7-2013

# Shortlisting Top-K Assignments

Yimin LIN
*Singapore Management University*, yimin.lin.2007@smu.edu.sg

Kyriakos MOURATIDIS
*Singapore Management University*, kyriakos@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Databases and Information Systems Commons, and the Numerical Analysis and Scientific Computing Commons

# Shortlisting Top-K Assignments

Yimin Lin
School of Information Systems
Singapore Management University
yimin.lin.2007@smu.edu.sg

Kyriakos Mouratidis
School of Information Systems
Singapore Management University
kyriakos@smu.edu.sg

## ABSTRACT

In this paper we identify a novel query type, the *top-K assignment query* ($\alpha$Top-$K$). Consider a set of objects $P$ and a set of suppliers $S$, where each object $p_i \in P$ must be assigned to one supplier $s_j \in S$. Assume that there is a cost $c_{ij}$ associated with every object-supplier pair $\langle p_i, s_j \rangle$. The matching with the smallest total cost would assign each object $p_i$ to the supplier $s_j$ with the minimum $c_{ij}$ value. In many scenarios, however, runner-up assignments may be required too, like for example when a decision maker needs to make additional considerations, not captured by $c_{ij}$ values. In this case, it is necessary to examine several shortlisted assignments before choosing one. This motivates the $\alpha$Top-$K$ query, which computes the $K$ best assignments, i.e., those achieving the $K$ smallest total costs. Algorithms for the traditional *assignment ranking* problem could be adapted to process the query, but their time requirements are prohibitive for large datasets (cubic to the input size). In this work we exploit the specific properties of the $\alpha$Top-$K$ problem and develop scalable methods for its processing. We also consider its *incremental* version, where $K$ is not specified in advance; instead, the best assignments are iteratively computed on demand. An empirical evaluation with real data verifies the practicality and efficiency of our framework.

## 1. INTRODUCTION

Consider the scenario of building a Boeing 747-400 aircraft. This plane includes 6 million parts and Boeing uses around 6 thousand different suppliers to source them [3]. Typically, every part can be provided by multiple alternative suppliers, albeit at different price. The smallest-cost supply plan $A$ would order each part $p_i$ from the supplier $s_j$ that offers it at the lowest price $c_{ij}$. In many cases, however, runner-up (suboptimal) assignments may be useful [5]. For instance, the smallest-cost supply plan $A$ might not be ideal for Boeing, because cost values $c_{ij}$ are unable to capture all factors that affect the final decision; e.g., instead of $A$, it might make sense to adopt another plan $A'$ with a slightly higher overall cost, which however assigns enough parts to a specific supplier in order to qualify for a service upgrade (e.g., enhanced after-sales support, delivery priority, etc). Such decision factors cannot be incorporated into individual $c_{ij}$ values (not accurately at least). Decision

making in this scenario necessitates shortlisting a number of plans based on total cost, with the purpose of investigating them closer. That motivates the $\alpha$Top-$K$ query. Given a set of objects $P$ (e.g., aircraft parts), a set of suppliers $S$, the object-supplier costs $c_{ij}$, and a weighted function $f$ that defines the overall cost of an assignment, $\alpha$Top-$K$ computes the $K$ assignments with the smallest total costs according to $f$. $K$ is a natural number, specified by the decision maker, that determines the number of alternative matchings[1] to be reported.

A variant of the problem with practical relevance is *incremental* $\alpha$Top-$K$, where $K$ is not known in advance, and the decision maker (or a semi-automated decision support process) iteratively requests for the next best assignment until a satisfactory one is found. This is particularly useful when certain constraints cannot be incorporated into traditional problem definitions, requiring iterative examination of alternatives in increasing cost order [23].

Our problem is related to *top-K computation*. A top-$K$ query receives as input a dataset $P$ and a scoring function $f$ [17]. The query returns the $K$ objects with the highest (or the smallest) scores according to $f$. To cope with large datasets, indexes have been used to accelerate search [26, 31]. An idea to solve $\alpha$Top-$K$ would be to somehow index the possible assignments, and retrieve the best $K$ using an existing top-$K$ algorithm. However, the number of candidate matchings explodes with the problem size, i.e., the assignments cannot be enumerated within reasonable time even for small datasets. To exemplify, assume that $|S| = 10$ suppliers and $|P| = 100$ objects. The number of possible assignments is $10^{100}$.

*Assignment ranking* [8, 5] is probably closer related to $\alpha$Top-$K$. The input includes an object set and a supplier set, where each object must be assigned to a supplier and every supplier can be assigned no more than a number of objects. The goal is to find the $K$ matchings with the smallest sum of costs across their object-supplier pairs. Assignment ranking algorithms could be adapted to $\alpha$Top-$K$. However, even the most efficient among them (termed CH [8]) has a time complexity of $O(K \cdot |P|^3)$ and needs $O(|P| \cdot |S|)$ space. These requirements are excessive even for medium-size problems, calling for novel methods specific to $\alpha$Top-$K$.

In this paper we develop algorithms for $\alpha$Top-$K$ queries in large datasets. Their crux is a visualization/examination of the search space via an *assignment tree*. The first algorithm, termed *Modified CH* (MCH), follows the CH paradigm but overcomes its inefficiencies using the assignment tree idea. The second dismisses the CH

---

[1] Terms assignment and matching are used interchangeably.

approach and further exploits the problem characteristics, leading to a new (and twice as fast) algorithm, named *Best Neighbor* (BN).

Our third algorithm, *Cross Probe* (CP), extends the applicability of our work in a number of directions. First, in centralized environments, it provides a tunable tradeoff between time and space requirements, thus catering for constrained memory scenarios. Second, CP also lays the foundation for *parallel* and *distributed processing*, a feature desirable due to the increasing availability of multi-core processors.

The rest of the paper is structured as follows. Section 2 reviews related work. Section 3 formalizes the problem. Section 4 presents the assignment tree idea and a preliminary solution. Sections 5, 6 and 7 describe MCH, BN and CP, respectively. Section 8 applies our framework to incremental $\alpha$Top-$K$. Section 9 discusses special cases and extensions. Section 10 evaluates our techniques on real and synthetic data. Finally, Section 11 concludes the paper.

## 2. RELATED WORK
In this section we survey related work on top-$K$ processing, the traditional optimal assignment and assignment ranking problems, as well as matching algorithms for large databases.

### 2.1 Top-$K$ Query Processing
The standard top-$K$ query retrieves the $K$ top-scoring objects in a dataset $P$ according to a scoring function $f$ over the data attributes. Top-$K$ queries have been studied in various domains, including relational databases [17], multimedia [7], web-accessible and distributed databases [11, 19], etc. There exist several approaches to reduce processing time, such as pre-computation [6, 15], histograms [4] and indexing [26, 31]. As explained in Introduction, top-$K$ methods are unable to process $\alpha$Top-$K$ queries.

The *rank join* problem is a case of top-$K$ processing inside a join operator. The input includes $m$ relations $L_i$ (for $1 \le i \le m$), each sorted on one of its attributes in descending order. Given a join condition and a scoring function $f$ defined over the sorting attributes, the rank join operator reports the $K$ join results with the highest scores. There are several algorithms for this problem [22, 27, 16, 25]. Their objective is to reduce the accessed portion of each input relation, and thus the I/O cost. Theoretically, the $\alpha$Top-$K$ problem can be formulated as a special rank join case where the join condition is always true, each sorted relation corresponds to an object $p_i$ (containing suppliers in ascending $c_{ij}$ order), and $f$ is a weighted sum function (which here we aim to minimize). However, rank join algorithms are impractical in our setting because they (i) assume joining a small number of relations (whereas we consider object populations in the order of thousands or millions), (ii) are designed for actual joins (not-always true condition) and (iii) cannot cope with the combinatorial explosion of candidate matchings when (i) and (ii) do not hold. Also, their goal is to reduce access cost (i.e., limit the read portion of each input relation – not an issue in our problem, as will become clear later) instead of minimizing the number of considered matchings (which is the main determinant of performance in $\alpha$Top-$K$ processing).

In Section 10.1 we include small-scale experiments with representative rank join algorithms HRJN* [16] and J* [22]. HRJN* is chosen because of its instance optimality for the rank join problem (instance optimality refers to the search depth within the input relations). The idea in HRJN* is to iteratively choose an input relation

to draw the next non-accessed tuple from, and join it with all accessed tuples from the other relations. The algorithm terminates when a threshold is reached. The J* algorithm is chosen because it follows a different paradigm from most rank join techniques. It treats the problem as a search in the Cartesian product of the input relations, and adopts an A* approach to guide the search [24]. Our experiments verify the ineffectiveness of rank join algorithms when adapted to our problem.

### 2.2 Optimal Assignment and Ranking
Our work is related to the traditional *optimal assignment* problem [1]. The input includes a set of objects $P$ and a set of suppliers $S$. Each object needs to be assigned to one supplier, while a supplier $s_j \in S$ may be assigned up to $n_j$ objects (called the *capacity* of $s_j$). Every object-supplier pair incurs a cost $c_{ij}$ if chosen in the final matching. The goal is to compute the assignment that has the smallest total cost (i.e., the smallest sum of costs across assigned object-supplier pairs). There are several optimal assignment algorithms (e.g., [13, 20]) among which the most efficient and commonly used is the *Successive Shortest Path Algorithm* (SSPA) [9] with $O(|P|^3)$ time and $O(|P| \cdot |S|)$ space requirements.

Optimal matching lies in the heart of *assignment ranking*. Here the input additionally includes an integer $K$, and the output consists of the matchings with the $K$ smallest total costs. The state-of-the-art methods for assignment ranking are Murty's algorithm [21] and Chegireddy and Hamacher's technique (CH) [8]. They are based on a similar principle, and although an improved implementation of Murthy's algorithm [23] achieves the same asymptotic complexity as CH, the latter is more efficient in practice [5].

CH builds a branch-decision tree, where every node is either a leaf or the parent of two other nodes. Each node computes and stores the two best assignments subject to specific constraints. The constraints are imposed by sets of object-supplier pairs $I$ and $O$ associated with the node. Set $I$ indicates pairs that must belong to the two assignments, and $O$ includes pairs that cannot participate in them. The root contains the optimal and the second best matchings, $A_1$ and $A_2$, computed by two executions of SSPA[2]. Let $e$ be a pair that is chosen in (i.e., belongs to) $A_1$ but not in $A_2$. The left child of the root has $I = \{e\}$ and $O = \emptyset$, while the right has $I = \emptyset$ and $O = \{e\}$. The top-2 assignments are computed by SSPA for each of the children of the root, subject to their own constraints. In the general case, given a node with constraints $I$ and $O$ whose top-2 matchings are $A$ and $A'$, and where $e$ is a pair that belongs only to $A$ but not to $A'$, the left child has $I_l = I \cup \{e\}$ and $O_l = O$, and the right $I_r = I$ and $O_r = O \cup \{e\}$. CH terminates after $K$ iterations, in each of which it splits the leaf node that has the $A'$ assignment with the smallest cost; the specific $A'$ is reported as the next best matching. An exception is the root, where both assignments $A$ and $A'$ are directly reported as the overall top-2 matchings $A_1$ and $A_2$.

$\alpha$Top-$K$ can be treated as an assignment ranking problem without capacity constraints (i.e., $n_j = |P|$ for each $s_j \in S$). Hence, theoretically, CH could be employed for its processing. However, the time complexity of CH is $O(K \cdot |P|^3)$ due to its repetitive SSPA calls, which explodes for moderate or large-size object sets. Also, CH requires an excessive $O(|P| \cdot |S|)$ space. In Section 5 we develop MCH that replaces the SSPA building block of CH with a fast assignment discovery mechanism specific to $\alpha$Top-$K$.

---

[2]SSPA can be modified to discover the second best assignment (with or without constraints) as described in [8].

We note that [18] applied the branching technique in Murty's algorithm (which is in principle similar to CH) to find top-$K$ shortest paths by essentially replacing its SSPA component with a "next shortest path" mechanism. In this sense it is related to MCH, although not applicable to $\alpha$Top-$K$. We also stress that MCH, albeit more practical than CH, is still more than two times slower than our best algorithm, BN.

## 2.3 Assignment Problems in Large Databases

Similar to our work, several recent studies consider assignment problems in large scale, where the standard operations research approaches fail. For instance, [30] considers the spatial version of the traditional *stable marriage* problem [12]. The object and supplier sets ($P$ and $S$) have spatial coordinates, and the preference of an object to a supplier is inversely proportional to the Euclidean distance between them. A stable marriage can be achieved if the closest object-supplier pair is iteratively output as part of the matching, and the corresponding object and supplier are removed from $P$ and $S$, respectively. The algorithm terminates when $P$ or $S$ is empty. Performance in [30] is improved via geometric optimizations.

[28] considers a preference-based stable marriage problem. Each object indicates a scoring function that determines its preference with respect to the suppliers. Processing follows the best match principle. Similarly to closest pairs in the spatial problem of [30], the result is produced by iteratively matching the object-supplier pair with the largest preference score in the system.

The spatial version of optimal matching is considered in [29]. Here the cost of each object-supplier assignment is equal to their Euclidean distance. The geometric characteristics of the problem are used to accelerate SSPA, while indexes are employed to support efficient retrieval according to spatial conditions (e.g., range and nearest neighbor queries).

All aforementioned methods (except [28]) are targeted at spatial data and rely on properties of the Euclidean space. [28] produces a stable matching (based on local criteria), which is by definition different than minimizing a global cost function. Most importantly, all algorithms in this subsection report a single assignment, as opposed to the very objective in $\alpha$Top-$K$ to produce multiple alternatives.

Although not a work on assignment computation, [2] addresses a problem of some relevance to ours, in the sense that it also needs to search through numerous combinations. Consider the purchase of a mobile device, for which a number of accessories (e.g., speakers, batteries, etc) are available, each at a different price. The objective is to identify all possible combinations of accessories whose purchase does not exceed the user's budget. A side-problem is to select the $K$ most representative among these combinations (similar to a set-cover problem). A second side-problem is to place the $K$ representative combinations into a linear order so that the result is visually optimal, i.e., consecutive combinations have as different composition from each other as possible. Although this is also a problem of a combinatorial nature, it is fundamentally different from ours, and so are the techniques proposed.

## 3. PROBLEM FORMULATION

In this section we formally define the $\alpha$Top-$K$ problem. Table 1 includes frequently used notation and its description. The input to the query consists of: (i) object set $P$, (ii) supplier set $S$, (iii) parameter $K$ (a natural number), (iv) $|P|$ cost lists, and (v) a cost function $f$. There is a cost list $L_i$ for every object $p_i \in P$. $L_i$

### Table 1: Notation

| Symbol | Description |
|--------|-------------|
| $P$ | object set |
| $S$ | supplier set |
| $|P|, |S|$ | cardinalities of sets $P$ and $S$, respectively |
| $K$ | number of assignments to be reported |
| $L_i$ | cost list of object $p_i \in P$ |
| $f(A)$ | the (aggregate) cost of assignment $A$ |
| $w_i$ | the weight of object $p_i \in P$ |
| $\mathcal{R}$ | $\alpha$Top-$K$ result; set of $K$ best assignments |
| $A_i$ | assignment with the $i$-th smallest cost |
| $H$ | min-heap for assignment search |
| $A'$ | second best matching in a node (for MCH) |
| $\mathcal{R}_{int}$ | interim result (for CP) |

includes tuples $\langle s_j, c_{ij} \rangle$, where $c_{ij}$ is the cost of assigning $p_i$ to $s_j$, and is sorted in ascending $c_{ij}$ order. For simplicity, we assume that each cost list includes a tuple for every supplier (i.e., each object can be assigned to any supplier); however, our algorithms apply trivially when only a subset of suppliers may offer an object (see Section 9). Cost lists may reside in main memory or on the disk.

A valid matching $A$ consists of $|P|$ object-supplier pairs $\langle p_i, s_j \rangle$, each implying that $p_i$ is assigned to $s_j$. An object $p_i$ appears in exactly one pair in $A$, i.e., it is assigned to one supplier. A supplier $s_j$, on the other hand, may appear in multiple pairs – in our motivating application, for instance, the same supplier could be used to offer different aircraft parts $p_i, p_j, p_k$, etc.

Table 2 is an example of $\alpha$Top-$K$ input. $P$ holds four objects ($p_1, ..., p_4$), and $S$ includes three suppliers ($s_1, s_2, s_3$). Each cell ($p_i, s_j$) in the table indicates the corresponding $c_{ij}$ cost. Figure 1 shows the four cost lists, sorted in ascending $c_{ij}$ order from bottom to top. We associate with each tuple $\langle p_i, s_j, c_{ij} \rangle$ a character for quick reference (shown next to the respective list element).

### Table 2: Object-Supplier Costs

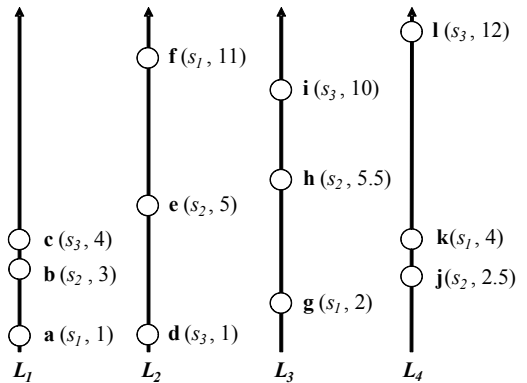|       | $s_1$ | $s_2$ | $s_3$ |
|-------|-------|-------|-------|
| $p_1$ | 1     | 3     | 4     |
| $p_2$ | 11    | 5     | 1     |
| $p_3$ | 2     | 5.5   | 10    |
| $p_4$ | 4     | 2.5   | 12    |



**Figure 1: Cost lists in running example**

The total cost of valid matching $A$, denoted by $f(A)$, is the weighted sum of individual $c_{ij}$ values across its object-supplier pairs, i.e.,

$$f(A) = \sum_{\langle p_i, s_j \rangle \in A} w_i \cdot c_{ij} \qquad (1)$$

where $w_i$ is a positive weight reflecting the significance of each $p_i \in P$. For ease of presentation we assume equal, unit weights, but also experiment with alternative settings in Section 10.

The output of the $\alpha$Top-$K$ query is a set $\mathcal{R}$ that contains the $K$ valid assignments with the smallest total costs; i.e., $|\mathcal{R}| = K$, and for each valid assignment $A \notin \mathcal{R}$ and for every $A_i \in \mathcal{R}$ it holds that $f(A) \geq f(A_i)$. We keep the assignments in $\mathcal{R}$ sorted in ascending cost order, and denote the $i$-th of them as $A_i$. In our running example, if $K = 4$, the $\alpha$Top-$K$ result is:

- $A_1 = \{\langle p_1, s_1 \rangle, \langle p_2, s_3 \rangle, \langle p_3, s_1 \rangle, \langle p_4, s_2 \rangle\}$; cost 6.5,
- $A_2 = \{\langle p_1, s_1 \rangle, \langle p_2, s_3 \rangle, \langle p_3, s_1 \rangle, \langle p_4, s_1 \rangle\}$; cost 8,
- $A_3 = \{\langle p_1, s_2 \rangle, \langle p_2, s_3 \rangle, \langle p_3, s_1 \rangle, \langle p_4, s_2 \rangle\}$; cost 8.5,
- $A_4 = \{\langle p_1, s_3 \rangle, \langle p_2, s_3 \rangle, \langle p_3, s_1 \rangle, \langle p_4, s_2 \rangle\}$; cost 9.5.

To simplify presentation, we refer to an assignment by its pair characters and its cost. For example, $A_1$ is represented as $\{a,d,g,j|6.5\}$, where 6.5 indicates $f(A_1)$.

## 4. A PRELIMINARY SOLUTION

In this section we present crucial observations that allow dealing with the combinatorial nature of the problem and lay the foundation for efficient computation of "the best" and "next best" matchings. These observations lead to a preliminary solution, termed *Expansion-Based* algorithm (EB). EB demonstrates the crux of our methodology, but is still impractical for large datasets, motivating the advanced algorithms in Sections 5, 6 and 7.

### 4.1 Crucial Observations

Our first observation is that the best (optimal) matching $A_1$ can be formed by assigning each object $p_i \in P$ to the supplier $s_j$ with the smallest $c_{ij}$ in cost list $L_i$. In other words, $A_1$ is derived by the first entry of every object's cost list (e.g., $\{a,d,g,j|6.5\}$ in Figure 1). This follows by the definition of the total cost in Equation 1.

Assume now that we want to find the second best matching $A_2$. $A_2$ can be derived by replacing *exactly one* of the pairs in $A_1$, say $\langle p_i, s_j \rangle$, with another assignment for $p_i$. This must be true, since replacing any two (or more) pairs of $A_1$ would increase the total cost more than making either of these replacements individually. Furthermore, the pair used to replace $\langle p_i, s_j \rangle$ should assign object $p_i$ to its second best supplier, i.e., to the supplier that corresponds to the second element of cost list $L_i$. Indeed, replacing $s_j$ with any other supplier in $L_i$ would rise the assignment cost more.

Consider Figure 1. Our observations suggest that the second best assignment $A_2$ is one of $\{b,d,g,j|8.5\}$, $\{a,e,g,j|10.5\}$, $\{a,d,h,j|10\}$, or $\{a,d,g,k|8\}$. By evaluating these four candidates, i.e., by calculating their total costs, we can report the smallest-cost one (i.e., $\{a,d,g,k|8\}$) as $A_2$. Although $K = 1$ and $K = 2$ are easy to handle, processing is not straightforward when $K > 2$. Before dealing with $K > 2$ we introduce the concept of *descendant* assignments.
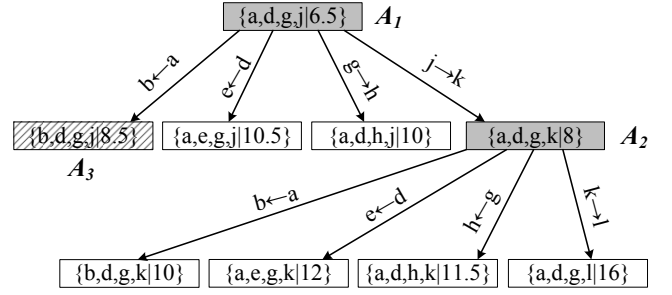


**Figure 2: Assignment tree example**

DEFINITION 1. **Descendant assignment:** *An assignment $A_{desc}$ is a descendant of another $A$ iff it can be derived by replacing exactly one of the pairs in $A$ with the pair that corresponds to the immediately next element in the respective object's cost list.*

In Figure 1 assignment $\{c,d,g,j\}$ is a descendant of $\{b,d,g,j\}$. Conversely, $\{a,d,g,j\}$, $\{b,e,h,j\}$, and $\{b,d,i,j\}$ are not. In general, each matching $A$ has $|P|$ descendants, and (with the exception of the root) is itself a descendant of at least one other assignment. By definition, the descendants of $A$ have costs no smaller than $f(A)$.

Our previous observations suggest that $A_2$ is among the descendants of $A_1$. To visualize the descendant relationships in our running example, we use the *assignment tree* in Figure 2. Each node corresponds to an assignment, with $A_1$ forming the root. The children of a node represent its descendant matchings[3]. The shaded nodes in the figure are the top-2 assignments, and the remaining nodes (leaves of the tree) are their descendants. Lemma 1 implies that the third best matching $A_3$ is among these leaves. In general, the following holds:

LEMMA 1. *Assuming that we have discovered the top-n assignments (where $n \in \mathbb{N}^*$), the next best matching $A_{n+1}$ is among their descendants.*

PROOF. We saw in the beginning of the section that the lemma holds for $n = 1$. The rationale for larger $n$ is similar. For simplicity, we make the assumption that there is no pair of assignments with identical costs. Recall that all assignments (except the root) are descendants of at least one other matching, and so is $A_{n+1}$. Let $\mathcal{R}$ be the set of the top-$n$ discovered matchings. We know that $A_{n+1}$ has a cost larger than any assignment in $\mathcal{R}$, and at the same time smaller than any other. In other words, all matchings with a smaller cost are already in $\mathcal{R}$. By Definition 1, $A_{n+1}$ may only be descendant of nodes with smaller costs, which (as proven previously) are all inside $\mathcal{R}$. $\square$

### 4.2 Expansion-based Algorithm

Lemma 1 may be directly applied via EB, a preliminary $\alpha$Top-$K$ algorithm. EB is reminiscent of a Dijkstra search [10] in the assignment tree, building the latter incrementally and on-the-fly. Initially, the tree contains only the root ($A_1$). Its descendants are evaluated and pushed into a min-heap $H$, with their total costs as sorting keys. We iteratively pop the top assignment of the heap $A_{top}$ and include

---

[3]A matching may be a descendant of multiple others, but the way our algorithms construct the assignment tree, only one parent node is recorded, as we describe later. Thus, the structure remains a tree and not a DAG.

it in the result as the next best assignment. Whenever a matching is popped, we evaluate its descendants and push them into the heap (provided that they have not been en-heaped previously, so as to avoid duplicates). The algorithm terminates when $K - 1$ assignments have been popped. $A_1$ and these $K - 1$ matchings are output as the $\alpha$Top-$K$ result. The correctness of EB is guaranteed by Lemma 1 and the sorting property of the min-heap. Algorithm 1 summarizes the process.

---

**Algorithm 1** Expansion-based Algorithm

1: Initialize empty set $\mathcal{R}$ and empty min-heap $H$
2: Form $A_1$ from first entries of cost lists; $\mathcal{R} := \{A_1\}$
3: Push the ($K$ best) descendants of $A_1$ into $H$
4: **while** $\mathcal{R}$ contains fewer than $K$ assignments **do**
5:     $A_{top} := \text{pop}(H)$
6:     $\mathcal{R} := \mathcal{R} \cup \{A_{top}\}$
7:     Push the ($K$ best) descendants of $A_{top}$ into $H$
8: Return $\mathcal{R}$

---

Table 3 shows the result set and heap contents of EB in the example of Figure 1 for $K = 4$. Iteration 1 reports $A_1$ and en-heaps its descendants. Iteration 2 pops $A_2$ and pushes its descendants into $H$. Note that at this stage the heap contents correspond to the leaves of the assignment tree in Figure 2. $A_3$ is the next popped matching (shown striped in Figure 2). The algorithm terminates when $H$ is popped for the third time, reporting $A_4$.

**Table 3: Example of EB ($K = 4$)**

| | Result set $\mathcal{R}$ | Heap contents |
|---|---|---|
| 1 | {a,d,g,j\|6.5} | **{a,d,g,k\|8}**,{b,d,g,j\|8.5} {a,d,h,j\|10},{a,e,g,j\|10.5} |
| 2 | {a,d,g,j\|6.5} {a,d,g,k\|8} | **{b,d,g,j\|8.5}**,{b,d,g,k\|10} {a,d,h,j\|10},{a,e,g,j\|10.5} {a,d,h,k\|11.5},{a,e,g,k\|12} {a,d,g,l\|16} |
| 3 | {a,d,g,j\|6.5} {a,d,g,k\|8} {b,d,g,j\|8.5} | **{c,d,g,j\|9.5}**,{b,d,g,k\|10} {a,d,h,j\|10},{a,e,g,j\|10.5} {a,d,h,k\|11.5},{a,e,g,k\|12} {b,d,h,j\|12},{b,e,g,j\|12.5} {a,d,g,l\|16} |
| 4 | {a,d,g,j\|6.5} {a,d,g,k\|8} {b,d,g,j\|8.5} {c,d,g,j\|9.5} | **{b,d,g,k\|10}**,{a,d,h,j\|10} {a,e,g,j\|10.5},{c,d,g,k\|11} {a,d,h,k\|11.5},{b,d,h,j\|12} {a,e,g,k\|12},{b,e,g,j\|12.5} {c,d,h,j\|13},{c,e,g,j\|13.5} {a,d,g,l\|16} |

Although EB is a first step towards efficient $\alpha$Top-$K$ processing, it is impractical for large scale problems. For every popped matching, there are $|P|$ descendants. Hence, to process an $\alpha$Top-$K$ query, $O(K \cdot |P|)$ assignments are pushed into $H$.

The situation can be improved based on the fact that no more than $K$ assignments will be output in total, and therefore only the $K$ best (i.e., smallest-cost) descendants of each reported assignment need to be pushed into $H$. Even with this optimization, EB en-heaps $O(K^2)$ assignments and exhibits a long running time, as we show in the experiments. EB, however, constitutes a significant improvement over assignment ranking algorithms and lays the foundation for our advanced algorithms. It is also used as a building block in Section 7 in a special case where only two cost lists exist; in this case it en-heaps around $2 \cdot K$ matchings only, yielding favorable performance.

## 5. MODIFIED CH ALGORITHM

In this section we develop the *Modified CH* algorithm (MCH). MCH embeds the crux of EB into the traditional CH paradigm. Traditional CH has a time complexity of $O(K \cdot |P|^3)$ and requires $O(|P| \cdot |S|)$ space, both due to its reliance on SSPA. SSPA is needed to compute the top-2 matchings ($A$ and $A'$) for each node of the branch-decision tree subject to specific constraints. The main idea in MCH is to drop SSPA, and instead utilize the EB discovery mechanism of Section 4 to derive assignments $A$ and $A'$.

We present the algorithm using our running example in Figure 1. We illustrate its branch-decision tree in Figure 3, denoting its nodes as $N_i$. Originally, the tree includes only the root $N_1$, which requires computing $A_1$ and $A_2$ without any constraint ($I = \emptyset$ and $O = \emptyset$). Instead of SSPA, we use the observations in Section 4.1; $A_1$ is created by the first elements of the cost lists, and $A_2$ is its smallest-cost descendant. To find $A_3$, the root must be split. To perform the split, the CH paradigm requires identifying the pair that belongs to $A_1 = \{a,d,g,j|6.5\}$ but not to $A_2 = \{a,d,g,k|8\}$. This pair is $j$. Thus, the left child has $I = \{j\}$ and the right $O = \{j\}$.
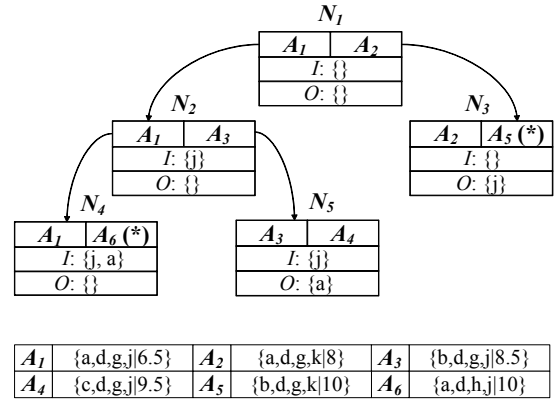


**Figure 3: Example of MCH ($K = 4$)**

Consider first the left child $N_2$ (where $I = \{j\}$ and $O = \emptyset$). Since $j$ is mandatory, we perform a top-2 search in the remaining three cost lists. The top assignment $A$ of the node is derived by the first elements of $L_1$, $L_2$, and $L_3$ (i.e., {a,d,g|4}), plus $j$ itself. The second best assignment $A'$ is formed by $j$ plus the smallest-cost descendant of {a,d,g|4}, i.e., the best among {b,d,g|6}, {a,e,g|8} and {a,d,h|7.5}, deriving $A' = \{b,d,g,j|8.5\}$. Note that in Figure 3 we represent each node as a table, where the first row includes its top assignment $A$ followed by the second best, $A'$; the mapping of assignment identifiers to their contents is shown below the tree.

The right child of the root, $N_3$, has constraints $I = \emptyset$ and $O = \{j\}$. Search for its top-2 assignments is identical to Section 4.1, the difference being that $j$ is ignored when considering $L_4$, and search starts as if $k$ was the first element in this cost list. That is, the top assignment $A$ for $N_3$ is {a,d,g,k|8}. The second best is the smallest-cost descendant of $A$; among {b,d,g,k|10}, {a,e,g,k|12}, {a,d,h,k|11.5}, and {a,d,g,l|16}, we pick the first as $A'$.

After splitting the root, the CH methodology suggests that the next best matching overall ($A_3$) is the smallest-cost $A'$ assignment in the leaves of the branch-decision tree. By comparing the second matchings of the leaves, {b,d,g,j|8.5} and {b,d,g,k|10} (for $N_2$ and $N_3$, respectively), we choose the former as $A_3$. Note that at this

stage we cannot tell the overall ranking of the remaining {b,d,g,k|10}. Leaf $N_2$ that owns $A_3$ is split into $N_4$ and $N_5$, and their own top and second best assignments are computed.

The fourth best assignment $A_4$ is chosen among the second best assignments $A'$ of the leaves ($N_3$, $N_4$, $N_5$), i.e., from $N_5$. Observe that at that point the overall ranking of the second best matchings in the remaining leaves is unknown, indicated by an asterisk (next to $A_5$ and $A_6$).

In general, the rule to compute the top-2 assignments in a node with constraint sets $I$ and $O$ is (i) to ignore the entire cost lists of all mandatory pairs in $I$, as they are directly included in both $A$ and $A'$, and (ii) in the remaining cost lists, to ignore the pairs that belong to $O$. Top-2 computation considers only the remaining lists, and proceeds similarly to Section 4.1. Algorithm 2 summarizes MCH. Note that the child node constraints $I$ and $O$ in Line 7 are produced from $N$ in the way described in Section 2.2.

---

**Algorithm 2** Modified CH

1: Initialize empty set $\mathcal{R}$
2: Form root of branch-decision tree from $A_1, A_2$
3: $\mathcal{R} := \{A_1\}$
4: **while** $\mathcal{R}$ contains fewer than $K$ assignments **do**
5: $\quad N :=$ the leaf with the smallest-cost $A'$
6: $\quad \mathcal{R} := \mathcal{R} \cup \{A'\}$
7: $\quad$ Split $N$; get top-2 matchings in children
8: Return $\mathcal{R}$

---

**Analysis:** Each output assignment requires splitting a node in the branch-decision tree. Thus, we have a maximum of $2 \cdot K$ nodes. Every node requires computing top-2 assignments (subject to $I$ and $O$ constraints) which takes $O(|P|)$ time; this is to find the min-score descendant $A'$ of the node's top assignment $A$. To report the next best assignment we need to search through the leaves of the tree at cost $O(K)$ – there are at most $K - 1$ leaves at any point. Overall, the time complexity of MCH is $O(K(|P| + K))$.

## 6. BEST NEIGHBOR ALGORITHM

MCH exploits in part the properties of the $\alpha$Top-$K$ query (in computing top-2 matchings), but it does not utilize in full the power of Lemma 1 and the problem's characteristics. Here we present the *Best Neighbor* algorithm (BN), which builds on the idea of the assignment tree to derive a more efficient $\alpha$Top-$K$ technique. It abandons CH and extends the EB principles instead.

The main problem of EB is that for every reported assignment, $K$ others (descendants) are pushed into the search heap. This leads to a large heap size and to prohibitively slow processing. BN solves this problem, and guarantees that no more than two assignments are en-heaped per reported matching, i.e., that no more than $2 \cdot K$ push operations are necessary overall. In addition to accelerating processing, this also keeps memory consumption low. The following two definitions are important for the description of BN.

DEFINITION 2. **Seed assignment:** *If an assignment $A$ is discovered through (i.e., as a descendant of) another $A_{seed}$, we call the latter the seed of $A$.*

DEFINITION 3. **Sibling assignments:** *We call siblings the assignments that have the same seed.*

Figure 4 illustrates the general idea in BN. The assignment tree refers to the same setting as Figure 2 and juxtaposes processing in
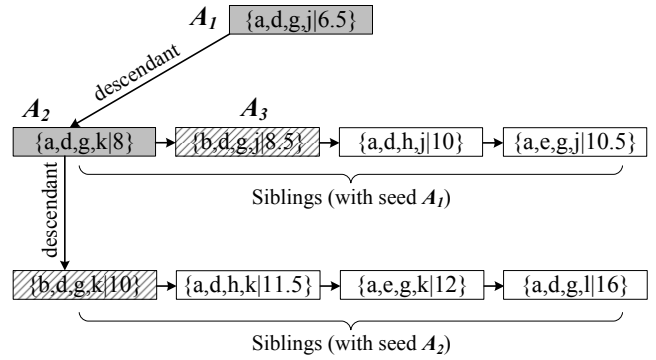


**Figure 4: Example of assignment tree in BN**

EB and BN. The crucial observation in BN is that in the second level of the assignment tree, for example, we may (i) sort the descendants of the root, (ii) drop their links with the root, except for the smallest-cost one, and (iii) link the siblings with each other in ascending cost order. In the figure, the descendants of the root form the horizontal chain below it, linked in ascending cost order from left to right. Dropping the seed-descendant links, as point (ii) suggests, does not affect the reporting order of the siblings, because anyway the smaller-cost ones will be visited before the larger in a Dijkstra-like search of the assignment tree. The benefit is that only the smallest-cost descendant (i.e., {a,d,g,k|8}, shown shaded) needs be en-heaped after popping the root. In turn, when $A_2 = $ {a,d,g,k|8} is reported, only the two striped nodes are en-heaped; the first is its immediately larger-cost sibling {b,d,g,j|8.5}, the second is its own smallest-cost descendant {b,d,g,k|10} (i.e., the first matching in the sibling chain that corresponds to seed $A_2$). Except for the root, each popped/reported assignment pushes exactly two others. Note that most siblings and descendants of a popped matching will never be en-heaped due to their large costs.

### 6.1 Algorithm Design and Optimizations

Search in the assignment tree requires addressing several issues, while it is also enables powerful optimizations.

**Limiting push operations:** The first issue arises from Definitions 2 and 3. Since an assignment $A$ may be a descendant of multiple others, there are many possible seeds for it. Similarly, $A$ has multiple sets of siblings (one for each possible seed). While we could maintain all seeds of $A$ and en-heap its next siblings for each seed when $A$ is popped, this would lead to many push operations. To avoid this, for every assignment $A$ we record a single seed $A_{seed}$, which is the first matching that led to en-heaping $A$. In Figure 4, for instance, {b,d,g,k|10} keeps $A_2 = $ {a,d,g,k|8} as its only seed. If $A$ is subsequently encountered via another seed $A_x$, and at some point it needs to be en-heaped as the (first or the) next costlier sibling among the descendants of $A_x$, we will detect that $A$ is already in the search heap[4], and instead push its immediately larger-cost sibling for seed $A_x$. This way we avoid unnecessary push operations and significantly accelerate processing (while retaining correctness, since siblings of $A_x$ are encountered in the right order).

**Assignment evaluation:** Another issue in BN is that when an assignment $A$ is popped, all its descendants must be evaluated in or-

---

[4]In our sorted-container heap implementation, re-en-heap is prevented by detection of sorting key (i.e., $f(A)$ value) duplication.

der to be sorted. Blindly applying Definition 1 to derive the descendants and computing the total cost of each of them is computationally expensive. An efficient way to evaluate and sort the descendants is the following. Each pair $\langle p_i, s_j \rangle$ in seed $A$ refers to the cost list $L_i$ for some object $p_i \in P$. The descendant that corresponds to $L_i$ has cost equal to the seed assignment's $f(A)$ plus the cost increase between seed's pair $\langle p_i, s_j \rangle$ and the next item in $L_i$. Consider the descendants of the root $A_1 = \{a,d,g,j|6.5\}$ in Figure 1. The cost of the descendant that corresponds to $L_2$, i.e., $\{a,e,g,j\}$, is $f(A_1)$ (which is 6.5) plus the cost difference between items $d$ and $e$ (which is 4). Since component $f(A_1)$ is common among descendants, to sort them we may only use the cost difference in the corresponding cost list, e.g., 2 in the case of $L_1$, 4 in the case of $L_2$, etc. In case of unequal weights $w_i$ in Equation 1, cost differences are multiplied by the corresponding weight $w_i$ before sorting.

**Sibling bookkeeping:** A key point about BN is that the descendants of every seed must be kept in memory. This is important so that when the next sibling for a seed must be en-heaped, we do not have to re-evaluate and re-sort all descendants to identify it. We keep the sorted descendants of each popped assignment in a separate structure, the *sibling table*. An issue with this approach is that every seed has $|P|$ descendants. This requires storing a total of $K \cdot |P|$ matchings in the sibling table and consumes a large amount of space. A key observation to cure the problem is that a maximum of $K$ descendants are necessary per seed, because in no case will we need to report more than $K$ assignments. Assume that (during BN execution) we pop an assignment $A_i$ and our result set at that point includes $|\mathcal{R}| < K$ results. We keep in the sibling table only the $K - |\mathcal{R}|$ smallest-cost descendants of $A_i$, because under no circumstances will we need more matchings to fill $\mathcal{R}$. In Figure 4 for $K = 4$, when $A_2$ is popped only its $K - |\mathcal{R}| = 2$ lowest cost descendants are kept in the sibling table (i.e., $\{b,d,g,k|10\}$ and $\{a,d,h,k|11.5\}$), and the rest ($\{a,e,g,k|12\}$ and $\{a,d,g,l|16\}$) are discarded.

**Assignment storage:** A necessary remark regards the storage of assignments. Since each matching includes $|P|$ object-supplier pairs, and we are targeting large datasets $P$, explicitly representing the matchings incurs a significant space overhead. Therefore, we record the contents of an assignment implicitly via a pair of values; the first indicates its seed, and the second the object identifier where it differs from the seed. For instance, in Figure 4, matching $\{a,d,h,k|11.5\}$ is represented as $\langle A_2, 3 \rangle$ (where $A_2$ is just an identifier of the seed assignment), implying that it can be derived from $A_2$ by replacing its pair for object $p_3$ with the immediately costlier one in list $L_3$. In turn, $A_2$ identifies the root as seed and indicates that it differs in the pair of $p_4$. The root itself is found from the first elements in the cost lists.

Regarding correctness of BN, the sorting of the search heap ensures reporting assignments in ascending cost order. This fact, in conjunction with Lemma 2, guarantees correctness.

LEMMA 2. *Assuming that BN has discovered so far the top-$n$ assignments (where $n \in \mathbb{N}^*$), the next best matching $A_{n+1}$ is inside its search heap.*

PROOF. Without loss of generality, assume that no pair of assignments have the same cost. All possible seeds of the next best matching $A_{n+1}$ have smaller costs, and therefore they must have been popped previously. Consider one of the possible seeds, say $A_i$. First, all the descendants of $A_i$ with cost smaller than $A_{n+1}$ have been reported already; otherwise, this would contradict the

fact that $A_{n+1}$ is the next best matching. Second, for the same reason, $A_{n+1}$ has the smallest cost among all the non-reported descendants of $A_i$. Thus, when $A_i$ (or its latest reported descendant with $A_i$ as seed) was popped, it either en-heaped $A_{n+1}$ as its best non-en-heaped descendant (as its next best sibling, respectively), or the only reason it failed to do so is because $A_{n+1}$ was already in the heap via another seed. □

Algorithm 3 provides the pseudo-code of BN. For every popped assignment $A_{top}$, Lines 9-10 en-heap its next (larger-cost) sibling with the same seed, skipping those that are already inside the search heap $H$. Lines 11-16 en-heap the smallest-cost descendant of $A_{top}$ that is not in $H$. The descendant stores $A_{top}$ as its seed. Note that the sibling table is organized per seed; e.g., Line 5 stores the descendants under seed $A_1$, and Line 17 under seed $A_{top}$.

---

**Algorithm 3** Best Neighbor Algorithm

---
1: Initialize empty set $\mathcal{R}$ and empty min-heap $H$
2: Form $A_1$ from first entries of cost lists; $\mathcal{R} := \{A_1\}$
3: Sort the descendants of $A_1$ in asc. cost order
4: Push the smallest-cost descendant into $H$
5: Store the $K - 1$ next descendants in sibl. table
6: **while** $\mathcal{R}$ contains fewer than $K$ assignments **do**
7:     $A_{top} := \text{pop}(H)$
8:     $\mathcal{R} := \mathcal{R} \cup \{A_{top}\}$
9:     $A_{nxt} :=$ next sibling of $A_{top}$ that is not in $H$
10:    Push $A_{nxt}$ into $H$
11:    Sort descendants of $A_{top}$ in ascending cost
12:    $A :=$ the smallest-cost descendant of $A_{top}$
13:    **if** $A$ is already in $H$ **then**
14:       $A :=$ next smallest cost desc. of $A_{top}$
15:       Go to Line 13
16:    Push $A$ into $H$ (with $A_{top}$ as its seed)
17:    Store $K - |\mathcal{R}|$ next descendants in sibl. table
18: Return $\mathcal{R}$

---

Table 4 demonstrates BN in the example of Figure 1. The seed of each en-heaped assignment is shown in parentheses next to it (in the right column of the table). Processing up to Iteration 2 is the same as discussed in Figure 4. Recall that when $A_2 = \{a,d,g,k|8\}$ is popped, only the two smallest-cost descendants are kept in the sibling table. Similarly, when $A_3 = \{b,d,g,j|8.5\}$ is reported, only its lowest cost descendant is maintained (because $K - |\mathcal{R}| = 1$ at that point). This descendant is $\{c,d,g,j|9.5\}$, and it is en-heaped in Iteration 3 along with the next sibling of $A_3$, $\{a,d,h,j|10\}$. Note that if $\{c,d,g,j|9.5\}$ happened to be already inside $H$ (via another seed), no descendant of $A_3$ would be en-heaped, since the sibling table includes no more.

**Table 4: Example of BN ($K = 4$)**

| | Result set $\mathcal{R}$ | Heap contents |
|---|---|---|
| 1 | {a,d,g,j\|6.5} | **{a,d,g,k\|8}**$(A_1)$ |
| 2 | {a,d,g,j\|6.5} | **{b,d,g,j\|8.5}**$(A_1)$ |
| | {a,d,g,k\|8} | {b,d,g,k\|10}$(A_2)$ |
| 3 | {a,d,g,j\|6.5} | **{c,d,g,j\|9.5}**$(A_3)$ |
| | {a,d,g,k\|8} | {b,d,g,k\|10}$(A_2)$ |
| | {b,d,g,j\|8.5} | {a,d,h,j\|10}$(A_1)$ |
| 4 | {a,d,g,j\|6.5} | **{b,d,g,k\|10}**$(A_2)$ |
| | {a,d,g,k\|8} | {a,d,h,j\|10}$(A_1)$ |
| | {b,d,g,j\|8.5} | {c,d,g,k\|11}$(A_4)$ |
| | {c,d,g,j\|9.5} | |

## 6.2 Analysis and Instance Optimality of BN

Each output assignment requires sorting its $|P|$ descendants (taking $O(|P|\log|P|)$ time) and en-heaping two new assignments. The "pop one, push two" feature of BN suggests that the heap size never exceeds $K$, implying an $O(\log K)$ cost per push/pop operation. Overall, the time complexity of BN is $O(K \cdot (|P|\log|P| + \log K))$.

Regarding access cost, we show that BN may access in each list $L_i$ at most one position deeper than an optimal-access-cost algorithm would. Let $d_i$ be the position of the deepest element in $i$-th list that appears in any of the final top-$K$ assignments. To ensure correctness, any $\alpha$Top-$K$ algorithm must reach at least depth $d_i$ in the $i$-th list. We prove that BN (and the same holds for EB) accesses at most $d_i + 1$ elements from list $L_i$. Assume that $A^i$ is the assignment in the $\alpha$Top-$K$ result that includes the deepest element in the $i$-th list (i.e., the element at position $d_i$). Since all assignments encountered by BN must have had their seed popped (and therefore reported as a result assignment), the deepest we look into $L_i$ is position $d_i + 1$ when considering the descendants of $A^i$.

Access cost is not a major concern in $\alpha$Top-$K$ processing per se. The result includes exactly $K$ matchings, which means that at most the first $K$ items from each cost list could appear in the top-$K$ assignments. The instance optimality of BN becomes important in case of incremental $\alpha$Top-$K$ where numerous assignments may be output before a satisfactory one is found.

## 7. CROSS PROBE ALGORITHM

In this section we propose the *Cross Probe* algorithm (CP), which extends our framework in two independent directions. First, in a centralized processing environment (as assumed so far), CP offers an alternative to MCH and BN with reduced space consumption. Second, CP enables parallel and distributed $\alpha$Top-$K$ processing. This allows for vast reductions in running time when multiple processors are available, and at the same time extends applicability to decentralized environments too. CP relies on Lemma 3 below.

Consider the merging of two (disjoint) object sets $P_1$ and $P_2$ to compute the $\alpha$Top-$K$ result of their union. Any produced matching $A$ must include all objects in $P_1 \cup P_2$. Thus, it can be broken into components $A_{P_1}$ and $A_{P_2}$, each specifying the suppliers for objects in $P_1$ and $P_2$, respectively. Either of $A_{P_1}$ and $A_{P_2}$ is a valid matching in the respective object set. Furthermore, from Equation 1 it follows that $f(A) = f(A_{P_1}) + f(A_{P_2})$. It holds that:

LEMMA 3. *Let $P_1$ and $P_2$ be two disjoint object sets, and $\mathcal{R}_1$, $\mathcal{R}_2$ be their respective $\alpha$Top-$K$ results (over the same supplier set $S$ and for the same parameter $K$). The $\alpha$Top-$K$ result for object set $P_1 \cup P_2$ comprises strictly assignments of the form $A_i \cup A_j$ where $A_i \in \mathcal{R}_1$ and $A_j \in \mathcal{R}_2$.*

PROOF. Let $\mathcal{R}$ be the $\alpha$Top-$K$ result for $P_1 \cup P_2$. Suppose that there is an assignment $A \in \mathcal{R}$ whose first component $A_{P_1}$ does not belong to the $\alpha$Top-$K$ result for $P_1$, i.e., $f(A_i) < f(A_{P_1})$ for each $A_i \in \mathcal{R}_1$ (for simplicity, we ignore ties). This means that for every $A_i \in \mathcal{R}_1$ we can construct an assignment $A_i \cup A_{P_2}$ for $P_1 \cup P_2$ with cost smaller than $A$ (the cost of the resulting assignment is $f(A_i) + f(A_{P_2})$ which is less than $f(A) = f(A_{P_1}) + f(A_{P_2})$, as follows from the hypothesis that $f(A_i) < f(A_{P_1}) \, \forall A_i \in \mathcal{R}_1$). In turn, this implies that there are $K$ matchings for $P_1 \cup P_2$ with costs smaller than $A$, which is a contradiction. The proof that the second component $A_{P_2}$ belongs to $\mathcal{R}_2$ is symmetric. $\square$

**Table 5: Example of CP** ($K = 4$)

| | New result $\mathcal{R}_{int}$ | Heap Contents |
|---|---|---|
| 1 | {$A_1$,W|7.5} | **{$A_1$,X|8.5}** |
| | | {$A_2$,W|9} |
| 2 | {$A_1$,W|7.5} | **{$A_2$,W|9}** |
| | {$A_1$,X|8.5} | {$A_2$,X|10} |
| | | {$A_1$,Y|10.5} |
| 3 | {$A_1$,W|7.5} | **{$A_3$,W|9.5}** |
| | {$A_1$,X|8.5} | {$A_2$,X|10} |
| | {$A_2$,W|9} | {$A_1$,Y|10.5} |
| 4 | {$A_1$,W|7.5} | **{$A_2$,X|10}** |
| | {$A_1$,X|8.5} | {$A_1$,Y|10.5} |
| | {$A_2$,W|9} | {$A_3$,X|10.5} |
| | {$A_3$,W|9.5} | {$A_4$,W|10.5} |

## 7.1 Memory-bound Processing

As we show in the experiments, the memory requirements of MCH and BN may grow substantially when $|P|$ or $K$ are very large (e.g., see Figures 9 and 11). CP trades, in a tunable way, computation efficiency for smaller space consumption. The idea is to partition the object set $P$ into equi-sized subsets, and process them one by one. CP first computes the $\alpha$Top-$K$ result of the first partition using BN, and places it into an interim result set $\mathcal{R}_{int}$ (we choose BN because it is significantly faster than competitors). Then, it runs BN on the second partition and computes its $\alpha$Top-$K$ result $\mathcal{R}_{tmp}$. Each assignment in $\mathcal{R}_{int}$ and $\mathcal{R}_{tmp}$ can be treated atomically as a list item with a cost, and therefore $\mathcal{R}_{int}$ and $\mathcal{R}_{tmp}$ can be seen as two cost lists. Lemma 3 guarantees that the $\alpha$Top-$K$ result on the union of the partitions can be found among all possible combinations of items from the two lists, which is itself an $\alpha$Top-$K$ problem. Hence, CP runs EB on $\mathcal{R}_{int}$ and $\mathcal{R}_{tmp}$, in order to compute the new interim result (we use EB because when only two lists are involved, it en-heaps just two assignments per reported matching (like BN) and is preferred due to its simplicity). Next, BN is run on the third partition, whose result is treated as a new list; EB is run on this list and the current interim result in order to produce the new interim result $\mathcal{R}_{int}$, and so on. When all partitions are processed, $\mathcal{R}_{int}$ is reported as the overall result $\mathcal{R}$. Algorithm 4 provides the pseudo-code of CP.

---

**Algorithm 4** Cross Probe Algorithm

1: Partition $P$ into $M$ equi-sized subsets ($P_1$ to $P_M$)
2: $\mathcal{R}_{int} := \text{BN}(P_1, K)$
3: **for** $i$ from 2 to $M$ **do**
4:      $\mathcal{R}_{tmp} := \text{BN}(P_i, K)$
5:      $\mathcal{R}_{int} := \text{EB}(\mathcal{R}_{tmp} \cup \mathcal{R}_{int}, K)$
6: Return $\mathcal{R}_{int}$

---

To illustrate, assume that $K = 4$ and the top-4 assignments in the current interim result $\mathcal{R}_{int}$ are $W, X, Y, Z$ with costs 1, 2, 4, and 7, respectively. Suppose that the new partition includes the objects in Figure 1 – we apply BN on them, receiving as $\mathcal{R}_{tmp}$ the assignments $A_1, A_2, A_3, A_4$, as described previously in Table 4. To produce the new interim result, we run EB with input $\mathcal{R}_{tmp}$ and the current $\mathcal{R}_{int}$. Table 5 sketches this EB process. Observe that the elements of both cost lists are not object-supplier pairs, but entire assignments. The first assignment in the new interim result is the combination of $A_1$ and $W$ with cumulative cost 7.5. Its descendants are {$A_1$,X|8.5} and {$A_2$,W|9} which are en-heaped, and so on. The fourth row of the table shows the new interim result.

As we show in the experiments, although CP is slower than MCH

and BN, it consumes very little space. The tradeoff between the space and time requirements of CP is controlled by the partition size. Small partitions imply little memory consumption but slow processing, and vice versa. Essentially, the larger the partitions, the more CP performs like BN.

## 7.2 Parallel and Decentralized Processing

Lemma 3 has another useful application. It allows for parallel processing and, thus, greater scalability. So far we considered $\alpha$Top-$K$ computation on a single processor. However, if several are available, the various object partitions of CP can be processed in parallel by different processors (to derive their local $\alpha$Top-$K$ results). Subsequently, the results of various partitions can be (treated as cost lists themselves and be) further processed to retrieve the overall $\alpha$Top-$K$ result. Note that Lemma 3 extends trivially to cases where more than two object sets are merged, i.e., we are not bound to two-way aggregation of partition results, but multi-way is also possible (albeit EB should be replaced by BN in this case).

A similar direction is to apply Lemma 3 to fully decentralized architectures. Assume that each object maintains its own cost list and also has some computation and communication capabilities, but there is no central processor to carry out query processing. The first object could send the top-$K$ elements of its cost list to the second object. The latter would run EB to produce the $\alpha$Top-$K$ result for the first two objects, forward it to the third, and so on until all objects have run EB over their local list and the partial result they received; the last object derives the overall result.

## 8. INCREMENTAL PROCESSING

So far we have focused on standard $\alpha$Top-$K$, but our methods apply to incremental $\alpha$Top-$K$ too. As explained in Introduction, this is useful when a decision maker (or an application) repetitively requests for the next best assignment until a satisfactory one is found.

By their nature, all our algorithms can incrementally output the next best matching, without knowing $K$ in advance. In EB, the heap can be popped iteratively, reporting its top as the next best matching. In Table 3, if no $K$ were specified, processing would be identical, and the fifth best matching would be the next assignment popped ($A_5 = \{b,d,g,k|10\}$). MCH also works incrementally by default, as its reporting/splitting mechanism is independent of $K$.

Incremental BN is similar to EB. However, we need to store all the descendants of each popped assignment, i.e., the $K - |\mathcal{R}|$ limit cannot be applied. In the example of Table 4, Iteration 3 would keep in the sibling table all descendants of $A_3$ (instead of just one). Since this may consume considerable space, only the first few descendants could be kept in the sibling table; if more are needed later, we may compute and store the next few descendants, and so on.

Regarding CP, we initialize a (local) incremental BN process on each partition. The local BN results form the input of a (global) incremental BN, which probes the local searches whenever a (global) en-heap requires accessing the next unavailable local item.

## 9. DISCUSSION

Here we cover several extensions and features of our methods.

**Selective assignment:** In Section 3 we made the assumption that every object can be assigned to any supplier. In practice, there may be cases where this does not hold. EB, MCH, and BN deal with this

situation by establishing the convention that any supplier $s_j$ which cannot provide an object $p_i$ has cost $c_{ij} = \infty$. To save space, a single infinite entry is needed per cost list. If a popped assignment includes an infinite entry, the algorithm has already reported all valid matchings and terminates.

**Object quantities:** Another note regards object quantities, i.e., situations where multiple instances of an object $p_i$ need to be assigned to suppliers. In this case, we may treat each instance as a separate cost list. This is necessary so that different instances of $p_i$ can be assigned to different suppliers. If there is an additional requirement that all instances of an object must be assigned to the same supplier, a single list is used with its $c_{ij}$ values multiplied by the number of instances (or weight $w_i$ incremented accordingly).

**Result post-processing:** The nature of $\alpha$Top-$K$ suggests that the top-$K$ assignments may share large numbers of common object-supplier pairs. In some applications, the decision maker might wish to examine small-cost assignments, which however are sufficiently different from each other. For situations like that, our framework could be invoked for a large $K$ value, e.g., in the order of several hundreds or more if necessary. The $\alpha$Top-$K$ results could then be post-processed to choose a few of them as representative (sufficiently distinct) alternatives, using a method similar in spirit to [2], described in Section 2.3.

## 10. EXPERIMENTS

In this section we evaluate our algorithms, focusing on processing time and space requirements. All methods access the first $K$ items per cost list at most. For MCH and BN the truncated cost lists are kept in memory throughout execution, while CP needs only the (truncated) lists of the current partition processed. We enhance MCH and CP with compact assignment representation, and with the optimized descendant evaluation technique in Section 6.1.

In Section 10.2 we present experiments with real data. We use the Jester collection[5], because it is a real dataset of sufficiently large scale that comprises actual preference values (ratings). Specifically, it includes ratings of 73,421 users on 100 jokes [14]. There are 4.1 million ratings, each being a real number in the range (-10, 10). Users play the role of $P$ (objects), jokes that of $S$ (suppliers), and ratings that of $c_{ij}$ values. We use the opposite of each rating to comply with the convention that lower $c_{ij}$ implies higher preference. On the average, each cost list holds 56 entries.

To establish the generality of our results, and gain control over more parameters (namely, $|P|$ and $|S|$), we also experimented on synthetic data in Section 10.3. For that set of experiments we generated the $c_{ij}$ values randomly and independently. All algorithms were executed on an Intel Xeon 64-bit 3.16GHz CPU.

By default, we assume equal weights $w_i$ for every object in $P$ (referring to Equation 1 and cost function $f$). However, we also examine unequal weights of different distributions.

Before presenting our full-scale experiments, in Section 10.1 we use small problem instances in order to verify (i) the deficiencies of EB and (ii) the inability of rank join algorithms to deal with our problem. We subsequently disqualify these approaches from our full-fledged empirical evaluation.

---

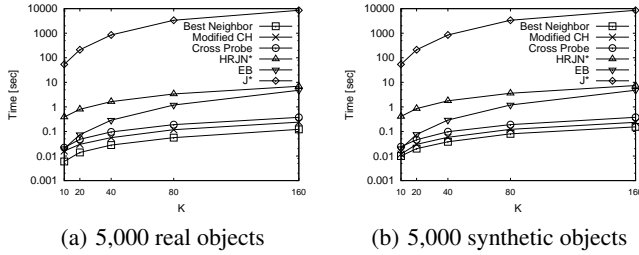[5]Available at http://eigentaste.berkeley.edu/dataset/

(a) 5,000 real objects     (b) 5,000 synthetic objects

**Figure 5: Impact of result size $K$ (5,000 objects only)**



(a) Processing time     (b) Memory requirements

**Figure 6: Impact of partition size on CP (Jester dataset)**



(a) Processing time     (b) Memory requirements

**Figure 7: Impact of result size $K$ (Jester dataset)**

## 10.1 Ineffectiveness of EB and Rank Join

In Section 2.1 we explained that $\alpha$Top-$K$ can theoretically be reduced to rank join processing, where the join condition is always true. Specifically, each sorted relation corresponds to a cost list $L_i$, every "join" result (i.e., every combination of entries in the cost lists) represents a possible matching $A$, and the sum of $c_{ij}$ values in $A$ indicate its cost $f(A)$. We use HRJN* and J* as representatives of the rank join class. We compare them against our methods for only 5,000 objects, so that they terminate within reasonable time. These objects are random samples of the Jester and the synthetic datasets, respectively. In our small-scale comparison we also include (the optimized version of) EB.

The computation time of the algorithms for various $K$ values is shown in Figures 5(a) and 5(b) for 5,000 real and 5,000 synthetic data, respectively. For CP, the partition size used is 1/10 of the object set $P$. We postpone the discussion on the trends and performance of our advanced algorithms for the subsequent sections on the complete datasets. Considering HRJN*, in the charts we observe its clear inability to scale, being 17-19 times slower than the least efficient of our advanced methods (CP). J* is slower than HRJN*, as also found in [16] for the rank join problem.

As explained in Section 2.1, the high-level reason is that rank join algorithms are designed for a different problem where there are real joins (whereas in our case every matching is a candidate), there is a small number of input relations, and the objective is to reduce I/O cost (not CPU time). To be specific, performance in our setting depends on the number of evaluated matchings. The issue in HRJN* is that *each* new tuple fetched from a relation (i.e., cost list) is joined with *all* tuples fetched so far in all other relations/lists. Suppose there are $|P| = 3$ lists. If we fetch an item from one list and the other two have been explored up to depths $d_1$ and $d_2$ respectively, then $d_1 \cdot d_2$ matchings need to be evaluated. This product has $|P| - 1$ factors and becomes intractable for large numbers of lists. The main problem with J*, on the other hand, is that it uses a form of uninstantiated elements in en-heaped assignments (which are therefore incomplete) in order to enable its A* pruning. When the number of lists is large, heap operations on incomplete assignments dominate its processing time and cripple performance.

EB is very inefficient as well, due to the en-heaping of $O(K^2)$ assignments. This also explains the trends of its processing time. In the following experiments we focus on large-scale problems and on the advanced $\alpha$Top-$K$ algorithms only (i.e., MCH, BN and CP).

## 10.2 Real Data

In this section we experiment on the Jester data. We first consider the effect of partition size on CP. Figure 6 shows its processing time and peak memory requirements; partition sizes are expressed as fractions of $|P|$, ranging from 1/160 to 1/5. $K$ is fixed to 20.
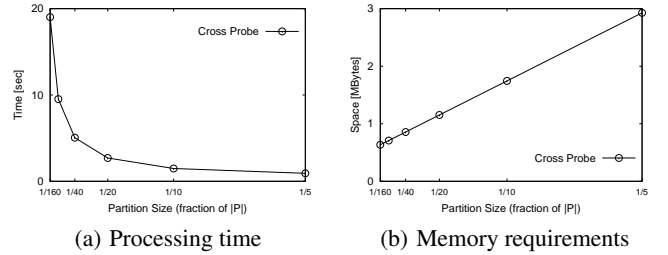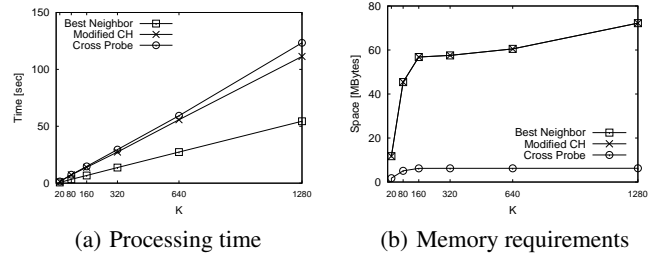
Small partitions imply slower processing. The reason is that for each partition, not all of its $K$ (local) assignments appear in the final $\alpha$Top-$K$ result, which means waste of computations for the extra matchings computed. The smaller the partition, the fewer the utilized local assignments. On the other hand, memory consumption is dominated by the cost lists, and thus requirements increase almost linearly with partition size. In the following we use 10 partitions in CP (fraction 1/10) as it strikes a good balance between space and time.

In Figure 7 we investigate the effect of $K$ on MCH, BN and CP. In MCH, every reported assignment requires splitting its owning node $N$ into two leaves. The first assignment of each leaf comes from $N$ itself, but the second best (i.e., $A'$) must be computed. Hence, roughly $2 \cdot K$ assignments are computed overall, which explains the near-linear increase of running time with $K$. In BN, every reported assignment requires en-heaping another two, leading again to a linear relationship between running time and $K$. CP relies on a BN building block and follows a similar trend.

The most efficient algorithm is BN – it is 2.0 to 2.2 times faster than the runner-up (MCH). This confirms that BN exploits to a larger degree the specific properties of the problem. Both methods form two new assignments per reported matching. The difference is that MCH computes them both from scratch (the $A'$ matchings in the leaves created by splitting $N$). BN, on the other hand, finds the sibling of the popped assignment directly (from the sibling table), and only needs to compute its best descendant. Often times, no sibling is en-heaped at all, since the $K - |\mathcal{R}|$ rule may have left no more descendants in the sibling table for some seeds, leading to further improvement. On the other hand, CP is the slowest method.

Memory requirements are dominated by the space allocated for the (truncated) cost lists. Space consumption for all approaches increases with $K$, because the necessary part of the cost lists grows. Interestingly, the increase rate drops when $K \geq 80$ and slows down further after $K = 160$. In Jester dataset the average cost list holds 56 entries. This means that the majority of lists are already stored in memory in their entirety for $K = 80$, and a further increase in
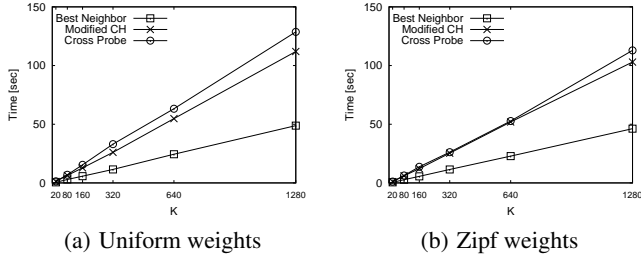
(a) Uniform weights

(b) Zipf weights

**Figure 8: Processing time for unequal weights $w_i$ (Jester)**



(a) Processing time

(b) Memory requirements

**Figure 9: Impact of number of objects $|P|$ (synthetic data)**



(a) Processing time

(b) Memory requirements

**Figure 10: Impact of number of suppliers $|S|$ (synthetic data)**

$K$ does not affect the space they occupy. For $K \geq 160$ almost all lists are entirely stored in memory, and the increase in space requirements is now due to encountering (and thus representing/en-heaping) more assignments as $K$ grows.

BN and MCH use similar space, since they keep the same number of cost lists. CP requires the least space, because at any point it keeps in memory only a subset of the $|P|$ cost lists. By trading processing time for lower space requirements (in a tunable way), CP can effectively capture the specific time-versus-space requirements of the application at hand.

In Figure 8 we repeat the previous experiment, using however unequal object weights $w_i$. In Figure 8(a) we plot processing time for weights uniformly distributed in range (0,1] and in Figure 8(b) for weights that follow Zipf distribution with parameter 0.2. The trends are similar to the equal-weight setting, indicating that the weight distribution does not significantly affect performance. Memory requirement plots are practically identical to Figure 7(b) and omitted.

## 10.3 Synthetic Data
Here we present experiment results with synthetic data, so that (i) we are able to vary the number of objects and suppliers to assess scalability (this is not possible with the real data) and (ii) verify that the performance trends in Section 10.2 are not specific to the Jester dataset. Table 6 shows value ranges for the tested parameters, with their defaults in bold. In each chart we vary one parameter and set the rest to their defaults. We finetuned the partition size for CP similarly to Figure 6 and chose size 1/10 of $|P|$ as the default.

**Table 6: Experiment Parameters**

| Parameter | Value Range |
|---|---|
| $K$ | **20**, 80, 160, 320, 640, 1280 |
| $|P|$ | 10, 50, **100**, 500, 1000 ($\times 1000$) |
| $|S|$ | 8, 32, **128**, 512, 2048 |

We first examine scalability with $|P|$ and $|S|$. In Figure 9 we vary the number of objects $|P|$ from $10^4$ to $10^6$ and keep the remaining parameters to their defaults. A larger object set implies that the number of possible assignments increases, making the problem tougher. The number of descendants per seed is $|P|$. The implication is that more assignments need to be evaluated and compared whenever a matching is included in the result for both BN and MCH. The former however is around 1.5 times more efficient in all cases. The effect is similar for CP, whose partitions grow larger (recall that we set their size to 1/10 of $|P|$). Also, memory consumption increases with $|P|$, since more cost lists are stored.

In Figure 10 we measure the impact of $|S|$ (number of suppliers) on performance. There is an interesting observation here. The CPU
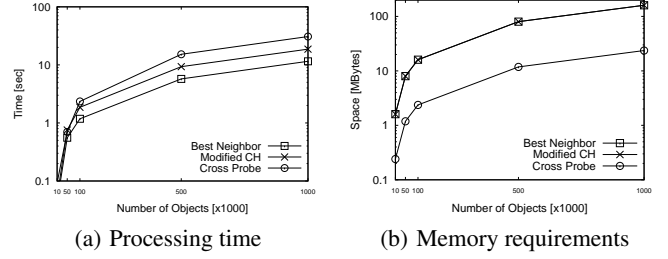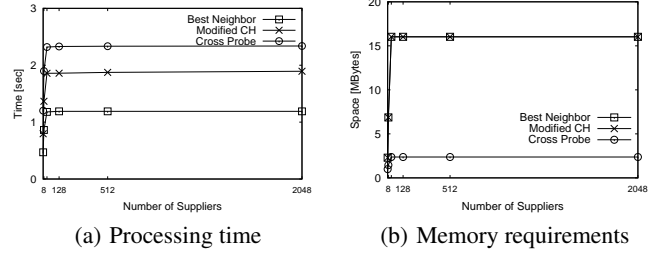
time for all methods originally increases until $|S|$ reaches $K$ (i.e., 20), and then it plateaus. The reason is that when $|S| < K$ the algorithms reach the end of several cost lists, thus needing to consider fewer alternatives/descendants. Since cost lists are shorter (having length $|S|$), memory requirements are lower too. On the other hand, when $|S| \geq K$, only the first $K$ items (i.e., the first 20) of each list need to be stored and processed. Therefore, for $|S| \geq 20$, processing time and peak memory stabilize.

In Figure 11 we vary $K$ and keep the remaining parameters to their defaults. The results are similar to Figure 7, with a slightly smaller gap between BN and alternatives. Finally, like in Jester data, alternative object weight settings have a minor effect on performance and the corresponding charts are omitted for brevity.
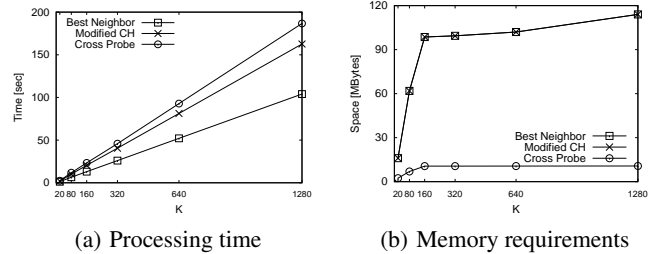


(a) Processing time

(b) Memory requirements

**Figure 11: Impact of result size $K$ (synthetic data)**

## 10.4 Parallel Processing
In Figure 12 we investigate the performance of BN and MCH when multiple processors are available. In the presence of $M$ processors, the object set is partitioned into $M$ subsets. Each partition is assigned to a processor, which applies BN on it (or MCH for the white bars). The $M$ derived results are assigned to one of the (now unoccupied) processors, which executes an $M$-way BN on them (or $M$-way MCH for the white bars) to produce the final $\alpha$Top-$K$ result. We use from 1 up to 4 processors and plot running times for Jester and synthetic data. The processing cost drops significantly with $M$, illustrating great scalability potential via parallelism. Total memory requirements are practically unaffected by $M$ and are
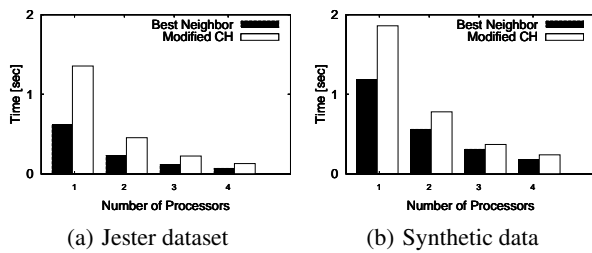
**Figure 12: Running time with multiple processors**

similar to those reported in previous experiments (for $M = 1$), thus omitted from the figure.

## 10.5 Summary of Experiments

BN is consistently the most efficient $\alpha$Top-$K$ algorithm. CP requires the least memory, trading efficiency for space; as the number of partitions drops, CP slowly degenerates to BN (which happens when a single partition is used). All advanced algorithms scale gracefully to object sets in the order of millions. Furthermore, our framework allows for vast performance improvements via parallelism when multiple processors are available.

## 11. CONCLUSION

In this paper we study the top-$K$ assignment query ($\alpha$Top-$K$). We propose highly scalable algorithms for its processing, one of which offers a controllable tradeoff between time and space requirements. We also extend our framework with parallel and decentralized versions of our methods. The practicality of our algorithms is verified through experiments on real and synthetic data. An interesting direction for future work is $\alpha$Top-$K$ processing with uncertain (or probabilistic) object-supplier costs, e.g., when the costs are replaced by ranges of possible values.

## 12. REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] S. Basu Roy, S. Amer-Yahia, A. Chawla, G. Das, and C. Yu. Constructing and exploring composite items. In *SIGMOD Conference*, pages 843–854, 2010.

[3] Boeing. www.boeing.com/commercial/747family/pf/pf_facts.html.

[4] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[5] R. E. Burkard, M. Dell'Amico, and S. Martello. *Assignment Problems*. SIAM, 2009.

[6] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In *SIGMOD Conference*, pages 391–402, 2000.

[7] S. Chaudhuri, L. Gravano, and A. Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Trans. Knowl. Data Eng.*, 16(8):992–1009, 2004.

[8] C. R. Chegireddy and H. W. Hamacher. Algorithms for finding k-best perfect matchings. *Discrete Applied Mathematics*, 18(2):155 – 165, 1987.

[9] U. Derigs. A Shortest Augmenting Path Method for Solving Minimal Perfect Matching Problems. *Networks*, 11(4):379–390, 1981.

[10] E. W. Dijkstra. A note on two problems in connexion with graphs. *Num. Mathematik*, 1:269–271, 1959.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[12] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69:9–15, 1962.

[13] A. V. Goldberg and R. Kennedy. An efficient cost scaling algorithm for the assignment problem. *Mathematical Programming*, 71(2):153–177, 1995.

[14] K. Y. Goldberg, T. Roeder, D. Gupta, and C. Perkins. Eigentaste: A constant time collaborative filtering algorithm. *Inf. Retr.*, 4(2):133–151, 2001.

[15] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB Journal*, 13(1):49–70, 2004.

[16] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.

[17] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):1–58, 2008.

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

[19] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[20] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

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

[22] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001.

[23] C. R. Pedersen, L. Relund Nielsen, and K. A. Andersen. An algorithm for ranking assignments using reoptimization. *Comput. Oper. Res.*, 35(11):3714–3726, 2008.

[24] S. J. Russell and P. Norvig. *Artificial intelligence - a modern approach: the intelligent agent book*. Prentice Hall, 1995.

[25] K. Schnaitter and N. Polyzotis. Evaluating rank joins with optimal cost. In *PODS*, pages 43–52, 2008.

[26] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound Processing of Ranked Queries. *Inf. Syst.*, 32(3):424–445, 2007.

[27] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, pages 277–288, 2003.

[28] L. H. U, N. Mamoulis, and K. Mouratidis. A fair assignment algorithm for multiple preference queries. *PVLDB*, 2(1):1054–1065, 2009.

[29] L. H. U, M. L. Yiu, K. Mouratidis, and N. Mamoulis. Capacity constrained assignment in spatial databases. In *SIGMOD Conference*, pages 15–28, 2008.

[30] R. C.-W. Wong, Y. Tao, A. W.-C. Fu, and X. Xiao. On efficient spatial matching. In *VLDB*, pages 579–590, 2007.

[31] Z. Zhang, S. won Hwang, K. C.-C. Chang, M. Wang, C. A. Lang, and Y.-C. Chang. Boolean + ranking: querying a database by k-constrained optimization. In *SIGMOD Conference*, pages 359–370, 2006.