Aggregated Search over Personal Process Description Graph

Jing Ouyang Hsu[1], Hye-young Paik[1], Liming Zhan[1], and Anne H. H. Ngu[2]

[1] University of New South Wales, Sydney, NSW, Australia
{jxux494, hpaik, zhanl}@cse.unsw.edu.au
[2] Texas State University, Austin, Texas, USA
angu@txstate.edu

Abstract. People share various processes in daily lives on-line in natural language form (e.g., cooking recipes, "how-to guides" in eHow). We refer to them as *personal process descriptions*. Previously, we proposed Personal Process Description Graph (PPDG) to concretely represent the personal process descriptions as graphs, along with query processing techniques that conduct exact as well as similarity search over PPDGs. However, both techniques fail if no single personal process description satisfies all constraints of a query. In this paper, we propose a new approach based on our previous query techniques to query personal process descriptions *by aggregation* - composing fragments from different PPDGs to produce an answer. We formally define the *PPDG Aggregated Search*. A general framework is presented to perform aggregated searches over PPDGs. Comprehensive experiments demonstrate the efficiency and scalability of our techniques.

1 Introduction

People are engaged in all kinds of *processes* all the time, such as cooking a dish, or filing a tax return. Although the area of business process management (BPM) [5] has produced solutions for modelling, automating and managing many of the business organizational workflows, still significant portion of the processes that people experience daily exist outside the realm of these technologies.

These experiences are often shared on the Web, in the form of how-to guides or step-by-step instructions. Although these are primarily describing a workflow, without the formal modelling expertise, they are written in natural language. To distinguish these texts from the conventional organizational workflow models, we refer to them as *personal process descriptions*. Many examples of personal process descriptions are found in cooking recipes, how-to guides or Q&A forums.

The natural language texts are not precise enough to be useful in utilizing the process information presented in them. For example, the state-of-the-art for search technologies over the existing personal process descriptions are still keyword/phrase-based and users would have to manually investigate the results.

In our previous work [18], we proposed a simple query language designed to perform exact-match search over the personal process descriptions. The language is supported by a graph-based, light-weight process model called PPDG (Personal Process Description Graph) which concretely represents the personal process description texts. We further extended our query technique to return *similar* process descriptions to a query input in [7]. Using these techniques, we can perform a *process-aware* search over PPDGs such as showing dependencies between data and actions.

However, when there is no single PPDG in the repository that satisfies all constraints in a query, these techniques cannot produce an answer. To overcome this limitation, we present a new approach to querying PPDGs, which can still produce an answer when a single PPDG cannot satisfy all query constraints. This technique, *Query By Aggregation*, involves decomposing a query into subqueries, matching multiple fragments over different PPDGs. The answer to a query is then generated by composing these fragments according to ranking criteria. This approach allows the user not only to better utilise existing process information in the PPDG repository, but also to discover and reuse process fragments to compose his/her own processes. We summarise our contributions below:

- We formally define the *PPDG Aggregated Search.*
- We present a general framework to perform an aggregated search over PPDGs, including: (i) a query decomposition algorithm to break down the query into two categories of subqueries - constant query and anonymous query, (ii) a tri-level index scheme based on our previous search techniques [7, 18] to reduce the search cost, and (iii) a ranking method to aggregate the matched fragments to obtain the closest query answers.
- We perform comprehensive experiments to demonstrate the efficiency and scalability of our techniques.

The paper is organized as follows: Section 2 defines the preliminary concepts and the problems. Section 3 shows the query decomposition algorithm. Section 4 discusses the algorithms to process the subqueries and aggregated search over the PPDG repository. Then we present the experiment results in Section 5. The related work is discussed in Section 6 followed by a conclusion in Section 7.

2 Preliminaries

In this section, we briefly introduce the PPDG and general querying of PPDGs. The full descriptions of these concepts are presented in [18] and [7], respectively. We then present an improved query technique named *PPDG Aggregated Search.*

2.1 Personal Process Description Graph (PPDG)

A PPDG represents a personal process description as a labeled directed graph. It describes the whole process of performing a personal process placing *equal emphasis* on both actions and input/output data associated with each action. The process on the right of Figure 1 depicts a PPDG of the ceremony day process experienced by a graduate. The circle nodes denote actions and data elements are represented by hexagonal nodes. In order to simplify the visualization of the graph, different types of actions/data are represented using the same notation. The details are stored in the schema associated with each PPDG.

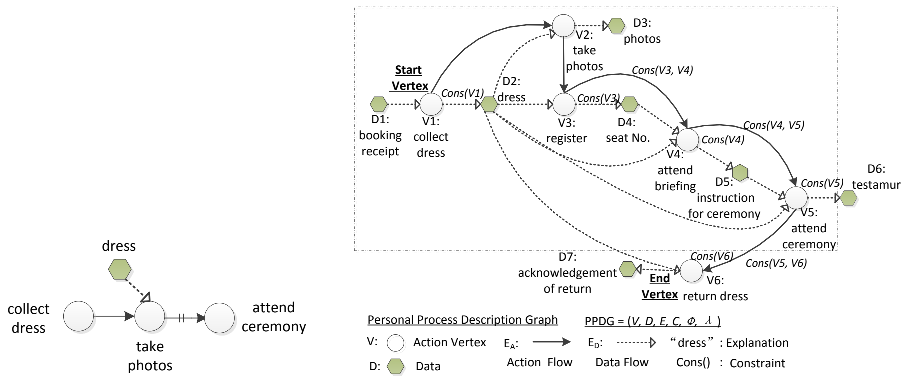

Fig. 1. An example of PPDG query input and output (an exact match)

The data elements and actions are connected to form 'action flow' and 'data flow'. Action flows, represented by solid lines, describe temporal sequence of the actions. For example, in the process on the right of Figure 1, 'V_3: *register*' takes place before 'V_4: *attend briefing*'. Data flows, represented by dotted lines, keep track of data sources and denote the relationships between the data and actions. For example, 'V_5: attend ceremony' takes two data inputs 'D_2: *dress*' and 'D_5: *instruction for ceremony*' and produces one data output 'D_6: *testamur*'.

A PPDG also stores constraints/conditions relating to an action, data or the flows. For example, a condition may specify a location or the time an action can take place. However, our current query techniques do not consider constraints in PPDG yet. For simplicity, we remove them from PPDG from here on. We define PPDG more formally as follows.

Definition 1. *A personal process description graph PPDG is a tuple* $PPDG := (A, D, E_A, E_D, \phi, \lambda)$ *where:*

- *A is a finite set of nodes* a_0, a_1, a_2,... *depicting the starting action* (a_0) *and actions* (a_1, a_2, ...).
- *D is a finite set of nodes* d_0, d_1, d_2,... *depicting the data input/output of an action.*
- E_A *is a finite set of directed action-flow edges* ea_1, ea_2,..., *where* $ea_i = (a_j, a_k)$ *leading from* a_j *to* a_k ($a_j \neq a_k$) *is an action-flow dependency. It reads* a_j *takes place before* a_k. *Each node can only be the source/target of at most one action-flow edge :* $ea = (a_i, a_j) \in E_A : ea' = (a_k, a_l) \in E_A \setminus ea : a_i \neq a_k$ *and* $a_j \neq a_l$.
- E_D *is a finite set of directed data-flow edges* ed_1, ed_2,..., *where* $ed_i = (a_j, d_k)$ *leading from* a_j *to* d_k *is a data-flow dependency. It reads* a_j *produces* d_k. $ed_l = (d_m, a_n)$ *leading from* d_m *to* a_n *is a data-flow dependency. It reads* a_n *takes* d_m.
- ϕ*: a function that maps Action Label to action nodes.*
- λ*: a function that maps Data Label to data nodes.*

We proposed a template-based query technique in [18], in which three types of query template constructs were defined: *atomic*, *path*, and *complex.* A PPDG query graph is defined as follows.

Definition 2 (PPDG Query Graph). *A PPDG query graph is a tuple* $PPDG\text{-}Q = (QA, QD, QE_A, QE_D, Q_\phi, Q_\lambda, QP, \Delta)$ *where:*

- QA *is a finite set of action nodes in a query.*

- *QD is a finite set of data nodes in a query.*
- *$QE_A \subseteq QA \times QA$ is the action flow relation between action nodes in a query.*
- *$QE_D \subseteq QA \times QD$ is the data flow relation between action nodes and data nodes in a query.*
- *Q_ϕ: a function that maps Action Label to action nodes.*
- *Q_λ: a function that maps Data Label to data nodes.*
- *QP is the path relation between action nodes which includes data nodes and data edges corresponding to each action node in query.*
- *Δ: $QP \rightarrow \{true, false\}$*

Figure 1 shows an example of PPDG query input and output[3]. A PPDG query graph on the left of Figure 1 consists of "*collect dress*" action, immediately followed by "*take photos*" action with input data "*dress*", followed by a *path query* edge leading to "*attend ceremony*" action. The edge with symbol "||" (also called *path edge*) represents a *path query* between action nodes "*take photos*" and "*attend ceremony*". In the above example, the path will match any action as well as connected data nodes from "*take photos*" to "*attend ceremony*". The subgraph inside dotted box on the right of Figure 1 is the result of the query.

However, in practice, performing an exact-match search over PPDGs may have limited utility because the PPDGs describing the same or similar processes are likely to have action/data labels and action/data flows expressed differently. Therefore, we also proposed the similarity search technique in [7] to expand the query results to include subgraphs that are similar to the query graph.

In this paper, we improve the query further by solving the problem of assembling fragments from different PPDGs to build an answer to a user query when a single PPDG cannot satisfy all the query constraints. Our goal is to support diversified forms of user queries and provide flexible ways to integrate or reuse information in PPDGs on demand.

2.2 Querying PPDG by Aggregated Search

Aggregated search is the task of searching and assembling information from a variety of sources, placing it into a single interface [10, 8]. Our approach to PPDG aggregated search is based on the notion of graph aggregation problem [9], and aggregated search problem in BPMN[4] models [12]. In both of them, the answer of a query graph can be represented as aggregation of fragments from different processes which are stored in the process repositories. In our work, we define aggregated search based on PPDG as follows.

Definition 3 (PPDG Aggregated Search). *Given a PPDG query q and a set of PPDGs $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, the problem of PPDG aggregated search is to find a subset $\mathcal{S} = \{P'_1, P'_2, \ldots, P'_m\}$ of $\mathcal{P} (m \leq n)$ and join their fragments $f_{P'_1}, f_{P'_2}, \ldots, f_{P'_m}$ to obtain a set of ranked PPDGs $\mathcal{R} = \{R_1, R_2, \ldots, R_k\}$, where each R matches the query q. That is, for each $R \in \mathcal{R}$, $R = f_{P'_1} \bowtie f_{P'_2} \bowtie \ldots \bowtie f_{P'_l} \mid l \leq m$ where f is a fragment (subgraph) of $P \in \mathcal{S}$.*

[3] We assume that the user can enter a query directly through our system by using an editor (similar to BPMN-Q editor [11]).

[4] Business Process Model and Notation, www.bpmn.org

We take the PPDG query shown in Figure 2 as an example. It describes a user query where the user wants to know what to do *before* booking the academic dress online and what needs to be done *between* getting the dress and attending ceremony. The prefix symbol “@” in the node label indicates an anonymous node (i.e. “@D” for an anonymous data node, “@V” for an anonymous action node).

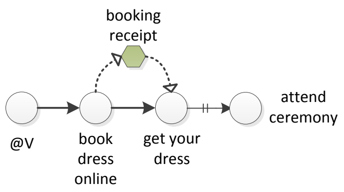

Fig. 2. Example of PPDG Query

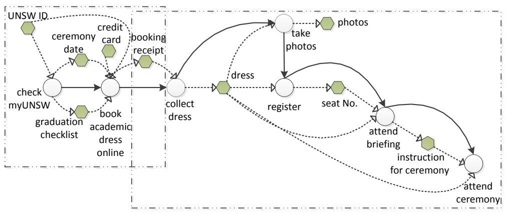

Fig. 3. Answer of PPDG Aggregated Query

Assume that the PPDG repository consists of two personal processes shown in Figure 4. Both of these two processes fail to match the said query if we matched it with each PPDG separately. There is no information about getting dress or attending ceremony in the process on the left of Figure 4. Similarly, the process on the right of Figure 4 does not mention booking dress online.

In the aggregated search approach, we decompose the original query into subqueries, match them individually against the PPDGs in the repository, and aggregate the results to form the answers. The answer of the said query over the two sample PPDGs is shown in Figure 3.

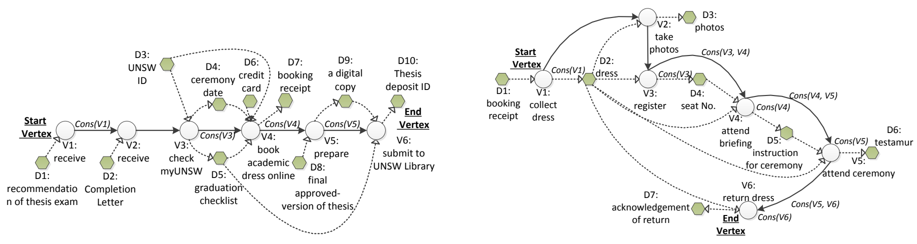

Fig. 4. Two PPDGs to Query

3 PPDG Query Decomposition

We can decompose a PPDG query into a set of subqueries which are classified into two categories as follows.

- A series of constant queries $\mathcal{Q}_c$: where each query has two constant (i.e., explicitly named) nodes connected by a direct flow edge.
- A series of anonymous queries $\mathcal{Q}_a$: where each query has (i) an anonymous node (i.e., unnamed node), or (ii) two constant nodes connected by a *path* edge.

Algorithm 1: ***Decomposition Query*** **(Q)**

```
Input  : a PPDG Query $Q$
Output: a series of Constant Queries $\mathcal{Q}_c$, a series of Anonymous Queries $\mathcal{Q}_a$
1  $L$ ←a list of action nodes in $Q$; $pos = 0$;
2  while $L \neq \phi$ do
3      $n_{action}$ ←the first node of $L$; $pos = pos + 1$;
4      if $n_{action}$ is a constant node then
5          for each data nodes $n_{data}$ of $n_{action}$ do
6              $type$ ←data type("D-in"/"D-out");
7              if $n_{data}$ is a constant node then
8                  $\mathcal{Q}_c \leftarrow \{n_{action}, n_{data}, type, pos\}$;
9              else
10                 $\mathcal{Q}_a \leftarrow \{n_{action}, n_{data}, type, pos\}$;
11         if $n_{action}$ has next action node $n'_{action}$ in $L$ then
12             if $n'_{action}$ is a constant node then
13                 if the connection is a path edge then
14                     $type$ ←action type("P");
15                 else
16                     $type$ ←action type("C");
17                 $\mathcal{Q}_c \leftarrow \{n_{action}, n'_{action}, type, pos\}$;
18             else
19                 $type$ ←action type("C-A");
20                 $\mathcal{Q}_a \leftarrow \{n_{action}, n'_{action}, type, pos\}$;
21     else
22         for each data nodes $n_{data}$ of $n_{action}$ do
23             $type$ ←data type("D-in"/"D-out");
24             $\mathcal{Q}_a \leftarrow \{n_{action}, n_{data}, type, pos\}$;
25         if $n_{action}$ has next action node $n'_{action}$ in $L$ then
26             $type$ ←action type("A-C");
27             $\mathcal{Q}_a \leftarrow \{n_{action}, n'_{action}, type, pos\}$;
28     Remove the first node of $L$;
29 return ($\mathcal{Q}_c$, $\mathcal{Q}_a$);
```

Unlike traditional graph, PPDG is a directed graph with two types of nodes and edges, so it is not straightforward to deploy the general decomposition methods to solve our problem. We fully decompose PPDG into atomic fragments below, because the size of most personal processes are not very big. To decompose the query Q into two series of subqueries $\mathcal{Q}_c$ and $\mathcal{Q}_a$, we traverse the query Q from the first action node to the last action node. For each action node n_{action}, we obtain its connected data nodes n_{data} and next action node n'_{action} (if have) to compose subqueries. Each subquery is represented as a tuple - $\{n_{action}, n_{data}\}$ or $\{n_{action}, n'_{action}\}$. Then we classify them into either $\mathcal{Q}_c$ or $\mathcal{Q}_a$. Each subquery can be further classified into the following six types: "C" or "P" - two nodes are both constant action nodes with a Constant edge or Path edge; "C-A" or "A-C" - one node is a Constant action node and the other is an Anonymous action node; "D-in" or "D-out" - one node is a Data node and the data flow to action node is "input" or "output", so we keep a *type* attribute in the subquery. The position *pos* of n_{action} in Q is stored in the subquery to keep the series information. Therefore, a subquery is defined as a tuple $\{n_{action}, n'_{action}$ or $n_{data}, type, pos\}$, and the action node n_{action} is the key of the tuple.

Algorithm 1 illustrates the details of decomposition process. We first put all action nodes of Q into a sorted list L in Line 1. From Line 2 to Line 28, we traverse the action nodes n_{action} in L one by one. We use a variable *pos* to store the position of n_{action}. If n_{action} is a constant node, we obtain all its connected data nodes from Q. If the data node n_{data} is also a constant node, we put the subquery $\{n_{action}, n_{data}, type, pos\}$ into $\mathcal{Q}_c$ in Line 8, otherwise $\mathcal{Q}_a$ in Line 10. Similarly, if the next connected action node n'_{action} for n_{action} exists, we check n'_{action} and put the subquery into corresponding subquery series ($\mathcal{Q}_c$ or $\mathcal{Q}_a$) in Lines 11-20. Note that if the edge between two action nodes is a *path* edge, we set the *type* as "P" in Line 14. If n_{action} is an anonymous node, we obtain its all connected data nodes and the next action node (if have) to compose the subqueries and put them into $\mathcal{Q}_a$ in Lines 22-27. After traversing all action nodes, we gain two series of queries $\mathcal{Q}_c$ and $\mathcal{Q}_a$.

4 PPDG Query Processing

After query decomposition, we use the two series of queries to find matched PPDG fragments and aggregate them to obtain the results. The framework of the aggregated search processing is presented as follows:

- ***Constant Query*:** We use each constant query in $\mathcal{Q}_c$ to perform similarity search on PPDGs to get the constant fragments with corresponding PPDGs,
- ***Anonymous Query*:** We use anonymous queries in $\mathcal{Q}_a$ based on the results in *Constant Query* phase to find the matched fragments,
- ***Aggregating*:** We rank and aggregate fragments to obtain the answers ordered by ranking score.

The experiments of our own proposed search techniques to process graph query over PPDGs showed that the main cost of the query processing is on matching the nodes between the query and PPDGs. Therefore, we designed an indexing technique to speed up the node matching process as part of the framework.

In what follows, we first present the details of the indexing, then we discuss using the index to obtain the matched fragments of constant queries and anonymous queries in Section 4.2 and Section 4.3, respectively. Finally, we propose a novel rank method to assemble the fragments to obtain the answer in Section 4.4.

4.1 Indexing PPDGs

In [7], we match two nodes in PPDGs by comparing their label similarity. For one node in the query, we obtain the word set from its label and calculate the similarity score between this word set and that of each node in a PPDG. The PPDG index has three levels "word(L1)-word set(L2)-PPDG(L3)", which is shown in Figure 5. All the PPDGs entries are stored in L3. The word sets from

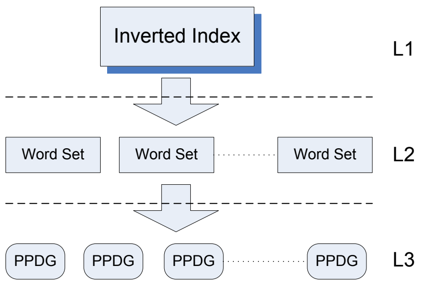

Fig. 5. Structure of PPDG Index

each PPDG are extracted and stored in L2. Each word set entry points to the PPDG that it originates from. We cluster those word sets which have common words by choosing one of them as the center. The similarity between each word set and the center is more than a given correlation radius η. All the word sets in one cluster can be merged to construct a new word set entry in L2, which points to multiple PPDGs. L1 is an inverted index. When a word is given, we can use the inverted index to find the corresponding word set entry in L2.

Algorithm 2: ***Constant Queries Search*** **($\mathcal{Q}_c$, $\mathcal{P}$)**

```
Input  : a series of Constant Queries $\mathcal{Q}_c$, a set $\mathcal{P}$ of PPDGs
Output: $\mathcal{Q}_c$ filled with matched fragments
for each $q \in \mathcal{Q}_c$ do
    $n = q.n_{action}$;
    Get word set of $n$ and obtain all matched PPDG graphs $\mathcal{P}_m$ by index;
    if $q.type$ is "C" then
        $n' = q.n'_{action}$;
        Get word set of $n'$, and obtain all matched PPDG graphs $\mathcal{P}'_m$ by index;
        $\mathcal{P}''_m = \mathcal{P}_m \cap \mathcal{P}'_m$;
        for each $P \in \mathcal{P}''_m$ do
            $f \leftarrow$ matched fragment;
            $score$ = similarity score between $q$ and $f$;
            $q.results \leftarrow \{f, score, P\}$;
    else
        $n' = q.n_{data}$;
        for each connected data node $n_d$ in each $P \in \mathcal{P}_m$ do
            if $n_d$ matches $n'$ then
                $f \leftarrow$ matched fragment;
                $score$ = similarity score between $q$ and $f$;
                $q.results \leftarrow \{f, score, P\}$;
return $\mathcal{Q}_c$;
```

4.2 Processing Constant Queries

To process constant queries, we launch our similarity query search proposed in [7] for each constant query to obtain the *similar* fragments. There is a small cost to do one similarity search. Since there are many constant queries generated for aggregated search, the total cost could be quite high. Therefore, we use the index built in Section 4.1 to reduce the query processing time.

Algorithm 2 illustrates the details of how to use the PPDG index to process the similarity search of the constant queries. We match each query q in the constant query set $\mathcal{Q}_c$ to process the similarity search over PPDG. For each query result, we record the fragment f, the similarity score between f and q, and the corresponding PPDG P, which comprise a result tuple $\{f, score, P\}$. Recall that the query q is represented as a tuple in Section 3. Here we add an attribute $results$ to q to store all the result tuples of q. For each constant query q, we put its key node n_{action} into n, and get the word set of n to obtain the matched PPDGs $\mathcal{P}_m$ by index in Lines 2- 3. Then we check the type of q in Line 4. If the second node n' is an action node, we use the similar method to get its matched PPDGs $\mathcal{P}'_m$ by index in Lines 5- 6. The two matched PPDGs $\mathcal{P}_m$ and $\mathcal{P}'_m$ are joined to obtain a new PPDG set $\mathcal{P}''_m$ containing both action nodes n and n'. We get matched fragment from each $P \in \mathcal{P}''_m$ and put the result tuple into $q.results$ from Line 8 to 11. If the second node n' is data node, we check each connected data node in $P \in \mathcal{P}_m$ to obtain the matched fragment from Line 13 to 18. Note that the edge direction must be matched when matching data node in Line 15. The fragment results are also put into $q.results$ in Line 18. After traversing all constant queries, the result is returned in Line 19.

4.3 Processing Anonymous Queries

Each anonymous query has at least one known action node, so we can use the same similarity matching technique in Section 4.2 to obtain the PPDGs set containing the node, and then find the matched anonymous node or path. According to the decomposition technique in Section 3, an anonymous query contains at least one constant node, which exists in a constant query, and we have already processed the constant node in Algorithm 2. These results can be stored in a map M. Then we search M first when the constant node in an anonymous query is given. If we cannot find the matched node in M, the normal index lookup is invoked to find the related PPDGs.

Algorithm 3 illustrates the details of anonymous query processing. Like Algorithm 2, we also use the result tuple $\{f, score, P\}$ to store the query answer. For each anonymous query q, we put its key node n_{action} in n, and get the related PPDGs $\mathcal{P}_m$ from map M or index in Lines 2-6. Note that n must be a constant node according to Section 3. Then we treat the anonymous part of q by the query type *q.type* separately from Line 7 to Line 24. If the anonymous node is an action node, we find the next/previous connected action node n' in $\mathcal{P}_m$, and put the query result tuple into q, which is shown in Lines 7-11. From Line 12 to Line 18, we process the case that the anonymous exists on edge (path query). The matched PPDGs $\mathcal{P}'_m$ of the second action node n' is gotten and joined with $\mathcal{P}_m$ to obtain the total matched PPDGs $\mathcal{P}''_m$ in Lines 14-15. Then we get the path from n to n' as the fragment result. Note that we must keep the order of the action series, which means n must in the front of n'. If the anonymous node is a data node, we find the connected data node n' in $\mathcal{P}_m$, and put the query result tuple into q, which is shown in Lines 20-24. After traversing all anonymous queries, the results are returned in Line 25.

Algorithm 3: *Anonymous Queries Search* ($\mathcal{Q}_a$, M, $\mathcal{P}$)

```
   Input  : a series of Anonymous Queries $\mathcal{Q}_a$, Map $M$, a set $\mathcal{P}$ of PPDGs
   Output: $\mathcal{Q}_a$ filled with matched fragments
1  for each $q \in \mathcal{Q}_a$ do
2      $n = q.n_{action}$;
3      if $\exists n \in M$ then
4          $\mathcal{P}_m = M(n)$;
5      else
6          $\mathcal{P}_m \leftarrow$ related PPDGs by query $n$ on index;
7      if q.type is "C-A" or "A-C" then
8          for each next connected action node $n'$ of $n$ in each $P \in \mathcal{P}_m$ do
9              $f \leftarrow$ matched fragment;
10             $score$ = similarity score between $n$ and corresponding node in $f$;
11             $q.results \leftarrow \{f, score, P\}$;
12     else if q.type is "P" then
13         $n' = q.n'_{action}$;
14         Obtain $\mathcal{P}'_m$ by searching $M$ or index;
15         $\mathcal{P}''_m = \mathcal{P}_m \cap \mathcal{P}'_m$;
16         $f \leftarrow$ matched fragment;
17         $score$ = similarity score between $q$ and $f$;
18         $q.results \leftarrow \{f, score, P\}$;
19     else
20         for each connected data node $n_d$ in each $P \in \mathcal{P}_m$ do
21             if the direction of $n_d$ matches q.type then
22                 $f \leftarrow$ matched fragment;
23                 $score$ = similarity score between $q$ and $f$;
24                 $q.results \leftarrow \{f, score, P\}$;
25 return ($\mathcal{Q}_a$);
```

4.4 Aggregating Fragments

After obtaining all matched fragments, we need to assemble them to obtain the required PPDGs. There are many ways to combine the fragments, therefore, we need to rank the possible aggregated results efficiently and recommend the user a list of aggregated PPDGs in an descending order by "score". In this section, we explain how to calculate such a score.

When we decompose the aggregated query into subqueries, the position of each query is kept, so we can assemble the aggregated result from each query tuple according to the position of the query. When we process the constant query or the anonymous query, we store the similarity *score* between the query and the fragment result. Intuitively, we could choose fragments with the highest score from each query tuple, and aggregate them to obtain the query result. Then the similarity score (SS) of a result R is calculated as follows.

$$SS(R) = \prod q.score(f) \quad (1)$$

where $q.score(f)$ represents the similarity score of a selected fragment f in R.

There may be some fragments coming from the same PPDG. The fragments from the same PPDG are preferred because intuitively they would form more coherent PPDG when put together, so we give the case a higher rank. For each possible aggregated result, we count the number n of fragments originated from each related PPDG. Then the similarity score of a result R can be adjusted as follows, which is called adjusted similarity score (ASS).

$$ASS(R) = \prod(\prod_{i=1}^{n} q.score(f_i) \times C^{n-1}) \quad (2)$$

where C ($C \geq 1$) represents a weighting factor for a scoring fragment and n is the number of fragments which come from the same PPDG containing f. Note that if the factor C is set to 1, ASS degenerates into SS.

5 Experiments

Now we present the results of a performance study to evaluate the efficiency and scalability of our proposed techniques. Following algorithms are evaluated.

- **Baseline:** Aggregated search by techniques proposed in [7].
- **INDEX:** Algorithms proposed in Section 4 by using the index.

Datasets We have evaluated our aggregated search techniques on both synthetic and real datasets.

The synthetic datasets are generated by randomization techniques. We create a word set containing 100 different words. Then we randomly choose n action nodes and $[0, 2n]$ data nodes to assemble p process graph. For each node, w words are randomly selected to make the label. After a process graph is built, we make several small changes, such as changing the labels of nodes and adding/deleting nodes, to obtain 99 similar process graph. The number p varies from $2K$ to $50K$ (default value = $10K$). The number n of action nodes in each process is randomly chosen in a range varying from $[5, 10]$ to $[35, 40]$ (default value = $[15, 20]$). The number of data nodes is randomly chosen in $[0, 2n]$. For each node, there are up to w words selected randomly. The w varies from 10 to 25 (default value = 15). By the default setting, the total number of nodes is up to $600K$ in our experiment. The cluster correlation radius of the index η varies from 0.2 to 0.8 (default value = 0.6). We select 100 process graphs and get their subgraphs to make 100 queries in our experiment. Some nodes in each query are randomly set as anonymous nodes. The size s of query, i.e. number of action nodes in query, varies from 3 to 9 (default value = 5). The average processing time of the 100 queries on each dataset represents the performance of our query processing mechanism. The factor C is set to 2 in all experiments.

The real dataset consists of 42 PPDGs about PhD programs collected from the Web and manually created by the authors. The dataset includes personal process descriptions on processes such as research degree admission, scholarship applications, and attending graduation ceremony. In this dataset, the queries are chosen manually.

All algorithms are implemented in C++ and compiled by Cygwin GCC 4.3.4. The experiments are conducted on a PC with Intel i7 2.80GHz CPU and 8G memory on Windows 7 Professional SP1.

Performance Evaluation

We evaluate the performance of our technique INDEX comparing with Baseline algorithm in the experiment.

Performance Tuning. The performance of our techniques is effected by the index. Especially, The correlation radius η between the data and the center of clusters in the index impacts the processing time of our algorithm. As expected, Figure 6 shows the processing time drops when η increases, because the wordsets with similarity η are clustered in one index entry. On the other hand, if the η drops, the index degenerates. When η is equal to 0.2, there is no significant improvement between the algorithm INDEX and the algorithm Baseline. We notice that the performance of INDEX does not change much when η increases from 0.6 to 0.8, therefore, we use 0.6 as the default setting of η in the following part.

Real vs Synthetic. we evaluate the performance of our techniques over the real and synthetic data. Due to the limited quantity of real process graphs, we magnify the result on the real data by 200 times in Figure 7. It is shown that our techniques give the similar performance on both datasets, and the index is very effective and reduces the processing time.

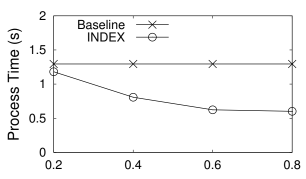

Fig. 6. Index Tuning

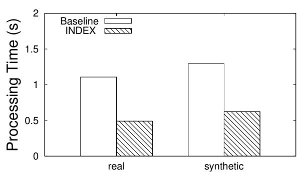

Fig. 7. Real vs Synthetic

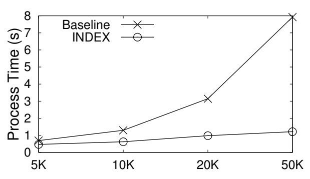

Fig. 8. Varying p

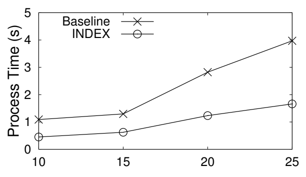

Fig. 9. Varying w

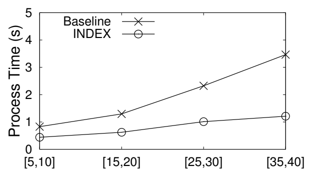

Fig. 10. Varying n

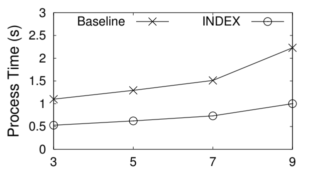

Fig. 11. Varying s

Evaluating Impacts by Different Setting. We study the scalability of our algorithms with regards to the different number of process graphs (p), number of words (w) in one graph node, number of action nodes (n), and the query size (s) in Figure 8-11. The processing time increases with the increase of the four parameters. The results also demonstrate that the indexing technique is effective and reduces the processing time in all settings. Clearly, the dataset size increases with the number of process graphs and action nodes thus the aggregated search processing becomes a little expensive. Longer word size makes it difficult to process the node matching, which increases the processing time. When the query size grows, the processing cost increases because more subqueries are involved in the aggregated search.

Aggregate Output. Finding a way to systematically measure the quality of possible aggregates for a given query is still an open research issue [8] and is one of our immediate future work plans. In this paper, we evaluate the average output size of our techniques compared with similarity search techniques in [7]. It is clear that the output size of the two approaches increases when the number of process graphs (p) rises as demonstrated in Figure 12. On the other hand, Figure 13 shows the output size of the two approaches drops when the number of words (w) increases, because it is harder to match the nodes when more words are involved in the labels. The two figures show regardless of the parameter changes, the aggregate approach outputs about three times more results than similarity approach. We also compare the similarity score between the results of aggregate search and similarity search, and it shows the score of top ranked answer in aggregate search is much higher than the one in similarity search. Therefore, our approach can give more recommendations to users.

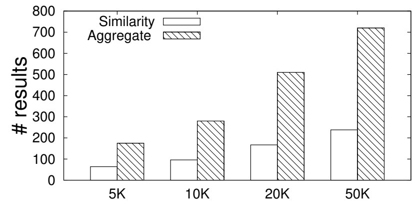

Fig. 12. Varying p

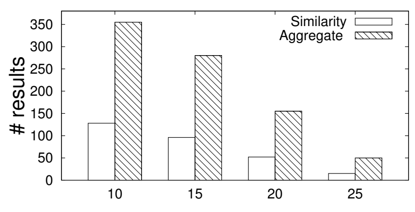

Fig. 13. Varying w

6 Related Work

We discuss related work broadly in two categories: business process querying and general graph querying approaches.

In the domain of Business Process Management (BPM), there are existing works that deal with querying business processes. In these work, queries are processed over BPMN (Business Process Modelling Notation) or equivalent notations. The main purpose of query processing is to extract actions (i.e., control flows). For example, the Business Process Query Language (BPQL) in [3] works on an abstract representation of BPEL[5] files, which focuses on querying actions only. The BPMN-Q is a visual language to query repositories of process models [1]. It processes the queries by matching a process model graph, converted from BPMN, to a query graph [11]. In [4], the authors describe the problem of retrieving process models in the repository that most closely resemble a given

[5] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

process model. At present, the authors have focused on developing the similarity metrics rather than efficient implementation of algorithms. In [15], the authors presented a survey on current approaches to querying business process repositories. However, the paper did not discuss how resources associated with tasks in business process can be queried.

In the graph querying domain, graph similarity search have received considerable attention. This includes Closure-Tree [6], K-AT [14], and SEGOS [17]. In general, subgraph similarity search is to retrieve the data graphs that approximately contain the query. Grafil [19] defines graph similarity as the number of missing edges in a maximum common subgraph. GrafD-index [13] defines similarity using maximum connected common subgraph. [20] studies the problem of graph similarity search with edit distance constraints. One of the closely related work to ours is Cooking Graphs [16]. A cooking graph describes a cooking process with cooking actions and relevant ingredients information. However, cooking graphs are tailored to recipes. The focus of Cooking Graph is to apply a graph mining technique on recipes to recognize cooking process patterns (frequent subgraphs) and recommend a suitable cooking recipe for a user. Our PPDG querying technique aims to provide a platform to support various analysis tasks on graph structure, not limited to just recommendation.

None of the above business process querying or graph querying approaches addresses aggregated search on graphs. The first definition of aggregated search was given in [10]. Many studies [8] from different IR domains developed their own aggregated search approaches based on this definition. However, to the best of our knowledge, there are few studies that address querying a process repository by aggregation. That is, decomposing a query graph into several fragments, querying the repository with each of the fragments, and constructing a new process by assembling all matched fragment results from different processes. A similar work presented in [2] is based on the notion of partial process model which describes a desired model through a combination of process model fragments and process model queries. However, they do not address querying of both control and data flow. In [9], authors propose the exact subgraph matching approach of assembling graphs to provide answers to a given query graph if no single candidate graph is isomorphic with the query. Another aggregated graph search paper [12] introduces a novel approach for querying and reusing knowledge contained in business process models repositories, which presents the solution for the similar subgraph matching. Due to the structure of PPDG and flexible attribute of personal processes, the above two approaches are not suitable for applying directly to query PPDG repositories.

7 Conclusion

In this paper, we have investigated aggregated search over Personal Process Description Graph (PPDG). We formally define the PPDG aggregated search and propose a novel approach based on our previous query techniques to query personal process descriptions by aggregation. We first decompose the PPDG query into two categories of subqueries - constant query and anonymous query, and then obtain the matched fragments from different PPDGs according to the decomposed subqueries. Afterward, the matched fragments are assembled to form the closest query answers by a ranking method. Furthermore, a tri-level index is built in our approach in order to accelerate the processing of decomposed subqueries and reduce the search cost. Finally, a comprehensive experimental study over both real and synthetic datasets demonstrates the efficiency and scalability of our techniques.

References

1. A. Awad. BPMN-Q: A language to query business processes. In *The 2nd International Workshop on EMISA*, pages 115–128, 2007.
2. A. Awad, S. Sakr, M. Kunze, and M. Weske. Design by selection: A reuse-based approach for business process modeling. In *ER*, pages 332–345, 2011.
3. C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *PVLDB*, pages 343–354, 2006.
4. R. Dijkman, M. Dumas, B. F. van Dongen, R. Käärik, and J. Mendling. Similarity of business process models: Metrics and evaluation. *Inf. Syst.*, 36(2):498–516, 2011.
5. M. Dumas, M. La Rosa, J. Mendling, and H. Reijers. *Fundamentals of Business Process Management.* Springer-Verlag Berlin Heidelberg, 2013.
6. H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, page 38, 2006.
7. J. O. Hsu, H. Paik, and L. Zhan. Similarity search over personal process description graph. In *Web Information Systems Engineering - WISE 2015, Proceedings, Part I*, pages 522–538, 2015.
8. A. Kopliku, K. Pinel-Sauvagnat, and M. Boughanem. Aggregated search: A new information retrieval paradigm. *ACM Comput. Surv.*, 46(3):41:1–41:31, 2014.
9. T. Le, H. Elghazel, and M. Hacid. A relational-based approach for aggregated search in graph databases. In *DASFAA*, pages 33–47, 2012.
10. V. Murdock and M. Lalmas. Workshop on aggregated search. *SIGIR Forum*, 42(2):80–83, 2008.
11. S. Sakr and A. Awad. A framework for querying graph-based business process models. In *WWW*, pages 1297–1300, 2010.
12. S. Sakr, A. Awad, and M. Kunze. Querying process models repositories by aggregated graph search. In *BPM Workshops*, pages 573–585, 2012.
13. H. Shang, X. Lin, Y. Zhang, J. X. Yu, and W. Wang. Connected substructure similarity search. In *SIGMOD*, pages 903–914, 2010.
14. G. Wang, B. Wang, X. Yang, and G. Yu. Efficiently indexing large sparse graphs for similarity search. *IEEE Trans. Knowl. Data Eng.*, 24(3):440–451, 2012.
15. J. Wang, T. Jin, R. K. Wong, and L. Wen. Querying business process model repositories - A survey of current approaches and issues. *World Wide Web*, 17(3):427–454, 2014.
16. L. Wang. *CookRecipe: towards a versatile and fully-fledged recipe analysis and learning system.* PhD thesis, City University of Hong Kong, 2008.
17. X. Wang, X. Ding, A. K. H. Tung, S. Ying, and H. Jin. An efficient graph indexing method. In *ICDE*, pages 210–221, 2012.
18. J. Xu, H. Paik, A. H. H. Ngu, and L. Zhan. Personal process description graph for describing and querying personal processes. In *Databases Theory and Applications - 26th Australasian Database Conference, ADC 2015*, pages 91–103, 2015.
19. X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.
20. X. Zhao, C. Xiao, X. Lin, W. Wang, and Y. Ishikawa. Efficient processing of graph similarity queries with edit distance constraints. *VLDB J.*, 22(6):727–752, 2013.