Goal-Oriented Web-site Navigation for On-line Shoppers

Thesis submitted in partial fulfillment of graduate requirements for the degree “Master of Sciences” in Tel Aviv University
The Blavatnik School of Computer Science

By
Tom Yam

Prepared under the supervision of Prof. Tova Milo

October 2009
Acknowledgements

I wish to set forth in words my deep and unadulterated gratitude and appreciation to my thesis supervisor, Prof. Tova Milo. Thanks to her guidance, insights, encouragements and, the sometimes needed, admonishes, this thesis is complete. I also wish to thank Daniel Deutch for his great assistance.

To all my lab-mates from TAU, many thanks for lots of helpful insights, great atmosphere at the lab and wonderfully organized social events.

My most sincere thanks goes to my family, my mother and father, my two endearing sisters Mor and Tal and my grandparents and grandfathers, Ora, Judith, Eliyahu and Daniel. Their unconditional love and support lights my life.
Abstract

The rapid development of Web-technologies makes Web-based applications increasingly complex. The number of navigation possibilities of such applications has grown tremendously, and is sometimes infinite. The hardship of finding the best navigation paths (sequence of actions/clicks) that will lead to the most desired outcomes, has consequently risen.

E-commerce Web-sites are affected by this trend. With the number of products and services offered constantly raising, finding the most desired products is becoming increasingly difficult. Recommender systems [40] are E-commerce applications that are aimed at addressing this problem, by suggesting those products to the user that best suit her needs and preferences, in a given situation and context. They have been utilized for recommending travel products, books, CDs, financial services, and in many other applications [4, 10, 43].

These systems may help users choose products, but they do not recommend how to navigate the application to get the best outcome. For example, they will not recommend a shopper to join the shop’s costumers club prior to purchasing. Traditional recommender systems are also incapacitated to handle a set of inter-connected products, e.g. recommend a set components that form together a new PC.

This thesis presents ShopIT (ShoppIng assitanT), a system that assists on-line shoppers by suggesting the most effective navigation paths for their specified criteria and preferences. The suggestions are continually adapted to choices/decisions taken by the users while navigating. We present here the data model underling the system, the weight metrics and query language used to describe the user preferences, as well as a set of novel efficient algorithms for adaptive TOP-K query evaluation.
Contents

1 Introduction 1

2 The underlying model 5
   2.1 Overview ................................................. 5
   2.2 BP Specifications ..................................... 6
   2.3 EX-Flow .................................................. 8
   2.4 Weighted BP Model ................................. 11
   2.5 Queries .................................................. 15
   2.6 Top-K Problems ....................................... 17

3 Top-K Algorithms 19
   3.1 Overview ............................................... 19
   3.2 TOP-K-Flows - First attempt ......................... 19
      3.2.1 Properties of the Naive Algorithm .............. 21
   3.3 TOP-K-Flows - Second attempt ...................... 25
      3.3.1 Algorithm .......................................... 27
      3.3.2 Complexity ......................................... 33
      3.3.3 Optimality ...................................... 35
   3.4 Query evaluation (static) .......................... 37
      3.4.1 Complexity & Optimality ....................... 39
   3.5 Query evaluation (on-the-fly) ...................... 39
      3.5.1 Algorithm ....................................... 40

4 Incremental Computation and Diversity 45
   4.1 Motivation .............................................. 45
   4.2 Incremental Computation ............................ 47
   4.3 Diversity ............................................... 49
      4.3.1 Algorithm ....................................... 50
Chapter 1

Introduction

The rapid development of Web-technologies has led Web-based applications to become increasingly more complex. The number of navigation possibilities of such applications has also raised tremendously, and is sometimes infinite. The hardship of finding the best navigation paths, i.e. which will lead to the most desired outcomes, has consequently risen.

E-commerce Web-sites are affected by this trend. With the number of products and services offered constantly raising, finding the most desired products is becoming increasingly difficult. Recommender systems [40] are E-commerce applications that are aimed at addressing this problem, by suggesting those products to the user that best suit her needs and preferences, in a given situation and context. They have been utilized for recommending travel products, books, CDs, financial services, and in many other applications [4, 10, 43].

These systems may help a user choose products, but they will not recommend her on how to navigate the application to get the best outcome. For example, they will not recommend a shopper to join the shop’s costumers club prior to purchasing.

The goal of this thesis is to formalize the notion of navigation assistance and to implement and present a system which assist users in navigation of applications. Several challenges arise in the development of such a system. First, the number of possible navigation options in a given application could not only be large but infinite, as users may navigate back and forth between pages. Hence, enumerating and ranking all relevant options is clearly not an option. Second, it is critical to maintain a fast response time in order to
provide a pleasant user experience, as the system is meant to assist users in real-time.

A Business Process (BP for short) consists of a set of activities which, when combined in a flow, achieve some business goal. Web-applications, especially E-commerce applications, are naturally modeled by BPs. Every navigation path (sequence of actions/clicks) in the modeled application corresponds to an execution flow (sequence of activities) in the modeling BP.

We address the afore-mentioned issues by utilizing a novel BPs model and algorithms for querying it for best execution flows.

Contributions

The contributions of this thesis are the following:

- We present a model for business processes and provide algorithms for top-k querying over it. Our algorithms employ sophisticated data structures in order to factorize the computation and achieve good performance.

- We introduce the notion of a navigation recommender system, a systems which assists users in navigation of applications. We show how such a system can be constructed using our model and top-k algorithms. We explain how previous computations can be exploited for further queries, allowing for incremental evaluation. We also show a method for creating diverse and useful (for the user) recommendations.

- We have implemented the above ideas in ShopIT - a system which, given an application’s model, provides recommendations for navigation in it. The ShopIT system was demonstrated at the VLDB’09 conference. An accompanying demo paper appeared in the conference proceedings [20].

The research work was composed of two parts. The first, more theoretical in nature included in-depth complexity and optimality analysis of the presented algorithms and is not part of this thesis. The second, focused on the more practical aspects and is the subject of this thesis. For completeness, we mention in this thesis all the results. Whenever the described result is not considered a contribution of this thesis we point to the relevant reference containing the full details.
Thesis outline

The rest of the thesis is structured as follows:

• Chapter 2 - We present our model for representing business processes.

• Chapter 3 - Upon the previously presented model, we introduce algorithms for finding the best navigation paths in a BP (top-k computations).

• Chapter 4 - We show how top-k computations could be used to assist a user in navigating an application.

• Chapter 5 - We present our implemented recommender system for applications navigation. We also show how it can be integrated into an application and show the benefits, to the user, of using it.

• Chapter 6 - We provide non-trivial implementation details and present empiric results of our proposed algorithms.

• Chapter 7 - We present related work.

• Chapter 8 - We conclude our work and present possible future work.

The thesis has three appendixes, in which we give a list of third-party software packages we used to develop our systems, instructions on how to install our navigation recommendations system and on how to configure it.
Chapter 2

The underlying model

2.1 Overview

A Business Process (BP for short) consists of a set of activities which, when combined in a flow, achieve some business goal. BPs are typically designed via high-level specifications, e.g. by using the BPEL standard specification language [8], which are later compiled into executable code. Since the BP logic is captured by the specification, tools for querying and analyzing the possible execution flows (EX-flows for short) of a BP specification are extremely valuable to companies [6].

Web-applications, especially E-commerce applications, are naturally modeled by BPs. A BP captures all the possible navigation paths in a modeled application. Every navigation path (sequence of actions/clicks) in the modeled application corresponds to an execution flow (sequence of activities) in the modeling BP.

A single BP typically induces a large (possibly infinite, for recursive BPs) set of possible EX-flows. Among all these EX-flows, users are often interested only in a subset that is relevant for their requirements. This subset is described via a query. Since the number of query answers (qualifying EX-flows) may itself be extensively large (or even infinite), it is important to identify the ”best” EX-flow, where the notion of goodness is captured by some weighting metric that depends on the user’s goals.

To weight EX-flows, we assume that each choice taken during the EX-flow bears some weight (denoted $cWeight$, for choice weight), and that $cWeights$ of choices throughout the EX-flow are aggregated to obtain the
weight of the entire flow (denoted \( fWeight \), for flow weight). For example, \( cWeight \) may be the price of the product chosen at a given point of the EX-flow, or the likelihood of a user clicking on a given store link. In the first case, summation may be used for aggregation; for the case of likelihoods we may use multiplication. Following common practice [24], we require the aggregation to be monotonic with respect to the progress of the flow. This captures most practical scenarios, e.g. the total price of a subset of a shopping cart does not exceed the price of the full cart, even in the presence of discount deals. Or in the case of likelihood, the likelihood for a part of a path to be followed cannot be less than the likelihood of the entire path to be followed.

Observe that the \( cWeight \) of a given choice may vary at different points of the EX-flow and may depend on the course of the flow so far and on previous choices. For instance, the price of a given product may be reduced if the user had previously subscribed to a customers club or bought more than two products from the same store; the likelihood of clicking on a certain store link may depend on stores previously visited. Thus \( cWeight \) is modeled as a function whose input includes not only the choice itself but also information about the history of the EX-flow thus far.

We distinguish the \( cWeight \) functions into three classes, varying by the amount of history they depend on. We will use this differentiation in the next chapter, to provide a complexity analysis of various Top-K algorithms.

**Chapter outline**  In this chapter we present the model we use to describe and weight the possible EX-flows of a BP. The rest of this chapter is structured as follows. In section 2.2 we present a basic model for describing BPs. In section 2.3 we formalize the notion of execution flows. To support an effective top-k analysis, we extend the above basic model to support weighted EX-flows, and present this in section 2.4.

### 2.2 BP Specifications

At a high-level, a BP specification encodes a set of activities and the order in which they may occur.

A BP specification is modeled as a set of node-labeled DAGs. Each DAG has a unique start node with no incoming edges and a unique end
node with no outgoing edges. Nodes are labeled by activity names and directed edges impose ordering constraints on the activities. Activities that are not linked via a directed path are assumed to occur in parallel. The DAGs are linked through implementation relationships; the idea is that an activity \( a \) in one DAG is realized via the activities in another DAG. We call such an activity compound to differentiate it from atomic activities which have no implementations. Compound activities may have multiple possible implementations, and the choice of implementation is controlled by a condition referred to as the guarding formula.

We assume a domain \( A = A_{\text{atomic}} \cup A_{\text{compound}} \) of activity names and a domain \( F \) of formulas in predicate calculus.

**Definition 2.2.1** A BP specification \( s \) is a triple \( (S, s_0, \tau) \), where \( S \) is a finite set of node-labeled DAGs, \( s_0 \in S \) is a distinguished DAG consisting of a single activity, called the root, \( \tau : A_{\text{compound}} \to 2^{S \times F} \) is the implementation function, mapping each compound activity name in \( S \) to a set of pairs, each consisting of an implementation (a DAG in \( S \)) and a guarding formula in \( F \).

**Example 2.2.2** Figure 2.1 shows an example BP specification. The root \( s_0 \) has precisely one activity named *ShoppingMall*. The latter has as its implementation the DAG \( S_1 \), which describes a group of activities comprising user login, the injection of an advertisement, the choice of a particular store, and the user exit (possibly by paying). The directed edges specify the order in which the activities may occur, e.g., a user has to login first before a store can be chosen. Some of the activities are not in a particular order, e.g., the injection of an advertisement and the choice of a store, which means that they occur in parallel.

Within \( S_1 \), we observe that *Login* and *ChooseStore* are compound activities; the *Login* activity has two possible implementations \( S_2 \) and \( S_3 \) that are guarded by respective formulas. The idea is that exactly one formula is satisfied at run-time, e.g., the user either choose to login with a Visa or a Mastercard credit card, and thus *Login* is implemented either by \( S_2 \) or \( S_3 \) respectively. A similar observation can be made for *ChooseStore*, which has several possible implementations, but exactly one will be chosen at run-time depending on which guarding formula is satisfied. Note that the specification is recursive as e.g. \( S_4 \) may call \( S_1 \).
We are going to use this BP specification as our running example, through the rest of this work.

Figure 2.1: BP Specifications from example 2.2.2

We note that satisfaction of guarding formulas is determined by external factors, such as the choices of the user or environment parameters. We assume that exactly one guarding formula can be satisfied when determining the implementation of a given compound activity occurrence. An important observation is that satisfaction of guarding formulas can change if activities occur several times. For instance, a user may choose to buy a “DVD” product the first time she goes through the activities of $S_4$, and a “TV” product the second time.

2.3 EX-Flow

An EX-flow is modeled as a nested DAG that represents the execution of activities from a BP. Since, in real-life, activities are not instantaneous, we model each occurrence of an activity $a$ by two "$a$"-labeled nodes, the first standing for the activity activation and the second for its completion point. These two nodes are connected by an edge. The edges in the DAG represent the ordering among activities activation/completion and the implementation relationships. To emphasize the nested nature of executions, the implementation of each compound activity appears in-between its activation and completion nodes. Of course, the structure of an EX-flow DAG
must adhere to the structure of the defining BP specifications, i.e., activities have to occur in the same ordering and implementation relationships must conform to function $\tau$.

**Definition 2.3.1** Given a BP specification $s = (S, s_0, \tau)$, $e$ is an execution flow (EX-flow) of $s$ if:

- **Base EX-Flow**: $e$ consists only of the activation and completion nodes of the root activity $s_0$ of $s$, connected by single edge, or,

- **Expansion Step**: $e'$ is an EX-flow of $s$, and $e$ is obtained from $e'$ by attaching to some activation-completion pair $(n_1, n_2)$ of an activity $a$ in $e'$ some implementation $e_a$ of $a$, through two new edges, called implementation edges, $(n_1, \text{start}(e_a))$ and $(\text{end}(e_a), n_2)$, and annotating the pair with the formula $f_a$ guarding $e_a^1$. We require that this $(n_1, n_2)$ pair does not have any implementation attached to it already in $e'$, whereas all its ancestor compound activities in $e'$ do have one. In the attached implementation $e_a$, each activity node is replaced by a corresponding pair of activation and completion nodes, connected by an edge.

We call $e$ an expansion of $e'$, denoted $e' \to e$.

We use $e' \to^* e$ to denote that $e$ was obtained from $e'$ by a sequence of expansions. An activity pair $a$ in a partial EX-flow $e$ is unexpanded if it is not the source of any implementation edge, implying that $a$ can be used to further expand $e$. We say that an EX-flow is partial if it has unexpanded activities, and full otherwise, and denote the set of all full flows of a BP specification $s$ by $\text{flows}(s)$.

For a graph $e$, we say that $e$ is an EX-flow if it is a (partial or full) flow of some BP specification $s$.

**Example 2.3.2** An example EX-flow of the shopping mall BP is given in figure 2.2. Ordering (implementation) edges are drawn by regular (dashed) arrows. The EX-flow describes a sequence of activities that occur during the BP execution. The user logs in with a Visa Credit Card, then chooses...

---

1Note that by attaching an implementation we attach an entire activities DAG to $(n_1, n_2)$
to shop at the BestBuy store. There, she chooses to look for a DVD player and selects one by Sony, then continues shopping at the same store, looking for a TV, and selects one also by Toshiba (this last part is omitted from the figure). Finally she exits and pays.

Figure 2.2: Execution flow from example 2.3.2

Clearly, a given EX-flow may be obtained via different expansion sequences that vary in the order in which parallel activities are expanded. To simplify the presentation, we impose a total order among unexpanded activities of any given partial EX-flow, and assume that at each step, the first unexpanded activity is expanded. (This total order can be achieved easily by a topological sort of the EX-flow DAG.) Thus, any partial EX-flow corresponds to a well defined and unique sequence of expansion steps from the base EX-flow. We stress that this assumption is made solely for presentation considerations—our results extend to a general context where multiple expansion orders are possible, as we will later explain in chapter 3.
2.4 Weighted BP Model

We next extend this basic model, to capture weighted EX-flows.

We assume an ordered domain $W$ of weights. We use three functions:
(1) $cWeight$ that describes the weight of each implementation choice, given its preceding EX-flow,
(2) $aggr$ that aggregates the $cWeights$ throughout the EX-flow, and
(3) $fWeight$ that describes the obtained EX-flow weight.

The $cWeight$ function

Given a BP specification $s$, $cWeight$ is a partial function that assigns a weight $w \in W$ to each pair (partial EX-flow $e$ of $s$, guarding formula $f$ in $s$), such that $f$ guards the compound activity node of $e$ that is next to be expanded. Intuitively, the value $cWeight(e, f)$ is the weight of the implementation guarded by $f$, given that $e$ is the flow that preceded it.

Example 2.4.1 Re-consider the EX-flow in our running example, and a $cWeight$ function that assigns, to each implementation choice (guarding formula), the additional cost that this choice incurs to the customer. In this case $W$ is the set of all positive numbers. The $cWeight$ of the $brand = \"Sony\"$ choice, given that the preceding flow indicates that the Sony product that is being purchased is a DVD, the shop is BestBuy and the user has identified herself as a Visa card holder, may be the Visa discounted price of a Sony DVD at BestBuy. The $cWeight$ of a Sony choice, given now that the product is a TV and that the preceding flow already includes a purchase of a Sony product (the DVD), may reflect, e.g., a 20% discount for the second product. The $cWeight$ of other choices (like the store or the product type) may be zero here, as they incur no additional cost to the user.

We distinguish three classes of $cWeight$ functions, called history-independent, bounded-history, and unbounded-history functions, reflecting how much of the preceding flow $e$ actually inflicts on the $cWeight$.

History independence: History-independent $cWeight$ functions compute $cWeight(e, f)$ based solely on the choice $f$ and ignore the history $e$ leading to it. More formally, for every guarding formula $f$, $cWeight(e, f) = cWeight(e', f)$ for each two partial EX-flows $e, e'$.

Example 2.4.2 To continue with our example, if $cWeight$ represents prices, it is history-independent if the prices of all products of all brands in all
stores are the same and no deals are available. The CWeight of choosing a store is also history-independent as it is always zero. For likelihood, history-independence means probabilistic independence between choices.

Bounded-history: Bounded-history cWeight functions capture the more common scenario where the cWeight value does depend on the history e of the EX-flow, but only in a bounded manner. That is, to determine the cWeight of any implementation choice at an activity node n, it suffices to consider only the implementation choices of the last b preceding compound activities, for some bound b. By “preceding” we refer here to activity nodes n that are ancestors of n in e (in contrast to nodes that occur in parallel and thus in general may or may not precede n). By “choice” we refer to the formula guarding the implementation selected for the activity. More precisely, recall that we assumed that given an EX-flow e, the expansion sequence leading to e is well defined. The last choice preceding (the expansion of) a node n in this sequence, denoted $\text{PrevChoice}(e,n)$, is the guarding formula of the implementation selected for the last compound activity node n in this sequence that is an ancestor of n. Similarly, $\text{PrevChoice}^2(e,n)$ are the last two preceding choices, and more generally $\text{PrevChoice}^i(e,n)$ is a vector consisting of the i last preceding choices. We then say that two activity nodes $n_1 \in e_1$ and $n_2 \in e_2$ were preceded by the same last b choices if $\text{PrevChoice}^b(n_1,e_1) = \text{PrevChoice}^b(n_2,e_2)$.

Now the notion of bounded-history cWeight functions is formally defined as follows.

**Definition 2.4.3** We say that cWeight is bounded-history, with bound b, if for every activity name $a$, every guarding formula $f$ of $a$, and every two pairs of [EX-flow,next-to-be-expanded-node] $(e,n), (e',n')$ where $\lambda(n) = \lambda(n') = a$ and $\text{PrevChoice}^b(e,n) = \text{PrevChoice}^b(e',n')$, it holds that $\text{cWeight}(e,f) = \text{cWeight}(e',f)$.

**Example 2.4.4** In our running example, assume that the price of a given brand depends only on the store, the type of the product being bought and whether or not the user has identified herself in this purchase as a Visa card holder. Here the bound b is 3: for nodes n with $\lambda(n) = \text{ChooseBrand}$, $\text{PrevChoice}^3(e,n)$ consists of the choices of credit card, store and product that preceded the choice of brand, and all $\text{ChooseBrand}$ nodes having the
same PrevChoice\textsuperscript{3} have the same cost (cWeight) for each possible choice of brand (guarding formula). In a similar way, a larger history bound can be used to take into consideration also (a bounded amount of) information on previously purchased products.

More generally, one may use a selective notion of history where choices are recorded in the history vector only for a subset of the compound activities (that are considered significant for the cWeight computation). For instance, consider the case where, in-between the choice of store and that of product, the user may navigate to a different Web-page, then return, then navigate again, and return, an unbounded number of times. To capture the dependency, ignoring the unbounded number of (irrelevant) choices made in-between, one may omit from the history vector these navigation activities. All our results extend to such a generalized setting.

Unbounded-history: Unbounded-history cWeight functions may use an unbounded portion of the flow history e to compute the next choice’s weight. For instance, if the price of a given product depends on the exact full sequence of searches that the user performed prior to the purchase, then the corresponding cWeight function is unbounded-history.

**Observation** Note that, by definition, for non-recursive BPs, cWeight functions are always bounded-history, with the bound being, at most, the BP nesting depth. Recursive BPs, on the other hand, may have unbounded-history cWeights. However, this is extremely rare in practice [38].

**The Aggregation function**

The weights along the EX-flow are aggregated using an aggregation function. The function $aggr : W \times W \rightarrow W$ receives two weights as inputs; the first intuitively corresponds to the aggregated weight computed so far, and the second is the new cWeight to be aggregated with the previous value. For instance, when computing purchase cost $aggr = +$ and $W = [0, \infty)$; when computing likelihood, $aggr = \times$ and $W = [0, 1]$.

We consider here aggregation functions that satisfy the following intuitive constraints:
1. \( \text{aggr} \) is associative and commutative, namely for each \( x, y, z \in W \),
\[
\text{aggr}(\text{aggr}(x, y), z) = \text{aggr}(x, \text{aggr}(y, z)), \quad \text{and} \quad \text{aggr}(x, y) = \text{aggr}(y, x).
\]

2. \( \text{aggr} \) is continuous, that is for each \( x, y, z \in W \), if \( \text{aggr}(x, y) <\text{aggr}(x, z) \) then there exists \( w \in W \) such that \( \text{aggr}(x, y) < \text{aggr}(x, w) < \text{aggr}(x, z) \).

3. \( \text{aggr} \) has a neutral value, denoted \( 1_{\text{aggr}} \). Namely for each \( x \in W \),
\[
\text{aggr}(x, 1_{\text{aggr}}) = \text{aggr}(1_{\text{aggr}}, x) = x.
\]

4. \( \text{aggr} \) is monotonically increasing or decreasing over \( W \). Namely, either for each \( s, x, y \in W \) \( x \geq y \implies \text{aggr}(s, x) \geq \text{aggr}(s, y) \) and \( \text{aggr}(s, x) \geq s \), or the same for \( \leq \).

Observe that the above mentioned aggregation functions \(+\) and \(\times\), for cost and likelihood, satisfy the constraints.

The \( f\text{Weight} \) function

Finally, the \( f\text{Weight} \) of an EX-flow is obtained by aggregating the \( \text{cWeights} \) of all choices made during the flow, and is defined as follows.

**Definition 2.4.5** Given a BP specification \( s \) with root \( s_0 \), a \( \text{cWeight} \) function and an aggregation function \( \text{aggr} \), the function \( f\text{Weight} \) that assigns weights to EX-flows of \( s \) (with respect to \( \text{cWeight} \), \( \text{aggr} \)) is defined as follows:

1. If \( e \) is an EX-flow consisting only of the root \( s_0 \),
\[
f\text{Weight}(e) = 1_{\text{aggr}}.
\]
   Else,

2. if \( e' \rightarrow e \) for some EX-flow \( e' \) of \( s \), then
\[
f\text{Weight}(e) = \text{aggr}(f\text{Weight}(e'), \text{cWeight}(e', f)), \quad \text{where} \ f \ \text{is the formula guarding the implementation that is added to} \ e' \ \text{to form} \ e.
\]

According to the definition above, \( f\text{Weight} \) is defined only for EX-flows that originate from the BP root activity. However, the definition naturally extends to sub-flows originating from an arbitrary compound activity node \( n \) in the EX-flow (treating it as a root). Namely, all \( \text{cWeight} \) values are defined as before, with \( f\text{Weight} \) aggregating only \( \text{cWeight} \) values within the sub-flow rooted at \( n \).
Example 2.4.6 To illustrate how all these notions work together, assume that a Sony TV and DVD, which individually cost 250$ and 150$ respectively, are sold together with 20% discount (i.e. for 320$). The cWeight for the first choice of the Sony TV (DVD) is 250$ (150$). The cWeight for the following choice, of DVD (TV), is 70$ (170$), computed as 320$ minus the cost already incurred for the first product. Using \(\text{+}\) for aggregation, the individual cWeights along the flow are summed up, yielding the total 320$ deal price.

Observe that when aggr is monotonically increasing (decreasing), so is fWeight, in the sense that the weight of an EX-flow increases (decreases) as the execution advances. Generally, when fWeight is increasing, e.g. for the overall price of purchases, we will be interested in the bottom-k full flows (e.g. the cheapest overall price). When fWeight is decreasing (as, e.g., for the likelihood of EX-flows), we will be interested in the top-k (e.g. the most likely) ones. Since all definitions and algorithms presented in this work apply symmetrically to both cases, we consider from now on only monotonically decreasing functions and top-k EX-flows.

2.5 Queries

Queries select EX-flows of interest using execution patterns, an adaptation of the tree/graph patterns offered by existing query languages for XML/graph-shaped data [11], to BP nested DAGs. We adopt the query language introduced in [6]. An execution pattern is a nested DAG of shape similar to that of an EX-flow. An edge in an execution pattern is either regular, i.e., it matches a single edge in the EX-flow DAG, or transitive, i.e., it matches a path\(^2\). Similarly, activity pairs may be regular or transitive for searching only in their direct implementation or zooming-in transitively inside it, respectively. Finally, activity nodes that are marked by a special “ANY” symbol may be matched to BP nodes bearing any label.

Definition 2.5.1 An execution pattern (EX-pattern for short) is a pair \(\hat{p} = (\hat{e}, T)\) where \(\hat{e}\) is an EX-flow whose nodes are labeled by labels from \(A \cup \{\text{ANY}\}\) and may be annotated by guarding formulas. \(T\) is a distinguished set of activity pairs and edges in \(\hat{e}\), called transitive.

\(^2\)which my correspond to several edges
Example 2.5.2 An example query (EX-pattern) is given in Fig. 2.3. The double-lined edges (double-boxed nodes) are transitive edges (activities). The query looks for EX-flows where the user chooses a DVD of brand Sony (possibly after performing some other activities, corresponding to the transitive edges), then chooses also a TV (of any brand). The ShoppingMall activity is transitive, indicating that its implementation may appear in any nesting depth; ChooseProduct is not transitive, requiring the brand choice to appear in its direct implementation.

Figure 2.3: Query (EX-Pattern) from example 2.5.2

Given a BP specification \( s \), a query (EX-pattern) \( p \) selects the EX-flows \( e \in \text{flows}(s) \) that contain an occurrence of \( p \). Informally, nodes and edges of the EX-pattern are matched to nodes and edges of EX-flows, respecting activity names, ordering and implementation edges. Formally,

Definition 2.5.3 Let \( p = (\hat{e}, T) \) be an execution pattern and let \( e \) be an EX-flow. An embedding of \( p \) into \( e \) is a homomorphism \( \psi \) from the nodes and edges in \( p \) to nodes, edges and paths in \( e \) s.t.

1. [nodes] activity pairs in \( p \) are mapped to activity pairs in \( e \). Node labels and formulas are preserved; a node labeled by \( \text{any} \) may be mapped
to a node with any activity name.

2. (edges) each (transitive) edge from node $m$ to node $n$ in $p$ is mapped to an edge (path) from $\psi(m)$ to $\psi(n)$ in $e$. If the edge $[n,m]$ belongs to a direct internal flow of a transitive activity, the edge (edges on the path) from $\psi(m)$ to $\psi(n)$ can be of any type (flow, or implementation choice) and otherwise must have the same type as $[n,m]$.

An EX-flow $e$ belongs to the query result if there exists some embedding of $p$ into $e$.

### 2.6 Top-K Problems

We conclude this chapter by presenting the **TOP-K-FLOWS** and the **TOP-K-ANSWERS** problems. In the next chapter we will propose several algorithms for solving these problems.

Given a BP specification $s$ and a monotonically decreasing $fWeight$ function for $s$, we call the problem of identifying the top-$k$ weighted EX-flows in $\text{flows}(s)$ the **TOP-K-FLOWS** problem (for $s$, $fWeight$, and $k$). Note that certain EX-flows may have equal weights, which implies that there may be several valid solutions to the problem, in which case we pick one arbitrarily. When a query $q$ is given, the problem of finding the top-$k$ weighted EX-flows among those satisfying the query, is called the **TOP-K-ANSWERS** problem (for $s$, $fWeight$, $k$ and $q$).

We will first consider **TOP-K-FLOWS**, as it is interesting by itself as a tool for BP analysis, and also since it forms, as we shall see in the sequel, an important ingredient in the solution of **TOP-K-ANSWERS**.
3.1 Overview

The previous chapter concluded with the introduction of the TOP-K-FLOWS and the TOP-K-ANSWERS problems. In this chapter we present our solutions to these problems.

In section 3.2 we present a simple algorithm for solving the TOP-K-FLOWS problem. This algorithm, as we shall see, suffers from various performance issues. In section 3.3 we present a refined algorithm for solving the TOP-K-FLOWS problem, built upon the simple algorithm. The refined algorithm successfully solves the performance issues. We also provide a complexity and an optimality analysis for the refined algorithm.

Finally, in sections 3.4 and 3.5 we present two approaches for solving the TOP-K-ANSWERS problem. While the former employs a static analysis approach, the latter employs an on-the-fly approach.

As mentioned in the introduction, the theoretical results, regarding the optimality and the complexity of the various algorithms presented here, are not considered a contribution of this thesis, and are only given here for completeness. The parts dealing with these results are thus brief and relevant pointers are provided.

3.2 TOP-K-Flows - First attempt

The FindFlows procedure, presented in algorithm 1, attempts to compute the top-k EX-flows of a given BP. It is given as input a BP specification s,
a weight function over its EX-flows, represented by $cWeight$ and $aggr$, and the number $k$ of requested results. Its output is an ordered queue $Out$ of top EX-flows. The algorithm uses a function $\text{AllExpansions}$ that, given a partial EX-flow $e$ returns all EX-flows $e'$ such that $e \rightarrow e'$, along with their corresponding guarding formulas $F'$.

$\text{FindFlows}$ operates in the spirit of the $A^*$ [15] search algorithm: The computation may be intuitively viewed as a gradual, greedy generation of a (possibly infinite) search tree, whose nodes correspond to possible (partial or full) EX-flows. The root node is an EX-flow consisting of only the BP root; at each step, we choose a leaf $e$ in the search tree, consider all of its possible next expansions and compute their $fWeight$. Each such expansion $e'$ becomes a child of $e$, and the computation continues with a tree leaf having the highest $fWeight$.

The algorithm maintains a priority queue $\text{Frontier}$ of (partial) EX-flows, ordered by $fWeight$. These are EX-flows which we might still need to consider. Initially $\text{Frontier}$ contains a single partial EX-flow, containing only the BP root (line 1). At each step, we pop the highest weighted EX-flow $e$ from $\text{Frontier}$ (line 3). If $e$ is a full EX-flow, we insert it to the output queue $Out$ (line 4-6). This is justified in Theorem 3.2.1 below. Otherwise, if its a partial EX-flow, we invoke $\text{HandlePartialFlow}$ over $e$ (line 7-9). $\text{HandlePartialFlow}$, presented in Algorithm 2, considers all possible direct expansions $e'$ of $e$, obtained by choosing some implementation, guarded by $F'$, at the next-to-be-expanded node of $e$ (lines 1-2). It computes the weight of each EX-flow and inserts it to (the global) $\text{Frontier}$ (lines 3-4).

We will refer to this algorithm as the naive (top-k) algorithm or as $\text{NaiveTopK}$, to reflect its simplicity.
Algorithm 1: FindFlows

Input: $s; cWeight; aggr; k$
Output: Out

1 insert $(s_0, 1_{aggr})$ into Frontier;
2 while $|Frontier| > 0 \land |Out| < K$ do
3 \hspace{1em} $(e, w_e) \leftarrow \text{pop}(Frontier)$;
4 \hspace{1em} if $e$ is a full EX-flow then
5 \hspace{2em} insert $(e, w_e)$ into Out;
6 \hspace{1em} end
7 \hspace{1em} else
8 \hspace{2em} \text{HandlePartialFlow}(e);
9 \hspace{1em} end
10 end

Algorithm 2: HandlePartialFlow

Input: Flow $e$

1 Expansions $\leftarrow \text{AllExpansions}(e)$;
2 foreach $(e', F') \in \text{Expansions}$ do
3 \hspace{1em} $w_{e'} \leftarrow \text{aggr}(w_e, cWeight(e, F'))$;
4 \hspace{1em} insert $(e', w_{e'})$ into Frontier;
5 end

3.2.1 Properties of the Naive Algorithm

We next analyze FindFlows. On the positive side, we show that the algorithm inserts EX-flows to Out in a correct order (theorem 3.2.1). As a consequence, the algorithm is valid, i.e. if it terminates, Out contains the top−k EX-flows.

Theorem 3.2.1 At any point of the algorithm execution, there exist no $e, e' \in \text{flows}(s)$ such that $e \in \text{Out}, e' \notin \text{Out}, \text{ and } fWeight(e) < fWeight(e')$.

Proof. The proof goes by contradiction. Let us assume the existence of such $e, e'$. We say that $e''$ is an ancestor of $e'$ if $e'' \rightarrow^* e'$. We denote by $e''$ the lowest ancestor of $e'$ that was already in Frontier at the time of moving $e$ to Out (there exists such EX-flow, perhaps consisting only of the root). $fWeight(e) \geq fWeight(e'')$, otherwise $e$ would have not been moved to Out. But $fWeight(e'') \geq fWeight(e')$ by monotonicity, thus $fWeight(e) \geq fWeight(e')$. □
On the negative side, though the algorithm is valid, it suffers from some significant drawbacks.

- First, when the BP contains recursion, the algorithm may repeatedly choose recursive expansions and fail to halt.
- Second, even for non-recursive BPs, it may suffer from bad performance: its run-time complexity depends on \(c\text{Weight}\) values and the amount of weight improvement obtained by each expansion, and is not necessarily a function of the BP size.
- Third, note that the algorithm explicitly generates EX-flows. As the size of even a single EX-flow may be exponential in the BP size, this may be costly.

Let us now observe some scenarios illustrating these issues.

**Example 3.2.2** Consider the following recursive BP specification, depicted in figure 3.1. The \(c\text{Weights}\) are in the \([0, 1]\) range, and \( \text{aggr} = \times \). Its root activity \(A\) has two possible implementations: the first, \(S_1\), guarded by a formula \(F_1\) with \(c\text{Weight}\) of 0, consists of a single atomic activity \(B\). The second, \(S_0\), guarded by \(F_2\) with \(c\text{Weight}\) of 0.5, is a recursive invocation of \(A\).

A top-k querying of these BP specification using the FindFlows algorithm illustrates the first drawback. The algorithm keeps considering recursive expansions of \(A\), each time obtaining EX-flows with decreasing \(f\text{Weight}\), but nevertheless higher than 0. The algorithm will never terminate because the top-1 EX-flow has \(f\text{Weight}\) of 0.

![Figure 3.1: BP Specifications from example 3.2.2](image-url)
Example 3.2.3 Consider the following two non-recursive BP specifications, depicted in figure 3.2. In both BPs, the cWeights are in the \([0, 1]\) range, and \(aggr = \times\).

A top-k querying of these BP specifications using the \(\text{FindFlows}\) algorithm illustrates the second drawback. The number of expansions needed to compute the top-1 EX-flow differs on the two BPs. The left BP requires 2 expansions while the right BP requires 3, even tough they are exactly of the same size and structure. For larger BPs, the difference will greatly increase.

![Figure 3.2: BP Specifications from example 3.2.3](image)

Example 3.2.4 Consider the following recursive BP specification, depicted in figure 3.3. The cWeights are in the \([0, 1]\) range, and \(aggr = \times\).

In this BP, every full EX-flow will consist at least 100 expansions of \(B\). This is because the More node will recurse until \(B\) has been expanded 100 times, before it can use \(Foo\) as its implementation to obtain a full EX-flow.

A top-k querying of these BP specification using the \(\text{FindFlows}\) algorithm illustrates the third drawback. The explicitly generated EX-flows will exponentially long, in the size of the BP. If we could use one expansions of \(B\) for all the EX-flows, the complexity would decline dramatically.

We have seen three examples, one for each of the drawbacks of the naive algorithm. We next give an additional example, which would hopefully help refine the reader’s intuition, for the next section.

Example 3.2.5 Consider the following non-recursive BP specification, depicted in figure 3.4. The cWeights are in the \([-\infty, 0]\) range, and \(aggr = -\).
The algorithm starts by expanding $A$, inserting $A \Rightarrow S_1$ and $A \Rightarrow S_2$ to the Frontier. The next step will be to pop $A \Rightarrow S_1$ from the Frontier and expand $B$, which is the currently the first expandable node. The result will be the full EX-flow $A \Rightarrow S_1 B \Rightarrow S_3$ and $A \Rightarrow S_1 B \Rightarrow S_4$. Next, the algorithm pops $A \Rightarrow S_2$ and expands $B$ for the second time.

We expand $B$ twice, although, had we cached the results of the first expansion, we would have been able to use that cached results instead of expanding $B$ for the second time.
The above example shows a case in which the naive algorithm makes redundant computations, thus being slower than an optimal algorithm. In the next section we present a refined version of the naive algorithm. The refined algorithm is designed to cope with the afore-mentioned issues, and is designed such that the no redundant computations are made.

### 3.3 TOP-K-Flows - Second attempt

The refined algorithm is based on the two observations presented next.

**Observation A** Some distinct activity nodes \( n, n' \) (appearing in the same or in different EX-flows) may be in fact equivalent, in the sense that every sub-flow that may originate from \( n \) may also originate from \( n' \), having exactly the same \( fWeight \). An algorithm may exploit this to compute \( fWeight \) values just once for each equivalence class. This equivalence notion is formally defined as follows.

**Definition 3.3.1** Given two pairs of \([\text{flow,first-expandable-node}], [e,n] \) and \([e',n'] \), we say that \([e,n],[e',n'] \) are equivalent if:

1. Nodes \( n \) and \( n' \) are labeled by the same activity name.
2. For all EX-flows \( \hat{e}, \hat{e}' \) s.t. \( e \rightarrow^* \hat{e}, e' \rightarrow^* \hat{e}' \), in which the sub-flows rooted at \( n \) and at \( n' \) (denoted \( \hat{e}_n \) and \( \hat{e}_{n'} \)) are isomorphic, \( fWeight(\hat{e}_n) = fWeight(\hat{e}_{n'}) \).

We use \( \text{equiv} \) to denote the set of all equivalence classes in a BP specification.

Let us exemplify this observation.

**Example 3.3.2** First observe that for history-independent cWeight functions, all pairs \([e,n],[e',n'] \) where \( n \) and \( n' \) has the same activity name are equivalent. This implies, for instance, that in example 3.2.2, the computation performed for recursive invocations of \( A \) may be factorized (and done just once), as sub-flows will have the same \( fWeight \) regardless in which of this recursive invocations they will be rooted. This also implies in example 3.2.5, according to this observation, we indeed may perform the expansion of \( B \) just once.
For bounded-history \( cWeight \) functions, on the other hand, two nodes are equivalent only if their history vector are the same. For instance, reconsider the bounded-history \( cWeight \) function from Example 2.2.2. Two instances of ChooseBrand here are equivalent if they both had the same preceding choices of store, product and credit card.

**Observation B** The monotonicity of \( fWeight \) facilitates incremental-style computation, i.e. if the \( j \)'th ranked sub-flow rooted at an activity node \( n \) uses sub-flows rooted at nodes of the same equivalence class, e.g. due to recursion, it can only use better ranked sub-flows rooted at nodes of the same equivalence class as \( n \). More formally,

**Lemma 3.3.3** For every equivalence class \( eq \) and a compound activity node \( n \in eq \), the following hold:

1. There exists a best ranked (top-1) EX-flow originating at \( n \) that contains no occurrence of any other node \( n' \in eq \). And,

2. For \( j > 1 \), there exists a \( j \)'th ranked EX-flow originating at \( n \) such that for any occurrence of a node \( n' \in eq \) in it, the sub-flow rooted at \( n' \) is one of its top \( j-1 \) EX-flows.

**Proof.** For (1), observe that for each EX-flow that contains an occurrence of \( n' \) we may, without reducing weight, eliminate the sub-flow in-between \( n \) and \( n' \), and replace the internal EX-flow of \( n \) by that of \( n' \). Similarly, for (2), for a \( j \)'th EX-flow violating the lemma constraints we may replace the sub-flow rooted at \( n' \) by one of the top \( j-1 \) EX-flows of \( n \).

The above lemma holds only for history-independent BP specifications. In the case of bounded-history BP specification, one may use the result of [17] to construct an equivalent history-independent BP specification \( s' \) whose size is linear in the number of equivalence classes induced by \( s \). It follows that lemma 3.3.3, and all inferred results, go through to such bounded-history cases, with a linear dependency on \( |equiv| \).

**Example 3.3.4** Reconsider the recursive BP specification given in example 3.2.2. The \texttt{NaiveTopK} algorithm keeps considering recursive expansions of \( A \), each time obtaining EX-flows with decreasing weight, but nevertheless higher than 0, and will never terminate.
However, following Lemma 3.3.3, we know this is redundant: to compute the top-1 EX-flow one may avoid considering EX-flows that contains a recursive call to $A$. The top-2 EX-flow may contain a recursive invocation of $A$, but the only sub-flow that needs to be considered as potential expansion for this occurrence of $A$ is the (already computed) top-1 EX-flow, and so on.

3.3.1 Algorithm

The refined TOP-K-FLOWS algorithm, depicted in algorithm 3, operates in two steps: first it calls a refined version of FindFlows which computes a compact representation of the top-k EX-flows, then it calls EnumerateFlows to explicitly enumerate the EX-flows from this compact representation.

<table>
<thead>
<tr>
<th>Algorithm 3: TOP-K</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> $s$; $cWeight$; $aggr$; $k$</td>
</tr>
<tr>
<td><strong>Output:</strong> $top-k$</td>
</tr>
<tr>
<td>1 Initialize $FTable$ ;</td>
</tr>
<tr>
<td>2 $tmp$ $\leftarrow$ $RefinedFindFlows(s, cWeight, aggr, k)$ ;</td>
</tr>
<tr>
<td>3 $top-k$ $\leftarrow$ $EnumerateFlows(tmp)$ ;</td>
</tr>
<tr>
<td>4 <strong>return</strong> $top-k$ ;</td>
</tr>
</tbody>
</table>

Following observations A and B, we define a table, $FTable$, which (compactly) maintains the top sub-flows ($k * \text{linear}(|\text{equiv}|)$ of them) for each equivalence class. It has rows corresponding to equivalence classes, and columns containing pointers to sub-flows. We also define a companion data-structure, $FlowsHold$, which maintains a list of partial EX-flows we should not consider yet. For each equivalence class, it contains a list of such EX-flows.

The algorithm utilizes these data-structures to achieve optimality with respect to the number of expansions made.
Algorithm 4: RefinedFindFlows

Input: \( s; cWeight; aggr; k \)
Output: Out

1. insert \((s_0, 1_{aggr})\) into Frontier;
2. while \(|Frontier| > 0 \land |Out| < K\) do
3. \((e, w_e) \leftarrow \text{pop}(Frontier)\);
4. if \(e\) is a full EX-flow then
5. \hspace{1em} \text{RefHandleFullFlow}(e);
6. else
7. \hspace{1em} \text{RefHandlePartialFlow}(e);
8. end
9. end

Algorithm 5: RefHandleFullFlow

Input: Flow \(e, w_e\)

1. foreach node \(n \in e\) do
2. \hspace{1em} \text{HandleFullPartialFlow}(n);
3. end
4. insert \((e, w_e)\) into Out;
Algorithm 6: RefHandlePartialFlow

**Input:** Flow $e$

1. **foreach** node $n \in e$ such that the flow rooted at $n$ has just fulfilled **do**
2.  
3.  
4.  
5.  
6.  
7.  
8.  
9.  
10. **end**
11. **end**
12. 
13. 
14. 
15. **else**
16. 
17. 
18. 
19. 
20. 
21. 
22. 
23. **end**
24. 

Algorithm 7: HandleFullPartialFlow

**Input:** Flow $e$, Node $n$

1. $flow_{pre} \leftarrow$ sub-flow of $e$ preceding $n$
2. $eq \leftarrow$ EquivalenceClass($flow_{pre}, n$)
3. $row \leftarrow FTable.getRow(eq)$
4. **if** $row = NULL$ **then**
5.  
6. **end**
7. $flow_{rooted} \leftarrow$ sub-flow of $e$ rooted at $n$
8. **if** $flow_{rooted} \notin row$ **then**
9.  
10. **else**
11. **end**
RefinedFindFlows  The main loop is very similar to FindFlows’s main loop. However, instead of simply inserting full EX-flows to Out (line 5) RefHandleFullFlow (algorithm 5) is used. Instead of invoking HandlePartialFlow on partial EX-flows, RefHandlePartialFlow (algorithm 6) is used (line 8).

Before explaining these new procedures, we shall outline some of the differences between the FindFlows algorithm and the RefinedFindFlows algorithm. Previously, Out contained the explicitly enumerated top-k EX-flows. Now, Out contains a compact representation of the EX-flows, based on the data stores at FTable. The FTable contains sub-flows, and is filled by the RefHandlePartialFlow and the RefHandleFullFlow procedures. Previously, given some EX-flow e we where only interested whether e was a partial or full EX-flow. We now consider a third case. We call it a “full-partial EX-flow” - a partial EX-flow, that one or more of its sub-flows are full. That is, the (sub-)flow rooted at one of e’s nodes, cannot be further expanded. We use these sub-flows to populate FTable.

Partial EX-flows  The refined treatment of partial EX-flows is detailed in algorithm 6.

RefHandlePartialFlow starts by searching for recently fulled sub-flows. By “recently”, we mean sub-flows we have yet to consider. For every such sub-flow, rooted at node n of e, we invoke HandleFullPartialFlow (lines 1-4), which we will explain later.

We than get the first expandable node, v, of e (line 5). We computes v’s equivalence class eq (line 6), and looks it up in the FTable (line 7). When the first expandable node and the EX-flow’s prefix, form some equivalence class eq, we say that the EX-flow’s first expandable is eq.

If no table-entry is found (line 8), it means that we have not encountered yet an equivalent node during the computation. We thus create a new row in FTable for this equivalence class (line 9). This row’s entries will be filled later, when corresponding full sub-flows are found. Then, we process v’s expansions as before (line 10) using algorithm 2.

Otherwise, if the appropriate row already exists in the FTable, we consider partial EX-flows that appear in this row but were not yet considered for expanding e (line 13). If no such EX-flow exists (line 14), (although the table entry itself does exist), it means that e was previously reached when expanding some other node v’ (which appears in e as well). Following
observation B, we may compute the next best EX-flow without computing further expansions of \( e \). Thus, we put \( e \) on hold (line 15). It will be released only later, upon finding a new full EX-flow originating in \( v' \) (see below).

Else, i.e. an unused EX-flow does exist, let \( unused \) denote the highest ranked such EX-flow. We first copy \( e \) and mark \( unused \) as if it was used to expand \( e \) at some point (lines 18-19). We then inserts the copy to \( OnHold \) (line 20). The copy will be returned to the \( Frontier \), when another \( e \) or an expansion of \( e \) becomes full. The original \( e \) is expanded with \( unused \), by “hanging” it on \( v \): we make \( v \) point to \( unused \) (lines 21). We than clear the used for expansion list of \( e \) (line 22), so it may use \( unused \) if it ever reaches that equivalence class again. We than insert the expanded \( e \) into \( Frontier \), for further expansions (line 23).

The \( FTable \), as we can see, serves as a synchronization method. Whenever the first expandable of a partial EX-flow \( e \) is some equivalence class \( eq \), we might be in three different cases.

- There is no corresponding \( FTable \) entry - \( e \) is the first EX-flow, whose first expandable is \( eq \), we consider. In this case, \( e \) will “spearhead” the computation. We expand \( e \) and insert all the expansions to \( Frontier \) but we also create an \( FTable \) entry (which will be filled later). This \( FTable \) entry marks that some EX-flow is already spearheading the computation for \( eq \). We do not need a second EX-flow to spearhead the computation because from observation A we know that the computation of that other EX-flow will end-up exactly as the computation of \( e \).

- There is a corresponding \( FTable \) entry and one of the sub-flows in that entry is not marked as used by \( e \) - another EX-flow already “spearhead” the computation for \( eq \) (and possibly is still spearheading). In this case we simply use that sub-flow to expand \( e \) and copy \( e \) to \( FlowsHold \). That copy might be expanded by other sub-flows from that \( FTable \) entry.

- There is a corresponding \( FTable \) entry and but all the sub-flows in that entry are marked as used by \( e \) - in that case we send \( e \) to \( FlowsHold \). \( e \) will remain there until one of the spearhead EX-flows will provide a full sub-flow for \( eq \).
**Full EX-flows**  The refined treatment of full EX-flows is detailed in algorithms 5 and 7. In `RefHandleFullFlow` we simply iterate over all the compound nodes of `e`, invoke `HandleFullPartialFlow` on each node and insert `e` to `Out`.

The real work is done at `HandleFullPartialFlow`. It receives as input an EX-flow, `e`, and a compound node of that EX-flow, `n`. It computes the sub-flow of `e` preceding `n` (line 1), `flow_pre`. `flow_pre` is used, in conjunction with `n`, to define the equivalence class `eq` (line 2), which is looked up in the `FTable` (line 3). If a corresponding table-row does not exist, we create one (lines 4-5). We then compute `flow_rooted`, the sub-flow of `e` rooted at `n` (line 7). We then search the table-row for `flow_rooted`. If it does not exist, we add it to the row (lines 8-9) and release from the `FlowsHold` all EX-flows in `eq`'s row (line 10).

We next illustrate how all components work together.

**Example 3.3.5**  Consider again the BP specification from example 3.2.2 and assume that we want to compute its top-2 EX-flows. Starting with the root activity `A`, `RefHandlePartialFlow` looks for its equivalence class in `FTable`. Since the table is empty, a new equivalence class containing only `A` is defined, and a row is generated for it in `FTable`. Then, the possible expansions of `A` are examined, and two possible expansions are added to `Frontier`: one full EX-flow (`A ⇒ S1`), corresponding to the implementation choice leading to `B`, and one partial EX-flow (`A ⇒ S0`), containing a recursive invocation of `A`. Next, `A ⇒ S0`, having the better `fWeight`, is popped from `Frontier`. The next activity to be expanded in is `A`, but as it appears in the table with no unused sub-flow, this expansion is removed from `Frontier` and stays on hold. Then `A ⇒ S1` is the only EX-flow in `Frontier` and is popped, then inserted to `Out` by `RefHandleFullFlow` (Indeed, `A ⇒ S1` is the top-1 EX-flow). Now, the entry in `FTable`, for the top-1 EX-flow rooted at `A`, is updated to be `A ⇒ S1`, and `A ⇒ S0` is “released” from `OnHold` and returned to `Frontier`, for computation of the 2nd-best EX-flow.

**EnumerateFlows**  To conclude, let us explain how `EnumerateFlows` extracts the EX-flows from their compact representation in `Out`. The EX-flows in `Out` contain activity nodes that point to entries in `FTable`, describing sub-flows. The nodes in each such description also possibly point to other table
entries and so on. \textbf{EnumerateFlows} thus simply follows these pointers to materialize the full EX-flow. This pointer chasing is guaranteed to terminate since, following observation B, pointers in \textit{FTable} induce no loops.

### 3.3.2 Complexity

We measure the complexity with respect to the input size and $|\text{equiv}|$, then relate $|\text{equiv}|$ to the three classes of cWeight functions, mentioned on the previous chapter. We assume that the computation of $\text{aggr}(x, y)$, for any $x, y$, takes $O(1)$ time.

We start by stating the run-time complexity of \textbf{RefinedFindFlows}, that generates a compact description of the top-k EX-flows, then continue with \textbf{EnumerateFlows}, that effectively enumerates them.

\textbf{Theorem 3.3.6} Given a BP specification $s$ (with cWeight and aggr) and a number $k$, if $|\text{equiv}|$ is finite, the time complexity of \textbf{RefinedFindFlows} in Algorithm \textbf{TopK} is polynomial in $|s|$, $k$ and $|\text{equiv}|$.

\textbf{Proof.} From Lemma 3.3.3, the number of entries in \textit{FTable} is $k \times |\text{equiv}| \times \text{linear}(|\text{equiv}|)$. Now, for each flow node $v$ considered during the course of the algorithm execution, either it already appears in \textit{FTable}, or it does not. The case where the sub-flow requested for $v$ does not appear in the table may only happen $k \times |\text{equiv}| \times \text{linear}(|\text{equiv}|)$ times, while computing the top-k EX-flows rooted at $v$. the cost of computation for such cases is $O(|\text{equiv}|)$ for searching the table, (assuming that we have an index that allows, in $O(1)$ time, to get the last (worst ranked) entry for a given row; otherwise there may be an additional factor of $k$) and then $O(1)$ of further computation considering direct expansions of $v$, a total of $O(k \times |\text{equiv}|^2 \times \text{linear}(|\text{equiv}|))$.

If the sub-flow considered for $v$ does already appear in \textit{FTable}, we only need to point the implementation of $v$ to the sub-flow that were already computed ($O(1)$). We next consider the number of times that this scenario may occur.

We start by considering the computation of the top-1 EX-flow. Now, consider some equivalence class $e$. Say that we have encountered some node $n \in e$, and then, before we are done computing the top-1 EX-flow rooted at $n$, we have encountered, at another point of the search tree, another node $n_0 \in e$. The course of the algorithm execution follows Observation B: it
suspends the computation for the top-1 EX-flow of $n_0$, until computation of the top-1 EX-flow of $n$ is done (by putting $n_0$ “on hold”). The number of such suspensions, while computing the top-1 EX-flow of $n$, is bounded by the size of the specification $s$, for each such $n \in e$ and for each $e$. The number of such equivalence classes is $|\text{equiv}|$. The same argument holds for computation of the $i$th highest weighted EX-flow, for each $i = 1, \ldots, k$, leading to a total bounded by $|s|^2 \ast |\text{equiv}| \ast k$ for this case.

The total complexity is thus polynomial in $|\text{equiv}|$, $k$, and $|s|$. □

We now give a brief overview of the complexity of the refined algorithm, for different $cWeight$ functions. For a more detailed analysis, the interested reader is referred to [18].

The size of $|\text{equiv}|$ is directly dependent on the $cWeight$ function. Recall that in the previous chapter we introduced three classes of $cWeight$ functions: history-independent, bounded-history and unbounded-history. We now consider the size of $|\text{equiv}|$ for the different classes of $cWeight$ functions.

**History-independent $cWeight$** Recall the if the $cWeight$ function is history-independent, then for every two EX-flows $e, e'$ and two activity nodes $n, n'$ (in $e$ and $e'$ resp.) that are labeled by the same activity name, $[e, n]$ and $[e', n']$ are equivalent. Thus, the number of equivalence classes $|\text{equiv}|$ equals to the number of activity names appearing in the BP specification $s$, hence is bounded by $|s|$. Consequently, the following proposition holds:

**Proposition 3.3.7** For history-independent $cWeight$ functions, $\text{RefinedFindFlows}$ is PTIME in $|s|$ and $k$.

**Bounded-history $cWeight$** Recall that if the $cWeight$ is bounded-history, with some bound $b$ on the history size, then the activity name that labels a node $n$, along with the sequence of the $b$ preceding choices, determines exactly the shape and cost of sub-flows that may originate from $n$. The number of such combinations of activity name and last $b$ choices, and thus $|\text{equiv}|$, is bounded by $|s|^{|b| + 1}$. Consequently, the following proposition holds:

**Proposition 3.3.8** For bounded-history $cWeight$ functions with bound $b$, $\text{RefinedFindFlows}$ is PTIME in $|s|$ and $k$, with an exponent depending on $b$.
The exponential dependency on the history size is inevitable unless $P = NP$ [18].

**Unbounded-History cWeight**  Here, the weight of a given choice may be different for each distinct history, and the number of such possible histories is infinite. In such case, the number of equivalence classes may be infinite and **RefinedFindFlows** may not halt. However this is inevitable [18].

To conclude this section, we consider the complexity of explicitly enumerating the top-k EX-flows.

**Theorem 3.3.9**  The complexity of **EnumerateFlows** is linear in the size of the output.

**Proof.** The proof follows immediately from the fact that the algorithm follows only pointers that appear in $Out$ and $FTable$, and which are part of the output EX-flows. □

### 3.3.3 Optimality

We next consider the optimality of **TopK**. We start by defining our optimality measures, then briefly present **TopK**’s optimality for weight functions of different properties. For a more detailed analysis, the interested reader is referred to [18].

**Optimality Measures**

We introduce the class of algorithms against which we compare **TopK**, the cost metric used for comparison, and the notions of optimality and instance-optimality.

**Competing algorithms**  We define the class $\mathcal{A}$ of all deterministic correct top-k algorithms that operate on the same input and have no additional information over **TopK**: an algorithm in $\mathcal{A}$ may only retrieve an EX-flow by multiple calls to **AllExpansions**. It may obtain the $cWeight$ of each expansion choice returned by these calls and can compute the $fWeight$ of the resulting EX-flows by applying $aggr$, but cannot use any other information that is not obtainable in the above manner.
**Cost metric**  We consider the number of calls to \textit{AllExpansions} as the dominant computational cost factor, as it indicates the number of distinct (sub-)flows generated and examined by the algorithm. The cost of an algorithm \( A \), when executed over an input instance \( I \) (denoted \( \text{cost}(A, I) \)) is thus defined as the number of calls it makes to \textit{AllExpansions}.

**Optimality and Instance Optimality**  Following [24], we use two notions of optimality (within the class \( A \)): \( A \in A \) is \textit{optimal} if there is no algorithm \( A' \in A \) and an input instance \( I \) such that \( \text{cost}(A', I) < \text{cost}(A, I) \). \( A \) is \textit{instance-optimal} if there exist constants \( c, c' \) such that for no \( A' \in A \) and instance \( I \), \( c \ast \text{cost}(A', I) + c' < \text{cost}(A, I) \).

**Optimality Results**

We show next that the (instance) optimality of our algorithm is influenced by properties of the global \textit{fWeight}.

**Strongly monotone \textit{fWeight}**  We say that \textit{fWeight} is \textit{strongly monotone} if for every two distinct (partial or full) EX-flows \( e, e' \), we have \( \text{fWeight}(e) \neq \text{fWeight}(e') \). In particular, this implies that the weight strictly decreases as the EX-flow advances. We prove the following.

**Theorem 3.3.10**  For strongly monotone \textit{fWeight} functions, \textbf{TopK} is optimal within \( A \).

**Semi-Strongly Monotone \textit{fWeight}**  We saw the optimality of \textbf{TopK} for strongly monotone \textit{fWeight} functions. However, in a realistic setting, some choices do not incur any change to \textit{fWeight}, e.g. in our shopping mall example, the choice of store or product type induce a zero added cost. Still, the number of consecutive choices that induce zero contribution to the \textit{fWeight}, is typically bounded. To model this we define \textit{semi-strongly monotone} \textit{fWeight} functions:

**Definition 3.3.11**  An \textit{fWeight} function is \textit{semi-strongly monotone} for a BP specification \( s \), if there exists some constant \( c \) such that for every EX-flow \( e \) of \( s \), \(|\{e' \mid e \rightarrow^* e', \text{fWeight}(e) = \text{fWeight}(e')\}| \leq c\).

**Theorem 3.3.12**  For semi-strongly monotone \textit{fWeight} functions, \textbf{TopK} is instance optimal within \( A \).
Indeed, we can also show that this is the best that can be achieved in this case. Namely,

**Theorem 3.3.13** No algorithm within $\mathcal{A}$ is optimal for all semi-strongly monotone $f$Weight functions.

**Weakly monotone fWeight** Finally, we consider the (not so common in practice) case of weakly monotone $f$Weight functions. Here users may perform an unbounded number of consecutive choices that incur no change to the EX-flow weight. Unfortunately, in this case our algorithm is not (instance) optimal, and we can show that in this case no (instance) optimal algorithm exists.

**Theorem 3.3.14** No algorithm within the class $\mathcal{A}$ is (instance) optimal for all weakly monotone $f$Weight functions.

### 3.4 Query evaluation (static)

The algorithms presented and analyzed in the previous sections solve the $\text{TOP-K-FLOWS}$ problem. In this section, and in the next, we present two approaches for solving the $\text{TOP-K-ANSWERS}$ problem, i.e. to find the top-k EX-flows conforming to a given query $q$.

Given a BP specifications $s$ and a query $q$, one way to solve the $\text{TOP-K-ANSWERS}$ problem is through the sequential execution of two algorithms.

1. The first algorithm is the query evaluation algorithm of [16] that, given a BP specification $s$ and a query $q$, constructs a BP specification $s'$, including only those EX-flows of $s$ that matches the query. Intuitively, $s'$ is the “intersection” of $s$ with $q$, obtained by considering all possible splits of the query into sub-queries, then matching these sub-queries to the DAGs in $s$.

2. The second algorithm is our $\text{TOP-K-FLOWS}$ algorithm, used to retrieve the top-k EX-flows of the just constructed $s'$.

**Creating $s'$** We will now explain how we create $s' = (S', s'_0, \tau')$ such that it will capture the set of query results.
First, consider queries without transitive nodes or edges. Let \( n_s \in s \) and \( n_q \in q \) be two nodes sharing the same (compound) activity name \( a \). We define a new activity name \([n_q, n_s, a]\). Note that a node \( n_q \) (resp. \( n_s \)) of the pattern \( q \) (BP \( s \)) may appear in several such new activities \([n_q, n_s^i, a]\) (resp. \([n_q^i, n_s, a]\)).

For compound (non-transitive) activities, the implementation \( \tau' \) of \([n_q, n_s, a]\) consists of all DAGs that represent possible embeddings of the direct implementation of \( n_q \) in \( q \) into the possible direct implementations of \( n_s \) in \( s \). The nodes in the resulting graph (for each of the possible embeddings) are labeled by triplets, as above, recording for every activity pair in \( p \), to which activity pair in \( s \) it was mapped in the given embedding. If no embedding was found, \([n_q, n_s, a]\) is marked as failure. The embeddings may be found using conventional algorithms for subgraph homomorphism. For efficiency, rather than constructing all the triplet activities names and their implementations, the algorithm operates in a top down manner. It starts by matching the pattern outer most activity with the BP root, building the corresponding \([n_p, n_s, a]\) activity. Then it compute its implementations, and the implementations of the activities appearing in them, and so on.

If the direct internal trace of \( n_p \) contains a node \( c_p \) labeled by a compound activity, such that \( c_p \) itself contains an internal implementation \( p' \), then the algorithm proceeds recursively as follows. In the previous step, \( c_p \) was matched to some node \( c_s \) (appearing in an implementation of \( n_s \)). As a result, an activity name \([c_p, c_s, c]\) was created, where \( c = \pi(\lambda(c_s)) \). Now, we consider all possible implementations of \( c_s \), and try to embed \( p' \) in each implementation. Each result will be added to the implementations set of \([c_p, c_s, c]\). We then proceed to the implementations of the compound activities appearing in the implementation of \( c_p \), and so on, recursively.

The recursion end when either when sub-pattern that currently needs to be handled is empty (in which case a successful match was found), or when no embeddings for the current sub-pattern are found (which we mark as failure). As a final step, the algorithm performs ”garbage collection” by recursively marking as failure activities for which all possible implementations contain failure activities, and then removing from \( s' \) all DAGs that contain failure such activities.

When the query contains transitive edges, we also define new activity name for every transitive edge \( e_q \in q \) and activity \( n_s \in s \). When the pattern
sub-graphs are embedded into the BP, the transitive edges \( e_q \in q \) (that connect two pattern nodes) are mapped to all possible paths in the BP (connecting the two corresponding BP nodes). In the output graph, a BP node \( n_s \) (with label \( a \)) that appear on such path is labeled by the triplet \([e_q, n_s, a]\).

Finally, when \( q \) contains transitive activities, the algorithm becomes somewhat more complex. Recall that transitive activities allow to navigate (transitively) inside the compound activities and query their internal EX-flow at any depth of nesting. Specifically, part of the direct implementation of \( n_q \) can be matched with the direct implementation of \( n_s \), while other parts may be matched at deeper levels of the implementation. To account for that, the algorithm considers all possible splits of the query into sub-queries, and the embeddings of those into the DAGs in \( s \).

### 3.4.1 Complexity & Optimality

The complexity of the first algorithm is \(|s||q|\) [16], and so is the maximal size of the resulting BP \( s' \). The second step, as theorem 3.3.6, is then polynomial in the size of its input \( s' \), and in \( k \) and \(|\text{equiv}|\), and is linear in the output size. That is, there is an exponential dependency between the complexity and the size of the query.

For completeness, we next present theoretical results related to the algorithm. These results are not considered a contribution of this thesis. It is shown in [18] that this exponential dependency is inevitable. It is also shown there that \textsc{Top-K-Answers} fails to be (instance) optimal, however, there exists no (instance) optimal algorithm for \textsc{Top-K-Answers} within \( \mathcal{A} \).

### 3.5 Query evaluation (on-the-fly)

A second approach for solving the \textsc{Top-K-Answers} problem is the “on-the-fly” query evaluation. In this method, the query evaluation is embedded into the \textsc{Top-K-Flows} algorithm. We modify the \( c\text{Weight} \) function such that EX-flow that do not satisfy the query will end up with a very low \( f\text{Weight} \). We then prune EX-flow with very low \( f\text{Weight} \) values, knowing that they do not satisfy the query.

\footnote{\( \mathcal{A} \) is adapted to the \textsc{Top-K-Answers} problem in a natural way, allowing algorithms to look at the entire query, but to access EX-flows only via \textit{AllExpansions}.}
Formally, let $0_{fWeight}$ denote the $fWeight$ threshold for pruning, i.e. EX-flow with $fWeight \leq 0_{fWeight}$ are pruned out.

Let us extend the requirements of aggregation functions, given in section 2.4, and consider only aggregation function that has a disqualifying value, denoted $0_{aggr}$. Namely, for each $x \in \mathcal{W}$, $aggr(x, 0_{aggr}) \leq 0_{fWeight}$.

The existence of that value allows us to use the $cWeight$ function to cause a pruning of an EX-flow (by giving one of its implementation choices weight of $0_{aggr}$).

**Example 3.5.1** For instance, for the $\times$ aggregation function where $\mathcal{W} = [0, 1]$ (corresponds to likelihood), $0_{fWeight}$ and $0_{aggr}$ could be 0. However, for practical reasons, we might decide that we are not interested in any EX-flow whose $fWeight \leq \epsilon$, for some $\epsilon \to 0$, and set $0_{fWeight}$ to $\epsilon$. In that case, $0_{aggr}$ could still be 0.

Unlike the case of $\times$ with $\mathcal{W} = [0, 1]$, in the case of the $-$ aggregation function where $\mathcal{W} = (-\infty, 0]$, $0_{fWeight}$ is more domain dependent, this is true whenever $\mathcal{W}$ is an open set. This is because any value which is small enough for some domain, might not be such for another domain.

### 3.5.1 Algorithm

As in the static-analysis based TOP-K-ANSWERS algorithm, presented in the previous section. The on-the-fly algorithm consists of running a TOP-K-FLOWS algorithm over a modified BP. This time, the TOP-K-FLOWS algorithm has to be modified as well. We first present the modification of the BP. We then present the modification needed when using the naive and when using the refine TOP-K-FLOWS algorithm. We shall refer to this algorithm as ON-THE-FLY.

**BP Modification**

Given a BP specifications $s = (S, s_0, \tau)$ and a $cWeight$ function $\delta$ we build upon them a new BP specifications $s' = (S', s'_0, \tau')$ and $cWeight$ function $\delta'$. The new root-DAG, $s'_0$ has one implementation, going to an extension of the previous root-DAG $s_0$. The extended $s_0$ consists of two connected activities. The first is the previous root-activity $actRoot$ (with all previous
implementation choices preserved). The second is actQuery, having a single implementation choice, mapping to a new single-activity DAG, called dagQuery. We extend $\delta$ by assigning $cWeight$ values to the new implementation choices. The $cWeight$ of the new root-activity’s implementation is always $1_{aggr}$. The $cWeight$ of the implementation choice of actQuery is $1_{aggr}$, if the preceding EX-flow satisfies $q$, and $0_{aggr}$ otherwise. This process is depicted in figure 3.5.

![Diagram of BP modification](image)

**Figure 3.5: Modifying the BP to allow on-the-fly query evaluation**

**Using the naive algorithm**

To solve TOP-K-ANSWERS over $s$, we use a modified TOP-K-FLOWS algorithm over $s'$. The modification lies in the $AllExpansions$ function, that given a partial EX-flow $e$ returns all EX-flows $e'$ s.t. $e \rightarrow e'$. The modified $AllExpansions$ returns only the EX-flows whose $fWeight$ values are greater than $0_{fWeight}$ (it essentially ignores (prunes) the other EX-flows, as they do not satisfy the query). To get the TOP-K-ANSWERS for $s$ we need to remove the first and last implementation choices (the implementations of $S'$ and of actQuery) from the TOP-K-FLOWS of $s'$.
Using the refined algorithm

This modification suffices for the naive FindFlows algorithm, however, the RefinedFindFlows algorithm requires another modification. First let us see the problem with using RefinedFindFlows. Whenever a new equivalence class \( eq \) is met during the computation, when considering some EX-flow \( e \), a corresponding entry in the \( FTable \) is created. This entry will cause other EX-flows, which start with \( eq \), to be sent to \( FlowsHold \). That is, until the expansions of \( e \) will provide a full sub-flow for \( eq \). When using on-the-fly query evaluation, we might drop all the expansions of \( e \). Thus causing some EX-flows to wait for \( e \)'s sub-flows, that will never arrive. We solve this by counting, for every \( FTable \) entry, how many EX-flows in the \( Frontier \) are spearheading the computation for that entry. We now have a forth case, when considering some partial EX-flow - there exist a table entry for the first expandable of that EX-flow but there are no EX-flow spearheading the computation for that EX-flow. We treat this case as if there is not \( FTable \) entry at all.

We now justify the correctness of the algorithm.

**Theorem 3.5.2** Give a BP specification \( s \), a query \( q \) and \( \text{TopK} \) as the underlying algorithm, \( \text{ON-THE-FLY} \) halts and returns the \( \text{TOP-K-ANSWERS} \) of \( q \) on \( s \), provided that \( |\text{equiv}| \) is finite.

**Proof.** Since \( |\text{equiv}| \) is finite, for any \( k \) \( \text{TopK} \) halts on \( s \) and returns the top-k EX-flows of \( s \). Consider the \( \text{TOP-K-ANSWERS} \) of \( q \) on \( s \). This EX-flows set is a subset of \( \text{TOP-r-FLOWS} \) for some \( r \geq k \). We could get \( \text{TOP-K-ANSWERS} \) by simply finding the \( \text{TOP-r-FLOWS} \) and then remove all EX-flows which are not satisfying the query. This is exactly what \( \text{ON-THE-FLY} \) does, with the exception that it throws the non-satisfying top EX-flows before treating them as full EX-flows (which includes the releasing of partial EX-flows from \( FlowsHold \)). Since we automatically release all EX-flows, corresponding to an \( FTable \) entry with no spearheading EX-flows, we essentially does treat those non-satisfying EX-flows as full, hence the algorithm does halts. \( \square \)

**Optimization**

We call this method “on-the-fly” because it prunes unsatisfying EX-flows during the top-k computation (with almost not preliminary computation).
But in the presented method, the actual pruning is essentially done at the end of the computation. That is, we do not check whether an EX-flow contradicts the query until the first expandable node of that EX-flow is \textit{actQuery}. By that time, had we queried over the original BP, that EX-flow would have been full. The problem is that we only check for query contradiction on \textit{actQuery}, when we could have checked it earlier. For instance, in the context of our online store. Suppose we are querying for purchases of a DVD player, which costs last 50\$. Every EX-flow, containing the choosing of a more expensive DVD player should be pruned out, the instance that player is chosen. However, in the presented method we will not prune that EX-flow, until it reaches \textit{actQuery}. We solve this by incorporating more “pruning points” in the BP. We can also modify the \textit{cWeight} functions of existing compound activities to essentially make them pruning points. This will enable us to prune EX-flows as soon as they violate the query, thus making the query process truly “on-the-fly”.

In our implementation we chose the “on-the-fly” approach due to its simplicity, the exemption of static analysis and the great performance achieved after using several pruning points.
Chapter 4

Incremental Computation and Diversity

4.1 Motivation

As we stated in previous chapters, when modeling an application using a BP specification, it captures all the navigation paths in the application. Every navigation path (sequence of actions/clicks) in the modeled application corresponds to an execution flow (sequence of activities) in the modeling BP. Top-k EX-flows querying of BPs is highly valuable and can be utilized for many tasks. In the context of our online store, for instance, from a user’s point of view, top-k querying could assist her in navigating the shop and attaining the best deals. We call a system, which utilizes top-k querying, for assisting users with navigating an application, a navigation recommender system (NRS, for short).

Let us start with an example, showing how top-k querying can be utilized for navigation assistance. Suppose a user wishes to purchase a fully operating PC, using our online store. The user has to sequentially choose compatible hardware components. Given the BP specification of the store, with price as the weighting function, querying it the for the top-10 EX-flows satisfying the query corresponding to the purchase of a motherboard, then a compatible CPU, than a compatible RAM, etc, reveals the navigation paths leading to the purchase of the 10 cheapest fully operating PCs. These EX-flows are essentially recommendations for how the user should navigate the shop in order to purchase a PC.
The user starts navigating the store, according to one of those recommendations (EX-flows), and chooses a motherboard. Out of the original 10 recommendations, only some are still relevant - those that start with the choosing of the same motherboard. To have more relevant recommendations and thus more information, the user has to execute another top-k query. This time, however, she is interested in the top-k EX-flows starting in correspondence to the actions she has already performed on the shop.

Finding these EX-flow by using the NaiveTopK or the TopK algorithms is easy, by simply initializing the Frontier with the shortest\(^1\) EX-flow starting in correspondence to the actions performed on the shop. It is straightforward to show that starting the computation with this EX-flow would result in the top-k EX-flows starting the same as this EX-flow, by treating the first expandable node of this EX-flow as the root of the BP.

After each action the user performs on the application, all the recommendations which do not continue with that action are no longer relevant. Relevant recommendations are obtained by executing a top-k query as mentioned above, i.e. initializing Frontier with the EX-flow corresponding to the user’s actions so far. Doing this process iteratively, i.e. executing such a top-k query after every action the user makes, is how a simple NRS provides navigation recommendations.

When using a NRS to navigate over some application, we first provide it with a BP specification of the application and a query representing the user’s goals. After that, an iterative process, like the one described above, in which the user makes an action on the application and requires new recommendations, takes place.

**Chapter outline** This process involves many top-k computations and thus might be costly. We would like to perform this computations as fast as possible. On section 4.2 we introduce the incremental computation mode, in which the computation state of a previous top-k computations is used to make a later top-k computations faster. We would like to produce useful recommendations, to assist the user as best as possible. On section 4.3 we introduce a problem similar to the **TOP-K-FLOWS**, whose solution provides recommendations which are more useful to the user.

\(^1\)in terms of number of implementation choices
4.2 Incremental Computation

A NRS executes many top-k queries over the same BP specification and query. The computation status, consisting of Frontier and Out in the case of the naive algorithm, and of FTable and FlowsHold as well, in the case of the refined algorithm, is lost after every query execution. We now show how we could use the computation status of a previous execution to optimize the performance.

Example 4.2.1 Observe the BP specification, given in figure 4.1. Let us assume an arbitrary user navigating an application, modeled by that BP specification, and using a NRS, with no query and k set to 2. Let us assume that the NaiveTopK algorithm is used. The user starts the application. The NRS, in turn, executes a top-k computation and outputs A ⇒ S1 B ⇒ S3 C ⇒ S5 and A ⇒ S1 B ⇒ S3 C ⇒ S6. Suppose the user advances by choosing S1. The NRS starts a new top-k computation. The computation starts by inserting the new prefix EX-flow A ⇒ S1 to the Frontier. The system outputs A ⇒ S1 B ⇒ S3 C ⇒ S5 and A ⇒ S1 B ⇒ S3 C ⇒ S6, same output as before.

The first top-k computation ended with the following status:
Frontier = {A ⇒ S1 B ⇒ S4, A ⇒ S2}
Out = {A ⇒ S1 B ⇒ S3 C ⇒ S5, A ⇒ S1 B ⇒ S3 C ⇒ S6}

When refined to EX-flows starting with A ⇒ S1:
Frontier = {A ⇒ S1 B ⇒ S4}
Out = Ø

This is exactly the computation status after the second top-k execution. We could have started the second top-k computation with this refined status, thus making no redundant computation.

Let us now assume the user chooses S4. The above computation status, when refined to EX-flows starting as A ⇒ S1 B ⇒ S4, is:
Frontier = {A ⇒ S1 B ⇒ S4}
Out = Ø

Again, we could have started the third computation with this refined status (although no computations would be saved this time).

For every two top-k computation starting with prefix EX-flows e and e’ correspondingly, where e →* e’, we can refine the computation status at the
end of the first computation to EX-flows starting as $e'$ and start the second computation with this refined status. This is justified in theorem 4.2.2.

**Theorem 4.2.2** Let $s$ be a BP specification and $q$ a query (possibly empty). Let $e$ and $e'$ be two EX-flows over $s$ such that $e \rightarrow^* e'$. Let $E_1$ and $E_2$ (for “End-status”) be the computation statuses after top-k computations (both are over $s$, with $q$ as a query) starting with EX-flow $e$ and $e'$ correspondingly.

If a top-k computation is started with $E_1$, refined by $e'$, the status at the end of the computation will be $E_2$.

**Proof.** The proof follows from the fact that $\rightarrow^*$ relation is transitive, that is, for every EX-flow $e''$ such that $e' \rightarrow^* e''$, also $e \rightarrow^* e''$. From that fact we know that $E_1$, reduced to EX-flows starting as $e'^2$, is a computation achieved by starting a computation with $e'$. When this status will be further expanded $E_2$ will result. □

We call this computation mode *incremental computation*. As we saw, employing it could save computations. The amount of computations saved

---

2In the case that this status is empty, we add $e'$ to it. In this case the theorem holds.
depends on the how the user continues and in general is higher as the user continues with higher ranked navigation options. Formalizing the notion of computations saving due to incrementally is work at progress.

4.3 Diversity

Let us assume an arbitrary user navigating in our online store. Let us assume the user’s goal is to purchase the cheapest fully operating PC possible. The user loads the BP specification of the store and a corresponding query to a NRS and starts navigating the store. Suppose the user starts by choosing a CPU for her new PC. The user is shown many different CPU models and requires assistant in choosing a CPU.

When querying for top-k EX-flows, we might get \( k \) EX-flows, all starting by choosing the same CPU (varying on later implementation choices). We saw this phenomenon in earlier examples. For instance, in example 4.2.1, the top-2 EX-flows, of the first computation, varies on \( C \) and not on \( A \), although the user is required to choose an implementation for \( A \). This is usually not a desired result. The user is currently choosing a CPU so the top-k EX-flows should be with respect to the chosen CPU. That is, we would like to output one EX-flow for each of the top-k CPUs the user can choose, i.e. those that choosing them will enable buying the cheapest overall PC. We do this by considering, for every possible CPU, the top-1 EX-flow staring in correspondence to the user’s actions so far and continuing with the choosing of that CPU. From those top EX-flows, we consider only the top-k. These are the EX-flows we would like to use in order to generate recommendations for the user.

Formally Given a BP specification and an EX-flow prefix (corresponding to the series of actions taken by the user so far). Let \( first \) denote the first expandable node prefix. Let \( imps \) denote the set of possible implementation choices of \( first \). For every implementation choice \( imp \in imps \), we consider the top-1 EX-flow starting as prefix and continuing by choosing \( imp \) as the implementation of \( first \). Among those top EX-flows, we consider only the top-k. We call the problem of finding these k (top) EX-flows the \textsc{Top-k-Diverse-Flows} problem.

This name derives from the fact that these EX-flows differ (diverse)
on choosing the implementation of first. Sometimes there are less than \( k \) implementation in imps. In such cases we are only interested in the \( \min(k, |imps|) = |imps| \) top EX-flows (of the top-1 EX-flows).

The \textit{diversity} of an EX-flows set is the number of different, with respect to the implementation choice of \textit{first}, EX-flows in that set. We will use this term in the next section, to present our solution to the \textsc{TOP-K-DIVERSE-FLOWS} problem.

4.3.1 Algorithm

We now present an algorithm for solving the \textsc{TOP-K-DIVERSE-FLOWS} problem. It is based using a \textsc{TOP-K-FLOWS} algorithm iteratively and modifying its computation status. The algorithms differs slightly when using the naive or the refined top-k algorithm as the underlying \textsc{TOP-K-FLOWS} algorithm. The boxed lines are for the refined top-k algorithm.

\begin{algorithm}
\caption{Diversity}
\textbf{Input:} \( dstDiv, prefix, status \)
\begin{algorithmic}[1]
\While {true}
\State \( prefixLen \leftarrow \text{number of implementation choices in prefix} \); \label{line:prefixLen}
\State \( prefixes \leftarrow \text{all different prefixes of length } prefixLen + 1 \text{ in } status.Out; \)
\If {\( |prefixes| \geq dstDiv \)} \label{line:diversity}
\State break \; \label{line:break_diversity}
\EndIf
\State move all flows from \( status.Out \) to \( tmpOut \); \label{line:move_Out}
\ForEach {\( prefix itr \in prefixes \)} \label{line:foreach_prefix}
\State move all flows in \( status.Frontier \) starting as \( itr \) to \( tmpFrontier \); \label{line:move_Frontier}
\State move all flows in \( status.FlowsHold \) starting as \( itr \) to \( tmpFlowsHold \); \label{line:move_FlowsHold}
\EndFor
\If {\( |status.Frontier| = 0 \)} \label{line:frontier_empty}
\State break \; \label{line:break_frontier_empty}
\EndIf
\State TopKFlows(1, \( status \)) \; \label{line:topKFlows}
\EndWhile
\State move all flows from \( tmpFrontier \) to \( status.Frontier \); \label{line:move_from_tempFrontier}
\State move all flows from \( tmpFlowsHold \) to \( status.FlowsHold \); \label{line:move_from_tempFlowsHold}
\end{algorithmic}
\end{algorithm}
**Analysis**  Let us now explain the algorithm. The algorithm receives as input a target diversity \((dstDiv)\), a prefix EX-flow \((prefix)\) and a top-k computation status\(^3\) \((status)\). The algorithm then enters a loop. In each iteration it first computes all the different prefixes in \(status.\text{Out}\) (line 3). The prefixes are of length \(prefixLen + 1\) which corresponds to the prefix EX-flow plus the next implementation choice. If this number of prefixes is greater than or equal to \(dstDiv\) (line 4) than we are done and we break from the loop (line 5). Otherwise, we continue by moving all the EX-flow of \(status.\text{Out}\) to \(tmp\text{Our}\) and all the \(status.\text{Frontier}\) EX-flows starting as one of those prefixes to \(tmp\text{Frontier}\) (line 8-9). If \(status.\text{Frontier}\) has emptied due to the refinement, there is no point in continuing, as we have already found all the possible continuations to the prefix EX-flow (line 12). If this is the case, we break from the loop (line 13). Otherwise, we end the iteration by starting the top-k algorithm on the refined status, trying to find one more diverse EX-flow (line 15). The found full EX-flow is guaranteed to be diverse from the other full EX-flow because all the partial EX-flows starting as those EX-flow were moved out from \(status.\text{Frontier}\).

After breaking from the loop, due to either satisfying our diversity goal or due to emptying the frontier, we return all the moved EX-flows back to \(status.\text{Frontier}\) (line 17). We do this because these EX-flows might be used for future computations. In the context of the CPUs example. Before returning the EX-flows to \(status.\text{Frontier}\), all the EX-flows there corresponded to choosing the last CPU found. We need to return the EX-flows in case the user chooses another CPU.

**On the refined algorithm**  When using the refined top-k algorithm as the underlying \texttt{TOP-K-FLOWS} algorithm, the boxed lines (10 and 18) are put into use, these are simply for moving EX-flow from the \texttt{FlowsHold} to \texttt{tmpFlowsHold} and back.

**Correctness**  The algorithm is correct, both in the case of the naive of the refined algorithm, for similar arguments to those presented in the proof of theorem 4.2.2.

\(^3\)The constitution depends on the underlying algorithm
**Complexity** The algorithm consists of $dstDiv$ invocations of a $\text{TOP-K-FL OWS}$ algorithm. All the complexity results of chapter 3 holds here as well, up to a multiplication by a constant.
Chapter 5

The ShopIT Recommendation System

5.1 Overview

So far, we presented a model for business processes abstraction (chapter 2), algorithms for performing top-k queries over it (chapter 3) and introduced the notion of NRS (chapter 4). We next describe the “ShopIT” recommendation system which employs the above concepts to provide recommendations for navigation.

We developed an online shop, simulating the computers department of Yahoo! Shopping [49]. The shop is based on real-life data obtained through the API [48] provided by Yahoo! Shopping. We loaded ShopIT with a BP specification of the shop.

We developed interfaces for communication between ShopIT and the store and built a “ShopIT empowered” shop, featuring recommendations by ShopIT. We presented the ShopIT empowered store at the 35th Very Large Data Bases convention in Lyon, France.

Chapter outline We start by introducing our online store (section 5.2). We than, in section 5.3, assume an arbitrary user shopping in online store. The user’s goal is to purchase a fully operating Pentium 4 based PC, of cheapest overall price. We than explore how this user may (try to) achieve this goal, using the regular online shop. We stress that the regular shop has no special features in it and it represents how things are done today. After
seeing that it is not an easy task, we see, in section 5.4, how this goal can be achieved much easily, using the ShopIT empowered store. We stress that the ShopIT empowered store is the same as the basic store, but with our NRS featured in it. We conclude this chapter by discussing the architecture used for integration the store and ShopIT.

5.2 The Shop

We start by introducing our sample online store and its user interface. Our online store is a mimic of the computers department of Yahoo! Shopping [49]. Our store and Yahoo! shopping have quite a different appearances, however the products and the pricing are the same\(^1\). Another difference is that in our shop we consider the price, the shipment time and the popularity of the different items, as we explain below. In Yahoo! Shopping, however, only the price is considered.

The store offers various hardware components, e.g. motherboards (MBs), processors (CPUs), random access memories (RAMs), hard-drivers (HDs) and Graphics Processing Units (GPUs). For every component the store offers many models, with a total of more than 12,000 distinct models. Every individual model is offered by one or more suppliers, with possibly varying prices and service qualities.

The store offers coupons, which when bought can affect the price of other products. Different coupons might have different benefits, e.g. 10% off, 15% off, 2 + 1, etc. Coupons are not always free and might be priced differently.

For every offer, we consider three parameters: price, shipment time and popularity. The price and shipment time of an offer are self-explanatory. The popularity of an offer is the probability that an arbitrary user will choose that offer among all the other offers of the same product. That is, if there are two offers for a Intel CPU, their polarities might be 33% and 67%. We also consider the popularity of a product, which is the probability that an arbitrary user will choose that product among all the other products of the same category.

The price is measured in United-States Dollars and was obtained through the Web API of Yahoo! Shopping. The shipment time is measured in days and is randomly generated. The popularity measure is also randomly gen-

\(^1\)to the time of the data extraction
erated. To the best of our knowledge, Yahoo! Shopping does not provide neither shipment time nor popularity information, hence the random generation.

5.3 Navigating without ShopIT

Let us consider the following scenario. Some arbitrary user wishes to purchases a fully operating Pentium 4 [29] based PC, with cheapest overall price. The user also requires a 512MB RAM and a 80GB hard-drive. A fully operating PC is a combination of a motherboard, cpu, ram, hard-driver and graphic-card that are compatible to one another. Finding this deal is not an easy task. Aside from the compatibility issues, a user should also consider the coupons, which could enable a cheaper deal. There are also technical-financial issues a user should be aware of, for instance, purchasing of some cheap motherboard might reduce the choice of a compatible RAMs to very expensive ones.

We now present the path a user might take to find the desired deal. Our user employs a greedy approach. In every category, she chooses the cheapest component of that category, which is also compatible to the components she has previously chosen.

In figure 5.1 we see the start page of the shop. In this figure we see two highlighted parts, marked “1” and “2”. The former is the menu panel, through which the user can navigate the different product categories. The later is a link to the coupons page (not available in the menu panel).

Our user ignores the coupons and starts by choosing a motherboard, arriving to the page depicted in figure 5.2. On the right we see a list of models, accompanied by their prices. On the left we see a list of attributes, through which we can refine the models list on the right. We can also change the ordering of the products, either by price, by shipment time or by popularity. Note that the products are already refined\(^2\) for our criteria of a Pentium 4 based PC and sorted by price. The user chooses the first motherboard, being the cheapest one.

The user arrives to the page depicted in figure 5.3. This is the product’s page. In this page we see the product’s technical specifications and the list of offers for that product. The offers might vary at price, shipment time and

\(^2\)For demonstration purposes, the refinements were preloaded.
popularity and we can change the ordering of the offers. Interested in the cheapest overall price, the user chooses the first offer.

Next, the user arrives to the page depicted in figure 5.4. This is the shopping cart page. We can see that the motherboard was indeed added to the shopping cart. The user decides to continue by choosing a RAM.

The user now arrives to the page depicted in figure 5.5, the RAMs page. Just like in the motherboards page, the models are already sorted by price and refined to user’s wish. However, the models might not be compatible to
the recently chosen motherboard. We chose a motherboard which uses the RDRAM [39] memory technology, so we must choose a RDRAM RAM chip. Using the refinement menu on the left, the user refined the list to RDRAM chips, the results are shown in figure 5.6. The price of the cheapest chip, which was 13.99$ before the refinement, is now 75.7$.

To achieve the goal of a cheapest deal, the user must start the whole process again, and choose a different motherboard, one that uses a cheaper
memory technology. In the next section, we show how the same goal can be achieved by using the ShopIT empowered store.

5.4 Navigating with ShopIT

We continue with the quest of our user to find the cheapest deal for a Pentium 4 based PC. In the previous section we have seen an attempt to
fulfill this goal, using the Yahoo! Shopping mimic store and saw that it is not an easy task. In this section, we see how it can be done, when using the ShopIT empowered store.

The first page of the ShopIT empowered store is shown in figure 5.7. The store can function exactly as the standard store, however it also features the ShopIT NRS. ShopIT is activated using the “activate ShopIT” button, to the right of the menu panel.

Once activated, the user arrives to the query page, depicted in figure 5.8. In this page, the user specifies her criterions for the entire PC. There are two types of criterions: global and local. Global criterions relate to the
Figure 5.6: the RAMs page - refined

entire deal, e.g. the entire PC should cost between X and Y dollars, the entire PC should arrive between X and Y days, etc. Local criterions relate to individual components, e.g. the processor should be Intel Pentium 4, the RAM should be 512MB in size, etc. The user can also specify how to rank the recommendation, either by total price, by total shipment time or by total popularity\(^3\). The checkboxes to the left of every products category are used to enable the user choose which components she is interested at.

\(^3\)the popularity of some deal corresponds to the probability of choosing this deal, from all the other deals.
Once the user has entered her requirements to the query page, she clicks the “compute recommendations” button (not visible in the figure), which brings her back to the start page, depicted in figure 5.9. On the right, we see the start page, as it was before. On the left, we see ShopIT’s recommendations. Each recommendation is a series of steps, corresponding to an EX-flow in the shop’s BP specification, which when followed will result in the purchase of a PC, satisfying the user’s requirements. The PC’s components are guaranteed to be compatible. For each recommendation we see its
Figure 5.8: the query page

$fWeight$, which in the case of price weighting, corresponds to the price of the entire PC bought, when the recommendation is followed step-by-step. Each recommendation is accompanied by an “execute” button, which when pressed sequentially executes all the step of the recommendation.

In figure 5.9 we see two recommendations. The first starts by going to the coupons page, while the second starts by going directly to the motherboards page. The overall price of the two recommendations differs slightly, due to the purchase of a coupon in the first recommendation and not in the second. When mouse-overing the first, non executed, step of a recommendation, the relevant part of the page, in which the action takes place, is highlighted. In figure 5.9 we mouse-over the first step of the first recommendation, hence the link to the coupons page is highlighted.
The user performs the first four steps of the recommendations. These steps are: going to the coupons page, choosing the 15% off coupon, choosing the offer by ServerSupply.com and going to the motherboards page. After executing these steps, the user arrives to the motherboards page, depicted in figure 5.10. On the left, we see ShopIT’s recommendations. The first four steps, of every recommendation, are faded, as they have already been executed by the user. On the right we see the motherboards list. The ordering of the motherboards has changed, they are no longer sorted by their price, but by the price of the cheapest PC, the user could attain, had she chosen that motherboard. The RDRAM motherboard, which was the first before, is now the 66th motherboard. The user is now equipped with the knowledge of how each choice she makes will affect the entire outcome.
On the products list we currently see the top-5 motherboards, with respect to the best PC that purchasing them enables. Below the products list we see a pagination panel. Through this panel we can request more recommendations.

![Motherboard Page with ShopIT Recommendations](image)

**Figure 5.10:** the motherboards page, with ShopIT’s recommendations

Suppose the user is indeed content with the first recommendation. She presses the “execute” button, adjacent to that recommendation. This brings her to the screen depicted in figure 5.11. This is the checkout screen, in which she can review the products she chose (using ShopIT as her proxy). ShopIT’s last recommendation is to press “checkout”, hence is highlighted.

In this section we have covered the main features of the ShopIT NRS and seen the added value to the user. In the next section we go “under-the-hood”
and examine the main components of the system and their interaction.

5.5 Under the Hood

The system’s architecture is given in figure 5.12. Let us now overview the main components and the interactions between them. This architecture can be applied on any online application and NRS, provided that they provide the necessary interfaces.

Query Engine The query engine provides top-k abilities to ShopIT. It has two components, a top-k algorithm component and an incremental computation component. It is initialized with a BP specification received from the store abstract model and a query and a ranking measure received from ShopIT. Once initialized it provides top-k computations for ShopIT, upon receiving of user actions.

Store Abstract Model Here we store the BP specification of the store. The process of model extraction is currently done by hand, with ad-hoc
tools. We note however that, in general, many Web-based applications are specified in declarative languages such as BPEL [8] (the standard for Web-based business processes) and then an automated extraction of their abstract model structure is possible [6]. How to perform an efficient model extraction is reserved for future work.

**ShopIT and Store** These components are very tight to each other hence presented together. The processes these two takes a part of are:

- The user interacts only with the store, which provides her with screen. Every action the user makes is reported by the store to ShopIT, which in turn passes the information to the query engine, in order to generate fresh top EX-flows. These EX-flows are returned to ShopIT, which provides the store with API for accessing them. The store renders the EX-flows as recommendations.

- ShopIT requires query and ranking information from the user. The user provides this information in the query page. This page is rendered by the store, according to instructions from ShopIT passed by a designated API. The store receives the query data and passes it to ShopIT.

- Some optional APIs between ShopIT and the store are needed for features such as highlighting of parts of the store page when mouse-overing the first step of a recommendation and sorting of products by
ShopIT.

The architecture presented above was used on our demonstration. It requires an “in-house” integration of ShopIT and the online application. For cases where access to the application’s internals is limited or not available, for instance, we might want to provide recommendations to the real Yahoo! Shopping Web-site, another architecture has to be used. The application has to provide different interfaces for the different features of ShopIT. For instance,

- The NRS needs to be informed on every action taken by the user on the online application. The online application has to provide a way for knowing when and which actions are taken.

- The execute-button feature (for executing the actions of an EX-flow) requires the online application to provide interface which receives an EX-flow, in some representation, and executes the actions of that EX-flow.

- The highlighting feature (for showing the user where an action takes place) requires the online application to provide a way of referencing the different elements of the page. For instance, in our implementation, different parts of the page were marked using the HTML “DIV” tag, which defines a division or a section in an HTML document [12]. ShopIT uses this information in order to highlight different parts of a page.
Chapter 6

Implementation & Experiments

6.1 Implementation

We implemented the TopK and the NaiveTopK algorithms and a NRS which supports incremental and diverse computation modes. The applications were developed in C++, under Windows XP. Most of the software was written using the C/C++ standard library, however, we used several third-party software packages. A list of third-party software packages we used is available on appending A.

We next give a very brief overview of non-trivial implementation details.

History model

To model non-trivial application, the cWeight function has to be history-dependent. That is, the cWeight of a guarding formula may vary, depending on the history. The cWeight computation is executed numerously during every top-k query evaluation so it has to be as efficient as possible.

Previously, we treated cWeight functions as black-box functions, receiving a history (given as an EX-flow) and outputting some weight. We now explain how this computation is done. A cWeight function is essentially a list of weight rules. Each weight rule consists of a condition on the history and a weight. When the condition of a weight rule is satisfied, its weight is chosen as the cWeight value. Computing cWeight is done by enumerating over the weight rules, evaluating their condition and outputting the weight
of the first satisfied weight rule. We assume that at-least one weight rule is satisfied for every history.

We now explain how we implemented the weight rules model. Every EX-flow was accompanied with an environment - a mapping of variables to values. We use numeric (doubles) and string values. Every guarding formula was accompanied with a series of commands to the environment, e.g. addition, subtraction, etc. Whenever we expand an EX-flow \( e \) with some implementation, guarded by guarding-formula \( F \), we execute \( F \)'s environment commands over \( e' \)'s environment. The weight rules relates to the variables of the environment. That is, we evaluate \( F \)'s weight rules over \( e' \)'s environment (prior to applying \( F \)'s environment formula). If the result is not zero than the rule is satisfied. For instance, the "The brand of recently purchased motherboard is Intel" rule can be easily modeled and evaluated. For every guarding formula representing the purchase of a motherboard we would add a \( \$brand = "x" ; \) command, where \( x \) is the brand of the corresponding motherboard. The condition of the weight rule would simply be \( \$brand == "Intel" ; \).

The weight rules and environment commands are written in a C-like language. During the BP generation phase every expression in that language is parsed using a bison/flex [35] parser to an Abstract Syntax Tree (AST). At run time, these ASTs are used to efficiently execute and evaluate the expressions over some environment.

Client-Daemon connection

To support incremental computation a client-daemon\(^1\) architecture is required. Such an architecture is required to enable the preservation of the computation states while allowing for other applications (users) to easily request top-k queries. In such architecture the user uses the client to send her requests. The client communicates with the daemon using an Interprocess Communications (IPC) channel, through which it sends requests and receive replies. The IPC channel chosen in our implementation is shared memory. To synchronize the communication, the method described in [46] was used.

\(^1\)program that runs in the background
Memory Management

The evaluation of a single top-k query could require millions of object allocations and de-allocation. Using the default memory allocator results in a memory fragmentation and as a consequence in a constant performance degradation. The allocation/deallocation of the “active” objects, i.e. those that are allocated and deallocated often, such as EX-flows, was done using a memory pool. The initial sizes of the memory pools were determined empirically and grows exponentially when needed.

6.2 Experiments

We present an experimental study of the proposed algorithms based on synthetic and real-life data. The study evaluates the performance of the TopK algorithm in practice relative to the NaiveTopK algorithm and the worst-case bound implied by our analysis, examining cases where optimality is guaranteed as well as cases where it is not.

The worst-case bound of TopK is reached when all entries of FTable are filled. To assess how much time this consumes we implemented a variant of TopK, called WC (for worst-case), that ignores the early stop condition and continues the processing until the table is full, and compared the performance of TopK to WC. Performance-wise, WC is similar to the algorithm in [17] which also constructs the full EX-flows table (the difference being that [17] considers likelihood of occurrence as the sole weighting scheme). A comparison of TopK to WC thus provides a comparison to [17] and demonstrates the significant performance gains achieved by our new algorithm.

We ran our experiments on a Lenovo T400 laptop, with Intel Core 2 Duo P8600 processor and 2GB RAM. We ran two series of experiments. First, we used synthetic data to vary the main parameters that may affect the relative performance of the algorithm. Second, we used real data, in the context of the Yahoo! Shopping Web-site, to evaluate how the performance compares in a real life setting.

6.2.1 Experiments with Synthetic Data

A balanced BP specification is a BP in which:

- Every activities-DAG has one activity in it.
• Every activity is either atomic or has a constant (\textit{width}) number of different implementations. And,

• The length, in terms of implementation choices, of all full EX-flows is constant - \textit{depth}.

The number of activities in a balanced BP specification of \textit{width} = \textit{w} and \textit{depth} = \textit{d} is \( \frac{w^d - 1}{w - 1} \) (sum of a geometrical column).

Experiment 1

The first experiment evaluates all three algorithms over a series of balanced BP specifications of \textit{width} = \textit{depth} = 7 (137257 activities). The BPs varies on the cWeight values generation. In every BP, the cWeight values are randomly generated from \textit{Uniform}(0, b) for some \( b \leq 1 \). We tested the algorithms for BPs with \( b \) ranging from 0 (all cWeight values are 0) to 1. The aggregation function is \( \times \).

Figure 6.1 shows the number of expansions that each algorithm required for performing a top-10 computation. The x-axis corresponds to \( b \) and the y-axis to number of expansions. Figure 6.2 shows how much time each algorithm required. Here, the x-axis corresponds to \( b \) and the y-axis corresponds to seconds.

![Graph showing expansions required](image)

Figure 6.1: Expansions required as a function of distribution size
In this set of computations, the \texttt{NaiveTopK} algorithm outperformed the \texttt{TopK} algorithm in terms of time and performed the same in terms of expansions count. This is due to the structure of the BPs which does not allow for any of \texttt{TopK}’s optimizations to matter. On the contrary, the \texttt{FTable} operations generate extra computation time, resulting in poorer performance.

The \texttt{WC} algorithm performs the same for all the distributions. This is because it expands all the EX-flows, no matter what, hence is not affected by the cWeight values. The performances of both the \texttt{NaiveTopK} and the \texttt{TopK} algorithms worsen as \( b \) gets smaller. As \( b \) gets smaller the variance of the cWeight values also gets smaller. As a result, the variance of the fWeight values also gets smaller. Hence, the algorithm has to expand a larger number of EX-flows. This is explained by the fact that when a top ranked EX-flows gets expanded, the number of EX-flows which are now higher ranked than it increases as the variance of the fWeight values decreases. Indeed, we see that all three algorithms performs the same for \( b = 0 \).

\textbf{Experiment 2}

Recall example 3.2.2 from chapter 3 (page 22). We used this example to illustrate how \texttt{TopK} could outperform \texttt{NaiveTopK} over recursive BP specifications. In this experiment we evaluated the two algorithms over a series of
example-3.2.2-like BPs. The BPs differ on the cWeight of guarding formula $F_2$ which ranges from 0.1 to $1.00E-10$.

Figure 6.3 shows that indeed NaiveTopK requires more expansions, as the cWeight of $F_2$ decreases, while TopK requires a constant number of expansions, independent of $F_2$'s cWeight.

Figure 6.3: Expansions required as a function of $F_2$'s cWeight

### 6.2.2 Experiments with Real Data

Our second set of experiments considered a real-life BP, modeling part of the Yahoo! Shopping Computer Store [49]. We obtained from the site (through a Web interface that it offers) all the products pricing information (including deals, reductions, etc.), and defined cWeight to capture this information. The resulting BP specification consists of 12,026 activities. The variance in cWeight values (costs) of choices for each compound activity (product type) is very high, e.g. the average RAM price is 192$, with a standard deviation of 510$. The full BP specification has approximately 2,000 trillions ($1.00E12$) equivalence classes.
Experiment 3

In this experiment, we measure the performance of NaiveTopK and WC over a series of small pieces of the ShopIT BP. These smaller BPs were created by limiting the number of products in every category. Figure 6.4 shows that NaiveTopK drastically outperforms WC, although neither is suitable for real-time applications.

![Figure 6.4: Time required as a function of equivalence classes](image)

Experiment 4

In this experiment we test the TopK algorithm over larger pieces of the ShopIT BP. This time the BPs are much larger, measuring at trillions of equivalence classes, instead of thousands. Figures 6.5 and 6.6 show how TopK performs over these BPs. Extrapolating the results of the previous experiment reveals TopK outperforms greatly the other algorithms. We could not provide a timing data for NaiveTopK and WC for these data sets as they crash due insufficient memory.

Experiment 5

In this experiment we test the incremental computation mode. Recall that when using a NRS, we execute a series of top-k queries. In incremental
computation we refine the computation status of a previous top-k computation and start the next computation with it. Figure 6.7 shows the run time of several successive top-k computations, with and without gradation. We see that the first computation takes the same amount of time in both computation modes. From the second computation, however, the incremental
computation mode drastically outperforms the non-incremental computation mode.

In this experiment we also evaluated our on-the-fly query evaluation. For all the top-k computation, the query of Pentium 4 based PC with 512MB RAM and 80GB hard-drive was enforced. This computation took only 25% of the time a full, that is, without a query, top-k computation took.

![Figure 6.7: Incremental vs Non-Incremental computation](image)

**Figure 6.7: Incremental vs Non-Incremental computation**

**Experiment 6**

In our final experiment we test the operation of our **TOP-K-DIVERSE-FLOWS** algorithm. We measured the time it took to our algorithm to find the top-20-diverse-flows. The diversity is in the choosing of a motherboard. Figure 6.8 shows how much time it took for every top-1 EX-flow. Clearly, the bulk of the computation was done for finding the top-1 EX-flow and from than very little computations were done for any additional diverse EX-flow.
Figure 6.8: Performance of TOP-K-DIVERSE-FLOWS
Chapter 7

Related Work

In this chapter we present work related to the work presented in this thesis. The work described in this thesis contains elements from the fields of

Application Abstraction

Probabilistic Databases (PDBs) [14], [44] and Probabilistic Relational Models (PRMs) [26] allow representation of uncertain information, but consider relational data and do not capture the dynamic nature of flow and the possibly unbounded number of recursive (possibly dependent) invocations. In terms of the possible worlds semantics, the number of worlds in our model is infinite (rather than large, yet finite, in PDBs). Extensions of PRMs to a dynamic setting, called Dynamic PRMs [41], do not allow for practically efficient algorithms. Probabilistic XML [3], [31] bears some resemblance to our model: the data is graph (tree) shaped, and it allows some dependencies between probabilistic events. However, our model is more complex: first, it represents nested DAG structures, rather than trees, entailing more intricate dependencies between events. Second, potentially infinite number of such nested DAGs are represented, due to possible recursive calls.

Top-K querying

Top-k queries were studied extensively in the context of relational and XML data [28]. Notably, [24] presented an instance-optimal algorithm for top-k queries that aggregate individual scores given to joining tuples, and had many follow ups [5, 45]. In our context, one may think of the cWeight as the equivalent of an individual score, and of fWeight as the aggregation of
cWeight values along a given EX-flow. Difficulties specific to our settings are that

1. The size of a given flow, thus the number of aggregated scores, is unbounded.

2. The particular properties of the cWeight functions are unique to EX-flows.

3. The number of items (EX-flows) that are ranked is infinite. The key challenge, thus, is to find a “small world” whose examination suffices for identifying the top-k EX-flows.

Note that while an infinite setting also appears in top-k queries over streamed data [32], works in this context aggregate over a bounded size sliding window, whereas we consider aggregation over flows of unbounded size.

Ranking by likelihood was also studied in several other settings. Probabilistic Databases (PDBs) [14, 44] and Probabilistic XML [3, 31] extend relational databases and XML, respectfully, to a probabilistic setting. For example, [44] and [31] study the problem of retrieving the top-k query results for queries over PDBs and Probabilistic XML, respectfully. Note that in contrast to relational data and XML, our model for BP flows allows representation of an infinite number of items, out of which the top-k are retrieved. Works on Probabilistic process specifications (like Markov Chains [30], Probabilistic Recursive State Machines [26], Stochastic Context Free Graph Grammars [13, 37], etc.) either suffer from low expressivity [23] or incur infeasibility of query evaluation [9].

Recommender Systems

We have already mentioned popular shopping Web-sites such as [49, 21]. Unlike ShopIT, their ranking mechanism ranks, separately, items in each distinct category, based on built-in specific ranking metric, e.g. price, popularity. The global effect of a full navigation flow that may include, e.g., registration to customers clubs, collection of coupon discounts, specific user choices, is not accounted for.
A variety of Recommender Systems (e.g. [42, 47]) appear in the literature. However, as mentioned in [28], they provide rather low flexibility, with a recommendation method that is hard-wired and not configurable to fit user needs\textsuperscript{1}. These works also typically do not support recommendations on multiple items. Successful commercial tools such as [36, 34] share similar characteristics and often specialize in specific domains, e.g. movies, music, restaurants. In contrast, we propose here a flexible generic approach that addresses the common, multi-item, shopping scenario and identifies navigation flows that best match the users criteria and preferences. The importance of customizable recommendation systems was recently recognized in [33], where such a flexible system was introduced in the context of relational data. The (possibly recursive) semi-structured shape of Web-applications introduces unique challenges for top-k computation, that are not found in a relational environment [19].

\textsuperscript{1}An exception are OLAP-based approaches that are still considered an open research problem.
Chapter 8

Conclusion and Future Work

We presented, in this work, a novel model for weighted BPs and explored top-k query evaluation in this context. We showed a naive algorithm for top-k querying and explored its drawbacks. Upon it, we developed a refined algorithm, which solves its predecessor’s drawbacks. We introduced a query language for BPs and showed two algorithms for finding the top-k EX-flows satisfying a query.

We introduced the notion of Navigation Recommender Systems as systems for assisting users with the navigation of complex applications. We showed how such a system can be built upon our BPs model and our top-k algorithms. We showed a way to perform incremental top-k computation and used it in the context of NRSs. We showed how useful and diverse recommendations can be created.

We implemented our top-k algorithms and built a NRS upon them. We implemented a sample online computers shop and integrated it with our NRS. We showed that the NRS empowered version of the store offers a better user experience than the regular version of the store.

Future Work

Several issues are still open and are left for future work. We briefly overview some of them:

- **Weight functions** - So far, we have limited our discussion to weight functions satisfying several requirements, the most limiting of them being the monotonicity requirement. Extending our algorithm to other weight function is an interesting problem.
• **Pre-fetching** - When using incremental top-k computation, we could use the time between computations to further expand the state of the previous execution. That is, after a top-k computation is done, we could continue to expand EX-flows, possibly finding more full EX-flows. A later computation would then have more information and might require less computations. Knowing how to continue the expansions and how much expansions to make is a very interesting problem.

• **Model Extraction** - In order to use our NRS on an application, we first need to model the application using our BPs model. The process of model extraction is currently done by hand. Automation can be achieved with the help of ad-hoc tools. We note however that, in general, some Web-based applications are specified in declarative languages such as BPEL [8] (the standard for Web-based business processes) and then an automated extraction of their abstract model structure is possible [6]. How to efficiently extract the model, of any online application, is an interesting problem.

• **Third-party integration** - In our implementation we demonstrate an integration of a our NRS to a specific application that we have developed. We might want to provide recommendations for a third-party application and might not have access to its internals. For instance, providing navigation recommendations for the real Yahoo! Shopping Web-site. Knowing which interfaces an application has to provide, in order to enable us assist its users, is an interesting problem.
Bibliography


   generation of recommender systems: A survey of the state-of-the-art
   and possible extensions. IEEE Transactions on Knowledge and Data


[9] T. Brazdil, A. Kucera, and O. Strazovsky. On the decidability of tem-
   poral properties of probabilistic pushdown automata. In Proc. of STACS,
   2005.


Appendix A

Third-Party Software Packages

Below is a list of software components used for the development of the query engine, the recommendation system and the sample online store. Most of these components are open-source projects. We would like to express our appreciation for the open-source community and for the developers of the following softwares in particular.

- andreas08 - the CSS template used for designing the sample Web store, created by Andreas Viklund. http://andreasviklund.com/templates/andreas08
- Altova XMLSpy - its code generator was used for creating XML parsers. http://www.altova.com/xmlspy.html.
- PHP - used for the development of the sample online store. http://www.php.net.
Throughout the development we used several “code snippets” from various sources. These are credited in the source code itself.
Appendix B

Installing the demonstration environment

The demonstration environment includes an online computers shop, a recommendation system and a link between the two.

Preliminaries

This environment requires the following third-party softwares installed to operate.

- **Apache HTTP Server** - or any other HTTP Web server. It may be downloaded from [25]. The system was developed on version 2.2.11.

- **PHP** - It may be downloaded from [27]. The system was developed on version 5.2.8.

- **MySQL** - It may be downloaded from [1]. The system was developed on version 5.1.31.

- **MySQL Connector/C** - It may be downloaded from [2]. The system was developed on version 6.0.1.

- **Graphviz** (optional) - Required only to generate images of BPs and EX-Flows. It may be downloaded from [22]. The system was developed on version 2.24

The **WampServer** [7] includes the first three components. It is recommended to install them through it.
The Shop

The shop’s files should be placed somewhere on the Apache `DocumentRoot` directory. The `DocumentRoot` may be changed at the `httpd.conf` configuration file. Let us assume that the shop’s files are placed in `DocumentRoot\shop`.

The Navigation Recommender System

The NRS consists of two executable files `TopKDaemon` and `TopKClient`. They can be placed anywhere. Let `TOPK` denote the path of the directory in which the `TopKClient` executable resides\(^1\). The `kPath` constant at `SHOP\library\topk.php` has to be set to `TOPK\TopKClient.exe`.

`TopKDaemon` requires a XML configuration file in order to run. The structure of the configuration file is given in appendix C. By default, it searches for `config.xml` in the working directory. However, any other path might be given, by starting the executable with “−c/−−config <path>”. If image generation is needed, the `DotAppPath` variable of the configuration has to be set to `GRAPHVIZ\bin\dot.exe`, assuming that `GARPHVIZ` is the path of the Graphviz software.

Shortcuts

It is recommended to create the following shortcuts.

- Start Daemon - `TOPK\TopKDaemon.exe`
- Stop Daemon - `TOPK\TopKClient.exe stop`

\(^1\)These executables may reside in different directories, as the only connection between them is through shared memory. The path of `TopKDaemon` is of no importance. However, we assume that they are placed in the same directory for simplicity of references.
Appendix C

Configuration

The top-k query engine requires an XML configuration file in order to run. By default, it searches for `config.xml` in the working directory. However, any other path might be given, by running the executable with 

\[-c/-\text{config}<\text{path}>\]

The structure of the configuration file is given below.

A configuration file has one `Root` element. This element contains one or more `Conf` elements, each identified by an `id` attribute. Only one `Conf` is used, it is chosen by `Root's activeId` attribute.

```
<Root activeId="int">
  series of Conf elements
</Root>
```

A `Conf` element has two child elements `Runs` and `DotAppPath`. The former containing a series of `Run` elements, these are executed sequentially by the query engine. The later contains the path to the `Graphviz - dot` application. This path is required only if image generation is used in any of the runs.

```
<Conf id="int">
  <Runs>
    series of Run elements
  </Runs>
  <DotAppPath>
    Path to the Graphviz - dot application
  </DotAppPath>
```

93
A *Run* element contains all the information the query engine requires to perform a query execution. Some elements has a default value. A default value is marked by bold typeface.

```
<Run>
  <Type>
    test - execute a query over a BP
    daemon - start as a daemon (to receive queries from a top-k client)
  </Type>
  <Algorithm>
    naive - use the NaiveTopK algorithm
    smart - use the TopK algorithm
  </Algorithm>
  <Aggr>
    mult, max, add
  </Aggr>
  <K>desired k</K>
  <IsDebug>
    true/false
  </IsDebug>
  <IsPrintFlows>
    true/false
    Weather or not to print the top EX-flows to the standard output
  </IsPrintFlows>
  <OutputFilesDirectory>
    Directory in which the output files will be placed
  </OutputFilesDirectory>
  <Name>
    A prefix added to every output file
  </Name>
  <BP>
    A BP element
```
The \( BP \) element contains information on how to generate the BP, the query is to be executed on. It is only relevant for the test mode.

\[
\begin{array}{l}
\text{\(<BP>\)} \\
\text{\(<Type>\)} \text{shopit - create a BP for the ShopIT demonstration} \\
\text{balanced - create a balanced BP (every node has } x \text{ implementation)} \\
\text{file - create a BP from specifications in a file} \\
\text{\(<IsPrintFlows>\)} \text{true/false} \\
\text{Wether or not to create an image of the BP} \\
\text{\(<IsPrintFlows>\)} \\
\text{\(<IsCreateXML>\)} \text{true/false} \\
\text{Wether or not to create an XML describing the BP} \\
\text{\(<IsCreateXML>\)} \\
\text{\(<ProdLimit>\)} \text{ShopIT: The number maximal number of products from every category} \\
\text{\(<ProdLimit>\)} \\
\text{\(<PcsToBy>\)} \text{ShopIT: The number of PCs needed to buy, in order to obtain a full EX-flow} \\
\text{\(<PcsToBy>\)} \\
\text{\(<Width>\)} \text{balanced: The size of the fork in every node} \\
\text{\(<Width>\)} \\
\text{\(<Depth>\)} \text{balanced: The depth of the BP} \\
\text{\(<Depth>\)}
\end{array}
\]
BP Structure files

The structure of a BP structure file is given below.

```xml
<BP>
  <DAGs>
    series of DAG elements
  </DAGs>
  <Implementations*>
    series of Implementation elements
  </Implementations>
</BP>

DAG element:

```xml
<DAG id="string, unique among all other DAG ids">
  <Nodes>
    series of Node elements
  </Nodes>
  <Edges*>
    series of Edge elements
  </Edges>
</DAG>

Node element:

```xml
<Node id="string, unique among all other node ids">
  <Label>the activity name</Label>
</Node>

Edge element:
<Edge>
  <Node1>string, id of the source node</Node1>
  <Node2>string, id of the destination node</Node2>
</Edge>

*Implementation* element:

<Implementation>
  <GuardingFormula>
    <Weight>
      <Default>numeric, default cWeight</Default>
      a series of *WeightRule* elements
    </Weight>
    <Formula>string, the environment formula</Formula>
  </GuardingFormula>
  <Node>string, id of the (source) node</Node>
  <DAG>string, id of the (destination) DAG</DAG>
</Implementation>

*WeightRule* element:

<WeightRule>
  <If>string, the condition of the rule</If>
  <Then>
    string, a formula to compute the cWeight or simple cWeight
  </Then>
</WeightRule>