Retrieving and Organizing Web Pages by "Information Unit"

Wen-Syan Li K. Selçuk Candan Quoc Vu * Divyakant Agrawal
C&C Research Laboratories, NEC USA, Inc.
110 Rio Robles, M/S SJ100, San Jose, CA 95134, USA
Email:wen,candan,qvu,agrawal@ccrl.sj.nec.com
Tel:(408)943-3008 Fax:(408)943-3099

ABSTRACT

Since WWW encourages hypertext and hypermedia document authoring (e.g., HTML or XML), Web authors tend to create *documents* that are composed of multiple pages connected with hyperlinks or frames. A Web document may be authored in multiple ways, such as (1) all information in one physical page, or (2) a main page and the related information in separate linked pages. Existing Web search engines, however, return only *physical pages*. In this paper, we introduce and describe the use of the concept of *information unit*, which can be viewed as a *logical* Web document consisting of multiple *physical* pages as one atomic retrieval unit. We present an algorithm to efficiently retrieve information units. Our algorithm can perform progressive query processing over a Web index by considering both document semantic similarity and link structures. Experimental results on synthetic graphs and real Web data show the effectiveness and usefulness of the proposed information unit retrieval technique.

Keywords

Web proximity search, link structures, query relaxation, progressive processing

1. INTRODUCTION

To find the announcements for conferences whose topics of interests include WWW, a user may issue a query "retrieve Web documents which contain keywords *WWW*, *conference*, and *topics*." We issued the above query to an Internet search engine and surprisingly found that the returned results did not include many relevant conferences. The main reason for this type of false drops is that the contents of HTML documents are often distributed among multiple physical pages and are connected through links or frames. With the current indexing and search technology, many search engines retrieve only those *physical* pages that have all the query keywords. It is crucial that the structure of Web documents (which may consist of multiple pages) is taken into consideration for information retrieval.

In this paper, we introduce the concept of an *information unit*, which can be viewed as a *logical* Web document, which may consist of multiple *physical* pages as one atomic retrieval unit. The concept of information units does *not* attempt to produce additional results that are likely to be more than users can browse. Instead, our approach is to keep those exactly matched pages at the top of the ranked list, while merging a group of partially matched pages into one unit in a more organized manner for easy visualization and access for the users. In other words, unlike the traditional keyword-based query relaxation, the information unit concept enables Web-structure based query relaxation in conjunction with the keyword-based relaxation.

Let us denote the set of Web pages containing a given keyword, K_i, as $[R_i]$. Furthermore, for a given $[R_i]$, let us define $[\rightarrow R_i]$ as the set of Web pages that contain a link to at least one of the pages in $[R_i]$. Similarly, let us denote $[R_i \rightarrow]$ as a set of pages that are linked from at least one of the pages in $[R_i]$. In the example in Figure 1, $[R_1]$, $[R_3 \rightarrow]$, and $[\rightarrow R_2]$ represent $\{URL1, URL3, URL4, URL5\}$, $\{URL1, URL2\}$, and $\{URL6, URL4, URL5, URL7\}$ respectively.

Figure 2 illustrates the query results that are generated progressively for a query with two keywords. In this figure, solid lines indicate *reuse* and dotted lines indicate *derive*. We denote *class 0* information units as documents which contain both K_1 and K_2; *class i* information units are a pair of documents such that one contains K_1 and the other K_2 and there is a path of length i between them. Note that **(1)** the intermediate query results of *Class 0*, $[R1]$ and $[R2]$, are reused (indicated by solid arrows) while processing the query for *Class 1*; **(2)** $[R1 \rightarrow]$ and $[R2 \rightarrow]$ are derived using *Class 0* results (indicated by dashed arrows); and **(3)** if necessary, computation of $([R1 \rightarrow] \cap [R2])$ and $([R1] \cap [R2 \rightarrow])$ can be parallelized.

Note, however, that this example considers only queries with two keywords, which are very common. Processing queries with three keywords is a substantial difficult task, since this involves graphs instead of paths. A study [1] has shown that most user queries on the Web typically involve two words. With query expansion, however, query lengths increase substantially. As a result, most existing search engines on the Web do not provide query expansion functionality. The information unit concept and technique provide an alternative to query relaxation not by keyword semantics, but by link structures.

This example highlights the three essential requirements of *information unit*-based retrieval on the Web:

*This work was performed when the author was with NEC USA Inc.

WWW10, May 1-5, 2001, Hong Kong.
ACM 1-58113-348-0/01/0005.

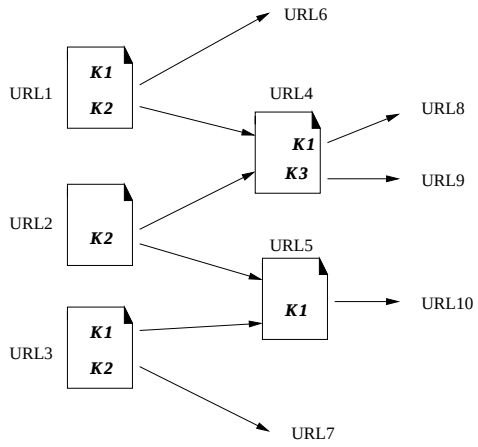

Figure 1: Examples for notations

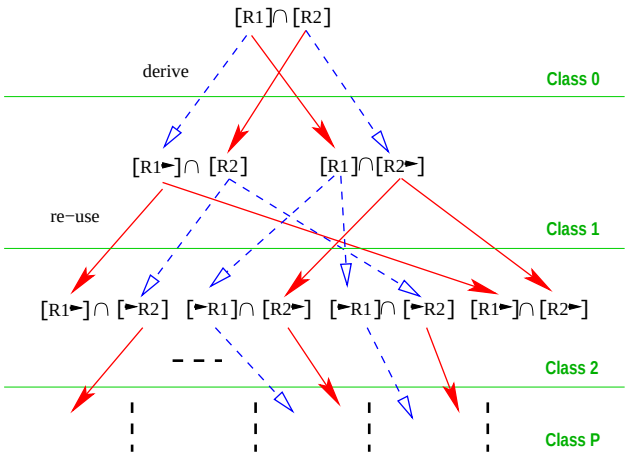

Figure 2: Query plan for finding information units for queries with two keywords

- Users are not just interested in a single result, but the top k results.
- While generating i^{th} result, we want to reuse existing $i-1$ results.
- And, since the Web is large and usually a simple search results in thousands of hits, preprocessing and any computation which requires touching or enumerating all pages is not feasible.

Note that this does not mean that we do not know of the structure of the Web. In our implementation, we used a Web index we maintain at NEC CCRL, to identify logical Web documents. The index contains information regarding the Web pages, as well as the incoming and outgoing links to them. In this paper, we present our research on progressive query processing for retrieving information units without pre-computation or the knowledge of the whole search space. We present experimental results on both synthetic graphs and real Web data.

The rest of the paper is organized as follows. In Section 2, we provide formal definitions of the more general problem of *information unit*-based retrieval. In Section 3, we describe the query processing techniques and generalize the framework to more complex scenario. In Section 4 we describe how our technique can be extended to deal with fuzziness in keyword-based retrieval, such as partial match and different importance of query keywords. In Section 5 we present experimental results for evaluating the proposed algorithm on actual Web data. In Section 6, we review related work and compare it with our work. Finally, we present concluding remarks in Section 7.

2. DATA AND QUERY MODELS FOR RETRIEVAL BY *INFORMATION UNIT*

In this section, we describe the query model for *information unit*-based retrieval and start with the definitions for the terms used in the rest of the paper:

- The Web is modeled as a directed graph $G(V, E)$, where V is the set of physical pages, E is the hyper- or semantic-links connecting these pages. We also define the undirected Web as $G^u(V, E^u)$, where E^u is the same as E except that the edges in E^u are undirected.
- The dictionary, D, is a finite set of keywords that can be used for querying the Web. A query, Q, which is a list of keywords, is a subset of D; i.e., $Q \subseteq D$.
- There is a page-to-keyword mapping $\pi : V \rightarrow 2^D$, which lists the keywords in a given page. There is also a keyword-to-page mapping $\kappa : D \rightarrow 2^V$, which lists the set of Web pages that contain the keyword.
- Finally, we also assume that there is a cost function, $\delta : E \rightarrow real$ (or $\delta : E^u \rightarrow real$), which models the distance of the pages from each other.

We denote the set of Web pages containing a given keyword, $K_i \in D$, as $[R_i]$; i.e., $[R_i] = \kappa(K_i)$. Furthermore, for a given $[R_i]$, $[\stackrel{n}{\rightarrow} R_i]$ is defined as the set of Web pages that can reach to at least one of the pages in $[R_i]$ with a path of length n. Similarly, we denote $[R_i \stackrel{n}{\rightarrow}]$ as a set of pages that are reachable from at least one of the pages in $[R_i]$ with a path of length n. Finally, we define $[R_i]^n$ as $([R_i \stackrel{n}{\rightarrow}] \cup [\stackrel{n}{\rightarrow} R_i])$.

Given the Web, $G = (V, E)$, its undirected version, $G^u = (V, E^u)$, and a query, $Q = \{K_1, \ldots, K_n\}$, an answer to Q is a set of pages, $V_Q = \{V_1, \ldots, V_m\}$, that covers all the keywords in the query; i.e.,

$$\pi(V_1) \cup \ldots \cup \pi(V_m) \supseteq \{K_1, \ldots, K_n\}.$$

A minimal answer to Q, then, is a set of pages, V_Q^m such that no proper subset, $V' \subset V_Q^m$, of V_Q^m is an answer to Q. The cost of a minimal answer, $\mu(V_Q^m)$, to Q is then the sum of the edge costs of the tree with minimal cost in G^u that contains all the vertices in V_Q^m.

Let us assume that the user issues a query, Q, with four keywords, i.e., $Q = \{K_1, K_2, K_3, K_4\}$. Let us also assume that there are three possible answers, A, B, and C, to this query as shown in Figure 3. In this figure, the edge weights may describe the importance of their association. Such association could depend on the type of connections. For example, the association or weights can be calculated based on the number of actual links between the two pages.

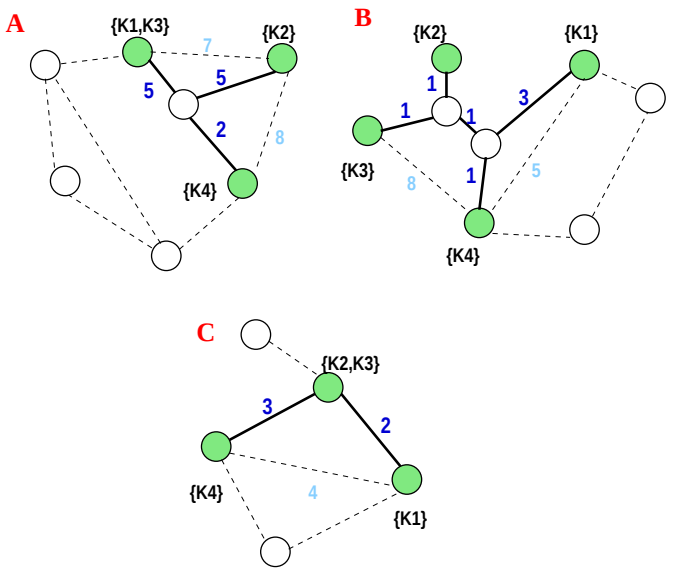

Figure 3: Three answers (A, B, and C) to query $Q = \{K_1, K_2, K_3, K_4\}$.

The solid lines in this figure denote the links that are going to be used for computing the cost of the information units (these are the lines on the smallest tree) and the dashed lines denote the links that are being ignored since their cost is not the minimum. For instance, there are at least two ways to connect all three vertices in cluster A. One of these ways is shown with dark lines, and the sum of the corresponding edge weights is 12. Another possible way to connect all three vertices would be to use the dashed edges with weights 7 and 8. Note that if we were to use the second option, the total edge weights would be 15; i.e., larger than 12 that we can achieve using the first option. Consequently, the cost of the best minimal answer is 12, not 15. In this example, $\mu(A) = 5 + 5 + 2 = 12$, $\mu(B) = 1 + 1 + 1 + 1 + 3 = 7$, and $\mu(C) = 2 + 3 = 5$.

The above query formulation, when instantiated with two keywords, can be easily solved by using the all-pairs shortest path solution (however, this algorithm would require a complete knowledge of the graph). In the next section, we examine algorithms for query processing that find the results by discovering the graph (or the Web) incrementally.

3. QUERY PROCESSING

Conventionally, answers to a Web query is an ordered list of pages, where the order reflects the rank of a page with respect to the given query. Consequently, we expect that the answers to an information unit query be an ordered set of *logical* Web documents; i.e., an ordered list of sets of Web pages. The ranks of the information units are computed by aggregating the cost of the edges involved in the graph representing the query result connected via link-in and link-out pages as described in the previous section. There are two ways to generate such an ordered list:

1. generate all possible results and, then, sort them based on their individual ranks, or

2. generate the results in the decreasing order of their individual ranks.

Clearly, the second option is more desirable since it generates the higher ranked results earlier thereby reducing the delay in responding to the user query. Furthermore, users on the Web in general specify queries in the form: "Give me the top k answers." Hence, the second approach of progressive query processing is more desirable for Web based applications. Another advantage of progressive query processing is that in some applications such as the Web based information systems, it is perhaps impossible to generate all possible answers.

In this section, we develop a progressive query processing algorithm for retrieving information units on the web. Since the search space is very large, the progressive algorithm relies on local information. The implication of relying on local information to produce answers is that the ranking of query results is approximate.

3.1 Abstract Formulation of the Problem

The problem of finding the minimum weighted connected subgraph, G', of a given graph G, such that G' includes all vertices in a given subset R of G is known as the *Steiner tree* problem[1] [2]. An extension of this problem, where we are given a set $\{[R_1], \ldots, [R_n]\}$ of sets of vertices such that the subgraph has to contain at least one vertex from each group $[R_i]$ is known as the *group* Steiner tree problem. The problem of finding the best answer with the minimal cost information unit to a query, $Q = \{K_1, \ldots, K_n\}$, can be translated into the problem of finding minimum-weighted group Steiner tree problem as follows: Let us be given an undirected Web, $G^u(V, E^u)$, and a query Q. Let also $[R_i]$, $1 \leq i \leq n$, be the set of vertices, $v_j \in V$, such that $\pi(v_j) \cap \{K_i\} \neq \emptyset$. Let us assume that the corresponding minimum weight group Steiner tree consists of a set of vertices, $V^s \subseteq V$ and a set of edges $E^s \subseteq E^u$. Then, the best answer with the minimal cost, $V_Q^{m,b}$, with respect to Q, is the maximal subset of V^s such that, for all $v_j \in V_Q^{m,b}$, $\pi(v_j) \cap Q \neq \emptyset$. Both minimum weight Steiner tree [2] and minimum weight group Steiner tree problems [3] are known to be NP-hard.

As a result of this NP-completeness result, the minimum weight group Steiner tree problem (and consequently, the minimum cost information unit problem) is not likely to have a polynomial time solution, except in certain special cases, such as when vertex degrees are bounded by 2 [4] or the number of groups is less than or equal to 2, i.e, $\{[R_1]\}$ or $\{[R_1], [R_2]\}$. However, there are a multitude of polynomial time approximation algorithms that can produce solutions with bounded errors. The most recent of these solutions, to our knowledge, is presented by Garg et al. [5]. This particular algorithm provides a randomized $O(log^3 V log n)$-approximation, where V is the number of vertices in the graph and n is the number of groups. An earlier result, by Bateman et al. [6] had a $(1 + \frac{ln\ n}{2})\sqrt{n}$ performance guarantee, which is independent of V yet non-logarithmic in n. However, since in the domain of Web querying, n is guaranteed to be much smaller than V, this earlier result is more applicable.

Note, however, that none of the above algorithms satisfy our essential requirements:

- We are not only interested in the minimum-weight group Steiner tree. Instead, what we need is the k^{th}

[1] If it exists, G' is guaranteed to be a tree.

minimum-weight group Steiner tree, where $k \geq 1$ is the rank of the corresponding answer in the result.

- We would prefer to generate k^{th} minimum-weight group Steiner tree, after we generate $(k-1)^{th}$ minimum-weight group Steiner tree, re-using results of the some of the earlier computation.

- Since the Web is large, we can not perform any pre-processing or any computation which would require us to touch or enumerate all pages on the Web.

The solution proposed by Garg et al., for instance, does not satisfy any of these requirements: it can only provide the best solution ($k = 1$), hence it is not a progressive algorithm. Moreover, since it uses linear programming, it requires enumeration of all vertices and the edges in the graph. Besides such approximation algorithms which provide performance guarantees, there are various heuristics proposed for the group Steiner tree problem. Some of these heuristics also provide performance guarantees, but these guarantees are not as tight. Such heuristics include minimum spanning tree heuristic [3], shortest path heuristic [3], and shortest path with origin heuristic [4]. None of these heuristics, however, satisfy our k^{th} minimum-weight result and progressive processing requirements.

3.2 Algorithm for Information Unit Retrieval

In this section, we develop a heuristic query processing algorithm, to retrieve information units, that adheres to the stated requirements. Unlike the other minimum spanning tree based algorithms, we do not generate the minimum spanning tree for the entire graph (or the entire Web). Furthermore, unlike other shortest-path based algorithms, we refrain ourselves from generating all possible shortest paths. Note that this does not mean that we do not know of the structure of the Web. In fact, in our implementation, we used the proposed algorithm in conjunction with a Web search index we maintain at NEC CCRL. The search index contains information regarding the Web pages, as well as the incoming and outgoing links to them.

The general idea of the algorithm is follows. Given a query $Q = \{K_1, K_2, \ldots, K_m\}$, we identify the corresponding set of pages $\mathcal{R} = \{[R_1], [R_2], \ldots, [R_m]\}$ such that K_i appears in each page in $[R_i]$. The algorithm starts with a graph represented by the vertices in $\mathcal{R}$ and then by exploring links (incoming or outgoing) with the minimum cost. During this exploration process, if we find a subgraph that satisfies the query, we output that as a query result. The rank of the query result is estimated by constructing a minimum cost spanning tree over the subgraph such that all the solution vertices are connected. Figure 4 depicts the essential parts of this algorithm.

As shown in the figure, the algorithm, *RetrieveInformationUnit*, assumes as input the graph, $G^u = (V, E^u)$, the set of initial pages $\mathcal{R} = \{[R_1], [R_2], \ldots, [R_m]\}$ corresponding to the query such that K_i is contained in the pages in $[R_i]$, and a parameter k which indicates the upper limit on the number of results that must be generated for query Q.

Lines 5-7 of Figure 4 are the initialization steps for the control variables. The main loop is depicted in lines 8-35 of the algorithm. The algorithm effectively grows a forest of MSTs. During each iteration of the loop, we choose a function, *chooseGrowthTarget* (Figure 5), to choose among different ways to grow the forest. Note that depending on the goal, we can use different choice strategies. In this paper, we describe two strategies: *minimum edge-based strategy* and *balanced MST strategy*. These two strategies will result in different degree of error and complexities. In Sections *3.3.1* and *3.3.2*, we evaluate the two strategies and discuss their quality and cost trade-offs.

Note that, in this algorithm, we assume that the costs of all neighboring vertices to be given, but in the real implementation this cost can be computed on-the-fly. The cost may be based on a variety of factors. For example, the neighboring vertex is in the same domain versus outside the domain or relevancy of links based on anchor text and/or URL strings. Much research on this issue has been done in the scope of efficient crawling [7].

After choosing an MST and an edge for growth, the algorithm checks if (line 10) the inclusion of this edge causes two MSTs to merge. If this is indeed the case, the algorithm next checks if the resulting subgraph can satisfy the query (lines 17-22). Essentially, this check determines if the new MST has a set of vertices (pages) such that the pages collectively include each keyword in the query Q. This step ensures that we only consider newly produced query results. The new query results are ranked by using the minimum spanning tree of the connected subgraph. Finally, the newly obtained (using subroutines *EnumerateFirst* and *EnumerateNext*) query results are output (lines 25-31); the algorithm terminates if k results are produced, otherwise the algorithm continues by adding the next minimum cost edge incident on the current set of vertices. In lines 33 and 34 the algorithm inserts the new MST into the growth candidates and prepares for a new iteration of the search.

Note that the minimum spanning tree computation (lines 11 and 14) of this algorithm is incremental. Since the algorithm visits edges in the increasing order of the costs, the tree is constructed incrementally by adding *neighboring edges* while growing a *forest* (multiple trees); consequently, it overcomes the weaknesses of both Kruskal's and Prim's algorithm when applied to the group Steiner tree generation on the Web. In particular, the algorithm proposed here does not require complete knowledge of the Web and it does not get stuck at one non-promising seed by growing a single tree. Next, we use an example to further illustrate the details of the algorithm. In this example, we will assume that the *chooseGrowthTarget* subroutine simply chooses the least cost edge for growth.

Figure 6(a) shows an undirected graph (the undirected representation of the Web and the keyword to page mapping, κ, for a query, Q, with four keywords, $\{K_1, K_2, K_3, K_4\}$. Note that for illustration purposes we show all the vertices (pages) in the example but they can be identified dynamically as the algorithm explores neighboring pages of the initial set of vertices represented by $[R_1], [R_2], [R_3]$, and $[R_4]$ (these vertices are shown by gray-filled circles). Let us assume that a user is interested in the best 2 "information units" that match Q. Below, we provide a step-by-step description of the query execution process:

1. Figure 6(b) shows the first step in the algorithm. An edge with the smallest cost has been identified and the endpoints of the edge is inserted into $touchedV$. Note that in the figure, all vertices in $touchedV$ are surrounded by a square and all edges included are shown as darker lines.

```
01MODULE RetrieveInformationUnit(G^u, [R_1], ..., [R_m], k);
02BEGIN     /* Compute the first k approximately optimal results for the given query Q = {K_1, ..., K_m} */
03  Solution = φ;
04  seed = [R_1] ∪ ··· ∪ [R_m];
05  growingMSTs = {(V, E)|V = {v} and v ∈ seed and E = ∅};
06  ∀M_i ∈ growingMSTs  cost(M_i) = 0;
07  growthCandidates = {⟨M_i, e_j⟩|M_i ∈ growingMSTs and e_j is incident on only one vertex on M_i};
08  WHILE (|Solution| < k) DO
09      growthTarget = ⟨M, e⟩ = chooseGrowthTarget(growthCandidates);
            e is of the form ⟨v_a, v_b⟩ where v_a ∈ M and v_b ∉ M*
10      IF ∀M_j ∈ growingMSTs  v_b ∉ M_j THEN
11          Insert v_b and e into M*; cost(M*) = cost(M) + cost(e);
12          Remove M from growingMSTs;
13      ELSE /*v_b ∈ M_j*/
14          Merge M and M_j into M* using edge e; cost(M*) = cost(M) + cost(M_j) + cost(e);
15          Remove M and M_j from growingMSTs;
17          i = 1; MoreResult = TRUE;
18          WHILE MoreResults ∧ i ≤ m DO          /* Restrict the solution spaces to current subcomponents */
19              [R_i]^a = [R_i] ∩ vertices in M; [R_i]^b = [R_i] ∩ vertices in M_j;
20              IF ([R_i]^a ∪ [R_i]^b) = φ THEN MoreResults = FALSE; END; /* IF */
21              i = i + 1;
22          END; /* WHILE */
23          IF MoreResults THEN ResultV = EnumerateFirst([R_1]^a, ..., [R_m]^a, [R_1]^b, ..., [R_m]^b))
24          ELSE ResultV = ⊥; END; /* IF */
25          WHILE (ResultV ≠ ⊥) DO
26              ResultGraph = M^+ = TrimIncrementally(M*, ResultV);
27              Cost(ResultGraph) = Sum of edges in the ResultGraph;
28              Insert ResultGraph in Solution in the increasing order of Cost(ResultGraph);
29              IF |Solution| ≥ k THEN EXIT; END; /* IF */
30              ResultV = EnumerateNext();
31          END; /* WHILE */
32      END; /* IF */
33      Insert M* in growingMSTs;
34      Update growthCandidates;
35  END; /* WHILE */
36END RetrieveInformationUnit.
```

Figure 4: A generalized algorithm for progressive querying of information unit

2. Figure 6(c) shows the step in which the algorithm identifies the first information unit. At this stage, the algorithm first identifies a new possible solution, and it verifies that the solution is within a single connected subgraph as a whole. Next, the algorithm identifies the corresponding Steiner tree through a sequence of minimum spanning tree computations and clean-ups (these steps are not shown in the figure). The resulting Steiner tree is shown with thicker lines. Note that the dark nodes denote the physical pages which form the information unit. Hence, at the end of this step, the algorithm outputs the solution, which has the total cost 8, as the first result.

3. Finally, Figure 6(d) shows the algorithm finding the second and last Steiner tree. In this step, the newly added edge reduces the number of individual connected components in the graph to one. Consequently, the algorithm identifies a new solution. The algorithm again identifies the corresponding Steiner tree through a sequence of minimum spanning tree computations and clean-ups; these steps are shown in the Figure 7:

   (a) Figure 7(a) shows the state of the graph at the beginning of this step. Note that, the graph denoted with darker lines is connected and subsumes all vertices, denoted darker, in $candidate_1$.

   (b) Figure 7(b) shows the minimum spanning tree, M^*.

   (c) Figure 7(c) shows the Steiner tree, M^+, obtained by removing the unnecessary vertices from M^*.

   At the end of this step, the algorithm outputs M^+, which has the total cost 16, as the second result. Since the user was interested in only two results, the algorithm terminates after this step.

3.3 Evaluation of the Algorithm

Since the proposed algorithm is based on local analysis and incomplete information, it is not surprising that it has limitations when compared to the optimal solution. Note that the algorithm is polynomial whereas, as we discussed earlier, finding the optimal solution is NP-hard. We now point to the cases in which the solution of our algorithm differs from the optimal solution through a series of examples. In this section, in order to show the trade-off between quality and cost of the heuristic, we will use two different *chooseGrowthTarget* functions which will result in different degrees of error and complexities. These two functions are shown in Figure 8. Intuitively, the first function chooses the smallest weighted edge at each iteration, whereas the second one aims at growing the MSTs at a balanced fashion. In Section 5, we will provide experimental evaluations of the

01MODULE *chooseGrowthTarget(growthCandidates)*;
02BEGIN /* Choose among all available growth candidates using a utility metric.*/
03 /* i.e. Minimun Edge-based Strategy or Balanced MST Strategy */
04 ...
05END *chooseGrowthTarget*.

01MODULE *EnumerateFirst*($[R_1]^a, \ldots, [R_m]^a, [R_1]^b, \ldots, [R_m]^b$);
02BEGIN /* Return a new information unit discovered by the merging of the two connected subcomponents CS_a and CS_b. Also, set up the data structures that the subsequent *EnumerateNext*() calls will use to return the remaining information units. The information units enumerated are in
$([R_1]^a \times [R_2]^b \times \ldots \times [R_m]^b) \bigcup ([R_1]^a \times [R_2]^a \times \ldots \times [R_m]^b) \bigcup \ldots \bigcup ([R_1]^b \times [R_2]^b \times \ldots \times [R_m]^a)$ */
03 ...
04END *EnumerateFirst*.

01MODULE *EnumerateNext()*;
02BEGIN /* Return a new information unit discovered by the merging of the two connected subcomponents CS_a and CS_b using the data structures set up during a previous execution of *EnumerateFirst*() call.
03 ...
04END *EnumerateNext*.

01MODULE *TrimIncrementally*(T_s^*, *ResultV*);
02BEGIN /* Trim the minimum spanning tree to find a Steiner tree specified by the vertices in *ResultV*.
Mark all the vertices in *ResultV*. Then remove all the unmarked leaves one by one.
The remaining graph is the Steiner tree.*/
03 ...
04END *TrimIncrementally*.

Figure 5: The subroutines used by the algorithm

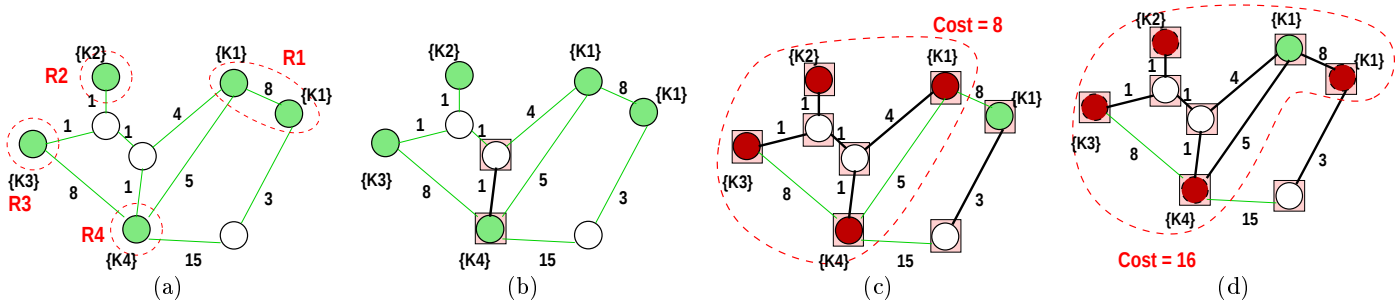

Figure 6: Group Steiner tree heuristic execution

algorithm and investigate the degree of divergence of both heuristics from the optimality.

The algorithm is *complete* in the sense that, given enough time, it can identify all information units in a given graph and *sound* in the sense that it does not generate incorrect information units. The completeness is due to the fact that, once an MST is generated, all information units on it are discovered. Since, given enough time, the algorithm will generate an MST that covers the entire graph, the algorithm is complete. Note that completeness does not imply that the information units will be discovered in the correct order.

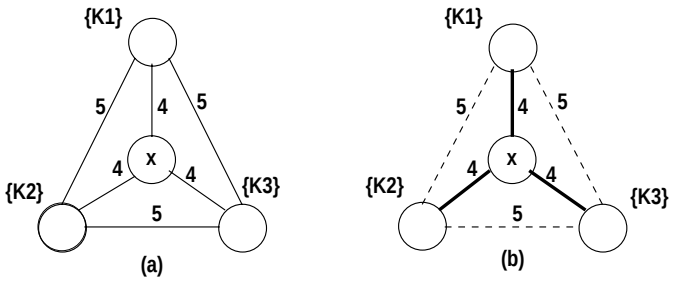

Figure 9: Sub-optimal solutions

3.3.1 Evaluation of the Minimum Edge-based Strategy

Let us consider the graph shown in Figure 9(a). As per the *chooseGrowthTarget*$_1$ subroutine, the edges that will be chosen will be $\langle K_1, x\rangle$, $\langle K_2, x\rangle$, and $\langle K_3, x\rangle$. Since the weight of all three edges is the same, the order in which the edges are included is non-deterministic and does not impact the final outcome. The state of the graph will be as shown in Figure 9(b). At this state, the algorithm will generate the one and only one information unit for the three keywords; the cost or rank associated with this information unit is the sum of the weights of the three darkened edges which is 12. However, there exists a Steiner tree of lower cost (actually there are three possible Steiner trees: $\{\langle K_1, K_2\rangle, \langle K_2, K_3\rangle\}$, $\{\langle K_2, K_3\rangle, \langle K_3, K_1\rangle\}$ and $\{\langle K_3, K_1\rangle, \langle K_1, K_2\rangle\}$ each with cost 10), which would have sufficed to form an information unit. This example illustrates that the proposed algorithm may not always generate optimal solutions.

Next consider the example illustrated in Figure 10(a) from which we want to extract three keyword information units.

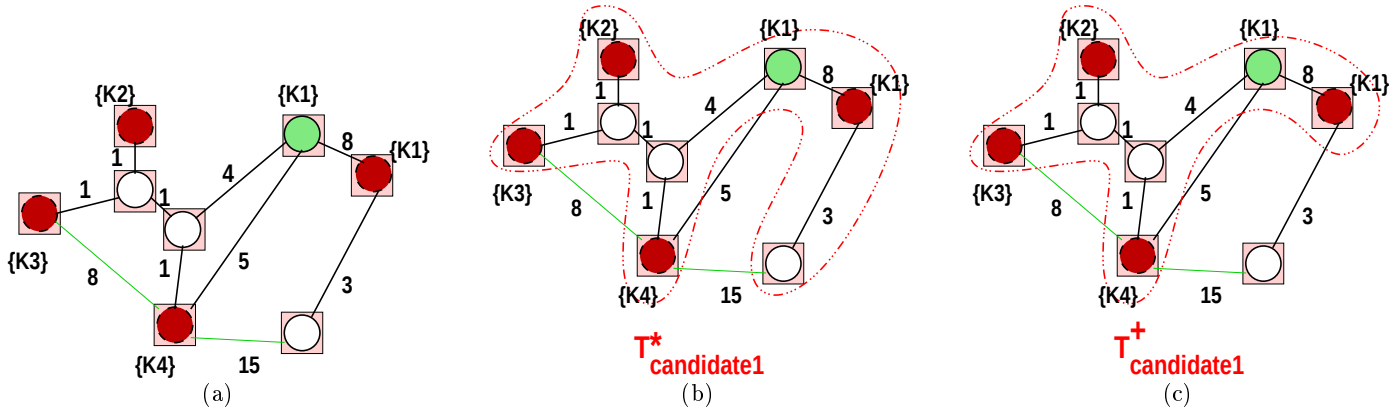

Figure 7: Use of minimum spanning trees and cleanup operation for Steiner tree heuristic

---

01MODULE *chooseGrowthTarget*$_1$ *(growthCandidates)*;
02BEGIN /* Let growthCandidates be $\{\langle M_i, e_j\rangle | M_i \in growingMSTs$ and e_j is incident on only one vertex on $M_i\}$*/
03 return($\langle M_i, e_j\rangle$) such that $cost(e_j)$ is minimum.
04END *chooseGrowthTarget*$_1$.

01MODULE *chooseGrowthTarget*$_2$ *(growthCandidates)*;
02BEGIN /* Let growthCandidates be $\{\langle M_i, e_j\rangle | M_i \in growingMSTs$ and e_j is incident on only one vertex on $M_i\}$*/
03 FORALL $\langle M_i, e_j\rangle \in growthCandidates$ DO
04 $cost(\langle M_i, e_j\rangle) = cost(M_i) + cost(e_j)$;
05 END; /* FORALL */
06 return($\langle M_i, e_j\rangle$) such that $cost(\langle M_i, e_j\rangle)$ is minimum.
07END *chooseGrowthTarget*$_2$.

Figure 8: Two subroutines for choosing among *growthCandidates*

Figure 10(b) illustrates the state of the system after the first edge (with cost 2) is added by the algorithm. After this edge, the next three edges that are included are all the edges of cost 5 as shown in Figure 10(c). At this point, the algorithm generates the information unit shown by the dotted region and this information unit has cost 15 associated with it. In the next step, the algorithm adds the edge with cost 6 connecting node v with the right hand side vertex with keyword K_3. As shown in Figure 10(d), the next information unit is output after this step is shown by the dotted region. However, the cost associated with this information unit is 13 which is smaller than the information unit that was generated earlier by the algorithm. This example illustrates that the proposed algorithm may not generate the results in the increasing order of the ranks.

Figure 11(a) shows the simplest case in which the heuristic does not return best solutions. Figure 11(b) shows the worst case scenario of quality estimation of the heuristic: when $c_n > max(c_1, \ldots, c_{(n-1)}))$, the minimum spanning tree that is grown between seed vertices v_1 and v_n (denoted with double circles) does not pass over c_n. Hence, if c_n is also smaller than $\sum_{i=1}^{(n-1)} c_i$, then the algorithm overestimates the cost of the solution. When there are two groups, as in Figure 11(b), the overestimation ratio, r, is

$$r = \frac{\sum_{i=1}^{(n-1)} c_i}{c_n} < \frac{(n-1) \times c_{max}}{c_{max}} = n - 1,$$

where n is the number of vertices in the group Steiner tree and c_{max} is the cost of the most expensive edge in the returned group Steiner tree. In general, when there are m groups (note the similarity between Figure 11(c) and 9(b)), the overestimation ratio becomes

$$r = \frac{\sum_{i=1}^{(n-1)} c_i}{\sum_{j=1}^{(m-1)} c_j^*} < \frac{(n-1) \times c_{max}}{(m-2) \times c_{min} + c_{max}} < n - 1,$$

where n is the number of vertices in the group Steiner tree, c_j^*s are the costs of the edges on an optimal group steiner tree connecting m vertices, c_{min} is the smallest cost of an edge and c_{max} is the largest cost of an edge in the returned group steiner tree.

Consequently, every solution, s, returned by the heuristic has a corresponding *range*, $[leastcost, maxcost]$, where $maxcost$ is the cost returned by the algorithm and $leastcost = \frac{maxcost}{r}$. Note that, consequently, the algorithm does not guarantee a progressive order of the results as discussed in earlier examples. In Section 5, we provide experimental evaluation of the algorithm to see how much it diverges from the optimal results. The results show that the divergence in reality is much less compared to the worst case analysis provided above (i.e., it is almost a constant factor) for information unit retrieval purposes.

Complexity. Let us assume that the maximum number of edges incident on a vertex is bounded by a constant ω (this is a reasonable assumption on the Web). Let us also assume that each of the first k results is embedded in a subgraph of diameter d. In the worst case, the number of vertices that will be touched by the edge-based growth strategy (es) is $\rho_{es} = |seed| \times 2^d$ (on the Web, the size of a subgraph

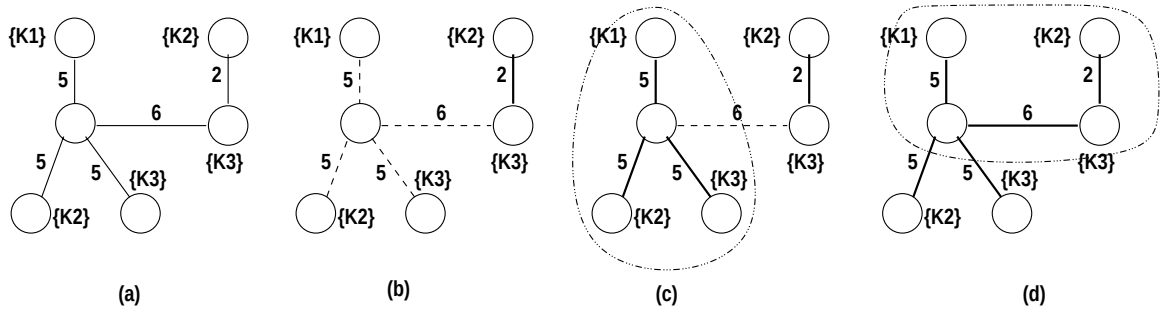

Figure 10: Out-of-order solutions

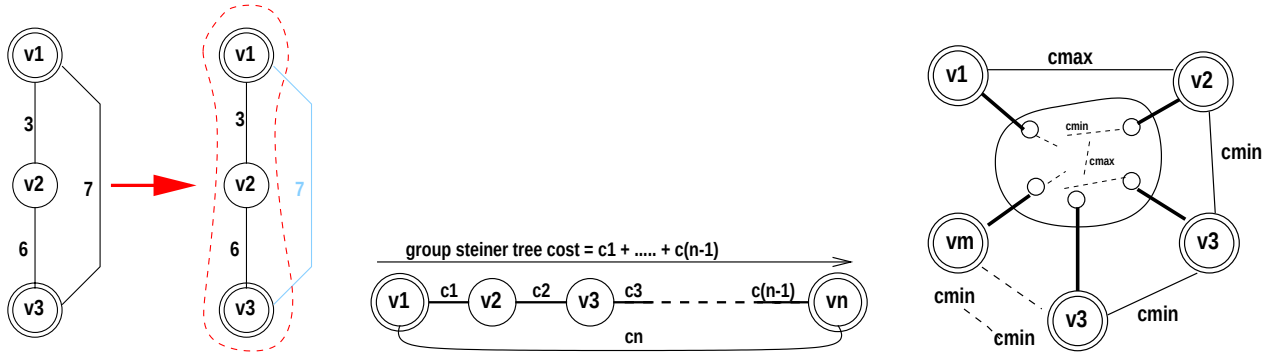

Figure 11: (a) A case where the heuristic fails. (b) and (c) worst case quality estimation scenarios: (b) two groups and $c_n > max(c_1, \ldots, c_{(n-1)})$; (c) m groups, c_{min} is the smallest cost of an edge and c_{max} is the largest cost of an edge in the returned group Steiner tree.

generally does not increase exponentially with its diameter; this is a worst case figure. Note also that for small diameters, $\rho_{es} << |V|$, where V is the set of all pages on the Web.)

The worst case execution time of the proposed algorithm, then, is $O(\rho_{es}\omega \log(\rho_{es}\omega) + (k+1)\rho_{es} + klog(k))$:

- Using a heap, maintaining the list of incident edges (maximum number of edges is $\rho_{es}\omega$) takes $O(\rho_{es}\omega \log(\rho_{es}\omega))$ time.
- Since the spanning tree is incrementally constructed, it takes $O(\rho_{es})$ time to construct the trees.
- For k solutions it takes $O(k\rho_{es})$ time to delete leaves of the spanning trees to get the approximate Steiner trees.
- It takes $O(klog(k))$ time to sort and return new results identified by the introduction of a new edge.

Note that, due to the large size of the Web, if the user wants to get all possible results (k is very large, consequently d_{es} is very large), even this polynomial worst-case time of this algorithm may not be acceptable. To deal with this problem, we can apply different constraints to reduce the execution time of the heuristic. These constraints include:

- Splitting the Web into domains, and limiting the search into only intra-domain Web documents. Since the input graph is divided into multiple, relatively small, sub-graphs, this significantly reduces the search space. As we pointed out in Section 1, the average size of a domain is around 100 documents.
- Assigning an upper bound on the total cost or on the number of vertices of an acceptable Steiner tree. For example, we may restrict the search of related pages to form an information unit within a radius of 2 links.
- Limiting the fan-out of every vertex in the graph to a predetermined small constant. This causes the graph to be sparse, thereby reducing the search time. Of course, it is a major challenge to identify which of the outgoing or incoming edges should be kept.

In Section 5, we provide experimental evaluation of the complexity of the algorithm (in terms of edges and nodes visited during the execution). Empirical evaluation is needed to determine the efficacy of the above constraints.

3.3.2 *Evaluation of the Balanced MST based Strategy*

In the previous section, we have seen that the overestimation ratio is proportional to the number of vertices in the MST. Consequently, in order to minimize the overestimation ratio, it is important to prevent the formation of long MSTs. Furthermore, in order to minimize the absolute value of the overestimation, we need to prevent very large MSTs to form. The next strategy that we describe improves on the previous one on these aspects. Later in Section 4, we show that the balanced MST based strategy is essential to the extension for dealing with fuzziness in keyword-based retrieval.

The subroutine *chooseGrowthTarget*$_2$ performs a look ahead before choosing the next MST. It identifies all possible ways to grow the forest and it chooses the growth option where the operation will result in the smallest growth.

Consequently, the minimum spanning trees are created in the increasing order of their cost, however, since MST to Steiner tree conversion is suboptimal, results may still be suboptimal.

Let us reconsider the graph shown in Figure 9(a). As per the *chooseGrowthTarget*$_2$ subroutine, the edges that will be chosen will again be $\langle K_1, x\rangle$, $\langle K_2, x\rangle$, and $\langle K_3, x\rangle$ (only these edges will result in an MST). However, we earlier have seen that this selection is suboptimal. Consequently, since it is based on MST, this second strategy does not necessarily result in an optimal solution and the overestimation ratio we found for the first strategy still holds.

This strategy prevents the formation of long chains and instead favors the creation of balanced size minimum spanning tree clusters. Note that since the overestimation ratio is proportional to the number of vertices in the MST, this results in a reduction in the amount of overestimation. This reduction is especially large when the edge weights in the graph are mostly the same, as when two edges have the same weight, the first strategy does not dictate a preference among them.

Complexity. The complexity of the second strategy is similar to the complexity of the first strategy: Let ϕ be equal to $|seed|$. As earlier, let us assume that the maximum number of edges incident on a vertex is bounded by a constant ω. Let us also assume that each of the first k results is embedded in a subgraph of diameter d. In the worst case, the number of vertices that will be touched by the balanced MST strategy (bs) is $\rho_{bs} = |seed| \times 2^d$. Then the complexity of the algorithm is $O(\phi \log(\phi) + (k+1)\rho_{bs} + klog(k))$:

- Using a heap structure, maintaining the list of the MSTs (maximum number of MSTs is ϕ) takes $O(\phi log(\phi))$ time.
- Since the spanning tree is incrementally constructed, it takes $O(\rho_{bs})$ time to construct the trees.
- For k solutions it takes $O(k\rho_{bs})$ time to delete leaves of the spanning trees to get the approximate Steiner trees.
- It takes $O(klog(k))$ time to sort and return new results identified by the introduction of a new edge.

Comparing the worst-case complexities of the balanced MST strategy, $O(\phi \log(\phi) + (k+1)\rho_{bs} + klog(k))$, and the edge-based strategy, $O(\rho_{es}\omega \log(\rho_{es}\omega) + (k+1)\rho_{es} + klog(k))$, is straightforward:

- $\phi = |seed|$ is definitely smaller than $\rho_{es} \times \omega = |seed| \times 2^d \times \omega$. Consequently, it is much cheaper to maintain the order of MSTs than to maintain the order of incident edges.
- Sorting and returning newly found results takes the same amount of time in both strategies.
- The amount of time spent for creating MSTs are the same in both algorithms: $\rho_{es} = \rho_{bs} = |seed| \times 2^d$.
- The time spent for trimming MSTs are the same in both algorithms: $k \times \rho_{es} = k \times \rho_{bs} = k \times |seed| \times 2^d$.

Consequently, in the worst case, the complexity of the balanced MST based growth strategy is less than the complexity of the edge based strategy.

Note, however, among the four components that we identified above, the time spent for creating the MSTs is the most important one, as it describes the number of webpages that may need to be visited and processed (unless the information is readily available as an index). For this component, both strategies have the same worst-case complexity $O(\rho_{es}) = O(\rho_{bs}) = O(|seed| \times 2^d)$. On the other hand, when the edge weights are distinct, edge-based strategy can cover larger distances (d) without visiting all the vertices in the same diameter, leading into a lower number of page visits. When the edge weights are mostly the same, on the contrary, the balanced MST-based strategy is likely to eliminate the formation of unnecessarily long branches of the MSTs, leading into a lower number of page visits.

4. DEALING WITH PARTIAL MATCHES AND FUZZINESS IN RETRIEVAL

In this section, we describe how to extend the algorithm to deal with fuzzy and partial keyword-based retrieval problems. More specifically, we address the following issues:

- *Disjunctive queries:* In this paper, we have formulated the Web query as a set of keywords, denoting a conjunctive query asking for all keywords. One extension to our system is to handle disjunctive queries. A disjunctive query is a query where any one of the keywords is enough to satisfy the query criterion. Combinations of conjunctions and disjunctions can also be handled through queries in conjunctive normal or disjunctive normal forms. The actual process for handling combinations of conjunctions and disjunctions is not discussed further in this paper.
- *Partial matches (or missing keywords):* In some cases, a user may issue a conjunctive query and may be willing to accept query results which do not have all the query terms. Clearly, for such a query, the user will prefer results which contains more keywords to the results which contain less keywords. One solution to such an extension is to translate a conjunction query Q to a disjunctive query Q'. This formulation, however, would not penalize results with missing keywords. Hence, the query processing algorithm needs to be modified so that the results with partial matches are ranked lower than the results with all the query terms.
- *Similarity-based keyword matching and keyword:* Because of the mismatch between authors' vocabularies and users' query terms, in some cases, users may be interested in not only results which exactly match with the query terms, but also results which contains keywords related to the query terms. For example, a logical document which contains keywords, "Web" and "symposium" may be an acceptable result to a query $Q = \{web, conference\}$ by keyword semantical relaxation. In this case, we assume that there is a similarity function, $\sigma : D \times D \rightarrow [0.0..1.0]$, where D is the dictionary, that describes the similarity of keywords. Such similarity functions have been studied elsewhere [8].
- *Keyword importance:* In the problem formulated in this paper, we assumed that each keyword in a given query, Q, is of the same importance. However, in some cases, we may want to give preference to some of the

1. $V_P = V$; $E_P^u = E^u$;
2. $\pi_P = \pi$; $\kappa_P = \kappa$; $\delta_P = \delta$;
3. For every vertex, $v_i \in V$, do
   (a) For every keyword, $K_j \in Q$, do
      i. If, $K_j \notin \pi(v_i)$, then
         A. Create a new vertex $v_{i,j}$;
         B. $\pi_P(v_{i,j}) = \{K_j\}$;
         C. $\kappa_P(K_j) = \kappa_P(K_j) \cup \{v_{i,j}\}$;
         D. Create a new edge, $e_{i,j}$, between v_i and $v_{i,j}$;
         E. $\delta_P(e_{i,j}) = \theta$;
         F. Add $v_{i,j}$ in V_P;
         G. Add $e_{i,j}$ in E_P^u;
4. Return $G_P^u(V_P, E_P^u)$.

Figure 12: Partial match transformation

keywords. For example, for a query Q with the keywords {*Y2K, solution, provider*}, the keyword *Y2K* may be more important than the two other terms. We can also assume that there is an importance function, $\gamma : D \rightarrow [0.0..1.0]$, where D is the dictionary, that describes the importance of the keywords for a given query.

We now describe the proposed solutions.

4.1 Partial Matches (Missing Keywords)

In order to consider query results which contain only a partial set of query terms, we need to adapt our technique to include group Steiner trees that do not fully cover all keywords. Instead of modifying the algorithm, we apply a graph transformation, $\mathcal{T}_P$, to the original graph representation of the Web, $G^u(V, E^u)$, so that the algorithm in Section 3.2 can be used to handle partial matches. The transformation assumes that the second growth strategy is used by the algorithm.

The transformation, $\mathcal{T}_P$, takes an undirected graph

$$G^u(V, E^u),$$

and a value of θ, as input and returns a graph $G_P^u(V_P, E_P^u)$ on which the problem of finding the minimal partial answer with the best score to a query can be translated into the problem of finding minimum-weighted group Steiner tree problem defined on the undirected graph $G_P^u(V_P, E_P^u)$. The value of θ describes the penalty for each missing keyword. The transformation, $\mathcal{T}_P$, is described in Figure 12. The input to the transformation are $G^u(V, E^u)$ and a query, $Q = \{K_1, K_2, \ldots, K_n\}$.

Description of the transformation: Let us assume that the set of nodes that contain keyword, K_j, is denoted as R_j. The transformation attaches to each node, $v_i \in V$, a set of pseudo nodes $\{v_{i,j} | v_i \notin R_j\}$. It updates the keyword/page mappings, π_P and κ_P, such that the only keyword that the new pseudo node contains is K_j. The transformation, then, updates the distance function, δ_P, such that the distance between v_i and each new vertex, $v_{i,j}$, is θ.

Note that the order in which these results will be discovered depends on the value of θ, which describes the penalty for each missing keyword and how important of exact match. In Figure 13, we show a query example where the value of θ is assigned as 4. Based on the progressive processing of the algorithm in Section 3.2, the system produces query results in Figures 13(a) and 13(b) before the result in Figure 13(c). However, if we assign value of θ to 7, the system will produce the result in Figure 13(c) first.

4.2 Fuzzy Keyword Matching

Both fuzzy keyword matching and keyword importance require preference-based processing: In the case of fuzzy keyword matching, we are given a set of keywords and we want to accept the result even if some of the keywords are not exactly matching, but are similar. Of course, we are trying to maximize the overall similarity. In the case of importance-based retrieval, on the other hand, each keyword has a different importance and we are trying to maximize the overall importance of the keywords. To handle this, we again use a graph transformation, $\mathcal{T}_S$, to handle similarity- and importance-based processing. The details of the transformation is beyond the scope of this paper.

5. EXPERIMENTAL EVALUATION

As discussed in Section 3, the proposed heuristic does not always generate information units with their optimal cost and it can also provide out-of-order solutions. However, if we consider the fact that the underlying problem of finding an information unit in the Web is NP-hard and that the graph has to be constructed incrementally, this is not unexpected. In this section, we use empirical analysis to evaluate the overall quality of the proposed algorithm.

5.1 Evaluation Criterion

One of the first experiments we conduct is on a set of synthetic graphs that are generated randomly. The parameters of these synthetic graphs are listed in Table 1. In order to have a yardstick to compare our results, we first perform an exhaustive search to find all information units along with their optimal costs. Next, we run our algorithm incrementally. We visualize the results of this experiment in three ways.

In the first measure, we compute the average cost of information units under both schemes (exhaustive and heuristic) as a function of top k results where k is varied from 10 to 100 in the increments of 10. This plot indicates the deviation or dispersion error of the heuristic. Note that since the heuristic generates sub-optimal results, and once a solution is generated it is never revised, we are bound to have some error even after exploring the entire graph.

The second plot in this experiment shows the percentage of nodes and edges used in generating top-k results. This plot allows us to measure the efficiency of progressive query processing.

The third plot shows the *recall ratio*, which captures the performance of the top-k results produced by the proposed algorithm. The recall ratio is computed as a percentage of information units in the answer set generated from the heuristic when compared to that from the exhaustive search. For example, if the query top-5 returns the 1^{st}, 2^{nd}, 3^{rd}, 5^{th}, and 7^{th} information units (as ranked by the exhaustive search), the recall ratio is 80% since the heuristic missed the 4^{th} information unit. This is similar to the recall measure used by information retrieval. Unfortunately, it penalizes the results by not giving any credits for the 7^{th} result, which can also be of some values.

Thus, we also visualize another parameter called *adjusted recall ratio*. The adjusted recall better reflects the utility of the results for information unit retrieval. The adjusted

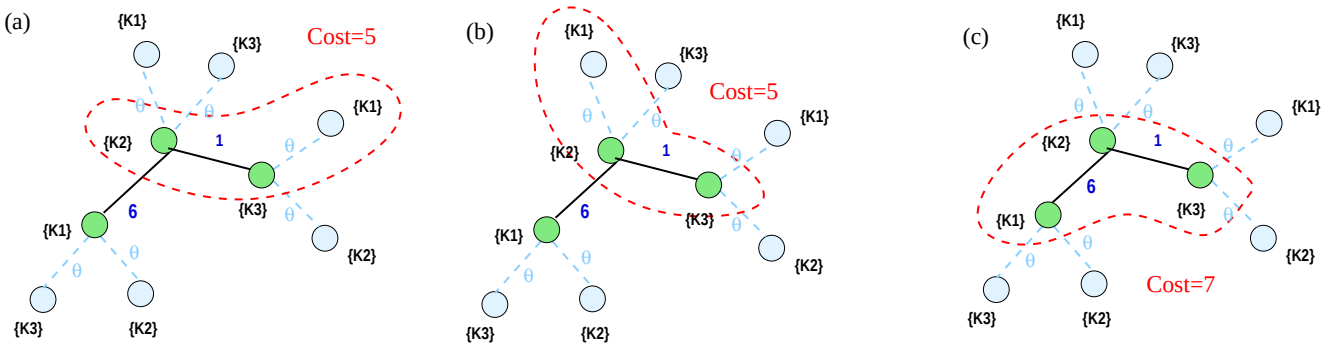

Figure 13: Examples of query results considering partial matches ($\theta = 4$)

Description	Name	Value
Number of Nodes	NumNodes	100/500/1000
Number of Edges	NumEdges	440/2461/4774
Minimum Node Degree	MinDegree	4
Maximum Node Degree	MaxDegree	12
Minimum Edge Weight	MinWeight	1
Maximum Edge Weight	MaxWeight	5
Number of Keyword		3
Occurrence of each kwd.		10

Table 1: Parameters used to generate the synthetic graphs

recall is calculated as follows: Obviously, since the query results is supposed to be a sorted list of information units, the information retrieval system should be penalized by providing the 5^{th} information unit instead of the 4^{th} information unit. Therefore, we give partial credit, $\frac{4}{5}$, for such recall instead of giving full credit of $\frac{4}{4}$. Second, we should give the 7^{th} information unit partial credit. In this example, we give a score of $\frac{5}{7}$. Note that this formulation has a nice property: If the 6^{th} information unit was returned as the last result, the system would give a score of $\frac{5}{6}$. This is appropriate since the 6^{th} information unit is of more value than the 7^{th} information unit. Similarly, if the 8^{th} information unit was returned, the system will give a score of $\frac{5}{8}$. This is also appropriate since the 8^{th} information unit is of more value than the 7^{th} information unit. The adjusted recall ratio for the top k results is calculated as follows:

$$\frac{1}{k} \times \sum_{i=1}^{k} \frac{i}{rank_i},$$

where $rank_i$ is the actual rank of the i^{th} result.

Let the user ask for the top 4 solutions and let the rank of the results returned by the algorithm be $\{2, 3, 5, 6\}$.

- *Recall ratio:* The algorithm returns 2 out of the required 4 solutions. Hence the recall ratio is 50%.
- *Adjusted recall ratio:* The adjusted recall ratio is:

$$\frac{1}{4} \times \left(\frac{1}{2} + \frac{2}{3} + \frac{3}{5} + \frac{5}{6}\right) = 65\%$$

The experiments are conducted for graphs of three different sizes: 100 nodes, 500 nodes, and 1000 nodes. The degree of each node (the total number of incoming and outgoing edges to a node) is uniformly distributed between $MinDegree$ and $MaxDegree$. Similarly, edge costs uniformly varies between $MinWeight$ and $MaxWeight$. The values we chose for each of these parameters is shown in Table 1. Note that we conducted all our experiments for three keywords since it is infeasible to conduct the exhaustive search that we use for comparison for a larger number of keywords. The number of occurrences of each keyword in the graph is set to 10. The above set up allows us to empirically evaluate the sub-optimality of the proposed heuristic when compared to the exhaustive search.

In the second experiment, instead of using synthetic data, we conducted experiments with real Web data. We downloaded the pages from `www-db.stanford.edu/people/`. The graph corresponding to the Web data consists of 236 nodes and 414 edges[2]. The weight associated with each edge was set to 1. On this data, we ran a three-keyword query involving keywords: $\{Ullman, Hector, Widom\}$. The reason for this choice was that the above three keywords had a reasonable number of occurrences in the graph: 14, 7, 7, respectively. In the next subsection, we summarize our findings with regard to these experiments.

5.2 Experimental Results on Synthetic Graphs

The experiment results show that, in a neighborhood with 100 nodes, it takes on the order of 300ms (only 10-20ms of which is the system time) to generate the top 10 retrieval results. When the user request is increased to top-25 retrieval, it takes on the order of 1700ms (only 20-30ms of which is the system time) to generate the results. Note that, since the process is progressive, top-25 generation time can be hidden while user is reviewing the top-10 results list.

Figures 14, 15, and 16 depict the result of performing three keyword queries over the 100 node, 500 node, and 1000 node synthetic graphs, respectively. Figures 14(a), 15(a), and 16(a) report the average cost of the information unit in the answer set of size from 10 to 100 in the increments of 10. As expected, due to the sub-optimal nature of the proposed heuristic, the average cost of information units is inferior to the one produced by the exhaustive search. In the case of 100 node graph the cost inflation is within two times the optimal cost and in the case of 500 nodes it is approximately 1.5 times the optimal cost. For 1000 nodes graph the cost

[2]Presentation slides and technical reports in postscript format were excluded.

inflation is also around three times the optimal cost. We expect if the variability of the weights of the edges is reduced, the inflation in the cost will become narrower. In fact, we observe this effect on the experiments that we run on real data (Section 5.3).

In Figures 14(b), 15(b), and 16(b), we report the percentage of nodes and edges visited to generate the answer sets under the proposed heuristic. Note that for the exhaustive search, the entire graph needs to be examined for each solution to ensure the minimal cost requirements (certain optimizations are, of course, possible). In contrast, we see that in the case of 100 nodes, we only explore about 45% of the nodes to produce up to 100 answers. The more dramatic observation is that only about 10% of edges are explored by the heuristic to produce top-100 answers. Note that to produce top-10 answers, the heuristic explores 30% of the nodes and 6% of the edges. This can be explained as follows. The algorithm initially needs to generate a sufficiently large connected component where the solutions can be found and this might take some time. However, after that the growth in exploration of the graph is fairly slow to produce new answers. This is a useful property since it can be combined with progressive querying to return incremental results (i.e., next-k) fairly quickly to the user. Another interesting observation is that portions of this plot are relatively flat. This is because once a connected subgraph is constructed it may have enough answers to return to the user without further exploration of the graph. The results with 500 and 1000 node graphs are similar. The range of nodes visited is from 40% to approximately 60% for 500 nodes and 30% to 40% for 100 nodes. The range for the percentage of edges visited is 8% to 15% for 500 nodes and 7% to 9% for 1000 nodes.

Finally, Figures 14(c), 15(c), and 16(c) report the recall ratio and the adjusted recall ratio of the proposed heuristic. As discussed in Section 3, the proposed heuristic generates result not necessarily in the ranked order. Furthermore, the ranking of results itself is not always as specified by the optimal order. Due to the combination of these two limitations of the proposed heuristic, we achieve less than perfect recall. As observed, the recall ratio for 100 nodes is in the range of 10% (when the size of the answer set is very small) to 50%. For 500 nodes the range is 20% to 55% and for 1000 nodes the range is 0% (for top-10 results) to 35%. If we exclude the top-10 data-point, the lower end of the recall ratio for the 1000 node graph becomes about 15%. As we have argued, the traditional recall measure imposes severe penalties for the result misses. Under this measure, for example, if in the top-10 answer set the 10^{th} element is replaced by the 11^{th} element, the loss of accuracy is 10%. Similarly, if the answer set includes 11^{th} to 20^{th} elements instead of the first 10, the loss of accuracy is 100%. The adjusted recall ratio defined above, on the other hand, is more realistic in the sense it penalizes the misses but does give partial credit for retrieving lower ranked results. In our experiments we found that the range for adjusted recall for 100 nodes is 20% to 60%; for 500 nodes it is 20% to 70%, and for 1000 nodes it is 0% (for the top-10 data-point) to 50%.

Figures 14(a), 15(a), and 16(a) report the average cost of the information unit in the answer set of size from 10 to 100 in the increments of 10. As we explained earlier, due to the sub-optimal nature of the proposed heuristic, the average cost of information units is inferior to the one produced by the exhaustive search. Another reason is that our algorithm only explores on an average less than 10% of edges. To identify how well our solutions are compared with the optimal solution *if* they are derived based on the same graph, we conducted additional experiments to find the optimal solutions for the same *subgraphs* our algorithm has explored rather than the *whole* graphs. We then compare the average cost of our *sub-optimal* solutions with the optimal solutions in Figure 17. The experimental results show that the average costs of our solutions are closer to the cost of the optimal solutions compared with the experimental results shown in Figures 14(a), 15(a), and 16(a), especially for larger graphs with 500 and 1000 nodes.

5.3 Evaluation on Real Web Data Set

Figure 18 reports the results of our experiments with actual Web data. Figure 18(a) illustrates the average costs under the two schemes. Here we see that the average cost of the information units is within 30% of that computed by the exhaustive algorithm. The reason for the small error in the cost inflation is because the edges in the Stanford Web data have a unit cost. As shown in Figure 18(b), a larger percentage of the edges are visited because of the low connectivity of the Web pages in the chosen dataset. Finally, Figure 18(c) reports the recall ratio which is in the range of 30% to 60%. The decline in recall between 50 and 100 results (x-axis) can be explained as follows: Note that from Figure 18(b) we can observe that the visited portion of the graph does not change much indicating that the graph is large enough to compute the answers in this range. Due to the greedy approach of the heuristic, when a given connected component has multiple answers, these answers are produced in a random order and not necessarily in the order of their costs. This contributes to the drop in recall. However, the adjusted recall ratio reaches almost 70% and the curve remains flat, which validates the performance of our heuristics algorithm since it provides the same level of recall utility to the users. More interestingly, our algorithm, shown in Figure 18(c), can provide 70% of adjusted recall ratio by exploring only about 30% of nodes and 25% of edges.

5.4 Discussion

In summary, our experiments on synthetic data are validated with actual data and are promising. In particular, the proposed heuristic generates information units of acceptable quality by exploring a very small part of the graph. By comparing the experimental results on the real Web data and on the synthetic graphs, we found that our heuristic algorithm performs much better on the real Web data in all categories. We examined the Web site connectivity of the Stanford site and found that the link fanout is around 4 on average. We exclude the search on presentation slides and technical reports in PDF or Postscript format. Another reason for such low fanout is that the personal home pages usually have low depth and the leaf nodes reduce the average fanout. We also found the link structures of real Web sites are more like "trees", rather than highly connected "graphs" used in experiments on synthetic data. We observe that the algorithm performs better in searching a tree-like structure with lower connectivity. We are pleased to see the heuristic algorithm performs better on the real Web data.

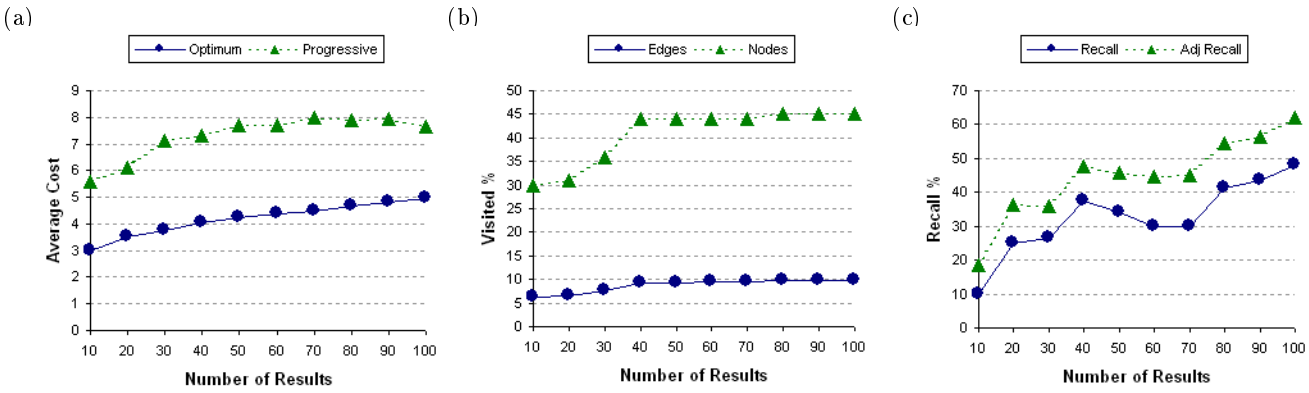

Figure 14: Experimental results on synthetic data with 100 nodes and 440 edges

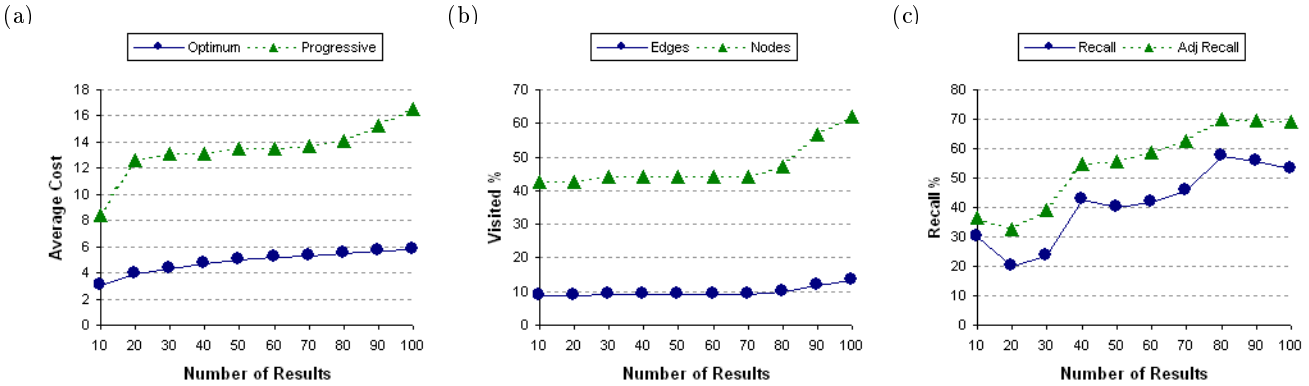

Figure 15: Experimental results on synthetic data with 500 nodes and 2461 edges

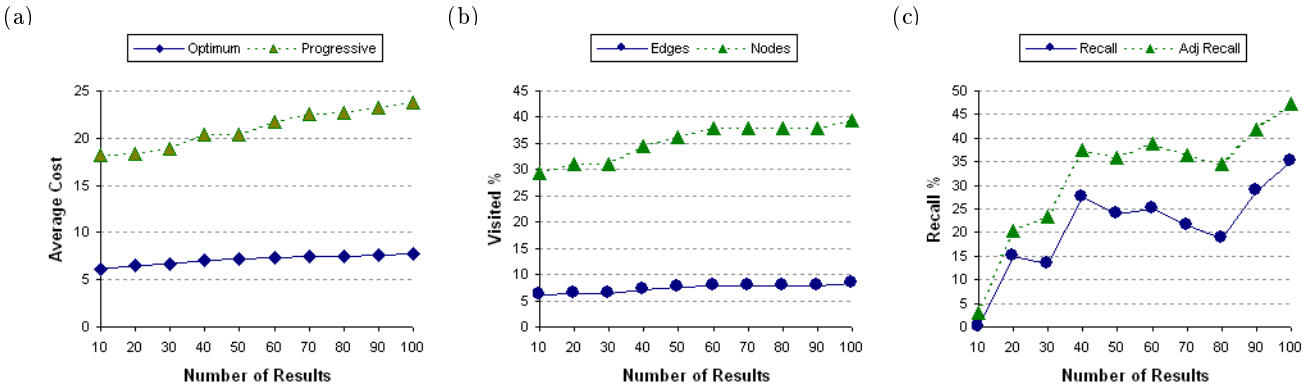

Figure 16: Experimental results on synthetic data with 1000 nodes and 4774 edges

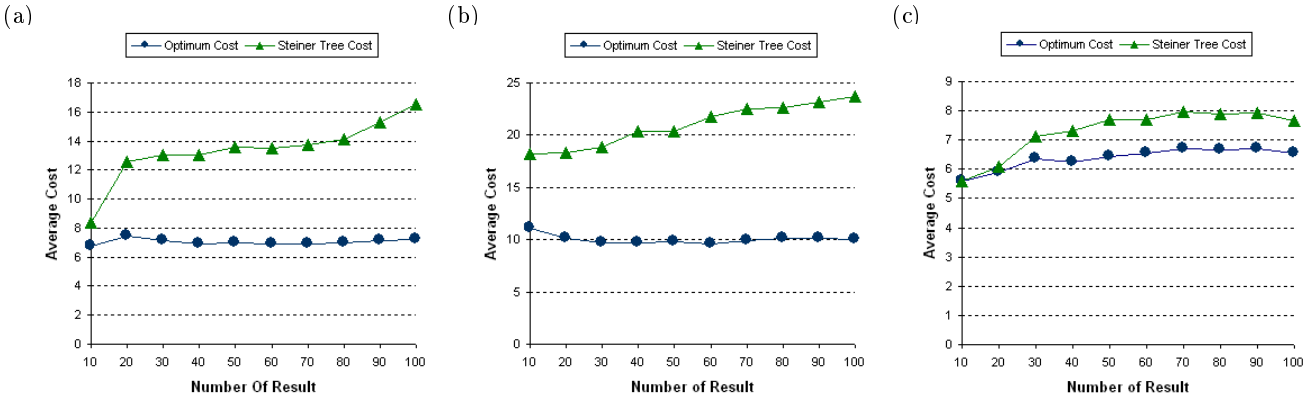

Figure 17: Comparisons between the costs of the results using the progressive information unit retrieval algorithm and the optimal solutions on synthetic data with (a) 100 nodes and 440 edges; (b) 500 nodes and 2461 edges; and (c) 1000 nodes and 4774 edges

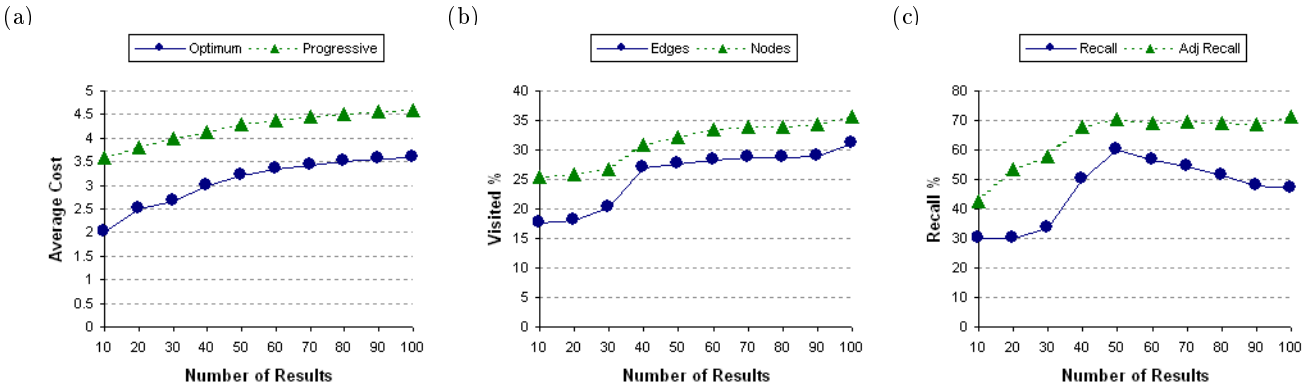

Figure 18: Experimental results on real Web data with 236 nodes and 409 edges

6. RELATED WORK

We have discussed existing work on group Steiner trees in Section 3. In this section, we give an overview of work in the area of integrating content search on the Web and Web structure analysis.

Although search engines are one of the most popular methods to retrieve information of interest from the Web, they usually return thousands of URLs that match the user specified query terms. Many prototype systems are built to perform clustering or ranking based on link structures [9, 10] or links and context [11, 12]. Tajima et al. [13] presented a technique which uses *cuts* (results of Wen structure analysis) as querying units for WWW. [14] first present the concept of information unit and [15] extends to rank query results involved multiple keywords by (1) finding minimal subgraphs of links and pages including all keywords; and (2) computing the score of each subgraph based on locality of the keywords within it.

Another solution to the above problem is the *topic distillation* [16, 17, 11, 9] approach. This approach aims at selecting small subsets of the *authoritative* and *hub* pages from the much larger set of domain pages. An authoritative page is a page with many inward links and a hub page is a page with many outward links. Authoritative pages and hub pages are mutually reinforcing: a good authoritative page is linked by good hub pages and vice versa. In [18], Bharat *et al.* present improvements on the basic topic distillation algorithm [16]. They introduce additional heuristics, such as considering only those pages which are in different domains and using page similarity for mutual authority/hub reinforcement.

Similar techniques to improve the effectiveness of search results are also investigated for database systems. In [19], Goldman *et al.* propose techniques to perform *proximity* searches over databases. In this work, proximity is defined as the shortest path between vertices (objects) in a given graph (database). In order to increase the efficiency of the algorithm, the authors also propose techniques to construct indexes that help in finding shortest distances between vertices. In our work, in the special case of two keywords we also use shortest distances. In the more general case, however, we use minimal group Steiner trees to gather results. Note that minimal group Steiner trees reduce to the shortest paths when the number of groups, that is, keywords, is limited to two. DataSpot, described in [20], aims at providing ranked results in a database which uses a schema-less semistructured graph called a Web View for data representation.

Compared with existing work, our work aims at providing more efficient graph search capability. Our work focuses on progressive query processing without the assumption of

that the complete graph is known. Our framework considers queries with more than 2 keywords, which is significantly more complex. In addition, we also present how to deal with partial matches and fuzziness in retrieval.

7. CONCLUDING REMARKS

In this paper, we introduced the concept of *information unit*, which is as a logical document consisting of multiple physical pages. We proposed a novel framework for document retrieval by information units. In addition to the results generated by existing search engines, our approach further benefits from the link structure to retrieve results consisting of multiple relevant pages associated by linkage and keyword semantics. We proposed appropriate data and query models and algorithms that efficiently solve the retrieval by information unit problem. The proposed algorithms satisfy the essential requirement of progressive query processing, which ensures that the system does not enumerate an unnecessarily large set of results when users are interested only in top matches. We presented a set of experiment results conducted on synthetic as well as real data. These experiments show that although the algorithm we propose is suboptimal (the optimal version of the problem is NP-hard), it is efficient and provides adequate accuracy.

8. REFERENCES

[1] Bruce Croft, R. Cook, and D. Wilder. Providing Government Information on the Internet: Experiences with THOMAS. In *Proceedings of Digital Libraries (DL'95)*, 1995.

[2] S.L. Hakimi. Steiner's Problem in Graphs and its Implications. *Networks*, 1:113–131, 1971.

[3] G. Reich and P. Widmayer. Approximate Minimum Spanning Trees for Vertex Classes. In *Technical Report, Inst. fur Informatik, Freiburg Univ.*, 1991.

[4] E. Ihler. Bounds on the Quality of Approximate Solutions to the Group Steiner Tree Problem. In *Proceedings of the 16^{th} Int. Workshop on Graph Theoretic Concepts in Computer Science. Lecture Notes in Computer Science*, pages 109–118, 1991.

[5] N. Garg, G. Konjevod, and R. Ravi. A Polylogarithmic Approximation Algorithm for the Group Steiner Tree Problem. In *Proceedings of the 9^{th} Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 253–259, 1998.

[6] C.D. Bateman, C.S. Helvig, G. Robins, and A. Zelikovsky. Provably Good Routing Tree Construction with Multi-port Terminals. In *Proceedings of the ACM/SIGDA International Symposium on Physical Design*, pages 96–102, Napa Valley, CA, April 1997.

[7] J. Cho, H. Garcia-Molina, and L. Page. Efficient Crawling through URL ordering. *Computer Networks and ISDN Systems. Special Issue on the Seventh International World-Wide Web Conference, Brisbane, Australia*, 30(1-7):161–172, April 1998.

[8] R. Richardson, Alan Smeaton, and John Murphy. Using Wordnet as a Knowledge base for Measuring Conceptual Similarity between Words. In *Proceedings of Artificial Intelligence and Cognitive Science Conference*, Trinity College, Dublin, 1994.

[9] David Gibson, Jon M. Kleinberg, and Prabhakar Raghavan. Inferring Web Communities from Link Topology. In *Proceedings of the 1998 ACM Hypertext Conference*, pages 225–234, Pittsburgh, PA, USA, June 1998.

[10] Lawrence Page and Sergey Brin. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Proceedings of the 7th World-Wide Web Conference*, pages 107–117, Brisbane, Queensland, Australia, April 1998.

[11] David Gibson, Jon Kleinberg, and Prabhakar Raghavan. Clustering Categorical Data: An Approach Based on Dynamic Systems. In *Proceedings of the 24th International Conference on Vary Large Databases*, pages 311–322, September 1998.

[12] Sougata Mukherjea and Yoshinori Hara. Focus+Context Views of World-Wide Web Nodes. In *Proceedings of the 1997 ACM Hypertext'97 Conference*, pages 187–196, Southampton, UK, March 1997.

[13] Keishi Tajima, Yoshiaki Mizuuchi, Masatsugu Kitagawa, and Katsumi Tanaka. Cut as a Querying Unit for WWW, Netnews, e-mail. In *Proceedings of the 1998 ACM Hypertext Conference*, pages 235–244, Pittsburgh, PA, USA, June 1998.

[14] Wen-Syan Li and Yi-Leh Wu. Query Relaxation By Structure for Document Retrieval on the Web. In *Proceedings of 1998 Advanced Database Symposium*, Shinjuku, Japan, December 1999.

[15] Keishi Tajima and Kenji Hatano and Takeshi Matsukura and Ryoichi Sano and Katsumi Tanaka. Discovery and Retrieval of Logical Information Units in Web. In *Proceedings of the 1999 ACM Digital Libraries Workshop on Organizing Web Space*, Berkeley, CA, USA, August 1999.

[16] Jon Kleinberg. Authoritative Sources in a Hyperlinked Environment. In *Proceedings of the 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998.

[17] Soumen Chakrabarti, Byron Dom, Prabhakar Raghavan, Sridhar Rajagopalan, David Gibson, and Jon Kleinberg. Automatic Resource Compilation by Analyzing Hyperlink Structure and Associated Text. In *Proceedings of the 7th World-Wide Web Conference*, pages 65–74, Brisbane, Queensland, Australia, April 1998.

[18] Krishna Bharat and Monika Henzinger. Improved Algorithms for Topic Distillation in a Hyperlinked Environment. In *Proceedings of the 21th Annual International ACM SIGIR Conference*, pages 104–111, Melbourne, Australia, August 1998.

[19] Roy Goldman, Narayanan Shivakumar, Suresh Venkatasubramanian, and Hector Garcia-Molina. Proximity Search in Databases. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 26–37, New York City, New York, August 1998. VLDB.

[20] Shaul Dar, Gadi Entin, Shai Geva, and Eran Palmon. DTL's DataSpot: Database Exploration Using Plain Language. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 645–649, New York City, New York, August 1998. VLDB.