Personal Process Description Graph for Describing and Querying Personal Processes

Jing Xu[1] Hye-young Paik[1] Anne H. H. Ngu[2]

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

Technical Report
UNSW-CSE-TR-201501
February 2015

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Unlike business processes which are template driven, personal processes are ad-hoc to the point where each personal process may have a unique structure and is certainly not as strictly defined as a business process. In order to describe, share and analyze personal processes more effectively, in this paper, we propose Personal Process Description Graph (PPDG) for describing personal processes. Based on the proposed model, a personal process query approach is developed to support different types of graph queries in a personal process graph repository. The approach follows a filtering and refinement framework to speed up the query computation. We conduct some experiments on real and synthetic datasets to demonstrate the efficiency of our techniques.

1 Introduction

People are often confronted with tasks that are infrequent (even one-off) and not well-described. Examples are applying for jobs, buying a house, or filing a tax return. We refer to such tasks as "personal processes" in which an individual performs a set of logically related tasks to accomplish some personal goals [9, 20]. Unlike business processes which are structured or template driven, personal processes are ad-hoc to the point where each personal process may have a unique structure and is certainly not as strictly defined as a business process.

To overcome their lack of familiarity, people typically seek out other people's experiences with the task. We realize that it is difficult and time-consuming to capture and share personal processes, although we have many kinds of channels to share variety of information such as photos or video. Furthermore, searching and understanding already shared processes are limited to a simple keyword search via search engines. We see the following problems in existing publicly available personal process repositories such as how-to sites or Q/A forums.

- With regards to capturing a personal process, the current model of describing processes, which is based on free texts and bullet points, makes it difficult to (1) understand the process especially if there are many steps involved; (2) compare and contrast different paths/ways to accomplish the goal; and (3) understand the dependencies between data and actions.
- For searching and analyzing a personal process, the current model of keyword search over the textual descriptions is limited and is not able to provide exploratory answers to query. That is, the search always returns a set of Web pages containing the descriptions of the whole processes even if the user just requires a fragment of the process or a specific task in the process.

In order to describe, share and analyze personal processes more effectively, we propose a novel framework for personal process management named ProcessVidere. Our ultimate aim is to provide a space for users to share their experiences, capture the process knowledge from people, analyze them effectively, and support users to re-use the whole or part of the processes for their own purposes. In this paper, as a concrete step towards building ProcessVidere, we present the following elements as the fundamental components of the system:

- Personal Process Description Graph (PPDG), a graph-based description language to present both control flow and data flow of a personal process.
- A template-based approach to perform structured queries over PPDG, as well as an implementation of the key template queries and preliminary performance evaluation results.

The paper is organized as follows: Section 2 introduces the PPDG language. Section 3 presents the basic design of PPDG query templates. Section 4 describes methods and algorithms for an efficient processing of the query templates. 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 Personal Process Description Graph (PPDG)

In this section, we introduce the syntax and formal definition of PPDG as well as motivation behind it. Let us consider "attending a graduation ceremony at UNSW". We interviewed six UNSW graduates and asked them to describe the process. A collective summary of the process is as follows:

> *Before the graduation day, a graduand may book a graduation gown online. The booking can be paid for by card (in which case a receipt is issued), or by cash when collected. On the day, the graduand may collect the dress, take a photo and register for the ceremony to obtain a seat number. After a briefing session, the graduand attends the ceremony. The dress is returned after the ceremony.*

Not every step is strictly followed, in fact, six different variations of this personal processes are obtained. A typical process modelling approach (as discussed in Section 6) would attempt to create a single reference model that describes the above process as complete as possible. However, building such a model for a flexible and ad-hoc process is difficult and often makes the model convoluted.

In ProcessVidere, instead of a single model, we consider a personal process as process steps experienced and described by a single person. Each description of the process becomes a feasible process, the so-called "model", that leads to the intended goal. For this reason, we intentionally do not use the word 'model' in our approach. PPDG described below is designed to capture the description of an individual experience of a process. Later in the paper, we will explain how queries are written over these descriptions to give a flexible view of the process.

2.1 PPDG Syntax Overview

Here we will briefly explain the syntax and visual notations before presenting formal definitions. PPDG represents a personal process P as a labelled directed

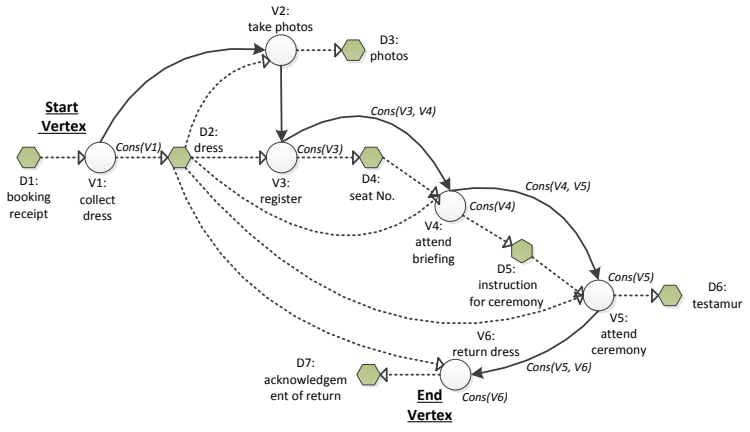


Figure 2.1: PPDG1: UNSW graduation ceremony process by Graduand A

graph. It describes the whole process of performing a personal process placing equal emphasis on both actions and input/output data related to each action

in the process. Figure 2.1 depicts a PPDG of the graduation ceremony process experienced by Graduand A. An action is noted by a circular node in the graph. Actions can be divided into two categories: a simple action and composite action which contains another process. A simple action can be one-off action, repeated action, or duration action. In order to make the visualization of the graph simple, all types of actions are represented using the same notation. The details are stored in the schema associated with the process graph.

Data elements in a process P are represented by hexagonal nodes. A data element can be either basic or composite (i.e., composition of basic data). In PPDG, data could be any artifacts that is available somewhere: physical items such as paper documents, academic dress, or digital items such as digital photos, text messages and so on. A data element can be external (i.e., it is not produced by any action) or processed (i.e., there is at least one action produced it).

The data elements and actions are connected to represent 'action flow' or 'data flow'. Action flows, represented by solid lines, describe temporal sequence of the actions. For example, in Figure 2.1, 'V_1: *collect dress*' takes places before 'V_2: *take photos*'. Data flows, represented by dotted lines, keep track of data sources and denotes the relationships between the data and actions. For example, 'V_4:attending briefing' takes two data inputs 'D_2: *dress*' and 'D_4: *seat no.*' and produces one data output 'D_5: *instructions for ceremony*'.

PPDG also stores constraints/conditions relating to action, data and the flows. If the constraints concern an action (referred to as 'action constraints'), it may include conditions such as the location or the time the action takes place. If the constraints are on the connecting edges of two actions (referred to as 'transition constraints', it may specify the conditions that should be met for the flow to take place. It can also be used to enforce the sequence of the two actions. We define PPDG more formally as follows.

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

- A *is a finite set of nodes* a_0, a_1, a_2,... *representing the starting action node* (a_0) *and actions* (a_1, a_2, ...).
- D *is a finite set of nodes* d_0, d_1, d_2,... *representing the data input and 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 as* 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 as* a_j *produces* d_k. $ed_l = (d_m, a_n)$ *leading from* d_m *to* a_n *is also a data-flow dependency. It reads* a_n *takes* d_m.
- C *is a finite set of conditions* c_1, c_2,... *with* $c_i = (< name, descr >, x_j)$ *being associated to* $x_j \in \{A, D, E_A, E_D\}$ *and having name and description of the condition.*
- ϕ: *a function that maps Label* $L(A_i)$ *to action nodes.*

- λ: *a function that maps Label* $L(D_i)$ *to data nodes.*

2.2 Constructing PPDG

The PPDGs presented here are manually transcribed from the textual description data we have gathered from students who experienced the process. Currently, we are developing a graphical tool, PPDG Editor, which will help construct a PPDG from textual "how-to" descriptions semi-automatically. In a similar fashion described in [16], a part-of-speech (POS) tagger can be utilized to recognize potential pairs of (action and data) (e.g., (book, academic dress), (pay, cash)). The editor provides graphical notations for the syntax elements in PPDG. Although we do not assume that the PPDG construction process can be totally automated, having an editor with smart natural language processing ability highly customized for PPDG syntax will help reduce the manual labour.

3 Querying in PPDG

A repository of PPDG is organized by categories and domains. The idea in ProcessVidere is that this repository can be simply keyword searched or browsed, but it can also be used to perform more structured queries for sophisticated analysis. Such queries can be issued over a single PPDG or a set of PPDGs relating to the same category.

3.1 Types of Query and Query Processing in PPDG

Taking the graduation ceremony as an example, the types of queries fall into two categories:

The Complete Picture Queries

This category of queries aims to return all the whole processes (i.e., a set of PPDGs) matching the query criteria. The main intention here is to gain an overall understanding about the process. An example complete picture query is "How to attend a graduation ceremony at UNSW?"

In answering this category of queries, we expect three possible cases: single match, similarity match and aggregation match. In the single match case, the above query will return a set containing all PPDGs that exactly match the query criteria. In the similarity match, the above query will return a ranked set of PPDGs that are considered 'similar' to the given query. The similarity measure may be determined by considering semantics of the text labels in the nodes and/or structure of the graphs. In the aggregation match case, the above query will return "the best" PPDG, possibly combining features from multiple PPDGs. The criteria for determining the best PPDG could depend on many factors such as shortest time taken to complete the process, or situational context of the person issuing the query.

The Fragments Queries

This category of queries aims to return all the partial processes matching the query criteria. The main intention is to obtain information about particular

aspects of the process. Some example fragment queries are: "What happens after attending the briefing session?", "How do I get a seat no.?", "I have collected the dress, what can I do with it?".

The complete picture queries can give you all the details of a process. However, typically a user will deal with many PPDGs returned as an answer, and manually examining PPDG individually is time consuming. Moreover, this may not be a suitable option if the user's query is more specific than the overall picture. Working out the specific details might be trivial in a simple process, but in many of the personal processes we examined, the flexible nature of the process can create quite complicated descriptions. More structured and fine-grained queries (which we refer to as fragments queries) will allow the users to obtain relevant information quicker and analyze the given processes in-depth.

Let us consider one example query: 'What do I need to do to attend the briefing'. In answering this, we expect three possible cases: single match, similarity match and aggregation match. In single match, the above query will return a set containing all fragments matching the query criteria. In each fragment, we will see a data node ($\in D$) (e.g., D_2:*dress*), an action node ($\in A$) (i.e., V_8:*attend briefing*) and the data-flow edge ($\in E_D$) between them. From Figure 2.1, the data items required for attending the briefing are '*dress*' and '*seat no.*'. Considering the fact that each PPDG describes a specific person's experience, the result of this query performed over the repository will give the user *all* data items known to have been associated with the action in question. That is, if another PPDG had '*student card*' linked with the action '*attend briefing*', this query will reveal it.

In the similarity match, the above query will return a ranked set of fragments containing all fragments that are considered 'similar' to the given query. In this ranked approach, if the '*student card*' is infrequently used, the fragment containing '*student card*' will have a lower score. In the aggregation match, the above query will return "the best" fragment, possibly combining features from multiple fragments. The criteria for determining the best fragment could depend on many factors such as frequency of a fragment, or situational context of the person issuing the query.

We believe in both categories, all three type of matches are necessary and will complement each other. In Section 3.2, as the first step, we propose a technique for the single match approach. We envisage that this basic set of query constructs will allow us to build more complex query approaches in our immediate future work to implement the similarity (e.g., the semantic mismatch issues in the text labels) or aggregation match. For simplicity, we remove C (conditions) from PPDG here on.

3.2 Query Templates and Their Basic Constructs

In this section, we present the basic design of the single match approach. We propose a template-based approach in which three types of query template constructs are defined: *atomic*, *path* and *complex* query templates. We also assume that the user can enter one of the query templates directly through ProcessVidere, using an editor not dissimilar to BPMN-Q query editor [13].

Figure 3.1 introduces another PPDG on graduation ceremony. Although it looks similar to Graduand A process (Figure 2.1), there are a few differences. For example, 'V_1*: collect dress*' takes 'D_1*: cash*' and produces 'D_3*: deposit*'.

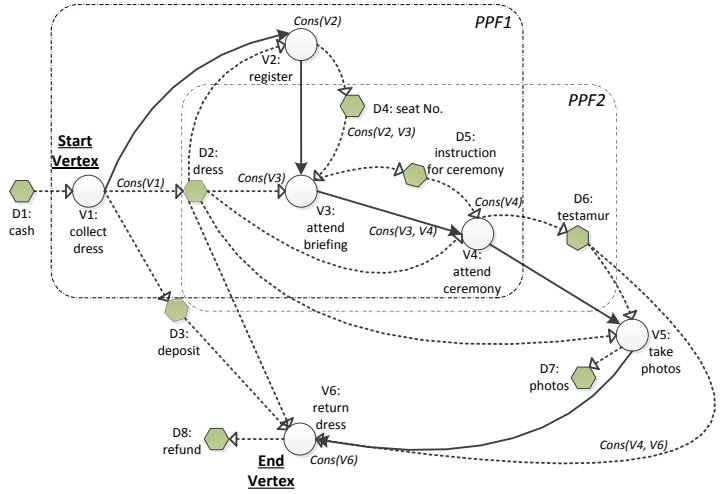


Figure 3.1: PPDG2: UNSW graduation ceremony process by Graduand B

'V_6*: return dress*' takes 'D_3*: deposit*' as well as the dress. Also, '*take photo*' appears after '*attend ceremony*' (rather than before '*register*').

Atomic Query Template

Atomic query templates are designed to match an action-flow edge ($\in E_A$) or a data-flow edge ($\in E_D$) and the nodes ($\in A, D$) that are directly connected by it. Q_1 to Q_6 in Figure 3.2 represent atomic query templates. The text label

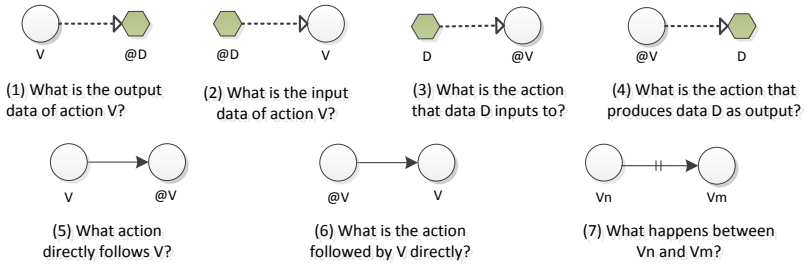


Figure 3.2: Atomic Query and Path Query Templates over PPDG

shown in each template describes the query contained in it. The prefix symbol "@" in the node label indicates a variable node (i.e., "@D" for a variable data node, "@V" for a variable action node).

Example 1 *Let us assume that we issue Q_2 with v as V_1: collect dress. The results (processed over PPDG1 and PPDG2) will be $\{PPDG1.ed_1$ = (D_1: booking receipt, V_1: collect dress), $PPDG2.ed_1$ = (D_1: cash, V_1: collect dress)$\}$. This indicates that collect dress can either take cash or a booking receipt.*

Example 2 *Similarly, let us assume that we issue Q_5 with v as V_1: collect dress. The results (processed over PPDG1 and PPDG2) will be $\{PPDG1.ea_1$*

= (V_1: collect dress, V_2: take photos), $PPDG2.ea_1$ = (V_1: collect dress, V_2: register)}. This indicates that collect dress can be followed by either taking photos or registering.

Path Query Template

Intuitively, a path query template is designed to match a fragment of PPDG contained by two action nodes. Visually, we denote the template as shown in Figure 3.2 (Q_7). The symbol "$\|$" is used to represent a path query between action nodes specified by V_n and V_m (where n, m are the action node numbers). To explain, we first define a path in PPDG.

Definition 2 *A path of PPDG is a tuple $P_{ath} = (A', D', E'_A, E'_D, \phi, \lambda)$ where*

- $P_{ath} \subseteq PPDG$.
- $x \in A'$ *iff:*
  - $x = source$.
  - $x = destination$.
  - x *lies on a path from source to destination in* $ppdg \in PPDG$.
- $\forall x, y \in A', e(x, y) \in E_A \rightarrow e(x, y) \in E'_A$
- $d \in D'$ *iff:*
  - d *is output data of source action in a path.*
  - d *is input data of destination action in a path.*
  - d *is input/output data of action* x *which lies on a path from source to destination in* $ppdg \in PPDG$.
- $\forall d \in D', e(d, x) \in E_D \rightarrow e(d, x) \in E'_D$ *and* $e(x, d) \in E_D \rightarrow e(x, d) \in E'_D$

A path query template will return a single matching path from each PPDG considered for processing.

Example 3 *Let us assume that we set $n = 1$ and $m = 4$ in Q_7, so that Q_7 is a path query to find personal process fragment between 'V_1:* collect dress*' and 'V_4:* attend ceremony*'. Therefore, the returned result of Q_7 (processed over PPDG2) is a path shown as $PPF1$ (dotted box) in Figure 3.1.*

Complex Query Template

A complex query template is a composite of atomic and/or path query templates. Q_8 and Q_9 in Figure 3.3 are the samples of complex queries that are composed of Q_5 and Q_7 and a constant action, Q_2 and Q_7, respectively. The complex query of PPDG can consist of Q_1 to Q_6 or Q_1 to Q_6 and Q_7 or Q_1 to Q_7 with constant actions/data.

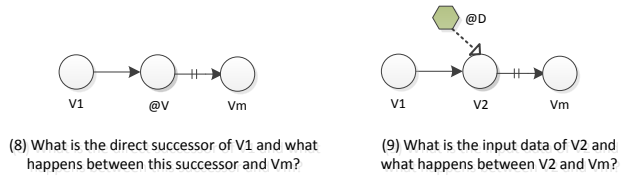


Figure 3.3: Complex Query Templates over PPDG

4 Query Processing

This section formally introduces the definitions of PPDG query and fragment, and then gives the schema of PPDG with detailed algorithms to perform three types of query template mentioned in Section 3.

Definition 3 (PPDG Query) *A query graph is a tuple* $PPDG - Q = (QA, QD, QAF, QADF, QP, T, S)$ *where*

- QA *is a finite set of action nodes in a query.*
- QD *is a finite set of data nodes in a query.*
- $QAF \subseteq QA \times QA$ *is the action flow relation between action nodes in a query.*
- $QADF \subseteq QA \times QD$ *is the data flow relation between action nodes and data nodes in a query.*
- QP *is the path relation between action nodes which includes data nodes and data edges corresponding to each action node in query.*
- T*: *$QA \rightarrow \{CONSTANT\ ACTION, VARIABLE\ ACTION\}$
- S*: *$QD \rightarrow \{CONSTANT\ DATA, VARIABLE\ DATA\}$

Definition 4 (PPDG Fragment (PPF)) *A connected subgraph* (A', D', E'_D)*, or* (A', E'_A)*, or* (A', D', E'_A, E'_D) *of a personal process description graph* (A, D, E_A, E_D) *, where* $A' \subseteq A$*,* $D' \subseteq D$*,* $E'_A \subseteq E_A$*,* $E'_D \subseteq E_D$*, is a fragment of the personal process description graph.*

A query template can return 0, 1 or more PPDG Fragments. Figure 3.1 shows examples of PPF in $PPF1$ (dotted box) and $PPF2$ (dotted box).

PPDG data schema

The database schema for our PPDG has the following tables:

- Graph(<u>ID</u>, Name, Description)
- Action(<u>ActionID, GraphID</u>, ActionLabel, Type1, Type2, SubProcess, Description)
- Data(<u>DataID, GraphID</u>, DataLabel, Type1, Type2, SubProcess, Artifacts)

- Edge_Action(sActionID, dActionID, GraphID)
- Edge_Data(DataID, ActionID, GraphID, Direction)

We store the descriptions of graphs, actions, and data in three tables, respectively. The relationships between actions and data are stored in two tables. Specially, the *Direction* item in table *Edge_Data* indicates whether the data is the input or output of an action.

Atomic Queries ($Q_1 \sim Q_6$)

Using the label of an action (data), we can get the *ActionID* (*DataID*) by matching the label in table *Action* (*Data*). Then the *IDs* of related actions and data can be derived from table *Edge_Action* and *Edge_Data*. After that, the label of the related actions and data can be found in table *Action* and *Data*. Each atomic query can be implemented by a single SQL statement. For instance, the results of Q_1 are obtained by the following SQL statement when giving $Q_1(ActionLabel, Direction = output)$.

```
SELECT d.DataID, d.GraphID, d.DataLabel
FROM query q, Action a, Data d, Edge_Data e
WHERE q.ActionLabel=a.ActionLabel and q.Direction=e.Direction
and e.ActionID=a.ActionID and e.GraphID=a.GraphID
and e.DataID=d.DataID and e.GraphID=a.GraphID;
```

The results of Q_5 are obtained by the following SQL statement when giving $Q_5(ActionLabel)$.

```
SELECT da.ActionID, da.GraphID, da.ActionLabel
FROM query q, Action sa, Action da, Edge_Action e
WHERE q.ActionLabel=sa.ActionLabel
and e.sActionID=sa.ActionID and e.GraphID=sa.GraphID
and e.dActionID=da.ActionID and e.GraphID=da.GraphID;
```

The other four atomic query templates are straightforwardly implemented with minor modifications of the SQL statements above.

Path Query (Q_7)

For the implementation of the path query template, intuitively, we can use the six atomic queries ($Q_1 \sim Q_6$) above to get the path step by step: (1) First, launch atomic query Q_5 from the first action node sA, and get a set $\mathcal{S}$ of action nodes that are direct successors of sA from different processes. (2) Then we perform the same atomic query for each action node in $\mathcal{S}$. (3) The same procedure is executed iteratively until we reach the dA or end of the process. The results are a set $\mathcal{A}$ of action paths. (4) For each action path in $\mathcal{A}$, we launch Q_1 and Q_2 to find the data nodes related to every action nodes. After that, the final results are obtained.

The cost of this initial algorithm is very high due to querying redundant processes which do not contain the end action node dA of the path. Therefore, we filter all the processes using the following SQL statement in order to obtain a small set $\mathcal{G}$ of processes which contain both sA and dA nodes before the above steps (1 to 4) are performed.

```
SELECT sA.GraphID
FROM query q, Action sA, Action dA
WHERE q.sActionLabel=sA.ActionLabel and q.dActionLabel=dA.ActionLabel
and sA.GraphID=dA.GraphID
```

Algorithm 1: *Path Query(sA, dA)*

```
   Input  : sA, dA: The start and end actions of the path
   Output: R: The result set of P_ath
 1 G ← all the processes contain both sA and dA;
 2 T = φ; T' = φ;
 3 for each graph ∈ G do
 4 |   T.graph=graph; T.queue.push(sA);
 5 |   T ← T;
 6 while T ≠ ∅ do
 7 |   for each T ∈ T do
 8 |   |   r=Q5(T.queue.back()) on T.graphID;
 9 |   |   if r ≠NULL then
10 |   |   |   T.queue.push(r);
11 |   |   |   if r==dA then
12 |   |   |   |   R ← T;
13 |   |   |   else
14 |   |   |   |   T' ← T;
15 |   T=T'; T' = ∅;
16 for each T ∈ R do
17 |   Find related data nodes of all action nodes in T by Q1 and Q2 → T.data;
18 return R;
```

The Algorithm 1 illustrates the details of path query processing. To enable computing the path in an iterative fashion, we use a tuple T to process the path query. T is employed to maintain the nodes information used for the path computation of a set of action and data nodes in a graph. Particularly, $T.graph$ indicates which graph the path locates in, $T.queue$ is a queue of action nodes on the path, and $T.data$ is a set of data nodes on the path. First, we filter the graph by the start and end actions of the path in Line 1. Then, Line 3-5 initialize T for each related graph and store them in a set $\mathcal{T}$. From Line 6 to Line 15, we find the actions on the path using the iterative method. Particularly, Line 8 launches Q_5 to find the next action node r of current action node in the tail of $T.queue$. If there is no result from Q_5, the T is pruned. Otherwise, we push r into $T.queue$ (Line 10). If r is dA, we get one exact result and put it into result set $\mathcal{R}$ (Line 12). If r is not dA, we continue to search for the next action (Line 14, 15). Finally, Line 16-17 launch Q_1 and Q_2 for each actions in $T \in \mathcal{R}$ to fill the data nodes for each path.

Complex Query

We decompose the complex query first, and then utilize methods of atomic and path queries mentioned above to find matched fragments/processes. The filtering and verification methods are used to improve the performance of the algorithm. To begin with, we split the complex query into one *node set* and four *pair sets*: Constant node set S_n, Constant pair set P_{con}, Variable pair set P_v, Path pair set P_{path} and Preprocessing pair set P_{pre}. S_n contains all constant nodes appearing in the query. In any of the four *pair sets*, *a pair* is described as $\{n, n'\}$ - n and n' are nodes linked by one edge or path in a graph. n or n'

Algorithm 2: ***Complex Query(Q)***

Input : Q: The complex query
Output: $\mathcal{R}$: The result set of the complex query

1 Decompose Q into nodes set S_n and pair sets $P_{con}, P_v, P_{path}, P_{pre}$;
2 $\mathcal{G} \leftarrow$ all the processes containing the nodes in S_n and matching the pairs in P_{con};
3 $\mathcal{C} = \phi$; $\mathcal{C}' = \phi$;
4 **for** *each graph* $\in \mathcal{G}$ **do**
5 $\quad T.graph$=$graph$; $T.R \leftarrow$ all nodes in S_n;
6 $\quad \mathcal{C} \leftarrow T$;
7 **for** *each pair* $\{n, n'\} \in P_v$ **do**
8 $\quad \mathcal{U} \leftarrow$ results of query $Q(n, n')$ on $\mathcal{G}$;
9 $\quad$ Join $\mathcal{U}$ and $\mathcal{C}$ on *graph* attribute; update $\mathcal{G}$;
10 $\quad$ **for** *each* $(U, T) \in \mathcal{U} \times \mathcal{C}$ **do**
11 $\quad\quad T.R \leftarrow U.r$;
12 $\quad\quad$ **if** *variable* $v \in P_{pre}$ *is identified by* $U.r$ **then**
13 $\quad\quad\quad T.Pair \leftarrow \{u,v'\}$ or $\{v',u\}$;
14 $\quad\quad \mathcal{C}' \leftarrow T$;
15 $\quad \mathcal{C}$=$\mathcal{C}'$; $\mathcal{C}'$=ϕ;
16 **for** *each pair* $\{n, n'\} \in P_{path}$ **do**
17 $\quad \mathcal{V} \leftarrow$ results of path query $Q(n, n')$ on $\mathcal{G}$;
18 $\quad$ Join $\mathcal{V}$ and $\mathcal{C}$ on *graph* attribute; update $\mathcal{G}$;
19 $\quad$ **for** *each* $(V, T) \in \mathcal{V} \times \mathcal{C}$ **do**
20 $\quad\quad T.R \leftarrow V.R$;
21 $\quad\quad \mathcal{C}' \leftarrow T$;
22 $\quad \mathcal{C}$=$\mathcal{C}'$; $\mathcal{C}'$=ϕ;
23 **for** *each* $T \in \mathcal{C}$ **do**
24 $\quad$ Process queries on all pairs in $T.Pair$;
25 $\quad \mathcal{R} \leftarrow T$ if all queries have results on T ;
26 **return** $\mathcal{R}$;

could be either a constant node or a variable node. We store the pairs with two constant action/data nodes linked by an edge in P_{con}. The pairs which can be processed directly by using atomic queries stored in P_v and those that can be processed by path query stored in P_{path}, respectively. And the pairs which need to be preprocessed before using the methods mentioned above are put into P_{pre}. That is, all the pairs in P_{pre} depend on the results from P_v to identify one node of each pair before using the methods of atomic queries or path query directly. We use S_n and P_{con} to find a candidate set $\mathcal{C}$ of graphs, and then prune and verify $\mathcal{C}$ by P_v, P_{path} and P_{pre}.

The details of the complex query processing is illustrated in Algorithm 2. Similar to Algorithm 1, we also use a tuple T to store the intermediate result. Particularly, $T.graph$ indicates which graph the path locates in, $T.R$ stores the known action/data nodes, and $T.Pair$ stores the pairs that exist in $T.graph$. Line 1 splits query Q into one nodes set and four pair sets. Then, we use S_n and P_{con} to filter all the processes to get a set $\mathcal{G}$ of processes in Line 2. Based on $\mathcal{G}$ and S_n, we get a candidate set $\mathcal{C}$ containing intermediate results T in Line 4-6. From Line 7 to 15, we process atomic queries on each pair in P_v, and

then use the results to prune and verify the candidate set. Then, we get the result set $\mathcal{U}$(Line 8) for each query. For each result $U \in \mathcal{U}$, $U.r$ is a result node on graph $U.graph$ for atomic query. Next, $\mathcal{U}$ and $\mathcal{C}$ are joined on their *graph* attribute(Line 9), so that the unjoined tuples in $\mathcal{C}$ can be pruned safely. If one variable node v of any pair in P_{pre} is identified by $U.r$, we replace the variable node v with $U.r$, and store the corresponding pair in $T.Pair$, because the pair only appears in $T.graph$(Line 10-13). From Line 16 to 22, we process path query on each pair in P_{path}, which can be used for further pruning. Finally, we process atomic query or path query on each pair in $T.Pair$ to obtain the complex query result set $\mathcal{R}$(Line 23-25).

Example 4 *To process query Q_8 in Figure 3.3 (set m=5), we split the query as S_n=$\{V_1, V_5\}$, P_{con}=$\{\phi\}$, P_v=$\{\{V_1, @V\}\}$, P_{path}=$\{\phi\}$, and P_{pre}=$\{\{@V, V_5\}\}$. Filtering by S_n and P_{con}, suppose we get the process in Figure 3.1, then we process the query on P_v by using $Q_5(V_1, @V)$ to get the result $\{V_1, V_2\}$. Note that V_2 matches the node in pair $\{@V, V_5\} \in P_{pre}$, so the new pair $\{V_2, V_5\}$ is stored in T.pair. P_{path} is empty and no further processing is necessary. Finally, we process the query on the pair $\in$ T.pair and assemble all results to get the fragment of graph.*

5 Performance Evaluations

In this section, we present results of a performance study to evaluate the efficiency and scalability of the proposed techniques in the paper.

Datasets We have evaluated our query processing techniques on both real and synthetic datasets. We invited 14 volunteers to describe the procedures they went through at Graduation Research School of UNSW, and generated 30 different processes with 295 action nodes and 441 data nodes. The synthetic datasets were generated by randomization techniques. By giving 30 action labels and 30 data labels, we randomly choose n action labels with $n + 2$ data labels to assemble p processes. The number p varies from 100 to 1000 (default value = 300). The number n of action nodes in each process varies from 10 to 25 (default value = 10). By the default setting, the total number of action nodes and data nodes were 3000 and 3600 respectively in our experiment. We had a total of 10 queries in the experiment. Each query was generated randomly. The average response time and I/O cost of the 10 queries on each dataset represent the performance of our query processing mechanism.

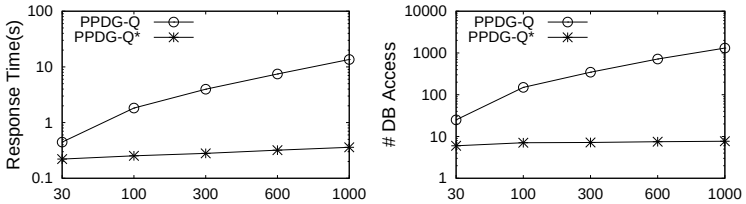
All algorithms have been implemented in Java and compiled with JRE 1.6. We used PostgreSQL to store the data. The experiments were run on a PC with Intel Xeon 2.40GHz dual CPU and 6G memory on Debian Linux.

We have measured the I/O performance of the algorithms by measuring the number of database access. Query response times were recorded to evaluate the efficiency of the algorithms, which contained the CPU time and the I/O latency.

Impact of the number of processes (p).

We varied the value of p and evaluated the performance of our algorithm against the real datasets where p is 30 and the synthetic datasets where p varies from 100 to 1000 in Figure 5.1. With a larger number of processes, more processes

are involved in the computation, thus incurring higher computation cost. From Figure 5.1(a) and Figure 5.1(b), we found the result of the response time and number of database access had the same increasing trend as the number of processes increased. That means I/O costs is the main contribution to the query processing cost. Due to the high cost of database access, we can optimize the technique of PPDG query (denoted PPDG-Q here) proposed in Section 4 by storing the actions node of $graph \in \mathcal{G}$ in cache. This optimization is represented as PPDG-Q* in our experiments. As expected, PPDG-Q* significantly outperforms PPDG-Q and is less sensitive to the growth of p.

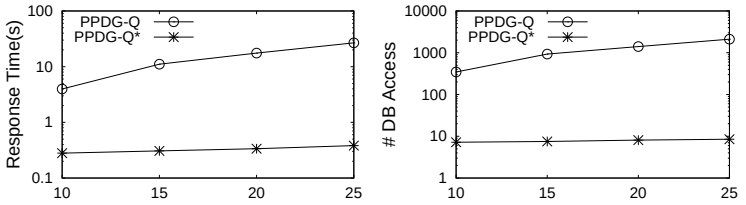


(a) *Response Time* (b) *number of DB Access*

Figure 5.1: Number of Process (p)

Impact of the number of action nodes in each process (n).

Figure 5.2 investigates the impact of the number of action nodes in each process on the algorithms where n grows from 10 to 25. With the growth of n, more action/data nodes are involved in the query computation, thus the response time and the number of database accesses increase. The results show that the scalability of PPDG-Q* is better than that of PPDG-Q regarding the growth of n.



(a) *Response Time* (b) *number of DB Access*

Figure 5.2: Action Nodes of Process (n)

6 Related Work

We discuss related work broadly in two categories: process modelling languages and process query approaches.

Modelling Processes. There are many models developed to represent business processes, Petri Net [17], YAWL [18] and Pi-Calculus [15] being some of

the examples. Recently, Business Process Model and Notation (BPMN) [1] has emerged as the *defacto* standard for business process modeling [13]. In all of these models, however, the main focus is in formalizing action flows (i.e., control flow dependencies) in a process. They tend to provide highly expressive and sophisticated constructs which make them a complicated tool to use for business process management. In [6] and [20], arguments are made for a modelling approach that reduces expressive power and simplifying notations for personal processes. There are some work that takes the simplification view. For example, in [10], a personal workflow management system is proposed using a model called *Metagraph*. It places the data flow at the center of the process, describing input/output of each task and the connections between the data. However, the model becomes very complicated and it is difficult to comprehend the temporal execution sequence of the actions. The cooking graph, proposed in CookRecipe [19] system, not only captures individual actions but also the diverse relationship between the actions. However, it is not suitable for the more generic scenarios of personal processes. [6] proposes a model that simplifies BPMN constructs to only include actions and sequential and parallel action flows. This model does not consider data items in the process.

Our work shares the same view that personal processes do not require complicated expressive power, for example, a parallel execution has no practical meaning when a process is viewed from a single person (i.e., single executor) view point. We also argue that describing data items explicitly is just as important as describing actions to support a useful set of analysis techniques over the personal process model. Therefore our work puts equal emphasis on both. This is a point that is often overlooked in all modelling languages, even the ones that are designed for personal processes such as the work mentioned above. Another salient point of our approach is to take the view that, rather than creating a master model that encompasses all possible scenarios of a process, creating a repository of individual scenarios and using a suite of query and analysis techniques to understand the process as required will lead to better support of the flexible nature of personal processes.

Querying Process Models. Graph query processing has been intensively studied by the database community in recent years and many approaches (e.g. [8], [12], [7], [21]) have been proposed to deal with different types of graph queries. Particularly, querying business process models is one of these applications and has attracted significant attention (e.g. [5], [3], [4], [14]) from academic researchers.

The Business Process Query Language BPQL in [5] works on an abstract representation of BPEL [2] files, which cannot be applied to PPDG proposed in our paper due to the inherent difference of the models. The BPMN-Q query language is a visual language to query repositories of process models which extends BPMN notations with very few additional constructs to serve its query purpose [3]. It is used to query business process models by matching a process model graph to a query graph [13]. While our PPDG describes personal processes from individual's view and uses a novel query paradigm which takes both actions and data into consideration. The approach presented in [4] is based on the notion of partial process models which describe a desired model through a combination of process model fragments and process model queries. Complex

query of PPDG has similar structure as partial process models, but the approach in [4] can not be used to perform PPDG queries because PPDG queries need to process both control and data flows. In [11], 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 [14] 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 graph repositories.

7 Conclusion

In this paper, we present Personal Process Description Graphs (PPDG) for describing personally experienced processes. A template-based query approach is proposed to support different types of graph queries in PPDG repository. Our experiments demonstrate the efficiency of this query approach.

PPDG is the first step towards providing a solution to sharing the process knowledge through a personal process repository, querying and analyzing personal processes and reusing the processes (either as a whole or fragments). Our immediate future work includes improving the query processing algorithms by introducing more punning rules, and utilizing other types of DBMS, *e.g.* graph database, further developing the algorithms to perform similarity and aggregation matches in PPDG repository. This is part of our on-going work for developing ProcessVidere to evaluate the feasibility and practicality of PPDG and its query approaches as the foundations of personal process management.

Bibliography

[1] `http://www.bpmn.org/`.

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

[3] Ahmed Awad. Bpmn-q: A language to query business processes. In *EMISA*, pages 115–128, 2007.

[4] Ahmed Awad, Sherif Sakr, Matthias Kunze, and Mathias Weske. Design by selection: A reuse-based approach for business process modeling. In *Conceptual Modeling - ER 2011, 30th International Conference, ER 2011, Brussels, Belgium, October 31 - November 3, 2011. Proceedings*, pages 332–345, 2011.

[5] Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying business processes. In *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 343–354, 2006.

[6] Marco Brambilla. Application and simplification of bpm techniques for personal process management. In *Business Process Management Workshops*, pages 227–233, 2012.

[7] Remco M. Dijkman, Marlon Dumas, Boudewijn F. van Dongen, Reina Käärik, and Jan Mendling. Similarity of business process models: Metrics and evaluation. *Inf. Syst.*, 36(2):498–516, 2011.

[8] Rosalba Giugno and Dennis Shasha. Graphgrep: A fast and universal method for querying graphs. In *16th International Conference on Pattern Recognition, ICPR 2002, Quebec, Canada, August 11-15, 2002.*, pages 112–115, 2002.

[9] Seyed Alireza Hajimirsadeghi, Hye-Young Paik, and John Shepherd. Processbook: Towards social network-based personal process management. In *Business Process Management Workshops*, pages 268–279, 2012.

[10] San-Yih Hwang and Ya-Fan Chen. Personal workflows: Modeling and management. In *Mobile Data Management*, pages 141–152, 2003.

[11] Thanh-Huy Le, Haytham Elghazel, and Mohand-Said Hacid. A relational-based approach for aggregated search in graph databases. In *Database Systems for Advanced Applications - 17th International Conference, DASFAA 2012, Busan, South Korea, April 15-19, 2012, Proceedings, Part I*, pages 33–47, 2012.

[12] Sherif Sakr and Ghazi Al-Naymat. Efficient relational techniques for processing graph queries. *J. Comput. Sci. Technol.*, 25(6):1237–1255, 2010.

[13] Sherif Sakr and Ahmed Awad. A framework for querying graph-based business process models. In *WWW*, pages 1297–1300, 2010.

[14] Sherif Sakr, Ahmed Awad, and Matthias Kunze. Querying process models repositories by aggregated graph search. In *Business Process Management Workshops - BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers*, pages 573–585, 2012.

[15] Howard Smith. Business process management–the third wave: business process modelling language (bpml) and its pi-calculus foundations. *Information & Software Technology*, 45(15):1065–1069, 2003.

[16] Bipin Upadhyaya, Ying Zou, Shaohua Wang, and Joanna Ng. Automatically composing services by mining process knowledge from the web. In *Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*, pages 267–282, 2013.

[17] Wil M. P. van der Aalst. The application of petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.

[18] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

[19] L. Wang. *CookRecipe: towards a versatile and fully-fledged recipe analysis and learning system.* PhD thesis, City University of Hong Kong, 2008.

[20] Ingo Weber, Hye-Young Paik, and Boualem Benatallah. Form-based web service composition for domain experts. *TWEB*, 8(1):2, 2013.

[21] Xiang Zhao, Chuan Xiao, Xuemin Lin, Wei Wang, and Yoshiharu Ishikawa. Efficient processing of graph similarity queries with edit distance constraints. *VLDB J.*, 22(6):727–752, 2013.