A Comparative Study of Spatial-Temporal Database Trends

Laila ElFangary[†], **Mahmoud Ahmed**[†], **and Shaimaa Bakr**[††]
[†] Information Systems Department, Faculty of Computers and Information, Helwan University, Cairo, Egypt
[††] Computer Science Dept., Cairo Higher Institute for Eng., Computer Science and Management, Cairo, Egypt

Summary
A comparative study is presented on the most known k-nearest neighbor search methods used by spatial-temporal database systems in order to provide the advantages and limitations of each algorithm used in system simulations. The scope is limited to the development of the grid indexing searching technique in terms of three different algorithms, including the well-known CPM, SEA-CNN, and CkNN algorithm. These algorithms don't make any assumptions about the movement of queries or objects. There are a number of functions proposed, which is used in: 1) partitioning the space around the query point in case of CPM and CkNN algorithms and 2) computing minimum and maximum distances between query and cell/level. All studied algorithms are compared together according to the required number of nearest neighbors, grid granularity, location update rate, speed, and population. An accuracy comparison is done between these algorithms to estimate the performance and determine the searching region error during query processing.

Key words:
continuous queries; grid index; kNN; NN accuracy; SR error.

1. Introduction

Due to the importance of Location-aware services, real-time spatial-temporal query processing algorithms that deal with large numbers of handheld devices and queries are needed [5,6]. These devices call for new spatial-temporal applications, in order to update any moving objects locations continuously over time [5, 7]. Spatial-temporal databases are the main topic in the geo-spatial and database communities [5, 10]. Due to rapid increase of spatial-temporal applications, new query processing techniques are taken into concideration to deal with both the spatial and temporal domains. The k-nearest neighbor [5, 8, 15] is a concept that is used to retrieve the k objects in a dataset that lie closest to a given query point. Dealing with continuous k-nearest neighbor (kNN) query over moving objects in location-dependent application requires real-time for updating moving objects and processing CkNN queries [5, 11]. There are two types of the Continuous k-nearest neighbor query: (i) a dynamic query object with a static data objects (e.g., finding the nearest gas station to a car), and (ii) both the query object and the data objects are dynamic (e.g., find the nearest police car to a moving vehicle) [2, 4, 5, 16].
There are many studies that deal with approximate kNN algorithms in order to minimize the processing cost (memory and time) [7, 17, 18, 19] which will affect the NN set results performance. Therefore, an accuracy study is done to stand on the performance of these approximate methods compared with an exact one. The accuracy is measured by comparing between the resulted and the exact NNs set in order to quantify the quality of results [7, 17, 18]. Nowadays, data indexing using grid index technique is taken into consideration in most of the existing algorithms in spatial database in order to reduce the processing time [1, 3, 5, 11, 12, 14]. Several researches are presented for answering continuously a collection of continuous CkNN queries through grid indexing [3, 5, 11, 12, 14]. In practice, construction of grid indexes enables allocation of different objects to their position or positions on the grid (static or dynamic objects respectively), then creating an index of object identifiers vs. grid cell identifiers for rapid access[5].

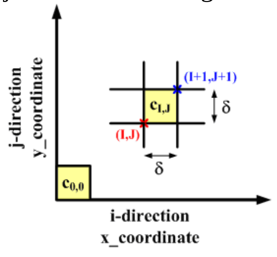

Fig. 1 Grid Space

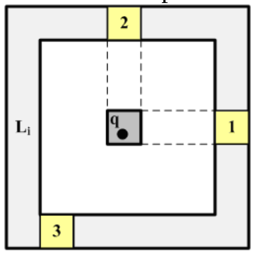

Fig. 2 Cell location w.r.t. query

For each cell $c_{I,J}$ has size $\delta*\delta$ as shown in figure 1, at column I and row J starting from low left corner of the grid space $c_{0,0}$ containing all objects and queries with x coordinate $\in [I.\ \delta\ ,\ (I+1).\ \delta]$ and y coordinate $\in [J.\ \delta\ ,\ (J+1).\ \delta]$ where the low left corner of the cell is denoted by (I, J) and the top right corner is denoted by (I +1, J +1) All objects and queries belong to the cell $c_{I,J}$ can be determined by [3, 5]:

$$I = \frac{x}{\delta} \tag{1}$$

$$J = \frac{y}{\delta} \tag{2}$$

The space around the cell $c_{I,J}$ can be divided into levels L_i where the zero level contains only the cell $c_{I,J}$ as shown in figure 2 [5]. There are two functions used by kNN algorithms, which are minimum and maximum distances between query point and cell boundary, which determine that the cell is affected by the query searching region (visited cells) or not in case of many algorithms like [3, 11, 14] and the cell is fully covered by the query searching

Manuscript received December 5, 2009
Manuscript revised December 20, 2009

region or not to obtain the bounded cells in case of CkNN algorithm [11] respectively.

1.1 Minimum query distance to a cell boundary

In order to compute the minimum distance *min_d_c* between a query point *q* and the nearest point of a certain cell $c_{I,J}$, the query coordinates *(q_x , q_y)*,cell width δ and cell location *(I,J)* have to be known. As shown in figure 2, there are three types of cell $c_{I,J}$ (i) cell has the same row *J* as query cell c_q, (ii) cell has the same column *I* as query cell c_q, (iii) and cell have row *J* and column *I* differ from query cell c_q. The algorithm used to compute *min_d_c* is illustrated below:

Algorithm 1. *min_Distance_Cell(c)*

```
1. If (c.I = q.cell.I)
2.     x_Dist = 0
3. Else If (c.I < q.cell.I)
4.     x_Dist = q.X – c.Right
5. Else
6.     x_Dist = c.X – q.X
7. If (c.J = q.cell.J)
8.     y_Dist = 0
9. Else If (c.J < q.cell.J)
10.    y_Dist = q.Y – c.Top
11. Else
12.    y_Dist = c.Y – q.Y
13. $min\_d_c$ = Square_Root(x_Dist * x_Dist + y_Dist * y_Dist)
```

1.2 Maximum query distance to a cell boundary

For all cell loctions shown in figure 2, the maximum cell distance *max_d_c* equals to the distance between the query point q and the farthest corner of the cell $c_{I,J}$ w.r.t. *q* . The algorithm used to compute *max_d_c* is illustrated below:

Algorithm 2. *maxDistance_Cell(c)*

```
1. If (c.I > q.cell.I)
2.     x_Dist = c.Right – q.loc.X
3. Else If (c.I < q.cell.I)
4.     x_Dist = q.loc.X – c.X
5. Else
6.     If (q.loc.X – q.cell.X < d/2)
7.         x_Dist = c.Right – q.loc.X
8.     Else
9.         x_Dist = q.loc.X – c.X
10. If (c.J > q.cell.J)
11.    y_Dist = c.Top – q.loc.Y
12. Else If (c.J < q.cell.J)
13.    y_Dist = q.loc.Y – c.Y
14. Else
15.    If (q.loc.Y – q.cell.Y < d/2)
16.        y_Dist = c.Top – q.loc.Y
17.    Else
18.        y_Dist = q.loc.Y – c.Y
19. $max\_d_c$ = Square_Root (x_Dist * x_Dist + y_Dist * y_Dist)
```

2. Shared execution algorithm

The *SEA-CNN* is an algorithm [5, 13, 14] which is used to answer continuously a group of *CkNN* queries. The main idea of *SEA-CNN* algorithm is minimizing redundent *I/O* operations by utilizing a *Shared Execution* paradigm. It has two main features which are: 1) *Incremental Evaluation* which is achieved by evaluating only the queries that their answers are affected by the movement of objects, 2) *Scalability* which is achieved by using a shared execution paradigm on concurrently running queries which is reducing repeated *I/O* operations. Then, evaluating a set of *CkNN* queries is reduced by a spatial join between the moving objects and query table as shown in figure 3.

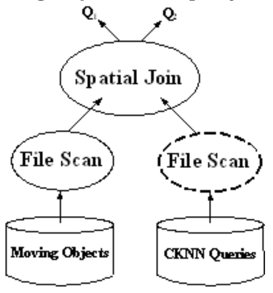

Fig. 3 Shared plan for two CkNN queries [5, 13, 14]

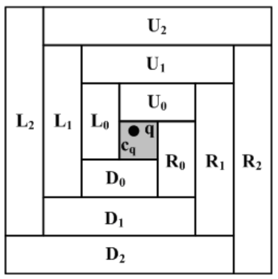

Fig. 4 Grid Conceptual Partitioning [3, 5]

3. Conceptual partitioning monitoring

CPM is an algorithm [3, 5] based on a conceptual partitioning of the space around the cell c_q which contains the query *q* into rectangles as shown in figure 4, in order to avoid unnecessary computations. Each rectangle is defined by a *direction* and a *level number*. The direction could be *U*, *D*, *L*, or *R* (for up, down, left and right) depending on the relative position of rectangle with respect to *q*. The level number indicates the number of rectangles between rectangle and c_q. The core of *CPM* is its *NN Computation* module, which retrieves the first-time results of incoming queries, and the new results of existing queries that change its location. This module produces and stores book-keeping information to facilitate fast *Update Handling*. If the new *NN* set of a query can be determined solely by the previous result and the set of updates, then access to the object grid *G* is avoided. Otherwise, *CPM* invokes the *NN re-computation* module, which uses the book-keeping information stored in the system to reduce the running time (compared to *NN Computation*).

3.1 Direction generator algorithm

In order to generate directions (*U*, *D*, *R*, and *L*) that bounds the current query searching region as shown in fig. 4, an algorithm is used which is illustrated below:

Algorithm 3. *DirGenerator(heapEntry)*

```
1. sign = 1
2. If (heapEntry.dir = Up or Down)
3.     If (heapEntry.dir = Down)
4.         sign = – 1
5.     If (heapEntry.cells.Right < GridSize * d)
6.         newEntry.cells.I = max(heapEntry.cells.I – 1 , 0)
7.         newEntry.cells.J = heapEntry.cells.J + sign
8.         newEntry.cells.Width = min(heapEntry.cells.Width
                       +2*d , heapEntry.cells.Right + d)
9.         newEntry.cells.Height = d
10.    Else If (heapEntry.cells.X > 0)
11.        newEntry.cells.I = heapEntry.cells.I – 1
12.        newEntry.cells.J = heapEntry.cells.J + sign
13.        newEntry.cells.Width = heapEntry.cells.Width + d
14.        newEntry.cells.Height = d
15.    Else
16.        newEntry.cells.I = heapEntry.cells.I – 1
17.        newEntry.cells.J = heapEntry.cells.J + sign
18.        newEntry.cells.Width = heapEntry.cells.Width + d
19.        newEntry.cells.Height = d
20. If (heapEntry.dir = Right or Left)
21.    If (heapEntry.dir = Left)
22.        sign = – 1
23.    If (heapEntry.cells.Top < GridSize * d)
24.        newEntry.cells.I = heapEntry.cells.I + sign
25.        newEntry.cells.J = max(heapEntry.cells.J – 1 , 0)
26.        newEntry.cells.Width = d
27.        newEntry.cells.Height = min(heapEntry.cells.Height
                       + 2*d , heapEntry.cells.Top + d)
28.    Else If (heapEntry.cells.Y > 0)
29.        newEntry.cells.I = heapEntry.cells.I + sign
30.        newEntry.cells.J = heapEntry.cells.J – 1
31.        newEntry.cells.Width = d
32.        newEntry.cells.Height = heapEntry.cells.Height + d
33.    Else
34.        newEntry.cells.I = heapEntry.cells.I + sign
35.        newEntry.cells.J = heapEntry.cells.J
36.        newEntry.cells.Width = d
37.        newEntry.cells.Height = heapEntry.cells.Height
38. newEntry.dist= heapEntry.dist + d
39. newEntry.cells.IsCellEntry = false
40. newEntry.cells.dir = heapEntry.dir
41. newEntry.cells.lvl = heapEntry.lvl + 1
42. Add newEntry into q.heap
```

3.2 Rectangle splitter generator algorithm

In order to split directions into cells to be sorted in a heap according to its minimum distance, an algorithm is used which is illustrated below:

Algorithm 4. *rectG(heapEntry)*

```
1. Delete heapEntry form q.heap
2. If (heapEntry.dir = Up or Down)
3.    cell_no = heapEntry.cells.Width / d
4.    for i = 0 to cell_no  do
5.        I = heapEntry.cells.I + i
6.        J = heapEntry.cells..J
7.        Add the cell with I , J and minDist(c, q) into q.heap
8. If (heapEntry.dir = Right or Left)
9.    cell_no = heapEntry.cells.Height / d
10.   for i = 0 to cell_no  do
11.       I = heapEntry.cells.I
12.       J = heapEntry.cells..J + i
13.       Add the cell c with I , J and minDist(c, q) into q.heap
14. DirGenerator(Dir , lvl)
```

4. Processing continuous k-NN queries in main memory grid index

C*k*NN [5, 11] processes new searching technique by *kNN search* algorithm. It searches for *nearest neighbors* of queries by partitioning grid space into levels. Figure 5 shows the mechanism with which the algorithm is implemented, the cells around the query are divided into levels *L*, where the level with index zero is the cell c_q containing the query point *q*, and during searching procedure, the cells of each level are visited in a clockwise direction, starting from the left bottom cell. And this gives the ability to obtain the result by checking few objects as possible. The *kNN search* algorithm has two phases: The first phase, if the number of objects in the current level and the total number of objects retrieved is not greater than *k*, and then the algorithm retrieves the objects from current level. During second phase, all cells in a cell level are sorted according to their minimum distance to the query. During query processing, *CkNN* tries to minimize the cost of C*k*NN query processing by reducing most unnecessary checking on queries / moving objects. For static C*k*NN queries, *incremental update* algorithm is processed. The *incremental update* algorithm makes the most of results obtained in last query processing, and attempts to reuse the data produced in query processing as more as possible.

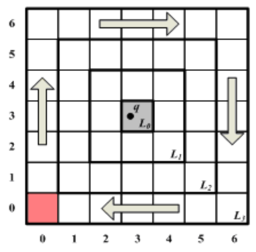

Fig. 5 Partition of Cell Levels

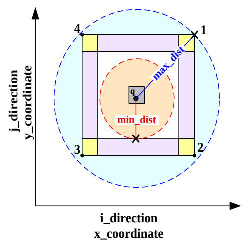

Fig. 6 Level distances to query

4.1 Level generator algorithm

In order to generate levels needed as shown in figure 5, an algorithm is used which is illustrated below:

Algorithm 5. *lvl_Gen(lvl)*

```
1. Read query entry q form QT and lvl;
2. Initialize an empty list cells;
3. If (lvl = 0)
4.    Add the cell contains q in cells list
```

```
5. Else
6.   Add the cell contains q in cells list
7.   row = q.I – lvl
8.   column = q.J – lvl
9.   If (row ≥ 0)
10.          for i = 0 to 2*lvl  do
11.             If (0 ≤ column < GridSize)
12.                      Add the cell with row & column in cells list
13.                      column = column + 1
14.             If (column < GridSize)
15.                      Add the cell with row & column in cells list
16. row = q.I – lvl + 1
17. column = q.J + lvl
18. If (column < GridSize)
19.    for i = 0 to 2*lvl  do
20.          If (0 ≤ row < GridSize)
21.             Add the cell with row & column in cells list
22.          row = row + 1
23.    If (row < GridSize)
24.          Add the cell with row & column in cells list
25. row = q.I + lvl
26. column = q.J + lvl – 1
27. If (row < GridSize)
28.    for i = 0 to 2*lvl  do
29.          If (0 ≤ column < GridSize)
30.             Add the cell with row & column in cells list
31.          column = column  –  1
32.    If (column ≥ GridSize)
33.          Add the cell with row & column in cells list
34. row = q.J + lvl – 1
35. column = q.J – lvl
36. If (column ≥ 0)
37.    for i = 0 to 2*lvl  do
38.          If (0 ≤ row < GridSize)
39.             Add the cell with row & column in cells list
40.          row = row – 1
```

4.2 Minimum Level distance of the query

In order to check the consistence/intersection of a level inside/with a query influence region, a minimum distance must be defined to compare it with the query best distance as shown in figure 6. The algorithm computes this distance is illustrated below:

Algorithm 6. *minDist_lvl(lvl)*

```
1. If (lvl = 0)
2.   minDist_lvl = 0
3. Else
4.   If (q.loc.X –  q.cell.X > d/2)
5.          minDist_lvl = q.cell.Right – q.loc.X + (lvl – 1)* d
6.   Else
7.          minDist_lvl = q.loc.X – q.cell.X + (lvl – 1)* d
8.   If (q.cell.Top – q.loc.Y < dist)
9.          minDist_lvl = q.cell.Top – q.loc.Y + (lvl – 1)* d
10.  Else If (q.loc.Y – q.cell.Y < minDist_lvl)
11.         minDist_lvl = q.loc.Y – q.cell.Y + (lvl – 1)* d
```

4.3 Maximum Level distance of the query

In order to check the consistence of a whole level inside a query influence region, a maximum distance must be defined to compare it with the query best distance as shown in figure 6. The algorithm computes this distance is illustrated below:

Algorithm 7. *maxDist_lvl(lvl)*

```
1. If (q.loc.X –  q.cell.X < d/2)
2.   max_x = (q.cell.I + lvl + 1) * d – q.loc.X
3. Else
4.   max_x = q.loc.X  – (q.cell.I – lvl) * d
5. If (q.loc.Y –  q.cell.Y < d/2)
6.   max_y =(q.cell.J + lvl + 1) * d – q.loc.Y
7. Else
8.   max_y = q.loc.Y  – (q.cell.J – lvl) * d
9. maxDist_lvl = Square-Root of (max_x ^ 2 + max_y ^ 2)
```

5. Experimental evaluation

An experimental evaluation has been implemented using C# programming language for all algorithms mentioned previously which are CkNN, CPM and SEA algorithms. Figures 7, 8 and 9 show the UML diagrams of classes used by SEA, CPM and CkNN respectively. These class diagrams are the backbone of the C# simulation program, which describe the static structure of the used system.

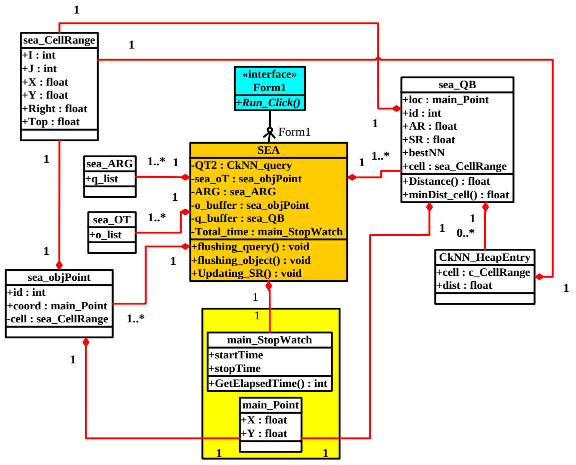

Fig. 7 UML diagram of classes used by SEA algorithm

All datasets used in this experimental study are created with the spatial-temporal generator mentioned in [9]. This generator is using the database of the road map of Oldenburg (a city in Germany) to obtain a set of objects/queries located on this map, where each object/query is represented by its identifier and coordinates at each timestamp. The velocity value of the generated object/query is *slow*, *medium*, and *fast* as mentioned in [9]. The queries are evaluated at every timestamp. The NN computation algorithm of CPM is used to compute the initial results of the queries in the implementation of SEA and CkNN, Table 1 shows the default values and ranges of the investigated parameters.

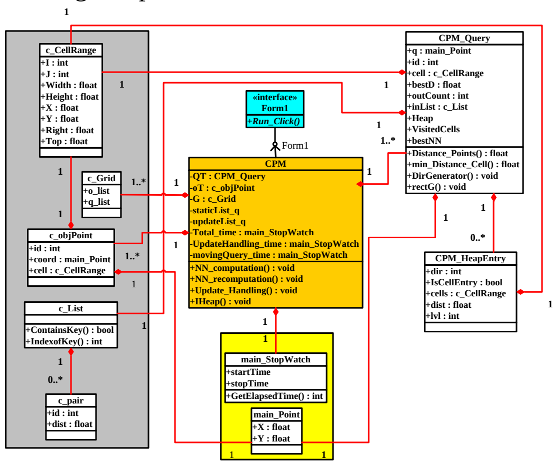

Fig. 8 UML diagram of classes used by CPM algorithm

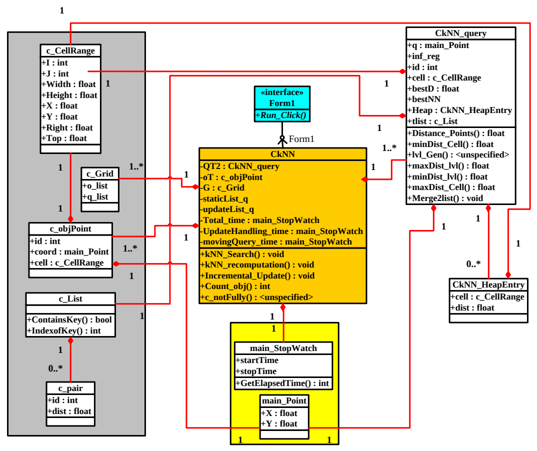

Fig. 9 UML diagram of classes used by CkNN algorithm

The used NN time is the time consumed to compute the NNs of dynamic queries from scratch, and UH time is the time consumed to update moving objects locations and updating the NNs of static queries. In each experiment a single parameter is varied, while setting the remaining ones to their default values. For all simulations we use an Intel 2GHz CPU with 1 GB memory.

Table 1 Experiments parameters (values ranges and defaults)

Parameter	Default value	Value range
Number of Grid cells.	32^2	32^2, 64^2, 128^2, 256^2, 512^2
Number of NN.	32	2, 16, 32, 48, 64
Number of objects.	10k	10, 30, 50k
Number of queries.	5k	1, 5, 10k
Update rate of objects.	50%	10, 30, 50%
Update rate of queries.	30%	10, 30, 50%
Speed of objects/queries.	medium	small, medium, fast

5.1 Effect of grid granularity

A comparison between the overall running time for CkNN and CPM algorithms by varying the grid cell size is shown in figures 10, 11 and 12 for objects population equal to 10k, 30k, and 50k respectively, where the number of cells of the grid space is ranging between 32^2 and 512^2. CkNN clearly consumes less CPU time than CPM for all grid sizes. CPM consumes more memory for query processing than CkNN due to unnecessary sorting of cells in case of dynamic and static queries. It is shown that the 128^2 grid (i.e., δ = Space width / 128) has the minimum cost for CkNN algorithm as mentioned in [11]. But in case of CPM, figure 10 shows that the 64^2 grid has the minimum cost and figures 11 and 12 show that the 128^2 grid has the minimum cost as mentioned in [3].

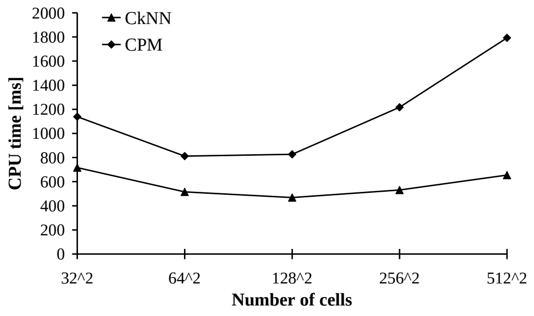

Fig. 10 CPU time versus number of grid space cells

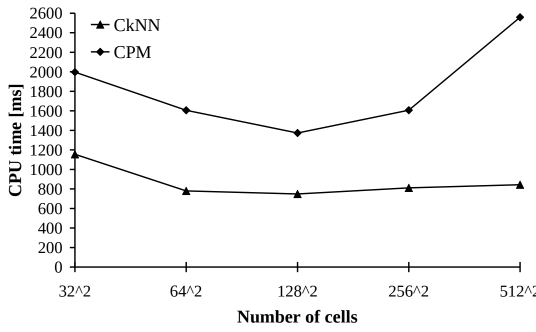

Fig. 11 CPU time versus number of grid space cells (objects = 30k)

As shown in figure 13, there are 4 cases used to show the effect of grid size in case of SEA algorithm which are:

- Case1 is using all default values in table 1.
- Case2, same as case1 but with 50,000 objects.
- Case3, same as case1 but only with 10% objects location update rate.
- Case4, same as case 2 but only with 10% objects location update rate.

For each case, there is an optimum value for best performance depending on the objects agility and population. Where decreasing object agility will decrease the overall processing time and also decrease its grid size optimum value as shown in cases (1, 3) and (2, 4). And increasing object population will increase the overall processing time and also increase its grid size optimum value as shown in cases (1, 2) and (3, 4).

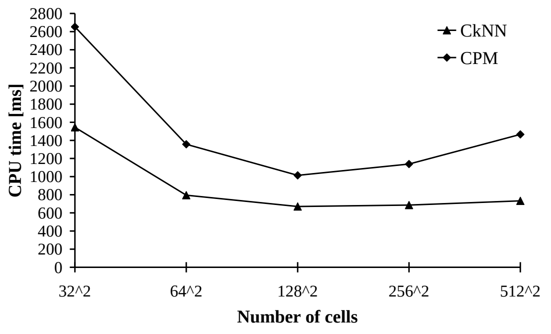

Fig. 12 CPU time versus number of grid space cells (objects = 50k)

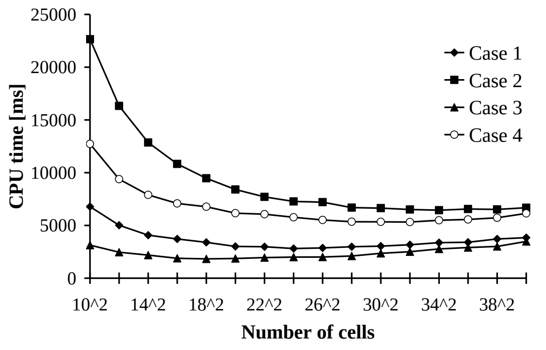

Fig. 13 Effect of grid size in case of SEA

Figures 10 and 13 show that SEA algorithm consumes more time than CkNN and CPM in all cases as mentioned in [3, 11], where both static and dynamic queries NNs answers are computed from scratch, and the searching process starting from far cells then getting closer until reaching the query cell and then getting far again, which will make unnecessary checking for more objects not in the final NNs set than in CkNN & CPM.

5.2 Effect of number of nearest neighbors

Figure 14 shows the CPU time on processing continuous *k*-NN queries which require various number of nearest neighbors (*k* = 2, 16, 32, 64) where all other parameters are the default values in table 1. It shows that the CPU time is increasing as a linear function with *k* as mentioned in [3, 11]. The reason is that when more NNs are required, the checked region extends (increasing the visited cells), where visiting cell is a complete scan over all objects inside cell bounds. This parameter is affected mainly by the object population around each query because if the population is high, the searching region will be smaller and vice versa. For SEA, cell accesses occur whenever some updates affect the answer region of a query (due to objects move away) and/or when the query moves. So the probability of increasing the query searching region due to these moving objects increases with increasing required k.

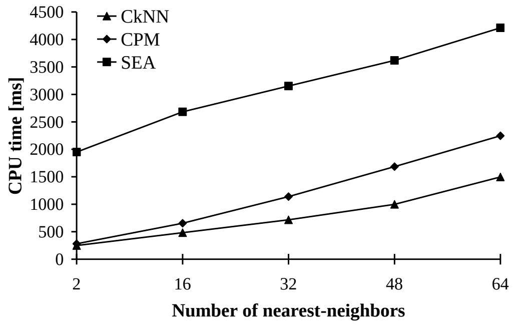

Fig. 14 CPU time versus number of nearest-neighbors

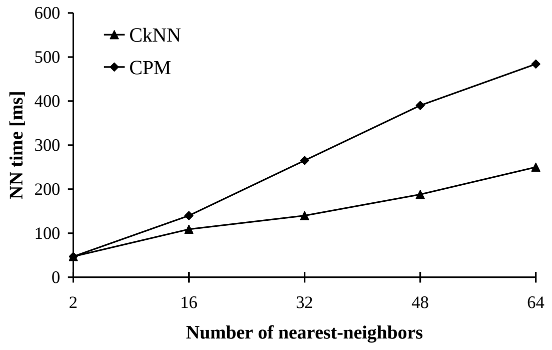

Fig. 15 NN computation time versus number of NNs

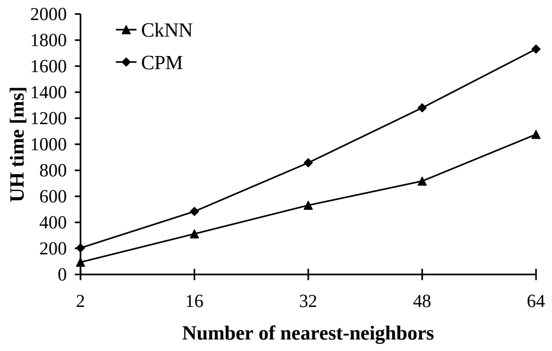

Fig. 16 UH computation time versus number of NNs

It is clear that CkNN outperforms CPM and SEA as *k* increases as mentioned in [3, 11], due to: (1) as shown in figure 15, CPM sorts all cells before checking objects in them (sorted heap), while CkNN directly retrieves objects from current level cells (if the number of objects is less than the required NNs) to build initial *k*-NN candidate and sorts fewer filtered cells to refine the query results; (2) as shown in figure 16, the incremental update algorithm of CkNN is more efficient than that of CPM, where the performance of *k*-NN re-computation algorithm of CkNN outperforms that of CPM, since CkNN reuses the remaining objects in NNs set (after objects updates) and complete this set by checking cells that are not completely inside the searching region for objects not in the NN set.

5.3 Effect of scalability

In order to quantify the effect of the scalability in terms of total number of objects/queries (i.e., population) on the performance of CkNN, CPM and SEA, where the number of objects ranged from 10,000 to 50,000 and the number of queries ranged from 1,000 to 10,000 respectively, while all other parameters are using default values from table 1.

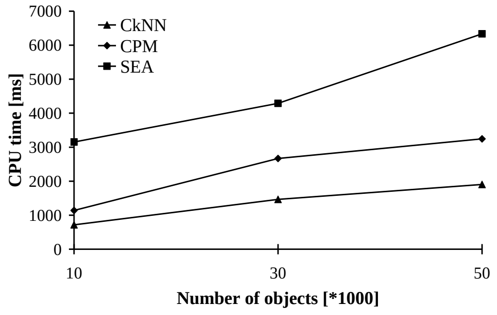

Fig. 17 CPU time versus number of objects

As the total number of objects/queries increase, the static and dynamic objects/queries increase. As shown in figures 17 and 18, the overall CPU time of CkNN, CPM and SEA increase linearly to the total number of objects /queries respectively as mentioned in [3, 11, 14], where the performance of CkNN is better than that of CPM and SEA. The change in overall CPU time in case of changing queries population is more sensitive than changing objects population (0.035 ms/object and 0.14825 ms/query).

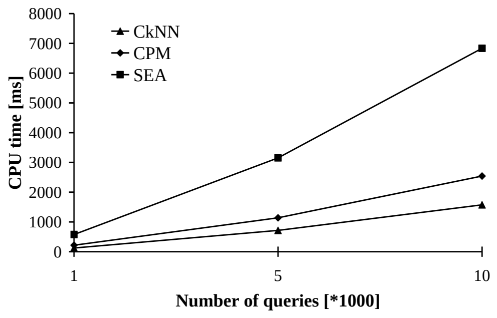

Fig. 18 CPU time versus number of queries

As shown in figure 19, as the objects population increases, the processing time for dynamic queries increases due to increasing the number of accessed objects inside visited cells. As shown in figure 20, the line slope between 10k and 30k is higher than the line between 30k and 50k, where the percentage of incoming objects increases (equals to or more than outgoing) avoiding re-computation processing. As shown in figures 21 and 22, the processing time increases with increasing the number of moving and static queries respectively.

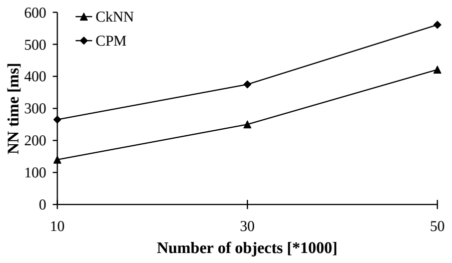

Figure 19 NN computation time versus number of objects

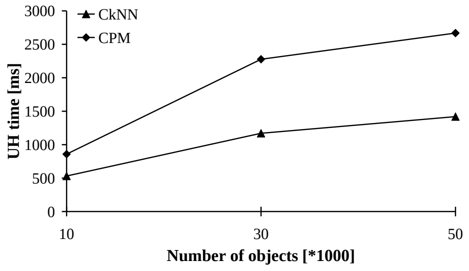

Figure 20 UH computation time versus number of objects

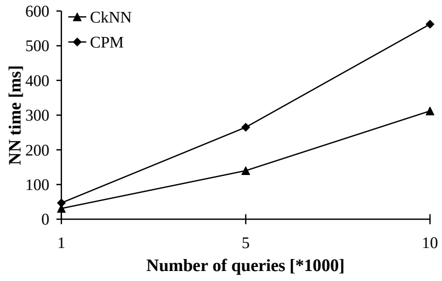

Figure 21 NN computation time versus number of queries

5.4 Effect of location update rate

In order to quantify the effect of the probability that objects/queries move within a timestamp (i.e., object/query agility) on the performance of CkNN, CPM and SEA, object/query agility vary from 10% to 50% and keep the remaining parameters fixed to their default values respectively.

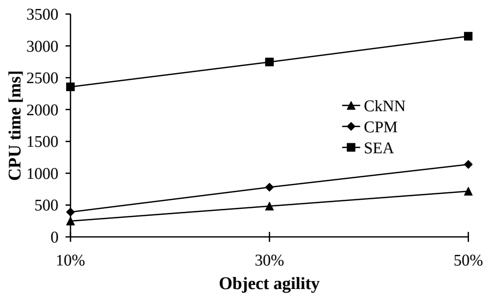

Figure 23 CPU time versus location update rate of objects

As shown in figures 23 and 24, the running time of CkNN, CPM and SEA are increasing linearly with increasing object and query agility respectively as mentioned in [3, 11, 14]. The performance of CkNN still outperforms CPM and SEA under all settings.

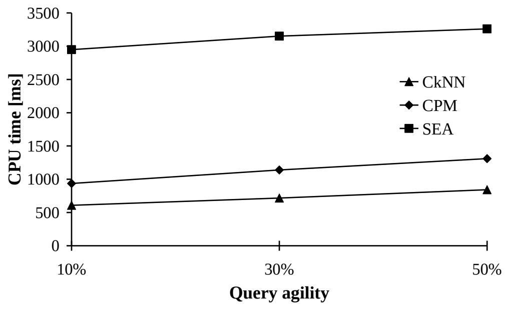

Figure 24 CPU time versus location update rate of queries

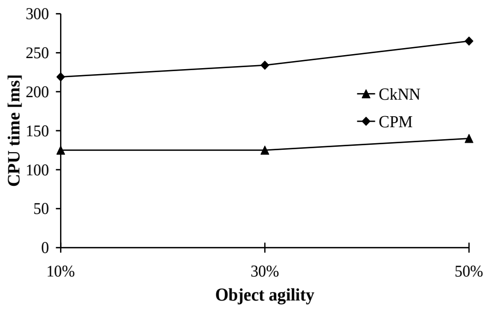

Figure 25 NN computation time versus object agility

Figure 25 illustrates the performance of moving queries processing over moving objects which is slightly increased due to: 1) computing NN set for those queries from scratch, 2) the object population around each query (if objects population increases, the number of accessed objects is increased and vice versa). Figure 26 shows that as object agility increases, the number of moving objects increases the time consumed in updating objects location and re-evaluating stationary queries. As shown in Figure 27, time consumed in evaluating NNs for moving queries in CkNN and CPM increases linearly due to increasing the number of moving queries. But from figure 28, the time consumed in re-evaluating NNs for static queries in CkNN and CPM is insensitive to the query agility.

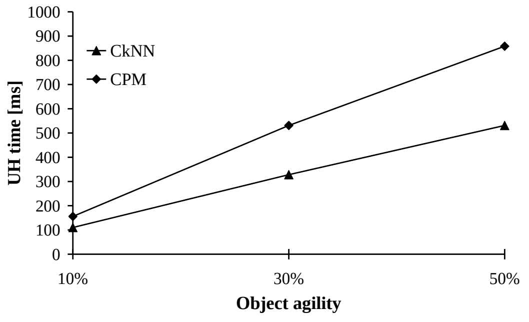

Figure 26 UH computation time versus object agility

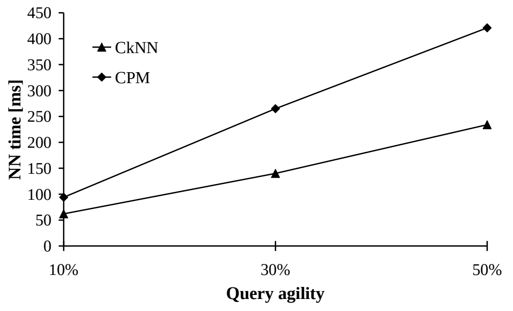

Figure 27 NN computation time versus query agility

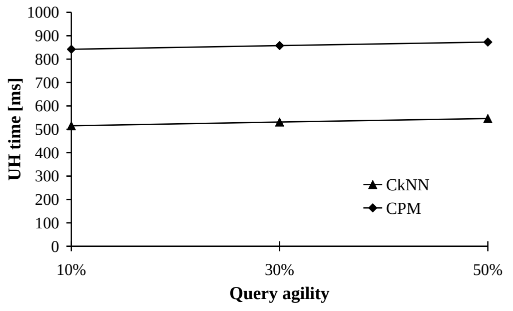

Figure 28 UH computation time versus query agility

5.5 Effect of object/query speed

The speed of the moving object / query is classified to three types in the used generator [9], slow, medium, and fast which are mentioned before.

Figures 29 and 30 compare the overall CPU time of CkNN, CPM and SEA with respect to the object/query speed respectively. The faster objects speed is, the farther they move. This will make more in/out objects inside NN set for static queries searching region, to check more objects by these queries. Therefore, the query processing cost increases as shown in figures 29 and 32. In case of SEA algorithm, as the object or query moves farther, the searching radius increases which will increase the CPU time as shown in figures 29 and 30.

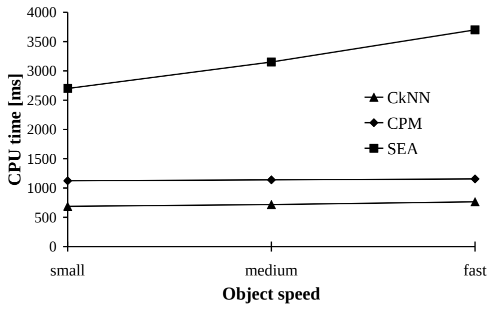

Figure 29 CPU time versus objects speed

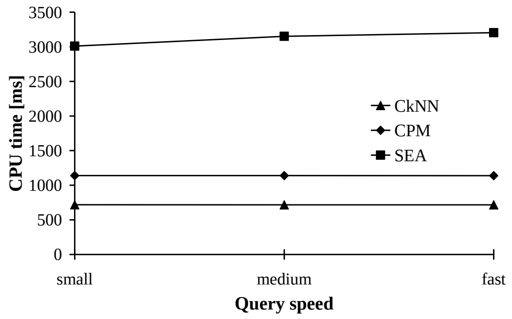

Figure 30 CPU time versus queries speed

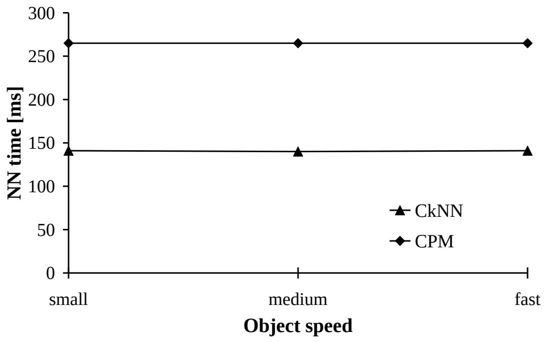

Figure 31 NN computation time versus object speed

In case of computing NNs of moving queries as shown in figure 31, the processing cost is not influenced by object speed, since CkNN and CPM compute NNs from scratch. On the contrary, as showed in figure 30, the performance of CkNN and CPM is not influenced by query speed, since these two methods compute *k*-NN of moved queries from scratch. These figures illustrate that CkNN outperforms CPM and SEA under all object/query speeds as mentioned in [3, 11].

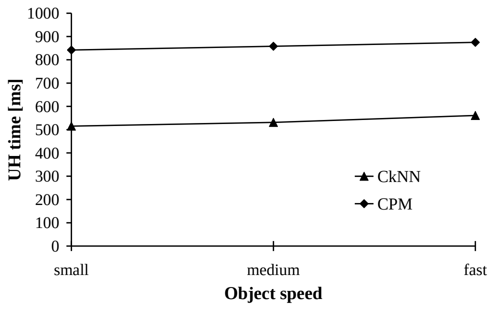

Figure 32 UH computation time versus object speed

6. Accuracy analysis during query processing

In this study, all proposed algorithms are exact. But the NN set results differ for CkNN, CPM and SEA algorithms during query processing depending on the average objects population in query cell and levels and the grid granularity. So an analysis will be done to determine the accuracy of the instantaneous NN set with respect to the final result and the current searching radius error with the final one to demonstrate well how these algorithms behave as searching process in-work. In order to investigate the accuracy during query processing, it has to be known that as the grid cell size increases, the average objects population for each cell decrease and as the required k objects increase, the visited cells increase too. Therefore, two different cases for two moving query point using the default values in table 1 with the same time-stamp will be investigated bellow:

6.1 Case 1

This case represents a query point with a high objects population inside its cell (for G = 32, c_q contains 54 objects but for G = 128, c_q contains 8 objects and lvl-1 contains 30 objects).

6.1.1 For G = 32

Figures 33, 34 show the incoming NN objects, where all objects distances are divided by the final query best distance, objects above 1 are not included in the final NN set (false NN objects) and objects on and below 1 are the final NN set (true NN objects).

Using SEA algorithm, all queries searching radius are computed before computing NN set. Then these queries are processed depending on the checked cell starting from far cells then getting closer until reaching the query cell and then getting far again. As in figure 33, all first 35 objects inserted into NN set are not in the final NN result having 0% NN accuracy, then the NN accuracy is increased as true NN objects are inserted till 100% as shown in figure 35 (false NN objects = 57).

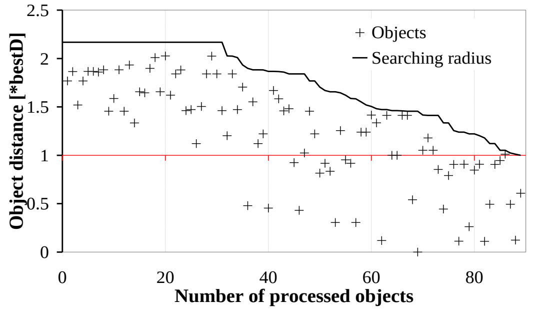

Fig. 33 Incoming NN objects for SEA (case-1: G = 32)

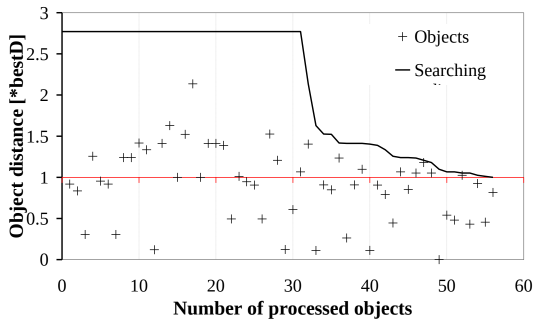

Fig. 34 Incoming NN objects for CkNN & CPM (case-1: G = 32)

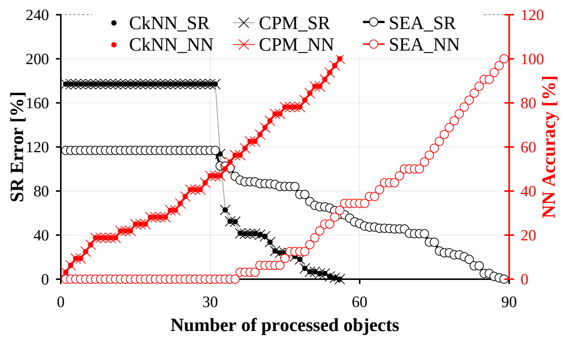

Figure 35 Searching radius error and NN accuracy (case-1: G = 32)

In this case, CPM and CkNN algorithms have the same mechanism, where c_q (level 0) is checked first (en-heaped into and de-heaped from SH only in case of CPM). The searching radius is set first to the maximum distance between the query cell and point. Hence this cell contains 54 objects in this case, a true NN objects are inserted as shown in figure 34 to increase the NN accuracy linearly with the number of inserted objects till 100% as shown in figure 35 (where false NN objects = 24). The searching radius error as shown in figure 35 is reduced after inserting first 32 objects, where in case of CkNN and CPM, the error suddenly decreases from 180% to 115% (NN accuracy ≈ 45%) but in case of SEA the error decreases from 120% to 105% (NN accuracy = 0%).

6.1.2 For G = 128

Using SEA algorithm, as shown in figure 36, the searching radius initially set as in case of G = 32 because it only depends on the displacement of the query and the movement of the last NN set candidates, all first 26 objects inserted into NN set are not in the final NN result having 0% NN accuracy, then the NN accuracy is increased as true NN objects are inserted till 100% as shown in figure 38 (where false NN objects = 36).

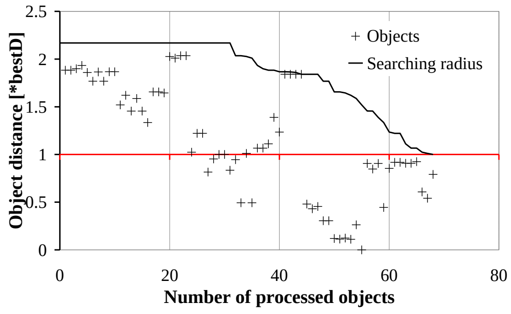

Fig. 36 Incoming NN objects for SEA (case-1: G = 128)

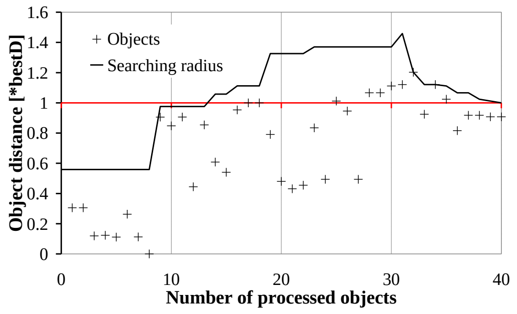

Fig. 37 Incoming NN objects for CkNN & CPM (case-1: G = 128)

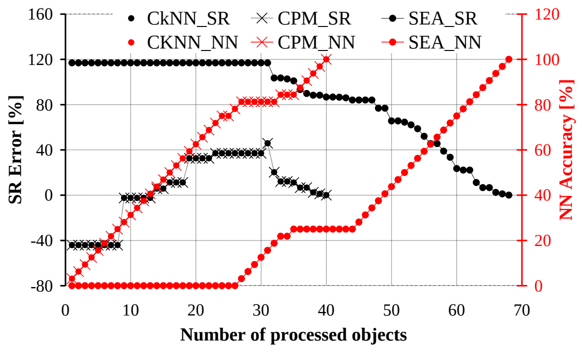

Fig. 38 Searching radius error and NN accuracy (case-1: G = 128)

Using CkNN algorithm, while the query cell contains 8 objects then all these objects are inserted directly in NN set. But for level 1 that contain 30 objects (more than 24), this level will be partitioned to 8 cells and sorted in heap in an ascending manner to be processed as CPM algorithm as shown in figure 37 where true NN objects are inserted to increase the NN accuracy linearly with the number of inserted objects till 100% as shown in figure 38 (where false NN objects = 8). The searching radius error as shown in figure 38 is reduced after inserting first 32 objects, where in case of CkNN and CPM the error decreases from 45% to 20% (NN accuracy ≈ 80%) but in case of SEA the error decrease from 120% to 100% (NN accuracy = 18%).

6.2 Case 2

This case represents a query point with low objects population inside its cell and levels (for G = 32, c_q contains 2 objects, lvl-1 = 21 objects and lvl-2 = 66 objects but for G = 128, c_q contains no objects, lvl-1 = 2 objects, lvl-2 = 0, lvl-3 = 9, lvl-4 = 1, lvl-5 = 10, lvl-6 = 8, lvl-7 = 17).

6.2.1 For G = 32

Using SEA algorithm, as shown in figure 39, all first 9 objects inserted into NN set are outside the final NN result having 0% NN accuracy, then the NN accuracy is increased as true NN objects are inserted till 100% as shown in figure 42 (where false NN objects = 25).

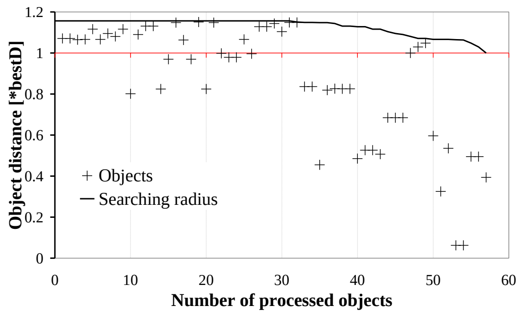

Fig. 39 Incoming NN objects for SEA (case-2: G = 32)

Using CPM algorithm, there are 12 cells de-heaped from SH (c_q, lvl-1 contains 8 cells and 3 cells from lvl-2), where only 8 cells have objects as shown in figure 40 (4 empty cells en-heaped and de-heaped). As true NN objects are inserted, the NN accuracy increases linearly with the number of inserted objects till 100% as shown in figure 42 (where false NN objects = 10). Using CkNN algorithm, 23 objects inside c_q and level 1 are inserted directly in NN set (less than 32). But for level 2 that contain 66 objects (more than 9), this level will be partitioned to 16 cells and sorted in heap in an ascending manner to be processed as CPM algorithm (only 3 cells de-heaped) as shown in figure 41 where true NN objects are inserted to increase the NN accuracy linearly with the number of inserted objects till 100% as shown in figure 42 (false NN = 10).

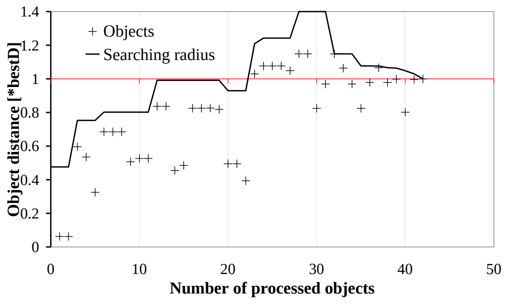

Fig. 40 Incoming NN objects for CPM (case-2: G = 32)

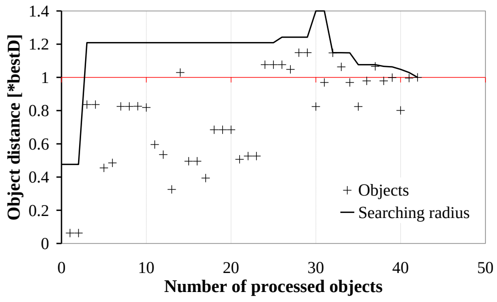

Fig. 41 Incoming NN objects for CkNN (case-2: G = 32)

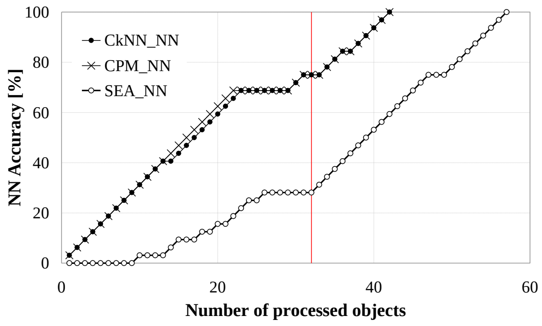

Fig. 42 NN accuracy during query processing (case-2: G = 32)

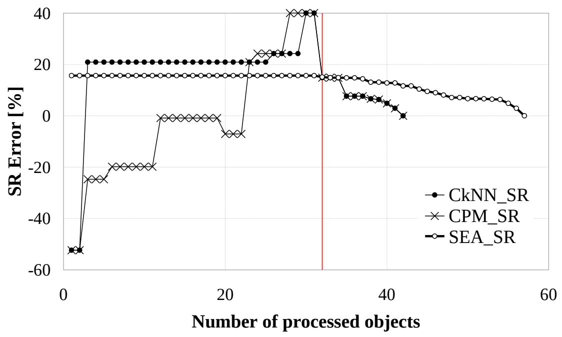

Fig. 43 Searching radius error during query processing (case-2: G=32)

The searching radius error as shown in figure 43 is reduced after inserting the first 32 objects, where in case of CkNN and CPM, the error decreases from 40% to 15% (NN accuracy ≈ 75%) but in case of SEA the error decreases from 16% to 15% (NN accuracy = 28%).

6.2.2 For G = 128

Using SEA algorithm, as shown in figure 44, all first 12 objects inserted into NN set are outside the final NN result having 0% NN accuracy, then the NN accuracy is increased as true NN objects are inserted till 100% as shown in figure 47 (where false NN objects = 25).

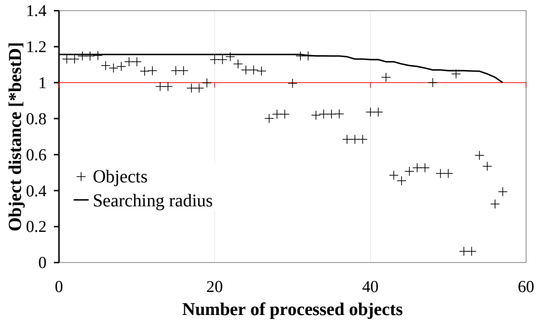

Fig. 44 Incoming NN objects for SEA algorithm (case-2: G = 128)

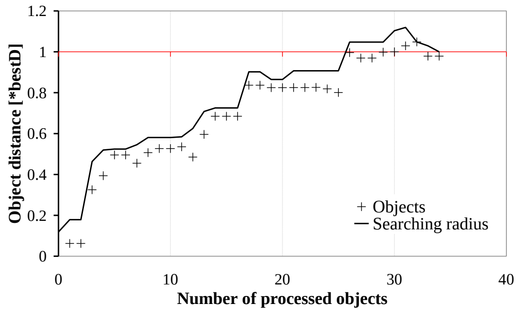

Fig. 45 Incoming NN objects for CPM algorithm (case-2: G = 128)

Using CPM algorithm, as shown in figure 45, there are 12 cells de-heaped from SH (c_q = 8, cells, lvl-1 = 16 cells, lvl-2 = 24, lvl-3 = 32, lvl-4 = 40, lvl-5 = 48, lvl-6 = 56 and 2 cells only from lvl-7), where only 20 cells have objects (207 empty cells en-heaped and de-heaped). As true NN objects are inserted, the NN accuracy increases linearly with the number of inserted objects till 100% as shown in figure 47 (where false NN objects = 2).

Using CkNN algorithm, as shown in figure 46, 30 objects inside levels from 0 to 6 are inserted directly in NN set (less than 32). But for level 7 that contain 17 objects (more than 2), this level will be partitioned to 56 cells and sorted in heap in an ascending manner to be processed as CPM algorithm (2 cells de-heaped) where true NN objects are inserted to increase the NN accuracy linearly with the number of inserted objects till 100% as shown in figure 47 (false NN objects = 5). The searching radius error as shown in figure 48 is reduced after inserting first 32 objects, where in case of CkNN and CPM the error decreases from 22% and 12% respectively to 5% (NN accuracy ≈ 75% and 94%) but in case of SEA, the error decreases from 16% to 5% (NN accuracy = 28%).

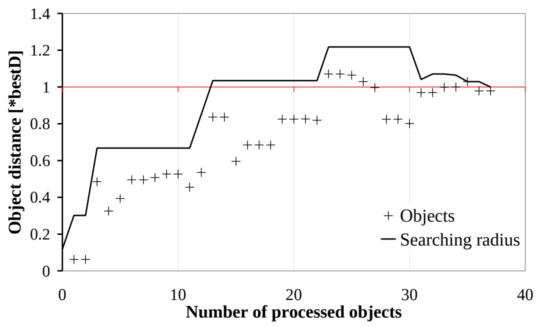

Fig. 46 Incoming NN objects for CkNN algorithm (case-2: G = 128)

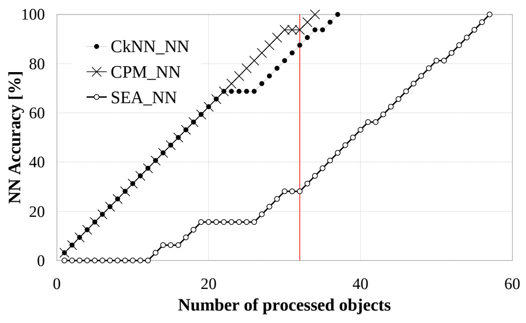

Fig. 47 NN accuracy during query processing (case-2: G = 128)

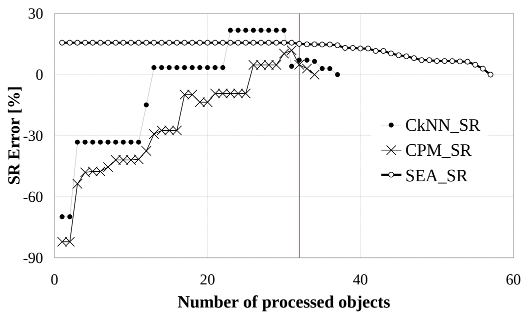

Fig. 48 Searching radius error during query processing (case-2: G = 128)

7. Conclusion

Three different algorithms, including the well-known CPM, SEA-CNN, and CkNN algorithm which are the most famous algorithms based on grid index technique are compared together. The implementation of these algorithms has been done using C# programming language. All simulations have been done on Intel 2 GHz CPU with 1

GB memory. The results of these simulations showed a good agreement with the theoretical comparison mentioned in [5]. In order to simulate these algorithms successfully, synthetic spatial-temporal data is generated using a well-defined objects generator [9].

A comprehensible performance evaluation between these algorithms has been done according to different parameters. These parameters are grid size, number of required nearest neighbors, total number of objects/queries, location update rate of objects/queries and object/query speed, where the performance of CkNN outperforms CPM and SEA under all conditions. SEA is the worst algorithm used compared with other algorithms, because for each query it visits all cells in its influence region (not using a sorted heap). CPM consumes more memory for query processing than CkNN algorithm, where CPM assigns each query a visit list and a sorted heap separately, but CkNN uses one sorted heap to process all queries. CPM re-computes query results from scratch for all moved queries. Although CPM utilizes a *visit list* as a cache of visited cells, all objects in those cells still need to be re-checked. This wastes the computation resource, since most of those objects are already checked in incremental update algorithm. On the contrary, CkNN keeps the objects in *kNN-list* and reuses them. Meanwhile, since the *k*-NN search algorithm of CkNN is more efficient, the overall running time of CkNN is less than that of CPM.

For grid size parameter, there is an optimum value for best performance depending on the objects agility and population, Where decreasing object agility will decrease the overall processing time and also decrease its grid size optimum value, and increasing object population will increase the overall processing time and also increase its grid size optimum value.

Finaly, an accuracy measure is done to evaluate the NN set results for proposed algorithms during query processing depending on the average objects population in each cell. This analysis determines the accuracy of the instantaneous NN set with respect to the final result and the current searching radius error with the final one to demonstrate how these algorithms behave as searching process in-work. Two different cases have been studied, for two moving queries points in high and low objects population region respectively. The SEA algorithm results show that for query with high objects population, as the grid size increases, the number of false objects decrease, but for queries with low objects population, as the grid size increases, the number of false objects increase. The CkNN and CPM algorithm results show that for both queries with high and low objects population, as the grid size increases, the number of false objects decrease. The CPM false objects is less than or equal to the CkNN false objects but CkNN consumes less cost due to its faster searching than CPM as mentioned before.

References

[1] D. Kalashnikov, S. Prabhakar, W. G.Aref, and S. Hambrusch, ”Efficient Evaluation of Continuous Range Queries on Moving Objects”, In proceeding of 13th International Conference on Database and Expert systems Applications (DEXA 2002), Lecture Notes In Computer Science; Vol. 2453, pages 731-740, 2002.

[2] Hae Don Chon, Divyakant Agrawal, and Amr El Abbadi, "Range and KNN Query Processing for Moving Objects in Grid Model", Mobile Network and Applications (MONET), Vol. 8, Issue No. 4, pages 401-412, August 2004.

[3] Kyriakos Mouratidis, Marios Hadjieleftheriou, and Dimitris Papadias. “Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring”, In proceeding of ACM SIGMOD, pages 634-645, Baltimore, Maryland, USA, June 14–16, 2005.

[4] Kyriakos Mouratidis. “Research Statement”, School of Information Systems, Singapore Management University, February 2007.

[5] Laila ElFangary , Mahmoud Ahmed and Shaimaa Bakr , “Review of k-Nearest Neighbor Methods Based on Grid indexing technique”, The 2009 World Congress in Computer Science, Computer Engineering, and Applied Computing (WORLDCOMP’09), Las Vegas, USA, July 13-16, 2009.

[6] Mohamed F. Mokbel, Xiaopeng Xiong, Walid G. Aref, Suzanne E. Hambrusch, Sunil Prabhakar, and Moustafa A. Hammad., “PLACE: A Query Processor for Handling Real-Time Spatial-Temporal Data Streams”, In proceedings of the 30th International Conference on Very Large Data Bases Conference (VLDB), pages 1377-1380, Toronto, Canada, 29th August - 3rd September, 2004.

[7] Mohamed F. Mokbel, "Scalable Continuous Query Processing in Location-Aware Database Services", PhD thesis, Purdue University, United States, August 2005.

[8] Uri Shaft and Raghu Ramakrishnan, "Theory of Nearest Neighbors Indexability", In ACM Transactions on Database Systems, Vol. 31, Issue No. 3, pages 814-838, September 2006.

[9] Thomas Brinkhoff, “A Framework for Generating Network-Based Moving Objects”, In GeoInformatica, Vol. 6, Issue No. 2, pp. 153-180, June 2002.

[10] Wang Huibing, Tang Xinming, Lei Bing, Yang Ping, and Chu Haifeng, “Modeling spatial-temporal data in version-difference model”, International Symposium on Spatio-temporal Modeling, Spatial Reasoning, Analysis, Data Mining and Data Fusion, Beijing, China, Aug 27- 29, 2005.

[11] Wei Zhang, Jianzhong Li, and Haiwei Pan,”Processing Continuous K-Nearest Neighbor Queries in Location-Dependent Applications”, International Journal of Computer Science and Network Security, Vol. 6, Issue No. 3A, pages 1-9 , March 2006.

[12] Xiaohui Yu, Ken Q. Pu, and Nick Koudas, "Monitoring k-Nearest Neighbor Queries over Moving Objects", 21st International Conference on Data Engineering (ICDE 2005), pages 631-642, Tokyo, Japan, April 5-8, 2005.

[13] Xiaopeng Xiong, Mohamed F. Mokbel, Walid G.Aref, Susanne E.Hambrusch, and Sunil Prabhakar, "Scalable Spatio-temporal Continuous Query Processing for Location-aware Services", In 16th International Conference on

Scientific and Statistical Databases, pages 317-326, Santorini Island, Greece, June 21-23, 2004.

[14] Xiaopeng Xiong, Mohamed F.Mokbel, and Walid G.Aref, "SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-Temporal Database", In the International Conference of Data Engineering (ICDE), pages 643-654, Tokyo, Japan, April 5-8, 2005.

[15] Xuegang Huang, Christian S.jensen, and Simonas Saltenis,"Multiple K-nearest Neighbor Query Processing in Spatial Network databases", 10th East-European Conference on Advancesin Databases and Information Systems(ADBIS 2006), pages 266-281, Hellas, September 3-7, 2006.

[16] Zhexuan Song and Nick Roussopoulos, "K-Nearest Neighbor Search for Moving Query Point", In proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases, Lecture Notes In Computer Science; Vol. 2121, pages 79-96, 2001.

[17] N. Koudas, B. Ooi, K. Tan, and R. Zhang, “Approximate NN Queries on Streams with Guaranteed Error/Performance Bounds” In proceeding of the 30th International Conference for Very Large Data Bases (VLDB’04), Toronto, Canada ,2004.

[18] Rimma V. Nehme," Continuous Query Processing on Spatio-Temporal Data Streams", M.Sc thesis, Faculty of Worcester Polytechnic Institute, June 2005.

[19] Yu-Ling Hsueh, Roger Zimmermann, and Meng-Han Yang. “Approximate Continuous K Nearest Neighbor Queries for Continuous Moving Objects with Pre-Defined Paths”, In proceedings of the 2nd International Workshop on Conceptual Modeling for Geographical Information Systems (CoMoGIS 2005), LNCS 3770, pp. 270-279, Klagenfurt, Austria, October 24-28, 2005.

Laila ElFangary is currently Associate Prof. in Information Systems Department, Faculty of Computers and Information - Helwan University, Cairo, Egypt.

Mahmoud Ahmed is currently a post doctoral fellow in University of Waterloo, Ontario, Canada and Assistant Prof. in Information Systems Department, Faculty of Computers and Information - Helwan University, Cairo, Egypt.

Shaimaa Bakr received the B.S. degree in Information Systems from Faculty of Computers and Information, Helwan University, Egypt in 2001. She works as an instructor in Computer Science Department, in Higher Institute of Engineering, Computer Science and Management, Cairo, Egypt. She is now finalizing her M.S. in Information Systems.