Ranking Information in Networks

Tina Eliassi-Rad[1] and Keith Henderson[2]

[1] Rutgers University (tina@eliassi.org)
[2] Lawrence Livermore National Laboratory (keith@llnl.gov)

Abstract. Given a network, we are interested in ranking sets of nodes that score highest on user-specified criteria. For instance in graphs from bibliographic data (e.g. PubMed), we would like to discover sets of authors with expertise in a wide range of disciplines. We present this ranking task as a *Top-K* problem; utilize fixed-memory heuristic search; and present performance of both the serial and distributed search algorithms on synthetic and real-world data sets.

Keywords: Ranking, social networks, heuristic search, distributed search.

1 Introduction

Given a graph, we would like to rank sets of nodes that score highest on user-specified criteria. Examples include (1) finding sets of patents which, if removed, would have the greatest effect on the patent citation network; (2) identifying small sets of IP addresses which taken together account for a significant portion of a typical day's traffic; and (3) detecting sets of countries whose vote agree with a given country (e.g., USA) on a wide range of UN resolutions. We present this ranking task as a *Top-K* problem. We assume the criteria is defined by L real-valued features on nodes. This gives us a matrix F, whose rows represent the N nodes and whose columns represent the L real-valued features. Examples of these node-centric features include degree, betweenness, PageRank, etc. We define an *L-tuple* $\langle v_1, \ldots, v_L \rangle$ as a selection of one value from each column. The score of an L-tuple is equal to the sum of the selected values (i.e., sum of the tuple's components). For a given parameter K, we require an algorithm that efficiently lists the top K L-tuples ordered from best (highest score) to worst. The L-tuple with the highest score corresponds to a set of nodes in which all features take on optimal values.

As described in the next section, we solve the Top-K problem by utilizing SMA* [4], which is a fixed-memory heuristic search algorithm. SMA* requires the specification of an additional parameter M for the maximum allotted memory-size. The choice for M has a dramatic effect on runtime. To solve this inefficiency problem, we introduce a parallelization of SMA* (on distributed-memory machines) that increases the effective memory size and produces super-linear speedups. This allows use to efficiently solve the Top-K ranking problem for large L and K (e.g., $L \in [10^2, 10^3]$ and $K \in [10^6, 10^9]$). Experiments on synthetic and real data illustrate the effectiveness of our solution.

2 A Serial Solution to the Top-K Ranking Problem

Given F, a matrix of L columns of real-valued features over N rows of vertices, and an integer K, our application of SMA* will report the top K scoring L-tuples that can be selected from F. First, we take each column of matrix F and represent it as its own vector. This produces a list X containing L vectors (of size $N \times 1$). We sort each vector in X in decreasing order. Then, we begin constructing D, another list of L vectors of real numbers. Each vector D_i has $|X_i| - 1$ entries, where $D_{ij} = X_{ij} - X_{i(j+1)}$. Note that the values in each D_i are all nonnegative real numbers; and D_i is unsorted.

The top-scoring L-tuple, R_1, can immediately be reported; it is simply the tuple generated by selecting the first element in each X_i. At this point, the problem can be divided into two subproblems whose structure is identical to the original problem (i.e. another instance of the Top-K problem). While there are several possible ways to make this division, we choose one that allows us to search efficiently. In particular, we choose the vector which would incur the least cost when its best element is discarded.[3] This can be determined by choosing the vector i with the minimum D_{i1}. Given this index i, we generate two subproblems as follows. In the first subproblem, we discard the best element in vector X_i. The resulting list of vectors will be called X^1. In the second subproblem, we keep the best element in vector X_i and remove all other elements from vector X_i. The resulting list of vectors here will be called X^0.

Let's illustrate this procedure with an example. Suppose X = {[10,8,5,2,1], [4,3,2,1], [30,25,24,23,22]}; so, D = {[2,3,3,1], [1,1,1], [5,1,1,1]}. (Recall that we assume lists in X are already sorted in decreasing order and some entries maybe missing.) The top-scoring L-tuple is $R_1 = \langle 10, 4, 30 \rangle$ with $score(R_1)$=10+4+30=44. Starting from X, the next best tuple can be generated by selecting the list i with the smallest D_{i1} and decrementing X_i in X: X^1 = {[10,8,5,2,1], [3,2,1], [30,25,24,23,22]}, D^1 = {[2,3,3,1], [1,1], [5,1,1,1]}, and $score(X^1)$=10+3+30=43. At this point, we can split the problem into two smaller problems. We can either "accept" the best decrement (X^1 above) or "reject" the best decrement and all future chances to decrement that list: X^0 = {[10,8,5,2,1], [4], [30,25,24,23,22]}, D^0 = {[2,3,3,1], [], [5,1,1,1]}, and $score(X^0)$=10+4+30=44. In this way, we can generate X^{11}, X^{10}, X^{01}, etc. This defines a binary tree structure in which each node has two children. Each series of superscripts defines a unique (L-tuple) node.

Our procedure for generating subproblems has three important properties. First, every L-tuple generated from X is either R_1, from X^1, or from X^0. Second, no L-tuple generated from X^1 or X^0 has a score greater than $score(R_1)$. Third, the top-scoring L-tuple from X^0 has the same score as R_1.

Given this formulation, a recursive solution to the Top-K problem is theoretically possible. In the base case, the input list X has L vectors of length one, and there is only one possible L-tuple to report (namely, R_1). Given arbitrary length vectors in X, we can divide the problem as above and merge the resultant top-K lists with R_1, discarding all but the top K elements of the merged list. This method, however, is impractical since each of the lists returned by the recursive calls could contain as many as K elements; hence, violating the requirement that space complexity must *not* be

[3] If all vectors in X contain exactly one element, no subproblems can be generated. In this case, the Top-K problem is trivial.

$O(K)$. But, a search-based approach allows us to generate the top K tuples one at a time. If we treat each Top-K instance as a node, and each subproblem generated from an instance as a child of that node, then we can treat the Top-K problem as search in a binary tree. The cost of traversing an edge is equal to the loss in score incurred by removing elements from X; thus the X^0 edge always has cost 0 and the X^1 edge has cost equal to $bestDiff(X) =_{def} min(D_{i1}|i = 1 \cdots L)$. In this context, A* search [3] clearly generates subproblems in order of their R_i scores. For the heuristic function $h(n)$, we use $bestDiff(X)$, which can be readily computed. Note that $bestDiff$ is monotone (and thus admissible) by the following argument:[4] If p is a 1-child of n (the X^1 subproblem), then $h(n) = cost(n, p)$ and $h(p) \geq 0$. Otherwise, $cost(n, p) = 0$ and $h(n) \leq h(p)$ by the definition of $bestDiff$.

Unfortunately, A* search requires storage of the $OPEN$ list in memory, and the size of $OPEN$ increases with every node expansion. This violates our memory requirements (of less than $O(K)$ storage), so we employ SMA* search [5] which stores a maximum of M nodes in memory at any given time during the execution. SMA* expands nodes in the same order as A* until it runs out of memory. At that point, the least promising node is deleted and its f-value is backed up in its parent.[5] SMA* is guaranteed to generate the same nodes as A* and in the same order. However, it may generate some intermediate nodes multiple times as it "forgets" and "remembers" portions of the tree. In certain cases, especially when the parameter M is small compared to the size of the search fringe, SMA* can experience "thrashing." This thrashing results from large parts of the search tree being generated and forgotten with very few new nodes being discovered. Our serial Top-K algorithm is essentially the same as SMA* [5] except for the use of heaps in selecting which of the X_i vectors to decrement at each node.

3 A Parallel Solution to the Top-K Ranking Problem

The aforementioned serial algorithm is very sensitive to the choice of M, the maximum amount of memory that can be allocated (see the Experiments Section). Here we present a parallel SMA* algorithm that offers dramatic improvement in runtime.

For a machine with P processing nodes, we use A* search to generate the P–1 best candidate subproblems. This covers the entire search tree. Because each subproblem is independent of the others, each processing node can perform SMA* search on its subproblem and incrementally report results to the master processing node, where the incoming tuple lists are merged and results are reported.

For small P, this algorithm works as expected and produces super-linear speedup. However as P increases, the performance boost can decrease quickly. This occurs because the runtime for this parallel algorithm is dominated by the single processing node with the longest runtime. If the initial allocation of subproblems to processing nodes is imbalanced, additional processing nodes may not improve performance at all. To ameliorate this problem, we adopt a load-balancing heuristic.

[4] Recall that a heuristic function $h(n)$ is monotone if $h(n) \leq cost(n, p) + h(p)$ for all nodes n and all successors p of n.

[5] The function f represents the total cost function, which equals the sum of the cost encountered so far and the estimated cost.

We run parallel SMA* on the input data with $K' \ll K$ as the threshold parameter. We then use the relative load from each subproblem as an estimate of the total work that will have to be done to solve that subproblem. We use these estimates to redistribute the initial nodes and repeat until there are no changes in the initial allocation of nodes. This distribution is then used to generate the top K L-tuples as described above. In our experiments, this heuristic correctly balanced the loads on the processing nodes. Moreover, the initial overhead to calculate the estimates was a negligible fraction of the overall runtime.

4 Experiments

4.1 Synthetic Data: Results and Discussion

To determine the runtime requirements for our Top-K algorithm, we generated a synthetic data set with $L = 100$ lists. Each list has between one and ten real values distributed uniformly in $[0, 1]$. Figure 1 shows runtime results for $M = 10^6$ nodes in memory. Note that as K increases, time complexity in K becomes near-linear. However, at approximately 20,000 tuples per second it is still too slow to be practical for large values of K.

Figures 2(a), 2(b), and 2(c) show the performance metrics for a strong scaling experiment with the parallel algorithm: *runtime*, *speedup*, *efficiency*. Runtime is the wall-clock runtime. Speedup is defined as the time taken to execute on one process (T_1) divided by the time taken to execute on P processes (T_p). Linear speedup is the ideal case where 2 processes take half the time, 3 processes take a third of the time, and so forth. Efficiency is defined as speedup divided by the number of processors. Ideal efficiency is always 1, and anything above 1 is super-linear. Values less than 1 indicate diminishing returns as more processors are added.

For the parallel Top-K experiments, we use the same 100 lists described above but with $M = 10^5$, $K = 2{\cdot}10^6$, and $K' = 10^4$. There is no I/O time in these experiments, as tuples are simply discarded once they are discovered. Note that the runtime for $P = 2$ is the same as the serial runtime plus overhead because in this case one processing node is the master and simply "merges" the results from the compute node. We see super-linear speedup for values of P up to 17 processes. However, the runtime for $P = 17$ is nearly identical to the runtime for $P = 9$. This is because in the $P = 17$ experiment, one of the large subproblems is not correctly split and redistributed. Figure 2(d) reveals the cause. When $P > 9$, the subproblems are not being correctly distributed among the nodes. Figure 3(d) shows the same results for the load-balanced version of the algorithm. In this case, it is clear that the nodes are being correctly distributed.

Figures 3(a), 3(b), and 3(c) show the parallel efficiency results for the load-balanced case. Our load-balancing heuristic drastically improves performance as the number of processors increases. These results include the time required to calculate the load-balance estimates, which explains the slightly longer runtimes at $P \leq 9$.

4.2 Real Data: Results and Discussion

We tested our Top-K algorithm on four real-world data sets – namely, patent citations, DBLP bibliographic data, IP traffic, and UN voting. For brevity, we only discuss two

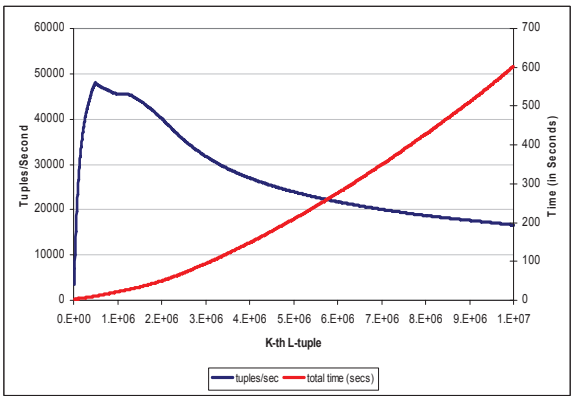
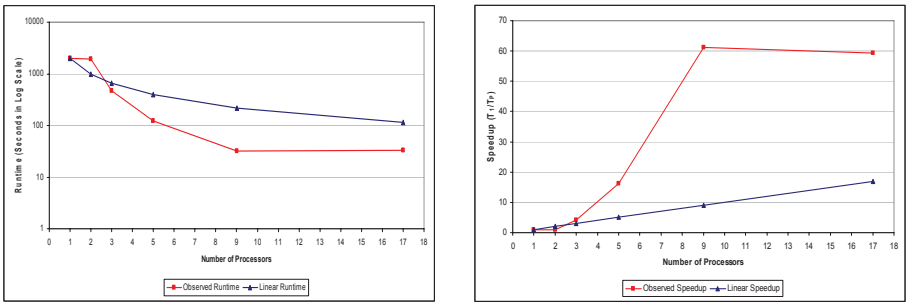


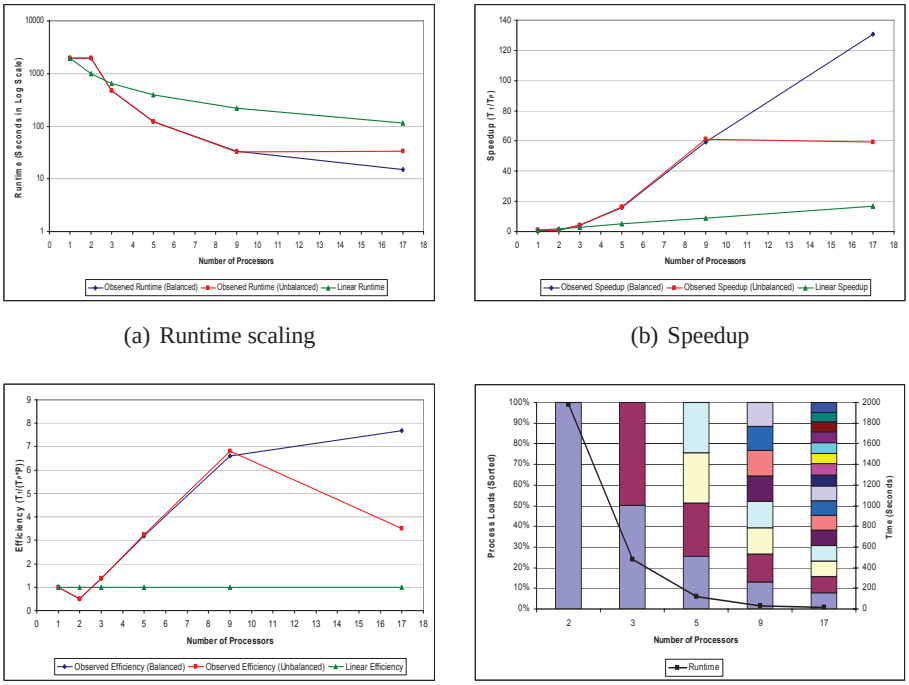
Fig. 1. Serial top-K: Runtime as a function of K



(a) Runtime scaling (b) Speedup

(c) Efficiency (d) Load balance

Fig. 2. Unbalanced parallel top-K: $K = 2 \cdot 10^6$, $L = 100$ lists, $M = 10^5$ nodes



(a) Runtime scaling (b) Speedup

(c) Efficiency (d) Load balance

Fig. 3. Balanced parallel top-K: $K = 2 \cdot 10^6$, $L = 100$ lists, $M = 10^5$ nodes

of them here. First, our patents data set is a collection of 23,990 patents from the U.S. Patent Database (subcategory 33: biomedical drugs and medicine). The goal here is to discover sets of patents which, if removed, would have the greatest effect on the patent citation network. In particular, we are interested in patents which are frequently members of such sets; and we want to find which centrality measures correspond to these frequent nodes in the citation network. We generated a citation network with 23,990 nodes and 67,484 links; then calculated four centrality metrics on each node: degree, betweenness centrality, random walk with restart score (RWR), and PageRank. In the Top-K framework, there are 4 vectors: one for each centrality measure. The indices into the vectors are patents. The entry *i* in vector *j* is the (normalized) centrality score *j* for patent *i*. Table 1 lists the patents with the highest frequency in the top-100K tuples. As expected, the patents associated with DNA sequencing have the highest frequencies. Table 2 presents the centrality scores for the patents listed in Table 1. As it can be seen, the centrality scores alone could not have found these highly impacting patents (since no patent dominates all four centrality scores).

Our second data set is composed of 4943 proposed UN resolutions, votes from 85 countries per resolution (yes, no, abstain, etc), and 1574 categories. Each proposed

Patent Number	Frequency (in Top-100K Tuples)	Patent Title
4683195	0.799022	Process for amplifying, detecting, and/or-cloning nucleic acid sequences
4683202	0.713693	Process for amplifying nucleic acid sequences
4168146	0.202558	Immunoassay with test strip having antibodies bound...
4066512	0.175828	Biologically active membrane material
5939252	0.133269	Detachable-element assay device
5874216	0.133239	Indirect label assay device for detecting small molecules ...
5939307	0.103829	Strains of Escherichia coli, methods of preparing the same and use ...
4134792	0.084579	Specific binding assay with an enzyme modulator as a labeling substance
4237224	0.074189	Process for producing biologically functional molecular chimeras
4358535	0.02674	Specific DNA probes in diagnostic microbiology

Table 1. Patents with the highest frequency in the top-100K tuples

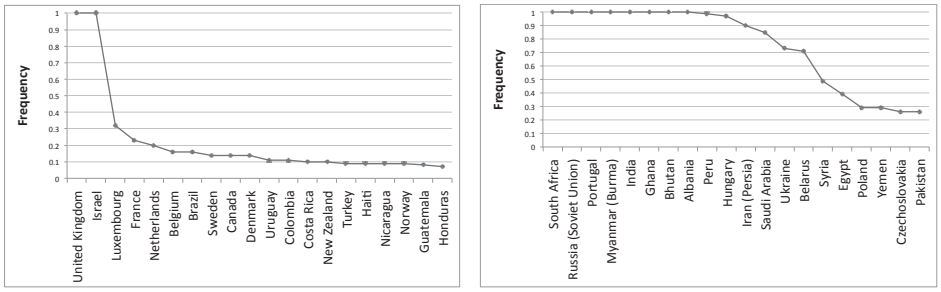
Patent #	Degree	Betweenness	RWR	PageRank
4683195	**1.000**	0.433	0.030	**1.000**
4683202	0.974	0.234	0.000	0.975
4168146	0.099	**1.000**	0.132	0.088
4066512	0.036	0.845	0.185	0.026
5939252	0.221	0.234	**1.000**	0.000
5874216	0.122	0.234	**1.000**	0.000
5939307	0.071	0.234	0.965	0.000
4134792	0.126	0.772	0.040	0.118
4237224	0.341	0.760	0.016	0.332
4358535	0.460	0.670	0.064	0.446

Table 2. Centrality scores for the 10 highest-frequency patents in the top-100K tuples

resolutions is assigned to one or more categories (with an average of 1.6 categories per proposal). We select the most frequent 20 categories. Then, we score each country in each category by how often it agrees (or disagrees) with the USA vote in that category: $\frac{1}{m}$ points per agreement, where m is the number of agreements per proposed resolution. The input to our top-K algorithm then is a matrix F with 85 rows (one per country) and 20 columns (one for each selected category). The entries in this matrix are the weighted agreement scores. We set K (in the top-K) to $100,000$. We repeat the experiment with scores for disagreements. Figures 4(a) and 4(b) depict the ranking results.

5 Related Work

Previous studies of the Top-K problem focus on cases where L is small (2 or 3) and each list is very long. Fredman ([2]) studies the problem of sorting $X + Y = \{x + y | x \in X, y \in Y\}$ in better than $O(n^2 \cdot log(n))$ time (where $|X| = |Y| = n$). Frederickson and Johnson ([1]) examine the slightly easier problem of discovering the K^{th} element of $X + Y$ in $O(n \cdot log(n))$ time. They also consider the problem of finding the K^{th} element of $\sum_{i=1}^{m} X_i$, proving that a polynomial algorithm in n for arbitrary $m \leq n$ and



(a) Top countries agreeing with the USA in the top-100K tuples

(b) Top countries disagreeing with the USA in the top-100K tuples

Fig. 4. UN resolutions: Voting records on the 20 most frequent categories

K would imply that $P = NP$. Our approach uses fixed-memory heuristic search to enumerate L-tuples. It has $O(n \cdot log(n))$ runtime and at most $O(K)$ storage.

6 Conclusions

We demonstrated that ranking sets of nodes in a network can be expressed in terms of the Top-K problem. Moreover, we showed that this problem can be solved efficiently using fixed memory by converting the problem into a binary search tree and applying SMA* search. We also parallelized SMA* for the Top-K problem in order to achieve super-linear speedup in a distributed-memory environment. Lastly we validated our approach through experiments on both synthetic and real data sets.

7 Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344.

References

1. G. N. Frederickson and D. B. Johnson. Generalized selection and ranking. In *Proc. of the 12th STOC*, pages 420–428, 1980.
2. M. L. Fredman. How good is the information theory bound in sorting? *Theoretic. Comput. Sci.*, 1:355–361, 1976.
3. P. E. Hart, N. J. Nilsson, and B. Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, 37:28–29, 1972.
4. S. Russell. Efficient memory-bounded search methods. In *ECAI*, pages 1–5, 1992.
5. S. J. Russell. Efficient memory-bounded search methods. In *Proc of the 10th ECAI*, pages 1–5, 1992.