Solving the Top-K Problem with Fixed-Memory Heuristic Search

Keith Henderson and Tina Eliassi-Rad
Lawrence Livermore National Laboratory*
Box 808, L-560, Livermore, CA 94551 USA
{keith,eliassi}@llnl.gov

Abstract

The *Top-K* problem is defined as follows. Given L lists of real numbers, find the top K scoring L-tuples. A tuple is scored by the sum of its components. Rare event modeling and event ranking are often reduced to the Top-K problem.

In this paper, we present the application of a fixed-memory heuristic search algorithm (namely, SMA*) and its distributed-memory extension to the Top-K problem. Our approach has efficient runtime complexity and super-linear speedup in distributed-memory setting. Experimental studies on both synthetic and real-world data sets show the effectiveness of our approach.

Introduction

Given a set of events with real-valued features, an important task is to model rare events and in general to rank events. Examples include (1) discover committees of authors which deliver expertise in a wide range of disciplines; (2) discover sets of patents which, if removed, would have the greatest effect on the patent citation network; and (3) find small sets of IP addresses which, when taken together, account for a significant portion of a typical day's traffic.

In this paper, we present the general task of event ranking in terms of the Top-K problem; and describe how a fixed-memory heuristic search algorithm – namely, SMA* (Russell 1992) – and its distributed-memory extension effectively solve this problem.

Top-K Problem Definition

The Top-K problem is defined as follows. Given L lists of real numbers, possibly of different lengths, an *L-tuple* is a selection of one value from each list. 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. We are particularly interested in input scenarios where both L and K are large (say, $L \in [10^2, 10^3]$ and $K \in [10^6, 10^9]$).

Given the above Top-K definition, the task of event ranking reduces to finding the top K L-tuples of a set of events, where each event is described by a vector of real-valued features. The L-tuple with the highest score corresponds to a theoretical event in which all features take on optimal values. The top K L-tuples represent a set of theoretical events, whose scores can be compared to observed data to detect extreme (e.g. rare) events.

Contributions

Due to the potentially huge number of results, any acceptable algorithm will not be able to store all of the generated L-tuples in memory. Therefore, a fixed (i.e., constant) memory-size algorithm, such as SMA* (Russell 1992), is needed.

SMA* and related algorithms require the specification of an additional parameter M for the maximum allotted memory-size. The choice for M has a dramatic effect on runtime (see the Experiments Section). However, parallelization of SMA* increases the effective memory size and produces super-linear speedups.

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 (1976) 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 (1980) examine the slightly easier problem of discovering the K^{th} element of $X + Y$ in $O(n \cdot log(n))$ time. They also considers 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 K would imply that $P = NP$.

The Top-K problem is related to the problem of determining order statistics for a collection of numbers. A well-known linear-time algorithm for computing the i^{th} order statistic of a list of numbers is presented in Blum et al. (1973) and Floyd and Rivest (1973).

Our approach uses fixed-memory heuristic search to enumerate L-tuples. In particular, we utilize and extend SMA* (Russell 1992). We found SMA* to be the best-suited fixed-memory heuristic search algorithm for the Top-K problem. SMA* is a simplification of MA* search (Chakrabarti et al. 1989), and is similar to RA* (Evett et al. 1990). Another member of this family of algorithms is IE (Russell 1992), but IE often leads to a higher re-expansion rate than SMA*.

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-TR-410187.

MREC (Sen and Bagchi 1989) and ITS (Ghosh, Mahanti, and Nau 1994) are extensions of IDA* (Korf 1985). They take advantage of extra available memory, but like IDA* they must search from the root node at each iteration. ITS is essentially another simplified version of MA* (Ghosh, Mahanti, and Nau 1994), while MREC can make poor use of available memory due to inefficiencies in its allocation procedure.

Serial SMA* for the Top-K Problem

Given X, a list of L lists of real numbers, and an integer K, our application of SMA* will report the top K scoring L-tuples that can be selected from X. We assume that each list in X is sorted in decreasing order; then, begin by constructing D, another list of L lists of real numbers. Each list 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 will allow us to search efficiently. In particular, we choose the list which would incur the least cost when its best element is discarded.[1] This can be determined by choosing the list 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 list X_i. The resulting list of lists will be called X^1. In the second subproblem, we keep the best element in list X_i and remove all other elements from list X_i. The resulting list of lists 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). 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.

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.

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

Given the above formulation, a recursive solution to the Top-K problem is theoretically possible. In the base case, the input list X has L lists of length one, and there is only one possible L-tuple to report (namely, R_1). Given arbitrary length lists 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 $O(K)$. 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 (Hart, Nilsson, and Raphael 1972) 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:[2] 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 (Russell 1992) 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.[3] 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.

The serial Top-K algorithm is described in Algorithm 1, which is essentially the same as SMA* (Russell 1992) except for modifications to demonstrate the use of heaps in selecting which of the X_i lists to decrement at each node. Algorithms 2 through 5 describe Top-K's auxiliary routines.[4]

[2] 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.

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

[4] We omit descriptions of auxiliary routines $backup()$ and $completed()$. See Russell (1992) for details.

Algorithm 1 Top-K

Require: integers K, L, M
Require: list of lists of positive reals D

```
count ← 0
open ← makeQueue()
root ← makeRoot()
queueInsert(open, root)
report(root)
while OPEN is not empty and count < K do
  if length(open) == M then
    worst ← highest f-value leaf node
    queueRemove(open, worst)
    delete(worst)
  end if
  best ← lowest f-value node in OPEN
  next ← successor(best)
  queueInsert(open, next)
  if completed(best) then
    backup(best)
    queueRemove(open, best)
  end if
  if next's top tuple has not been seen then
    report(next)
  end if
end while
```

Algorithm 2 makeRoot()

```
v ← node()
v.heap ← makeHeap()
for i = 1 to L do
  heapInsert(v.heap, {key = D[i][0], val = (i, 0)})
end for
v.g ← 0
v.h ← heapMin(v.heap).key
v.f ← v.g + v.h
v.leftChild ← v.rightChild ← NULL
```

Algorithm 3 successor(v)

```
if v has a right child then
  return left(v)
end if
if v has a left child then
  return right(v)
end if
if v has generated children but they have been deleted
then
  return least-recently generated child
end if
return left(v)
```

Algorithm 4 left(v)

```
u ← node()
u.heap ← heapCopy(v.heap)
{diff, (i, j)} ← heapPop(u.heap)
heapPush(u.heap, {key = D[i][j + 1], val = (i, j +
1)})
u.g ← v.g + diff
u.h ← heapMin(u.heap).key
u.f ← u.g + u.h
u.leftChild ← u.rightChild ← NULL
v.leftChild ← u
return u
```

Algorithm 5 right(v)

```
u ← node()
u.heap ← heapCopy(v.heap)
heapPop(u.heap) /* Discard list D_i in this branch. */
u.g ← v.g
u.h ← heapMin(u.heap).key
u.f ← u.g + u.h
u.leftChild ← u.rightChild ← NULL
v.rightChild ← u
return u
```

Parallel-SMA* for the Top-K 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.

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.

Experiments

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, 3, and 4 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 5 reveals the cause. When $P > 9$, the subproblems are not being correctly distributed among the nodes. Figure 6 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 7, 8, and 9 show the parallel efficiency results for the load-balanced case. As previously discussed, the 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 $P \leq 9$.

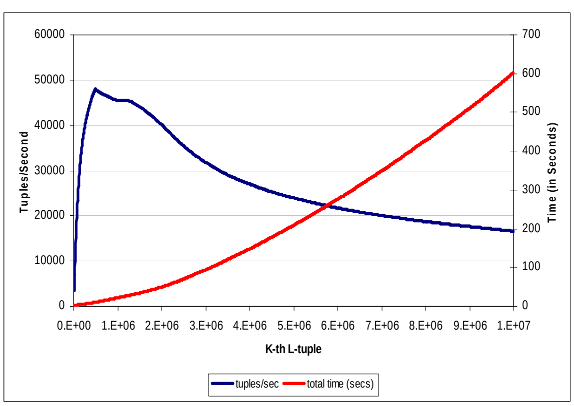

Figure 1: Serial Top-K: Runtime as a Function of K

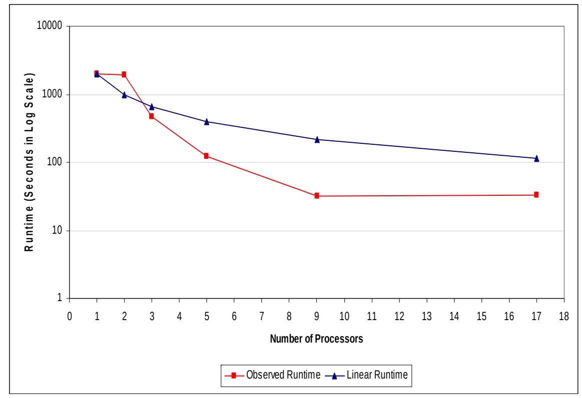

Figure 2: Unbalanced Parallel Top-K: Runtime Scaling

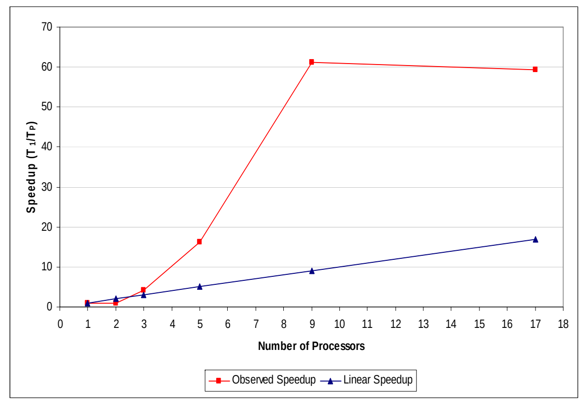

Figure 3: Unbalanced Parallel Top-K: Speedup

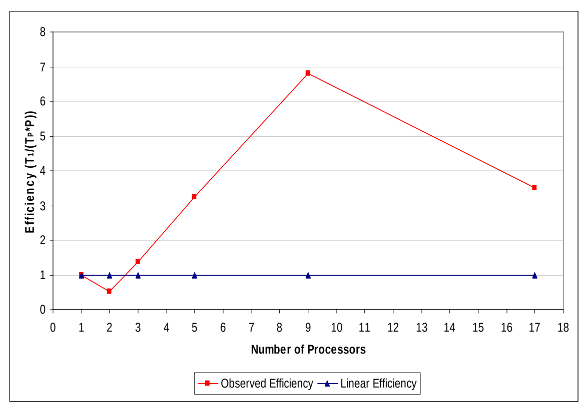

Figure 4: Unbalanced Parallel Top-K: Efficiency

Real Data: Results and Discussion

We tested the Top-K algorithm on four real-world data sets (namely, patents, IP traffic, DBLP, and UN voting). For brevity, we only discuss two of them here. The first data set is a collection of 23,990 patents from the U.S. Patent

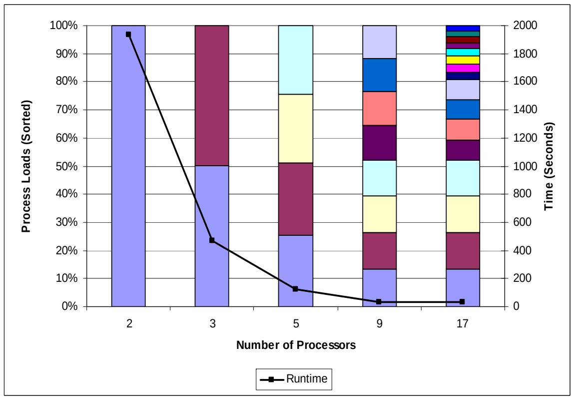

Figure 5: Unbalanced Parallel Top-K: Load Balance

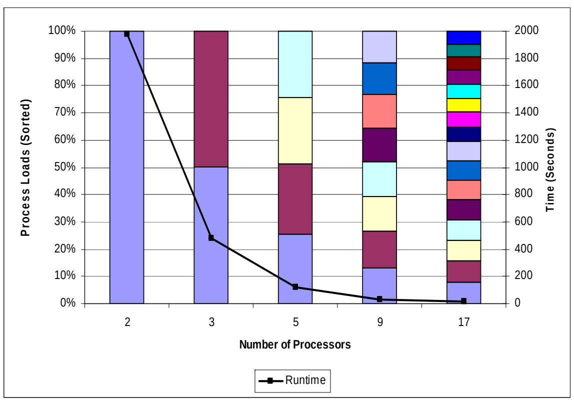

Figure 6: Balanced Parallel Top-K: Load Balance

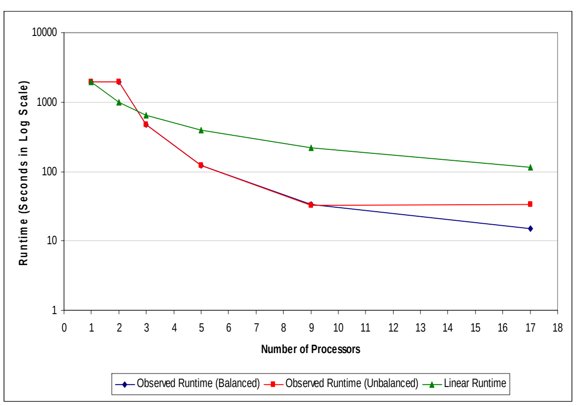

Figure 7: Balanced Parallel Top-K: Runtime Scaling

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

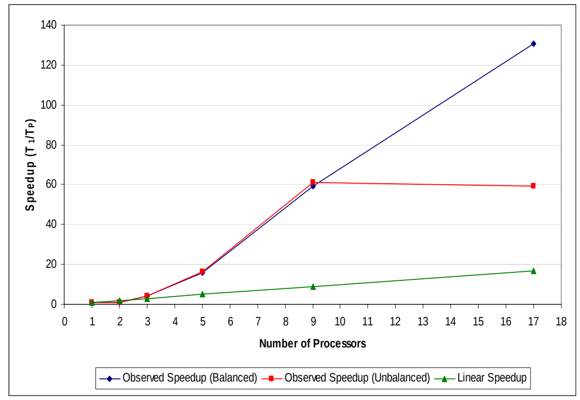

Figure 8: Balanced Parallel Top-K: Speedup

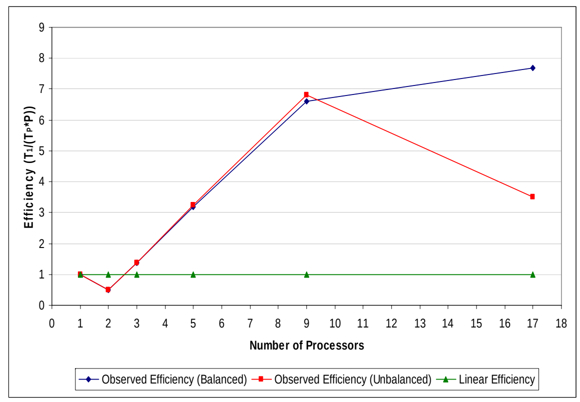

Figure 9: Balanced Parallel Top-K: Efficiency

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 lists: one for each centrality measure. The indices into the lists are patents. The entry i in list 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.

The second data set is a trace of IP network traffic captured at a 5-day meeting. Here, the goal is to find recurring daily events in which no particular IP is dominating the communication bandwidth. There are a total of 34,787 IP-addresses with approximately 16M communications between them. In the Top-K framework, there are 24 lists: one per hour of the day. The indices into the lists are IP addresses. The entry i in list j is the (normalized) maximum number of bytes sent by IP i during hour j over all 5 days. Figure 10 depicts the number of unique IPs in the top-100K

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	Betweeness	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

tuples, the number of IPs communicating, and the total number of bytes sent. Only the Top-K tuples discover the hours that are least stable w.r.t. IP communications. In order, these are 10 AM, 3 PM, and 8 AM.

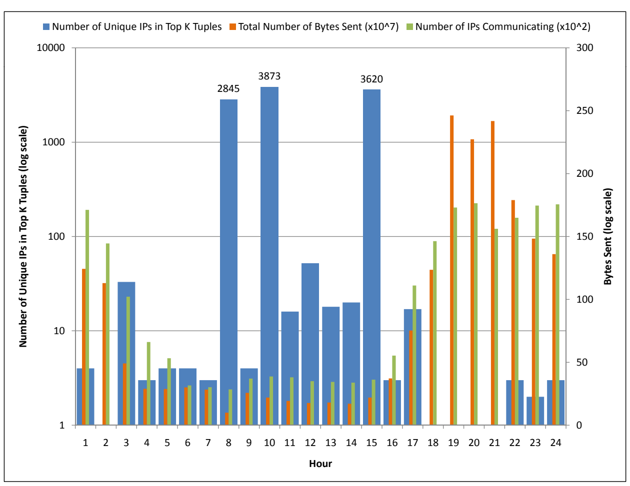

Figure 10: IP Network Traffic Data: 10 AM, 3 PM, and 8 AM are the least stable hours w.r.t. IP communications. Other measures cannot find these unstable hours.

Conclusions

In this paper, we demonstrated that the Top-K 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.

References

Blum, M.; Floyd, R. W.; Pratt, V.; Rivest, R. L.; and Tarjan, R. E. 1973. Time bounds for selection. Technical Report CS-TR-73-349, Stanford University, Stanford, CA.

Chakrabarti, P. P.; Ghose, S.; Acharya, A.; and de Sarkar, S. C. 1989. Heuristic search in restricted memory. *AIJ* 41(2):197–222.

Evett, M.; Hendler, J.; Mahanti, A.; and Nau, D. 1990. PRA*: A memory-limited heuristic search procedure for the connection machine. In *Proc. of the 3rd Symp. on the Frontiers of Massively Parallel Computation*.

Floyd, R. W., and Rivest, R. L. 1973. Expected time bounds for selection. Technical Report CS-TR-73-349, Stanford University, Stanford, CA.

Frederickson, G. N., and Johnson, D. B. 1980. Generalized selection and ranking. In *STOC*, 420–428.

Fredman, M. L. 1976. How good is the information theory bound in sorting? *Theoretic. Comput. Sci.* 1:355–361.

Ghosh, S.; Mahanti, A.; and Nau, D. S. 1994. ITS: an efficient limited-memory heuristic tree search algorithm. In *AAAI*, 1353–1358.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1972. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.* 37:28–29.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *AIJ* 27(1):97–109.

Russell, S. J. 1992. Efficient memory-bounded search methods. In *ECAI*, 1–5.

Sen, A. K., and Bagchi, A. 1989. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, 297–302.