Capacity Constrained Assignment in Spatial Databases*

Leong Hou U
University of Hong Kong
hleongu@cs.hku.hk

Man Lung Yiu
Aalborg University
mly@cs.aau.dk

Kyriakos Mouratidis
Singapore Management University
kyriakos@smu.edu.sg

Nikos Mamoulis
University of Hong Kong
nikos@cs.hku.hk

ABSTRACT

Given a point set P of customers (e.g., WiFi receivers) and a point set Q of service providers (e.g., wireless access points), where each $q \in Q$ has a capacity $q.k$, the *capacity constrained assignment* (CCA) is a matching $M \subseteq Q \times P$ such that (i) each point $q \in Q$ ($p \in P$) appears at most k times (at most once) in M, (ii) the size of M is maximized (i.e., it comprises $\min\{|P|, \sum_{q \in Q} q.k\}$ pairs), and (iii) the total assignment cost (i.e., the sum of Euclidean distances within all pairs) is minimized. Thus, the CCA problem is to identify the assignment with the optimal overall quality; intuitively, the quality of q's service to p in a given (q, p) pair is anti-proportional to their distance. Although max-flow algorithms are applicable to this problem, they require the complete distance-based bipartite graph between Q and P. For large spatial datasets, this graph is expensive to compute and it may be too large to fit in main memory. Motivated by this fact, we propose efficient algorithms for *optimal assignment* that employ novel edge-pruning strategies, based on the spatial properties of the problem. Additionally, we develop *approximate* (i.e., suboptimal) CCA solutions that provide a trade-off between result accuracy and computation cost, abiding by theoretical quality guarantees. A thorough experimental evaluation demonstrates the efficiency and practicality of the proposed techniques.

1. INTRODUCTION

Consider a point set P of customers (e.g., WiFi receivers) and a point set Q of service providers (e.g., wireless access points). Suppose that each service provider $q \in Q$ is able to serve at most $q.k$ customers and every customer has at most one service provider. A subset $M \subseteq Q \times P$ is said to be a *valid matching* if (i) each point $q \in Q$ ($p \in P$) appears at most $q.k$ times (at most once) in M and (ii) the size of M is maximized (i.e., it is $\min\{|P|, \sum_{q \in Q} q.k\}$). To quantify the quality of the matching M, we define its *assignment cost* as:

$$\Psi(M) = \sum_{(q,p) \in M} dist(q, p) \quad (1)$$

where $dist(q, p)$ denotes the Euclidean distance between q and p. Intuitively, a high-quality matching should have low assignment cost.

Figure 1 illustrates a scenario where P={p_1, ..., p_{12}}, Q={q_1, q_2, q_3}, $q_1.k = q_3.k = 3$, and $q_2.k = 5$. Intuitively, assigning to each q_i the points p_j that fall inside its Voronoi cell (indicated by dashed lines in the figure) leads to the minimum matching cost. However, this approach ignores the service provider capacities. In our example, it assigns 5, 3, and 4 objects to q_1, q_2 and q_3, respectively, violating the capacity constraints of q_1 and q_3. The optimal CCA matching, on the other hand, would assign $\{p_2, p_3, p_4\}$ to q_1, $\{p_5, ..., p_9\}$ to q_2, and $\{p_{10}, p_{11}, p_{12}\}$ to q_3, as shown by the three ellipses. In the general case $\sum_{q \in Q} q.k \neq |P|$, i.e., the customers may be fewer or more than the cumulative capacity of the service providers. CCA assigns every $p_j \in P$ to a $q_i \in Q$, unless all service providers have reached their capacity. In Figure 1, for instance, p_1 is not assigned to any q_i, since they are all full. Conversely, it is possible that some service providers are not fully utilized. In any case, CCA computes the maximum size matching with the minimum assignment cost, subject to the capacity constraints.

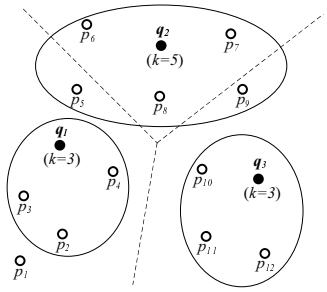


Figure 1: Spatial assignment example

Besides the aforementioned wireless communication scenario, CCA arises in many resource allocation applications that require matching between users and facilities based on capacity constraints and spatial proximity. For instance, the municipality could assign children to schools (with certain capacity each) such that the average (or, equivalently, the summed) traveling distance of children to their schools is minimized. Another application (in welfare states) is the assignment of residents to designated, public clinics of given individual capacities.

CCA can be reduced to the well-known *minimum cost flow* (MCF) problem in a complete distance-based bipartite graph between Q and P [1]. In the operations research literature [13], there is an abundance of MCF algorithms based on this reduction. These solutions, however, are only applicable to small-sized datasets and main memory processing. In particular, the best of them have a cubic time complexity (elaborated on in Section 2.2), and require that the bipartite graph (which contains $|Q| \cdot |P|$ edges) resides in memory. For moderate and large size datasets, this graph requires a pro-

*Supported by grant HKU 7155/06E from Hong Kong RGC.

hibitive amount of space (exceeding several times the typical memory sizes), and leads to an excessive computation cost as the CCA complexity increases with the number of graph edges.

Motivated by the lack of CCA algorithms for large datasets, we develop efficient and highly scalable CCA techniques that produce an *optimal assignment*. Specifically, we assume that P resides in secondary storage, indexed by a spatial access method, while Q fits in main memory; in most real-world applications $|Q| << |P|$ and the capacities $q_i.k$ are in the order of tens or hundreds. We use the MCF reduction as a foundation, but we achieve space and computation scalability by exploiting the spatial properties of the problem and incrementally including into the graph only the necessary edges. Targeted at a disk-resident P, our methods take into account and reduce the I/O cost by incorporating elaborate index-based enhancements. Furthermore, we extend our framework with *approximate* solutions that leverage similar edge-pruning strategies and provide a tunable trade-off between processing cost and assignment quality; we analyze the inaccuracy incurred and devise theoretical bounds for the deviation from the optimal matching.

The rest of the paper is organized as follows. Section 2 covers background and existing work related to our problem. Section 3 presents the central theorem our approach is stemming from, along with our optimal CCA algorithms that utilize it. Section 4 studies the trade-off between computation cost and matching quality, and develops approximate CCA solutions with guaranteed matching quality. Section 5 empirically evaluates our exact and approximate CCA methods using synthetic and real datasets. Finally, Section 6 concludes the paper with future research directions.

2. BACKGROUND AND RELATED WORK

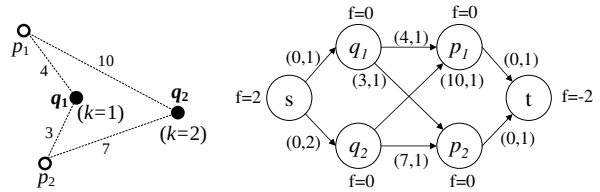
CCA can be reduced to a flow problem on a graph. In Section 2.1 we describe the graph formulation of CCA, and in Section 2.2 we describe a traditional main memory algorithm for the corresponding flow problem. Even though this solution is inapplicable to our setting, it is fundamental to our techniques. In Section 2.3 we survey spatial queries and algorithms related to our approach. Table 1 summarizes the notation used in this and the following sections.

Symbol	Description
Q	set of service providers (points)
P	set of customers (points)
$dist(q_i, p_j)$	the Euclidean distance between q_i and p_j
$e(q_i, p_j)$	the (directed) edge from q_i to p_j
s	the source node
t	the sink node
$v.\alpha$	minimum cost from s to node v
$v.\tau$	potential value of node v
$v.prev$	prev. node of v in shortest path from s to v
v_{min}	the last node in current sp that belongs to P

Table 1: Notation

2.1 Minimum Cost Flow on Bipartite Graph

CCA can be reduced to a maximum flow problem on a (directed) bipartite graph [1]. Consider the example in Figure 2(a), where $P = \{p_1, p_2\}$, $Q = \{q_1, q_2\}$, and $q_1.k = 1$, $q_2.k = 2$. This CCA problem is represented by the *flow graph* shown in Figure 2(b). The flow graph is a complete bipartite graph between Q and P, extended with two special nodes s and t (called the *source* and the *sink*, respectively) and $|Q| + |P|$ extra edges from/to these nodes. Specifically, letting V be the set of nodes in the graph, then $V = Q \bigcup P \bigcup \{s, t\}$. Each node $v \in V$ has a fixed *balance* $f(v)$. For every $p \in P$ and $q \in Q$, the balance is set to 0. For s and t, $f(s) = \gamma$ and $f(t) = -\gamma$, where γ is the *required flow* and $\gamma = \min\{|P|, \sum_{q \in Q} q.k\}$. In our example, $\gamma = \min\{2, 3\} = 2$ and the balances are shown next to each node in the figure.



(a) Spatial locations (b) Flow graph

Figure 2: CCA reduction to the MCF problem

Let E represent the set of edges in the flow graph. Each edge $e(v_i, v_j) \in E$ has a *cost* $w(v_i, v_j)$ and a *capacity* $c(v_i, v_j)$. The set of edges E comprises: (i) an edge $e(s, q_i)$ for every service provider $q_i \in Q$, with cost 0 and capacity $q_i.k$ (modeling the capacity constraint of the service provider), (ii) an edge $e(q_i, p_j)$ for every pair of service provider $q_i \in Q$ and customer $p_j \in P$, with cost $dist(q_i, p_j)$ (e.g., in Figure 2, $w(q_1, p_2) = dist(q_1, p_2) = 3$) and capacity 1 (implying that pair (q_i, p_j) can appear at most once in the final matching M), and (iii) an edge $e(p_j, t)$ for every customer $p_j \in P$, with cost 0 and capacity 1 (implying that p_j is assigned to at most one service provider). In Figure 2(b), the label of each edge indicates (in parentheses) its cost and capacity.

Given the above graph, the *minimum cost flow* (MCF) problem is to associate an integer *flow value* $x(v_i, v_j) \in [0, c(v_i, v_j)]$ with each edge $e(v_i, v_j) \in E$ such that for every node $v \in V$ it holds that:

$$\sum_{e(v, v_m) \in E} x(v, v_m) - \sum_{e(v_m, v) \in E} x(v_m, v) = f(v) \qquad (2)$$

and the following objective function $\mathcal{Z}(x)$ is minimized:

$$\mathcal{Z}(x) = \sum_{e(v_i, v_j) \in E} w(v_i, v_j) \cdot x(v_i, v_j) \qquad (3)$$

An optimal CCA assignment is derived by solving the MCF problem and including in M these and only these pairs (q_i, p_j) for which $x(p_j, q_i) = 1$ [1]. Intuitively, every edge $e(p_j, q_i)$ with $x(p_j, q_i) = 1$ incurs cost $w(p_j, q_i) = dist(p_j, q_i)$ and $\Psi(M) = \mathcal{Z}(x)$. Also, the required flow γ ensures (according to Equation 2) that M has the full size, i.e., that M covers the maximum possible number of customers.

Several algorithms have been proposed in the literature for solving MCF in main memory [1]. The *Hungarian algorithm* [8, 11] constructs a cost matrix with $|Q| \cdot |P|$ entries, performs subtraction/addition for entries in specific rows/columns, until each row/column has at least one zero value. This solution is limited to small problem instances; it becomes infeasible even for moderate-sized problems, as the aforementioned matrix may not fit in main memory.

The *cost scaling algorithm* [1, 5] solves the assignment problem using a reduction to MCF. It processes the latter

through successive approximations in a number of steps that is logarithmic to the maximum edge cost in the flow graph. This approach is inapplicable to CCA, since it works only for one-to-one matching (i.e., $q.k = 1$ for all $q \in Q$) and strictly integer edge costs (versus real-valued ones in CCA).

2.2 Successive Shortest Path Algorithm

The *Successive Shortest Path Algorithm* (SSPA) is a popular technique for the MCF problem defined above. SSPA receives as input the flow graph defined in Section 2.1 and performs $\gamma = \min\{|P|, \sum_{q \in Q} q.k\}$ iterations. In each iteration, it computes the shortest path from the source s to the sink t, and reverses the path's edges. After the last iteration, every (directed) edge from a point in P to a point in Q corresponds to a pair in the optimal matching M.[1]

Algorithm 1 is the detailed pseudo-code of SSPA. In each loop, SSPA invokes Dijkstra's algorithm to compute the shortest path *sp* between the source and the sink; the algorithm adheres to the edge directions and cannot pass through edges $e(s, q_i)$ (or, $e(p_j, t)$) that were already included in $c(s, q_i)$ ($c(p_j, t)$, respectively) shortest paths at previous loops. For a visited node v (i.e., a node de-heaped during Dijkstra's algorithm), we use $v.\alpha$ to refer to its minimum distance from the source, and $v.prev$ to indicate the node it was reached from. We denote by v_{min} the last node in the current shortest path that belongs to P (note that *sp* may be passing via multiple points of P). Upon *sp* computation in Line 2, SSPA traces it back and reverses the direction of all the edges it contains (Lines 5, 6); we say that this step *augments* the path into the graph. Then, SSPA updates the *potential* (to be discussed shortly) of the nodes visited by Dijkstra's algorithm (Lines 8, 9), and the costs of the edges incident to these nodes (Lines 10, 11).

Algorithm 1 Successive Shortest Path Algorithm (SSPA)

```
   algorithm SSPA(Set Q, Set P, Edge set E)
 1: for loop:=1 to γ do
 2:     v_min:=Dijkstra(Q, P, E)
 3:     v:=v_min //v is a local variable of node type
 4:     while v.prev ≠ ∅ do
 5:         add e(v, v.prev) to E, with c(v, v.prev)=−c(v.prev, v)
 6:         delete e(v.prev, v) from E
 7:         v:=v.prev                ▷ proceed with the previous object
 8:     for all visited nodes v_i do
 9:         v_i.τ:=v_i.τ − v_i.α + v_min.α
10:         for all edges e(v_i, v_j) incident to v_i do
11:             w(v_i, v_j):=dist(v_i, v_j) − v_i.τ + v_j.τ
```

An important step in SSPA is the edge cost updating performed in Line 11. To ensure that no edge cost becomes negative (which is a requirement for the correctness of Dijkstra's algorithm), SSPA uses the concept of *node potentials*. The potential $v.\tau$ of a node $v \in V$ is a non-negative real value that is initialized to 0 for all $v \in V$ before the first SSPA loop, and is subsequently updated in Lines 8, 9 whenever v is visited (i.e., de-heaped) by Dijkstra's algorithm. The cost of an edge $w(v_i, v_j)$ varies during the execution of SSPA, and is defined as $dist(v_i, v_j) - v_i.\tau + v_j.\tau$ at all times (we establish the convention that $dist(v_i, v_j) = 0$ if any of v_i, v_j is s or t). The node potentials and the definition of edge costs play an important role in SSPA and in our methods described in Section 3.

[1]Note that this is equivalent to forming M by edges $e(p_j, q_i)$ with flow 1, as described in Section 2.1. For simplicity, in our SSPA description we omit flow computation per se, and focus on retrieving the optimal CCA matching directly.

Example: Consider the CCA example and flow graph in Figure 2. SSPA performs in total $\gamma = 2$ iterations. Figure 3(a) shows the flow graph of Figure 2(b) appended with the initial potentials next to each node (all set to 0). In the first iteration, SSPA finds the shortest path $sp_1 = \{s, q_1, p_2, t\}$ from the source to the sink. Then, it augments *sp* and updates the flow graph to be used in the next iteration; Figure 3(b) illustrates the reversed *sp* edges, the updated node potentials and the new edge costs. Figure 3(c) shows in bold the shortest path $sp_2 = \{s, q_2, p_2, q_1, p_1, t\}$ found in the second iteration. Note that sp_2 cannot pass through edges $e(s, q_1)$ and $e(p_2, t)$, since they have already been used $c(s, q_1) = 1$ and $c(p_2, t) = 1$ times in previous shortest paths (i.e., in sp_1). Figure 3(d) augments sp_2 and updates the flow graph. Even though this is the last iteration of SSPA, it exemplifies an interesting case. Edge $e(s, q_2)$ is part of sp_2, but it is not "completely" reversed; its capacity is 2, and only one of its "instances" is reversed, which leads to (i) decreasing its capacity by 1 (instead of deleting it), and (ii) creating reverse edge $e(q_2, s)$ with capacity 1 and cost 0. To complete the example, optimal assignment M corresponds to edges from P to Q in the resulting flow graph after the $\gamma = 2$ iterations, i.e., it contains (q_1, p_1) and (q_2, p_2).

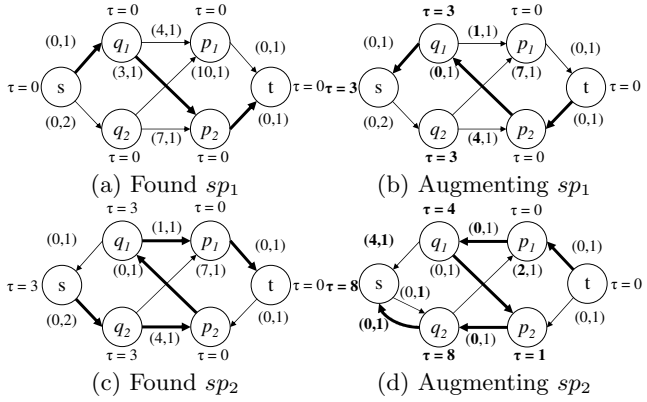


Figure 3: Example of *SSPA*

SSPA requires that the entire flow graph resides in main memory. The graph contains an excessive number of $O(|Q| \cdot |P|)$ edges, which do not fit in memory for large problem instances. Moreover, the time complexity of SSPA is $O(\gamma \cdot (|E| + |V| \cdot log|V|))$, where $O(|E| + |V| \cdot log|V|)$ is the cost to compute a shortest path. Since in our targeted applications $|E|$ is quite large ($O(|P| \cdot |Q|)$), SSPA is particularly slow. Another fundamental problem of SSPA is that it is designed for main memory processing and ignores the I/O cost, which generally is the most critical performance factor in the processing of disk-resident data.

2.3 Spatial Queries

Point sets are usually indexed by spatial access methods in order to accelerate query processing. The R-tree [6] and its variants (e.g., [9, 2]) are the most common such indexes. The R-tree is a disk-based, balanced tree that groups together nearby points into leaf nodes, and recursively groups these leaf nodes into higher level nodes (again based on their proximity) up to a single root. Each non-leaf entry is as-

sociated with a minimum bounding rectangle (MBR) that encloses all the points in the subtree pointed by it.

Typical spatial search operations on a point set P are range and nearest neighbor (NN) queries. Given a range value r and a query point q, the r-range query retrieves all points of P within (Euclidean) distance r from q. If P is indexed by an R-tree, this query is evaluated by following recursively R-tree entries that intersect the circular disk with center at q and radius r. The K-nearest neighbor (KNN) query receives as input an integer K and a query point q, and returns the K points of P that are closest to q. The state-of-the-art KNN processing technique is the best-first NN algorithm [7], which employs a heap for organizing encountered R-tree entries and visiting them in ascending order of their distance from q, until K points are discovered.

In the context of large spatial databases, assignment problems have recently received considerable attention. Specifically, [12, 14] study the *spatial matching* (SM) join. Given two point sets P and Q, the SM join iteratively outputs the closest pair [4] (p, q) in $P \times Q$, reports (p, q) as an assigned pair, and removes both p and q from their corresponding datasets before the next iteration. This procedure continues until either P or Q becomes empty. [14] enhances the performance of a naïve (i.e., repetitive closest pair) algorithm with several geometric observations. Although SM is related to CCA, it is a different problem by definition; SM greedily performs local assignments instead of minimizing the global assignment cost.

3. EXACT METHODS

In this section we present our methods for computing optimal CCA assignments. In accordance with most real-world scenarios, we consider that Q (the set of of service providers) is much smaller than P (the set of customers). We assume that Q fits into main memory, while P is stored on the disk. In the following we consider two-dimensional points and that P is indexed by an R-tree. However, our algorithms can easily extend to problems of higher dimensionality and other spatial access methods.

As explained in Section 2.2, SSPA is not applicable to large CCA problem instances, as its (complete bipartite) flow graph leads to excessive memory consumption and expensive shortest path computations. To alleviate the space and running time problems incurred by the huge flow graph, we develop *incremental* SSPA-based algorithms that start from an empty flow graph and insert edges into it gradually. Intuitively, edges with low edge weights are highly probable to indicate pairs in the optimal assignment. A fundamental theorem (presented below) formalizes this intuition and excludes from consideration edges whose cost is too high to affect the result of SSPA. Additionally, our techniques exploit the spatial index of P to further improve performance.

Our general idea is to perform the search in a subgraph with edge set $E_{sub} \subseteq E$, where E is the complete set of flow graph edges. We refer to the Euclidean distance between the nodes of an edge as its *length*. Let function $\phi(\cdot)$ take as input a set of edges and return the minimum edge length in it. To facilitate the derivation of distance bounds, we require E_{sub} to be *distance-bounded*, as defined below.

DEFINITION 1. *An edge set $E_{sub} \subseteq E$ is said to be* distance-bounded *if*

$$\forall\, e(q_i, p_j) \in E_{sub},\ dist(q_i, p_j) \leq \phi(E - E_{sub})$$

In other words a distance-bounded E_{sub} contains those and only those edges in E that have length less than or equal to a threshold (i.e., $\phi(E - E_{sub})$). Conversely, all the remaining edges (i.e., edges in $E - E_{sub}$) have length greater than or equal to that threshold. We stress that function $\phi(\cdot)$ and Definition 1 refer to edge lengths, and not to their costs (note that costs vary during the execution of our algorithms because the node potentials are updated).

Suppose that we are given a distance-bounded edge set E_{sub}. Consider an execution of Dijkstra's algorithm on E_{sub} that computes the shortest path sp between the source and the sink, and the potential values $v_i.\tau$ for every node v_i, derived as described in Section 2.2. The following theorem determines the condition that should hold so that sp is the shortest path on the complete edge set E.

THEOREM 1. *Consider a distance-bounded edge set $E_{sub} \subseteq E$. Let sp be the shortest path (between source and sink) in E_{sub} and $\tau_{max} = \max\{v_i.\tau | v_i \in V\}$ be the maximum potential value. If the total cost of sp is at most $\phi(E - E_{sub}) - \tau_{max}$, then sp is also the shortest path (between source and sink) on the complete flow graph.*

PROOF. Consider the edges in $E - E_{sub}$. First, their minimum length is $\phi(E - E_{sub})$. Second, as explained in Section 2.2, their costs are defined as $w(v_i, v_j) = dist(v_i, v_j) - v_i.\tau + v_j.\tau$. Since $dist(v_i, v_j) \geq \phi(E - E_{sub})$, $v_i.\tau \leq \tau_{max}$, and $v_j.\tau \geq 0$, it holds that

$$w(v_i, v_j) \geq \phi(E - E_{sub}) - \tau_{max}, \forall e(c_i, v_j) \in E - E_{sub}$$

According to the above (and since edge costs are always non-negative), any path passing through an edge in $E - E_{sub}$ has at least a cost of $\phi(E - E_{sub}) - \tau_{max}$. Therefore, if the shortest path sp (on E_{sub}) has total cost no greater than $\phi(E - E_{sub}) - \tau_{max}$, then it must be the shortest path in the entire E too. □

In the following we investigate approaches for gradually expanding the subgraph E_{sub} and use it to derive CCA pairs. Our first solution incrementally enlarges range searches around points in Q. The other two aim at further reducing the size of E_{sub} by replacing range queries with incremental nearest neighbor searches [7].

3.1 Range Incremental Algorithm

Our first method is the *Range Incremental Algorithm* (RIA). Algorithm 2 presents the pseudo-code of RIA. The procedure starts with an initial range T equal to a system parameter θ. For every point $q_i \in Q$, RIA performs a T-range search in P; for each retrieved point $p_j \in P$, edge $e(q_i, p_j)$ is inserted into E_{sub}. RIA invokes SSPA in the resulting E_{sub}.

Observe that T serves as a lower bound for $\phi(E - E_{sub})$ (i.e., $\phi(E - E_{sub}) \geq T$). Assume that a Dijkstra execution in Line 6 finds a shortest path sp. If the total cost of sp is less than $T - \tau_{max}$ (Line 7)[2], then it is also less than $\phi(E - E_{sub}) - \tau_{max}$. In this case, sp is *valid* according to Theorem 1; i.e., it is a shortest path in the entire E too. Thus, we augment sp, updating potential values and sp edges in the graph (Lines 8-10) as in the basic SSPA technique. Otherwise (i.e., sp cost is higher than $T - \tau_{max}$), the sp is not valid and is not augmented; RIA performs new range searches with an

[2]To clarify the condition in Line 7, the total cost of sp is by definition equal to $v_{min}.\alpha$, since $c(v_{min}, t)$ is always 0.

extended T in order to insert more edges into E_{sub} (Lines 12-15). Specifically, we extend T by θ and execute an *annular range search* for each point $q_i \in Q$, so that points of P within the distance range $(T-\theta, T]$ from q_i are identified (and the corresponding edges are inserted into E_{sub}). Then, RIA resumes from the iteration it stopped. RIA continues this way and terminates when $\gamma = \min\{|P|, \sum_{q\in Q} q.k\}$ valid shortest paths are found in total. It can be easily shown that the RIA matching is identical to that of SSPA, which considers the entire E.

Algorithm 2 Range Incremental Algorithm (RIA)

```
    algorithm RIA(Set Q, Set P, Value θ)
 1: T:=θ; τ_max:=0; E_sub:=∅
 2: for all q_i ∈ Q do
 3:     P'=Range-Search(q_i,T)
 4:     insert edge e(q_i, p_j) into E_sub, for each p_j ∈ P'
 5: for loop:=1 to γ do
 6:     v_min:=Dijkstra(Q, P, E_sub)
 7:     if v_min.α ≤ T − τ_max then
 8:         v:=ReverseEdges()
 9:         UpdatePotentials()
10:         τ_max:=max {q_i.τ | q_i ∈ Q}        ▷ the highest potential
11:     else
12:         loop--; T:=T + θ
13:         for all q_i ∈ Q do
14:             P'=Annular-Range-Search(q_i,T − θ,T)
15:             insert edge e(q_i, p_j) into E_sub, for each p_j ∈ P'
```

3.2 Nearest Neighbor Incremental Algorithm

RIA constrains the search on a small edge set E_{sub} by using system parameter θ. However, it is hard to fine-tune θ or derive it analytically. When θ is too large, set E_{sub} grows, leading to long computation time. In case θ is too small, RIA performs numerous range searches, incurring high I/O cost. To tackle this problem, we develop a *Nearest Neighbor Incremental Algorithm* (NIA), which performs incremental nearest neighbor search [7] to expand edge set E_{sub}. Algorithm 3 is the pseudo-code of NIA. We use a min-heap H, that organizes encountered edges in ascending cost order. Specifically, we first compute for each point $q_i \in Q$ its nearest neighbor p_j in P and insert the corresponding edge $e(q_i, p_j)$ into H. In each loop, NIA de-heaps the shortest edge $e(q_i, p_j)$ from H and inserts it into E_{sub} (Lines 7, 8). Then, it computes the next nearest neighbor of q_i and inserts the corresponding edge into H (Lines 9, 10). Next, it computes the shortest path sp in the new E_{sub}.

Due to the min-heap ascending ordering and the incremental nearest neighbor search, it is guaranteed that the top edge in H has the minimum weight of edges in $E - E_{sub}$. Letting $TopKey(H)$ be the key (i.e., length) of the top entry in H, it holds that (i) E_{sub} is a distance-bounded edge set and (ii) $\phi(E - E_{sub}) = TopKey(H)$. From Theorem 1 it follows that if the cost of sp (i.e., $v_{min}.\alpha$) is no greater than $TopKey(H) - \tau_{max}$, then sp is a valid shortest path and is thus augmented into the graph.

Otherwise (i.e., if the sp cost is larger than $TopKey(H) - \tau_{max}$), sp is invalid and ignored. In this case, NIA de-heaps the top edge $e(q_i, p_j)$ from H and inserts it into E_{sub}. For the q_i node of the de-heaped edge, NIA finds its next nearest neighbor in P. Letting p_m be this neighbor, edge $e(q_i, p_m)$ is inserted into H (with key equal to its length). A new shortest path is computed in the expanded E_{sub} and the procedure is repeated; the current iteration is considered complete when a valid shortest path is computed and augmented. Overall, NIA terminates after γ completed iterations (equivalently, after augmenting γ valid shortest paths).

Algorithm 3 Nearest Neighbor Incremental Algorithm (NIA)

```
    algorithm NIA(Set Q, Set P)
 1: H:=new min-heap
 2: τ_max:=0; E_sub:=∅
 3: for all q_i ∈ Q do
 4:     p_j:=NN of q_i in P
 5:     insert ⟨e(q_i, p_j), dist(q_i, p_j)⟩ into H
 6: for loop:=1 to γ do
 7:     de-heap the top entry ⟨e(q_i, p_j), dist(q_i, p_j)⟩ from H
 8:     insert edge e(q_i, p_j) into E_sub
 9:     p_m:=next NN of q_i in P
10:     insert ⟨e(q_i, p_m), dist(q_i, p_m)⟩ into H
11:     v_min:=Dijkstra(Q, P, E_sub)
12:     if v_min.α ≤ TopKey(H) − τ_max then
13:         v:=ReverseEdges()
14:         UpdatePotentials()
15:         τ_max:=max {q_i.τ | q_i ∈ Q}        ▷ the highest potential
16:     else
17:         loop--                              ▷ Invalid path; go to Line 7
```

3.3 Incremental On-demand Algorithm

In this section, we present the *Incremental On-demand Algorithm* (IDA), which improves on NIA by pruning more edges and accelerating sp computations. IDA is based on the concept of *full* service providers and *full* customers.

DEFINITION 2. *A service provider $q_i \in Q$ is said to be* full *when edge $e(s, q_i)$ has already been used $q_i.k$ times in previous (valid) shortest paths.*

For a full q_i, since $e(s, q_i)$ (with a fixed cost 0) has reached its capacity, Dijkstra's algorithm can no longer pass through this edge. In other words, the shortest path from s to q_i can no longer be this edge and, thus, $q_i.\alpha$ (i.e., the minimum cost from s to q_i) may be greater than 0. This fact is exploited by IDA, which leads to a more effective pruning of edges incident to q_i.

IDA uses an edge heap H just like NIA. Unlike NIA, where the key of the edges in H is their length $dist(q_i, p_m)$, in IDA the key of an edge $e(q_i, p_m)$ is $q_i.\alpha + dist(q_i, p_m)$. The rationale is that if q_i is full, any sp going through q_i should have cost at least $q_i.\alpha + dist(q_i, p_m)$. This leads to earlier termination and smaller E_{sub}, since edges reachable through full service providers are not de-heaped (and, thus, not inserted into E_{sub}) unnecessarily early.

As $q_i.\alpha$ varies, whenever some Dijkstra execution visits a full $q_i \in Q$ and updates $q_i.\alpha$ to a new value, IDA accordingly updates the key of its corresponding edge $e(q_i, p_j)$ in H to the new $q_i.\alpha + dist(q_i, p_j)$. Note that (in both NIA and IDA) for every $q_i \in Q$ there is exactly one edge in H from q_i to some $p_j \in P$ at all times. It is easy to show the correctness of IDA, after replacing $\phi(E - E_{sub})$ by $\Phi(E - E_{sub})$ in Theorem 1. $\Phi(E - E_{sub})$ models the minimum possible cost an sp could have if it passed through some edge in $E - E_{sub}$.

Similar to full service providers, IDA also exploits the properties of *full* customers to improve the running time and, specifically, to accelerate shortest path computations. Below we formally define full customers and provide a theorem that allows sp retrieval without invoking Dijkstra's algorithm.

DEFINITION 3. *A customer $p_j \in P$ is said to be* full *when edge $e(p_j, t)$ has already been used in a previous (valid) shortest path.*

THEOREM 2. *If no $q \in Q$ is full, then the shortest path (between source s and sink t) passes through a single edge $e(q_i, p_j)$; i.e., $sp = \{e(s, q_i), e(q_i, p_j), e(p_j, t)\}$, where $q_i \in Q$, $p_j \in P$. Furthermore, $e(q_i, p_j)$ is the shortest edge in E_{sub} with a non-full p_j.*

PROOF. Since no $q \in Q$ is full, all $q \in Q$ are inserted into the Dijkstra heap and visited (with cost $q.\alpha = 0$) before any $p \in P$. Therefore, after de-heaping the first $p_j \in P$, and if p_j is full, Dijkstra cannot return to any $q \in Q$. As a result, the current sp must be passing through exactly one edge $e(q_i, p_j)$ (with a non-full p_j) followed by $e(p_j, t)$, i.e., $sp = \{e(s, q_i), e(q_i, p_j), e(p_j, t)\}$. Since q_i and p_j are non-full, $w(s, q_i) = w(p_j, t) = 0$ and the sp cost is $w(q_i, p_j)$.

It remains to show that the cost order among edges $e(q, p) \in E_{sub}$ with non-full p coincides with their length order. As described in Section 2.2, $w(q, p) = dist(q, p) - q.\tau + p.\tau$. Note that a node $p \in P$ becomes full when Dijkstra's algorithm visits it for the first time. Equivalently, all non-full ones have never been visited by Dijkstra's algorithm and their potentials remain 0 since the initialization of the problem. As a result, $p.\tau = 0$, and $w(q, p) = dist(q, p) - q.\tau$. Also, the fact that all $q \in Q$ are non-full leads to their potentials being updated in every IDA iteration to the same exact value (in Line 9 in Algorithm 1). Thus, the cost order among edges with non-full p coincides with their distance order. □

According to the above theorem, as long as no service provider $q \in Q$ is full, IDA computes the current sp, without invoking Dijkstra's algorithm, by iteratively de-heaping edges $e(q_i, p_j)$ from H[3]. If p_j is full, we directly insert it into E_{sub} and de-heap the next entry; otherwise we report $sp = \{e(s, q_i), e(q_i, p_j), e(p_j, t)\}$. Note that after de-heaping any edge $e(q_i, p_j)$ from H, we en-heap the edge from q_i to its next nearest customer (as in Lines 7-10 of Algorithm 3).

Algorithm 4 is the pseudo-code of IDA. Lines 1-5 initialize E_{sub} identically to NIA. At Line 9 we compute the current sp. Note that if no service provider is full, we derive sp using Theorem 2 and the method described above (we omit this enhancement from the pseudo-code for readability). At Lines 10-12, if the last sp computation visited some full $q \in Q$ and altered its $q.\alpha$ value, then we accordingly update the key of its corresponding edge $e(q, p)$ in H to the new $q.\alpha + dist(q, p)$ (Line 12). Lines 13-14 retrieve the next NN of q_i (q_i refers to $e(q_i, p_j)$ de-heaped at Line 7) and insert the corresponding edge into H. Note that we perform this after updating the $q.\alpha$ values at Lines 10-12 so that the en-heaped edge has an up-to-date key.

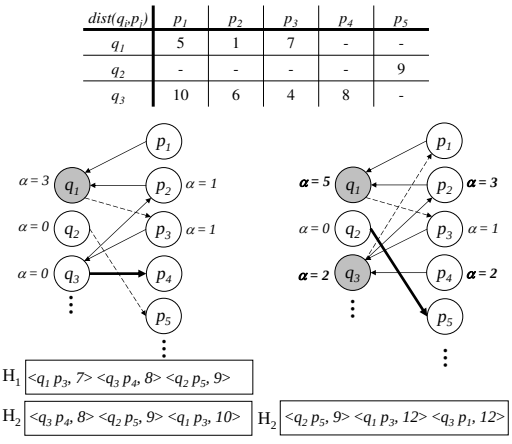
Example: Consider the example in Figure 4(a), where the table at the top illustrates the lengths of all encountered edges (i.e., edges in E_{sub} and in the heap). The flow graph shown skips the source and sink for clarity and includes only edges between service providers and customers. Service provider q_1 (shown shaded) is full with $q_1.\alpha = 3$. Dashed edges $e(q_1, p_3), e(q_2, p_5)$ and the bold one $e(q_3, p_4)$ have been en-heaped but not yet inserted into E_{sub}. At the bottom, H_1 and H_2 illustrate the heap contents in NIA and IDA, respectively, assuming that so far they proceeded identically.

[3]While no q is full, all keys in H are equal to the corresponding edge lengths.

Algorithm 4 Incremental On-demand Algorithm (IDA)

```
   algorithm IDA(Set Q, Set P)
 1: H:=new min-heap
 2: τ_max:=0; E_sub:=∅
 3: for all q_i ∈ Q do
 4:     p_j:=first NN of q_i in P
 5:     insert ⟨e(q_i, p_j), dist(q_i, p_j)⟩ into H
 6: for loop:=1 to γ do
 7:     de-heap ⟨e(q_i, p_j), key⟩ from H
 8:     insert e(q_i, p_j) into E_sub
 9:     v_min:=Dijkstra(Q, P, E_sub)
10:     for all visited q ∈ Q do
11:         if q is full and q.α changed in Line 9 then
12:             update q.α in H
13:     p_m:=next NN of q_i in P
14:     insert ⟨e(q_i, p_m), q_i.α + dist(q_i, p_m)⟩ into H
15:     if v_min.α ≤ TopKey(H) − τ_max then
16:         v:=ReverseEdges()
17:         UpdatePotentials()
18:         τ_max:=max {q_i.τ | q_i ∈ Q}        ▷ the highest potential
19:     else
20:         loop--                              ▷ Invalid path; go to Line 7
```

Their difference is the key of $e(q_1, p_3)$, which is 7 in NIA and 10 in IDA (since $dist(q_1, p_3) = 7$ and $q_1.\alpha = 3$). This leads to a different insertion order into E_{sub} and a faster IDA termination. For the current sp to be valid, in Line 12 of Algorithm 3 (in Line 15 of Algorithm 4), NIA (IDA) requires that its cost is no greater than 7-τ_{max} (8-τ_{max}), where 7 (8) is the $TopKey(H_1)$ value ($TopKey(H_2)$, respectively). This implies that the current IDA iteration has higher chances to terminate without needing to insert new edges and re-invoke Dijkstra's algorithm.



(a) IDA versus NIA (b) Key update

Figure 4: Utilizing full service providers in IDA

Let us now focus on IDA. Since the top edge in H_2 is $e(q_3, p_4)$ (shown bold), we insert it into E_{sub}. Figure 4(b) shows the new flow graph, assuming that the subsequent Dijkstra execution returned an sp passing through $e(q_3, p_4)$. Assuming that $q_3.k = 2$, augmenting this sp makes q_3 full with $q_3.\alpha = 2$, and alters $q_1.\alpha$ to 5. Since $q_1.\alpha$ has changed, IDA updates the key of $e(q_1, p_3)$ in H_2 to $dist(q_1, p_3) + q_i.\alpha = 12$. Then, we find the next NN of q_3 (i.e., p_1) and insert the corresponding edge $e(q_3, p_1)$ into H_2 with key $q_3.\alpha + dist(q_3, p_1) = 12$. The bold edge (i.e., $e(q_2, p_5)$) is the one to be inserted next into E_{sub}.

3.4 Optimizations

In this section we describe two enhancements that apply to NIA and IDA. Section 3.4.1 proposes a technique that accelerates Dijkstra's algorithm by reusing its previous computations. Section 3.4.2 presents an incremental all nearest neighbor (ANN) search that reduces the I/O cost.

3.4.1 Reducing Dijkstra Executions

Unlike the bulk discovery and insertion of edges (through range search) in RIA, NIA/IDA apply incremental NN search to discover the edges one-by-one, keeping E_{sub} small. However, since E_{sub} expands slowly, NIA/IDA may perform numerous Dijkstra executions. To accelerate processing, we reduce the cost of Dijkstra executions in NIA/IDA by reusing (i) the $v_i.\alpha$ values computed in the previous *sp* computation and (ii) utilizing the entries that remained inside the Dijkstra heap upon termination. Assume that in the current NIA/IDA iteration some (invalid) *sp* has been computed, and that we need to find a new *sp* after inserting a new edge $e(q,p)$ into E_{sub} (in Line 8 of Algorithm 3 or Algorithm 4, respectively). Let H_d be the Dijkstra search heap after last *sp* computation.

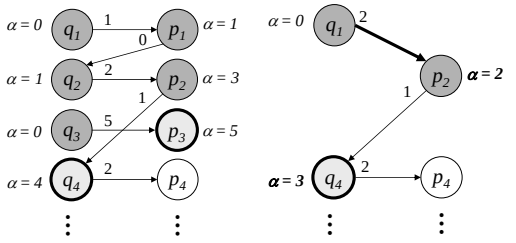
Our objective is (i) to identify the visited nodes v whose $v.\alpha$ value is affected by $e(q,p)$ (i.e., $e(q,p)$ leads to a shortest path from the sink to v) and, eventually, (ii) to update the keys of nodes inside H_d. This is performed by the *Path Update Algorithm* (PUA) to be described shortly. Upon termination of PUA, a new Dijkstra execution is performed, which however directly uses the updated H_d and avoids visiting nodes de-heaped in previous *sp* computation(s) in the current NIA/IDA iteration.

PUA initializes an empty min-heap H_f to play the role of a Dijkstra-like search heap among previously visited nodes. H_f organizes its entries (nodes) in ascending order of their α values. First, we insert into H_f the q node of the new edge $e(q,p)$. Next, we iteratively de-heap the top node v_i from H_f and examine whether nodes v_j connected to v_i can be reached through a shortest path via v_i. In particular, if $v_j.\alpha > v_i.\alpha + w(v_i,v_j)$ then $v_j.\alpha$ is updated to $v_i.\alpha + w(v_i,v_j)$ and $v_j.prev$ is set to v_i (to indicate that v_j is now reachable via v_i). If v_j is in H_d or H_f, its key is updated to $v_j.\alpha$ in its containing heap. Otherwise (i.e., if v_j is neither in H_d nor H_f), it is inserted into H_f with key $v_j.\alpha$. PUA terminates when H_f becomes empty. Algorithm 5 presents PUA.

Algorithm 5 Path Update Algorithm (PUA)

```
   algorithm PUA(Set Q, Set P, Heap H_d, Edge set E_sub,
   Edge e(q,p))
 1: H_f:=new min-heap
 2: insert ⟨q, q.α⟩ into H_f
 3: while H_f is not empty do
 4:     de-heap top node v_i (with the lowest v_i.α value) from H_f
 5:     for all edges e(v_i, v_j) ∈ E_sub outgoing from v_i do
 6:         if v_j.α > v_i.α + w(v_i, v_j) then
 7:             v_j.α:=v_i.α + w(u, v); v_j.prev:=v_i
 8:             if v_j ∈ H_d then
 9:                 update v_j.α in H_d
10:             else if v_j ∈ H_f then
11:                 update v_j.α in H_f
12:             else
13:                 insert ⟨v_j, v_j.α⟩ into H_f
```

Example: We illustrate the PUA technique with an example. Figure 5(a) shows the current E_{sub} edges between (some nodes of) sets Q and P, the α values of these nodes, and the edge costs (numbers above each edge) after the last Dijkstra execution. The visited nodes are illustrated shaded, while the nodes remaining in H_d are q_4 and p_3 (having bold borders and lighter gray color). Consider that edge $e(q_1,p_2)$ with cost $w(q_1,p_2)=2$ is inserted into E_{sub}. Figure 5(b) shows the new edge (in bold) and the PUA steps. First, q_1 is inserted into H_f with key $q_1.\alpha = 0$. Its de-heaping leads to adjacent node p_2 which is reachable with a lower cost (than the current $p_2.\alpha$) via q_1. Thus, p_2 is inserted into H_f with key equal to the new $p_2.\alpha = q_1.\alpha + w(q_1,p_2) = 2$. Similarly, the de-heaping of p_2 leads to updating the key of q_4 in H_d to the new $q_4.\alpha = 3$. After these changes, the new *sp* can be computed by directly using $H_d = \{\langle q_4,3\rangle, \langle p_3,5\rangle\}$ in the new Dijkstra execution. Note that the shortest paths to (and, accordingly, the α values of) q_2, q_3, p_1, p_3 have not been affected by the insertion of $e(q_1,p_2)$ and the new *sp* computation avoids unnecessary costs for them.



(a) Previous state (b) Insertion of $e(q_1,p_2)$

Figure 5: Example of PUA

PUA can utilize results only among Dijkstra executions taking place as part of the same NIA/IDA iteration. The reason why reusing cannot span multiple iterations is that *sp* augmentation (which signals the end of an iteration) alters many edges, by reversing their directions and modifying their costs. Another important remark is that IDA uses the above PUA-based optimization only after some of the service providers become full, because until then shortest paths are computed using Theorem 2 directly.

3.4.2 Incremental ANN Processing

Our CCA algorithms invoke numerous NN search operations around the service providers to the R-tree R_P that indexes the customers P. To reduce the I/O cost, we employ an incremental *all-nearest-neighbors* technique. First, we form service provider groups G_m based on their Hilbert space-filling curve ordering. For each group G_m we maintain a min-heap H_m that organizes encountered R-tree entries e in ascending $mindist(MBR(G_m), MBR(e))$ order. For every $q_i \in G_m$ we maintain a candidate min-heap res_i that orders all encountered customers (i.e., candidate NNs) in ascending distance from q_i.

When we need to compute the (next) NN of some $q_i \in G_m$, we iteratively de-heap and visit the top R-tree entry in H_m. If the de-heaped entry is a point $p \in P$, we insert it into the candidate min-heap of every service provider in G_m. The procedure terminates when the top candidate p_j in res_i has key smaller than or equal to the top entry in H_m; i.e., $dist(q_i,p_j)$ is smaller than or equal to $mindist(MBR(G_m), MBR(e))$ for every unvisited R-tree entry e. At that point, we de-heap p_j from res_i and report it as the (next) NN of q_i. Algorithm 6 is a pseudo-code for the above procedure.

Algorithm 6 Incremental ANN Search

```
algorithm ANN(Group G_m, R-tree R_P, Service provider q_i)
while top entry in res_i has key > key of top entry in G_m do
    de-heap top entry e from H_m
    if e is an directory entry of R-tree R_P then
        visit node pointed by e and insert its entries into H_m
    else                          ▷ E is a leaf level entry, i.e., a point p ∈ P
        for all q_k in G_m do
            insert ⟨p, dist(p, q_k)⟩ into res_k
de-heap top entry ⟨p_j, dist(p_j, q_i)⟩ from res_i
return p_j as the next NN of q_i
```

4. APPROXIMATE METHODS

Time-critical applications may favor fast answers over exact ones. This motivates us to develop approximate CCA solutions. In this section, we propose a methodology that provides a tunable trade-off between result accuracy and response time, and comes with theoretical guarantees for the assignment cost.

Our general approach consists of three phases. The first one is the *partitioning phase*, in which we form groups G_m of either the points in Q or points in P, so that the diagonal of their MBR does not exceed a threshold δ. Parameter δ is used to control the quality of the assignment; the smaller δ is the better the computed matching approximates the optimal. The second phase, called *concise matching*, solves optimally a small CCA problem extracting one representative point per group G_m and using the set of representatives as the set of service providers (customers). Finally, the *refinement phase* uses the assignment produced in the previous step to derive a matching on the entire sets P and Q.

Sections 4.1 and 4.2 describe two methods, called *Service Provider Approximation* (SA) and *Customer Approximation* (CA). SA and CA follow different approaches for partitioning and subsequent concise matching. Specifically, SA groups the service providers and solves concise matching in the entire P, while CA groups the customers and performs concise matching in the entire Q.[4] Section 4.3 describes refinement techniques that could be used with either SA or CA. Finally, Section 4.4 provides error bounds for both approaches.

4.1 Service Provider Approximation

Partitioning in SA is performed on set Q. The points $q \in Q$ are sorted according to their Hilbert values and processed in this order. We start with zero service provider groups. Each point q, in turn, is inserted into an existing group G_m so that the diagonal of G_m's MBR does not exceed δ. If no such group is found, then a new group is formed to include q. The process is repeated until all $q \in Q$ are grouped.

We proceed to concise matching by extracting one representative point per group. The representative point $\overline{g_m}$ of a group G_m has capacity $\overline{g_m.k} = \sum_{q \in G_m} q.k$ and is located at its *geometric centroid*; each coordinate of $\overline{g_m}$ is equal to the weighted average of points inside G_m. Weighting is performed according to the capacities $q.k$ of points $q \in G_m$, e.g., the x-coordinate $\overline{g_m}.x$ of $\overline{g_m}$ is $\frac{1}{\sum_{q \in G_m} q.k} \sum_{q \in G_m} (q.x \cdot q.k)$.

Figure 6 shows a scenario where $Q = \{q_1, ..., q_{10}\}$. Assume that SPP produces the illustrated groups G_1, G_2, G_3 according to parameter δ. The dashed lines correspond to group MBR diagonals and their lengths cannot be longer than δ. The representatives of these groups are shown as gray points $\overline{g_1}, \overline{g_2}$ and $\overline{g_3}$. Assuming that all $q \in Q$ have capacity $q.k = 2$, then the representative capacities are $\overline{g_1}.k = 6, \overline{g_2}.k = 6, \overline{g_3}.k = 8$.

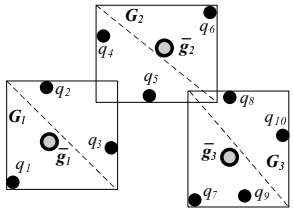


Figure 6: Service provider partitioning

The resulting representatives form set Q' which is used as an approximation of Q. The concise matching of SA solves an exact CCA problem over Q' and P. This step is performed by the IDA algorithm described in Section 3.3, because (as will be demonstrated by our experiments) it is the most efficient among the exact methods. The matching M' produced by this step will be refined into the final matching M using one of the techniques presented in Section 4.3.

4.2 Customer Approximation

CA is similar to SA, but groups customers instead of service providers. Recall that P is indexed by an R-tree. We first initialize a set S of customer groups to $\emptyset$. Given parameter δ, we traverse the R-tree. Starting from the root entries, we compare the MBR diagonal of each of them with δ. If the diagonal of entry e is smaller than or equal to δ, we insert it into S (the corresponding group of customers are those in the subtree rooted at e). Otherwise (i.e., e's diagonal is larger than δ), we visit the corresponding node and recursively repeat this procedure for its entries.

R-tree leaves are an exception to this procedure. In particular, if δ is small, it is possible that we reach an entry e corresponding to an R-tree leaf whose diagonal is larger than δ. An option would be to insert into S all points in e, but this would result in a large S. Thus, we handle e as follows. We conceptually split its MBR into two equal halves on its longest dimension. We repeat this process until the diagonal of each partition becomes smaller than or equal to δ. Then, we insert the resulting conceptual entries into S.

Upon termination of the above procedure, all entries in S have diagonal smaller than δ and the union of points in their subtrees is the entire P. The size of S, however, can be reduced (without violating the δ constraint) by an extra step that merges its contents. Specifically, we use a procedure similar to SA and group entries in S into conceptual hyper-entries whose diagonal does not exceed δ.

Let S be the final set of entries (conceptual or not). We produce a set P' of customer representatives as follows. For each $e \in S$ we derive a representative point $\overline{g}$ located at the geometric centroid of e. The representative has *weight* $\overline{g.w}$ equal to the number of points in the subtree of e.

To exemplify CA partitioning, assume that the R-tree of P and parameter δ are as shown in Figure 7 (the R-tree is illustrated both in the spatial domain and as stored on the disk). We first access the root, and consider its entries e_1 and e_2. Entry e_2 has smaller diagonal than δ and is inserted

[4]We note here that we attempted to combine SA and CA (i.e., to group both Q and P), but this led to a very poor matching. Thus, we omit this hybrid method.

into S. This is not the case for e_1, whose pointed entries are loaded from the disk. Among e_1's entries, e_4 and e_5 satisfy the diagonal condition and are included in S. On the other hand, e_3 is a leaf and still has diagonal larger than δ. Thus, we conceptually divide it into two new entries on its long dimension (i.e., x dimension). The resulting $e_{3,1}$ and $e_{3,2}$ have small enough diagonal and are placed into S. Entries inserted into S are shown shaded. In the last step, we merge entries into larger ones (while still satisfying the δ condition); e_4 and e_5 form a hyper-entry whose boundaries are shown dashed. Every entry in the final S implicitly defines a group of customers G_m. Set P' contains the representatives of the final entries in S, i.e., $P' = \{\overline{g_1}, \overline{g_2}, \overline{g_3}, \overline{g_4}\}$.

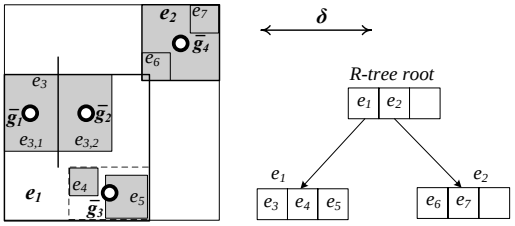


Figure 7: Customer partitioning

In the concise matching phase, CA computes the optimal matching M' between P' and Q. This is performed in main memory (where P' and Q reside) using IDA. Note that in this setting points in P' also have capacities (the representative weights). This is not a problem, since IDA (as well as RIA and NIA) can handle capacities in the customer side of the flow graph too. The difference is that M' may assign "instances" of a representative to multiple service providers.

4.3 Refinement Phase

In both SA and CA, we are given a matching M' between one approximate set (i.e., Q' or P') and one original set (P or Q, respectively). In either case, M' specifies for each group G_m of service providers (customers) which customers (instances of service providers) are assigned to it. In other words, in both SA and CA the refinement phase has to solve several smaller problems of assigning a set of customers P'' to a set of service providers Q'' (where the number of points $p \in P''$ to be assigned to each $q \in Q''$ is given by the concise matching phase). We could run an exact algorithm for each of these smaller problems. This, however, is expensive. Instead, we propose the following two heuristics[5], receiving smalls sets P'' and Q'' as input.

NN-based refinement: This approach computes the (next) NN of each $q \in Q''$ in a round-robin fashion in set P''. When discovering the NN p of service provider q, we include pair (q, p) in the final assignment M and remove p from P''. If q has reached its number of instances to be assigned to P'', we also delete q from Q''.

Exclusive NN refinement: According to this strategy, we identify the $p \in P''$ with the minimum distance from any $q \in Q''$ that has not reached its number of instances to be assigned to P'' (according to M'). We insert into the final assignment M the corresponding pair (q, p) and proceed with the next customer in P''.

[5]We experimented with several other alternatives but these two methods were both efficient and quite accurate.

4.4 Assignment Cost Guarantee

Let M be the matching computed by SA and M_{CCA} be the optimal matching. The *assignment cost error* of M is:

$$Err(M) = \Psi(M) - \Psi(M_{CCA}), \quad (4)$$

where $\Psi(M)$ and $\Psi(M_{CCA})$ are defined as in Equation 1. We show that $Err(M)$ is at most $2 \cdot \gamma \cdot \delta$, where $\gamma = \min\{|P|, \sum_{q \in Q} q.k\}$. Thus, we are able to control the assignment cost error through parameter δ.

THEOREM 3. *The assignment error of SA is upper bounded by* $2 \cdot \gamma \cdot \delta$.

PROOF. Note that approximate matching M has the full size γ, since concise matching leaves customers unassigned only if all service providers are fully utilized (i.e., they have reached their capacity). From the optimal matching M_{CCA}, we derive another matching M'_{CCA} by replacing each pair $(q, p) \in M_{CCA}$ with pair $(\overline{g}, p)$, where $\overline{g}$ is the representative of q's group. After the replacement, the cost of each pair increases/decreases by at most δ (since δ is the maximum possible distance between q and the weighted centroid $\overline{g}$). Thus, $\Psi(M'_{CCA}) \leq \Psi(M_{CCA}) + \gamma \cdot \delta$.

Note that M'_{CCA} is not necessarily the optimal matching between Q' (i.e., the set of service provider representatives) and P. Let M' be the optimal matching between Q' and P. We know that $\Psi(M') \leq \Psi(M'_{CCA})$. Combining the two inequalities, we derive $\Psi(M') \leq \Psi(M_{CCA}) + \gamma \cdot \delta$.

SA replaces the pairs of M' heuristically to form the final matching M, incuring a maximum error of δ per pair. Hence, $\Psi(M) \leq \Psi(M') + \gamma \cdot \delta$. From the last two inequalities, we infer that $\Psi(M) \leq \Psi(M_{CCA}) + 2 \cdot \gamma \cdot \delta$. □

The assignment error of CA is bounded as follows.

THEOREM 4. *The assignment error of CA is upper bounded by* $\gamma \cdot \delta$.

PROOF. The proof follows the same lines as that of SA, the difference being that the maximum possible distance between a customer p and its group representative $\overline{g}$ is $\frac{\delta}{2}$ (since $\overline{g}$ is always the geometric centroid of p's group MBR). □

5. EXPERIMENTS

This section empirically evaluates the performance of our algorithms. All methods were implemented in C++ and experiments were performed on a Pentium D 3.0GHz machine, running on Ubuntu 7.10. Section 5.1 describes the datasets, the parameters under investigation, and other settings used in our evaluation. In Section 5.2 we study the performance of our algorithms on optimal CCA computation. Section 5.3 explores the efficiency and assignment cost error of our techniques on approximate CCA computation.

5.1 Data Generation and Problem Settings

The CCA problem takes two spatial datasets as input: the service provider set Q and the customer set P. Both datasets were generated on the road map of San Francisco (SF) [3], using the generator of [15]. In particular, the points fall on edges of the road network, so that 80% of them are spread among 10 dense clusters, while the remaining 20% are uniformly distributed in the network. This dataset selection simulates a real situation where some parts of the city are denser than others. To establish the generality of our

methods, we also present results for different distributions. All datasets are normalized to lie in a $[0, 1000]^2$ space.

By default, the capacity k of all $q \in Q$ is 80 and the dataset cardinalities are $|Q|$=1K and $|P|$=100K. Parameter θ of RIA is fine-tuned (and set to 0.8), for fairness in the comparison with NIA and IDA. Table 2 shows the parameters under investigation. We assume that the service provider dataset Q is small enough to fit in main memory. Each P dataset is indexed by an R-tree with 1Kbyte page size. We use an LRU buffer with size 1% of the tree size. We record the memory usage (i.e., $|E_{sub}|$, number of edges in the subgraph) and the CPU time. Also, we measure I/O time by charging 10ms per page fault [10].

Parameter	Default	Range
$\|Q\|$ (in thousands)	1	0.25, 0.5, 1, 2.5, 5
$\|P\|$ (in thousands)	100	25, 50, 100, 150, 200
Capacity k	80	20, 40, 80, 160, 320
Diagonal δ	SA: 40, CA: 10	10, 20, 40, 80, 160

Table 2: System parameters

5.2 Experiments on Optimal Assignment

SSPA requires that the complete flow graph is stored in main memory (as described in Section 2.2). For our default setting this leads to space requirements that exceed several times the available system memory. To provide, however, an intuition about (i) the inherent complexity of the problem and (ii) the relative performance of SSPA versus our algorithms, we experiment on a smaller problem; we generate P and S as described in Section 5.1, with $|Q| = 250$ and $|P| = 25$K, so that the flow graph fits in main memory. For RIA, NIA, and IDA, P is indexed by a memory-based R-tree. SSPA does not utilize an index, as it involves no spatial searches. Figure 8 shows the CPU time (in logarithmic scale) versus capacity k in this small problem. Our methods are one to three orders of magnitude faster than SSPA. We postpone the explanation of the observed trends for Figure 9 (with disk-resident P), but stress the excessive time requirements of SSPA and the efficiency of our methods.

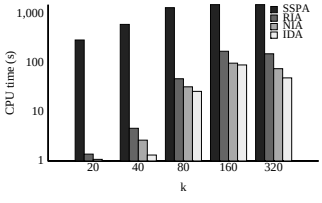
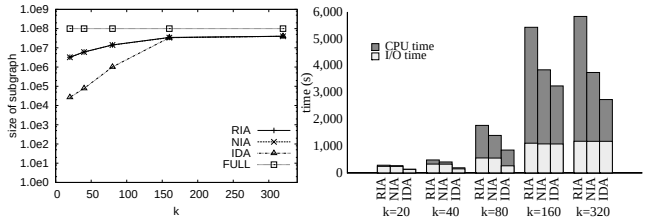


Figure 8: CPU time vs. k, $|Q| = 250$, $|P| = 25$K

In the remaining experiments, we focus on disk-based P and large problem instances, excluding the inapplicable SSPA. Figure 9(a) shows the subgraph size E_{sub} as a function of k (setting $|Q|$ and $|P|$ to their default values). We include the complete bipartite graph size $|E_{FULL}| = |Q| \cdot |P|$ as a reference (indicated by FULL). Due to the application of Theorem 1, our algorithms (RIA, NIA, IDA) use/store only a fragment of the complete bipartite graph. IDA explores fewer edges than RIA and NIA for small values of k. The reason behind this is that for $k \cdot |Q| < |P|$, providers are likely to become full early and the tighter bounds of IDA over NIA/RIA can be effectively utilized. On the other hand, if $k \cdot |Q| > |P|$, few or no providers become full, so IDA does not achieve additional pruning compared to NIA/RIA.

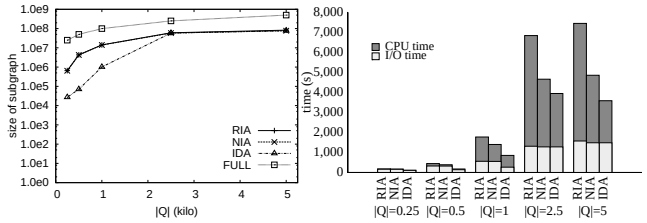


(a) $|E_{sub}|$ (b) total time

Figure 9: Performance vs. k, $|Q| = 1$K, $|P| = 100$K

Figure 9(b) shows the total execution time in the previous experiment, and breaks it into I/O and CPU cost. The I/O time depends primarily on (and thus follows the increasing trend of) $|E_{sub}|$. The CPU time also rises with k, since the flow graph size and the number of iterations γ increase with k. For large k values, however, the increase for RIA is not as steep, while for INA and NIA the CPU cost drops slightly. This happens because the capacity constraint is looser and, essentially, the problem becomes easier. NIA has lower CPU time than RIA because NIA adds new edges one-by-one and keeps the subgraph small. Note that even for large k (where the final $|E_{sub}|$ is similar for RIA and NIA), the early iterations of NIA run on a smaller E_{sub} which increases only towards its final iterations. On the other hand, IDA is faster than NIA because (i) Theorem 2 computes the first assignments fast and (ii) the utilization of full service providers (i.e., with non-zero $q_i.\alpha$ values) avoids unnecessary edge insertions into E_{sub} and leads to earlier termination.

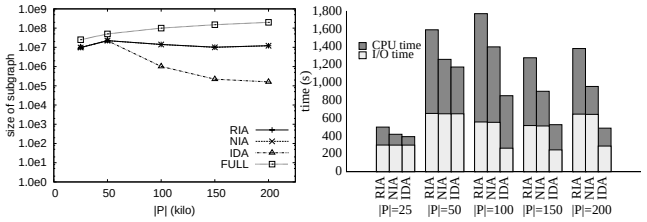
The next experiment investigates the effect of service provider cardinality $|Q|$ (in Figure 10). In general, the relative performance of the algorithms is consistent with our observations in Figure 9; IDA prunes more edges than NIA/RIA when $k \cdot |Q| < |P|$. The cost of the problem increases with $|Q|$, but saturates when $k \cdot |Q| > |P|$, since the optimal assignment is found before long edges (from service providers to their furthest neighbors) are examined.



(a) $|E_{sub}|$ (b) total time

Figure 10: Performance vs. $|Q|$, $k = 80$, $|P| = 100$K

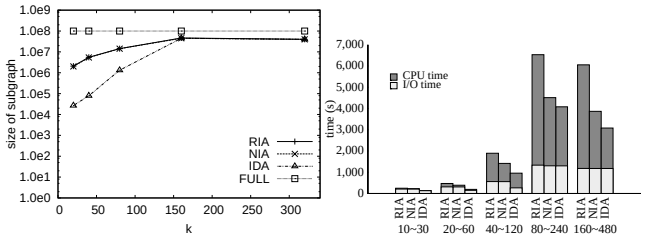
Figure 11 investigates the effect of $|P|$. When $|P|$ increases, the complete flow graph grows but the subgraph explored by our algorithms shrinks. Intuitively, if there are too many customers, the NNs of each service provider are closer, and stand a higher chance to be assigned to it; i.e., the problem becomes easier and fewer E_{sub} edges (and, thus, computations) are needed. However, for $|P| = 200$K the customer R-tree has one more level than smaller cardinalities, incurring more I/Os and a higher overall cost. Note that the difference of IDA from RIA/NIA grows as $|P|$ becomes larger compared to $k \cdot |Q|$ (for the reasons mentioned earlier).



(a) $|E_{sub}|$ (b) total time

Figure 11: Performance vs. $|P|$, $k = 80$, $|Q| = 1K$

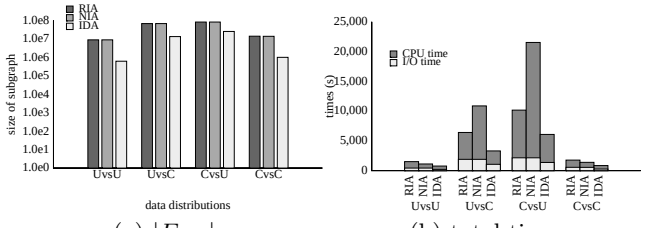
So far we assumed that all service providers have equal capacities $q.k$. Figure 12 compares the algorithms for problems where the providers have different k, taken randomly from the ranges shown as labels on the horizontal axis. The results are similar to those in Figure 9; i.e., mixed k values do not affect the effectiveness of our pruning techniques.



(a) $|E_{sub}|$ (b) total time

Figure 12: Perf. for mixed k, $|Q| = 1K$, $|P| = 100K$

Figure 13 compares the algorithms when Q and P follow varying distributions; *uniform* (U) places points uniformly in the SF network, while *clustered* (C) generates datasets in the way described in Section 5.1. For example, label "UvsC" on the horizontal axis corresponds to uniform service providers and clustered customers. We observe that the cost for computing the optimal assignment increases considerably when the two sets are distributed differently. If Q is uniform and P is clustered (e.g., customers gather in central squares during New Year's Eve), some providers are far from their nearest customer clusters and compete for points far from them, thus increasing the size of the examined subgraph. If Q is clustered and P is uniform (e.g., service providers concentrate around certain regions), the providers cannot fill their capacities with customers near them, and need to expand their search ranges very far. In both cases, NIA is slower than RIA, because the incremental edge retrieval (that is slower than a batch range-based insertion in RIA) is invoked numerous times.



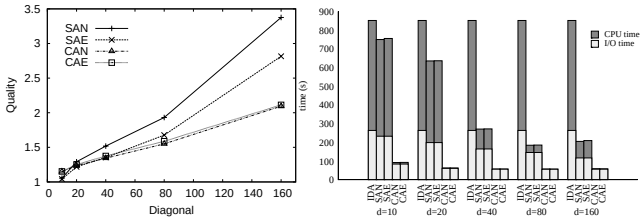
(a) $|E_{sub}|$ (b) total time

Figure 13: Different distributions (default $k, |Q|, |P|$)

5.3 Experiments on Approximate Assignment

In this section, we evaluate the accuracy of our approximate CCA methods (i.e., SA and CA) presented in Section 4, and compare their execution time with IDA (the best exact algorithm). We measure the accuracy of an approximate matching M by $\Psi(M)/\Psi(M_{CCA})$, where M_{CCA} is the optimal assignment. For each of SA and CA, we implemented both the NN-based and exclusive NN refinement techniques (indicated by "N" and "E" after SA or CA in chart labels).

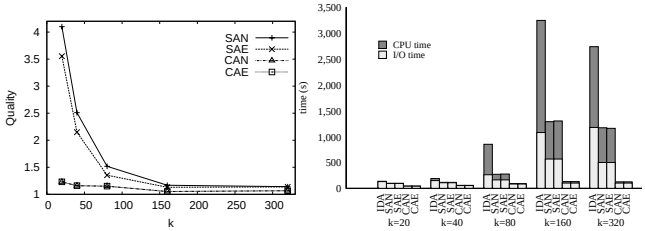
Figure 14 shows the approximation quality and the running time as a function of the diagonal parameter δ (used in the partitioning phase). Observe that the CA variants are significantly better than those of SA in terms of quality and efficiency for all values of δ. An exception is $\delta = 10$ where SA achieves a better approximation, at a cost, however, that is comparable to IDA (since almost every provider forms a group by itself). As expected, accuracy and execution cost drop with δ. CA with as small δ as 10 achieves great performance improvement over IDA, while producing a matching only marginally worse than the optimal.



(a) quality (b) total time

Figure 14: Quality vs. δ (default $k, |Q|, |P|$)

In the remaining experiments, we set δ to 40 for SA, and to 10 for CA, as those values achieve the best efficiency/accuracy trade-off. We evaluate the approximate solutions using the defaults and ranges in Table 2 for k, $|Q|$, and $|P|$. In Figure 15, we vary k and observe that the approximation quality improves with it. As k increases, the providers are assigned more distant customers; i.e., both $\Psi(M)$ and $\Psi(M_{CCA})$ grow. On the other hand, the provider/customer group MBRs remain constant (as δ is fixed) and, hence, the relative error of a suboptimally assigned customer drops. The CA variants are more robust (i.e., less affected by k) than SA, with a 12% error in the default, and 23% in the worst case. The execution time of SA/CA follows the trend of IDA, due to their IDA-based concise matching (but both SA and CA are several times faster).

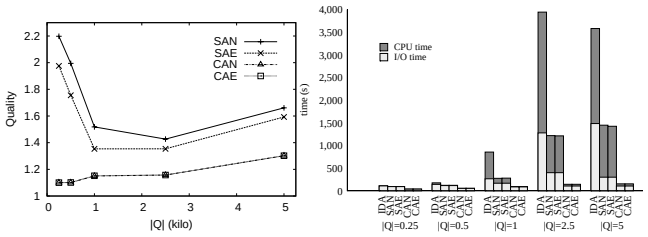


(a) quality (b) total time

Figure 15: Performance vs. k, $|Q| = 1K$, $|P| = 100K$

Figure 16 evaluates the approximation methods for various service provider cardinalities. Again, CA is more accurate than SA, while there are only marginal differences

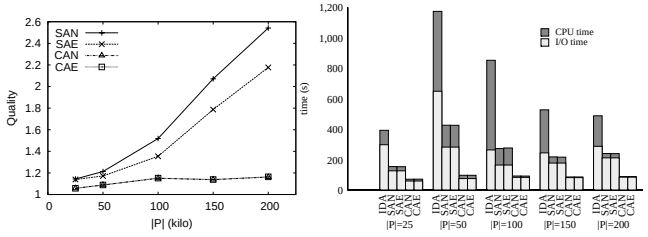
between its CAN and CAE variants. The quality of CA worsens with $|Q|$, because the more service providers around a customer group, the higher the chances for a suboptimal pair in M. On the other hand, in SA the provider groups have a fixed maximum diagonal δ, but their density varies. Very low or very large densities lead to poor approximations.



(a) quality (b) total time

Figure 16: Performance vs. $|Q|$, $k = 80$, $|P| = 100K$

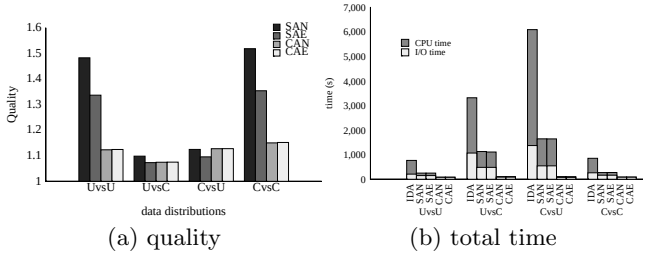
In Figure 17, we investigate the effect of $|P|$. The increase of $|P|$ reduces the accuracy of SA; as the space around every provider group becomes denser with customers, the potential for suboptimal matchings becomes higher. The accuracy of CA is affected to a lesser degree by $|P|$. The slight error increase is because CA groups more customers together, implying a coarser partitioning and worse approximation.



(a) quality (b) total time

Figure 17: Performance vs. $|P|$, $k = 80$, $|Q| = 1K$

Figure 18 compares the approximate methods for different Q and P distributions. CA performs best in terms of running time for all distributions. CA is also more accurate than SA for similarly distributed Q and P (which is the case in most applications). For differently distributed Q and S, the quality of SA and CA is comparable, and close to optimal. To summarize the approximation experiments, CA typically computes a near-optimal matching, while being orders of magnitude faster than IDA.



(a) quality (b) total time

Figure 18: Different distributions (default $k, |Q|, |P|$)

6. CONCLUSION

In this paper, we identify the *capacity constrained assignment* (CCA) problem, which retrieves the matching (between two spatial point sets) with the lowest assignment cost, subject to capacity constraints. CCA is important to applications involving assignment of users to facilities based on spatial proximity and capacity limitations. We present efficient CCA techniques that expand the search space incrementally and effectively prune it. We also develop approximate CCA solutions that provide a trade-off between computation cost and matching quality. According to our experimental results, IDA is the best algorithm for the exact CCA problem, while CA is the method of choice for approximate CCA matching.

In our assumed setting, the set of service providers fits in main memory, while the customers are indexed by a disk-based R-tree. In the future, we plan to extend our framework to the scenario where both sets are disk-resident, incorporating hash-based techniques.

7. REPEATABILITY ASSESSMENT RESULT

All the results in this paper were verified by the SIGMOD repeatability committee. Code and/or data used in the paper are available at:
http://www.sigmod.org/codearchive/sigmod2008/

8. REFERENCES

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

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD*, 1990.

[3] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2):153–180, 2002.

[4] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. In *SIGMOD*, 2000.

[5] A. V. Goldberg and R. Kennedy. An Efficient Cost Scaling Algorithm for the Assignment Problem. *Mathematical Programming*, 71:153–177, 1995.

[6] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.

[7] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[8] J. Munkres. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[9] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*, 1987.

[10] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, fifth edition, 2005.

[11] I. H. Toroslu and G. Üçoluk. Incremental Assignment Problem. *Inf. Sci.*, 177:1523–1529, 2007.

[12] L. H. U, N. Mamoulis, and M. L. Yiu. Continuous Monitoring of Exclusive Closest Pairs. In *SSTD*, 2007.

[13] J. Vygen. *Approximation Algorithms for Facility Location Problems (Lecture Notes)*. University of Bonn, 2004.

[14] R. C.-W. Wong, Y. Tao, A. Fu, and X. Xiao. On Efficient Spatial Matching. In *VLDB*, 2007.

[15] M. L. Yiu and N. Mamoulis. Clustering objects on a spatial network. In *SIGMOD*, 2004.