Continuous Monitoring of Exclusive Closest Pairs*

Leong Hou U[1], Nikos Mamoulis[1], and Man Lung Yiu[2]

[1] Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong,
{hleongu,nikos}@cs.hku.hk
[2] Department of Computer Science
Aalborg University
DK-9220 Aalborg, Denmark
mly@cs.aau.dk

Abstract. Given two datasets A and B, their exclusive closest pairs (ECP) join is a one-to-one assignment of objects from the two datasets, such that (i) the closest pair (a, b) in $A \times B$ is in the result and (ii) the remaining pairs are determined by removing objects a, b from A, B respectively, and recursively searching for the next closest pair. An application of exclusive closest pairs is the computation of (car, parking slot) assignments. In this paper, we propose algorithms for the computation and continuous monitoring of ECP joins in memory, given a stream of events that indicate dynamic assignment requests and releases of pairs. Experimental results on a system prototype demonstrate the efficiency of our solutions in practice.

1 Introduction

Due to the increasing popularity of location-based services, continuous monitoring of spatial queries emerges as an important research topic. Existing work [18, 11, 21, 13, 16, 17] focuses on *range* or k *nearest neighbor* (kNN) queries on moving objects. These problems can also be viewed as continuous joins between queries and data objects, according to their spatial relationship. However, there has not been much research done related to the continuous monitoring of spatial join results. Several variants of spatial join queries exist, such as the intersection join [2], the distance (or similarity) join [14], the all k nearest neighbors join [24], and the k (inclusive) closest pairs query (kICP) [9, 4].

In this paper, we study an interesting type of spatial joins that has received little attention in the past. We call this operation the k *exclusive closest pairs* join (kECP). kECP produces k one-to-one assignments of objects between two datasets A and B, such that (i) the closest pair (a, b) in $A \times B$ belongs to the result and (ii) the remaining pairs are determined by removing objects a, b from A, B respectively, and recursively searching for the next closest pair. Thus, each object appears only once in the result.

A real-life application of a kECP query is the car-parking assignment problem. Consider a set A of car drivers that request for a parking slot and another set B of available slots. The well-known assignment problem [19] searches for the 1-to-1 assignment of cars to parking spaces, such that the sum of travel distances is minimized. However, in a world of selfish users, it is more reasonable to assign each car $c \in A$ to the parking space $p \in B$ that may not be taken by another driver c', which happens to be closer

* Supported by grant HKU 7160/05E from Hong Kong RGC.

to p than c is. Therefore, our formulation of the kECP query (assuming that k is the minimum of cardinalities $|A|$ and $|B|$) searches for a practical solution to the problem.

We propose a technique for computing the ECP pairs efficiently given a set of cars and a set of parking slots. In addition, we extend it to monitor the ECP results, in a dynamic environment, where parking requests from cars and availability events from parking slots arrive from a data stream. Due to such events, ECP assignments must be deleted (i.e., when a car un-parks), new assignments must be added (i.e., when a new car requests parking), and current assignments may have to be changed. For instance, assume that pair (c, p) is in the current assignment and a new parking slot p' becomes available which is closer to c than p is. In this case, c must be re-assigned to p' and p should become available for other cars. This change may trigger a "chaining" effect which could alter the whole assignment. Our method processes incoming events in an appropriate order, such that the correct ECP results are maintained correctly and efficiently.

We assume that a centralized server monitors the locations of objects. When an object moves to another location, it informs the server about its new location. Since the frequent updates render disk-based management techniques inefficient, our solution is based on a memory grid-based indexing approach [16, 18].

Our contributions can be summarized as follows:

- We identify ECP as a new type of spatial join that finds application in real-life dynamic allocation problems (e.g., car/parking assignment).
- We show that the kECP for $k = \min\{|A|, |B|\}$ is equivalent to a special case of the stable marriage problem [7], where assignment preferences are derived from the distance function. Based on this observation, we adapt the Gale-Shapley algorithm [6] to solve kECP queries by computing only a small fraction of the distances dynamically and on-demand.
- We define a dynamic version of ECP for moving objects and streaming events that indicate (i) availability of slots and (ii) demand for new kECP pairs. We propose an appropriate extension of our static kECP query evaluation algorithm that solves this continuous kECP query.
- We conduct a set of experiments to verify the efficiency of the proposed methods for a wide range of problem parameters.

The rest of the paper is organized as follows. Section 2 surveys related work on closest pair queries in spatial data, continuous monitoring problems, and the stable marriage problem. Section 3 formally defines ECP and presents our solution to it for a static input. Section 4 presents an update framework for ECP calculation with two optimizations to improve its performance. Our solutions are evaluated in Section 5. Finally, Section 6 concludes the paper, giving directions for future work.

2 Background and Related Work

2.1 Closest Pairs Queries in Spatial Databases

Computation of closest pairs queries have been studied for several decades. Main-memory algorithms, such as the Neighbor Heuristic [1] and Fast Pair [3, 5], focus on 1CP problems. Fast Pair was shown to have the best overall performance. However, this

method is not directly applicable to: (i) kCP queries for arbitrary values of k, and (ii) other variants of CP queries.

Some previous work [4, 9, 22] employ spatial indexes to solve kICP queries in secondary memory. [4, 9] assume that the datasets are indexed by R-trees [8]. On the other hand, Yang et al. [22] extended the R-tree to a b-Rdnn tree, by augmenting each non-leaf entry with the maximum nearest neighbor distance (with respect to the other dataset) of points in its subtree. During query evaluation, such distances are utilized for reducing the search space. [22] showed that their approach outperforms previous R-tree based methods. Since these methods operate on indexed data they may not be applied in a dynamic environment. A high rate of streaming events imposes a high burden to the update of the indexes, which in combination with the expensive refreshing of the query results, renders the overall approach inefficient or impossible. In addition, although an kICP algorithm can be tuned to process the kECP query (i.e., by remembering assigned points and avoiding their re-assignment), such an approach would require a large amount of memory (for k=$\min\{|A|, |B|\}$, as large as the size of a dataset).

2.2 Continuous Monitoring of Spatial Queries

Various spatial applications, like the car-parking problem of the Introduction, handle large amounts of information at fast arrival rate. Several extensions of R-trees have been developed for supporting frequent updates of spatial data. Lee et al. [15] proposed the FUR-tree (Frequent Update R-tree), which uses localized bottom-up update strategies into the traditional R-tree. Recently, Xiong et al. [20] developed the RUM-tree (R-tree with Update Memo), which was shown to have better update performance than FUR-tree. [12] applied an event-driven approach to maintain query results for kNN and spatial join queries, with the assumption that moving objects can be modeled by linear motion functions.

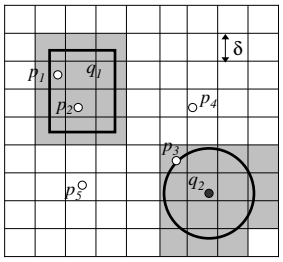


Fig. 1. Monitoring spatial queries

Continuous monitoring of *multiple* spatial queries (e.g., range [16, 17] and kNN [21, 23, 18]) adopt the *shared execution paradigm* to reduce the processing cost. Instead of monitoring the results for different queries separately, the problem is viewed as a large spatial join between the query objects and data objects. As illustrated in Figure 1, grid cells (of cell length δ) are employed for indexing the objects. In practice, memory grid cells [23, 18] are used (instead of disk-based structures) in order to handle very high update rate. q_1 corresponds to a range query (shown in bold rectangle) and its *influence region* consists of the (gray) cells that intersect with q_1. Since data object updates outside the influence region cannot affect the query result, the processing cost

is significantly reduced. As another example, q_2 represents a NN query (shown in bold circle). Its difference from q_1 is that its influence region is a circular region centered at q_2 with dynamic radius equal its NN distance. For example, when the NN of q_2 moves closer to (further from) q_2, then the influence region of q_2 shrinks (grows).

Observe that continuous monitoring of range/kNN queries is different from that of kECP queries. For range and kNN queries, only query results near a change triggered by a streaming event (i.e., appearance, disappearing, or movement of an object) need to be updated (i.e., only for queries whose influence region intersects the location of the change). On the other hand, as we will discuss in Section 4, a streaming event can generate a sequence of changes in the kECP result. Thus, the idea of influence regions is not appropriate for kECP monitoring, which calls for novel techniques.

2.3 The Stable Marriage Problem

The kECP join is closely related to the classic stable marriage problem [6, 7]. Given two set of objects A and B, M is said to be a *matching* between A and B if (i) M is a set of $\min\{|A|, |B|\}$ pairs of objects (a, b) where $a \in A$, $b \in B$, and (ii) each object $a \in A$ ($b \in B$) appears in at most one pair in M. A matching M is *stable* if there are no pairs (a, b) and (a', b') in M such that a prefers b' to b and b' prefers a to a'. Given the preference lists of all objects $a \in A$ and $b \in B$, the stable marriage problem seeks for a stable matching. In our context, the preference list of an object a is implicitly defined by the total order defined by the Euclidean distance; if a is closer to b than to b', then a prefers b to b'.

[7] is a nice reference text that introduces the stable marriage problem and presents solutions to it, for special cases of the input. For the generic problem, Gale and Shapley [6] proved that, if $|A| = |B|$, it is always possible to find a solution and provided an algorithm for this. For the ease of discussion, we call the objects in A and B as senders and receivers, respectively. In the first round, each sender (in A) calls its most preferred receiver (in B). If a receiver hears from at least one sender, then the receiver matches with the best sender (according to the receiver's preference) and the corresponding sender is removed from A. The above procedure is applied iteratively in subsequent rounds, but with an additional rule: if a receiver has been assigned a sender a_{old} (in previous rounds) and now it hears from a better sender a_{new} (in the current round), then the receiver matches with the new sender and the remaining set of senders becomes $A := \{a_{old}\} \cup A - \{a_{new}\}$. Eventually, the stable matching between A and B is obtained after all objects in A or B have been removed.

Job	**Preference**	**Applicant**	**Preference**
a_1	$b_1 \succ b_3 \succ b_2$	b_1	$a_1 \succ a_3 \succ a_2$
a_2	$b_1 \succ b_2 \succ b_3$	b_2	$a_2 \succ a_3 \succ a_1$
a_3	$b_2 \succ b_3 \succ b_1$	b_3	$a_2 \succ a_1 \succ a_3$

Table 1. Example of stable marriage

For example, Table 1 illustrates a set A of three jobs and a set B of three applicants, such that the applicants (jobs) can be totally ordered based on their qualification (preference) for the job (applicant). In the first round of the Gale-Shapley algorithm, both jobs a_1 and a_2 call the applicant b_1, who prefers a_1 to a_2. Thus, b_1 matches with a_1 and a_1 is removed from A. Also, a_3 calls b_2, b_2 matches with a_3 and a_3 is removed

from A. In the second round, a_2 calls b_2. Since b_2 prefers the new job a_2 to its old job a_3, b_2 now matches with a_2 instead and the job a_3 is added back to A. In the third round, a_3 calls b_3 and b_3 matches with a_3. Thus, the stable matching contains the pairs $(a_1, b_1), (a_2, b_2), (a_3, b_3)$. Note that at least one pair is finalized at each round, thus the worst-case time complexity of the algorithm is $O(|A| \times |B|)$.

The stable marriage algorithm is asymmetric; if the roles of A and B are reversed, a different solution may be found. Furthermore, it has been shown that it is sender-optimal (i.e., A-optimal if A is the sender dataset); its execution will derive the optimal pair in B for any $a \in A$, for any order of examined objects from A. Thus, there is a unique solution when taking A as the sender input and another unique solution when taking B as the sender. We now prove that if the preference list is derived by a symmetric weight function w (e.g., Euclidean distance), such that $w(a, b) = w(b, a), \forall a \in A, b \in B$, then these two solutions are identical.

Theorem 1. *If preferences are defined by a weight function w, such that a prefers b to b' if and only if $w(a, b) < w(a, b')$ and b prefers a to a' if and only if $w(b, a) < w(b, a')$, and $w(a, b) = w(b, a)$, for any $a, a' \in A$, $b, b' \in B$ then the optimal stable marriage result is unique independently on whether A or B is the sender set.*

Proof. Without loss of generality, assume that $|A| = |B| = n$. Let M_A=$\{\{(a_{(1)}, b_{(1)})\}$, $\{(a_{(2)}, b_{(2)})\}, \ldots, \{(a_{(n)}, b_{(n)})\}\}$ be the A-optimal matching, such that $(a_{(i)}, b_{(i)})$ models the pair which is finalized at the i-th loop of the Gale-Shapley algorithm.[1] Let the B-optimal matching, generated by the Gale-Shapley algorithm, be $M_B = \{\{(b'_{(1)}, a'_{(1)})\}$, $\{(b'_{(2)}, a'_{(2)})\}, ..., \{(b'_{(n)}, a'_{(n)})\}\}$. We will first prove that $a_{(1)} = a'_{(1)}$ and $b_{(1)} = b'_{(1)}$, i.e., the first assignments output by the two runs of the algorithm are identical. Since $(a_{(1)}, b_{(1)})$ is the first finalized pair of the A-sender run, $w(a_{(1)}, b_{(1)})$ should be the smallest $w(a, b)$, for any $a \in A$, $b \in B$. Similarly, $w(b'_{(1)}, a'_{(1)})$ should be the smallest $w(b, a)$, for any $a \in A$, $b \in B$. Since $w(a, b) = w(b, a)$, it must be $a_{(1)} = a'_{(1)}$ and $b_{(1)} = b'_{(1)}$. By induction, we can prove that $a_{(i)} = a'_{(i)}$ and $b_{(i)} = b'_{(i)}$, for $1 \leq i \leq n$, since by removing pairs $\{(a_{(1)}, b_{(1)}), (a_{(2)}, b_{(2)}), \ldots, (a_{(i)}, b_{(i)})\}$ from the problem we showed that the first pair $(a_{(i+1)}, b_{(i+1)})$ in the resulting subproblem is identical for both A-sender and B-sender runs. □

A subtle issue to note is that the uniqueness argument for the A-sender (or B-sender) Gale-Shapley's output and Theorem 1 holds only for cases where the preference lists are *unique*, strictly total orders. Non-unique orders can be derived from weight functions w, for which there exist pairs (a, b) and (a', b'), such that $w(a, b) = w(a', b')$ and $(a = a' \wedge b \neq b')$ or $(a \neq a' \wedge b = b')$. In such cases, e.g., $a = a' \wedge b \neq b'$, object a has the same preference to b and b', therefore the stable marriage result may not be unique; there could be a stable solution that includes (a, b) and another that includes (a, b').

3 The Static kECP Query

In this section, we define and solve the *static* case of the kECP query, where the kECP result is requested for two sets of static points. For completeness, we also provide the definition of the k *inclusive* closest pairs (kICP) query.

[1] Without loss of generality, we assume that only one pair is finalized at each loop. If there are multiple such pairs we could modify the algorithm to output only the one with the smallest $w(a, b)$, without affecting the correctness of the result.

Definition 1. *Given two set of points A, B and a $k < |A \times B|$, the k* inclusive closest pairs *$kICP(A, B)$ is defined as the set $S \subset A \times B$, such that $|S| = k$ and $\forall (a, b) \in S, (a', b') \in (A \times B) - S, d(a, b) \leq d(a', b')$.*

Definition 2. *Given two set of points A and B, the k* exclusive closest pairs *$kECP(A, B)$ is recursively defined as:*
$kECP(A, B) = kICP(A, B)$, for $k = 1$, and
$kECP(A, B) = 1ECP(A, B) \cup (k-1)ECP(A - \{a\}, B - \{b\})$, otherwise.

Note that the maximum possible value for k is $\min\{|A|, |B|\}$ in kECP and $k \leq |A| \cdot |B|$ in kICP. It is easy to prove that kECP, for $k = \min\{|A|, |B|\}$ is a special case of the stable marriage problem, where the preference order is derived by the weight function $w(a, b) = d(a, b)$ (d denotes Euclidean distance). Therefore the Gale-Shapley stable marriage algorithm (SMA) can be applied to solve kECP queries; the preference list of a point $a \in A$ is constructed by placing points in B in ascending order of their distances to a. The preference lists of points $b \in B$ are generated symmetrically. After running SMA on the preference lists, the obtained results correspond to the results of ECP. Since $d(a, b) \equiv d(b, a)$, and assuming that the distances between a point $a \in A$ and the points in B are distinct (and vice versa)[2], SMA will derive the unique ECP result according to Theorem 1, no matter whether we take A of B as the sender set.

Nevertheless, the direct application of SMA requires the computation of a large number of distances and large space to store them (for $|A| \cdot |B|$ distances), thus it does not scale well for large problems. We conjecture that the spatial properties of the query, in combination with appropriate indexes can be utilized to accelerate SMA. For example, we need not compute the distance of a point $a \in A$ to *all* in B before running SMA; instead, we can applying spatial ranking techniques [10, 18] to generate the preference list of a incrementally and on-demand.

We adopt CPM; the grid-based technique of [18] for indexing data points in our problem, due to its good performance in environments with frequent updates. CPM is the state-of-the-art grid-based index for monitoring NN queries. Each query point is associated with a heap such that the objects and grid cells are visited in ascending order of their distances from the query point. In this way, query results can be computed fast and unnecessary accesses to other points are avoided. In particular, [18] propose a conceptual partitioning of the cells (see Figure 2) for reducing distance computations. Each rectangle DIR_{lvl} is associated with a direction DIR and a level number lvl. The direction can be U (up), D (down), L (left), or R (right). The level number denotes the number of rectangles between DIR_{lvl} and the cell containing the query point q.

Our static ECP algorithm (see Algorithm 1) uses the CPM index to search for the optimal matching, and (due to the hardness of the ECP problem) it is more sophisticated compared to the simple NN algorithm of [18]. Recall that we have two datasets A and B in our problem. In order to optimize performance, we consider the smallest dataset as a *query* set (that will generate nearest neighbor lists to be used as preference lists in the stable marriage evaluation). Accordingly, the other dataset represents an *objects* set. For the ease of exposition, let A be the query set and B be the objects set. For each point o in A and B keep track of the following information: (i) its current ECP object

[2] This is a realistic assumption since distances are real numbers and they are unlikely to coincide.

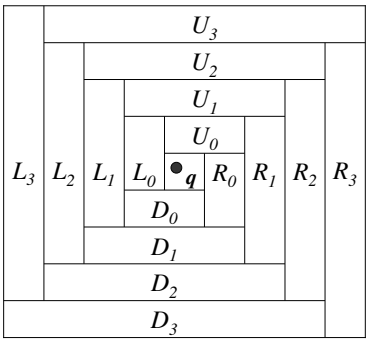


Fig. 2. Conceptual Partitioning Monitoring (CPM) space division

($o.\psi$), and (ii) the distance ($o.\lambda$) to that object. Initially, $o.\lambda$ is set to ∞, and $o.\psi$ is set to NULL. Table 2 summarizes the notation used in our algorithm description.

Symbol	Description
A, B	a set of points (cars), a set of points (parking slots)
a (b)	a point in A (B)
$d(a, b)$	Euclidean distance between a and b
$d_{min}(r, a)$	minimum distance between rectangle r and point a
DIR_{lvl}	rect. of direction (DIR) in level lvl (in CPM)
$a.\psi$ ($b.\psi$)	a's (b's) current ECP point
$a.\lambda$ ($b.\lambda$)	the distance between a (b) and its ECP point

Table 2. Notation

In its initialization phase (Lines 1–5), SECP allocates a min-heap $a.H$ for each object $a \in A$, and inserts in it $Cell(a)$ (i.e., the cell containing a) and all 0-level CPM rectangles (see Figure 2) that surround a. During SECP, $a.H$ contains cells, rectangles, and/or objects from B and can identify the one with the smallest d_{min} to a in O(1) time.[3] In addition, all points in A are inserted to a *patients* set P, containing query points that have not found their exclusive closest pair yet.

SECP then starts a sequence of iterations (Lines 7-16); after each loop a number of ECP pairs are identified and inserted to the result. At the i-th iteration, for each query point $a \in P$, SECP incrementally retrieves from B nearest neighbors of a which are no further than the i-th level rectangle of the CPM partition and attempts to find the ECP pair of a in them (Lines 10-12).

We now describe in more detail the core search module of SECP which is called at Line 12. Algorithm 2 is a pseudo-code for this ϵ-bounded incremental nearest neighbor search with integrated ECP assignment (ϵ-INNECP). ϵ-INNECP browses the nearest neighbors of a query point a incrementally, subject to the constraint that their distance to a is not greater than ϵ. At Lines 3-9, it processes the element on top of the $a.H$ heap, if it is a rectangle or a cell, exactly like the original NN algorithm of [18]. If the

[3] Given a point p and a rectangle r, $d_{min}(p, r)$ is the minimum distance between p and any possible point in r.

Algorithm 1 SECP

```
    V, P, P' : Queue
    Result : Heap
    algorithm SECP(Integer k)
 1: for all a ∈ A do
 2:     insert ⟨Cell(a), d_min(a, Cell(a))⟩ into a.H
 3:     for each direction DIR do
 4:         insert ⟨DIR_0, d_min(a, DIR_0)⟩ into a.H
 5:     insert a into P
 6: loop:=0
 7: while |Result| < k do
 8:     loop:=loop + 1
 9:     maxdist := (loop − 1/2) · δ
10:     while P ≠ ∅ do
11:         dequeue an object a from P
12:         ε-INNECP(a, maxdist, V, P, P')
13:     for all b ∈ V do
14:         if b = (b.ψ).ψ then
15:             insert (b.ψ, b) into Result
16:     P:=P'; P':=∅
```

next $a.H$ entry is an object b, it is processed according to Lines 11-19. If $d(a, b)$ is smaller than $b.\lambda$ (this happens if b is unassigned or b has been previously assigned to a further query point), then the current ECP of a (resp. b) is tentatively set to b (resp. a). If b is unassigned, we insert it into a *candidates* list V. Otherwise, the previous assigned pair of b ($b.\psi \in A$), is added to P and marked as unassigned. Then, $b.\lambda$ and $b.\psi$ are updated as $d(a, b)$ and a respectively. Search terminates if a is assigned to a point $b \in B$ (while-loop break of Line 19) or if a has not been assigned after all its ϵ-bounded nearest neighbors in B have been examined. In the latter case, a is inserted into next loop's patients list P' (Line 21). Note that ϵ-INNECP does not search for neighbors of a beyond ϵ distance from a, and ϵ is increased at each loop.

After each loop of SECP has examined all points in P, for each b in the candidate list V, it checks whether $a = b.\psi$ has also $a.\psi = b$ (Lines 13-15 of SECP). In this case (a, b) is definitely a pair in the ECP result. The reason is that $d(a, b) \leq \epsilon$ and there could not be an unassigned neighbor to a (or b) with a smaller distance (those have already been retrieved by ϵ-INNECP). The algorithm terminates when the number of results reaches k. Otherwise, ϵ-INNECP is invoked again with a new distance $\epsilon = (loop - 0.5) \cdot \delta$, where *loop* is the current loop and δ is the extent of a grid cell.

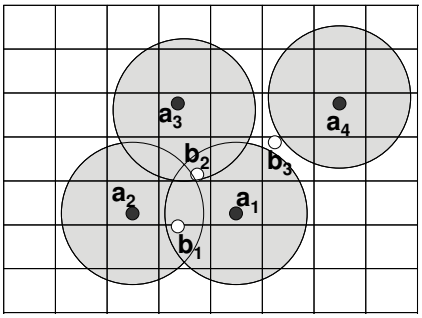
Figure 3 exemplifies how SECP algorithm works. Assume that 4 cars (in A) and 3 parking slots (in B) remain unassigned after the first loop. Then, ϵ is set to $(2 - 0.5) * \delta$, thus the maximum search range around each $a \in P$ is shown by the gray circles in Figure 3a. Assume that the order of points in P is (a_1, a_2, a_3, a_4). Figure 3b shows the running steps of this example in *loop*=2. At the first call of ϵ-INNECP, a_1 is assigned to b_1, since b_1 is the NN of a_1 and b_1 is currently unassigned. Then, a_2 takes b_1 and a_1 is put back to P (Lines 16-17 of ϵ-INNECP). This happens because (i) b_1 is the NN of

Algorithm 2 ϵ-bounded INN search and tentative ECP assignment

algorithm ϵ-INNECP(Object a, Distance ϵ, Queue V, P, P')

1: **while** $a.H \neq \varnothing$ and $a.H$'s top entry's distance $\leq \epsilon$ **do**
2: $\langle o, o_{dist} \rangle := deheap(a.H)$
3: **if** o is a cell c **then**
4: **for all** objects $b' \in c$ **do**
5: insert $\langle b', d(a, b') \rangle$ into $a.H$
6: **else if** o is a rectangle DIR_{lvl} **then**
7: **for** each cell c' in DIR_{lvl} **do**
8: insert $\langle c', d_{min}(a, c') \rangle$ into $a.H$
9: insert $\langle DIR_{lvl+1}, d_{min}(a, DIR_{lvl+1}) \rangle$ into $a.H$
10: **else** ▷ o is an object b
11: **if** $b.\lambda > o_{dist}$ **then** ▷ b prefers a to its previous pair
12: set $a.\psi := b$ and $a.\lambda := o_{dist}$ ▷ update ECP for a
13: **if** $b.\psi$ is NULL **then**
14: insert b into V ▷ insert to ECP candidates V
15: **else**
16: $(b.\psi).\psi$:=NULL; $(b.\psi).\lambda$:=∞ ▷ unset pair of b
17: insert $b.\psi$ into P
18: set $b.\psi := a$ and $b.\lambda := o_{dist}$ ▷ update ECP for b
19: break ▷ break while-loop
20: **if** $a.\psi$=NULL **then** ▷ a has not been assigned in this loop
21: insert a into P'

a_2 and (ii) a_1 is the current ECP pair of b_1 and $d(a_2, b_1) < d(a_1, b_1)$. The algorithm continues and eventually outputs the assignments (a_2, b_1) and (a_1, b_2), whereas $P' = \{a_3, a_4\}$ are moved to the next loop (so is b_3). Although ϵ-INNECP runs with a larger searching area in the next loop, it avoids accessing unnecessary elements, because it continues searching using the current min-heap $a.H$ for each $a \in P$.



(a) locations of points

Loop	a	$a.\psi$	P	P'
2	a_1	b_1	(a_2, a_3, a_4)	-
2	a_2	b_1	(a_3, a_4, a_1)	-
2	a_3	b_2	(a_4, a_1)	-
2	a_4	-	(a_1)	(a_4)
2	a_1	b_2	(a_3)	(a_4)
2	a_3	-	-	(a_4, a_3)

(b) Iterations (Lines 10-12)

Fig. 3. An example of SECP ($loop$=2)

4 Continuous Monitoring of ECP Pairs

In this section, we set up the problem of monitoring ECP pairs dynamically and propose a solution that uses the SECP algorithm presented in the previous section. To motivate

our problem setting, we base it on a realistic application, where the ECP join between a set of moving cars (C) and a set of static parking slots (S) is to be computed and incrementally maintained. When the car-parking assignment system starts up, it receives a number of events E_r from cars ($c \in C$) in the monitored area, corresponding to assignment requests. It then runs a static ECP join algorithm to determine the slots to be assigned to these cars.

While the system is running, it receives events from cars and pushes them into a buffer Buf. At regular time intervals (e.g., every few seconds), the events collected in Buf are handled in batch. Three types of events are collected in Buf: E_r events from cars that have just requested to park, E_p events from cars that have just parked to their assigned slot, and E_m events from cars that have just unparked and they are moving. Accordingly, we can divide the sets of cars (and slots) into four classes based on their current state, as specified in Table 3. Figure 4 shows how streaming events or system decisions define the transitions of cars and parking slots among states. We assume that at each timestamp the system receives a number of E_r, E_p, and E_m events from cars. First, all E_p events are processed, which change the statuses of the corresponding cars and slots from C_a to C_p and S_a to S_p, respectively. Then, the E_m events are processed and the corresponding cars in C_p and slots in S_p will move to classes C_m and S_f, respectively (we will explain the role of S_f shortly). Finally, the E_r events move cars from C_m state to C_r state. Unassigned cars in C_r and currently assigned cars in C_a must be processed by a *continuous* ECP algorithm based on the following.

- If an assigned car $c \in C_a$ can be assigned a better slot (due to the availability of a new free parking slot which is closer) then perform this change.
- For all cars in $c \in C_r$, find their ECP pairs after having considered the optimal re-assignments for cars in C_a.

Symbol	Description
C_m	set of cars which move and do not want to park
C_r	set of cars which move and request to park
C_a	set of cars which move and are assigned to a parking slot
C_p	set of parked cars
S_e	set of slots which are unoccupied and unassigned
S_a	set of slots which are assigned but not occupied
S_p	set of slots which are currently occupied
S_f	set of slots which are set free at the current timestamp

Table 3. Classification of objects based on their current status

Note that a re-run of the ECP join for the union of $C_r \cup C_a$ cars could result in the unfavorable assignment of a $c \in C_a$ to a slot which is further than its currently assigned slot. In order to avoid such situations[4], we must run a special version of ECP that handles cars in C_a separately.

Our continuous ECP algorithm (CECP) (see Algorithm 3) is based on the realistic assumption that only slots in S_f can change a current assignment $(c_a, c_a.\psi)$ for $c_a \in C_a$

[4] Imagine that you've been assigned to a parking and while moving towards it, the system informs you that you have to change to a further slot!

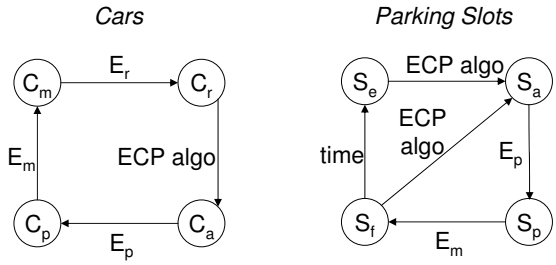


Fig. 4. State transition diagrams for objects

to a better one. The rationale is that once assigned to its slot, c_a will have moved towards it, so it is unlikely for a slot in S_e (i.e., the empty slots from the previous timestamp) will suit c_a now (since it did not suit it in the previous timestamp). Based on this assertion, we examine all slots in S_f to see if any of them could change the current assignment of a $c_a \in C_a$ to a better one. If a slot $s_f \in S_f$ can replace the current assignment $c_a.\psi$ of a car c_a, we perform this change and push $c_a.\psi$ to S_f (since it could update the assignment of another car). Otherwise, we put s_f to S_e (the set of empty slots). After all slots in S_f have been examined and the set becomes empty, we perform a static ECP join for the pair of requesting cars and empty slots (C_r, S_e). For this join, we use the SECP algorithm described in Section 3. We now discuss two optimization techniques for speeding up the search operation at Line 3 of CECP.

Algorithm 3 Continuous ECP

```
   algorithm CECP(C,S)
1: while S_f ≠ ∅ do                                          ▷ first phase
2:     s_f := remove slot s_f from S_f
3:     if for a c_a ∈ C_a d(c_a, c_a.ψ) > d(c_a, s_f) then
4:         move c_a.ψ to S_f; set c_a.ψ := s_f
5:         move s_f to S_a;
6:     else
7:         move s_f to S_e;
8: SECP(C_r, S_e)                                            ▷ second phase
```

4.1 Distance-bounded search

For each s_f, CECP scans C_a to find a car $c_a \in C_a$ for which s_f can replace $c_a.\psi$ or verify that no such car exists in C_a. This search can be accelerated if the cars in C_a are checked in increasing distance from s_f. Therefore, before CECP begins for the current timestamp, we organize the existing C_a (from the previous timestamp) in a CPM index. In addition, we compute the maximum distance Γ of any assigned pair in C_a (i.e., $\Gamma = \max\{d(c_a, c_a.\psi) | c_a \in C_a\}$). This preprocessing phase requires a only single pass over C_a, whereas the resulting index can be used for any $s_f \in S_f$.

For each s_f, we examine the objects $c_a \in C_a$ incrementally according to their distance to s_f (i.e., we perform a NN search on the CPM-index [18]). This way, the chances to find an assignment for s_f early are maximized because assigned cars close to s_f are

examined earlier. More importantly, NN search can terminate as soon as $d(s_f, c_a) \geq \Gamma$, for a neighbor c_a of s_f.

4.2 Partitioning in CPM cells

Recall that each $s_f \in S_f$ attempts to find any $c_a \in C_a$, for which $dist(c_a, s_f) < dist(c_a, c_a.\psi)$. If the distance between c_a and its assigned slot s_a ($c_a.\psi$) is smaller than the minimum distance between c_a and the boundary of the CMP cell $Cell(c_a)$ which encloses c_a (i.e., $d(c_a, c_a.\psi) \leq d_{min}(c_a, Cell(c_a))$), then c_a cannot be re-assigned to any s_f outside $Cell(c_a)$. For example, consider three assigned pairs (c_0, s_0), (c_1, s_1), (c_2, s_2), and a newly available slot s_f, as shown in Figure 5. Since $d(c_0, s_0) \leq d_{min}(c_0, Cell(c_0))$ and $s_f \notin Cell(c_0)$, we know that c_0 cannot be re-assigned to s_f.

We can extend this argument for arbitrary cars as follows. For each $c_a \in C_a$, we define $level(c_a)$ to be the minimum number of CPM levels around $Cell(c_a)$ such that c_a cannot be re-assigned to s_f, for any s_f further than these levels. This can be computed by comparing $d(c_a, c_a.\psi)$ to $d_{min}(c_a, L)$ where L is the boundary (MBR) of successive cell layers around c_a. For example, in Figure 5, $level(c_0) = 0$, $level(c_1) = 1$, and $level(c_2) = 2$.

The idea behind our second optimization is to partition the cars c_a in each cell, based on their $level(c_a)$. For example, in Figure 5, c_0 belongs to the level-0 partition of $Cell(c_0)$, c_1 belongs to the level-1 partition of $Cell(c_1)$, and c_2 belongs to the level-2 partition of $Cell(c_2)$. Then, for each s_f, when we examine a cell C during NN search, we only check all $c_a \in C$, for which $level(c_a) \geq s_f.cpmlevel$, where $s_f.cpmlevel$ is the current search level around s_f. The further C is from s_f the more partitions inside it will be pruned. For example, in Figure 5, while searching for a better assignment containing s_f, when visiting $Cell(c_0)$, we don't have to check its level-0 partition (which contains c_1). Similarly, when visiting $Cell(c_2)$, we can prune its level-0 and level-1 partitions (but not the level-2 partition which contains c_2; therefore c_2 has to be examined).

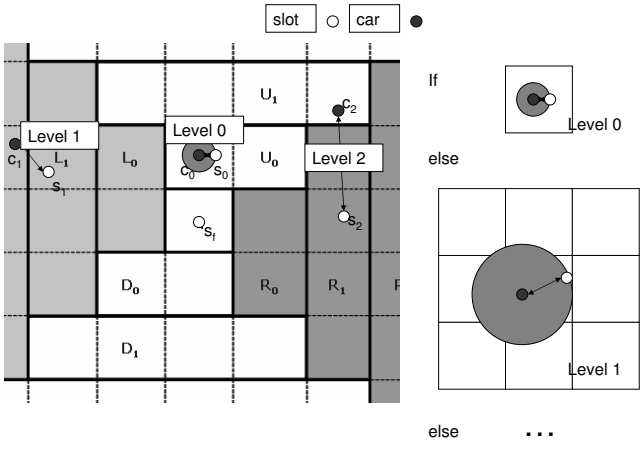


Fig. 5. Partitioning of objects to levels

5 Experimental Evaluation

In this section we experimentally evaluate the efficiency of our proposed ECP algorithms using synthetic data. First, we compare the SECP algorithm proposed in Section 3 with two alternative approaches to the same problem. Second, we validate CECP; the algorithm for continuous monitoring of ECP pairs proposed in Section 4. The algorithms were implemented in C++ and all experiments were performed on a Pentium IV 1.8GHz machine with 512MB memory, running Windows XP.

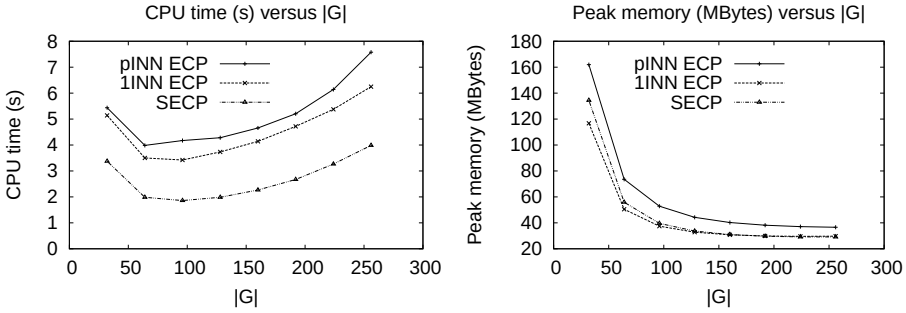
5.1 ECP Computation

To our knowledge, this is the first paper studying ECP computation, so there are no previous approaches to compare SECP with. Clearly, computing the distances between all pairs of points and running the stable-marriage algorithm (SMA) would be very inefficient. Alternatively, we compare SECP with two alternative methods, which (like SECP) avoid computing all distances:

- p**INN ECP search.** This method is similar to our SECP. The difference is that at each step it (incrementally) fills a list with the next p nearest neighbors in B for each unassigned A-point, where p is an input parameter of the algorithm. Given the pNN lists of all such A-points to their p-th neighbors, let ϵ be the smallest of these distances. We can use this distance as a bound and run Lines 10-12 of SECP to finalize ECP pairs for some of the A-points. For this purpose, we directly use the pNN lists, instead of re-computing the nearest points by running ϵ-INNECP. At each step, after all A-points have been processed, some of them will have found their ECP pair. For the remaining ones, we continue the INN search until their NN set contains exactly p neighbors. For these points we repeat the whole process at the next step.
- 1**INN ECP search.** In the initial state of the 1INN ECP algorithm, for each unassigned query a we maintain a CPM heap H for it and use it to find the nearest CPM element of a (this could be a rectangle, a cell, or an object). The nearest elements of all unassigned $a \in A$ are stored in a candidate queue (CQ) which is a priority queue organizing them in ascending order of the distance. At each step of the algorithm, we pop the top element from CQ. If this is a rectangle or a cell, we proceed to find the next nearest CPM element of the corresponding query object and push it into CQ. If the popped element from CQ is an unassigned object, it must be the ECP result of the corresponding query (by definition). If it is an assigned one, we ignore it and get the next nearest neighbor of the query object, which is pushed back to CQ. This process is continued until all ECP results are computed.

We evaluate the performance of the static ECP algorithms with synthetic datasets (to study their scalability with respect to various parameters and due to lack of real data for ECP problems). In each dataset, the coordinates of points are random values uniformly generated in a $[0, 10000] \times [0, 10000]$ space. By default, the total number of queries and objects is 100K and there are as many objects as queries (i.e., $|A| = |B|$ and $|A| + |B|$ =100K). By default the CPM grid used was 128×128 and the value of p used by the pINN ECP search algorithm is 8 (we found out by experimentation that this method performs best for $p = 8$). The k parameter of the ECP join is set to $k = \min\{|A|, |B|\}$ (i.e., we seek for the maximum possible assignment).

Figure 6 shows the performances of the three static ECP algorithms for CPM grid sizes $|G| \times |G|$ ranging from 32×32 to 256×256. Although the grid sizes with the best CPU time performance are between 64×64 and 96×96, $|G| = 128$ presents a good trade-off between the CPU time and the memory usage by all three algorithms. Furthermore, as we will show later, the 128×128 grid outperforms other sizes when larger amounts of data are searched. Note that our SECP algorithm outperforms the other two methods, while having only slightly higher memory requirements than 1INN ECP. The reason behind the good performance of SECP is that (i) unlike pINN ECP, it searches only up to the necessary nearest neighbors for each query and (ii) it avoids using and updating the huge CQ heap of 1INN ECP.



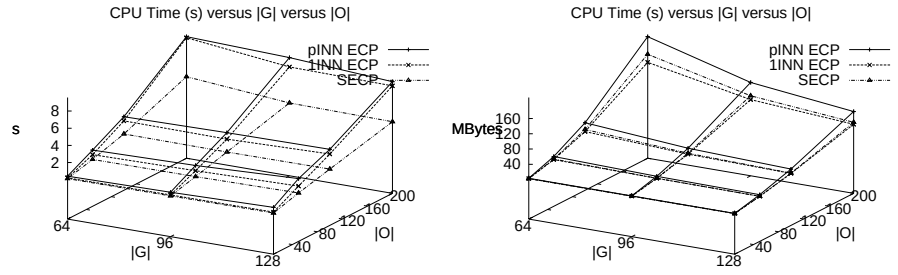
(a) CPU time (b) Peak memory

Fig. 6. Effect of $|G|$

Figure 7 compares the three algorithms for various grid sizes and database sizes $|O| = |A| + |B|$. The results are consistent with the previous experiment. SECP performs the best in terms of CPU while the costs of pINN and 1INN are more sensitive to the database size. Note that when the number of objects increases finer grids become more efficient; this is expected since the space becomes denser and using a finer partitioning pays off. Note that more memory is required for smaller grid sizes, since more individual objects (instead of cells and rectangles) enter the search heaps of the queries. Again, SECP has slightly higher memory requirements than 1INN. Finally, Figure 8 shows the performance as a function of different data size ratios ($|A|/|B|$) and database sizes ($|O| = |A| + |B|$). SECP has the best performance and its relative difference to other methods increases with the database size and $|A|/|B|$ ratio. We do not need to consider ratios larger than 1, since the ECP computation is symmetric (the smallest dataset is taken as the *query* dataset A).

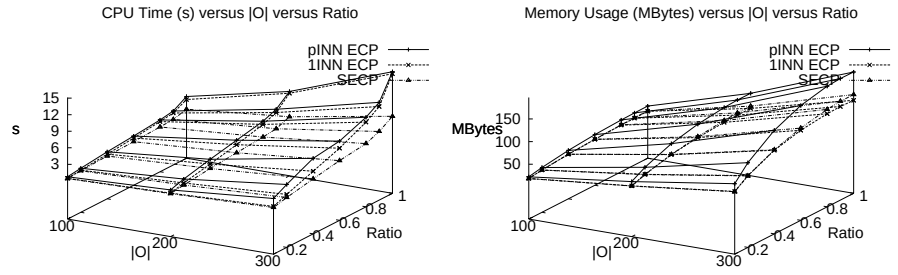
5.2 Maintenance of ECP results

We developed a data generator that simulates a real-life car-parking assignment problem and monitoring problem, based on the specifications of Section 4. The generator starts with a set of parking slots and a set of cars which are uniformly distributed in a $[0, 10000] \times [0, 10000]$ space. A parking-request probability P_{req}, an unparking probability P_{unpark}, and a velocity V are assigned to each car. Initially, all cars are moving to a random direction and they request for parking with probability P_{req} at each timestamp. If a car c issues a parking request to the system (E_r) it moves to the parking



(a) CPU time (b) Peak memory

Fig. 7. Combined effect of $|G|$ and $|O|$



(a) CPU time (b) Peak memory

Fig. 8. Combined effect of $|O|$ and $|A|/|B|$ ratio

request state and the system attempts to assign a slot to it. Once a slot s is assigned to c, c moves towards s according to its velocity and when it reaches s it parks, issuing a E_p event. After c has parked, at each subsequent timestamp it has P_{unpark} probability to issue a E_m event. A car that unparks sets its slot free and starts moving to a direction 90 degrees different than its direction when moving towards its parking slot. At each timestamp, the system processes all incoming events according to Section 4.

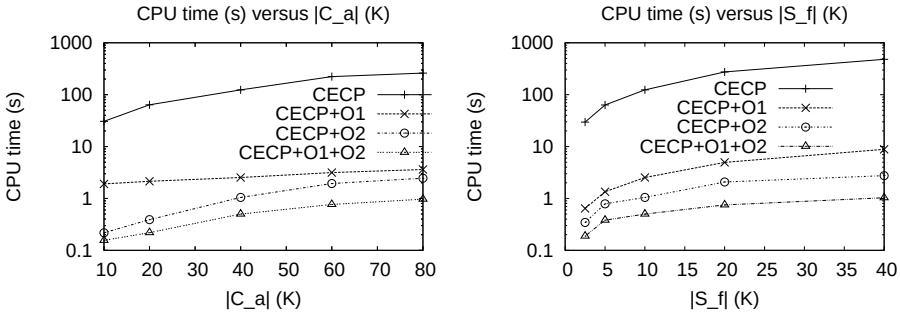
Table 4 shows the parameters of the generator, their range of values and their default value in bold font. In each experiment, only one parameter varies while the others are fixed to their default values. We measured the average CPU cost and memory requirements of the CECP algorithm for each timestamp, after letting the system to run for 1000 timestamps.

In the first experiment, we verify the effectiveness of the optimizations of Sections 4.1 and 4.2 in CECP. These optimizations aim at reducing the re-assignment cost for cars in C_a using S_f (i.e., Lines 1–7 of Algorithm 3). Figure 9 shows the re-assignment cost of CECP without, with one (CECP+O1 or CECP+O2), and with both (CECP+O1+O2) optimizations, for different time instants with different sizes of C_a and S_f. Optimization 2 (Section 4.2) incurs larger improvement compared to optimization 1 (Section 4.1) since it avoids additional accesses. The combination of both methods

Parameter	Values
Number of cars, $\|C\|$	**600K**
Number of slots, $\|S\|$	**150K**
Parking request probability, $P_{req}\%$	0.5%, 1%, **2**$\%$, 4%, 8%
Unparking probability, $P_{unpark}\%$	0.5%, 1%, **2**$\%$, 4%, 8%
Average velocity of cars, V	1.67, 3.33, **5.27**, 6.67, 13.33

Table 4. Stream generation parameters

result in the best performance for CECP at all settings. In the remaining experiments we use both optimizations in the first phase of CECP.



(a) CPU time (sec.), $|S_f| = 10$K (b) CPU time ($|C_a| = 40$K)

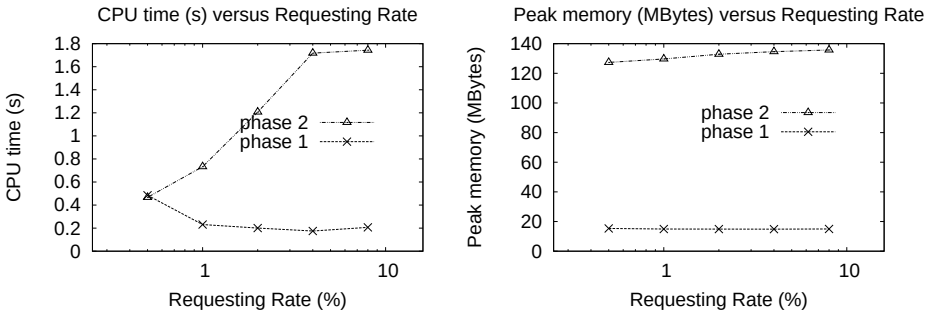
Fig. 9. Effect of $|C|$ and $|S_f|$

Figure 10a shows the average performance per timestamp of both CECP phases for different values of P_{req}. The first phase (i.e., the handling of S_f and C_a) uses both optimizations of Sections 4.1 and 4.2. The second phase (Line 8 of Algorithm 3) is performed by the SECP algorithm. For small values of P_{req} the distances between assigned cars and their slots tend to be large, a fact that increases the cost of CECP's first phase (as many re-assignments are performed). Larger P_{req} reduces the cost of the first phase due to the decrease of the average distance between assigned pairs. On the other hand, as P_{req} increases $|C_r|$ becomes larger and the second phase of CECP becomes more expensive. Figure 10a shows that the memory requirements of both phases of CECP are slightly affected by P_{req}, with the same trend as the CPU time difference.

Figure 11 shows the effect of P_{unpark} on the performance of the algorithm, after fixing P_{req} and V to their default values. There is a slight increase on the CPU time and memory requirements for both phases as P_{unpark} increases (due to the increase of $|S_f|$. Finally, Figure 12 shows that our problem is not sensitive to the objects velocity (P_{req} and P_{unpark} are fixed to their default values).

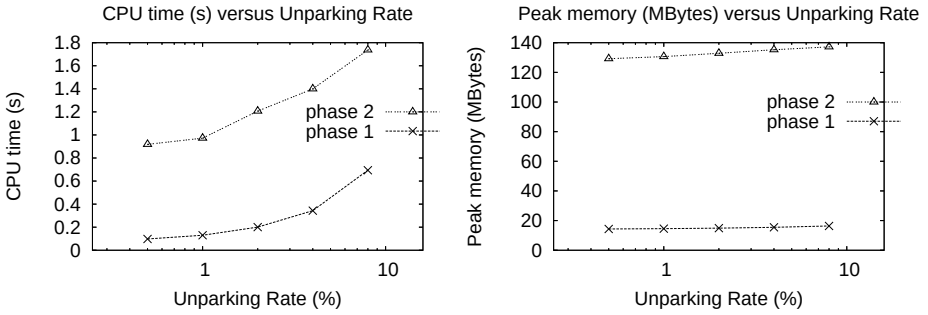
6 Conclusion

In this paper we identified the exclusive closest pairs (ECP) problem, which is a spatial assignment problem. A motivating application of it is the matching of cars and parking slots. We proposed an efficient main-memory algorithm for solving the static version of the problem. In addition, we defined the problem of continuous monitoring ECP pairs in



(a) CPU time (b) Peak memory

Fig. 10. Effect of requesting rate



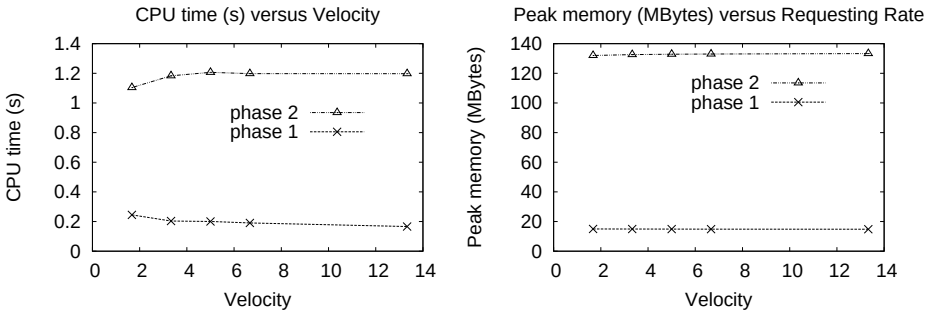
(a) CPU time (b) Peak memory

Fig. 11. Effect of leaving rate

a dynamic environment where assignment requests and de-assignment notifications arrive from a stream. We presented a thorough experimental evaluation that demonstrates the efficiency of the proposed solutions on synthetically generated data that simulate a real-life dynamic car/parking assignment problem. In the future, we will consider other types of one-to-one assignments (e.g., finding and maintaining an assignment that minimizes an aggregate distance).

References

1. M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press, Inc., 1973.
2. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD Conference*, pages 237–246, 1993.
3. J. Cardinal and D. Eppstein. Lazy algorithms for dynamic closest pair with arbitrary distance measures. In *Algorithm Engineering and Experiments Workshop (ALENEX)*, 2004.
4. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD Conference*, pages 189–200, 2000.
5. D. Eppstein. Fast hierarchical clustering and other applications of dynamic closest pairs. *ACM Journal of Experimental Algorithms*, 5:1, 2000.
6. D. Gale and L. S. Shapley. College admissions and the stability of marriage. *Amer. Math.*, 69:9–14, 1962.



(a) CPU time (b) Peak memory

Fig. 12. Effect of different velocities

7. D. Gusfield and R. W. Irving. *The Stable Marriage Problem, Structure and Algorithms*. MIT Press, 1989.
8. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD*, 1984.
9. G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD Conference*, pages 237–248, 1998.
10. G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.
11. H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD Conference*, 2005.
12. G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of k-nn and spatial join queries on continuously moving points. *ACM Trans. Database Syst.*, 31(2):485–536, 2006.
13. N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB*, pages 804–815, 2004.
14. N. Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *ICDE*, 1998.
15. M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in r-trees: A bottom-up approach. In *VLDB*, pages 608–619, 2003.
16. M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD Conference*, pages 623–634, 2004.
17. M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. In *STDBM*, pages 57–64, 2004.
18. K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD Conference*, pages 634–645, 2005.
19. E. D. Nering and A. W. Tucker. *Linear Programs & Related Problems: A Volume in the Computer Science and Scientific Computing Series*. Academic Press, Inc., 1992.
20. X. Xiong and W. G. Aref. R-trees with update memos. In *ICDE*, 2006.
21. X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.
22. C. Yang and K.-I. Lin. An index structure for improving nearest closest pairs and related join queries in spatial databases. In *IDEAS*, pages 140–149, 2002.
23. X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.
24. J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. In *SSDBM*, pages 297–306, 2004.