4-2010

Continuous Spatial Assignment of Moving Users

Hou U LEONG
University of Hong Kong

Kyriakos MOURATIDIS
kyriakos@smu.esu.sg, kyriakos@smu.edu.sg

Nikos MAMOULIS
University of Hong Kong

Published in VLDB Journal , 2010, 19 (2), 141-160. doi: 10.1007/s00778-009-0144-3

Leong Hou U · Kyriakos Mouratidis · Nikos Mamoulis

Continuous Spatial Assignment of Moving Users

Received: date / Accepted: date

Abstract Consider a set of servers and a set of users, where each server has a *coverage region* (i.e., an area of service) and a *capacity* (i.e., a maximum number of users it can serve). Our task is to assign every user to one server subject to the coverage and capacity constraints. To offer the highest quality of service, we wish to minimize the average distance between users and their assigned server. This is an instance of a well-studied problem in operations research, termed *optimal assignment*. Even though there exist several solutions for the static case (where user locations are fixed), there is currently no method for dynamic settings.

In this paper, we consider the *continuous assignment problem* (CAP), where an optimal assignment must be constantly maintained between mobile users and a set of servers. The fact that the users are mobile necessitates real-time reassignment so that the quality of service remains high (i.e., their distance from their assigned servers is minimized). The large scale and the time-critical nature of targeted applications require fast CAP solutions. We propose an algorithm that utilizes the geometric characteristics of the problem and significantly accelerates the initial assignment computation and its subsequent maintenance. Our method applies to different cost functions (e.g., average squared distance) and to any Minkowski distance metric (e.g., Euclidean, L_1 norm, etc).

Leong Hou U
Department of Computer Science
University of Hong Kong
E-mail: hleongu@cs.hku.hk

Kyriakos Mouratidis
School of Information Systems
Singapore Management University
E-mail: kyriakos@smu.edu.sg

Nikos Mamoulis
Department of Computer Science
University of Hong Kong
E-mail: nikos@cs.hku.hk

1 Introduction

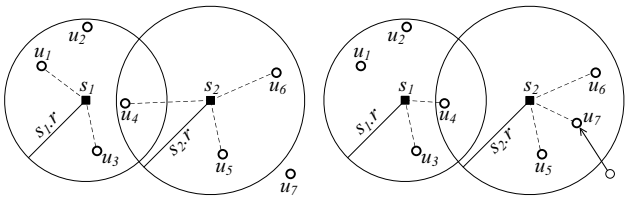
Consider a set of users U and a set of servers S. Each $s \in S$ has a coverage radius $s.r$ and a capacity $s.c$, implying that s can serve users within distance $s.r$ from its location, whose total number does not exceed $s.c$. Our objective is to construct a valid assignment $A \subseteq U \times S$ between users and servers such that each user u in A is assigned to exactly one server s and the maximum possible number of users are served (i.e., the cardinality of A is maximized). Furthermore, among the assignments of maximal size, we want to find the one that minimizes $avg_{(u,s)\in A} dist(u, s)$, i.e., the average distance between the users and their assigned servers. This optimization criterion is in accordance with the objectives of access point placement problems in wireless networks (see for example [20]). The primary *maximality* requirement demands that no user is left unassigned, unless he/she is outside the coverage region of all servers or it is impossible to assign him/her to a server without leaving some other user unassigned. Minimizing the average distance of served users is a secondary optimization criterion. In this paper, we consider the *continuous assignment problem* (CAP), where an optimal assignment, as defined above, must be constantly reported as the users arbitrarily change locations and log in/out.

Continuous assignment is essential to several applications, including examples in mobile communications, wireless networking, etc. For instance, set S could correspond to the base stations of a telecommunications company, and U to cell-phone users. The company wishes to serve as many users as possible, and to provide them with the best quality of service (i.e., signal strength). The base stations have coverage and capacity constraints and, thus, the dynamic assignment of mobile users to the stations is a CAP instance.

As another example, consider a Wi-Fi service that provides Internet connection via a set S of wireless access points (WAPs). Clients registered with the service use their portable devices (e.g., PDAs) to access the Internet. Every WAP $s \in S$ can serve a limited number of users ($s.c$), who must additionally be within a certain radius ($s.r$) from it. Under these

constraints, the service assigns each user to a WAP. Primarily, it wishes to serve as many users as possible. Moreover, it needs to minimize the average distance between users and their assigned WAP, in order to achieve high overall connectivity. Taking into account that users may unpredictably move and log in/out, this is a continuous assignment problem.

To illustrate CAP, consider two sets $U = \{u_1, ..., u_7\}$ and $S = \{s_1, s_2\}$, as shown in Figure 1(a). Let $s_1.c = 2, s_2.c = 3$ and their disks of coverage be as illustrated. The dashed lines indicate the optimal assignment A between U and S; i.e., users u_1, u_3 are served by s_1 and users u_4, u_5, u_6 by s_2. User u_7 is unassigned, as it is outside both coverage disks. On the other hand, u_2 is unassigned because s_1 could not possibly serve him/her without dropping another user. Note that u_4 is assigned to s_2 even though s_1 is closer; s_1 cannot serve u_4, because this requires dropping either u_1 or u_3, and leads to a total of only 4 served users (instead of 5).



(a) Initial assignment (b) Result maintenance

Fig. 1 Continuous optimal assignment

Figure 1(b) continues the previous example, assuming that after the initial assignment, user u_7 moves. His/her new location is in the coverage region of s_2, but the latter is full. Since u_7 is closer to s_2 than u_4, s_2 drops u_4 and admits u_7 to reduce the average distance in the system. In turn, for the same reason, u_4 is taken on by s_1, which leaves u_1 unassigned. The new optimal assignment A' (indicated by dashed lines) shows that a location update may affect distant users and servers.

The above scenario considers the *strict* CAP case, where a stationary user may be completely dropped or handed over by his/her current server when some other user's movement allows for a better assignment. Depending on the application, this may or may not be desirable. In Figure 1(b), for example, even though the new assignment minimizes the average distance in the system, users u_1 and u_4 experience a disruption of service due to loss of access or server swap. This might be unacceptable in applications like the assignment between WAPs and PDAs, discussed earlier. We, thus, distinguish another problem variant, termed *connected* CAP. Connected CAP requires that currently assigned users who remain within the coverage region of their server will not be dropped by it, while the reported assignment is optimal subject to this constraint.

Optimal assignment is an old and well-studied topic in operations research. Existing methods, however, are explicitly designed for static environments. In this paper, we propose a CAP algorithm applicable to both the strict and the connected variants, called *continuous spatial assignment* (CSA). CSA exploits the geometric properties of the problem and follows the incremental execution paradigm to reduce the computation cost. The main idea behind our approach is to identify, by local decisions, user-server pairs that certainly belong to the optimal assignment and users or servers that cannot participate in the solution. Subsequently, the resulting CAP (with U, S, and the capacities of servers reduced accordingly) is split into smaller, independent problems, of lower complexity. These problems are eventually solved by an off-the-shelf optimal assignment algorithm. An additional advantage of the problem decomposition is that these independent tasks can be parallelized, leading to an even lower computation cost. CSA applies to coverage regions of arbitrary shape and can be easily adapted to different cost functions (provided that they are monotonically increasing with distance) and to any Minkowski distance metric.

The rest of the paper is structured as follows. Section 2 reviews related work. Sections 3 and 4 present the first-time assignment computation and its subsequent maintenance, respectively. Section 5 describes data-structures and implementation techniques in CSA. Section 6 experimentally evaluates our approach, while Section 7 concludes the paper.

2 Related Work

In this section, we survey related work on optimal assignment (Section 2.1), continuous query evaluation over moving objects (Section 2.2), and resource allocation methods for large spatial datasets (Section 2.3).

2.1 Optimal Assignment

The *optimal assignment problem* (also known as linear assignment) is one of the oldest problems in operations research [2,25]. Algorithms proposed for its processing exploit a reduction to the *minimum cost flow problem* (MCF). In the MCF formulation, the *flow network* is a weighted, directed graph $G(V, E)$. The set of vertices V is $U \cup S \cup \{source, sink\}$, where $source$ and $sink$ are two fictitious vertices.

E is the set of arcs in the flow network. Each arc has a *weight* and a *capacity* (note that the arc capacity is a concept different from the server capacity). For each user u, E includes an arc from u to all servers s whose coverage region contains u; the weight of arc $\overrightarrow{us}$ is $dist(u, s)$[1] and its capacity is one, implying that pair $\langle u, s\rangle$ can appear at most once in the assignment. Also, there is an arc from $source$ to every $u \in U$, with weight zero and capacity one, imposing the constraint that u can be assigned to at most one server.

[1] In the general MCF formulation, the weight is defined as the cost of assigning u to s, i.e., it could be $dist^2(u, s)$ or another value, depending on the optimization function.

E additionally contains an arc from every $s \in S$ to $sink$, with weight zero and capacity equal to $s.c$, which models the constraint that s can serve at most $s.c$ users.

Figure 2 shows the flow network for the example in Figure 1(a). The flow network for Figure 1(b) is identical, except for the inclusion of one more arc from u_7 to s_2 (since u_7 is now inside the coverage disk of s_2). Omitting the details, the MCF solution is the *maximal* subset $A \subseteq E$ of $\vec{us}$ arcs that have the *minimum* summed weight and respect the server capacities; each $\vec{us} \in A$ indicates the assignment of u to s. Note that although the solution to an MCF may generally include real flows (which in our case would imply that a user is partially assigned to multiple severs), this is not the case when all arc capacities are integer, i.e., the solution is guaranteed to include only integer flows [2,25] (equivalently, to perform only complete assignments).

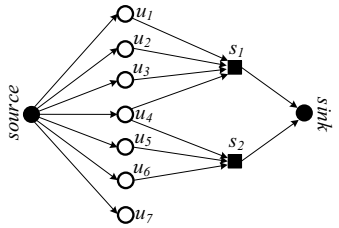


Fig. 2 Flow network example

There are several algorithms to solve the MCF problem defined above, including adaptations of the primal simplex (linear programming) method [14], signature [4] and relaxation [5,6] techniques. The Hungarian[2] [19,24] and the successive shortest path (SSP) [9] algorithms are the most popular and have the lowest worst-case time complexity of $O(n(m + n \log n))$, where n is the number of vertices in V and m the number of arcs in E; lower worst-case bounds are possible for special cases of the problem (not including ours), using the cost-scaling technique [10,13].

The Hungarian algorithm uses a $N \times M$ matrix (where N and M are the numbers of users and servers, respectively), each entry of which corresponds to the distance between a user and a server. During execution, entries of the table are marked and unmarked, and its rows and columns are covered and uncovered. Upon termination, if a table entry is marked, the corresponding user-server pair is included in the optimal assignment A.

SSP iteratively finds shortest paths in the flow network from $source$ to $sink$. After every shortest path computation, the direction of some arcs is reversed and their weights are updated before the next iteration. Upon termination, A is formed by $\langle u, s \rangle$ pairs corresponding to all $\vec{su}$ (i.e., reversed) arcs. SSP requires less space than the Hungarian algorithm for general flow networks, since the latter's table has $O(MN)$ size regardless of the denseness of the network.

[2] Also referred to as Munkres assignment algorithm.

Asymptotic performance aside, the most efficient algorithm for sparse flow networks is *QuickMatch* [26]. *QuickMatch* is basically SSP enhanced by several heuristics and optimizations, the most important being the elimination of $source$ and $sink$ (along with their incident arcs) to accelerate shortest path computations. In our implementation, we use the *QuickMatch* enhancements, but generally refer to the algorithm as SSP in the rest of the paper. We do not elaborate more on SSP, since we use it as an off-the-shelf black box in our method.

[3] proposes a computational geometric approach for a special case of optimal assignment. It is based on *power diagrams*, a variation of the Voronoi diagram. This method computes a weight $s.w$ for every server such that the assignment of every user u to the server s with the smallest $pow(u, s) = dist^2(u, s) - s.w$ minimizes the summed squared distance (between users and their assigned server), while it respects the server capacity constraints. This method is particular to the summed squared distance and the Euclidean space. Also, it requires that (i) the number of users is equal to the sum of server capacities (i.e., $N = \sum_{s \in S} s.c$) and (ii) users can be assigned to any server, regardless of their distance (i.e., coverage region constraints cannot be dealt with).

A recent study [30] presents techniques for the capacity-constrained assignment problem in disk-resident datasets. These methods are extensions of SSP that add user-server arcs to the flow network incrementally and on-demand (in order to keep it small), until the optimal assignment is finalized. This work focuses on *static* data, stored on the disk and indexed by R-trees. On the other hand, in this paper, we study the *dynamic* maintenance of the optimal assignment and we consider data managed in main memory, since we aim at time-critical applications where users are mobile and updates are frequent. Also, [30] assumes that the servers have infinite coverage (i.e., there are no $s.r$ constraints); here we consider a more realistic situation, where servers have a bounded coverage region.

All aforementioned approaches consider static assignment (i.e., they compute an one-time A over static users and terminate). They could be applied to our continuous scenario by recomputing A from scratch when the users move, but this incurs excessive processing cost as we show experimentally. Our method, CSA, updates A incrementally instead. Note that when CSA runs for the first time, it computes an initial "static" assignment A over the current snapshot of user locations; our empirical study demonstrates that even for this static computation, CSA outperforms existing approaches (in the considered spatial assignment setting) due to its effective geometric pruning strategies.

To our knowledge, there is no previous work on the continuous assignment problem we study in this paper. Even though there exist methods for "dynamic optimal assignment" ([29] and references therein), the term refers to a problem completely different from ours that is related more to vehicle routing and scheduling. As an example of this problem, consider a dispatch service managing a fleet of vehi-

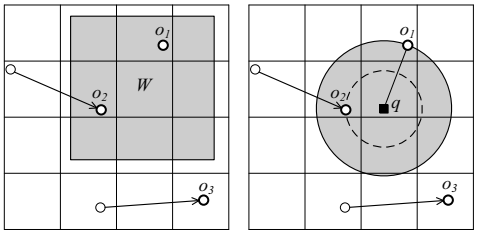
cles. The customers request from the service that a particular load is moved (from and to a particular location), while vehicles can transfer only one load at a time. The objective is to minimize the distances traveled by the vehicles, as vehicles become available and customer requests arrive dynamically. This problem incorporates the expectation of future events (thus termed "non-myopic") and the assignment at a single snapshot is typically not optimal, i.e., a vehicle with a large capacity may be held for some future (larger) load, even if an optimal assignment in the current time instance would dispatch it right away.

Interestingly, the problem of maintaining an optimal user assignment to access points with overlapping coverage regions has not received attention by the network community. Instead, most of the related work in this field (e.g., [15,20, 28]) has focused on planning problems; i.e., optimizing the locations of the access points and/or the frequency channels allocated to them. Based on the estimated user positions and their expected traffic volume, these techniques place access points at a subset of candidate locations so that their bandwidth requirements are similar, and assign to them different frequency channels. The problem we address in this paper is essentially different. First, the servers are at given, predetermined locations. Second, we assign users, instead of placing access points and allocating bandwidth. Third, we *monitor* the optimal assignment in a dynamic setting, rather than producing an one-time result.

2.2 Continuous Spatial Queries

There is a growing literature on continuous evaluation of spatial queries over moving objects. Research in this field usually assumes that the data objects, the queries, and their results fit in main memory. The objects are indexed by a regular grid because this plain index facilitates faster updates compared to more complicated spatial access methods [16]. Algorithms for continuous range processing incrementally insert into/delete from the result those objects that cross the query window boundaries [21,16]. Figure 3(a) exemplifies range monitoring for query window W (shown in gray). The current result contains object o_1. In the next time instance, o_2 and o_3 move as shown, and report their new locations to the monitoring server. The latter focuses only on objects moving into/out of W; i.e., it ignores o_3 and appends o_2 to the result.

Continuous nearest neighbor (NN) queries are processed in a similar way, the difference being that the monitoring region is no longer fixed, but shrinks/expands depending on the distance of the current NN. Algorithms for continuous NN processing [34,22,33] also use a grid index and avoid unnecessary computations for objects far away from the query point. Figure 3(b) illustrates NN monitoring at query point q. The current NN is o_1 and the monitoring region is the disk around q with radius $dist(q, o_1)$. Processing is limited to cells intersecting this disk. For instance, the movement of o_3 is ignored. On the other hand, o_2 is examined and reported



(a) Range monitoring (b) NN monitoring

Fig. 3 Continuous spatial queries

as the new NN. For subsequent processing, the monitoring disk is updated (shrunk) to the disk with radius $dist(q, o_2)$ (shown with a dashed boundary). Similar grid-based approaches have also been proposed for reverse nearest neighbor (RNN) queries [32,17]. A continuous RNN query at point q reports all objects that have q as their NN.

In some applications the mobile objects may have computational capabilities. Motivated by this fact, [27] first introduced the concept of *safe regions* to reduce the number of location updates. Specifically, each object is assigned a spatial region, such that it needs to issue an update only if it exits this region (because, otherwise, it does not influence the result of any query). For instance, this approach could avoid an update transmission from object o_3 in Figures 3(a) and 3(b). Safe regions have been applied to range [27,7,12] and NN monitoring [23].

[18] does not consider continuous spatial queries per se, but is related to our problem. It assumes a set of servers located along a line (i.e., an one-dimensional space). The users move on the same line, and each of them is assigned to his/her closest server. The objective is to approximately answer *count* and *max distance* queries with small space and time requirements; i.e., for every server, the system reports the number of its assigned users and the distance of the furthest among them. [18] additionally includes an approximate method for computing which K servers (K is an input parameter) among the available ones should be kept "active" so that the count or maximum distance values are below a given threshold. This work is intrinsically different from ours because its aim is not to produce an assignment, but to report aggregate values or subsets of servers.

2.3 Resource Allocation in Spatial Databases

Some facility location/resource allocation problems have recently been studied in the context of large spatial datasets. [35] solves the *min-dist optimal-location* problem. Given a static set of users and a set of existing servers, the problem is to find a location where if a new server is placed, the average distance between the users and their closest server will be minimized. This is not an assignment problem, since users are always taken on by their closest server. Also, user updates and server capacities/coverage regions are not taken into account.

[31] solves a spatial matching problem for indexed datasets. Consider a set of static servers S and a set of static users U. The objective is to compute an assignment A such that there is no user-server pair $\langle u, s\rangle \in U \times S$ where both u's assigned server s' and s's assigned user u' are further than $dist(u, s)$ from them. Equivalently, A contains all server-user pairs produced by an incremental closest-pair algorithm that ignores previously reported servers and users. This is an instance of the traditional stable marriage problem [11] in the spatial domain. Besides solving a problem inherently different from optimal assignment, [31] applies only to one-time assignment over static datasets.

3 Initial Assignment in CSA

We assume two-dimensional user/server locations and use Euclidean distance for realizing $dist()$, even though our techniques also apply to higher dimensions and other metrics. The *cost* of an assignment A is defined as the average distance $dist(u, s)$ of the $\langle u, s\rangle$ pairs it contains. Nevertheless, $dist(u, s)$ may be replaced in the cost definition by any monotonically increasing function of $dist(u, s)$ (e.g., $dist^2(u, s)$, or $3\sqrt{dist(u, s)} + 5$), without affecting the effectiveness or the correctness of CSA[3].

The servers are stationary and their coverage regions may have irregular shapes. For ease of presentation, however, we assume that the coverage region of every server s is a circular disk (denoted by $disk(s)$) with center at s and radius $s.r$. Different servers may have different capacities and radii. We say that a user u is *covered* by s if u is located inside $disk(s)$. If two servers cover one or more common users, we say that these servers are *conflicting*.

On the other hand, the users dynamically connect/disconnect to/from the system, and move arbitrarily; we make no assumptions about their log in/out patterns, velocities, trajectories, or movement frequencies. Connected users send location updates to a central *coordinator* whenever they move. We collectively refer to user updates and connect/disconnect requests as *events*. The coordinator's task is to process the events and maintain an optimal assignment A. Processing takes place at discrete timestamps; the reported assignment A at each timestamp is optimal subject to the most current user information available at the coordinator.

When the coordinator produces A for the first time, it needs to compute it from scratch. In this section, we focus on this initial assignment by CSA. However, the presented concepts and geometric optimizations also form the foundation of assignment maintenance in subsequent timestamps (elaborated on in Section 4).

The initial assignment between U and S is computed in three steps. The first one (Section 3.1) quickly produces some assigned pairs with local geometric decisions and excludes users/servers from consideration in the next stages; this is equivalent to deleting vertices and their incident arcs from the flow network. The second step (Section 3.2) distinguishes different types of servers and breaks the problem into smaller MCF instances. The third step (Section 3.3) categorizes these MCF instances into two classes; the first can be solved purely geometrically (at a cost lower than an MCF algorithm), while the second class is processed by some existing MCF method on a flow network whose size is reduced by spatial optimizations.

Before presenting the above in detail, we define the concept of *conflict graph* CG that is central to CSA. Note that we refer to the vertices and arcs in CG as nodes and edges, respectively, to distinguish them from those in a flow network.

Definition 1 ***Conflict graph*:** An undirected graph where every node corresponds to a server, and every edge $\bar{ss'}$ to a pair of conflicting severs s and s' (i.e., servers that cover at least one common user).

CSA initially constructs the complete CG. Figure 4 demonstrates CG in an example used throughout this section. The squares s_i model servers with capacities $s_i.c = 2$, the circles (drawn with continuous line) are the boundaries of their coverage disks, and the hollow points are users. CG includes all servers as nodes, and connects with an edge every pair of conflicting servers (as shown in the figure). Note that even though the coverage disks of s_3 and s_4 overlap, there is no edge between them because they are not conflicting for any user. In the following, we show how CG is used to efficiently compute the optimal assignment A shown on the right, through a sequence of three steps. The process is based on several lemmas (among the corresponding proofs we provide only the most complicated ones for brevity).

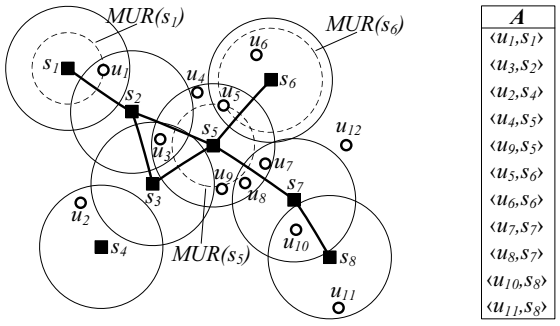


Fig. 4 Conflict graph CG and optimal assignment A

3.1 Step 1: Local Elimination

The local elimination step excludes from consideration some servers and users. Every user that is not covered by any server can be ignored. This is the case for user u_{12} in Figure 4, who is no longer taken into account. Servers that cover no

[3] The adaptation of CSA to such cases is discussed in Section 4.5.

users are also ignored. Furthermore, the elimination step directly makes some assignments (i.e., inserts user-server pairs into A) based on *minimum unshared regions*.

Definition 2 ***Minimum unshared region*:** The minimum unshared region $MUR(s)$ of a server s is defined as the circular disk with center at s and radius equal to the distance of the closest user that s is conflicting for with another server. If s is not conflicting with any server, then $MUR(s)$ is the entire coverage region of s.

Figure 4 illustrates $MUR(s_1)$, $MUR(s_5)$, and $MUR(s_6)$ with dashed boundaries. $MUR(s_4)$ is the entire $disk(s_4)$, because s_4 is not conflicting with any other server. The remaining MURs are omitted from the figure for clarity. Definition 2 implies that users strictly inside $MUR(s)$ (i.e., excluding those on its boundary) (i) are only covered by s and (ii) are closest to s than any user outside $MUR(s)$. Note that the MURs of different servers may overlap (e.g., $MUR(s_5)$ and $MUR(s_6)$) but the above properties still hold, since their overlapping area contains no users by definition. The MUR characteristics lead to the following lemma.

Lemma 1 *Let $s.k$ be the number of users strictly in the interior of $MUR(s)$ (i.e., excluding users on its boundary). The $\min\{s.c, s.k\}$ users in $MUR(s)$ that are closest to s form pairs with s that appear in A.*

Proof We prove that the lemma is correct (i.e., it identifies user-server pairs that indeed belong to the optimal assignment A) by contradiction. Let U_s be the set of users assigned to s by the lemma. Assume that $u \in U_s$ but $\langle u, s\rangle \notin A$. Since u is only covered by s, this means that u is not assigned to any server. Furthermore, due to the maximality of A (i.e., the requirement that the maximum possible number of users are served), the hypothesis implies that s takes on some $u' \notin U_s$ in place of u. As U_s contains the closest users to s, it holds that[4] $dist(u, s) < dist(u', s)$. This is a contradiction; if one of u and u' had to be left unassigned, s would take on the closest (i.e., u) in order to minimize the assignment cost.

According to Lemma 1, CSA computes the MUR of every server s and directly assign to it $\min\{s.c, s.k\}$ users; i.e., we insert the corresponding user-server pairs into A and ignore these users in later steps. Furthermore, if $\min\{s.c, s.k\} = s.c$, we delete s (and its incident edges) from CG and ignore it in the following steps. Otherwise ($s.k < s.c$), we decrease $s.c$ by $s.k$. In Figure 4, for example, we include $\langle u_6, s_6\rangle$ in A and set $s_6.c = 1$; if two or more users were inside $MUR(s_6)$, s_6 would be deleted. The case of s_4 reveals another situation where server elimination is possible. User u_2 is strictly inside $MUR(s_4)$ and is thus assigned to s_4. Even though $s_4.c$ is now 1 (i.e., still greater than zero), s_4 can be eliminated since there are no users left inside $disk(s_4)$.

[4] Note that $dist(u, s)$ cannot be equal to $dist(u', s)$, because U_s excludes users on the boundary of $MUR(s)$.

3.2 Step 2: Isolating Smaller MCF Instances

The rationale behind the second step of CSA is to break the problem into independent, smaller ones. Starting at the server granularity, we say that two servers *affect* each other if altering one's set of assigned users may result in different user assignment to the other, and vice versa. At first sight, it seems that only conflicting servers affect one another (equivalently, only pairs of servers that are adjacent in CG). This, however, is not true; two servers may affect each other via a chain of intermediate conflicting servers. Consider Figure 5(a), where $s_1.c = 2$, $s_2.c = 1$, and $s_3.c = 3$. We compare the optimal assignment in the case where u_8 does not exist in the system versus the case where it does, i.e., we juxtapose assignments A and A' for $U = \{u_1, ..., u_7\}$ and $U' = U \cup u_8$, respectively. A is indicated by the dashed lines, while the table next to the figure illustrates A' and its differences from A. Particularly, removed user-server pairs are shown in strikethrough and added pairs in bold. Note that even though there is no edge s_1–s_3 in CG, the presence of u_8 in $disk(s_3)$ affects the assignments of s_1 (which takes on u_4 in place of u_1).

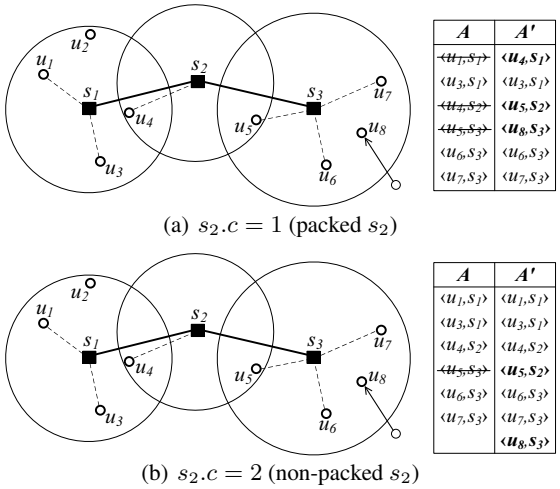


Fig. 5 Example of inter-server influence

On the other hand, not all connected servers affect each other. Assume, for instance, that in the previous example s_2 had capacity $s_2.c = 2$ (instead of 1). Assignment A (in the absence of u_8) would be as indicated by the dashed lines in Figure 5(b). The table next to the figure shows the assignment A' (in the presence of u_8). As opposed to Figure 5(a), the addition of u_8 does not affect s_1. Observe that any removal/addition of users in $disk(s_3)$ (even if $disk(s_3)$ were left empty, or flooded with numerous users) cannot affect s_1, because s_2 "isolates" s_3 from s_1. The reason for this behavior (versus the case where $s_2.c = 1$) is that the increased capacity of s_2 is now enough to accommodate all users inside $disk(s_2)$. This is a crucial observation in CSA that allows performing assignments to some servers locally,

without considering the entire S and U. To formalize it, we distinguish two types of servers, *packed* and *non-packed*.

Definition 3 ***Packed/non-packed server*:** A server s is packed if its coverage region contains more than $s.c$ users. Otherwise, s is non-packed.

Server s_2 in Figure 5(a) is packed, because capacity $s_2.c = 1$ is smaller than 2 (the number of users inside $disk(s_2)$). Conversely, in Figure 5(b) where $s_2.c = 2$, s_2 is non-packed. Lemma 2 uses the classification in Definition 3 to constrain inter-server influence.

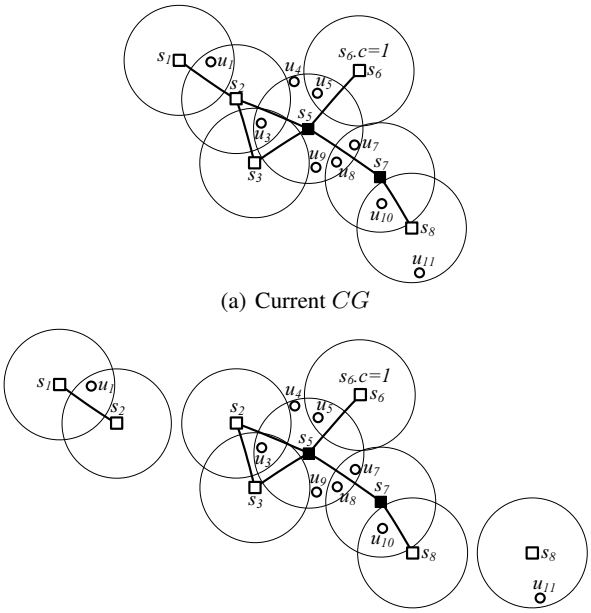
Lemma 2 *Consider a pair of non-conflicting servers s and s'. If all paths between s and s' in CG pass through some non-packed server, then s and s' do not affect each other (i.e., the user assignments of s do not influence those of s', and vice versa).*

Proof Let s_{np} be a non-packed server in a path between s and s'. Assume that altering the user assignment to s necessitates that it hands over/takes on some users to/from another server, and that this process reaches and affects s_{np}. There are two cases. If some users are taken from s_{np}, then s_{np} has more vacancies. This additional freedom of s_{np}, however, cannot lead to a smaller distance or a larger number of served users; in the previous assignment A (i.e., before altering s's assignment), s_{np} also had the flexibility of taking on any of its covered users (being non-packed), but that was not beneficial. Thus, it does not lead to a better assignment in the current state either, i.e., reassignment stops at s_{np}. In the second case, some additional users are assigned to s_{np}. Since s_{np} covers no more than $s.c$ users in total, it can directly accommodate the additional users without propagating the reassignment any further (i.e., towards s'). The situation where the assignments of s' are altered is symmetric. The above conclusion applies to all other paths connecting s and s', with some intermediate non-packed server playing the role of s_{np}. Hence, the lemma holds.

Based on Lemma 2, CSA decomposes CG into smaller MCF problems as follows. We first conceptually delete from CG all non-packed servers and their incident edges. Let CG_{pack} be the resulting graph. We partition CG_{pack} into *maximal connected components* (MCC); each MCC is a connected subgraph of CG_{pack} where the inclusion of any packed server would result in the subgraph being disconnected. From the MCC definition it follows that if two packed servers s and s' are in different MCCs, then they may only be connected through some non-packed server, i.e., they do not affect each other. On the other hand, packed servers in an MCC do affect their adjacent non-packed servers (in the original CG) but none any further (according to Lemma 2). We, hence, call these adjacent non-packed servers the *boundary nodes* of the MCC.

Figure 6(a) continues the example of Figure 4 and shows CG after the local elimination step discussed in Section 3.1 (i.e., with s_4, u_2, u_6, u_{12} eliminated and $s_6.c = 1$). In this and in all the following illustrations of this section, non-packed servers are represented as hollow squares, while packed ones as solid. CG_{pack} consists of s_5, s_7 and the edge between them. There is only one MCC (the entire CG_{pack}) and its boundary nodes are servers s_2, s_3, s_6, s_8, as each of them is adjacent to s_5 or s_7.

Since boundary nodes "isolate" the MCCs, we form an MCF instance (i.e., a subproblem) for every MCC as follows. The subproblem includes as servers the MCC ones plus the boundary nodes, and as users the ones covered by the MCC servers. We call the resulting MCF instance a *packed subproblem*. The middle graph in Figure 6(b) represents the packed subproblem that corresponds to the MCC of s_5, s_7. Note that even though edge s_2s_3 links two boundary nodes, it is part of the subproblem because it corresponds to a user (i.e., u_3) that is also covered by an MCC server (s_5).



(a) Current CG

(b) Extracted MCF subproblems

Fig. 6 Problem decomposition

After the packed servers, we need to extract MCF instances for the non-packed ones too. We first form $CG_{non-pack}$ by removing from CG all packed servers along with their incident edges. For every MCC in $CG_{non-pack}$, we extract a *non-packed subproblem* that is defined on the MCC servers and their covered users (excluding the ones previously included in a packed subproblem). In Figure 6(b), $CG_{non-pack}$ contains two MCCs (with servers $\{s_1, s_2\}$ and $\{s_8\}$, respectively). Their corresponding non-packed subproblems are represented by the leftmost and rightmost connected graphs of the figure. Note that server s_3 belongs to $CG_{non-pack}$, and is part of the left MCC since it is connected with s_2. It is removed from the non-packed MCC $\{s_1, s_2\}$, however, be-

cause there are no remaining users inside $disk(s_3)$ (u_3 was previously included in the packed subproblem).

Users appear only in one MCF instance (be it packed or non-packed). The same holds for the servers, with the exception of boundary nodes. Specifically, a non-packed server may be a boundary node of (and, thus, belong to) multiple packed subproblems, and may additionally appear in a non-packed subproblem too, such as s_2 and s_8 in Figure 6(b). The capacities of these multiple server instances are set to the number of covered users in each subproblem; recall that boundary nodes are non-packed servers and, thus, they could potentially take on all users covered by them.

An important point concerns the partitioning of CG_{pack} and of $CG_{non-pack}$ into MCCs. The MCCs of an undirected graph can be found using depth-first or breadth-first searches, one for each MCC [8]. In particular, starting from a random node, the graph is traversed until the search terminates. The visited nodes form one MCC. The next MCC is found similarly, initiating the traversal at some random unvisited node. The procedure continues until all nodes are visited by some traversal (and, thus, inserted into some MCC).

3.3 Step 3: Processing MCF Instances

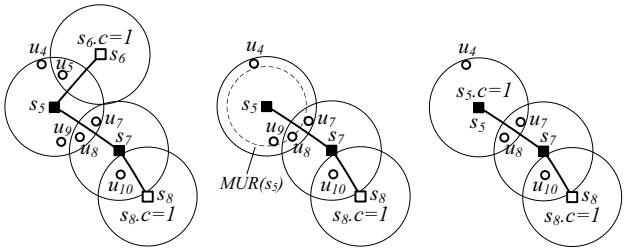
To complete the initial assignment, we need to solve the MCF instances produced in the previous step. We could directly apply a standard MCF algorithm like SSP (described in Section 2.1) to all subproblems and incorporate their individual assignments into the reported A. This, however, can be improved on. In particular, non-packed subproblems can be solved by plain geometric computations, avoiding a costly MCF algorithm. Also, packed instances (even though they require invoking an MCF technique) can be optimized based on spatial criteria. Processing of the former subproblem class is based on Lemma 3.

Lemma 3 *The optimal assignment in any non-packed subproblem is derived by assigning each user to its closest covering server.*

Continuing the example in Figure 6(b), the non-packed problem instance that corresponds to $\{s_1, s_2\}$ is solved by assigning u_1 to the closest server (i.e., to s_1). In the second non-packed subproblem, there is only one server (s_8), and user u_{11} is assigned to it. On the other hand, packed subproblems (like the middle one in Figure 6(b)) are solved by a standard MCF technique. To accelerate the latter, we can geometrically reduce the size of its input (i.e., number of flow network vertices/arcs) using Lemmas 4 and 5 below.

Lemma 4 *In a packed subproblem, if the closest among the servers that cover a user is non-packed, then the user can be assigned to it directly.*

In the packed subproblem of Figure 6(b), u_3 is covered by s_2, s_3, s_5, among which s_2 is the closest. Since s_2 is non-packed, it directly takes u_3 on. Additionally, this leads to the elimination of s_2, s_3 from the subproblem, because (after the assignment of u_3) they cover no users. The resulting subproblem is shown in Figure 7(a). The size of packed MCF instances may be further reduced utilizing the concept of *strongly packed servers*.



(a) $A = A \cup \langle u_5, s_6 \rangle$ (b) $A = A \cup \langle u_9, s_5 \rangle$ (c) Final MCF

Fig. 7 Reducing the size of a packed subproblem

Definition 4 ***Strongly packed server*:** A server s is strongly packed if it covers $s.c$ or more users that are not covered by any other server.

In Figure 7(a), server s_5 uniquely[5] covers users u_4 and u_9. Since $s_5.c = 2$, s_5 is a strongly packed server. Note that in the above definition $s.c$ refers to the remaining capacity of s (after possible assignments in previous stages of CSA). Lemma 5 builds on the concept of strongly packed servers.

Lemma 5 *Let s be a strongly packed server. If s is conflicting for a user u with some non-packed server, then pair $\langle u, s \rangle$ cannot be in the optimal assignment.*

Proof Let s_{np} be a non-packed server that also covers u. We prove the lemma by contradiction, i.e., showing that if u were assigned to s, the maximality requirement would be violated. Let A' be an assignment where s takes on u. Assume that U_s and $U_{s_{np}}$ are the sets of users served in A' by s and s_{np}, respectively. Since u occupies one of s's vacancies, at least one of its uniquely covered users (say u_{unique}) is left unassigned. Consider now the assignment A'' that results from A' by (i) assigning u to s_{np}[6] and (ii) having s take on u_{unique} instead of u. Assignment A'' serves one more user (i.e., u_{unique}) compared to A', because s and s_{np} together serve $U_s \cup U_{s_{np}} \cup u_{unique}$ versus just $U_s \cup U_{s_{np}}$. This means that A' does not serve the maximum possible number of users and, hence, cannot be the optimal assignment.

In Figure 7(a), Lemma 5 implies that u_5 cannot be assigned to (the strongly packed) server s_5, because it is also covered by (the non-packed) s_6. Furthermore, since s_6 is the only other server covering u_5, pair $\langle u_5, s_6 \rangle$ is inserted into A, and s_6 (who is no longer covering any users) is eliminated from the subproblem. This leads to the MCF instance shown in Figure 7(b). Note that Lemma 5 does not assign u_5

[5] Uniquely here means that s is the only covering server.

[6] Note that s_{np} is non-packed and, thus, it is feasible to take on u in addition to users in $U_{s_{np}}$.

to some server per se; it simply disqualifies s_5 from taking it on. For instance, if u_5 were also covered by s_7, it could not be assigned directly to any of s_6, s_7. Instead, we would ignore edge $s_5\bar{s}_6$ and, accordingly, exclude arc $\vec{u_5 s_5}$ from the flow network of the packed subproblem.

To further reduce the size of the subproblem, we recompute server MURs, taking into account only the remaining users, and subsequently we reapply Lemma 1. In Figure 7(b), for example, $MUR(s_5)$ is defined by $dist(u_8, s_5)$, since u_8 is the closest user that s_5 is currently conflicting for. User u_9 is strictly inside $MUR(s_5)$ and, thus, we directly assign u_9 to s_5. The derived MCF instance is shown in Figure 7(c), having eliminated u_9 and set $s_5.c = 1$. Note that if the resulting MCF instance contains a server that is not conflicting with any other, we directly assign to it its closest covered users (until its capacity is reached or all users are assigned).

To conclude, the final subproblem derived can be solved with some traditional MCF algorithm, such as SSP or Hungarian (in our implementation we use the former due to its graceful performance, as discussed in Section 2.1). In Figure 7(c), the MCF algorithm assigns u_4 to s_5, $\{u_7, u_8\}$ to s_7, and u_{10} to s_8. Overall, the computed assignment is shown in the table of Figure 4. Algorithm 1 is a pseudo-code for the initial assignment computation by CSA.

Algorithm 1 Initial assignment in CSA

Initial_Assignment(U, S)
1: Eliminate users covered by no servers
2: Eliminate servers covering no users
3: Compute MURs and perform assignments using Lemma 1
4: Decompose the problem into packed and non-packed subproblems
5: Solve non-packed subproblems using Lemma 3
6: **for all** packed subproblems **do**
7: Reduce the subproblem size using Lemmas 4 and 5
8: Compute MURs and perform assignments using Lemma 1
9: Find the optimal assignment of the subproblem using SSP
10: Return the union of pairs produced in steps $3, 5, 7, 8, 9$ above

A nice feature of Algorithm 1 is that the subproblems produced in step 4 (be them packed or non-packed) can be processed in parallel, since they are completely independent. Therefore, given multiple processors, we can greedily assign/distribute the subproblems among them. In the best case where the subproblems are no more than the processors, the overall cost will be the time spent in steps 1 to 4, plus the cost of the largest subproblem.

4 Assignment Maintenance in CSA

In this section, we show how CSA processes events (location updates and connect/disconnect requests from the users) and accordingly maintains an optimal assignment A. As mentioned in Section 1, we distinguish two variants of the problem, suiting different applications and system requirements. The first variant is the *strict* CAP, where the reported A is optimal at every timestamp, regardless of the previous assignment and connection status of the users. The second variant is the *connected* CAP, where A is optimal subject to an additional constraint; users who remain within the coverage region of their currently assigned server will keep being served by the same server, unless they send a disconnect request. The first-time assignment is computed identically in both cases (i.e., as presented in Section 3). In subsequent timestamps, the corresponding version of CSA produces optimal assignments for either variant. We describe CSA maintenance in the strict case first, and present the connected one later (Section 4.4).

We classify the events into *insertions* and *deletions*. An insertion is a request for service from a new user, while a deletion is a disconnect request from an existing one. A location update is treated as a deletion (at the old user position) and an insertion (at his/her new position). Sections 4.1 and 4.2 describe handling of single insertions and deletions, individually. Since multiple events may arrive at the coordinator in a timestamp, Section 4.3 integrates the above into a maintenance algorithm (i.e., CSA) that processes multiple insertions and deletions in a batch.

To facilitate presentation, we introduce some notation. We denote by A the assignment reported at the previous timestamp, and by A' the updated assignment (to be produced) for the current timestamp. We represent the set of users assigned to server s in A and A' as U_s and U'_s, respectively. Also, we indicate by $C(u)$ the set of servers that cover user u, and denote by $NN(u)$ the closest among them (to u). Furthermore, we use the concept of *full* servers that is central to event processing. We say that a server s is full if it is serving exactly $s.c$ users[7]. Note that this is different from the definition of a packed server. For instance, in the assignment produced in Figure 4, servers s_5, s_6, s_7, and s_8 are full, because each has reached its capacity (i.e., 2).

4.1 Processing a Single Insertion

Consider the insertion of a user u. The first step in CSA is to add new edges into CG for pairs of servers that were previously not adjacent, but are now conflicting for u (i.e., between servers that belong to $C(u)$). Next, CSA attempts to assign u with local decisions, restricting as much as possible the scope of necessary reassignments (and, thus, reducing the computation cost).

If u is not covered by any server (i.e., $C(u) = \emptyset$), no assignment update is required, i.e., $A' = A$. Fast processing is also possible when $C(u) \neq \emptyset$ and $NN(u)$ is not full, according to Lemma 6.

[7] The capacities are reset to their original values in the beginning of each timestamp; recall that during the first-time A computation, for example, we were decreasing the capacities whenever making an assignment. Such capacity changes are temporary, and limited within the scope of one timestamp.

Lemma 6 *Let s be the closest covering server of inserted user u (i.e., $s = NN(u)$). If s were not full in A, u is directly assigned to it (i.e., $A' = A \cup \langle u, s \rangle$).*

Assume that at timestamp $t = 0$ the user locations and assignment A were as shown in Figure 4. Consider that at $t = 1$ a user u connects to the system and requests service. If u is covered by no servers, it is ignored ($A' = A$). Otherwise, if $NN(u)$ is not among the full servers (e.g., if $NN(u)$ is s_3), u is directly assigned to it. Optimizations for fast event processing are also possible when $NN(u)$ is full. Specifically, Lemma 7 applies when u is uniquely covered by a strongly packed server s.

Lemma 7 *Assume that inserted user u is uniquely covered by a server s which (i) was strongly packed in the previous timestamp, and (ii) was assigned (in A) only uniquely covered users. If u is closer to s than the furthest user in U_s, u replaces this user in A'. Otherwise, u is left unassigned.*

Proof The maximality of A implies that any strongly packed server is full; i.e., s cannot simply take u on. Let u_{cur} be the furthest user in U_s (from s). If $dist(u, s) \geq dist(u_{cur}, s)$, u is left unassigned, because taking him/her on in place of any user in U_s would increase the assignment cost, while serving the same number of users. On the other hand, if $dist(u, s) < dist(u_{cur}, s)$, taking u in place of u_{cur} leads to a smaller assignment cost (which, additionally, is lower than replacing any other user in U_s with u). Also, no further reassignment is necessary; if $\langle u_{cur}, s \rangle$ was preferable over $\langle u', s \rangle$ for some $u' \notin U_s$ (in A), then $\langle u, s \rangle$ is also preferable to $\langle u', s \rangle$ (in A').

Figure 8 illustrates the handling of inserted user u by Lemma 7, assuming that the state of the system in the previous timestamp was as in Figure 4. Irrelevant servers are omitted, while dashed lines indicate the previous assignment A. User u is uniquely covered by strongly packed server s_5, which was assigned only uniquely covered users in A. The furthest of them (from s_5) is u_4. Since $dist(u, s_5) < dist(u_4, s_5)$, u is assigned to s_5 in place of u_4. If u were further than $dist(u_4, s_5)$, then no reassignment would be made.

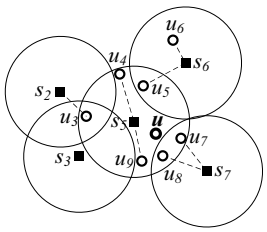


Fig. 8 Example of Lemma 7 (u replaces u_4)

If none of the above lemmas eliminates/assigns the inserted user u, we need to identify and solve the MCF instance u belongs to. Observe that for Lemma 6 to fail, $s = NN(u)$ must be full in A (i.e., s covered at least $s.c$ users in the previous timestamp). It follows that after the insertion of u into $disk(s)$, server s is packed (in the current timestamp). Therefore, the subproblem of u is the packed MCF instance that includes s.[8] We identify the MCC of s in the updated CG and solve the corresponding MCF instance according to Section 3.3 (steps 7, 8, 9 of Algorithm 1). Note that computing s's MCC is performed by one depth-first or breadth-first search in CG, initiated at s and ignoring edges between packed and non-packed servers, i.e., it does not require explicit computation of CG_{pack} or identifying all packed MCCs.

4.2 Processing a Single Deletion

Consider now the deletion of a user u. We first update CG; if two servers in $C(u)$ were conflicting only for u (and no other user), we remove the edge between them. If $C(u) = \emptyset$ or u were unassigned in A, we simply remove u and report $A' = A$. Otherwise, let s_u be the server assigned to the deleted user u (in A). According to Lemma 8 below, fast processing is possible if s_u were not full in A.

Lemma 8 *If s_u were not full in A, we simply delete u, i.e., report $A' = A - \langle u, s_u \rangle$.*

Proof Since s_u was not full in A, the extra vacancy created by the deletion of u cannot lead to a better assignment; s_u had available slots in the previous timestamp, and if a better use of them were possible, they would have already been exploited in A.

In Figure 4, the deletion of u_{12} would not cause any change ($A' = A$) because $C(u_{12}) = \emptyset$. Also, the deletion of u_1, u_2, or u_3 would be treated by simply removing their corresponding pair from A, since their assigned servers (s_1, s_4, and s_2, respectively) were not full in the previous timestamp. Local decisions are also possible for a full s_u, based on Lemma 9.

Lemma 9 *If s_u were serving all its covered users in the previous timestamp, we simply delete u, i.e., report $A' = A - \langle u, s_u \rangle$.*

In Figure 4, servers s_6 and s_8 serve all their covered users. Thus, the deletion of u_5, u_6, u_{10} or u_{11} would be simply dealt with by removing the corresponding pair from A.

If all above criteria/lemmas fail, we treat s_u as packed and isolate/solve the corresponding subproblem in the way explained at the end of Section 4.1. Note that s_u is not necessarily packed in the current timestamp, but we treat it as packed in order to identify all the servers it affects (according to Lemma 2).

[8] Note that all servers in $C(u)$ are conflicting with s for u and, thus, belong to the same MCF instance.

4.3 Processing Multiple Insertions/Deletions

In the general case, multiple events arrive at the coordinator in a timestamp. If the events were treated individually one after another, some subproblems would be solved multiple times, since different events may affect the same server or subproblem. We avoid waste of computations by first scanning all events, marking the affected servers, and then solving their MCF instances in the updated CG.

In particular, we first decompose the events (that arrived in the current timestamp) into a set of insertions E_{ins} and deletions E_{del}. Recall that a location update from a user is treated as a deletion at his/her old position and an insertion at the new one. If multiple updates arrive in the current timestamp from a user u, then we aggregate them. For example, if a user moves from location u to u' and then to u'', the coordinator processes these two events as a deletion at u and an insertion at u''. If the user additionally sent a disconnect request, a single deletion at u is processed.

Next, we update CG taking into account all events. Based on the observation that processing is faster for servers that are not full, we consider E_{del} before E_{ins}. For every deletion in E_{del}, we attempt to locally remove the corresponding user u (according to the criteria and lemmas in Section 4.2). If we fail, we mark s_u as *affected* but do not process its MCF instance yet. Note that if s_u were previously marked as *affected* by another deletion, we skip local processing attempts. We continue with the next deletions in E_{del} until all of them pass this first processing phase.

Then, we scan the insertions in E_{ins}. For each of them, we first check whether any server in $C(u)$ (where u is the inserted user) is already *affected*. If so, we proceed with the next insertion in E_{ins} (as u is contained in the affected server's subproblem and will be handled in the last phase). Otherwise, we attempt to make local reassignments using the criteria and lemmas in Section 4.1. If these criteria fail, we mark $NN(u)$ as *affected*. We stop when the complete E_{ins} is scanned.

In the last phase, we identify the MCF of each *affected* server s in the way described at the end of Section 4.1, while treating all the encountered affected servers as packed. Each of the identified subproblems is processed by steps $7, 8, 9$ of Algorithm 1. Note that multiple affected servers may belong to (and thus be handled simultaneously in) the same subproblem.

Algorithm 2 summarizes the complete assignment maintenance procedure. Observe that step 21 can be parallelized in the way described at the end of Section 3.3. The correctness of CSA on the whole is proven below.

Theorem 1 *CSA assignment is optimal at every timestamp.*

Proof The initial assignment is correct, according to the lemmas presented in Section 3. Thus, CSA is correct if its event processing in every subsequent timestamp is correct. The CSA maintenance module, essentially, processes first the events that can be handled locally (be them insertions or deletions), and then solves (optimally) isolated MCF subproblems. Correctness for each locally processed event is guaranteed by the corresponding lemmas in Sections 4.1 and 4.2. This means that assignment optimality is retained from event to event in the first phase. Each of the remaining events requires solution of an MCF instance in the last phase. These subproblems are unambiguously defined by the current (i.e., updated) CG and can be processed individually (according to Lemma 2), yielding the CSA maintenance procedure correct overall. It follows that CSA reports an optimal assignment at every timestamp.

Algorithm 2 Assignment maintenance in CSA

Maintenance(A, E_{ins}, E_{del})
1: $A' = A$
2: **for all** events u in E_{del} (i.e., deleted users) **do**
3: **for all** pairs of servers s, s' in $C(u)$ that conflict only for u **do**
4: Remove edge $\bar{ss'}$ from CG ▷ CG maintenance
5: **if** u were assigned to server s_u in A and s_u is not *affected* **then**
6: **if** s_u were not full **or** s_u served all covered users in A **then**
7: $A' = A' - \langle u, s_u \rangle$
8: **else**
9: Mark s_u as *affected*
10: Remove u from the system
11: **for all** events u in E_{ins} (i.e., inserted users) **do**
12: **for all** pairs of servers s, s' in $C(u)$ where edge $\bar{ss'} \notin CG$ **do**
13: Add edge $\bar{ss'}$ to CG ▷ CG maintenance
14: **if** no server in $C(u)$ is *affected* **then**
15: **if** server $s = NN(u)$ were not full in A **then**
16: $A' = A' \cup \langle u, s \rangle$
17: **else if** server $s = NN(u)$ satisfies Lemma 7 **then**
18: $A' = A' \cup \langle u, s \rangle - \langle u', s \rangle$ (u' selected by Lemma 7)
19: **else**
20: Mark $NN(u)$ as *affected*
21: Solve the MCF subproblems of affected servers, and update A'
22: Return A'

So far we assumed that there are no updates in S. However, there may be cases where some servers go off-line or some new ones are installed. This type of events are generally infrequent, and recomputing A from scratch when they occur is an acceptable solution. On the other hand, if some time-critical applications cannot tolerate the incurred processing cost, we can improve performance based on concepts similar to the above. Specifically, assume that a server s goes off-line (i.e., is deleted from S). We treat s as a packed server with zero capacity, and isolate/solve its subproblem. Processing is similar in the case where s is a new server inserted into S, the difference being that we now consider the actual $s.c$.

4.4 Event Processing in Connected CAP

In the connected variant of CAP, assignment maintenance is performed in two stages. Let E be the set of all events received in the current timestamp. In the first stage, we preserve the no-disruption-of-service property of connected CAP.

In particular, we consider all users u that (i) were assigned to a server in the previous timestamp, and (ii) do not request disconnection in the current timestamp. Let s be the assigned server of such a user u (i.e., $\langle u, s\rangle \in A$). If u remained static or moved to another location within $disk(s)$, we insert $\langle u, s\rangle$ into A' and reduce $s.c$ by one. In the latter case, we also delete u's location update from E.

In the second stage, we process all remaining events in E as described in Section 4.3. In this setting, however, we ignore users dealt with in the first stage (to avoid their reassignment) and consider the modified server capacities when applying the corresponding lemmas/assignment decisions. The correctness of Algorithm 2 (proven by Theorem 1) guarantees that modified CSA maintains an optimal assignment, with respect to maximality, assignment cost, and the additional feasibility constraint imposed by the connected CAP.

4.5 Discussion on CSA Generality

So far we assumed the Euclidean distance. It is important to stress, however, that CSA can be extended to other Minkowski metrics too. Consider, for instance, Definition 2 and Lemma 1. If we define $MUR(s)$ to be the locus of points p in space where $dist(p, s)$ is smaller than the distance between s and its closest conflicting user, the lemma and its proof hold without changes. If the L_1 metric were used, for example, the MURs would be squares with sides oriented at a $45°$ angle to the coordinate axes (instead of disks), but the lemma would apply directly. The situation is even simpler in our other definitions and assignment criteria, since (i) CG can be formed by primitive operations on sets of covered users, and (ii) our optimizations/lemmas require plain comparisons among user-server distances.

Furthermore, it is easy to see that CSA applies to different cost functions, as long as they are monotonically increasing with distance. The only necessary modification is in step 9 of Algorithm 1 and step 21 of Algorithm 2, where arcs $\vec{us}$ in the flow network of SSP have weights defined according to the specified cost function (instead of $dist(u, s)$), as mentioned in Section 2.1.

CSA is designed for central processing, i.e., there is a central coordinator which receives all the location updates and maintains the assignment. However, Lemma 2 allows for distributed processing at the servers themselves as follows. Initially, each server s identifies all others s' whose coverage regions overlap with its own (these are candidate conflicting servers), and exchanges its packed/non-packed status with them. Then, the implicit CG is decomposed into MCCs/MCFs, starting from random "seed" servers, who relay messages to their neighboring ones (practically executing concurrent breadth-first traversals). After this step, the servers within each MCF choose a coordinator among them for the specific subproblem, and send to it the locations of users they cover (boundary nodes are responsible for determining which of their covered users belong to which MCF). The local coordinator solves the subproblem and performs the initial assignment therein. When a user is deleted (or moves), his/her currently assigned server processes the deletion locally (according to Lemmas 8 and 9), and if this fails, it determines its MCF and acts as a coordinator to update the assignment. When a user is inserted (or moves to a new position), he/she informs its closest covering server, which in turn either processes the insertion locally, or determines its MCF and solves it. All servers in the system should be synchronized to process updates at the same timestamps, in order to avoid solving the same subproblem multiple times and, more importantly, to achieve consistency.

5 CSA Implementation

In this section, we describe data-structures and techniques used in the implementation of CSA. Each server s (user u) has a unique identifier $s.id$ ($u.id$). Information for every $s.id$ is kept in a table that contains $s.r$, $s.c$, and the number of covered users; the latter is used to quickly determine whether s is packed. CG is stored in the form of adjacency lists (one list for every $s.id$). Similar to most spatial monitoring systems (overviewed in Section 2.2), we index the users by a regular, main memory grid, due to its graceful performance in highly dynamic settings.

Every grid cell c stores the set of users that fall inside its spatial extent (as a hash table on user id), and for each of these users it records his/her coordinates and the id of his/her currently assigned server (or *NULL* if unassigned). Additionally, each cell c is associated with an *overlap list* $c.list$ that includes the $s.id$ of every server whose coverage region overlaps with the cell's spatial extent. In Figure 9, for example, the overlap lists of the light gray cells contain either s_1 or s_2, while those of the dark gray cells contain both s_1 and s_2 (e.g., $c_1.list = \{s_1\}$, $c_4.list = \{s_2\}$, $c_2.list = \{s_1, s_2\}$, $c_3.list = \emptyset$).

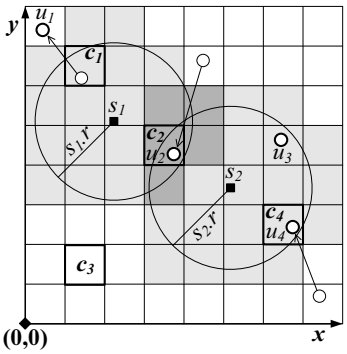


Fig. 9 Grid index and overlap lists

Overlap lists are used to implicitly maintain the set of users covered by each server, and to efficiently identify pairs of conflicting servers. In our example, the entrance of u_4 into $disk(s_2)$ (the movement of u_1 out of $disk(s_1)$) is detected

due to the presence of s_2 in $c_4.list$ (of s_1 in $c_1.list$). Note that overlap lists are used as fast filters, but exact distance comparisons with server coverage radii are necessary for refinement (e.g., a part of c_1 is outside $disk(s_1)$). Overlap lists are also used to efficiently maintain CG. Initially, there is no edge in CG between s_1 and s_2 as the dark gray cells contain no users. After the movement of u_2, however, their conflict is captured by the fact that $c_2.list$ contains them both and, thus, edge $s_1\bar{s}_2$ is inserted into CG.

6 Experimental Evaluation

In this section, we evaluate CSA. We first focus on strict CAP (Section 6.1), assuming a single or multiple available processors. Next, we consider connected CAP (Section 6.2). Finally, we compare CSA with non-optimal, heuristic algorithms (Section 6.3).

In our experiments we generate sets of users U with various cardinalities N that move in a road network (main roads in North America [1], with 175,813 nodes and 179,179 edges), whose node coordinates are normalized to a $[0, 1000]^2$ space. In our default setting, the initial locations of the users (at timestamp 0) are generated by a Gaussian distribution with mean at the center of the network and standard deviation σ equal to 0.2 times the length of the shortest path from the center to the furthest node. This distribution models the situation where many users gather in the center of a city during the working hours of a weekday. In each of the following timestamps, a percentage f of the users send an event to the coordinator. Each event is either a location update or a disconnect request, at a ratio α (e.g., $\alpha = 0.8$ means that 80% of the events are location updates and 20% disconnect requests). Every disconnect request is followed by a connect request from a newly generated user. Each location update corresponds to a moving user; in the given timestamp, the user covers a distance v in the road network. We generate sets of (stationary) servers S with various cardinalities M, following a uniform distribution in the same road network[9]. The servers have equal capacity c and coverage radius r. Table 1 presents the parameters under investigation, and their various examined values, where the ones in bold are the defaults. In each experiment we vary one parameter, and set the remaining ones to their default values. Our simulations include 50 timestamps, and the reported measurements are the average observed values per timestamp. All experiments were executed on an Intel Core 2 Duo E7200 machine with 2 GB of memory.

Our evaluation considers three methods, CSA, R-CSA, and SSP. SSP and R-CSA recompute the assignment from scratch in every timestamp, using SSP and Algorithm 1, respectively. We include the *QuickMatch* [26] optimizations in all three methods. For fairness, we enhance SSP and R-CSA with a grid index with overlap lists (as described in Section 5) to efficiently update their flow network and conflict graph, respectively. The grid granularity (i.e., cell size) for all algorithms is set according to the analysis in [34].

[9] Note that we also examine non-uniform servers, as well as various user/server distribution combinations.

Parameter	Examined values
Number of users (N)	25, 50, **100**, 200, 400 (K)
Standard deviation (σ)	0.05, 0.1, **0.2**, 0.5, 1
Event rate (f)	5%, 10%, **20%**, 50%, 100%
User velocity (v)	0.1, 0.25, **0.5**, 1, 2 (edges/timestamp)
Number of servers (M)	100, 500, **1000**, 1500, 2000
Server capacity (c)	16, 32, 64, **128**, 256, 512
Coverage radius (r)	2.5, 5, **10**, 15, 20, 25

Table 1 System parameters and their examined values

We focus on the performance improvement of CSA over SSP, as it illustrates the significance of our contribution towards the *continuous* assignment problem, i.e., the main topic of this paper. On the other hand, the comparison between R-CSA and SSP demonstrates the superiority of Algorithm 1 over the traditional approach for *static* problem instances.

6.1 Strict CAP

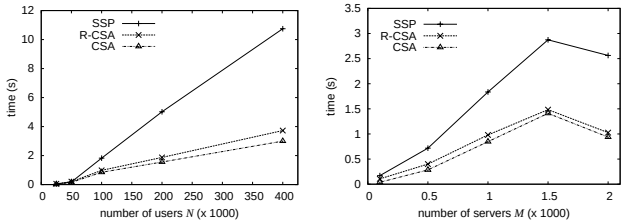
We first consider strict CAP and assume that a single processor is available. Table 2 presents average measurements and statistics in our default setting, such as the assignment cost and size (i.e., the average distance and the number of served users, respectively). The table also includes other important characteristics of the problem. The number of MCCs is the number of (either packed or non-packed) subproblems solved on the average in the last step of CSA/R-CSA. Due to its incremental nature, CSA processes almost 7 times fewer MCCs than R-CSA. The next row contains the number of SSP calls per timestamp, i.e., the number of MCCs that could not be processed purely geometrically. Both CSA and R-CSA avoid a considerable number of SSP calls, demonstrating the effectiveness of our local assignment criteria. The average (node) degree in CG indicates the number of conflicts a server has. The average overlap degree is the number of coverage disks each server's disk overlaps with.

Measurement	SSP	R-CSA	CSA
Assignment cost	4.40		
Assignment size	46616.18		
CPU time	1.77s	0.98s	0.81s
No. of MCCs	–	394.28	58.04
No. of SSP executions	1	47.86	30.64
No. of packed servers	–	473.72	
No. of full servers	–	258.60	
Average degree in CG	–	2.30	
Average overlap degree	–	4.13	

Table 2 Measurements and statistics in the default setting

In Figure 10(a), we vary the number of users N and plot the CPU time per timestamp (in seconds). The processing cost increases with N for all methods, because the flow network of SSP has more edges, while in CSA/R-CSA there

are more conflicts among the servers. CSA is always better that SSP, and their difference grows with N (with CSA being over 3.2 times faster for $N \geq 200$K), because a small M/N ratio implies that most of the assignments in CSA are made locally/inexpensively by the MURs in conjunction with Lemma 1. CSA is around 20% to 35% faster than R-CSA, as a result of the incremental evaluation techniques in Section 4. On the other hand, the processing time of R-CSA is up to 2.8 times shorter than SSP, indicating the efficiency of our methods for static assignments too.

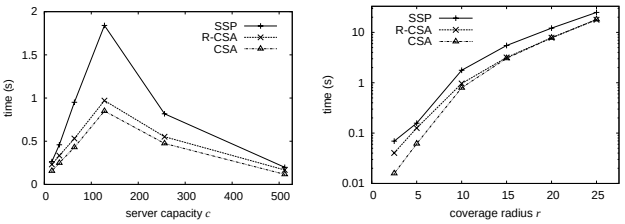


(a) Effect of N (b) Effect of M

Fig. 10 CPU time vs. user and server cardinality

Figure 10(b) measures the effect of the server cardinality M on the performance of the algorithms. Again, CSA and R-CSA are significantly faster than SSP (the largest difference being a 4.5 and 2.8 times improvement, respectively, for $M = 100$). The improvements of CSA and R-CSA over SSP are larger for small M, because their problem decomposition is more effective (i.e., leads to smaller subproblems). The running times of all algorithms initially increase and then decrease. For small M (e.g., $M = 100$) the processing cost is lower because (i) the problem size is smaller, and (ii) the servers are far from each other, thus conflicting for fewer users. On the other hand, when M is very large, the servers are under-utilized (or, non-packed/not full, in CSA terminology), and the problem becomes easier because many users are taken on by their closest server. Again, CSA/R-CSA benefit more from this fact (compared to SSP), because non-packed/not full servers allow for more local decisions (i.e., using Lemmas 3, 4, 6, 8). We mention for completeness that for $M = 100$ the average size of the assignment is 5,906, while for $M = 2000$ it is 72,807; in the latter case, even though there are enough servers for all users, some users are not served because U follows a different distribution from S. In summary, Figure 10 shows that CSA and R-CSA scale well with N and M.

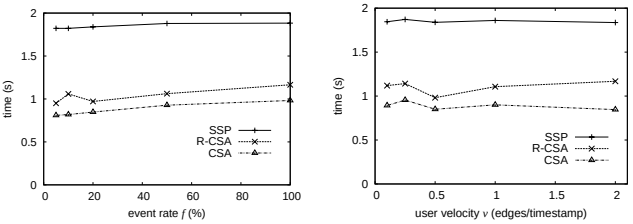
Figure 11(a) investigates the effect of the server capacity c. The running time of the algorithms initially increases and then decreases. There is a correspondence between this trend and that in Figure 10(b); when the capacity is too large (small), the situation is similar to having a large (small) number of servers M. Figure 11(b) shows the computation time (in logarithmic scale) as a function of the coverage radius r. The problem becomes harder when r increases, because the search space grows (i.e., there are more potential assignments to consider); it is worth mentioning that for $r = 25$ the average degree in CG is 13.2 and the average overlap degree is 23.8. In all cases CSA is faster than SSP by a wide margin (taking less than half the time in all cases). The difference between CSA and R-CSA diminishes with r, because as r increases the CG contains fewer and larger MCCs. In turn, the growing MCC sizes imply costlier MCF computations per affected server (in CSA).



(a) Effect of c (b) Effect of r

Fig. 11 CPU time vs. server capacity and coverage radius

Figure 12(a) examines the effect of the event rate f. SSP and R-CSA compute the assignment from scratch in every timestamp, so they are not affected significantly by f; there are fluctuations because different f leads to different system states (i.e., user positions) per timestamp. The running time of CSA increases slightly with f due to its incremental nature. However, even for $f = 100\%$ it outperforms SSP by 1.9 times and R-CSA by 1.2 times; note that $f = 100\%$ means that all users send an event in every timestamp (80% of them move, and the remaining 20% disconnect and are replaced by new connecting users). Figure 12(b) shows the processing time as a function of the user velocity v, varying from 0.1 to 2 times the average edge length (around 4 distance units in our $[0, 1000]^2$ space) per timestamp. Velocity v does not affect the algorithms significantly, and the fluctuations are due to the different system states generated for different v.

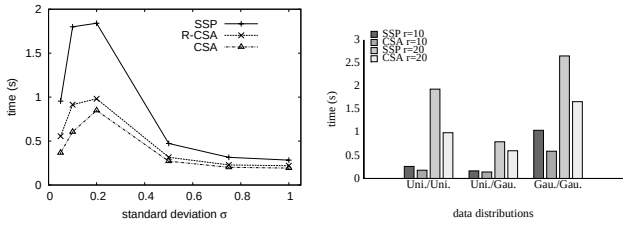


(a) Effect of f (b) Effect of v

Fig. 12 CPU time vs. event rate and user velocity

In Figure 13(a) we vary the standard deviation σ of U's Gaussian distribution. The running time of all algorithms initially increases and then decreases. When σ is small, U is very skewed (concentrated around the center of the datas-

pace), which leaves most servers highly under-utilized (non-packed/not full in CSA terms), and thus easy to handle. As σ grows, more servers become packed/full, which increases the complexity. There is some σ, however, after which the servers around the center of the network are no longer overloaded; larger σ values bring S closer to being uniform, leading to more under-utilized servers and a lower running time.



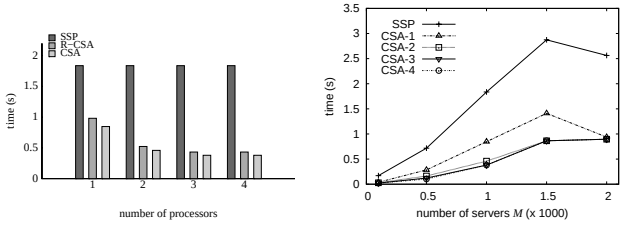
(a) Effect of σ (b) Distribution combinations

Fig. 13 CPU time for various distributions

Figure 13(b) investigates different distribution combinations for U and S, keeping the other parameters to their defaults. Some running times are too small, so we additionally include experiments for $r = 20$ where measurements are more obvious. To avoid cluttering, we exclude R-CSA from the chart; its relative performance to CSA and SSP is similar to our default distributions. In the labels, "Gau." stands for Gaussian and "Uni." for uniform, e.g., label "Uni./Gau." corresponds to uniform users and Gaussian servers. In "Uni./Uni.", most of the servers are under-utilized (recall that the summed capacity of the servers is 128K, i.e., greater than N) and the problem is faster to compute. The situation is similar for "Uni./Gau.", where the centrally located servers are under-utilized, while further from the center (of the dataspace) the servers are sparse and have very few conflicts. In "Gau./Gau.", the running times are larger than "Uni./Uni." and "Uni./Gau.", because the search space grows; i.e., in SSP there are many edges in the flow network, while in CSA there are many conflicts among servers. In all distribution combinations, and for both $r = 10$ and $r = 20$, CSA is faster than SSP, verifying its general superiority.

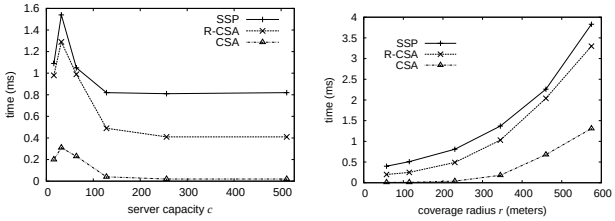
In Figure 14, we investigate performance when multiple processors are available. Scheduling is performed by assigning the subproblem with the most servers to the first unoccupied processor. In Figure 14(a), we vary the number of processors, while setting all other parameters to their defaults. CSA and R-CSA exploit parallelism, and their running time drops when more processors are available. However, there is no improvement when moving from 3 processors to 4, because the bottleneck is one large subproblem that occupies a processor and leaves the others idle. Recall that S follows a Gaussian distribution, and there is always a large MCC at the center of the dataspace.

In Figure 14(b), we investigate the effect of M. In the legends, the number after "CSA-" indicates the number of available processors (e.g., 2 for CSA-2). For clarity in the



(a) Effect of no. of processors (b) Effect of M

Fig. 14 CPU time in the presence of multiple processors



(a) Effect of c (b) Effect of r

Fig. 15 CPU time for real data

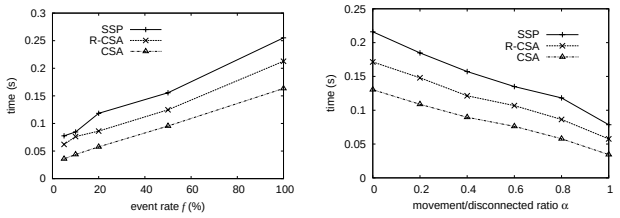
chart, we exclude measurements for R-CSA, which follows the trends of CSA. CSA-4 is 7.5 times faster than SSP for $M = 100$, as in this case there are few conflicts and CG is decomposed into multiple subproblems. CSA-2 and CSA-3 are not much slower than CSA-4, due to the existence of a large MCC, as explained in Figure 14(a). For $M = 2000$ the size of this MCC is the largest, leading to almost identical performance for CSA-2, CSA-3, and CSA-4.

All above experiments use synthetic data, because the available real datasets are small and do not provide control over most of the parameters under investigation. For generality, however, in Figure 15 we include two experiments on strict CAP (with a single processor) using real data. Specifically, we model U as the locations of 14,698 taxis in Singapore, recorded on 5 July 2008 from 20:00 to 20:10. On the average, around 151 location updates are reported per second. The servers follow the taxi distribution; their positions (which are fixed) are drawn at random among the taxi locations at exactly 20:10. By default, there are $M = 150$ servers, with capacity $c = 128$ and coverage radius $r = 230$ meters; M and r are set in proportion to the default problem size and coverage area in the synthetic experiments. We use timestamps of 1 second, and report the average processing time of CSA, R-CSA, and SSP per timestamp.

In Figures 15(a) and 15(b) we measure the effect of c and r, respectively. The observed trends and the reasons behind them are similar to Figures 11(a) and 11(b). A difference is that the peak of the CPU time is at capacity $c = 32$ (in Figure 15(a)) versus 128 (in Figure 11(a)). Another difference is that CSA achieves greater gains compared to R-CSA and SSP (running 12 and 20 times faster in the default setting); the event rate is lower than in the synthetic experiments, thus further benefiting the incremental evaluation in CSA.

6.2 Connected CAP

In this section, we evaluate CSA, R-CSA, and SSP on the connected CAP. Figure 16(a) plots their running time versus the event rate f, setting all other parameters to their defaults (shown in Table 1). CSA consistently outperforms SSP and R-CSA (being up to 2.2 and 1.7 times faster). The CPU time of all algorithms is smaller than in strict CAP, because the no-disruption-of-service constraint reduces drastically the problem size (many users retain their assignment). Figure 16(b) keeps the default $f = 20\%$, but varies the movement/disconnect ratio α; for $\alpha = 0$ all events are connect/disconnect requests, while for $\alpha = 1$ all events are location updates. A small α implies that many users disconnect, thus releasing their assigned slot and increasing the problem complexity.



(a) Effect of f (b) Effect of α

Fig. 16 CPU time for connected CAP

6.3 Comparison with Approximate Methods

In our final experiment, we investigate the applicability of approximate CAP algorithms. If non-maximal/sub-optimal solutions are acceptable, a way to reduce the complexity of an MCF is to eliminate some of the arcs in its flow network. Specifically, each server s could be allowed a maximum of k_{max} incoming arcs $\vec{us}$ (where $k_{max} \geq s.c$), i.e., only the k_{max} nearest users covered by s are considered for assignment to it (ignoring all remaining users in $disk(s)$). By definition, this technique achieves a lower assignment cost than the exact MCF solution (because only the nearest users are considered), but it sacrifices maximality, i.e., it may serve fewer than the maximum possible number of users. This approach can be directly applied to SSP (to reduce the complexity of the entire MCF over U and S), but also to CSA (to accelerate the processing of packed subproblems). We call the resulting methods A-SSP and A-CSA, where "A" stands for *approximate*. Table 3 investigates the effect of parameter k_{max} on A-SSP and A-CSA for strict CAP in the default setting. For reference, recall that the running time of CSA is 0.81s and the exact solution has assignment size 46,616.18 (i.e., 46,616.18 users are served) and assignment cost 4.40. A-CSA is clearly faster than A-SSP, while both methods produce assignments of very similar sizes and costs. Note also that he performance of A-CSA converges to that of CSA for large values of k_{max}. In the following, we ignore A-SSP. Also, we use $k_{max} = 192$ for A-CSA, as it leads to the best tradeoffs.

	A-SSP			A-CSA		
k_{max}	Time	Size	Cost	Time	Size	Cost
128	0.15s	37492.96	3.756	0.13s	37817.7	3.799
192	0.38s	41960.46	4.071	0.21s	41942.1	4.069
256	0.65s	43976	4.273	0.46s	43945.92	4.269
384	1.32s	44853.96	4.397	0.62s	44823	4.393
640	1.85s	45193.7	4.468	0.72s	45161.22	4.464

Table 3 Effect of approximation parameter k_{max}

Another intuitive approximation approach is to use some greedy heuristic strategy, based on local geometric decisions, and specifically on NN queries. We considered several alternatives, the best one being the *continuous heuristic assignment* (CHA), which works as follows. At each timestamp, CHA applies an iterative process. Initially, it computes for every server s its nearest covered user (NN). The NN-server pair with the smallest distance is appended to A, the server's capacity is decreased by one, and the next neighboring (unassigned) user inside its coverage disk is retrieved. The new closest NN-server pair is placed inside A and the process continues until all servers have capacity zero or cover no more (unassigned) users. For the incremental NN searches, we use the algorithm of [22]. Note that CHA respects the server capacities, but it does not guarantee maximality.

Figure 17 considers strict CAP in the default setting and compares the running time of CSA, A-CSA, and CHA for four different combinations of user/server distributions. Table 4 presents the average assignment size and average assignment cost in the same experiment. The approximate methods are faster than CSA for "Gau./Uni." and "Gau./Gau.", but CSA is slightly better in the other two settings[10]. The most important observation, however, is that A-CSA and CHA fail to find a maximal assignment, unnecessarily leaving without service a significant percentage of users (up to about 4.5% and 10%, respectively). Interestingly, this percentage is higher in these settings where the approximate algorithms run faster than CSA. The reason is that users who are left unassigned by A-CSA/CHA (due to their failure to achieve maximality) are usually assigned to remote servers in the optimal solution. A byproduct of this fact is that A-CSA/CHA produce assignments with lower average costs. In summary, if we value the maximality of served users, then CSA should be preferred over approximate approaches even in those settings where the latter are faster.

[10] Note that the reason for A-CSA being slightly slower than CSA is that it uses k_{max}-NN searches (versus the faster $s.r$-range searches) in order to construct the flow networks of packed MCFs.

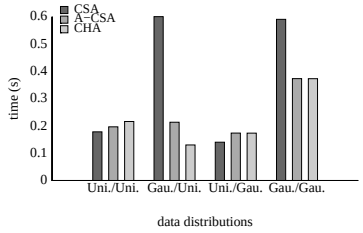


Fig. 17 Comparison with A-CSA and CHA (CPU time)

distributions	CSA		A-CSA		CHA	
	Size	Cost	Size	Cost	Size	Cost
Uni./Uni.	64585	4.989	63163.9	4.887	63591.7	4.898
Gau./Uni.	45193.72	4.468	41942.1	4.069	43490.4	4.180
Uni./Gau.	38643.24	4.323	38480.16	4.301	38393.16	4.280
Gau./Gau.	86483.56	4.151	77743.58	3.610	82548.46	3.824

Table 4 Comparison with A-CSA and CHA (assignment size and cost)

7 Conclusions

In this paper, we study the problem of continuously maintaining the optimal assignment of mobile users to a set of servers. Each server has a limited capacity and a bounded coverage region. The objective is primarily to maximize the number of served users, and secondarily to minimize the average distance between servers and users in the assignment. The continuous assignment problem arises in many applications, including examples in telecommunications and wireless networking.

We propose a comprehensive method for this problem, termed *continuous spatial assignment* (CSA). We exploit several geometric observations to accelerate the initial assignment computation and its subsequent maintenance (in the presence of location updates and connect/disconnect requests from the users). Based on a sequence of local and inexpensive geometric decisions, we exclude certain user-server pairs from consideration and assign others directly. Next, we reduce the problem to a set of smaller ones, and solve them using an off-the-shelf assignment algorithm (i.e., SSP).

We experimentally evaluate CSA and demonstrate its efficiency compared to the direct use of SSP (the most efficient optimal assignment method) at each timestamp; CSA is typically more than two times faster than SSP. Furthermore, we show that the decomposition of the problem into smaller, independent ones allows for even higher performance gains if these tasks are parallelized. In the future, we plan to develop fast, approximate CAP techniques that ensure maximality and provide assignment cost guarantees.

References

1. www.maproom.psu.edu/dcw/
2. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications, first edn. Prentice Hall (1993)
3. Aurenhammer, F., Hoffman, F., Aronov, B.: Minkowski-type theorems and least-squares partitioning. In: Symposium on Computational Geometry, pp. 350–357 (1992)
4. Balinski, M.: Signature methods for the assignment problem. Operations Research **33**, 527–537 (1985)
5. Bertsekas, D.: A new algorithm for the assignment problem. Mathematical Programming **21**(1), 152–171 (1981)
6. Bertsekas, D.: The auction algorithm: A distributed relaxation method for the assignment problem. Annals of Operations Research **14**(1), 105–123 (1988)
7. Cai, Y., Hua, K.A., Cao, G.: Processing range-monitoring queries on heterogeneous mobile objects. In: MDM, pp. 27–38 (2004)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, second edn. The MIT Press (2001)
9. Derigs, U.: A shortest augmenting path method for solving minimal perfect matching problems. Networks **11**(4), 379–390 (1981)
10. Gablow, H.N., Tarjan, R.E.: Faster scaling algorithms for general graph matching problems. Journal of the ACM **38**(4), 815–853 (1991)
11. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. The American Mathematical Monthly **69**, 9–15 (1962)
12. Gedik, B., Liu, L.: Mobieyes: A distributed location monitoring service using moving location queries. IEEE Transactions on Mobile Computing **5**(10), 1384–1402 (2006)
13. Goldberg, A.V., Kennedy, R.: An efficient cost scaling algorithm for the assignment problem. Mathematical Programming **71**(2), 153–177 (1995)
14. Hung, M.: A polynomial simplex method for the assignment problem. Operations Research **31**, 595–600 (1983)
15. Hurley, S.: Planning effective cellular mobile radio networks. IEEE Transactions on Vehicular Technology **51**(2), 243–253 (2002)
16. Kalashnikov, D.V., Prabhakar, S., Hambrusch, S.E.: Main memory evaluation of monitoring queries over moving objects. Distributed and Parallel Databases **15**(2), 117–135 (2004)
17. Kang, J.M., Mokbel, M.F., Shekhar, S., Xia, T., Zhang, D.: Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In: ICDE, pp. 806–815 (2007)
18. Korn, F., Muthukrishnan, S., Srivastava, D.: Reverse nearest neighbor aggregates over data streams. In: VLDB, pp. 814–825 (2002)
19. Kuhn, H.W.: The hungarian method for the assignment problem. Naval Research Logistic Quarterly **2**, 83–97 (1955)
20. Lee, Y., Kim, K., Choi, Y.: Optimization of ap placement and channel assignment in wireless lans. In: LCN, pp. 831–836 (2002)
21. Mokbel, M.F., Xiong, X., Aref, W.G.: Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In: SIGMOD Conference, pp. 623–634 (2004)
22. Mouratidis, K., Hadjieleftheriou, M., Papadias, D.: Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In: SIGMOD Conference, pp. 634–645 (2005)
23. Mouratidis, K., Papadias, D., Bakiras, S., Tao, Y.: A threshold-based algorithm for continuous monitoring of k nearest neighbors. IEEE Transactions on Knowledge and Data Engineering **17**(11), 1451–1464 (2005)
24. Munkres, J.: Algorithms for the assignment and transportation problems. Journal of the Society for Industrial and Applied Mathematics **5**(1), 32–38 (1957)
25. Murty, K.: Network Programming, first edn. Prentice Hall (1992)
26. Orlin, J.B., Lee, Y.: Quickmatch: a very fast algorithm for the assignment problem. Working papers 3547-93, Massachusetts Institute of Technology (MIT), Sloan School of Management (2003). Available at http://ideas.repec.org/p/mit/sloanp/2460.html
27. Prabhakar, S., Xia, Y., Kalashnikov, D.V., Aref, W.G., Hambrusch, S.E.: Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. IEEE Transactions on Computers **51**(10), 1124–1140 (2002)
28. Raniwala, A., Gopalan, K., Chiueh, T.: Centralized channel assignment and routing algorithms for multi-channel wireless mesh networks. SIGMOBILE Mob. Comput. Commun. Rev. **8**(2), 50–65 (2004). DOI http://doi.acm.org/10.1145/997122.997130

29. Spivey, M.Z., Powell, W.B.: The dynamic assignment problem. Transportation Science **38**(4), 399–419 (2004)
30. U, L.H., Yiu, M.L., Mouratidis, K., Mamoulis, N.: Capacity constrained assignment in spatial databases. In: SIGMOD (2008)
31. Wong, R.C.W., Tao, Y., Fu, A.W.C., Xiao, X.: On efficient spatial matching. In: VLDB, pp. 579–590 (2007)
32. Xia, T., Zhang, D.: Continuous reverse nearest neighbor monitoring. In: ICDE, p. 77 (2006)
33. Xiong, X., Mokbel, M.F., Aref, W.G.: Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: ICDE, pp. 643–654 (2005)
34. Yu, X., Pu, K.Q., Koudas, N.: Monitoring k-nearest neighbor queries over moving objects. In: ICDE, pp. 631–642 (2005)
35. Zhang, D., Du, Y., Xia, T., Tao, Y.: Progressive computation of the min-dist optimal-location query. In: VLDB, pp. 643–654 (2006)