Query-Feature Graphs: Bridging User Vocabulary and System Functionality

Adam Fourney
afourney@cs.uwaterloo.ca

Richard Mann
mannr@uwaterloo.ca

Michael Terry
mterry@cs.uwaterloo.ca

Cheriton School of Computer Science
University of Waterloo

ABSTRACT

This paper introduces query-feature graphs, or QF-graphs. QF-graphs encode associations between high-level descriptions of user goals (articulated as natural language search queries) and the specific features of an interactive system relevant to achieving those goals. For example, a QF-graph for the GIMP graphics manipulation software links the query "GIMP black and white" to the commands "desaturate" and "grayscale." We demonstrate how QF-graphs can be constructed using search query logs, search engine results, web page content, and localization data from interactive systems. An analysis of QF-graphs shows that the associations produced by our approach exhibit levels of accuracy that make them eminently usable in a range of real-world applications. Finally, we present three hypothetical user interface mechanisms that illustrate the potential of QF-graphs: search-driven interaction, dynamic tooltips, and app-to-app analogy search.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

General terms: Design, Human Factors

Keywords: Query-feature Graph, Search-driven Interaction, Dynamic Tooltips, Analogy Search

INTRODUCTION

When faced with a new task to accomplish, users of interactive systems must often navigate a *gulf of execution* [17]: They have a goal, they are able to succinctly express their goal, but they are unsure of how to accomplish the goal using the interactive system.

One of the hurdles in overcoming this knowledge gap is a difference in terminology. More specifically, the way users conceptualize and articulate their needs does not always match the (rather terse) vocabulary of the interactive system. In the research described in this paper, we are interested in creating a bridge between the user's vocabulary and the vocabulary of the system. Additionally, we wish to be able to automatically construct and update this bridge as a system's user base grows and as people's use of the system evolves.

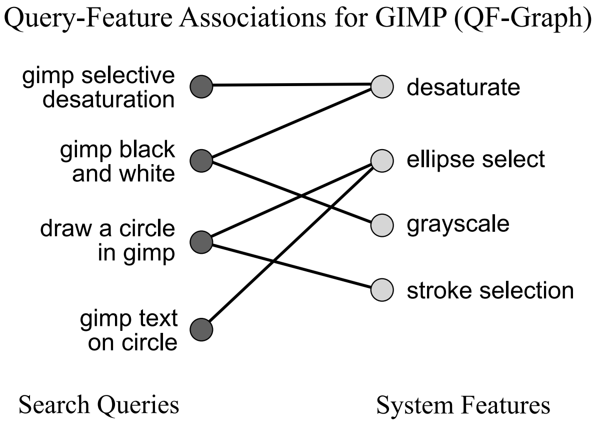


Figure 1: The query-feature graph pairs tasks, as naturally expressed in user search queries, with relevant system features.

Currently, users often bridge gulfs of execution by turning to Internet search engines. In the ideal case, a search engine returns links to web pages that describe how to achieve the desired goal. As an example, consider the search "GIMP black and white," a commonly executed query used to learn how to simulate the effect of black and white film using the GIMP raster graphics editor [8]. Since GIMP has no command named "black and white," users who wish to achieve this effect must learn that commands such as "desaturate," "grayscale," or "channel mixer" will yield the desired effect. Indeed, searching for the phrase "GIMP black and white" returns web pages describing the use of these very commands.

There are several notable aspects of this existing, manual process. First, the user articulates their goal in their own words, typically by expressing the desired outcome, rather than by using the low-level language of the interface and its commands. Second, the search engine (ideally) directs users to relevant web pages. Third, the web pages most likely to assist the user in achieving their goal combine natural language descriptions of the task with the *specific names* of the commands, tools, and preferences necessary to accomplish the

UIST'11, October 16-19, 2011, Santa Barbara, CA, USA.
Copyright 2011 ACM 978-1-4503-0716-1/11/10...$10.00.

task. In essence, the web pages serve as "Rosetta Stones" between users' conceptualizations of tasks and the actual tools necessary to accomplish those tasks.

Inspired by this manual process, this paper presents a system that automatically uncovers these relationships between users' vocabularies and the relevant system components. These pairings are represented in what we call a *query-feature* graph (Figure 1), or *QF-graph* (where "feature" in this context refers to elements in an interactive system).

The QF-graph is a weighted bipartite graph with nodes representing queries on one side, and nodes representing system terminology on the other. Edges express an association between a query and an interface component. The strength of these associations are encoded as edge weights. While a QF-graph can conceivably be assembled in a variety of ways, this paper describes how to construct QF-graphs using search query logs, search engine results, web page content, and localization data from interactive systems. As we will later argue, this mode of construction confers with it a number of advantages.

Once formed, a QF-graph can provide the foundation for a range of novel interaction techniques. In this paper, we illustrate its potential by outlining three possible interaction techniques:

- a search-driven interface in which users type the task they wish to accomplish, and the interface assembles a list of the most relevant commands for the task
- dynamic and ever-evolving tooltips that display tasks which reflect how the user community uses a given command
- app-to-app analogy search, which provides a mapping between the tools necessary to perform a task in one interface and the equivalent tools in a second interface

Collectively, the QF-graph, its mode of construction, our validation of the technique using real-world data, and the example uses of the QF-graph constitute the primary contributions of this paper.

In the remainder of this paper, we first contextualize this research with respect to prior work, then describe the query-feature graph and its automated construction in more detail. We then present results from our analysis of the technique in order to validate the quality of the query-feature associations expressed in the QF-graph. We describe three novel interaction techniques enabled by a QF-graph and conclude with a discussion of the approach's limitations and directions for future research.

RELATED WORK

The overall vision guiding this work is to create a system that can connect users' articulations of their goals with the actual system functionality necessary to achieve these goals. In this section, we describe existing mechanisms and past research that aim to help users make these connections, then discuss previous work that informed our approach of using search queries and their related web pages to construct the QF-graph.

Software commonly includes built-in help that can be searched. However, current offerings represent static collections of documents developed by the software creators. As a result, application-provided help may not always match the vocabulary of the users, nor evolve to mirror how the software is actually used in practice. Recognizing this deficiency, many software applications now augment built-in help with the capability to execute the help query on web-based knowledge bases. For example, Microsoft Word for Windows allows users to perform a help search on Microsoft's web servers if none of the local results meet their needs. While these facilities offer a useful dynamic extension to traditional static built-in help, searches are typically limited to the software producer's web properties and still require the user to manually link results with the actual functionality in the application.

This process of manually linking web-based information to the task at hand was observed by Brandt et al. in a study of software developers [5]. For example, it was noticed that developers often incorporate example code found on the web into their own code projects. These findings led the researchers to develop Blueprint, a system that more deeply integrates search-based practices into an IDE. Using Blueprint, programmers can search for, navigate, and import code examples from the web by simply typing a few keywords in the source code editor and pressing a hotkey. This mode of interaction resembles the familiar auto-completion features of modern IDEs, but greatly extends their capabilities to include knowledge culled from the web. Our work developing QF-graphs shares similar motivations to this prior research, though our focus is on explicitly linking user vocabularies with system vocabularies.

In recent years, search-like interfaces for finding and issuing commands in a user interface have grown in popularity. For example, in all modern operating systems, users are able to enter a few keywords into a search field to launch applications and load documents. Mac OS X takes this idea further, and offers a search field in the system-wide help menu that provides users with a means of searching and executing commands found in the top few levels of any application's menuing system. Similarly, programs like Quicksilver [4], Enso [11], and Ubiquity [15] allow keyword search to be used to both issue commands and specify command parameters.

This "search line" style of interaction has also been explored in the research community. Notably, Inky provides "command line functionality for the web," allowing users to send emails, book flights, and reserve conference rooms, all by entering a few keywords in a text field [14]. Inky provides some tolerance of user input by recognizing a limited set of synonyms, correcting spelling, and offering "sloppy syntax." In our own work, we are similarly motivated to allow users to express their intentions without needing to know complex syntax or specialized terminology.

The GEKA project by Jeff Hendy et al. [10], achieves search-driven interaction in the form of "graphically enhanced keyboard accelerators." These enhanced keyboard accelerators make heavy use of query auto-completion to quickly search through menus, toolbars and dialogs for commands. Users can also specify command parameters within their input.

While Inky, GEKA, and kin optimize the execution of commands, they cater to users who already know what commands they would like to issue. For users unfamiliar with a task or interface, these tools will be less useful. Our work building the QF-graph is intended to help this latter class of users by providing a bridge between high-level descriptions of tasks and the actual system features necessary to complete those tasks.

From this user-centric view of QF-graphs, we turn now to work related to the construction of a QF-graph.

The query-feature graph is similar in spirit to the query-document bipartite graph discussed by Beeferman and Berger in [2]. The graph described by Beeferman associates search queries with web documents, and is constructed by recording "click-through" patterns: As people perform web searches and visit resultant pages, their transactions serve to associate queries with documents. A similar query-document graph was described by Baeza-Yates and Tiberi in [1]. In contrast, we build a bipartite graph between queries and system functionality using question-answering techniques. Specifically, we use the question answering passage retrieval algorithm (QAP) described in [6] to form query-feature associations (again, where "features" in this context refer to the commands, tools, preferences, and other elements of an interactive system). Since click-through data is not made available to the public, our approach is arguably more accessible and actionable in generating QF-graphs.

In sum, given a high-level objective to accomplish with an interactive system, a number of mechanisms exist to assist users in translating that goal to actual system functionality. However, at present, users must manually find and establish these connections between tasks and system features. In the section that follows, we describe how we automatically establish these associations via query-feature graphs.

THE QUERY-FEATURE GRAPH

The query-feature graph directly associates user search queries with relevant system features (commands, menu items, dialogs, preferences, etc.) via an undirected weighted bipartite graph. Formally, the query-feature graph, $G = (\{Q, F\}, E)$, is composed of the following components:

- **Graph Vertices:**
  $Q = \{q_i\,; 1 \leq i \leq N\}$

  Where Q is a set of distinct search queries pertaining to the use of a given interactive system.

  $F = \{f_j\,; 1 \leq j \leq M\}$

  Where F is a set of features, commands, menu items or other interface components present in the system.

- **Graph Edges:**
  $E = \{(q_i, f_j, w_{ij})\,; q_i \in Q, f_j \in F, w_{ij} \in \mathbb{R}\}$

  Where E is a set of 3-tuples, each representing a weighted edge from a query vertex to a feature vertex. Each weight, w_{ij}, expresses the strength of the association.

System	Description of Feature Vertex Set F
Kindle	**210 commands** discovered through manual exploration of the interface.
GIMP	**830 commands** enumerated by the ingimp [21] project.
Inkscape	**1785 strings** extracted from Inkscape's primary en-US string table.
Firefox	**583 strings** extracted from Firefox's primary en-US string table.
Chrome (Chromium)	**3088 strings** extracted from Chrome's primary en-US string table.

Table 1: The five interactive systems for which we generated QF-graphs, along with a description of the data source used to popular each graph's feature vertex set.

We create a QF-graph by first enumerating the relevant search queries and system features that populate the vertex sets Q and F, respectively. We then establish associations between queries and features using techniques from question-answering research. We describe each of these steps in detail below.

Enumerating relevant search queries (Populating Q)

Assembling a meaningful set of queries, Q, is the first challenge in creating a QF-graph. While any string of characters is a valid search query, it is desirable that Q contain queries that best reflect common tasks and user vocabularies. An obvious way to populate this set, then, is to sample the search query logs of web search providers, looking for queries mentioning the interactive system. However, search query logs are not made publicly available.

To approximate search query logs, we build upon our previous research by employing the *CUTS* procedure described in [8]. CUTS leverages query auto-completion services (e.g., "Google Suggest") to sample popular queries from the logs of top-tier search engines. For publicly available software applications, this technique has been demonstrated to reveal tens or hundreds of thousands of queries for a given system.

Enumerating system features (Populating F)

To enumerate the features, F, of a system, we extract all strings contained within the system's internal string tables. String tables are used for language localization, and contain the text of all commands, error messages, and other user interface elements used by the application. As such, there are typically hundreds or thousands of strings in a string table.

Table 1 lists the five interactive systems we use as examples throughout the rest of this paper. With the exception of the Amazon Kindle, the listed applications are open source, and the string tables are distributed along with each project's source code. In order to obtain a list of features for the Amazon Kindle, we systematically crawled the Kindle interface and manually transcribed the UI components encountered.

Associating Queries with Features (Populating E)

Associating queries with features is the final step in constructing a QF-graph. In this section, we present the specific challenges inherent in this final step, then describe our use of the question-answering approach, *QAP*, to establish query-feature relationships.

Challenges

The goal of a QF-graph is to associate the text of queries with the text representing software features. However, both queries [12] and software features typically consist of only a few words, limiting the range of approaches that can be used to establish associations. More specifically, simplistic term-matching approaches such as the vector space model of information retrieval [18] cannot be readily applied: In the vector space model, the similarity of two phrases depends on the set of terms that the two phrases have in common. If the phrases share few words, then their similarity score is low. In our case, the use of term overlap is further confounded by the fact that many queries are task or goal-related and tend to have few words in common with any particular feature of the system.

In order to address the issue of term sparsity, one can simply emulate existing search practices: Each search query can be submitted to a search engine, the relevant web pages can be retrieved, and the commands, actions or tools mentioned within the web pages can be identified. This process enables us to create associations between search queries and related system functionality.

The strategy of using document retrieval to expand the set of terms associated with a short phrase is not new. Specifically, Bernstein et al. employed document retrieval to help cluster short Twitter messages [3], and Shen et al. used document retrieval to help classify search queries [19]. However, while this approach has been found to be very effective in these latter contexts, in the context of pairing search queries with specific elements of an interface, this technique breaks down when the resulting documents are multi-topical. For example, a query to learn how to perform a particular task with an application can yield web pages on user forums, frequently asked question (FAQ) pages, or blogs, all of which are inherently multi-topical documents that can reference a wide range of system features in the same document. The challenge, then, is to determine which portion of a document is most relevant to a given search query.

To address this problem of multi-topical documents, one can repeat the basic search process *within* each document, identifying and retrieving short passages that are most relevant to the original query. These passages can then be processed to identify what system features they mention.

This pipeline of retrieving relevant documents, retrieving relevant passages, then identifying a set of relevant system components, is essentially the same as that employed by many question answering (QA) services (e.g., see [20]). As such, we employed the QAP (Question Answering Passage) algorithm originally described by Clarke et al. [6] to discover associations between queries and system functionality. We briefly describe QAP in the next section. Interested readers are directed to [6] and [7] for a more thorough treatment of the QAP algorithm.

An Overview of QAP

QAP proceeds in three distinct steps: 1) retrieve short document passages relevant to the user's question or query, 2) identify potential answers in those passages, and 3) rank or otherwise validate potential answers so that a final summary can be presented to the user.

STEP 1: PASSAGE RETRIEVAL

The first step of QAP is to retrieve passages relevant to the user's query. QAP employs a cover-density ranking approach that treats all document substrings that both begin and end with a query keyword as potential passages. The details of this cover-density ranking are beyond the scope of this paper, but the ranking weighs the number of query keywords contained within each substring against the substring's length. Favourable ranks are assigned to short substrings that contain many query keywords. Once substrings are scored, longer fixed-length passages are extracted by expanding each substring about its midpoint. We elected to extract passages consisting of 300 words after early experiments suggested that this value was effective for the types of web pages and documents we were analyzing (forums, blogs, etc.). The original QAP paper, [6], utilized passages consisting of 200 words.

The original QAP papers employed cover-density ranking over all documents in the corpus. In the QA literature, it is more common to first limit the search space by creating a short list of documents that are potentially relevant to the original question or search query [20]. Passage retrieval is then employed over this smaller set of documents. To approximate this first step, we simply used Google's public search API [9] to retrieve the top 8 documents for each query. Whether using the original QAP formulation, or our modification, each document contributes only its highest scoring passage to the next step of the process.

STEP 2: ANSWER EXTRACTION

The second step of QAP is to identify potential answers mentioned within the top k scoring passages. In our case $k = 8$. In general, k can be any integer not exceeding the total number of passages (and documents, since each contributes only its top passage). In the original QAP work, each query was analyzed to determine the form of its expected answer. Depending on the question, answers might take the form of dates, proper names, cities, numbers etc. Such answers can be detected using regular expressions, or by matching a passage's phrases against lists of potential answers. In the context of our problem, document phrases are matched against the features enumerated in the QF-graph's vertex set F.

STEP 3: RANKING ANSWERS

The final step of QAP is to rank the potential answers that were identified in passages. To do this, QAP exploits the built-in redundancy of the web: The web is a large corpus, and the answer to any question is likely to be found in many documents. As such, QAP ranks answers using the following measure:

$$\text{score}(q, f) = n_{q,f} \times log\left(\frac{|D|}{d_f}\right) \quad (1)$$

Here, $n_{q,f} \leq k$ represents the number of passages returned for query q in which the feature f is mentioned. Similarly, d_f is the number of distinct documents in the corpus in which the feature's corresponding phrase f occurs, and $|D|$ is the number of documents in the corpus overall. The term $|D|/d_f$ is just the familiar inverse-document frequency (idf) of the phrase/system feature f. Since we do not have access to the full Google document corpus, we estimate the idf statistic from our smaller corpus of web pages relating to the interactive system under investigation. Depending on the interactive system, this corpus consists of tens or hundreds of thousands of web documents collected using standard web crawling practices.

Using QAP to generate edges

At the heart of the QAP question answering algorithm is the function $score(q, f)$, given by equation 1 above. This function assigns a numeric score to the tuple (q, f), expressing the strength of the association between the query q and system feature f. This score provides a means for weighting the edges in the QF-graph. Specifically, G's edges are defined as follows:

$$\begin{aligned} E \;=\{ \quad & (q_i, f_j, w_{ij});\; q_i \in Q,\; f_j \in F, \\ & w_{ij} = \text{score}(q_i, f_j) \quad \} \end{aligned} \quad (2)$$

Additionally, notice that QAP accepts any sequence of words as input. As such, the QF-graph can be actively updated as new searches are performed. This is accomplished by simply appending new queries to the query vertex set Q, and executing QAP to establish each new query's associations with the fixed set of system features F.

In theory, this method should establish reasonable connections between user queries and specific elements in the user interface. The next section presents results from our analysis examining the quality of these connections.

EVALUATION

To evaluate the quality of a QF-graph, we employed a variety of metrics. Some of these metrics are standardized and are used throughout information retrieval literature. Another was developed by us for this particular problem domain. Since results are often difficult to interpret on their own, we compare QAP results to those achieved when using a more basic term-matching approach for associating queries with features (specifically, the standard vector space model [18]).

Experiment Setup

The QF-graphs produced by our technique were evaluated using the following high-level steps. First, a set of test queries is chosen. Second, for each test query, QAP's results are recorded. Finally, the relevance of each result is judged by an expert. Since we have already covered how we construct the QF-graph, we describe the first and last steps of this process.

Selecting test queries

For this experiment, we randomly selected 20 queries from the query-feature graphs pertaining to each of the five interactive systems listed in Table 1. Because queries were selected from the query-feature graphs (which itself was built using queries harvested via CUTS), the queries represent real-world user searches.

Judging relevance

Many queries describe a goal or a task that the user would like to perform. We would like to know which system features are relevant to the query, but we would also like to know which sets of features are sufficient for completing the task implied by the search. To accommodate both needs, we manually crafted *solutions* for each test query. A solution is a collection of relevant commands or system features that "solves" or accomplishes the goal implied by the query. As an example, there are two solution sets for the query "firefox how to clear cookies":

1. Clear Recent History, Cookies, Clear Now
2. Preferences, Privacy, Show Cookies,
   Remove All Cookies

A system feature is said to be relevant to a query if the feature appears in any of the query's solutions. Similarly, we say that a set of features, S, *covers a solution* when S contains references to all system features required to implement that solution. By this definition, the full set of features F covers all solutions, and the empty set $\emptyset$ covers no solutions.

Given a selection of test queries and their associated solutions, it is possible to employ a number of metrics to measure the quality of the QAP query-feature associations. We describe these metrics next.

Performance Metrics

We employed four search-quality metrics to assess our QF-graphs: Mean precision at 1, percent correct at 10, mean average precision, and mean precision at *. Each of these metrics is described below.

- **Mean precision at 1**
  Mean precision at 1, denoted $\overline{P_{@1}}$, is a standard information retrieval metric [22] that measures the proportion of test queries whose top-ranking QAP result is judged to be relevant.

- **Percent correct at 10**
  The *percent correct at 10* measure (%C@10) is another simple metric that measures the proportion of test queries for which at least one correct solution is covered by the query's top-10 QAP results.

- **Mean average precision**
  Mean average precision (MAP) is a widely used information retrieval metric that averages the precision of a set of

search results, measured at various levels of recall (see [22] for more details). It is generally described as a measure of the area underneath the precision-recall (or, receiver operating characteristic) curve. MAP scores range from 0 to 1, with higher scores indicating better results.

- **Mean precision at ***
  In question answering literature, one standard measure of performance is *mean reciprocal rank* (MRR) [20]. A query's reciprocal rank is simply the multiplicative inverse of the rank corresponding to the first correct answer in the queries' list of results. The mean reciprocal rank metric is the average of the reciprocal ranks of all test queries.

  In our domain, many queries cannot be "answered" by a single result, but rather are "covered" by a set of results that combine to form a correct solution. As such, we developed a metric similar in spirit to MRR, described below, which we refer to as *mean precision at **.

  For each query q, let r^* be the smallest integer such that q's top r^* ranking QAP results cover one complete solution for the query. Precision at r^*, denoted $Prec_*(q)$, is the proportion of the top r^* results that are judged to be relevant to the query. $\overline{P_*}$ is the arithmetic mean of these $Prec_*(q)$ scores across all test queries.

  Importantly, the $\overline{P_*}$ measure is equivalent to MRR in cases where the query's solution is covered by a single command or system feature.

Experiment Results

Results from applying these four metrics to the experimental QF-graphs are listed in Table 2. On average, 77% percent of the test queries were "answered" (or covered) by the top-10 results returned by QAP. Moreover, the first QAP result was relevant to the query in 63% of test cases.

Regarding individual applications, results for GIMP are quite good in part because we were able to obtain a very accurate list of GIMP's commands, but also because GIMP command names are sufficiently technical to not conflict with everyday language, making it easier to identify commands mentioned in web pages. Similarly, results for Firefox are superior to those of Chrome because Firefox's localization format allowed us to easily identify strings associated with menu items, buttons and other command-related UI components. Conversely, Chrome's localization database does not provide this information. As a result, Chrome's QF graph contains a much broader set of strings (which explains why table 1 lists nearly six times as many strings from Chrome as it does for Firefox). While most of these spurious strings rarely occur in search queries or web documents, some strings (e.g., Chrome's "And then click") are sufficiently common to introduce errors.

To provide further context for interpreting these results, and to measure the impact of QAP, we repeated the experiment using a typical implementation of the vector space model to match query phrases to system features. This approach is similar to those employed in existing interface search tools (e.g., Mac OS X's help menu search), and ranks query-feature associations by averaging the importance weights of the words that both the query and system feature have in common. A word's importance weight is simply its inverse document frequency, described previously. Results from these trials are listed in Table 3.

System	$\overline{P_{@1}}$	$\overline{P_*}$	MAP	%C@10
GIMP	0.800	0.725	0.467	90%
Firefox	0.750	0.601	0.496	75%
Chrome	0.500	0.598	0.382	75%
Inkscape	0.500	0.536	0.264	70%
Kindle	0.600	0.633	0.458	75%
Overall	0.630	0.619	0.413	77%

Table 2: Performance when using QAP for discovering query-feature associations for five different interactive systems.

System	$\overline{P_{@1}}$	$\overline{P_*}$	MAP	%C@10
GIMP	0.450	0.302	0.132	30%
Firefox	0.500	0.501	0.264	70%
Chrome	0.050	0.197	0.074	45%
Inkscape	0.150	0.160	0.081	35%
Kindle	0.400	0.384	0.124	50%
Overall	0.310	0.309	0.135	46%

Table 3: Performance when using the vector space model for discovering query-feature associations.

In all cases, with all metrics, QAP's results are superior to those obtained when using simple term matching (the standard vector space model).

From these experiments we conclude that, for a given query, the top-10 query-feature associations discovered by QAP are reasonable, and, in every case, the results obtained using QAP are better than those produced by more typical implementations of interface search. Finally, these experiments establish a baseline with which future research can be compared, and improvements measured.

APPLICATIONS

QF-graphs can serve as the computational back-end for a number of novel interface mechanisms. In this section, we describe how the QF-graph can improve search-driven interaction, support dynamic tooltips, and enable application-to-application analogy search. We use example data from actual QF-graphs to illustrate the utility of QF-graphs in these hypothetical interface mechanisms.

Search-driven Interaction

The concept of search-driven interaction is simple: The user types in a few keywords and the system returns a ranked list of relevant system commands and interface components. This style of interaction can be useful when an application's features number in the hundreds or thousands [16].

QF-graphs provide a direct means of supporting search-driven interaction. When the user enters a query, the QF-graph is consulted to retrieve relevant system features. If the QF-graph does not contain an entry for the query, QAP can be used to provide this information on demand.

To illustrate the potential of this approach, the following examples demonstrate search results obtained using QF-graphs for three different interactive systems. For each example, we provide the top five results returned from querying the QF-graph. The queries themselves were drawn from the corpora of queries produced by CUTS, and thus represent frequently issued queries by users.

Query: *"gimp convert to black and white"*
As noted in the Introduction, this query is issued by users who would like to convert color digital images to images that consists only of shades of gray. This effect is most easily achieved using GIMP's, *grayscale* or *desaturate* commands, but can also be accomplished by selecting a "monochrome" option in the *channel mixer* tool, or by *decomposing* the image in order to extract its Luminosity channel (in HSL color space).

QF-graph search results:

- Channel mixer
- Grayscale
- Desaturate
- Channels
- Decompose

Query: *"how to get the kindle to read to you"*
In this case, the user would like to enable the text-to-speech feature of the Amazon Kindle. This is accomplished by pressing the *Aa* hardware button, then navigating to the *Text-to-speech* section of the dialog, and finally selecting the *turn on* command.

QF-graph search results:

- Aa
- Text to speech
- Turn on
- Web browser
- Down (directional keypad)

Query: *"change download location firefox"*
Here, a user of the Firefox web browser would like to change the location to which Firefox saves downloaded files. This can be achieved by opening Firefox's general preferences and entering a different value in a text field titled "Save files to." Alternatively, the user can check the radio button titled "Always ask me where to save files."

QF-graph search results:

- Save files to
- Always ask me where to save files
- Always ask
- Location
- Save

Importantly, each of these three queries are task-related and do not mention any system commands by name. Nevertheless, the QF-graph returns results directly relevant to the goals implied by the queries, demonstrating the utility and robustness of the approach.

Rank	Query
1	gimp draw circle
2	draw a circle in gimp
3	drawing a circle in gimp
4	gimp drawing circle
5	gimp tutorial circle
6	draw ellipse gimp
7	gimp ellipse draw
...	

Table 4: A partial list of queries neighbouring GIMP's "ellipse select" command.

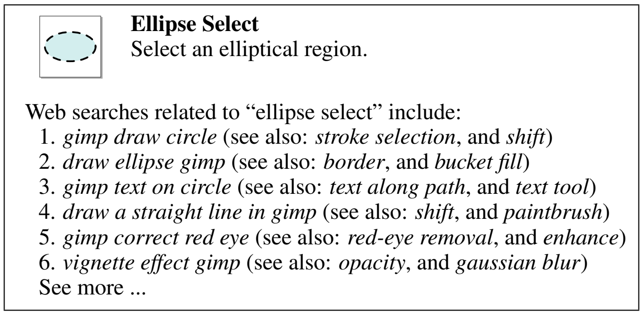
Dynamic tooltips

The features represented in a QF-graph can also be "queried" to determine the set of search queries associated with a given feature. This capability motivates *dynamic tooltips*.

Dynamic tooltips extend standard tooltips or balloon help by proactively describing the range of tasks that utilize the command or interface component currently in focus. These task descriptions are derived from QF-graphs, which are themselves derived from real-world user search queries and from web content, such as FAQs, forums, tutorials, and blog posts. As a result, these tooltips dynamically track and reflect current use of the software by the community. Figure 2 provides an example of the contents such a dynamic tooltip could display for GIMP's "ellipse select" command (where the contents are derived from GIMP's actual QF-graph).

To generate the contents of a dynamic tooltip, the system first determines which queries are associated with a given system feature f. The set of related queries Q_f is simply the set of vertices neighbouring f in the QF-graph. As an example, Table 4 provides a partial list of the queries neighbouring the "ellipse select" command in GIMP's query-feature graph.

An exhaustive examination of these queries (in this case, numbering over 750), reveals references to a variety of tasks other than drawing circles or ellipses (e.g., writing text along a circle, or correcting red eye). However, while there may exist numerous queries, many of these queries refer to the same topic, as can be seen in the partial list of queries associated with "ellipse select" shown in Table 4. In other words, there is considerable redundancy in Q_f.



Ellipse Select
Select an elliptical region.

Web searches related to "ellipse select" include:

1. *gimp draw circle* (see also: *stroke selection*, and *shift*)
2. *draw ellipse gimp* (see also: *border*, and *bucket fill*)
3. *gimp text on circle* (see also: *text along path*, and *text tool*)
4. *draw a straight line in gimp* (see also: *shift*, and *paintbrush*)
5. *gimp correct red eye* (see also: *red-eye removal*, and *enhance*)
6. *vignette effect gimp* (see also: *opacity*, and *gaussian blur*)

See more ...

Figure 2: The dynamic tooltip for GIMP's "ellipse select" command. The list of related searches is derived from GIMP's QF-graph, as are the pairs of commands associated with each search query.

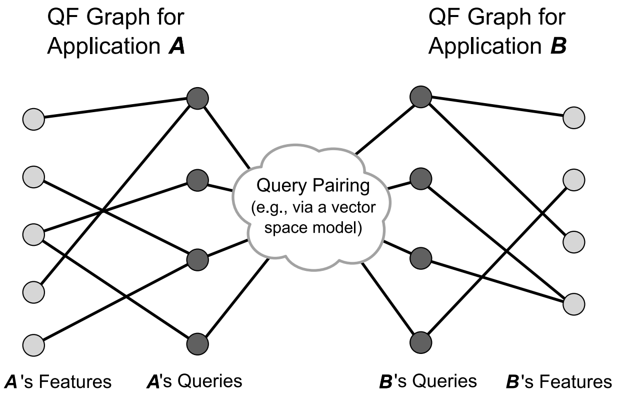

QF Graph for Application *A*
QF Graph for Application *B*
Query Pairing (e.g., via a vector space model)
A's Features
A's Queries
B's Queries
B's Features

Figure 3: App-to-app analogy search relates features found in one application to similar features found in other applications. To accomplish this, the QF-graphs of two applications are essentially joined together by using the standard vector space model to identify similar queries.

By removing redundancy in the set Q_f, one can more concisely express the variety of tasks in which the command f is involved. In order to remove redundancy in Q_f, we can repeatedly remove queries q whose query-feature edge weight is less than that of some *equivalent query*, $p \in Q_f$. We consider two queries, q and p, equivalent if they share 4 of their top-5 search results. Applying this procedure to the set of queries related to GIMP's "ellipse select" tool yields the queries listed previously in Figure 2.

The queries listed in a dynamic tooltip can also be augmented with an additional list of related commands. More specifically, the top two features associated with each query (excluding the command for which the tooltip is generated) can be displayed to provide more context about the tools necessary to complete the task represented by the query. As an example, the dynamic tooltip for the "ellipse select" tool in Figure 2 lists an example query, "gimp text on circle", along with two additional commands: "text along path" and the "text tool". Both of these latter commands can be used in conjunction with "ellipse select" to print text along the circumference of a circle. In some cases, these additional commands may be sufficient to describe the set of commands necessary to accomplish a task, thus aiding the user in learning how to use the interface. This specific use of QF-graphs is similar in spirit to the user and item-based command recommendations discussed by Matejka et al. in [13]. However, the recommendations generated from QF-graphs are likely to be more task-specific and task-focused since they derive from web pages and tutorials describing specific tasks.

App-to-App Analogy Search

Within a given domain, competing applications often provide similar functionality, but use different naming conventions or vocabularies for those features. As an example, in the domain of web browsers, Firefox's "private browsing" feature is equivalent to Chrome's "Incognito" mode (both modes limit the amount of information tracked and exchanged when browsing the web). Despite these different system names, users issue similar queries when searching for these capabilities (since queries typically express a high-level goal). These similarities in user queries make it possible to associate queries in one application to queries in a second, comparable application.

Linking queries from two different applications serves to connect the QF-graphs of the two applications (Figure 3). Once connected, the paired graphs enable *analogy* search, or the ability to directly relate the commands of one application to similar commands in the second application.

As a demonstration of this concept, the results of applying analogy search to the aforementioned private browsing example are listed below. As can be seen, analogy search is able to correctly associate Firefox's "start private browsing" command to Chrome's "new incognito window" command.

Analogy: *Chrome commands similar to Firefox's "Start Private Browsing" command:*

Results:
- New incognito window
- Incognito
- Session
- And then click

In this example, analogy search begins by identifying the top 10 queries in Firefox's QF-graph related to the "Start Private Browsing" command. For each query, a term frequency inverse document frequency (tf-idf) vector is created, as is standard in the vector space model of information retrieval [18]. A weighted sum of the 10 query vectors is then computed in order to synthesize a single feature vector for the "Start Private Browsing" command. Weights in the summation correspond to the query-feature edge weights in the QF graph. This amalgamation of queries into a communal feature vector helps to mitigate the term sparsity problem, which was previously discussed in the section entitled "Associating Queries with Features." Once the feature vector is computed, an identical process is performed for each of Chrome's commands. The similarity between pairs of commands is then computed as the dot product of the two vectors, and the highest scoring Chrome commands are listed in the search results.

As a further example of the effectiveness of app-to-app analogy search, consider two somewhat different applications: GIMP and Inkscape. GIMP is a raster graphics editor, while Inkscape is a vector graphics editor. Importantly, GIMP and Inkscape both edit images, but do so using vastly different metaphors and data (namely, pixels vs. vectors).

As an example of how these applications differ, GIMP allows users to crop an image using a "crop" tool. To achieve a similar effect in Inkscape, users must first select objects of interest, then either set the "clipping region," or "fit the page to the selection." Despite these marked differences, app-to-app analogy search is able to correctly associate GIMP's crop tool with the appropriate Inkscape commands.

Analogy: *Inkscape commands similar to GIMP's "crop" command:*

Results:
- Crop marks
- Select all in all layers
- Select
- Fit page to selection

Finally, it is also possible to use analogy search to identify related commands *within the same* application. The following example illustrates this point:

Analogy: *GIMP commands similar to GIMP's "stretch contrast" command:*

Results:
- White Balance
- Auto (Levels)
- Stretch Contrast
- Colors

In the above example, analogy search correctly inferred that GIMP's "stretch contrast", "white balance" and "auto levels" commands are related in that they are often used in conjunction (or in place of) one another, in order to enhance a digital photograph. (In this case, each of these commands is used to manipulate an image's histogram.)

DISCUSSION

We conclude our discussion of QF-graphs by considering some of their limitations, challenges in automatically creating query-feature associations, and directions for future work.

"Feature" ambiguity in web documents

In this paper, we identified system features referenced within web pages by simply searching those web pages for instances of phrases matching the names of commands, menus, and other interface components. This approach works rather well for commands with technical names (e.g., "unsharp mask"), or for longer phrases ("Always check to see if Firefox is the default browser on startup"). Such phrases are unlikely to appear accidentally in documents. However, for short commands (e.g., "Delete", "Save"), there is considerable ambiguity, and it is difficult to decide if the document is referencing a command, or if the phrase is simply part of the document's prose. In this paper, the QAP scoring function (equation 1) addresses the problem by exploiting the redundancy afforded by multiple relevant passages, all the while using inverse document frequency to reduce the impact of common phrases. In other words, the scoring function requires that a feature with a common name appear in many relevant passages in order to achieve a high score.

In future research, we would like to explore more sophisticated means of identifying references to system features. In examining tutorials and forum postings, we have noticed that people often specify the full paths of commands in their text. As an example, rather than simply writing "grayscale", many authors write "Image→Mode→Grayscale" when referring to GIMP's grayscale command. In the latter case, it is clear that the terms "Image", "Mode" and "Grayscale" are references to interface components. Identifying these types of patterns could increase the confidence that the use of a word actually refers to an element in the system. Matching the phrase to the known hierarchical organization of commands within an interface could further increase confidence measures.

Exploiting temporal associations

Additionally, just as search queries are often task-related, so too are the documents they retrieve. As such, many documents specify sequences of commands that must be executed in a particular order to achieve a desired outcome. These command sequences are not reflected in the current QF-graph, nor in the search results returned by QAP. This can be disconcerting when the order of commands returned does not match the order in which those commands should be executed. As an example, consider the query "how to draw a circle in gimp". Depending on the strengths of the query-feature associations, our system may return a ranked list of commands where "stroke selection" appears before "ellipse select" (where both commands must be used to draw a circle in GIMP since it provides no tools for drawing geometric primitives).

In the future, we would like to extract command sequences from documents, and use this sequencing information to improve the range of possible applications of QF-graphs. As an example, it would be beneficial if search-driven interaction could return sequences of actions rather than individual commands. Similarly, the availability of sequencing information could allow the recommendations made by dynamic tooltips to automatically update as the user progresses through a given task.

CONCLUSION

In this paper, we have presented QF-graphs, and demonstrated how they can be constructed automatically from logs of search queries, web pages, and localization data. While it is conceivable that QF-graphs can be constructed using alternative means or data sources, the approach outlined in this paper confers a number of advantages. Specifically, by drawing from query logs and web pages, we ensure that QF-graphs graph can be continuously updated as system usage patterns change. Moreover, a completely automated approach ensures that data from thousands of users can be considered when associating queries with system features.

This paper also outlined how QF-graphs can be used to advance search-driven interaction, while paving the way for new interaction techniques such as dynamic tooltips and application-to-application analogy search. Collectively, these mechanisms help to bridge the gulf of execution in cases where users are able to articulate their goals as search queries, but are unsure of how to accomplish those goals in an interactive system.

ACKNOWLEDGEMENTS

We would like to thank Charles Clarke for discussions which helped to clarify various details regarding the QAP passage retrieval algorithm.

REFERENCES

1. Ricardo Baeza-Yates and Alessandro Tiberi. Extracting semantic relations from query logs. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 76–85, New York, NY, USA, 2007. ACM.
2. Doug Beeferman and Adam Berger. Agglomerative clustering of a search engine query log. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 407–416, New York, NY, USA, 2000. ACM.
3. Michael S. Bernstein, Bongwon Suh, Lichan Hong, Jilin Chen, Sanjay Kairam, and Ed H. Chi. Eddi: interactive topic-based browsing of social status streams. In *Proceedings of the 23nd annual ACM symposium on User interface software and technology*, UIST '10, pages 303–312, New York, NY, USA, 2010. ACM.
4. Blacktree Software. Quicksilver: OS X at your fingertips. `http://qsapp.com/`, Retrieved April, 2011.
5. Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM.
6. Charles L. A. Clarke, Gordon V. Cormack, and Thomas R. Lynam. Exploiting redundancy in question answering. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '01, pages 358–365, New York, NY, USA, 2001. ACM.
7. Charles L. A. Clarke, Gordon V. Cormack, and Elizabeth A. Tudhope. Relevance ranking for one to three term queries. *Information Processing and Management*, 36(2):291 – 311, 2000.
8. Adam Fourney, Richard Mann, and Michael Terry. Characterizing the usability of interactive applications through query log analysis. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 1817–1826, New York, NY, USA, 2011. ACM.
9. Google Corperation. Google custom search APIs and tools. `http://code.google.com/apis/customsearch/`, Retrieved April, 2011.
10. Jeff Hendy, Kellogg S. Booth, and Joanna McGrenere. Graphically enhanced keyboard accelerators for GUIs. In *Proceedings of Graphics Interface 2010*, GI '10, pages 3–10, Toronto, Ont., Canada, 2010. Canadian Information Processing Society.
11. Humanized Inc. Enso. `http://www.humanized.com/enso/`, Retrieved April, 2011.
12. Melanie Kellar, Carolyn Watters, and Michael Shepherd. A field study characterizing web-based information-seeking tasks. *J. Am. Soc. Inf. Sci. Technol.*, 58(7):999–1018, 2007.
13. Justin Matejka, Wei Li, Tovi Grossman, and George Fitzmaurice. Communitycommands: command recommendations for software applications. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, UIST '09, pages 193–202, New York, NY, USA, 2009. ACM.
14. Robert C. Miller, Victoria H. Chou, Michael Bernstein, Greg Little, Max Van Kleek, David Karger, and mc schraefel. Inky: a sloppy command line for the web with rich visual feedback. In *Proceedings of the 21st annual ACM symposium on User interface software and technology*, UIST '08, pages 131–140, New York, NY, USA, 2008. ACM.
15. Mozilla Labs. Ubiquity: An experimental interface based on natural language input. `https://mozillalabs.com/ubiquity/`, Retrieved April, 2011.
16. Don Norman. The next UI breakthrough: command lines. *interactions*, 14:44–45, May 2007.
17. Donald A. Norman. *Cognitive Engineering*, chapter 3. User Centered System Design: New Perspectives on Human-computer Interaction. Lawrence Erlbaum Associates, 1986.
18. G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18:613–620, November 1975.
19. Dou Shen, Rong Pan, Jian-Tao Sun, Jeffrey Junfeng Pan, Kangheng Wu, Jie Yin, and Qiang Yang. Q2C@UST: our winning solution to query classification in KDDCUP 2005. *SIGKDD Explor. Newsl.*, 7:100–110, December 2005.
20. Stefanie Tellex, Boris Katz, Jimmy Lin, Aaron Fernandes, and Gregory Marton. Quantitative evaluation of passage retrieval algorithms for question answering. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, SIGIR '03, pages 41–47, New York, NY, USA, 2003. ACM.
21. Michael Terry, Matthew Kay, Brad Van Vugt, Brandon Slack, and Terry Park. ingimp: introducing instrumentation to an end-user open source application. In *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 607–616, New York, NY, USA, 2008. ACM.
22. Andrew Turpin and Falk Scholer. User performance versus precision measures for simple search tasks. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 11–18, New York, NY, USA, 2006. ACM.