Computation and Monitoring of Exclusive Closest Pairs

Leong Hou U, Nikos Mamoulis, and Man Lung Yiu

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. A real application of exclusive closest pairs is the computation of (car, parking slot) assignments. This paper introduces the problem and proposes several solutions that solve it in main-memory, exploiting space partitioning. In addition, we define a dynamic version of the problem, where the objective is to continuously monitor the ECP join solution, in an environment where the joined datasets change positions and content. Finally, we study an extended form of the query, where objects in one of the two joined sets (e.g., parking slots) have a capacity constraint, allowing them to match with multiple objects from the other set (e.g., cars). We show how our techniques can be extended for this variant and compare them with a previous solution to this problem. Experimental results on a system prototype demonstrate the efficiency and applicability of the proposed algorithms.

Index Terms—H.2.4.h Query processing, H.2.4.k Spatial databases

1 Introduction

In this paper, we study an interesting type of spatial join operation, called the *exclusive closest pairs* join (ECP). ECP produces (all) 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 the ECP query is the car-parking assignment problem, where each car driver $a \in A$ requests for a parking slot from the set B of available slots. The well-known optimal matching (OM) problem [17] searches for the 1-to-1 assignment of cars to parking spaces, such that the sum/average of travel distances is minimized (i.e., optimal). However, in the real world, selfish users do not sacrifice individual convenience for the overall benefit of other 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 to p than c is. In addition, solving the exact OM problem is expensive; algorithms like successive shortest path (SSPA) [4] and the Hungarian algorithm [12] have $O(n^3)$ time complexity, where $|A| = |B| = n$. As we show in our experimental study, the ECP query produces results of similar quality to the OM solution and yet ECP can be computed at least 2 orders of magnitude faster than the exact OM algorithms. Therefore, our formulation of the ECP query searches for a practical solution to the assignment problem.

Figure 1 shows examples of previously studied join operators and the newly introduced ECP query for two cars $A = \{c_1, c_2\}$ and four parking slots $B = \{p_1, p_2, p_3, p_4\}$. Note that the distance join [11], apart from having to determine an appropriate value for ϵ, may result to assignments of the same parking slot to multiple cars and may leave a car without an assignment. The all k nearest neighbor join (AkNN) [2], [21], [26] has the same problem; multiple cars may have the same slot as their nearest neighbor. Another related query is the bichromatic reverse nearest neighbor (BRNN) query [18] (i.e., list for each car c, a parking slot p (if any) having c as its nearest car). Again, this query solves the problem only partially, since a single parking slot may have multiple reverse nearest neighbors, whereas only one of them can be assigned to it. The k inclusive closest pairs (kICP) query [3] lists the top k car-parking pairs with the smallest distance, allowing one object to participate to more than one query results. Due to this property, the query has limited applicability for the assignment problem we examine, since drivers are not certain about the availability of their nearest parking slots (as in the case of AkNN queries). On the other hand, the ECP query reserves a parking slot to one driver only. For example, after c_2 is assigned to p_1, the query does not consider p_1 for the assignment of other cars. In addition, each car is assigned to a parking space as close as possible to it, that is not nearer to some other unassigned car.

The above example demonstrates the utility of ECP joins for real assignment problems, where other spatial

- *L. H. U and N. Mamoulis are with the Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: {hleongu,nikos}@cs.hku.hk*
- *M. L. Yiu is with the Department of Computer Science, Aalborg University, DK-9220 Aalborg, Denmark. E-mail: mly@cs.aau.dk.*
- *This work was supported by grant HKU 7160/05E from Hong Kong RGC.*

Manuscript received XXXX XX, 200X; revised XXXX XX, 200X.

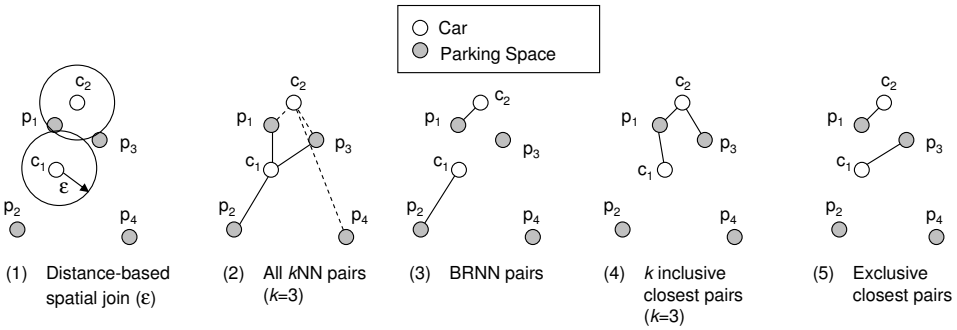


Fig. 1. Spatial join queries

join queries are inadequate. It is easy to show [19], [20] that the ECP join is a special case of the stable marriage (SM) problem [6], where preferences are derived from symmetric object distances. This equivalence ensures that we obtain the ECP join result by running a SM algorithm. Stable marriage algorithms, however, require the complete preference lists of all objects, which comes at $O(|A| \cdot |B|)$ preprocessing time and space requirements. On the other hand, we propose ECP algorithms, which compute on-demand only a small fraction of these distances, by adapting the Gale-Shapley SM algorithm [6] to exploit space-partitioning and nearest neighbor searching.

Our first algorithm, called CPMECP, utilizes the *conceptual partitioning scheme* [16] to retrieve for all objects in one dataset (e.g., A) their nearest neighbors in the other (e.g., B) incrementally. CPMECP progressively outputs ECP assignments, using distance bounds to confirm them. We then propose to evaluate ECP with a *plane scan* approach (PSECP), which sorts both datasets along the x-axis and scans them concurrently. We instantiate two algorithms from PSECP by exploiting space partitioning techniques: GRIDECP and STRIPECP, which partition the space into grid cells and horizontal stripes respectively. After analyzing the pros and cons of CPMECP and PSECP, we finally develop a hybrid ECP algorithm (HYECP), which finds the closest ECP pairs using PSECP and then switches to CPMECP computation for the remaining ones.

All the above algorithms are designed for memory-resident data. The first reason for this design choice is that main memories today are large enough to accommodate spatial data with the magnitude of the problem examined in this paper (i.e., car-parking assignment).[1] The second reason is that we consider the case where objects have high update rates. Therefore, in this paper, we also extend the ECP computation problem to an ECP monitoring problem, 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' closer to c becomes available. In this case, c must be reassigned to p' and p should become available for other cars. This change may trigger a "chaining" effect which could alter the whole assignment. We design an ECP monitoring algorithm, which is based upon the ECP join computation methods. 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.

Apart from maintaining the ECP results for dynamic data, we also study an extended form of the ECP join, called capacity constrained ECP query (CAPECP), originally defined in [20]. In this case, the objects in one of the two datasets (e.g., parking slots) have a capacity constraint (e.g., multiple cars can be accommodated by them). CAPECP, for each identified closest pair (c, p), removes c from A (like ECP) and reduces the capacity of p by 1. If the latter reaches 0, then p is removed from B. The motivating application for CAPECP is that multiple objects (e.g., cars) can be served by a facility (e.g., parking lot). In Section 3.4 we provide modifications of the proposed ECP algorithms for CAPECP.

Although so far we have only mentioned the car-parking problem as the main application of ECP, this new query and its CAPECP variant can be used to solve a wide range of (static or dynamic) assignment problems between objects and facilities, where the assignment preference is expressed by the Euclidean distance between them. Examples include cars to parking slots, tourists to landmarks, police cars to emergency incidents, mobile internet users to wireless routers, etc. As discussed above, we found that ECP produces results of good quality even if the OM measure is used.

The rest of the paper is organized as follows. Section 2 surveys related work on closest pair queries in spa-

1. Besides, our CPMECP solution can directly be applied for secondary-memory data indexed by R-trees. In this case, we just have to replace the CPM-based neighbor search module with a nearest neighbor algorithm for R-trees (e.g., [9]).

tial data, continuous monitoring problems, and spatial assignment problems. Section 3 describes our proposed algorithms for ECP computation in main memory. In Section 4 we introduce the ECP monitoring problem and present an algorithm for its efficient evaluation. 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 [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, such as the problem we study in this paper.

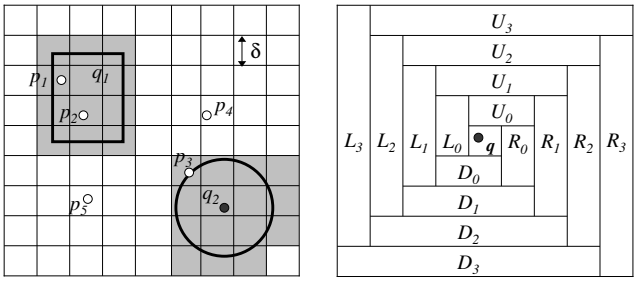
Some previous work [3], [8], [24] employ spatial indexes to solve kICP (i.e., kCP) queries in secondary memory. [3], [8] assume that the datasets are indexed by R-trees [7]. On the other hand, Yang et al. [24] extended the R-tree, by augmenting each non-leaf entry with the maximum nearest neighbor distance (with respect to the other dataset) of points in its subtree. During query evaluation, such distances are utilized for reducing the search space. [24] 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 ECP query (i.e., by remembering assigned points and avoiding their re-assignment), such an approach would require a large amount of memory (as large as the size of a dataset).

2.2 Continuous Monitoring of Spatial Queries

Several extensions of the R-tree have been developed for supporting frequent updates of spatial data. Lee et al. [13] proposed the FUR-tree (Frequent Update R-tree), which uses localized bottom-up update strategies into the traditional R-tree. Recently, Xiong et al. [22] developed the RUM-tree (R-tree with Update Memo), which was shown to have better update performance than FUR-tree. [10] 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.

Continuous monitoring of *multiple* spatial queries (e.g., range [14], [15] and kNN [23], [25], [16]) 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 2a, grid cells (of cell length δ) are employed for indexing the objects. In practice, memory grid cells [25], [16] 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).

Conceptual Partitioning Monitoring (CPM) [16] is the state-of-the-art grid-based index for monitoring NN queries and it has good performance in environments with frequent updates. Given a query point q, and assuming that the objective is to find its nearest neighbors (NN) incrementally, CPM partitions the space around $Cell(q)$ (the cell containing q), as shown in Figure 2b. The grid cells are implicitly grouped into rectangles based on their direction and distance with respect to $Cell(q)$. 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. To retrieve the NN of q incrementally, we initialize a heap (priority queue) $q.H$ to contain $Cell(q)$ and the neighboring CPM rectangles (U_0, D_0, L_0, R_0) to $Cell(q)$. The contents of $q.H$ are deheaped in ascending order of their d_{min} to q.[2] If a rectangle (resp., cell) is deheaped, the cells (resp. points) in it are added on $q.H$. A deheaped point corresponds to the next nearest neighbor of q. This way, NNs are retrieved efficiently and unnecessary accesses to distant points are avoided.



(a) Regular Grid (b) CPM

Fig. 2. Monitoring Spatial Queries

2.3 Spatial Matching Queries

Wong et al. [20], independently to us, defined an extended version of the ECP query and proposed an

2. 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, called CHAIN, to solve this problem in main memory, assuming that the input datasets A and B are indexed by a main-memory R-tree. CHAIN can solve both the ECP problem and its CAPECP extension discussed in the Introduction; objects may have a capacity, indicating the maximum number of times they participate in the assignment.

CHAIN first picks a random object c from dataset A and finds its NN c_{NN} in B (using B's index). Then, CHAIN finds the NN c' of c_{NN} in A (using A's index). If $c' \neq c$, then c_{NN} is pushed to a queue Q. Otherwise (i.e., if $c' = c$), pair (c, c_{NN}) is output as a result pair and c, c_{NN} are deleted from A, B, respectively (by applying deletion operations to the corresponding spatial indexes). The algorithm continues by dequeueing the next object x from Q (or picks a random object from A if Q is empty) and testing if x's NN in the other dataset has x as its nearest neighbor (i.e., the same test described above). Depending on the result of this test, the corresponding ECP pair is output or x's NN is pushed to Q. Eventually, CHAIN terminates after all ECP pairs have been identified this way. For the CAPECP version of the problem, CHAIN operates similarly, but when pairs are identified the capacities of the corresponding objects are decremented, until they reach zero, in which case they are deleted.

The main drawback of CHAIN is that objects in identified ECP pairs are deleted from the indexes and subsequent 1-NN searches reach the next nearest neighbors after such deletions. The overhead of continuously updating the indexes is very high, even when they are memory-based. As we will show in Section 5, our algorithms are 1-2 orders of magnitude faster than CHAIN [20]. In addition, we propose an adaptation of CHAIN that applies incremental NN search (i.e., without performing explicit object deletions), and show that even though this algorithm performs better than the original CHAIN, it is still significantly slower than our methods.

3 ECP Evaluation

As shown in the preliminary version of this paper [19] (and independently proved in [20]), the ECP query is a special case of the classic stable marriage problem and can be solved by applying an algorithm like Gale-Shapley's stable matching algorithm (SMA) [6]. Nevertheless, such a direct application of SMA requires the computation of a large number ($|A| \cdot |B|$) of distances and large space to store them, 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 apply spatial ranking techniques [9], [16] to generate the preference list of a incrementally and on-demand.

In this section, we propose efficient algorithms for evaluation of ECP queries in main memory. Section 3.1 describes a ECP computation algorithm, which integrates CPM with SMA. Since CPM was originally designed for kNN monitoring, it has high memory requirements for processing SMA. Thus, in Section 3.2 we propose an alternative approach that is reminiscent to plane sweep methods from Computational Geometry and has lower memory requirements. The two approaches are finally integrated into a hybrid solution in Section 3.3.

3.1 CPM-based Computation

The first algorithm, called CPMECP, adopts Conceptual Partitioning Monitoring (CPM) [16] (see Section 2.2) for indexing the points and supporting incremental NN search. CPMECP (Algorithm 1) uses the CPM index to compute the ECP result, by extending the aforementioned nearest neighbor search algorithm. Recall that we have two datasets A and B in our problem. In order to optimize performance, we consider the smaller dataset as a *query* set (that generate 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 we keep track of the following information: (i) its current ECP object ($o.\psi$), and (ii) the distance ($o.\lambda$) to that object. Initially, $o.\lambda$ is set to ∞, and $o.\psi$ is set to NULL.

Algorithm 1 CPMECP

```
   V, P, P' : Queue
   Result : Heap
   algorithm CPMECP(Point set A, B)
 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| < min{|A|, |B|} do      ▷ check if all results found
 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':=∅
```

In its initialization phase (Lines 1–5), CPMECP allocates an NN heap $a.H$ for each object $a \in A$, and inserts in it the points in $Cell(a)$ and all 0-level CPM rectangles that surround a. During CPMECP, $a.H$ contains cells, rectangles, and/or objects from B and can identify the one with the smallest d_{min} to a in O(1) time. 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.

CPMECP 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 $a \in P$, CPMECP 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 CPMECP 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 [16]. If the 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.

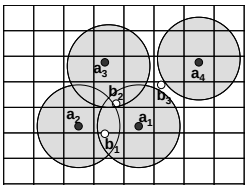
Algorithm 2 ϵ-bounded INN search and tentative ECP assignment

```
    algorithm ε-INNECP(Point a, Distance ε, Queue V, P, P′)
 1: while a.H ≠ ∅ and a.H's top entry's distance ≤ ε do
 2:     deheap ⟨o, o_dist⟩ from a.H
 3:     if o is a cell c then
 4:         for all objects b′ ∈ c do
 5:             insert ⟨b′, d(a, b′)⟩ into a.H
 6:     else if o is a rectangle DIR_lvl then
 7:         for each cell c′ in DIR_lvl do
 8:             insert ⟨c′, d_min(a, c′)⟩ into a.H
 9:         insert ⟨DIR_lvl+1, d_min(a, DIR_lvl+1)⟩ into a.H
10:     else                                          ▷ o is an object b
11:         if b.λ > o_dist then          ▷ b prefers a to its previous pair
12:             a.ψ:=b; a.λ:=o_dist;                  ▷ update ECP for a
13:             if b.ψ is NULL then
14:                 insert b into V          ▷ insert to ECP candidates V
15:             else
16:                 (b.ψ).ψ:=NULL; (b.ψ).λ:=∞          ▷ unset pair of b
17:                 insert b.ψ into P
18:             b.ψ:=a; b.λ:=o_dist                   ▷ update ECP for b
19:             break                                 ▷ break while-loop
20: if a.ψ=NULL then          ▷ a has not been assigned in this loop
21:     insert a into P′
```

After each loop of CPMECP 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 CPMECP). 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 $\min\{|A|,|B|\}$. 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 CPMECP 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 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 $a \in P$.

Although CPMECP, as described above, applies on data indexed in main memory, it can be easily adapted to apply on secondary memory data indexed by an R-tree. We only need to replace function **ϵ-INNECP** (line 12 in CPMECP) by a *bounded* NN search to the R-tree, which terminates when either the next NN is found or when the bound has been exceeded.



(a) set 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 CPMECP (*loop*=2)

3.2 Plane Scan Based Methods

Observe that in CPMECP a heap must be maintained for each $a \in A$. Its memory usage is acceptable for the scenario of [16] where a point set (say, A) is much smaller than the other (B). However, in our problem setting, two sets can have comparable sizes and thus CPMECP has high memory usage. In particular, the heaps for unassigned objects grow larger with the range of search (i.e., the number of loops), as more cells and more objects are added to them.

In this section, we propose an alternative approach (Algorithm 3) for processing the ECP join of two datasets. This PSECP algorithm first sorts A and B along an axis (e.g., the x-axis). For the sake of discussion, we use a_i (b_j) to denote the i-th (j-th) point of A (B) in the sorted order. Then, it scans the two sorted lists concurrently to determine the nearest neighbors of each $a \in A$

in B and of each $b \in B$ in A. At each loop, corresponding to a scan for both lists, we determine pairs of objects mutually having each other as nearest neighbors, remove them from their lists, and add them to the ECP result. The scalability of this method is improved, when used as a module of a divide and conquer technique, which we describe afterwards.

Algorithm 3 PSECP

```
   algorithm PSECP(Point set A, B)
 1: sort points in A along the x-axis; sort points in B along the x-axis
 2: let a_i (b_j) be the i-th (j-th) point of A (B) in the sorted order
 3: while A ≠ ∅ ∧ B ≠ ∅ do
 4:     PlaneScan(A,B)                          ▷ find candidate pairs
 5:     PlaneScan(B,A)                          ▷ refine candidate pairs
 6:     for all a ∈ A do
 7:         if a = (a.ψ).ψ then                 ▷ mutual NN pair
 8:             A:=A − a; B:=B − a.ψ
 9:             insert (a, a.ψ) into Result
```

3.2.1 The basic plane scan algorithm

Algorithm 4 is a pseudo-code of the plane scan module of Algorithm 3. The algorithm is similar to a plane sweep method which computes the intersection join of two rectangle sets, however it is unidirectional. PlaneScan(A,B) scans the x-sorted lists A and B concurrently and for every encountered point $a_i \in A$, it performs a search to find its nearest neighbor in B (Algorithm 4). This is done by two local scans from the current b_j; one backward and one forward. The search at each direction stops when the x-distance of the next examined point is greater than the distance $a_i.\lambda$ to the current nearest neighbor. The event of meeting a point b_j is handled symmetrically. Finally, we perform a scan to the assignment lists and report in the ECP join all pairs (a, b) for which $a.\psi = b$ and $b.\psi = a$. The remaining points are handled at subsequent iterations of PSECP (calls to Algorithm 4). Observe that PSECP is guaranteed to terminate after finite loops because each iteration produces at least one ECP pair (i.e., the closest pair of the current sets A and B).

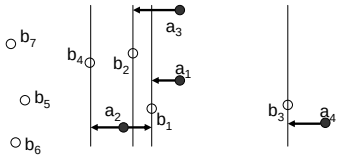
In our implementation, the call PlaneScan(B,A) at Line 7 (of Algorithm 3) is combined with the identification of the pairs (a, b) to be output (Lines 8-11) during the same scan. In addition, during the PlaneScan(B,A), we confine the search only to objects $b_j \in B$, for which $(b_j.\psi).\psi = b_j$ has been set by the PlaneScan(A,B) run. Only these objects can be returned as ECP results at the current loop (if their NN does not change during the PlaneScan(B,A)). As a result, the scanning/searching cost for dataset B is reduced.

To exemplify the PSECP functionality, consider the set of points in Figure 4a. During the first loop, after the PlaneScan(A,B) call, we get $a_2.\psi = b_1$, $a_1.\psi = b_1$, $a_3.\psi = b_2$, and $a_4.\psi = b_3$. In addition, we have $b_1.\psi = a_2$, $b_2.\psi = a_3$, and $b_3.\psi = a_4$. Therefore only the subset $\{b_1, b_2, b_3\}$ of B must be examined during the PlaneScan(B,A) call, to verify whether their current NN assignments are correct. From these objects, only

Algorithm 4 Plane Scan Algorithm

```
   algorithm PlaneScan(Point set A, B)
 1: j:=1                                       ▷ current position of point in B
 2: for all i from 1 to |A| do                 ▷ current position of point in A
 3:     while a_i.x ≥ b_j.x and j < |B| do
 4:         j:=j + 1                           ▷ skip the current point in B
 5:     h:=j − 1
 6:     while h ≥ 1 and a_i.λ ≥ dist_x(a_i, b_h) do
 7:         if a_i.λ > d(a_i, b_h) then
 8:             a_i.ψ:=b_h; a_i.λ:=d(a_i, b_h)
 9:         if b_h.λ > d(a_i, b_h) then
10:             b_h.ψ:=a_i; b_h.λ:=d(a_i, b_h)
11:         h:=h − 1
12:     h:=j
13:     while h ≤ |B| and a_i.λ ≥ dist_x(a_i, b_h) do
14:         execute Lines 7-10
15:         h:=h + 1
```

b_1 and b_3 have their current pairs as their actual NN (the actual NN of b_2 is a_1 and not a_3). Thus, only pairs $(a_2, b_1), (a_4, b_3)$ are output after the first loop of PSECP. Figure 4b shows the candidate assignments (after Line 6 of PSECP) and the results output after each loop.



(a) Points and scanlines

Loop	Candidates	Results
1	$a_2 \leftrightarrow b_1$ $a_3 \leftrightarrow b_2$ $a_4 \leftrightarrow b_3$	$a_2 \leftrightarrow b_1$ $a_4 \leftrightarrow b_3$
2	$a_1 \leftrightarrow b_2$	$a_1 \leftrightarrow b_2$
3	$a_3 \leftrightarrow b_4$	$a_3 \leftrightarrow b_4$

(b) Running steps

Fig. 4. An example of plane scan

3.2.2 Grid-based plane scan

Although PSECP saves redundant accesses significantly, its plane scan along x-sorted lists may lead to unnecessary temporary assignments (that do not translate to actual results). For instance, in Figure 4a, the plane scan for the point a_2 visits the points b_4, b_2, and b_1 (in the order). Even though b_1 is the closest to a_2, we need to examine other points (e.g., b_2) closer to a_2 along the x-axis.

In order to reduce unnecessary assignments, we extend PSECP to a divide-and-conquer solution, called GRIDECP. First, all objects are indexed by a (hierarchical) grid G with 4^l cells, where l denotes the number of grid levels. Second, for each cell c_m, objects in the cell are sorted and the PSECP method is applied on them to obtain local candidate ECP pairs.

To verify whether a local candidate pair $(a_i, a_i.\psi)$ (found in the cell c_m) is an actual ECP result, we need to consider their *minimum possible distances* to other points outside c_m. In the following, we use $d_{min}(a_i, (G - c_m).B)$ to denote a lower-bound value of the minimum distance between a_i and points of B outside c_m. The derivation of this bound will be elaborated at the end of the section. When $d(a_i, a_i.\psi) < d_{min}(a_i, (G - c_m).B)$, a_i must be closer to $a_i.\psi$ than to other points of B. Similarly, we compute $d_{min}(a_i.\psi, (G - c_m).A)$ and determine whether

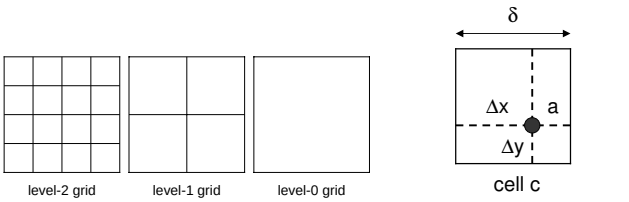
$a_i.\psi$ is closer to a_i than to other points of A. The pair $(a_i, a_i.\psi)$ is immediately reported as a result pair when both of these conditions are satisfied. In case no further result pairs can be obtained within individual cells, we merge four adjacent cells into one (see Figure 5a) and create new grid with 4^{l-1} cells. The GRIDECP algorithm continues until $\min\{|A|, |B|\}$ pairs are generated.

It remains to discuss the derivation of the bound $d_{min}(a, (G-c).B)$, for a point $a \in A$ in the cell c. Figure 5b illustrates a simple method for deriving the bound. Let δ be the side length of a grid cell. Suppose that the lower x and y coordinates of the cell c are $c.x_l$ and $c.y_l$ respectively. Clearly, the minimum distance from a to any point outside c is indicated by one of the dotted lines. By letting $\Delta x = a.x - c.x_l$ and $\Delta y = a.y - c.y_l$, the bound can be defined as:

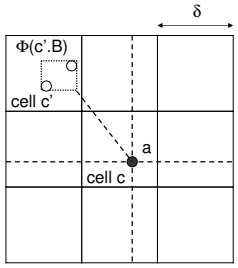
$$d_{min}^{basic}(a, (G-c).B) = \min\{\Delta x, \delta - \Delta x, \Delta y, \delta - \Delta y\}$$

Although the above derivation is simple, the bound may be loose, affecting the effectiveness of discovering ECP results. We now show how to derive a tighter bound, as depicted in Figure 5c. For each cell c', we maintain the *minimum bounding rectangle* $MBR(c'.B)$ of all points of B in the cell c'. Observe that points of B outside the cell c falls in either region: (i) the eight neighbor cells surrounding c, and (ii) the region beyond those neighbor cells. Regarding (i), the minimum distance bound with respect to a neighbor cell c' is $d_{min}(a, MBR(c'.B))$. For (ii), the minimum distance is taken as $\delta + d_{min}^{basic}(a, (G-c).B)$. Combining them, we obtain the following bound for $d_{min}(a, (G-c).B)$:

$$d_{min}^{tight}(a, (G-c).B) = \min\{\delta + d_{min}^{basic}(a, (G-c).B), \min_{\text{neighbor cell } c' \text{ of } c, |c'.B|>0} d_{min}(a, MBR(c'.B))\}$$



(a) Merging grid cells (b) Deriving d_{min}^{basic}



(c) Deriving d_{min}^{tight}

Fig. 5. Grid cells and derivation of the lower bound distance $d_{min}(a, (G-c).B)$

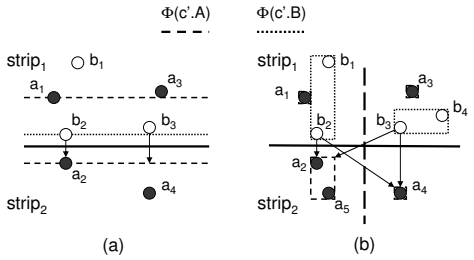


Fig. 6. An example of strip-based plane scan

3.2.3 Strip-based plane scan

GRIDECP partitions the space using a grid but it has to manage a large number of cells and applies numerous plane scans for computing ECP results. To alleviate this, we propose to modify the algorithm into the STRIPECP algorithm, which divides the space into horizontal stripes.

In STRIPECP, plane scan is applied on each stripe (equivalent to a row of grid cells), like the GRIDECP algorithm. For each discovered pair $(a_i, a_i.\psi)$ within a stripe, we need to compare its distance $d(a_i, a_i.\psi)$ against the lower-bound distance d_{min} of a_i and $a_i.\psi$ to the borders of neighbor stripes. However, if the border for all points in a strip is used, then the above bound could be too loose and ineffective for pruning. Thus, we also apply the borders at the grid-cell level.

As an example, consider the objects shown in Figure 6a. In $strip_1$, we discover the assignment $a_3 \leftrightarrow b_3$ and test whether it is a valid ECP pair. If we simply consider the distance of b_3 to the border $strip_2$ (see Figure 6a), then such a (lower-bound) distance is less than $d(a_3, b_3)$ and we cannot immediately conclude (a_3, b_3) to be an actual ECP result. On the other hand, if we use the borders at the grid-level (see Figure 6b), we can immediately output the pair (a_3, b_3). Note that only the grid-level borders of neighboring cells outside the stripe are used, since the results are already confirmed to be valid within the stripe by the plane scan algorithm. As confirmed by our experiments, STRIPECP achieves better scalability than GRIDECP.

3.3 A Hybrid Solution

In previous sections, we proposed two alternative approaches for ECP evaluation in main memory; a CPM-based solution (CPMECP) and two plane-scan based methods (STRIPECP being the best one). We observed by experimentation that each solution has its own advantages and drawbacks. Although CPMECP is effective in finding the ECP pairs without redundant calculations, it maintains a huge number of heaps (one for each object) and suffers from large memory usage. On the other hand, STRIPECP has low memory requirements but performs many redundant computations for candidate pairs in many loops. As discussed in Section 3.2, objects of false-hit pairs at each loop need to be examined in

the next one. If there is no large reduction in the set of remaining objects, then the number of loops becomes high, degrading the efficiency of the algorithm.

To demonstrate this problem, we execute STRIPECP on sample datasets with sizes $|A| = |B| = 100$K. Figure 7 shows the fraction of total output results as a function of time. Observe that the number of objects drops fast initially, but the reduction rate decreases in the later loops. It takes 4.5 seconds to compute all results but uses 2.6 seconds for deriving the last 15% ECP results. After the first few loops, the algorithm produces few ECP assignments and requires many iterations for processing few remaining objects.

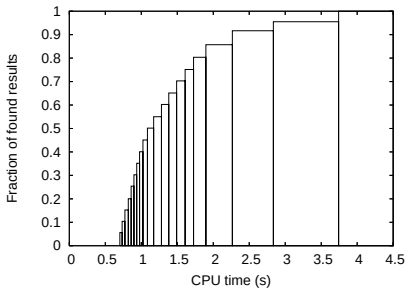


Fig. 7. Output progress of STRIPECP on test datasets

To alleviate this problem, we consider switching STRIPECP to another method when the number of remaining objects becomes too small. As discussed below, CPMECP works well for a small number of objects, for which the heap bookkeeping is cheap. Thus, we propose a *hybrid* solution, called HYECP, which first applies STRIPECP on the input datasets and then switches to CPMECP when the number of remaining ECP pairs (yet to be identified) drops below $\omega \cdot \min\{|A|, |B|\}$, where ω is a user-defined parameter between 0 and 1.

3.4 The CAPECP Query

In this section, we present a generalized version of the ECP query and discuss how the proposed algorithms can be adapted to process this query. CAPECP is similar to ECP, except that each object may have a *capacity*, which is a positive integer indicating how many objects from the other dataset can match with it. For example, if the capacity $b.\sigma$ of the parking slot $b \in B$ is 10, then b can accommodate at most 10 cars from A. When the next closest pair (a, b) in $A \times B$ is identified, the capacities of objects a and b are decreased by one. If the capacity of an object (e.g., $b.\sigma$) reaches 0 then the object is removed from the corresponding dataset (e.g., B). CAPECP iteratively finds and reports the next closest pair, performing capacity adjustments (and deletions if applicable), until all objects from one of the two datasets have been deleted.

It is easy to see that CAPECP can be simulated and solved as an ECP query, by replacing each object p with $p.\sigma$ identical points. However, this approach boosts the size of the datasets and degrades performance. We now discuss the (minor) modifications necessary to our proposed ECP algorithms for handling CAPECP queries. We make the following changes to lines 13 to 18 of CPMECP (Algorithm 2). For each point, e.g., $a \in A$, the candidate λ is replaced by a heap h_λ which stores up to $a.\sigma$ elements. After this modification, we keep at most $a.\sigma$ NN candidates for each a. If any element in h_λ passes the verification test in Algorithm 1, Lines 14 to 15, then this pair must be in the CAPECP result. We refer to this modified CPMECP algorithm by CPMCAPECP. In PSECP, we remove points from A and B when they are in confirmed ECP pairs. For CAPECP queries, the only modification to PSECP is to replace the removal of points (Line 8 of Algorithm 3), by reduction of their σ values wherever applicable.

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 static ECP algorithms 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.

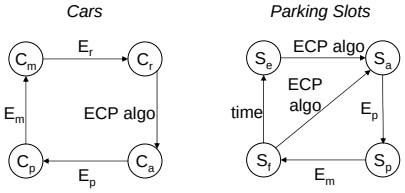
While the system is running, it receives events from cars and pushes them into a buffer Buf. At regular time intervals (e.g., every few seconds), events collected in Buf are handled in batch. Three types of events are collected in Buf: E_r events from cars that have just requested to park, E_p events 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 divide the sets of cars (and slots) into four classes based on their current state, as specified in Figure 8a. Figure 8b shows how streaming events or system decisions define the transitions of cars and parking slots among states. Suppose 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 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 these rules.

- 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	cars which move and request to park
C_a	cars which move and are assigned to a parking slot
C_p	parked cars
S_e	(set of) slots which are unoccupied and unassigned
S_a	slots which are assigned but not occupied
S_p	slots which are currently occupied
S_f	slots which are set free at the current timestamp

(a) Possible states



(b) State transition diagrams

Fig. 8. States of cars/slots and their transitions

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[3], we must run a special version of ECP that handles cars in C_a separately.

Our continuous ECP algorithm (CECP) (see Algorithm 5) is based on the realistic assumption that only slots in S_f can change a current assignment $(c_a, c_a.\psi)$ for $c_a \in C_a$ 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 one of the static ECP algorithms described in Section 3 (i.e., CPMECP, GRIDECP, STRIPECP, HYECP). We now discuss two optimization techniques for speeding up the search operation at Line 3 of CECP.

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 $s_f \in S_f$.

3. Imagine that you have been assigned to a parking and while moving towards it, the system informs you that you have to change to a further slot!

Algorithm 5 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 ∀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: execute a static ECP algorithm on (C_r, S_e)       ▷ second phase
```

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 [16]). 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 border of the CPM 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 9. 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 border (MBR) of successive cell layers around c_a. For example, in Figure 9, $level(c_0) = 0$, $level(c_1) = 1$, 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 9, 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 9, while searching for a better assignment containing s_f,

when visiting $Cell(c_0)$, we do not 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).

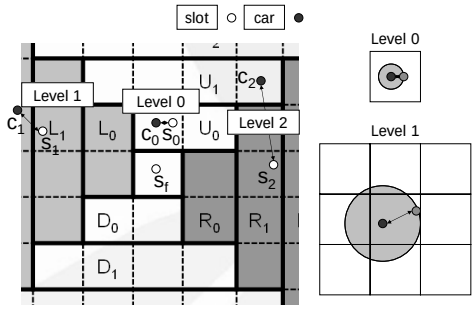


Fig. 9. Partitioning of objects to levels

5 Experimental Evaluation

This section experimentally evaluates the efficiency of our proposed ECP algorithms using synthetic data. The algorithms were implemented in C++ and all experiments were performed on a Pentium IV 1.8GHz machine with 512MB memory, running Windows XP. The comparison figures that we show already include the cost of creating and maintaining the necessary data structures for the algorithms (e.g., the CPM grid for ECPCPM).

Section 5.1 compares the static ECP algorithms (namely, CPMECP, GRIDECP, STRIPECP, HYECP) proposed in Section 3. Note that CPMECP has already been compared with two alternative ECP solutions based on the CPM grid in our preliminary work [19]. In the comparison, we also included the CHAIN algorithm of [20] and an optimized version of this method, called CPMCHAIN, which operates on a CPM-grid (indexing the objects) instead of a main-memory R-tree. When the capacity of an object reaches zero, the object is marked as "deleted", rather than directly removing it from the index. In addition, CPMCHAIN performs NN searches incrementally, by maintaining a search heap for each object in order to continue from the previous NN every time the next NN has to be explored (i.e., the current NN is matched to another object).

Section 5.2 evaluates the quality of the ECP result compared to the optimal matching. Section 5.3 evaluates the CECP algorithm (for continuous monitoring), proposed in Section 4).

5.1 Efficiency of ECP Computation

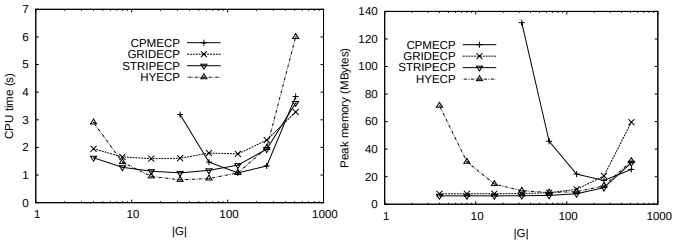
We evaluate the performance of the proposed ECP algorithms with synthetic datasets. In each dataset, the coordinates of points are random values uniformly generated in the 2D space $[0, 10000] \times [0, 10000]$. Besides uniform points, we also generated datasets following Gaussian and Zipfian distributions. In the Gaussian distribution the coordinates of the points are generated to have as mean the center of the space and the deviation is set to 2500. For the Zipfian distribution, the skew parameter is fixed to 0.8 and the x- and y- coordinates are skewed towards 0. We study the performance of our ECP algorithms with respect to various parameters, which are displayed in Table 1 (their default values are shown in bold). In each experiment, only one parameter varies while the others are fixed to their default values. The parameter $|G|$ denotes the number of cells per axis. CPMECP and GRIDECP operate on a grid with $|G|^2$ cells, STRIPECP partitions the space into $|G|$ strips, and HYECP operates initially using $|G|$ strips and then using a $|G| \times |G|$ grid. Since different ECP algorithms achieve their best performance at different values of $|G|$, their default values will be determined in the next experiment.

TABLE 1
Range of parameter values

Parameter	Values
Number of cells per axis, $\|G\|$	4, 8, 16, 32, 64, 128, 256, 512
Total data size, $\|O\|$ ($\times$1000)	15, 30, **60**, 120, 240
Cardinality ratio, $\|A\|/\|B\|$	0.1, 0.3, 0.5, 0.7, 0.9, **1.0**
Data Distribution	**Uniform**, Gaussian, Zipfian
Switching Ratio, ω (for HYECP)	0.7, 0.8, **0.9**

Figure 10 illustrates the execution time and memory usage as a function of $|G|$. Note that CPMECP achieves a good tradeoff between CPU time and memory usage, at $|G| = 128$. STRIPECP incurs low CPU time at $|G| = 16$. Since STRIPECP maintains only a few extra information for reflecting changes in the grid, its memory usage is not sensitive to $|G|$. GRIDECP shows similar results as STRIPECP. HYECP has a good balance between CPU time and memory usage, at $|G| = 32$. In the subsequent experiments, we set the following default values for $|G|$: 128 for CPMECP, 16 for GRIDECP and STRIPECP, and 32 for HYECP. The 'U'-shape of the curves can be explained as follows. For CPMECP, very high (very low) values of $|G|$ increase the search cost and the memory usage since too many empty cells (unused objects) are inserted into the heaps for performing the incremental NN search. For PS-based approaches (GRIDECP, STRIPECP, HYECP), when $|G|$ is too low (i.e., each cell has large area), we encounter the scenario of Figure 4 and each plane scan may sweep over a large region. In case $|G|$ is too high, each cell has small area and many points are located close to cell boundaries. As a result, ECP candidates of those points cannot be immediately reported and they need to be further processed after the cells are merged.

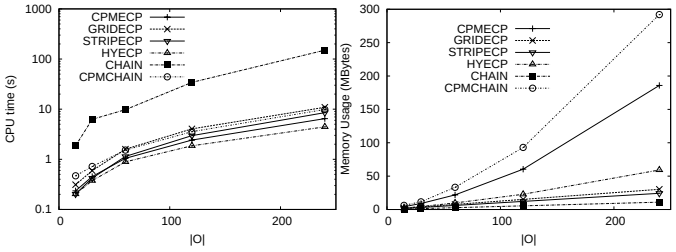
We proceed to study the scalability of our ECP algorithms and compare them with CHAIN. Figure 11a shows the effect of the data sizes $|O|$ on the running time. The cost of algorithms increases with $|O|$ and their performance gap widens at high values of $|O|$. CHAIN is one to two orders of magnitude slower than the other methods (including CPMCHAIN). There are



(a) CPU time (b) Peak memory

Fig. 10. Effect of $|G|$

two reasons for the bad performance of CHAIN. First, each NN query of CHAIN is run from scratch, while incremental NN search is performed by the other approaches. Second, CHAIN performs expensive deletions to the main-memory R-trees every time an ECP pair is detected. CPMCHAIN is faster than CHAIN, but still significantly slower than HYECP (our best technique). Regarding memory usage, observe in Figure 11b that, CPMECP has high storage demands when $|O|$ is large. This drawback restricts the application of CPMECP in real scenarios with limited available memory. CHAIN has the lowest memory usage overhead, since it does not maintain any heaps for incremental NN searches. CPMCHAIN has the highest memory usage than all methods because it maintains NN heaps for all objects (like CPMECP), however, search is performed in a depth-first fashion (using Q). The NN queries performed by this method might search more space than CPMECP (which operates in a breadth-first fashion, for all points at the same time). In summary, HYECP is the best method, since it has the lowest computational cost, while not demanding excessive memory.

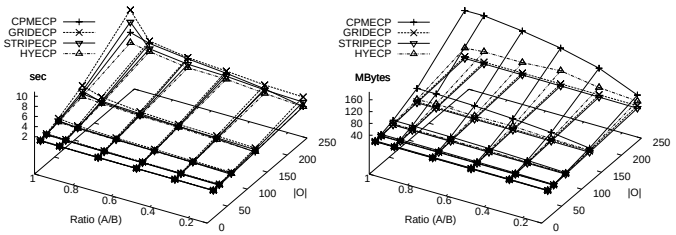


(a) CPU time (b) Peak memory

Fig. 11. Effect of data size $|O|$

In the next experiment (Table 2), we compare the algorithms for input datasets of varying data distribution. Again, HYECP has the best performance for all pairs of inputs. Observe that HYECP has very low cost for pairs with the same data distribution. Cases with a Gaussian dataset take more time to compute because most of the (Gaussian) points are located at the center and the distance of a ECP pair can be at most half the diagonal length of the spatial domain. Cases with a Zipfian dataset are even more expensive because most of the (Zipfian) points are located near the origin point (0,0) and the distance of a ECP pair can reach the diagonal length of the spatial domain.

Next, we investigate the combined effect of the cardinality ratio $|A|/|B|$ and data size $|O|$ on the performance of the ECP algorithms. Since ECP is a symmetric problem (see Section 3), it suffices to consider only cardinality ratio in the interval $(0, 1]$. We did not include CHAIN and CPMCHAIN in the experiments, since CHAIN is 1-2 orders of magnitude slower than the other methods and CPMCHAIN is always outperformed by CPMECP computationally and in terms of memory requirements. As shown in Figure 12a, HYECP outperforms its competitors for most of the cases. For the case of very low cardinality ratio (e.g., $|A|/|B| = 0.1$), HYECP is slightly slower than CPMECP; in these cases, switching between CPMECP and PSECP does not pay off. Regarding the most expensive case (highest values of $|O|$ and $|A|/|B|$), HYECP saves 30% CPU time and 70% peak memory usage over CPMECP. Moreover, CPMECP is more sensitive to the value of $|O|$ than the others. Observe that STRIPECP achieves its best performance when the ratio $|A|/|B|$ is between 0.6 to 0.8. In case the ratio is too high or too low, STRIPECP produces a lot of false hits. Figure 12b shows that HYECP has a good tradeoff between CPU time and memory usage. Also, the memory usage of CPMECP is more sensitive to the parameter $|O|$ than the others.



(a) CPU time (b) Peak memory

Fig. 12. Effect of cardinality ratio $|A|/|B|$ and data size $|O|$

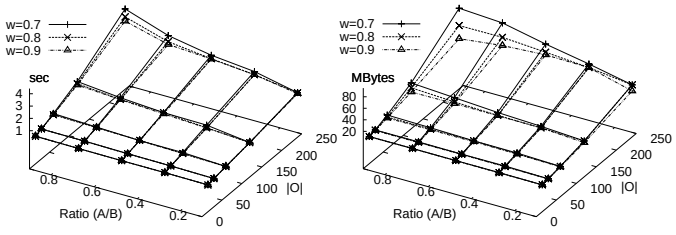
Recall that HYECP switches from plane scan searching to CPM searching when the number of remaining ECP pairs (yet to be identified) drops below $\omega \cdot \min\{|A|, |B|\}$, where ω is a parameter value between 0 and 1. Figure 13 shows the performance of HYECP for different values of ω. HYECP achieves the best performance (in terms of CPU time and memory usage) at $\omega = 0.9$.

Next, we investigate the performance of the algorithms for CAPECP queries on the default data, where all objects in dataset B have equal capacity σ (varying capacity does not affect the relative performance of the algorithms). We included CHAIN and CPMCHAIN in the comparison. As Figure 14 shows, σ does not affect the cost and memory requirements of the methods.

Figure 15 shows the performance of algorithms with respect to the data ratio $|A|/|B|$, by fixing the capacity σ to 16. STRIPECP incurs higher cost at lower data cardinality ratio, affecting the performance of the first

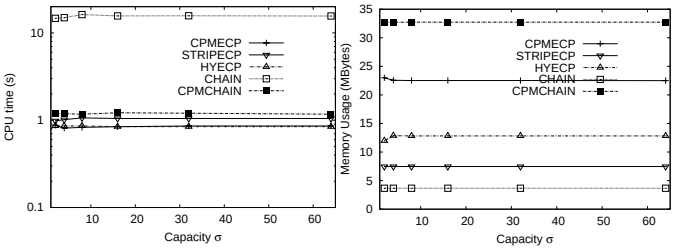
TABLE 2
Effect of different data distributions (memory: MBytes; time: seconds)

Distribution		CPMECP		STRIPECP		HYECP		CHAIN		CPMCHAIN	
A	B	memory	time	memory	time	memory	time	memory	time	memory	time
Uni	Uni	15.87	1.16	7.53	1.22	9.91	1.06	10.07	9.781	22.21	1.75
Gau	Gau	17.24	1.19	7.53	1.22	10.11	1.08	10.07	24.72	24.21	1.80
Zipf	Zipf	25.31	1.48	7.53	2.53	9.93	1.09	10.07	43.02	37.31	3.56
Uni	Gau	64.63	8.17	7.53	47.53	34.18	4.00	10.07	130.11	122.66	16.97
Gau	Uni	86.44	13.08	7.53	44.39	44.69	5.58	10.07	159.41	121.30	16.51
Gau	Zipf	161.18	31.09	7.53	224.16	148.86	13.99	10.07	31.55	465.50	54.70
Zipf	Gau	332.72	50.64	7.53	222.99	164.69	24.27	10.07	72.88	469.94	64.61
Uni	Zipf	110.10	27.05	7.53	288.34	97.69	10.98	10.07	32.77	458.77	54.81
Zipf	Uni	344.03	53.69	7.53	272.02	151.07	22.25	10.07	34.25	451.95	57.64



(a) CPU time (b) Peak memory

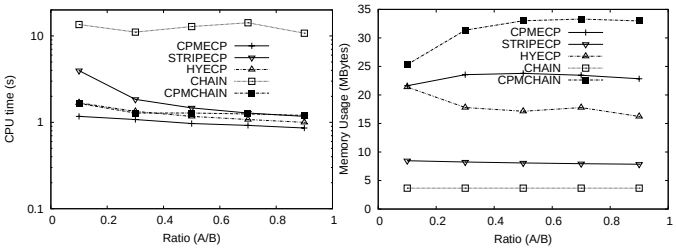
Fig. 13. Effect of ω in HYECP



(a) CPU time (b) Peak memory

Fig. 14. Effect of capacity σ

phase in HYECP. At these settings, CPMECP is faster than HYECP. Nevertheless, STRIPECP and HYECP have lower memory consumption than CPMECP.

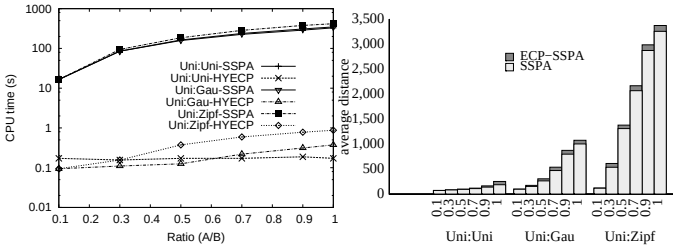


(a) CPU time (b) Peak memory

Fig. 15. Effect of ratio $|A|/|B|$ with capacity $\sigma = 16$

5.2 Comparison of ECP result with the optimal matching

In the next experiment, we investigate how close the ECP result is to the optimal matching. In specific, we implemented the successive shortest path algorithm (SSPA) [4], having as objective to find the 1-1 assignment between A and B with the minimum average distance between the assigned pairs. Figure 16 compares SSPA with HYECP in terms of computational cost and quality of resulting matching on three pairs of datasets taken from Table 2. The results are similar for other pairs. HYECP is at least two orders of magnitude faster than SSPA in all cases. The costs of both methods increase with the cardinality ratio, as the size of the matching (and the effort to find it) increases. The optimal average matching distance (generated by SSPA) and the average matching distance difference between ECP and SSPA are shown in Figure 16b, for various data distributions and cardinality ratio. Observe that the quality of matching obtained by ECP stays close to the optimal.



(a) CPU time (b) Quality

Fig. 16. Comparison with the optimal assignment

5.3 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 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

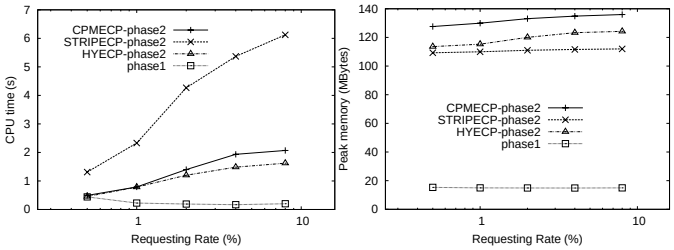
its velocity and when it reaches s it parks, issuing a E_p event. After c has parked, at each subsequent timestamp it has P_{unpark} probability to issue a E_m event. A car that unparks sets its slot free and starts moving to a direction 90 degrees different than its direction when moving towards its parking slot. At each timestamp, the system processes all incoming events according to Section 4. Table 3 shows the parameters of the generator, their range of values and their default value in bold font.

TABLE 3
Stream generation parameters

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

The effectiveness of the optimizations of Sections 4.1 and 4.2 in the first phase of CECP was verified in the preliminary version of our paper [19]. There, we showed that these optimizations reduce the cost of the basic version of CECP by two orders of magnitude. Here, we investigate the overall performance of CECP if the two optimizations are used in the first phase and for different versions of the offline ECP method used in the second phase (i.e., CPMECP, STRIPECP, or HYECP). We skip the comparison of GRIDECP since it has worse performance than STRIPECP in all previous experiments. Figure 17a shows the average performance per timestamp of both CECP phases for different values of P_{req}. 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 for all algorithms. Figure 17b 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. CPMECP has higher memory overhead than the others, which is similar to our static experiments. HYECP has equal or lower CPU cost than CPMECP.

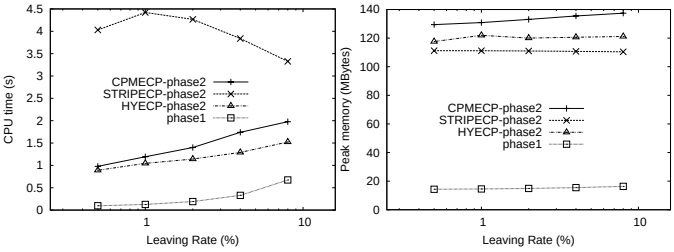
Figure 18a shows the average performance per timestamp of both CECP phases for different values of P_{req}. The first phase (i.e., the handling of S_f and C_a) uses both optimizations of Sections 4.1 and 4.2. For small values of P_{req} the distances between assigned cars and their slots tend to be large, 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. In the second phase, HYECP is the best algorithm in terms of CPU and has lower memory overhead than CPMECP. Figure 18b 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.



(a) CPU time (b) Peak memory

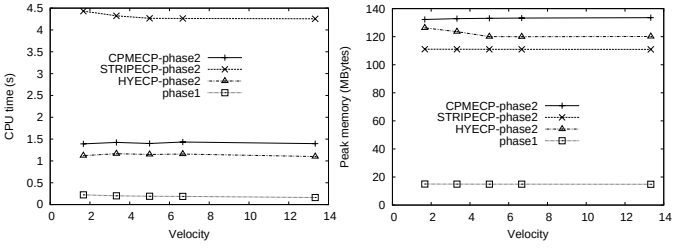
Fig. 17. Effect of requesting rate

Figure 18 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 19 shows that our problem is not sensitive to the objects velocity (P_{req} and P_{unpark} are fixed to their default values).



(a) CPU time (b) Peak memory

Fig. 18. Effect of leaving rate



(a) CPU time (b) Peak memory

Fig. 19. Effect of different velocities

6 Conclusion

In this paper we identified the exclusive closest pairs (ECP) problem, which is a spatial assignment problem. A motivating application is the matching of cars and

parking slots. We proposed main-memory algorithms for solving the static version of the problem. Apart from the CPM-based algorithm, which was included in an earlier version of the paper [19], we designed a plane-sweep based approach with low memory requirements. This method was combined with the CPM-based algorithm to result in a highly optimized hybrid method. We compared our proposal with an ECP algorithm independently proposed in [20] and showed that it is 1-2 orders of magnitude faster. In addition, we defined the problem of continuous monitoring ECP pairs in a dynamic environment where assignment requests and de-assignment notifications arrive from a stream. Finally, we showed that our approaches can be seamlessly applied for a generalization of the ECP problem, where objects may have capacities. Via a thorough experimental evaluation we demonstrated 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] C. Böhm and F. Krebs. The *k*-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.

[3] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *SIGMOD*, 2000.

[4] U. Derigs. A shortest augmenting path method for solving minima perfect matching problems. *Networks*, 11(4):379–390, 1981.

[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.

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

[8] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD*, 1998.

[9] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[10] 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.

[11] N. Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *ICDE*, 1998.

[12] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics*, 2005.

[13] 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*, 2003.

[14] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, 2004.

[15] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in place. In *STDBM*, 2004.

[16] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.

[17] 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.

[18] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, 2001.

[19] L. H. U, N. Mamoulis, and M. L. Yiu. Continuous monitoring of exclusive closest pairs. In *SSTD*, 2007.

[20] R. C.-W. Wong, Y. Tao, A. W.-C. Fu, and X. Xiao. On efficient spatial matching. In *VLDB*, 2007.

[21] C. Xia, H. Lu, B. C. Ooi, and J. Hu. Gorder: An efficient method for knn join processing. In *VLDB*, 2004.

[22] X. Xiong and W. G. Aref. R-trees with update memos. In *ICDE*, 2006.

[23] 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*, 2005.

[24] C. Yang and K.-I. Lin. An index structure for improving nearest closest pairs and related join queries in spatial databases. In *IDEAS*, 2002.

[25] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.

[26] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. In *SSDBM*, 2004.

Leong Hou U Leong Hou U received Bachelor Degree in Computer Science and Information Engineering in 2003 from the National Chi Nan University, Taiwan, and received the Master Degree in E-Commerce in 2005 from the University of Macau, Macau. He is currently a PhD candidate at the Department of Computer Science, University of Hong Kong, under the supervision of Dr. N. Mamoulis. His research focuses on evaluating different kind of queries on complex types of data.

Nikos Mamoulis Nikos Mamoulis received a diploma in Computer Engineering and Informatics in 1995 from the University of Patras, Greece, and a PhD in Computer Science in 2000 from the Hong Kong University of Science and Technology. He is currently an associate professor at the Department of Computer Science, University of Hong Kong, which he joined in 2001. In the past, he has worked as a research and development engineer at the Computer Technology Institute, Patras, Greece and as a post-doctoral researcher at the Centrum voor Wiskunde en Informatica (CWI), the Netherlands. His research focuses on management and mining of complex data types. He has served on the program committees of over 40 international conferences and workshops on data management and data mining. He was the general chair of SSDBM 2008 and a co-organizer of SSTDM 2006. He is an editorial board member for Geoinformatica Journal and a field editor of the Encyclopedia of Geographic Information Systems.

Man Lung Yiu Man Lung Yiu received the bachelor's degree in computer engineering and the PhD degree in computer science from the University of Hong Kong in 2002 and 2006, respectively. He is currently an assistant professor in the Department of Computer Science, Aalborg University. His research interests include databases and data mining, especially advanced query processing and mining techniques for complex types of data.