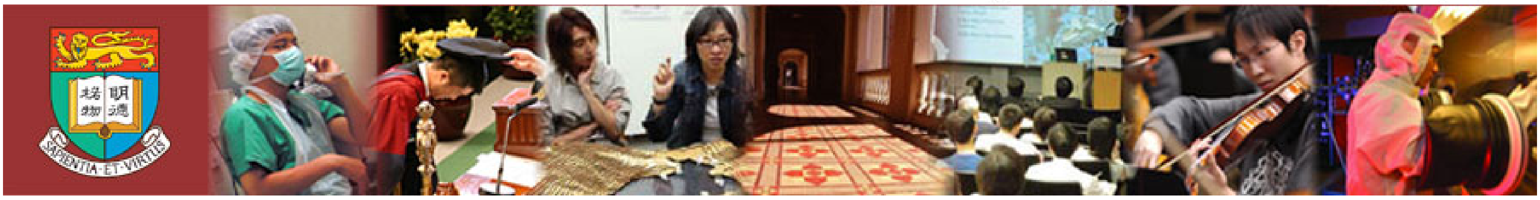


Optimal Matching between Spatial Datasets under Capacity Constraints

LEONG HOU U[1]
University of Hong Kong
and
KYRIAKOS MOURATIDIS[2]
Singapore Management University
and
MAN LUNG YIU[3]
Hong Kong Polytechnic University
and
NIKOS MAMOULIS[1]
University of Hong Kong

Consider a set of *customers* (e.g., WiFi receivers) and a set of *service providers* (e.g., wireless access points), where each provider has a *capacity* and the quality of service offered to its customers is anti-proportional to their distance. The *capacity constrained assignment* (CCA) is a matching between the two sets such that (i) each customer is assigned to at most one provider, (ii) every provider serves no more customers than its capacity, (iii) the maximum possible number of customers are served, and (iv) the sum of Euclidean distances within the assigned provider-customer pairs is minimized. Although max-flow algorithms are applicable to this problem, they require the complete distance-based bipartite graph between the customer and provider sets. 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 incremental techniques that maintain an optimal assignment (in the presence of updates) with a processing cost several times lower than CCA re-computation from scratch. Finally, we present *approximate* (i.e., suboptimal) CCA solutions that provide a tunable 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.

Categories and Subject Descriptors: H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms: Algorithms

Additional Key Words and Phrases: Optimal Assignment, Spatial Databases

Supported by grant HKU 7155/09E from Hong Kong RGC, and by the Research Center, School of Information Systems, Singapore Management University.

Authors' addresses: [1]Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong; email: {hleongu,nikos}@cs.hku.hk; [2]School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902; email: kyriakos@smu.edu.sg; [3]Department of Computing, Hong Kong Polytechnic University, Hung Hom, Hong Kong; email: csmlyiu@comp.polyu.edu.hk.

1. INTRODUCTION

Assume that we want to assign a set of WiFi receivers to a set of wireless access points. An access point can serve up to a maximum number of receivers. A receiver can be assigned to one access point only, and their distance determines the signal strength (and, thus, the quality of service). Ideally, we would like to serve as many receivers as possible, and at the same time minimize the average (or, equivalently, the summed) distance from their access points. Similar problems arise in many resource allocation applications that require a matching between users and facilities based on capacity constraints and spatial proximity. Such a scenario is the assignment of students to schools (with certain capacity each) so that the average 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. In the commercial world, a franchise (e.g., supermarket chain, fast-food chain) could match its outlets (with certain capacities) and the (residential locations of) its employees. The above situations are instances of the *capacity constrained assignment* (CCA) problem.

Formally, the problem input consists of a set of customers P and a set of service providers Q. In addition to spatial coordinates, set Q also includes the capacity $q.k$ of each provider $q \in Q$. A matching $M \subseteq Q \times P$ is said to be *valid* if (i) each provider $q \in Q$ (customer $p \in P$) appears at most $q.k$ times (at most once) in M and (ii) the size of M is maximized (i.e., it contains $\min\{|P|, \sum_{q \in Q} q.k\}$ provider-customer pairs) [Irving et al. 2003]. Among all possible valid matchings, CCA computes a matching M that minimizes the *assignment cost* $\Psi(M)$, defined as:

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

where $dist(q,p)$ denotes the Euclidean distance between q and p. Essentially, the assignment cost determines the quality of a matching; i.e., the output M of CCA achieves the optimal overall quality.

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 customers p_j that fall inside its Voronoi cell (indicated by dashed lines in the figure) leads to the minimum matching cost [Okabe et al. 2000]. However, this approach ignores the service provider capacities. In our example, it assigns 5, 3, and 4 customers 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.

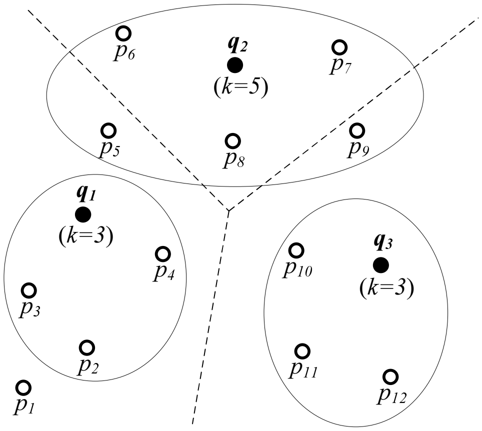


Fig. 1. Spatial assignment example

CCA can be reduced to the well-known *minimum cost flow* (MCF) problem in a complete distance-based bipartite graph between Q and P [Ahuja et al. 1993; Murty 1992]. In the operations research literature [Vygen 2004], there is an abundance of MCF algorithms based on this reduction. These solutions, however, are only applicable to small-sized datasets. In particular, the best of them have a cubic time complexity (elaborated 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 takes up a prohibitive amount of space (exceeding several times the typical memory sizes), and leads to an excessive computation cost, because the CCA complexity increases with the number of edges in the graph.

Motivated by the lack of CCA algorithms for large datasets, we develop efficient and highly scalable techniques that produce an *optimal assignment*. 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. Furthermore, we extend our framework with an update method, which utilizes an existing matching to derive the new optimal assignment when the customer dataset is updated. Additionally, we design *approximate* solutions that 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.

A preliminary version of this paper appears in [U et al. 2008b]. The best exact algorithm proposed in that version is IDA (described in Section 3.3). Here, we enhance IDA with an optimization (Section 3.3.2), but we additionally propose a new algorithm, SIA, that vastly outperforms IDA. Furthermore, the current version considers incremental assignment maintenance in the presence of customer updates, a topic ignored by our preliminary work. Moreover, a byproduct of SIA is faster approximation techniques; although our approximate algorithms are practically the same, they can now achieve better quality in shorter time, using SIA as a building block (instead of IDA).

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, and then describes our optimal CCA algorithms that utilize it. Section 4 extends our framework with an incremental maintenance technique that processes customer updates. Section 5 studies the trade-off between computation cost and matching quality, and develops approximate CCA solutions with guaranteed error bounds. Section 6 empirically evaluates our exact and approximate CCA methods using synthetic and real datasets. Finally, Section 7 summarizes and concludes the paper.

2. BACKGROUND AND RELATED WORK

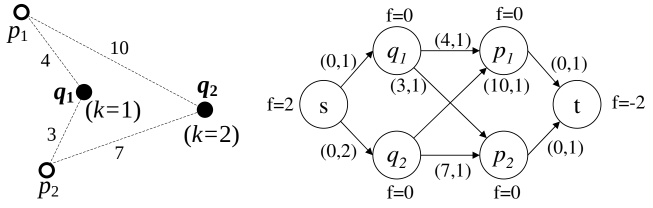
CCA can be reduced to a flow problem on a graph. In Section 2.1 we describe the graph formulation of CCA, and in Section 2.2 we describe a traditional algorithm for the corresponding flow problem. Even though this solution is inapplicable to our setting, it is fundamental to our techniques. In Section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

Symbol	**Description**
Q	set of service providers (points)
P	set of customers (points)
$dist(q_i, p_j)$	Euclidean distance between q_i and p_j
$e(q_i, p_j)$	(directed) edge from q_i to p_j
$w(q_i, p_j)$	cost of edge $e(q_i, p_j)$
s	source node
t	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}	last node in the current shortest path that belongs to P
γ	required flow ($min\{\|P\|, \sum_{q \in Q} q.k\}$)

Table I. Notation

2.1 Minimum Cost Flow on Bipartite Graph

CCA can be reduced to a maximum flow problem on a (directed) bipartite graph [Ahuja et al. 1993]. 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

Fig. 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:

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

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

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

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 $x(p_j, q_i) = 1$. 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 [Ahuja et al. 1993], including adaptations of the primal simplex (linear programming) method [Hung 1983], cost scaling [Gabow and Tarjan 1991; Goldberg and Kennedy 1995], signature [Balinski 1985] and relaxation [Bertsekas 1981; 1988] techniques. The most general approaches with the lowest complexity are the *Hungarian algorithm* and the *successive shortest path algorithm* (SSPA; described in detail in Section 2.2).

The Hungarian algorithm [Kuhn 1955; Munkres 1957] 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 cost matrix may not fit in main memory.

The aforementioned methods solve a static (i.e., one-time) MCF problem. Conversely, [Toroslu and Üçoluk 2007] proposes an update module for the Hungarian algorithm. Given the current MCF solution, this method computes the new solution if a provider-customer pair (along with their incident edges) is appended to the current flow graph. The general idea is to add the new customer and the new provider to the existing cost matrix and perform a single iteration of the Hungarian algorithm on it. This method explicitly considers one-to-one assignment and assumes that the number of providers is equal to the number of customers (i.e., $q.k = 1$ for all $q \in Q$, and $|Q| = |P|$). Furthermore, it cannot handle deletions, and requires that a provider is inserted for every inserted customer.

[Mills-Tettey et al. 2007] extends the idea in [Toroslu and Üçoluk 2007] to handle multiple updates and address deletions (while still assuming one-to-one assignment). Specifically, letting $|I|$ and $|D|$ be the number of insertions and deletions, [Mills-Tettey et al. 2007] applies the insertions/deletions onto the cost matrix and runs on it $\max\{|I|, |D|\}$ iterations of the Hungarian algorithm. The main limitation of [Mills-Tettey et al. 2007] is that, similar to [Toroslu and Üçoluk 2007], it uses the voluminous cost matrix of the Hungarian algorithm (with size $O(|Q| \cdot |P|)$), which yields it inapplicable to moderate or large problem instances due to its huge space requirements.

We note that updating an MCF (or CCA) solution is different from the *dynamic optimal assignment* in vehicle routing [Spivey and Powell 2004]; the latter is a scheduling problem that incorporates the anticipation of future events and defines optimality differently (i.e., the individual assignments reported are non-optimal for the current state of the system, but are expected to optimize some criterion in the long run).

2.2 Successive Shortest Path Algorithm

SSPA is a popular technique for the MCF problem defined above [Derigs 1981]. It 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 in 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

[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.

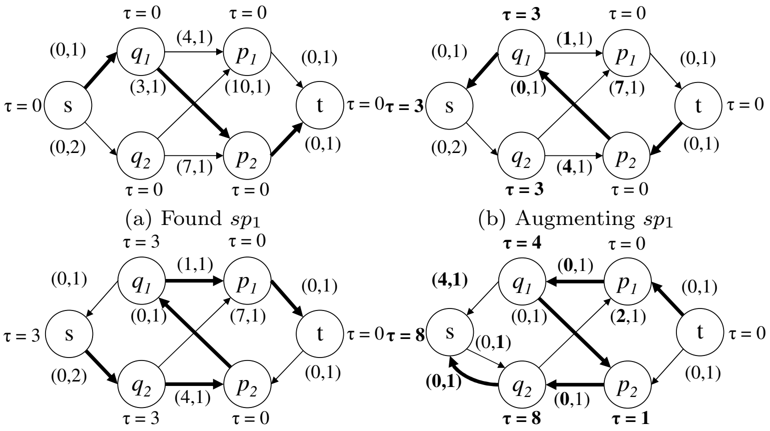
back and reverses the direction of all the edges it contains (Lines 4–6); we say that this step *augments* the path into the graph. Then, SSPA updates the *potential* (to be discussed shortly) of the nodes visited by Dijkstra's algorithm (Lines 7, 8), and the costs of the edges incident to these nodes (Lines 9–13).

Algorithm 1 Successive Shortest Path Algorithm (SSPA)

```
    algorithm SSPA(Set $Q$, Set $P$, Edge set $E$)
 1: for $loop$:=1 to $\gamma$ do
 2:     $v_{min}$:=Dijkstra($Q$, $P$, $E$)
 3:     $v$:=$v_{min}$ //$v$ is a local variable of node type
 4:     while $v.prev \neq \emptyset$ do
 5:         reverse $e(v, v.prev)$ in $E$
 6:         $v$:=$v.prev$                         ▷ proceed with the previous node
 7:     for all visited nodes $v_i$ do
 8:         $v_i.\tau$:=$v_i.\tau - v_i.\alpha + v_{min}.\alpha$
 9:         for all edges $e(v_i, v_j)$ incident to $v_i$ do
10:             if $v_i \in Q \cup \{s\}$ then
11:                 $w(v_i, v_j)$:=$dist(v_i, v_j) - v_i.\tau + v_j.\tau$
12:             if $v_i \in P$ then
13:                 $w(v_i, v_j)$:=$-dist(v_i, v_j) - v_i.\tau + v_j.\tau$
```

An important step in SSPA is the edge cost updating performed in Lines 11 and 13. 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 Line 8 every time 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 $\pm dist(v_i, v_j) - v_i.\tau + v_j.\tau$ 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.

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, and (ii) creating reverse edge $e(q_2, s)$ with capacity 1 and cost 0. To complete



(a) Found sp_1 (b) Augmenting sp_1

(c) Found sp_2 (d) Augmenting sp_2

Fig. 3. Example of SSPA

the example, the 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).

SSPA requires that the entire flow graph is available. 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.

2.3 Spatial Queries

Point sets are usually indexed by spatial access methods in order to accelerate query processing. The R-tree [Guttman 1984] and its variants (e.g., [Sellis et al. 1987; Beckmann et al. 1990]) are the most common such indexes. The R-tree is a 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 associated 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 first KNN method for R-trees [Roussopoulos et al. 1995] searches the tree in a depth-first manner, by recursively visiting the node with the minimum distance from q. The state-of-the-art KNN processing technique is the best-first NN algorithm [Hjaltason and Samet 1999], 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.

Assignment problems in large spatial databases have recently received consider-able attention. [U et al. 2008a] and [Wong et al. 2007] study the *spatial matching* (SM) join, which is an instance of the traditional *stable marriage problem* [Gale and Shapley 1962; Gusfield and Irving 1989] in the spatial domain. Given two point sets P and Q, the SM join iteratively outputs the closest pair [Corral et al. 2000] (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. [Wong et al. 2007] enhances the performance of a naïve (i.e., repetitive closest pair) algorithm with several geometric observations. SM is related, yet different by definition from CCA; SM greedily performs local assignments instead of minimizing the global assignment cost.

[Papadias et al. 2005] proposes the *aggregate nearest neighbor* (ANN) search problem, which retrieves the K points (of a point set P) with the smallest sums of distances from a set Q of query points. There are several major differences between ANN search and CCA: (i) the ANN result set contains K points in P, whereas CCA computes a matching between P and Q, and (ii) in ANN there are no capacity considerations (i.e., each examined point in P is evaluated according to its aggregate distance from *all* points in Q).

Another related problem in spatial databases is *min-dist optimal-location* (MDOL) computation [Zhang et al. 2006]. Given a set of points P, a set of existing providers Q, and a user-specified spatial region R (i.e., a permissible range for installing a new provider), the MDOL problem is to derive the location inside R where if a new provider is placed, it will minimize the average distance between each point in P and its closest provider. MDOL is different from CCA, because it outputs a single point (as opposed to a matching) and it implicitly assigns each point in P to its closest provider, ignoring any possible capacity constraints.

[Ester et al. 1995] and [Mouratidis et al. 2008] propose K*-medoid* methods for large spatial datasets indexed by R-trees. In this problem, the objective is to choose K points (i.e., medoids) from an input dataset P so that the sum of distances between all points in P and their closest medoid is minimized. The problem is NP-hard, so both techniques use heuristics and produce suboptimal answers. The former approach follows the hill-climbing paradigm, while the latter uses the Hilbert space-filling curve [Bially 1969] to group the points into K groups and extract one medoid per group. The problem is intrinsically different from CCA, since the input includes a single dataset and there are no capacity considerations.

3. EXACT METHODS

In this section we present methods for computing an optimal assignment. In accordance with most real-world scenarios, we consider that Q (the set of service providers) and P (the set of customers) contain two-dimensional points. However, our algorithms can easily extend to problems of higher dimensionality. Both datasets are stored in main memory. This is a reasonable choice, because as shown in the preliminary version of this paper, the I/O cost of our CCA techniques exceeds that of fetching the entire (index of the) datasets from the disk. Keeping Q and P in primary storage is sensible, as their size accounts for a small fraction of the overall memory requirements (CCA is a memory-intensive problem).

Our techniques direct the search towards one of Q or P, performing multiple NN computations in it. To accelerate these operations, a spatial index is assumed on the corresponding dataset. In the basic form of our methods (and unless otherwise specified), P is the indexed dataset. However, an enhanced approach presented later may require that the search is directed towards Q, necessitating an index on Q instead. The index may be any data-partitioning spatial access method; in the following we consider R-trees due to their prevalence and wide applicability.

3.1 Fundamental Primitives

As explained in Section 2.2, SSPA is not applicable to large CCA problem instances. 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 small costs 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.

Our general idea is to execute 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 3.1. 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 3.1 refer to edge lengths, and not to their costs (note that costs $w(q_i, p_j)$ 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 in 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 in the complete edge set E.

THEOREM 3.2. *Consider a distance-bounded edge set $E_{sub} \subseteq E$. Let sp be the shortest path (between source and sink) in E_{sub} and $\tau_{max} = \max\{q_i.\tau | q_i \in Q\}$ 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) in 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, the cost of an edge $e(q_i, p_j)$ is defined as $w(q_i, p_j) = dist(q_i, p_j) - q_i.\tau + p_j.\tau$. Since $dist(q_i, p_j) \geq \phi(E - E_{sub})$, $q_i.\tau \leq \tau_{max}$, and $p_j.\tau \geq 0$, it holds that

$$w(q_i, p_j) \geq \phi(E - E_{sub}) - \tau_{max}, \forall e(q_i, p_j) \in E - E_{sub}$$

According to the above (and since edge costs are always non-negative), any path passing through an edge $e(q_i, p_j)$ in $E - E_{sub}$ has at least a cost of $\phi(E - E_{sub}) - \tau_{max}$. Therefore, if the shortest path sp (in 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 algorithms enlarge E_{sub} with incremental nearest neighbor searches [Hjaltason and Samet 1999] around points in Q. Section 3.2 presents a basic approach (termed NIA), while Section 3.3 proposes an improved method (IDA) that utilizes some central observations in the flow subgraph. Section 3.4 describes a third approach (SIA) that reduces running time based on a simplification of the flow graph. Section 3.5 discusses an optimization in the implementation of these three methods.

3.2 Nearest Neighbor Incremental Algorithm

Our basic approach is the *nearest neighbor incremental algorithm* (NIA). Algorithm 2 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 every loop, NIA de-heaps the shortest edge $e(q_i, p_j)$ from H and inserts it into E_{sub} (Lines 9, 10). Then, it computes the next nearest neighbor of q_i and inserts the corresponding edge into H (Lines 11, 12). 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 length 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 3.2 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 $TopKey(H) - \tau_{max}$, or the sink is unreachable), 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 into the graph. Overall, NIA terminates after γ completed iterations (equivalently, after augmenting γ valid shortest paths), where γ is the required flow and equals $min\{|P|, \sum_{q \in Q} q.k\}$.

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.

3.3.1 *Exploiting Full and Non-Full Nodes in E_{sub}.* IDA utilizes observations in the flow subgraph evolution during execution. Specifically, it relies on the concept of *full* service providers and *full* customers.

Algorithm 2 Nearest Neighbor Incremental Algorithm (NIA)

```
algorithm NIA(Set $Q$, Set $P$)
$H$:=new min-heap
$\tau_{max}$:=0; $E_{sub}$:=$\emptyset$
for all $q_i \in Q$ do
    $p_j$:=first NN of $q_i$ in $P$
    insert $\langle e(q_i, p_j), dist(q_i, p_j) \rangle$ into $H$
for $loop$:=1 to $\gamma$ do
    $v_{min}.\alpha := \infty$
    while $v_{min}.\alpha > TopKey(H) - \tau_{max}$ do
        de-heap the top entry $\langle e(q_i, p_j), dist(q_i, p_j) \rangle$ from $H$
        insert edge $e(q_i, p_j)$ into $E_{sub}$
        $p_m$:=next NN of $q_i$ in $P$
        insert $\langle e(q_i, p_m), dist(q_i, p_m) \rangle$ into $H$
        $v_{min}$:=Dijkstra($Q$, $P$, $E_{sub}$)
    ReverseEdges()
    UpdatePotentials()
    $\tau_{max}$:=max $\{q_i.\tau | q_i \in Q\}$                ▷ the highest potential
```

Definition 3.3. 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 $q_i.\alpha$. 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 3.2. $\Phi(E - E_{sub})$ models the minimum possible cost an *sp* could have if it passes 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 3.4. 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 3.5. *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., $sp = \{s, q_i, p_j, t\}$, 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., $sp = \{s, q_i, p_j, t\}$. 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 (Line 8 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. If p_j is full, we directly insert it into E_{sub} and de-heap the next entry; otherwise, we report $sp = \{s, q_i, p_j, t\}$. 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 9–12 of Algorithm 2).

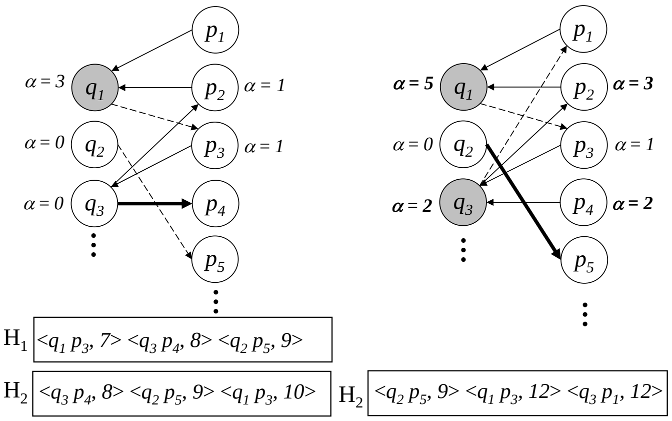
Algorithm 3 is the pseudo-code of IDA. Lines 1–5 initialize E_{sub} identically to NIA. In Line 11 we compute the current sp. Note that if no service provider is full, we derive sp using Theorem 3.5 and the method described above (we omit this enhancement from the pseudo-code for readability). In Lines 12–14, 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 14). Lines 15–16 retrieve the next NN of q_i (q_i refers to $e(q_i, p_j)$ de-heaped in Line 9) and insert the corresponding edge into H. Note that we perform this after updating the $q.\alpha$ values in Lines 12–14 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. 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 8 of Algorithm 2 (Algorithm 3), 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).

Algorithm 3 Incremental On-demand Algorithm (IDA)

```
    algorithm IDA(Set $Q$, Set $P$)
 1: $H$:=new min-heap
 2: $\tau_{max}$:=0; $E_{sub}$:=$\emptyset$
 3: for all $q_i \in Q$ do
 4:     $p_j$:=first NN of $q_i$ in $P$
 5:     insert $\langle e(q_i, p_j), dist(q_i, p_j)\rangle$ into $H$
 6: for $loop$:=1 to $\gamma$ do
 7:     $v_{min}.\alpha := \infty$
 8:     while $v_{min}.\alpha > TopKey(H) - \tau_{max}$ do
 9:         de-heap $\langle e(q_i, p_j), key\rangle$ from $H$
10:         insert $e(q_i, p_j)$ into $E_{sub}$
11:         $v_{min}$:=Dijkstra($Q, P, E_{sub}$)
12:         for all visited $q \in Q$ do
13:             if $q$ is full and $q.\alpha$ changed in Line 11 then
14:                 update $q.\alpha$ in $H$
15:         $p_m$:=next NN of $q_i$ in $P$
16:         insert $\langle e(q_i, p_m), q_i.\alpha + dist(q_i, p_m)\rangle$ into $H$
17:     ReverseEdges()
18:     UpdatePotentials()
19:     $\tau_{max}$:=max $\{q_i.\tau | q_i \in Q\}$                ▷ the highest potential
```

$dist(q_i,p_j)$	p_1	p_2	p_3	p_4	p_5
q_1	5	1	7	-	-
q_2	-	-	-	-	9
q_3	10	6	4	8	-



α = 3
α = 0
α = 0
q_1
q_2
q_3
p_1
p_2
p_3
p_4
p_5
α = 1
α = 1
H_1 <q_1 p_3, 7> <q_3 p_4, 8> <q_2 p_5, 9>
H_2 <q_3 p_4, 8> <q_2 p_5, 9> <q_1 p_3, 10>
α = 5
α = 0
α = 2
α = 3
α = 1
α = 2
H_2 <q_2 p_5, 9> <q_1 p_3, 12> <q_3 p_1, 12>

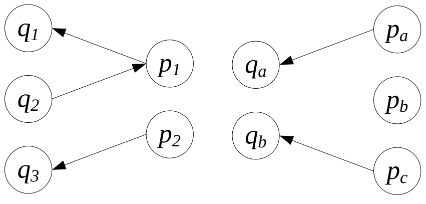
(a) IDA versus NIA (b) Key update

Fig. 4. Utilizing full service providers in IDA

This implies that the current IDA iteration has higher chances to terminate without needing to insert new edges and re-invoke Dijkstra's algorithm.

Let us now focus on IDA. Since the top edge in H_2 is $e(q_3, p_4)$ (shown in bold), we insert it into E_{sub}. Figure 4(b) shows the new flow graph, assuming that $q_3.k = 2$

$dist(q_i,p_j)$	$q_1\ (p_a)$	$q_2\ (p_b)$	$q_3\ (p_c)$
$p_1\ (q_a)$	3	4	9
$p_2\ (q_b)$	9	9	7



(a) Original CCA (b) Transformed CCA

Fig. 5. Example of role reversal between Q and P

and that the subsequent Dijkstra execution returned an *sp* passing through $e(q_3, p_4)$. 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}.

3.3.2 *Reverse Handling for Over-Capacity Cases.* IDA exploits an additional observation, related to the supply-demand relationship as defined by the cumulative provider capacity and the number of customers. Specifically, we distinguish between the under-capacity ($\sum_{q\in Q} q.k \leq |P|$) and over-capacity ($\sum_{q\in Q} q.k > |P|$) cases. In the over-capacity case, providers take longer to get full (with some never becoming full). This significantly limits the effectiveness of the first enhancement described in Section 3.3.1 (i.e., the $q_i.\alpha$ values remain 0 for many/all IDA iterations). To solve this problem, we process over-capacity problems by reversing the roles of Q and P; the resulting problem is now an under-capacity one and IDA is performed on it instead. This technique reduces the size of E_{sub}, as well as the number of Dijkstra executions.

To exemplify, consider Figure 5(a). All service providers have capacity 1 (rendering this an over-capacity problem) and the provider-customer distances are given in the table above the flow graph. Assume that we apply IDA directly (without role reversal). The first IDA iteration inserts $e(q_1, p_1)$ into E_{sub}, and augments the corresponding path (as it is valid according to Theorem 3.2). The second iteration initially inserts $e(q_2, p_1)$, but the corresponding *sp* is invalid. Thus, it additionally inserts $e(q_3, p_2)$ and terminates after augmenting the new *sp*. Figure 5(a) shows the final E_{sub}.

What we propose, on the other hand, is to reverse the roles of Q and P; Figure 5(b) shows the transformed (under-capacity) problem. After the first IDA iteration (which includes $e(q_a, p_a)$ into E_{sub}), $q_a.\alpha = \infty$. This updates the key of $e(q_a, p_b)$ in edge heap H to ∞ and prevents it from being inserted into E_{sub} in the second iteration. Figure 5(b) shows the final E_{sub}, which only includes 2 edges (versus 3 in Figure 5(a)). Also, only 2 shortest paths were computed (versus 3 in the original, over-capacity problem).

Note that, even though node reversal reduces the number of E_{sub} edges and Dijkstra executions, the number of IDA iterations remains $\gamma = \min\{|P|, \sum_{q \in Q} q.k\}$. Another remark is that the reversal requires an R-tree on Q to accelerate NN searches in it, as mentioned in the beginning of Section 3.

3.4 Simplified Graph Incremental Algorithm

There is an SSPA implementation (we refer to it as SSPA$^+$) which has the same space requirements and asymptotic performance as the original SSPA, but runs faster in practice. SSPA$^+$ requires that the summed provider capacity is equal to the number of customers [Ahuja et al. 1993; Orlin and Lee 1993], i.e., equi-capacity case, where $|P| = \sum_{q \in Q} q.k$. The main idea in SSPA$^+$ is to eliminate the source s and the sink t from the flow graph (along with their incident edges). In lack of source s, shortest paths can start from any provider q and end at any customer p, provided that less than $q.k$ paths have started at q, and no path has ended at p yet (or, in IDA terminology, that both q and p are non-full). Correctness is guaranteed after $\gamma = |P| = \sum_{q \in Q} q.k$ iterations, regardless of the order that providers were chosen as the shortest path starting points. SSPA$^+$ is more efficient than SSPA because its Dijkstra searches start from the providers directly (instead of s), leading to fewer heap operations and smaller heap sizes.

Our third CCA algorithm, termed *simplified graph incremental algorithm* (SIA), exploits the simplified flow graph of SSPA$^+$. Prior to describing SIA, we first extend SSPA$^+$ to cases where $|P| \neq \sum_{q \in Q} q.k$. We only consider the under-capacity case ($\sum_{q \in Q} q.k < |P|$), because SIA employs the reverse handling of Section 3.3.2 for over-capacity problems.

If $\sum_{q \in Q} q.k < |P|$, we add one fictitious service provider q_e with capacity $q_e.k = |P| - \sum_{q \in Q} q.k$ into the flow graph; there are $|P|$ artificial edges from q_e to all customers p with $dist(q_e, p) = \infty$ (practically, $dist(q_e, p)$ is set to a very large number). Note that the resulting problem is an equi-capacity one, and SSPA$^+$ is now applicable. After inserting q_e, the required flow becomes $\gamma = |P|$. However, it can be easily seen that none of the artificial edges of q_e would appear in an optimal assignment (for the original problem), and that any sp from q_e would not pass through or affect any edge $e(q, p)$ for $q \neq q_e$. Therefore, we do not need to perform shortest path searches from q_e, and only have to execute $\sum_{q \in Q - \{q_e\}} q.k$ iterations of SSPA$^+$ (instead of $|P|$).

Although SSPA$^+$ is faster than SSPA and can be extended to under- and over-capacity cases as shown above, it does not overcome the main limitation of SSPA, which is its prohibitive space requirements. Specifically, SSPA$^+$ requires the entire bipartite graph in main memory, and is thus inapplicable to moderate and large size CCA problems. In the following, we show how our CCA (and, specifically, the IDA) pruning techniques can be applied in conjunction with SSPA$^+$.

3.4.1 *Pruning Techniques in SIA.* In NIA and IDA, edge pruning is based on Theorem 3.2. Below, we present an adaptation of this theorem that reflects the lack of sink and source nodes. First, however, we must redefine the concept of a distance-bounded edge set. In contrast with Definition 3.1, E now refers to the complete bipartite graph between Q and P (i.e., it excludes the edges incident to s and t). Also, since the source is missing, a non-full provider q_i plays its role. We define $E_{sub} \subseteq E$ to be *distance-bounded with respect to* q_i *with bound* Ξ as follows.

Definition 3.6. An edge set $E_{sub} \subseteq E$ is said to be *distance-bounded w.r.t.* q_i *with bound* Ξ if

$$\forall\, e(q_k, p_j) \in E - E_{sub},\; q_k.\alpha + dist(q_k, p_j) > \Xi$$

Note that $q_k.\alpha$ represents the minimum cost from q_i to q_k, and that $q_i.\alpha = 0$, since q_i now plays the role of s. Essentially, the above definition requires that E_{sub} contains all edges within cost Ξ from q_i. Under the above definition, Theorem 3.7 is the crux of SIA.

THEOREM 3.7. *Consider a distance-bounded edge set* $E_{sub} \subseteq E$ *w.r.t. provider* q_i *with bound* Ξ. *Let sp be the shortest path from* q_i *to a non-full* $p_j \in P$ *and* $\tau'_{max} = \max\{v.\tau | v \in Q \wedge v.\alpha \leq \Xi\}$. *If the total cost of sp is at most* $\Xi - \tau'_{max}$, *then sp is also the shortest path (between* q_i *and a non-full customer) in the complete flow graph* E.

PROOF. Suppose that the shortest path from q_i (to a non-full customer) in E contains an edge $e(q_k, p_j)$ that is not in E_{sub}. The cost of this path is $q_k.\alpha + dist(q_k, p_j) - q_k.\tau + p_j.\tau$ plus the cost of the remaining path from p_j to the end-node (non-full customer). Since $e(q_k, p_j) \notin E_{sub}$, it follows from Definition 3.6 that:

$$\begin{aligned} q_k.\alpha + dist(q_k, p_j) &\geq \Xi \Rightarrow \\ q_k.\alpha + dist(q_k, p_j) - q_k.\tau &\geq \Xi - q_k.\tau \end{aligned}$$

For a path passing through q_k to be shorter than sp, it should hold that $q_k.\alpha \leq \Xi - \tau'_{max} \leq \Xi$. Thus, $q_k \in \{v | v \in Q \wedge v.\alpha \leq \Xi\}$, and hence $q_k.\tau \leq \tau'_{max}$. Therefore,

$$\begin{aligned} q_k.\alpha + dist(q_k, p_j) - q_k.\tau &\geq \Xi - \tau'_{max} \Rightarrow \\ q_k.\alpha + dist(q_k, p_j) - q_k.\tau + p_j.\tau &\geq \Xi - \tau'_{max} \Rightarrow \\ q_k.\alpha + w(q_k, p_j) &\geq \Xi - \tau'_{max} \end{aligned}$$

The above contradicts our assumption, because it implies that any path passing through edge $e(q_k, p_j)$ has cost higher than $\Xi - \tau'_{max}$ and, thus, larger than sp. Hence, if the shortest path sp (in E_{sub}) has total cost no greater than $\Xi - \tau'_{max}$, then it must be the shortest path in the entire E too. □

Algorithm 4 presents the pseudo-code of SIA, which is practically an adaptation of IDA based on Theorem 3.7. In the beginning of each iteration, SIA selects a source q_i in round-robin fashion among the non-full providers[2]. After q_i is determined, SIA finds its first NN and places the corresponding edge into heap H. Unlike IDA which en-heaps $|Q|$ edges (initialization Lines 3–5 in Algorithm 3), SIA inserts a single edge into H; since E_{sub} is empty, $q_k.\alpha = \infty$ for all service providers other than q_i and, thus, Theorem 3.7 ignores them at this stage. If during some Dijkstra execution (Line 12) a provider q is encountered for the first time in the

[2] The source selection strategy does affect performance, albeit to a small degree. We tried various heuristics, as well as exhausting the excess capacity of a provider before proceeding to the next non-full one, but the round-robin technique yielded the best performance.

current SIA iteration, then the edge with its first NN is inserted into H (Lines 14–16). This implies that q's edges start being considered for inclusion into E_{sub}, as $q.\alpha$ is no longer ∞. The validity of a shortest path according to Theorem 3.7 is checked in Line 9, where $TopKey(H)$ plays the role of bound Ξ; it is easy to see that $\Xi = TopKey(H)$ since edge insertion in Line 16 adds $q.\alpha$ to the edge length, and uses their sum as the sorting key. An important remark is that (unlike IDA) SIA re-initializes heap H and sets $v.\alpha = \infty$ for all $v \in E_{sub}$ at the beginning of every iteration (Line 3), because once a different provider q_i is chosen as source (Line 4), the previous node distances and H contents are invalidated. Finally, note that the node potentials are maintained across iterations (i.e., they are not reset in each iteration) to avoid negative edges in E_{sub}.

Algorithm 4 Simplified Graph Incremental Algorithm (SIA)

```
    algorithm SIA(Set $Q$, Set $P$)
 1: $\tau'_{max}$:=0; $E_{sub}$:=$\emptyset$
 2: for $loop$:=1 to $\gamma$ do
 3:     $H$:=new min-heap; set $v.\alpha := \infty$ for each $v \in E_{sub}$
 4:     select a non-full $q_i \in Q$ in round-robin fashion
 5:     $q_i.\alpha := 0$
 6:     $p_j$:=first NN of $q_i$ in $P$
 7:     insert $\langle e(q_i, p_j), dist(q_i, p_j) \rangle$ into $H$
 8:     $v_{min}.\alpha := \infty$
 9:     while $v_{min}.\alpha > TopKey(H) - \tau'_{max}$ do
10:         de-heap $\langle e(q_k, p_j), key \rangle$ from $H$
11:         insert $e(q_k, p_j)$ into $E_{sub}$
12:         $v_{min}$:=Dijkstra($Q, P, E_{sub}$)
13:         for all visited $q \in Q$ do
14:             if $q$ is not in $H$ then                  ▷ i.e., $q.\alpha$ used to be $\infty$
15:                 $p_j$:=get next NN of $q$ in $P$
16:                 insert $\langle e(q, p_j), q.\alpha + dist(q, p_j) \rangle$ into $H$
17:             if $q.\alpha$ changed in Line 12 then
18:                 update $q.\alpha$ in $H$
19:         $p_m$:=next NN of $q_k$ in $P$
20:         insert $\langle e(q_k, p_m), q_k.\alpha + dist(q_k, p_m) \rangle$ into $H$
21:     ReverseEdges()
22:     UpdatePotentials()
23:     $\tau'_{max} = \max\{v.\tau | v \in Q \wedge v.\alpha \leq TopKey(H)\}$
```

SIA benefits from the flow graph simplification (i.e., the removal of source and sink), because Dijkstra executions become faster in a fashion similar to SSPA$^+$. Compared to IDA, the path verification bound in SIA becomes tighter; τ'_{max} (in Theorem 3.7) is expected to be smaller than τ_{max} (in Theorem 3.2) because it is computed over a subset of the providers. This leads to a more effective *sp* verification mechanism, implying faster execution and smaller E_{sub}. Figure 6 demonstrates these two advantages with an example.

There are $|Q| = 3$ providers with capacity 1 each, and $|P| = 3$ customers. The flow graph corresponds to either algorithm (IDA and SIA) after two executions; for

$dist(q_i,p_j)$	p_1	p_2	p_3
q_1	3	20	5
q_2	20	5	19
q_3	4	10	20

H_1 $<q_3\,p_2,\ 10>\ <q_1\,p_2,\ 20>\ <q_2\,p_3,\ \infty>$

H_2 $<q_3\,p_2,\ 10>\ <q_1\,p_2,\ 20>$

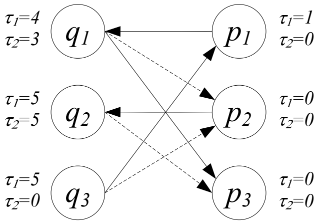


Fig. 6. IDA versus SIA

SIA, we assume that the first iteration chose q_1 as the source, and the second chose q_2. The solid edges have been inserted into E_{sub}, while the dashed edges indicate the next NN of the service providers. Potential values τ_1 and τ_2 above each node correspond to IDA and SIA, respectively. In the third iteration, the edge heap of IDA is H_1 and of SIA (with source q_3) is H_2; note that H_2 does not contain $e(q_2, p_3)$ because $q_2.\alpha = \infty$ (i.e., q_2 was not visited by any Dijkstra execution in the current iteration). The shortest path in both[3] IDA and SIA is $sp = \{q_3, p_1, q_1, p_3\}$ with cost 6. $TopKey(H)$ is also the same in both cases (i.e., 10). Nevertheless, τ_{max} is 5 but τ'_{max} is only 3. Therefore, IDA (where $TopKey(H) - \tau_{max} < 6$) needs to insert more edges into H before it can deem sp valid. On the other hand, SIA augments sp directly without further expansion.

3.5 Optimization: Reducing Dijkstra Executions

All our algorithms (NIA, IDA, SIA) apply incremental NN search to discover the edges one-by-one, in order to keep E_{sub} small. However, since E_{sub} expands slowly, they may perform numerous Dijkstra executions. To accelerate processing, we reduce the cost of Dijkstra executions by (i) reusing 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 iteration (of either NIA, IDA, or SIA) 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}. Let H_d be the Dijkstra search heap after the 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 source 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 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 shorter 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

[3] The sp in IDA starts from s and ends at t, but we only show the intermediate nodes for simplicity.

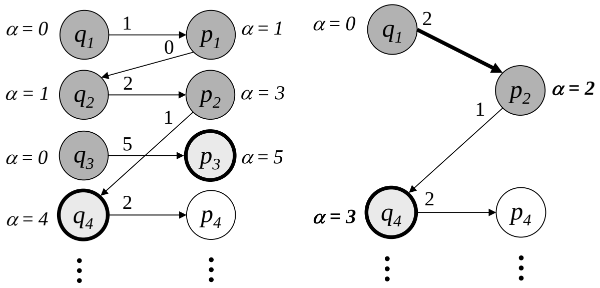
to v_i (to indicate that v_j is now reachable via v_i). If v_j is in H_d or H_f, its key is updated to $v_j.\alpha$ in its containing heap. Otherwise (i.e., if v_j is neither in H_d nor H_f), it is inserted into H_f with key $v_j.\alpha$. PUA terminates when H_f becomes empty. Algorithm 5 presents PUA.

Algorithm 5 Path Update Algorithm (PUA)

```
algorithm PUA(Set $Q$, Set $P$, Heap $H_d$, Edge set $E_{sub}$, Edge $e(q,p)$)
$H_f$:=new min-heap
insert $\langle q, q.\alpha \rangle$ into $H_f$
while $H_f$ is not empty do
    de-heap top node $v_i$ (with the lowest $v_i.\alpha$ value) from $H_f$
    for all edges $e(v_i, v_j) \in E_{sub}$ outgoing from $v_i$ do
        if $v_j.\alpha > v_i.\alpha + w(v_i, v_j)$ then
            $v_j.\alpha$:=$v_i.\alpha + w(u,v)$; $v_j.prev$:=$v_i$
            if $v_j \in H_d$ then
                update $v_j.\alpha$ in $H_d$
            else if $v_j \in H_f$ then
                update $v_j.\alpha$ in $H_f$
            else
                insert $\langle v_j, v_j.\alpha \rangle$ into $H_f$
```

Example: We illustrate the PUA technique with an example, assuming processing with NIA/IDA. Figure 7(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 7(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* search avoids unnecessary computations for them. PUA works similarly with SIA, since it concerns only the Dijkstra building block. Assuming that SIA chooses q_1 as the source, the difference in the above example is that q_3 would not be visited and would not be in H_d.

PUA can utilize results only among Dijkstra executions that take place in the same 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/SIA use the above PUA-based optimization only after some of the service providers become full, because until then shortest paths are computed using Theorem 3.5 directly.



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

Fig. 7. Example of PUA

4. INCREMENTAL CCA MAINTENANCE

In the previous section we computed a one-time, static assignment. However, in practice, the customers may issue updates, i.e., move to a new location. Since re-assignment from scratch is expensive, in this section we propose an approach to incrementally update an existing matching. Specifically, we extend our best exact method, SIA, to process updates of customer locations; we focus on customer updates, as usually the provider set is rather static.

In Section 2.1 we described existing methods for assignment maintenance, i.e., [Toroslu and Üçoluk 2007; Mills-Tettey et al. 2007]. The main drawback of these techniques is that they require the complete cost matrix of the Hungarian algorithm to fit in main memory. The size of this matrix is equivalent to the entire flow graph E. The central challenge in this section is how to extend our framework so that optimality can be maintained efficiently with small space requirements.

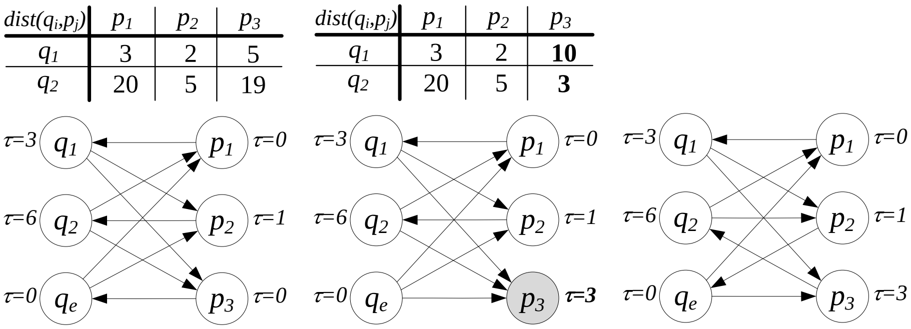
First, in Section 4.1, we adapt the (Hungarian-based) method of [Mills-Tettey et al. 2007] to work with SSPA$^+$; note that Hungarian and SSPA are practically dual views of the same process and that a mapping between them is possible. Since the resulting SSPA$^+$ approach still requires the entire flow graph in memory, in Section 4.2 we show how SIA and our CCA pruning techniques can be applied (instead of SSPA$^+$) to maintain the assignment using only a flow subgraph E_{sub}. Section 4.3 describes how arbitrary customer insertions and deletions (i.e., general updates other than just movements) can be dealt with.

4.1 Preliminary Solution

Suppose that an optimal assignment M is stored in our system when we receive location updates from some customers. Our task is to compute the new optimal assignment M' according to the current customer positions.

Our preliminary solution is an SSPA$^+$-based adaptation of [Mills-Tettey et al. 2007]; it follows the same steps using the flow graph instead of the Hungarian cost matrix. The entire E is assumed to be available (since SSPA$^+$ is used). First, we add an imaginary provider q_e with capacity $q_e.k = |P| - \sum_{q \in Q} q.k$ into the flow graph[4]. There are $|P|$ artificial edges from q_e to all customers p with $dist(q_e, p) =$

[4] We focus on the under-capacity case, since the over-capacity one can be dealt with by reverse



$dist(q_i,p_j)$	p_1	p_2	p_3
q_1	3	2	5
q_2	20	5	19

$dist(q_i,p_j)$	p_1	p_2	p_3
q_1	3	2	**10**
q_2	20	5	**3**

(a) Original assignment M (b) Movement of p_3 (c) New assignment M'

Fig. 8. SSPA$^+$ update example

∞. All customers that were unassigned in M are given to q_e; i.e., the corresponding edges are reversed. In Figure 8(a), for example, both q_1 and q_2 have capacity 1, and they were assigned p_1 and p_2, respectively. Customer p_3 was not assigned. The fictitious q_e is added in the flow graph with capacity 1, and p_3 is assigned to it (as indicated by the reversal of edge $e(q_e, p_3)$).

Let p be a moving customer, and assume that it is currently assigned to provider $q \in Q \cup \{q_e\}$. Since the new assignment of p is unknown, we reverse edge $e(p, q)$ in E. First, this results in q becoming non-full. Second, since the Euclidean distance of p from the service providers has changed, some edges may result in negative costs (due to the obsolete potentials). Thus, we update them as indicated by Theorem 4.1.

THEOREM 4.1. *If the costs of some edges $e(q_i, p)$ incident to the updated customer p are negative, we can safely eliminate all negative costs by setting: $p.\tau = \max_{e(q_k,p)\in E \wedge w(q_k,p)<0} \{-dist(q_k, p) + q_k.\tau\}$.*

PROOF. First, we prove that no negative cost exists after updating $p.\tau$. Since $p.\tau$ is increased, all non-negative edge costs remain non-negative. Let us focus now on the negative cost ones. For such an edge $e(q_i, p)$, since $w(q_i, p)$ was negative before the update, it holds that[5] $dist(q_i, p) - q_i.\tau + p.\tau \leq 0 \Rightarrow dist(q_i, p) - q_i.\tau \leq 0 \Rightarrow -dist(q_i, p) + q_i.\tau \geq 0$. After the update, the cost becomes:

$$\begin{aligned} w(q_i, p) &= dist(q_i, p) - q_i.\tau + p.\tau \\ &= dist(q_i, p) - q_i.\tau + \max_{e(q_k,p)\in E \wedge w(q_k,p)<0} \{-dist(q_k, p) + q_k.\tau\} \\ &= \max_{e(q_k,p)\in E \wedge w(q_k,p)<0} \{-dist(q_k, p) + q_k.\tau\} - (-dist(q_i, p) + q_i.\tau) \\ &\geq 0 \end{aligned}$$

The inequality holds because $\max_{e(q_k,p)\in E \wedge w(q_k,p)<0}\{-dist(q_k, p)+q_k.\tau\} \geq -dist(q_i, p)+q_i.\tau$, while both quantities are positive for edges with previously negative costs (as shown above).

handling (see Section 3.3.2).

[5] In this inequality $p.\tau$ refers to the potential of p before the update.

Next, we show that the change does not affect the SSPA$^+$ computation. Due to the inclusion of q_e, the only non-full customer is p. Therefore, if an *sp* is computed, it must end at p. Since the costs of all edges that are incident to p are increased by the same amount, the *sp* computation is not affected by the new $q.\tau$. □

Consider again the example in Figure 8(a), and assume that p_3 moves to a new location. Figure 8(b) shows its new distances from the providers and the updated flow graph. We first reverse $e(q_e, p_3)$. If we keep using the potential values from the previous flow graph (in Figure 8(a)), then $w(q_2, p_3) = dist(q_2, p_3) - q_2.\tau + p_3.\tau = -3$ which is negative. Therefore, we apply Theorem 4.1, and set $p_3.\tau = 3$. After this change, all edge costs are non-negative (note that $w(q_2, p_3) = 0$, which is permissible).

After applying the above process (edge reversal and potential update) for each moving customer, we resume SSPA$^+$. Specifically, letting there be $|U|$ updates, there are $|U|$ edge reversals. This means that there are some non-full providers (with total free capacity $|U|$) and exactly $|U|$ non-full customers. Therefore, we perform $|U|$ SSPA$^+$ iterations; i.e., we compute and augment $|U|$ shortest paths, choosing sources in round-robin fashion among the non-full providers. The new optimal assignment M' is derived from the resulting flow graph, ignoring the assignments made to q_e.

Continuing our example in Figure 8(b), we perform $|U| = 1$ iterations of SSPA$^+$. The only non-full provider is q_e and it is used as source. The shortest path is $\{q_e, p_2, q_2, p_3\}$ which is augmented. This reverses the included edges as shown in Figure 8(c), practically assigning p_3 to q_2 and leaving p_2 unassigned (specifically, p_2 is assigned to q_e and, thus, is left out of M'). Note that, unlike Section 3.4, we cannot simply remove q_e. In our example, for instance, if there was no q_e, there would be no non-full provider to choose as the source of *sp*. Algorithm 6 is the pseudo-code of the complete SSPA$^+$ update process (U is the set of customer updates).

Algorithm 6 SSPA$^+$ Update Process

```
algorithm SSPA-Update(Set $Q$, Set $P$, Edge set $E$, Set $U$)
insert $q_e$ into the system ($Q := Q \cup \{q_e\}$)
set $q_e.k := |P| - \sum_{q_i \in Q} q_i.k$
for all $p \in U$ do
    reverse $e(p, q)$ in $E$                    ▷ $q$ is $p$'s assigned provider in $M$
    update $p.\tau$ according to Theorem 4.1
run $|U|$ iterations of SSPA$^+$ in $E$
```

4.2 SIA Update Process

Our preliminary solution above requires that the entire flow graph fits in memory, and is thus inapplicable to large problems. Using Algorithm 6 as a basis, in this section we replace SSPA$^+$ by SIA, so that only a flow subgraph $E_{sub} \subseteq E$ is used. Our approach requires that the potentials of all providers and customers are recorded after the last assignment (i.e., M) computation. It also requires that the expansion distance $q.\eta$ is kept for each provider q; this is the distance of the last

NN of q retrieved during M's computation. The $q.\eta$ values are used to construct a new E_{sub} when a set of customer updates U arrives (although storing the previous E_{sub} and amending it is possible, this is not much faster than E_{sub} reconstruction as described below).

We build a new E_{sub} as follows. For every $q \in Q$, we perform a range query (with center at q and radius $q.\eta$) on the updated P, and insert into E_{sub} an edge $e(q,p)$ for each retrieved customer p. If p was previously assigned to q, then we reverse edge $e(q,p)$. We set the potentials as stored after M's computation. Subsequently, we proceed as in Section 4.1, i.e., we add the imaginary provider q_e, reverse edges that correspond to assignments of moving customers, and handle any negative costs according to Theorem 4.1. Finally, we perform $|U|$ iterations of SIA in E_{sub} and derive the new assignment M'. Algorithm 7 outlines the SIA maintenance process.

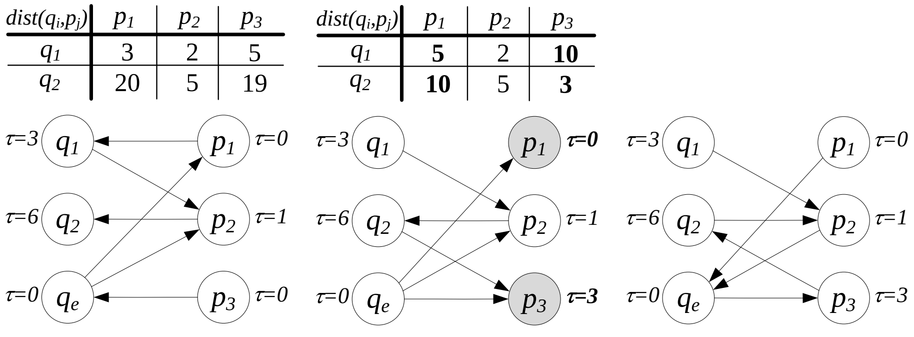
Algorithm 7 SIA Update Process

```
    algorithm SIA-Update(Set $Q$, Set $P$, Edge set $E_{sub}$, Set $U$)
 1: for all $q \in Q$ do
 2:     for all $p \in P$ where $dist(q,p) \leq q.\eta$ do
 3:         insert $e(q,p)$ into $E_{sub}$
 4:         if $(q,p) \in M$ then
 5:             reverse $e(q,p)$ in $E_{sub}$
 6: set node potentials as stored after $M$'s computation
 7: insert $q_e$ into the system ($Q := Q \cup \{q_e\}$)
 8: set $q_e.k := |P| - \sum_{q_i \in Q} q_i.k$
 9: for all $p \in U$ do
10:     reverse $e(p,q)$ in $E_{sub}$
11:     update $p.\tau$ according to Theorem 4.1
12: run $|U|$ iterations of SIA in $E_{sub}$
```

A performance issue with Algorithm 7 relates to the existence of q_e in the flow graph when SIA is executed in Line 12. Specifically, many (among the $|U|$) SIA iterations use q_e as the source. Shortest path computation from q_e is expensive, due to its large number of incident edges; i.e., there are $|P|$ adjacent nodes (all $p \in P$), all of which are en-heaped during Dijkstra computation. Furthermore, multiple sp computations may be required in each iteration (until a valid one is found), exacerbating the problem. In Section 4.2.1 we describe an optimization to mitigate this deficiency.

4.2.1 *Optimization: Removing q_e.* As explained previously, q_e is necessary to ensure optimality. Thus, it cannot be simply deleted/ignored. Our main idea is to first use q_e while optimally solving CCA in E_{sub} (not E), and then remove q_e so that SIA can continue from E_{sub} as per normal and derive the optimal solution in the entire E.

Specifically, our improved update process includes three stages. Stage one runs SSPA$^+$ in E_{sub} (including q_e, but ignoring any edge outside E_{sub}); this leads to an optimal assignment within E_{sub}. Stage two expands and updates E_{sub} so that SIA can run on it. Stage three removes q_e from E_{sub} and produces the new assignment M' using SIA. The performance benefits of this approach over the basic Algorithm



$dist(q_i,p_j)$	p_1	p_2	p_3
q_1	3	2	5
q_2	20	5	19

$dist(q_i,p_j)$	p_1	p_2	p_3
q_1	**5**	2	**10**
q_2	**10**	5	**3**

(a) Original assignment M (b) Movement of p_1 and p_3 (c) E_{sub} after stage one

Fig. 9. SIA update example

7 stem from the fact that (i) in stage one, although q_e still exists, a single sp computation is necessary per SSPA$^+$ iteration (the derived sp needs not be checked for validity, as we only require optimality within E_{sub}), and (ii) in stage three, SIA runs on a graph without q_e.

Consider the example in Figure 9(a). Assuming that the previous assignment M was computed by SIA, the edges of the flow graph correspond to the previous E_{sub}. Suppose that customers p_1 and p_3 move, and their new distances are shown in Figure 9(b). The same figure illustrates the reconstructed E_{sub} (following Lines 1–11 of Algorithm 7). Note that edge $e(p_1, q_1)$ is not included in E_{sub}, since $dist(q_1, p_1)$ is now larger than $q_1.\eta = 3$. At this point, our enhancement takes over.

In stage one, $|U| = 2$ and the non-full providers are q_e and q_1. When q_1 is chosen as source, SSPA$^+$ fails to find any path to a non-full customer; q_1 is only connected to p_2 which is full. However, to ensure optimality (within E_{sub}), $|U|$ paths must be augmented successfully. Therefore, the capacity of q_e is increased by one[6] ($q_e.k = 1 + 1 = 2$) and both SSPA$^+$ iterations use q_e as source. The two shortest paths of q_e are $\{q_e, p_1\}$ and $\{q_e, p_2, q_2, p_3\}$. The resulting E_{sub} is shown in Figure 9(c) and contains three assignments $e(p_1, q_e)$, $e(p_2, q_e)$, and $e(p_3, q_2)$.

Observe that SSPA$^+$ ensures non-negative edge costs in E_{sub} but not in the entire E. Preservation of non-negative costs in E is an invariant in our CCA techniques and must be ensured before SIA can be applied. Formally, the flow subgraph is *valid with respect to* E if using its potentials does not lead to negative costs for any edge in E. Stage two checks whether E_{sub} is valid w.r.t. E and, if it is not, it expands/updates it accordingly. The details of stage two will be presented shortly.

Returning to our example in Figure 9(c), assume that E_{sub} is deemed valid by stage two. Stage three first removes q_e from the flow subgraph, along with its incident edges (i.e., $e(p_1, q_e)$, $e(p_2, q_e)$ and $e(q_e, p_3)$). Finally, SIA is executed on the resulting E_{sub}, augmenting paths from non-full providers until they all become full. Provider q_1 is non-full and is chosen as source. Since SIA fails to find a path to any non-full customer, it inserts into its edge heap H an edge from q_1 to its next

[6] In the general case, $q_e.k$ is increased by the number of iterations that q_1 must be used as the source in order to become full.

NN that lies further than $q_1.\eta = 3$ (i.e., customer p_1). It proceeds like Algorithm 4, expanding E_{sub} until a valid sp is found and augmented. The new optimal assignment M' is extracted from the flow subgraph upon termination of SIA.

4.2.1.1 *Stage two.* The pseudo-code of stage two is given in Algorithm 8. It performs validity checking and expansion at the same time. Initially, for each provider $q_i \in Q$, it retrieves the next NN $p_j \in P$ where $dist(q_i, p_j) > q_i.\eta$. If $dist(q_i, p_j) - q_i.\tau \geq 0$, then the cost of all edges $e(q_i, p_k) \in E - E_{sub}$ is non-negative[7]. If the above inequality holds for all providers $q_i \in Q$, then the flow subgraph is valid w.r.t. E.

If the inequality does not hold for a provider q_i, Algorithm 8 keeps retrieving the next NNs of q_i and inserting the corresponding edges into E_{sub} until $dist(q_i, p_j) - q_i.\tau \geq 0$ (where p_j is the last retrieved NN of q_i). If any of these included edges have negative costs (Line 6), stage two treats each of them as an update of the corresponding customer. Specifically, the insertion of a negative cost edge $e(q_i, p_j)$ into E_{sub} can be dealt with transparently if we assume that the (previously undefined) distance of p_j from q_i is now $dist(q_i, p_j)$, which is equivalent to a movement of p_j. Thus, Lines 6–9 alter accordingly E_{sub} (in a fashion similar to Lines 10, 11 in Algorithm 7) and add p_j into update set U. Note that inserted edges with non-negative costs do not affect the validity of E_{sub} (and do not have to be treated as updates).

Finally, after negative cost edges are guaranteed not to exist in E, Line 12 reruns stages one and two in the updated E_{sub} using U as the set of updates. Note that this recursively alters and expands E_{sub}, until it is valid w.r.t. E and ready to be processed by SIA in stage three.

Algorithm 8 Stage Two

```
    algorithm StageTwo(Set $Q$, Set $P$, Edge set $E_{sub}$, Set $U$)
 1: $U := \emptyset$
 2: for all $q_i \in Q$ do
 3:     $p_j$ :=first NN of $q_i$ in $P$ where $dist(q_i, p_j) > q_i.\eta$
 4:     while $dist(q_i, p_j) - q_i.\tau < 0$ do
 5:         insert $e(q_i, p_j)$ into $E_{sub}$
 6:         if $w(q_i, p_j) < 0$ and $\exists q_k \in Q$ such that $e(p_j, q_k) \in E_{sub}$ then
 7:             reverse $e(p_j, q_k)$ in $E_{sub}$
 8:             update $p_j.\tau$ according to Theorem 4.1
 9:             insert $p_j$ into $U$ if $p_j \notin U$
10:         $p_j$ :=next NN of $q_i$ in $P$
11: if $U \neq \emptyset$ then
12:     run stages one and two in $E_{sub}$
```

[7] It holds by definition that $p_k.\tau \geq 0$. Since $dist(q_i, p_j) - q_i.\tau \geq 0$ and $dist(q_i, p_k) \geq dist(q_i, p_j)$, it follows that $w(q_i, p_k) = dist(q_i, p_k) - q_i.\tau + p_k.\tau \geq 0$.

4.3 Insertion and Deletion Handling

So far we considered customer updates that correspond to movements. However, insertions of new customers and/or deletions of existing ones may occur. Handling is identical to Section 4.2 by establishing the following two conventions:

Insertion of customer p_j into P. We assume that the distance of p_j from each $q_i \in Q$ was previously infinite, while its new distance from them is the actual $dist(q_i, p_j)$.

Deletion of customer p_j from P. We implicitly keep p_j in the system and assume that it has moved from its previous location to a new one, which is infinitely far from every $q_i \in Q$.

Regarding deletion handling, explicit storage of each deleted customer would unnecessarily increase the size of E_{sub} (and thus the storage and computation requirements). To overcome this problem, we introduce a fictitious customer p_e whose distance form each provider is infinite, and whose capacity is increased by one for each customer deletion.

Note that we can handle multiple insertions and deletions (and movements) at the same time. Having modeled each insertion/deletion as a customer movement, we can process multiple ones simultaneously, using the technique in Section 4.2 without any modification.

5. 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. We consider approximation only for one-time, static assignments, because our approximate methods are very fast compared to the exact ones (as we show later), and incremental evaluation would not pay off if suboptimal answers are acceptable.

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 or customers. Finally, the *refinement phase* uses the assignment produced in the previous step to derive a matching on the entire sets P and Q.

We assume a data-partitioning spatial access method on each of Q and P. In addition to the concise matching phase (where an index is necessary on the dataset where the search is directed to), the partitioning phase also requires an index on the partitioned dataset. We consider R-trees because of their widespread use and favorable data grouping properties [Theodoridis et al. 2000].

Section 5.1 describes two alternative partitioning and concise matching approaches, while Section 5.2 describes refinement techniques that could be used with either of these alternatives. Section 5.3 derives error bounds for our approximate solutions.

5.1 Partitioning and Concise Matching

We distinguish between two methods, called *service provider approximation* (PA) and *customer approximation* (CA). PA and CA follow different approaches for partitioning and subsequent concise matching. Specifically, PA 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.

PA and CA proceed symmetrically, treating P in place of Q and vice versa. For the sake of presentation, we focus on CA. Partitioning in CA is performed on P. We first initialize a set S of customer groups to $\emptyset$. Given parameter δ, we traverse the R-tree of P. 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 customers in their subtrees is the entire P. The size of S, however, can be reduced by an extra step that merges its contents into hyper-entries (MBRs) whose diagonal does not exceed δ. This is performed as follows. The entries in S are sorted according to the Hilbert value of their bottom-left MBR corner. For each entry e, we scan the following ones in the sorted order. If merging e with one of them leads to a permissible diagonal, the two entries are replaced in S by a hyper-entry and scanning resumes, potentially merging additional entries into the hyper-entry. This process is repeated for the remaining entries in S.

Based on the resulting S, we produce a set P' of customer representatives as follows. For each entry (normal, conceptual, or hyper-entry) $e \in S$ we derive a representative point $\overline{g}$ located at the *geometric centroid* of e, i.e., the intersection of its MBR diagonals. The representative has *weight* $\overline{g.w}$ equal to the number of customers in the subtree(s) of e. The resulting set P' serves as an approximation of P.

To exemplify CA partitioning, assume that the R-tree of P and parameter δ are as shown in Figure 10 (the R-tree is illustrated both in the spatial domain and as stored physically). 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 into S. This is not the case for e_1, whose pointed entries are accessed. 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 longest 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);

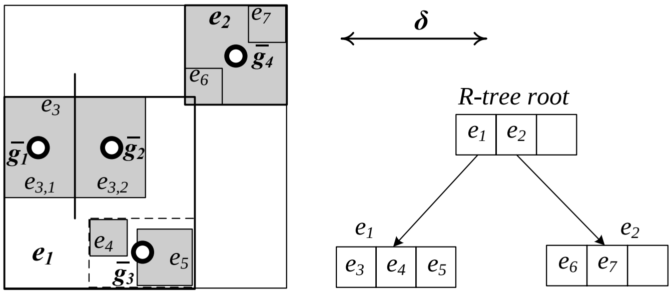


Fig. 10. Customer partitioning

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. The representatives of these groups are points $\overline{g_1}, \overline{g_2}, \overline{g_3}$, and $\overline{g_4}$, and together comprise set P'.

In the concise matching phase, CA computes the optimal matching M' between P' and Q. This is performed using SIA, because (as we demonstrate in our experiments) it is the most efficient among the exact methods. Note that in this setting points in P' also have capacities (the representative weights). This is not a problem, since SIA (as well as NIA and IDA) 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. The matching M' produced by this step will be refined into the final matching M using one of the techniques presented in Section 5.2.

As mentioned earlier, PA proceeds like CA, the difference being that partitioning takes place in Q (utilizing its R-tree) and that concise matching is performed between P and the set of provider representatives Q'. Note that in PA the representative $\overline{g_m}$ of a provider group G_m has capacity $\overline{g_m.k} = \sum_{q \in G_m} q.k$. Before describing the refinement phase, it is worth mentioning that we attempted to combine CA and PA (i.e., to group both P and Q), but this led to very poor matching quality. Thus, we ignore this hybrid method in the following.

5.2 Refinement Phase

In both PA and CA, the input of the refinement phase is 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 PA 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[8], receiving small sets

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

P'' and Q'' as input.

Provider-centric refinement: This approach computes the (next) NN of each $q \in Q''$ in 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''.

Customer-centric 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''.

5.3 Assignment Cost Guarantee

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

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

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

THEOREM 5.1. *The assignment error of either CA or PA is upper bounded by $\gamma \cdot \delta$.*

PROOF. We focus on CA, as the proof for PA follows the same lines. First, note that the 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 $(q, \overline{g})$, where $\overline{g}$ is the representative of p's group. After the replacement, the cost of each pair increases/decreases by at most $\frac{\delta}{2}$ (since the representative $\overline{g}$ of each group G_m lies no further than $\frac{\delta}{2}$ from any customer in G_m). Thus, $\Psi(M'_{CCA}) \leq \Psi(M_{CCA}) + \gamma \cdot \frac{\delta}{2}$.

Note that M'_{CCA} is not necessarily the optimal matching between Q and P' (i.e., the set of customer representatives). 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 $\Psi(M') \leq \Psi(M_{CCA}) + \gamma \cdot \frac{\delta}{2}$.

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

To confirm the tightness of the bound in Theorem 5.1, we demonstrate an example where the upper bound assignment error $\gamma \cdot \delta$ is close to the actual assignment cost error $Err(M)$. We focus on the approximate matching M computed by the CA method; the situation is similar for PA. Consider the example in Figure 11, where $Q = \{q\}$ (there is one provider) and $P = \{p_1, p_2, \cdots, p_{4k}\}$ (there are $4k$ customers). Note that customers $p_1, p_2, \cdots, p_k$ are located at the same position. Similarly, other customers share other locations. Suppose that the capacity of provider q is k.

The CA method groups customers $p_1, p_2, \cdots, p_{2k}$ into entry e_1, and places the remaining customers into entry e_2. For each entry e_i, its diagonal length is exactly

δ. The gray points correspond to entry centroids $\overline{g_i}$ (i.e., customer representatives). Assume that the minimum distance of q from the MBRs of e_1 and e_2 is μ and 2μ, respectively, where μ is a value arbitrarily close to zero. After partitioning into e_1 and e_2, CA's concise matching assigns representative $\overline{g_1}$ to provider q, because $dist(q, \overline{g_1}) < dist(q, \overline{g_2})$. Next, the refinement phase assigns customers $p_{k+1}, p_{k+2}, \cdots, p_{2k}$ to q. Since μ tends to zero, the assignment cost of this approximate matching is $\Psi(M) = k \cdot \delta/\sqrt{2}$. Consider now the optimal matching M_{CCA}; in M_{CCA}, provider q is assigned customers $p_{2k+1}, p_{2k+2}, \cdots, p_{3k}$. Since μ tends to zero, the optimal assignment cost is $\Psi(M_{CCA}) = 0$.

By Theorem 5.1, the upper bound assignment error in the above example is $k \cdot \delta$ because the matching size γ is k. On the other hand, the actual assignment cost error is $Err(M) = \Psi(M) - \Psi(M_{CCA}) = k \cdot \delta/\sqrt{2}$. In other words, the actual error $Err(M)$ is 0.7071 of the upper bound error $k \cdot \delta$, and therefore there exists a problem instance in which Theorem 5.1 provides a reasonably tight bound of the actual error. Note however that, unlike this purposely selected pathological example, in the general case the actual assignment cost error is much lower (as we demonstrate in the experiments).

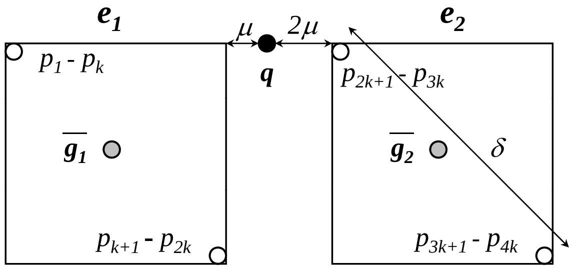


Fig. 11. Tightness example

6. EXPERIMENTS

This section empirically evaluates the performance of our algorithms. All methods were implemented in C++ and experiments were executed on an Intel Core 2 Duo E7200 machine with 4 GBytes RAM, running on Ubuntu 8.04. Section 6.1 describes the datasets, the parameters under investigation, and other settings used in our evaluation. Section 6.2 investigates the performance of our algorithms on optimal CCA computation. Section 6.3 evaluates the efficiency of our incremental maintenance techniques under different update models. Section 6.4 explores the efficiency and assignment cost error of our approximate CCA methods.

6.1 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) [Brinkhoff 2002], using the generator of [Yiu and Mamoulis 2004]. 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 were normalized to lie in a $[0, 1000]^2$ space.

In Section 6.3, we use two update models that simulate different practical scenarios. The first is *Movement*, which simulates small movements in the road network; each update is a customer movement to a random position within its current edge. The second is *Insertion/Deletion*, where arbitrary insertions and deletions occur in P. Customers are deleted randomly, and an equal number of new ones are inserted at locations that follow the original distribution of P.

By default, the capacity k of all $q \in Q$ is 80 and the dataset cardinalities are $|Q| = 1$K and $|P| = 100$K. Table II shows the parameters under investigation and their examined values. Both datasets are stored in main memory and indexed by R-trees with 1 KByte node size. Unless otherwise specified, for each experiment we report the memory usage (i.e., $|E_{sub}|$, number of edges in the subgraph) and the CPU time.

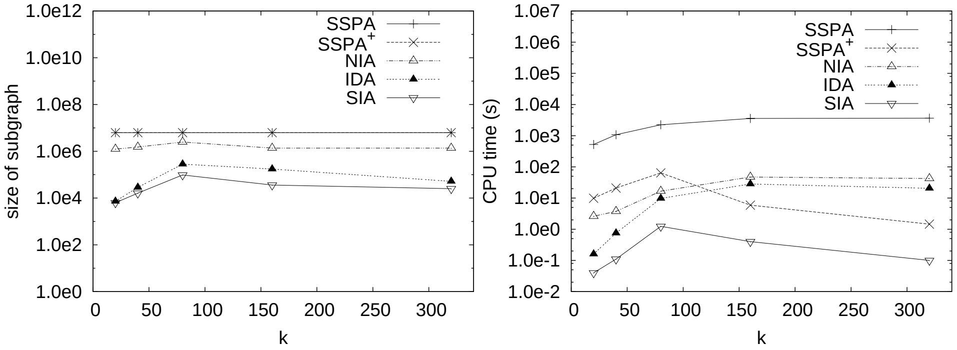
Parameter	**Default**	**Range**
$|Q|$ (in thousands)	1	0.25, 0.5, 1, 2.5, 5
$|P|$ (in thousands)	100	25, 50, 100, 150, 200
Capacity k	80	20, 40, 80, 160, 320
Diagonal δ	5, 10	2.5, 5, 10, 20, 40, 80
Update ratio	1%, 10%	1%, 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 100%

Table II. System parameters

6.2 Experiments on Optimal Assignment

SSPA and SSPA$^+$ require that the complete flow graph is stored in main memory. For our default setting this leads to space requirements that exceed the available system memory. To provide, however, an intuition about (i) the inherent complexity of the problem and (ii) the relative performance of SSPA/SSPA$^+$ versus our algorithms, we experiment on a smaller problem. We generated Q and P as described in Section 6.1, with $|Q| = 250$ and $|P| = 25$K, so that the flow graph fits in main memory. SSPA and SSPA$^+$ do not utilize an index, as they involve no spatial searches. Figure 12 shows $|E_{sub}|$ and the CPU time (in logarithmic scale) versus the provider capacity k in this small problem. Our best method, SIA, is one to two (three to four) orders of magnitude faster than SSPA$^+$ (SSPA, respectively) in all cases. Moreover, SIA has two to three orders smaller space requirements than SSPA and SSPA$^+$. For under-capacity problems, the second fastest method is IDA. For over-capacity ones, however, both NIA and IDA are slower than SSPA$^+$, because they build on SSPA and follow its trend.

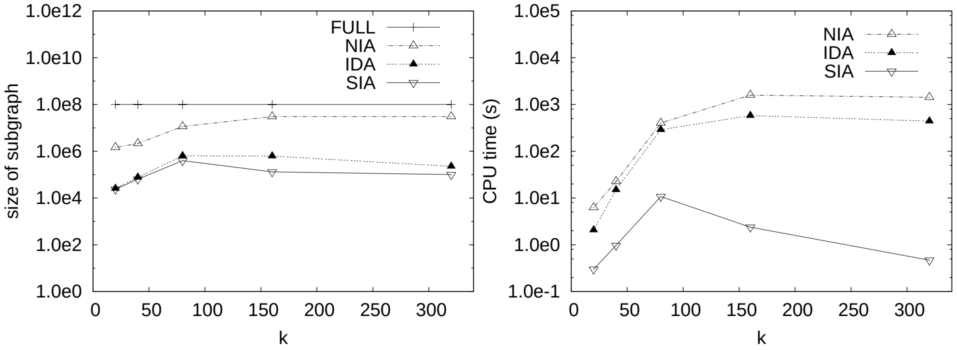
In the remaining experiments, we focus on large problem instances, excluding the inapplicable SSPA and SSPA$^+$. In Figure 13(a) we measure $|E_{sub}|$ as we vary k and set the remaining parameters to their default values. We include the complete bipartite graph size $|E| = |Q| \cdot |P|$ (indicated by "FULL") as a reference for the



(a) $|E_{sub}|$

(b) CPU time

Fig. 12. Performance vs. k, $|Q| = 250$, $|P| = 25$K



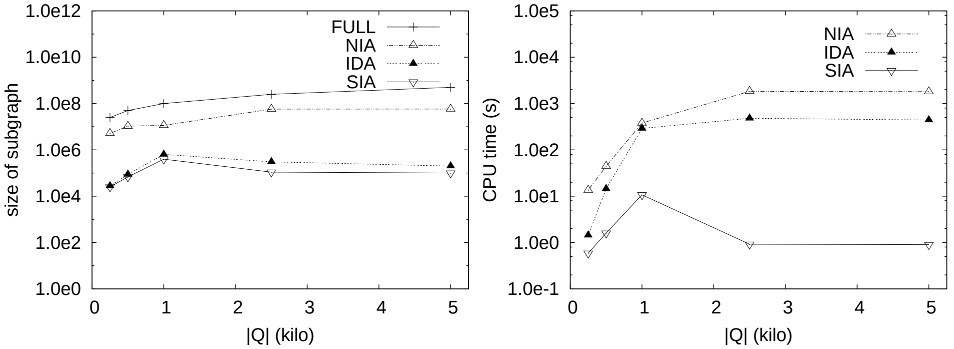
(a) $|E_{sub}|$

(b) CPU time

Fig. 13. Performance vs. k, $|Q| = 1$K, $|P| = 100$K

space requirements of SSPA/SSPA$^+$. Due to the application of Theorems 3.2 and 3.7, our algorithms (NIA, IDA, SIA) use/store only a fragment of the complete bipartite graph. IDA/SIA prune more edges than NIA, because they exploit the distinction between full and non-full nodes in E_{sub}. The difference becomes more obvious in the over-capacity case, because IDA/SIA additionally benefit from the role reversal between P and Q.

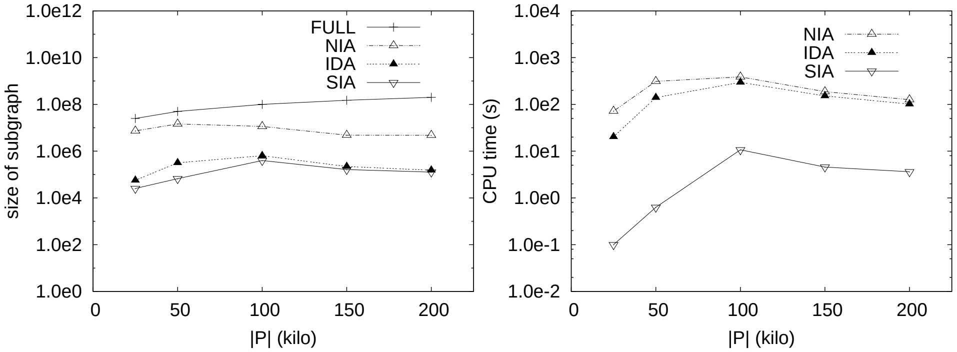
Figure 13(b) shows the execution time in the previous experiment. In the under-capacity case, the processing cost for all methods increases with the provider capacity, because the number of iterations is proportional to k (i.e., $\gamma = |Q| \cdot k$). In the over-capacity case, γ is independent of k and equal to $|P|$. However, when k grows, the capacity constraint becomes looser and the problem becomes easier. Practically, this means that more customers are assigned to their closest provider. Another observation is that SIA vastly outperforms IDA, although they use similar pruning techniques. The reason is the lack of source/sink nodes in SIA. This accelerates *sp* computations because (i) the Dijkstra heap is kept small, and (ii) fewer



(a) $|E_{sub}|$

(b) CPU time

Fig. 14. Performance vs. $|Q|$, $k = 80$, $|P| = 100$K



(a) $|E_{sub}|$

(b) CPU time

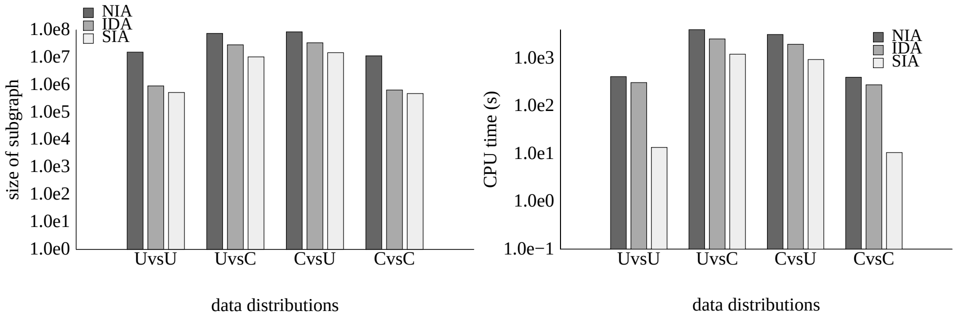
Fig. 15. Performance vs. $|P|$, $k = 80$, $|Q| = 1$K

heap operations are required.

Figure 14 investigates the effect of service provider cardinality $|Q|$. The relative performance of the algorithms is consistent with our observations in Figure 13; IDA and SIA prune more edges than NIA, while SIA additionally benefits from its simplified flow graph. The size of E_{sub} increases with $|Q|$, but saturates when $k \cdot |Q|$ exceeds $|P|$, since the optimal assignment is found before long edges (from providers to their furthest neighbors and vice versa) are examined.

Figure 15 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.

Figure 16 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 6.1. For example, label "UvsC" on the



(a) $|E_{sub}|$

(b) CPU time

Fig. 16. Performance under different distributions (default $k, |Q|, |P|$)

horizontal axis corresponds to uniform service providers and clustered customers. SIA is the method of choice for all distribution combinations. 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, again, need to expand their search ranges very far. In both cases, E_{sub} is larger and the problem becomes more complex (compared to similarly distributed Q and P).

In the next experiment, we study the performance of our algorithms on various real datasets. From the Geographic Names Information System (GNIS)[9], we obtained three real point sets of building locations in the United States: School (SCH, 172K points), Church (CHU, 181K points), and Populated Place (POP, 177K points). Table III shows the subgraph size and CPU time of NIA, IDA and SIA for different combinations of these datasets. All parameters are set to their default values; for each tested combination of Q and P we randomly selected points from the corresponding real datasets so that their sizes are equal to the default values ($|Q| = 1$K and $|P| = 100$K). The performance trends are similar to Figure 16. When Q and P are drawn from different datasets (i.e., follow a different distribution), the size of E_{sub} and the CPU time increase because some customers are assigned to very distant servers. SIA outperforms NIA and IDA in all cases, using several times (or orders) smaller E_{sub} and requiring a fraction of their CPU time.

6.3 Experiments on Assignment Maintenance

In this section we evaluate CCA maintenance in the *Movement* and *Insertion/Deletion* update models. We compare the processing time of the incremental technique in Section 4 (denoted by "SIA Update") with SIA re-computation from scratch (illustrated as "SIA" in the charts).

[9] http://geonames.usgs.gov/domestic/

Q	P	**NIA**	**IDA**	**SIA**	**NIA**	**IDA**	**SIA**
		size of subgraph			**CPU time (s)**		
SCH	SCH	9.7E+07	7.0E+05	3.8E+05	660.81	325.93	7.18
SCH	CHU	9.9E+07	3.5E+07	1.5E+07	4589.99	2861.52	1206.15
SCH	POP	9.8E+07	1.7E+07	7.8E+06	3070.91	1799.09	707.37
CHU	SCH	1.0E+08	4.0E+07	2.3E+07	7179.93	4891.94	3185.43
CHU	CHU	1.0E+08	2.6E+06	4.9E+05	673.24	337.74	10.56
CHU	POP	6.1E+07	3.6E+07	2.3E+07	5114.81	3508.26	2018.18
POP	SCH	1.0E+08	1.2E+07	5.6E+06	1969.50	1199.09	366.06
POP	CHU	1.0E+08	3.2E+07	7.9E+06	3068.73	1754.63	610.17
POP	POP	9.8E+07	8.5E+05	5.0E+05	595.86	248.33	10.76

Table III. Performance on real datasets (default $k, |Q|, |P|$)

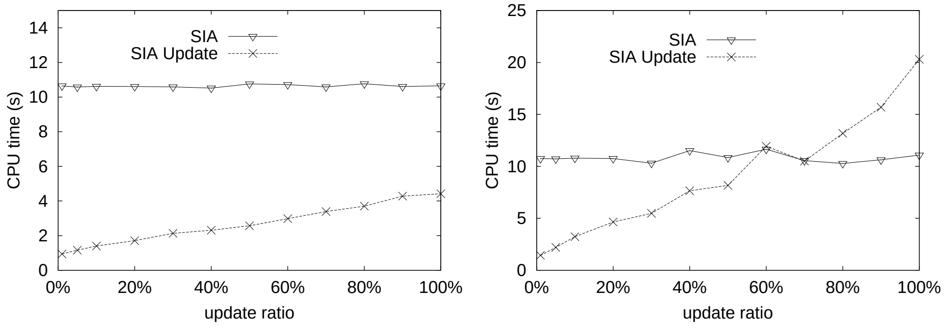
Figure 17(a) shows the execution time as a function of the update ratio for *Movement* updates. SIA Update is 2.4 time faster than SIA even when the update ratio is 100%. This is an impressive result, considering that all previously assigned pairs are invalidated (i.e., their edges are reversed when reforming E_{sub}). As the customers move within their current road segment, their distances from the providers change only slightly. This means that most of the $|U| = |P|$ iterations are successfully performed in stage one of our update process (see Section 4.2.1), and the reconstructed E_{sub} does not have to be expanded much in stages two and three. Another reason is that E_{sub} in SIA Update is built to the largest degree by range searches (Lines 1–3 in Algorithm 7), which are faster than incremental NN searches.

In Figure 17(b) we investigate the effect of the update ratio in the *Insertion/Deletion* model. The incremental SIA method is less efficient than in the *Movement* case, because the inserted customers typically do not lie close to the deleted ones. Thus, E_{sub} needs to be expanded significantly in stages two and tree. Regardless of this fact, however, our update method is faster than SIA re-computation for update ratios up to 60%.

In Figure 17, as well as in the remainder of this section, we focus on CPU time and omit $|E_{sub}|$ measurements. The size of the flow subgraph in SIA Update is similar to SIA (as reported in Section 6.2). For completeness, we mention that our incremental method has between 0.1% and 2% (between 1% and 11%) larger E_{sub} than SIA in all cases in Figure 17(a) (in Figure 17(b), respectively).

In the remaining experiments, each chart presents results for two update ratios; 1% and 10%. There are two lines for the incremental SIA (denoted by "SIA Update-1%" and "SIA Update-10%"), and one for SIA re-computation[10]. In Figure 18 we vary k, while setting the remaining parameters to their default values shown in Table II. Similarly, in Figures 19 and 20 we vary $|Q|$ and $|P|$, respectively. In the default setting, when the update ratio is 10%, incremental SIA is 7.5 (3.4) times faster than SIA re-computation for *Movement* (for *Insertion/Deletion*) updates. However, when $\sum_{q \in Q} q.k \ll |P|$ or $\sum_{q \in Q} q.k \gg |P|$, the benefits of incremental SIA are not as significant. The reason is that in these cases, the problem becomes

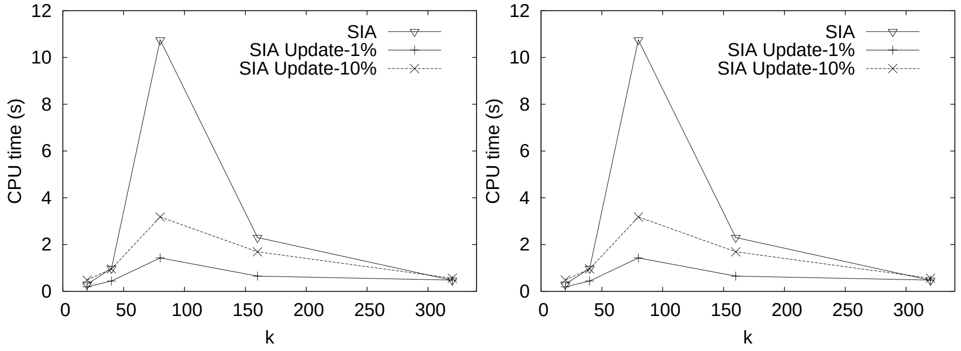
[10] The reported measurements for SIA re-computation are average values for both update ratios, as its performance is practically unaffected by the number of updates.



(a) Movements

(b) Insertions/Deletions

Fig. 17. Maintenance performance vs. update ratio (default k, $|Q|$, $|P|$)



(a) Movements

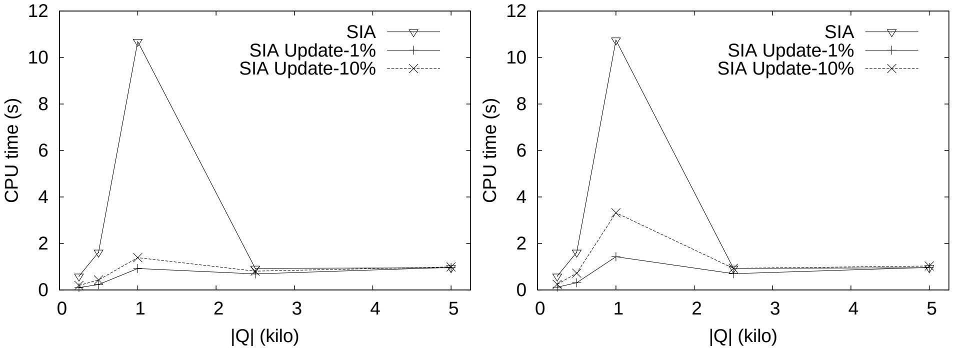
(b) Insertions/Deletions

Fig. 18. Maintenance performance vs. k, $|P| = 100\text{k}$, $|Q| = 1\text{K}$

easier (equivalently, the running time becomes short), leaving little space for improvement; note that in the extreme cases (e.g., for k=20 and k=320 in Figure 18) the reconstruction of E_{sub} alone accounts for 70% of the total processing time in SIA Update.

In Figure 21 we investigate the effect of different Q and P distributions on maintenance performance. When Q and P follow different distributions, the benefits of incremental SIA are more pronounced. In these cases, E_{sub} is very large (see Figure 16) and shortest path computation becomes costlier. This fact amplifies the advantage of SIA Update to execute only $|U|$ iterations instead of γ.

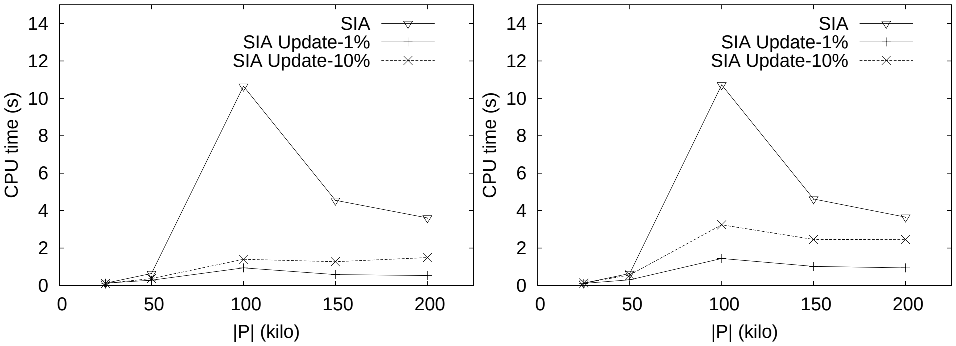
To summarize the CCA maintenance experiments, our incremental technique is several times faster than SIA re-computation from scratch in near equi-capacity problems. This is always the case for *Movement* updates and for up to 60% update ratios in the *Insertion/Deletion* model. In seriously under-capacity or over-capacity cases, the problem becomes easier overall, and the improvement is not as significant.



(a) Movements

(b) Insertions/Deletions

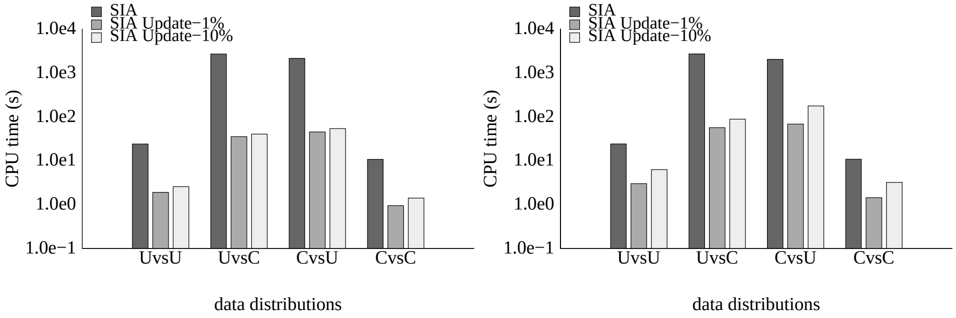
Fig. 19. Maintenance performance vs. $|Q|$, $k = 80$, $|P| = 100\text{K}$



(a) Movements

(b) Insertions/Deletions

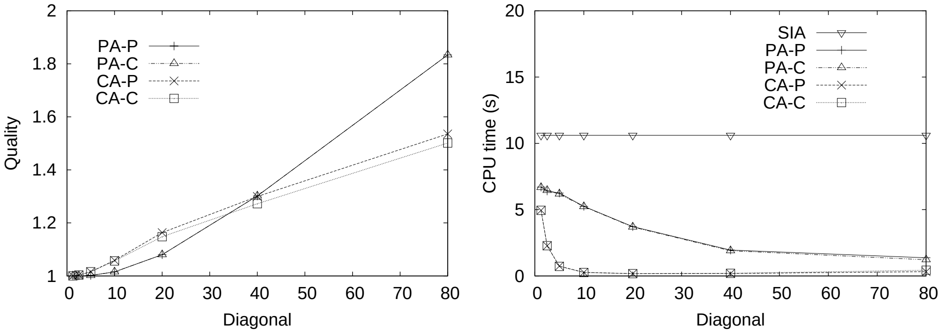
Fig. 20. Maintenance performance vs. $|P|$, $k = 80$, $|Q| = 1\text{K}$



(a) Movements

(b) Insertions/Deletions

Fig. 21. Maintenance performance under different distributions (default k, $|Q|$, $|P|$)



(a) quality (b) CPU time

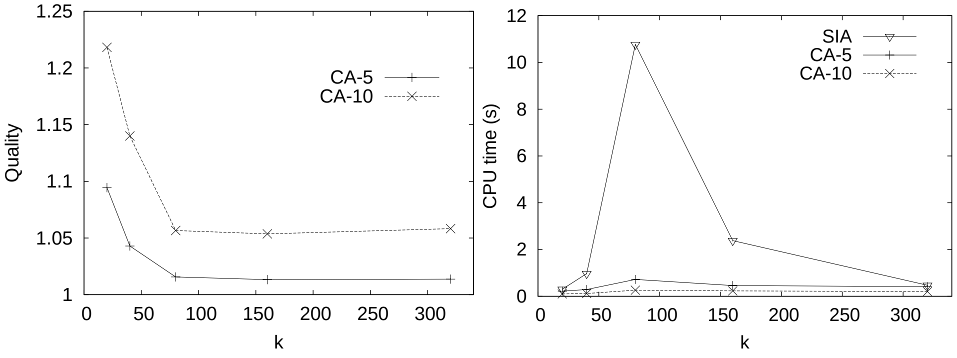
Fig. 22. Approximation quality vs. δ (default $k, |Q|, |P|$)

6.4 Experiments on Approximate Assignment

In this section we evaluate the accuracy of our approximate CCA methods (i.e., PA and CA) presented in Section 5, and compare their execution time with SIA (the fastest 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 PA and CA, we implemented both provider-centric refinement and customer-centric refinement (indicated by a "P" or "C" after the method's name in the charts).

Figure 22 shows the approximation quality and the running time as a function of the diagonal parameter δ (used in the partitioning phase). As expected, accuracy and execution cost drop with δ for both CA and PA. However, CA is significantly faster. The reason is that the flow subgraph in its concise matching phase is much smaller than in PA. For the same value of δ, the number of customer/provider groups (and, thus, the number of representatives) in CA and PA is similar. However, each representative in CA (PA) has a number of incident edges that is proportional to $|Q|$ (proportional to $|P|$, respectively). Therefore, E_{sub} in PA is roughly $|P|/|Q|$ times larger than in CA. For small values of δ, PA yields a low approximation error, but its running time is too high (around half of SIA). On the other hand, CA with a small δ (5 and 10) achieves great performance improvement (15 times and 41.5 times) over SIA, while producing a matching only marginally worse than the optimal. Between its variants, CA-C (i.e., CA with customer-centric refinement) performs better. In the remaining experiments we focus on CA-C with $\delta = 5$ and $\delta = 10$ (denoted by "CA-5" and "CA-10"), because these two settings achieve the best efficiency/accuracy trade-offs.

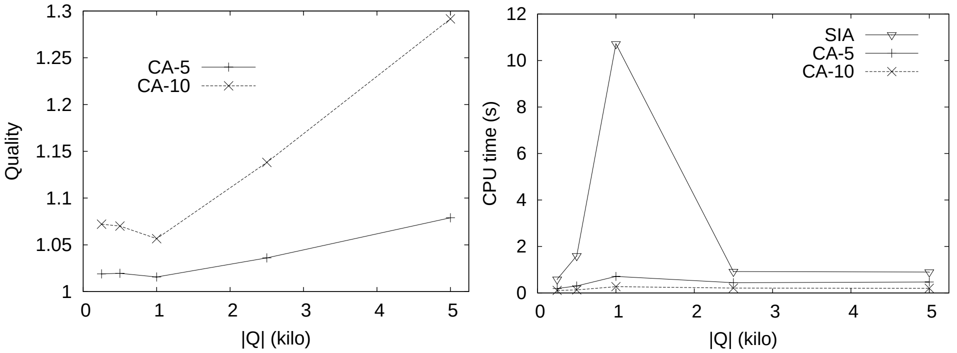
Next, we evaluate the approximate solutions varying one of k, $|Q|$, and $|P|$, while setting the other two parameters to their default values shown in Table II. In Figure 23, we examine the effect of k and observe that the approximation quality improves when it grows. As k increases, the providers are assigned more distant customers; i.e., both $\Psi(M)$ and $\Psi(M_{CCA})$ grow. On the other hand, the customer group MBRs remain constant (as δ is fixed) and, hence, the relative error of a suboptimally assigned customer drops. When $k \geq 80$, the error is lower than 1.5% and 6% for CA-5 and CA-10, respectively. The execution time of CA follows the



(a) quality

(b) CPU time

Fig. 23. Approximation performance vs. k, $|Q| = 1\text{K}$, $|P| = 100\text{K}$



(a) quality

(b) CPU time

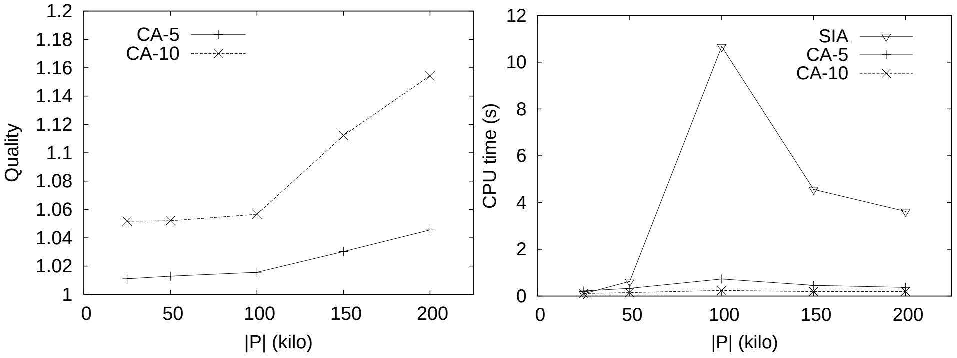
Fig. 24. Approximation performance vs. $|Q|$, $k = 80$, $|P| = 100\text{K}$

trend of SIA, due to its SIA-based concise matching.

Figure 24 investigates the effect of $|Q|$ on the approximation quality and running time. The error of CA increases with $|Q|$, because the more service providers around a customer group, the higher the chances for a suboptimal pair in M. The quality of CA-10 is more sensitive to $|Q|$, because more wrong assignments are made by its concise matching.

Next, in Figure 25, we vary the number of customers $|P|$. A large $|P|$ reduces the accuracy of CA, because the customer groups become more crowded, implying a coarser partitioning and a worse approximation. On the other hand, when $|P|$ is only 25K, the concise matching of CA degenerates to SIA (i.e., most groups contain a single customer), leading to a marginally higher processing cost than SIA due to the unnecessary overheads for partitioning and refinement.

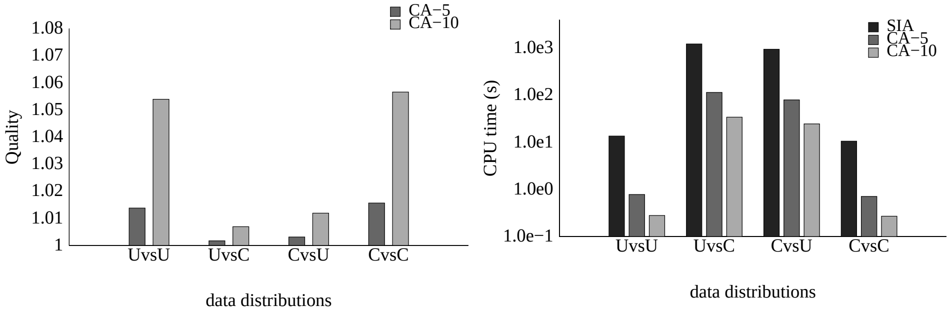
Figure 26 examines the effect of different Q and P distributions. CA-5 and CA-10 perform well in terms of running time and accuracy for all combinations. For differently distributed Q and P, CA-5 and CA-10 have less than 1.5% error while



(a) quality

(b) CPU time

Fig. 25. Approximation performance vs. $|P|$, $k = 80$, $|Q| = 1$K



(a) quality

(b) CPU time

Fig. 26. Approximation performance under different distributions (default $k, |Q|, |P|$)

they are at least one order of magnitude faster than SIA. CA is less accurate (but faster) for similarly distributed Q and P, because multiple alternatives exist for each provider-customer assignment with similar costs (i.e., distances) each.

In summary, CA with customer-centric refinement is our best approximate method. It typically computes a near optimal matching, while being several times faster than SIA. Importantly, CA is also very robust, avoiding SIA's undesirable peak (in processing time) for near equi-capacity problems.

7. CONCLUSION

In this paper we study the *capacity constrained assignment* (CCA) problem, which retrieves the matching (between two spatial datasets) 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 algorithms that gradually expand the search space and effectively prune it. In addition to one-time (static) CCA methods, we propose incremental techniques that amend an existing assignment subject to a batch of location updates, in order to retain optimality. Finally, we develop approximate

CCA solutions that provide a trade-off between computation cost and matching quality.

An interesting direction for future work is the predictive CCA problem. Here, each customer (i.e., every point of one dataset) follows a predefined trajectory, and the challenge is to compute all future optimal assignments along with their respective time intervals. An additional challenge in this model, is to allow updates in the predefined trajectories, requiring incremental (on-the-fly) amendments of the predictions made previously. The predictive CCA problem is motivated by intelligent transportation systems, where position anticipation and trajectory mining are possible.

REFERENCES

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

Balinski, M. 1985. Signature Methods for the Assignment Problem. *Operations Research 33*, 527–537.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD Conference*. 322–331.

Bertsekas, D. P. 1981. A New Algorithm for the Assignment Problem. *Mathematical Programming 21,* 1 (December), 152–171.

Bertsekas, D. P. 1988. The Auction Algorithm: A Distributed Relaxation Method for the Assignment Problem. *Ann. Oper. Res. 14,* 1-4, 105–123.

Bially, T. 1969. Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction. *IEEE Transactions on Information Theory 15,* 6, 658–664.

Brinkhoff, T. 2002. A Framework for Generating Network-Based Moving Objects. *GeoInformatica 6,* 2, 153–180.

Corral, A., Manolopoulos, Y., Theodoridis, Y., and Vassilakopoulos, M. 2000. Closest Pair Queries in Spatial Databases. In *SIGMOD Conference*. 189–200.

Derigs, U. 1981. A Shortest Augmenting Path Method for Solving Minimal Perfect Matching Problems. *Networks 11,* 4, 379–390.

Ester, M., Kriegel, H.-P., and Xu, X. 1995. Knowledge Discovery in Large Spatial Databases: Focusing Techniques for Efficient Class Identification. In *SSD*. 67–82.

Gabow, H. N. and Tarjan, R. E. 1991. Faster Scaling Algorithms for General Graph-Matching Problems. *J. ACM 38,* 4, 815–853.

Gale, D. and Shapley, L. S. 1962. College Admissions and the Stability of Marriage. *The American Mathematical Monthly 69,* 1, 9–15.

Goldberg, A. V. and Kennedy, R. 1995. An Efficient Cost Scaling Algorithm for the Assignment Problem. *Math. Program. 71*, 153–177.

Gusfield, D. and Irving, R. W. 1989. *The Stable Marriage Problem: Structure and Algorithms.* MIT Press.

Guttman, A. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*. 47–57.

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

Hung, M. 1983. A Polynomial Simplex Method for the Assignment Problem. *Operations Research 31*, 595–600.

Irving, R. W., Manlove, D., and Scott, S. 2003. Strong Stability in the Hospitals/Residents Problem. In *STACS*. 439–450.

Kuhn, H. W. 1955. The Hungarian Method for the Assignment Problem. *Naval Research Logistic Quarterly 2*, 83–97.

Mills-Tettey, G. A., Stentz, A. T., and Dias, M. B. 2007. The Dynamic Hungarian Algorithm for the Assignment Problem with Changing Costs. Tech. Rep. CMU-RI-TR-07-27, Robotics Institute. July.

Mouratidis, K., Papadias, D., and Papadimitriou, S. 2008. Tree-Based Partition Querying: A Methodology for Computing Medoids in Large Spatial Datasets. *VLDB J. 17,* 4, 923–945.

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

Murty, K. G. 1992. *Network Programming.* Prentice-Hall, Inc.

Okabe, A., Boots, B., Sugihara, K., and Chiu, S. N. 2000. *Spatial Tessellations : Concepts and Applications of Voronoi Diagrams*, second ed. Wiley.

Orlin, J. B. and Lee, Y. 1993. QuickMatch–A Very Fast Algorithm for the Assignment Problem. Working papers 3547-93., Massachusetts Institute of Technology (MIT), Sloan School of Management.

Papadias, D., Tao, Y., Fu, G., and Seeger, B. 2005. Progressive Skyline Computation in Database Systems. *ACM Trans. Database Syst. 30,* 1, 41–82.

Roussopoulos, N., Kelley, S., and Vincent, F. 1995. Nearest Neighbor Queries. In *SIGMOD Conference.* 71–79.

Sellis, T. K., Roussopoulos, N., and Faloutsos, C. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB.* 507–518.

Spivey, M. Z. and Powell, W. B. 2004. The Dynamic Assignment Problem. *Transportation Science 38,* 4, 399–419.

Theodoridis, Y., Stefanakis, E., and Sellis, T. K. 2000. Efficient Cost Models for Spatial Queries Using R-Trees. *IEEE Trans. Knowl. Data Eng. 12,* 1, 19–32.

Toroslu, I. H. and Üçoluk, G. 2007. Incremental Assignment Problem. *Inf. Sci. 177,* 6, 1523–1529.

U, L. H., Mamoulis, N., and Yiu, M. L. 2008a. Computation and Monitoring of Exclusive Closest Pairs. *IEEE Trans. Knowl. Data Eng. 20,* 12, 1641–1654.

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

Vygen, J. 2004. *Approximation Algorithms forFacility Location Problems (Lecture Notes).* University of Bonn.

Wong, R. C.-W., Tao, Y., Fu, A. W.-C., and Xiao, X. 2007. On Efficient Spatial Matching. In *VLDB.* 579–590.

Yiu, M. L. and Mamoulis, N. 2004. Clustering Objects on a Spatial Network. In *SIGMOD Conference.* 443–454.

Zhang, D., Du, Y., Xia, T., and Tao, Y. 2006. Progressive Computation of the Min-Dist Optimal-Location Query. In *VLDB.* 643–654.