
University of Concepcion
Concepcion, Chile
Email: {patriciopinto,natalycruces,cecihernandez}@udec.cl

***Abstract*—Listing relevant patterns from graphs is becoming increasingly challenging as Web and social graphs are growing in size at a great rate. This scenario requires to process information more efficiently, including the need of processing data that cannot fit in main memory. Typical approaches for processing data using limited main memory include the streaming and external memory models. This paper addresses the problem of listing dense subgraphs from Web and social graphs using little memory.**

We propose an external memory algorithm based on K-way merge-sort for clustering and reordering input graphs. We also propose mining heuristics that work well with different stream orders such as URL, BFS, and cluster-based. Our experimental evaluation shows that on Web graphs, in comparison with the in-memory algorithm, the streaming mining heuristic is able to find between 70 and 96% of edges participating in dense subgraphs, uses only between 17 and 25% of the memory, and running times are between 34 and 65%. We further consider an application that uses these dense subgraphs for compressing Web graphs with a representation that enables querying the collection of subgraphs for pattern recovery and basic statistics without decompression.

***Keywords*-Web Graphs, Graph Pattern Listing, Streaming Algorithms, External Memory Algorithms**

I. Introduction

Discovering patterns from graphs is important for many applications including the Web, social networks, and biological applications among many others. For instance, patterns found in Web graphs and social networks are used to discover link spams and in ranking algorithms for searching the Web. In biological networks, graphs patterns, such as cliques in protein structures are used for modeling and predictions [23]. However, these graphs are growing at an incredible rate. For instance, the Web consists of more than a trillion of pages, increasing in number every day [1]. Social networks are also growing very fast, Facebook is over 1.1 billions active users and twitter was over 500 millions in July, 2013 [2].

At such growth rates comes a need to process that information more efficiently, including the need of processing data that do not fit in main memory. Typical approaches using limited memory include the streaming and external memory models. In the streaming model data is processed sequentially in one or a few passes using limited memory, whereas in the external memory model the idea is to keep data in secondary storage and bring data selectively to main memory to improve I/O performance.

This paper addresses the problem of finding and listing graph patterns based on dense subgraphs from large graphs. We propose an external memory algorithm based on K-way merge-sort for clustering and reordering input graphs. We also propose streaming mining heuristics which work well with different stream orders such as URL, BFS, and cluster-based. We provide experimental evaluation that shows that our external memory algorithm reduces memory usage preserving the quality of the in-memory solution. We also provide a streaming mining heuristic that uses little memory and achieves good quality for stream orders that exploit locality of reference. We further consider an application for compressing Web graphs and social networks that is based on the listed dense subgraphs. The compressed structure not only allows the recovery of the collection of dense subgraphs but also enables basic queries such as obtaining the number and average size of cliques and bicliques as well as other dense subgraphs.

II. Related Work

Finding relevant patterns on graphs have been addressed for some time in different applications. In the context of the Web, Donato et al. [9] used several web mining techniques to discover the structure and evolution of the Web graph, such as weakly and strongly connected components, depth-first and breath-first search. Other proposals use graph algorithms to detect spam farms [22], [12]. Saito et al. [22] present a method for spam detection based on classical graph algorithms such as identification of weak and strong components, maximal clique enumeration and minimum cuts. Gibson et al. [12] propose large dense subgraph extraction as a primitive for spam detection. Their algorithm used efficient heuristics based on the idea of shingles [4].

The streaming model is an important model of computation for processing massive data sets [18], [19]. The most restrictive streaming model allows $O(\text{ polylog } n)$ space and a single or a few passes over the data. To address graph problems in the streaming model, several more relaxed

[1] http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html
[2] http://www.statisticbrain.com/twitter-statistics/

978-1-4799-6953-1/14 $31.00 © 2014 IEEE
DOI 10.1109/LAWeb.2014.16

models have been proposed, such as the semi-streaming, w-stream and stream-sort models. The semi-streaming model allows one or a few sequential read-only passes through the graph, but in addition allows $O(n \text{ polylog } n)$ space, which means that vertices and some information about them can be stored, but not all the edges [10], [19], [27]. The w-stream model is similar, but it allows temporary streams to read/write on disk [21], [27]; and the stream-sort model, which not only creates intermediate streams but also sorts them in a single pass [2], [27].

Feigenbaum et al. [10] propose semi-streaming algorithms for finding approximations to the unweighted maximum dense subgraph matching problem with an approximation ratio $2/3-\epsilon$ in $O(\frac{\log 1/\epsilon}{\epsilon})$ passes using $O(n \log n)$ memory, where $0 < \epsilon < 1/3$. The proposed algorithm is based on first finding a bipartition, then using a matching of the graph, and then finding a set of simultaneous length-3 augmenting paths. The maximum dense subgraph matching algorithm increases the size of the matching by repeatedly finding a set of simultaneously length-3 augmenting paths. They also provide a semi-streaming algorithm for finding a weighted matching. They use edge weights of edges in the stream and compare them with the sum of the weights of the edges in the current matching M. Demetrescu et al. [8] show that the single-source shortest path problem in directed graphs can be solved in the w-stream model in $O(n \log^{3/2} n)/\sqrt{s})$ passes. For undirected connectivity they propose an $O(n \log n)/s$ passes algorithm.

Aggarwal et al. [1] propose a model for dense pattern mining using summarization of graph streams. They define dense patterns based on node-affinity and edge-density of patterns in a general way. Their approach removes small and large adjacency lists a priori because the dense pattern mining definition does not consider them as relevant. On the other hand, running time is in the order of thousand processed edges per second. More recently, Sariyüce et al. [24] propose incremental streaming algorithms for k-core decomposition, where a k-core is defined as a maximal connected subgraph in which every vertex is connected to at least k nodes in the subgraph. The core decomposition of a graph is the problem of finding the set of maximum k-cores of all vertices in the graph. Thus, an algorithm to find k-cores of a graph removes all vertices with degree less than k with their corresponding adjacency edges. The authors propose streaming algorithms supporting insertion and removal of edges for dynamic networks. The algorithms require reordering unprocessed vertices in subgraphs. Stanton and Kliot [25] address the problem of distributed graph partitioning using streaming for directed or undirected graphs. They evaluate different heuristics performed on various stream orders. They find that a greedy linear deterministic algorithm works best together with BFS stream order.

External memory algorithms define memory layouts that are suitable for graph algorithms reducing random accesses to disk. This model has been used for different basic problems, such as scanning, sorting, permuting, and other graph algorithms such as traversal algorithms and graph connectivity[26], [17]. Cheng et al. [7] propose a external memory algorithm for the maximal clique enumeration problem on undirected graphs. The algorithm is based on defining a partition-based strategy that avoids random access. Their algorithm I/O complexity is $O(k \cdot scan(|V+E|)$, $O(kT)$ CPU time, and $O(M)$ memory space, where $k = min\{\frac{|V|(\phi_{deg})^2}{M}, |V|\}$, T is the CPU time complexity of the in-memory algorithm, M is the memory size, and ϕ_{deg} is the maximum vertex degree.

III. Problem Definition

Let $G(V, E)$ a directed graph, with $n = |V|$ vertices and $m = |E|$ edges. We define the neighbors of a vertex v as $adj(v) = \{u : (v, u) \in E\}$. We assume that input graphs use the adjacency list representation, where vertices have unique ids and each adjacency list describes the set of neighbors of a vertex.

Our dense subgraph pattern is described in Definition 1. This definition is based on a complete bipartite graph pattern with an important difference. We add self-loops in each vertex adjacency list, that is, for each vertex v we add edge (v, v) in its adjacency list. This allows us to discover cliques as well as complete bipartite cliques. Similar patterns are defined by Kumar [14], where they only consider complete bipartite graphs.

Definition 1: A *Dense subgraph is based on a bipartite pattern with set overlap* $H(S, C)$ of $G = (V, E)$ is a graph $G'(S \cup C, S \times C)$, where $S, C \subseteq V$.

Note that Definition 1 not only includes cliques ($S = C$) and bicliques ($S \cap C = \emptyset$), but also more general subgraphs.

The problem we want to solve is given a graph G, discover dense subgraphs and list them using small memory. We do not aim for an exact solution, but rather for fast and memory efficient heuristics. We focus on large graphs such as Web and social networks, where $n << m$. These graphs are power-law and present high similarity of adjacency lists and locality of reference. In order to design memory efficient algorithms, we consider the external memory and streaming models. In the context of the external memory model, we use the standard I/O complexity notation [26] in the analysis: M is the main memory size, B is the disk block size, $scan(N) = \Theta(\frac{N}{B})$ I/O, and $sort(N) = \Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. To use the streaming model, we consider an input *graph stream* as a stream of adjacency lists as a sequence $X = \{v_0 : adj(v_0), v_1 : adj(v_1), ..., v_n : adj(v_n)\}$.

IV. Heuristics and Algorithms

We first discuss the algorithm that we use as reference (*in-memory*) algorithm (Algorithm 1). We have used it successfully for discovering graph patterns for compressing Web and social graphs [15], [16]. The Algorithm 1 is iterative,

and each iteration consists of a clustering and a mining phase. The clustering phase consists of grouping similar adjacency lists (lists that have similar neighbors) and the mining phase processes the information on each cluster in order to discover the most relevant dense subgraphs, and then extract them from the graph. The next iteration takes as input the remaining graph given by the previous iteration.

The clustering algorithm is based on finding similar adjacency lists using min-hashing similarity [4]. The algorithm represents each adjacency list with P fingerprints (hash values), generating a matrix of fingerprints of $|V|$ rows and P columns. Then it traverses the matrix column-wise. At stage i the matrix rows are sorted lexicographically by their first i column values, and the algorithm groups the rows with the same fingerprints in columns 1 to i, with $0 < i \leq P$. When the number of rows in a group falls below a small number ($threshold$), it is converted into a cluster formed by the vertexes corresponding to the rows. As shown on a previous work, it is sufficient to use $P = 2$ [16] for achieving good results.

During the second phase a mining algorithm is applied on each cluster to discover and extract dense subgraphs. This algorithm first computes frequencies of the nodes mentioned in the adjacency lists of a cluster, and sorts the list by decreasing frequency. Then, each list is inserted into a prefix tree (Definition 2), using a function cost based on the size of dense subgraphs (Definition 3).

Definition 2: We denote a *prefix tree* as $T = (N, A)$, where N is the set of nodes in the tree and $a = (n_x, n_y) \in A$, where $n_x, n_y \in N$ and n_x is the parent of n_y. We define a branch b as the path from the root to a leaf. Each node n_i in a branch has a label and a set S. The label in the node represents a vertex in an adjacency list and the set S consists of all the vertices that share the adjacency list from the root to the n_i in the branch. We define a set C of a node n_i as the set of all the node labels from the root to n_i. The prefix tree allows different nodes in the tree to have the same label.

Definition 3: We consider dense subgraphs whose sizes follow the function $f(T) = max\{|S_i| \cdot |C_i|\}$ on different branches of the tree, with $|S_i| > 1$ and $|C_i| > 1$.

V. External Memory Algorithm

The K-way external merge sort works in two phases [11]. The first is a "run formation" phase, where N input data are streamed in main memory using memory pieces of size M. Each piece of size M is sorted, having at the end of the phase N/M sorted runs. The second phase is the "merge phase", where groups of K runs are merged together. Runs in the merge phase are sorted using buffers of size B. The merge phase might take more than one pass; in each pass one buffer of size B from each run is maintained in main memory and one buffer is used for streaming out sorted runs. Sorting K runs is done using a Heap data structure. Since the memory usage of the algorithm is bound to M and the buffer size is B, $K = \frac{M}{B} - 1$ buffers are used for input and one for output. The overall I/O performance of the algorithm is $O(N/B \log_{M/B} N/M)$, which is a $sort(N)$ primitive in the external memory model.

Algorithm 1 In-memory algorithm for listing dense subgraphs.

Input: G: input graph, P: number of fingerprints, $threshold$: threshold for clustering, $Iters$:iterations, $size\_thr$: minimum size to list ($|S| \cdot |C|$).
Output: Output: $dscol$: Collection of dense subgraphs.

```
1: dscol ← ∅
2: for (i = 1 to Iters) do
3:    Matrix M ← computeFingerprints(G, P)
4:    Clusters C ← getClusters(M, G)
5:    for (c ∈ C ) do
6:       ds ← mine(G, c, size_thr)
7:       dscol.add(ds)
8:    end for
9: end for
```

The external memory algorithm we propose uses the K-way external merge-sort in two different ways: One for sorting the matrix of nP hashes, for the clustering phase and the other for reordering the input graph by cluster id. During the clustering phase, the algorithm computes hashes and sorts them by columns using external merge-sort; clusters ids are defined based on the conditions of pairs of hashes (given that $P = 2$). The vertices ids of each cluster are used for sorting the input graph based on cluster ids. In summary, this external memory algorithm requires two external $sorts$, one over the matrix nP and one for permuting the graph based on vertex id. Therefore, the algorithm is $O(sort(2n) + sort(n + m))$, that is, $O(sort(n + m))$ I/O complexity. This complexity does not consider the mining part of the algorithm.

For the mining phase, the reordered graph is scanned by cluster. We use whatever main memory requires each cluster, which requires at most $O(V_c + E_c)$ time, where V_c is the number of vertices in the cluster and E_c the number of edges. Such external algorithm is given in Algorithm 2.

VI. Streaming Algorithms and Stream Orders

The main idea of our streaming heuristics is to take advantage of the locality of reference found on Web and social graphs. The question we want to answer is whether we can apply a mining algorithm reading a sequential window of neighbors (w) from the input graph stream. We based the heuristic on the mining algorithm we use in the second phase of the *in-memory* algorithm. Our streaming algorithm belongs to the w-stream model. The idea of this model is at each pass one input stream is read, one output stream is written, and data items have to be processed using limited

Algorithm 2 External memory algorithm based on K-way merge sort for listing dense subgraphs.

Input: G, P, $threshold$, $Iters$, $size\_thr$, M, B, K.
Output: Output: $dscol$: Collection of dense subgraphs.

1: $dscol \leftarrow \emptyset$
2: $msFinger.init(M, B, K)$, $msPerm.init(M, B, K)$
3: **for** $(i \leftarrow 1$ **to** $Iters)$ **do**
4: **for** $((v, adj) \in G)$ **do**
5: $fingers \leftarrow computeFingerprints(v, adj, P)$
6: $msFinger.add(v, fingers)$
7: **end for**
8: $sortFingersFile \leftarrow msFinger.extsort()$
9: $cls \leftarrow msFinger.getClusters(sortFingersFile)$
10: $permGpFile \leftarrow msPerm.extperm(M, B, K, cls)$
11: **for** $(cluster \in permGpFile)$ **do**
12: $ds \leftarrow mine(cluster, size\_thr)$
13: $dscol.add(ds)$
14: **end for**
15: **end for**

space. The output stream produced at pass i is the input stream at pass $i + 1$.

Our hypothesis is that if the graph stream has locality of reference, then we can detect different clusters implicitly just sorting the adjacency list by decreasing neighbor frequency and then building a prefix tree every time we find a different frequent first neighbor in any of the adjacency lists. Therefore, after sorting the adjacency lists by decreasing frequency we define a forest of prefix trees, where the root of each prefix tree serves as the identification of clusters in the window. We define a *forest* of prefix trees in Definition 4.

Definition 4: We denote FT a prefix tree forest as a collection of prefix trees, $FT = (r_1, T_1), (r_2, T_2), ..., (r_k, T_k)$, where r_i is the node id of the root of prefix tree T_i, for any $0 < i < k$, with k prefix trees in a window w.

Figure 1 shows an example that illustrates the algorithm. The example shows a window, w, of an input graph. Figure 1-(a) shows the frequency count of each of the neighbors in the adjacency lists showed in Figure 1-(b).

Figure 1-(b) also shows the same graph partition sorted by decreasing neighbor frequency, where we can observe that two clusters are identified. With this information, it is possible to build two prefix trees, where the root of the first is the node 2 and the root of the second is 14 (Figure 1-(c)). Figure 1-(c) also shows the identified dense subgraphs, where the first is $S = (1, 2, 3, 5)$, $C = (2, 3, 4)$, and the second is $S = (11, 12, 13)$, $C = (14, 18, 16)$. Finally, Figure 1-(d) shows our application which compresses the listed dense subgraphs by using a compact representation that is explained in Section VIII. This representation has been previously used for compressing Web and social graphs [15].

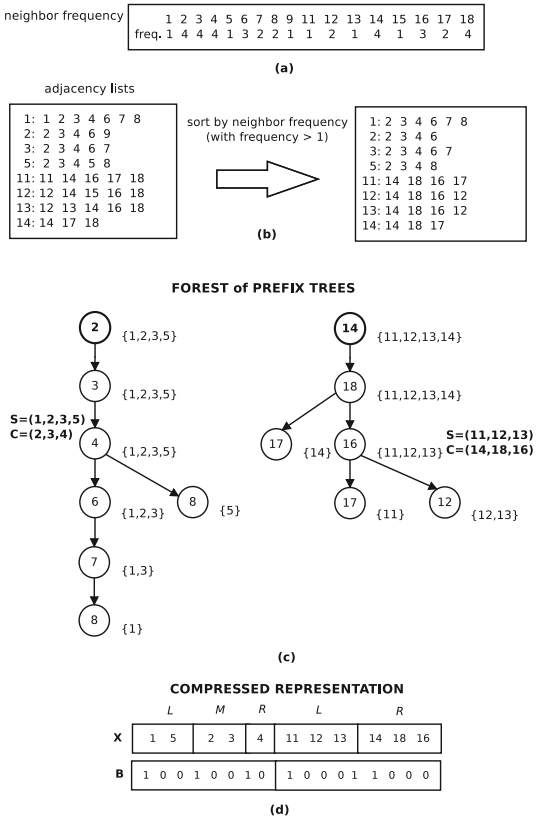

Figure 1. Streaming heuristic example. (a) shows the frequency of the neighbors of the adjacency list showed in (b). (c) shows the forest of prefix trees associated with the adjacency lists sorted by neighbor frequency described in Algorithm 3. (d) shows the compression application which defines the compact representation of the dense subgraphs described in Section VIII.

Algorithm 3 shows the mining algorithm considering a window, w, of edges and a forest of prefix trees.

A. Stream Orders

We consider different stream orders that are good candidates for providing locality [3] and are easy to compute. Specifically, we use the following stream orders:

- URL: URL-based node ordering. In the case of Web graphs, it consists of ordering URLs lexicographically.
- BFS: This node ordering is given by traversing the graph using breadth-first search algorithm starting at a random node.
- CLLP: This node ordering is based on Layered Label Propagation (LLP) clustering described by Boldi et al. [3].
- CHASH: This is not a node ordering in a strict sense, but rather a cluster-based adjacency list ordering. The idea is to group together adjacency lists based on their similarity using the algorithm described in section IV.

Algorithm 3 Streaming algorithm (FT) for listing dense subgraphs.

Input: X:graph stream, w: window of neighbors, $Iters$, $size\_thr$.
Output: Output: $dscol$: Collection of dense subgraphs.

```
1: for (i ← 1 to Iters) do
2:   while (X_i ≠ ∅) do
3:     sortedadjs = readWSortFreq(w, X_i)
4:     Forest FT ← ∅
5:     Tree T
6:     for (adj_u ∈ sortedadjs) do
7:       first ← getFirstElem(adj_u)
8:       T ← FT.find(first)
9:       if (T) then
10:        T.insert(adj_u)
11:      else
12:        T = createT(adj_u)
13:        FT.add(T)
14:      end if
15:    end for
16:    for (T ∈ FT) do
17:      ds ← extractDS(T, size_thr)
18:      dscol.add(ds)
19:    end for
20:  end while
21: end for
```

VII. Experimental Evaluation

We implemented our algorithms in C++. We used a Linux PC with a processor Intel Xeon at 2.4GHz, with 64 GB of RAM and 12 MB of cache. We ran each experiment ten times and considered mean values since the standard deviation was not significant.

We used snapshots of Web graphs and social networks displayed in Table I, which are available by the WebGraph framework project at http://law.dsi.unimi.it. All the algorithms are set to list dense subgraphs with $size\_thr = |S| \cdot |C| \geq 6$ to avoid finding dense subgraphs too small that do not contribute greatly in our compression application. We executed the external memory algorithm described in Algorithm 2, setting M, B, and K so that the sort is done in two passes. We use the URL and CLLP orders provided by the WebGraph project, and compute BFS and CHASH orders. We also consider w of 500, 1000, 2000, and 5000 in the streaming algorithms, since greater values for w did not improve results.

We consider the in-memory algorithm as a reference (shown in Algorithm 1), external memory algorithm (Extmem) (shown in Algorithm 2), the streaming algorithm using only one prefix tree per window (T) and the streaming algorithm considering a forest of prefix trees (FT) described in Algorithm 3.

Dataset	$\|V\|$	$\|E\|$
Eu-2005	862,664	19,235,140
Indochina-2004	7,414,866	194,109,311
Uk-2002	18,520,486	298,113,762
Arabic-2005	22,744,080	639,999,458
Dblp-2011	986,324	6,707,236
LiveJournal-2008	5,363,260	79,023,142

Table I
Main statistics of the Web graphs and social networks.

We examine whether the streaming heuristics are effective for finding and listing dense subgraphs for Web and social graphs. We measure the effectiveness in terms of the number of edges of the graph that participate in dense subgraphs and the required memory and CPU time. We compare streaming algorithms (T) and (FT) using the stream orders presented in section VI-A.

Figure 2 shows how well the streaming algorithms behave in terms of the number of edges participating in dense subgraphs with respect to the time they need to list the dense subgraphs for Web graphs. We observe that there is good locality of reference that allows the FT heuristic to obtain a much better effectiveness that using just one prefix tree (T) per window. We also observe that using a window size $w = 5000$ (right charts) instead of $w = 1000$ allows us to capture more edges in all dense subgraphs using almost the same CPU time. We also observe that the FT algorithm only needs about 5 iterations to capture the dense subgraphs. The best results in terms of edges in dense subgraphs are achieved with CLLP and CHASH orders on Web graphs. Figure 3 shows that in the case of social networks the stream order has more impact than in Web graphs, where the CLLP order exploits locality of reference better than the other stream orders. The figure also shows that social networks need more iterations than Web graphs to list more dense subgraphs.

Second, Table II shows performance ratios in terms of memory usage, total number of edges in dense subgraphs, and CPU time with respect to the in-memory algorithm for 5 iterations on Web graphs and social networks. In addition, Table II shows the Speedup, which is defined as the ratio between the $Edges/secs$ of our streaming algorithm and the $Edges/secs$ of the in-memory algorithm. We compare our results using the FT streaming algorithm over the stream orders URL, BFS, CLLP and CHASH and our external memory algorithm with respect to the in-memory algorithm given in Algorithm 1. *Memory ratio* is the ratio between the amount of memory used by the corresponding algorithm and the in-memory algorithm. *Edge ratio* is the ratio between the number of edges participating in the extracted dense subgraphs obtained by the algorithm and the number of edges achieved using the in-memory algorithm, and *Time ratio* is the ratio between the execution time of the corre-

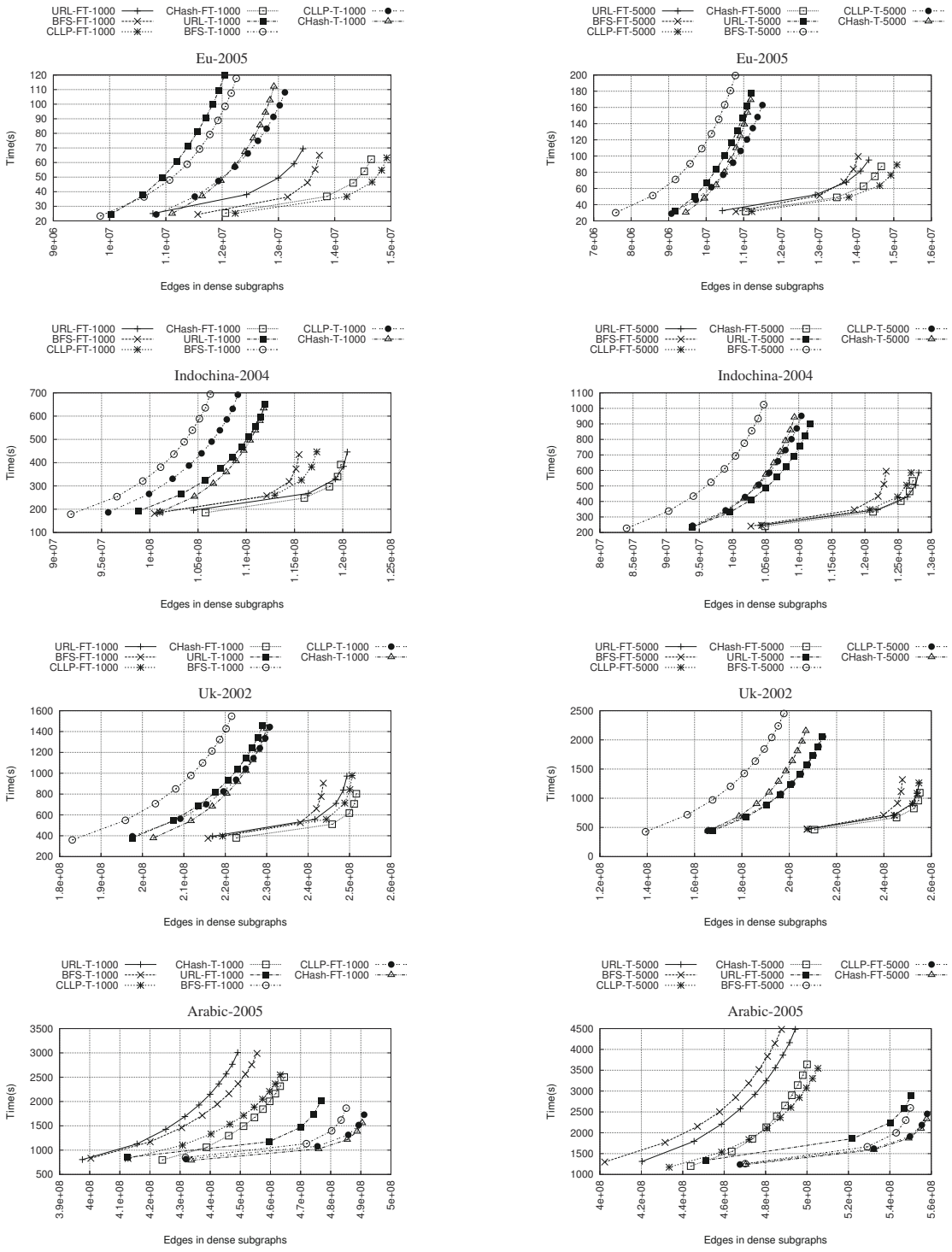

Figure 2. CPU time required for listing edges participating on dense subgraphs in Web Graphs using algorithms T and FT and different stream orders for $w = 1000$ (left charts) and $w = 5000$ (right charts).

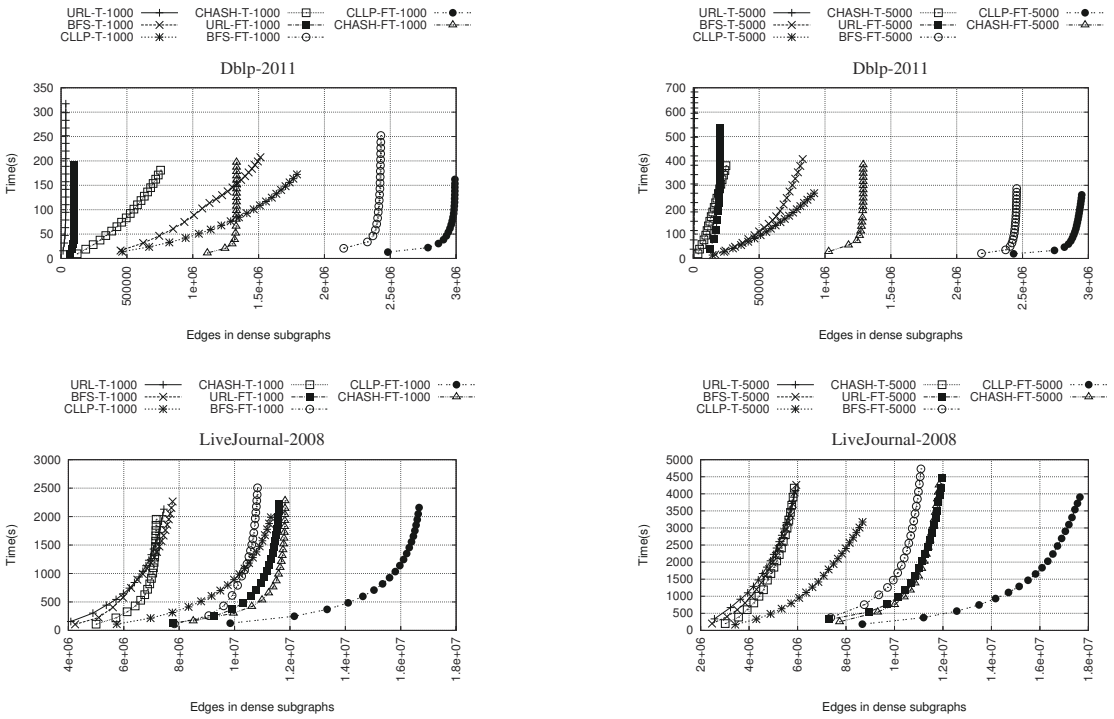

Figure 3. CPU time required for listing edges participating on dense subgraphs in social networks using algorithms T and FT and different stream orders for $w = 1000$ (left charts) and $w = 5000$ (right charts).

sponding algorithm and the execution time of the in-memory algorithm.

We observe in Table II that our external memory algorithm preserves the quality of the in-memory algorithm, using between 53 and 74% of the memory, but doubling running times. Comparing with the in-memory algorithm, we show that for Web graphs, the streaming algorithm (FT) with CLLP order uses between 17 and 25% of the memory between 34 and 65% of the running time, a speedup between 1.21 and 2.15, and achieves between 70 and 96% of the edges in dense subgraphs. However, on social networks the heuristic is less effective being able to retrieve between 40 and 67% of the edges in dense subgraphs, which shows that it pays off to reorder the graph (as in the in-memory algorithm) after each iteration in order to find more subgraphs.

VIII. Application: Compression

The results of listing dense subgraphs can be used for different applications. Here we consider using them for compressing Web graphs and social networks. We represent the collection of dense subgraphs using compact data structures via a symbol sequence and a compressed bitmap. Definitions 5 and 6 describe how dense subgraphs can be used to compress graphs.

Definition 5: Let $G(V, E)$ be a directed graph, and let $H(S_r, C_r)$ be edge-disjoint dense subgraphs of G. Then the corresponding compressed representation of G is $(\mathcal{H}, \mathcal{R})$, where $\mathcal{H} = \{H(S_1, C_1), \ldots, H(S_N, C_N)\}$ and $\mathcal{R} = G - \bigcup H(S_r, C_r)$ is the remaining graph.

Definition 6: Let $\mathcal{H} = \{H_1, \ldots, H_N\}$ be the dense subgraph collection found in the graph. We represent $\mathcal{H}$ as a sequence of integers X with a corresponding bitmap B. Sequence $X = X_1 : X_2 : \ldots : X_N$ represents the sequence of dense subgraphs and bitmap $B = B_1 : B_2 : \ldots B_N$ is used to mark the separation between each subgraph. We now describe how a given X_r and B_r represent the dense subgraph $H_r = H(S_r, C_r)$.

We define X_r and B_r based on the overlapping between the sets S and C. Sequence X_r will have three components: L, Q, and R, written one after the other in this order. Component L lists the elements of $S - C$. Component Q lists the elements of $S \cap C$. Finally, component R lists the elements of $C - S$. Bitmap $B_r = 10^{|L|}10^{|Q|}10^{|R|}$ gives alignment information to determine the limits of the components. In this way, we avoid repeating nodes in the intersection, and have sufficient information to determine all the edges of the dense subgraph. In other words, this

Metric	Order	Datasets					
		Eu-2005	**Indochina-2004**	**Uk-2002**	**Arabic-2005**	**Dblp-2011**	**LiveJournal-2008**
Memory ratio	URL	0.24	0.19	0.19	0.21	0.07	0.12
	BFS	0.21	0.16	0.16	0.19	0.30	0.10
	CLLP	0.25	0.17	0.21	0.24	0.32	0.19
	CHASH	0.26	0.23	0.20	0.26	0.20	0.10
	Extmem	0.59	0.74	0.57	0.53	0.71	0.70
Edge ratio	URL	0.84	0.71	0.96	0.92	0.04	0.30
	BFS	0.82	0.68	0.93	0.92	0.57	0.28
	CLLP	0.88	0.70	0.96	0.94	0.67	0.40
	CHASH	0.86	0.70	0.96	0.94	0.30	0.30
	Extmem	1.00	1.00	1.00	1.00	1.00	1.00
Time ratio	URL	0.69	0.34	0.63	0.70	1.25	0.38
	BFS	0.72	0.35	0.66	0.63	0.49	0.47
	CLLP	0.65	0.34	0.63	0.59	0.46	0.29
	CHASH	0.63	0.31	0.55	0.56	0.72	0.37
	Extmem	2.18	1.82	2.90	2.48	2.40	1.99
Speedup	URL	1.07	2.17	1.48	1.52	0.03	0.79
	BFS	1.01	2.05	1.38	1.70	1.15	0.60
	CLLP	1.21	2.15	1.48	1.83	1.47	1.36
	CHASH	1.20	2.36	1.72	1.92	0.41	0.81

Table II
FT STREAMING HEURISTIC WITH RESPECT TO IN-MEMORY ALGORITHM. $Speedup = \frac{OurAlg.Edges/secs}{InMemoryAlg.Edges/secs}$.

representation allows us to use a sequence X which length is given by $|X| = \sum_r |S_r| + |C_r| - |S_r \cap C_r|$.

Definition 6 describes the compact representation of $\mathcal{H}$ using a symbol sequence and a bitmap. The remaining graph $\mathcal{R}$, in Definition 5, can be compressed using any other compression technique. Here we only show how to compress $\mathcal{H}$ and provide experimental results for that. In order to achieve compression we represent our sequence X and bitmap B using compact data structures. We use Wavelet Trees (WT) [13] using the implementation without pointers [6] for representing the integer sequence X, and compressed bitmaps [20] for the bitmap B. We use the compact data structure, *libcds*, implementation library version 1.0 available at http://www.github.com/fclaude/libcds.

Table III shows some statistics of the listed dense subgraphs using the streaming algorithm FT and CLLP stream order with 5 iterations. We show the distribution of cliques, bicliques and other dense subgraphs with their corresponding average size.

Table IV shows the edge representation in the dense subgraph collection. INMEM shows the percentage of edges captured by the streaming algorithm (FT) using CLLP stream order with respect to the total of edges captured by the in-memory algorithm. RE is the percentage of edges captured by FT with respect to the total number of edges of the graph. Table IV also shows the compression efficiency achieved by the compact representation of the dense subgraphs found using the FT streaming algorithm with 5 iterations. Compression is given by bpe (bits per edge) which corresponds to the space (in bits) of the compact structure of the dense subgraph collection divided by total number of edges in the collection.

Dataset	% CL	size	% BI	size	% DS	size
Eu-2005	4.85	9.95	47.28	22.08	47.85	21.99
Indochina	6.24	5.98	37.61	25.42	56.13	22.94
Uk-2002	3.22	5.31	42.61	20.13	54.16	25.12
Arabic-2005	3.05	5.31	48.06	25.27	48.87	25.83
Dblp-2011	18.27	4.00	11.74	8.30	69.98	6.00
LiveJournal-2008	2.92	3.50	45.44	8.42	51.63	7.95

Table III
PERCENTAGE OF CLIQUES (CL), BICLIQUES (BI), AND THE REST OF DENSE GRAPHS (DS) FOUND, WITH CORRESPONDING AVERAGE SIZE.

Data Set	INMEM (%)	RE (%)	bpe
Eu-2005	88.1	81.0	1.68
Indochina	70.2	66.0	1.26
Uk-2002	96.0	87.2	1.60
Arabic-2005	94.0	88.4	1.27
Dblp-2011	67.6	43.1	5.64
LiveJournal-2008	40.3	18.2	7.95

Table IV
EDGE REPRESENTATION AND COMPRESSION EFFICIENCY IN $\mathcal{H}$.

IX. CONCLUSIONS

This paper proposes an external memory algorithm for finding and listing dense subgraphs in Web and social

graphs. The algorithm preserves the quality of the in-memory algorithm, it reduces memory usage, but double the CPU time. We also present a streaming mining heuristic that takes advantage of the similarity and locality of reference that provide some graph stream orders. We provide experimental evaluation that shows that on Web graphs, in comparison with the in-memory algorithm, the streaming mining heuristic FT using CLLP stream order is able to find between 70 and 96% of edges participating in dense subgraphs, uses only between 17 and 25% of the memory, CPU times are between 34 and 65%, and provides a speedup on $Edges/secs$ between 1.21 and 2.15. However, the results show that on social graphs the streaming algorithm is less effective as we are able to find between 40 and 67% of the edges, using between 19 and 32% of memory, CPU times between 29 and 46%, and a speedup between 1.36 and 1.47. Furthermore, we show an effective way of compressing listed dense subgraphs using compact data structures.

REFERENCES

[1] C. C. Aggarwal, Y. Li, P.S. Yu, R. Jin. *On Dense pattern mining in graph streams*, PVLDB, 2010, 3(1), pp. 975–984.

[2] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. *On the streaming model augmented with a sorting primitive*, FOCS, 2004, pp. 540–549.

[3] P. Boldi and M. Rosa, M. Santini, S. Vigna. *Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks*, WWW, 2011, pp.587–596.

[4] A.Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. *Min-Wise independent permutations*, J. Comput. Syst. Sci., 60(3), 2000, pp. 630–659.

[5] G. Buehrer and K. Chellapilla. *A scalable pattern mining approach to Web graph compression with communities*, WSDM, 2008, pp. 95–106.

[6] F.Claude and G. Navarro. *Practical rank/select queries over arbitrary sequences*, SPIRE, 2008, pp. 176–187.

[7] J. Cheng, L. Zhu, Y. Ke, and S. Chu. *Fast algorithms for maximal clique enumeration with limited memory*, SIGKDD, 2012, pp. 1240–1248.

[8] C. Demetrescu, and I. Finocchi, and A. Ribichini. *Trading off space for passes in graph streaming problems*, ACM Transactions on Algorithms, 6(1), 2009.

[9] D. Donato, S. Leonardi, S. Millozzi, and P. Tsaparas. *Mining the inner structure of the Web graph*, WebDB, 2005, pp. 145–150.

[10] J. Feigenbaum and S. Kannan and A. McGregor and S. Suri and J. Zhang. *On graph problems in a semi-streaming model*, J. Theor. Comput. Sci., 348(2-3), pp. 207–216, 2005.

[11] H. Garcia-Molina, J. Ullman, and J. Widom. *Database systems - the complete book*, Prentice Hall Press, Upper Saddle River, NJ, USA, 1 edition, 2002.

[12] D. Gibson, R. Kumar, and A. Tomkins. *Discovering large dense subgraphs in massive graphs*, VLDB, 2005, pp. 721–732.

[13] R. Grossi, A. Gupta, and J.S. Vitter. *High-order entropy-compressed text indexes*, SODA, 2003, pp. 841–850.

[14] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. *Trawling the Web for emerging cyber-communities*, Computer Networks, 1999, 31(11-16), pp. 1481–1491.

[15] C. Hernández, and G. Navarro. *Compressed Representation of Web and Social Networks via Dense Subgraphs*, SPIRE, 2012, pp. 264–276.

[16] C. Hernández, and G. Navarro. *Compressed representations for Web and social graphs*, Knowl. Inf. Syst. 40(2), 2014, pp. 279–313.

[17] I. Katriel, and U. Meyer. *Elementary graph algorithms in external memory*, Algorithms for Memory Hierarchies, 2002, pp. 62–84.

[18] I. Munro, and M. Paterson. *Selection and sorting with limited storage*, Theor. Comput. Sci., 1980, (12), pp. 315–323.

[19] S. Muthukrishnan. *Data Streams: Algorithms and applications*, Foundations and Trends in Theoretical Computer Science, (2), 2005.

[20] R. Raman, V. Raman, S.S. Rao. *Succinct indexable dictionaries with applications to encoding k-ary trees and multisets*, SODA, 2002, pp. 233–242.

[21] M. Ruhl, *Efficient algorithms for new computational models*, PhD. Thesis, MIT, 2001.

[22] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. *A large-scale study of Link spam detection by graph algorithms*, AIRWEB, 2007.

[23] R. Samudrala and J. Moult. *A graph-theoretic algorithm for comparative modeling of protein structure*, Journal of Molecular Biology, 279(1), 1998, pp. 287–302.

[24] A. E. Sariyüce, B. Gedik, G. Jacques-Silva, K. Wu, and Ü. V. Çatalyürek. *Streaming algorithms for k-core decomposition*, PVLDB, 2013, 6(6), pp. 433–444.

[25] I. Stanton, and G. Kliot. *Streaming graph partitioning for large distributed graphs*, SIGKDD, 2012, pp. 1222–1230.

[26] J.C Vitter. *External memory algorithms and data structures*, ACM Comput. Surv., 2(33), 2001, pp. 209–271.

[27] J. Zhang. *A survey on streaming algorithms for massive graphs*, Advances in Database Systems, Springer US, pp. 393–420.

[28] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.