ISSN 1884-0760

GRACE TECHNICAL REPORTS

# Graph Generation via Reverse Iterative Query Processing

Makoto ONIZUKA, Hiroyuki KATO, Soichiro HIDAKA,
Keisuke NAKANO, Zhenjiang HU

GRACE-TR 2016–02 March 2016

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

**WWW page: http://grace-center.jp/**

# Graph Generation via Reverse Iterative Query Processing

**Makoto Onizuka**[1], **Hiroyuki Kato**[2], **Soichiro Hidaka**[2], **Keisuke Nakano**[3], **Zhenjiang Hu**[2]
[1]Graduate School of Information Science and Technology, Osaka University
[2]National Institute of Informatics/SOKENDAI
[3]Faculty of Informatics and Engineering, University of Electro-Communications

**Abstract**

Automatic generation of application-specific big graphs is becoming one of the most important features of a benchmark. In this paper, we report our on-going work and partial results on big graph generation based on reverse query processing. We show how to deal with iterative queries by transforming iterative query computation to computation of the fixed point of a flat query function, from which a set of constraints can be systematically derived for reverse query processing. And we propose three approaches to parallelizing reverse query processing, namely by query rewriting, virtual node introduction and Kronecker's product.

## 1 Introduction

Graphs capture complex data dependencies and play a significant role in a wide range of applications (McCune, Weninger, and Madey 2015), such as page-ranking, k-means clustering, semi-supervised learning based on random graph walks, web search based on link analysis, and social community detection based on label propagation. To effectively test and benchmark these applications, automatic generation of application-specific big graphs is becoming one of the most important features of a benchmark (Capota et al. 2015). As synthetic data model specifications (such as maximum of the node degrees and topological structures) evolve over time, the data generator programs implementing these models should be adapted continuously – a task that often becomes more tedious as the set of model constraints grows.

One promising method to address this problem is to use the known technique called reverse query processing (RQP) (Binnig, Kossmann, and Lo 2007), which gets a query and a result as input and returns a possible database instance that could have produced that result for that query. With RQP, we may use a query to declaratively specify the intended behavior of an application (e.g., computation of page ranking), and then generate a set of test data from the query and an expected result and test the application. Moreover, if we specify statistical properties of data (e.g., community-like graphs (Leskovec et al. 2008)) using queries, we can generate data with such properties. This makes data generation be more flexible and adaptive to different requirements.

However, there are two challenges in importing RQP for big graph generation. First, it is often that an application behavior on graphs is specified by iterative queries rather than simple SQL queries that can be addressed so far (Binnig, Kossmann, and Lo 2007). This calls for a method to reverse iterative queries. Second, to generate big graphs, it is impossible to adopt existing sequential approach to data generation with a set of global constraints. Rather we need to investigate how to reverse the query in a divide-and-conquer manner so that RQP can be done efficiently in parallel.

In this paper, we report our on-going work and partial results on big graph generation based on RQP. We start by showing how to deal with the first problem by transforming iterative query computation to computation of the fixed point of a flat query function, from which a set of constraints can be systematically derived for reverse query processing. This empowers the existing RQP, but yet cannot generate big graphs due to its sequential computation manner.

Then, we proceed to solve the scalability problem by optimizing and parallelizing the RQP of the fixed point computation. First, we show that by fusing query sequences and promoting the convergence constraints to the flat query for the fixed point computation, we may rewrite it to an efficient and parallelzable form in which the group-by/aggregation computation can be computed in parallel. Second, (if the first approach cannot apply,) we may construct and solve a set of suitable constraints to introduce virtual connecting nodes such that generation of a big graph can be divided into that of smaller graphs. Third, (if the second approach cannot apply,) we may map the fixed point computation of a flat query into a computation for finding an $n$-dimensional vector $v$ satisfying $Av = v$ for a given $n \times n$ matrix $A$. With Kronecker's product, we can obtain a divide-and-conquer algorithm to compute an approximate of such $v$.

The rest of this paper is organized as follows. We show how to extend RQP to iterative queries in Section 2. Then we propose our three approaches to divide-and-conquer parallelization, by query rewriting in Section 3, by virtual node introduction in Section 4, and by Kronecker's product in Section 5. We conclude the paper in Section 6.

## 2 Applying RQP to Iterative Queries

In this section, we describe how to apply RQP to small graph generation. In RQP, actual data generation is done using Model checker by giving constraints, which are satisfied in the generated data. We start by given an overview on RQP, then how to extract such constraints from iterative queries is described. A key is to transform iterative queries into fixpoint queries.

## 2.1 Reverse query processing

RQP, which performs in relational databases, gets a query Q and a result R as input and returns a database instance D such that;

$$R = Q(D).$$

To this end, RQP extracts constraints, which satisfies the above equation, from the query Q. Then, a database instance D is generated by using Model checker by giving the constrains. To extract the constraints, reverse relational algebra (RRA) [1] is defined so that for each algebraic operator *op* in relational algebra, right-inverse of *op* is defined, that is the following equation holds:

$$op(op^{-1}(R)) = R.$$

In summary, RQP proceeds the following steps;

- (step 1): Constructing a reverse query tree T for a given SQL query Q.
- (step 2): Extracting constraints from T for a given database schema S.
- (step 3): Data instantiation satisfying the constraints.

## 2.2 Iterative queries and fixpoint computations

Consider applying RQP to interative queries. To apply RQP to iterative a query IQ, we have to extract appropreate constraints from IQ because, in RQP, the actual data instantiation is done using Model cheker (SMT solver) by giving constraints collected from the input queries.

However, RQP is not applicable to iterative queries, because RQP is designed only for non-iterative queries. There is a challenge to apply RQP to iterative queries. We could apply RQP to each iteration of iterative queries, however it is not obvious when to stop the iteration of the reversed query, or it is difficult to generate input data that satisfies initial constrant after the reversed iterations; all the page scores should be equal in PageRank computation for example.

Our idea to tackle this problem is to associate iterative queries with fixpoint computations. Now, we consider the input, the output and a reverse operation of IQ to extract constraints. Typical iterative queries for graph anaysis can be represented as the following form:

```
1: g = (inuput graph structure)
2: v_new = (initializing data)
3: do
4:   v_old = v_new
5:   v_new = update(g, v_old)
6: until converge(v_new, v_old)
7: return v_new
```

In this form, the input is a graph structure g in line 1 and an initial value of analysis v_new in line 2, and the output is a analysis value v_new in line 7.

We observe that there are two characteristic aspects in such iterative queries. The first one is that an input graph structure is unchanged in each iteration, and the second is that terminal condition of the iteration is that analysis value is unchanged (or, the difference of analysis values between previous iteration and current iteration is less than some threshhold. ). From these observation, we can see an iterative query as a following fixpoint equation in line 2

```
1: g = (inuput graph structure)
2: v = update(g,v)
3: return v
```

Actually, the consraints we should extract from the fixepoint equation are in the direct definition of the graph analysis computations. Note that how to extract such constraints from iterative queries in OptIQ (Onizuka et al. 2013) by using query rewriting will be described in the next section. In the rest of this section, we will see a PageRank example to show how to generate a small graph using a SMT solver by giving constraints.

### PageRank example

Usually, PageRank computation can be done in iterative way. However, when we consider the constraints of the graph we want to generate, the direct solution can be used. The direct solution of the PageRank computation is fomulated by the following equation.

A simplified PageRank equation: [2]:

$$P(n) = \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

where $P(n)$ is the PageRank $P$ of a page $n$, $L(n)$ is the set of pages that link to $n$, and $C(m)$ is the out-degree of node $m$ (the number of links on page $m$). We can see the PageRank equation as the above form of fixpoint equation by assocating update and g with $\sum_{m \in L(n)}$ and $C(m)$, respectively.

Now, we can consider the constraints from this equation. The constraints to give an SMT solver are (a) the total number of nodes as $Int$, (b) the PageRank $P$ of a page (node) $n$ is represented as a function $rank : Node \to Real$, (c) the weighted edge is represented as a function $edge : Node \times Node \to Real$, (d) the above equation is represented as a function $sumOfOutRank : Node \times Node \to Real$ with the assertion that for all nodes, the sum of the weighted edges equals to the rank value, and (e) some other constraints such as no self cycles are described.

We have tested to generate graphs by using an SMT solver, Yices[3]. Figure 1 shows the input file which describes the above constraints with the total number of nodes 5. Unfortunatelly, this method does not scale for big graphs generation due to the global constraint of PageRank computation, namely we have to check that for all nodes the PageRank constraint is hold. This preliminary experiment drives us to develop a novel big data generation method. The following sections report the partial results of our effort.

[1]In precise, RRA is not an algebra and its operators are not operators because they allow different outputs for the same input.

[2]For simplicity, we use the random jump factor $\alpha$ as 0 in the precise definiton: $P(n) = \alpha(\frac{1}{|G|}) + (1-\alpha)\sum_{m \in L(n)} \frac{P(m)}{C(m)}$ without loss of genericily.

[3]http://yices.csl.sri.com/

```
;; (a) number-of-nodes : total number of nodes
(define number-of-nodes::int 5)
;; node : type of node
(define-type node (subrange 0 (- number-of-nodes 1)))

;; (b) rank function
(define rank::(-> node real))
(assert (= (rank 0) 3/10))
(assert (= (rank 1) 3/10))
(assert (= (rank 2) 2/10))
(assert (= (rank 3) 1/10))
(assert (= (rank 4) 1/10))

;; (c) weithted edge function
(define edge::(-> node node
                  (subtype (x::real) (and (<= 0 x) (<= x 1)))))

;; (d) PageRank constraint
(define sum-of-out-rank::(-> node node real)
  (lambda (v::node n::node)
    (if (< n 0) 0 (+ (edge v n) (sum-of-out-rank v (- n 1))))))
(assert (forall (v::node)
          (= (rank v) (sum-of-out-rank v (- number-of-nodes 1)))))

;; (e) no self cycle
(assert (forall (v::node) (= (edge v v) 0)))

;; (e)
(assert (forall (v::node)
(exists (w::real)
(forall (u::node)
  (or (= (edge v u) 0)
      (= (edge v u) w))))))
```

Figure 1: Yices, an SMT solver, inputs for PageRank

## 3 Extracting Constraints from Iterative Queries in OptIQ

There are two problems for RQP to be applied to iterative queries. As we have discussed in Section 2, the first one is that it is not obvious when to stop the iteration of the reversed query, or it is difficult to generate input data that satisfies initial constant after the reversed iterations. The second problem is that RQP for iterative queries is not efficient when the queries are expressed in a sequence of simple queries, which is easier for programmers to implement compared to implement a single complicated query. Consider each iteration is expressed in the form of T = q(T, D) where T is update table, which is updated in each iteration until convergence, and D is an input table, which is not updated during iterations. Let assume iterative query q is expressed in a sequence of queries, say q = $q_n ... * q_0$. In the reverse query processing, T is input to the reversed query of $q_n$, and then its result is input to $q_{n-1}$. This processing is continued until $q_0$. The result of reverse query of $q_0$ must be identical to T, because T is converged at the starting time in the reverse query processing, To satisfy this constraint for a sequence of queries causes a serious issue in efficiency, because we have to backtrack the reverse query processing from $q_0$ to $q_n$ when the result of reversed query of $q_0$ is not identical to T.

We propose an approach for reverse query processing for iterative queries. Our approach uses two techniques to tackle the above difficulties. The first technique is to rewrite iterative queries to non-iterative ones, fixpoint queries. This query rewrite does not change the semantics of queries, because the output of the forward query is converged, so iterative queries can be rewritten to the queries without iterations. The second technique is to rewrite a sequence of queries in each iteration so that multiple queries referring to input table are to be merged into a single query. We can avoid backtracking and achieves efficient reverse query processing. After applying the above two techniques, we can apply RQP to the fixpoint queries and extract constraints from the reversed queries and input them into SMT solvers to generate input data from reversed queries.

### 3.1 Query rewrite to fixpoint queries

The first technique is to rewrite iterative queries to fixpoint queries. After the convergence of forward query, iterative queries are no need to iterate any more and the convergence constraint becomes a constant constraint for output data.

An iterative query is expressed in a general form of:

```
iterate
   set T = q(T,D)
until $\phi$(new,old) on T
```

where T is update table, which is updated in each iteration until convergence, D is an input table, and q is an iterative query. According to the query specification in OptIQ (Onizuka et al. 2013), the convergence constraint $\phi$ is expressed with the difference between new record (before executing the iteration) and old record (value after executing the iteration) on update table T. new and old are special aliases that refer to new table (T on the left-hand side of T = q(T,D)) and old update table (T on the right-hand side of T = q(T,D)), respectively. This iterative query is rewritten to a fixpoint query after convergence:

```
1st query:
  set T = q(T,D)
2nd query:
  select *
  from T
  where $\phi$(new,old)
  and new.key = old.key;
```

where new.key and old.key is key attribute of new and old update table, respectively. The 1st query is extracted from the iterative query; it is the inside part of the iteration. The 2nd query is obtained by adding key attribute condition, new.key = old.key, to the convergence constraint, because the new and old record share the same value on the key attribute of the update table.

### 3.2 Merging multiple queries

The second technique is to rewrite a sequence of queries in each iteration into a single query so as to avoid backtracking during reverse query processing.

We first explain how backtrack occurs when a query is expressed in a query sequence. An iterative query for k-means clustering is written in OptIQ (Onizuka et al. 2013) as follows:

**Schema:**

```
Centroid(id,pos)
Point(id,cid,pos)
```

**Query:**

```
iterate
  let Point =
     select id, closest(p,Centroid) as c, pos
     from Point as p;
```

```
5:  set Centroid =
6:     select c.id, avg(pos)
7:     from Point
8:     group by c;
9: until |new.pos-old.pos| < ε on Centroid
```

where `closest(p,Centroid)` returns the closest record in Centroid to `p` and it is defined as:

```
closest(p,Centroid) =
     select c
     from Centroid as c
     order by distance(c,p)
     limit 1
```

Notice that there are two queries (1st query in line 2-4 and 2nd query in line 5-8) that commonly refers to Point table that is the input table of the forward query processing. We generate Point records by following the reverse order of the queries, 2nd query and then 1st query. If the Point records generated by the 2nd query does not satisfy the constraint specified by the 1st query, we have to backtrack the reverse query processing so that the generated Point records would satisfy the constraint of the 1st query.

We merge multiple queries into a single query so as to avoid the backtracking. Here we employ techniques for traditional query simplification (selection/projection push down, identity query removal) and/or scan sharing (Nykiel et al. 2010; He et al. 2010). The scan sharing is an optimization technique for forward query processing, but we employ this technique also to backward query processing. The idea of scan sharing is that scanning same tables in different queries are shared so that those queries are evaluated by scanning the same tables at the same time. By applying this technique to the backward query processing, the constraints on the input table expressed in multiple queries are merged into a single query.

The merging queries works as follows for k-means clustering. First, we simplify the query by pushing `closest(p,Centroid)` down to the 2nd query from the 1st query:

```
1: iterate
2:  let Point =
3:     select id, cid, pos
4:     from Point
5:  set Centroid =
6:     select c.id, avg(pos)
7:     from Point as p
8:     group by closest(p,Centroid) as c;
9: until |new.pos-old.pos| < ε on Centroid
```

Then, the 1st query is an identity query, so we can omit it.

```
1: iterate
2:  set Centroid =
3:     select c.id, avg(pos)
4:     from Point as p
5:     group by closest(p,Centroid) as c;
6: until |new.pos-old.pos| < ε on Centroid
```

## 3.3 Examples

We explain how the above two techniques work for the examples of k-means clustering and PageRank computation.

**k-means clustering example** We apply the first technique to the query obtained after merging queries for k-means clustering example and then obtain fixpoint queries as follows:

```
1st query:
 1: set Centroid =
 2:    select c.id, avg(pos)
 3:    from Point as p
 4:    group by closest(p,Centroid) as c;
2nd query:
 5: select *
 6: from Centroid
 7: where |new.pos-old.pos| < ε
 8: and new.key = old.key;
```

Since the select clause in line 2 forms new update table (`new`) after the query and the result of closest function, `c`, refers to old update table (`old`), `new.pos-old.pos` and `new.key = old.key` are rewritten to `avg(pos)-c.pos` and `c.id = c.id`, respectively. Then we can remove `c.id = c.id` because it is a tautology. Finaly, we can merge the above queries into a single query:

```
 1: set Centroid =
 2:   select c.id, avg(pos)
 3:    from Point as p
 4:    group by closest(p,Centroid) as c
 5:    having |avg(pos)-c.pos| < ε;
```

Notice that the constraint expressed by this query (in line 4 and 5) is independently solved for every Centroid record in reverse query processing. So, we can solve this constraint by divide and conquer algorithms. For each Centroid record, we can generate Point records so that they satisfy the constraints. This result is quite different from other cases, such as PageRank computation, because we cannot remove the constraint between records (`new.key = old.key`) in general. We describe the detail of the PageRank computation example in the next section.

In addition, so as to more efficiently generate Point records that satisfy the constraint expressed by `closest(p,Centroid)` (in line 4), we construct Voronoi diagram (Aurenhammer 1991) by setting Centroid records as Voronoi seeds. We can generate Point records inside of each Voronoi, then those records automatically satisfy the closest constraint.

**PageRank computation example** An iterative query for PageRank computation is written in OptIQ (Onizuka et al. 2013) as follows:

**Schema:**
```
 1: Graph(src,dest,score)
 2: Count(src,count)
 3: Score(dest,score)
```

**Query:**
```
iterate
  set Score =
     select n.dest, sum(n.score/Count.count)
     from Graph as n, Count
     where n.src = Count.src
     group by n.dest;
  set Graph =
     select m.src, m.dest, Score.score
     from Graph as m, Score
```

```
    where m.src = Score.dest;
until |new.score-old.score| < ϵ on Score
```

The two queries inside of the iterate clause are merged by using table decomposition and subquery lifting (Onizuka et al. 2013).

```
initialize
  IT_Count = select IT.src, IT.dest,Count.count
             from IT, Count
             where IT.src = Count.src;
iterate
  set Score =
    select ic.dest, sum(VT.score/ic.count)
    from Score as sc, IT_Count as ic
    where sc.dest = ic.src
    group by ic.dest;
until |new.score-old.score| < ϵ on Score
```

The iterate part of this query is rewritten to fixpoint queries as follows:

```
1st query:
  set Score =
    select ic.dest, sum(VT.score/ic.count)
    from Score as sc, IT_Count as ic
    where sc.dest = ic.src
    group by ic.dest;
2nd query:
  select *
  from Score
  where |new.score-old.score| < ϵ
  and new.key = old.key;
```

Since the select clause of the 1st query forms new update table (`new`) and Score `sc` refers to old update table (`old`), `new.score-old.score` and `new.key = old.key` are rewritten to `sum(VT.score/ic.count)-sc.score` and `ic.dest = sc.dest`, respectively. Finally, we can merge the queries and obtain a query.

```
  set Score =
    select ic.dest, sum(VT.score/ic.count)
    from Score as sc, IT_Count as ic
    where sc.dest = ic.src
    group by ic.dest
    having |sum(VT.score/ic.count)-sc.score| < ϵ
    and ic.dest = sc.dest;
```

We cannot easily solve the constraint of this query by divide and conquer algorithms because this query has a global constraint between records. There is a self-cyclic constraint for ic, `ic.dest = sc.dest = ic.src`, so the constraint affects globally to the input data. We will describe efficient techniques for solving the constraints of PageRank computation in Sections 4 and 5.

## 4 Graph Generation using Virtual Nodes

This section proposes another divide-and-conquer approach. We have seen in Section 2 that directly generating the graph at a time using (global) constraint across generated graph does not scale. Alternatively, we construct and solve a set of smaller constraints for smaller graphs with virtual connecting nodes between them so that the graph generated by connecting these graphs via the virtual nodes still satisfies the PageRank constraints. This approach scales in the sense that this decomposition can be applied recursively to these smaller graphs. Though this approach is specific to PageRank, we do not introduce any approximation, like that in Section 5. However the present approach may fail to generate graphs for given score, if the suitable partition cannot be selected.

Inspired by this approach, we further discuss the possible divide-and-conquer approach to *forward* PageRank computation provided that particular graph partition that the former approach would generate is possible for a given graph. Desikan et al. (Desikan et al. 2006) also proposed a divide-and-conquer approach. They partition into two groups of partitions so that former group have no incoming edge from another partition and have outgoing edges only to another group of partition. Our partition allow existence of cycles across partitions, though we have constraints in the number of crossing edges.

### 4.1 Divide-and-conquer approach to PageRank

Computation of PageRank is difficult to decompose for strongly-connected graphs, because the score of a node can affect any other node in the connected components. Decomposition of the problem within these components is not trivial and may lead to generation of unnatural graphs unless we carefully consider the boundary of the component. In this section, we impose a boundary condition that enables the independent generation of PageRank graphs so that the combination of these graphs are again PageRank graphs.

For the *forward* PageRank computation $\mathsf{pr} : \mathit{Graph} \to \mathit{Score}$ for given graph $g \in \mathit{Graphs}$ (we denote the set of nodes $V$ and edges $E \subseteq V \times V$ by $g.\mathrm{V}$ and $g.\mathrm{E}$, respectively) that is supposed to assign each node in graph $g$ the score of node $v \in g.\mathrm{V}$ as the function $\mathsf{score}_g = g.\mathrm{V} \to \mathit{Real}$ such that $\sum_{v \in g.\mathrm{V}} \mathsf{score}_g(v) = 1$, we define the *backward* PageRank $\mathsf{pr}^{-1} : \mathit{Score} \to \mathit{Graph}$ for given score[4] $s \in \mathit{Score}$ with $\sum_{v \in \mathrm{dom}(s)} s(v) = 1$ and $s \in \mathrm{Range}(\mathsf{pr})$ is to compute one of $\mathsf{pr}$'s (right) inverse $g \in \mathit{Graph}$ such that $\mathsf{pr}\, g = s$. We call a graph $g$ a PageRank graph for scores $s$ when $\mathsf{pr}\, g = s$.

**Example 1 (PageRank Graph)** *Figure 2 shows a graph that is generated from scores* $\{3/13, 3/13, 3/13, 2/13, 1/13, 1/13\}$. *We omitted the denominators of the scores (of the nodes) and weights (of the edges) in the figure.*

Then our divide-and-conquer computation of the backward PageRank by the decomposition $s = s_1 \oplus s_2$ of node scores is to compute graphs $g = g_1 \otimes g_2$ such that

$$\begin{aligned} g_1 &= \mathsf{pr}^{-1}\ s_1 \\ g_2 &= \mathsf{pr}^{-1}\ s_2 \end{aligned}$$

and

$$\mathsf{pr}\ (g_1 \otimes g_2) = s_1 \oplus s_2$$

We divide the backward PageRank computation $\mathsf{pr}^{-1}$ for $s$ into computations of $\mathsf{pr}^{-1}$ for $s_1$ and $s_2$ (Divide) and obtain

[4]Some application may not be interested in the absolute values of scores for individual nodes, so the input can be just a histogram of scores. This relaxation includes specification of top-$k$ scores only. It is our future work to consider these relaxed inputs.

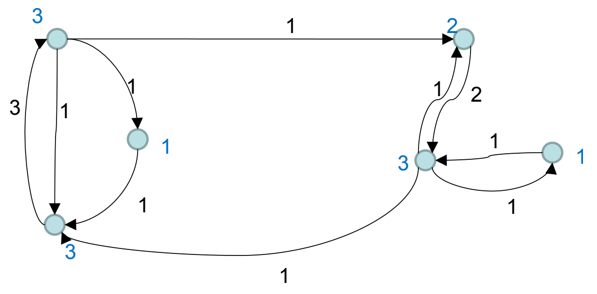


Figure 2: PageRank Graph Example

the entire result (Conquer) by combining the results using $\otimes$.

## 4.2 Introduction of virtual nodes

Our graph decomposition is based on the following observations by introducing *virtual nodes*. Consider the decomposition of a graph $g$ into two graphs $g_1$ and $g_2$ by decomposing the set of nodes of $g$ into two sets. Then $g_1$ and $g_2$ are created by these nodes, edges that do not cross the boundary of the node sets, and for each subgraph, a virtual node that represents the other subgraph, and edges incoming/outgoing the virtual node that consist of the edges that crosses the boundary. Formally, given decomposition of nodes $g.\mathrm{V} = V_1 \cup V_2$ for graph $g$ such that $\mathsf{pr}\, g = s$,

$$\begin{aligned} g_1.\mathrm{V} &= V_1 \cup \{w_1\} \\ g_1.\mathrm{E} &= \{(u,v) \mid (u,v) \in g.\mathrm{E}, u \in V_1, v \in V_1\} \\ &\cup \{(u,w_1) \mid (u,v) \in g.\mathrm{E}, u \in V_1, v \in V_2\} \\ &\cup \{(w_1,v) \mid (u,v) \in g.\mathrm{E}, u \in V_2, v \in V_1\} \end{aligned}$$

and $g_2$ given similarly, we have

$$\begin{aligned} &\mathsf{pr}\, g_1 = s|_{V_1} \cup \{w_1 \mapsto s_\mathrm{v}\} \\ &\mathsf{pr}\, g_2 = s|_{V_2} \cup \{w_2 \mapsto s_\mathrm{v}\} \\ &\text{where } s_\mathrm{v} = \textstyle\sum_{(u,v)\in g.\mathrm{E}, u\in V_1, v \in V_2} \mathsf{w}_g(u) \\ &\qquad\quad\; = \textstyle\sum_{(u,v)\in g.\mathrm{E}, u\in V_2, v \in V_1} \mathsf{w}_g(u) \end{aligned}$$

where $\mathsf{degree}_g : g.\mathrm{V} \to \mathit{Int} \stackrel{\mathrm{def}}{=} \{v \mapsto \sum_{(v,u)\in g.\mathrm{E}} 1 \mid v \in g.\mathrm{V}\}$ represents the number of outgoing edges for node $v \in g.\mathrm{V}$, $\mathsf{w}_g(v) \stackrel{\mathrm{def}}{=} \mathsf{score}_g(v) / \mathsf{degree}_g(v)$ the weight of the node $v$ for its adjacent nodes and $f|_S$ denotes the restriction of the domain of the function $f$ to the set $S$. The nodes $w_1$ and $w_2$ correspond to the virtual nodes. The (common) score of them is equal to the total flow that is incoming/outgoing from/to one partition to the other. Note that the total score is not equal to 1 anymore for each partition.

This observation implies that the forward computations after decomposition into $g_1$ and $g_2$ result in the same scores for each subset of the nodes before the decomposition, which enables the divide-and-conquer approach summarized as Algorithm 1 by decomposing the nodes into two sets, or equivalently decomposing scores (via split), independently generate the graphs allocating the score of the virtual nodes as the flow of the scores between the sets, and combine the graphs. Indeed, we have, given score partition $s = \mathsf{score}_{g_1} \oplus \mathsf{score}_{g_2}$ defined as

$$\mathsf{score}_{g_1} \oplus \mathsf{score}_{g_2} \stackrel{\mathrm{def}}{=} \mathsf{score}_{g_1}|_{g_1.V_1} \cup \mathsf{score}_{g_2}|_{g_2.V_2}$$

with the virtual nodes $w_1$ and $w_2$ having the common score $s_\mathrm{v}$,

$$\mathsf{pr}\ ((\mathsf{pr}^{-1}\ s_1) \otimes (\mathsf{pr}^{-1}\ s_2)) = s$$

$$\begin{aligned} &(g_1 \otimes g_2).\mathrm{E} \stackrel{\mathrm{def}}{=} \\ &\quad \{(u,v) \mid (u,v) \in g_1.\mathrm{E}, u \neq w_1, v \neq w_1\} \\ &\cup \{(u,v) \mid (u,v) \in g_2.\mathrm{E}, u \neq w_2, v \neq w_2\} \\ &\cup \{(u,v) \mid (u,w_1) \in g_1.\mathrm{E}, (w_2,v) \in g_2.\mathrm{E}, \\ &\qquad \textstyle\sum_{(u',w_1)\in g_1.\mathrm{E}} \mathsf{w}_{g_1}(u') = \mathsf{w}_{g_2}(w_2)\} \\ &\cup \{(u,v) \mid (u,w_2) \in g_2.\mathrm{E}, (w_1,v) \in g_1.\mathrm{E}, \\ &\qquad \textstyle\sum_{(u',w_2)\in g_2.\mathrm{E}} \mathsf{w}_{g_2}(u') = \mathsf{w}_{g_1}(w_1)\} \end{aligned}$$

Note that the flow (score) should be given appropriately for the existence of the PageRank graph. It is also worth noting that by recursively decomposing scores, a partition may include at most two virtual nodes, one of which is already introduced during the previous decomposition. Therefore, the virtual nodes are represented as sets $V_\mathrm{v}$, and passed to $\mathsf{pr}^{-1}$ as an additional argument so that the recursive call can impose virtual node conditions to these nodes.

We delegate the base case (PRINV in Algorithm 1) to SMT solvers as described in Section 4.3.

**Example 2 (D & C Graph Generation)** *Figure 3 shows the PageRank graphs $g_1$ and $g_2$ that are generated from scores $s_1 = \{v_1 \mapsto 3/13, v_2 \mapsto 3/13, v_3 \mapsto 1/13, w_1 \mapsto 2/13\}$ and $s_2 = \{v_4 \mapsto 3/13, v_5 \mapsto 2/13, v_6 \mapsto 1/13, w_2 \mapsto 2/13\}$. $2/13$ is chosen as the score $s_\mathrm{v}$ of the virtual nodes $w_1$ and $w_2$. We have $s_1 = \mathsf{pr}\, g_1$ and $s_2 = \mathsf{pr}\, g_2$. Figure 4 shows the graph $g_1 \otimes g_2$.*

As a boundary condition around the virtual nodes, the distribution of the weights of the edges incoming to a virtual node, and the distribution of the weights of the edges outgoing from the virtual node of the other partition should coincide. Alternatively, since there is no restriction in the distribution of the incoming edges of a node, if the number of the outgoing edge of a virtual node is only one, then there is no restriction on the incoming edges of the other virtual node. This simplified restriction is applied in the above definition of $\otimes$ and used in the constraint to SMT solvers described in Section 4.3.

**Example 3 (Boundary Condition)** *The graphs in Figure 3 satisfies the stronger boundary condition in that the number*

**Algorithm 1** D & C graph generation using virtual nodes

```
1:  procedure $\mathsf{pr}^{-1}(s, V_v)$
2:                                  ▷ generates a graph $g$ s.t. $\mathsf{pr}\ g = s$
3:      if nondivisible $s\ V_\mathrm{v}$ then
4:          return $\mathrm{PRINV}(s, V_v)$
5:                                  ▷ generate a graph using SMT solver
6:      else
7:          $((s_1, V_{\mathrm{v}1}), (s_2, V_{\mathrm{v}2})) \leftarrow \mathsf{split}\ s\ V_\mathrm{v}$
8:          ▷ s.t. $s_1 \oplus s_2 = s$ for the virtual node $v_\mathrm{v} \in V_\mathrm{v}$
9:          $g_1 \leftarrow \mathsf{pr}^{-1}(s_1, V_{\mathrm{v}1})$
10:         $g_2 \leftarrow \mathsf{pr}^{-1}(s_2, V_{\mathrm{v}2})$
11:         $\{v_\mathrm{v}\} \leftarrow \mathrm{dom}(s_1) \cap \mathrm{dom}(s_2)$
12:         return $g_1 \otimes_{v_\mathrm{v}} g_2$
```

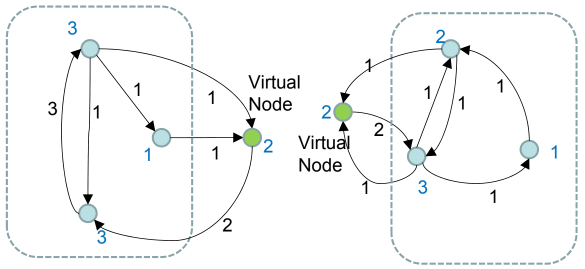


Figure 3: Divide-and-Conquer Generation of PageRank Graph (flow=2/13)

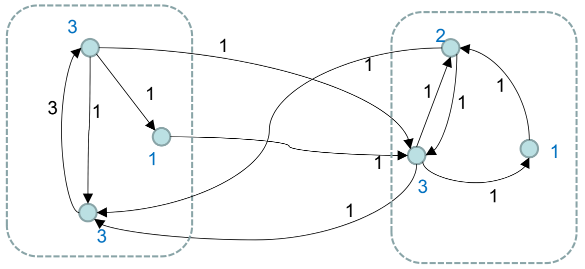


Figure 4: Conquer Phase

*of outgoing edges in the virtual nodes $w_1$ and $w_2$ are 1, allowing independent generation of PageRank graphs from $s_1$ and $s_2$. Note that $s_1$ and $s_2$ should also be in the range of* pr.

## 4.3 Generation of PageRank graphs in SMT-LIB2

After decomposing input scores to make the number of scores small enough (as determined by the predicate **indivisible** in Algorithm 1), we use SMT solvers to generate PageRank graphs (**PRINV** in Algorithm 1). We show the pseudo-code to represent the input to the solver in Figure 5. The code is similar to that of Figure 1 except that Figure 5 is adapted to the SMT-LIB2 (Barrett, Stump, and Tinelli 2010) standard syntax, introduced the virtual nodes and their boundary conditions (see the comments in the code). The scores of the non-virtual nodes are given by the first set of assert commands while the scores of virtual nodes are used by the next set of such commands. The rest of the code fragment, including the boundary conditions are the same for every invocation of **PRINV**. The result is extracted as the value assignments of the function *edge*. Thanks to the standardization by SMT-LIB2, we were able to run the code across multiple platforms including Yices2 (Dutertre 2014), Z3 (De Moura and Bjørner 2008) and CVC4 (Barrett et al. 2011). We use quantifier-free, uninterpreted function with linear integer and real arithmetics (QF_UFLIRA) as the configuration for the logic and theories. $edge\ v\ v'$ represents the fraction of the score of node $v$ that is contributed to node $v'$. Its non-zero value implies the existence of an edge between these nodes. The predicate $zeroOrFix$ represents that the contribution mentioned above is uniform. $pagerankCond\ v$ represents the PageRank condition for node $v$ that ensures the sum of the contributions from its incoming edges constitutes the score of $v$, and the sum of the contribution for $v$'s outgoing edges equals to the score. The predicate $weight$ encodes an existential quantification that ensures uniformity of weights of the edges outgoing from a node. To avoid trivial solutions in which every node has only a self-cycle, we add the corresponding condition using function $edge$. The boundary condition for the virtual node $v_{\mathrm{v}}$ is represented by $rank\ v_{\mathrm{v}} = weight\ v_{\mathrm{v}}$, indirectly encodes that the number of outgoing edges is one, so that all the score contributes to the weight.

```
$rank : Node \to Real$
$edge : Node \times Node \to Real$
$weight : Node \to Real$

assert $rank\ v_0 = s(v_0)$
assert $rank\ v_1 = s(v_1)$
...
assert $rank\ v_{\mathrm{v}1} = s_{\mathrm{v}1}$    (* virtual node *)
assert $rank\ v_{\mathrm{v}2} = s_{\mathrm{v}2}$
...
$rank\ v_{\mathrm{v}1} = weight\ v_{\mathrm{v}1}$ (* boundary condition *)
$rank\ v_{\mathrm{v}2} = weight\ v_{\mathrm{v}2}$
...

$sumInRank : Node \to Real$
$sumInRank\ v = \sum_{v' \in V} edge\ v'\ v$
$edgeOutRange : Node \to Bool$
$edgeOutRange\ v = \bigwedge_{v' \in V} 0 \leq edge\ v\ v' \leq 1$
$sumOutRank : Node \to Real$
$sumOutRank\ v = \sum_{v' \in V} edge\ v\ v'$
$zeroOrFix : Node \times Node \to Bool$
$zeroOrFix\ v\ n = edge\ v\ n = 0 \lor edge\ v\ n = weight\ v$
$allZeroOrFix\ Node \to Bool$
$allZeroOrFix\ v = \bigwedge_{v' \in V}\ zeroOrFix\ v\ v'$

$pagerankCond : Node \to Bool$
$pagerankCond\ v =$
      $rank\ v = sumInRank\ v$
   $\land\ rank\ v = sumOutRank\ v$
   $\land\ allZeroOrFix\ v$ (* uniform weight *)
   $\land\ edgeOutRange\ v$ (* weight range *)
   $\land\ edge\ v\ v = 0$  (* non-self-cycle *)
assert $\bigwedge_{v \in V}\ pagerankCond\ v$
```

Figure 5: SMT Solver inputs

Example 4 shows a generation of $g_1$ in Example 2.

**Example 4 (Generation of a Graph in a Partition)** $V = \{0, 1, 2, 3\}$ $V_{\mathrm{v}} = \{2\}$ $s(0) = 3/13, s(1) = 3/13, s(3) = 1/13, s(2) = 2/13$ *generates edges* $\{(0,1), (0,2), (0,3), (1,0), (2,1), (3,2)\}$.

### Memoization of subgraphs

The notion of virtual nodes enables us to reuse the generated graphs. We can memoize a graph generated by scores including that of virtual nodes by registering the ratios of scores, and reuse the graphs by looking up the graphs needed, by the scaled scores.

**Example 5 (Graph Generation by Reusing Memo Entries)** *Given scores in Example 1, we can first choose scores $\lbag 3/13, 3/13, 2/13 \rbag$ with flow=$1/13$, scaling by 13, and reuse the graph at the left bottom of figure 6 for one partition, and graph with scores $\lbag 3, 1, 1 \rbag$ for the other partition, and connect (Figure 7).*

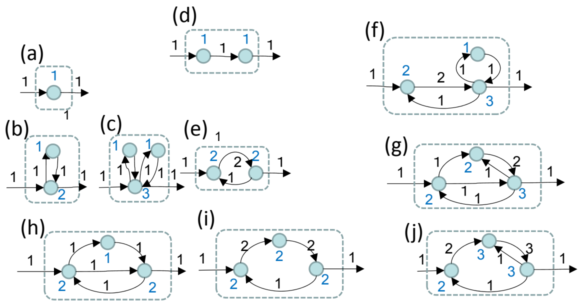
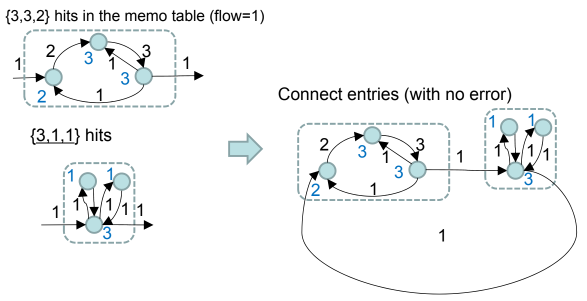


Figure 6: Memo Table for flow=1


{3,3,2} hits in the memo table (flow=1)
{3,1,1} hits
Connect entries (with no error)
Figure 7: Graph Generation by Reusing Memo Tables

## 4.4 Discussions

**Restrictions on the topology of generated graphs** We have introduced a restriction on the boundaries of partitions to enable divide-and-conquer approach, namely, every edge crossing the partitions should be connected to a common node. Natural question here is whether this restriction is realistic. We have investigated a real-world graph to see if such partitioning is possible.

Figures 8 and 9 are two different manual partitions of Zachary's Karate Club (Zachary 1977) that satisfy the restriction. Though boundary of partitions are not drawn, please consider that nodes placed on the left half of the figure belongs to the left partition and the rest of the nodes belong to the other partition. By no means these are sufficient to show that the restriction is realistic enough. However, at least we have shown that there exist a graph that can be generated by our D&C approach.

**Forward D & C computation by Rescaling scores of subgraphs** As we discussed so far, D & C computation of

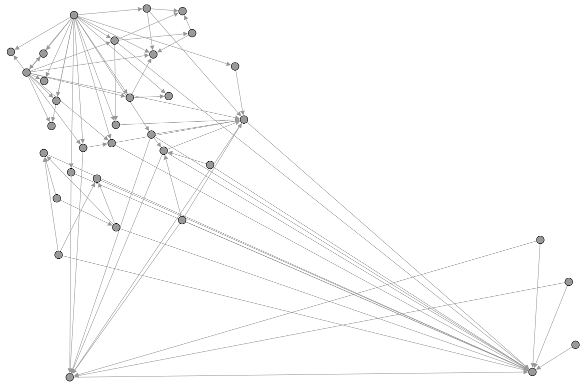


Figure 8: Partitioning example (1) of Zachary's Karate Club (Zachary 1977)

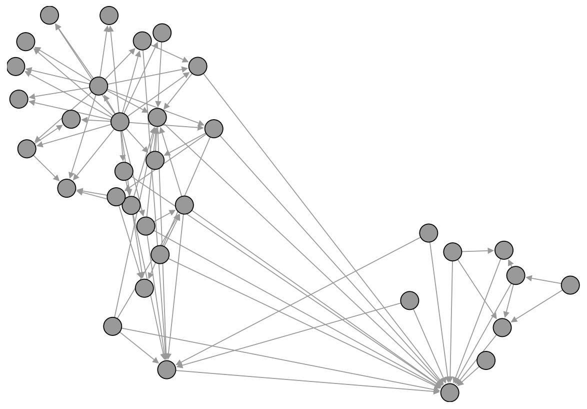


Figure 9: Partitioning example (2) of Zachary's Karate Club (Zachary 1977)

forward PageRank is generally impossible due to global influence of scores. However, as long as there is a partition that satisfies the boundary constraint, we can independently compute the PageRank of subgraphs including virtual nodes, and then uniformly rescale the scores of each partition so that the score of virtual node coincides and the sum of the ordinary (non-virtual) nodes of both partitions is equal to 1. Suppose the scores of virtual nodes are $s_{\mathrm{v}1}$ and $s_{\mathrm{v}2}$ for partition 1 and 2, respectively, then the scale factors $\alpha_1$ and $\alpha_2$ for each partition are

$$\alpha_1 = \frac{s_{\mathrm{v}2}}{s_{\mathrm{v}2}(1 - s_{\mathrm{v}1}) + s_{\mathrm{v}1}(1 - s_{\mathrm{v}2})}$$

$$\alpha_2 = \frac{s_{\mathrm{v}1}}{s_{\mathrm{v}1}(1 - s_{\mathrm{v}2}) + s_{\mathrm{v}2}(1 - s_{\mathrm{v}1})}$$

**Systematic Generation of Memo Entries** Memo entries like exemplified by Figure 6 may be generated in a systematic manner. Indeed, starting from graph (a) which is a minimum entry having only one node and no internal edge, we can obtain graph (b) by superimposing a cyclic graph $g$ with $g.\mathrm{V} = \{1, 2\}$ and $g.\mathrm{E} = \{(1, 2), (2, 1)\}$ and scores

$\{1 \mapsto 1, 2 \mapsto 1\}$, graph (c) by further superimposing the same graph $g$, graph (h) by splitting the node with score 2 in graph (b), graph (i) by shifting part of flow (=1) directing to the right to the upper node in graph (h), graph (j) by superimposing graph $g$ to graph (i), graph (b) by sequentially connecting graph (a) to itself, and so on. It is our future work to construct memo tables without using SMT solvers efficiently using these basic operations.

## 5 Graph Generation by Kronecker's Product

In this section, we show that could reverse iterative queries in a divide-and-conquer manner through finding a matrix satisfying an equation. As we know, many of the iterative computation problems such as PageRank, random walk and graph reachability are formalized as problems for finding an $n$-dimensional vector $\mathbf{v}$ satisfying $A\mathbf{v} = \mathbf{v}$ for a given $n \times n$ matrix $A$ that is an adjacency matrix of the input graph. Contrary, in our context where an appropriate input is expected to be generated from an output, we should find a solution that is a matrix $A$ satisfying $A\mathbf{v} = \mathbf{v}$ for a given vector $\mathbf{v}$. Note that the solution is not unique in general and an identity matrix is always a trivial solution. In the PageRank example, the trivial solution corresponds to a graph consisting of only self cycles, which is not desired for our purpose of data generation.

Our goal is to find a nontrivial $n \times n$ matrix $A$ satisfying $A\mathbf{v} = \mathbf{v}$ for a given $\mathbf{v}$. It is so hard in particular when $n$, the number of nodes of the graph, is very large. We give the solution as a Kronecker's product $A = A_1 \otimes \cdots \otimes A_k$ where $A_i$ $(i = 1, \ldots, k)$ has almost the same dimension so as to generate a portable and distributable data.

We design a divide-and-conquer algorithm to find a set of small matrices whose Kronecker's product is equivalent to a solution $A$ of $A\mathbf{v} = \mathbf{v}$. Suppose that there is a method for obtaining $m$-dimensional vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ from a $2m$-dimensional vector $\mathbf{v}$ such that $\mathbf{v}_1 \otimes \mathbf{v}_2 = \mathbf{v}$ holds. We can find $A_i (i = 1, 2)$ satisfying $A_i\mathbf{v}_i = \mathbf{v}_i$ with a recursive procedure. Then $A = A_1 \otimes A_2$ is found to satisfy $A\mathbf{v} = \mathbf{v}$ since we have

$$\begin{aligned} A\mathbf{v} &= (A_1 \otimes A_2)(\mathbf{v}_1 \otimes \mathbf{v}_2) \\ &= (A_1\mathbf{v_1}) \otimes (A_2\mathbf{v}_2) \\ &= \mathbf{v_1} \otimes \mathbf{v}_2 \\ &= \mathbf{v}. \end{aligned}$$

We may stop the recursive step at a threshold to switch the other algorithm for finding a small matrix.

The algorithm above cannot be complete to attain our goal, however. The problem is that there may be no vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ such that $\mathbf{v} = \mathbf{v}_1 \otimes \mathbf{v}_2$. For example, $\mathbf{v} = (1\ 2\ 2\ 4)^T$ has a solution $\mathbf{v}_1 = \mathbf{v}_2 = (1\ 2)^T$, while $\mathbf{v} = (1\ 2\ 3\ 4)^T$ and $\mathbf{v} = (1\ 2\ 4\ 2)^T$ have no solution. A possible approach for evading the problem is to try to minimize $\|\mathbf{v} - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$ where $\|\mathbf{x}\|_F$ denotes the Frobenius Norm of a vector $\mathbf{x}$. In the case of $\mathbf{v} = (1\ 2\ 3\ 4)^T$, we could choose $\mathbf{v}_1 \approx (0.946\ 2.138)^T$ and $\mathbf{v}_2 \approx (1.347\ 1.911)^T$ to have $\mathbf{v}_1 \otimes \mathbf{v}_2 \approx (1.274\ 1.808\ 2.880\ 4.086)^T$. Another approach for the problem is to allow to permute $\mathbf{v}$ before finding $\mathbf{v}_1$ and $\mathbf{v}_2$, that is, $\pi(\mathbf{v}) = \mathbf{v}_1 \otimes \mathbf{v}_2$ with a proper permutation $\pi$. In the case of $\mathbf{v} = (1\ 2\ 4\ 2)^T$, one may notice that there is a solution for its permutation $(1\ 2\ 2\ 4)^T$. The order can be arbitrary arranged in our purpose when the matrix represents an adjacent matrix

**Algorithm 2** Kronecker approximation for vectors

```
 1: procedure KroneckerApprox($\mathbf{v}$)
 2:     ▷ finds $\mathbf{v}_1$ and $\mathbf{v}_2$ s.t. $\pi(\mathbf{v}) \approx \mathbf{v}_1 \otimes \mathbf{v}_2$ with some $\pi$
 3:     $\mathbf{v}_2 \leftarrow$ a $|\mathbf{v}|/2$-vector whose all values are $2/|\mathbf{v}|$
 4:     repeat
 5:         $\mathbf{v}_1 \leftarrow \arg\min_{\mathbf{v}_1} \|\pi(\mathbf{v}) - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$
 6:                                             ▷ $\mathbf{v}_2$ and $\pi$ fixed
 7:         $\mathbf{v}_2 \leftarrow \arg\min_{\mathbf{v}_2} \|\pi(\mathbf{v}) - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$
 8:                                             ▷ $\mathbf{v}_1$ and $\pi$ fixed
 9:         $\pi \leftarrow \arg\min_{\pi} \|\pi(\mathbf{v}) - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$
10:                                             ▷ $\mathbf{v}_1$ and $\mathbf{v}_2$ fixed
11:     until $\|\pi(\mathbf{v}) - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$ converges
12:     return $(\mathbf{v}_1, \mathbf{v}_2)$
```

From these observations, we take an approximative approach in which we find vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ that minimize $\|\pi(\mathbf{v}) - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$ with varying permutation $\pi$. Algorithm 2 shows the procedure which gives a pair of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ that minimizes $\|\pi(\mathbf{v}) - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$ for a given $\mathbf{v}$ varying the permutation $\pi$. With $\pi$ fixed, we could solve the minimization problem by Singular-Value-Decomposition (SVD) approximation. Let $M_\mathbf{v}$ be a matrix such that the vector $\mathbf{v}$ is obtained by stacking the columns of $M_\mathbf{v}$. To minimize $\|\mathbf{v} - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$ for $\mathbf{v}$, an SVD $U_1 \Sigma U_2^T$ of the matrix $M_\mathbf{v}$ with diagonal matrix $\Sigma$ whose (1,1)-component $\sigma_{11}$ is a maximum entry gives a solution $(\mathbf{v}_1, \mathbf{v}_2) = (\sigma_{11}\mathbf{u}_1, \mathbf{u}_2)$ where $\mathbf{u}_i$ is a vector obtained by stacking the columns of $U_i$ (Loan and Pitsianis 1993). The decomposition is known to be achieved by an iterative algorithm of linear procedures (Pitsianis 1997; Johns, Mahadevan, and Wang 2007) as follows:

```
repeat
    $\mathbf{v}_1 \leftarrow \arg\min_{\mathbf{v}_1} \|\mathbf{v} - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$      ▷ $\mathbf{v}_2$ fixed
    $\mathbf{v}_2 \leftarrow \arg\min_{\mathbf{v}_2} \|\mathbf{v} - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$      ▷ $\mathbf{v}_1$ fixed
until $\|\mathbf{v} - \mathbf{v}_1 \otimes \mathbf{v}_2\|_F$ converges
```

Algorithm 2 extends the iterative procedure by varying the permutation $\pi$. To find a better permutation, the same order of the kronecker product of fixed vectors is used.

Using Algorithm 2, it is easy to find a set of matrices derive an approximation $A \approx A_1 \otimes \cdots \otimes A_k$ by divide-and-conquer. Algorithm 3 shows the procedure which derives a matrix $A$ as a Kronecker's product such that $A\mathbf{v} = \mathbf{v}$ for a given vector $\mathbf{v}$. $|\mathbf{v}|$ denotes the dimension of the vector $\mathbf{v}$ and $\sigma$ is a threshold at which the graph generation algorithm is switched into the other precise but possibly inefficient algorithm *GraphGen* to restrain the error introduced by approximation. We may choose a constraint-solver-based algorithm shown in 2 as *GraphGen*. A virtual-node based algorithm shown in 4 is also a candidate though it is specific to PageRank.

## 6 Conclusion and future work

This paper reports our work in progress on reversing iterative queries for systematic generation of graphs. The key idea is to reduce graph generation as a reverses of a fixed point computation, and to make it scalable through parallelization based on the divide-and-conquer approach.

Quite a lot of work is to be done in the future. First, in this paper we consider as an input the complete specification of the problem, like complete list of scores for every nodes in case of PageRank. However some application may require only the stochastic profile of them, like top-$k$ or histogram of scores. We would like to make use of this relaxation to generate graphs more efficiently. Second, the Divide-and-conquer approach may repeatedly produce the same subgraph at different levels and branches of recursions. We have discussed in Section 4.4 to use memoization to avoid re-computation, but the memo table entries themselves were constructed using SMT solvers. We could instead use combination of basic operations to construct these memo tables systematically. Finally, we should seriously implement our new graph generation method and evaluate it with practical graph generations.

## References

Aurenhammer, F. 1991. Voronoi diagrams&mdash;a survey of a fundamental geometric data structure. *ACM Comput. Surv.* 23(3):345–405.

Barrett, C.; Conway, C. L.; Deters, M.; Hadarean, L.; Jovanović, D.; King, T.; Reynolds, A.; and Tinelli, C. 2011. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, 171–177. Berlin, Heidelberg: Springer-Verlag.

Barrett, C.; Stump, A.; and Tinelli, C. 2010. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

Binnig, C.; Kossmann, D.; and Lo, E. 2007. Reverse query processing. In Chirkova, R.; Dogac, A.; Özsu, M. T.; and Sellis, T. K., eds., *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, 506–515. IEEE.

**Algorithm 3** Generating a matrix $A$ as a Kronecker's product

```
1:  procedure FIXEDPOINTINVERSION($\mathbf{v}$)
2:                              ▷ generates a matrix $A$ s.t. $A\mathbf{v} = \mathbf{v}$
3:      if $|\mathbf{v}| < \sigma$ then
4:          $GraphGen(s)$
5:              ▷ call the other algorithm for smaller vectors
6:      else
7:          $(\mathbf{v}_1, \mathbf{v}_2) \leftarrow$ KRONECKERAPPROX($\mathbf{v}$)
8:                      ▷ so that $\mathbf{v}_1 \otimes \mathbf{v}_2 \approx \pi(\mathbf{v})$ with some $\pi$
9:          $A_1 \leftarrow$ FIXEDPOINTINVERSION($\mathbf{v}_1$)
10:         $A_2 \leftarrow$ FIXEDPOINTINVERSION($\mathbf{v}_2$)
11:         return $A_1 \otimes A_2$
```

Capota, M.; Hegeman, T.; Iosup, A.; Prat-Pérez, A.; Erling, O.; and Boncz, P. A. 2015. Graphalytics: A big data benchmark for graph-processing platforms. In Larriba-Pey, J., and Willke, T. L., eds., *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*, 7:1–7:6. ACM.

De Moura, L., and Bjørner, N. 2008. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, 337–340. Berlin, Heidelberg: Springer-Verlag.

Desikan, P. K.; Pathak, N.; Srivastava, J.; and Kumar, V. 2006. Divide and conquer approach for efficient pagerank computation. In *Proceedings of the 6th International Conference on Web Engineering*, ICWE '06, 233–240. New York, NY, USA: ACM.

Dutertre, B. 2014. Yices 2.2. In Biere, A., and Bloem, R., eds., *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, 737–744. Springer.

He, B.; Yang, M.; Guo, Z.; Chen, R.; Su, B.; Lin, W.; and Zhou, L. 2010. Comet: batched stream processing for data intensive distributed computing. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, 63–74.

Johns, J.; Mahadevan, S.; and Wang, C. 2007. Compact spectral bases for value function approximation using kronecker factorization. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 1*, AAAI'07, 559–564. AAAI Press.

Leskovec, J.; Lang, K. J.; Dasgupta, A.; and Mahoney, M. W. 2008. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, 695–704. New York, NY, USA: ACM.

Loan, C. F., and Pitsianis, N. 1993. *Linear Algebra for Large Scale and Real-Time Applications*. Dordrecht: Springer Netherlands. chapter Approximation with Kronecker Products, 293–314.

McCune, R. R.; Weninger, T.; and Madey, G. 2015. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.* 48(2):25:1–25:39.

Nykiel, T.; Potamias, M.; Mishra, C.; Kollios, G.; and Koudas, N. 2010. MRShare: Sharing across multiple queries in mapreduce. *PVLDB* 3(1):494–505.

Onizuka, M.; Kato, H.; Hidaka, S.; Nakano, K.; and Hu, Z. 2013. Optimization for iterative queries on mapreduce. *PVLDB* 7(4):241–252.

Pitsianis, N. P. 1997. *The Kronecker Product in Approximation and Fast Transform Generation*. Ph.D. Dissertation, Cornell University, Ithaca, NY, USA. UMI Order No. GAX97-16143.

Zachary, W. W. 1977. An information flow model for conflict and fission in small groups. *Journal of Anthropological Research* 33(4):452–473.