Optimizing and Parallelizing Ranked Enumeration

Konstantin Golenberg*
The Hebrew University
Jerusalem 91094, Israel
konstg01@cs.huji.ac.il

Benny Kimelfeld
IBM Research—Almaden
San Jose, CA 95120, USA
kimelfeld@us.ibm.com

Yehoshua Sagiv*
The Hebrew University
Jerusalem 91094, Israel
sagiv@cs.huji.ac.il

ABSTRACT

Lawler-Murty's procedure is a general tool for designing algorithms for enumeration problems (i.e., problems that involve the production of a large set of answers in ranked order), which naturally arise in database management. Lawler-Murty's procedure is used in a variety of modern database applications; particularly in those related to keyword search over structured data. Essentially, this procedure enumerates by invoking a series of instances of an optimization problem (i.e., finding the best solution); solving the optimization problem is the only part that depends on the specific task at hand. The topic of optimizing and parallelizing Lawler-Murty's procedure is investigated. Naive parallelism can be carried out by concurrently solving independent instances of the optimization problem. This can be improved by printing the next answer, in the enumeration order, as soon as none of the concurrent instances can produce a better answer. However, this approach alone suffers from poor utilization of available threads. That leads to the idea of freezing an instance of the optimization problem. Interestingly, not only is freezing beneficial to the parallel execution of Lawler-Murty's, it also substantially reduces the running time of the serial execution. Additional improvements of the freezing technique are then developed to further enhance the utilization of threads, and they result in a significant overall speedup. The effectiveness of the proposed approach is demonstrated on keyword search over data graphs, wherein an extensive experimental study is described.

1. INTRODUCTION

Answer enumeration arises in a computational problem where the output is too large (e.g., of size super-polynomial in that of the input) and, hence, a user cannot be expected to wait for the whole of it to be computed. Instead, an efficient algorithm is expected to incrementally enumerate output items (*answers* or *solutions*) *throughout* the computation, rather than after the completion of the computation. This is a common case in query evaluation over databases. Not surprisingly, ranked enumeration is the focus of extensive research in the database area. A theoretical guarantee that captures the notion of incremental evaluation is that of *polynomial delay* between consecutive answers [6]. Very often, answers are *ranked* by their estimated desirability (e.g., by a scoring function that estimates relevancy or by a user-specified ORDER BY), and then we desire *ranked enumeration*, which means that answers are produced in the order of their ranks (e.g., by decreasing score or increasing weight). Furthermore, when ranking is involved, the user is likely to be satisfied (and halt the computation) immediately after getting the *top-k* answers.

An example of the need for ranked enumeration is in keyword search over structured data. The typical approach to this problem [1,4,5,7,11,16,19] considers a *data graph* with keywords attached to some of the nodes. An answer is a connected subgraph (usually a tree) that contains the keywords. A central component in the score of an answer is the size of its subgraph. *Small* subgraphs indicate closer relationships among the keywords and, hence, they are likely to be ranked higher. Unfortunately, the number of such subgraphs can be huge; theoretically, this number can be exponentially larger than the size of the data, even if only two keywords are involved [11]. Hence, in that setting, there is no alternative to enumerating the answers (or producing just the top-k), in ranked order (e.g., by increasing size or weight).

In ranked enumeration, finding the first answer is nothing else than an *optimization problem*, for which we have an abundance of tools (e.g., shortest-path algorithms, linear programming, Viterbi algorithms, and so on). For $i > 1$, finding the ith answer amounts to computing the best answer, under the restriction that it is not among the first $i-1$ answers. Handling that restriction is the core difficulty in designing enumeration and top-k algorithms. We are aware of only very few general techniques for handling this difficulty. A central technique (with applications discussed below) is Lawler-Murty's procedure [15, 17],[1] which reduces an enumeration problem to an optimization problem with very simple constraints. We explain the details of this procedure in Section 2. Roughly speaking, to apply the procedure to a specific setting, one needs just to design an efficient solution to the constrained optimization problem.

*Work supported by The German-Israeli Foundation for Scientific Research & Development (Grant 973–150.6/2007).

Proceedings of the VLDB Endowment, Vol. 4, No. 11

[1] Among the enumeration techniques discussed in a tutorial by Cohen et al. [2], Lawler-Murty's is the only one that is applicable to ranked enumeration (except for extremely restricted cases of ranking functions).

Lawler-Murty's procedure is a generalization of Yen's algorithm [23] for finding the k shortest simple paths of a graph. As we discuss next, this procedure has been used for solving a large number of enumeration and top-k problems, in both systems and theory.

Lawler-Murty's procedure is implemented in various applications, such as the bioinformatic system of Takigawa and Mamitsuka [20] that finds connections (through chemical reactions, genes, etc.) between metabolites. This procedure has also been used for solving problems related to shortest paths, such as finding k minimum spanning trees [3, 8] and finding k minimum Steiner trees [11]. Another early application of the procedure is to finding k-best solutions in resource allocation [9].

More recently, Kimelfeld and Ré [10] used Lawler-Murty's procedure in order to enumerate the output of a string transducer, when applied to a Markov sequence; there, each answer is a string that is scored with a probability. The system built by Talukdar et al. [21] automatically suggests queries for designing search forms in the context of integrating biomedical information. Their query-generation algorithm is an adaptation of Kimelfeld and Sagiv's algorithm for top-k Steiner trees [11], which in turn is an application of Lawler-Murty's procedure (as mentioned earlier). Parameswaran and Garcia-Molina [18] show how Lawler-Murty's procedure can be used within the task of generating (multiple) course recommendations in the context of *CourseRank*, which is a system used by Stanford students for planning academic programs. Kimelfeld and Sagiv used the procedure to enumerate answers of structured queries with ranking, such as SQL with ORDER-BY [12] or flexible twig queries over XML [13]. They also showed that the procedure can be extended to simultaneously support both ranking and incompleteness when queries are trees [14].

In all of the above applications, Lawler-Murty's procedure facilitates the design of ranked-enumeration algorithms (with provable guarantees of correctness and polynomial delay). But in practice, the main drawback of this procedure is a high execution cost. In particular, our initial implementation suffered from very poor performance in terms of running time. Optimizing Lawler-Murty's procedure is, of course, beneficial to each of its existing applications; furthermore, due to the generality of this procedure (which is essentially a reduction from ranked enumeration to optimization), we believe that optimizing it to a practical level would give rise to effective ranked enumeration in many new applications. Initially, we attempted to improve performance by a simple multi-core parallelization, which unfortunately resulted in a negligent utilization of the core multiplicity. We eventually realized that a proper parallelization of Lawler-Murty's procedure requires a nontrivial understanding of and intervention in the execution flow of the procedure.

The goal of the research presented in this paper is to develop practical techniques for optimizing and parallelizing Lawler-Murty's procedure. Our case study is an implementation of keyword search on data graphs, where the techniques we present are shown to be highly beneficial. Nevertheless, we focus on techniques that are relevant to essentially *every* application of Lawler-Murty's procedure. Hence, we believe that all systems implementing this procedure can significantly benefit from our techniques.

Particularly, we introduce the idea of *freezing* that substantially reduces the amount of computations done in solving constrained optimization problems. We show how to combine freezing with parallelism. This combination cuts even further the needed computations. In the parallel case, freezing has a greater effect when it is applied early. Doing so entails a significant change in the logic of Lawler-Murty's procedure. The experiments show an improvement by up to a factor of 26 compared to the original Lalwer-Murty's procedure.

2. LAWLER-MURTY'S PROCEDURE

In this section, we describe Lawler-Murty's procedure [15, 17]. The importance of this procedure lies in its generality and efficiency guarantees. Lawler-Murty's procedure can enumerate (in ranking order) all the solutions to an optimization problem, provided that we have an efficient algorithm for finding the best solution (to that problem) under a certain type of constraints. In Appendix A, we give an intuitive explanation of Lawler-Murty's procedure. Figure 1 gives the pseudocode. The input is a dataset D and an instance P of the enumeration problem at hand. Each *solution* S to P is represented as set of *objects* (or *elements*) of D, where the exact definition of an object depends on the dataset D and the enumeration problem. As an example, if we want to find all the simple paths between a given pair of nodes s and t, then D is a graph, P is the given pair of nodes, an object is an edge, and a solution is a set of edges that form a simple path between s and t. We assume that some function defines the *weight* (also called *value*) of each solution to P. The output is an enumeration of all the solutions to P by increasing weight. In practice, we often stop Lawler-Murty's procedure after getting the top-k solutions.

Lawler-Murty's procedure uses two types of constraints in order to define subsets of the solution space. I denotes a set of *inclusion constraints*, and E denotes a set of *exclusion constraints*. Each constraint is represented by an object of the input dataset D, so we view both I and E as sets of objects. A solution S *satisfies* I and E if it includes all the objects of I and none of the objects of E. By a slight abuse of terminology, we may refer to I and E simply as constraints, rather than sets thereof.

In the algorithm of Figure 1, a triplet $[I, E, S]$ comprises a set I of inclusion constraints, a set E of exclusion constraints, and the top-ranked solution S among all those satisfying I and E. In particular, $[\emptyset, \emptyset, S]$ (where $\emptyset$ is the empty set of constraints) is the triplet that contains the top-ranked solution, which should be printed first.

The procedure $\mathsf{SolveOpt}(D, P, I, E)$ has the following arguments: a dataset D, an instance P of the given problem, a set I of inclusion constraints, and a set E of exclusion constraints. It computes the top-ranked solution among all those satisfying I and E; if no solution exists, $\perp$ is returned. This procedure depends on and should be designed for the specific enumeration problem at hand.

We now describe in detail the algorithm of Figure 1. Line 1 initializes an empty priority queue. This queue stores triplets $[I, E, S]$ according to the value (i.e., weight) of S. In particular, the top of the queue has the smallest value. In line 2, SolveOpt is called with the empty sets of inclusion and exclusion constraints, and it returns the top-ranked solution S for the dataset D and the instance P. If S exists (i.e., $S \neq \perp$), then line 4 inserts the triplet $[\emptyset, \emptyset, S]$ into the priority queue.

While the queue is not empty, the loop of line 5 repeats the following operations. Line 6 pops the top of the queue

Algorithm: *Lawler-Murty*(D, P)

```
1:  Queue ← an empty priority queue
2:  S ← SolveOpt(D, P, ∅, ∅)
3:  if S ≠⊥ then
4:     Queue.insert([∅, ∅, S])
5:  while Queue ≠ ∅ do
6:     [I, E, S] ← Queue.removeTop()
7:     print(S)
8:     create the pairs of constraints [I_1, E_1], ..., [I_n, E_n] for
       the optimization problems spawned by [I, E, S]
9:     for i ← 1, ..., n do
10:       S_i ← SolveOpt(D, P, I_i, E_i)
11:       if S_i ≠⊥ then
12:          Queue.insert([I_i, E_i, S_i])
```

Figure 1: Lawler-Murty's procedure.

to get the triplet $[I, E, S]$. The solution S is printed in line 7. Line 8 depends on the given enumeration problem and, hence, is written just as a high-level operation. This line partitions (into subsets) all the solutions (except S) that satisfy I and E. Each subset is defined by a pair $[I_i, E_i]$ of (sets of) inclusion and exclusion constraints. The pairs $[I_1, E_1], \ldots, [I_n, E_n]$ are derived from the triplet $[I, E, S]$ that was popped from the queue in line 6. The way of deriving these pairs is particular to the enumeration problem at hand. The number n of pairs depends on the size of S. The loop of line 9 iterates over all these pairs, and for each $[I_i, E_i]$ it finds in line 10 the top-ranked solution S_i (for the dataset D and the instance P) that satisfies I_i and E_i. If S_i exists, then line 12 inserts the triplet $[I_i, E_i, S_i]$ into the queue.

In summary, line 8 uses $[I, E, S]$ to create n instances of the optimization problem (i.e., finding the top-ranked solution under constraints). We refer to these instances (allowing for a slight abuse of terminology) as the optimization problems *spawned* by $[I, E, S]$.

An important property of Lawler-Murty's procedure is the following. If SolveOpt(D, P, I, E) finds only an approximation of the top-ranked solution under constraints, but with a guaranteed approximation ratio, then the enumeration is in a guaranteed approximate order [11]. This is useful when it is too hard (from either a theoretical or practical point of view) to find the exact top-ranked solution.

2.1 Research Goal and Methodology

Our goal is to enhance the practical efficiency of Lawler-Murty's procedure by developing optimization and parallelization techniques that do not depend on a particular enumeration problem. However, testing these techniques can be done only in the context of specific problems. In particular, the experiments that we describe in Section 8 test the effectiveness of our techniques with respect to the problem of keyword search over data graphs, which is described next.

When applying keyword search to data graphs, the goal is to enumerate reduced subtrees that contain all the terms of a given query. (A subtree is *reduced* if it contains all the query terms, but none of its proper subtrees also includes all of them.) In comparison to paths, it is considerably more difficult to develop a practically efficient algorithm for finding the top-ranked reduced subtree under constraints. In earlier work [4], we developed an algorithm that finds a 2-approximation of the minimal-height reduced subtree under inclusion and exclusion constraints; that algorithm was used within Lawler-Murty's procedure in an implementation of an engine for keyword search over data graphs.

2.2 Progressive Lower Bound

Although our techniques are general, we require the implementation of SolveOpt(D, P, I, E) to have the following property. It should be an iterative process, where each iteration gets closer to the solution, and provides a lower bound on the solution. Moreover, we assume that the lower bound improves with each additional iteration. This property, called *progressive lower bound*, usually holds for the problems solved by means of Lawler-Murty's procedure. For example, for the problem of path enumeration, Dijkstra's algorithm is used for finding the shortest path. Each iteration of this algorithm gives (when popping the priority queue) a better lower bound on the length of the shortest path between the given pair of nodes. Other examples are algorithms in the spirit the Viterbi algorithm [22] (which have been used within Lawler-Murty's procedure [10]).

3. FREEZING SPAWNED PROBLEMS

After popping the top of the queue (in line 6 of Figure 1), new optimization problems are spawned and they need to be solved (in line 10) before the top of the queue can be removed again. Thus, even if only one of the spawned problems requires a long computation, the whole algorithm is slowed down considerably. In principle, there is no need to completely solve the spawned problems before popping the next element from the top of the queue. Instead, it is sufficient to know that every spawned problem either has already been solved or cannot produce a solution that is better (i.e., has a smaller value) than the current top of the queue. By employing the progressive lower bound, we may conclude that the latter part of the condition holds.

We can suspend the computation of a spawned problem when it reaches a lower bound that is greater than the value at the top of the queue. Ideally, we would like to freeze a snapshot of the (state of the) computation. However, this requires a lot of memory. So, we simply freeze the problem itself and record the lower bound that has been reached. Formally, a *frozen* problem is a triplet $[I, E, l]$, where I and E are the sets of inclusion and exclusion constraints, respectively, and l is the lower bound that has been reached when the computation is suspended. Thus, when the computation of a frozen problem is resumed, it should start from scratch.

When a problem is frozen, its triplet $[I, E, l]$ is inserted into the queue (and the lower bound l is used as the value that determines the position in the queue). Thus, the queue stores triplets of the form $[I, E, X]$, where X is either a solution or a lower bound (and it is possible to distinguish between the two).

An important issue is how to decide when to freeze the computation of an optimization problem. We could do it as soon as the lower bound becomes equal to or greater than the value at the top of the queue. However, in order to avoid freezing and restarting of the same problem too often, we use a *freezing threshold*. We freeze an optimization problem when its lower bound exceeds the freezing threshold. We set the freezing threshold by applying the function F to the value v at the top of the queue. Based on some experiments described in Section 8.2, we have chosen $F(v) = 1.1v$, namely, the freezing threshold is 10% more than the value at

Algorithm: *Serial-with-Freezing*(D, P)

```
1:  Queue ← an empty priority queue
2:  S ← SolveOpt(D, P, ∅, ∅)
3:  if S ≠⊥ then
4:     Queue.insert([∅, ∅, S])
5:  while Queue ≠ ∅ do
6:     [I, E, X] ← Queue.removeTop()
7:     if Queue ≠ ∅ then
8:        f ← F(Queue.topValue())
9:     else
10:       f ← ∞
11:    /* note that in lines 16 and 22, SolveOpt returns a solu-
          tion or a lower bound l such that l > f */
12:    if X is a solution then
13:       print(X)
14:       create the constraint pairs [I_1, E_1], ..., [I_n, E_n] for
          the optimization problems spawned by [I, E, X]
15:       for i ← 1, ..., n do
16:          X_i ← SolveOpt(D, P, I_i, E_i)
17:          if X_i ≠⊥ then
18:             Queue.insert([I_i, E_i, X_i])
19:             if [I_i, E_i, X_i] becomes the top of Queue then
20:                f ← F(value(X_i))
21:    else /* X is a lower bound l */
22:       X' ← SolveOpt(D, P, I, E)
23:       if X' ≠⊥ then
24:          Queue.insert([I, E, X'])
```

Figure 2: Serial algorithm with freezing.

the top of the queue. For the sake of efficiency, the algorithm stores $F(v)$ in a dedicated variable f, which is updated when either the queue is popped or a new element is inserted and becomes the new top of the queue. If the queue becomes empty after popping its top, then the freezing threshold is set to infinity.

Figure 2 describes the incorporation of freezing in Lawler-Murty's procedure. The first six lines are the same as those of Figure 1. In particular, line 6 pops the queue, but now X could be either a solution or a lower bound on a frozen problem. Lines 7–10 update the freezing threshold f as follows. If the queue is not empty, then line 8 assigns to f the result of applying F to the value at the top of the queue. Otherwise (i.e., the queue is empty), infinity is assigned to f. Line 11 is a comment that says the following. Inside the loop of line 5, an execution of SolveOpt(D, P, I, E) tests whether the progressive lower bound l exceeds f. This test is done whenever l changes. If $l > f$ becomes true before a solution is found, then SolveOpt(D, P, I, E) returns l. Line 12 tests whether X is a solution. If so, lines 13–20 are executed. The first six of them (i.e., lines 13–18) are the same as lines 7–12 of Figure 1, except for the fact that the X_i returned in line 16 could be a lower bound rather than a solution. Note that (as earlier) if X_i is $\perp$, it means that there is no solution. Line 19 checks whether the newly inserted triplet has become the top of the queue. If so, the freezing threshold is updated immediately in line 20. Lines 22–24 are executed if X is a lower bound. In this case, there are no spawned optimization problems. The computation of the frozen problem starts from scratch in line 22, and its returned value X' is either a solution or a new lower bound. The triplet $[I, E, X']$ is inserted into the queue (unless $X' = \perp$, i.e., no solution). Note that there is no need to update the freezing threshold after the insertion, because this will be done in lines 7–10 during the next iteration of the main loop of line 5.

4. RELEVANCE MONITORING

Our goal is to parallelize Lawler-Murty's procedure. The basic idea is to handle the spawned optimization problems in parallel. (Freezing is not used in the parallel algorithm of this section.) Thus, a single, dedicated thread executes the *main task*, which is the algorithm of Figure 1 except for lines 10–12. Instead of solving the spawned problems, the main task creates a new *optimization task* for each one of these problems. A thread pool is employed to run the optimization tasks. Each optimization task executes lines 10–12 of the original Lawler-Murty's procedure of Figure 1.

The problem with the above approach is that the main task has to wait until all the spawned problems are solved before popping the next element from the queue. Consequently, when some but not all of the optimization tasks require a long time, the CPU cores are underutilized. To overcome this problem, we use *relevance monitoring*, which has some similarity to freezing, but the two are not the same. Relevance monitoring sometimes makes it possible to pop the next element from the top of the queue before all the spawned problems are solved. Thus, the next solution can be printed more quickly. Unlike freezing, however, relevance monitoring cannot suspend the execution of an optimization task, even if that task runs for a very long time.

The main idea is to employ the progressive lower bound (of Section 2.2) as follows. An optimization task monitors whether the lower bound exceeds the value at the top of the queue. If so, the main task does not have to wait for that optimization task. That is, the main task continues with the next iteration when the following holds. Every optimization

Algorithm: *Relevance-Monitoring*(D, P) (main task)

```
1:  Semaphore ← a counting semaphore initialized to 0
2:  Queue ← an empty priority queue
3:  S ← SolveOpt(D, P, ∅, ∅)
4:  if S ≠⊥ then
5:     Queue.insert([∅, ∅, S])
6:  while Queue ≠ ∅ do
7:     [I, E, S] ← Queue.removeTop()
8:     print(S)
9:     if Queue ≠ ∅ then
10:       v ← Queue.topValue()
11:    else
12:       v ← ∞
13:    lock the task list T
14:    for each task t on T do
15:       t.toRelease ← true
16:    m ← T.size()
17:    unlock T
18:    create the constraint pairs [I_1, E_1], ..., [I_n, E_n] for the
       optimization problems spawned by [I, E, S]
19:    for i ← 1, ..., n do
20:       create an optimization task for (D, P, I_i, E_i), and
          set its toRelease to true
21:    Semaphore.acquire(n + m)
```

Figure 3: Main task with monitoring.

Task for optimization problem (D, P, I, E)

```
1: the task (i.e., this) adds itself to the task list T
2: S ← SolveOpt(D, P, I, E) /* also runs the relevance monitoring of Figure 5 whenever the lower bound changes */
3: if S ≠ ⊥ then
4:    Queue.insert([I, E, S])
5:    if [I, E, S] becomes the top of Queue then
6:       v ← Queue.topValue()
7: lock the task list T
8: the task (i.e., this) removes itself from T
9: unlock T
10: if toRelease = true then
11:    Semaphore.release(1)
```

Figure 4: Optimization task with monitoring.

Relevance Monitoring

```
1: if toRelease = true then
2:    if lower-bound > relevance-threshold then
3:       toRelease ← false
4:       Semaphore.release(1)
```

Figure 5: Code of relevance monitoring.

task either has finished or has a lower bound (on its solution) that is greater than the value at the top of the queue.

The *relevance threshold* is the value v at the top of the queue. If the lower bound of an optimization task t exceeds v, then t is no longer relevant to the current iteration of the main task. However, when the next iteration starts, the queue is popped and, as a result, the relevance threshold changes. Hence, task t becomes relevant again (if it has not already finished).

An optimization task is created with a Boolean variable called *toRelease*, which is initialized to **true**. This variable indicates whether the optimization task is relevant to the current iteration of the main task. When the optimization task is no longer relevant, it *releases a permit* and changes its *toRelease* to **false**. The main task waits until it *acquires permits* from all the optimization tasks. The *task list*, denoted by T, stores pointers to the optimization tasks.

The parallel algorithm that uses relevance monitoring consists of the main task of Figure 3 and the optimization task of Figure 4. We first describe the main task. This task uses a counting semaphore, which is initialized to zero in line 1. Lines 2–8 of Figure 3 are the same as lines 1–7 of Figure 1. In particular, the queue is popped in line 7 and the obtained solution is printed in line 8. In lines 9–12, the new relevance threshold is assigned to v. Specifically, if the queue is not empty, then v gets the value at the top of the queue; otherwise, v becomes infinity. The loop of line 14 is needed so that all the tasks will become relevant again. In this loop, the task list is traversed; for every task, line 15 sets its *toRelease* to **true**. An exclusive lock on the task list is obtained before the traversal (in line 13). After the traversal ends, line 16 sets the variable m to the number of tasks on the list, and then the lock is released in line 17. Line 18 creates the constraints for the spawned optimization problems. The loop of line 19 creates optimization tasks for the spawned problems. Each created task (in line 20) has **true** for its *toRelease*. In line 21, the main task needs to acquire $n + m$ permits before continuing with the next iteration. Note that n is the number of spawned optimization problems and m is the number of tasks that were traversed on the task list.

Next, we describe the optimization task of Figure 4. In line 1, the task adds itself to the task list. Line 2 calls the procedure for solving the optimization problem. This line uses a modified algorithm (for the optimization problem) so that whenever the lower bound changes, the relevance monitoring of Figure 5 is executed. Namely, if *toRelease* is **true** and the lower bound exceeds the relevance threshold, then *toRelease* is changed to **false** and one permit is released.

Now, we continue with the description of Figure 4. Note that line 2 does not do any freezing; that is, it returns either a solution or $\perp$ (if there is no solution). Line 3 tests whether a solution was found and if so, line 4 adds the corresponding triplet to the queue. Line 5 tests whether that triplet is the new top of the queue, and if so, line 6 updates the relevance threshold. In line 8, the task removes itself from the task list. Before this operation, the list is locked exclusively (in line 7). The lock is released (in line 9) after the removal. Finally, in line 10, the task tests whether its *toRelease* is **true** and if so, it releases one permit in line 11.

We now discuss the issue of locks. For clarity's sake, the pseudo code explicitly shows locking and unlocking only in some places. Additional locking is done as follows. First, the queue is implemented as a thread-safe collection. Second, line 1 of Figure 4 actually includes the steps of obtaining an exclusive lock on T before the task adds itself to T, and releasing the lock when this operation ends. Third, the following segments of code actually begin with obtaining a lock on the relevant *toRelease*, and they end with unlocking that *toRelease*: *(1)* Line 15 of Figure 3, *(2)* Lines 10–11 of Figure 4, *and (3)* Lines 3–4 of Figure 5. Appendix B proves the correctness of the parallel algorithm of this section.

5. PARALLELISM WITH FREEZING

Recall that the serial algorithm with freezing (Figure 2) enhances the efficiency of Lawler-Murty's procedure (Figure 1) by suspending long computations of optimization problems. The parallel algorithm with relevance monitoring gains efficiency by utilizing multiple cores. In this section, we combine freezing and relevance monitoring. Due to space limitations, our discussion is short and high level; the full details are in Appendix C.

We found that a straightforward combination of freezing and relevance monitoring suffers from low utilization of the cores, for the following reason. Recall that the algorithm with freezing stores two types of triplets in the queue: solutions and lower bounds of frozen problems. Popping a solution creates work for multiple threads (to solve the spawned problems). However, popping a lower bound launches only one thread that runs the optimization task t for solving the frozen problem. One could suggest, in that case, to further pop additional triplets from the queue to utilize available threads. However, that could lead to the following violation of the algorithm's correctness. Recall that if one of these additional triplets is for a solution S, then the main task needs to print S; but it *cannot* print S, since the ongoing execution of t may produce a solution that is better than S!

So, instead of loading available threads by popping triplets from the queue, we let these threads handle only frozen problems. For that, we use two priority queues (instead of just

Table 1: Dataset statistics.

dataset	file size	#nodes	#edges
M	1.44M	22.5K	139K
PD	74.2M	356K	6.3M
FD	692M	4.16M	70.5M

one). One is the ordinary queue of Lawler-Murty's procedure, and it stores solutions; we will continue to refer to it as "the queue." The second is a priority queue, called the *freezer*, that stores frozen problems (the top of this queue has the frozen problem with the smallest lower bound).

Finally, when assigning frozen problems to threads, we will select those with a small lower bound, because they are likely to produce better solutions. Specifically, in addition to the relevance threshold v and the freezing threshold f, we use an *unfreezing threshold* u, which is between v and f. A frozen problem is assigned to a thread if its lower bound is below u. (Based on experiments, we found that $u = 1.05v$ and $f = 1.1v$ are good choices.)

6. EARLY FREEZING

When the main task pops a triplet $[I, E, S]$, it may create too few spawned optimization problems to fully utilize all available threads. To overcome this problem, we introduce the technique of *early freezing*. Again, our discussion is short and high level; the full details are in Appendix D. The main idea is to create spawned problems when a solution S is found and inserted into the queue (by an optimization task), rather than when it is popped and printed (by the main task). The spawned problems are immediately inserted into the freezer upon their creation. However, doing so raises the following problem with the current design of the algorithm. The main task is the one that creates spawned problems, but it is not aware of when the computation of an optimization problem is done. So, we delegate the task of spawning problems to the optimization tasks. In addition, before an optimization task terminates, it may pop the freezer and create a new task, in order to keep the available threads fully utilized.

In principle, now the main task needs only to remove solutions from the queue and print them. However, the main task can do it so quickly that some frozen problems will have a lower bound that is smaller than the current solution at the top of the queue. Therefore, the main task should start a new iteration by first checking whether there are frozen problems with a lower bound that is smaller than the solution at the top of the queue. If so, the main task should create new optimization tasks for these frozen problems before printing the next solution.

7. MULTIPLE POPPING

A simple optimization, which can be easily incorporated in all the parallel algorithms we developed, is based on the following (straightforward) observation. If we pop a solution from the queue and the next solution has the same value, then we can immediately pop the next one as well. We can easily modify each of our algorithms so that all the solutions with the same weight are popped at the beginning of each iteration, instead of one at a time. This optimization is called *multiple popping,* and as shown in Section 8, it is often surprisingly beneficial.

8. EXPERIMENTS

As a case study for our algorithms, we used the problem of enumerating reduced subtrees, which is central to keyword search over data graphs [1, 4, 7, 11]. The input consists of a data graph and a query (i.e., a set of keywords). The latter is the instance of the enumeration problem. The algorithm of [4] was used to implement the procedure that solves the optimization problem under constraints.

8.1 The Experimental Setup

The experiments are based on two datasets. One is Mondial[2] and the other is DBLP.[3] From DBLP, we created a third, smaller dataset that includes only proceedings and journals that are related to database research. It should be emphasized that we did *not* remove any fields from the original datasets. We use the following names for the datasets. *M* is Mondial, *PD* is the partial DBLP, and *FD* is the full DBLP. Some statistics about the datasets are given in Table 1. In particular, the file size refers to the source data in XML, whereas the numbers of nodes and edges are for the data graph generated from the XML.

For each dataset, we use two different schemes for assigning weights to the nodes and edges of the graph. These weighting schemes, which are described in Appendix E, have been proven in practice as effective in generating relevant answers to a large number of queries. Thus, each dataset has two versions, which are called M1, M2, PD1, etc.

For each dataset, the queries range from 2 to 10 terms. For each size (i.e., number of terms), we created four queries. Statistics about the keyword selectivity of the queries are given in Appendix F. To accommodate space limitations, we do not show results for every query size separately, but rather divide the queries into three groups: *short* (2–4 query terms), *medium* (5–7 query terms), and *long* (8–10 query terms). The running time for each group is the average over all queries in the group, and it does not include overheads, such as looking up a keyword index and so on.

The experiments were run on a Linux server with two Intel Xeon X5550 (2.67GHz) processors and 48GB memory (1333MHz RDIMMs). Each processor has four cores (with two hyper threads per core). The algorithms were implemented in Java 1.6 and executed by a Java 64-bit server.

8.2 Serial Algorithms

We considered two serial algorithms: the original Lawler-Murty's procedure of Figure 1, and the serial algorithm with freezing (abbr. SF) of Figure 2. The detailed experimental results are in Appendix G, and they show the following. Freezing cuts the running time by at least one third, and in many cases, by one half or more. For 100 solutions, the reduction is at least one half in most cases, and two thirds in half of the cases. Almost always, freezing saves a larger percentage of the running time when more solutions are generated. It is not surprising: as the number of solutions grows, the difference in weights becomes smaller and, hence, frozen problems have to be recomputed less frequently.

Figure 6 shows how the running time changes when the freezing threshold varies. This experiment was done on the

[2] `http://www.dbis.informatik.uni-goettingen.de/Mondial/#XML`

[3] `http://dblp.uni-trier.de/xml/`

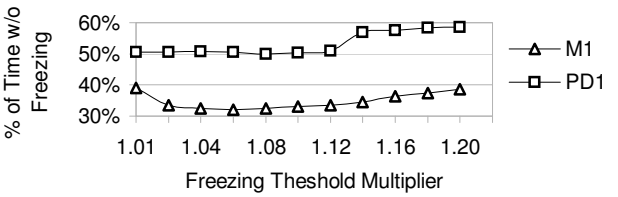

Figure 6: Varying the freezing threshold.

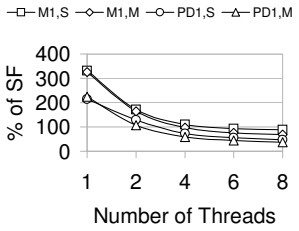

Figure 7: RM short.

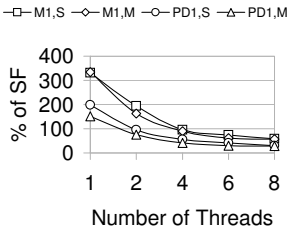

Figure 8: RM long.

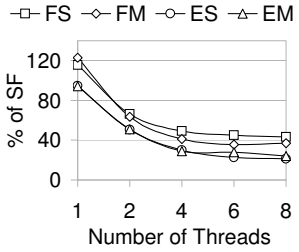

Figure 9: M1 short.

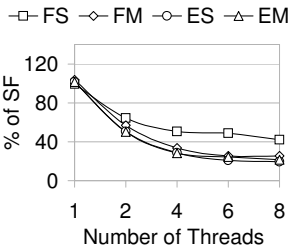

Figure 10: M1 medium.

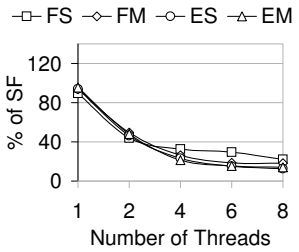

Figure 11: M1 long.

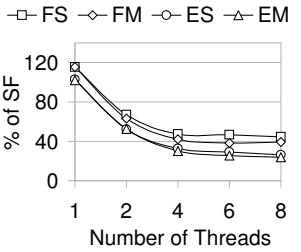

Figure 12: M2 short.

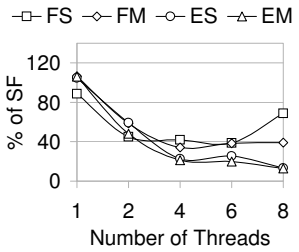

Figure 13: M2 medium.

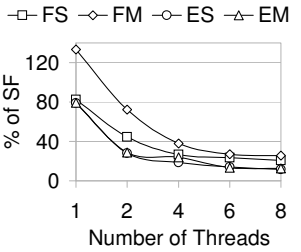

Figure 14: M2 long.

datasets M1 and PD1 when generating 100 answers. The figure shows the running time of SF as a percentage of the original Lawler-Murty's procedure. As can be seen, reasonable choices for $F(v)$ are between $1.04v$ and $1.12v$.

8.3 Parallel Algorithms

We show results for the parallel algorithms using SF as the baseline. That is, the running time is given as a percentage of (the running time of) SF. The results are for 100 solutions. The graphs depict the running time vs. the number of threads in the pool for executing optimization tasks. Note that in addition to those threads, there is also one thread for the main task.

First, we discuss the relevance-monitoring (RM) algorithm of Section 4. Figures 7 and 8 present the running time of RM vs. the number of threads for short and long queries, respectively. Each figure is for two datasets, namely, M1 and PD1. For each dataset, there are two graphs, one is for single popping (S) and the other is for multiple popping (M). For one thread, the running times are much worse than those of SF, because the RM algorithm does not use freezing. Only when there are 4 or more threads, does the running time start to be better than SF. This shows that the effect of freezing cannot be realized by merely using more threads. We do not give more results for the RM algorithm due to a lack of space and, anyhow, they are about the same. The bottom line is clear: parallelizing without incorporating freezing is not an effective approach.

We now consider the parallel algorithms with freezing (Section 5) and early freezing (Section 6). We give results only for 100 solutions, because those for 10 are similar. That similarity is due to the fact that both the algorithm that is tested and the baseline use freezing, so the ratio of the running times does not depend on the number of solutions.

The results are organized in twelve figures as follows. Each figure is for one dataset (M1, M2, FD1, or FD2) and one group of query sizes (short, medium, or long). (The results for PD1 and PD2 do not provide any additional insight and, hence, are not shown.) Each figure has four graphs for the following algorithms: freezing with single popping (FS), freezing with multiple popping (FM), early freezing with single popping (ES), and early freezing with multiple popping (EM). For example, Figures 9, 10, and 11 are for running short, medium and long queries, respectively, on M1.

The main conclusions from Figures 9–20 are as follows. Freezing with multiple popping is usually better than freezing with single popping. However, when early freezing is used, there is no significant difference between the two types of popping. The bottom line is that early freezing is the best algorithm (except for some cases where only one or two threads are used). The difference between freezing and early freezing is most noticeable in short queries, for the following reason. When queries are short, solutions are small. Hence, a solution spawns only a few optimization problems and, as a result, freezing underutilizes the thread pool.

The speedup of the early-freezing algorithm is excellent. For eight threads, the percentages measuring the running time have averages (over all datasets) of 13, 16 and 24 for long, medium and short queries, respectively. In the case of six threads, the averages are almost the same: 15, 20 and 26. (These averages are for multiple popping; for single popping they are almost identical: 12, 17, 25 for 8 threads, and 15, 21, 25 for 6 threads.) Note that 12% amounts to a speed up by a factor of 8, which is almost too good to be true for 8 threads. The reason for such a performance is the following. When several optimization problems are solved in parallel, chances are higher that the freezing threshold will occasionally decrease. (Lines 9–11 of Figure 26 decrease the freezing threshold when the inserted solution becomes the top of the queue.) Thus, not only does the early-freezing algorithm operate in parallel, its total amount of work is actually less than that of the serial algorithm.

We now explain why there is hardly any improvement (and even a degradation in Figure 13) in the speedup from

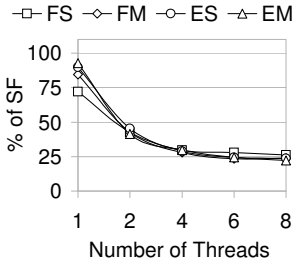

Figure 15: FD1 short.

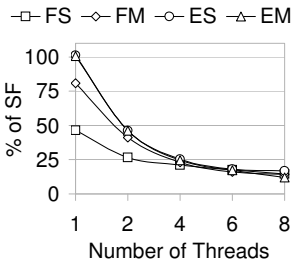

Figure 16: FD1 medium.

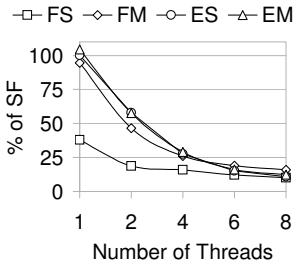

Figure 17: FD1 long.

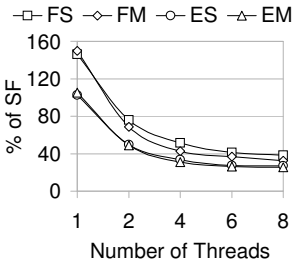

Figure 18: FD2 short.

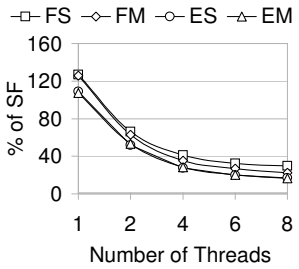

Figure 19: FD2 medium.

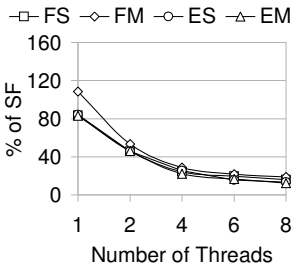

Figure 20: FD2 long.

six to eight threads. Recall that we have eight cores in total. (The hyper threads of each core do not add true parallelism.) In addition to the thread pool (for executing optimization tasks), there is also one thread for the main task, as well as some other threads for garbage collection, operating-system tasks, and so on. This means that when six threads are allocated to the thread pool, all the cores are employed.

Next, we explain why FS (i.e., freezing with serial popping) is sometimes better than FM. When multiple popping is used, the freezing threshold f increases more quickly, because the top of the queue becomes larger faster. Indeed, f may go down again as spawned problems are solved, but that tends to happen more slowly when there are fewer threads. Thus, multiple popping is not effective when there is only a small number of threads. As the number of threads grows, FM catches up FS and even surpasses it, because FM is more adept at keeping many threads busy. In contrast, early freezing does not suffer from this phenomenon, because it avoids creating new optimization tasks when the thread pool is fully utilized (see lines 19–26 of Figure 26).

Sometimes, FS on a single thread is faster than the serial baseline SF. For example, in Figure 17, its running time is just 38% of the baseline. The reason for that is the following. FS and SF are quite different in how they operate, even when FS has just a single thread for optimization tasks (and another one for the main task). For example, FS has both a queue and a freezer, whereas SF has only a queue. As a result of those differences, there is no guarantee that FS and SF produce equal-weight solutions in the same order. Sometimes there are many equal-weight solutions. Moreover, some of those solutions spawn a few optimization problems whereas others spawn many. An algorithm that first prints solutions that spawn only a few problems will run faster than an algorithm that starts with solutions that spawn many problems.

9. REFERENCES

[1] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.

[2] S. Cohen, B. Kimelfeld, and Y. Sagiv. Enumerating large query results, 2009. An ICDE 2009 tutorial.

[3] H. N. Gabow. Two algorithms for generating weighted spanning trees in order. *SIAM J. Comput.*, 6(1):139–150, 1977.

[4] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, pages 927–940. ACM, 2008.

[5] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[6] D. Johnson, M. Yannakakis, and C. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27:119–123, 1988.

[7] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[8] N. Katoh, T. Ibaraki, and H. Mine. An algorithm for finding k minimum spanning trees. *SIAM J. Comput.*, 10(2):247–255, 1981.

[9] N. Katoh, T. Ibaraki, and H. Mine. An algorithm for the k best solutions of the resource allocation problem. *J. ACM*, 28(4):752–764, 1981.

[10] B. Kimelfeld and C. Ré. Transducing markov sequences. In *PODS*, pages 15–26. ACM, 2010.

[11] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.

[12] B. Kimelfeld and Y. Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In *NGITS*, pages 141–152. Springer, 2006.

[13] B. Kimelfeld and Y. Sagiv. Twig patterns: From XML trees to graphs. In *WebDB*, pages 26–31, 2006.

[14] B. Kimelfeld and Y. Sagiv. Combining incompleteness and ranking in tree queries. In *ICDT*, volume 4353 of *Lecture Notes in Computer Science*, pages 329–343. Springer, 2007.

[15] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[16] Y. Luo, W. Wang, and X. Lin. SPARK: A keyword search engine on relational databases. In *ICDE*, pages 1552–1555. IEEE, 2008.

[17] K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

[18] A. G. Parameswaran and H. Garcia-Molina. Recommendations with prerequisites. In *RecSys*, pages 353–356. ACM, 2009.

[19] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD Conference*, pages 681–694. ACM, 2009.

[20] I. Takigawa and H. Mamitsuka. Probabilistic path ranking based on adjacent pairwise coexpression for metabolic transcripts analysis. *Bioinformatics*, 24(2):250–257, 2008.

[21] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. *PVLDB*, 1(1):785–796, 2008.

[22] A. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, 1967.

[23] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

APPENDIX

A. ABOUT LAWLER-MURTY'S

We now intuitively explain how Lawler-Murty's procedure works. The circle of Figure 21(a) contains the solutions that we want to enumerate. Pictorially, solutions that are closer to the center have a higher rank. So, S_1 is the top-ranked solution. An optimization algorithm that computes the best solution can be used for finding S_1. After printing S_1, the remainder of the circle is divided into disjoint subsets. Generally, the number of these subsets depends on S_1. In Figure 21(a), it is assumed that there are four subsets and they are shown as the slices separated by thick lines. Now, we should find the top-ranked solution in each slice, thereby obtaining S_2, S_3, S_5 and S_7. These four solutions are inserted into a priority queue. Clearly, the second solution in the ranking order is the one at the top of the queue, namely, S_2. After printing S_2, the remainder of the slice (from which S_2 was taken) is divided into sub-slices, namely, the ones separated by dotted lines. In each sub-slice, we find the top-ranked solution and add it to the queue. Now, the queue has six solutions, namely, S_3, S_5, S_7 and the three new solutions S_6, S_{10} and S_{11}. The algorithm likewise continues until the queue is empty (or sufficiency many solutions have been printed): the solution at the top of the queue is popped and printed, its slice is divided into sub-slices, the top-ranked solution in each sub-slice is added to the queue, and so on. So, two problems should be solved when using Lawler-Murty's procedure: how to divide each slice into sub-slices, and how to find the top-ranked solution in each slice. Lawler and Murty addressed the first problem by means of *inclusion* and *exclusion* constraints, as explained in Section 2.

As a concrete example of using Lawler-Murty's procedure, we consider Yen's algorithm [23] for enumerating (by increasing length) all simple paths between two given nodes. In Figure 21(c), the shortest path between nodes a and d has three edges and it goes through nodes b and c (Dijkstra's algorithm can be used for finding this path). So, three subsets are created, as illustrated in Figure 21(b): *(1)* the subset of all paths between a and d that do not contain the edge (a, b), *(2)* the subset of all paths between a and d that contain the edge (a, b), but not (b, c), *and (3)* the subset of all paths between a and d that contain the edges (a, b) and (b, c), but not (c, d). In other words, each subset comprises all paths that contain some (possibly empty) prefix of the shortest path from a to d and do not contain the next edge that follows that prefix. Observe that, as required, the subsets are indeed disjoint, and their union contains all paths between a and d except for the (already printed) shortest one.

Now we have to find the shortest path in each subset. This can be done easily by a reduction to the regular shortest-path problem. For example, suppose that we want to find the shortest path in the third subset. Nodes a and b (as well as their incident edges) are removed from the graph. The edge (c, d) is also removed and, in the resulting graph that is shown in Figure 21(d), we find the shortest path between c and d. We add to that path the edges (a, b) and (b, c) in order to obtain the shortest path of the third subset.

B. PROOF OF CORRECTNESS

In this section, we prove the correctness of the parallel algorithm with relevance monitoring (described in Section 4). Let *RM* denote the parallel algorithm with relevant monitoring (Figure 3), and let *LM* denote the original Lawler-Murty's procedure (Figure 1). It suffices to show that RM prints solutions in the same order as LM (possibly, up to reordering among solutions with the same value). At the conceptual level, the main difference between LM and RM is in the content of the queue when a solution is popped by the main task. In LM, the queue contains the solutions of all the optimization problems spawned thus far. In RM, the queue contains just some of those solutions, while the rest are *hidden* since they are still computed by optimization tasks of the task list. However, lines 9–21 of Figure 3 (and the code of an optimization task of Figure 4) ensure that when the main task pops a solution in line 7, every hidden solution has a lower bound that is larger than the value at the top of the queue and, hence, would not have been chosen even if it had been on the queue. Specifically, this is done as follows.

Lines 9–12 of Figure 3 update the relevance threshold v to the value at the top of the queue. For all the optimization tasks traversed on the task list (in lines 13–17) or created in line 20, the *toRelease* is set to **true**. Line 21 waits for permits from all of these tasks, while each one of them releases a permit in two cases: line 2 of Figure 4, and line 11 of that figure. In the former case, the lower bound exceeds v (hence, as said above, the solution would not have been chosen). In the latter case, the solution has been computed and inserted into the queue (hence, is no longer hidden).

Additional important observations are as follows. Operations on the task list require a lock. An optimization task releases a permit (if needed) in lines 10–11 of Figure 4 only after removing itself from the task list in line 8. Finally, when the relevance threshold is updated by an optimization task in line 6 of Figure 4, it can only decrease.

C. PARALLEL-FREEZING ALGORITHM

In this section, we give the details of the algorithm discussed in Section 5, which combines freezing and relevance monitoring. Figure 22 describes the main task. Lines 1–22 are the same as lines 1-20 of Figure 3, except for the following additions: line 3 initializes the freezer, and lines 14 updates f immediately after the relevance threshold v gets a new value.

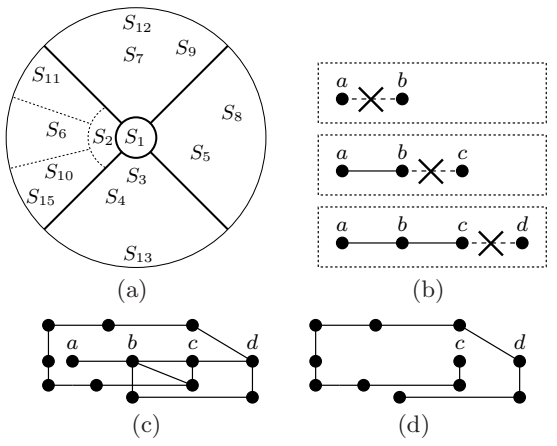

Figure 21: Illustration of *(a)* Lawler-Murty's procedure *and (b–d)* Yen's algorithm.

Algorithm: *Parallel-Freezing*(D, P) (main task)

```
1: Semaphore ← a counting semaphore initialized to 0
2: Queue ← an empty priority queue
3: Freezer ← an empty priority queue
4: S ← SolveOpt(D, P, ∅, ∅)
5: if S ≠⊥ then
6:    Queue.insert([∅, ∅, S])
7: while Queue ≠ ∅ do
8:    [I, E, S] ← Queue.removeTop()
9:    print(S)
10:   if Queue ≠ ∅ then
11:      v ← Queue.topValue()
12:   else
13:      v ← ∞
14:   f ← F(v) /* update the freezing threshold */
15:   lock the task list T
16:   for each task t on T do
17:      t.toRelease ← true
18:   m ← T.size()
19:   unlock T
20:   create the pairs of constraints [I_1, E_1], ..., [I_n, E_n] for
      the optimization problems spawned by [I, E, S]
21:   for i ← 1, ..., n do
22:      create an optimization task for (D, P, I_i, E_i, 0), and
         set its toRelease to true
23:   k ← 0
24:   u ← U(v) /* update the unfreezing threshold */
25:   while Freezer ≠ ∅ and Freezer.topValue() < u do
26:      [I, E, l] ← Freezer.removeTop()
27:      create an optimization task to solve (D, P, I, E, l),
         and set its toRelease to true
28:      k ← k + 1
29:   Semaphore.acquire(n + m + k)
```

Figure 22: Main task with parallel freezing.

The main difference between Figures 3 and 22 is the addition of lines 23–28 to the main loop of line 6 (which is line 7 in Figure 22). These lines remove from the freezer all the optimization problems that have a lower bound that is smaller than the unfreezing threshold u (which is updated in line 24). Line 27 creates an optimization task for each of these problems. As usual, the new tasks have their *toRelease* set to **true**. The variable k counts the number of tasks created in the loop of line 25. Line 29 is similar to the last line of Figure 3, except that now the main task has to wait for $n + m + k$ permits.

A minor, yet important difference between Figure 22 and the algorithms of previous sections is the following. An optimization task has five (rather than four) parameters. The fifth is the lower bound l that is returned when an optimization problem is frozen (as explained below). When tasks for spawned optimization problems are created in line 22, the fifth parameter is zero.

Figure 23 describes the optimization task, which differs from Figure 4 in two places. The first is the implementation of SolveOpt(D, P, I, E, l) in line 2. In addition to solving the optimization problem, this procedure also executes the code of Figure 24 whenever the lower bound changes. This code is obtained from that of Figure 5 by two modifications. First, lines 1–2 are new. Line 1 tests whether the lower bound is greater than the freezing threshold and if so, line 2 returns that bound and SolveOpt(D, P, I, E, l) terminates. Second, lines 3–6 of Figure 24 are the same as the four lines of Figure 5, except that in the test of line 2 (of Figure 5) we added the disjunct that checks whether l is greater than the relevance threshold. Note that the optimization task can release a permit as soon as it starts executing line 2 of Figure 23, provided that the following holds: the lower bound l from the previous computation of the optimization problem exceeds the current relevance threshold. This may happen if the relevance threshold has not increased sufficiently (or even decreased) since the problem was unfrozen.

SolveOpt(D, P, I, E, l) is also modified as follows. When it starts to recompute a frozen optimization problem (i.e., $l > 0$), it first of all checks whether l is greater than the current freezing threshold f. (Recall that l is the lower bound that was obtained when that optimization problem was previously computed.) If $l > f$, then SolveOpt(D, P, I, E, l) immediately returns l and terminates. Our experiments have shown that, in some cases, this test cuts down a lot of unnecessary work.

The second difference between Figures 4 and 23 is the replacement of lines 3–6 of the former with lines 3–10 of the latter. Line 3 of Figure 23 tests that line 2 did not return $\perp$ (otherwise, there is no solution and the execution continues to line 11). If the test is true, then line 4 checks whether a solution was found in line 2. If so, line 5 inserts the corresponding triplet into the queue. Line 6 tests whether that triplet has become the top of the queue and if so, lines 7 and 8 update the relevance threshold v and the freezing threshold f, respectively. If the test of line 4 is **false**

Task for optimization problem (D, P, I, E, l)

```
1: the task (i.e., this) adds itself to the task list T
2: X ← SolveOpt(D, P, I, E, l) /* this line also executes the
   code of Figure 24 whenever the lower bound changes */
3: if X ≠⊥ then
4:    if X is a solution then
5:       Queue.insert([I, E, X])
6:       if [I, E, X] becomes the top of Queue then
7:          v ← Queue.topValue()
8:          f ← F(X)
9:    else
10:      Freezer.insert([I, E, X])
11: lock the task list T
12: the task (i.e., this) removes itself from T
13: unlock T
14: if toRelease = true then
15:    Semaphore.release(1)
```

Figure 23: Optimization task with parallel freezing.

Monitoring with Freezing

```
1: if lower-bound > f then
2:    return lower-bound
3: if toRelease = true then
4:    if lower-bound > relevance-threshold or
      l > relevance-threshold then
5:       toRelease ← false
6:       Semaphore.release(1)
```

Figure 24: Monitoring with parallel freezing.

Algorithm: *Early-Freezing*(D, P) (main task)

```
 1: Semaphore ← a counting semaphore initialized to 0
 2: Queue ← an empty priority queue
 3: Freezer ← an empty priority queue
 4: S ← SolveOpt(D, P, ∅, ∅)
 5: if S ≠⊥ then
 6:    Queue.insert([∅, ∅, S])
 7:    create the pairs [I_1, E_1], ..., [I_n, E_n] of constraints for
       the optimization problems spawned by [∅, ∅, S]
 8:    for i ← 1, ..., n do
 9:       Freezer.insert([I_i, E_i, value(S)])
10: while Queue ≠ ∅ or Freezer ≠ ∅ do
11:    if Queue ≠ ∅ then
12:       v ← Queue.topValue()
13:    else
14:       v ← ∞
15:    lock Freezer
16:    k ← 0
17:    lock T
18:    for each task t on T do
19:       if t.revealed = false then
20:          t.revealed ← true
21:          if t.l < v then
22:             t.toRelease ← true
23:             k ← k + 1
24:    unlock T
25:    while Freezer ≠ ∅ and Freezer.topValue() < v do
26:       [I, E, l] ← Freezer.removeTop()
27:       create an optimization task t to solve (D, P, I, E, l),
          and set both its toRelease and revealed to true
28:       add t to the task list T
29:       k ← k + 1
30:    unlock Freezer
31:    Semaphore.acquire(k)
32:    if Queue ≠ ∅ then
33:       [I, E, S] ← Queue.removeTop()
34:       print(S)
35:       if Queue ≠ ∅ then
36:          v ← Queue.topValue()
37:       else
38:          v ← ∞
39:       f ← F(v) /* update the freezing threshold */
40:       lock the task list T
41:       for each task t on T do
42:          t.toRelease ← true
43:          if t.revealed = false then
44:             t.revealed ← true
45:       m ← T.size()
46:       unlock T
47:       Semaphore.acquire(m)
```

Figure 25: Main task with early freezing.

(i.e., line 2 returned a lower bound), then line 10 inserts the corresponding triplet into the freezer.

D. EARLY-FREEZING ALGORITHM

In this section, we give the details of the early-freezing algorithm, which was discussed in Section 6. Figure 25 describes the main task. Lines 1–6 are the same as in Figure 22. In particular, line 6 inserts the first solution into the queue. Lines 7–9 create the optimization problems spawned by the first solution and insert their triplets into the freezer. Note that a triplet is inserted into the freezer with a lower bound that is equal to the value of the solution from which it was created. The main loop of line 10 is executed while either the queue or the freezer is nonempty. Lines 11–14 update the relevance threshold v. Lines 17–24 are needed to make sure that the main task is aware of optimization tasks that were created by other optimization tasks, as explained later. The loop of line 25 pops from the freezer (in line 26) all the triplets with a lower bound that is smaller than v. Line 27 creates optimization tasks for these triplets. Line 28 adds each new task to the task list T. Note that the freezer is locked between lines 15 and 30. The variable k counts the number of optimization tasks that have to release a permit before the main task can continue beyond line 31.

Line 32 checks that the queue is not empty. If so, the main task executes lines 33–47, where lines 33–46 are the same as lines 8–19 of Figure 22, except for the addition of lines 43–44 that will be explained later. In line 47, the main task waits for m permits before continuing with the next iteration of line 10.

Figure 26 describes the optimization task. Note that the task need not add itself to the task list, because that is done when it is removed from the freezer. Line 1 solves the optimization problem (and it operates exactly as line 2 of Figure 23). Lines 2–13 differ in two ways from lines 3–10 of

Task for optimization problem (D, P, I, E, l)

```
 1: X ← SolveOpt(D, P, I, E, l) /* this line also executes the
    code of Figure 24 whenever the lower bound changes */
 2: if X ≠⊥ then
 3:    if X is a solution then
 4:       if shutdown-bit = false then
 5:          create the pairs of constraints
             [I_1, E_1], ..., [I_n, E_n] for the optimization
             problems spawned by [I, E, X]
 6:          for i ← 1, ..., n do
 7:             Freezer.insert([I_i, E_i, value(X)])
 8:       Queue.insert([I, E, X])
 9:       if [I, E, X] becomes the top of Queue then
10:          v ← Queue.topValue()
11:          f ← F(X)
12:    else
13:       Freezer.insert([I, E, X])
14: lock the task list T
15: the task (i.e., this) removes itself from T
16: unlock T
17: if toRelease = true then
18:    Semaphore.release(1)
19: lock Freezer
20: p ← size of the thread pool
21: if T.size() < p and shutdown-bit = false then
22:    if Freezer ≠ ∅ and Freezer.topValue() < f then
23:       [I, E, l] ← Freezer.removeTop()
24:       create an optimization task t to solve (D, P, I, E, l),
          and set both its toRelease and revealed to false
25:       add t to the task list T
26: unlock Freezer
```

Figure 26: Optimization task with early freezing.

Table 2: Keyword selectivity (min, max, avg).

kw.	M			PD			FD		
2	1	3	1.3	55	13690	2763	1672	67550	21865
3	1	94	13	23	14154	3800	1049	183947	41566
4	1	227	17	20	31164	3375	583	220529	31541
5	1	441	27	2	1090	400	10	16890	5571
6	1	227	17	5	10223	1870	38	113313	22591
7	1	227	23	1	5706	783	6	43621	10098
8	1	401	19	26	8357	857	562	75923	9781
9	1	849	44	23	13690	1153	342	67550	10222
10	1	1438	51	18	14154	1823	234	183947	19376
All	1	1438	29	1	31164	1567	6	220529	16531

Figure 23. First, if X is a solution, then before inserting it into the queue, lines 5–7 create the triplets for the spawned problems and insert them into the freezer. Second, the test of line 4 is new, and is needed for dealing with the following practical issue.

When applying ranked enumeration, one usually desires just the first several (top-K) solutions in ranked order, rather than all of them; then, the main task is required to print only K solutions. Up to now, adjusting our algorithms to this requirement has not been a problem (and indeed, has been ignored in the paper), since the main task could force termination as soon as K solutions have been printed. But in early freezing, optimization tasks create new frozen problems and (as explained later) may also create new optimization tasks; therefore, terminating the whole parallel algorithm cannot be done solely by the main task. To accommodate that, we introduce a global *shutdown bit* that is initially set to **false**. To terminate, the main task needs just to set this bit to **true** (for clarity's sake, we did not write the code for doing that in Figure 25). In turn, the test of line 4 of Figure 26 ensures that new frozen problems are not created if the shutdown bit is **true**. Another reference to this bit (in line 21) is discussed later.

We continue with the description of the code in Figure 26. In lines 14–18 (which are the same as lines 11–15 of Figure 23), the task removes itself from the task list T and releases a permit if its *toRelease* is **true**.

Lines 19–26 ensure proper utilization of the thread pool, and they are executed while holding an exclusive lock on the freezer. If the current number of tasks on T is smaller than the size of the thread pool and the shutdown bit is **false**, then one triplet is popped from the freezer in line 23 provided that the freezer is not empty and its top is smaller than the freezing threshold. Lines 24–25 create an optimization task t for the popped triplet and add t to T. The main task is not aware of this new optimization task. Therefore, *revealed* is a new variable that is assigned **false** in line 24. The main task sets *revealed* to **true** when it reveals the existence of the new task while traversing the task list in the loop of either line 18 or 41. The main task has to wait for a permit from each revealed task (unless the test of line 21 is **false**, i.e., the lower bound from the previous computation is equal to or greater than the relevance threshold).

E. ASSIGNING WEIGHTS TO A GRAPH

Our data graphs are generated from XML. Every node has weight 1. The weight of an edge created for an IDREF is 0. All edges e created from elements to subelements or due to IDREFS have weight $f(e)$, where f is a numerical function on edges. For IDREF and IDREFS, *opposite* edges are added automatically if they do not already exist. For an opposite edge $\hat{e}$ that is automatically added, the weight is $o \cdot f(\hat{e})$, where o is a constant parameter. Hence, the weights on a data graph are fully defined by f and o. The pair $\langle f, o\rangle$ is what we call a *weighting scheme*.

Two of the functions f we use are based on the following definition. Consider an edge $e = (v, \hat{v})$. Let $idg^e(\hat{v})$ be the number of edges that enter $\hat{v}$ from nodes having the same label as v. Similarly, $odg^e(v)$ is the number of edges leaving v to nodes with the same label as $\hat{v}$. We define $w_\alpha(e) = \ln(1+\alpha\cdot idg^e(\hat{v})+(1-\alpha)\cdot odg^e(v))$, where $0 \le \alpha \le 1$ is a parameter.

The weighting schemes of the datasets are as follows. For M1: $\langle w_{0.1}, 1.1\rangle$. For M2: $\langle (w_{0.9})^{-1}, 3\rangle$. For both PD1 and FD1: $\langle 1, 1.1\rangle$. For both PD2 and FD2: $\langle w_{0.1}, 3\rangle$. (When we write 1 for f, we mean that f is the constant function 1.)

When given a query Q, we attach to the data graph a node for each keyword q of Q. For each node v that contains a keyword q of Q (but is not the node for q), we add an edge from v to q. The weight of (v, q) in all the datasets is $w = \ln(e + tf(q, v)^{-1}) \cdot ct(Q, v)^{-1}$, where e is Euler's constant, $tf(q, v)$ is the term frequency of q in v, and $ct(Q, v)$ is the number of keywords of Q that appear in v.

F. KEYWORD SELECTIVITY

In Table 2, the *selectivity* of a keyword q is defined as the number of nodes that contain q. The numbers shown in the table are the minimum, maximum and average for each group of four queries with the same number of keywords, except for the last line that refers to all the queries.

G. SERIAL-FREEZING EXPERIMENT

Table 3 gives the experimental results comparing the original Lawler-Murty's procedure of Figure 1 with the serial algorithm that uses freezing (abbr. SF), which is shown in Figure 2. The results are for 10 and 100 solutions. For each combination of a dataset, query size and number of solutions, the running time is given as two numbers. The first is the absolute time in seconds. The second number (shown inside parentheses) is the running time of SF as a percentage of Lawler-Murty's procedure.

Table 3: Execution times for serial with freezing (in seconds and percentage of LM).

Data	Size	No. of Solutions	
		10	100
M1	short	0.03 (42%)	0.26 (30%)
	medium	0.13 (60%)	0.44 (43%)
	long	0.4 (51%)	1.36 (29%)
M2	short	0.02 (37%)	0.18 (18%)
	medium	0.11 (56%)	0.39 (30%)
	long	0.38 (59%)	1.61 (48%)
PD1	short	0.8 (61%)	3.63 (45%)
	medium	10.51 (65%)	35.54 (60%)
	long	15.61 (61%)	44.09 (46%)
PD2	short	1.33 (58%)	6.93 (26%)
	medium	16.51 (53%)	70.46 (58%)
	long	29.63 (46%)	82.45 (36%)
FD1	short	9.08 (47%)	43.69 (43%)
	medium	63.01 (50%)	225.67 (46%)
	long	178.99 (46%)	543.94 (32%)
FD2	short	11.04 (25%)	98.5 (27%)
	medium	129.37 (43%)	567.34 (36%)
	long	480.07 (42%)	1736.08 (32%)