Math. Maced.
Vol. 8 (2010)
95-106

PARALLEL ALGORITHMS FOR TRANSITIVE REDUCTION FOR WEIGHTED GRAPHS

DRAGAN BOŠNAČKI, WILLEM LIGTENBERG, MAXIMILIAN ODENBRETT*, ANTON WIJS*, AND PETER HILBERS

Dedicated to Academician Ǵorǵi Čupona

Abstract. We present a generalization of transitive reduction for weighted graphs and give scalable polynomial algorithms for computing it based on the Floyd-Warshall algorithm for finding shortest paths in graphs. We also show how the algorithms can be optimized for memory efficiency and effectively parallelized to improve the run time. As a consequence, the algorithms can be tuned for modern general purpose graphics processors. Our prototype implementations exhibit significant speedups of more than one order of magnitude compared to their sequential counterparts.

Transitive reduction for weighted graphs was instigated by problems in reconstruction of genetic networks. The first experiments in that domain show also encouraging results both regarding run time and the quality of the reconstruction.

1. Introduction

[0]

The concept of transitive reduction for graphs was introduced in [1] and a similar concept was given previously in [8]. Transitive reduction is in a sense the opposite of transitive closure of a graph. In transitive closure a direct edge is added between two nodes i and j, if an indirect path, i.e., not including edge (i, j), exists between i and j. In contrast, the main intuition behind transitive reduction is that edges between nodes are removed if there are also indirect paths between i and j.

In this paper we present an extension of the notion of transitive reduction to weighted graphs. The idea was triggered by applications in bioinformatics, in particular, in reconstruction of genetic networks from perturbation experiments. In this kind of experiments usually one gene is disabled and the effects on the other genes are monitored. Based on the data, interactions between genes can be deduced.

The problem of incorrectly inferring non-exisiting direct interactions between genes is inherent to such experiments, as illustrated by the following example:

2000 *Mathematics Subject Classification.* 68R10;05C85.

Key words and phrases. transitive reduction, weighted graphs, algorithms.

[0*] Supported by the Netherlands Organisation for Scientific Research (NWO) project 612.063.816 *Efficient Multi-Core Model Checking*

Suppose that gene a directly activates gene b and gene b activates gene c, but there is no direct influence between gene a and gene c. Also, let us assume that genes b and c cannot be activated by any other gene factor. Now, if we disable gene a, for instance by deleting it from the genome, this would deactivate gene b, but also indirectly gene c. Thus, our measurement of the expression of genes a, b, and c will show that a influences both b and c. However, since we are usually interested only in *direct* influences, the influence of a on c *transitively* via b is obsolete and it can be considered as a side effect of the measurement method. Thus, it would be useful if we could systematically remove such spurious indirect interactions from the reconstructed network. This translates to the aforementioned concept of transitive reduction. Assuming that the nodes of the network are genes and the edges are interactions, the rule is to remove a direct interaction between each gene pair g and g', if there is an alternative chain of interactions between g and g'.

Using *transitive reduction* for filtering out indirect influences was first proposed in [10]. There are several other relevant publications on this subject [6, 5]. In all previous works on transitive reduction the networks were represented as directed graphs without weights on the edges. However, in the network inference algorithms interaction strengths between genes usually play an important role. Motivated by this fact, we work with the concept of transitive reduction of weighted graphs, where the weights correspond to interaction strengths (influences).

In this paper we present two algorithms for transitive reduction of weighted graphs. The algorithms are based on the Floyd-Warshall algorithm for finding paths between all pairs of nodes in a graph. As a consequence, the algorithms are of polynomial complexity and can be efficiently parallelized. Moreover, we also show how memory requirements of the algorithms can be lowered. These algorithms are developed for general purpose graphics processing units (GPUs). GPUs have been extensively used outside the computer graphics area for various applications, including bioinformatics and systems biology. Since GPUs are standard in modern computers, parallel algorithms become an attractive possibility to speed up computations. In that sense our algorithms are already beyond the state of the art regarding the biological applications for gene network reconstruction. With the current version one can tackle full genome (all genes of the organism disabled one by one) perturbation experiments which will be important in the future.

With regard to the definition of transitive closure for weighted graphs and the general theoretical background probably the closest to our work is [3]. Their paper is also motivated by a biological application, in particular, analysis of protein-protein interaction networks. The authors define the notion of transitive closure of weighted graphs, but stop short of introducing transitive reduction. We reuse some of their ideas about transitive affinity to define transitive influence on a path in the network. We extend their work by introducing the restricted transitive influence and showing its properties which were necessary for the development and correctness proof of the algorithm.

The only work that we could find in the literature that deals explicitly with parallel algorithms for transitive reduction of unweighted graphs is [2]. Although

parallel versions of the Floyd-Worshall algorithm [4, 11] are well known, to the best of our knowledge, besides ours, there does not exist an implementation of a transitive reduction algorithm on GPUs.

After the work from this paper was presented on the conference to which this special issue is dedicated, we learned about a very similar concept of weighted transitive reduction that was introduced in [7] and which was published almost simultaneously. The main difference with our definition of transitive reduction is that they use signs in the interactions. As a consequence, in presence of negative cycles, the algorithm becomes NP-complete, which might lead to poor scalability. Our experiments indicate that our approach without signs produces the same quality of results with the advantage of seamless scalability because of the polynomial complexity and the parallelization of our algorithms.

In [9] transitive reduction for weighted graphs is briefly discussed together with several other methods for discrimination between direct and indirect interactions in networks. Similarly to our approach, they use a Floyd-Warshall based algorithm for pruning the obsolete interactions. Compared to our approach, the authors introduce quite different method to extend the notion of edge weights to paths in the graph based on probabilities. This method takes also the signs of the interactions into account.

Paper layout. In the next section we present the basic definitions and properties that are used in the paper. The definition of transitive reduction is given in Section 3. Section 4 deals with the algorithms. The last section concludes with some directions for future work.

2. Preliminaries

2.1. **Graphs, Transitive Closure, and Transitive Reduction.** A *directed graph* $G = (V, E)$ is a pair of sets of nodes V and edges $E \subseteq V \times V$. Without loss of generality, in what follows we identify the set of nodes V with the set of numbers $\underline{n} = \{1, 2, \ldots, n\}$, where n is the number of nodes in V. The graph can be equivalently represented by an *adjacency matrix* A. For unweighted graphs, the elements a_{ij} of the matrix have value 1, if there is an edge between nodes i and j, and 0, otherwise. A *path* from node i to node j is a sequence $P_{ij} = (k_0, k_1, k_2 \ldots k_{m-1}, k_m)$, where $k_0 = i$, $k_m = j$ and $(k_{l-1}, k_l) \in E$, for $0 < l \leq m$. Nodes k_l, where $1 \leq l \leq m-1$ are called *intermediate* nodes. We denote the set of edges of P_{ij} with $Edges(P_{ij})$. A *cycle* is a path P_{ij} for which the first and the last node coincide, i.e., $i = j$. A cycle that consists of only one edge is called a *self loop*. A graph is *acyclic* if it does not contain cycles. The set of all paths in the graph G is denoted as Π. Similarly, $\Pi_{ij} \subseteq \Pi$ is the set of all paths between i and j.

The *transitive closure* of a graph G is the graph $G^T = (V, E^T)$ with $(i, j) \in E^T$ if and only if there exists a path from i to j in G. The *transitive reduction* of an acyclic graph G is the unique [1] smallest (i.e., with the least number of edges) graph $G^t = (V, E^t)$ such that $(G^t)^T = G^T$.

For an acyclic graph G it can also be shown [1] that the transitive reduction G^t can be obtained by removing each edge $(i, j) \in E$ from the original graph G for which there is an indirect path, i.e., not including (i, j), between i and j in G.

The definition of transitive reduction can be extended in a natural way to graphs with cycles. However, the reduced graphs are not unique and in general cannot be generated by deleting edges from the graph. To solve this, in [1] the strongly connected components of the graph are shrunk to single nodes and transitive reduction is applied on the resulting acyclic graph.

In genetic networks sometimes both direct and indirect influences can exist at the same time. Unfortunately, in such cases, the transitive reduction as defined above, will still remove the direct influence. To avoid this anomaly, we introduce the notion of transitive reduction on weighted graphs where the weights correspond to the strengths of influences between the nodes. Knowing the influence strengths allows us to refine the edge removal criterion compared to the unweighted case: an edge (i, j) is removed from the original graph only if there exists a stronger indirect influence of i to j along some path P_{ij}. In other words, if an edge is at least as strong as all indirect influences, then it is kept in the graph.

To formally capture the above concepts, we associate with the graph a function $w : E \to \mathbb{R}$ which maps each edge to a real number. The adjacency matrix A is replaced by a matrix of weights W in which each element $w_{ij} = w(i, j)$ gives the direct influence between nodes i and j. A special value denotes the absence of a direct influence/edge between two nodes. In our case this value is 0. In what follows we use interchangeably w_{ij} and $w(i, j)$ to denote edge weights.

2.2. **Transitive Influence.** Since the above mentioned informal criterion of edge removal operates with influence strengths along paths, we use *transitive influence* [3] as a measure of the influence strength along the path. As shown in [3] in many applications – including genetic networks – a natural requirement is that the transitive influence is associative. In the sequel we work with the transitive influence $T_{\min}$ which is determined by the minimal edge influence of all edges in the path.

Definition 1. *Given a path P_{ij}, its* transitive influence $T_{\min}$ *is defined as*

$$T_{\min}(P_{ij}) = \min_{(k,l) \in Edges(P_{ij})} \{w_{kl}\}.$$

Intuitively, T_{min} expresses the weakest link principle which states that the influence along the path is as strong as its weakest direct influence. Later we discuss also some alternative influences on which the algorithms of this paper can also be applied. In what follows we omit the subscript min whenever it is clear from the context that we use $T_{\min}$.

In general, there can exist many paths between two given nodes i and j. Since we are interested in the strongest influence between i and j this leads to the following definition:

Definition 2. *For two fixed nodes i and j in $\underline{n}$ we define* maximal transitive influence *between i and j as*

$$h_{ij} = \max_{P_{ij} \in \Pi_{ij}} T(P_{ij}),$$

if there exists a path P_{ij} between i and j. Otherwise, $h_{ij} = 0$

Assuming the weakest link influence $T_{\min}$ the definition becomes

$$h_{ij} = \max_{P_{ij} \in \Pi_{ij}} \left(\min_{(k,l) \in Edges(P_{ij})} \{w_{kl}\} \right).$$

For a given graph G the maximal transitive influence between any pair of nodes is uniquely defined. It follows directly from Def. 2 that for all $i, j \in \underline{n}$

$$h_{ij} \geq w_{ij}. \tag{2.1}$$

since $P_{ij} = (i, j) \in \Pi_{ij}$.
For the design of the algorithms that we introduce in the sequel, as well as for showing their correctness, we need some more definitions and properties related to transitive influence. We aim at iteratively computing the maximal transitive influence by systematically enlarging the set of considered paths, until we have included all paths and have found the optimal one.

In this context, for fixed nodes i, j, k, we consider paths which pass only through a given subset of V. We define $\Pi_{ij}^k \subseteq \Pi_{ij}$ as the set that consists of all paths P_{ij} for which it holds: if l is an intermediate node of P_{ij}, then $l \leq k$. All edges (i, j) are also in Π_{ij}^k, since as direct paths they do not contain any intermediate nodes. The paths in Π_{ij}^k can begin and/or end in nodes greater than k, however all intermediate nodes are less than or equal to k. Now we can introduce *k-restricted transitive influence h^k* which is defined only on paths in Π_{ij}^k:

Definition 3. *For all $i, j \in \underline{n}$ and $k > 0$* k-restricted transitive influence *h_{ij}^k is defined as*

$$h_{ij}^k = \max_{P_{ij} \in \Pi_{ij}^k} T(P_{ij})$$

It is convenient to define also h_{ij}^0. Then we can formulate the following relation which makes it possible to recursively/iteratively compute h_{ij}^k and it is a basis for the Floyd-Warshall like algorithm that we present later.

Proposition 2.1. *Let $h_{ij}^0 = w_{ij}$, for all $(i, j) \in \underline{n} \times \underline{n}$. Then, for all $i, j, k \in \underline{n}$*

$$h_{ij}^k = \max(h_{ij}^{k-1}, \min(h_{ik}^{k-1}, h_{kj}^{k-1})).$$

Proof. We observe that we can reuse the computation of h^{k-1} in the sense that to find the optimal path for h_{ij}^k we have to consider only some extra paths which are in Π_{ij}^k, but not in Π_{ij}^{k-1}. To this end, using also properties of the max function, we rewrite the equation which defines h_{ij}^k in Def. 3 as:

$$h_{ij}^k = \max_{P_{ij} \in \Pi_{ij}^k} T(P_{ij}) = \max(\max_{P_{ij} \in \Pi_{ij}^{k-1}} T(P_{ij}), \max_{P_{ij} \in \Pi_{ij}^k \setminus \Pi_{ij}^{k-1}} T(P_{ij})) \tag{2.2}$$

Term $\max_{P_{ij} \in \Pi_{ij}^{k-1}} T(P_{ij})$ is computed taking into account only the paths which contain intermediate nodes $l \leq k-1$ and it is actually the definition of h_{ij}^{k-1}. Hence, we need to show that the second argument of the outer max function equals $\min(h_{ik}^{k-1}, h_{kj}^{k-1})$. As emphasized above, the set of paths $\Pi_{ij}^k \setminus \Pi_{ij}^{k-1}$ contains exactly the paths which are not taken into account in the computation of h_{k-1}. Each of these paths have node k as an intermediate node.

It is sufficient to consider only paths between i and j that contain only one occurrence of k as an intermediate node. This is because the presence of cyclic paths P_{kk} as a part of a path P_{ij} does not change $T(P_{ij})$. To see this, consider a path P_{ij} which contains a cycle P_{kk}. Let $T(P_{kk}) < T(P'_{ij})$, where P'_{ij} is the path which is obtained by removing P_{kk} from P_{ij}. Since we are looking for a path with maximal transitive influence we can always disregard P_{kk} since P'_{ij} is a better candidate path in the computation of the maximal T. In case $T(P_{kk}) > T(P'_{ij})$, $T(P_{kk})$ again does not play a role in the computation of $T(P_{ij})$ since P'_{ij} cannot be omitted from P_{ij}. Hence, each considered path P_{ij} between nodes i and j can be split into two parts P_{ik} between nodes i and k and P_{kj}, between k and j. Notice that both P_{ik} and P_{kj} are in Π_{ij}^{k-1} since they do not feature k as an intermediate node. Thus, the paths in $\Pi_{ij}^k \setminus \Pi_{ij}^{k-1}$ can be covered in the computation by considering all concatenations of pairs of such paths (P_{ik}, P_{kj}), which gives:

$$\max_{P_{ij} \in \Pi_{ij}^k \setminus \Pi_{ij}^{k-1}} T(P_{ij}) = \max_{(P_{ik}, P_{kj}) \in \Pi_{ij}^{k-1} \times \Pi_{ij}^{k-1}} T((P_{ik}, P_{kj})) \tag{2.3}$$

Since we assume $T_{\min}$, we have $T((P_{ik}, P_{kj})) = \min(T(P_{ik}), T(P_{kj}))$. Using the fact that set of transitive affinities $T(P_{ij}$ together with max and min form a distributive lattice, the right hand side of 2.3 can be rewritten as:

$$\begin{aligned} \max_{(P_{ik}, P_{kj}) \in \Pi_{ij}^{k-1} \times \Pi_{ij}^{k-1}} T((P_{ik}, P_{kj})) &= \max_{(P_{ik}, P_{kj}) \in \Pi_{ij}^{k-1} \times \Pi_{ij}^{k-1}} \min(T(P_{ik}), T(P_{kj})) \\ &= \min(\max_{P_{ik} \in \Pi_{ij}^{k-1}} T(P_{ik}), \max_{P_{kj} \in \Pi_{ij}^{k-1}} T(P_{kj})) \\ &= \min(h_{ik}^{k-1}, h_{kj}^{k-1}) \end{aligned}$$

By putting the last equalities together with 2.2 and 2.3 we obtain the desired result.

□

The next two propositions ensure that each new iteration in the computation of h_{ij} is an improvement with regard to the previous one:

Proposition 2.2. *For all* $i, j, k \in \underline{n}$

$$h_{ij}^{k-1} \leq h_{ij}^k.$$

Proof. This inequality follows directly from Prop. 2.1, since one of the arguments of the max function is h_{ij}^{k-1}. □

Sequence $\{h_{ij}^k\}_{k=1,\ldots,n}$ converges to h_{ij}^n, i.e.

Proposition 2.3. *For all* $i, j \in \underline{n}$

$$h_{ij}^n = h_{ij}.$$

This holds since all nodes are allowed to occur in the paths along which we compute h_{ij}^n.

The next property plays an important role in the parallelization of the algorithms that we present in the next sections:

Proposition 2.4. *For all* $i, j, k \in \underline{n}$*:* $h_{ik}^k = h_{ik}^{k-1}$ *and* $h_{kj}^k = h_{kj}^{k-1}$.

Proof. By Prop. 2.1 and the properties of max we have:

$$h_{ik}^k = \max(h_{ik}^{k-1}, \min(h_{ik}^{k-1}, h_{kk}^{k-1})) = h_{ik}^{k-1}$$

and

$$h_{kj}^k = \max(h_{kj}^{k-1}, \min(h_{kk}^{k-1}, h_{kj}^{k-1})) = h_{kj}^{k-1}.$$

□

3. Transitive Reduction of a Weighted Graph

For weighted graphs, the informal criterion for preserving an edge (i, j) in the reduced graph, that we mentioned above, was that the direct influence is at least as strong as the maximal indirect transitive influence, which is formally expressed as:

$$h_{ij} \leq w_{ij}. \tag{3.1}$$

Together inequalities 2.1 and 3.1 imply $h_{ij} = w_{ij}$. Hence, we have the following definition:

Definition 4. *The* transitive reduction of a weighted graph $G = (V, E, w)$ *is the graph* $G^t = (V, E^t, w^t)$ *with* $E^t = \{(i, j) \in E | w_{ij} = h_{ij} \}$ *and* $w^t(i, j) = w(i, j)$*, for all* $(i, j) \in E^t$.

In other words, edge (i, j) is preserved in the reduced graph E^t if and only if its weight equals the maximal transitive influence between nodes i and j. The edge weights remain unchanged in the reduced graph. Unlike its counterpart for unweighted graphs, the above definition includes also cyclic graphs and their reduced graph G^t is unique.

In general, also the refined notion of transitive reduction with weights does not completely resolve the anomaly of removing an edge from the graph which corresponds to an influence that actually exists in the real network. One way to further improve the filtering of the edges is to introduce *thresholds*. We introduce an upper threshold t_u determining that any edge (i, j) with $w_{ij} \geq t_u$ is unconditionally kept in E^t, i.e., regardless of the transitive influence between i and j. In this way we ensure that direct influences with sufficiently high strengths are not removed

from the network. Similarly, we use a lower threshold t_l such that any edge (i, j) with $w_{ij} \leq t_l$ is unconditionally removed from the network. Provided we use the transitive influence T_{min}, such a filtering with lower threshold t_l is actually independent of the transitive reduction concept and it can be done as a preprocessing step before the transitive reduction.

In presence of thresholds the definition of transitive reduction needs to be adjusted:

Definition 5. *The* transitive reduction of a weighted graph $G = (V, E, w)$ with an upper threshold t_u and a lower threshold t_l *is the graph* $G^t = (V, E^t, w^t)$ *with* $E^t = \{(i, j) \in E | (w(i, j) = h_{ij} \wedge \neg w(i, j) \leq t_l) \vee w(i, j) \geq t_u\}$, *and for all* $(i, j) \in E^t$ *it holds* $w^t(i, j) = w(i, j)$.

By setting $t_u = 0$ or $t_l = 0$ we obtain transitive reduction without upper or lower threshold, respectively.

4. Algorithms for Transitive Reduction

In this section we present algorithms for transitive reduction of weighted graphs. They are based on the Floyd-Warshall algorithm [4, 11] and as such they can be efficiently parallelized. Although the algorithms are of polynomial complexity for practical applications this parallelization is important since it allows scalability. For instance, in the applications to genetic networks, whole genome knockout experiments with tens of thousands of nodes can be processed with modern desktop computers.

4.1. Straightforward Algorithm for Transitive Reduction of Weighted Acyclic Graphs.

The first algorithm that we present is a direct implementation of Def. 4. The input of the algorithm is the weight matrix W of the graph. The output is the modified weight matrix of the reduced graph in which the weights of the removed edges are set to 0. The core of Algorithm 1 consists of the nested for loops in lines 3-5 and is borrowed from [3]. It computes the transitive influences between each pair of nodes in the matrix $\{g_{ij}\}$. This is done by direct application of Prop. 2.1 which is implemented by the assignment in line 5. After that, in lines 6-7 the edges are removed for which the weights are smaller than the transitive influences.

Since in the second part of Alg. 1 only the edges (i, j) are removed for which $w_{ij} < g_{ij}$, for the correctness of the algorithm as a whole it is sufficient to show that after the termination of the for loop in line 3, for each pair i, j it holds $g_{ij} = h_{ij}$. To this end we show by induction that after the execution of the k-th iteration of the line 3 loop, it holds $g_{ij} = h^k_{ij}$. This is in fact a loop invariant that is valid before entering the line 3 loop, since after the initialization done in the line 1 for loop we have $g_{ij} = w_{ij} = h^0_{ij}$. By the induction hypothesis the invariant holds also before entering the line 4 loop in the $(k + 1)$-th iteration. By Prop. 2.1 the invariant is preserved by the assignment in line 5. Because of Prop. 2.4 these assignments can be performed in an arbitrary order since the value of g_{ik} and g_{kj} remains the same in iterations k and $k + 1$. As a consequence, the loop in line 4 can be executed in

Algorithm 1 Transitive reduction for weighted acyclic graphs (straightforward)

Input: Weight matrix $\{w_{ij}\}_{i,j\in\underline{n}}$
Output: Weight matrix $\{g_{ij}\}_{i,j\in\underline{n}}$ of the reduced graph
1: **for** $(i,j) \in \underline{n} \times \underline{n}$ **do in parallel**
2: $\quad g_{ij} := w_{ij}$
3: **for** $k = 1, \ldots, n$ **do sequentially**
4: $\quad$ **for** $(i,j) \in \underline{n} \times \underline{n}$ **do in parallel**
5: $\quad\quad g_{ij} := \max(g_{ij}, \min(g_{ik}, g_{kj}))$
6: **for** $(i,j) \in \underline{n} \times \underline{n}$ **do in parallel**
7: $\quad$ **if** $w_{ij} < g_{ij}$ **then**
8: $\quad\quad w_{ij} := 0$

parallel, which we exploit to improve the efficiency of the algorithm. Finally, by Prop. 2.3 after the last n-th iteration we have $g_{ij} = h^n_{ij} = h_{ij}$.

4.2. **A Memory Efficient Algorithm.** Algorithm 1 needs two matrices: the weight matrix plus the auxiliary matrix g_{ij} to store the (restricted) transitive influences. This can be a drawback if one is confronted with limited memory, which can be the case if we use GPUs. To alleviate this problem we give a version of the algorithm which requires less memory. Conceptually, it still uses two matrices: the input matrix of weights (in this case it is assumed that this is given directly as the auxiliary matrix $\{g_{ij}\}$) and Boolean matrix $\{r_{ij}\}$, which indicates if an edge (i,j) should be removed. The advantage of the Boolean matrix is that it needs only one bit per element. Actually, in practice this bit can be implemented as a sign of the values in $\{g_{ij}\}$, which are usually floating points. Thus matrix g_{ij} is sufficient. The output of the algorithm is the modified weight matrix $\{g_{ij}\}$.

Algorithm 2 has a structure similar to Algorithm 1. In lines 1 and 2 the values r_{ij} are initialized to *False* which means that initially we assume conservatively that no edge needs to be removed. Lines 4-8 are expanded version of lines 3-5 of Alg. 1 and as such it computes the restricted transitive influences of the original graph. It is straightforward to check that regarding the assignment to g_{ij}, lines 5-7 are equivalent to the statement $g_{ij} := \max(g_{ij}, \min(g_{ik}, g_{kj}))$ in line 5 of Alg. 2. In addition, in line 8, the value of r_{ij} is set to *True* to record that edge (i,j) should be removed. This is based on the observation that for an edge to be removed it suffices that in at least one iteration the weight of the edge is smaller than the (restricted) influence of the graph.

The last part of Alg. 2, lines 7 and 8, is analogous to the final part of Alg. 1, lines 10 and 11. Based on the value of r_{ij} edges are removed from the original graph.

To establish the soundness of Algorithm 2 we need to show that if an edge (i,j) is removed in lines 10 and 11, i.e., if $r_{ij} = \mathit{True}$, then also according to the definition, this edge needs to be removed from the original graph, i.e., r_{ij} implies $w_{ij} < h_{ij}$.

Algorithm 2 Transitive reduction for weighted acyclic graphs (memory efficient)

Input: Weight matrix $\{g_{ij}\}$ of the original graph
Output: Modified weight matrix $\{g_{ij}\}$ of the reduced graph

```
1: for (i, j) ∈ n × n do in parallel
2:    r_ij := False
3: for k = 1, ..., n do sequentially
4:    for (i, j) ∈ n × n do in parallel
5:       g* = min(g_ik, g_kj)
6:       if g_ij < g* then
7:          g_ij := g*
8:          r_ij := True
9: for (i, j) ∈ n × n do in parallel
10:   if r_ij then
11:      g_ij := 0
```

Analogously to Alg. 1, after the termination of the k-th iteration of the parallel loop in line 3 it holds

$$g_{ij} = h_{ij}^k. \tag{4.1}$$

Again we establish validity of this invariant by means of induction. The invariant 4.1 holds before entering the line 3 loop, since by definition for the elements of the input matrix we have $g_{ij} = w_{ij} = h_{ij}^0$. It is straightforward to check that the property is preserved by the iteration step: the statements in lines 5-7 of Alg. 1 are equivalent to the assignment $g_{ij} := \max(g_{ij}, \min(g_{ik}, g_{kj}))$ in line 5 of Alg. 2. For, if $g_{ij} < g^* = \min(g_{ik}, g_{kj})$ then $\max(g_{ij}, \min(g_{ik}, g_{kj})) = g^*$. Otherwise, the maximum is g_{ij} and its value remains unchanged. Thus, by the induction hypothesis 4.1 we have that before the assignment in line 7 in the k-th iteration it holds: $g_{ij} = h_{ij}^{k-1}$, and also by Prop. 2.4, $g_{ik} = h_{ik}^{k-1} = h_{ik}^k$ and $g_{kj} = h_{kj}^{k-1} = h_{kj}^k$. By Prop.2.1 this implies that after the assignment, invariant 4.1 is established.

Boolean r_{ij} can be changed to *True* only by the assignment in line 8. So, once modified (to *True*) the value is never reverted to *False*. Let us consider the greatest $k \geq 1$ for which r_{ij} is modified. The assignment is performed under the condition $g_{ij} < g^*$. Taking into account 4.1 and the assignment in line 7 and using Prop. 2.2 we conclude that $h_{ij}^{k-1} = g_{ij} < g_{ij}^* = h_{ij}^k$. Hence, $r_{ij} = \mathit{True}$ after the termination of the loop 3, implies the following chain of (in)equalities $w_{ij} = h_{ij}^0 \leq h_{ij}^{k-1} < h^k \leq h_{ij}^n = h_{ij}$, which establishes the required inequality and with that the soundness of the algorithm.

For the completeness we have to show that for each edge (i, j) such that $w_{ij} < h_{ij}$, value r_{ij} becomes eventually *True* after the execution of loop 3. We do this by contradiction. Suppose that there exist such an edge (i, j) with $w_{ij} < h_{ij}$, but after the loops are terminated $r_{ij} = \mathit{False}$. Since r_{ij} is changed if and only if g_{ij} is changed (lines 7 and 8) we conclude that also g_{ij} is unchanged after the termination of the loops and therefore after the last iteration $g_{ij} = w_{ij}$, by the definition of the input of the algorithm. On the other hand, we already showed

that after the termination of loop 3 also $g_{ij} = h_{ij}$ and therefore $w_{ij} = h_{ij}$ which contradicts our initial assumption.

4.3. **Algorithm for Transitive Reduction with Upper Threshold.** The idea behind the third version of the algorithm with upper threshold is similar. The only modification is in the condition which determines when an influence of the edge is updated and consequently the corresponding element of the Boolean array is set. In this version of the algorithm there are edges which are "protected" in the sense that they are preserved, the Boolean element is not set, if the edge weight is greater than the given threshold t_u. The transitive reduction with upper threshold can be generated by an algorithm which is obtained from Alg. 2 by replacing the condition $g_{ij} < g^*$ in line 6 by

$$g_{ij} < g^* \textbf{ and } g_{ij} < t_u.$$

The correctness of the modified algorithm can be shown in a way analogous to Alg. 2. The only difference is that the loop relation 4.1 becomes $g_{ij} \leq h^k_{ij}$. As a consequence, r_{ij} implies $(w_{ij} < h_{ij}) \wedge (w_{ij} < t_u)$. The completeness arguments are the same as for Alg. 2.

4.4. **Complexity, Generalizations, GPUs.** All of the presented algorithms are obviously of polynomial complexity $O(n^3)$, where n is the number of nodes in the graph. They can be also implemented efficiently on parallel architectures, in particular shared memory ones, like modern GPUs. Experiments with our prototype implementations show significant speed ups of more than 40 compared to their sequential counterparts. Regarding the parallel version of the algorithms, assuming $p \geq n^2$, where p is the number of processors and n the size of the graph, we obtain time complexity $O(n)$, i.e., linear in the number of nodes.

Besides the 'weakest' link transitive influence T_{min} that we used throughout the paper, one can use the complementary T_{max} transitive influence in which the transitive influence of a path is determined as its maximal edge weight. Analogously, instead of the maximal transitive influence we have to work with a minimal one. Also we need to use ∞ as a special value that indicates absence of edge. In general, the edge weights should be from a set which is a lattice and in the definitions supremum (greatest upper bound) and infimum (least lower bound) can substitute instead of maximum and minimum, respectively. In the proofs the only constraints that we used on T is that it was associative and the counterparts of the max and min operations have to be mutually distributive. In that case we can reuse completely the schema of our algorithm by only replacing the operations and the designated values.

5. Conclusions

We presented a generalization of the notion of transitive reduction for weighted graphs. Both algorithms are of polynomial complexity $O(n^3)$, where n is the number of nodes in the graph. They can be also implemented efficiently on parallel architectures, in particular shared memory ones, like modern GPUs.

We implemented both sequential and parallel versions of the algorithms. Currently we have been performing experiments for inference of genetic networks. The first experiments are quite encouraging, both in run time and quality of the network reconstruction, especially taking into account the relative simplicity of the concept.

An interesting avenue for future work would be to find other applications for transitive reductions of weighted graphs beyond genetic networks. Probably to this end one would need to use different definitions of transitive influence and counterparts to the minimum and maximum operations.

References

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. *The Transitive Reduction of a Directed Graph*, SIAM Journal on Computing, **1**(2) (1972), 131–137.

[2] P. Chang and L. J. Henschen. *Parallel transitive closure and transitive reduction algorithms*, in: *International Conference on Databases, Parallel Architectures and Their Applications, PARBASE-90*, (1990) 152–154.

[3] C. Ding, X. He, H. Xiong, H. Peng, and S. R. Holbrook. *Transitive closure and metric inequality of weighted graphs; detecting protein interaction modules using cliques*, Int. J. Data Min. Bioinformatics, **1(2)** (2006), 162–177.

[4] R. W. Floyd. *Algorithm 97: Shortest path*, Commun. ACM, **5(6)**, (1962), 345.

[5] A. Goralcíková and V. Koubek. *A reduct-and-closure algorithm for graphs*, in: J. Becvár (ed.), *Mathematical Foundations of Computer Science 1979*, Lecture Notes in Computer Science **74**, Springer Berlin / Heidelberg (1979), 301–307.

[6] D. Gries, A. J. Martin, J. L. van de Snepscheut, and J. T. Udding. *An algorithm for transitive reduction of an acyclic graph*, Sci. Comput. Program., **12(2)** (1989), 151–155.

[7] S. Klamt, R. J. Flassig, and K. Sundmacher. *Transwesd: inferring cellular networks with transitive reduction* Bioinformatics, **26** (2010), 2160–2168.

[8] D. M. Moyles and G. L. Thompson. *An algorithm for finding a minimum equivalent graph of a digraph*, J. ACM, **16** (1969), 455–460.

[9] A. Tresch, T. Beissbarth, H. Sültmann, R. Kuner, A. Poustka, and A. Buness. *Discrimination of Direct and Indirect Interactions in a Network of Regulatory Effects*, Journal of Computational Biology, **14(9)** (2007), 1217–1228.

[10] A. Wagner, *How to reconstruct a large genetic network from n gene perturbations in fewer than n^2 easy steps*, Bioinformatics, **17(12)** (2001), 1183–1197.

[11] S. Warshall, *A theorem on boolean matrices*, J. ACM, **9(1)** (1962), 11–12.

Department of Biomedical Engineering, Eindhoven University of Technology,
PO Box 513, 5600 MB, Eindhoven, The Netherlands
E-mail address: dragan@win.tue.nl