Analysing Big Social Graphs Using a List-based Graph Folding Algorithm

Eugenia Muntyan[1], Nikolay Sergeev[1], and Alexey Tselykh[2,a]

[1]*Southern Federal University, Department of Computer Engineering, Taganrog, Russia*
[2]*Southern Federal University, Department of Information Security Systems, Taganrog, Russia*

Abstract. In this paper, we explore the ways to represent big social graphs using adjacency lists and edge lists. Furthermore, we describe a list-based algorithm for graph folding that makes possible to analyze conditionally infinite social graphs on resource constrained mobile devices. The steps of the algorithm are (a) to partition, in a certain way, the graph into clusters of different levels, (b) to represent each cluster of the graph as an edge list, and (c) to absorb the current cluster by the cluster of the next level. The proposed algorithm is illustrated by the example of a sparse social graph.

1 Introduction

Currently, there is a growing interest in modeling and analyzing interrelationships between professional, religious, ethnic, and other types of social groups [1-4]. The knowledge underlying these models comprises both the knowledge of personality psychology and social psychology of interpersonal relationships as well as anthropology, sociology, theology, ethnography, conflict theory, and history.

The basic unit of modeling here is either an actor (human person or group) or, in Actor-Network Theory (ANT) [5], an output of actor's activity – a material artefact. Both are collectively called actants.

In terms of graph-hypergraph paradigm [6], an entity (actor or actant) can be represented by a vertex in a graph or by a hyperedge in a hypergraph. In second case, a set of vertices forming a hyperedge can be associated with certain characteristics of an entity: situation awareness, evaluation of event, communication or individual activities.

The two most common representations of a social graph are an adjacency matrix (sociomatrix) and an incidence matrix [7]. Both are memory-intensive, especially for sparse high-dimensional graphs, when most of the elements of the matrix have 0 value.

In this paper, we use a list data structure [8, 9] to represent and analyze sparse social graphs. Furthermore, we describe a list-based algorithm for graph folding that eventually makes possible to analyze conditionally infinite social graphs on resource constrained mobile devices.

Using a list data structure to analyze social graphs was previously considered in [10-13].

2 Research methodology

2.1 List representations of a graph

There are two ways to represent a graph as a list. Adjacency list describes the set of neighbors of a vertex in the graph; its main data structure is an array of linked lists, one linked list for each vertex. Edge list is a list of node pairs (edges).

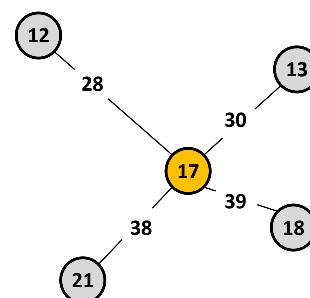

Figure 1. Undirected graph

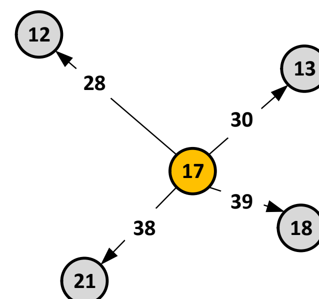

Figure 2. Directed graph

[a] Corresponding author: tselykh@sfedu.ru

© The Authors, published by EDP Sciences. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (http://creativecommons.org/licenses/by/4.0/).

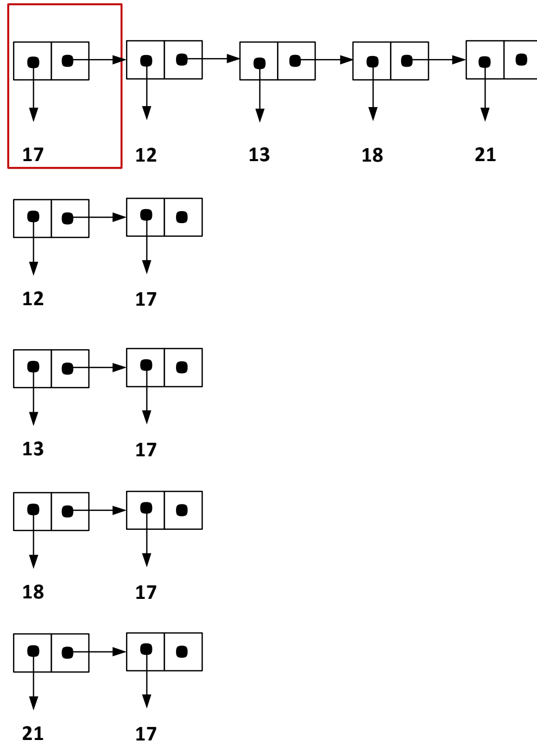

Figure 3. Adjacency list

As an example, consider an undirected graph G_U in Fig. 1 and an appropriate directed graph G_D in Fig. 2.

The adjacency list of a graph is an *n*-element array of linked lists, one for each vertex, where the i_{th} element points to a linked list of edges for vertex *i*. This linked list represents the edges by the vertices adjacent to vertex *i*. Every element (node) of the list consists of two fields. The left field is a pointer to the vertex index, e.g. 17, while the right field is a pointer to the next node as shown in Fig 3.

Adjacency list of the graph G_U is X_c = (17, 12, 13, 18, 21), (12, 17), (13, 17), (18, 17), (21, 17) as shown in Fig. 3.

Adjacency list for an appropriate directed graph G_D is X_c = (17, 12, 13, 18, 21).

The edge list of the graph is a list, or array, of edges. When representing undirected graph with an edge list, all the edges are repeated in reverse order.

Edge list for graph in Fig. 1 is X_p = (17, 12), (17, 13), (17, 18), (17, 21), (12, 17), (13, 17), (18, 17), (21, 17) as shown in Fig. 4. In the above example, the graph has 4 edges, so the edge list consists of 8 pairs.

Edge list for an appropriate directed graph in Fig. 2 is X_p = (17, 12), (17, 13), (17, 18), (17, 21). Graphical representation of X_p corresponds to the left part of the edge list in Fig. 4.

2.2 Operations on the lists

In this section, we consider split and join operations on the lists. Appropriate functions were described in [8].

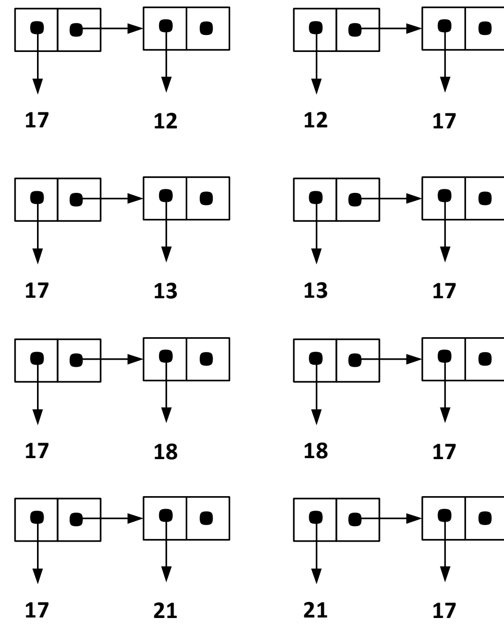

Figure 4. Edge list

To split the list, we use functions *hd* (*x*) and *tl* (*x*). These functions get the head (the first element) and tail (all but the first element) of a list, respectively.

Given the list *x* = (17, 12, 13, 18, 21), the following values are returned: *hd* (*x*) = (17) and *tl* (*x*) = (12, 13, 18, 21). If we take *tl* (*x*) as a new list x_1 = (12, 13, 18, 21), then *hd* (x_1) = (12), and *tl* (x_1) = (13, 18, 21). Similarly, we construct lists x_2 and x_3. Summarizing the example, list *x* can be represented as follows:

hd (*x*), *hd* (*tl* (*x*)), *hd* (*tl* (*tl* (*x*))), *hd* (*tl* (*tl* (*tl* (*x*)))), …

To join lists, we use function *cons*. Given *a* – an element or a list and *b* – a list, the function *cons* (*a, b*) is a new node in the list.

In our case, if *a* = 17 and *b* = (12, 13, 18, 21), then the following value is returned: *cons* (*a, b*) = (17, 12, 13, 18, 21).

List *z* = (z_1, z_2, z_3, z_4, …, z_n) is constructed using a formula

cons (z_1, *cons* (z_2, *cons* (z_3, *cons* (z_4, …, *cons* (z_n) …)))).

We can make a conclusion that operators *hd (x)* and *tl (x)* are inversely related to *cons*: *hd* (*cons* (*a, b*)) = *a*, *tl* (*cons* (*a, b*)) = *b*. The reverse rule is not always true. *cons* (*hd* (*x*), *tl* (*x*)) returns a new element, that contains the same pointers as the source element *x*. Function *cons* (*hd* (*x*), *tl* (*x*)) returns a copy of the cell *x*, but not the cell *x* itself.

The concept of a doubly linked list is given in [9]. It can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders. Each node of a doubly linked list contains three fields: an integer value (*key*), the link to the next node (*next*), and the link to the previous node (*prev*) as shown in Fig. 5.

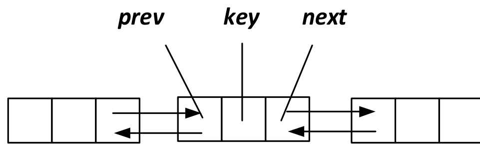

Figure 5. The structure of an element of a doubly linked list

Consider a doubly linked list $x = (x_1, x_2, x_3, x_4, \ldots, x_n)$. Given a node x_i, *next* (x_i) refers to the next node, and *prev* (x_i) refers to the previous node. If *prev* (x_i) = NIL, then x_i is the entry point called the head of the list *hd* (x). If *next* (x_i) = NIL, then x_i is the last node in the list called the tail of the list *tl* (x).

There are 2-element and 5-element lists among the adjacency lists considered earlier. The 5-element list x = (17, 12, 13, 18, 21) implies that we can get from node 17 to nodes 12, 13, 18 and 21. Meanwhile, there's no other way to get from node 12 to node 13, but through the node 17.

Eventually we end up with the edge lists discussed earlier. From this we can conclude that it is reasonable to represent graphs as lists of edges.

3 Graph folding algorithm

Let G_1 and G_2 be graphs and $f\colon G_1 \rightarrow G_2$ be a continuous function. Then f is called a graph map, if

(i) For each vertex $v \in V(G_1)$, $f(v)$ is a vertex in $V(G_2)$.

(ii) For each edge $e \in E(G_1)$, $\dim(f(e)) \leqslant \dim(e)$.

A graph map $f\colon G_1 \rightarrow G_2$ is called a graph folding if f maps vertices to vertices and edges to edges, i.e., for each vertex $v \in V(G_1)$, $f(v)$ is a vertex in $V(G_2)$ and for each edge $e \in E(G_1)$, $f(e)$is an edge in $E(G_2)$. [14]

In this section, we describe a list-based algorithm for graph folding in a sequence of steps (1) to (8).

First, we introduce the following notations:

Nodes = {1…(*n*-1)} – the vertex set of a social graph;

$Node_j$ – the j_{th} vertex in the graph;

cNode – the source vertex in the graph;

$Cluster_i$ – the cluster at the i_{th} layer;

Neighbors (*cNode*) – the set of neighboring (adjacent) vertices for *cNode*;

C – the number of neighboring vertices.

The basic steps of the algorithm are:

1) Define the level of the cluster, *i*=0;
2) Define the source vertex (*cNode* = $Node_j$);
3) Perform the search function to discover neighboring vertices for *cNode*, *C* = *get Neighbors* (*cNode*);
4) If $C \neq 0$, then $i = i + 1$;
5) Define the cluster of the i_{th} level as an aggregated set of source vertex *cNode* and its neighbors *Neighbors* (*cNode*), herewith the cluster is a list $Cluster_i$ = *cons* (*cNode*, *Neighbors* (*cNode*)), where *hd* ($Cluster_i$) = cNode, *tl* ($Cluster_i$) = *Neighbors* (*cNode*);
6) Fold the vertices of the cluster at the i_{th} level into a new source vertex *cNode*, *cNode* = $Cluster_i$;
7) return to step (4);
8) if *C*=0 then terminate.

We explain algorithm using the example in Fig. 6. The initial graph has *n* vertices numbered from 1 to 42. They are connected by edges numbered from 1 to 73. The size of incidence matrix is 42x73. The size of adjacency matrix is 42x42. Both matrices are sparse.

We designate vertex 17 as a source vertex *cNode*. Vertex 17 has neighboring vertices 12, 13, 18, and 21.

According to algorithm, the cluster of the i_{th} level $Cluster_i$ (*i*=1) contains a source vertex *cNode* = 17 and its *Neighbors* (*cNode*) = 12, 13, 18, and 21. They can be represented with an edge list $Cluster_{1\text{-}4}$ = (17, 12), (17, 13), (17, 18), (17, 21). Using an algorithm, we fold the cluster at the 1st level into a new vertex and assign it the next available index *cNode* = 43.

For source vertex *cNode* = 43 in the cluster of the 2nd level, *Neighbors* (*cNode*) = 3, 9, 7, 10, 19, 27, 16. The edge list is $Cluster_{2\text{-}8}$ = (43, 3), (43, 7), (43, 9), (43, 10), (43, 19), (43, 16), (43, 27), (9, 10).

We emphasize the fact that some merging vertices are connected by multiple edges, e.g. in the list (43, 9) the vertices are connected by three edges: 16, 17 and 18.

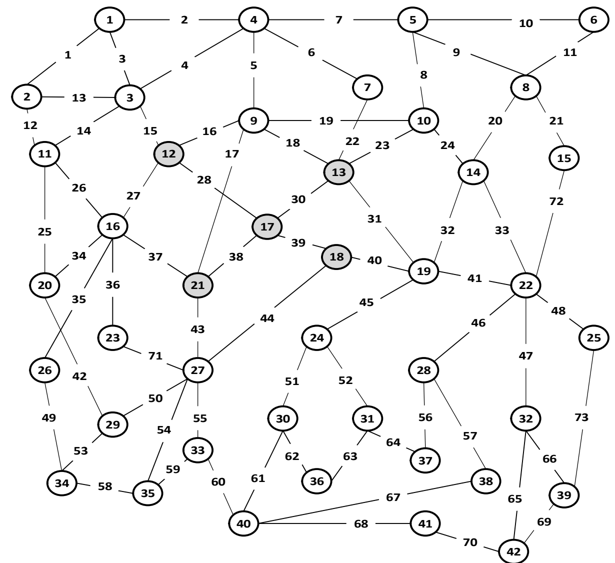

Figure 6. Graph with a highlighted cluster at the 1st level

We note the case when two or more merging vertices are linked not only to the source vertex but are interconnected to each other. For appropriate mapping of merged vertices, we add list (9, 10).

We proceed with designating *cNode* = 44 as a source vertex with *Neighbors* (*cNode*) = 1, 2, 4, 5, 11, 14, 20, 22, 23, 24, 26, 29, 33, and 35.

Edge lists with merging vertices in the cluster of the 3rd level are produced accordingly: $Cluster_{3\text{-}22}$ = (44, 1), (44, 2), (44, 4), (44, 5), (44, 11), (44, 14), (44, 20), (44, 22), (44, 23), (44, 24), (44, 26), (44, 29), (44, 33), (44, 35), (2, 11), (2, 1), (1,4), (4,5), (14, 22), (11, 20), (20, 29), (33, 35).

In Fig. 7, it is shown an example of merging vertices in the cluster of the 4th level into a new vertex *cNode* = 45. *Neighbors* (*cNode*) = 6, 8, 15, 25, 32, 28, 31, 30, 40, 34. $Cluster_{4\text{-}13}$ = (45, 6), (45, 8), (45, 15), (45, 25), (45, (45, 30), (45, 31), (45, 32), (45, 34), (45, 40), (8, 6), (8, 28), 15), (40, 30).

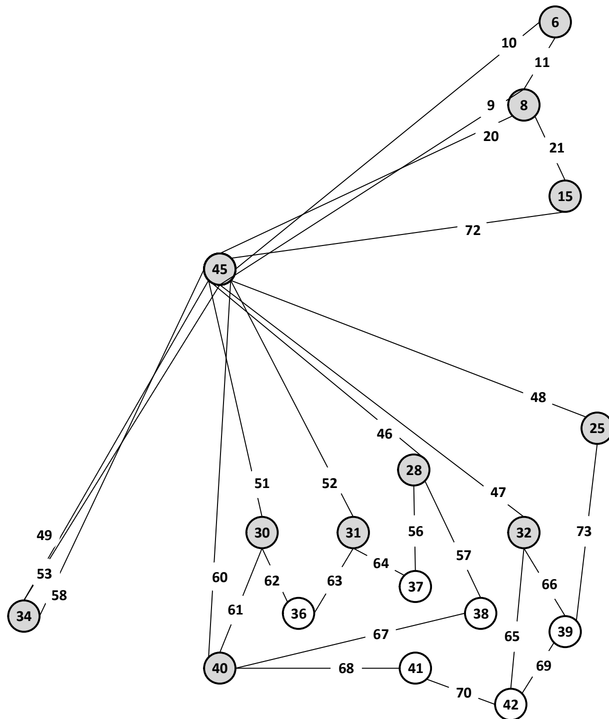

Figure 7. Folded graph with a highlighted cluster at the 4th level

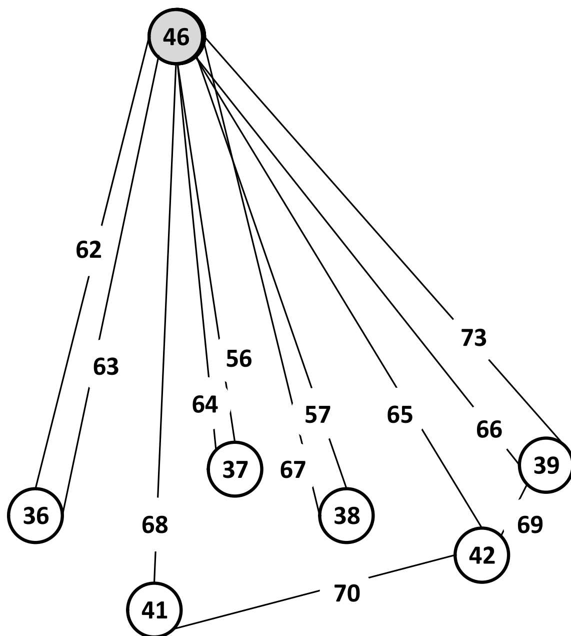

Figure 8. Folded graph with a highlighted cluster at the 5th level

The resulting graph contains vertex 46 and its *Neghbors* (*cNode*) = 36, 41, 37, 38, 42, 39 (Fig. 8). The edge lists for the cluster of the 5th level is the following: $Cluster_{5\text{-}8}$ = (46, 36), (46, 37), (46, 39), (46, 38), (46, 41), (46, 42), (42, 41), (42, 39).

4 Conclusion

In this paper, we described a list-based algorithm for graph folding. According to the results of experiments, by reducing calculation redundancy, the above algorithm is 30% less memory-intensive and 20-30% faster, depending on the density of the graph, in comparison to algorithms using adjacency/incidence matrices and iteration over vertices. In our case it was the only possible algorithm for analyzing conditionally infinite graphs on resource constrained mobile devices.

5 Acknowledgements

This work was supported by the grant of Southern Federal University, research project No. 07/2017-28 and by the grant of Russian Foundation for Basic Research, project No. 17-01-00243.

References

1. R.A. Grier, B. Skarin, A. Lubyansky and L. Wolpert, *In 2nd Intl. Conf. on Computational Cultural Dynamics* (2008).
2. C.C. James and Ö. ÖZdamar, Foreign Policy Analysis, **5**, 17-36 (2009).
3. J. Kopecky, N. Bos and A. Greenberg, *In 19th Annual Conf. on Behavior Representation in Modeling and Simulation* (136-143, 2010).
4. G.M. Levchuk, F. Yu, T. Haiying, K.R. Pattipati, Y. Levchuk and E. Entin, *In 12th Intl. Command and Control Research and Technology Symposium* (2007).
5. B. Latour, *Reassembling the Social: An Introduction to Actor-Network-Theory* (Oxford University Press, 2007).
6. N. Sergeev, A.A. Tselykh and A.N. Tselykh, *In 4th Intl. Conf. on Security of Information and Networks* (2013).
7. S. Wasserman and K. Faust, *Social Network Analysis: Methods and Applications* (Cambridge University Press, 1994).
8. J.M. Foster, *List Processing* (Macdonald, London, 1968).
9. T.H. Cormen, C.E. Leiserson, R.L. Rivest and C. Stein, *Introduction to Algorithms* (MIT Press, 2009).
10. F. Hildorsson, *Scalable Solutions for Social Network Analysis* (Diva portal, 2009).
11. C. Wickramaarachchi, A. Kumbhare, M. Frincu, C. Chelmis, and V.K. Prasanna, *In IEEE/ACM 15th Intl. Symposium on Cluster, Cloud, and Grid Computing* (829-834, 2015).
12. F. Claude and G. Navarro. *In Algorithms and Applications* (77-91, Springer-Verlag, 2010)
13. P. Liakos, K. Papakonstantinopoulou and A. Delis, *In ACM 25th Intl. Conf. Inf. Knowl. Manag.* (2317-2322, 2016).
14. E.M. El-Kholy and H.A. Mohamed, Journal of Applied and Computational Mathematics, **7**(1) (2017).