The Kinetic Facility Location Problem

Bastian Degener[1,2] Joachim Gehweiler[2] Christiane Lammersen[2]

{bastian.degener, joge, christiane.lammersen}@uni-paderborn.de

[1]International Graduate School Dynamic Intelligent Systems,
[2]Heinz Nixdorf Institute, Computer Science Departement,
University of Paderborn, 33095 Paderborn, Germany

February 2008

Technical Report tr-ri-08-288

Abstract

We present a deterministic kinetic data structure for the facility location problem that maintains a subset of the moving points as facilities such that, at any point of time, the accumulated cost for the whole point set is at most a constant factor larger than the current optimal cost. At any point of time, the cost that arise for a point depends on the status and the position of the point. In the case that a point is currently a facility it causes maintenance cost, otherwise it causes connection cost depending on its demand and its distance to the nearest facility. In our scenario, each point can change its status between client and facility and moves continuously along a known trajectory in a d-dimensional Euclidean space, where d is a constant.

Our kinetic data structure needs $\mathcal{O}(n(\log^d(n)+\log(nR)))$ space, where $R := \frac{\max_{p_i \in \mathcal{P}} f_i \cdot \max_{p_i \in \mathcal{P}} d_i}{\min_{p_i \in \mathcal{P}} f_i \cdot \min_{p_i \in \mathcal{P}} d_i}$, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ is the set of given points, and f_i, d_i are the maintenance cost and the demand of a point p_i, respectively. In the case that each trajectory can be described by a bounded degree polynomial, the data structure processes $\mathcal{O}(n^2 \log^2(nR))$ events, each requiring $\mathcal{O}(\log(nR))$ status changes and $\mathcal{O}(\log^{d+1}(n) \cdot \log(nR))$ time. This results in a total processing time of $\mathcal{O}(n^2 \log^{d+1}(n) \cdot \log^3(nR))$. To the best of our knowledge, this is the first kinetic data structure for the facility location problem.

1 Introduction

The facility location problem is a fundamental combinatorial problem in computer science. In its classical interpretation, the goal is to find an optimal placement of industrial facilities or warehouses, such that the combined cost for the maintenance of the facilities and the transportation cost for the customers are minimized.

In this work, we consider a scenario of continuously moving objects. Each object can either be a facility or a client. Applications for this scenario are for example in sensor networks and mobile ad-hoc networks. In these networks, nodes move continuously and interact with each other. Often they are organized in a hierarchical way, where the upper layer offers the lower layer a certain service, such as a routing infrastructure. Each node can act as a server, but, at any time, cost arises for each node that is set up or maintained as a server. This additional overhead for a server is caused by a higher energy consumption due to message passing, storing of routing tables etc. Since each node should be able to access a service as fast as possible, there is also a cost for each client, namely the delay time which depends on the distance to the nearest server. Now imagine that, to decrease the total cost for the system, nodes are allowed to change their status from server to client or vice versa. Then we have the kinetic facility location problem. In particular, one scenario, in which the above described hierarchical communication system is used, are teams of robots that are deployed in unknown terrain and that have to satisfy certain tasks autonomously. Providing the required services, including storing the data, setting up communication channels, or maintaining equipment, induces some concurrent cost. Finding the proper subset of robots that should provide this service, while the team is moving

through the terrain, is the kinetic facility location problem. However, there are other applications as well. Assume, for example, a future generation of route guiding systems, in which road networks are continuously monitored by satellites, and traffic congestion parameters (e.g., current location and length) are communicated in real-time to the on-board navigation units in cars. As a communication channel to a satellite induces very high cost, it is not desirable to establish communication channels to all on-board units. Thus, for cost minimization, we have to choose a subset of on-board units which are connected to the satellites and act as relays for all units in neighboring cars. Solving this problem is exactly the facility location problem: A relay unit (a facility) induces maintenance cost (the satellite connection) and a non-relay unit (a client) causes connection cost (radio connection to a relay unit) depending on the distance to the nearest relay unit. Since cars move, it is straightforward to assume the setting to be in motion.

Kinetic Data Structures. The kinetic data structure (KDS) framework is well-suited to maintain a combinatorial structure of continuously moving objects and common in the field of computational geometry [8, 11, 21]. In this framework, we are given a set of objects and a *flight plan*, i.e., each object moves continuously along a known trajectory. Furthermore, at any point of time, it is possible to change the flight plan by performing a so called *flight plan update*, which means that one object changes its trajectory. The main idea is now that the continuous motion of the objects is utilized in a way that updates take place only at discrete points of time and can be processed fast. As a result, a lot of computational effort can be saved maintaining the KDS compared to handling just a series of instances of the corresponding static problem. To guarantee that the required properties of the combinatorial structure are satisfied at any point of time, a KDS ensures that certain *certificates* are always hold up. Certificates provide a proof that the combinatorial structure has the desired property. Whenever a certificate fails, we call this an *event*, and an update is required. To be able to handle each of these events at the correct time, an *event queue* is maintained.

There are four important properties to measure the quality of a KDS. The worst-case amount of time to process an update is called *responsiveness*. The second and third property are *compactness* and *locality*. The compactness is given by the ratio between the maximum size of the event queue and the complexity of the moving objects. The latter term addresses the maximum number of events in the queue, in which one object can be involved in. As a result, the locality is a measure on how easily flight plan updates can be performed. The fourth property, the *efficiency* of a KDS, is the ratio between the number of total events processed by the KDS and the minimum number of events that would have been sufficient to maintain a solution for the given kinetic problem. We say that a KDS is responsive, compact, local, and efficient, respectively, if the associated value is at most poly-logarithmic in the complexity of the moving points.

One challenge to construct a KDS for the facility location problem is that the underlying combinatorial structure of an exact solution is not stable. More precisely, a slight change of the position of a point in an exact solution might require an update on all the points to restore an optimal solution. Therefore, we use a new approach to comply with the condition that an update takes only poly-logarithmic time. This requires that the influence of opening or closing a facility is bounded in an appropriate way.

Related Work. The facility location problem has been extensively studied in combinatorial optimization and operations research. In general, the problem is known to be $\mathcal{NP}$-complete. The first constant factor approximation algorithm was given by Shmoys et al. [35]. Many other approximation algorithms followed [14, 15, 28, 29, 33, 36]. Currently, the best approximation algorithm for general metrics achieves an approximation ratio of 1.52 [32]. This is close to the best known lower bound of 1.463 [20]. For the Euclidean case, there exists a randomized PTAS [31]. The facility location problem has also been investigated in other settings, for instance, distributed [19, 34] and dynamic settings [27], but so far no algorithm is known in the kinetic setting. Unfortunately, it does not seem that the only known $(1+\varepsilon)$-approximation given in [31] can be translated to this setting, since the authors use the Arora-scheme including dynamic programming techniques, which does not well comply with kinetization.

The KDS framework was introduced and applied on the convex hull problem by Basch et al. [11]. Later KDSs for measuring various descriptors of the extent of a set of points, including the diameter, width, or smallest bounding box, have been designed [6, 8]. Several further algorithms that use the KDS framework have been developed, e.g., algorithms for kinetic collision detection [10, 25, 30], kinetic planar subdivisions [4, 3, 7], kinetic range searching [1, 12], kinetic kd-trees [2, 5], and kinetic connectivity for unit disks, rectangles, and hypercubes [22, 26]. Only some results are known for problems related to clustering, to which the facility location problem belongs to. For instance, Gao et al. [18] provided a KDS to maintain an expected constant factor approximation for the minimal number of centers to cover all points for a given radius. The centers that they considered are a subset of the moving nodes, whereas Bespamyatnikh et al. [13] studied k-center problems for $k = 1$ in the KDS framework, where the centers are not necessarily located at the moving points. Another algorithm for the kinetic k-center problem can be found in [17]. Hershberger [24] proposed a kinetic algorithm for maintaining a covering of the moving points in $\mathbb{R}^d$ by unit boxes such that the number of boxes is always within a factor of 3^d of the optimal static covering at any instance. Recently, Czumaj et al. [16] presented a KDS for the Euclidean MaxCut problem. For other work on KDSs, we refer to the survey by Guibas [21].

Har-Peled [23] considered the k-center problem in a mobile setting different from the KDS framework, where the number of centers is fixed to exactly k. Instead of handling events, a static set is provided, which ensures a constant factor approximation at all times. However, a set of size $k^{\mu+1}$ is required, where μ is the degree of the polynomial of the trajectories.

Our Contribution. We present a KDS for the facility location problem that gets as input a set of n points in $\mathbb{R}^d$, where d is a constant, and for each given point a trajectory. At any point of time, each point is either a facility (then we call the point *open*) or a client (then we call the point *closed*). The cost that arises for a facility persist during the entire time it is open. Analogously, a client has to pay some cost for its connection to a facility permanently. Our KDS maintains a subset of the moving points as facilities such that, at any time, the sum of the maintenance cost for the facilities and the connection cost for the clients is at most a constant factor larger than the current optimal cost. To be able to ensure this, we keep up the invariant that, on the one hand, for each client there exists a facility in a certain local neighborhood and, on the other hand, no facility has another facility in a certain local neighborhood. The problem is now that restoring the invariant at one point (by changing the status of the point from open to close or vice versa) can lead to a violation of the invariant at many other points. Our main technical contribution is a technique that allows us to restore the invariant in poly-logarithmic time.

To find an initial set of facilities, we use a modified version of the algorithm of Mettu and Plaxton [33]. Each following update can be processed in $\mathcal{O}(\log^{d+1}(n) \cdot \log(nR))$ time, where R is the ratio of the product of the maximum maintenance cost and demand to the product of their corresponding minimum values. Hence, our data structure is responsive, provided that $R = \mathcal{O}(n^{\text{polylog}(n)})$. We point out that the number of status changes per event is $\mathcal{O}(\log(nR))$. In the case that each trajectory can be described by a bounded degree polynomial, we can bound the number of events by $\mathcal{O}(n^2 \log^2(nR))$ and get a total processing time of $\mathcal{O}(n^2 \log^{d+1}(n) \cdot \log^3(nR))$. Flight plan updates can be easily performed with low computational cost, so that the criterion of locality holds. And finally, our data structure is compact because the total space requirement is $\mathcal{O}(n(\log^d(n) + \log(nR)))$.

Organization of the paper. After giving a formal definition of the kinetic facility location problem and introducing some basics used throughout the paper, we describe how the KDS for the facility location problem is designed. Then we prove that at any time it is guaranteed that our current set of facilities leads to a total cost which is at most a constant factor larger than the current optimal cost. Afterwards, we analyze our KDS in terms of its running time and its space requirement. Finally, we conclude with Section 5.

2 Preliminaries

We define the kinetic facility location problem as follows. Let $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ be a set of n independently moving points in $\mathbb{R}^d$, where d is a constant. Let $p_i(t)$ denote the position of p_i at time t and let $\mathcal{P}(t) = \{p_1(t), p_2(t), \ldots, p_n(t)\}$. Furthermore, let $\mathcal{M}$ be the description of the movement of the points in $\mathcal{P}$. At any point of time t, the set $\mathcal{P}(t)$ is divided into two subsets, namely the current set of *facilities* $\mathcal{F}(t)$ and the current set of *clients* $\mathcal{G}(t) = \mathcal{P}(t) \backslash \mathcal{F}(t)$. For each point $p_i(t) \in \mathcal{P}(t)$, there exists a non-negative *maintenance cost* f_i, that has to be paid at time t if $p_i(t)$ is a facility, and a non-negative *demand* d_i. Note that both the maintenance cost and the demand of a point do not change over time. The problem is now to maintain, at each point of time t, a subset $\mathcal{F}(t) \subseteq \mathcal{P}(t)$, such that

$$cost(\mathcal{F}(t)) := \sum_{p_i(t) \in \mathcal{F}(t)} f_i + \sum_{p_j(t) \in \mathcal{G}(t)} d_j \cdot D(p_j(t), \mathcal{F}(t))$$

is minimized. Here, $D(p_j(t), \mathcal{F}(t))$ is the minimum Euclidean distance from $p_j(t)$ to a facility in $\mathcal{F}(t)$. We let $\mathcal{F}_{\text{OPT}}(t)$ denote an optimal set of facilities at time t.

In the following, we introduce some basics required for our approach to handle the kinetic facility problem. The main idea is to use a set of nested cubes around each point and to update the KDS each time a point enters or leaves a cube of another point.

Cubes. For a point $p_i(t) \in \mathcal{P}(t)$ and a non-negative value r, we define $\mathcal{C}(p_i(t), r)$ to be the axis-parallel cube whose center is the point $p_i(t)$ and whose side length is $2r$. Given such a cube $\mathcal{C}(p_i(t), r)$, we let $weight(\mathcal{C}(p_i(t), r))$ denote the sum of the demands of all the points in $\mathcal{P}(t)$ that are located in the cube $\mathcal{C}(p_i(t), r)$, i.e., we define

$$weight(\mathcal{C}(p_i(t), r)) := \sum_{p_j(t) \in \mathcal{P}(t) \cap \mathcal{C}(p_i(t), r)} d_j \,.$$

Radius associated with a point. For each point $p_i(t) \in \mathcal{P}(t)$, we calculate a special radius $r_i^*(t)$ which is an approximation for the radius $r_i(t)$ of the ball with center $p_i(t)$ that is used in [33] and satisfies

$$\sum_{p_j(t) \in \mathcal{P}(t) | D(p_i(t), p_j(t)) \leq r_i(t)} d_j \cdot (r_i(t) - D(p_i(t), p_j(t))) = f_i \,.$$

Due to this definition, $r_i(t)$ ranges from $\frac{\min_{p_j \in \mathcal{P}} f_j}{n \cdot \max_{p_j \in \mathcal{P}} d_j}$ to $\frac{\max_{p_j \in \mathcal{P}} f_j}{\min_{p_j \in \mathcal{P}} d_j}$. To obtain a constant factor approximation for $r_i(t)$, we define $r_i^*(t)$ to be the value 2^{k^*}, such that $k^* = k_0 + \lceil \log(4\sqrt{d}) \rceil$ and k_0 is the minimum integer k with $\log(\frac{\min_{p_j \in \mathcal{P}} f_j}{n \cdot \max_{p_j \in \mathcal{P}} d_j}) \leq k \leq \log(\frac{\max_{p_j \in \mathcal{P}} f_j}{\min_{p_j \in \mathcal{P}} d_j})$, for which $weight(\mathcal{C}(p_i(t), 2^{k_0})) \geq 2^{-k_0}$ holds. The choice of k^* is explained in Section 4. Hence, due to our definition, we have to consider only $\mathcal{O}(\log(nR))$ possible values for the radii, where $R := \frac{\max_{p_i \in \mathcal{P}} f_i \cdot \max_{p_i \in \mathcal{P}} d_i}{\min_{p_i \in \mathcal{P}} f_i \cdot \min_{p_i \in \mathcal{P}} d_i}$.

Note that the cube $\mathcal{C}(p_i(t), r_i^*(t))$ is a ball with radius $r_i^*(t)$ with respect to the L_1-metric. Hence, we approximate the set of radii $\bigcup_{p_i(t) \in \mathcal{P}(t)} r_i(t)$ of the balls for a point set $\mathcal{P}(t)$, where the distance measure is given by the L_2-metric, by a set of radii of special balls for $\mathcal{P}(t)$, where the distance measure is given by the L_1-metric.

Walls around a point. We consider a set of $\mathcal{O}(\log(nR))$ nested cubes for each point $p_i(t) \in \mathcal{P}(t)$. In particular, there is the cube $\mathcal{C}(p_i(t), 2^k)$ with radius 2^k for each $k \in \{\lceil \log(\frac{\min_{p_j \in \mathcal{P}} f_j}{n \cdot \max_{p_j \in \mathcal{P}} d_j}) \rceil + \lceil \log(4\sqrt{d}) \rceil, \lceil \log(\frac{\min_{p_j \in \mathcal{P}} f_j}{n \cdot \max_{p_j \in \mathcal{P}} d_j}) \rceil + 1 + \lceil \log(4\sqrt{d}) \rceil, \ldots, \lfloor \log(\frac{\max_{p_j \in \mathcal{P}} f_j}{\min_{p_j \in \mathcal{P}} d_j}) \rfloor + \lceil \log(4\sqrt{d}) \rceil\}$. The side faces of the cube defined by $\mathcal{C}(p_i(t), 2^k)$ form a wall around $p_i(t)$, which we call $W_{i,k}(t)$. Hence, there exists a set of $\mathcal{O}(\log(nR))$ walls for $p_i(t)$. We use this set of walls to determine the points of time when an update of p_i in our KDS is required. In general, an event occurs each time when any point crosses any wall of another point.

Range Trees. We maintain two $(d+1)$-dimensional dynamic range trees as data structures denoted by T_1 and T_2. At any time, range tree T_1 is used to manage the current set of facilities, and T_2 stores the current set of clients. Apart from the fact that the two data structures contain different point sets, they are constructed in the same way. In the first d levels of the range trees, the points are handled according to their coordinates and in the $(d+1)$-st level according to their special radii. Additionally, with each node v in every binary search tree of the $(d+1)$-st level, we store the sum of the demands of all the points contained in the subtree of v. Beside the two range trees, we maintain a binary search tree T that contains for each point in $\mathcal{P}$ its coordinates currently stored in T_1 or T_2 and its status. The points are sorted according to their indices.

The dynamic data structure described in [12] supports all required properties of T_1 and T_2 efficiently. In particular, a range tree for a set of n points in $\mathbb{R}^{d+1}$ has size $\mathcal{O}(n \log^d(n))$ and can be built in $\mathcal{O}(n \log^{d+1}(n))$ time. We can maintain this data structure in $\mathcal{O}(\log^{d+1}(n))$ worst case time per insertion and deletion. Given any orthogonal range in $\mathbb{R}^{d+1}$, we can output the points inside the range in $\mathcal{O}(\log^{d+1}(n) + N)$ time, where N is the output size. Due to the additional information stored at each node in level $d+1$ of the range trees, we can also compute the sum of the demands of all the points in a certain range in $\mathcal{O}(\log^{d+1}(n))$ time. Finally, we can output the status of a given point in $\mathcal{O}(\log(n))$ time by querying T.

The movement of the points is reflected by insertion and deletion operations on T_1 and T_2 upon an event. That means that the actual position of any point p_i is represented by its coordinates at the latest event it was involved in. Although its exact coordinates might slightly deviate between two events that involve p_i, we will show that, at any point of time t, $p_i(t)$ is stored in the correct range with respect to the walls of all other points. That means, if $p_i(t)$ is located between two walls $W_{j,k(t)}$ and $W_{j,k'(t)}$ of any point $p_j(t)$ then this is also reflected in the positions of p_i and p_j stored in the range trees at time t.

The Mettu-Plaxton Algorithm. We denote the initial position of a point $p_i \in \mathcal{P}$ by $p_i(t_0)$. Furthermore, we use the expression that we open a facility at a certain point to declare that the status of this point changes from closed to open. Analogously, closing a facility means a status change from open to closed.

Algorithm 2.1 METTU-PLAXTON($\mathcal{P}$)

```
1: calculate the radius r_i^*(t_0) for each point p_i(t_0) ∈ P(t_0)
2: sort all points in ascending order according to their radii
3: let p_1(t_0), p_2(t_0), ..., p_n(t_0) be the sorted sequence
4: for i = 1 to n do
5:    if there is no facility in C(p_i(t_0), 2 · r_i^*(t_0)) then
6:       open facility at p_i(t_0)
```

To get an initial set of facilities, we apply Algorithm 2.1 on the input points. This algorithm is almost the algorithm of Mettu and Plaxton [33]. However, the radius $r_i^*(t_0)$ is the above described approximation of the original value $r_i(t_0)$. In Section 4, we prove that this modification still yields a constant factor approximation for the facility location problem.

Note that this greedy algorithm is only used to find an initial solution. Unfortunately, we cannot apply this approach to obtain a KDS with poly-logarithmic update time. The reason is that similar to maintaining an exact solution for the facility location problem, keeping up the solution provided by the original Mettu-Plaxton algorithm is not stable. That means, a slight perturbation of the input might result in $\Omega(n)$ status changes, whereas we are looking for stable solutions, where only a poly-logarithmic number of changes occur upon on an event.

3 The Kinetic Data Structure

This section addresses the design of our KDS for the facility location problem. In particular, we describe how the event queue is structured and how an update of the KDS is processed.

3.1 Event Queue

To maintain a subset of the moving points as facilities, such that the associated cost is at most a constant factor larger than the current optimal cost, we have to update our KDS at certain points of time. More precisely, we perform an update each time a point $p_j(t)$ crosses a wall $W_{i,k}(t)$, where $\lceil \log(\frac{\min_{p_j \in \mathcal{P}} f_j}{n \cdot \max_{p_j \in \mathcal{P}} d_j}) \rceil + \lceil \log(4\sqrt{d}) \rceil \leq k \leq \lfloor \log(\frac{\max_{p_j \in \mathcal{P}} f_j}{\min_{p_j \in \mathcal{P}} d_j}) \rfloor + \lceil \log(4\sqrt{d}) \rceil$, of another point $p_i(t)$.

In order to keep track of these events, we need another data structure beside the two range trees: For each dimension ℓ, $1 \leq \ell \leq d$, we store all n points and all $\mathcal{O}(n \cdot \log(nR))$ wall faces that are orthogonal to the ℓ-th coordinate axis in a list sorted by the ℓ-coordinate. For each consecutive pair in each of the d lists, we keep up one certificate to certify the sorted order of the lists. We define the failure time of the certificate for any pair of consecutive objects to be the first future time when these objects swap their places in their sorted list. The failure times of all certificates are maintained in one event queue.

For simplicity we assume that the points are in general position. Then at most two events occur at the same time. If this happens we handle them in an arbitrary order. Certainly, it is not the case that each event implicates that a point crosses a wall of another point (as, e.g., swapping of two wall faces also causes an event), but definitely every crossing of a wall is discovered by a failure of at least one certificate. The event queue has the following complexity:

Lemma 3.1 *The event queue for the kinetic facility location problem has size $\mathcal{O}(n \log(nR))$, can be initialized in $\mathcal{O}(n \log^2(nR))$ time, and updated in $\mathcal{O}(\log(nR))$ time. Provided that each trajectory can be described by a bounded degree polynomial, the total number of events is $\mathcal{O}(n^2 \log^2(nR))$. A flight plan update involves $\mathcal{O}(\log(nR))$ certificates and requires $\mathcal{O}(\log^2(nR))$ time.*

Proof: The initialization of the lists and the event queue can be done by simple sorting operations in $\mathcal{O}(n \log^2(nR))$ time. In each following update we have to re-calculate the points of time of the next position swaps for the two objects involved in the current event with their two neighbors in the corresponding list. Thus, a constant number of events have to be updated in the event queue, which requires $\mathcal{O}(\log(nR))$ time. Furthermore, a flight plan update of a point causes a re-calculation of the points of time of the next position swap in all d sorted lists for the point and all its wall faces with the associated neighbors in the lists. Afterwards, the involved certificates are updated in the event queue. This can be accomplished in $\mathcal{O}(\log^2(nR))$ time. The upper bounds on the space requirement and the total number of events are obvious. □

3.2 Handling an Update

In this subsection, we describe how an event, occurring at any point of time t, is handled (cf. Alg. 3.1, line 5 ff). As the first step the event queue is updated as explained in Subsection 3.1. All further steps are performed to keep up one invariant consisting of the following conditions:

a) for each closed point $p_i(t) \in \mathcal{G}(t)$ there is an open point $p_j(t) \in \mathcal{F}(t)$ with $r_j^*(t) \leq r_i^*(t)$ in $\mathcal{C}(p_i(t), 4 \cdot r_i^*(t))$ and

b) for each open point $p_i(t) \in \mathcal{F}(t)$ there is no other open point $p_j(t) \in \mathcal{F}(t)$ with $r_j^*(t) \leq r_i^*(t)$ in $\mathcal{C}(p_i(t), 2 \cdot r_i^*(t))$.

We will show that, at each point of time t, if the invariant is satisfied, then $cost(\mathcal{F}(t))$ is at most a constant factor larger than $cost(\mathcal{F}_{\text{OPT}}(t))$. Moreover, the asymmetric choice of condition a) and b) enables our KDS to be stable. Thus, the goal is to restore the invariant each time an event occurs. For simplicity of description, we say that a point $p_i(t)$ violates the invariant at a point of time t in the following case: Either $p_i(t)$ is closed but there is no facility with radius smaller than or equal to $r_i(t)$ in $\mathcal{C}(p_i(t), 4 \cdot r_i^*(t))$, or $p_i(t)$ is open but there is another facility with radius smaller than or equal to $r_i(t)$ in $\mathcal{C}(p_i(t), 2 \cdot r_i^*(t))$.

Now, let us assume that the invariant is satisfied by the time when an event e occurs. Then the only possibility that the invariant gets violated is that e indicates that one point crosses a wall of

another point. Thus, in case that an event is not a wall crossing, handling e is finished after updating the event queue. To detect wall crossings, we check if one of the objects involved in the considered certificate is a point and if the other one is the face of a wall. Then we update both associated points, the point that might cross the wall and the point whose wall might be crossed, in the range trees and check if the first point really crosses a wall of the second point.

Next, we describe the steps that have to be performed in case that any point $p_j(t)$ crosses a wall of any other point $p_i(t)$ at any time t. At first, we update the radius $r_i^*(t) = 2^{k^*}$, such that $k^* = k_0 + \lceil \log(4\sqrt{d}) \rceil$ and k_0 is the minimum integer k with $\log(\frac{\min_{p_j \in \mathcal{P}} f_j}{n \cdot \max_{p_j \in \mathcal{P}} d_j}) \leq k \leq \log(\frac{\max_{p_j \in \mathcal{P}} f_j}{\min_{p_j \in \mathcal{P}} d_j})$, for which $weight(\mathcal{C}(p_i(t), 2^{k_0})) \geq 2^{-k_0}$ holds. Note that k_0 can be found by performing a binary search on its $\mathcal{O}(\log(nR))$ possible values, where each step of the binary search requires only one range query on both T_1 and T_2. Afterwards, we test if $p_i(t)$ violates the invariant by using a range query on T_1. If this is the case, we change the status of $p_i(t)$ (cf. Alg. 3.2). For simplicity of description, we assume that the range trees are always up to date, i.e., at any time the position of each point stored in the range trees is equal to its real current position. We will show in Section 4 that our KDS works as desired, although the position of a point in the range tree can slightly deviate from its real current position. As an effect of changing the radius or the status of one point, the invariant might be violated by many other points (e.g., their open facility has been closed). In the following, we will show how to deal with this problem.

Algorithm 3.1 KineticFL($\mathcal{P}, \mathcal{M}$)

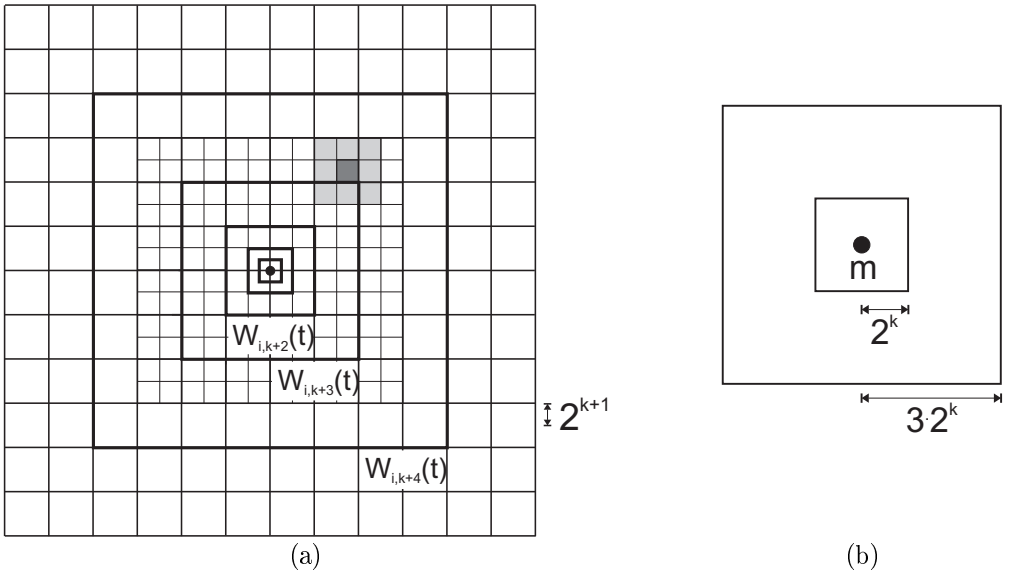
```
1:  Mettu-Plaxton(P)
2:  initialize event queue Q
3:  while Q is not empty do
4:    e ← dequeue(Q)
5:    update Q
6:    if e indicates that p_j(t) crosses a wall of p_i(t) for any i, j then
7:      update position of p_i and p_j in T_1 and T_2
8:      update r_i^*(t) ← 2^{k^*} in T_1 and T_2
9:      if p_i(t) violates the invariant then
10:       change status of p_i(t)
11:     if radius or status of p_i(t) changed then
12:       Restore(p_i(t), k^*)
```

Algorithm 3.2 Restore($p_e(t), k^*$)

```
1: for k ← k^* − 1 to ⌊log(max_{p_j∈P} f_j / min_{p_j∈P} d_j)⌋ + ⌈log(4√d)⌉ do
2:   define cubical shells S_1 := C(p_e(t), 4 · 2^{k+1}) and S_2 := C(p_e(t), 6 · 2^{k+1}) \ S_1
3:   for each cubelet C with center m_C and radius 2^k in cubical shell S_1 do
4:     if ∃ facility with radius < 2^k in C(m_C, 3 · 2^k) then
5:       close all facilities with radius 2^k in C
6:   for each cubelet C with center m_C and radius 2^k in cubical shell S_1 ∪ S_2 do
7:     if ∄ facility with radius ≤ 2^k in C(m_C, 3 · 2^k) then
8:       open one point with radius 2^k in C (if existing)
```

Algorithm Restore. Suppose that, due to an event at any point of time t, the radius or the status of a point $p_e(t)$ changed and its new radius is $r_e^*(t) = 2^{k^*}$. First, we restore the invariant at all points with radius 2^{k^*-1}, to ensure that no point with radius less than or equal to 2^{k^*-1} violates the invariant. Then we handle all points with radius 2^{k^*} that violates the invariant, then the ones with radius 2^{k^*+1}, ..., up to the biggest possible radius. Now, we describe in general how the invariant is restored at all the points with radius 2^k that violate the invariant.

We define the two cubical shells $S_1 := \mathcal{C}(p_e(t), 4 \cdot 2^{k+1})$ and $S_2 := \mathcal{C}(p_e(t), 6 \cdot 2^{k+1}) \setminus S_1$. Both cubical shells are divided into equally sized cubelets with radius 2^k. Figure 1 (a) illustrates this decomposition in the plane for k and the next iteration $k+1$.



(a) (b)

Figure 1: (a) Decomposition into cubelets. (b) Tested area.

To guarantee that no open point with radius 2^k violates the invariant, we perform the following test for each cubelet in S_1: Let m be the center point of the considered cubelet. If there is a facility with radius less than 2^k in $\mathcal{C}(m, 3 \cdot 2^k)$, then close all facilities with radius 2^k in $\mathcal{C}(m, 2^k)$. Note that there is at most one such facility. The considered area around a cubelet is illustrated in Figure 1 (b).

In order to ensure that no closed point with radius 2^k violates the invariant neither, we test each cubelet in $S_1 \cup S_2$ one after the other, whether there exists a facility with radius less than or equal to 2^k in $\mathcal{C}(m, 3 \cdot 2^k)$. If this is not the case, then we open a point with radius 2^k in the cubelet (if there is such a point). No matter, whether we opened a point or not, it is guaranteed, that for each closed point $p_j(t)$ with $r_j^*(t) = 2^k$ in the cubelet, there is a facility in $\mathcal{C}(p_j(t), 4 \cdot r_j^*(t))$.

4 Quality and Complexity of the Kinetic Data Structure

At first, we prove that the invariant is satisfied each time our KDS has handled an update. Afterwards, this fact is used to show that our KDS maintains, at any point of time t, a set of facilities $\mathcal{F}(t)$ such that $cost(\mathcal{F}(t)) = \mathcal{O}(cost(\mathcal{F}_{\mathrm{OPT}}(t)))$. Finally, we analyze the complexity.

Maintenance of the Invariant. The difficulty in proving the correctness of maintaining the invariant is that both range trees contain out-dated information. It is guaranteed that the radius of each point stored in the range trees is equal to its current special radius, but this is not true for the position. More precisely, at any time the coordinates of each point stored in the range trees are determined by its position at the latest event it was involved in. Nevertheless, we will prove that our algorithm works as desired. For any point of time t and any $p_i(t) \in \mathcal{P}(t)$, let $p_i^T(t)$ be the position of p_i stored in the range trees at time t. The following claim shows that, at any time, every point is stored in the correct range with respect to the walls of all other points.

Claim 4.1 *Let $p_i, p_j \in \mathcal{P}$, let k be an integer in $\{\lceil \log(\frac{\min_{p_j \in \mathcal{P}} f_j}{n \cdot \max_{p_j \in \mathcal{P}} d_j}) \rceil + \lceil \log(4\sqrt{d}) \rceil, \lceil \log(\frac{\min_{p_j \in \mathcal{P}} f_j}{n \cdot \max_{p_j \in \mathcal{P}} d_j}) \rceil + 1 + \lceil \log(4\sqrt{d}) \rceil, \ldots, \lfloor \log(\frac{\max_{p_j \in \mathcal{P}} f_j}{\min_{p_j \in \mathcal{P}} d_j}) \rfloor + \lceil \log(4\sqrt{d}) \rceil\}$, and let t be any point of time between two successive events which involve p_i and p_j. If and only if we have $p_j^T(t) \in \mathcal{C}(p_i^T(t), 2^k)$, then $p_j(t) \in \mathcal{C}(p_i(t), 2^k)$ is true as well.*

Proof: Let $t_1 < t$ be the latest point of time when p_i and p_j have been involved in one event. Furthermore, $p_j^T(t) \in \mathcal{C}(p_i^T(t), 2^k)$ implies that we have updated p_i and p_j at time t_1, such that $p_j(t_1) \in \mathcal{C}(p_i(t_1), 2^k)$ and $p_j^T(t_1) \in \mathcal{C}(p_i^T(t_1), 2^k)$. Now let us assume that we have $p_j^T(t) \in \mathcal{C}(p_i^T(t), 2^k)$ but $p_j(t) \notin \mathcal{C}(p_i(t), 2^k)$. Thus, there must be a point of time t_2 with $t_1 < t_2 < t$ when the point $p_j(t_2)$ crosses the wall $W_{i,k}(t_2)$. Then t_1 could not be the latest point of time when p_i and p_j have been involved in one event, a contradiction. Analogously, we can show that $p_j^T(t) \notin \mathcal{C}(p_i^T(t), 2^k)$ implies $p_j(t) \notin \mathcal{C}(p_i(t), 2^k)$. □

Next, we prove that the invariant is satisfied as long as algorithm KineticFL does not call algorithm Restore.

Claim 4.2 *The invariant is satisfied after the first step of algorithm* KineticFL.

Proof: For each point $p_i \in \mathcal{P}$, we have $p_i^T(t_0) = p_i(t_0)$. Since algorithm Mettu-Plaxton treats the points in ascending order according to their radii and opens a point $p_i(t_0)$ with radius $r_i^*(t_0)$ if there is no other facility in $\mathcal{C}(p_i(t_0), 2 \cdot r_i^*(t_0))$, no facility violates the invariant.

Furthermore, algorithm Mettu-Plaxton does not open a point $p_i(t_0)$ with radius $r_i^*(t_0)$ if and only if there is another facility in $\mathcal{C}(p_i(t_0), 2 \cdot r_i^*(t_0)) \subseteq \mathcal{C}(p_i(t_0), 4 \cdot r_i^*(t_0))$. Because this facility has been treated earlier than $p_i(t_0)$, its radius is less than or equal to $r_i^*(t_0)$. Thus, there exists a facility with radius less than or equal to $r_i^*(t_0)$ in $\mathcal{C}(p_i(t_0), 4 \cdot r_i^*(t_0))$. Hence, no closed point violates the invariant. □

Claim 4.3 *Let e be any event such that algorithm* KineticFL *does not change the radius or the status of any point. If the invariant is satisfied before e, then it holds after e as well.*

Proof: We have to consider two cases. In the first case no point crosses a wall of another point. This implies that no radius changes its value, so that the invariant is valid. Furthermore, our algorithm does not change the radius or the status of any point, so that the claim holds.

Let t be the point of time when event e occurs. Then, in the second case, we have that a wall $W_{i,k}(t)$ of a point $p_i(t)$ is crossed by another point $p_j(t)$, but our algorithm does not change the radius or the status of $p_i(t)$. It follows that $p_i(t)$ does not violate the invariant because otherwise our algorithm would have changed its status. Due to the fact that $p_i(t)$ is unchanged and only the wall $W_{i,k}(t)$ is crossed at time t, no point in $\mathcal{P}(t) \backslash \{p_i(t)\}$ violates the invariant. □

The following claims show that the invariant is restored after each call of algorithm Restore.

Claim 4.4 *Let $p_e(t)$ be a point whose radius or status changed due to an event e. Let $r_e^*(t) = 2^{k^*}$ be the updated radius of $p_e(t)$. If no point with radius less than or equal to 2^{k^*-2} violates the invariant before e, then this holds after e as well.*

Proof: Due to the definition of the special radii and the fact that only one point has crossed one wall of $p_e(t)$, the radius of p_e has been at least 2^{k^*-1} before e. While processing event e, we only change the status of points with radius larger than or equal to 2^{k^*-1}. These status changes cannot effect the invariant at points with radius less than or equal to 2^{k^*-2}. Thus, the claim follows. □

Claim 4.5 *Let $p_e(t)$ be a point whose radius or status changed due to an event e. Let $r_e^*(t) = 2^{k^*}$ be the updated radius of $p_e(t)$. If the invariant is satisfied before e and no open point with radius less than or equal to $2^{\ell-1}$ violates the invariant before running the outer* ***for****-loop of algorithm* Restore *for $k = \ell$, where $k^* - 1 \leq \ell \leq \lfloor \log(\frac{\max_{p_j \in \mathcal{P}} f_j}{\min_{p_j \in \mathcal{P}} d_j}) \rfloor + \lceil \log(4\sqrt{d}) \rceil$, then, after running this* ***for****-loop, no open point with radius 2^ℓ violates the invariant.*

Proof: The proof is by contradiction. Let us assume that after running the outer **for**-loop of algorithm Restore for $k = \ell$ there is an open point $p_i(t)$ with radius $r_i^*(t) = 2^\ell$ that has another open point $p_j(t)$ with radius $r_j^*(t) \leq r_i^*(t)$ in $\mathcal{C}(p_i(t), 2 \cdot r_i^*(t))$.

Note that p_e is updated in the range trees at time t, so that $p_e^T(t) = p_e(t)$. Due to Claim 4.1, we know that $p_i(t) \in S_1$ if and only if $p_i^T(t) \in S_1$. Furthermore, due to Claim 4.1, if we have $p_i(t) \in S_2$, then we also know that $p_i^T(t) \notin S_1$. Keeping this in mind, we have to consider the following five possible combinations for $p_i(t)$ and $p_i^T(t)$:

i) $p_i(t) \in S_1$ and also $p_i^T(t) \in S_1$

ii) $p_i(t) \in S_2$ and also $p_i^T(t) \in S_2$

iii) $p_i(t) \in S_2$, but $p_i^T(t) \notin S_1 \cup S_2$

iv) $p_i(t) \notin S_1 \cup S_2$ and also $p_i^T(t) \notin S_1 \cup S_2$

v) $p_i(t) \notin S_1 \cup S_2$, but $p_i^T(t) \in S_1 \cup S_2$

Cases i), ii), and v). Subcase $r_j^*(t) < r_i^*(t)$: Due to the fact that $r_j^*(t) < 2^\ell$, we have opened p_j before running the outer **for**-loop for $k = \ell$. It follows that $p_i^T(t) \in \mathcal{C}(m, 2^\ell)$ and $p_j^T(t) \notin \mathcal{C}(m, 3 \cdot 2^\ell)$ for one center m of a considered cubelet, because otherwise we either would have closed $p_i(t)$ or would not have opened $p_i(t)$. As a consequence, $p_j^T(t) \notin \mathcal{C}(p_i^T(t), 2^{\ell+1}) = \mathcal{C}(p_i^T(t), 2 \cdot r_i^*(t))$. Now, due to Claim 4.1, we have $p_j(t) \notin \mathcal{C}(p_i(t), 2 \cdot r_i^*(t))$, which is a contradiction.

Subcase $r_j^*(t) = r_i^*(t)$: We have to consider the case that neither p_i nor p_j is opened while running the outer **for**-loop for $k = \ell$ and the case that at least one of p_i and p_j is opened during this **for**-loop. In the first case, it follows that p_i and p_j must have been open before running the outer **for**-loop for $k = \ell$. As a consequence, both points have been open before e or one point is p_e. Then either the invariant was violated before e or changing the status of p_e violated the invariant, a contradiction. In the latter case, we have opened at least one point of p_i and p_j while running the outer **for**-loop for $k = \ell$. W.l.o.g., let us assume that we have opened p_i before we have opened p_j. Then we must have that $p_j^T(t) \in \mathcal{C}(m, 2^\ell)$ and $p_i^T(t) \notin \mathcal{C}(m, 3 \cdot 2^\ell)$ for one center m of a considered cubelet. It follows that $p_j^T(t) \notin \mathcal{C}(p_i^T(t), 2^{\ell+1}) = \mathcal{C}(p_i^T(t), 2 \cdot r_i^*(t))$. Due to Claim 4.1, we have $p_j(t) \notin \mathcal{C}(p_i(t), 2 \cdot r_i^*(t))$, which is a contradiction.

Cases iii) and iv). Subcase $r_j^*(t) < r_i^*(t)$: Due to the fact that $r_j^*(t) < 2^\ell$, we have opened p_j before running the outer **for**-loop for $k = \ell$. Furthermore, it follows from $p_i^T(t) \notin S_1 \cup S_2$ that we must have opened p_i before running the outer **for**-loop for $k = \ell$ as well. Hence, we have that both p_i and p_j have been open before running this **for**-loop. Consequently, the invariant must have been violated at point $p_j(t)$ with $r_j^*(t) \leq 2^{\ell-1}$ before running the outer **for**-loop for $k = \ell$, a contradiction.

Subcase $r_j^*(t) = r_i^*(t)$: Here we can use the same argumentation as for cases i), ii), and v) in subcase $r_j^*(t) = r_i^*(t)$ with the modification that we know that p_i has been opened before running the outer **for**-loop for $k = \ell$. The reason is that $p_i^T(t) \notin S_1 \cup S_2$, so that we do not change its status during the running of this **for**-loop. □

Claim 4.6 *Let $p_e(t)$ be a point whose radius or status changed due to an event e. Let $r_e^*(t) = 2^{k^*}$ be the updated radius of $p_e(t)$. If the invariant is satisfied before e and no closed point with radius less than or equal to $2^{\ell-1}$ violates the invariant before running the outer **for**-loop of algorithm* RESTORE *for $k = \ell$, where $k^* - 1 \leq \ell \leq \lfloor \log(\frac{\max_{p_j \in \mathcal{P}} f_j}{\min_{p_j \in \mathcal{P}} d_j}) \rfloor + \lceil \log(4\sqrt{d}) \rceil$, then, after running this **for**-loop, no closed point with radius 2^ℓ violates the invariant.*

Proof: The proof is by contradiction. Let us assume that after running the outer **for**-loop of algorithm RESTORE for $k = \ell$ there is a closed point $p_i(t)$ with radius $r_i^*(t) = 2^\ell$ that has no open point with radius less than or equal to $r_i^*(t)$ in $\mathcal{C}(p_i(t), 4 \cdot r_i^*(t))$. We have to consider the same five cases as in the proof of Claim 4.5.

Cases i), ii), and v). The assumption implies that $p_i^T(t) \in \mathcal{C}(m, 2^\ell)$ and there is an open point $p_j^T(t)$ in $\mathcal{C}(m, 3 \cdot 2^\ell)$ for any center m of a considered cubelet, because otherwise we would have opened a point with radius 2^ℓ in $\mathcal{C}(m, 2^\ell)$. As a consequence, $p_j^T(t) \in \mathcal{C}(p_i^T(t), 2^{\ell+2}) = \mathcal{C}(p_i^T(t), 4 \cdot r_i^*(t))$. Now, due to Claim 4.1, we have $p_j(t) \in \mathcal{C}(p_i(t), 4 \cdot r_i^*(t))$, which is a contradiction.

Cases iii) and iv). Let t' be any point of time between the occurrence of e and the latest event before. Then there was an open point $p_j(t')$ with radius less than or equal to $r_i^*(t)$ in $\mathcal{C}(p_i(t'), 4 \cdot r_i^*(t'))$ because otherwise the invariant was violated before e. Due to Claim 4.1, $p_j^T(t') \in \mathcal{C}(p_i^T(t'), 4 \cdot r_i^*(t'))$ is also true. From $p_i^T(t) \notin S_1 \cup S_2 = \mathcal{C}(p_e^T(t), 6 \cdot 2^{\ell+1})$ follows that $p_e^T(t) \notin \mathcal{C}(p_i^T(t), 4 \cdot 2^\ell)$, so that $p_e \neq p_j$. Due to $p_i \neq p_e$, $p_j \neq p_e$, and $p_j^T(t') \in \mathcal{C}(p_i^T(t'), 4 \cdot r_i^*(t'))$, we have that $p_j^T(t) \in \mathcal{C}(p_i^T(t), 4 \cdot r_i^*(t))$ is also true. Thus, if p_i violates the invariant after e, then we must have closed p_j during processing e. We only close points with radius less than or equal to $r_i^*(t)$ in S_1. Since $p_i(t) \notin S_1 \cup S_2$ and $p_j(t) \in S_1$, we get $p_j^T(t) \notin \mathcal{C}(p_i^T(t), 2 \cdot 2^{\ell+1}) = \mathcal{C}(p_i^T(t), 4 \cdot r_i^*(t))$, a contradiction. □

Now, we can combine the obtained results to the following lemma:

Lemma 4.1 *The invariant is satisfied after algorithm* KineticFL *has handled an event.*

Proof: Due to Claim 4.2 and Claim 4.3, the invariant is satisfied as long as algorithm KineticFL does not call algorithm Restore. Now, we show that this is also true after processing algorithm Restore.

Let $p_e(t)$ be the point whose radius or status changed due to an event e, and let $r_e^*(t) = 2^{k^*}$ be its updated radius. Because of the precondition given above and Claim 4.4 the lemma is true for all open points $p_i(t)$ with radius $r_i(t)^* = 2^\ell$ where $\ell \leq k^* - 2$, and because of Claim 4.5 it follows for $\ell \geq k^* - 2$. Hence, it is true for all open points.

A similar argumentation holds for the closed points. Because of the precondition given above and Claim 4.4 the lemma follows for all closed points $p_i(t)$ with radius $r_i(t)^* = 2^\ell$ where $\ell \leq k^* - 2$, and because of Claim 4.6 it follows for $\ell \geq k^* - 2$. Thus, it is true for all closed points. □

The Special Radii. Similar to the definition of cubes, for any point $p_i(t) \in \mathcal{P}(t)$, we define $\mathcal{B}(p_i(t), r)$ to be the ball with center $p_i(t)$ and radius r. Given such a ball $\mathcal{B}(p_i(t), r)$, we let $weight(\mathcal{B}(p_i(t), r))$ denote the sum of the demands of all the points in $\mathcal{P}(t)$ that lie in the ball $\mathcal{B}(p_i(t), r)$. Now, we can prove that, at any point of time t, the special radius $r_i^*(t)$ of any point $p_i(t) \in \mathcal{P}(t)$ is a constant factor approximation for the value $r_i(t)$ that is used in [33] and satisfies

$$\sum_{p_j(t) \in \mathcal{B}(p_i(t), r_i(t))} d_j \cdot (r_i(t) - D(p_i(t), p_j(t))) = f_i \,.$$

For the uniform facility location problem, the authors in [9] gave an appropriate lower and upper bound for the value $r_i(t)$ and showed how to approximate $r_i(t)$ by counting the number of points in a certain distance of $p_i(t)$. We generalize their two results to the non-uniform case:

Lemma 4.2 *For any point of time t and for each $p_i(t) \in \mathcal{P}(t)$, we have*

$$\frac{f_i}{weight(\mathcal{B}(p_i(t), r_i(t)))} \leq r_i(t) \leq \frac{2 \cdot f_i}{weight(\mathcal{B}(p_i(t), \frac{r_i(t)}{2}))} \,.$$

Proof: From the definition of $r_i(t)$ follows $\sum_{p_j(t) \in \mathcal{B}(p_i(t), r_i(t))} d_j \cdot r_i(t) \geq f_i$, so that

$$r_i(t) \geq \frac{f_i}{\sum_{p_j(t) \in \mathcal{B}(p_i(t), r_i(t))} d_j} = \frac{f_i}{weight(\mathcal{B}(p_i(t), r_i(t)))} \,.$$

Furthermore, we get

$$\begin{aligned} f_i &= \sum_{p_j(t)\in\mathcal{B}(p_i(t),r_i(t))} d_j \cdot (r_i(t) - D(p_i(t),p_j(t))) \\ &\geq \sum_{p_j(t)\in\mathcal{B}(p_i(t),\frac{r_i(t)}{2})} d_j \cdot (r_i(t) - D(p_i(t),p_j(t))) \\ &\geq \frac{r_i(t)}{2} \cdot \sum_{p_j(t)\in\mathcal{B}(p_i(t),\frac{r_i(t)}{2})} d_j = \frac{r_i(t)}{2} \cdot weight(\mathcal{B}(p_i(t), r_i(t)/2)) \,, \end{aligned}$$

which completes the proof. □

Lemma 4.3 *For any point of time t, let k_1 be the minimum integer k with $\lceil \log(\frac{\min_{p_j\in\mathcal{P}} f_j}{n\cdot\max_{p_j\in\mathcal{P}} d_j}) \rceil \leq k \leq \lfloor \log(\frac{\max_{p_j\in\mathcal{P}} f_j}{\min_{p_j\in\mathcal{P}} d_j}) \rfloor$, such that*

$$weight(\mathcal{B}(p_i(t), 2^k)) \geq f_i \cdot 2^{-k} \,.$$

Then $\frac{1}{2} \cdot r_i(t) \leq 2^{k_1} \leq 2 \cdot r_i(t)$ holds.

Proof: Due to the choice of k_1, we have $weight(\mathcal{B}(p_i(t), 2^{k_1-1})) < f_i \cdot 2^{-(k_1-1)}$. It follows that, for any $r_i(t) < 2^{k_1-1}$, we get

$$weight(\mathcal{B}(p_i(t), r_i(t))) \leq weight(\mathcal{B}(p_i(t), 2^{k_1-1})) < f_i \cdot 2^{-(k_1-1)} < f_i \cdot \frac{1}{r_i(t)} \,.$$

Now, we have $r_i(t) < \frac{f_i}{weight(\mathcal{B}(p_i(t),r_i(t)))}$, which is a contradiction to Lemma 4.2. Hence, $r_i(t) \geq 2^{k_1-1}$ must be true, which proves the second inequality.

Furthermore, for any $r_i(t) > 2^{k_1+1}$, we have

$$weight(\mathcal{B}(p_i(t), r_i(t)/2)) \geq weight(\mathcal{B}(p_i(t), 2^{k_1})) \geq f_i \cdot 2^{-k_1} > f_i \cdot \frac{2}{r_i(t)} \,.$$

In this case, it follows that $r_i(t) > \frac{2f_i}{weight(\mathcal{B}(p_i(t),r_i(t)/2))}$, which is again a contradiction to Lemma 4.2. Thus, we have $r_i(t) \leq 2^{k_1+1}$, which proves the first inequality. □

Our algorithm uses the approach of [9], but we approximate the sum of the demands of all the points in a distance 2^k, for an integer k, by a cube with radius 2^k. This leads to the following result:

Lemma 4.4 *For any point of time t, let k_0 be the minimum integer k with $\lceil \log(\frac{\min_{p_j\in\mathcal{P}} f_j}{n\cdot\max_{p_j\in\mathcal{P}} d_j}) \rceil \leq k \leq \lfloor \log(\frac{\max_{p_j\in\mathcal{P}} f_j}{\min_{p_j\in\mathcal{P}} d_j}) \rfloor$, such that $weight(\mathcal{C}(p_i(t), 2^k)) \geq 2^{-k}$. Then $\frac{1}{4\sqrt{d}} \cdot r_i(t) \leq 2^{k_0} \leq 2 \cdot r_i(t)$ holds.*

Proof: Let k_1 be the minimum integer k with $\lceil \log(\frac{\min_{p_j\in\mathcal{P}} f_j}{n\cdot\max_{p_j\in\mathcal{P}} d_j}) \rceil \leq k \leq \lfloor \log(\frac{\max_{p_j\in\mathcal{P}} f_j}{\min_{p_j\in\mathcal{P}} d_j}) \rfloor$, such that $weight(\mathcal{B}(p_i(t), 2^k)) \geq 2^{-k}$. Then the radius of $\mathcal{C}(p_i(t), 2^{k_0})$ is at most 2^{k_1}, since each point in $\mathcal{P}(t)$ that is located in $\mathcal{B}(p_i(t), 2^{k_1})$ is also located in $\mathcal{C}(p_i(t), 2^{k_1})$, so that $weight(\mathcal{C}(p_i(t), 2^{k_1})) \geq 2^{-k_1}$. Furthermore, the radius of $\mathcal{C}(p_i(t), 2^{k_0})$ is larger than $\frac{1}{\sqrt{d}} \cdot 2^{k_1-1}$. The reason is that $weight(\mathcal{B}(p_i(t), 2^{k_1-1})) < 2^{-(k_1-1)}$ and $weight(\mathcal{C}(p_i(t), \frac{1}{\sqrt{d}} \cdot 2^{k_1-1})) \leq weight(\mathcal{B}(p_i(t), 2^{k_1-1}))$, so that

$$weight(\mathcal{C}(p_i(t), 2^{k_1-1-\log(\sqrt{d})})) = weight(\mathcal{C}(p_i(t), \frac{1}{\sqrt{d}} \cdot 2^{k_1-1})) < 2^{-(k_1-1)} < 2^{-(k_1-1-\log(\sqrt{d}))}.$$

Now, due to the fact that $2^{k_0} > \frac{1}{\sqrt{d}} \cdot 2^{k_1-1}$ and Lemma 4.3, the lemma follows. The maximum and minimum radius of $\mathcal{C}(p_i(t), 2^{k_0})$ is illustrated in Figure 2. □

Due to Lemma 4.4, we have $2^{-\log(4\sqrt{d})} \cdot r_i(t) \leq 2^{k_0} \leq 2 \cdot r_i(t)$. Therefore, to get $r_i(t) \leq r_i^*(t)$, we set $r_i^*(t) = 2^{k^*} = 2^{k_0+\lceil\log(4\sqrt{d})\rceil)}$. This implies $r_i(t) \leq r_i^*(t) \leq 2^{3+\lceil\log(\sqrt{d})\rceil} \cdot r_i(t)$.

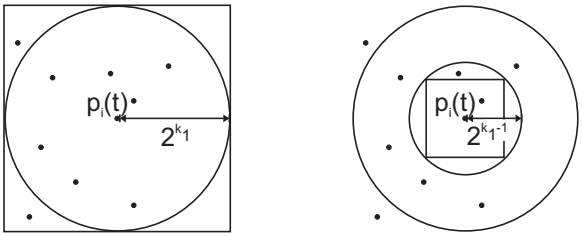


Figure 2: Maximum and minimum radius of $\mathcal{C}(p_i(t), 2^{k_0})$.

Lemma 4.5 *The KDS for the facility location problem for an arbitrary but fixed dimension d maintains, at any point of time t, a subset $\mathcal{F}(t) \subseteq \mathcal{P}(t)$ such that*

$$cost(\mathcal{F}(t)) \leq (64d+1) \cdot cost(\mathcal{F}_{\text{OPT}}(t)) .$$

Proof: For each point $p_i(t) \in \mathcal{P}(t)$, there is a facility $p_j(t) \in \mathcal{F}(t)$ with radius $r_j^*(t) \leq r_i^*(t)$ in $\mathcal{C}(p_i(t), 4 \cdot r_i^*(t))$. Furthermore, we know that $r_i(t) \leq r_i^*(t) \leq 2^{3+\lceil \log(\sqrt{d}) \rceil} \cdot r_i(t)$. Thus, we get $D(p_i(t), p_j(t)) \leq \sqrt{d} \cdot 4 \cdot r_i^*(t) \leq \sqrt{d} \cdot 4 \cdot 2^{3+\lceil \log(\sqrt{d}) \rceil} \cdot r_i(t) \leq 64d \cdot r_i(t)$. Now, the lemma follows from the analysis in [33]. Details can be found in the appendix. □

Complexity. In the remainder of this section, we analyze our KDS in terms of its complexity. Due to Lemma 3.1, we have already proven that our KDS is compact and local. The following lemma shows that the requirement for responsiveness is also fulfilled.

Lemma 4.6 *Each event requires $\mathcal{O}(\log(nR))$ status changes and $\mathcal{O}(\log^{d+1}(n) \cdot \log(nR))$ update time.*

Proof: Due to Lemma 3.1, the time to update the event queue is $\mathcal{O}(\log(nR))$. In the case that the event does not indicate that one point crosses a wall of another point, we have already finished processing the event. Otherwise, we have to update two points in the range trees and the radius of one point. Because we use binary search for updating the radius, both operations require $\mathcal{O}(\log\log(nR) \cdot \log^{d+1}(n))$ time. The time required to check, whether a point violates the invariant or not, is $\mathcal{O}(\log^{d+1}(n))$. To change the status of one point, we have to perform one remove and one insert operation on the range trees. This can be done in $\mathcal{O}(\log^{d+1}(n))$ time. Next we examine the time needed for algorithm RESTORE. We consider the running time resulting for restoring the invariant at points with radius 2^k. The number of cubelets with radius 2^k in $\mathcal{C}(p_e(t), 6 \cdot 2^{k+1})$ is 12^d. The query of open or closed points for one cubelet can be answered by T_1 and T_2 in time $\mathcal{O}(\log^{d+1}(n))$. Afterwards, there has to be at most one point inserted and deleted in T_1 and T_2. This requires $\mathcal{O}(\log^{d+1}(n))$ time. By summation over all radii, we get a total running time of $\mathcal{O}(\log(nR) \cdot 12^d \cdot \log^{d+1}(n)) = \mathcal{O}(\log^{d+1}(n) \cdot \log(nR))$.

There can exist at most one facility with radius 2^k in a cubelet with radius 2^k, because otherwise at least one facility would violate the invariant. Hence, the number of facilities with radius 2^k that are closed while running algorithm RESTORE is constant. Furthermore, we open at most one facility in each cubelet, so that the number of opened facilities with radius 2^k is also constant. Due to the fact that we handle $\mathcal{O}(\log(nR))$ radii, there are $\mathcal{O}(\log(nR))$ status changes per event. □

Due to Lemma 3.1 and Lemma 4.6, the total processing time is $\mathcal{O}(n^2 \log^{d+1}(n) \cdot \log^3(nR))$. Thus, we get the following result:

Theorem 4.1 *Let $\mathcal{P}$ be a set of n independently moving points in $\mathbb{R}^d$, where d is a constant. Then there is a KDS for the facility location problem that maintains, at any point of time t, a set $\mathcal{F}(t) \subseteq \mathcal{P}(t)$, such that $cost(\mathcal{F}(t)) \leq (64d+1) \cdot cost(\mathcal{F}_{\text{OPT}}(t))$. The KDS has a space requirement of $\mathcal{O}(n(\log^d(n) + \log(nR)))$, where $R = \frac{\max_{p_i \in \mathcal{P}} f_i \cdot \max_{p_i \in \mathcal{P}} d_i}{\min_{p_i \in \mathcal{P}} f_i \cdot \min_{p_i \in \mathcal{P}} d_i}$. Each update operation requires $\mathcal{O}(\log(nR))$ status changes and $\mathcal{O}(\log^{d+1}(n) \cdot \log(nR))$ time. In the case that each trajectory can be described by a bounded degree polynomial, the total number of updates is $\mathcal{O}(n^2 \log^2(nR))$, which results in a total processing time of $\mathcal{O}(n^2 \log^{d+1}(n) \cdot \log^3(nR))$. A flight plan update involves $\mathcal{O}(\log(nR))$ certificates and requires $\mathcal{O}(\log^2(nR))$ time.*

5 Conclusion and Future Work

In this paper, we initiated the study on the kinetic facility location problem. In particular, we proposed a KDS that maintains a subset of the moving input points as facilities such that, at any point of time, the associated total cost is at most a constant factor larger than the current optimal cost. We showed that our KDS is compact, local, and responsive. We also consider our algorithm to be efficient, although we cannot prove this in the formal sense of KDS, because it is hard to lower bound the number of mandatory events in a non-trivial way. We did not focus on fine-tuning the approximation factor and leave this as furure work.

Due to the local structure of the solution and the motivation from mobile ad-hoc networks, it might be worthwhile to extend our results to a dynamic or distributed setting as well. Future work in the area of kinetic facility location problems could include to consider additional opening cost that arises at the moment when a point changes its status from client to facility. Here we point out that in our scenario the opening cost per event would be already bounded, because we open at most a logarithmic number of facilities per event.

References

[1] P. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points.. In *Proceedings of the ACM SIGMOD Symposium on Principles of Database Systems (PODS)*, pp. 175–186, 2000.

[2] M. Abam and M. de Berg and B. Speckmann. Kinetic KD-trees and longest-side KD-trees. In *Proceedings of the 23rd annual symposium on Computational Geometry*, pp. 364–372, 2007.

[3] P. Agarwal, J. Basch, M. de Berg, L. Guibas, and J. Hershberger. Lower bounds for kinetic planar subdivisions. *Proceedings of the 15th Annual ACM Symposium on Computational Geometry*, pp. 247–254, 1999.

[4] P. Agarwal, J. Erickson, and L. Guibas. Kinetic binary space partitions for intersecting segments and disjoint triangles. *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 107–116, 1998.

[5] P. Agarwal, J. Gao, and L. Guibas. Kinetic Medians and *kd*-Trees. *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, pp. 5–16, 2002.

[6] P. Agarwal, L. Guibas, J. Hershberger, and E. Veach. Maintaining the Extent of a Moving Point Set. *Discrete & Computational Geometry*, 26(3):353–374, 2001.

[7] P. Agarwal, L. Guibas, T. Murali, and J. Vitter. Cylindrical static and kinetic binary space partitions. *Computational Geometry: Theory and Applications*, 16(2):103–127, 2000.

[8] P. Agarwal, S. Har-Peled, and K. Varadarajan. Approximating extent measures of points. *Journal of the ACM*, 51(4):606–635, 2004.

[9] M. Bădoiu, A. Czumaj, P. Indyk, and C. Sohler. Facility location in sublinear time. *Proceedings of the 32nd Annual International Colloquium on Automata, Languages and Programming (ICALP)*, 3580:866–877, 2005.

[10] J. Basch, J. Erickson, L. Guibas, J. Hershberger, and L. Zhang. Kinetic collision detection between two simple polygons. *Journal of Computational Geometry: Theory and Applications*, 27(3): 211–235, 2004.

[11] J. Basch and L. Guibas and J. Hershberger. Data structures for mobile data. *Journal of Algorithms*, 31(1): 1–28, 1999.

[12] J. Basch and L. Guibas and L. Zhang. Proximity problems on moving points. In *Proceedings of the 13th annual Symposium on Computational Geometry*, pp. 344–351, 1997.

[13] S. Bespamyatnikh, B. Bhattacharya, D. Kirkpatrick, and M. Segal. Mobile facility location. In *Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications*, pp. 46–53, 2000.

[14] M. Charikar and S. Guha. Improved combinatorial algorithms for the facility location and k-median problems. *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 378–388, 1999.

[15] F. A. Chudak. Improved approximation algorithms for uncapacitated facility location. *Proceedings of the 6th International Integer Programming and Combinatorial Optimization Conference (IPCO)*, pp. 180–194, 1998.

[16] A. Czumaj, G. Frahling, and C. Sohler. Efficient kinetic data structures for MaxCut. *Proceedings of the 19th Canadian Conference on Computational Geometry (CCCG)*, 2007.

[17] J. Gao, L. Guibas, and A. Nguyen. Deformable spanners and applications. In *Proceedings of the 20th Annual ACM Symposium on Computational Geometry*, pp. 190–199, 2004.

[18] J. Gao, L. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Discrete mobile centers. *Journal of Discrete and Computational Geometry*, 30(1):45–63, 2003.

[19] J. Gehweiler and C. Lammersen and C. Sohler. A distributed O(1)-approximation algorithm for the uniform facility location problem. In *Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 237–243, 2006.

[20] S. Guha and S. Khuller. Greedy strikes back: Improved facility location algorithms. *Journal of Algorithms*, 31(1): 228–248, 1999.

[21] L. Guibas. Kinetic data structures: A state of the art report. In *Proceedings of the 3rd Workshop on Algorithmic Foundations of Robotics*, 1998.

[22] L. Guibas, J. Hershberger, S. Suri, and L. Zhang. Kinetic connectivity for unit disks. *Proceedings of the 16th Annual ACM Symposium on Computational Geometry*, pp. 331–340, 2000.

[23] S. Har-Peled. Clustering Motion. *Proceedings of the 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 84–93, 2001.

[24] J. Hershberger. Smooth kinetic maintenance of clusters. *Proceedings of the Annual ACM Symposium on Computational Geometry*, pp. 48–57, 2003.

[25] J. Hershberger. Kinetic collision detection with fast flight plan changes. *Journal of Information Processing Letters*, 92(6): 287–291, 2004.

[26] J. Hershberger and S. Suri. Simplified kinetic connectivity for rectangles and hypercubes. *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 158–167, 2001.

[27] P. Indyk. Algorithms for dynamic geometric problems over data streams. *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 373–380, 2004.

[28] K. Jain, M. Mahdian, and A. Saberi. A new greedy approach for facility location problems. *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 731–740, 2002.

[29] K. Jain and V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *Journal of the ACM*, 48(2): 274–296, 2001.

[30] D. Kirkpatrick, J. Snoeyink, and B. Speckmann. Kinetic collision detection for simple polygons. *Proceedings of the 16th Annual ACM Symposium on Computational Geometry*, pp. 322–330, 2000.

[31] S. Kolliopoulos and S. Rao. A Nearly Linear-Time Approximation Scheme for the Euclidean k-Median Problem. *SIAM Journal on Computing*, 37(3):757–782, 2007.

[32] M. Mahdian, Y. Ye, and J. Zhang. Improved approximation algorithms for metric facility location problems. *Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, pp. 229–242, 2002.

[33] R. R. Mettu and C. G. Plaxton. The online median problem. In *SIAM J. Comput.*, 32(3): 816–832, 2003.

[34] T. Moscibroda and R. Wattenhofer. Facility Location: Distributed Approximation. In *Proceedings of the 24th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pp. 108–117, 2005.

[35] D. B. Shmoys, E. Tardos, and K. Aardal. Approximation algorithms for facility location problems. *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 265–274, 1997.

[36] M. Thorup. Quick k-median, k-center, and facility location for sparse graphs. *SIAM Journal on Computing*, 34(2):405–432, 2005.

A Proof of Lemma 4.5

The following analysis is basically the same as in [33]. Only a few adjustments to our scenario have been made.

Claim A.1 *For any point $p_i(t) \in \mathcal{P}(t)$, there exists a point $p_j(t) \in \mathcal{F}(t)$, such that $r_j^*(t) \leq r_i^*(t)$ and $D(p_i(t), p_j(t)) \leq 64d \cdot r_i(t)$.*

Proof: Since, for each point $p_i(t) \in \mathcal{P}(t)$, there is a facility $p_j(t) \in \mathcal{F}(t)$ with radius $r_j^*(t) \leq r_i^*(t)$ in $\mathcal{C}(p_i(t), 4 \cdot r_i^*(t))$, we get $D(p_i(t), p_j(t)) \leq \sqrt{d} \cdot 4 \cdot r_i^*(t)$. Now, due to Lemma 4.4 and the definition of $r_i^*(t)$, we have $D(p_i(t), p_j(t)) \leq \sqrt{d} \cdot 4 \cdot 2^{3+\lceil \log(\sqrt{d}) \rceil} \cdot r_i(t) \leq 64d \cdot r_i(t)$. □

Claim A.2 *Let $p_i(t)$ and $p_j(t)$ be distinct points in $\mathcal{F}(t)$. Then $D(p_i(t), p_j(t)) > 2 \cdot \max\{r_i(t), r_j(t)\}$.*

Proof: W.l.o.g., $r_j^*(t) \leq r_i^*(t)$. From the fact that the invariant is always restored after an event occurred, it follows that $p_j(t) \notin \mathcal{C}(p_i(t), 2 \cdot r_i^*(t))$. Thus, we have $D(p_i(t), p_j(t)) > 2 \cdot r_i^*(t) \geq 2 \cdot r_i(t)$ and $D(p_i(t), p_j(t)) > 2 \cdot r_i^*(t) \geq 2 \cdot r_j^*(t) \geq 2 \cdot r_j(t)$. □

For any point $p_j(t) \in \mathcal{P}(t)$ and an arbitrary set of facilities $\mathcal{X}(t) \subseteq \mathcal{P}(t)$, let

$$charge(p_j(t), \mathcal{X}(t)) = D(p_j(t), \mathcal{X}(t)) + \sum_{p_i(t) \in \mathcal{X}(t)} \max\{0, r_i(t) - D(p_i(t), p_j(t))\}\,.$$

Claim A.3 *For an arbitrary set of facilities $\mathcal{X}(t) \subseteq \mathcal{P}(t)$, we get*

$$\sum_{p_j(t) \in \mathcal{P}(t)} charge(p_j(t), \mathcal{X}(t)) \cdot d_j = cost(\mathcal{X}(t))\,.$$

Proof: We get

$$\begin{aligned}
&\sum_{p_j(t) \in \mathcal{P}(t)} charge(p_j(t), \mathcal{X}(t)) \cdot d_j \\
= &\sum_{p_i(t) \in \mathcal{X}(t)} \sum_{p_j(t) \in \mathcal{B}(p_i(t), r_i(t))} (r_i(t) - D(p_i(t), p_j(t))) \cdot d_j + \sum_{p_j(t) \in \mathcal{P}(t)} D(p_j(t), \mathcal{X}(t)) \cdot d_j \\
= &\sum_{p_i(t) \in \mathcal{X}(t)} f_i + \sum_{p_j(t) \in \mathcal{P}(t)} D(p_j(t), \mathcal{X}(t)) \cdot d_j\,.
\end{aligned}$$

□

Claim A.4 *Let $p_j(t) \in \mathcal{P}(t)$ be a point, let $\mathcal{X}(t) \subseteq \mathcal{P}(t)$ an arbitrary set of facilities, and let $p_i(t) \in \mathcal{X}(t)$. If $D(p_j(t), p_i(t)) = D(p_j(t), \mathcal{X}(t))$ then $charge(p_j(t), \mathcal{X}(t)) \geq \max\{r_i(t), D(p_j(t), p_i(t))\}$.*

Proof: If $p_j(t) \notin \mathcal{B}(p_i(t), r_i(t))$ then $charge(p_j(t), \mathcal{X}(t)) \geq D(p_j(t), p_i(t)) > r_i(t)$. Otherwise, $charge(p_j(t), \mathcal{X}(t)) \geq (r_i(t) - D(p_j(t), p_i(t))) + D(p_j(t), p_i(t)) = r_i(t) \geq D(p_j(t), p_i(t))$. □

Claim A.5 *Let $p_j(t) \in \mathcal{P}(t)$ be a point and let $p_i(t) \in \mathcal{F}(t)$. If $p_j(t) \in \mathcal{B}(p_i(t), r_i(t))$, then*

$$charge(p_j(t), \mathcal{F}(t)) \leq r_i(t)\,.$$

Proof: By Claim A.2, there is no point $p_\ell(t) \in \mathcal{F}(t)$ such that $i \neq \ell$ and $p_j(t) \in \mathcal{B}(p_\ell(t), r_\ell(t))$. The lemma now follows from the definition of $charge(p_j(t), \mathcal{F}(t))$, since $D(p_j(t), \mathcal{F}(t)) \leq D(p_j(t), p_i(t))$. □

Claim A.6 *Let $p_j(t) \in \mathcal{P}(t)$ be a point and let $p_i(t) \in \mathcal{F}(t)$. If $p_j(t) \notin \mathcal{B}(p_i(t), r_i(t))$, then*

$$charge(p_j(t), \mathcal{F}(t)) \leq D(p_j(t), p_i(t))\,.$$

Proof: The lemma follows immediately unless there is a point $p_\ell(t) \in \mathcal{F}(t)$ such that $p_j(t) \in \mathcal{B}(p_\ell(t), r_\ell(t))$. If such a point $p_\ell(t)$ exists, then Claims A.2 and A.5 imply $D(p_i(t), p_\ell(t)) > 2 \cdot \max\{r_i(t), r_\ell(t)\}$ and $charge(p_j(t), \mathcal{F}(t)) \leq r_\ell(t)$, respectively. The lemma now follows since

$$D(p_j(t), p_i(t)) \geq D(p_i(t), p_\ell(t)) - D(p_j(t), p_\ell(t)) > 2r_\ell(t) - r_\ell(t) = r_\ell(t)\,.$$

□

Claim A.7 *For any point $p_j(t) \in \mathcal{P}(t)$ and an arbitrary set of facilities $\mathcal{X}(t) \subseteq \mathcal{P}(t)$,*

$$charge(p_j(t), \mathcal{F}(t)) \leq (64d + 1) \cdot charge(p_j(t), \mathcal{X}(t))\,.$$

Proof: Let $p_i(t)$ be some point in $\mathcal{X}(t)$ such that $D(p_j(t), p_i(t)) = D(p_j(t), \mathcal{X}(t))$. By Claim A.1, there exists a point $p_\ell(t) \in \mathcal{F}(t)$ such that $r^*_\ell(t) \leq r^*_i(t)$ and $D(p_i(t), p_\ell(t)) \leq 64d \cdot r_i(t)$.

If $p_j(t) \in \mathcal{B}(p_\ell(t), r_\ell(t))$, then $charge(p_j(t), \mathcal{F}(t)) \leq r_\ell(t)$ by Claim A.5. The lemma follows since $r_\ell(t) \leq r^*_\ell(t) \leq r^*_i(t) \leq \sqrt{d} \cdot 4 \cdot 2^{3+\lceil \log(\sqrt{d}) \rceil} \cdot r_i(t) \leq 64d \cdot r_i(t)$ and Claim A.4 implies $charge(p_j(t), \mathcal{X}(t)) \geq r_i(t)$.

If $p_j(t) \notin \mathcal{B}(p_\ell(t), r_\ell(t))$, then $charge(p_j(t), \mathcal{F}(t)) \leq D(p_j(t), p_\ell(t))$ by Claim A.6. Thus,

$$charge(p_j(t), \mathcal{F}(t)) \leq D(p_j(t), p_i(t)) + D(p_i(t), p_\ell(t)) \leq D(p_j(t), p_i(t)) + 64d \cdot r_i(t)\,.$$

The lemma now follows by Claim A.4, since the ratio of $D(p_j(t), p_i(t)) + 64d \cdot r_i(t)$ to the maximum of $r_i(t)$ and $D(p_j(t), p_i(t))$ is at most $64d + 1$. □

Due to Claims A.3 and A.7, we have $cost(\mathcal{F}(t)) \leq (64d + 1) \cdot cost(\mathcal{X}(t))$ for an arbitrary set of facilities $\mathcal{X}(t) \subseteq \mathcal{P}(t)$. Thus, the approximation factor is also true for an optimal set of facilities $\mathcal{F}_{\text{OPT}}(t)$, which completes the lemma.