Efficient Path Oracles for Proximity Queries on Point Clouds (Technical Report)

ANONYMOUS and ANONYMOUS

The prevalence of computer graphics technology boosts the development of point clouds, which offer advantages over *Triangular Irregular Networks*, i.e., *TIN*s, in proximity queries. All existing on-the-fly shortest path query algorithms and oracles on a *TIN* are expensive, and no algorithms can answer shortest path queries on a point cloud directly. Thus, we propose two types of efficient shortest path oracles on a point cloud. They answer the shortest path query between (1) a pair of *Points-Of-Interests* (*POIs*), and (2) any point and a POI, respectively. We propose four adaptations of them to answer the query between any point and a POI (or any point if no POIs are given). We also propose two efficient proximity query algorithms using these oracles. Our two oracles and their proximity query algorithms outperform the best-known adapted oracle by 12 to 42,000 times in terms of the oracle construction time, oracle size and proximity query time, respectively[1].

1 INTRODUCTION

Conducting proximity queries on a 3D surface is a topic of widespread interest in both academia and industry [22, 48]. In academia, proximity queries (i.e., *shortest path queries* [15, 26, 27, 29, 44–46, 50–54, 58, 59], *k-Nearest Neighbor* (*kNN*) *queries* [18, 20, 40, 45, 48, 53] and *range queries* [36, 53, 60]) is a prevalent database research topic. In industry, Google Earth [4] and Cyberpunk 2077 [2] utilize the shortest path passing on a 3D surface (such as Earth) for route planning.

Point cloud and *TIN*: There are different representations of a 3D surface, including a point cloud and a *Triangular Irregular Network* (*TIN*). Figure 1 (a) is a 3D surface of Mount Rainier [32] (a national park in the USA) in an area of 20km × 20km. (1) A *point cloud* is represented by a set of 3D *points*. Figure 1 (b) is the point cloud of this surface. Given a point cloud, we create a *point cloud graph* in Figure 1 (c). Its *vertices* consist of the points in the point cloud. Its *edges* consist of a set of edges between each vertex and its 8 neighbor vertices in the 2D plane. Each edge's weight is set to the Euclidean distance between its two vertices. (2) A *TIN* contains a set of triangular *faces*. Each face consists of three *edges* connecting at three *vertices*. Figure 1 (d) is a *TIN* of this surface. We focus on three paths. (1) The blue path passing on a point cloud (graph) in Figures 1 (b & c). (2) The blue *surface path* [27] passing on (the faces of) a *TIN* in Figure 1 (d). (3) The purple dashed *network path* [27] passing on (the edges of) a *TIN* in Figure 1 (d).

1.1 Motivation

1.1.1 **Advantages of point cloud**. Point clouds have four advantages compared with *TIN*s.

(1) *More direct access to point cloud data*. We can obtain a point cloud using an iPhone LiDAR scanner [42] or a satellite [57] in 3s for a region of 1km^2. But, a *TIN* is usually converted from a point cloud via *triangulation* [12], where all vertices of faces are points in the point cloud.

(2) *Lower hard disk usage of a point cloud*. We only store point information of a point cloud, but store vertex, edge and face information of a *TIN*.

(3) *Smaller shortest path query time on a point cloud*. Calculating the shortest path passing on a point cloud is faster than calculating the shortest surface path passing on a *TIN*. Since the former path only passes on the edges of a point cloud graph, but the latter path passes on the faces of a *TIN* (the query region is larger than a point cloud graph). Calculating the shortest path passing on a point cloud has a similar time of calculating the shortest network path passing on a *TIN*.
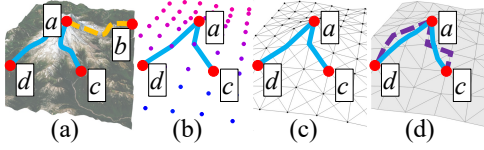

Authors' address: Anonymous; Anonymous.

Fig. 1. (a) A 3D surface, (b) paths passing on a point cloud, (c) a point cloud graph and (d) blue surface and purple network paths passing on a *TIN*

Table 1. P2P, A2P, A2A, AR2P and AR2AR queries

Query	Source or query object	Destination or target objects	3D surface
P2P	POI	POI(s)	point cloud/*TIN*
A2P	any point[a]	POI(s)	point cloud
A2A	any point[a]	any point[a]	point cloud
AR2P	arbitrary point[b]	POI(s)	*TIN*
AR2AR	arbitrary point[b]	arbitrary point[b]	*TIN*

Remark: [a]on (a set of points of) a point cloud in discrete space, and [b]on (the faces of) a *TIN* in continuous space.

(4) *Small distance error of the shortest path passing on a point cloud*. In Figures 1 (b & d), the blue shortest path passing on a point cloud is similar to the blue shortest surface path passing on a *TIN* (since each point connects with 8 neighbor points). But, in Figure 1 (d), the blue shortest surface path and purple dashed shortest network path passing on a *TIN* differ significantly (since each vertex only connects with 6 neighbor vertices).

1.1.2 P2P, A2P and A2A queries.
We study the shortest path query between an object X and an object Y, or the *kNN* and range queries among a query object X and target objects Y on a point cloud. There are three types of queries. Consider a set of pre-selected *Points-Of-Interests* (*POIs*) on a point cloud. (1) In the *POI-to-POI* (*P2P*) *query*, both X and Y are POIs (or a set of POIs). (2) In the *Any point-to-POI* (*A2P*) *query*, X is any point on the point cloud, and Y is a POI (or a set of POIs). X and Y can be swapped. (3) In the *Any point-to-Any point* (*A2A*) *query*, both X and Y are any point on the point cloud. The first three rows in Table 1 illustrate them.

1.1.3 Usage of oracles.
It is fast to answer the shortest path query on a point cloud *on-the-fly* (i.e., path pre-computation and index are not involved). But, we can pre-compute and store paths among a set of selected points in an index, called an *oracle* (a common term in database community [13, 44, 54, 56, 59, 61]). Then, we can use it to answer proximity queries more efficiently.

1.1.4 Snowfall example.
We performed a snowfall evacuation case study [33] in Mount Rainier [32] to evacuate tourists to nearby hotels.

(1) *P2P query*: In Figure 1 (a), we find shortest paths from a viewing platform (e.g., POI a) to k-nearest hotels (e.g., POIs b to d) due to hotels' limited capacity. c and d are the k-nearest hotels to a where $k = 2$. We find shortest paths quickly for life-saving.

(2) *A2P query*: If visitors can be anywhere with unknown locations before the oracle is constructed, we find shortest paths from a visitor to k-nearest hotels. If visitors are at viewing platforms and hotels are not available (only tents are available with unknown locations before the oracle is constructed), we find shortest paths from a viewing platform to k-tents.

(3) *A2A query*: We find shortest paths from a visitor to k-tents. But, if either hotels are available or visitors are at viewing platforms, the oracle for the A2A query is not suitable due to its large construction time. So, it is necessary to design an oracle for the A2P query.

1.1.5 Solar storm example.
We performed a solar storm evacuation case study [9] for NASA's Mars 2020 rover. During solar storms, rovers find shortest escape paths quickly from their current locations (any location) on Mars to shelters (POIs) to avoid damage. Since the memory size of a rover is 256MB [8], the oracle for the A2A query is not suitable due to its large size. So, it is necessary to design an oracle for the A2P query.

1.2 Challenges

1.2.1 **Non-existence of on-the-fly algorithms**. There is no study answering shortest path query on a point cloud *on-the-fly* directly. Existing algorithms [38, 41, 55] convert a point cloud to a *TIN*, and then calculate the shortest path passing on this *TIN*, which are very slow. The best-known *TIN* exact on-the-fly shortest surface path query algorithm [15, 46] uses a line to connect the source and destination on a 2D *TIN* unfolded by the 3D *TIN*. But, the unfolding technique is complicated. The best-known *TIN* approximate on-the-fly shortest surface path query algorithm [26, 51] uses Dijkstra's algorithm on a graph constructed by *TIN*'s vertices and discrete Steiner points placed on *TIN*'s edges. But, the graph has a complicated structure. Our experimental results show that they need several days for *kNN* or range queries.

1.2.2 **Non-existence of oracles**. There is no study answering shortest path query on a point cloud using *oracles* directly. The only related studies are oracles [25, 44, 45] on a *TIN*. We can adapt them to a point cloud by converting the point cloud to a *TIN*, and then constructing these oracles on this *TIN*. But, the best-known adapted *TIN* (1) oracle [44, 45] for the P2P query and (2) oracle [25] for both the A2P and A2A queries on a point cloud have a large oracle construction time. This is because their *loose criterion for algorithm earlier termination* drawback. They pre-compute shortest surface paths passing on the *TIN* from each POI (or point) to other POIs (or points) using the *Single-Source All-Destination* (*SSAD*) algorithm [15, 26, 27, 46, 51], i.e., a Dijkstra's algorithm [21], and store paths in a set. Although they provide a criterion to *terminate* the *SSAD* algorithm *earlier*, different POIs (or points) have the *same* criterion, meaning that it is not tight for some of these POIs (or points). Even after it has visited most POIs (or points), their earlier termination criterion are still not reached. Our experimental results show that their oracle construction time is one day.

1.3 Our First-Type Oracle

We propose an efficient shortest path oracle on a point cloud called *Rapid Construction path Oracle*, i.e., *RC-Oracle*. It answers $(1 + \epsilon)$-approximate P2P shortest path queries, where $\epsilon > 0$ is the *error parameter*. It can significantly reduce the oracle construction time for two reasons. (1) *Rapid point cloud on-the-fly shortest path query*: When constructing *RC-Oracle*, we propose an efficient algorithm *Fast on-the-Fly shortest path query*, i.e., *FastFly*, on a point cloud graph. It is a Dijkstra's algorithm [21] efficiently calculating the exact shortest path passing on a point cloud directly. (2) *Rapid oracle construction*: When constructing *RC-Oracle*, we use algorithm *FastFly*, i.e., a *SSAD* algorithm, to calculate the shortest path passing on the point cloud from for each POI to other POIs *simultaneously*. We set *tight* earlier termination criterion for different POIs. We adapt it to be *RC-Oracle-A2P-Small Construction time* (*RC-Oracle-A2P-SmCon*), *RC-Oracle-A2P-Small Query time* (*RC-Oracle-A2P-SmQue*) and *RC-Oracle-A2A* for A2P and A2A queries.

1.4 Our Second-Type Oracle

We propose a different efficient shortest path oracle on a point cloud called *TIght result path Oracle*, i.e., *TI-Oracle*. It answers $(1 + \epsilon)$-approximate A2P shortest path queries. It can significantly reduce the oracle construction time and oracle size due to the *tight shortest paths result*. Since when constructing *TI-Oracle*, we only calculate and store shortest paths passing on the point cloud among *some points close to the given POIs*, instead of among *all the points*. This saves time and space. We adapt it to be *TI-Oracle-A2A* for A2A queries.

1.5 Contributions and Organization

1.5.1 **Contributions of this journal paper**. We summarize our contributions.

(1) We propose algorithm *FastFly* on a *point cloud graph*, to efficiently calculate the shortest path passing on a point cloud directly. Although it is just a Dijkstra's algorithm, the proposed point cloud graph allows us to address the research gap of not having studies that find the shortest path on a point cloud (with many advantages compared with a *TIN*) directly. We also propose six oracles (four about *RC-Oracle* and two about *TI-Oracle*) to efficiently answer the P2P, A2P and A2A shortest path queries on a point cloud. We also propose two efficient proximity query algorithms to answer $(1 + \epsilon)$-approximate *kNN* and range queries using the first four and last two oracles.

(2) We provide theoretical analysis on six oracles' oracle construction time, oracle size, shortest path query time and error bound. We also provide theoretical analysis on algorithm *FastFly* and proximity query algorithms' query time and error bound.

(3) Our six oracles and their proximity query algorithms outperform the best-known adapted *TIN* oracles [25, 44, 45] on a point cloud in terms of the oracle construction time, oracle size and proximity (e.g., *kNN*) query time. Our experimental results show that for the P2P query on a point cloud with 2.5M points and 500 POIs, these values are 80s ≈ 1.3 min, 50MB and 12.5s for *RC-Oracle*, respectively. But, these values are 78,000s ≈ 21.7 hours, 1.5GB and 150s for the best-known adapted *TIN* oracle [44, 45], respectively. For the A2P query on a point cloud with 250k points and 500 POIs, these values are 25s, 28MB and 2.2s for *TI-Oracle*, respectively. But, these values are 1,050,000s ≈ 12 days, 300GB and 600s ≈ 10 min for the best-known adapted *TIN* oracle [25], respectively.

1.5.2 ***Contributions compared to the previous conference paper***. This journal paper extends the previous conference paper [52] by expanding from *P2P and A2A* queries to *P2P, A2P and A2A* queries. The previous paper has (1) algorithm *FastFly*, (2) *RC-Oracle* and *RC-Oracle-A2A*, and (3) their proximity query algorithm. We summarize our new contributions compared to it.

(1) We propose four new oracles *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* and *TI-Oracle-A2A*, and the proximity query algorithm for the last two oracles. The first three oracles for the A2P query perform better in cases of: (i) fewer proximity queries, (ii) more proximity queries when the density of POIs is high, and (iii) more proximity queries when the density of POIs is low.

(2) We provide theoretical analysis on the new oracles and new proximity query algorithm.

(3) Our three new oracles for the A2P query outperform the previously proposed best-known oracle *RC-Oracle-A2A* [52] in terms of the oracle construction time and oracle size. Their proximity query time (with their proximity query algorithms) are comparable to *RC-Oracle-A2A*. Our experimental results show that for the A2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time, oracle size and *kNN* query time are 250s ≈ 4.1 min, 280MB and 22s for *TI-Oracle*, respectively. But, these values are 42,000 ≈ 11.6 hours, 100GB and 12.5s for *RC-Oracle-A2A*, respectively. The oracle construction time of *RC-Oracle-A2P-SmCon* and *RC-Oracle-A2P-SmQue* are 320 times and 158 times better than *RC-Oracle-A2A*, respectively.

1.5.3 ***Organization***. The remainder of the paper is organized as follows. Section 2 provides the problem definition. Section 3 covers the related work. Sections 4 and 5 present *RC-Oracle*, *TI-Oracle*, and their adaptations. Section 6 covers the empirical studies and Section 7 concludes the paper.

2 PROBLEM DEFINITION

2.1 Notation and Definitions

2.1.1 ***Point cloud***. Given a set of points, we let C be a point cloud of these points with size N. The difference between "a set of points" and "a point cloud" is as follows. "A set of points" is a normal computer science term which describes a set containing a number of points. "A point cloud" is a special terminology in the literature describing the set of points describing a 3D surface or object. Each point $p \in C$ has three coordinate values, denoted by x_p, y_p and z_p. We let x_{max} and

x_{min} (resp. y_{max} and y_{min}) be the maximum and minimum x (resp. y) coordinate value for all points on C. We let $L_x = x_{max} - x_{min}$ (resp. $L_y = y_{max} - y_{min}$) be the side length of C along x-axis (resp. y-axis), and $L = \max(L_x, L_y)$. Figure 2 (a) shows a point cloud C with $L_x = L_y = 4$. Given a point p in C, we define $N(p)$ to be a set of neighbor points of p, which denotes the closest 8 surrounding points of p in the 2D plane. In Figure 2 (a), given a green point q, $N(q)$ is denoted as 7 blue points and 1 red point s. Let P be a set of POIs, which is a subset of C with size n, where $n \leq N$.
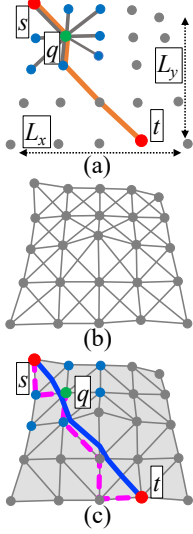
Fig. 2. (a) A point cloud, (b) a point cloud graph and (c) a *TIN*

Table 2. Summary of frequent used notations

Notation	Meaning
C	The point cloud with a set of points
N	The number of points of C
L	The maximum side length of C
P	The set of POI
n	The number of vertices of P
G	The point cloud graph of C
$d_E(p, p')$	The Euclidean distance between point p and p'
$\Pi^*(s, t\|C)$	The exact shortest path passing on C between s and t
$\|\Pi^*(s, t\|C)\|$	The distance of $\Pi^*(s, t\|C)$
ϵ	The error parameter
T	The *TIN* converted from C
$\Pi^*(s, t\|T)$	The exact shortest surface path passing on T between s and t
$\Pi_W(s, t\|T)$	The shortest network path passing on T between s and t
n'	The number of target objects in *kNN* or range query
$\Pi_A(s, t\|C)$	The shortest path passing on C between s and t returned by oracle A, where $A \in \{RC\text{-}Oracle, RC\text{-}Oracle\text{-}A2P\text{-}SmCon, RC\text{-}Oracle\text{-}A2P\text{-}SmQue, RC\text{-}Oracle\text{-}A2A, TI\text{-}Oracle, TI\text{-}Oracle\text{-}A2A\}$

2.1.2 **Point cloud graph**. We define G to be a point cloud graph of C. Let $G.V$ and $G.E$ be the set of vertices and edges of G. Each point in C is denoted by a vertex in $G.V$. For each point $q \in C$, $G.E$ consists of a set of edges between q and $q' \in N(q)$. The weight of each edge is set to the Euclidean distance between its two vertices. Figure 2 (b) shows a point cloud graph. Given a pair of points p and p' in 3D space, we define $d_E(p, p')$ to be the Euclidean distance between p and p'. Given a pair of points s and t on C, let $\Pi^*(s, t|C)$ be the exact shortest path passing on (G of) C between s and t. Figure 2 (a) shows an example of $\Pi^*(s, t|C)$ in orange line. We define $|\cdot|$ to be the distance of a path (e.g., $|\Pi^*(s, t|C)|$ is the distance of $\Pi^*(s, t|C)$).

2.1.3 **TIN**. Let T be a *TIN* that contains a set of vertices, edges and (triangular) faces. It is triangulated [12] from C where all vertices of faces are points in C, i.e., each (internal) point is part of four or more triangles. It is not a Delaunay triangulation [17]. Figure 2 (c) shows a T. In this figure, given a green vertex q, the neighbor vertices of q are 6 blue vertices. Given a pair of vertices s and t on T, let $\Pi^*(s, t|T)$ be the exact shortest surface path passing on (the faces of) T between s and t. Similarly, let $\Pi_W(s, t|T)$ be the shortest netWork path passing on (the edges of) T between s and t. Figure 2 (c) shows an example of $\Pi^*(s, t|T)$ in blue line and $\Pi_W(s, t|T)$ in pink line.

2.1.4 **Proximity queries**. (1) In the shortest path query, given a source s and a destination t, we answer the shortest path between s and t. (2) In the *kNN* query, given a query object q, a set of n' target objects O and a user parameter k, we answer all shortest paths from q to its k nearest target

objects. (3) In the range query, given q, O and a user parameter r, we answer all shortest paths from q to target objects whose distance to it is at most r.

2.1.5   *P2P, A2P and A2A queries on point cloud*. (1) In the P2P query, the shortest path passing on C from a source (a POI in P) to a destination (a POI in P) can pass on points in $C - P$ (and also P). (2) In the A2P query, either the source or destination is any point on C, while the other is a POI in P. Without loss of generality, we let the source be any point on C and destination be a POI in P. (3) In the A2A query, both the source and destination are any point on C.

(1) In the P2P query, for *kNN* and range queries, both query and target objects are POIs in P. (2) In the A2P query, there are two cases for *kNN* and range queries. (i) The query object is any point on C, and the target objects are POIs in P. (ii) The query object is a POI in P, and the target objects are any point on C. In other words, the A2P query is the same as the P2A query. (3) In the A2A query, for *kNN* and range queries, both query and target objects are any point on C.

Consider an area of points on C. (1) If the area is pre-selected that covers a portion of C, then it is equivalent to the P2P query, since we can regard points in the area as pre-selected POIs. (2) If the area is not pre-selected, then it is equivalent to the A2P or A2A query, since we can regard points in the area as any point in C. Consider our snowfall or solar storm evacuation example. If the locations of visitors, tents and rovers are unknown before the oracle is constructed, i.e., their corresponding area is not pre-selected, we need to build the oracle for the A2P or A2A query for efficient rescuing. In other words, "any point" in the A2P or A2A queries is opposite to "pre-selected POI" in the P2P query, i.e., we do not know the position of the source or destination before the oracle is constructed.

2.1.6   *P2P, AR2P and AR2AR queries on TIN*. Apart from the queries on a point cloud C, we also have similar queries on a *TIN* T. The P2P query on C is the same as on T. But, since an object may lie on the face of T (i.e., not just lie on the vertex of T), to adapt the A2P and A2A queries on C to T, we change "any point on (a set of points of) C" to "arbitrary point on (the faces of) T". "Any point on C" comes from a discrete space (i.e., a set of a certain number of points of C), and "arbitrary point on T" comes from a continuous space (i.e., the faces of T). Then, we obtain *ARbitrary point-to-POI* (*AR2P*) and *ARbitrary point-to-ARbitrary point* (*AR2AR*) *query* on T. They are more general than the A2P and A2A queries on C since an object may lie on the faces of T. The first, fourth and fifth rows in Table 1 illustrate them.

2.1.7   *Oracle*. An oracle [13, 44, 54, 56, 59, 61] is an index that stores the shortest path passing on C among a set of selected points. We can use it to answer shortest path queries among all pairs of selected points efficiently. Given a pair of points s and t on C, let $\Pi_A(s, t|C)$ be the shortest path between s and t returned by oracle A, where $A \in \{$*RC-Oracle, RC-Oracle-A2P-SmCon, RC-Oracle-A2P-SmQue, RC-Oracle-A2A, TI-Oracle, TI-Oracle-A2A*$\}$. We have $|\Pi_A(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for any pair of POIs or points s and t stored in oracle A. Table 2 shows a notation table.

2.2   Problem

The problem is to design efficient oracles on a point cloud for answering proximity queries.

3   RELATED WORK

3.1   On-the-fly Algorithms

There is no study answering shortest path query on a point cloud *on-the-fly* directly. Existing algorithms [38, 41, 55] convert a point cloud to a *TIN* in $O(N)$ time, and then calculate the shortest path passing on this *TIN*. There are two types of *TIN* shortest path query algorithms, i.e., (1) *shortest surface path* [15, 26, 28, 30, 46, 47, 51] and (2) *shortest network path* [27] query algorithms.

3.1.1 **TIN on-the-fly shortest surface path query algorithms**. There are two more sub-types. (1) *Exact algorithms*: Study [30] and study [47] use continuous Dijkstra and checking window algorithm for query both in $O(N^2 \log N)$ time. The best-known *TIN* exact on-the-fly shortest surface path query algorithm $\underline{D}irect\underline{I}on\text{-}\underline{O}riented$, i.e., algorithm *DIO* [15, 46] uses a line to connect the source and destination on a 2D *TIN* unfolded by the 3D *TIN*. It runs in $O(N^2)$ time. (2) *Approximate algorithms*: All algorithms [26, 28, 51] use Dijkstra's algorithm on a graph constructed by *TIN*'s vertices and discrete Steiner points placed on *TIN*'s edges. The best-known *TIN* $(1+\epsilon)$-approximate on-the-fly shortest surface path query algorithm $\underline{E}fficient$ $\underline{S}teiner$ $\underline{P}oint$, i.e., algorithm *ESP* [26, 51] runs in $O(\gamma N \log(\gamma N))$. Note that $\gamma = \frac{l_{max}}{\epsilon l_{min} \sqrt{1-\cos\theta}}$, l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of the *TIN*, and θ is the minimum inner angle of any face in the *TIN*.

3.1.2 **TIN on-the-fly shortest network path query algorithm**. Since the shortest network path does not cross the faces of a *TIN*, it is an approximate path. The best-known *TIN* approximate on-the-fly shortest network path query algorithm $\underline{DIJ}stra$, i.e., algorithm *DIJ* [27] uses Dijkstra's algorithm on *TIN*'s edge. It runs in $O(N \log N)$ time.

Adaptations: (1) Given a point cloud, we adapt algorithms *DIO* [15, 46], *ESP* [26, 51] and *DIJ* [27] to be algorithms *DIO-Adapt*, *ESP-Adapt* and *DIJ-Adapt*. Specifically, we first convert the point cloud to a *TIN*, and then compute the shortest path passing on the *TIN*. (2) Given a point cloud without data conversion, algorithm *DIO* cannot be directly adapted to the point cloud, because there is no face to be unfolded in a point cloud. But, algorithms *ESP* and *DIJ* can be adapted to the point cloud (if we let the path pass on the point cloud graph), and they become algorithm *FastFly*.

Drawback: These algorithms are time-consuming. Our experimental results show algorithms *DIO-Adapt*, *ESP-Adapt* and *DIJ-Adapt* first need to convert a point cloud with 2.5M points to a *TIN* in 21s. Then they perform the *kNN* query for all 500 objects on this *TIN* in 290,000s $\approx$ 3.4 days, 161,000s $\approx$ 1.9 days and 3,990s $\approx$ 1.1 hours, respectively. Performing the same query for algorithm *FastFly* on the point cloud needs 4,000s $\approx$ 1.1 hours.

3.2 Oracles for the Shortest Path Query

There is no study answering shortest path query on a point cloud using *oracles* directly. The only related studies are oracles on a *TIN*. Specifically, $\underline{S}pace$ $\underline{E}fficient$ $\underline{O}racle$ (*SE-Oracle*) [44, 45] and $\underline{U}pdatable$ $\underline{P}ath$ $\underline{O}racle$ (*UP-Oracle*) [54] answer P2P shortest path queries on a *TIN* using an oracle. $\underline{E}fficiently$ $\underline{AR}bitrary$ $points\text{-}to\text{-}arbitrary$ $points$ $\underline{O}racle$ (*EAR-Oracle*) [25] answers AR2AR shortest path queries on a *TIN* using an oracle. They store *TIN* shortest surface paths. By performing a linear scan using the shortest path query results, they can answer other proximity queries.

Adaptations: (1) Given a point cloud, we adapt *SE-Oracle* [44, 45], *UP-Oracle* [54] and *EAR-Oracle* [25] to be *SE-Oracle-Adapt*, *UP-Oracle-Adapt* and *EAR-Oracle-Adapt*. Specifically, we convert the point cloud to a *TIN*, and then construct these oracles on the *TIN*. (2) Given a point cloud without data conversion, *SE-Oracle*, *UP-Oracle* and *EAR-Oracle* can be adapted to the point cloud using algorithm *FastFly* (if we let the path pass on the point cloud graph). They only differ in the oracle construction compared with our oracles. But, their oracle construction time remains large due to the reasons to be discussed in their drawbacks.

3.2.1 **SE-Oracle-Adapt**. It uses a *compressed partition tree* [44, 45] and *well-separated node pair sets* [14] to store the $(1 + \epsilon)$-approximate P2P shortest surface paths passing on the converted *TIN*. Its oracle construction time, oracle size and shortest path query time are $O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ and $O(h^2)$, respectively. Note that h is the compressed partition tree's height and $\beta \in [1.5, 2]$ is a constant. It is the best-known adapted *TIN* oracle for the P2P query on a point cloud.

3.2.2 **UP-Oracle-Adapt**. It builds an updatable oracle on the converted *TIN*. It uses a complete graph to store pairwise P2P shortest surface paths passing on the converted *TIN* before updates. If the *TIN* updates, it updates the complete graph for affected regions. Its oracle construction time, oracle size and shortest path query time are $O(nN^2 + n^2)$, $O(n^2)$ and $O(1)$, respectively.

3.2.3 **EAR-Oracle-Adapt**. Similar to *SE-Oracle-Adapt*, it also uses well-separated node pair sets. But, it adapts *SE-Oracle-Adapt* from the P2P query on a point cloud to the A2P and A2A queries on a point cloud. Specifically, it places Steiner points on the faces of the converted *TIN*, and uses *highway nodes* as POIs in well-separated node pair sets construction. Its oracle construction time, oracle size and shortest path query time are $O(\lambda \xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$, $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ and $O(\lambda \xi \log(\lambda \xi))$, respectively. Note that λ is the number of highway nodes in a minimum square, ξ is the square root of the number of boxes, and m is the number of Steiner points per face. It is the best-known adapted *TIN* oracle for the A2P and A2A queries on a point cloud.

Drawbacks: (1) *SE-Oracle-Adapt*'s oracle construction time is large due to the *loose criterion for algorithm earlier termination*. For POIs in the same level of the compressed partition tree, they have the *same* earlier termination criteria of using the *SSAD* algorithm, meaning that it is not tight for some of these POIs. (2) *UP-Oracle-Adapt* does not have any pruning technique during oracle construction, so it performs worse than *SE-Oracle-Adapt* in terms of the oracle construction time. (3) *EAR-Oracle-Adapt* also has the *loose criterion for algorithm earlier termination* drawback. Our experimental results show that for the P2P query on a point cloud with 2.5M points and 500 POIs, the oracle construction time of *SE-Oracle-Adapt*, *SE-Oracle-Adapt* and *RC-Oracle* are 78,000s ≈ 21.7 hours, 160,000s ≈ 1.8 days and 80s ≈ 1.3 min, respectively. For the A2P query on a point cloud with 250k points and 500 POIs, the oracle construction time of *EAR-Oracle-Adapt* and *TI-Oracle* are 1,050,000s ≈ 12 days and 25s, respectively.

3.3 Oracles for Other Proximity Queries

There is no study answering proximity queries on a point cloud using *oracles* directly. The only related studies are oracles on a *TIN*. Specifically, studies [19, 20] use a multi-resolution terrain model to answer AR2P *kNN* queries on a *TIN* in $O(N^2)$ time. *SUrface Oracle* (*SU-Oracle*) [40] uses a surface oracle to answer AR2P *kNN* queries on a *TIN* in $O(N \log^2 N)$ time. We can adapt *SU-Oracle* to be *SU-Oracle-Adapt* on a point cloud in a similar way to *SE-Oracle-Adapt*. *SU-Oracle-Adapt* is the best-known adapted *TIN* oracle to directly answer *kNN* queries. But, studies [44, 45] show its *kNN* query time is up to 5 times larger than that of using *SE-Oracle-Adapt* with a linear scan of the shortest path query result. This is because *SU-Oracle-Adapt* only stores the first nearest POI of the given query point. It still needs to use on-the-fly algorithm to find other k-nearest POIs ($k > 1$), such results are not stored in the oracle. In addition, study [48] builds an oracle to answer the dynamic version of the *kNN* query, and study [49] builds an oracle to answer the reverse nearest neighbor query, but they are not our main focus.

3.4 Comparisons

We compare different algorithms that support the shortest path query on a point cloud in Table 3. When constructing *RC-Oracle*, we have tight earlier termination criteria for different POIs when using algorithm *FastFly*. We denote the naive version of our oracle as *RC-Oracle-Naive* if no earlier termination criterion is used. Our oracles outperform other baselines.

Table 3. Comparison of algorithms (support the shortest path query) on a point cloud

Algorithm	Oracle construction time		Oracle size		Shortest path query time		Error	Query type
Oracle-based algorithm								
SE-Oracle-Adapt [44, 45]	$O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh\log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small	P2P
UP-Oracle-Adapt [54]	$O(nN^2 + n^2)$	Large	$O(n^2)$	Large	$O(1)$	Small	Small	P2P
EAR-Oracle-Adapt [25]	$O(\lambda\xi mN^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh\log N)$	Large	$O(\frac{\lambda mN}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda\xi\log(\lambda\xi))$	Medium	Small	P2P, A2P, A2A
RC-Oracle-Naive	$O(nN\log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Small	Small	P2P
RC-Oracle	$O(\frac{N\log N}{\epsilon} + n\log n)$	**Small**	$O(\frac{n}{\epsilon})$	**Small**	$O(1)$	**Small**	**Small**	**P2P**
RC-Oracle-A2P-SmCon	$O(\frac{N\log N}{\epsilon} + n\log n)$	**Small**	$O(\frac{n}{\epsilon})$	**Small**	$O(N\log N)$	**Medium**	**Small**	**P2P, A2P**
RC-Oracle-A2P-SmQue	$O(\frac{N\log N}{\epsilon} + n\log n)$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(1)$	**Small**	**Small**	**P2P, A2P**
RC-Oracle-A2A	$O(\frac{N\log N}{\epsilon})$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(1)$	**Small**	**Small**	**P2P, A2P, A2A**
TI-Oracle	$O(\frac{N\log N}{\epsilon} + Nn + n\log n)$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(1)$	**Small**	**Small**	**P2P, A2P**
TI-Oracle-A2A	$O(\frac{N\log N}{\epsilon} + N\sqrt{N} + \sqrt{N}\log\sqrt{N})$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(1)$	**Small**	**Small**	**P2P, A2P, A2A**
On-the-fly algorithm								
DIO-Adapt [15, 46]	-	N/A	-	N/A	$O(N^2)$	Large	Small	P2P, A2P, A2A
ESP-Adapt [26, 51]	-	N/A	-	N/A	$O(\gamma N\log(\gamma N))$	Large	Small	P2P, A2P, A2A
DIJ-Adapt [27]	-	N/A	-	N/A	$O(N\log N)$	Medium	Medium	P2P, A2P, A2A
FastFly	-	**N/A**	-	**N/A**	$O(N\log N)$	**Medium**	**No error**	**P2P, A2P, A2A**

Remark: $n << N$. h is the compressed partition tree's height. β is the largest capacity dimension [44, 45]. λ is the number of highway nodes in a minimum square. ξ is the square root of the number of boxes, m is the number of Steiner points per face. $\gamma = \frac{l_{max}}{\epsilon l_{min}\sqrt{1-\cos\theta}}$. θ is the minimum inner angle of any face in T, l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of T.

4 RC-ORACLE AND ITS ADAPTATIONS

We illustrate *RC-Oracle*. In Figure 3 (a), we have a point cloud C, a set of POIs P and an error parameter ϵ. In Figures 3 (b) - (e), we construct *RC-Oracle* by calculating shortest paths among these POIs. In Figure 3 (f), we answer the shortest path query between two POIs using *RC-Oracle*.
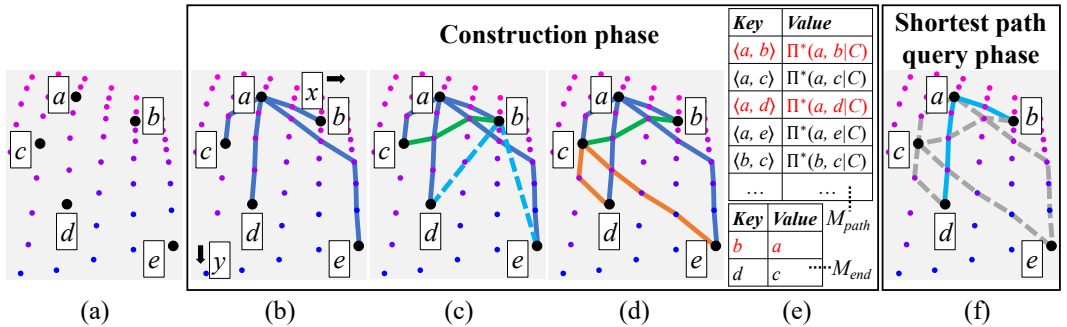

Fig. 3. *RC-Oracle* framework details

4.1 Overview of *RC-Oracle* and Its Adaptations

We introduce the two components and two phases.

4.1.1 Components of RC-Oracle and Its Adaptations. There are two components. Both of them are *hash tables* [16] that store key-value pairs. We use *RC-Oracle* as an example to illustrate.

(1) **The path map table** M_{path}: Each key-value pair stores a pair of endpoints u and v, as a key $\langle u, v \rangle$, and the corresponding exact shortest path $\Pi^*(u, v|C)$ passing on C, as a value. The endpoint is a POI in P or any point on C, depending on the oracle P2P, A2P or A2A query types. Given a key, a hash table returns the value in $O(1)$ time, and this applies to all hash tables in this paper. In Figure 3 (d), there are 7 exact shortest paths passing on C, and they are stored in M_{path} in Figure 3 (e). For the exact shortest paths passing on C between b and c, M_{path} stores $\langle b, c \rangle$ as a key and $\Pi^*(b, c|C)$ as a value.

(2) **The endpoint map table** M_{end}: Each key-value pair stores an endpoint u as a key, and another endpoint v as a value. u is the endpoint that we do not store all the exact shortest paths passing on C in M_{path} with u as a source. v is the endpoint close to u, where we concatenate $\Pi^*(u, v|C)$ and the exact shortest paths passing on C with v as a source, to approximate shortest paths passing on C with u as a source. In Figure 3 (d), a is close to b, we concatenate $\Pi^*(b, a|C)$ and the exact shortest paths passing on C with a as a source, to approximate shortest paths passing on C with b as a source. So, we store b as a key, and a as a value in M_{end} in Figure 3 (e).

4.1.2 *Phases of RC-Oracle and Its Adaptations*. There are two phases.

(1) *RC-Oracle* (see Figures 3 and 4): (i) In the construction phase, given C, P and ϵ, we pre-compute the exact shortest paths passing on C between some selected pairs of POIs. We store the calculated paths in M_{path}, and store the non-selected POIs and their corresponding selected POIs in M_{end}. (ii) In the shortest path query phase, given a pair of POIs in P, M_{path} and M_{end}, we answer the path results between this pair of POIs efficiently.

Construction phase	Point cloud C, set of POIs P and error parameter ε	➤	Path calculation	➤	Path map table M_{path} and endpoint map table M_{end}
Shortest path query phase	Pair of POIs s and t in P, M_{path} and M_{end}	➤	Path query	➤	Shortest path $\Pi_{RC\text{-}Oracle}(s, t|C)$
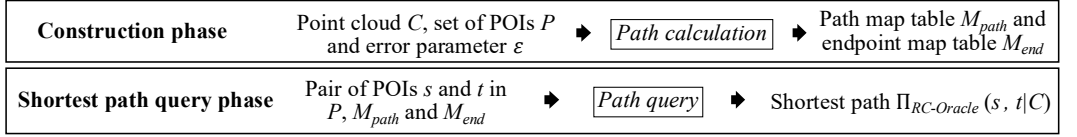
Fig. 4. *RC-Oracle* framework overview

(2) *RC-Oracle-A2P-SmCon*: (i) In the construction phase, given C, P and ϵ, the procedure is the same as *RC-Oracle*. (ii) In the shortest path query phase (see Figures 5 and 6), given any point (e.g., f) on C and a POI in P, we efficiently compute the exact shortest paths passing on C between this point and some selected POIs. We store the calculated paths in M_{path}, and store this point and its corresponding selected POIs in M_{end}. Then, given M_{path} and M_{end}, we return the path results between this point and this POI.
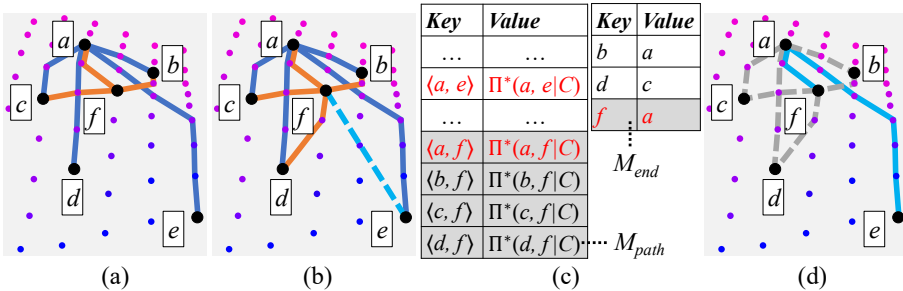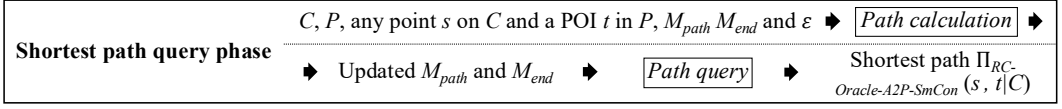

Fig. 5. *RC-Oracle-A2P-SmCon* shortest path query phase details

Shortest path query phase	C, P, any point s on C and a POI t in P, M_{path} M_{end} and ε ➡ $\boxed{\textit{Path calculation}}$ ➡
	➡ Updated M_{path} and M_{end} ➡ $\boxed{\textit{Path query}}$ ➡ Shortest path $\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(s, t\|C)$

Fig. 6. *RC-Oracle-A2P-SmCon* shortest path query phase overview

(3) *RC-Oracle-A2P-SmQue*: (i) In the construction phase, given C, P and ϵ, the procedure is similar to *RC-Oracle*. The only difference is that the destinations are not POIs in P, but all points on C. (ii) In the shortest path query phase, given any point on C and a POI in P, we answer the path results between this point and POI efficiently.

(4) *RC-Oracle-A2A*: (i) In the construction phase, given C and ϵ, the procedure is similar to *RC-Oracle*. The only difference is that no POI is given as input, we need to create POIs that have the same coordinate values as all points on C. (ii) In the shortest path query phase, given any pair of points on C, we answer the path results between this pair of points efficiently.

4.2 *RC-Oracle* and Its Proximity Query Algorithms

4.2.1 **Key Idea of *RC-Oracle***. We introduce the key idea of the small oracle construction time, small oracle size and small shortest path query time of *RC-Oracle* as follows.

(1) **Small oracle construction time**: We give the reason why *RC-Oracle* has a small oracle construction time.

(i) *Rapid point cloud on-the-fly shortest path query by algorithm FastFly*: When constructing *RC-Oracle*, given C and a pair of POIs s and t on C, we use algorithm *FastFly* on a point cloud graph of C between s and t. It is a Dijkstra's algorithm [21] calculating the exact shortest path passing on C directly. Figure 7 (a) shows the shortest path passing on a point cloud, and Figure 7 (b) (resp. Figure 7 (c)) shows the shortest surface (resp. network) path passing on a *TIN*. The paths in Figures 7 (a) and (b) are similar, but calculating the former path is much faster than the latter path. The path in Figure 7 (c) has a larger error than the path in Figure 7 (a). Thus, we use algorithm *FastFly* for constructing *RC-Oracle*.
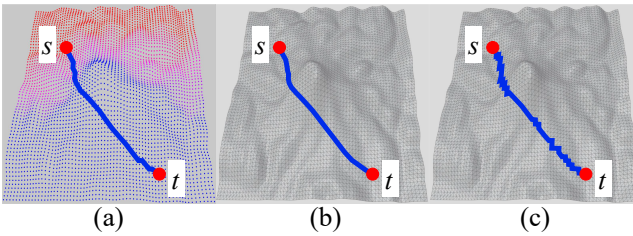
Fig. 7. (a) The shortest path passing on a point cloud, the shortest (b) surface and (c) network path passing on a *TIN*
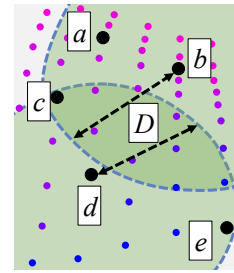
Fig. 8. *SE-Oracle-Adapt*

(ii) *Rapid oracle construction*: When constructing *RC-Oracle*, we regard each POI as a source and use algorithm *FastFly*, i.e., a *SSAD* algorithm, for n times for oracle construction. We set a *tight* earlier termination criteria for each POI to terminate the *SSAD* algorithm earlier for time-saving. There are two versions of a *SSAD* algorithm. (a) Given a source POI and a set of destination POIs, the *SSAD* algorithm can terminate earlier if it has visited all destination POIs. (b) Given a source POI and a *termination distance* (denoted by D), the *SSAD* algorithm can terminate earlier if the searching distance from the source POI is larger than D. We use the first version. For each POI, we

consider more geometry information of the point cloud (e.g., Euclidean distances and previously calculated shortest distances). Then, we can set *tight* earlier termination criteria to calculate the corresponding destination POIs. This ensures that the number of destination POIs is minimized, and these destination POIs are closer to the source POI compared with other POIs.

We use an example for illustration. In Figure 3 (a), we have a set of POIs a, b, c, d, e. In Figures 3 (b) - (d), we process these POIs based on their y-coordinate, i.e., we process them in the order of a, b, c, d, e. In Figure 3 (b), for a, we use the *SSAD* algorithm (i.e., *FastFly*) to calculate shortest paths passing on C from a to all other POIs. We store the paths in M_{path}. In Figure 3 (c), for b, if b is close to a, and if b is far away from d, i.e., $\frac{2}{\epsilon} \cdot |\Pi^*(a,b|C)| < d_E(b,d)$, then we can use $\Pi^*(b,a|C)$ and $\Pi^*(a,d|C)$ to approximate $\Pi^*(b,d|C)$. The first term "if b is close to a" is judged using the previously calculated $|\Pi^*(a,b|C)|$. The second term "if b is far away from d" is judged using the Euclidean distance $d_E(b,d)$. The case is similar to e by changing d to e. Thus, we just need to use the *SSAD* algorithm with b as a source, and terminate earlier when it has visited c. We store the path in M_{path}, and b as key and a as value in M_{end}. In Figure 3 (d), for c, we repeat the process as for a. We store the paths in M_{path}. Similarly, for d, we use $|\Pi^*(c,d|C)|$ and $d_E(c,e)$ to determine whether we can terminate the *SSAD* algorithm earlier with d as a source. We found that there is even no need to use the *SSAD* algorithm with d as the source. For *different* POIs b and d, we use *customized* termination criteria to calculate a tight and different set of destination POIs for time-saving. The criteria is $|\Pi^*(a,b|C)|$ and $d_E(b,d)$ for b, and is $|\Pi^*(c,d|C)|$ and $d_E(c,e)$ for d. We store d as key and c as value in M_{end}. In Figure 3 (e), we have M_{path} and M_{end}.

However, in *SE-Oracle-Adapt*, it has the *loose criterion for algorithm earlier termination* drawback. It first constructs the compressed partition tree. Then, it pre-computes the shortest surface paths passing on T using the *SSAD* algorithm (i.e., *DIO-Adapt*) with each POI as a source for n times, to construct the well-separated node pair sets. It uses the second version of the *SSAD* algorithm and sets termination distance $D = \frac{8r}{\epsilon} + 10r$, where r is the radius of the source POI in the compressed partition tree. Given two POIs p and q in the same level of the tree, their termination distances are the same (suppose that the value is d_1). However, for p, it is enough to terminate the *SSAD* algorithm when the searching distance from p is larger than d_2, where $d_2 < d_1$. This results in a large oracle construction time. In Figure 8, when d is processed, suppose that b and d are in the same level of the tree, and they use the *same* termination criteria to get the *same* termination distance D. Since $|\Pi^*(d,e|T)| < D$, for d, it cannot terminate the *SSAD* algorithm earlier until e is visited, which means that its termination criteria is loose. The two versions of the *SSAD* algorithm have similar ideas, we achieve a small oracle construction time mainly by using *tight* and customized termination criteria for different POIs.

(2) **Small oracle size**: We introduce the reason why *RC-Oracle* has a small oracle size. We only store a small number of paths in *RC-Oracle*, i.e., we do not store the paths between any pair of POIs. In Figure 3 (d), for a pair of POIs b and d, we use $\Pi^*(b,a|C)$ and $\Pi^*(a,d|C)$ to approximate $\Pi^*(b,d|C)$. We will not store $\Pi^*(b,d|C)$ in M_{path} for memory saving.

(3) **Small shortest path query time**: We use an example to introduce the reason why *RC-Oracle* has a small shortest path query time. In Figure 3 (f), we query the shortest path passing on C (i) between a source a and a destination d, and (ii) between a source b and a destination d. (i) For a and d, since $\langle a,d \rangle \in M_{path}.key$, we can directly return $\Pi^*(a,d|C)$. (ii) For b and d, since $\langle b,d \rangle \notin M_{path}.key$, b and d are both keys in M_{end}, we use the key b with a smaller y-coordinate value to retrieve the value a in M_{end}. Then, in M_{path}, we use $\langle b,a \rangle$ and $\langle a,d \rangle$ to retrieve $\Pi^*(b,a|C)$ and $\Pi^*(a,d|C)$, for approximating $\Pi^*(b,d|C)$.

4.2.2 *Implementation Details of RC-Oracle (Construction Phase)*. We give the construction phase of *RC-Oracle*.

Notation: Let P_{rema} be a set of remaining POIs of P that we have not used algorithm *FastFly* to calculate the exact shortest paths passing on C with POIs in P_{rema} as sources. P_{rema} is initialized to be P. Given an endpoint (a point on C or a POI in P) q, let $D(q)$ be a set of endpoints that we need to use *FastFly* to calculate the exact shortest paths passing on C from q to $p_i \in D(q)$ as destinations. In the construction phase of *RC-Oracle*, q and each element in $D(q)$ are POIs in P. $D(q)$ is empty at the beginning. In Figure 3 (c), $P_{rema} = \{c, d, e\}$ since we have not used *FastFly* to calculate the exact shortest paths with c, d, e as source. $D(b) = \{c\}$ since we need to use *FastFly* to calculate the exact shortest path from b to c.

Detail and example: Algorithm 1 shows the construction phase of *RC-Oracle* in detail, and the following illustrates it with an example.

Algorithm 1 *RC-Oracle-Construction* (C, P, ϵ)

Input: A point cloud C. A set of POIs P. An error parameter ϵ.
Output: A path map table M_{path}. An endpoint map table M_{end}.
1: $P_{rema} \leftarrow P, M_{path} \leftarrow \emptyset, M_{end} \leftarrow \emptyset$
2: **if** $L_x \geq L_y$ (resp. $L_x < L_y$) **then**
3:      Sort POIs in P_{rema} in ascending order using x-coordinate (resp. y-coordinate)
4: **while** P_{rema} is not empty **do**
5:      $u \leftarrow$ a POI in P_{rema} with the smallest x-coordinate / y-coordinate
6:      $P_{rema} \leftarrow P_{rema} - \{u\}, P'_{rema} \leftarrow P_{rema}$
7:      Calculate the exact shortest paths passing on C from u to each POI in P'_{rema} simultaneously using algorithm *FastFly*
8:      **for** each POI $v \in P'_{rema}$ **do**
9:          $key \leftarrow \langle u, v \rangle$, $value \leftarrow \Pi^*(u, v | C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
10:      Sort POIs in P'_{rema} in ascending order based on $|\Pi^*(u, v | C)|$ for each $v \in P_{rema}$
11:      **for** each sorted POI $v \in P'_{rema}$ such that $d_E(u, v) \leq \epsilon L$ **do**
12:          $P_{rema} \leftarrow P_{rema} - \{v\}, P'_{rema} \leftarrow P'_{rema} - \{v\}, D(v) \leftarrow \emptyset$
13:          **for** each POI $w \in P'_{rema}$ **do**
14:              **if** $\frac{2}{\epsilon} \cdot |\Pi^*(u, v | C)| < d_E(v, w)$ and $v \notin M_{end}.key$ **then**
15:                 $key \leftarrow v$, $value \leftarrow u$, $M_{end} \leftarrow M_{end} \cup \{key, value\}$
16:              **else if** $\frac{2}{\epsilon} \cdot |\Pi^*(u, v | C)| \geq d_E(v, w)$ **then**
17:                 $D(v) \leftarrow D(v) \cup \{w\}$
18:          Calculate the exact shortest paths passing on C from v to each POI in $D(v)$ simultaneously using algorithm *FastFly*
19:          **for** each POI $w \in D(v)$ **do**
20:              $key \leftarrow \langle v, w \rangle$, $value \leftarrow \Pi^*(v, w | C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
21: **return** M_{path} and M_{end}

---

(1) *POIs sort* (lines 2-3): In Figure 3 (b), since $L_x < L_y$, the sorted POIs are a, b, c, d, e.

(2) *Shortest paths calculation* (lines 4-20): There are two steps.

(i) *Exact shortest paths calculation* (lines 5-9): In Figure 3 (b), a has the smallest y-coordinate based on the sorted POIs in P_{rema}. We delete a from P_{rema}, so $P_{rema} = P'_{rema} = \{b, c, d, e\}$. We calculate the exact shortest paths passing on C from a to b, c, d, e (in purple lines) using algorithm *FastFly*, and store each POIs pair as a key and the corresponding path as a value in M_{path}.

(ii) *Shortest paths approximation* (lines 10-20): In Figure 3 (c), b is the POI in P'_{rema} closest to a, c is the POI in P'_{rema} second closest to a, so the following order is $b, c, \ldots$. There are two cases:

- *Approximation loop start* (lines 11-20): In Figure 3 (c), we first select a's closest POI in P'_{rema}, i.e., b. Since $d_E(a, b) \leq \epsilon L$, it means a and b are not far away. We start the approximation loop, delete b from P_{rema} and P'_{rema}, so $P_{rema} = P'_{rema} = \{c, d, e\}$. There are three steps:

- *Far away POIs selection* (lines 13-15): In Figure 3 (c), $\frac{2}{\epsilon} \cdot |\Pi^*(a,b|C)| < d_E(b,d)$, $\frac{2}{\epsilon} \cdot |\Pi^*(a,b|C)| < d_E(b,e)$, $d \notin M_{end}.key$ and $e \notin M_{end}.key$, it means d and e are far away from b. So, we can use $\Pi^*(b,a|C)$ and $\Pi^*(a,d|C)$ to approximate $\Pi^*(b,d|C)$, and use $\Pi^*(b,a|C)$ and $\Pi^*(a,e|C)$ to approximate $\Pi^*(b,e|C)$. We get $\Pi_{RC\text{-}Oracle}(b,d|C)$ by concatenating $\Pi^*(b,a|C)$ and $\Pi^*(a,d|C)$, and get $\Pi_{RC\text{-}Oracle}(b,e|C)$ by concatenating $\Pi^*(b,a|C)$ and $\Pi^*(a,e|C)$. We store b as key and a as value in M_{end}.
- *Close POIs selection* (line 13 and lines 16-17): In Figure 3 (c), $\frac{2}{\epsilon} \cdot |\Pi^*(a,b|C)| \geq d_E(b,c)$, it means c is close to b. So, we cannot use any existing exact shortest paths passing on C to approximate $\Pi^*(b,c|C)$, and then we store c into $D(b)$.
- *Selected exact shortest paths calculation* (lines 18-20): In Figure 3 (c), when we have processed all POIs in P'_{rema} with b as a source, we have $D(b) = \{c\}$. We use algorithm *FastFly* to calculate the exact shortest path passing on C between b and c, i.e., $\Pi^*(b,c|C)$ (in green line). We store $\langle b,c \rangle$ as a key and $\Pi^*(b,c|C)$ as a value in M_{path}. Note that we can terminate algorithm *FastFly* earlier since we just need to visit POIs that are close to b, and we do not need to visit d and e.
- *Approximation loop end* (line 11): In Figure 3 (c), since we have processed b, and $P'_{rema} = \{c,d,e\}$, we select a's closest POI in P'_{rema}, i.e., c. Since $d_E(a,c) > \epsilon L$, it means a and c are far away, and it is unlikely to have a POI m that satisfies $\frac{2}{\epsilon} \cdot |\Pi^*(a,c|C)| < d_E(c,m)$. So, we end the approximation loop and terminate the iteration.

(3) *Shortest paths calculation iteration* (lines 4-20): In Figure 3 (d), we repeat the iteration, and calculate the exact shortest paths passing on C with c as a source (in orange lines).

4.2.3 Implementation Details of RC-Oracle (Shortest Path Query Phase). We give the shortest path query phase of *RC-Oracle*.

Detail and example: Given a pair of POIs s and t in P, there are two cases (s and t are interchangeable, i.e., $\langle s,t \rangle = \langle t,s \rangle$):

(1) *Exact shortest path retrieval*: If $\langle s,t \rangle \in M_{path}.key$, we use $\langle s,t \rangle$ to retrieve $\Pi^*(s,t|C)$ as $\Pi_{RC\text{-}Oracle}(s,t|C)$ in $O(1)$ time. In Figures 3 (d) and (e), given a and d, since $\langle a,d \rangle \in M_{path}.key$, we retrieve $\Pi^*(a,d|C)$.

(2) *Approximate shortest path retrieval*: If $\langle s,t \rangle \notin M_{path}.key$, it means $\Pi^*(s,t|C)$ is approximated by two exact shortest paths passing on C in M_{path}, and (i) either s or t is a key in M_{end}, or (ii) both s and t are keys in M_{end}. Without loss of generality, suppose that (i) s exists in M_{end} if either s or t is a key in M_{end}, or (ii) the x- (resp. y-) coordinate of s is smaller than or equal to t when $L_x \geq L_y$ (resp. $L_x < L_y$) if both s and t are keys in M_{end}. For both of two cases, we use the key s to retrieve the value s' in M_{end} in $O(1)$ time, and then use $\langle s,s' \rangle$ and $\langle s',t \rangle$ to retrieve $\Pi^*(s,s'|C)$ and $\Pi^*(s',t|C)$ in M_{path} in $O(1)$ time. We then use $\Pi^*(s,s'|C)$ and $\Pi^*(s',t|C)$ as $\Pi_{RC\text{-}Oracle}(s,t|C)$ to approximate $\Pi^*(s,t|C)$. (i) In Figures 3 (d) and (e), given b and e, $\langle b,e \rangle \notin M_{path}.key$, and b is a key in M_{end}. So, we use the key b to retrieve the value a in M_{end}. Then, in M_{path}, we use $\langle b,a \rangle$ and $\langle a,e \rangle$ to retrieve $\Pi^*(b,a|C)$ and $\Pi^*(a,e|C)$, for approximating $\Pi^*(b,e|C)$. (ii) In Figures 3 (d), (e) and (f), given b and d, $\langle b,d \rangle \notin M_{path}.key$, b and d are both keys in M_{end}, and $L_x < L_y$. So, we use the key b with a smaller y-coordinate value to retrieve the value a in M_{end}. Then, in M_{path}, we use $\langle b,a \rangle$ and $\langle a,d \rangle$ to retrieve $\Pi^*(b,a|C)$ and $\Pi^*(a,d|C)$, for approximating $\Pi^*(b,d|C)$).

4.2.4 Proximity Query Algorithms using RC-Oracle. We introduce the key idea of proximity query algorithms using *RC-Oracle*. In Figure 9, given C, a query object q, a set of n' target objects O on C, a value k in *kNN* query and a value r in range query, we can answer *kNN* and range queries using *RC-Oracle*. In the P2P query, the query object is a POI in P, and the target objects are POIs in P. A naive algorithm performs the shortest path query n' times with q as a source and performs a linear scan on the results. Then, it returns all shortest paths passing on C from q to its k

nearest target objects of q or target objects whose distance to q is at most r. We propose an efficient   480
algorithm for it. Intuitively, when constructing these oracles, we have used the *SSAD* algorithm   481
to calculate shortest paths passing on C with q as a source and sorted these paths in ascending   482
order based on their distance in M_{path}. We can use a set of lists to store these sorted paths, where   483
each list stores a set of sorted paths with one POI as the source. For these paths, we do not need   484
to perform linear scans over all of them in proximity queries for time-saving. Then, we give the   485
notation, detail and example (using *kNN* query with $k = 1$).   486

kNN or range query	C, query object q, set of target objects O and value k or r	➧	*Proximity query*	➧	*kNN* or range query path result

Fig. 9. Proximity query algorithm overview

Notation: Let $list_u$, $list_v$, ... be a set of lists, where each list stores a set of sorted paths calculated   487
by *SSAD* algorithm with one POI $u, v, \ldots$ as the source. Let M_{list} be a hash table that stores key-value   488
pairs. Each key-value pair stores an endpoint u as a key, and the corresponding list $list_u$ as a value.   489
u is the endpoint that we use as a source to calculate the exact shortest paths passing on C. $list_u$ 490
is the corresponding sorted paths. In Figure 3 (d), we store $\Pi^*(a, b|C)$, $\Pi^*(a, c|C)$, $\Pi^*(a, d|C)$ and   491
$\Pi^*(a, e|C)$ in order in $list_a$. We store $\Pi^*(b, c|C)$ in $list_b$. We store a as a key, and $list_a$ as a value in   492
M_{list}. We also store b as a key, and $list_b$ as a value in M_{list}.   493
Detail and example: There are two cases.   494
(1) *Approximation needed in direct result return*: If $q \in M_{end}.key$, it means we need to use two   495
paths in M_{path} to approximate some other paths in a later stage, we use the key q to retrieve the   496
value q' in M_{end}, there are two more cases. In Figures 3 (d) and (e), given b as the query object, since   497
$b \in M_{end}.key$, we use the key b to retrieve the value a in M_{end}, there are two more cases.   498
(i) *Linear scan*: For the target objects with a smaller or same x- (resp. y-) coordinate compared   499
with q' when $L_x \geq L_y$ (resp. $L_x < L_y$), we perform a linear scan on the shortest path query result   500
between q and them. We maintain *kNN* or range query result. In Figures 3 (d) and (e), since $L_x < L_y$,   501
the POI with a smaller or same y-coordinate compared with a is $\{a\}$, we perform a linear scan on   502
the shortest path query result between b and $\{a\}$. The *kNN* result stores $\{\Pi^*(a, b|C)\}$.   503
(ii) *Direct result return*: For the target objects (not including q) with a larger x- (resp. y-) coordinate   504
compared with q' when $L_x \geq L_y$ (resp. $L_x < L_y$), there are further more two cases. In Figures 3 (d)   505
and (e), since $L_x < L_y$, the POIs with a larger y-coordinate compared with a are $\{c, d, e\}$, there are   506
further more two cases.   507
- *Direct result return without approximation*: If the endpoint pairs of q and these target objects   508
  are keys in M_{path}, it means that we have used the *SSAD* algorithm with q as a source for such   509
  objects and already sorted such paths in order in $list_q$. We can use q to retrieve $list_q$ in M_{list}. We   510
  maintain *kNN* or range query result. In Figures 3 (d) and (e), since $\langle b, c \rangle \in M_{path}.key$, we know   511
  that $\Pi^*(b, c|C)$ is sorted in order in $list_b$. We use b to retrieve $list_b$ in M_{list}. The sorted path in   512
  $list_b$ is $\Pi^*(b, c|C)$. The *kNN* result stores $\{\Pi^*(a, b|C)\}$.   513
- *Direct result return with approximation*: If the endpoint pairs of q and these target objects are   514
  not keys in M_{path}, it means that we have used the *SSAD* algorithm with q' as a source for such   515
  objects and already sorted such paths in order in $list_{q'}$. We can use q' to retrieve $list_{q'}$ in M_{list}.   516
  We just need to use the exact distance between q' and these target objects plus $|\Pi^*(q', q|C)|$,   517
  to get the approximate distance between q and o. We maintain *kNN* or range query result. In   518
  Figures 3 (d) and (e), since $\langle b, d \rangle \notin M_{path}.key$ and $\langle b, e \rangle \notin M_{path}.key$, we know that $\Pi^*(a, d|C)$ 519
  and $\Pi^*(a, e|C)$ are sorted in order in $list_a$. We use a to retrieve $list_a$ in M_{list}. The sorted paths in   520

$list_a$ are $\Pi^*(a,d|C)$ and $\Pi^*(a,e|C)$. So, $\Pi(b,d|C)$ and $\Pi(b,e|C)$ are also sorted in order. The *kNN* result stores $\{\Pi^*(a,b|C)\}$.

(2) *Approximation not needed in direct result return*: If $q \notin M_{end}.key$, it means we do need to use two paths in M_{path} to approximate all other paths in a later stage, there are two more cases. In Figures 3 (d) and (e), given c as the query object, since $c \notin M_{end}.key$, there are two more cases.

(i) *Linear scan*: For the target objects with a smaller or same x- (resp. y-) coordinate compared with q when $L_x \geq L_y$ (resp. $L_x < L_y$), we perform a linear scan on the shortest path query result between q and them. We maintain *kNN* or range query result. In Figures 3 (d) and (e), since $L_x < L_y$, the POIs with a smaller y-coordinate compared with c are $\{a,b\}$, we perform a linear scan on the shortest path query result between c and $\{a,b\}$. The *kNN* result stores $\{\Pi^*(a,c|C)\}$.

(ii) *Direct result return*: For the target objects with a larger x- (resp. y-) coordinate compared with q when $L_x \geq L_y$ (resp. $L_x < L_y$), we have used the *SSAD* algorithm with q as a source for such objects and already sorted such paths in order in $list_q$. We can use q to retrieve $list_q$ in M_{list}. We maintain *kNN* or range query result. In Figures 3 (d) and (e), since $L_x < L_y$, the POIs with a larger y-coordinate compared with c are $\{d,e\}$, we know that $\Pi^*(c,d|C)$ and $\Pi^*(c,e|C)$ are sorted in order in $list_c$. We use c to retrieve $list_c$ in M_{list}. The sorted paths in $list_c$ are $\Pi^*(c,d|C)$ and $\Pi^*(c,e|C)$. The *kNN* result stores $\{\Pi^*(c,d|C)\}$.

4.2.5 *Theoretical Analysis about RC-Oracle*. We show some theoretical analysis.

(1) **Algorithm *FastFly* and *RC-Oracle***: The analysis of algorithm *FastFly* is in Theorem 4.1, and the analysis of *RC-Oracle* is in Theorem 4.2.

THEOREM 4.1. *The shortest path query time and memory consumption of algorithm FastFly are* $O(N \log N)$ *and* $O(N)$, *respectively. It returns the exact shortest path passing on the point cloud.*

PROOF. Since algorithm *FastFly* is a Dijkstra's algorithm and there are total N points, we obtain the shortest path query time and memory consumption. Since Dijkstra's algorithm returns the exact shortest path, algorithm *FastFly* returns the exact shortest path passing on the point cloud. □

THEOREM 4.2. *The oracle construction time, oracle size and shortest path query time of RC-Oracle are* $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$ *and* $O(1)$, *respectively. It always has* $|\Pi_{RC-Oracle}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ *for any pair of POIs s and t in P.*

PROOF. Firstly, we show the *oracle construction time*. (i) In *POIs sort* step, it needs $O(n \log n)$ time since there are n POIs and we use quick sort. (ii) In *shortest paths calculation* step, it needs $O(\frac{N \log N}{\epsilon} + n)$ time. (a) It uses $O(\frac{1}{\epsilon})$ POIs as sources to run algorithm *FastFly* for exact shortest paths calculation according to standard packing property [24], where each algorithm *FastFly* needs $O(N \log N)$ time. (b) For other $O(n)$ POIs, which are not used as sources to run algorithm *FastFly*, we calculate the Euclidean distance from these POIs to other POIs in $O(1)$ time for shortest paths approximation. (iii) So, the oracle construction time is $O(\frac{N \log N}{\epsilon} + n \log n)$.

Secondly, we show the *oracle size*. (i) For M_{end}, its size is $O(n)$ since there are n POIs. (ii) For M_{path}, its size is $O(\frac{n}{\epsilon})$. (a) We store $O(\frac{n}{\epsilon})$ exact shortest paths passing on C from $O(\frac{1}{\epsilon})$ POIs (that are used as sources to run algorithm *FastFly*) to other $O(n)$ POIs. (b) We also store $O(n)$ exact shortest paths passing on C from $O(n)$ POIs (that are used as sources to run algorithm *FastFly*) to other $O(1)$ POIs. (iii) So, the oracle size is $O(\frac{n}{\epsilon})$.

Thirdly, we show the *shortest path query time*. (i) If $\Pi^*(s,t|C) \in M_{path}$, the shortest path query time is $O(1)$. (ii) If $\Pi^*(s,t|C) \notin M_{path}$, we need to use s to retrieve s' in M_{end} in $O(1)$ time, and use $\langle s, s' \rangle$ and $\langle s', t \rangle$ to retrieve $\Pi^*(s,s'|C)$ and $\Pi^*(s',t|C)$ in M_{path} in $O(1)$ time. The shortest path query time is still $O(1)$. Thus, the shortest path query time of *RC-Oracle* is $O(1)$.

Fourthly, we show the *error bound*. Given a pair of POIs s and t, if $\Pi^*(s, t|C)$ exists in M_{path}, then there is no error. Thus, we only consider the case that $\Pi^*(s, t|C)$ does not exist in M_{path}. Suppose that u is a POI close to s, such that $\Pi_{RC\text{-}Oracle}(s, t|C)$ is calculated by concatenating $\Pi^*(s, u|C)$ and $\Pi^*(u, t|C)$. This means that $\frac{2}{\epsilon} \cdot \Pi^*(u, s|C) < d_E(s, t)$. So, we have $|\Pi^*(s, u|C)| + |\Pi^*(u, t|C)| < |\Pi^*(s, u|C)| + |\Pi^*(u, s|C)| + |\Pi^*(s, t|C)| = |\Pi^*(s, t|C)| + 2 \cdot |\Pi^*(u, s|C)| < |\Pi^*(s, t|C)| + \epsilon \cdot d_E(s, t) \leq |\Pi^*(s, t|C)| + \epsilon \cdot |\Pi^*(s, t|C)| = (1 + \epsilon)|\Pi^*(s, t|C)|$. The first inequality is due to triangle inequality. The second equation is because $|\Pi^*(u, s|C)| = |\Pi^*(s, u|C)|$. The third inequality is because we have $\frac{2}{\epsilon} \cdot \Pi^*(u, s|C) < d_E(s, t)$. The fourth inequality is because the Euclidean distance between two points is no larger than the shortest distance on the point cloud between the same two points. $\quad\square$

(2) **Relationships of the shortest distance on a point cloud, and the shortest surface or network distance on a *TIN***: We show the relationship of $|\Pi^*(s, t|C)|$ with $|\Pi_W(s, t|T)|$ and $|\Pi^*(s, t|T)|$ in Lemma 4.3.

LEMMA 4.3. *Given a pair of points s and t on C, we have (i) $|\Pi^*(s, t|C)| \leq |\Pi_W(s, t|T)|$ and (ii) $|\Pi^*(s, t|C)| \leq k' \cdot |\Pi^*(s, t|T)|$, where $k' = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$.*

PROOF. (i) In Figure 2 (a), given a green point q on C, it can connect with one of its 8 neighbor points (7 blue points and 1 red point s). In Figure 2 (c), given a green vertex q on T, it can only connect with one of its 6 blue neighbor vertices. So, $|\Pi^*(s, t|C)| \leq |\Pi_W(s, t|T)|$. (ii) We let $\Pi_E(s, t|T)$ be the shortest path passing on the edges of T (where these edges belong to the faces that $\Pi^*(s, t|T)$ passes) between s and t. According to left hand side equation in Lemma 2 of study [27], we have $|\Pi_E(s, t|T)| \leq k' \cdot |\Pi^*(s, t|T)|$. Since $\Pi_W(s, t|T)$ considers all the edges on T, $|\Pi_W(s, t|T)| \leq |\Pi_E(s, t|T)|$. Thus, we finish the proof by combining these inequalities. $\quad\square$

(3) **Proximity query algorithms**: We provide analysis on the proximity query algorithms using *RC-Oracle*. For the *kNN* and range queries, both of them return a set of target objects. Given a query object q, we let v_f (resp. v_f') be the furthest returned target object to q calculated using the exact distance on C (resp. the approximated distance on C returned by *RC-Oracle*). We define the error rate of the *kNN* and range queries to be $\frac{|\Pi^*(q, v_f'|C)|}{|\Pi^*(q, v_f|C)|}$, which is a real number no smaller than 1. Then, we show the query time and error rate of *kNN* and range queries using *RC-Oracle* in Theorem 4.4.

THEOREM 4.4. *The query time and error rate of both the kNN and range queries by using RC-Oracle are $O(n')$ and $(1 + \epsilon)$, respectively.*

PROOF SKETCH. The *query time* is due to the usage of the shortest path query phase for n' times in the worst case. The *error rate* is due to its definition and the error of *RC-Oracle*. The detailed proof appears in the appendix. $\quad\square$

4.3 *RC-Oracle-A2P-SmCon* and Its Proximity Query Algorithms

4.3.1 Key Idea of RC-Oracle-A2P-SmCon. We introduce the key idea of the efficient adaptation from *RC-Oracle* to *RC-Oracle-A2P-SmCon* in the A2P query. This makes sure that *RC-Oracle-A2P-SmCon*'s oracle construction time remains the same as *RC-Oracle*, and shortest path query time is smaller than algorithm *FastFly*, i.e., the *SSAD* algorithm. The adaptation is achieved by using the *SSAD* algorithm with the assistance of *RC-Oracle*, such that the *SSAD* algorithm can *terminate earlier*, i.e., similar to the *rapid oracle construction* reason for *RC-Oracle*.

Since *RC-Oracle-A2P-SmCon* has the same construction phase as *RC-Oracle* in Figures 3 (b) - (e), we only illustrate its shortest path query phase with an example. In Figure 5 (a), for a non-POI point f, we first use the *SSAD* algorithm with f as a source, and visit all POIs with the y-coordinate smaller than or equal to f (i.e., a, b, c). Note that in this figure, it seems that the y-coordinate of

c is larger than f in the 3D point cloud. But indeed, their y-coordinates are the same (in 2D). At the same time, before the termination of the *SSAD* algorithm, if we can also visit the POIs with the y-coordinate larger than f, we also calculate shortest paths passing on C between f and these POIs. In Figure 5 (b), we need to find a POI such that we have used this POI as a source in the *SSAD* algorithm, this POI is not a key in M_{end}, and the exact distance on C between f and this POI is the smallest. This POI is a. If f is close to the POI a, and f is far away from e, i.e., $\frac{2}{\epsilon} \cdot |\Pi^*(a, f|C)| < d_E(e, f)$, then we can use $\Pi^*(f, a|C)$ and $\Pi^*(a, e|C)$ to approximate $\Pi^*(f, e|C)$. The first term "if f is close to a" is judged using the previously calculated $|\Pi^*(a, f|C)|$. The second term "if f is far away from e" is judged using the Euclidean distance $d_E(e, f)$. Thus, we just need to continue the previous *SSAD* algorithm with f as a source, and terminate earlier when it has visited d. We store the paths from the *SSAD* algorithm in M_{path}, and store f as key and a as value in M_{end}. Note that the *SSAD* algorithm (i.e., *FastFly*) is a Dijkstra's algorithm, so given a source, after we terminate it, we can save the result of the Dijkstra's algorithm from the memory to the hard disk. If we continue the *SSAD* algorithm with the same source, we can retrieve the previously saved result from the hard disk to the memory, and there is no need to start from scratch for time-saving. In Figure 5 (c), we have the updated M_{path} and M_{end}. In Figure 5 (d), we need to query the shortest path passing on C between e and f. Similar to the shortest path query phase of *RC-Oracle*, since $\langle e, f \rangle \notin M_{path}.key$, f is key in M_{end}, we use the key f to retrieve the value a in M_{end}. Then, in M_{path}, we use $\langle e, a \rangle$ and $\langle a, f \rangle$ to retrieve $\Pi^*(e, a|C)$ and $\Pi^*(a, f|C)$, for approximating $\Pi^*(e, f|C)$.

However, the shortest path query time of simply using algorithm *FastFly* with f as a source without pruning any other destination POIs is large. Our experimental result shows that the shortest path query time for *RC-Oracle-A2P-SmCon* is half of algorithm *FastFly*.

4.3.2 Implementation Details of RC-Oracle-A2P-SmCon (Shortest Path Query Phase).
The construction phase of *RC-Oracle-A2P-SmCon* is the same as *RC-Oracle*. We give its shortest path query phase as follows.

Notation: Given a source q, we re-use the notation $D(q)$ as in the construction phase of *RC-Oracle*, but q is a point on C and elements in $D(q)$ are POIs in P in the shortest path query phase of *RC-Oracle-A2P-SmCon*. Let $P' = P - D(q)$ be a set of POIs that are in P and not in $D(q)$. In Figure 5 (a), $D(f) = \{a, b, c\}$ since we need to use *FastFly* to calculate the exact shortest path from f to a, b, c, and $P' = \{d, e\}$.

Detail and example: Algorithm 2 shows the shortest path query phase of *RC-Oracle-A2P-SmCon* in detail, and the following illustrates it with an example.

(1) *New shortest paths calculation* (lines 1-18): In Figure 5 (a), given f as a source that is not a POI, and e as a destination that is a POI, there are five steps. If both source and destination are POIs, we can skip these five steps.

(i) *Smaller x- or y-coordinate POIs exact shortest paths calculation* (lines 3-6): In Figure 5 (a), since $L_x < L_y$, we have $D(f) = \{a, b, c\}$. We calculate the exact shortest paths passing on C from f to a, b, c (in orange lines) using algorithm *FastFly*.

(ii) *Approximate POI selection and destination POIs update* (lines 7-10): In Figure 5 (a), we have a as the POI, since the exact distance on C between f and a is the smallest, and $a \notin M_{end}.key = \{b, d\}$. During the execution of algorithm *FastFly*, if we can also visit the POIs with the y-coordinate larger than f, we also calculate shortest paths passing on C between f and these POIs, and update $D(f)$ to cover those POIs. In this figure, there are no such POIs and we do not need to $D(f)$. So, $D(f) = \{a, b, c\}$ and $P' = \{d, e\}$.

(iii) *Far away POIs selection* (lines 11-13): In Figure 5 (b), $\frac{2}{\epsilon} \cdot |\Pi^*(a, f|C)| < d_E(e, f)$ and $f \notin M_{end}.key$, it means e is far away from f. So, we can use $\Pi^*(f, a|C)$ and $\Pi^*(a, e|C)$ to approximate

---

Algorithm 2 *RC-Oracle-A2P-SmCon-Query* $(C, P, s, t, M_{path}, M_{end}, \epsilon)$

Input: A point cloud C. A set of POIs P. A source s that is any point on C. A destination t that is a POI in P. A path map table M_{path}. An endpoint map table M_{end}. An error parameter ϵ.

Output: An updated path map table M_{path}. An updated endpoint map table M_{end}. The shortest path $\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(s, t|C)$ between s and t passing on C.

1: **if** $\langle s, t \rangle \notin M_{path}.key$ and $s \notin M_{end}.key$ and $t \notin M_{end}.key$ **then**
2:     $D(s) \leftarrow \emptyset$
3:     **if** $L_x \geq L_y$ (resp. $L_x < L_y$) **then**
4:         **for** each POI $u \in P$ such that the x-coordinate (resp. y-coordinate) of u is smaller than or equal to s **do**
5:             $D(s) \leftarrow D(s) \cup \{u\}$
6:     Calculate the exact shortest paths passing on C from s to each POI in $D(s)$ simultaneously using algorithm *FastFly* and store the result of the algorithm from the memory to the hard disk
7:     $u \leftarrow$ the POI in $D(s)$ such that the exact distance on C between s and u, i.e., $|\Pi^*(s, u|C)|$, is the smallest and $u \notin M_{end}.key$
8:     **for** each POI $v \in P'$ such that v is also visited during the execution of algorithm *FastFly* **do**
9:         $D(s) \leftarrow D(s) \cup \{v\}$
10:     $P' \leftarrow P - D(s)$
11:     **for** each POI $v \in P'$ **do**
12:         **if** $\frac{2}{\epsilon} \cdot |\Pi^*(u, s|C)| < d_E(s, v)$ and $s \notin M_{end}.key$ **then**
13:             $key \leftarrow s$, $value \leftarrow u$, $M_{end} \leftarrow M_{end} \cup \{key, value\}$
14:         **else if** $\frac{2}{\epsilon} \cdot |\Pi^*(u, s|C)| \geq d_E(s, v)$ **then**
15:             $D(s) \leftarrow D(s) \cup \{v\}$
16:     Continue the previous algorithm *FastFly* with s as a source by retrieving the previously saved result from the hard disk to the memory, to calculate the exact shortest paths passing on C from s to each POI in $D(s)$ simultaneously
17:     **for** each POI $v \in D(s)$ **do**
18:         $key \leftarrow \langle s, v \rangle$, $value \leftarrow \Pi^*(s, v|C)$, $M_{path} \leftarrow M_{path} \cup \{key, value\}$
19: Use the same shortest path query phase of *RC-Oracle* to retrieve $\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(s, t|C)$
20: **return** M_{path}, M_{end} and $\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(s, t|C)$

$\Pi^*(f, e|C)$. We get $\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(f, e|C)$ by concatenating $\Pi^*(f, a|C)$ and $\Pi^*(a, e|C)$. We store f as key and a as value in M_{end}.

(iv) *Close POIs selection* (line 11 and lines 14-15): In Figure 5 (b), $\frac{2}{\epsilon} \cdot |\Pi^*(a, f|C)| \geq d_E(d, f)$, it means d is close to f. So, we cannot use any existing exact shortest paths passing on C to approximate $\Pi^*(f, d|C)$. We store d into $D(f)$.

(v) *Selected exact shortest paths calculation* (lines 16-18): In Figure 5 (b), when we have processed all POIs in P' with f as a source, we have $D(f) = \{a, b, c, d\}$. We continue the previous algorithm *FastFly* with f as a source to calculate the exact shortest path passing on C from f to each POI in $D(f)$ (in orange lines). We store each endpoint pair as a key and the corresponding path as a value in M_{path}. Since we terminate the previous algorithm *FastFly* when it visits a, b, c, there is no need to start from scratch for time-saving. Note that we can terminate algorithm *FastFly* earlier since we just need to visit POIs that are close to f, and we do not need to visit e.

(2) *Shortest path query* (line 19): In Figure 5 (d), given f as a source that is not a POI, and e as a destination that is a POI, we use the same shortest path query phase of *RC-Oracle* to query $\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(f, e|C)$. That is, $\langle f, e \rangle \notin M_{path}.key$, and f is a key in M_{end}. So, we use the key f to retrieve the value a in M_{end}. Then, in M_{path}, we use $\langle f, a \rangle$ and $\langle a, e \rangle$ to retrieve $\Pi^*(f, a|C)$ and $\Pi^*(a, e|C)$, for approximating $\Pi^*(f, e|C)$.

4.3.3 ***Proximity Query Algorithms using RC-Oracle-A2P-SmCon***. Recall that there is a query object q and a set of n' target objects O on C. In the A2P query, there are two cases. (1) The query object is any point on C, and the target objects are POIs in P. (2) The query object is a POI in P, and the target objects are any point on C. For both cases, the proximity query algorithms using *RC-Oracle-A2P-SmCon* are similar to using *RC-Oracle*. We just need to use Algorithm 2 lines 1-18 to calculate new shortest paths for q if needed, and then perform similarly as in *RC-Oracle*. The only difference between the two cases is as follows. For the first case, we do not need to perform linear scans for paths calculated using algorithm *FastFly* in the shortest path query phase of *RC-Oracle-A2P-SmCon*. Since they are already sorted in ascending order with q as the source. For the remaining paths in the first case and all paths in the second case, we perform linear scans on them for proximity queries.

4.3.4 ***Theoretical Analysis about RC-Oracle-A2P-SmCon***. We show some theoretical analysis. (1) **RC-Oracle-A2P-SmCon**: The analysis of *RC-Oracle-A2P-SmCon* is in Theorem 4.5.

THEOREM 4.5. *The oracle construction time, oracle size and shortest path query time of RC-Oracle-A2P-SmCon are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$ and $O(N \log N)$, respectively. It always has $|\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmCon}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any point s on C and any POI t in P.*

PROOF. Compared with *RC-Oracle*, in the shortest path query time, we change $O(1)$ to $O(N \log N)$, since *RC-Oracle-A2P-SmCon* uses algorithm *FastFly*. In the error bound, we change "any pair of POIs s and t in P" to "any point s on C and any POI t in P". The others are the same as *RC-Oracle*.   □

(2) **Proximity query algorithms**: We show the query time and error rate of *kNN* and range queries using *RC-Oracle-A2P-SmCon* in Theorem 4.6.

THEOREM 4.6. *The query time and error rate of both the kNN and range queries by using RC-Oracle-A2P-SmCon are $O(N \log N + n')$ and $(1+\epsilon)$, respectively.*

PROOF SKETCH. The *query time* is due to the usage of the new shortest paths calculation step in $O(N \log N)$ time for only once, and the usage of the shortest path query step for n' times in the worst case. The *error rate* is due to its definition and the error of *RC-Oracle-A2P-SmCon*.   □

4.4 *RC-Oracle-A2P-SmQue* and Its Proximity Query Algorithms

4.4.1 ***Key Idea of RC-Oracle-A2P-SmQue***. We introduce the key idea of the efficient adaptation from *RC-Oracle* to *RC-Oracle-A2P-SmQue* in the A2P query. This makes sure that *RC-Oracle-A2P-SmQue*'s oracle construction time will not increase a lot, and shortest path query time remains the same as *RC-Oracle*. We still regard each POI as a source and use algorithm *FastFly* for n times. The only difference from *RC-Oracle* is that, in *RC-Oracle-A2P-SmQue*, the destinations are not POIs in P, but all points on C. We do not use any point on C as a source and use algorithm *FastFly* for N times, and set destinations to be POIs in P. This will be very slow compared with *RC-Oracle-A2P-SmQue*.

4.4.2 ***Proximity Query Algorithms using RC-Oracle-A2P-SmQue***. Recall that there is a query object q and a set of n' target objects O on C, and there are two cases in the A2P query. For both cases, the proximity query algorithms using *RC-Oracle-A2P-SmQue* are similar to using *RC-Oracle*. The only difference between the two cases is as follows. For the second case, we do not need to perform linear scans for paths calculated using algorithm *FastFly* in the shortest path query phase of *RC-Oracle-A2P-SmQue*. Since they are already sorted in ascending order with q as the source. For the remaining paths in the second case and all paths in the first case, we perform linear scans on them for proximity queries.

4.4.3 ***Theoretical Analysis about RC-Oracle-A2P-SmQue***. We show some theoretical analysis.

(1) **RC-Oracle-A2P-SmQue**: The analysis of *RC-Oracle-A2P-SmQue* is in Theorem 4.7.

THEOREM 4.7. *The oracle construction time, oracle size and shortest path query time of RC-Oracle-A2P-SmQue are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{N}{\epsilon})$ and $O(1)$, respectively. It always have $|\Pi_{RC\text{-}Oracle\text{-}A2P\text{-}SmQue}(s, t|C)| \leq (1+\epsilon)|\Pi^*(s, t|C)|$ for any point s on C and any POI t in P.*

PROOF. Compared with *RC-Oracle*, in the oracle size, we change n to N since *RC-Oracle-A2P-SmQue* regards all points on C as possible destinations during oracle construction. In the error bound, we change "any pair of POIs s and t in P" to "any point s on C and any POI t in P". The others are the same as *RC-Oracle*. □

(2) **Proximity query algorithms**: We show the query time and error rate of *kNN* and range queries using *RC-Oracle-A2P-SmQue* in Theorem 4.8.

THEOREM 4.8. *The query time and error rate of both the kNN and range queries by using RC-Oracle-A2P-SmQue are $O(n')$ and $(1+\epsilon)$, respectively.*

PROOF SKETCH. The proof is the same as *RC-Oracle*. □

4.5 *RC-Oracle-A2A* and Its Proximity Query Algorithms

4.5.1 ***Key Idea of RC-Oracle-A2A***. We introduce the key idea of the efficient adaptation from *RC-Oracle* to *RC-Oracle-A2A*. We just need to create POIs that have the same coordinate values as all points on C.

4.5.2 ***Proximity Query Algorithms using RC-Oracle-A2A***. Recall that there is a query object q and a set of n' target objects O on C. In the A2A query, the query object is any point on C, and the target objects are any point on C. Our proximity query algorithms using *RC-Oracle-A2A* are similar to using *RC-Oracle*.

4.5.3 ***Theoretical Analysis about RC-Oracle-A2A***. We show some theoretical analysis.

(1) **RC-Oracle-A2A**: The analysis of *RC-Oracle-A2A* is in Theorem 4.9.

THEOREM 4.9. *The oracle construction time, oracle size and shortest path query time of RC-Oracle-A2A are $O(\frac{N \log N}{\epsilon})$, $O(\frac{N}{\epsilon})$ and $O(1)$, respectively. It always has $|\Pi_{RC\text{-}Oracle\text{-}A2A}(s, t|C)| \leq (1+\epsilon)|\Pi^*(s, t|C)|$ for any pair of points s and t on C.*

PROOF. Compared with *RC-Oracle*, in the oracle construction time and oracle size, we change n to N since *RC-Oracle-A2A* creates POIs that have the same coordinate values as all points on the point cloud. In the error bound, we change "any pair of POIs s and t in P" to "any pair of points s and t on C". The others are the same as *RC-Oracle*. □

(2) **Proximity query algorithms**: We show the query time and error rate of *kNN* and range queries using *RC-Oracle-A2A* in Theorem 4.10.

THEOREM 4.10. *The query time and error rate of both the kNN and range queries by using RC-Oracle-A2A are $O(n')$ and $(1+\epsilon)$, respectively.*

PROOF SKETCH. The proof is the same as *RC-Oracle*. □

4.6 Adaptation to AR2P and AR2AR Queries on *TIN*s

We can adapt our oracles for A2P and A2A queries on point clouds to AR2P and AR2AR queries on *TIN*s. Apart from converting the given *TIN*s to point clouds (the vertices of the *TIN* correspond to the points of the point cloud), we need one more step. This step addresses the more general case of AR2P and AR2AR queries compared with A2P and A2A queries, i.e., the source or destination may lie on the face of a *TIN* but not only the point of a point cloud. We denote *RC-Oracle-Adapt-AR2AR* to be the adapted point cloud oracle of *RC-Oracle-A2A* for AR2AR queries on a *TIN*, and use it for illustration (all other oracles are similar). Specifically, if both source s and destination t lie on faces of the *TIN*, we denote the set of three vertices of the faces that s and t lie in to be V_s and V_t, respectively. Let $\Pi_{RC\text{-}Oracle\text{-}Adapt\text{-}AR2AR}(s, t|T)$ be the calculated shortest path of *RC-Oracle-Adapt-AR2AR* passing on a *TIN* T between s and t. It is calculated by concatenating the line segment (s, u), the path between two vertices u and v returned by *RC-Oracle-Adapt-AR2AR*, and the line segment (v, t), such that $|\Pi_{RC\text{-}Oracle\text{-}Adapt\text{-}AR2AR}(s, t|T)| = \min_{\forall u \in V_s, v \in V_t} [|(s, u)| + |\Pi_{RC\text{-}Oracle\text{-}Adapt\text{-}AR2AR}(u, v|T)| + |(v, t)|]$. Note that $|(s, u)|$ and $|(v, t)|$ are distances of (s, u) and (v, t), respectively. The case that s or t lie on the vertices of the *TIN* is similar but simpler than this. All the theoretical analysis of *RC-Oracle-Adapt-AR2AR* is the same as *RC-Oracle-A2A*. The details proof of the error bound regarding path concatenation appears in the appendix.

5 TI-ORACLE AND ITS ADAPTATIONS

We illustrate *TI-Oracle*. In Figure 10 (a), we have a point cloud C, a set of POIs P and an error parameter ϵ. In Figures 10 (b) - (h), we divide the points into several regions (similar to Voronoi cells [17] in a Voronoi diagram) based on POIs. We construct *TI-Oracle* by calculating two types of shortest paths. The first type is *intra-paths* (in green lines of Figure 10 (d)) from POIs to points in the same region. The second type is *inter-paths* (in blue lines of Figure 10 (g)) among intersection points of these regions and the point cloud. In Figures 10 (i) - (k), for a point k that is not a POI, we calculate *intra-paths* (in orange lines of Figure 10 (i)) from k to points in the same region. Then, we calculate the shortest path between k and another POI using intra-paths and inter-paths stored in *TI-Oracle*.

5.1 Overview of *TI-Oracle* and Its Adaptations

We introduce the two concepts, six components and two phases.

5.1.1 Concepts of *TI-Oracle* and Its Adaptations.
There are two concepts. We use *TI-Oracle* as an example to illustrate.

(1) **The partition cell** is a region that contains a set of points on the point cloud. Before we give more details, we introduce the Voronoi diagram and the Voronoi cell [17]. Given a space and a set of POIs, a Voronoi diagram partitions the space into a set of disjoint Voronoi cells using the POIs. Similarly, given a point cloud C and a set of POIs in P, we partition C into a set of partition cells using P. The boundary of each partition cell lies on the edges of the point cloud graph of C. If a point lies inside but does not lie on the boundary of a partition cell, we say that this point *belongs to* this partition cell. In Figures 10 (b) and (c), given C and P, we use the process of constructing Voronoi cells to obtain the partition cells of C based on P. In Figure 10 (d), there are five partition cells in different colors. f belongs to the partition cell corresponding to a, but g and h do not belong to the partition cell corresponding to a (since they lie on the boundary of this partition cell).

(2) **The boundary point** is an intersection point between C and partition cell's boundary. Given C and P, we obtain a set of boundary points B. In Figure 10 (d), g, h, i are three boundary points.
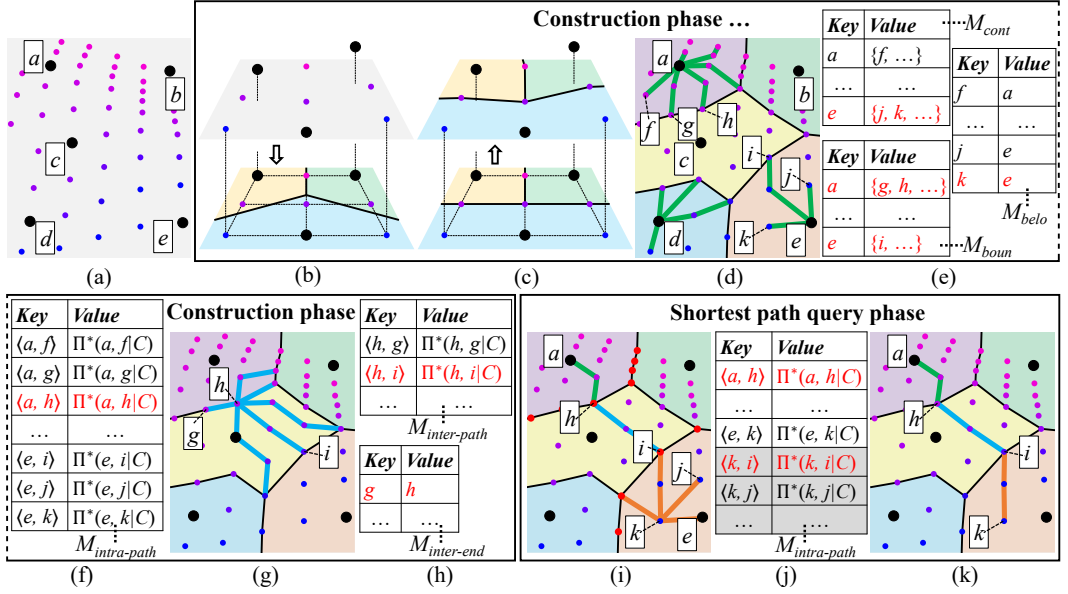
Fig. 10. *TI-Oracle* framework details

*5.1.2 **Components of TI-Oracle and Its Adaptations***. There are six components. All of them
are *hash tables* that store key-value pairs.

(1) **The containing point map table** M_{cont}: Each key-value pair stores an endpoint u as a key,
and a set of points $\{v_1, v_2, \dots\}$ as a value. The endpoint is a POI in P or any point on C, depending
on the oracle A2P or A2A query types. u is used for creating the partition cell, and $\{v_1, v_2, \dots\}$ are
the points on C except u that belong to the partition cell corresponding to u. In Figure 10 (d), f is a
point on C that belongs to the partition cell corresponding to a, so we store a as a key and $\{f, \dots\}$
as a value in M_{cont} in Figure 10 (e). Similarly, we store e as a key and $\{j, k, \dots\}$ as a value in M_{cont}.

(2) **The boundary point map table** M_{boun}: Each key-value pair stores an endpoint u as a key,
and a set of boundary points $\{v_1, v_2, \dots\}$ as a value. u is used for creating the partition cell, and
$\{v_1, v_2, \dots\}$ are the boundary points of the partition cell corresponding to u. In Figure 10 (d), g, h are
two boundary points of the partition cell corresponding to a, so we store a as a key and $\{g, h, \dots\}$
as a value in M_{boun} in Figure 10 (e). Similarly, we store e as a key and $\{i, \dots\}$ as a value in M_{boun}.

(3) **The belonging point map table** M_{belo}: Each key-value pair stores a point u on C as a key
and another endpoint v as a value. v is used for creating the partition cell, and u belongs to the
partition cell corresponding to v. In Figure 10 (d), f belongs to the partition cell corresponding to a,
so we store f as a key and a as a value in M_{belo} in Figure 10 (e). Similarly, we store j as a key and e
as a value, and k as a key and e as a value in M_{belo}.

(4) **The intra-path map table** $M_{intra-path}$: Consider an endpoint u and a point v on C, and v
belongs to the partition cell corresponding to u or v is the boundary point of the partition cell
corresponding to u. We call the exact shortest path passing on C between u and v as the *intra-path*.
In $M_{intra-path}$, each key-value pair stores u and v, as a key $\langle u, v \rangle$, and the corresponding intra-path
$\Pi^*(u, v | C)$, as a value. In Figure 10 (d), there are 6, 4 and 3 intra-paths in green lines with a, d and e
as a source, respectively, and they are stored in $M_{intra-path}$ in Figure 10 (f). For intra-paths between
a and f, we store $\langle a, f \rangle$ as a key and $\Pi^*(a, f | C)$ as a value in $M_{intra-path}$. Similarly, we store $\langle a, g \rangle$ as

a key and $\Pi^*(a, g|C)$ as a value in $M_{intra\text{-}path}$. In Figure 10 (i), there are 3 intra-paths in orange lines with k as a source, where k is not a POI.

(5) **The inter-path map table** $M_{inter\text{-}path}$: Consider a pair of boundary points u and v. We call a path passing on C between u and v as the *inter-path*, and denote it as $\Pi_{inter\text{-}path}(u, v|C)$. We call the exact shortest path passing on C between u and v as the *exact inter-path*. In $M_{inter\text{-}path}$, each key-value pair stores u and v, as a key $\langle u, v \rangle$, and the corresponding exact inter-path $\Pi^*(u, v|C)$, as a value. By regarding all the boundary points as POIs in *RC-Oracle*, $M_{inter\text{-}path}$ in *TI-Oracle* and *TI-Oracle-A2A* is the same as M_{path} in *RC-Oracle*. In Figure 10 (g), $\Pi_{inter\text{-}path}(h, i|C)$ is the same as $\Pi^*(h, i|C)$, and $\Pi_{inter\text{-}path}(g, i|C)$ is approximated by $\Pi^*(g, h|C)$ and $\Pi^*(h, i|C)$. There are 6 exact inter-paths in blue lines, and they are stored in $M_{inter\text{-}path}$ in Figure 10 (h). For the exact inter-paths between h and g, we store $\langle h, g \rangle$ as a key and $\Pi^*(h, g|C)$ as a value in $M_{inter\text{-}path}$. Similarly, we store $\langle h, i \rangle$ as a key and $\Pi^*(h, i|C)$ as a value in $M_{inter\text{-}path}$.

(6) **The inter-endpoint map table** $M_{inter\text{-}end}$: Each key-value pair stores a boundary point u as a key and another boundary point v as a value. By regarding all the boundary points as POIs in *RC-Oracle*, $M_{inter\text{-}end}$ in *TI-Oracle* and *TI-Oracle-A2A* is the same as M_{end} in *RC-Oracle*. In Figure 10 (g), g is close to h, we concatenate $\Pi^*(g, h|C)$ and the exact shortest paths passing on C with h as a source, to approximate shortest paths passing on C with g as a source. So, we store g as a key, and h as a value in $M_{inter\text{-}end}$ in Figure 10 (h).

5.1.3 *Phases of TI-Oracle and Its Adaptations*. There are two phases.

(1) *TI-Oracle* (see Figures 10 and 11): (i) In the construction phase, given C, P and ϵ, we divide C into several partition cells based on P, to obtain a set of boundary points B. We store the points that belong to each partition cell in M_{cont} and M_{belo}, and store the boundary points of each partition cell in M_{boun}. We pre-compute the exact shortest paths passing on C between each POI and (a) each point that belongs to the partition cell generated by this POI, and (b) each boundary point of the same partition cell. We store these calculated intra-paths in $M_{intra\text{-}path}$. We pre-compute the exact shortest paths passing on C between some selected pairs of boundary points (by regarding boundary points as POIs in *RC-Oracle*). We store these calculated exact inter-paths in $M_{inter\text{-}path}$, and store the non-selected boundary points and their corresponding selected boundary points in $M_{inter\text{-}end}$. (ii) In the shortest path query phase, given any point s on C and a POI in P, we calculate the exact shortest paths passing on C between s and (a) each point that belongs to the partition cell that s lies inside, and (b) each boundary point of the same partition cell. We store the calculated paths in $M_{intra\text{-}path}$. Then, given M_{cont}, M_{boun}, M_{belo}, $M_{intra\text{-}path}$, $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$, we answer the path results between s and this POI.

Construction phase	Point cloud C, set of POIs P and error parameter ε ➧	*Partition cells calculation* ➧	
	➧ Set of boundary points B, containing point map table M_{cont}, boundary point map table M_{boun} and belonging point map table M_{belo} ➧	*Intra and inter path calculation* ➧	Intra-path map table $M_{intra\text{-}path}$, inter-path map table $M_{inter\text{-}path}$ and inter-endpoint map table $M_{inter\text{-}end}$
Shortest path query phase	C, P, any point s on C and a POI t in P, B, M_{cont}, M_{boun}, M_{belo}, $M_{intra\text{-}path}$, $M_{intra\text{-}path}$ and $M_{inter\text{-}end}$ ➧	*Intra path calculation* ➧	
	➧ Updated $M_{intra\text{-}path}$ ➧	*Path query* ➧	Shortest path $\Pi_{TI\text{-}Oracle}(s, t|C)$

Fig. 11. *TI-Oracle* framework overview

(2) *TI-Oracle-A2A*: (i) In the construction phase, given C and ϵ, the procedure is similar to *TI-Oracle*. The only difference is that no POI is given as input, we need to randomly select some points

on the point cloud as POIs to construct the partition cells. (ii) In the shortest path query phase, given any pair of points s and t on C, the procedure is similar to *TI-Oracle*, the only difference is that we perform the same query twice for both s and t.

5.2 *TI-Oracle* and Its Proximity Query Algorithms

*5.2.1 **Key Idea of TI-Oracle***. We introduce the key idea of the construction of partition cells of *TI-Oracle*, the key idea of the small oracle construction time and small oracle size of *TI-Oracle*, and the key idea of shortest path query of *TI-Oracle* as follows.

(1) **Construction of partition cells**: In Figure 10 (a), we have C and P. In Figure 10 (b), we project C in the 2D plane. We build a Voronoi diagram in Euclidean space using the grid-based 2D point cloud and POIs with the sweep line algorithm [17] in $O(n \log n)$ time, and obtain a set of Voronoi cells. In Figure 10 (c), we obtain a set of partition cells in the 2D plane by correcting the boundary of each Voronoi cell, such that the boundary of each partition cell lies on edges of the point cloud graph of C. Then, we project the partition cells in the 2D plane back to the 3D space to obtain the partition cells of C.

We discuss more about the boundary correction step. In Figure 12 (a), we have a part of the Voronoi diagram with three Voronoi cells and a point cloud graph (without the diagonal edges) of C in the 2D plane. We have some intersection points between the boundary of the 2D Voronoi cells and the 2D point cloud graph, e.g., the blue points. We move each point to one of the two closest points on C of the edge that this intersection point lies on, i.e., following the red arrows. We also have some intersection points among the boundary of the 2D Voronoi cells, e.g., the green point. We move each point to one of the four closest points on C of the square that this intersection point lies on, i.e., following the orange arrow. In Figure 12 (b), we connect these intersection points, to form the boundary (in pink lines) of partition cells in the 2D plane.

(a)

⇓

(b)
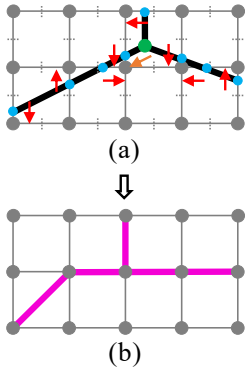
Fig. 12. Boundary correction of the 2D partition cells

Table 4. Point cloud datasets

Name	$|N|$
Original dataset	
BearHead (BH_c) [3, 44, 45]	0.5M
EaglePeak (EP_c) [3, 44, 45]	0.5M
GunnisonForest (GF_c) [6]	0.5M
LaramieMount (LM_c) [7]	0.5M
RobinsonMount (RM_c) [11]	0.5M
Small-version dataset	
BH_c-small	10k
EP_c-small	10k
GF_c-small	10k
LM_c-small	10k
RM_c-small	10k
Multi-resolution dataset	
BH_c multi-resolution	1M, 1.5M, 2M, 2.5M
EP_c multi-resolution	1M, 1.5M, 2M, 2.5M
GF_c multi-resolution	1M, 1.5M, 2M, 2.5M
LM_c multi-resolution	1M, 1.5M, 2M, 2.5M
RM_c multi-resolution	1M, 1.5M, 2M, 2.5M
EP_c-small multi-resolution	20k, 30k, 40k, 50k

(2) **Small oracle construction time**: We give the reason why *TI-Oracle* has a small oracle construction time. It is due to the *tight shortest paths result* of *TI-Oracle*. During construction, we only calculate intra-paths and exact intra-paths by algorithm *FastFly*, i.e., the *SSAD* algorithm. We can terminate it earlier for both types of paths.

We use an example for illustration. In Figure 10 (a), we have a set of POIs a, b, c, d, e. In Figures 10 (b) and (c), we construct partition cells. In Figures 10 (d) - (f), by using the partition cells, we store the corresponding information in M_{cont}, M_{boun} and M_{belo}. We regard each POI as a source and use the *SSAD* algorithm to calculate intra-paths in green lines. We terminate earlier when each has visited all the points that belong to the partition cell generated by each POI, and the boundary points of the same partition cell. We use the POI to retrieve such points in M_{cont} and M_{boun}. We store intra-paths in $M_{intra\text{-}path}$. In Figures 10 (g) and (h), we regard boundary points as POIs in *RC-Oracle* to calculate exact inter-paths in blue lines. Due to the *rapid oracle construction* advantage of *RC-Oracle*, we can also terminate the corresponding *SSAD* algorithm earlier. We store exact inter-paths in $M_{inter\text{-}path}$, and the corresponding boundary points in $M_{inter\text{-}end}$. No matter whether the density of POIs is high or low (i.e., POIs are close to or far away from each other), boundary points are always close to each other. So, we can always utilize the *rapid oracle construction* advantage of *RC-Oracle*. We can use the previously calculated shortest paths to approximate other shortest paths, to terminate the *SSAD* algorithm with each boundary point as a source earlier.

However, in *RC-Oracle-A2A*, it needs to use the *SSAD* algorithm with each point on C as a source to cover all other points on C, which results in a large oracle construction time. In addition, in *RC-Oracle-A2P-SmQue*, when the density of POIs is low (i.e., POIs are far away from each other), it is difficult to utilize the *rapid oracle construction* advantage of *RC-Oracle*. Since it is difficult to use the previously calculated shortest paths to approximate other shortest paths, and difficult to terminate the *SSAD* algorithm with each POI as a source earlier.

(3) **Small oracle size**: We give the reason why *TI-Oracle* has a small oracle size. We only store a small number of intra-paths and inter-paths in *TI-Oracle*, i.e., we do not store the paths between any pair of points on C. In Figure 10 (d), we only store intra-paths in green lines. In Figure 10 (g), for a pair of boundary points g and i, we use the exact inter-paths $\Pi^*(g, h|C)$ and $\Pi^*(h, i|C)$ in blue lines to approximate $\Pi^*(g, i|C)$. We will not store $\Pi^*(g, i|C)$ in $M_{inter\text{-}path}$ for memory saving.

(4) **Shortest path query**: We use an example to illustrate the shortest path query phase of *TI-Oracle*. In Figures 10 (i) - (j), given a source point k that is not a POI, we use the *SSAD* algorithm with k as a source, to calculate intra-paths in orange lines. We can terminate earlier when it has visited all the points that belong to the partition cell that k lies inside and the boundary points of the same partition cell. We store intra-paths in $M_{intra\text{-}path}$. Given a destination POI, there are two cases. (i) If the destination POI is e such that $\langle e, k \rangle \in M_{intra\text{-}path}.key$, we can directly return $\Pi^*(e, k|C)$. (ii) If the destination POI is a such that $\langle a, k \rangle \notin M_{intra\text{-}path}.key$, we have the following. We let B_s and B_t be two sets of boundary points of the partition cells that source s and destination t belong to, respectively. We can use $s = k$ to retrieve e in M_{belo}, and then use e to retrieve $B_s = \{i, \dots\}$ in M_{boun}. We can use $t = a$ to retrieve $B_t = \{h, \dots\}$ in M_{boun}. B_s and B_t are red points around k and a. We calculate the result by concatenating $\Pi^*(k, k'|C)$, $\Pi_{inter\text{-}path}(k', a'|C)$ and $\Pi^*(a', a|C)$ such that $|\Pi^*(k, k'|C)| + |\Pi_{inter\text{-}path}(k', a'|C)| + |\Pi^*(a', a|C)|$ is the smallest for any $k' \in B_s$ and $a' \in B_t$. In this case, $k' = i$ and $a' = h$. We can use $\langle i, k \rangle$ and $\langle a, h \rangle$ to retrieve $\Pi^*(i, k|C)$ and $\Pi^*(a, h|C)$ in $M_{intra\text{-}path}$, respectively. We can use the shortest path query phase in *RC-Oracle*, h, i, $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$ to calculate $\Pi_{inter\text{-}path}(h, i|C) = \Pi^*(h, i|C)$.

5.2.2 *Implementation Details of TI-Oracle (Construction Phase)*. We give the construction phase of *TI-Oracle*.

Notation: Given a source q, we re-use the notation $D(q)$ as in the construction phase of *RC-Oracle*, but q is a POI in P and elements in $D(q)$ are points on C in the construction phase of *TI-Oracle*.

Detail and example: Algorithm 3 shows the construction phase of *TI-Oracle* in detail, and the following illustrates it with an example.

Algorithm 3 *TI-Oracle-Construction* (C, P, ϵ)

Input: A point cloud C. A set of POIs P. An error parameter ϵ.
Output: A set of boundary points B. A containing point map table M_{cont}. A boundary point map table M_{boun}. A belonging point map table M_{belo}. An intra-path map table $M_{intra\text{-}path}$. An inter-path map table $M_{inter\text{-}path}$. An inter-endpoint map table $M_{inter\text{-}end}$.
1: Project C in the 2D plane, and build a Voronoi diagram in Euclidean space using the grid-based 2D point cloud and P with the sweep line algorithm to generate a set of Voronoi cells
2: Project the partition cells in the 2D plane back to the 3D space to obtain the partition cells of C
3: $B \leftarrow$ the boundary points, $M_{cont} \leftarrow \emptyset, M_{boun} \leftarrow \emptyset, M_{belo} \leftarrow \emptyset, M_{intra\text{-}path} \leftarrow \emptyset, M_{inter\text{-}path} \leftarrow \emptyset, M_{inter\text{-}end} \leftarrow \emptyset, M_{cout} \leftarrow \emptyset$
4: **for** each POI $u \in P$ **do**
5:     $value_1 \leftarrow \emptyset, value_2 \leftarrow \emptyset$
6:     **for** each point v on C **do**
7:         **if** v is a point that belongs to the partition cell corresponding to u **then**
8:             $value_1 \leftarrow value_1 \cup \{v\}$
9:         **if** v is a boundary point of the partition cell corresponding to u **then**
10:            $value_2 \leftarrow value_2 \cup \{v\}$
11:     $key \leftarrow u, M_{cont} \leftarrow M_{cont} \cup \{key, value_1\}, M_{boun} \leftarrow M_{boun} \cup \{key, value_2\}$
12: **for** each point u on C such that $u \notin B$ **do**
13:     **for** each POI $v \in P$ such that u belongs to the partition cell corresponding to v **do**
14:         $key \leftarrow u, value \leftarrow v, M_{belo} \leftarrow M_{belo} \cup \{key, value\}$
15: **for** each POI $u \in P$ **do**
16:     $D(u) \leftarrow$ retrieved from M_{cont} using u as key $\cup$ retrieved from M_{boun} using u as key
17:     calculate the exact shortest paths passing on C from u to each point in $D(u)$ simultaneously using algorithm *FastFly*
18:     **for** each point $v \in D(u)$ **do**
19:         $key \leftarrow \langle u, v \rangle, value \leftarrow \Pi^*(u, v | C), M_{intra\text{-}path} \leftarrow M_{intra\text{-}path} \cup \{key, value\}$
20: $\{M_{inter\text{-}path}, M_{inter\text{-}end}\} \leftarrow$ *RC-Oracle-Construction* (C, B)
21: **return** $B, M_{cont}, M_{boun}, M_{belo}, M_{intra\text{-}path}, M_{inter\text{-}path}$ and $M_{inter\text{-}end}$

---

(1) *Partition cells calculation and initialization* (lines 1-3): In Figures 10 (b) and (c), we obtain a set of partition cells. In Figure 10 (d), there are five partition cells in different colors.

(2) M_{cont}, M_{boun} and M_{belo} *calculation* (lines 4-14): In Figure 10 (d), f is a point (resp. j, k are two points) on C that belongs to the partition cell corresponding to a (resp. e). g, h are two boundary points (resp. i is a boundary point) of the partition cell corresponding to a (resp. e). f belongs to (resp. j, k belong to) the partition cell corresponding to a (resp. e). So, we have M_{cont}, M_{boun} and M_{belo} in Figure 10 (e).

(3) *Intra-paths calculation* (lines 15-19): In Figure 10 (d), for POI a, $D(a) = \{f, g, h, \dots\}$. For each POI a, b, c, d, e, we calculate intra-paths in green lines using algorithm *FastFly*, and store the key-value pairs in $M_{intra\text{-}path}$ in Figure 10 (f).

(4) *Inter-paths calculation* (line 20): In Figure 10 (g), we regard boundary points as POIs in *RC-Oracle* to calculate inter-paths in blue lines, and store the output in $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$ in Figure 10 (h).

5.2.3   ***Implementation Details of TI-Oracle (Shortest Path Query Phase)***. We give the shortest path query phase of *TI-Oracle*.

Notation: Given a source q, we re-use the notation $D(q)$ as in the construction phase of *RC-Oracle*, but q and elements in $D(q)$ are points on C in the shortest path query phase of *TI-Oracle*.

Detail and example: Algorithm 4 shows the shortest path query phase of *TI-Oracle* in detail, and the following illustrates it with an example.

Algorithm 4 *TI-Oracle-Query* ($C, P, s, t, B, M_{cont}, M_{boun}, M_{belo}, M_{intra\text{-}path}, M_{inter\text{-}path}, M_{inter\text{-}end}$)

Input: A point cloud C. A set of POIs P. A source s that is any point on C. A destination t that is a POI in P. A set of boundary points B, the containing point map table M_{cont}. A boundary point map table M_{boun}. A belonging point map table M_{belo}. An intra-path map table $M_{intra\text{-}path}$. An inter-path map table $M_{inter\text{-}path}$. An inter-endpoint map table $M_{inter\text{-}end}$.

Output: An updated intra-path map table $M_{intra\text{-}path}$. The shortest path $\Pi_{TI\text{-}Oracle}(s, t|C)$ between s and t passing on C.

1: **if** $\langle s, t \rangle \in M_{intra\text{-}path}.key$ **then**
2:      Use $\langle s, t \rangle$ to retrieve $\Pi^*(s, t|C)$ as $\Pi_{TI\text{-}Oracle}(s, t|C)$
3: **else if** $\langle s, t \rangle \notin M_{intra\text{-}path}.key$ **then**
4:      $B_s \leftarrow \emptyset, B_t \leftarrow \emptyset$
5:      **if** $s \in M_{belo}.key$ **then**
6:          $u \leftarrow$ retrieved from M_{belo} using s as key
7:          $D(s) \leftarrow$ retrieved from M_{cont} using u as key (except s) $\cup$ retrieved from M_{boun} using u as key
8:          Calculate the exact shortest paths passing on C from s to each point in $D(s)$ simultaneously using algorithm *FastFly*
9:          **for** each point $v \in D(s)$ such that $\langle s, v \rangle \notin M_{intra\text{-}path}.key$ **do**
10:             $key \leftarrow \langle s, v \rangle, value \leftarrow \Pi^*(s, v|C), M_{intra\text{-}path} \leftarrow M_{intra\text{-}path} \cup \{key, value\}$
11:          $B_s \leftarrow$ retrieved from M_{boun} using u as key
12:      **else if** $s \in B$ (resp. $s \in P$) **then**
13:          $B_s \leftarrow \{s\}$ (resp. retrieved from M_{boun} using s as key)
14:      **if** $t \in B$ (resp. $t \in P$) **then**
15:          $B_t \leftarrow \{t\}$ (resp. retrieved from M_{boun} using t as key)
16:      Calculate $\Pi_{TI\text{-}Oracle}(s, t|C)$ by concatenating $\Pi^*(s, s'|C)$, $\Pi_{inter\text{-}path}(s', t'|C)$ and $\Pi^*(t', t|C)$ such that $|\Pi_{TI\text{-}Oracle}(s, t|C)| = \min_{\forall s' \in B_s, \forall t' \in B_t} [|\Pi^*(s, s'|C)| + |\Pi_{inter\text{-}path}(s', t'|C)| + |\Pi^*(t', t|C)|]$
17:      $\Pi^*(s, s'|C)$ and $\Pi^*(t', t|C)$ are retrieved from $M_{intra\text{-}path}$ using $\langle s, s' \rangle$ and $\langle t', t \rangle$ as key, $\Pi_{inter\text{-}path}(s', t'|C)$ is calculated by the shortest path query phase of *RC-Oracle* using $s', t', M_{inter\text{-}path}$ and $M_{inter\text{-}end}$
18: **return** $M_{intra\text{-}path}$ and $\Pi_{TI\text{-}Oracle}(s, t|C)$

---

(1) *Same partition cell* (lines 1-2): In Figures 10 (d) and (f), given k as a source that is not a POI, and e as a destination that is a POI, since $\langle e, k \rangle \in M_{intra\text{-}path}.key$, we directly retrieve $\Pi^*(e, k|C)$.

(2) *Different partition cell* (lines 3-17): In Figures 10 (i) and (j), given k as a source that is not a POI, and a as a destination that is a POI, since $\langle a, k \rangle \notin M_{intra\text{-}path}.key$, there are two steps.

(i) *Intra-paths calculation* (lines 5-15): In Figures 10 (e), (i) and (j), $k \in M_{belo}.key$, we use key k to retrieve the POI e in M_{belo}, use key e to retrieve $\{j, \dots\}$ (except k) in M_{belo} and $\{i, \dots\}$ in M_{boun}, so we have $D(k) = \{i, j, \dots\}$. We calculate intra-paths in orange lines using algorithm *FastFly*, and store the key-value pairs in $M_{intra\text{-}path}$ in Figure 10 (j). In Figure 10 (i), we have a set of red points $B_s = \{i, \dots\}$ around k, and a set of red points $B_t = \{g, h, \dots\}$ around a. If $s = i \in B$, then $B_s = \{i\}$. If $s = e \in P$, then $B_s = \{i, \dots\}$. If $t = h \in B$, then $B_t = \{h\}$.

(ii) *Shortest path query* (lines 16-17): In Figure 10 (k), we use intra-paths $\Pi^*(a, h|C)$ and $\Pi^*(k, i|C)$ in green and orange lines, and the inter-path $\Pi_{inter\text{-}path}(h, i|C) = \Pi^*(h, i|C)$ in blue line to approximate $\Pi_{TI\text{-}Oracle}(a, k|C)$.

5.2.4 *Proximity Query Algorithms using TI-Oracle*.

We introduce the key idea of proximity query algorithms using *TI-Oracle*. Figure 9 shows an overview. Given C, a query object q, a set of n' target objects O on C, a value k in *kNN* query and a value r in range query, we can answer *kNN*

and range queries using *TI-Oracle*. In the A2P query, there are two cases. (1) The query object is any point on C, and the target objects are POIs in P. (2) The query object is a POI in P, and the target objects are any point on C. For both cases, the proximity query algorithms using *TI-Oracle* are the same. Similarly to *RC-Oracle*, a naive algorithm performs the shortest path query n' times with q as a source and performs a linear scan on the results. Then, it returns all shortest paths passing on C from q to its k nearest target objects of q or target objects whose distance to q is at most r. We also propose an efficient algorithm for it. Intuitively, when we perform the linear scan using the shortest path query phase of *TI-Oracle*, we use the shortest path query phase of *RC-Oracle* to find an inter-path between the same pair of boundary points more than once. One such query only needs $O(1)$ time. But, if the exact shortest path passing on C between this pair of boundary points is not stored in $M_{inter\text{-}path}$, the experimental running time is increased. Since we need to search in $M_{inter\text{-}end}$, and again in $M_{inter\text{-}path}$ for path appending. To handle this, if this happens, after we find such an inter-path using the shortest path query phase of *RC-Oracle*, we use an additional table to store it. So, when we need this inter-path again later, we can directly return the result in this additional table for time-saving. Then, we give the notation, detail and example (using *kNN* query with $k = 1$).

Notation: Let $M'_{inter\text{-}path}$ be an additional inter-path map table similar to $M_{inter\text{-}path}$. But, $M'_{inter\text{-}path}$ not only stores the exact inter-paths, but also stores inter-paths returned by *RC-Oracle* with $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$ as input. In Figure 10 (g), $M_{inter\text{-}path}$ only stores 6 exact inter-paths in blue lines with h as a source. But, apart from these, $M'_{inter\text{-}path}$ also stores the inter-path between g and i.

Detail and example: We perform a linear scan on the shortest path query result between q and each object in O. Then, we obtain *kNN* or range query result. In the A2P query, there are two cases for the query object and target objects. Without loss of generality, we answer the proximity query with the query object being any point on C, and the target objects being POIs in P. In Figure 10 (i) - (j), given k as the query object, we perform a linear scan on the shortest path query result between k and each target object in O. The *kNN* result stores $\{\Pi(e, k|C)\}$. For each shortest path query, we follow Algorithm 4, there is only one change in line 17. We first initialize $M'_{inter\text{-}path}$ to be empty. For finding $\Pi_{inter\text{-}path}(s', t'|C)$, we first search in $M'_{inter\text{-}path}$. There are two cases.

(1) *Inter-path retrieval by $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$*: If $\langle s', t' \rangle \notin M'_{intra\text{-}path}.key$, we calculate $\Pi_{inter\text{-}path}(s', t'|C)$ by the shortest path query phase of *RC-Oracle* using s', t', $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$. We store $\langle s', t' \rangle$ as key and $\Pi_{inter\text{-}path}(s', t'|C)$ as value in $M'_{inter\text{-}path}$. In Figures 10 (f) - (h), suppose that $M'_{intra\text{-}path}$ is empty. Given g, i, $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$, $\Pi_{inter\text{-}path}(g, i|C)$ is approximated by $\Pi^*(g, h|C)$ and $\Pi^*(h, i|C)$. We store $\langle g, i \rangle$ as key and $\Pi_{inter\text{-}path}(g, i|C)$ as value in $M'_{inter\text{-}path}$.

(2) *Inter-path retrieval by $M'_{inter\text{-}path}$*: If $\langle s', t' \rangle \in M'_{intra\text{-}path}.key$, we use $\langle s', t' \rangle$ to retrieve $\Pi_{inter\text{-}path}(s', t'|C)$ in $M'_{inter\text{-}path}$. In Figure 10 (g), we may need to find the inter-path between g and i again. But, since $\langle g, i \rangle \in M'_{intra\text{-}path}.key$, we can directly use one table $M'_{inter\text{-}path}$ to retrieve $\Pi_{inter\text{-}path}(g, i|C)$, without searching in two tables $M_{inter\text{-}path}$ and $M_{inter\text{-}end}$.

5.2.5 *Theoretical Analysis about TI-Oracle*. We show some theoretical analysis.

(1) **TI-Oracle**: The analysis of *TI-Oracle* is in Theorem 5.1.

THEOREM 5.1. *The oracle construction time, oracle size and shortest path query time of TI-Oracle are $O(\frac{N \log N}{\epsilon} + Nn + n \log n)$, $O(\frac{N}{\epsilon})$ and $O(1)$, respectively. It always have $|\Pi_{TI\text{-}Oracle}(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C)|$ for any point s on C and any POI t in P.*

PROOF. Firstly, we show the *oracle construction time*. (i) In *partition cells calculation* step, it needs $O(n \log n)$ time. Since there are n POIs and we use the sweep line algorithm [17] that runs in $O(n \log n)$ time to construct them. (ii) In M_{cont}, M_{boun} and M_{belo} *calculation* step, it needs $O(Nn)$

time. Since there are N points and n POIs (i.e., partition cells), for each point, we need to check which partition cell it belongs to. (iii) In *intra-paths calculation* step, it needs $O(n)$ time. Since we use $O(n)$ POIs as sources to run algorithm *FastFly* for calculating intra-paths, where each algorithm *FastFly* needs $O(1)$ time (the destination boundary points are close to their corresponding POI). (iv) In *inter-paths calculation* step, it needs $O(\frac{N \log N}{\epsilon})$ time. Since there are at most $O(N)$ boundary points and we use *RC-Oracle* to calculate inter-paths, we just need to change n to N in the oracle construction time of *RC-Oracle*. (v) So, the oracle construction time is $O(\frac{N \log N}{\epsilon} + Nn + n \log n)$.

Secondly, we show the *oracle size*. (i) For M_{cont}, M_{bond}, M_{belo} and $M_{intra-path}$, their sizes are all $O(N)$ since there are N points on C. (ii) For $M_{inter-end}$ and $M_{inter-end}$, their sizes are $O(N)$ and $O(\frac{N}{\epsilon})$, respectively. Since there are at most $O(N)$ boundary points, $M_{inter-end}$ and $M_{inter-end}$ correspond to M_{end} and M_{end} in *RC-Oracle*, we just need to change n to N for these two tables in *RC-Oracle*. (iii) So, the oracle size is $O(\frac{N}{\epsilon})$.

Thirdly, we show the *shortest path query time*. (i) If $\Pi^*(s, t|C) \in M_{intra-path}$, the shortest path query time is $O(1)$. (ii) If $\Pi^*(s, t|C) \notin M_{intra-path}$, we run algorithm *FastFly* to calculate intra-paths in $O(1)$ time, and run the shortest path query phase of *RC-Oracle* to calculate inter-paths in $O(1)$ time. Thus, the shortest path query time of *TI-Oracle* is $O(1)$.

Fourthly, we show the *error bound*. Given s and t, let s' and t' be the boundary points of the partition cell that s and t belong to, such that we concatenate $\Pi^*(s, s'|C)$, $\Pi_{inter-path}(s', t'|C)$ and $\Pi^*(t', t|C)$ to calculate $\Pi_{TI\text{-}Oracle}(s, t|C)$. Let p and q be the boundary points of the partition cell that s and t belong to, such that they lie on $\Pi^*(s, t|C)$. We have $|\Pi_{TI\text{-}Oracle}(s, t|C)| = |\Pi^*(s, s'|C)| + |\Pi_{inter-path}(s', t'|C)| + |\Pi^*(t', t|C)| \leq |\Pi^*(s, p|C)| + |\Pi_{inter-path}(p, q|C)| + |\Pi^*(q, t|C)| \leq |\Pi^*(s, p|C)| + (1 + \epsilon)|\Pi^*(p, q|C)| + |\Pi^*(q, t|C)| \leq (1 + \epsilon)|\Pi^*(s, p|C)| + (1 + \epsilon)|\Pi^*(p, q|C)| + (1 + \epsilon)|\Pi^*(q, t|C)| = (1 + \epsilon)|\Pi^*(s, t|C)|$. The first equation is because $\Pi_{TI\text{-}Oracle}(s, t|C)$ is calculated by the $\Pi^*(s, s'|C)$, $\Pi_{inter-path}(s', t'|C)$ and $\Pi^*(t', t|C)$. The second inequality is because s' and t' are the boundary points that result in the shortest distance of $\Pi_{TI\text{-}Oracle}(s, t|C)$. The third inequality is because $|\Pi_{inter-path}(p, q|C)| \leq (1 + \epsilon)|\Pi^*(p, q|C)|$, i.e., the error bound of *RC-Oracle* for the inter-path. The fourth inequality is because $|\Pi^*(s, p|C)| \leq (1 + \epsilon)|\Pi^*(s, p|C)|$ and $|\Pi^*(q, t|C)| \leq (1 + \epsilon)|\Pi^*(q, t|C)|$. The fifth inequality is because p and q are the boundary points that result in the shortest distance of $\Pi^*(s, t|C)$. □

(2) **Proximity query algorithms**: We show the query time and error rate of *kNN* and range queries using *TI-Oracle* in Theorem 5.2.

THEOREM 5.2. *The query time and error rate of both the kNN and range queries by using TI-Oracle are $O(n')$ and $(1 + \epsilon)$, respectively.*

PROOF SKETCH. The *query time* is due to the usage of the shortest path query phase for n' times. The *error rate* is due to its definition and the error of *TI-Oracle*. □

5.3 TI-Oracle-A2A and Its Proximity Query Algorithms

5.3.1 Key Idea of TI-Oracle-A2A.
We introduce the key idea of the efficient adaptation from *TI-Oracle* to *TI-Oracle-A2A*. In the oracle construction phase, since no POI is given, we randomly select some points (e.g., $\sqrt{N}$ points) as POIs. Then, we follow the same oracle construction phase as *TI-Oracle* to construct *TI-Oracle-A2A*. In the shortest path query phase, given any pair of points s and t on C, we follow the same shortest path query phase as *TI-Oracle*. The only difference is that we use the *SSAD* algorithm twice for both s and t as sources to calculate intra-paths.

5.3.2 Proximity Query Algorithms using TI-Oracle-A2A.
Recall that there is a query object q and a set of n' target objects O on C. In the A2A query, the query object is any point on C, and the

target objects are any point on C. Our proximity query algorithms using *TI-Oracle-A2A* are similar to using *TI-Oracle*.

5.3.3 **Theoretical Analysis about TI-Oracle-A2A.** We show some theoretical analysis.

(1) **TI-Oracle-A2A**: The analysis of *TI-Oracle-A2A* is in Theorem 5.3.

THEOREM 5.3. *The oracle construction time, oracle size and shortest path query time of TI-Oracle-A2A are* $O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$, $O(\frac{N}{\epsilon})$ *and* $O(1)$, *respectively. It always have* $|\Pi_{TI\text{-}Oracle\text{-}A2A}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ *for any pair of points s and t on C.*

PROOF. Compared with *TI-Oracle*, in the oracle construction time and oracle size, we change n to $\sqrt{N}$ since *TI-Oracle-A2A* selects $\sqrt{N}$ points as POIs. In the error bound, we change "any point s on C and any POI t in P" to "any pair of points s and t on C". The others are the same as *TI-Oracle*. □

(2) **Proximity query algorithms**: We show the query time and error rate of *kNN* and range queries using *TI-Oracle-A2A* in Theorem 5.4.

THEOREM 5.4. *The query time and error rate of both the kNN and range queries by using TI-Oracle-A2A are* $O(n')$ *and* $(1 + \epsilon)$, *respectively.*

PROOF SKETCH. The proof is the same as *TI-Oracle*. □

6 EMPIRICAL STUDIES

6.1 Experimental Setup

We conducted the experiments on a Linux machine with 2.2 GHz CPU and 512GB memory with both point clouds and *TIN*s as input. All algorithms were implemented in C++. Our experimental setup generally follows the setups in the literature [26, 27, 44, 45, 51, 52].

6.1.1 **Datasets.** (1) Point cloud datasets: We conducted our experiment based on 34 (= 5 + 5 + 24) real point cloud datasets in Table 4, where the subscript c means a point cloud. For BH_c and EP_c datasets, they are represented as a point cloud with 8km × 6km covered region. For GF_c, LM_c and RM_c, we first obtained the satellite model from Google Earth [4] with 8km × 6km covered region, and then used Blender [1] to generate the point cloud. These five original datasets have a resolution of 10m × 10m [40, 44, 45, 51, 52]. We extracted 500 POIs using OpenStreetMap [44, 45] for these datasets in the P2P query. For small-version datasets, we use the same region of the original datasets with a (lower) resolution of 70m × 70m and the dataset generation procedure in [45, 51, 52] to generate them. In addition, we have six sets of multi-resolution datasets with different numbers of points generated similarly. (2) *TIN* datasets: We triangulate [12] 34 point cloud datasets and generate another 34 *TIN* datasets, and use t as the subscript. Storing a point cloud with 2.5M points needs 39MB, but storing a *TIN* converted from this point cloud needs 170MB.

6.1.2 **Algorithms.** (1) To solve our problem on point clouds, we adapted existing algorithms on *TIN*s, by converting the given point clouds to *TIN*s via triangulation [12] so that the existing algorithms could be performed. Their algorithm names are appended by "*-Adapt*". We have 4 on-the-fly algorithms. (i) *DIO-Adapt* [15, 46]: the best-known adapted *TIN* exact on-the-fly shortest surface path query algorithm. (ii) *ESP-Adapt* [26, 51]: the best-known adapted *TIN* approximate on-the-fly shortest surface path query algorithm. (iii) *DIJ-Adapt* [27]: the best-known adapted *TIN* approximate on-the-fly shortest network path query algorithm. (iv) *FastFly*: our algorithm. We have 13 oracles. (v) *SE-Oracle-Adapt* [44, 45]: the best-known adapted *TIN* oracle for the P2P query on a point cloud. (vi) *UP-Oracle-Adapt* [54]: adapted *TIN* oracle for the P2P query on a point cloud. (vii) *EAR-Oracle-Adapt* [25]: the best-known adapted *TIN* oracle for both the A2P and A2A queries

on a point cloud. (viii) *RC-Oracle-Naive*: the naive version of *RC-Oracle* without shortest paths approximation step for the P2P query on a point cloud. (ix) *RC-Oracle*: the oracle for the P2P query on a point cloud proposed in the previous conference paper [52]. (x) *SE-Oracle-Adapt-A2A*: the adapted *SE-Oracle-Adapt* (by placing POIs on faces of the converted *TIN* of the point cloud, see more details in [44, 45]) for the A2A query on a point cloud. (xi) *UP-Oracle-Adapt-A2A*: the adapted *UP-Oracle-Adapt* in a similar way of *SE-Oracle-Adapt-A2A* for the A2A query on a point cloud. (xii) *RC-Oracle-Naive-A2A*: the adapted *RC-Oracle-Naive* in a similar way of *RC-Oracle-A2A* for the A2A query on a point cloud. (xiii) *RC-Oracle-A2A*: the oracle for the A2A query on a point cloud proposed in the previous conference paper [52]. (xiv, xv & xvi) *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle*: the oracles for the A2P query on a point cloud proposed in this journal paper. (xvii) *TI-Oracle-A2A*: the oracle for the A2A query on a point cloud proposed in this journal paper.

(2) To solve the existing problem on *TIN*s, we adapted our algorithms on point clouds, by converting the given *TIN*s to point clouds so that our algorithms could be performed. This involves regarding the vertices of the *TIN* as the points of the point cloud. For AR2P and AR2AR queries, we also take the additional step in Section 4.6. Our algorithm names are appended by "*-Adapt*". Similarly, we have 4 on-the-fly algorithms. (i) *DIO* [15, 46]. (ii) *ESP* [26, 51]. (iii) *DIJ* [27]. (iv) *FastFly-Adapt*. We have 13 oracles. (v) *SE-Oracle* [44, 45]. (vi) *UP-Oracle* [54]. (vii) *EAR-Oracle* [25]. (viii) *RC-Oracle-Naive-Adapt*. (ix) *RC-Oracle-Adapt*. (x) *SE-Oracle-AR2AR*. (xi) *UP-Oracle-Adapt-AR2AR*. (xii) *RC-Oracle-Naive-Adapt-AR2AR*. (xiii) *RC-Oracle-Adapt-AR2AR*. (xiv) *RC-Oracle-Adapt-AR2P-SmCon*. (xv) *RC-Oracle-Adapt-AR2P-SmQue*. (xvi) *TI-Oracle-Adapt*. (xvii) *TI-Oracle-Adapt-AR2AR*.

6.1.3 **Query generation**. We conducted all proximity queries, i.e., (1) shortest path query, (2) all objects kNN query, and (3) all objects range query. (1) For the shortest path query, we issued 100 query instances. For each instance, we randomly chose a source and a destination. They can be a POI, any point on a point cloud or *TIN*, depending on the P2P, A2P, A2A, AR2P or AR2AR query types. The average, minimum and maximum results were reported. In the experimental result figures, the vertical bar and the points mean the minimum, maximum and average results. (2 & 3) For all objects kNN query and range query, we perform the proximity query algorithms for our oracles, and a linear scan for other baselines (as described in [45]) using all objects as query objects. We randomly select 500 objects on the point cloud or *TIN*. They can be a POI, any point on a point cloud or *TIN*, depending on the P2P, A2P, A2A, AR2P or AR2AR query types. Since we perform linear scans or use the calculated distance stored in M_{path}, $M_{intra-path}$ or $M_{inter-path}$ for proximity query, the value of k and r will not affect their query time, we set $k = 3$ and $r = 1$km.

6.1.4 **Factors and measurements**. We studied three factors. (1) ϵ (i.e., the error parameter). (2) n (i.e., the number of POIs). (3) N (i.e., the number of points or vertices in a point cloud or *TIN*). We used nine measurements to evaluate algorithm performance. (1) *Oracle construction time*. (2) *Memory consumption* (i.e., the space consumption when running the algorithm). (3) *Oracle size*. (4) *Query time* (i.e., the shortest path query time). (5 & 6) *kNN and range query time* (i.e., all objects kNN and range query time). (7) *Distance error* (i.e., the error of the distance returned by the algorithm compared with the exact distance). (8 & 9) *kNN and range query error* (i.e., the error rate of the kNN and range query defined in Section 4.2.5).

6.2 Experimental Results for *TIN*s

We first study proximity queries on *TIN*s (studied by previous studies [25, 44, 45]) to justify why our proximity queries on *point clouds* are useful. We have the following settings. (1) The distance of the path calculated by *DIO* is used for distance error calculation since the path is the exact shortest surface path passing on the *TIN*. (2) For the P2P query on a *TIN*, we compared the following

algorithms. We compared *SE-Oracle, UP-Oracle, EAR-Oracle, RC-Oracle-Naive-Adapt, RC-Oracle-Adapt, DIO, ESP, DIĴ* and *FastFly-Adapt* on small-version datasets. There are 50 POIs by default. We compared *RC-Oracle-Adapt, DIO, ESP, DIĴ* and *FastFly-Adapt* on original datasets since the rest have expensive oracle construction time. There are 500 POIs by default. (3) For the AR2P and AR2AR queries on a *TIN*, we compared the following algorithms. We compared *SE-Oracle-AR2AR, UP-Oracle-AR2AR, EAR-Oracle, RC-Oracle-Naive-Adapt-AR2AR, RC-Oracle-Adapt-AR2AR, RC-Oracle-Adapt-AR2P-SmCon, RC-Oracle-Adapt-AR2P-SmQue, TI-Oracle-Adapt, TI-Oracle-Adapt-AR2AR* and *FastFly-Adapt* on small-version datasets. There are 50 POIs for the AR2P query by default. We compared *RC-Oracle-Adapt-AR2AR, RC-Oracle-Adapt-AR2P-SmCon, RC-Oracle-Adapt-AR2P-SmQue, TI-Oracle-Adapt, TI-Oracle-Adapt-AR2AR* and *FastFly-Adapt* on original datasets since the rest are not feasible on original datasets. There are 500 POIs for the AR2P query by default.

6.2.1 **Baseline comparisons.** We study the effect of ϵ and n for the P2P, AR2P and AR2AR queries on a *TIN* here. We study the effect of N for these queries in the appendix.

Effect of ϵ for the P2P query on a *TIN*. In Figure 13, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_c-*small* dataset by setting N to be 10k and n to be 50 for baseline comparisons for the P2P query on a *TIN*. Although a *TIN* is given as input, *RC-Oracle-Adapt* performs better than all other oracles in terms of the oracle construction time, oracle size and shortest path query time. Although *FastFly-Adapt* needs to convert a point cloud from the given *TIN*, the shortest path query time of *FastFly-Adapt* is 100 times smaller than that of *DIO*. Since the query region of the path calculated by *FastFly-Adapt* is smaller than that of *DIO*. The distance error of *FastFly-Adapt* (i.e., 0.002) is very small compared with *DIO* (i.e., without error), and much smaller than that of *DIĴ* (i.e., 0.1). This means that the shortest distance on a point cloud (resp. the shortest network distance on a *TIN*) is 1.002 (resp. 1.1) times larger than the shortest surface distance on a *TIN*. The *kNN* and range query error are all equal to 0 (due to the small distance error), so their results are omitted.
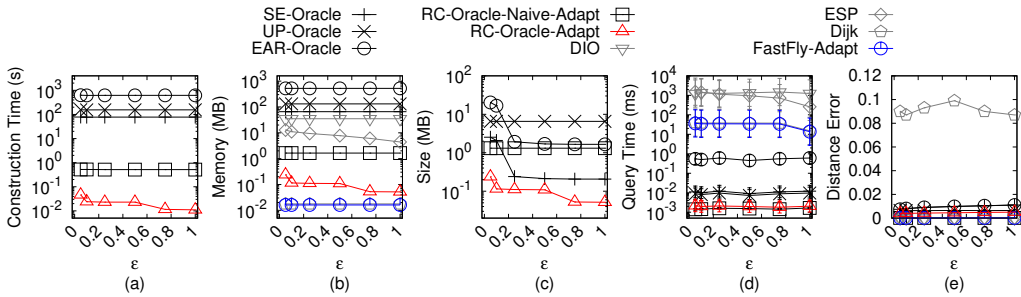
Fig. 13. Baseline comparisons (effect of ϵ on BH_t-*small TIN* dataset for P2P query)

Effect of n for the P2P query on a *TIN*. In Figure 14, we tested 5 values of n from {50, 100, 150, 200, 250} on EP_t dataset by setting N to be 10k and ϵ to be 0.1 for baseline comparisons for the P2P query on a *TIN*. In Figure 14 (a), when n increases, the construction time of all oracles increases. In Figure 14 (b), when n increases, the memory consumption of *RC-Oracle-Adapt* exceeds that of *DIĴ* and *FastFly-Adapt*. This is because *RC-Oracle-Adapt* is an oracle that is affected by n, it needs more memory consumption during the oracle construction to calculate more paths among these POIs when n increases. But, *DIĴ* and *FastFly-Adapt* are on-the-fly algorithms which are not affected by n, their memory consumption only measures the space consumption for calculating one path.

Effect of ϵ for the AR2P query on a *TIN*. In Figure 15, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_c-*small* dataset by setting N to be 10k and n to be 50 for baseline comparisons
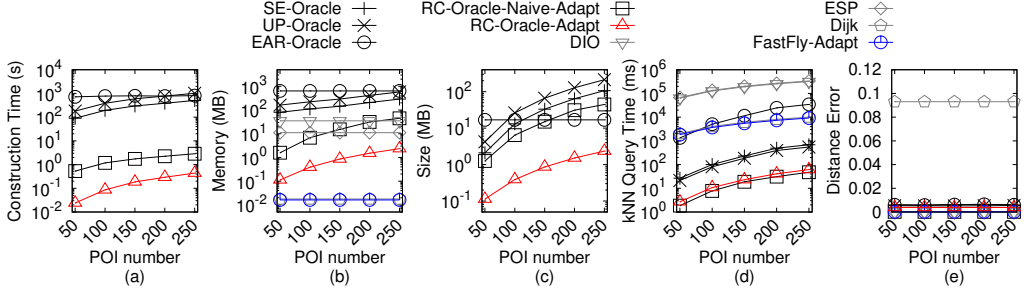
Fig. 14. Baseline comparisons (effect of n on EP_t-small TIN dataset for P2P query)

for the AR2P query on a *TIN*. The oracle construction time, memory usage and oracle size of *RC-Oracle-Adapt-AR2P-SmCon* are the smallest in all oracles since it has the same oracle construction process as *RC-Oracle-Adapt*. *RC-Oracle-Adapt-AR2P-SmCon* performs better than *FastFly-Adapt*, since it can terminate earlier when using *FastFly-Adapt* in the shortest path query phase. But, its shortest path query time is larger than other oracles, so it performs well in the case of fewer proximity queries. The oracle construction time and shortest path query time of *RC-Oracle-Adapt-AR2P-SmQue* and *TI-Oracle-Adapt* are also very small. Since they terminate algorithm *FastFly-Adapt* earlier during oracle construction and store tight information in the oracles.
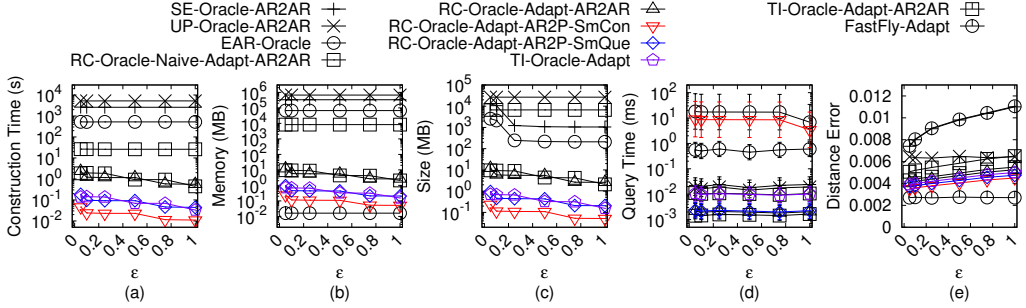
Fig. 15. Baseline comparisons (effect of ϵ on BH_t-small TIN dataset for AR2P query)

Effect of n for the AR2P query on a *TIN*. In Figure 16, we tested 5 values of n from {50, 100, 150, 200, 250} on EP_t dataset by setting N to be 10k and ϵ to be 0.1 for baseline comparisons for the AR2P query on a *TIN*. When $n < 100$ (resp. $n \geq 100$), the oracle construction time of *RC-Oracle-Adapt-AR2P-SmQue* is smaller (resp. larger) than that of *TI-Oracle-Adapt*. Thus, the former (resp. latter) performs well when the density of POIs is high (resp. low).

AR2AR query on a *TIN*. In Figures 15 and 16, we also compared oracles for the AR2AR query on a *TIN*. *SE-Oracle-AR2AR, UP-Oracle-AR2AR, EAR-Oracle, RC-Oracle-Naive-Adapt-AR2AR, RC-Oracle-Adapt-AR2AR* and *TI-Oracle-Adapt-AR2AR* can answer the AR2AR query on a *TIN*. The last two oracles still perform better than other oracles in terms of oracle construction time, oracle size and shortest path query time. Since they terminate algorithm *FastFly-Adapt* earlier during oracle construction and store tight information in the oracles.

6.3 Experimental Results for Point Clouds

Now, we understand the effectiveness of proximity queries on *point clouds*. In this section, we then study proximity queries on *point clouds* using the algorithms in Table 3. We have the following
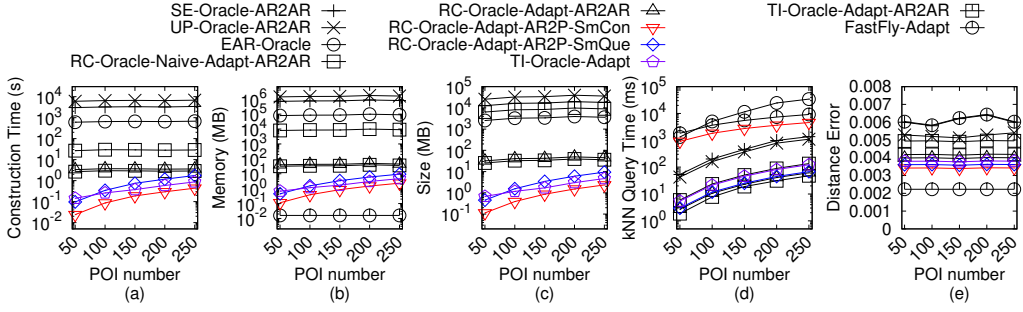
Fig. 16. Baseline comparisons (effect of n on EP_t-small TIN dataset for AR2P query)

setting. (1) The distance of the path calculated by *FastFly* is used for distance error calculation since the path is the exact shortest path passing on the point cloud. (2) We compared similar algorithms on small-version or original datasets with the same reasons for *TIN*s.

6.3.1 ***Baseline comparisons***. We study the effect of ϵ, n and N for the P2P, A2P and A2A queries on a point cloud here.

Effect of ϵ for the P2P query on a point cloud. In Figure 17, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on GF_c-*small* dataset by setting N to be 10k and n to be 50 for baseline comparisons for the P2P query on a point cloud. The oracle construction time, memory consumption and oracle size of *RC-Oracle* are smaller than *SE-Oracle-Adapt*. Since *RC-Oracle* has *rapid oracle construction* advantage, it can terminate algorithm *FastFly* earlier with less time and memory, and stores fewer paths. The shortest path query time of *RC-Oracle* is smaller than *SE-Oracle-Adapt*. Since their theoretical values are $O(1)$ and $O(h^2)$, respectively. In Figures 17 (a & b), varying ϵ has a large effect on the oracle construction time and memory consumption of *RC-Oracle*. When ϵ increases from 0.05 to 1, their values increase by 5 times. But, due to the log scale used in the experimental figures, the effect is not obvious. Varying ϵ has a small effect on the oracle construction time and memory consumption of *SE-Oracle-Adapt* and *EAR-Oracle-Adapt*. Since even when ϵ is large, they cannot terminate the *SSAD* algorithm earlier for most cases due to their *loose criterion for algorithm earlier termination* drawback. The shortest path, *kNN* and range queries time of *RC-Oracle* are much smaller than the on-the-fly algorithms. The distance error of *RC-Oracle* is close to 0.
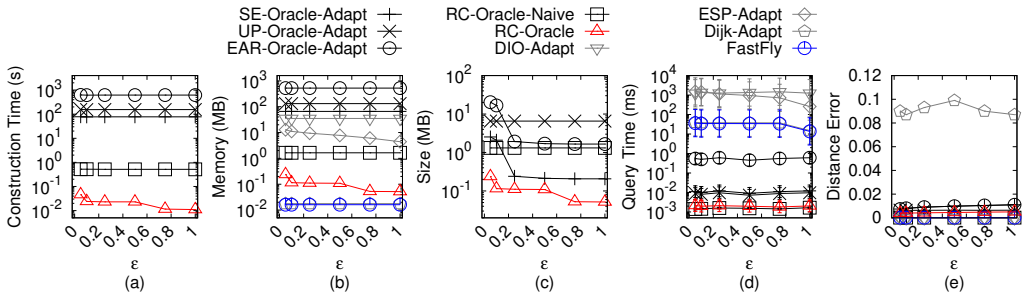
Fig. 17. Baseline comparisons (effect of ϵ on GF_c-*small* point cloud dataset for P2P query)

Effect of n for the P2P query on a point cloud. In Figure 18, we tested 5 values of n from $\{500, 1000, 1500, 2000, 2500\}$ on LM_c dataset by setting N to be 0.5M and ϵ to be 0.25 for baseline comparisons for the P2P query on a point cloud. Since *RC-Oracle* is an oracle, its *kNN* query time

is smaller than on-the-fly algorithms. Algorithm *FastFly* runs faster than other *TIN* on-the-fly algorithms since it calculates the shortest path passing on a point cloud. Algorithm *FastFly* calculates neighbors during shortest path calculation, and other *TIN* on-the-fly algorithms do so beforehand. But, the point cloud neighbors calculation time is 10^4 to 10^6 times smaller than the shortest path calculation time. Thus, the former time can be neglected.

Effect of N (scalability test) for the P2P query on a point cloud. In Figure 19, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on RM_c dataset by setting n to be 500 and ϵ to be 0.25 for baseline comparisons for the P2P query on a point cloud. The oracle construction time of *RC-Oracle* is only 80s $\approx$ 1.3 min for a point cloud with 2.5M points and 500 POIs, which shows its scalability. The range query time of *RC-Oracle* is the smallest.
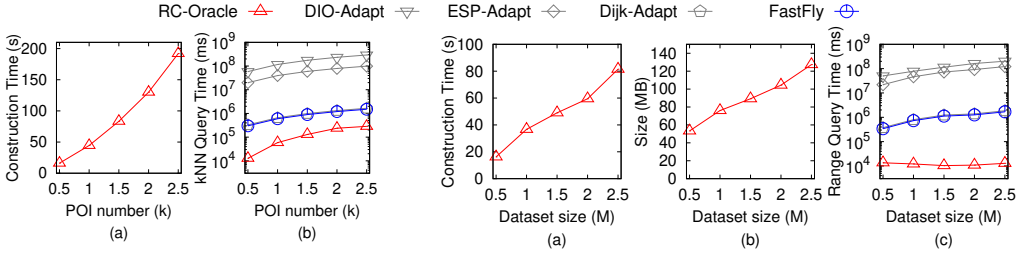
Fig. 18. Baseline comparisons (effect of n on LM_c point cloud dataset for P2P query)

Fig. 19. Baseline comparisons (effect of N on RM_c point cloud dataset for P2P query)

Effect of ϵ for the A2P query on a point cloud. In Figure 20, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on GF_c-*small* dataset by setting N to be 10k and n to be 50 for baseline comparisons for the A2P query on a point cloud. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* perform better than *EAR-Oracle-Adapt*, *RC-Oracle-A2A* and *TI-Oracle-A2A*. Since *EAR-Oracle-Adapt* has the *loose criterion for algorithm earlier termination* drawback, *RC-Oracle-A2A* and *TI-Oracle-A2A* are not designed for the A2P query on a point cloud.
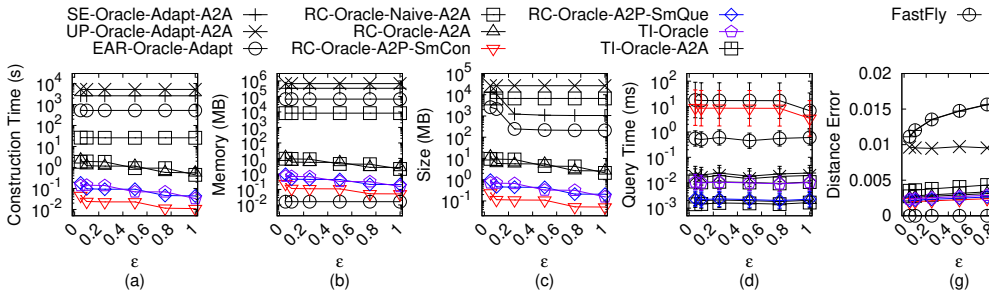
Fig. 20. Baseline comparisons (effect of ϵ on GF_c-*small* point cloud dataset for A2P query)

Effect of n for the A2P query on a point cloud. In Figure 21, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on LM_c dataset by setting N to be 0.5M and ϵ to be 0.25 for baseline comparisons for the A2P query on a point cloud. The oracle construction time, memory usage and oracle size of *RC-Oracle-A2P-SmCon* are the smallest in all oracles. *RC-Oracle-A2P-SmCon* performs better than *FastFly*. But, its shortest path query time is larger than other oracles, so it performs well in the case of fewer proximity queries. The reason is the same as that of *RC-Oracle-Adapt-AR2P-SmCon* for *TIN*s. The oracle construction time and shortest path query time

of *RC-Oracle-A2P-SmQue* and *TI-Oracle* are also very small. The reason is the same as that of *RC-Oracle-Adapt-AR2P-SmQue* and *TI-Oracle-Adapt* for *TIN*s. When $n < 500$ (resp. $n \geq 500$), the oracle construction time of *RC-Oracle-A2P-SmQue* is smaller (resp. larger) than that of *TI-Oracle*. Thus, the former (resp. latter) performs well when the density of POIs is high (resp. low).

Effect of N (scalability test) for the A2P query on a point cloud. In Figure 22, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on RM_c dataset by setting n to be 500 and ϵ to be 0.25 for baseline comparisons for the A2P query on a point cloud. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* are scalable when N is large.
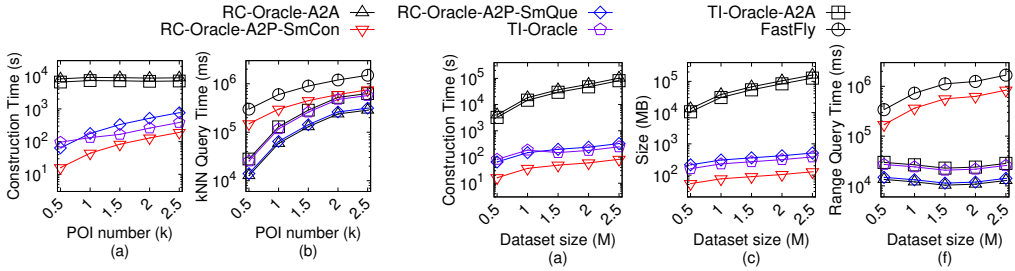
Fig. 21. Baseline comparisons (effect of n on LM_c point cloud dataset for A2P query)

Fig. 22. Baseline comparisons (effect of N on RM_c point cloud dataset for A2P query)

A2A query on a point cloud. In Figures 20, 21 and 22, we also compared oracles for the A2A query on a point cloud. *SE-Oracle-Adapt-A2A*, *UP-Oracle-Adapt-A2A*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive-A2A*, *RC-Oracle-A2A* and *TI-Oracle-A2A* can answer the A2A query on a point cloud. The last two oracles still perform better than the first three oracles in terms of oracle construction time, oracle size and shortest path query time.

6.3.2 *Ablation study*.
We further adapt *SE-Oracle-Adapt*, *UP-Oracle-Adapt*, *EAR-Oracle-Adapt*, *SE-Oracle-Adapt-A2A* and *UP-Oracle-Adapt-A2A* to be *SE-Oracle-FastFly-Adapt*, *UP-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt*, *SE-Oracle-FastFly-Adapt-A2A* and *UP-Oracle-FastFly-Adapt-A2A*. It means that we use algorithm *FastFly* to directly calculate the shortest path passing on a point cloud without converting to a *TIN*.

In Figure 23, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on LM_c dataset by setting N to be 0.5M and n to be 500 for ablation study for the P2P query on a point cloud. We compared *SE-Oracle-FastFly-Adapt*, *UP-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt* and *RC-Oracle*. They only differ by the oracle construction. *RC-Oracle* performs better than all other oracles.

In Figure 24, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on RM_c dataset by setting N to be 0.5M and n to be 500 for ablation study for the A2P query on a point cloud. We compared *SE-Oracle-FastFly-Adapt-A2A*, *UP-Oracle-FastFly-Adapt-A2A*, *EAR-Oracle-FastFly-Adapt*, *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* and *TI-Oracle-A2A*. They only differ by the oracle construction. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* perform better than all other oracles.

In Figure 24, we also compared oracles for the A2A query on a point cloud. We compared *SE-Oracle-FastFly-Adapt-A2A*, *UP-Oracle-FastFly-Adapt-A2A*, *EAR-Oracle-FastFly-Adapt*, *RC-Oracle-A2A* and *TI-Oracle-A2A*. *RC-Oracle-A2A* and *TI-Oracle-A2A* perform better than all other oracles.

6.3.3 *Comparisons with other proximity queries oracles and variation oracles on a point cloud*.
We compared *SU-Oracle-Adapt* [40] (i.e., the oracle designed for the *kNN* query) and some
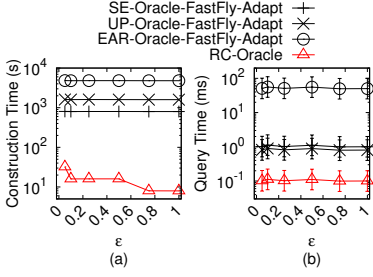
Fig. 23. Ablation study on LM_c point cloud dataset for P2P query
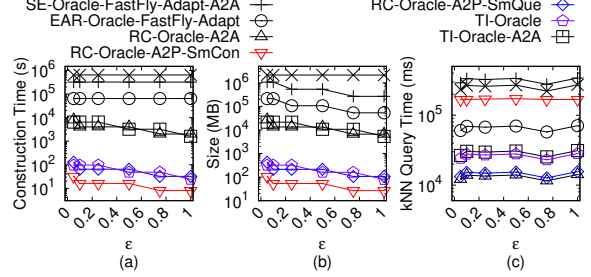
Fig. 24. Ablation study on RM_c point cloud dataset for A2P query

variations of our oracles related to *SU-Oracle-Adapt* in the appendix. Our oracles and variations still outperform other baselines.

6.3.4 **Case study (snowfall)**. We performed a snowfall evacuation case study [33] in Mount Rainier [32] to evacuate tourists to nearby hotels. The blizzard wreaking havoc across the USA in December 2022 killed more than 60 lives [31] and one may be dead due to asphyxiation [39] if s/he gets buried in the snow. A human will be buried in the snow in 2.4 hours[2], and the evacuation (i.e., the time of human's walking from the viewing platforms to hotels) can be finished in 2.2 hours[3]. Thus, the calculation of shortest paths is expected to be finished within 12 min (= 2.4 − 2.2 hours). Due to avalanches, we capture the point cloud dataset *after* snowfall (i.e., the point cloud dataset is updated), so the oracle construction time is considered after snowfall.

Consider the P2P query on a point cloud with 2.5M points and 500 POIs (250 viewing platforms and 250 hotels). The oracle construction time for *RC-Oracle* and the best-known adapted *TIN* oracle *SE-Oracle-Adapt* are 80s ≈ 1.3 min and 78,000s ≈ 21.7 hours, respectively. The query time for calculating 10 nearest hotels of each viewing platform for them are 6s and 75s, respectively. The query time of the same query for *FastFly* and the best-known adapted *TIN* approximate on-the-fly shortest surface path query algorithm *ESP-Adapt* are 2,000s ≈ 33 min and 80,500s ≈ 22.5 hours, respectively. Thus, only *RC-Oracle* is suitable since 1.3 min + 6s ≤ 12 min, but 21.7 hours + 75s ≥ 12 min, 33 min ≥ 12 min and 22.5 hours ≥ 12 min.

Consider the A2P query on a point cloud under the same setting. The oracle construction time for *TI-Oracle*, *RC-Oracle-A2A* and the best-known adapted *TIN* oracle *EAR-Oracle-Adapt* are 250s ≈ 4.1 min, 42,000 ≈ 11.6 hours and 10,500,000s ≈ 121 days, respectively. The query time for the same query for them are 11s, 6s and 600s ≈ 10 min. Thus, only *TI-Oracle* is suitable since 4.1 min + 11s ≤ 12 min, but 11.6 hours + 6s ≥ 12 min and 121 days + 600s ≥ 12 min.

6.3.5 **Case study (solar storm)**. We performed a solar storm evacuation case study [9] for NASA's Mars 2020 rover (costing USD 2.5 billion [37]). During solar storms, rovers need to find shortest escape paths quickly from their current locations (any location) on Mars to shelters (POIs) to avoid damage. The memory size of a rover is 256MB [8]. Consider the A2P query on a point cloud with 250k points and 500 POIs. The oracle construction time for *TI-Oracle* and *RC-Oracle-A2A* are

---

[2]2.4 hours = $\frac{10\text{centimeters} \times 24\text{hours}}{1\text{meter}}$, since the snowfall rate (i.e., the snow depth in a given time [35, 43]) in Mount Rainier is 1 meter per 24 hours [34], and it becomes difficult to walk and get buried in the snow if the snow is deeper than 10 centimeters [23].

[3]2.2 hours = $\frac{11.2\text{km}}{5.1\text{km/h}}$, since the average distance between the viewing platforms and hotels in Mount Rainier National Park is 11.2km [5], and the average human walking speed is 5.1 km/h [10].

25s and 4,200 ≈ 1.2 hours, respectively. The oracle size for them are 28MB and 10GB, respectively. Thus, only *TI-Oracle* is suitable since 28MB ≤ 256MB, but 10GB ≥ 256MB.

6.3.6 **Summary**. Consider the oracle construction time, oracle size and proximity (e.g., *kNN*) query time. For the P2P query on a point cloud with 2.5M points and 500 POIs, these values are 80s ≈ 1.3 min, 50MB and 12.5s for *RC-Oracle*, respectively. They are up to 975 times, 30 times and 12 times better than the best-known adapted *TIN* oracle *SE-Oracle-Adapt* for the P2P query on a point cloud, respectively. For the A2P query on a point cloud with 250k points and 500 POIs, these values are 25s, 28MB and 2.2s for *TI-Oracle*, respectively. They are up to 42,000 times, 10,800 times and 27 times better than the best-known adapted *TIN* oracle *EAR-Oracle-Adapt* for the A2P query on a point cloud, respectively.

7　CONCLUSION

In our paper, we propose six efficient shortest path oracles called *RC-Oracle*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *RC-Oracle-A2A*, *TI-Oracle* and *TI-Oracle-A2A*. They can answer $(1 + \epsilon)$-approximate P2P, A2P and A2A shortest path queries on a point cloud. We also propose efficient proximity query algorithms using these oracles. They can $(1 + \epsilon)$-approximate *kNN* and range queries on a point cloud. Our six oracles and their proximity query algorithms have the state-of-the-art performance in terms of the oracle construction time, oracle size and proximity query time compared with the best-known adapted oracle.

This journal paper is an extension of the previous conference paper that additionally studies the A2P query. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* and *TI-Oracle-A2A* are four new proposed oracles. The first three new oracles for the A2P query outperform the previously proposed best-known oracle *RC-Oracle-A2A* in terms of the oracle construction time and oracle size. The proximity query time of these new oracles (using their proximity query algorithms) are comparable to *RC-Oracle-A2A*.

For the future work, we can extend our oracles from a static point cloud to an updated point cloud. Consider two point clouds before and after updates, denoted by C_{be} and C_{af}, respectively. Suppose that we construct an oracle on C_{be}. After point cloud updates, if C_{be} and C_{af} do not differ a lot, there is no need to construct the oracle on C_{af} from scratch. We aim to efficiently update the oracle on C_{af} using the previous oracle on C_{be}. Then, we can use the updated oracle on C_{af} for proximity queries. In our snowfall evacuation case study, due to avalanches, C_{be} is updated and we need to capture C_{af} after snowfall. We hope to efficiently update the oracle on C_{af}, to answer proximity queries for life-saving.

REFERENCES

[1] 2025. *Blender*. https://www.blender.org
[2] 2025. *Cyberpunk 2077*. https://www.cyberpunk.net
[3] 2025. *Data geocomm*. http://data.geocomm.com/
[4] 2025. *Google earth*. https://earth.google.com/web
[5] 2025. *Google map*. https://www.google.com/maps
[6] 2025. *Gunnison national forest*. https://gunnisoncrestedbutte.com/visit/places-to-go/parks-and-outdoors/gunnison-national-forest/
[7] 2025. *Laramie mountain*. https://www.britannica.com/place/Laramie-Mountains
[8] 2025. *Mars 2020 mission perseverance rover brains*. https://mars.nasa.gov/mars2020/spacecraft/rover/brains/
[9] 2025. *NASA's Maven observes martian light show caused by major solar storm*. https://www.nasa.gov/missions/nasas-maven-observes-martian-light-show-caused-by-major-solar-storm/
[10] 2025. *Preferred walking speed*. https://en.wikipedia.org/wiki/Preferred_walking_speed
[11] 2025. *Robinson mountain*. https://www.mountaineers.org/activities/routes-places/robinson-mountain
[12] Haoan Feng, Yunting Song, and Leila De Floriani. 2024. Critical features tracking on triangulated irregular networks by a scale-space method. In *ACM International Conference on Advances in Geographic Information Systems (GIS)*. 54–66.

[13] Sainyam Galhotra, Rahul Raychaudhury, and Stavros Sintos. 2024. k-Clustering with comparison and distance oracles. In *ACM International Conference on Management of Data (SIGMOD)*, Vol. 2. 1–26.

[14] Paul B Callahan and S Rao Kosaraju. 1995. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM (JACM)* 42, 1 (1995), 67–90.

[15] Jindong Chen and Yijie Han. 1990. Shortest paths on a polyhedron. In *Symposium on Computational Geometry (SOCG)*. 360–369.

[16] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[17] Mark De Berg. 2000. *Computational geometry: algorithms and applications*. Springer Science & Business Media.

[18] Ke Deng, Heng Tao Shen, Kai Xu, and Xuemin Lin. 2006. Surface k-nn query processing. In *IEEE International Conference on Data Engineering (ICDE)*. 78–78.

[19] Ke Deng and Xiaofang Zhou. 2004. Expansion-based algorithms for finding single pair shortest path on surface. In *International Workshop on Web and Wireless Geographical Information Systems (WWGIS)*. 151–166.

[20] Ke Deng, Xiaofang Zhou, Heng Tao Shen, Qing Liu, Kai Xu, and Xuemin Lin. 2008. A multi-resolution surface distance model for k-nn query processing. *The VLDB Journal (VLDBJ)* 17, 5 (2008), 1101–1119.

[21] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (1959), 269–271.

[22] David Eriksson and Evan Shellshear. 2016. Fast exact shortest distance queries for massive point clouds. *Graphical Models* 84 (2016), 28–37.

[23] Fresh Off The Grid. 2025. *Winter hiking 101: everything you need to know about hiking in snow.* https://www.freshoffthegrid.com/winter-hiking-101-hiking-in-snow/

[24] Anupam Gupta, Robert Krauthgamer, and James R Lee. 2003. Bounded geometries, fractals, and low-distortion embeddings. In *IEEE Symposium on Foundations of Computer Science (SFCS)*. 534–543.

[25] Bo Huang, Victor Junqiu Wei, Raymond Chi-Wing Wong, and Bo Tang. 2023. Ear-oracle: on efficient indexing for distance queries between arbitrary points on terrain surface. In *ACM International Conference on Management of Data (SIGMOD)*, Vol. 1. 1–26.

[26] Manohar Kaul, Raymond Chi-Wing Wong, and Christian S Jensen. 2015. New lower and upper bounds for shortest distance queries on terrains. In *International Conference on Very Large Data Bases (VLDB)*, Vol. 9. 168–179.

[27] Manohar Kaul, Raymond Chi-Wing Wong, Bin Yang, and Christian S Jensen. 2013. Finding shortest paths on terrains by killing two birds with one stone. In *International Conference on Very Large Data Bases (VLDB)*, Vol. 7. 73–84.

[28] Mark Lanthier, Anil Maheshwari, and J-R Sack. 2001. Approximating shortest paths on weighted polyhedral surfaces. *Algorithmica* 30, 4 (2001), 527–562.

[29] Lian Liu and Raymond Chi-Wing Wong. 2011. Finding shortest path on land surface. In *ACM International Conference on Management of Data (SIGMOD)*. 433–444.

[30] Joseph SB Mitchell, David M Mount, and Christos H Papadimitriou. 1987. The discrete geodesic problem. *SIAM Journal on Computing (JOC)* 16, 4 (1987), 647–668.

[31] Mithil Aggarwal. 2022. *More than 60 killed in blizzard wreaking havoc across U.S.* https://www.cnbc.com/2022/12/26/death-toll-rises-to-at-least-55-as-freezing-temperatures-and-heavy-snow-wallop-swaths-of-us.html

[32] National Park Service. 2022. *Mount rainier.* https://www.nps.gov/mora/index.htm

[33] National Park Service. 2025. *Mount rainier annual snowfall totals.* https://www.nps.gov/mora/planyourvisit/annual-snowfall-totals.htm

[34] National Park Service. 2025. *Mount rainier frequently asked questions.* https://www.nps.gov/mora/faqs.htm

[35] National Weather Service. 2025. *Measuring snow.* https://www.weather.gov/dvn/snowmeasure

[36] Hoong Kee Ng, Hon Wai Leong, and Ngai Lam Ho. 2004. Efficient algorithm for path-based range query in spatial databases. In *IEEE International Database Engineering and Applications Symposium (IDEAS)*. 334–343.

[37] Niall McCarthy. 2021. *Exploring the red planet is a costly undertaking.* https://www.statista.com/chart/24232/life-cycle-costs-of-mars-missions/

[38] Sebastian Pütz, Thomas Wiemann, Jochen Sprickerhof, and Joachim Hertzberg. 2016. 3D navigation mesh generation for path planning in uneven terrain. *Symposium on Intelligent Autonomous Vehicles (IAV)* 49, 15 (2016), 212–217.

[39] Russell LaDuca. 2020. *What would happen to me if I was buried under snow?* https://qr.ae/prt6zQ

[40] Cyrus Shahabi, Lu-An Tang, and Songhua Xing. 2008. Indexing land surface for efficient knn query. In *International Conference on Very Large Data Bases (VLDB)*, Vol. 1. 1020–1031.

[41] Barak Sober, Robert Ravier, and Ingrid Daubechies. 2020. Approximating the riemannian metric from point clouds via manifold moving least squares. *arXiv preprint arXiv:2007.09885* (2020).

[42] Spatial. 2022. *LiDAR scanning with spatial's ios app.* https://support.spatial.io/hc/en-us/articles/360057387631-LiDAR-Scanning-with-Spatial-s-iOS-App

[43] The Conversation. 2025. *How is snowfall measured? A meteorologist explains how volunteers tally up winter storms.* https://theconversation.com/how-is-snowfall-measured-a-meteorologist-explains-how-volunteers-tally-up-

winter-storms-175628

[44] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, and David M. Mount. 2017. Distance oracle on terrain surface. In *ACM International Conference on Management of Data (SIGMOD)*. 1211–1226.

[45] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, David M Mount, and Hanan Samet. 2022. Proximity queries on terrain surface. *ACM Transactions on Database Systems (TODS)* (2022).

[46] Victor Junqiu Wei, Raymond Chi-Wing Wong, Cheng Long, David M Mount, and Hanan Samet. 2024. On efficient shortest path computation on terrain surface: a direction-oriented approach. *IEEE Transactions on Knowledge & Data Engineering (TKDE)* 1 (2024), 1–14.

[47] Shi-Qing Xin and Guo-Jin Wang. 2009. Improving chen and han's algorithm on the discrete geodesic problem. *ACM Transactions on Graphics (TOG)* 28, 4 (2009), 1–8.

[48] Songhua Xing, Cyrus Shahabi, and Bei Pan. 2009. Continuous monitoring of nearest neighbors on land surface. In *International Conference on Very Large Data Bases (VLDB)*, Vol. 2. 1114–1125.

[49] Da Yan, Zhou Zhao, and Wilfred Ng. 2012. Monochromatic and bichromatic reverse nearest neighbor queries on land surfaces. In *ACM International Conference on Information and Knowledge Management (CIKM)*. 942–951.

[50] Yinzhao Yan and Raymond Chi-Wing Wong. 2021. Path advisor: a multi-functional campus map tool for shortest path. In *International Conference on Very Large Data Bases (VLDB)*, Vol. 14. 2683–2686.

[51] Yinzhao Yan and Raymond Chi-Wing Wong. 2024. Efficient shortest path queries on 3d weighted terrain surfaces for moving objects. In *IEEE International Conference on Mobile Data Management (MDM)*.

[52] Yinzhao Yan and Raymond Chi-Wing Wong. 2024. Proximity queries on point clouds using rapid construction path oracle. In *ACM International Conference on Management of Data (SIGMOD)*, Vol. 2. 1–26.

[53] Yinzhao Yan and Raymond Chi-Wing Wong. 2026. Efficient proximity queries on simplified height maps. In *ACM International Conference on Management of Data (SIGMOD)*, Vol. 3. 1–26.

[54] Yinzhao Yan, Raymond Chi-Wing Wong, and Christian S Jensen. 2024. An efficiently updatable path oracle for terrain surfaces. *IEEE Transactions on Knowledge & Data Engineering (TKDE)* 1 (2024), 1–14.

[55] Hongchuan Yu, Jian J Zhang, and Zheng Jiao. 2014. Geodesics on point clouds. *Mathematical Problems in Engineering (MPE)* (2014).

[56] Zichao Qi, Yanghua Xiao, Bin Shao, and Haixun Wang. 2013. Toward a distance oracle for billion-node graphs. *International Conference on Very Large Data Bases (VLDB)* 7, 1 (2013), 61–72.

[57] Yoones Rezaei and Stephen Lee. 2023. sat2pc: Generating building roof's point cloud from a single 2d satellite images. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. 221–230.

[58] Jagan Sankaranarayanan and Hanan Samet. 2009. Distance oracles for spatial networks. In *IEEE International Conference on Data Engineering (ICDE)*. 652–663.

[59] Jagan Sankaranarayanan, Hanan Samet, and Houman Alborzi. 2009. Path oracles for spatial networks. *International Conference on Very Large Data Bases (VLDB)* 2, 1 (2009), 1210–1221.

[60] Farhan Tauheed, Laurynas Biveinis, Thomas Heinis, Felix Schurmann, Henry Markram, and Anastasia Ailamaki. 2012. Accelerating range queries for brain simulations. In *IEEE International Conference on Data Engineering (ICDE)*. 941–952.

[61] Victor Junqiu Wei, Raymond Chi-Wing Wong, and Cheng Long. 2020. Architecture-intact oracle for fastest path and time queries on dynamic spatial networks. In *ACM International Conference on Management of Data (SIGMOD)*. 1841–1856.

A   ADAPTATION TO AR2P AND AR2AR QUERIES ON *TINS*

Recall that we discussed in the main paper that we can adapt our oracles for A2P and A2A queries on point clouds to AR2P and AR2AR queries on *TINs*. We used *RC-Oracle-Adapt-AR2AR* for illustration. All the theoretical analysis of *RC-Oracle-Adapt-AR2AR* is the same as *RC-Oracle-A2A*. We mainly prove its error bound. We let $\Pi_X(s, t|T)$ be the calculated shortest path of *RC-Oracle-Adapt-AR2AR* passing on a point cloud graph between s and t, where $X = $ *RC-Oracle-Adapt-AR2AR* . Note that the points of this point cloud are formed by the vertices of T and $\Pi_X^*(s, t|T)$ is the exact shortest path passing on this graph between s and t. For the AR2AR query on *TINs*, *RC-Oracle-Adapt-AR2AR* always has $|\Pi_X(s, t|T)| \leq (1 + \epsilon)|\Pi^*(s, t|T)|$ for any pair of vertices s and t on the faces T. This is because we let $p \in V_s$ and $q \in V_t$ be two vertices that lie on the path $\Pi_X(s, t|T)$, so $|\Pi_X(s, t|T)| = |(s, p)| + |\Pi_X(p, q|T)| + |(q, t)| \leq |(s, p')| + |\Pi_X(p', q'|T)| + |(q', t)|$. We let $p' \in V_s$ and $q' \in V_t$ be two vertices that lie on the path $\Pi_X^*(s, t|T)$, so $|\Pi_X^*(s, t|T)| = |(s, p')| + |\Pi_X^*(p', q'|T)| + |(q', t)|$. *RC-Oracle-Adapt-AR2AR* always has $|\Pi_X(p', q'|T)| \leq (1 + \epsilon)|\Pi_X^*(p', q'|T)|$. So, we obtain

$|\Pi_X(s,t|T)| = |(s,p)| + |\Pi_X(p,q|T)| + |(q,t)| \leq |(s,p')| + |\Pi_X(p',q'|T)| + |(q',t)| \leq |(s,p')| + (1+$
$\epsilon)|\Pi_X^*(p',q'|T)| + |(q',t)| \leq (1+\epsilon)|(s,p')| + (1+\epsilon)|\Pi_X^*(p',q'|T)| + (1+\epsilon)|(q',t)| = (1+\epsilon)|\Pi_X^*(s,t|T)|.$

B EMPIRICAL STUDIES

B.1 Experimental Results for *TIN*s
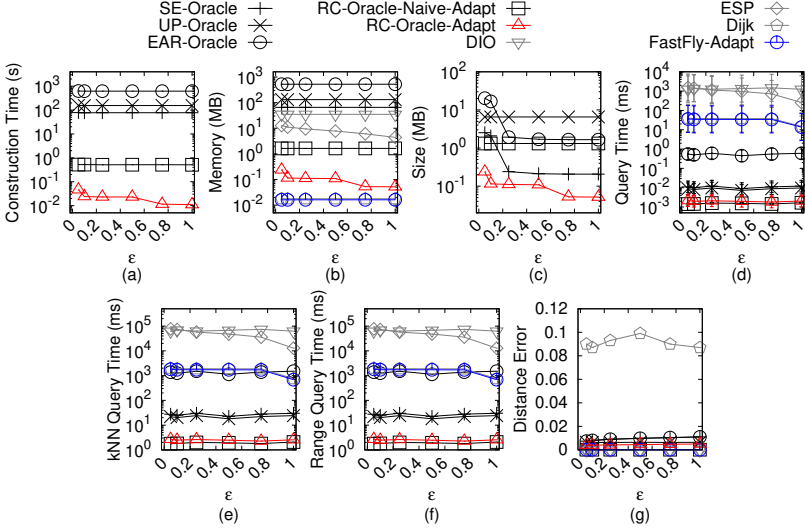
Fig. 25. Baseline comparisons (effect of ϵ on BH_t-*small TIN* dataset for P2P query)
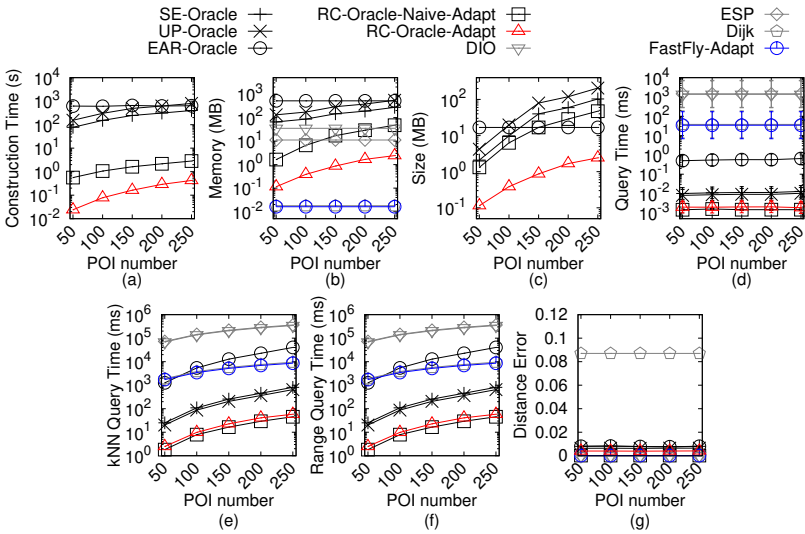
Fig. 26. Baseline comparisons (effect of n on BH_t-*small TIN* dataset for P2P query)

*B.1.1 **Baseline comparisons for the P2P query**. We study the P2P query on *TIN*s. We compared SE-Oracle, UP-Oracle, EAR-Oracle, RC-Oracle-Naive-Adapt, RC-Oracle-Adapt, DIO, ESP, DIJ and*
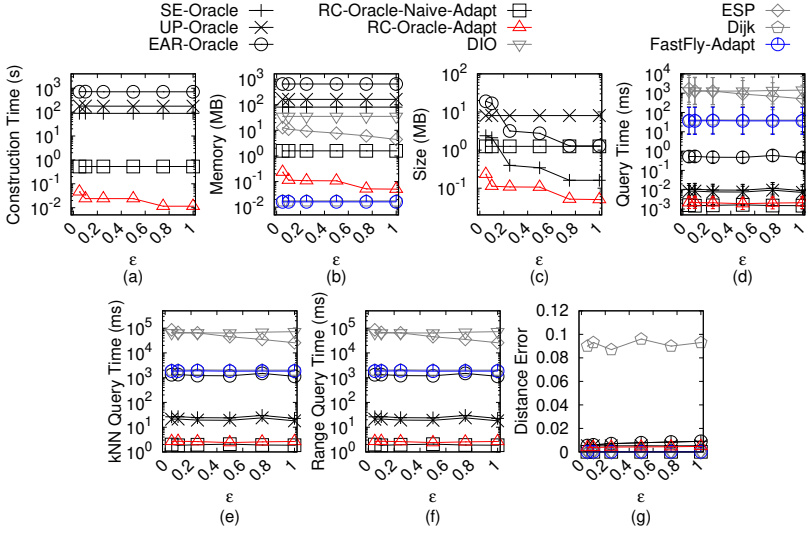
Fig. 27. Baseline comparisons (effect of ϵ on EP_t-small TIN dataset for P2P query)
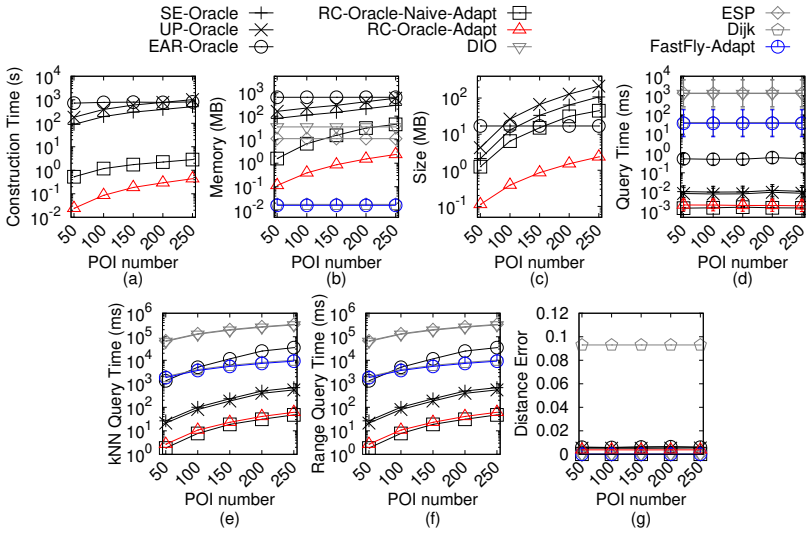
Fig. 28. Baseline comparisons (effect of n on EP_t-small TIN dataset for P2P query)

FastFly-Adapt on small-version datasets with default 50 POIs. We compared *RC-Oracle-Adapt*, *DIO*, *ESP*, *DIJ* and *FastFly-Adapt* on large-version datasets with default 500 POIs. The *kNN* query error and range query error are all equal to 0 for all experiments (since the distance error is very small), so their results are omitted. (1) Figure 25 and Figure 26 show the result on BH_t-small TIN dataset for the P2P query when varying ϵ and n, respectively. (2) Figure 27, Figure 28 and Figure 29 show the result on EP_t-small TIN dataset for the P2P query when varying ϵ, n and N, respectively. (3) Figure 30 and Figure 31 show the result on GF_t-small TIN dataset for the P2P query when varying ϵ and n, respectively. (4) Figure 32 and Figure 33 show the result on LM_t-small TIN dataset for the P2P query when varying ϵ and n, respectively. (5) Figure 34 and Figure 35 show the result
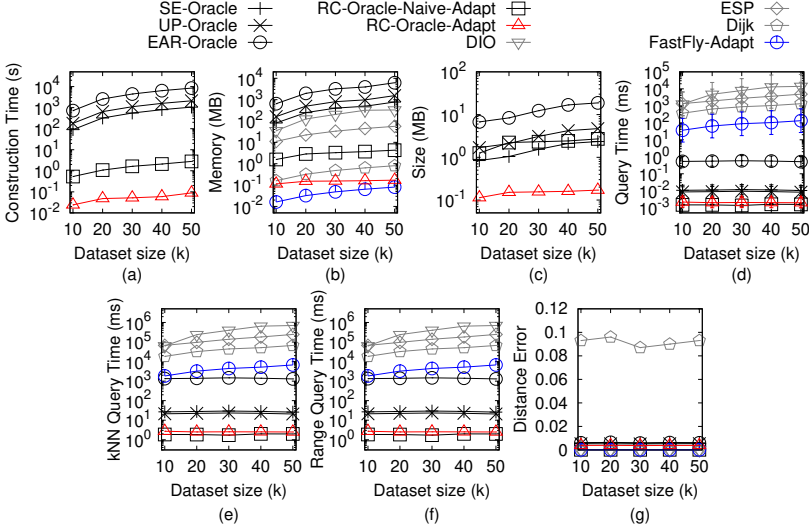
Fig. 29. Baseline comparisons (effect of N on EP_t-small TIN dataset for P2P query)
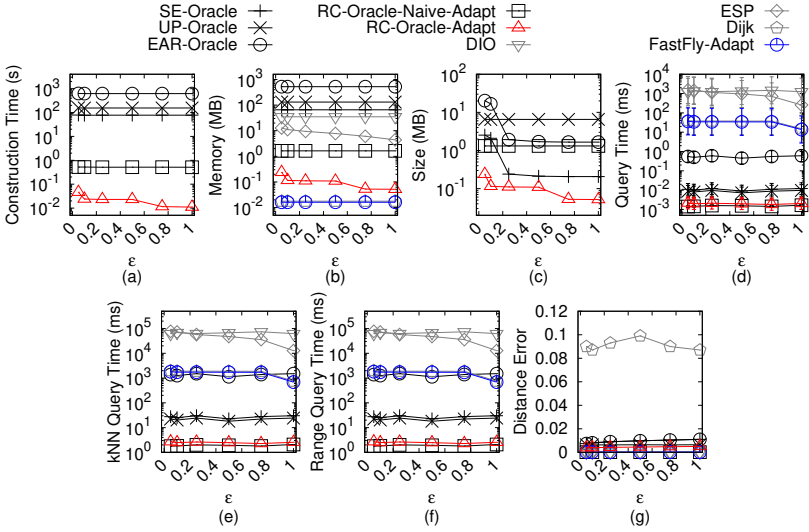
Fig. 30. Baseline comparisons (effect of ϵ on GF_t-small TIN dataset for P2P query)

on RM_t-small TIN dataset for the P2P query when varying ϵ and n, respectively. (6) Figure 36, Figure 37 and Figure 38 show the result on BH_t TIN dataset for the P2P query when varying ϵ, n and N, respectively. (6) Figure 36, Figure 37 and Figure 38 show the result on BH_t TIN dataset for the P2P query when varying ϵ, n and N, respectively. (7) Figure 39, Figure 40 and Figure 41 show the result on EP_t TIN dataset for the P2P query when varying ϵ, n and N, respectively. (8) Figure 42, Figure 43 and Figure 44 show the result on GF_t TIN dataset for the P2P query when varying ϵ, n and N, respectively. (9) Figure 45, Figure 46 and Figure 47 show the result on LM_t TIN dataset for the P2P query when varying ϵ, n and N, respectively. (10) Figure 48, Figure 49 and Figure 50 show the result on RM_t TIN dataset for the P2P query when varying ϵ, n and N, respectively.
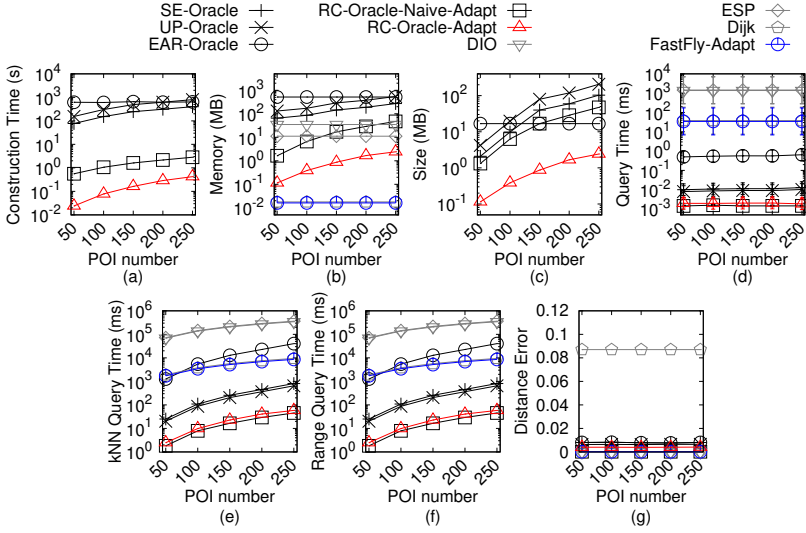
Fig. 31. Baseline comparisons (effect of n on GF_t-small TIN dataset for P2P query)
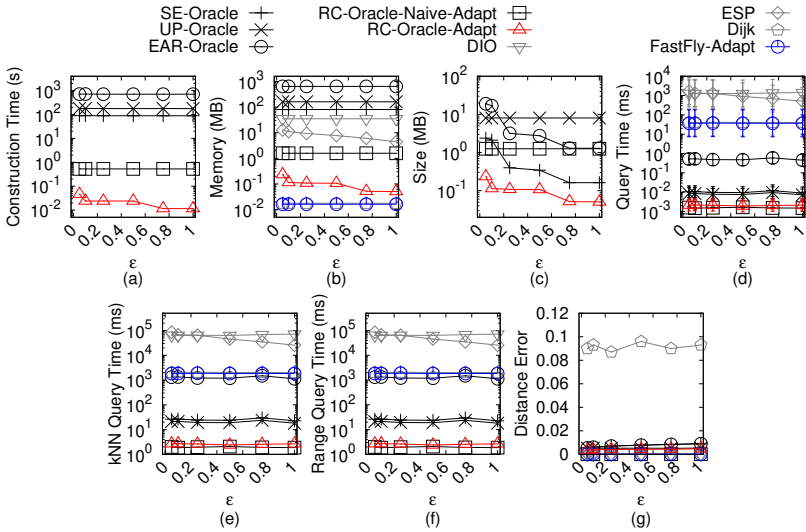
Fig. 32. Baseline comparisons (effect of ϵ on LM_t-small TIN dataset for P2P query)

Effect of ϵ. In Figure 25, Figure 27, Figure 30, Figure 32 and Figure 34, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_t-small, EP_t-small, GF_t-small, LM_t-small and RM_t-small dataset by setting N to be 10k and n to be 50. In Figure 36, Figure 39, Figure 42, Figure 45 and Figure 48, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting N to be 0.5M and n to be 500. Even though varying ϵ will not affect *RC-Oracle-Adapt* a lot, *RC-Oracle-Adapt* still performs much better than the best-known oracle *SE-Oracle*, and other algorithms / oracles.

Effect of n. In Figure 26, Figure 28, Figure 31, Figure 33 and Figure 35, we tested 5 values of n from $\{50, 100, 150, 200, 250\}$ on BH_t-small, EP_t-small, GF_t-small, LM_t-small and RM_t-small dataset
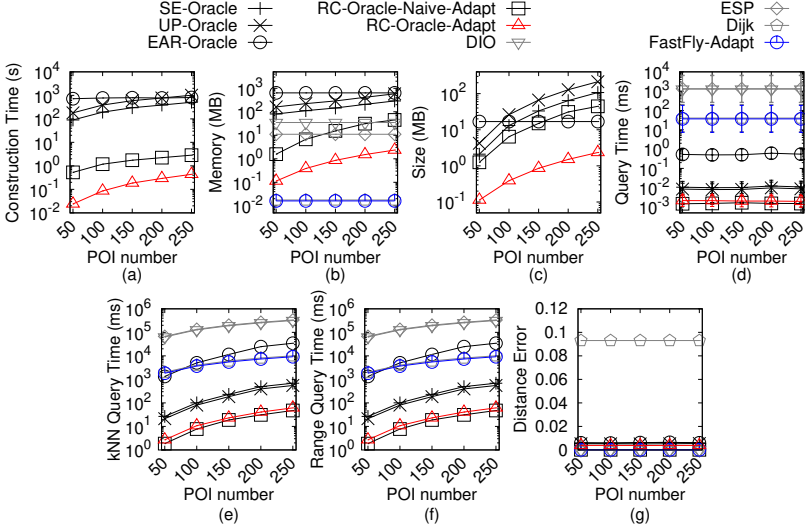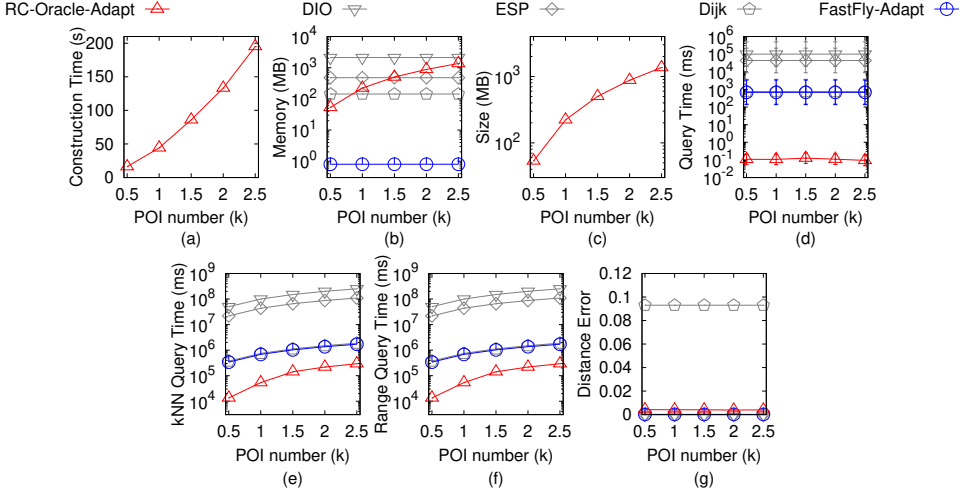
Fig. 33. Baseline comparisons (effect of n on LM_t-small TIN dataset for P2P query)
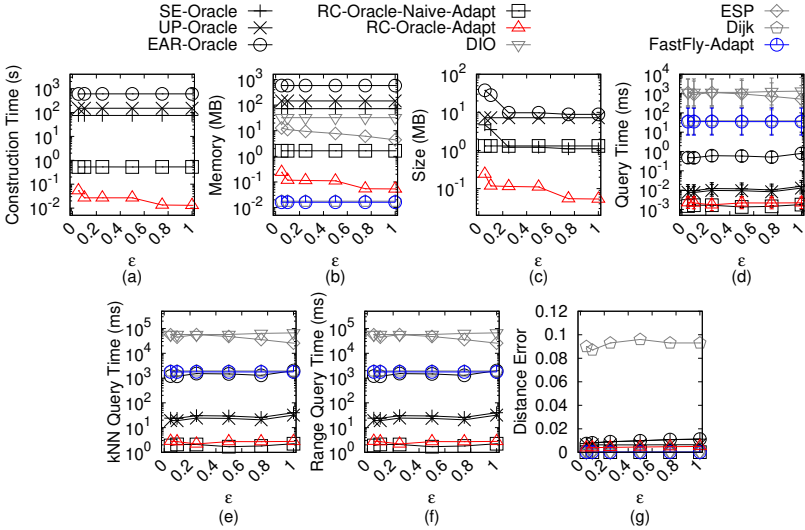
Fig. 34. Baseline comparisons (effect of ϵ on RM_t-small TIN dataset for P2P query)

by setting N to be 10k and ϵ to be 0.1. In Figure 37, Figure 40, Figure 43, Figure 46 and Figure 49, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting N to be 0.5M and ϵ to be 0.25. The oracle construction time and shortest path query time for *SE-Oracle* and *UP-Oracle* are large compared with *RC-Oracle-Adapt*, which shows the superior performance of *RC-Oracle-Adapt* in terms of the oracle construction and shortest path querying.

Effect of N (scalability test). In Figure 29, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_t-small dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 38, Figure 41, Figure 44, Figure 47 and Figure 50, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_t-small, EP_t-small, GF_t-small, LM_t-small and RM_t-small dataset by setting n to be 500 and ϵ to be
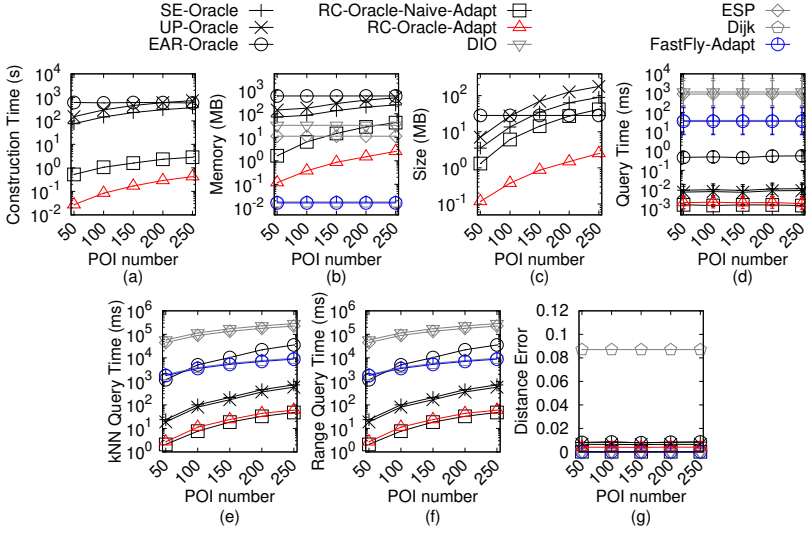
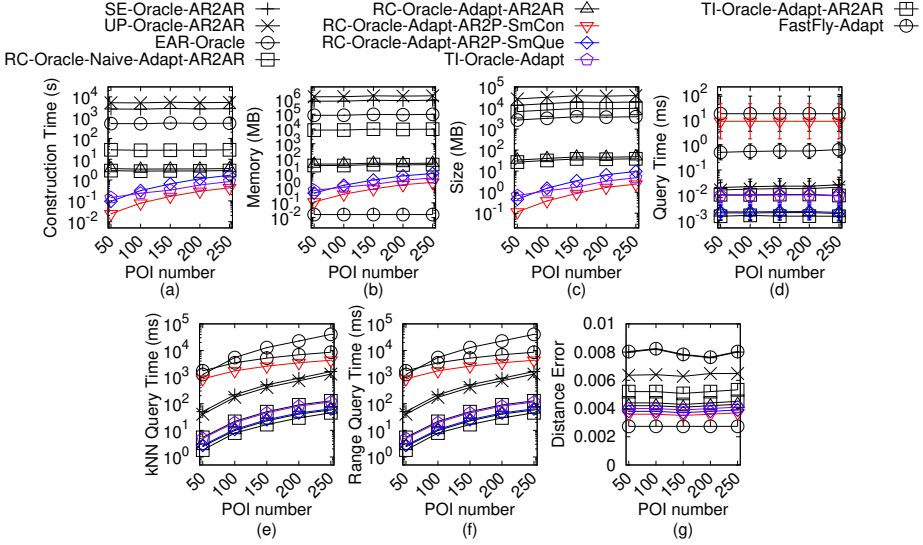Fig. 35. Baseline comparisons (effect of n on RM_t-small TIN dataset for P2P query)
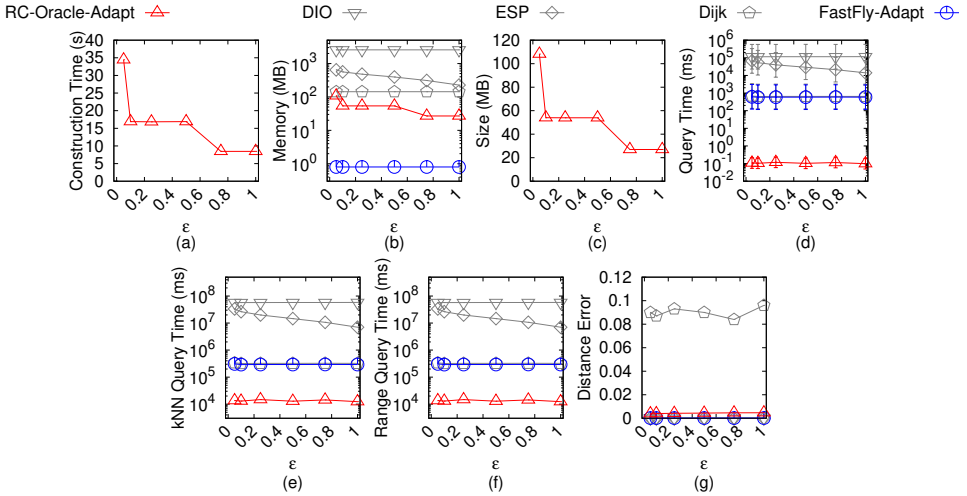
Fig. 36. Baseline comparisons (effect of ϵ on BH_t TIN dataset for P2P query)

0.25 for scalability test. *RC-Oracle-Adapt* performs better than all the remaining algorithms in terms of the oracle construction time, oracle size and shortest path query time. The shortest path query time of *FastFly-Adapt* is 100 times smaller than that of *DIO*, and the distance error of *FastFly-Adapt* (with distance error close to 0) is much smaller than that of *DIJ* (with distance error 0.03).

B.1.2 ***Baseline comparisons for the AR2P query****. We study the AR2P query on *TIN*s. We compared *SE-Oracle-AR2AR, UP-Oracle-AR2AR, EAR-Oracle, RC-Oracle-Naive-Adapt-AR2AR, RC-Oracle-Adapt-AR2AR, RC-Oracle-Adapt-AR2P-SmCon, RC-Oracle-Adapt-AR2P-SmQue, TI-Oracle-Adapt, TI-Oracle-Adapt-AR2AR* and *FastFly-Adapt* on small-version datasets with default 50 POIs.
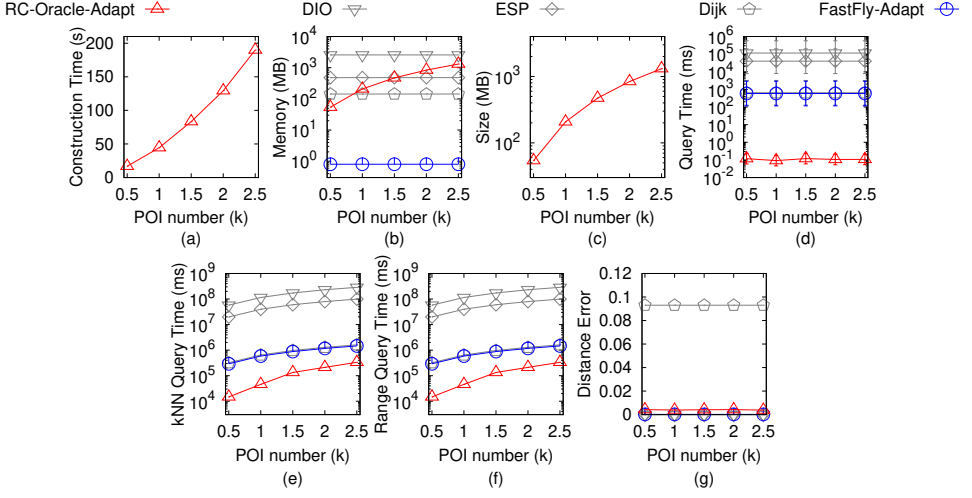
Fig. 37. Baseline comparisons (effect of n on BH_t TIN dataset for P2P query)
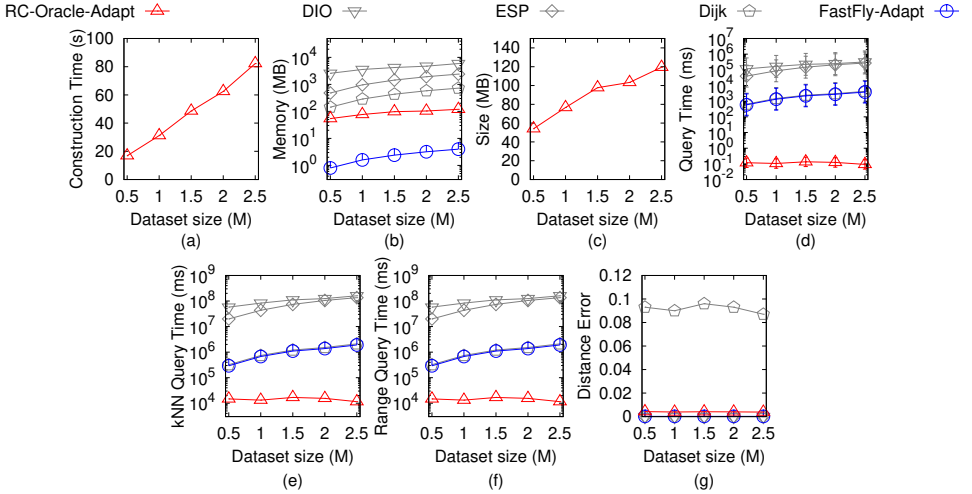
Fig. 38. Baseline comparisons (effect of N on BH_t TIN dataset for P2P query)

We compared *RC-Oracle-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2P-SmCon*, *RC-Oracle-Adapt-AR2P-SmQue*, *TI-Oracle-Adapt*, *TI-Oracle-Adapt-AR2AR* and *FastFly-Adapt* on large-version datasets with default 500 POIs. The *kNN* query error and range query error are all equal to 0 for all experiments (since the distance error is very small), so their results are omitted. (1) Figure 51 and Figure 52 show the result on BH_t-*small TIN* dataset for the AR2P query when varying ϵ and n, respectively. (2) Figure 53, Figure 54 and Figure 55 show the result on EP_t-*small TIN* dataset for the AR2P query when varying ϵ, n and N, respectively. (3) Figure 56 and Figure 57 show the result on GF_t-*small TIN* dataset for the AR2P query when varying ϵ and n, respectively. (4) Figure 58 and Figure 59 show the result on LM_t-*small TIN* dataset for the AR2P query when varying ϵ and n, respectively. (5) Figure 60 and Figure 61 show the result on RM_t-*small TIN* dataset for the AR2P query when varying
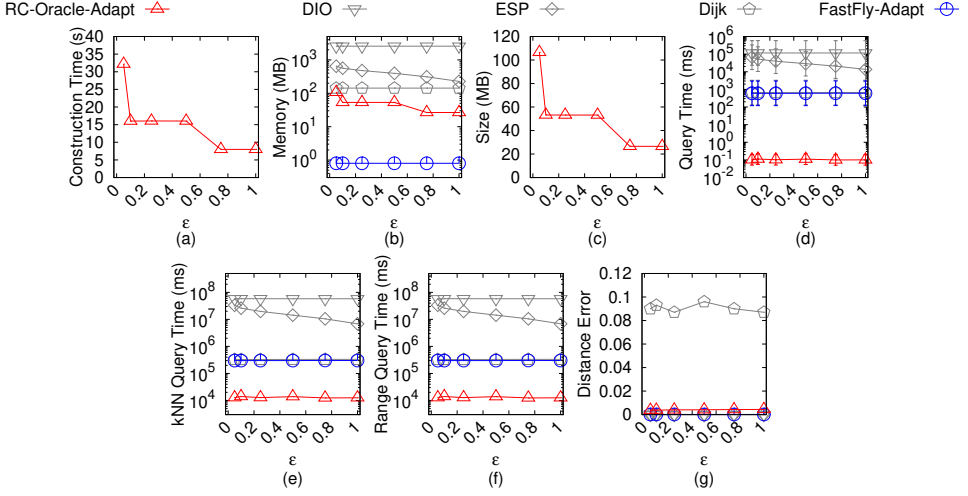
Fig. 39. Baseline comparisons (effect of ϵ on EP_t TIN dataset for P2P query)
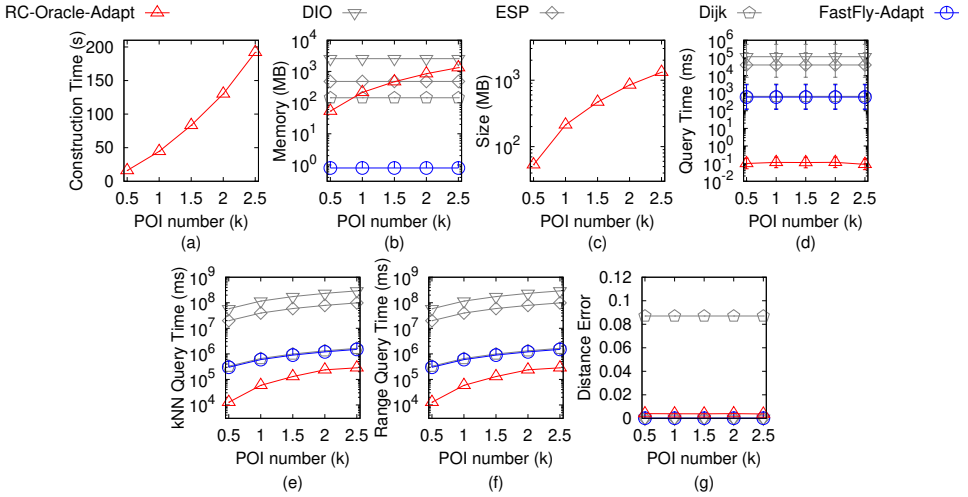
Fig. 40. Baseline comparisons (effect of n on EP_t TIN dataset for P2P query)

ϵ and n, respectively. (6) Figure 62, Figure 63 and Figure 64 show the result on BH_t TIN dataset for the AR2P query when varying ϵ, n and N, respectively. (6) Figure 62, Figure 63 and Figure 64 show the result on BH_t TIN dataset for the AR2P query when varying ϵ, n and N, respectively. (7) Figure 65, Figure 66 and Figure 67 show the result on EP_t TIN dataset for the AR2P query when varying ϵ, n and N, respectively. (8) Figure 68, Figure 69 and Figure 70 show the result on GF_t TIN dataset for the AR2P query when varying ϵ, n and N, respectively. (9) Figure 71, Figure 72 and Figure 73 show the result on LM_t TIN dataset for the AR2P query when varying ϵ, n and N, respectively. (10) Figure 74, Figure 75 and Figure 76 show the result on RM_t TIN dataset for the AR2P query when varying ϵ, n and N, respectively.
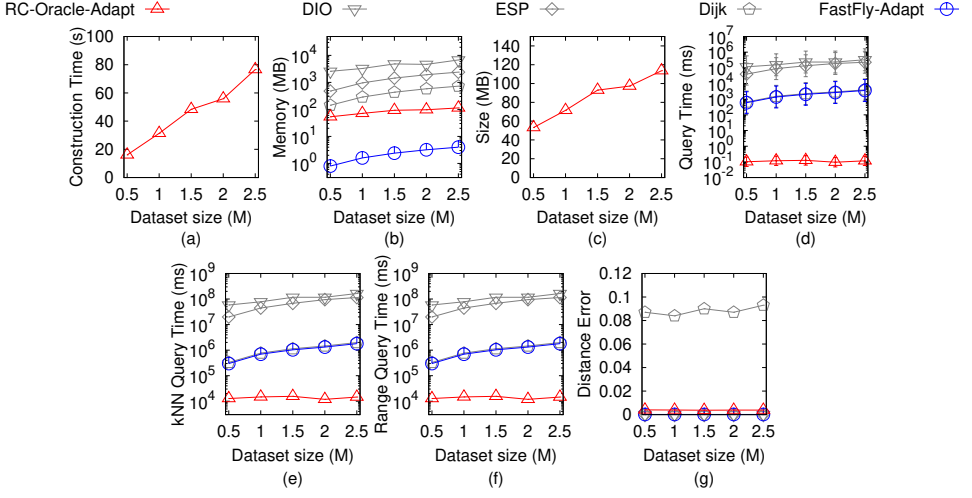
Fig. 41. Baseline comparisons (effect of N on EP_t TIN dataset for P2P query)
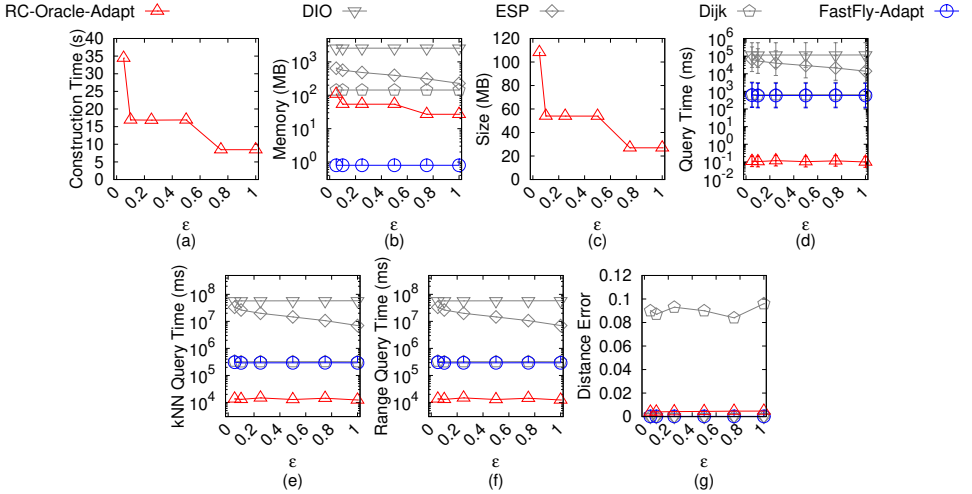
Fig. 42. Baseline comparisons (effect of ϵ on GF_t TIN dataset for P2P query)

Effect of ϵ. In Figure 51, Figure 53, Figure 56, Figure 58 and Figure 60, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_t-small, EP_t-small, GF_t-small, LM_t-small and RM_t-small dataset by setting N to be 10k and n to be 50. In Figure 62, Figure 65, Figure 68, Figure 71 and Figure 74, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting N to be 0.5M and n to be 500. The oracle construction time, memory usage and oracle size of *RC-Oracle-Adapt-AR2P-SmCon* is the smallest in all oracles since it has the same oracle construction process as *RC-Oracle-Adapt*. But, the shortest path query time of *RC-Oracle-Adapt-AR2P-SmCon* is larger than other oracles (but still smaller than *FastFly-Adapt*) since it can terminate earlier when using *FastFly-Adapt* in the shortest path query phase. Thus, it performs well in the case of fewer proximity queries. The oracle construction time of *RC-Oracle-Adapt-AR2P-SmQue* and
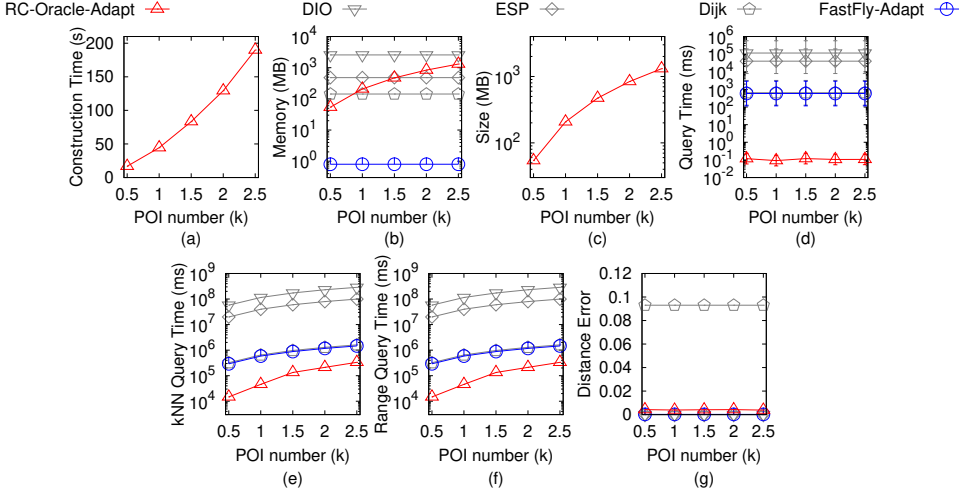
Fig. 43. Baseline comparisons (effect of n on GF_t TIN dataset for P2P query)
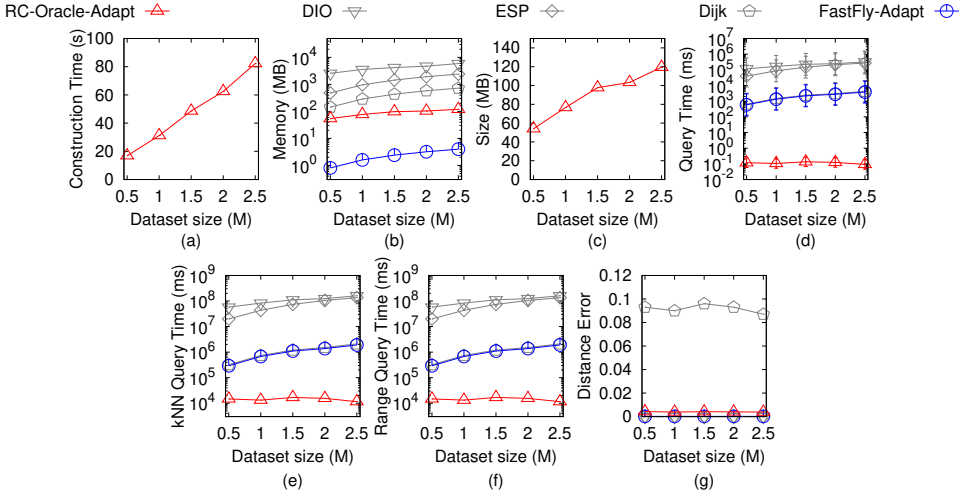
Fig. 44. Baseline comparisons (effect of N on GF_t TIN dataset for P2P query)

TI-Oracle-Adapt are also very small and their shortest path query time are also very small due to their earlier termination during oracle construction and tight information stored in the oracles.

Effect of n. In Figure 52, Figure 54, Figure 57, Figure 59 and Figure 61, we tested 5 values of n from {50, 100, 150, 200, 250} on BH_t-*small*, EP_t-*small*, GF_t-*small*, LM_t-*small* and RM_t-*small* dataset by setting N to be 10k and ϵ to be 0.1. In Figure 63, Figure 66, Figure 69, Figure 72 and Figure 75, we tested 5 values of n from {500, 1000, 1500, 2000, 2500} on BH_t, EP_t, GF_t, LM_t and RM_t dataset by setting N to be 0.5M and ϵ to be 0.25. When $n < 100$ (resp. $n \geq 100$), the oracle construction time of *RC-Oracle-Adapt-AR2P-SmQue* is smaller (resp. larger) than that of *TI-Oracle-Adapt*, and it verifies our claim that the former (resp. latter) one performs well when the density of POIs is high (resp. low).
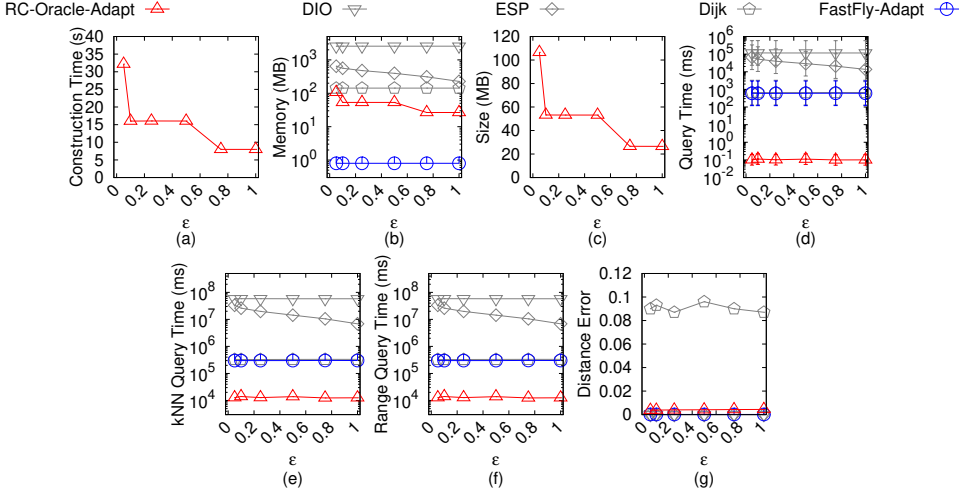
Fig. 45. Baseline comparisons (effect of ϵ on LM_t TIN dataset for P2P query)
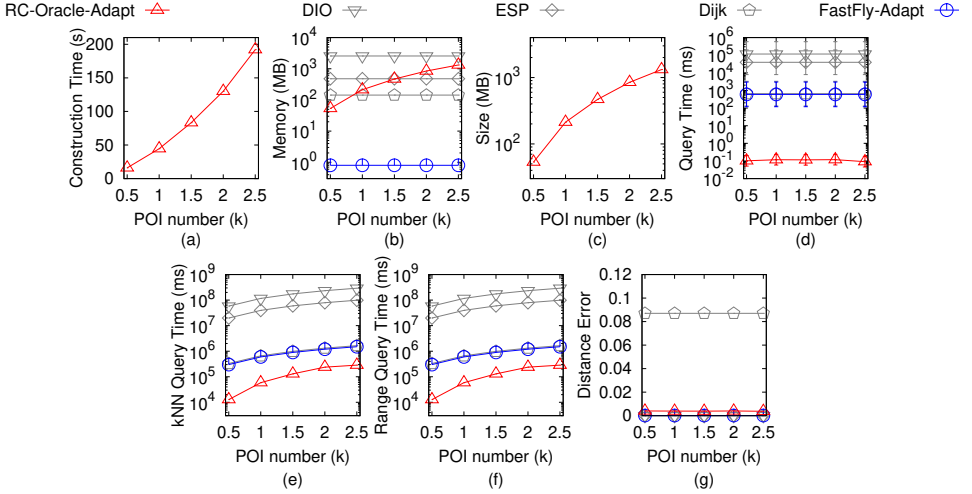
Fig. 46. Baseline comparisons (effect of n on LM_t TIN dataset for P2P query)

Effect of N (scalability test). In Figure 55, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_t-*small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 64, Figure 67, Figure 70, Figure 73 and Figure 76, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_t-*small*, EP_t-*small*, GF_t-*small*, LM_t-*small* and RM_t-*small* dataset by setting n to be 500 and ϵ to be 0.25 for scalability test. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* are scalable when N is large.

B.1.3 **Baseline comparisons for the AR2AR query**. We study the AR2AR query on *TIN*s. In the same figures of baseline comparisons for the AR2P query on *TIN*s, we compared *SE-Oracle-AR2AR*, *UP-Oracle-AR2AR*, *EAR-Oracle*, *RC-Oracle-Naive-Adapt-AR2AR*, *RC-Oracle-Adapt-AR2AR* and *TI-Oracle-Adapt-AR2AR*. The last two oracles still perform better than the first two oracles
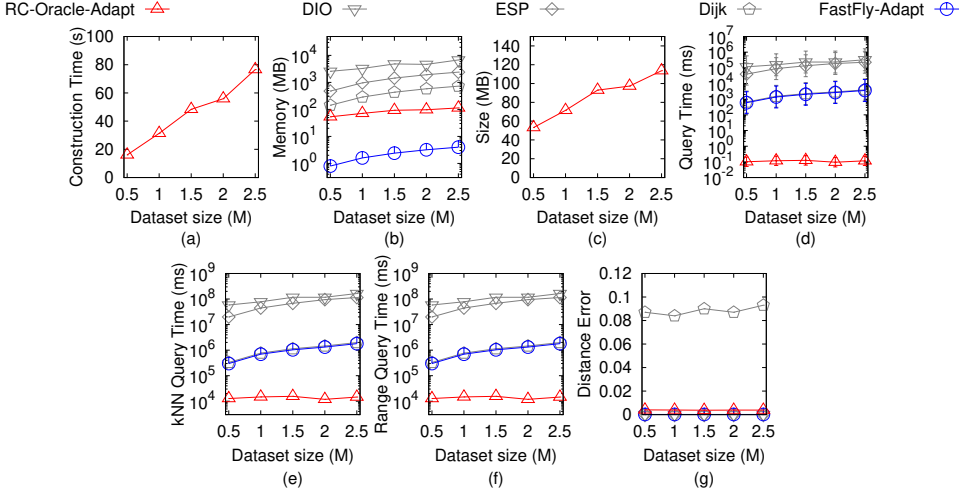
Fig. 47. Baseline comparisons (effect of N on LM_t TIN dataset for P2P query)
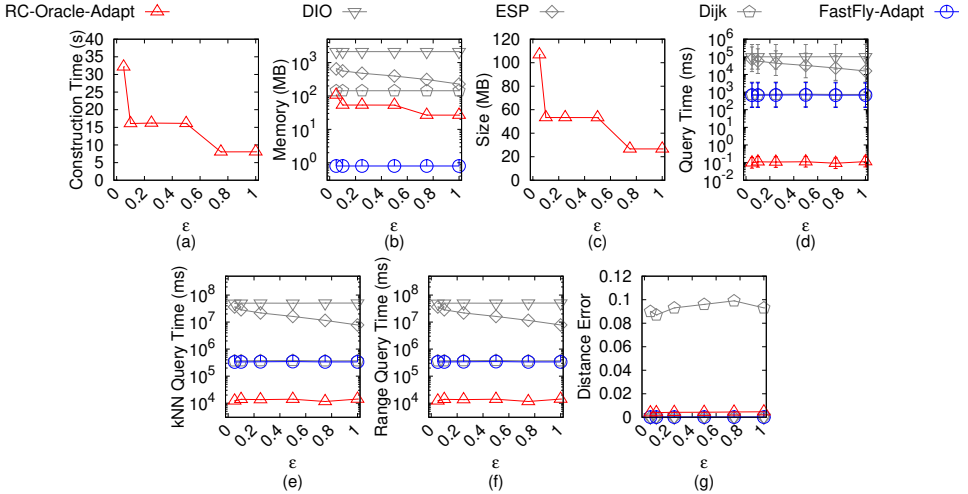
Fig. 48. Baseline comparisons (effect of ϵ on RM_t TIN dataset for P2P query)

in terms of oracle construction time, oracle size and shortest path query time due to their earlier termination during oracle construction and tight information stored in the oracles.

B.2 Experimental Results for Point Clouds

B.2.1 ***Baseline comparisons for the P2P query***. We study the P2P query on *point clouds* for baseline comparisons. We compared *SE-Oracle-Adapt*, *UP-Oracle-Adapt*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive*, *RC-Oracle*, *DIO-Adapt*, *ESP-Adapt*, *DIJ-Adapt* and *FastFly* on small-version datasets with default 50 POIs. We compared *RC-Oracle*, *DIO-Adapt*, *ESP-Adapt*, *DIJ-Adapt* and *FastFly* on large-version datasets with default 500 POIs. (1) Figure 77 and Figure 78 show the result on BH_c-*small* point cloud dataset for the P2P query when varying ϵ and n, respectively. (2) Figure 79,

Fig. 49. Baseline comparisons (effect of n on RM_t TIN dataset for P2P query)
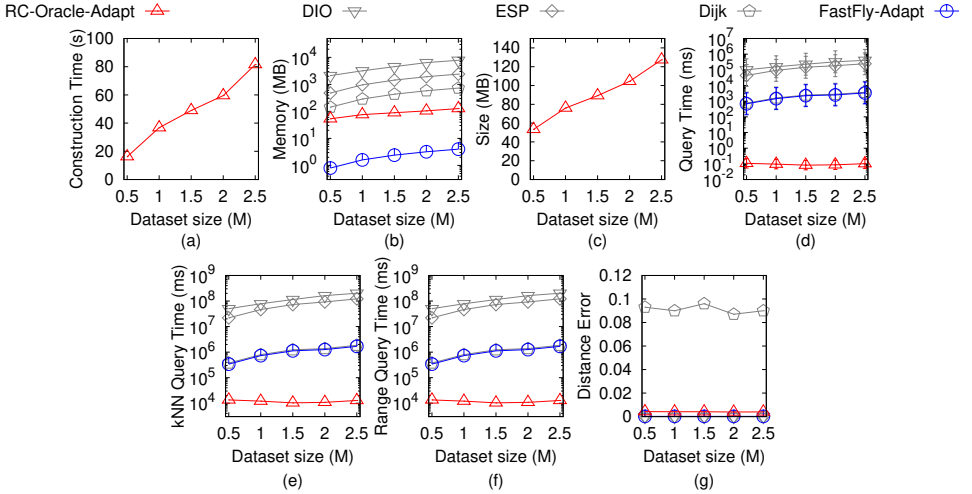
Fig. 50. Baseline comparisons (effect of N on RM_t TIN dataset for P2P query)

Figure 80 and Figure 81 show the result on EP_c-*small* point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (3) Figure 82 and Figure 83 show the result on GF_c-*small* point cloud dataset for the P2P query when varying ϵ and n, respectively. (4) Figure 84 and Figure 85 show the result on LM_c-*small* point cloud dataset for the P2P query when varying ϵ and n, respectively. (5) Figure 86 and Figure 87 show the result on RM_c-*small* point cloud dataset for the P2P query when varying ϵ and n, respectively. (6) Figure 88, Figure 89 and Figure 90 show the result on BH_c point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (7) Figure 91, Figure 92 and Figure 93 show the result on EP_c point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (8) Figure 94, Figure 95 and Figure 96 show the result on GF_c point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (9) Figure 97, Figure 98 and Figure 99 show
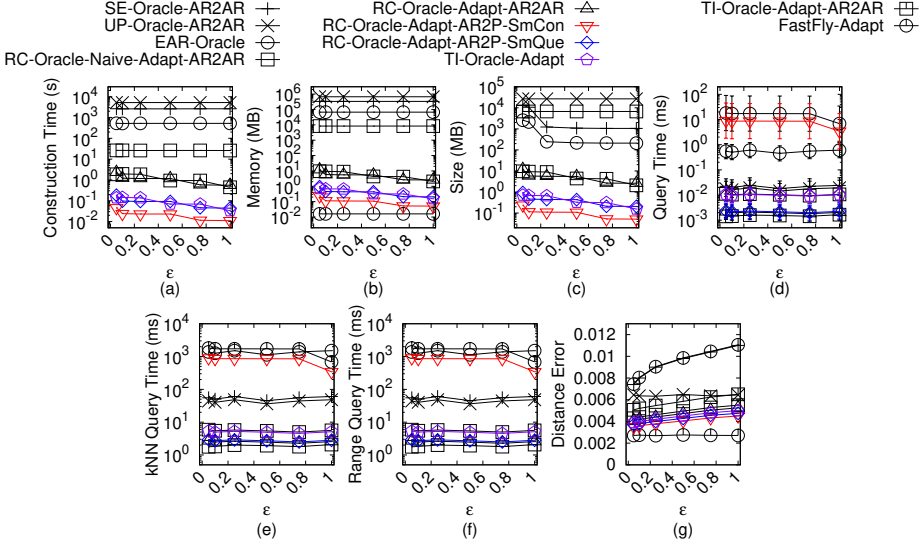
Fig. 51. Baseline comparisons (effect of ϵ on BH_t-small TIN dataset for AR2P query)

Fig. 52. Baseline comparisons (effect of n on BH_t-small TIN dataset for AR2P query)

the result on LM_c point cloud dataset for the P2P query when varying ϵ, n and N, respectively. (10) Figure 100, Figure 101 and Figure 102 show the result on RM_c point cloud dataset for the P2P query when varying ϵ, n and N, respectively.

Effect of ϵ. In Figure 77, Figure 79, Figure 82, Figure 84 and Figure 86, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_c-small, EP_c-small, GF_c-small, LM_c-small and RM_c-small dataset by setting N to be 10k and n to be 50. In Figure 88, Figure 91, Figure 94, Figure 97 and Figure 100, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_c, EP_c, GF_c, LM_c and RM_c dataset by setting N to be 0.5M and n to be 500. Even though varying ϵ will not affect *RC-Oracle* a
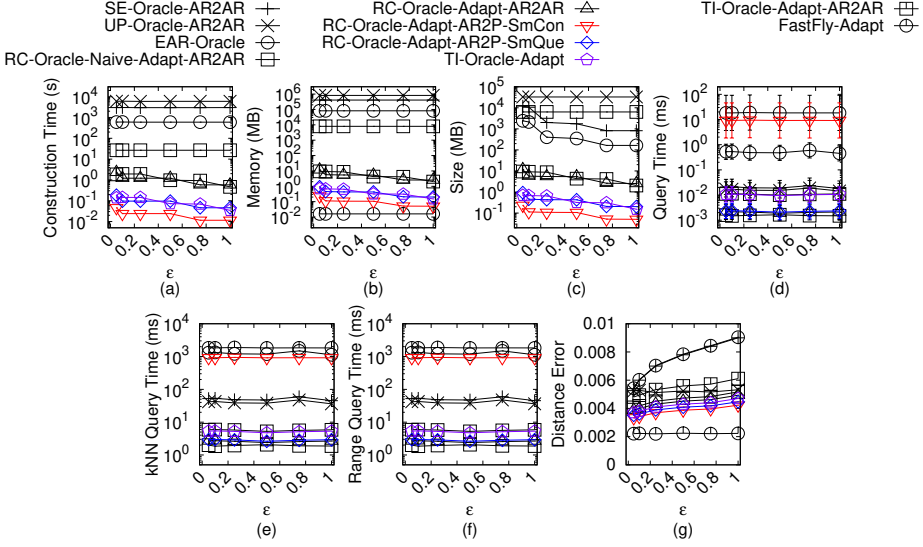
Fig. 53. Baseline comparisons (effect of ϵ on EP_t-small TIN dataset for AR2P query)
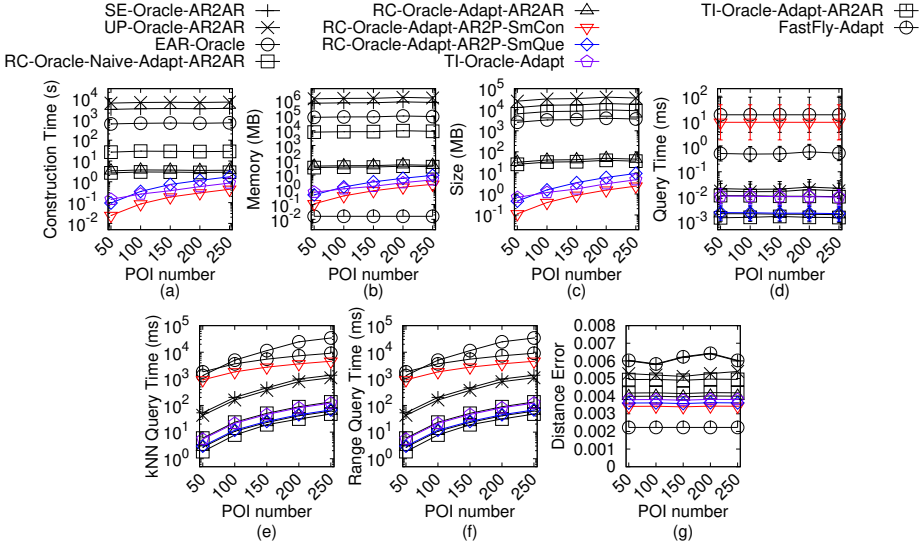
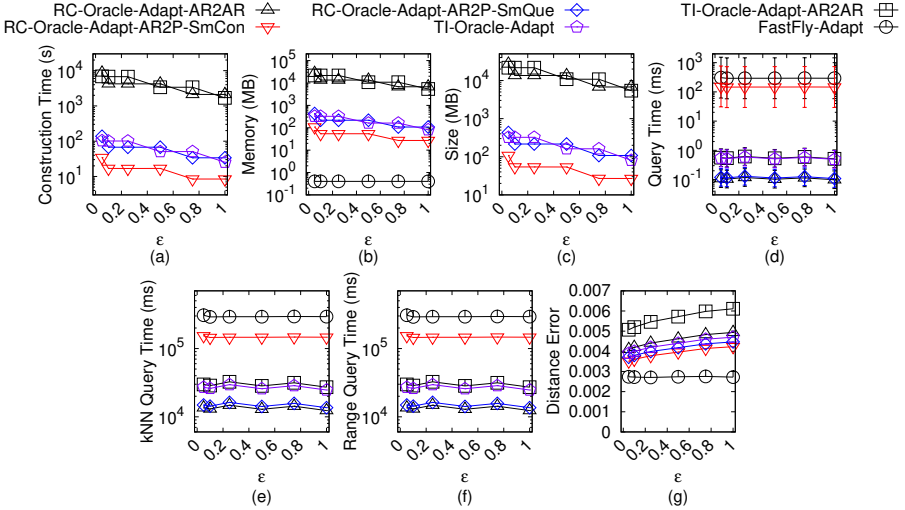Fig. 54. Baseline comparisons (effect of n on EP_t-small TIN dataset for AR2P query)

lot, *RC-Oracle* still performs much better than the best-known adapted oracle *SE-Oracle-Adapt*, and other algorithms / oracles.

Effect of n. In Figure 78, Figure 80, Figure 83, Figure 85 and Figure 87, we tested 5 values of n from $\{50, 100, 150, 200, 250\}$ on BH_c-small, EP_c-small, GF_c-small, LM_c-small and RM_c-small dataset by setting N to be 10k and ϵ to be 0.1. In Figure 89, Figure 92, Figure 95, Figure 98 and Figure 101, we tested 5 values of n from $\{500, 1000, 1500, 2000, 2500\}$ on BH_c, EP_c, GF_c, LM_c and RM_c dataset by setting N to be 0.5M and ϵ to be 0.25. The oracle construction time and shortest path query time
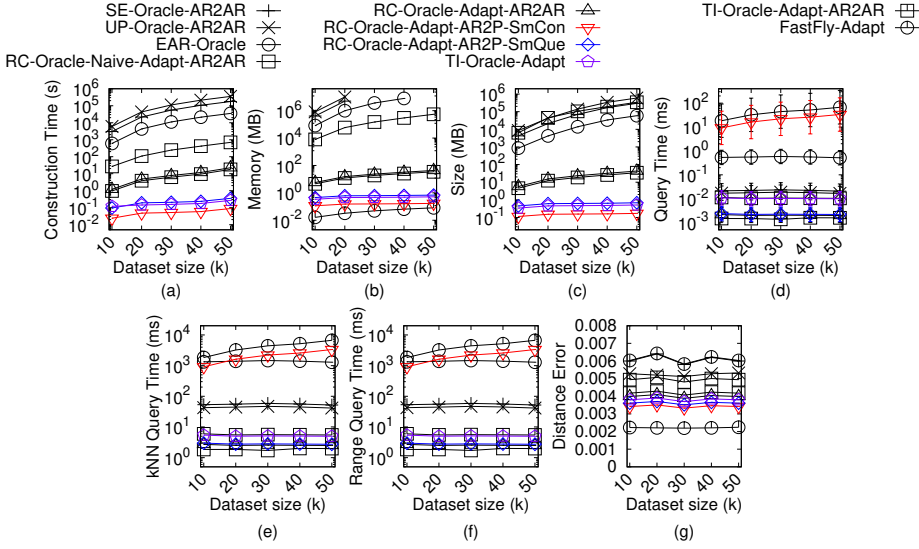
Fig. 55. Baseline comparisons (effect of N on EP_t-small TIN dataset for AR2P query)
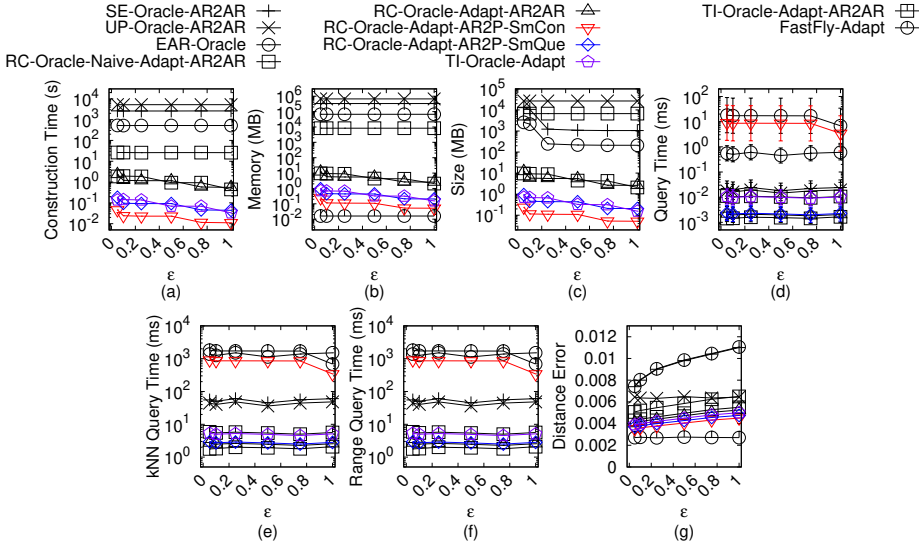
Fig. 56. Baseline comparisons (effect of ϵ on GF_t-small TIN dataset for AR2P query)

for *SE-Oracle-Adapt* and *UP-Oracle-Adapt* are large compared with *RC-Oracle*, which shows the superior performance of *RC-Oracle* in terms of the oracle construction and shortest path query.

Effect of N (scalability test). In Figure 81, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_c-small dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 90, Figure 93, Figure 96, Figure 99 and Figure 102, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_c-small, EP_c-small, GF_c-small, LM_c-small and RM_c-small dataset by setting n to be 500 and ϵ to be 0.25 for scalability test. *RC-Oracle* performs better than all the remaining algorithms in terms of the oracle construction time, oracle size and shortest path query time.
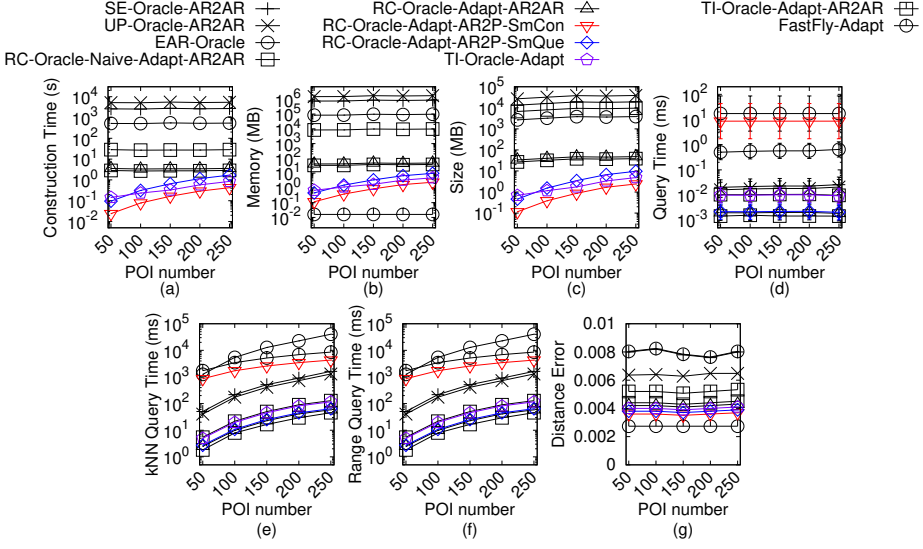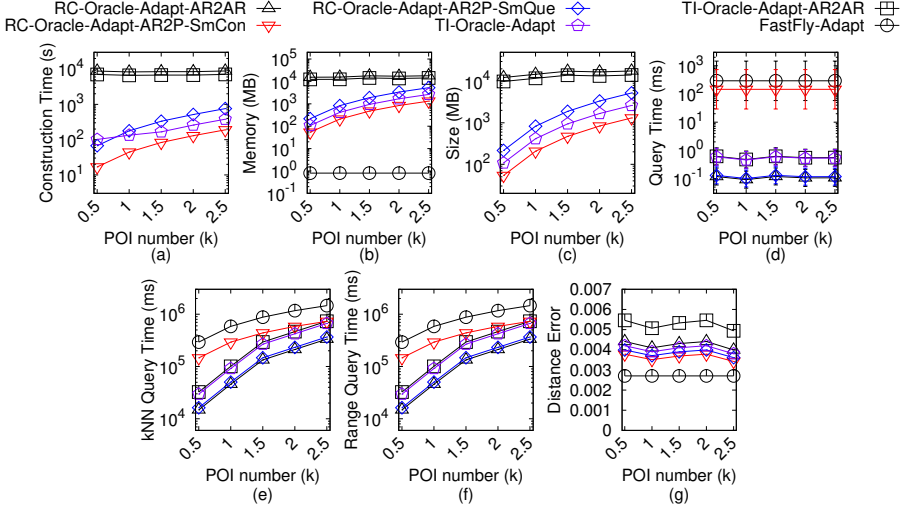
Fig. 57. Baseline comparisons (effect of n on GF_t-small TIN dataset for AR2P query)
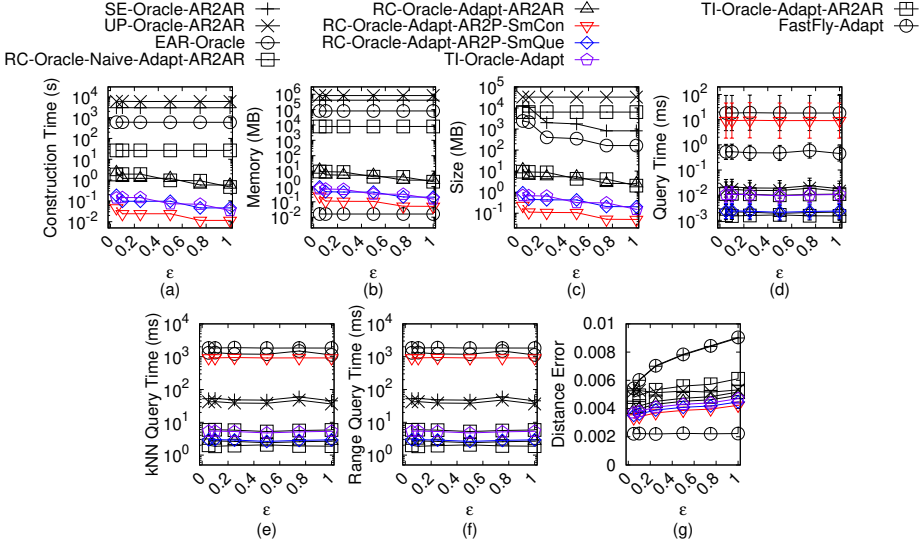
Fig. 58. Baseline comparisons (effect of ϵ on LM_t-small TIN dataset for AR2P query)

B.2.2 **Ablation study for the P2P query.** We study the P2P query on *point clouds* for ablation study. We compared *SE-Oracle-FastFly-Adapt*, *UP-Oracle-FastFly-Adapt*, *EAR-Oracle-FastFly-Adapt* and *RC-Oracle*.

In Figure 103, Figure 104, Figure 105, Figure 106 and Figure 107, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_c, EP_c, GF_c, LM_c and RM_c dataset by setting N to be 0.5M and n to be 500. The oracle construction time, oracle size and shortest path query time of *RC-Oracle* perform better than *SE-Oracle-FastFly-Adapt*, *UP-Oracle-FastFly-Adapt* and *EAR-Oracle-FastFly-Adapt*, which shows the usefulness of the oracle part of *RC-Oracle*.
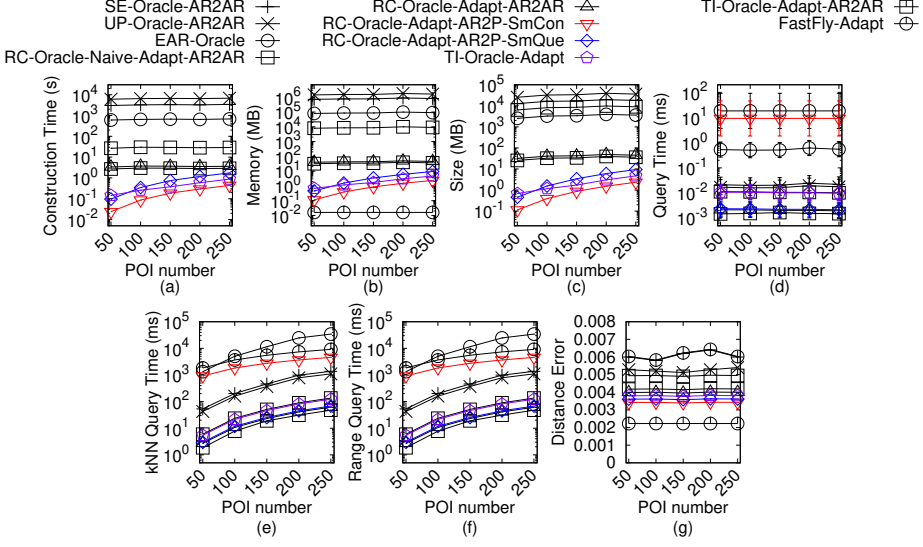
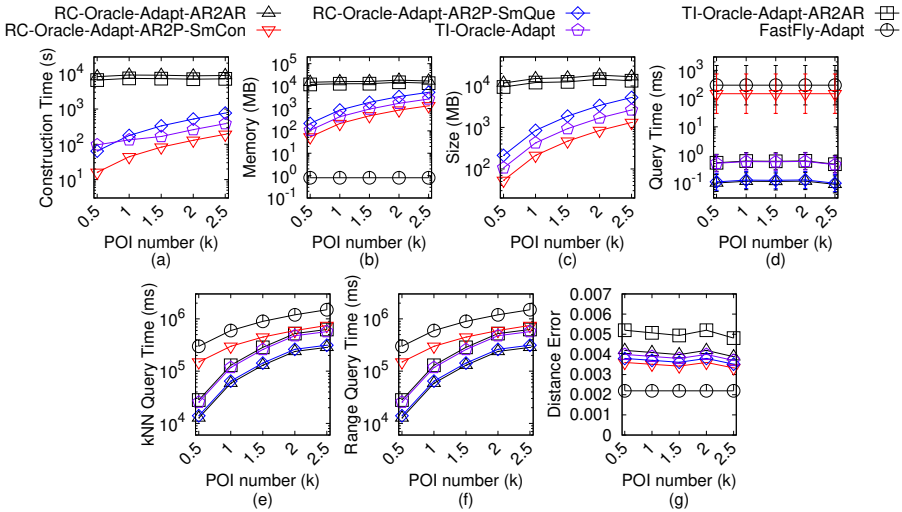Fig. 59. Baseline comparisons (effect of n on LM_t-small TIN dataset for AR2P query)
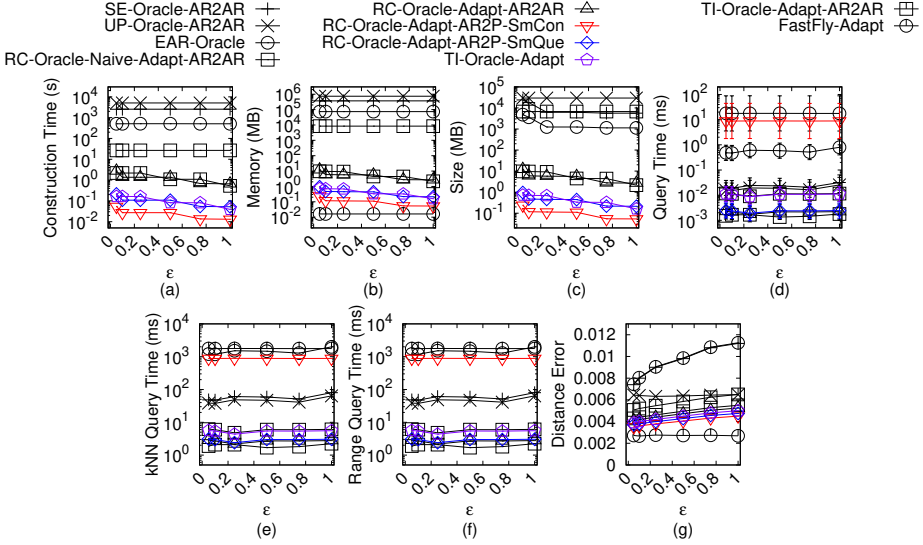
Fig. 60. Baseline comparisons (effect of ϵ on RM_t-small TIN dataset for AR2P query)

B.2.3 Comparisons with other proximity queries oracles and variation oracles for the P2P query.
We study the P2P query on *point clouds* for comparisons with other proximity queries oracles and variation oracles. We denote *RC-Oracle-NaiveProx* to be *RC-Oracle* using the naive proximity query algorithm. We compared *SU-Oracle-Adapt*, *RC-Oracle-NaiveProx* and *RC-Oracle*.

In Figure 108, Figure 109, Figure 110, Figure 111 and Figure 112, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_c-small, EP_c-small, GF_c-small, LM_c-small and RM_c-small dataset by setting N to be 10k and n to be 50. The oracle construction time, oracle size and *kNN* query time of *RC-Oracle* perform better than *SU-Oracle-Adapt* and *RC-Oracle-NaiveProx*. For a point cloud
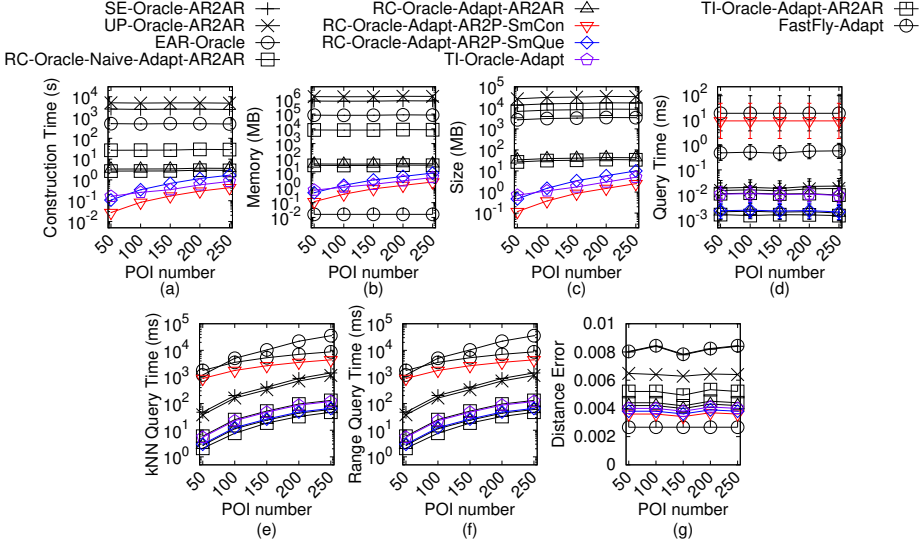
Fig. 61. Baseline comparisons (effect of n on RM_t-small TIN dataset for AR2P query)
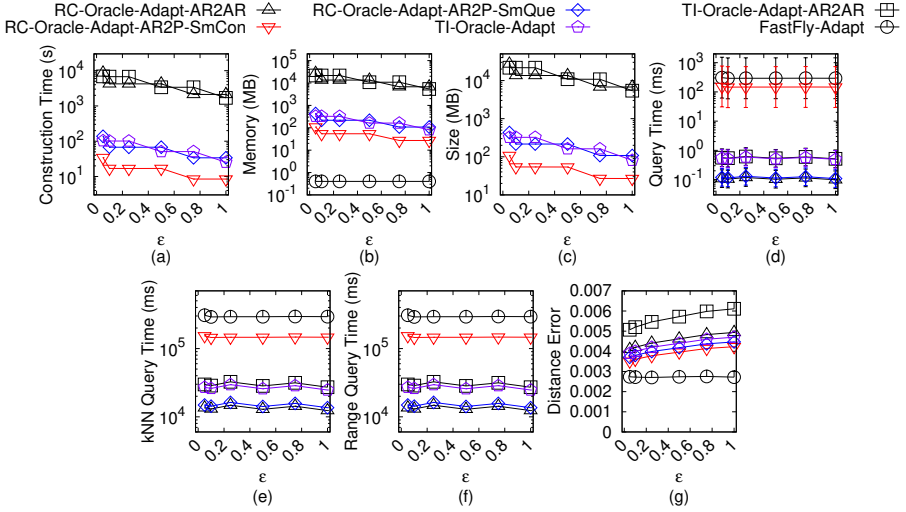
Fig. 62. Baseline comparisons (effect of ϵ on BH_t TIN dataset for AR2P query)

with 2.5M points and 500 POIs, the *kNN* query time of *RC-Oracle* is 12.5s. But, the time is 1520s ≈ 25 min for *SU-Oracle-Adapt*, and 25s for *RC-Oracle-NaiveProx*. Since the shortest path query time of *RC-Oracle* is $O(1)$, and we do not need to perform linear scans over all the POIs in our efficient proximity query algorithm of *RC-Oracle*.

B.2.4 ***Baseline comparisons for the A2P query***. We study the A2P query on *point clouds* for baseline comparisons. We compared *SE-Oracle-Adapt-A2A*, *UP-Oracle-Adapt-A2A*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive-A2A*, *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle*, *TI-Oracle-A2A* and *FastFly* on small-version datasets with default 50 POIs. We compared
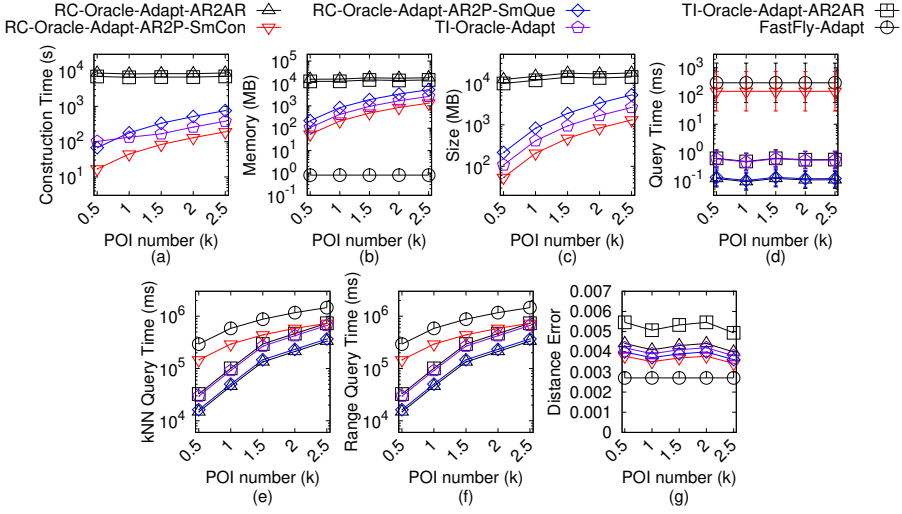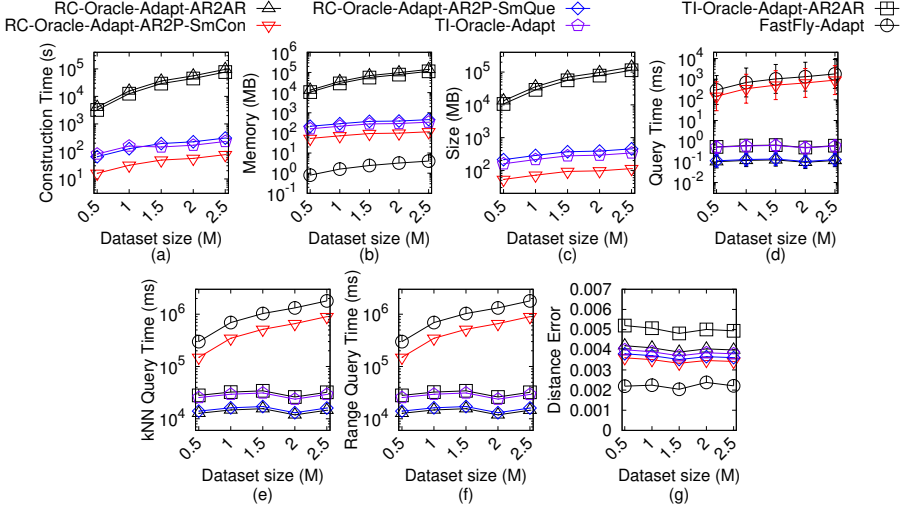
Fig. 63. Baseline comparisons (effect of n on BH_t TIN dataset for AR2P query)
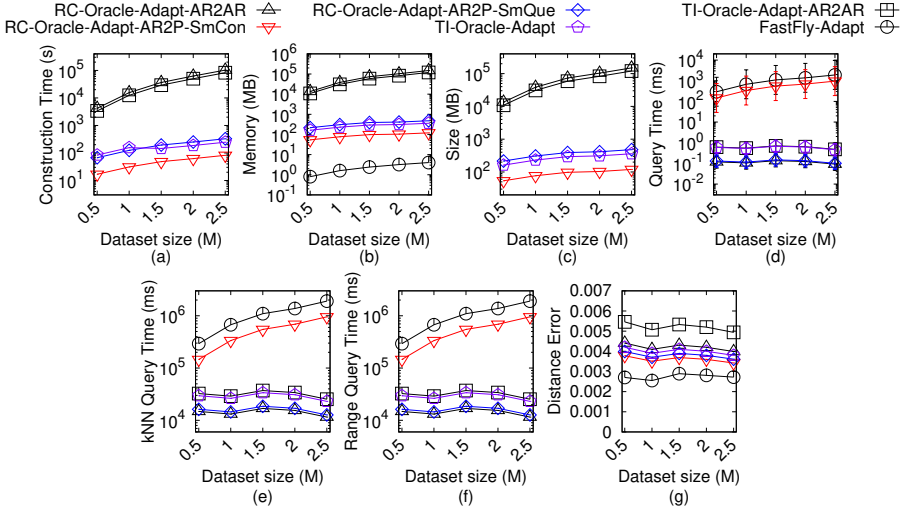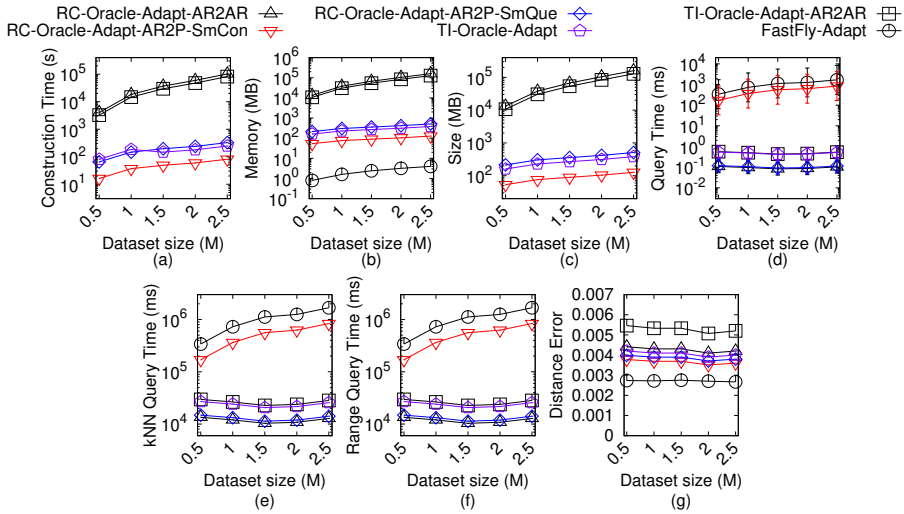

Fig. 64. Baseline comparisons (effect of N on BH_t TIN dataset for AR2P query)

RC-Oracle-A2A, RC-Oracle-A2P-SmCon, RC-Oracle-A2P-SmQue, TI-Oracle, TI-Oracle-A2A and *FastFly* on large-version datasets with default 500 POIs. (1) Figure 113 and Figure 114 show the result on BH_c-*small* point cloud dataset for the A2P query when varying ϵ and n, respectively. (2) Figure 115, Figure 116 and Figure 117 show the result on EP_c-*small* point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (3) Figure 118 and Figure 119 show the result on GF_c-*small* point cloud dataset for the A2P query when varying ϵ and n, respectively. (4) Figure 120 and Figure 121 show the result on LM_c-*small* point cloud dataset for the A2P query when varying ϵ and n, respectively. (5) Figure 122 and Figure 123 show the result on RM_c-*small* point cloud dataset for the A2P query when varying ϵ and n, respectively. (6) Figure 124, Figure 125 and Figure 126 show the result on BH_c point cloud dataset for the A2P query when varying ϵ, n and N, respectively.
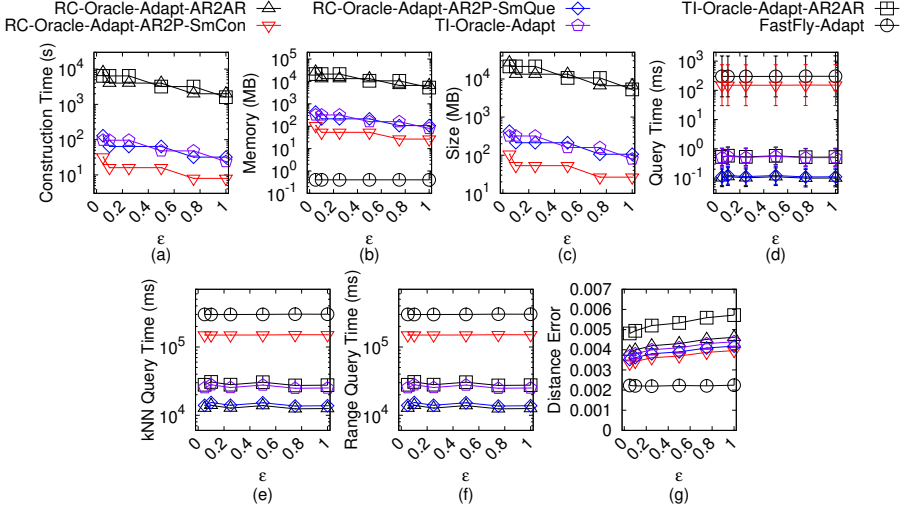
Fig. 65. Baseline comparisons (effect of ϵ on EP_t *TIN* dataset for AR2P query)
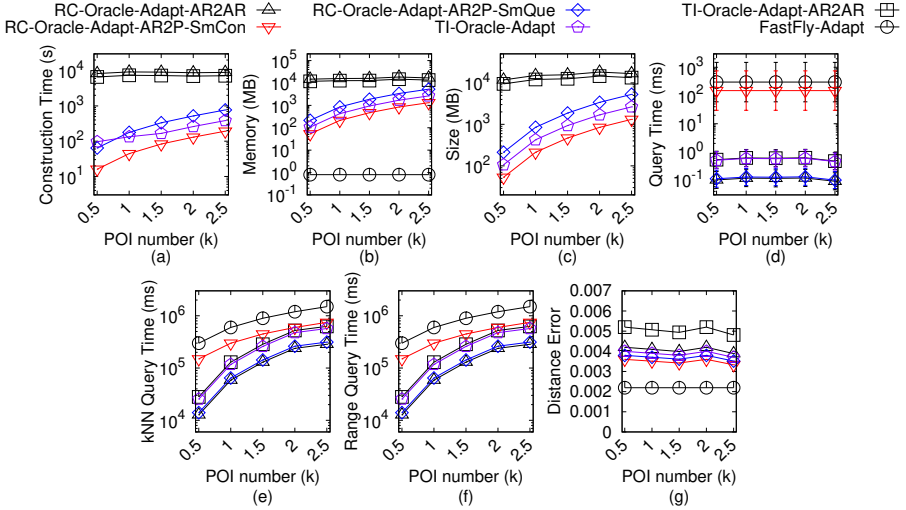
Fig. 66. Baseline comparisons (effect of n on EP_t *TIN* dataset for AR2P query)

(6) Figure 124, Figure 125 and Figure 126 show the result on BH_c point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (7) Figure 127, Figure 128 and Figure 129 show the result on EP_c point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (8) Figure 130, Figure 131 and Figure 132 show the result on GF_c point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (9) Figure 133, Figure 134 and Figure 135 show the result on LM_c point cloud dataset for the A2P query when varying ϵ, n and N, respectively. (10) Figure 136, Figure 137 and Figure 138 show the result on RM_c point cloud dataset for the A2P query when varying ϵ, n and N, respectively.

Effect of ϵ. In Figure 113, Figure 115, Figure 118, Figure 120 and Figure 122, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_c-*small*, EP_c-*small*, GF_c-*small*, LM_c-*small* and RM_c-*small*
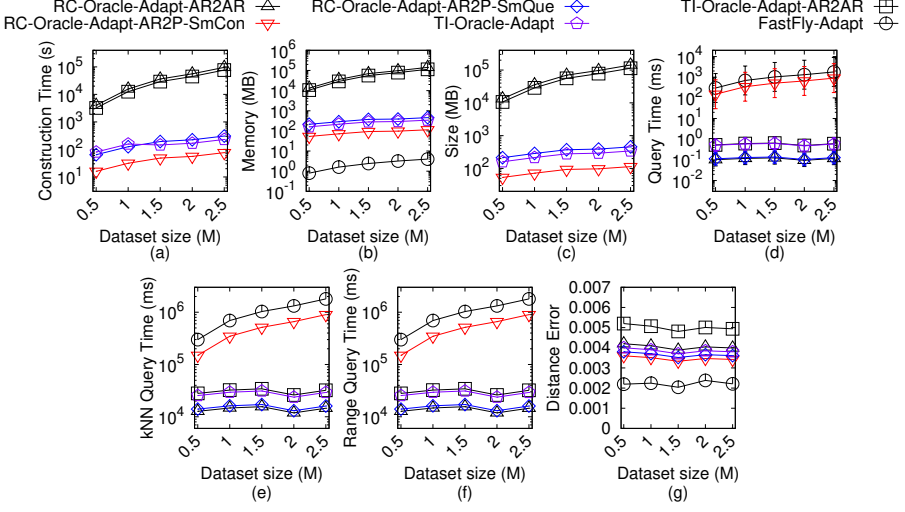
Fig. 67. Baseline comparisons (effect of N on EP_t TIN dataset for AR2P query)

Fig. 68. Baseline comparisons (effect of ϵ on GF_t TIN dataset for AR2P query)

dataset by setting N to be 10k and n to be 50. In Figure 124, Figure 127, Figure 130, Figure 133 and Figure 136, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_c, EP_c, GF_c, LM_c and RM_c dataset by setting N to be 0.5M and n to be 500. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* perform better than the best-known adapted oracle *EAR-Oracle-Adapt* for the A2P query on a point cloud in terms of oracle construction time, oracle size and shortest path query time. Since *EAR-Oracle-Adapt* has the *loose criterion for algorithm earlier termination* drawback.

Effect of n. In Figure 114, Figure 116, Figure 119, Figure 121 and Figure 123, we tested 5 values of n from $\{50, 100, 150, 200, 250\}$ on BH_c-*small*, EP_c-*small*, GF_c-*small*, LM_c-*small* and RM_c-*small* dataset by setting N to be 10k and ϵ to be 0.1. In Figure 125, Figure 128, Figure 131, Figure 134 and Figure 137, we tested 5 values of n from $\{500, 1000, 1500, 2000, 2500\}$ on BH_c, EP_c, GF_c, LM_c and RM_c

Fig. 69. Baseline comparisons (effect of n on GF_t TIN dataset for AR2P query)
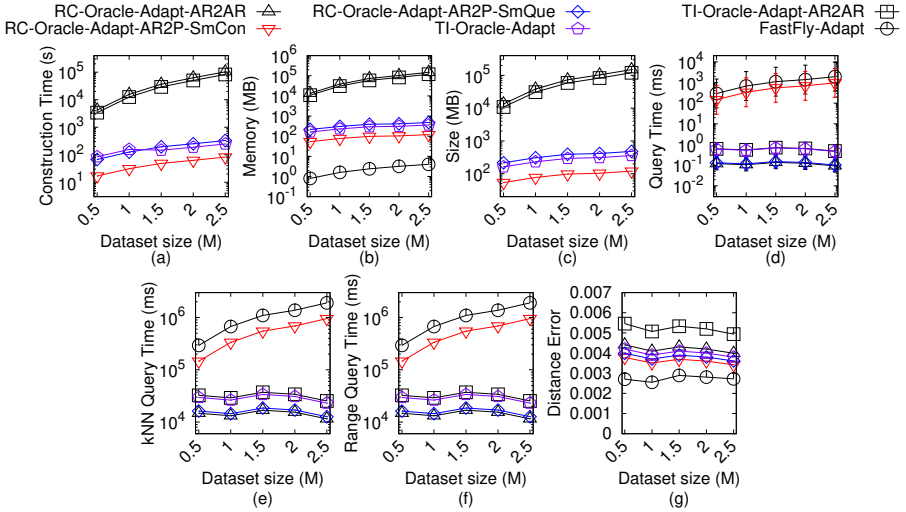
Fig. 70. Baseline comparisons (effect of N on GF_t TIN dataset for AR2P query)

dataset by setting N to be 0.5M and ϵ to be 0.25. (1) *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* also perform better than *RC-Oracle-A2A*. (2) The oracle construction time, memory usage and oracle size of *RC-Oracle-AR2P-SmCon* are the smallest. But, the shortest path query time of *RC-Oracle-AR2P-SmCon* is larger than other oracles (but still smaller than *FastFly*) due to the same reason as that of *RC-Oracle-Adapt-AR2P-SmCon* for *TIN*s. Thus, it performs well in the case of fewer proximity queries. (3) The oracle construction time of *RC-Oracle-AR2P-SmQue* and *TI-Oracle* are also very small and their shortest path query time are also very small due to the same reason as those of *RC-Oracle-Adapt-AR2P-SmQue* and *TI-Oracle-Adapt* for *TIN*s. When $n < 500$ (resp. $n \geq 500$), the oracle construction time of *RC-Oracle-AR2P-SmQue* is smaller (resp. larger) than
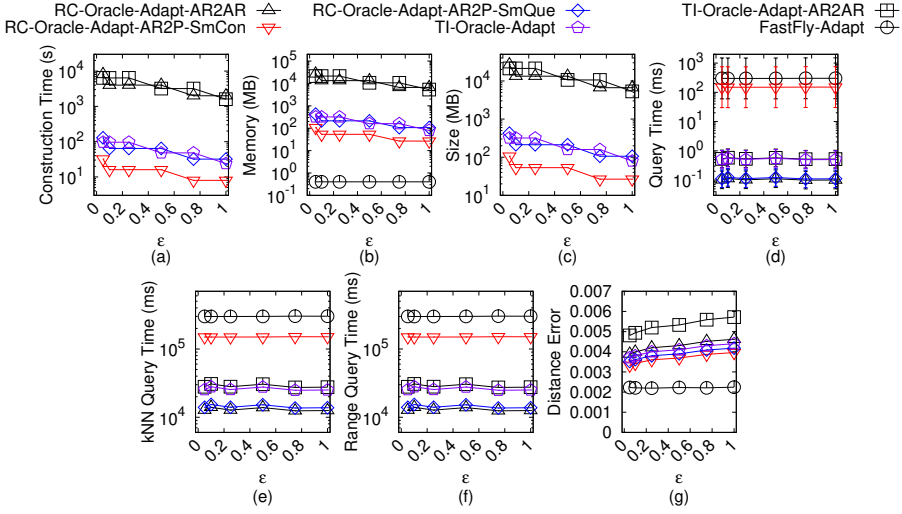
Fig. 71. Baseline comparisons (effect of ϵ on LM_t TIN dataset for AR2P query)

Fig. 72. Baseline comparisons (effect of n on LM_t TIN dataset for AR2P query)

that of *TI-Oracle*, and it verifies our claim that the former (resp. latter) one performs well when the density of POIs is high (resp. low).

Effect of N (scalability test). In Figure 117, we tested 5 values of N from {10k, 20k, 30k, 40k, 50k} on EP_c-*small* dataset by setting n to be 50 and ϵ to be 0.1 for scalability test. In Figure 126, Figure 129, Figure 132, Figure 135 and Figure 138, we tested 5 values of N from {0.5M, 1M, 1.5M, 2M, 2.5M} on BH_c-*small*, EP_c-*small*, GF_c-*small*, LM_c-*small* and RM_c-*small* dataset by setting n to be 500 and ϵ to be 0.25 for scalability test. *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue* and *TI-Oracle* are scalable when N is large.

Fig. 73. Baseline comparisons (effect of N on LM_t TIN dataset for AR2P query)
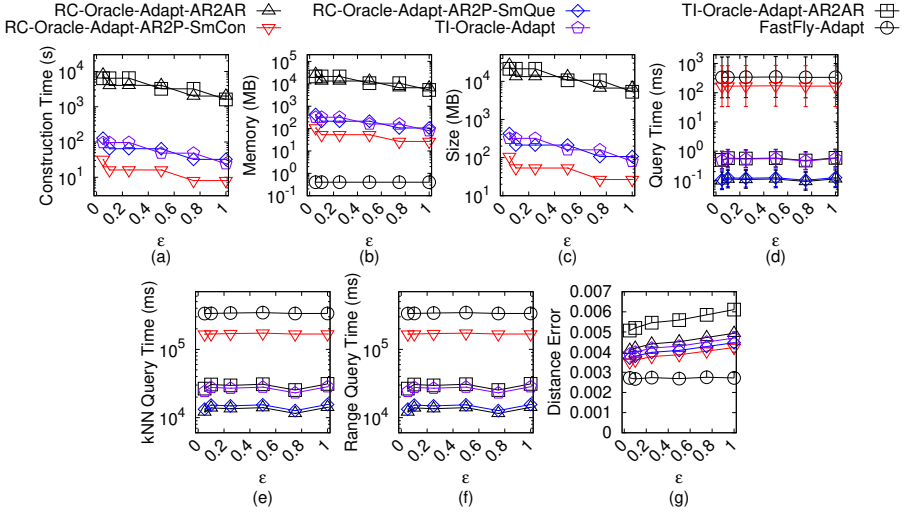
Fig. 74. Baseline comparisons (effect of ϵ on RM_t TIN dataset for AR2P query)

B.2.5 Ablation study for the A2P query. We study the A2P query on *point clouds* for ablation study. We compared *SE-Oracle-FastFly-Adapt-A2A, UP-Oracle-FastFly-Adapt-A2A, EAR-Oracle-FastFly-Adapt, RC-Oracle-A2A, RC-Oracle-A2P-SmCon, RC-Oracle-A2P-SmQue, TI-Oracle* and *TI-Oracle-A2A*.

In Figure 139, Figure 140, Figure 141, Figure 142 and Figure 143, we tested 6 values of ϵ from $\{0.05, 0.1, 0.25, 0.5, 0.75, 1\}$ on BH_c, EP_c, GF_c, LM_c and RM_c dataset by setting N to be 0.5M and n to be 500. The oracle construction time, oracle size and shortest path query time of *RC-Oracle-A2P-SmQue* and *TI-Oracle* also perform better than *SE-Oracle-FastFly-Adapt-A2A, UP-Oracle-FastFly-Adapt-A2A* and *EAR-Oracle-FastFly-Adapt*.
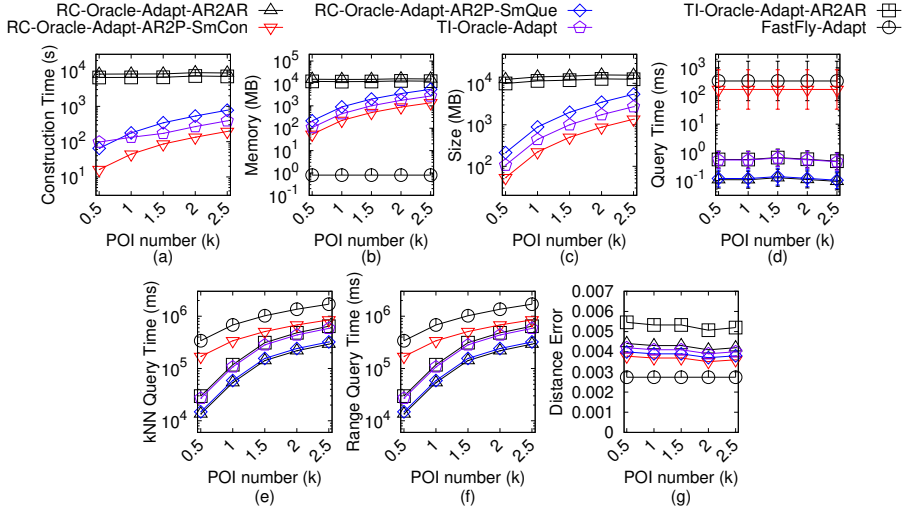
Fig. 75. Baseline comparisons (effect of n on RM_t TIN dataset for AR2P query)

Fig. 76. Baseline comparisons (effect of N on RM_t TIN dataset for AR2P query)

B.2.6 *Comparisons with other proximity queries oracles and variation oracles for the A2P query*. We study the A2P query on *point clouds* for comparisons with other proximity queries oracles and variation oracles. We adapt *SU-Oracle-Adapt* to be *SU-Oracle-Adapt-A2A* in the similar way of *TI-Oracle-A2A* for the A2A query on a point cloud. We denote *RC-Oracle-A2A-NaiveProx*, *RC-Oracle-A2P-SmCon-NaiveProx* and *RC-Oracle-A2P-SmCon-NaiveProx* using the naive proximity query algorithm as mentioned in Section 4.2.4. We denote *TI-Oracle-NaiveProx* and *TI-Oracle-A2A-NaiveProx* to be *TI-Oracle* and *TI-Oracle-A2A* using the naive proximity query algorithm as mentioned in Section 5.2.4. We denote *TI-Oracle-Rtree* and *TI-Oracle-A2A-Rtree* to be *TI-Oracle* and *TI-Oracle-A2A* without using M_{belo} but using an R-tree and a set of 2D boxes as tree nodes to store
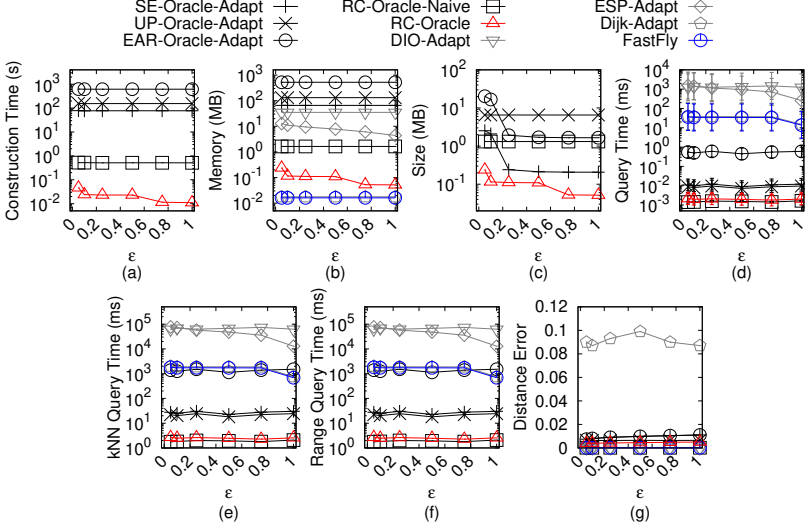
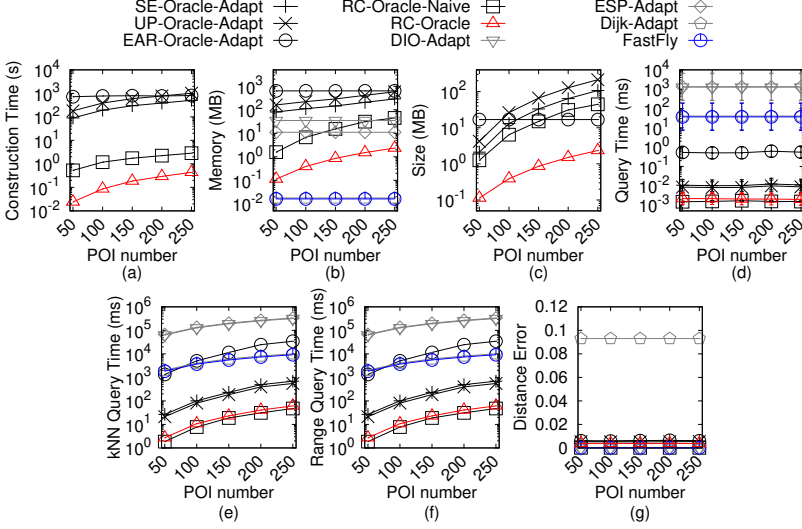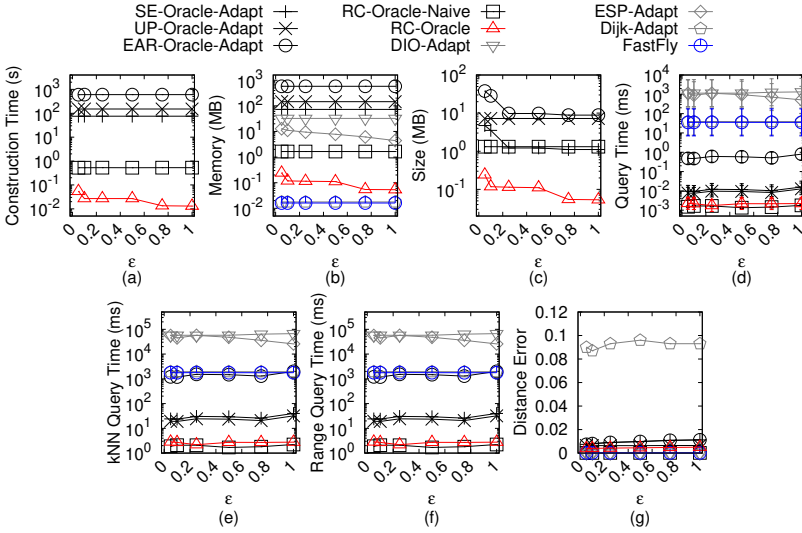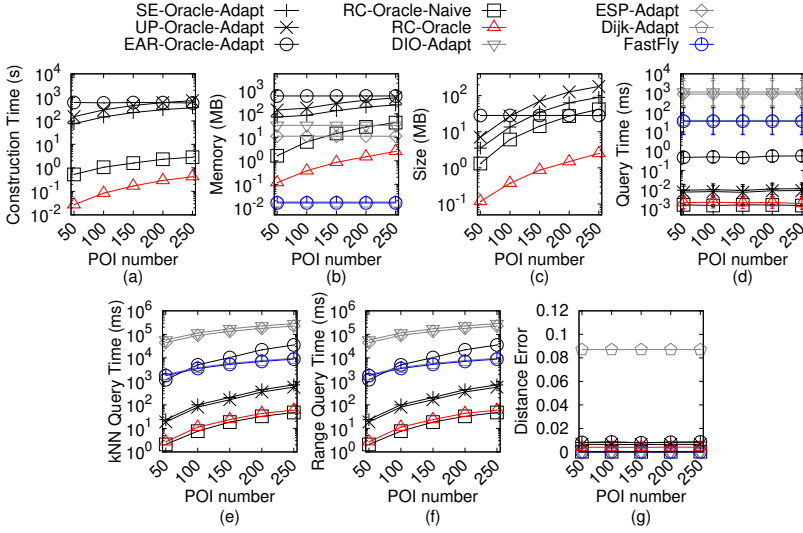Fig. 77. Baseline comparisons (effect of ϵ on BH_c-small point cloud dataset for P2P query)
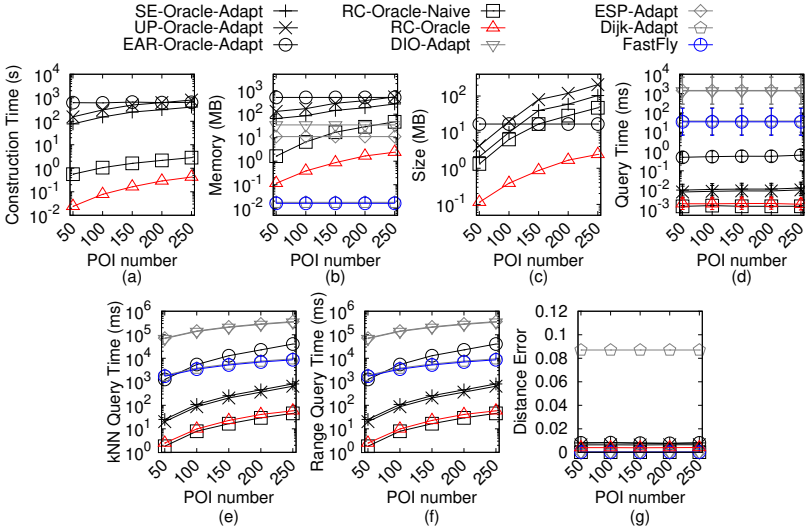
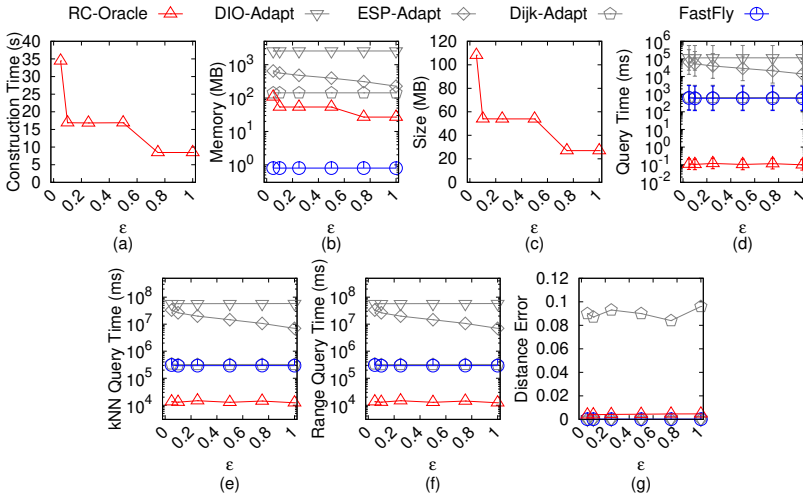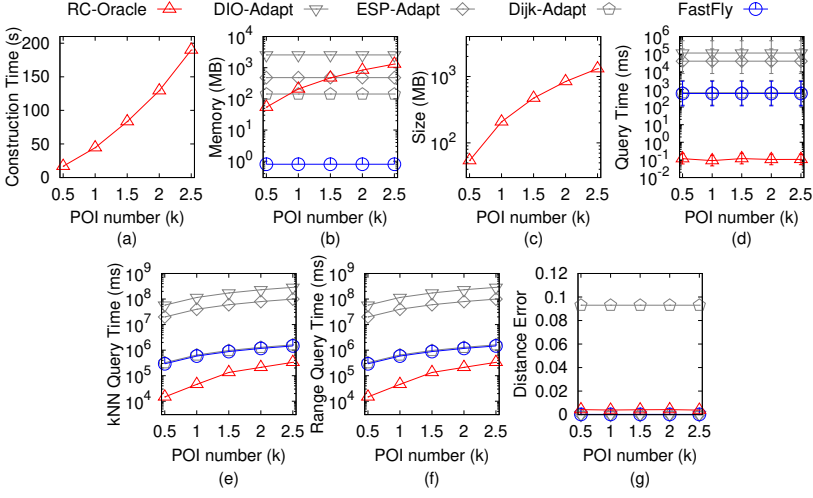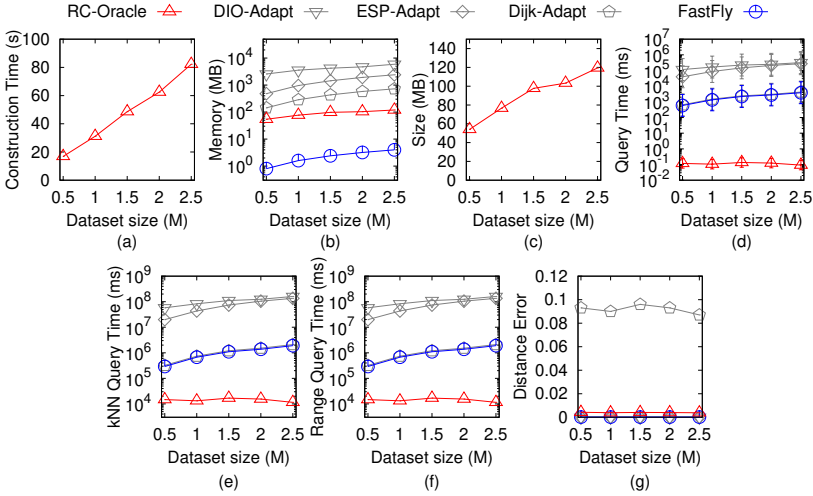Fig. 78. Baseline comparisons (effect of n on BH_c-small point cloud dataset for P2P query)

the belonging point information used in study [40]. We denote *TI-Oracle-TigLoo* and *TI-Oracle-A2A-TigLoo* to be *TI-Oracle* and *TI-Oracle-A2A* without using partition cells but using tight and loose surface oracles used in study [40]. We compared *SU-Oracle-Adapt*, *SU-Oracle-Adapt-A2A*, *RC-Oracle-A2A-NaiveProx*, *RC-Oracle-A2P-SmCon-NaiveProx*, *RC-Oracle-A2P-SmQue-NaiveProx*, *TI-Oracle-NaiveProx*, *TI-Oracle-Rtree*, *TI-Oracle-TigLoo*, *TI-Oracle-A2A-NaiveProx*, *TI-Oracle-A2A-Rtree*, *TI-Oracle-A2A-TigLoo*, *RC-Oracle-A2A*, *RC-Oracle-A2P-SmCon*, *RC-Oracle-A2P-SmQue*, *TI-Oracle* and *TI-Oracle-A2A*.

In Figure 144, Figure 145, Figure 146, Figure 147 and Figure 148, we tested 6 values of ϵ from {0.05, 0.1, 0.25, 0.5, 0.75, 1} on BH_c, EP_c, GF_c, LM_c and RM_c dataset by setting N to be 0.5M and n to be 500.
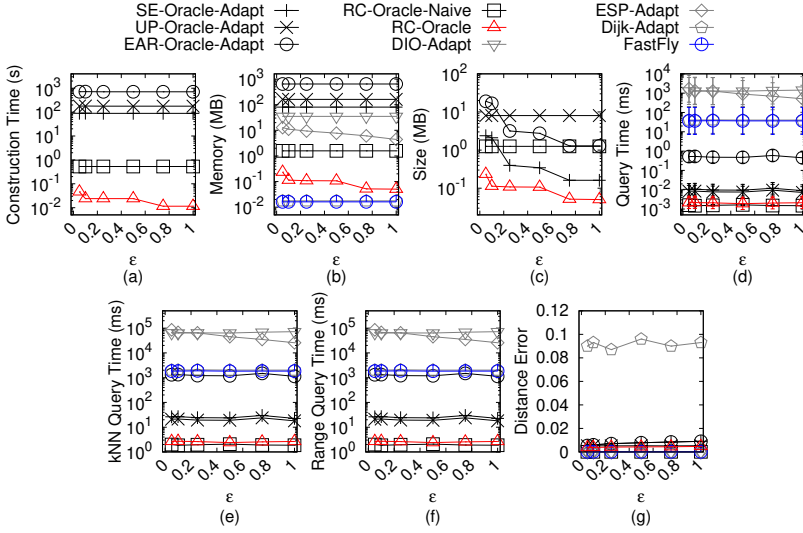
Fig. 79. Baseline comparisons (effect of ϵ on EP_c-small point cloud dataset for P2P query)
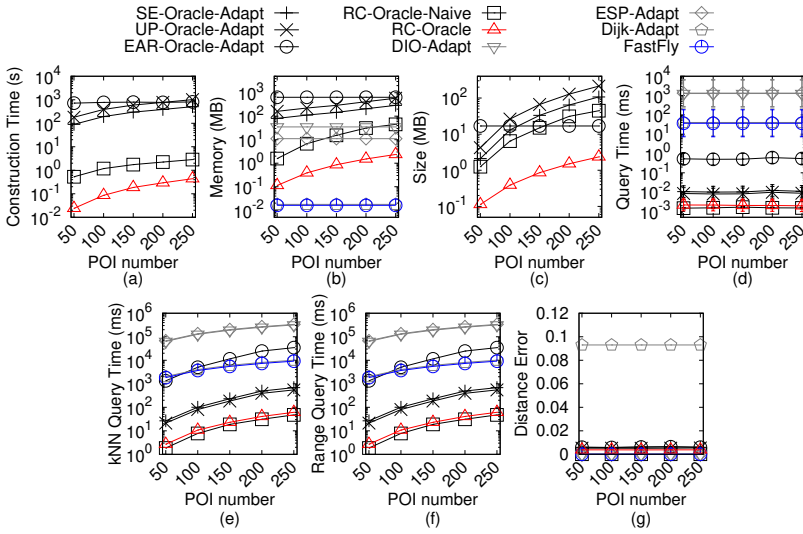
Fig. 80. Baseline comparisons (effect of n on EP_c-small point cloud dataset for P2P query)

(1) Our oracles and variations perform better than *SU-Oracle-Adapt* and *SU-Oracle-Adapt-A2A*. (2) For a point cloud with 2.5M points and 500 POIs, the oracle size and *kNN* query time of *TI-Oracle* are 350MB and 23s, but are 348MB and 104s of *TI-Oracle-Rtree*. The latter one needs to store the R-tree, 2D boxes and points coordinate information. So, it can only slightly reduce the oracle size compared with the former one. But, the latter one uses the R-tree to find which partition cell of a query point belongs to, and the leaf nodes of 2D boxes contain more than one endpoint used for creating the partition cell (see Figure 9 of study [40]). So, it significantly increases the shortest path query time compared with the former one. (3) For a point cloud with 2.5M points and 500 POIs, the oracle construction time and *kNN* query time of *TI-Oracle* are 250s $\approx$ 4.1 min and 23s, but are 500s
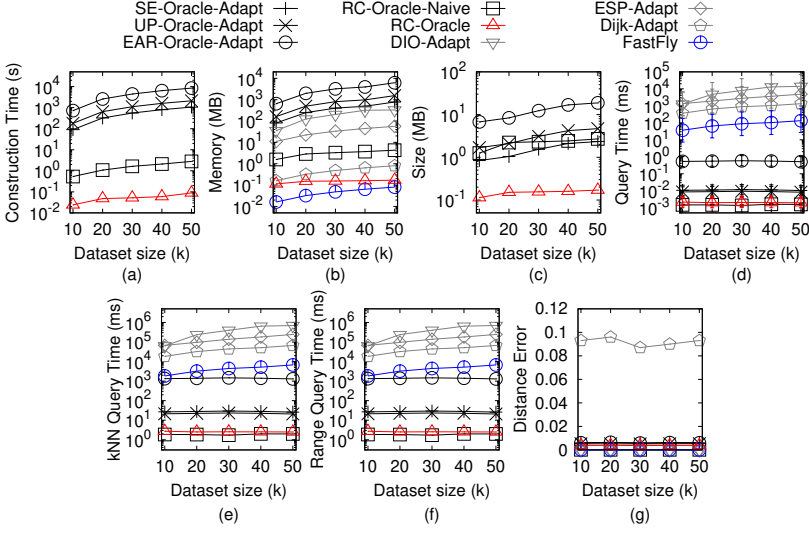
Fig. 81. Baseline comparisons (effect of N on EP_c-small point cloud dataset for P2P query)
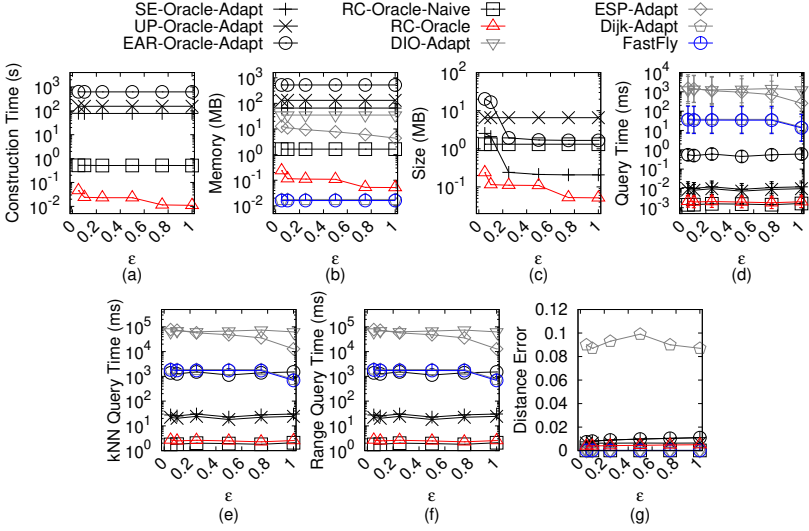
Fig. 82. Baseline comparisons (effect of ϵ on GF_c-small point cloud dataset for P2P query)

$\approx$ 8.2 min and 20s of *TI-Oracle-TigLoo*. The latter one uses tight and loose surface oracles. So, it can gradually expand in *kNN* queries to slightly reduce *kNN* query time compared with the former one. But, the latter one contains more (almost twice compared with *TI-Oracle* using partition cells) boundary points between tight and loose surface oracles and the point cloud. So, it significantly increases the oracle construction time compared with the former one.

B.2.7 ***Baseline comparisons for the A2A query****. We study the A2A query on point clouds for baseline comparisons. In the same figures of baseline comparisons for the A2P query on point
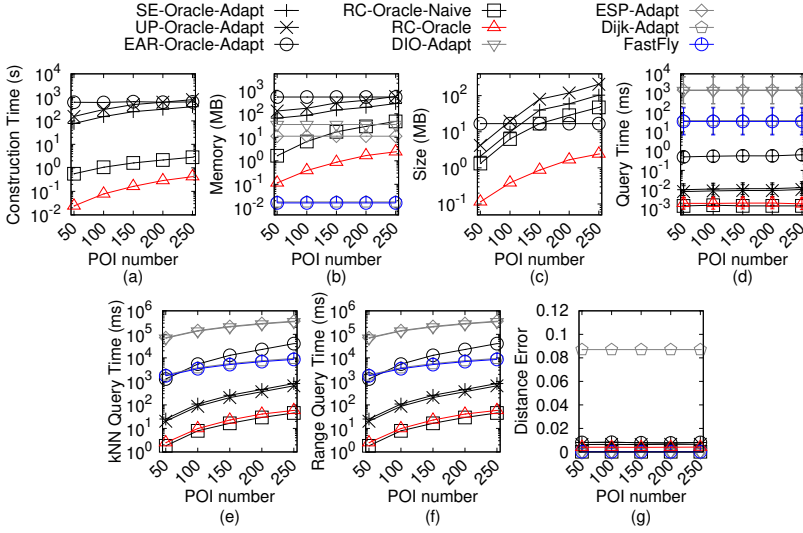
Fig. 83. Baseline comparisons (effect of n on GF_c-$small$ point cloud dataset for P2P query)
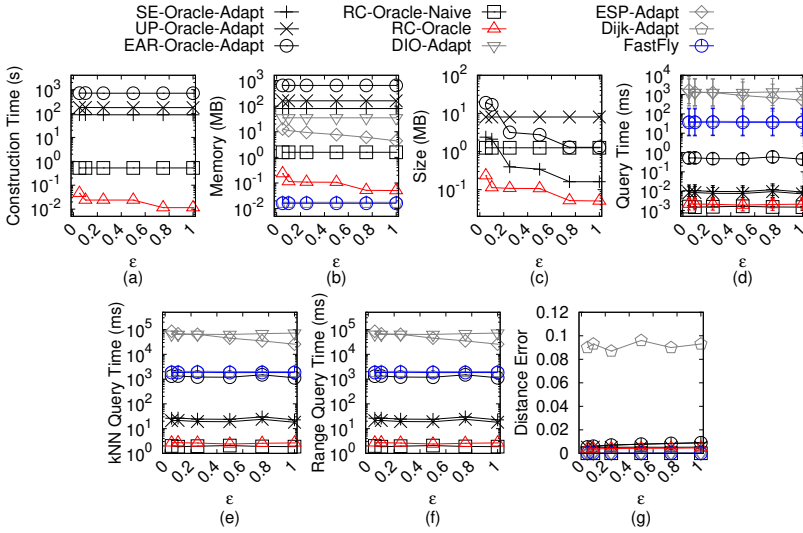
Fig. 84. Baseline comparisons (effect of ϵ on LM_c-$small$ point cloud dataset for P2P query)

clouds, we compared *SE-Oracle-Adapt-A2A*, *UP-Oracle-Adapt-A2A*, *EAR-Oracle-Adapt*, *RC-Oracle-Naive-A2A*, *RC-Oracle-A2A* and *TI-Oracle-A2A*. The last two oracles still perform better than the first two oracles in terms of oracle construction time, oracle size and shortest path query time.

*B.2.8   **Ablation study for the A2A query for ablation study**.* We study the A2A query on point clouds for ablation study. In the same figures of ablation study for the A2P query on point clouds, we compared *SE-Oracle-FastFly-Adapt-A2A*, *UP-Oracle-FastFly-Adapt-A2A*, *EAR-Oracle-FastFly-Adapt*, *RC-Oracle-A2A* and *TI-Oracle-A2A*. The last two oracles still perform better than the first two oracles in terms of oracle construction time, oracle size and shortest path query time.

Fig. 85. Baseline comparisons (effect of n on LM_c-small point cloud dataset for P2P query)

Fig. 86. Baseline comparisons (effect of ϵ on RM_c-small point cloud dataset for P2P query)

B.2.9 *Comparisons with other proximity queries oracles and variation oracles for the A2A query.* We study the A2A query on *point clouds* for comparisons with other proximity queries oracles. In the same figures of comparisons with other proximity queries oracles and variation oracles for the A2P query on point clouds, we compared *SU-Oracle-Adapt-A2A*, *RC-Oracle-A2A* and *TI-Oracle-A2A*. The last two oracles still perform better than the first oracle in terms of oracle construction time, oracle size and shortest path query time. In the comparisons with other proximity queries oracles and variation oracles for the A2P query on point clouds, we have shown that *TI-Oracle* performs better than *TI-Oracle-NaiveProx*, *TI-Oracle-Rtree* and *TI-Oracle-TigLoo*. So, there is

Fig. 87. Baseline comparisons (effect of n on RM_c-*small* point cloud dataset for P2P query)

Fig. 88. Baseline comparisons (effect of ϵ on BH_c point cloud dataset for P2P query)

no need to compare *TI-Oracle-A2A* using the naive proximity query algorithms, using the R-tree and using tight and loose surface oracles.

B.3 Generating Datasets with Different Dataset Sizes

The procedure for generating the point cloud datasets with different dataset sizes is as follows. We mainly follow the procedure for generating datasets with different dataset sizes in the [29, 44, 45]. Let C_t be our target point cloud that we want to generate with qx_t points along x-coordinate, qy_t points along y-coordinate and N_t points, where $N_t = qx_t \cdot qy_t$. Let C_o be the original point cloud that we currently have with qx_o edges along x-coordinate, qy_o edges along y-coordinate and N_o points, where $N_o = qx_o \cdot qy_o$. We then generate $qx_t \cdot qy_t$ 2D points (x, y) based on a Normal distribution

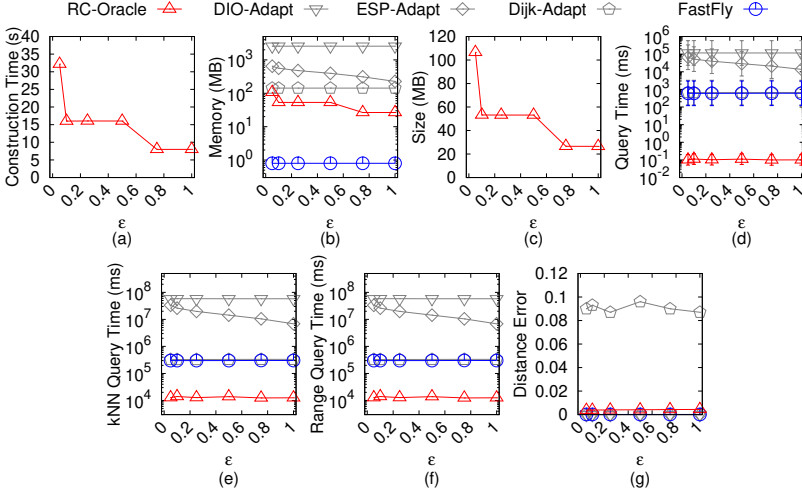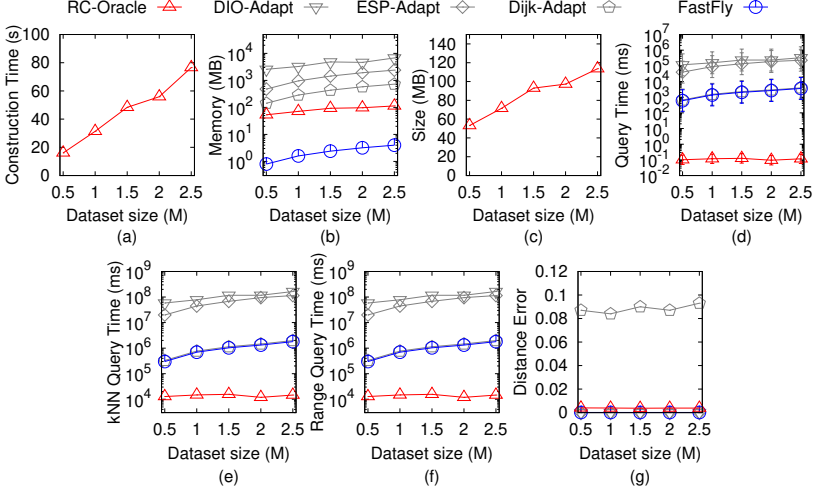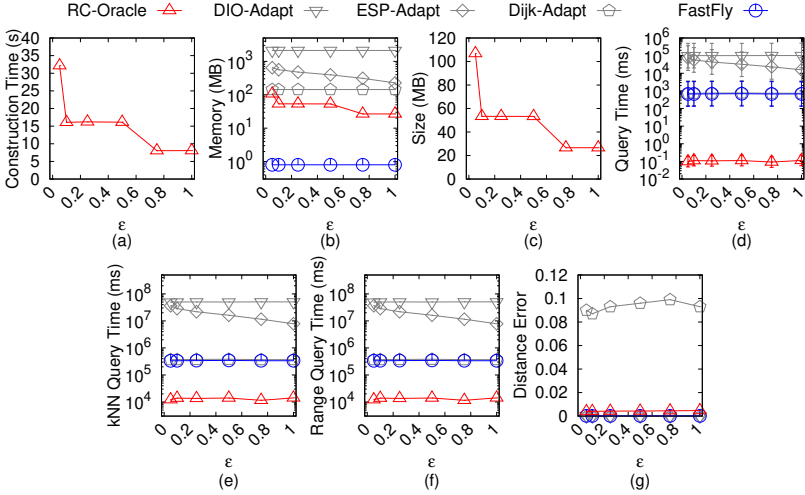Fig. 89. Baseline comparisons (effect of n on BH_c point cloud dataset for P2P query)

Fig. 90. Baseline comparisons (effect of N on BH_c point cloud dataset for P2P query)

$N(\mu_N, \sigma_N^2)$, where $\mu_N = (\overline{x} = \frac{\sum_{q_o \in C_o} x_{q_o}}{q x_o \cdot q y_o}, \overline{y} = \frac{\sum_{q_o \in C_o} y_{q_o}}{q x_o \cdot q y_o})$ and $\sigma_N^2 = (\frac{\sum_{q_o \in C_o} (x_{q_o} - \overline{x})^2}{q x_o \cdot q y_o}, \frac{\sum_{q_o \in C_o} (y_{q_o} - \overline{y})^2}{q x_o \cdot q y_o})$. In the end, we project each generated point (x, y) to the implicit surface of C_o and take the projected point as the newly generated C_t.

C SUMMARY OF ALL NOTATION

Table 5 shows a summary of all notation.

D COMPARISON OF ALL ALGORITHMS

Table 6 shows a comparison of all algorithms (support the shortest path query) in terms of the oracle construction time, oracle size and shortest path query time. Table 7 shows a comparison of
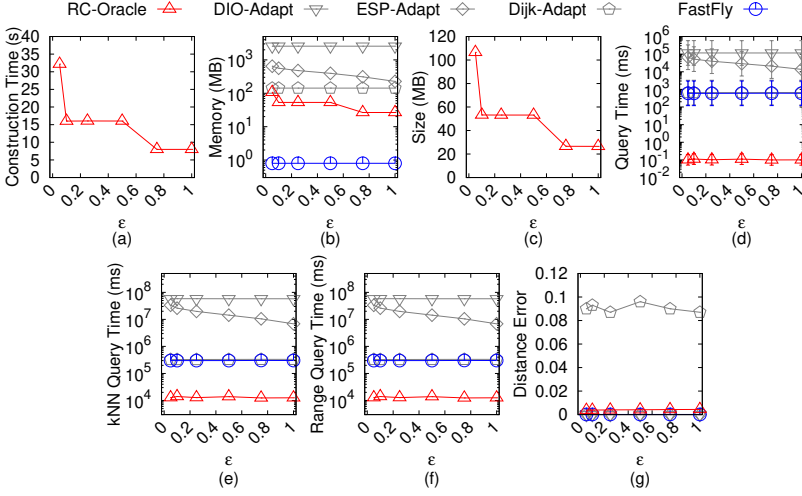
Fig. 91. Baseline comparisons (effect of ϵ on EP_c point cloud dataset for P2P query)
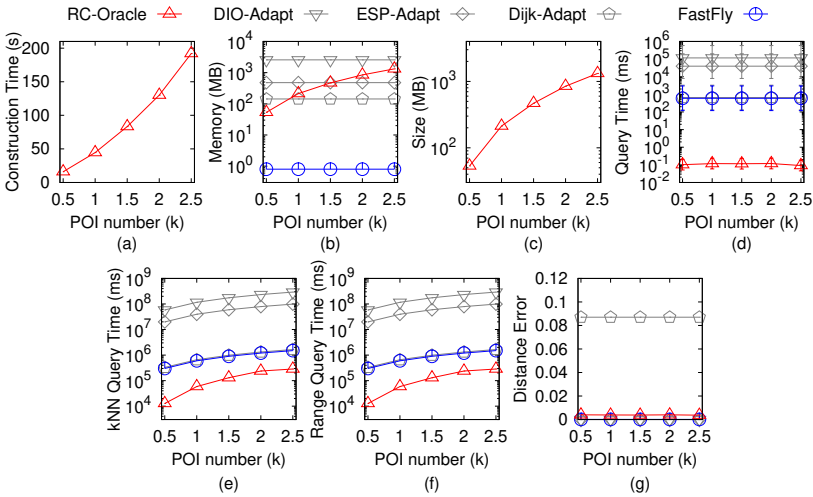
Fig. 92. Baseline comparisons (effect of n on EP_c point cloud dataset for P2P query)

other proximity queries oracles and their variation oracles in terms of the oracle construction time, oracle size and *kNN* query time.

E PROOF

PROOF OF LEMMA 4.4. Firstly, we show the query time of both the *kNN* and range queries algorithm. In the *kNN* or range query, the worst case is that we need to perform a linear scan to check the distance between the query object and all other target objects using the shortest path query phase of *RC-Oracle* in $O(1)$ time. Since there are total n' target objects, the query time is $O(n')$. However, the real query time is smaller than $O(n')$. This is because in most cases, we do not
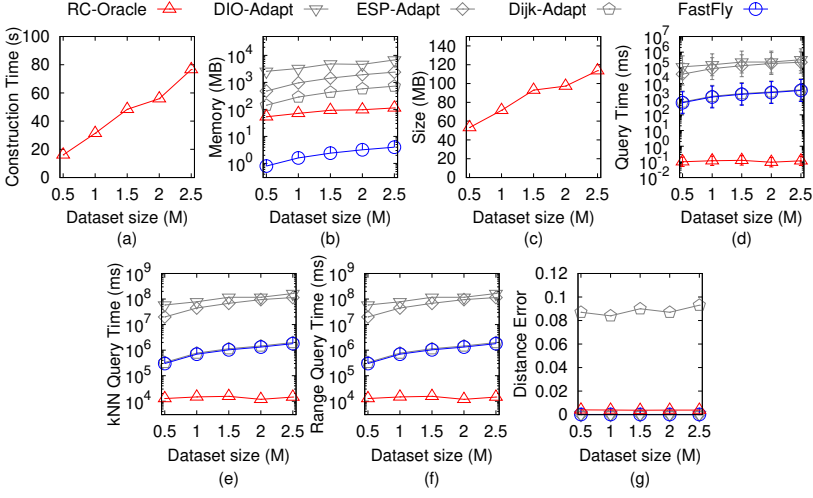
Fig. 93. Baseline comparisons (effect of N on EP_c point cloud dataset for P2P query)
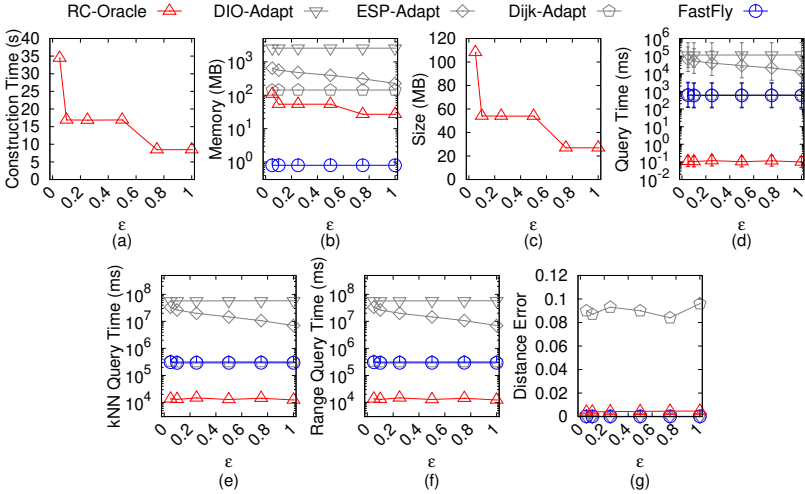
Fig. 94. Baseline comparisons (effect of ϵ on GF_c point cloud dataset for P2P query)

need to perform a linear scan (since we have already sorted some distances in order during the construction phase of *RC-Oracle*).

Secondly, we show the error rate of both the *kNN* and range queries algorithm for *RC-Oracle*. We give some definitions first. For simplicity, given a query POI $q \in P$, we let X be a set of POIs containing the *exact* (1) k nearest POIs of q or (2) POIs whose distance to q are at most r, calculated using the exact distance on C. Furthermore, given a query POI $q \in P$, we let X' be a set of POIs containing (1) k nearest POIs of q or (2) POIs whose distance to q are at most r, calculated using the approximated distance on C returned by *RC-Oracle*. In Figure 1 (a), suppose that the exact k nearest POIs ($k = 2$) of a is c, d, i.e., $X = \{c, d\}$. Suppose that our *kNN* query algorithm finds the k nearest POIs ($k = 2$) of a is b, c, i.e., $X' = \{b, c\}$. Recall that we let v_f (resp.
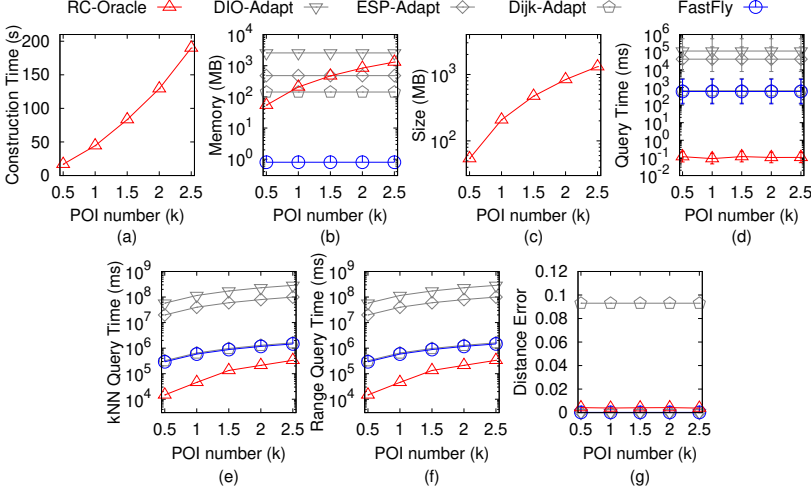
Fig. 95. Baseline comparisons (effect of n on GF_c point cloud dataset for P2P query)
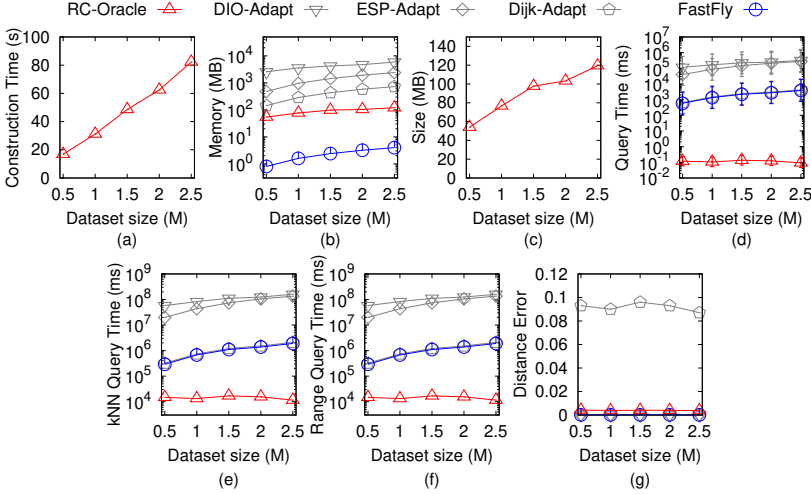
Fig. 96. Baseline comparisons (effect of N on GF_c point cloud dataset for P2P query)

v'_f) be the furthest target object to q in X (resp. X'), i.e., $|\Pi^*(q, v_f|C)| \leq \max_{\forall v \in X} |\Pi^*(q, v|C)|$ (resp. $|\Pi^*(q, v'_f|C)| \leq \max_{\forall v' \in X'} |\Pi^*(q, v'|C)|$). In Figure 1 (a), d is the furthest POI to a in these two POIs, i.e., $v_f = d$ and b is the furthest POI to a in these two POIs, i.e., $v'_f = b$. We further let w_f (resp. w'_f) be the furthest target object to q in X (resp. X') based on the approximated distance on C returned by *RC-Oracle*, i.e., $|\Pi_{RC\text{-}Oracle}(q, w_f|C)| \leq \max_{\forall w \in X} |\Pi_{RC\text{-}Oracle}(q, w|C)|$ (resp. $|\Pi_{RC\text{-}Oracle}(q, w'_f|C)| \leq \max_{\forall w' \in X'} |\Pi_{RC\text{-}Oracle}(q, w'|C)|$). Recall the error rate of the *kNN* and range queries is $\alpha = \frac{|\Pi^*(q, v'_f|C)|}{|\Pi^*(q, v_f|C)|}$. In Figure 1 (a), the error rate is $\frac{|\Pi^*(a, b|C)|}{|\Pi^*(a, d|C)|}$. Since the approximated distance on C returned by *RC-Oracle* is always longer than the exact distance on C, we have $|\Pi_{RC\text{-}Oracle}(q, v'_f|C)| \geq |\Pi^*(q, v'_f|C)|$. Thus, we have $\alpha \leq \frac{|\Pi_{RC\text{-}Oracle}(q, v'_f|C)|}{|\Pi^*(q, v_f|C)|}$. By the definition of v_f and

Fig. 97. Baseline comparisons (effect of ϵ on LM_c point cloud dataset for P2P query)
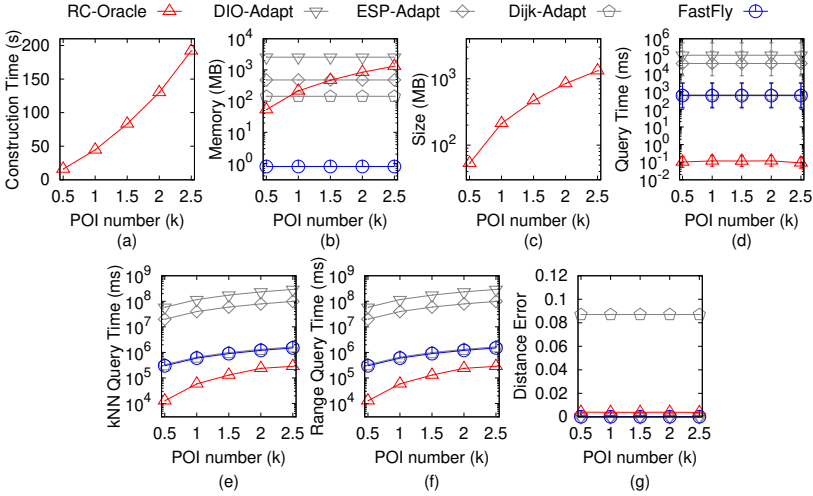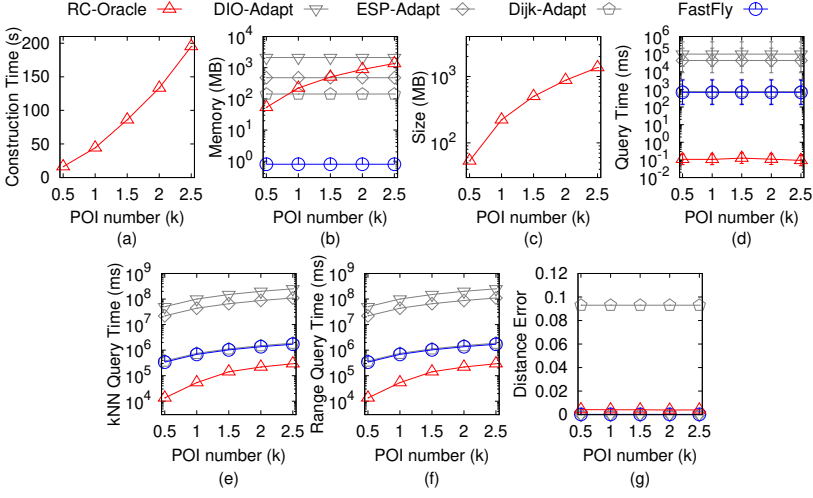

Fig. 98. Baseline comparisons (effect of n on LM_c point cloud dataset for P2P query)

w_f, we have $|\Pi^*(q, v_f|C)| \geq |\Pi^*(q, w_f|C)|$. Thus, we have $\alpha \leq \frac{|\Pi_{RC\text{-}Oracle}(q, v'_f|C)|}{|\Pi^*(q, w_f|C)|}$. By the definition of

v'_f and w'_f, we have $|\Pi_{RC\text{-}Oracle}(q, v'_f|C)| \leq |\Pi_{RC\text{-}Oracle}(q, w'_f|C)|$. Thus, we have $\alpha \leq \frac{|\Pi_{RC\text{-}Oracle}(q, w'_f|C)|}{|\Pi^*(q, w_f|C)|}$.

Since the error rate of the approximated distance on C returned by $RC\text{-}Oracle$ is $(1 + \epsilon)$, we have

$|\Pi_{RC\text{-}Oracle}(q, w_f|C)| \leq (1 + \epsilon)|\Pi^*(q, w_f|C)|$. Then, we have $\alpha \leq \frac{|\Pi_{RC\text{-}Oracle}(q, w'_f|C)|(1+\epsilon)}{|\Pi_{RC\text{-}Oracle}(q, w_f|C)|}$. By our kNN

and range queries algorithm, we have $|\Pi_{RC\text{-}Oracle}(q, w'_f|C)| \leq |\Pi_{RC\text{-}Oracle}(q, w_f|C)|$. Thus, we have

$\alpha \leq 1 + \epsilon$.   □

    PROOF OF LEMMA 4.6. For the query time of both the kNN and range queries algorithm, it is

similar to $RC\text{-}Oracle$. But, $RC\text{-}Oracle\text{-}A2P\text{-}SmCon$ also has the new shortest paths calculation step

Fig. 99. Baseline comparisons (effect of N on LM_c point cloud dataset for P2P query)

Fig. 100. Baseline comparisons (effect of ϵ on RM_c point cloud dataset for P2P query)

before the shortest path query step. *RC-Oracle-A2P-SmCon* just need to use the new shortest paths calculation step once in $O(N \log N)$ time. After that, it needs the shortest path query step in $O(n')$ time. Thus, the total query time of both the *kNN* and range queries algorithm are $O(N \log N + n')$ time. For the error rate, since *RC-Oracle-A2P-SmCon* is also a $(1+\epsilon)$-approximate shortest path oracle, its error rate of both the *kNN* and range queries algorithm is the same as that of *RC-Oracle*. □

PROOF OF LEMMA 4.8. The query time and error rate of both the *kNN* and range queries algorithm for *RC-Oracle-A2P-SmQue* are the same as that of *RC-Oracle*. Since its shortest path query time is the same as that of *RC-Oracle*, and it is also a $(1 + \epsilon)$-approximate shortest path oracle. □

Fig. 101. Baseline comparisons (effect of n on RM_c point cloud dataset for P2P query)
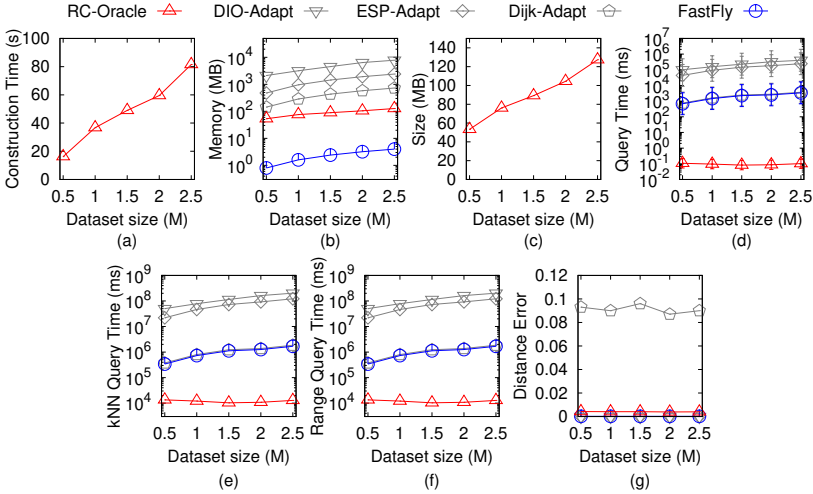
Fig. 102. Baseline comparisons (effect of N on RM_c point cloud dataset for P2P query)

PROOF OF LEMMA 4.10. The query time and error rate of both the *kNN* and range queries algorithm for *RC-Oracle-A2A* are the same as that of *RC-Oracle*. Since its shortest path query time is the same as that of *RC-Oracle*, and it is also a $(1 + \epsilon)$-approximate shortest path oracle.   □

PROOF OF LEMMA 5.2. The query time and error rate of both the *kNN* and range queries algorithm for *TI-Oracle* are the same as that of *RC-Oracle*. Since its shortest path query time is the same as that of *RC-Oracle*, and it is also a $(1 + \epsilon)$-approximate shortest path oracle.   □

PROOF OF LEMMA 5.4. The query time and error rate of both the *kNN* and range queries algorithm for *TI-Oracle-A2A* are the same as that of *RC-Oracle*. Since its shortest path query time is the same as that of *RC-Oracle*, and it is also a $(1 + \epsilon)$-approximate shortest path oracle.   □
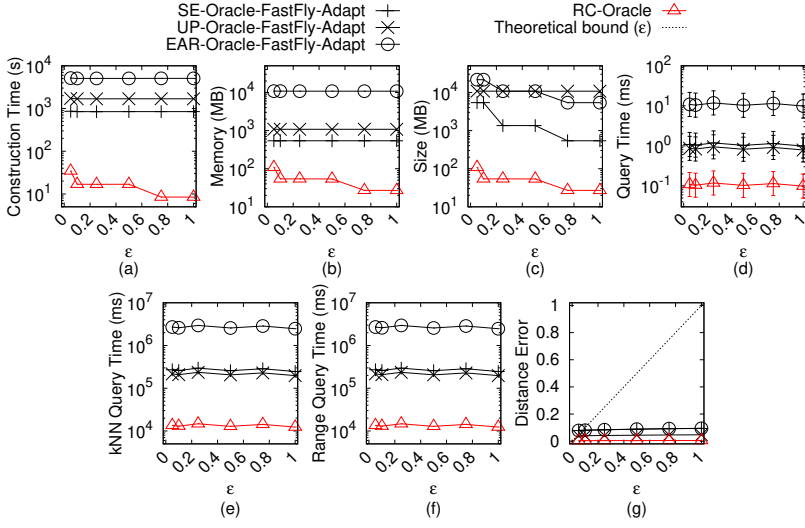
Fig. 103. Ablation study on BH_c point cloud dataset for P2P query
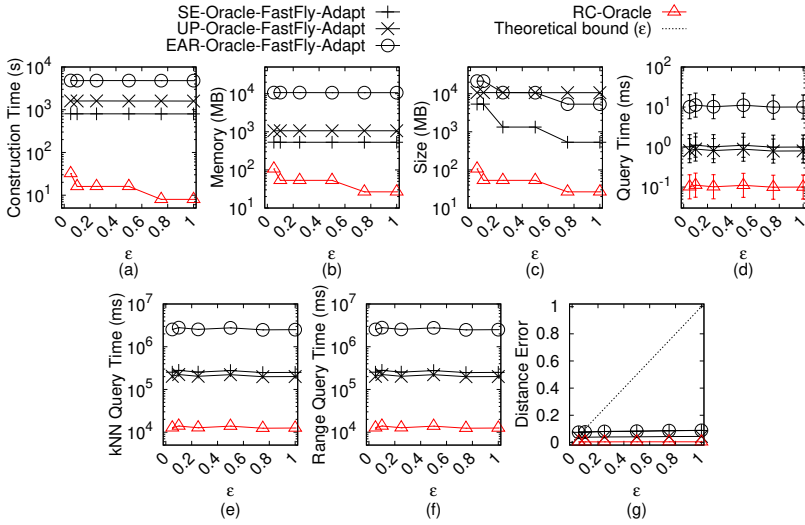
Fig. 104. Ablation study on EP_c point cloud dataset for P2P query

THEOREM E.1. *The shortest path query time and memory consumption of algorithm DIO-Adapt are* $O(N^2)$ *and* $O(N^2)$, *respectively. Consider a point cloud* C *and a TIN* T *converted from* C. *Compared with* $\Pi^*(s, t|T)$, *algorithm DIO-Adapt returns the exact shortest surface path passing on* T. *Compared with* $\Pi^*(s, t|C)$, *algorithm DIO-Adapt returns the approximate shortest path passing on* C.

PROOF. Firstly, we show the *shortest path query time* of algorithm *DIO-Adapt*. The proof of the shortest path query time of algorithm *DIO-Adapt* is in [15, 46]. But since algorithm *DIO-Adapt* first needs to convert C to T, it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *DIO-Adapt* is $O(N + N^2) = O(N^2)$.
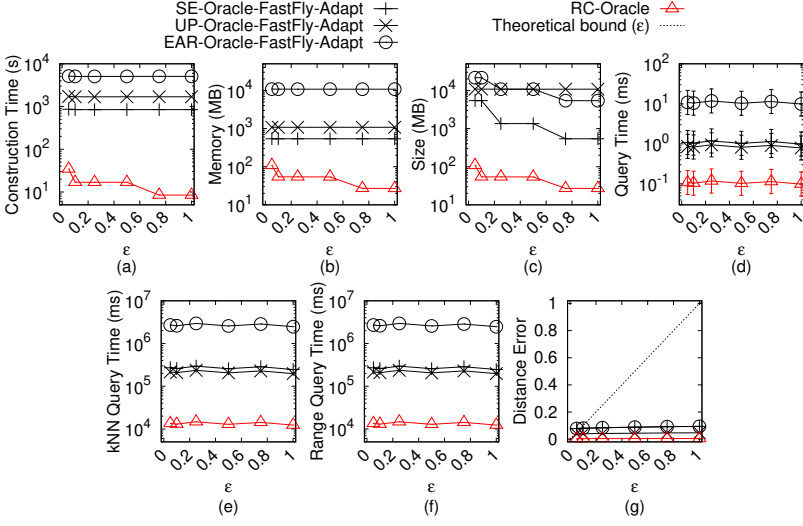
Fig. 105. Ablation study on GF_c point cloud dataset for P2P query
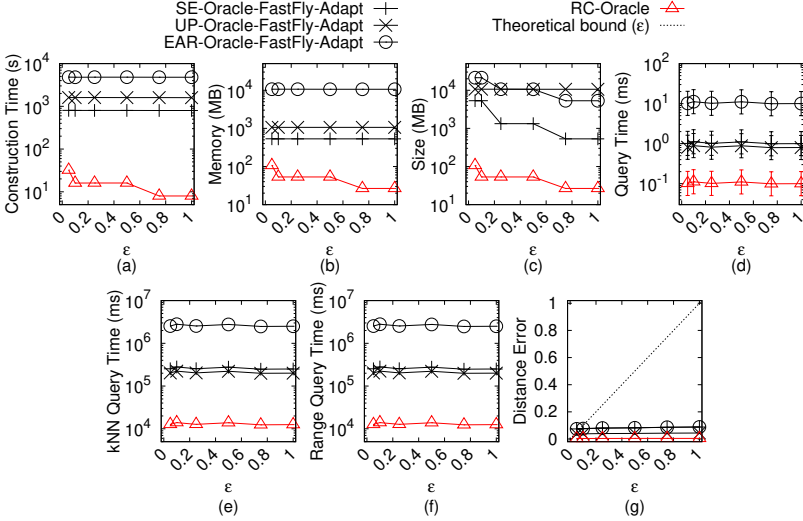
Fig. 106. Ablation study on LM_c point cloud dataset for P2P query

Secondly, we show the *memory consumption* of algorithm *DIO-Adapt*. The proof of the memory consumption of algorithm *DIO-Adapt* is in [15, 46]. Thus, the memory consumption of algorithm *DIO-Adapt* is $O(N^2)$.

Thirdly, we show the *error bound* of algorithm *DIO-Adapt*. Compared with $\Pi^*(s, t|T)$, the proof that algorithm *DIO-Adapt* returns the exact shortest path passing on T is in [15, 46]. Since T is converted from C, so algorithm *DIO-Adapt* returns the exact shortest surface path passing on the *TIN*. Compared with $\Pi^*(s, t|C)$, since we regard $\Pi^*(s, t|C)$ as the exact shortest path passing on C, algorithm *DIO-Adapt* returns the approximate shortest path passing on C. □
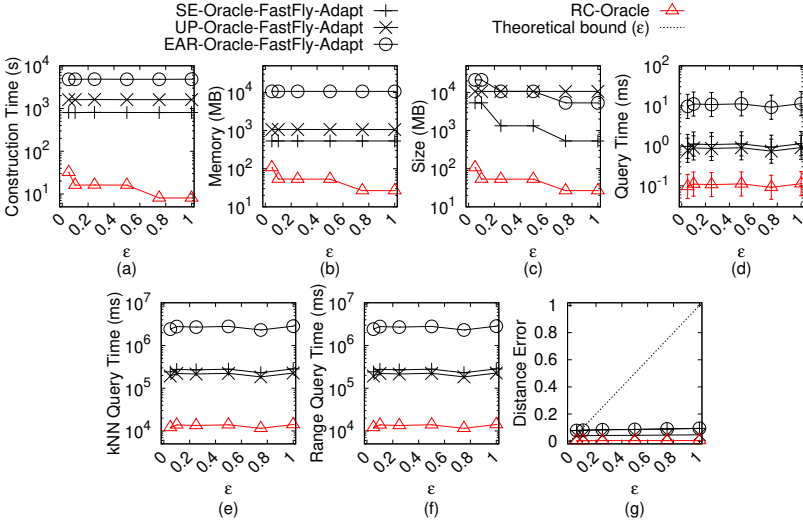
Fig. 107. Ablation study on RM_c point cloud dataset for P2P query
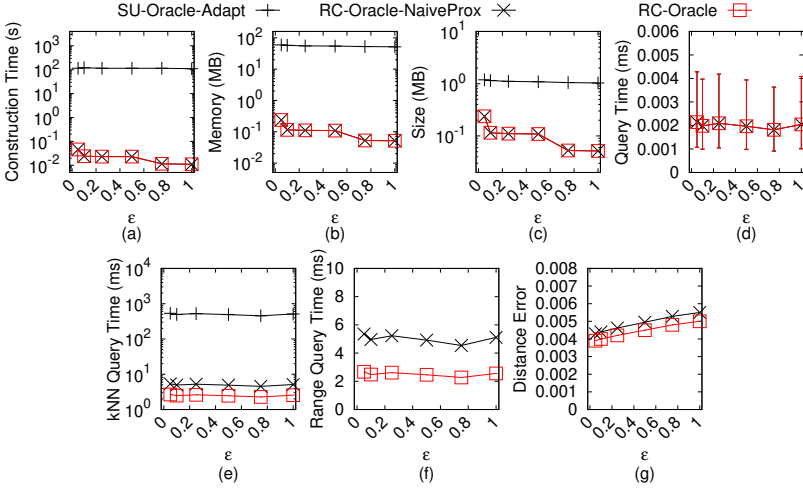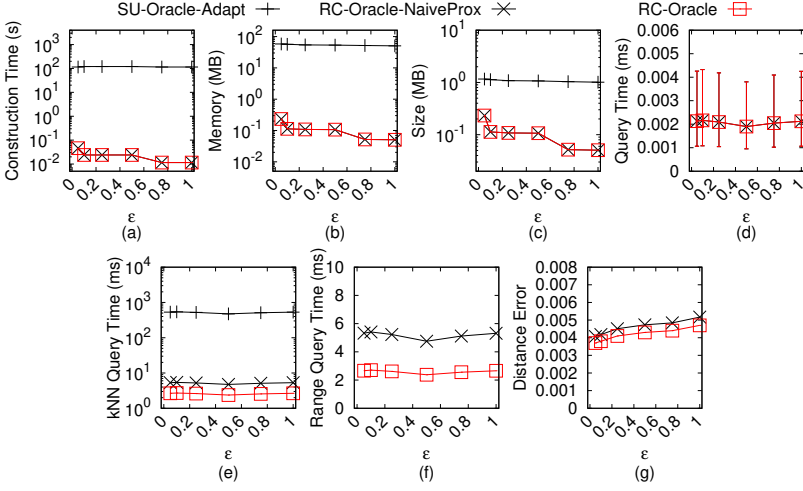

Fig. 108. Comparisons with other proximity queries oracles and variation oracles on BH_c-small point cloud dataset for P2P query

THEOREM E.2. *The shortest path query time and memory consumption of algorithm ESP-Adapt are $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$ and $O(N)$, respectively. Consider a point cloud C and a TIN T converted from C. Compared with $\Pi^*(s,t|T)$, algorithm ESP-Adapt always has $|\Pi_{ESP-Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for any pair of vertices s and t on T, where $\Pi_{ESP-Adapt}(s,t|T)$ is the shortest surface path of algorithm ESP-Adapt passing on T between s and t. Compared with $\Pi^*(s,t|C)$, algorithm ESP-Adapt returns the approximate shortest path passing on C.*

PROOF. Firstly, we show the *shortest path query time* of algorithm *ESP-Adapt*. The proof of the shortest path query time of algorithm *ESP-Adapt* is in [26]. In Section 4.2 of [26], the shortest path query time of algorithm *ESP-Adapt* is $O((N+N')(\log(N+N')+(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}})^2))$, where

Fig. 109. Comparisons with other proximity queries oracles and variation oracles on EP_c-small point cloud dataset for P2P query
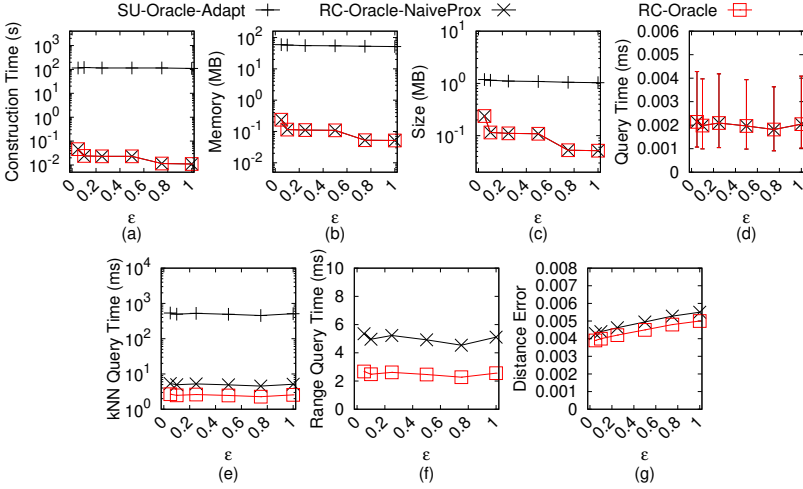
Fig. 110. Comparisons with other proximity queries oracles and variation oracles on GF_c-small point cloud dataset for P2P query

$N' = O(\frac{l_{max}K}{l_{min}\sqrt{1-\cos\theta}}N)$ and K is a positive parameter at least 1. By Theorem 1 of [26], we obtain that its error bound ϵ is equal to $\frac{1}{K-1}$. Thus, we can derive that the shortest path query time of algorithm *ESP-Adapt* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}) + \frac{l_{max}^2}{(\epsilon l_{min}\sqrt{1-\cos\theta})^2})$. Since for N, the first term is larger than the second term, so we obtain the shortest path query time of algorithm *ESP-Adapt* is $O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$. But since algorithm *ESP-Adapt* first needs to convert C to T, so it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *ESP-Adapt* is $O(N + \frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}})) = O(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}\log(\frac{l_{max}N}{\epsilon l_{min}\sqrt{1-\cos\theta}}))$. In [51], it omits the constant term in the shortest path query time. After adding back these terms, the shortest path query time is the same.
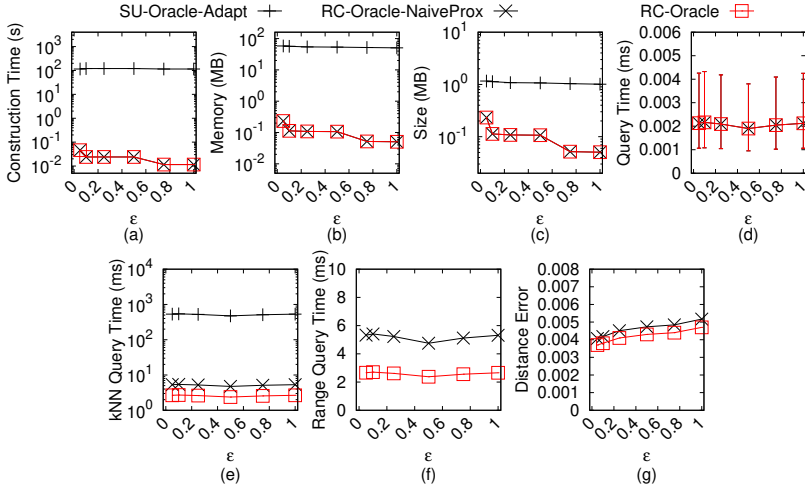
Fig. 111. Comparisons with other proximity queries oracles and variation oracles on LM_c-small point cloud dataset for P2P query
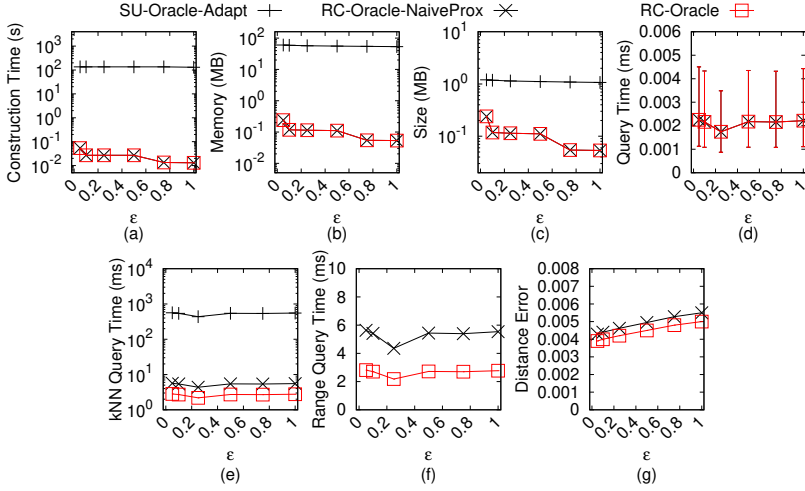

Fig. 112. Comparisons with other proximity queries oracles and variation oracles on RM_c-small point cloud dataset for P2P query

Secondly, we show the *memory consumption* of algorithm *ESP-Adapt*. Since algorithm *ESP-Adapt* is a Dijkstra algorithm and there are total N vertices on T, the memory consumption is $O(N)$. Thus, the memory consumption of algorithm *ESP-Adapt* is $O(N)$.

Thirdly, we show the *error bound* of algorithm *ESP-Adapt*. Compared with $\Pi^*(s, t|T)$, the proof of the error bound of algorithm *ESP-Adapt* is in [26, 51]. Since T is converted from C, so algorithm *ESP-Adapt* always has $|\Pi_{ESP-Adapt}(s, t|T)| \leq (1 + \epsilon)|\Pi^*(s, t|T)|$ for any pair of vertices s and t on T. Compared with $\Pi^*(s, t|C)$, since we regard $\Pi^*(s, t|C)$ as the exact shortest path passing on C, algorithm *ESP-Adapt* returns the approximate shortest path passing on C. □

THEOREM E.3. *The shortest path query time and memory consumption of algorithm DIJ-Adapt are $O(N \log N)$ and $O(N)$, respectively. Consider a point cloud C and a TIN T converted from C.*
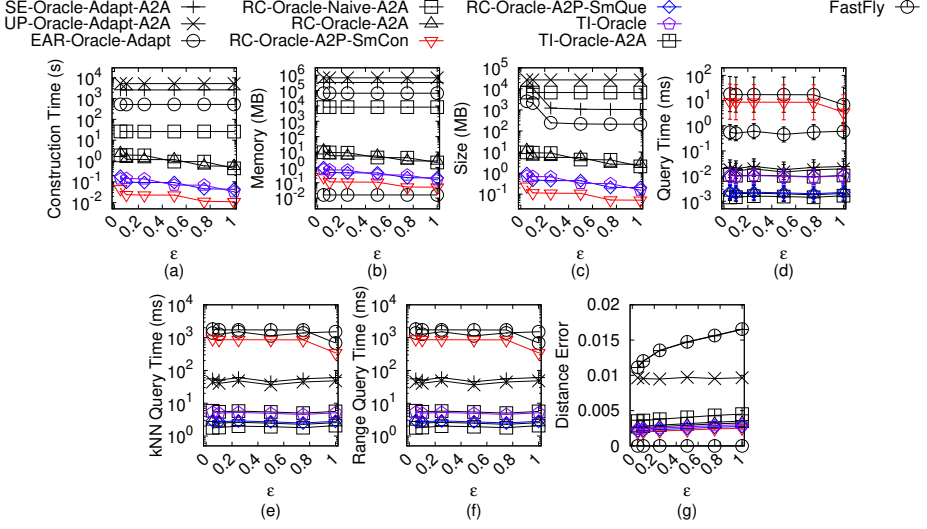
Fig. 113. Baseline comparisons (effect of ϵ on BH_c-small point cloud dataset for A2P query)
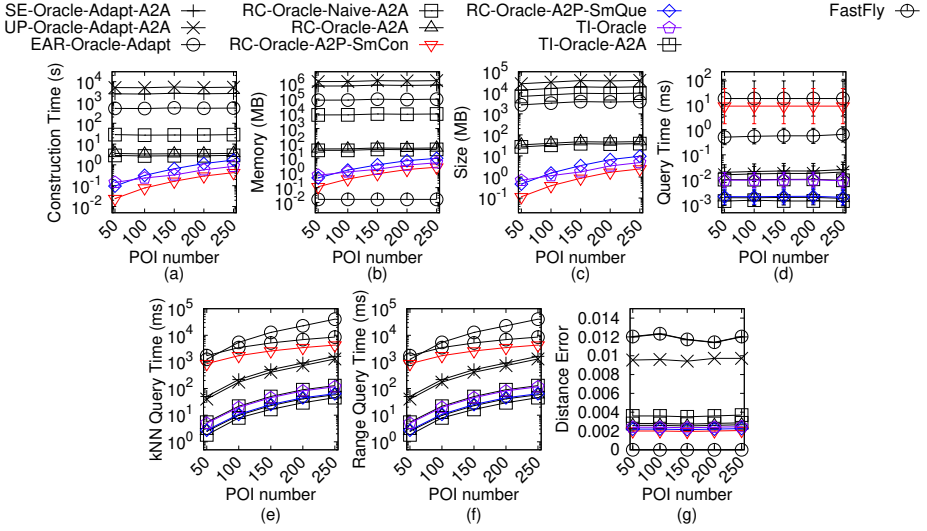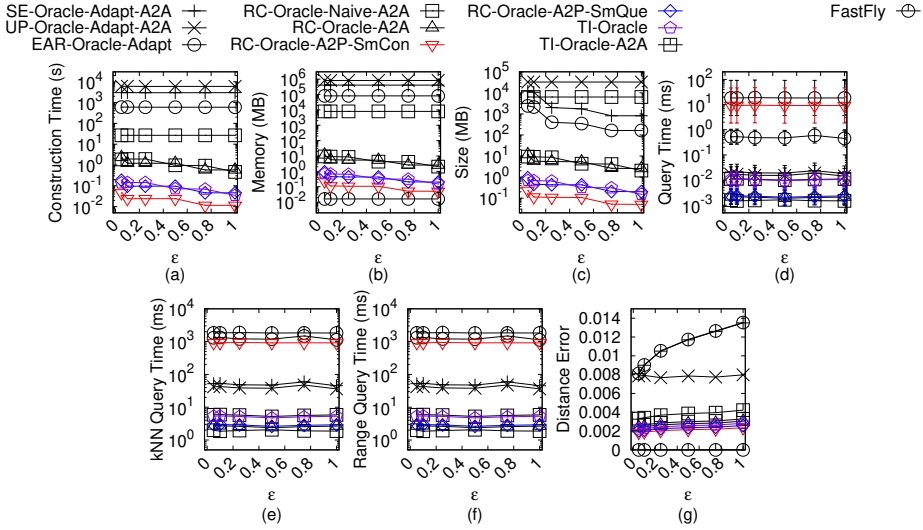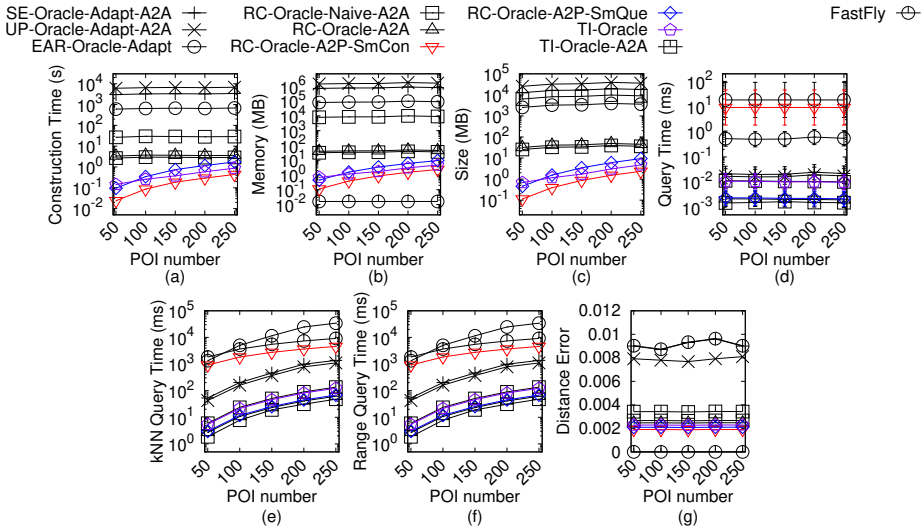

Fig. 114. Baseline comparisons (effect of n on BH_c-small point cloud dataset for A2P query)

Compared with $\Pi^*(s, t|T)$, algorithm DIJ-Adapt always has $|\Pi_{DIJ\text{-}Adapt}(s, t|T)| \le k' \cdot |\Pi^*(s, t|T)|$ for any pair of vertices s and t on T, where $\Pi_{DIJ\text{-}Adapt}(s, t|T)$ is the shortest network path of algorithm DIJ-Adapt passing on T between s and t, $k' = \max\{\frac{2}{\sin\theta}, \frac{1}{\sin\theta\cos\theta}\}$. Compared with $\Pi^*(s, t|C)$, algorithm DIJ-Adapt returns the approximate shortest path passing on C.

PROOF. Firstly, we show the *shortest path query time* of algorithm *DIJ-Adapt*. Since algorithm *DIJ-Adapt* only calculates the shortest network path passing on T, it is a Dijkstra algorithm and there are total N points, so the shortest path query time is $O(N \log N)$. But since algorithm *DIJ-Adapt* first needs to convert C to T, it needs an additional $O(N)$ time for this step. Thus, the shortest path query time of algorithm *DIJ-Adapt* is $O(N + N \log N) = O(N \log N)$.

Fig. 115. Baseline comparisons (effect of ϵ on EP_c-small point cloud dataset for A2P query)

Fig. 116. Baseline comparisons (effect of n on EP_c-small point cloud dataset for A2P query)

Secondly, we show the *memory consumption* of algorithm *DIJ-Adapt*. Since algorithm *DIJ-Adapt* is a Dijkstra algorithm and there are total N vertices on T, the memory consumption is $O(N)$. Thus, the memory consumption of algorithm *DIJ-Adapt* is $O(N)$.

Thirdly, we show the *error bound* of algorithm *DIJ-Adapt*. Recall that $\Pi_W(s, t|T)$ is the shortest network path passing on T between s and t. So, actually $\Pi_W(s, t|T)$ is the same as $\Pi_{DIJ-Adapt}(s, t|T)$. We let $\Pi_E(s, t|T)$ be the shortest path passing on the edges of T (where these edges belong to the faces that $\Pi^*(s, t|T)$ passes) between s and t. Compared with $\Pi^*(s, t|T)$, we know $|\Pi_E(s, t|T)| \leq k' \cdot |\Pi^*(s, t|T)|$ (according to left hand side equation in Lemma 2 of [27]) and $|\Pi_W(s, t|T)| \leq |\Pi_E(s, t|T)|$ (since $\Pi_W(s, t|T)$ considers all the edges on T). So, we have algorithm *DIJ-Adapt* always
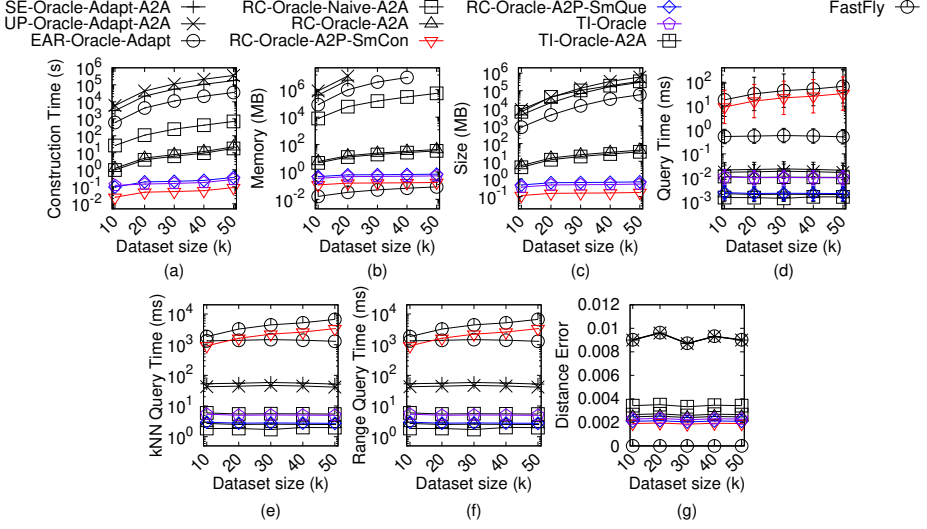
Fig. 117. Baseline comparisons (effect of N on EP_c-small point cloud dataset for A2P query)
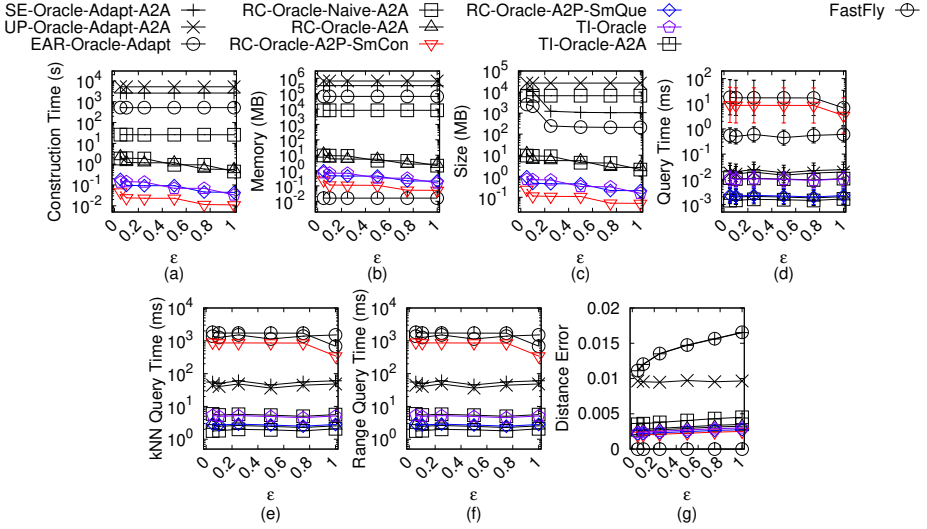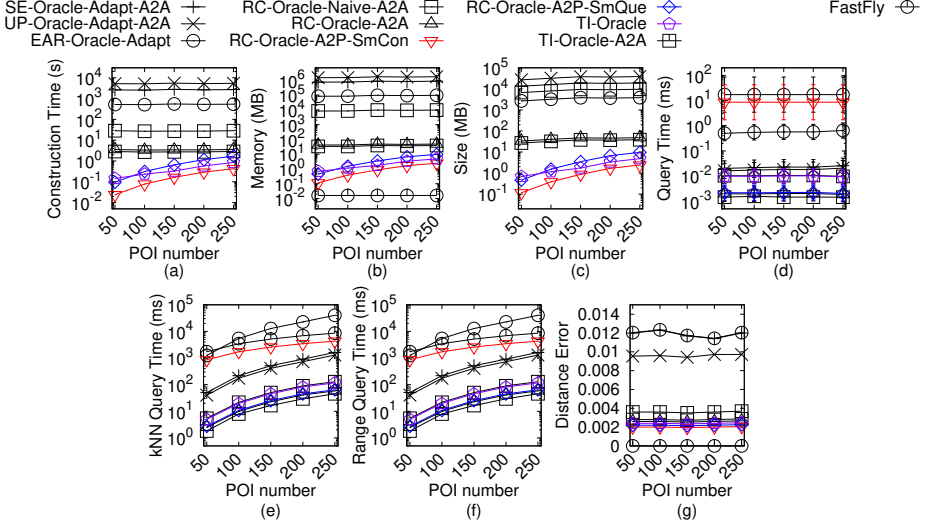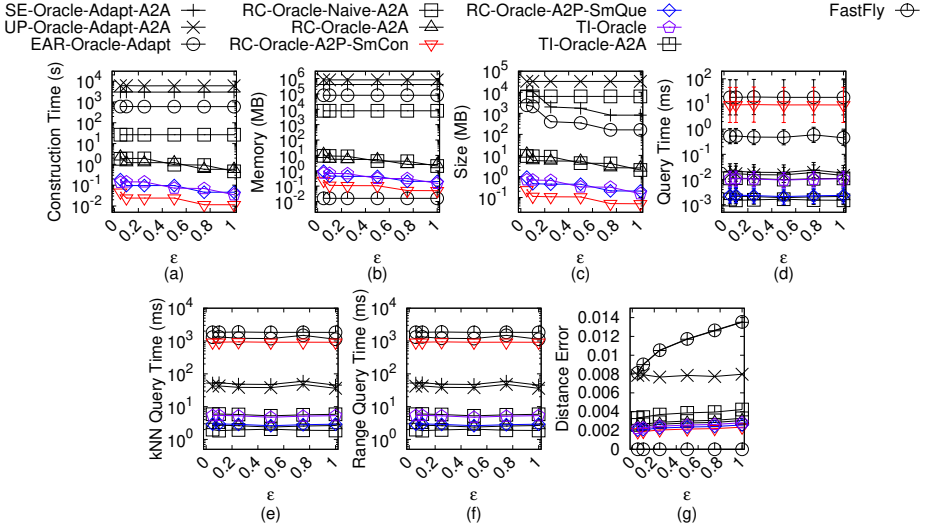

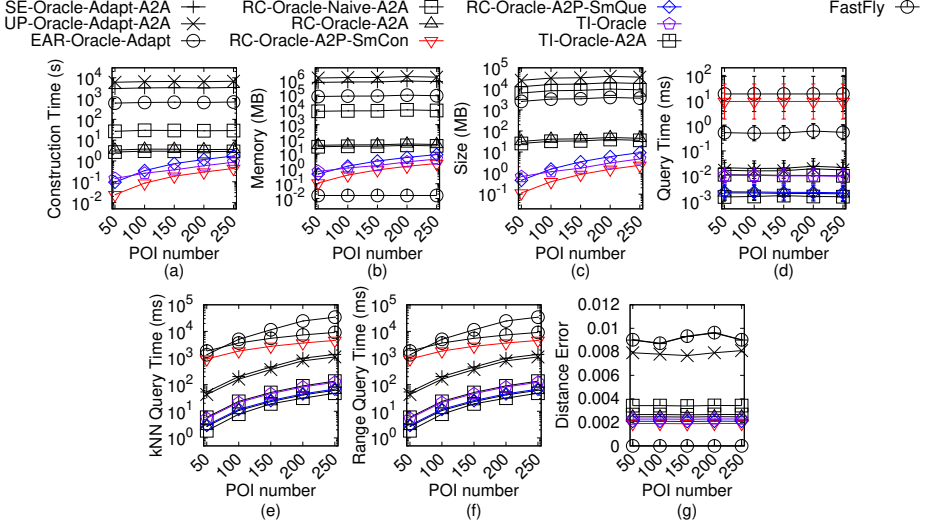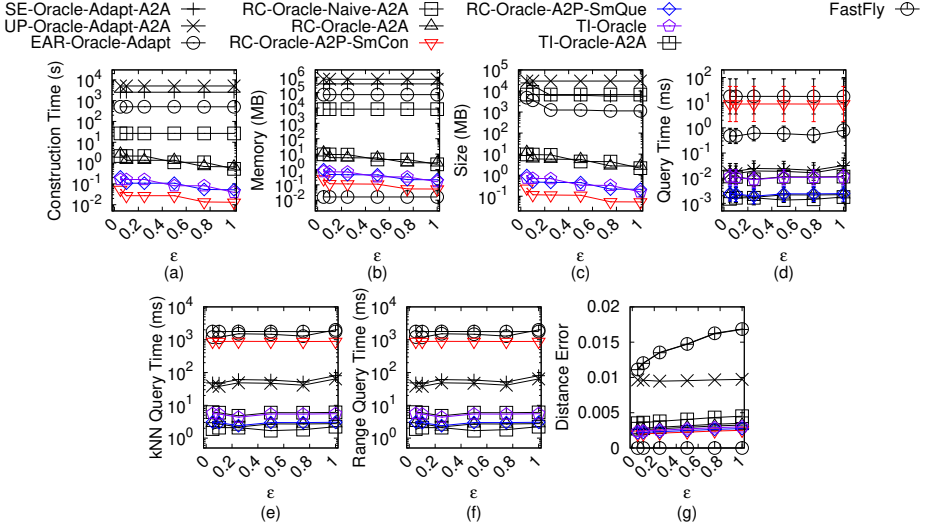Fig. 118. Baseline comparisons (effect of ϵ on GF_c-small point cloud dataset for A2P query)

has $|\Pi_{DIJ\text{-}Adapt}(s, t|T)| \leq k' \cdot |\Pi^*(s, t|T)|$ for any pair of vertices s and t on T. Compared with $\Pi^*(s, t|C)$, since we regard $\Pi^*(s, t|C)$ as the exact shortest path passing on C, algorithm *DIJ-Adapt* returns the approximate shortest path passing on C. □

THEOREM E.4. *The oracle construction time, oracle size and shortest path query time of SE-Oracle-Adapt are* $O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ *and* $O(h^2)$, *respectively. Consider a point cloud C and a TIN T converted from C. Compared with $\Pi^*(s, t|T)$, SE-Oracle-Adapt always has* $(1 - \epsilon)|\Pi^*(s, t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt}(s, t|T)| \leq (1 + \epsilon)|\Pi^*(s, t|T)|$ *for any pair of POIs s and t in P,*

Fig. 119. Baseline comparisons (effect of n on GF_c-small point cloud dataset for A2P query)

Fig. 120. Baseline comparisons (effect of ϵ on LM_c-small point cloud dataset for A2P query)

where $\Pi_{SE\text{-}Oracle\text{-}Adapt}(s, t|T)$ is the shortest surface path of SE-Oracle-Adapt passing on T between s and t. Compared with $\Pi^*(s, t|C)$, SE-Oracle-Adapt returns the approximate shortest path passing on C. 1866

1867

PROOF. Firstly, we show the *oracle construction time* of *SE-Oracle-Adapt*. The oracle construction time of the original oracle in [44, 45] is $O(nu + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, where u is the on-the-fly shortest path query time. In *SE-Oracle-Adapt*, we use algorithm *DIO* for the shortest path query on T, which has shortest path query time $O(N^2)$ according to Theorem E.1. So, we substitute u with N^2. But since *SE-Oracle-Adapt* first needs to convert C to T, it needs an additional $O(N)$ time for this step. Thus, the oracle construction time of *SE-Oracle-Adapt* is $O(N + nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh \log n) = O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$. 1868 1869 1870 1871 1872 1873

Fig. 121. Baseline comparisons (effect of n on LM_c-small point cloud dataset for A2P query)

Fig. 122. Baseline comparisons (effect of ϵ on RM_c-small point cloud dataset for A2P query)

Secondly, we show the *oracle size* of *SE-Oracle-Adapt*. The proof of the oracle size of *SE-Oracle-Adapt* is in [44, 45]. Thus, the oracle size of *SE-Oracle-Adapt* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we show the *shortest path query time* of *SE-Oracle-Adapt*. The proof of the shortest path query time of *SE-Oracle-Adapt* is in [44, 45]. Thus, the shortest path query time of *SE-Oracle-Adapt* is $O(h^2)$.

Fourthly, we show the *error bound* of *SE-Oracle-Adapt*. The on-the-fly shortest path query algorithm in *SE-Oracle-Adapt* is algorithm *DIO*, which returns the exact surface shortest path passing on T according to Theorem E.1, the error of *SE-Oracle-Adapt* is due to the oracle part.
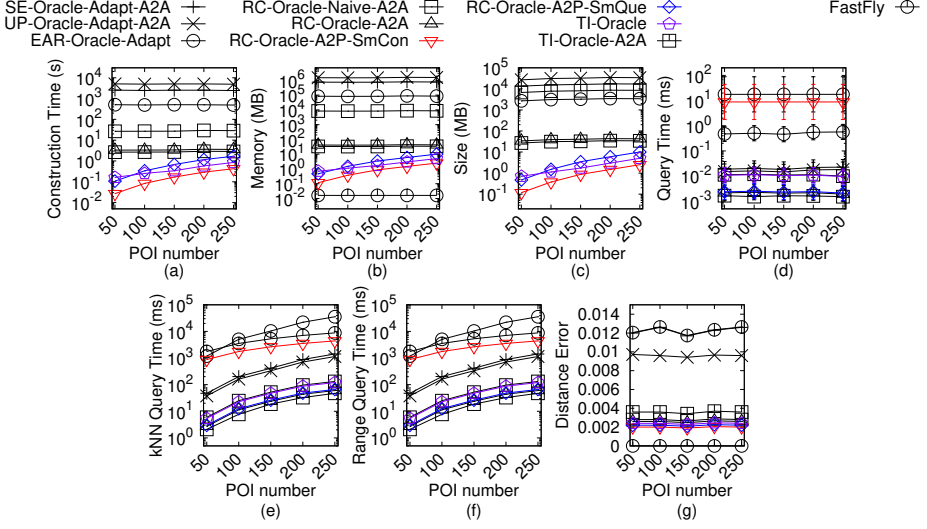
Fig. 123. Baseline comparisons (effect of n on RM_c-*small* point cloud dataset for A2P query)
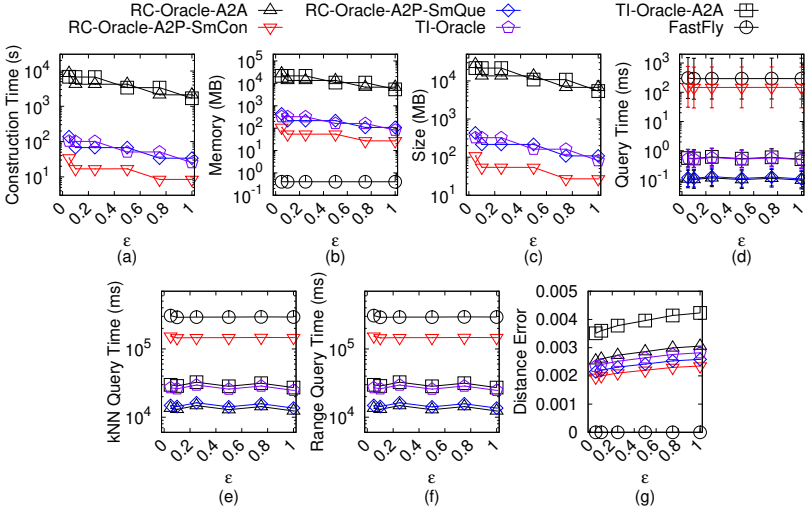
Fig. 124. Baseline comparisons (effect of ϵ on BH_c point cloud dataset for A2P query)

Compared with $\Pi^*(s, t|T)$, the proof of the error bound of the oracle part regarding *SE-Oracle-Adapt* is in [44, 45]. Since T is converted from C, we obtain that *SE-Oracle-Adapt* always has $(1 - \epsilon)|\Pi^*(s, t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt}(s, t|T)| \leq (1 + \epsilon)|\Pi^*(s, t|T)|$ for any pair of POIs s and t in P. Compared with $\Pi^*(s, t|C)$, since we regard $\Pi^*(s, t|C)$ as the exact shortest path passing on C, *SE-Oracle-Adapt* returns the approximate shortest path passing on C. □

THEOREM E.5. *The oracle construction time, oracle size and shortest path query time of SE-Oracle-FastFly-Adapt are* $O(nN \log N + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, $O(\frac{nh}{\epsilon^{2\beta}})$ *and* $O(h^2)$, *respectively. Consider a point cloud* C. *SE-Oracle-FastFly-Adapt always has* $(1 - \epsilon)|\Pi^*(s, t|C)| \leq |\Pi_{SE\text{-}Oracle\text{-}FastFly\text{-}Adapt}(s, t|C)| \leq$
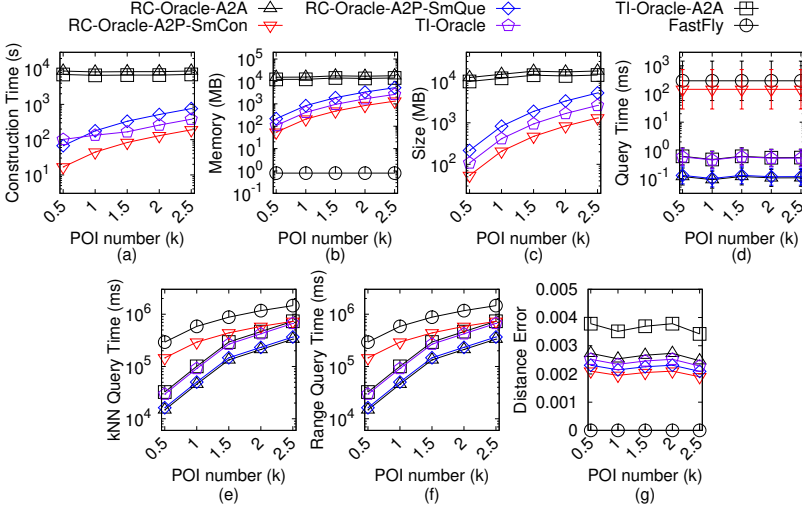
Fig. 125. Baseline comparisons (effect of n on BH_c point cloud dataset for A2P query)
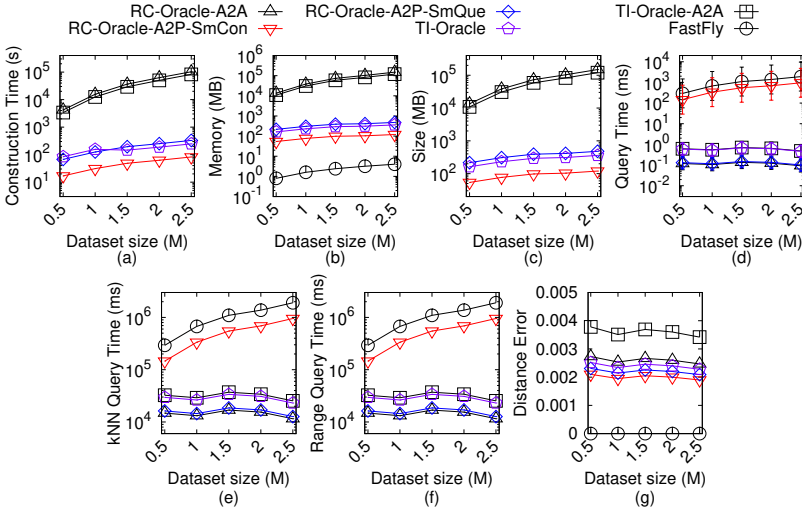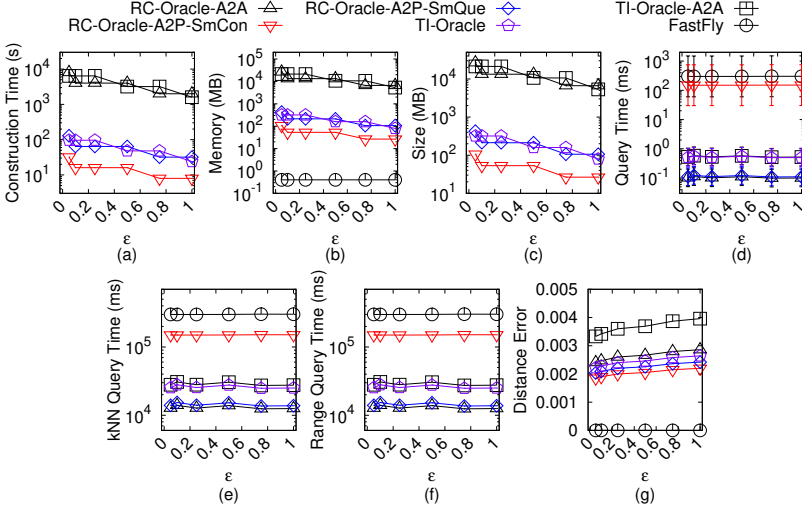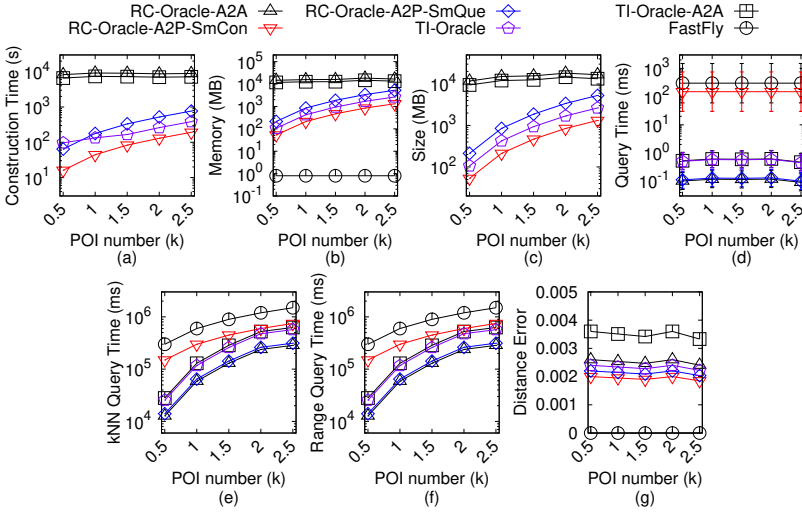

Fig. 126. Baseline comparisons (effect of N on BH_c point cloud dataset for A2P query)

$(1 + \epsilon)|\Pi^*(s, t|C)|$ for any pair of POIs s and t in P, where $\Pi_{SE\text{-}Oracle\text{-}FastFly\text{-}Adapt}(s, t|C)$ is the shortest path of SE-Oracle-FastFly-Adapt passing on C between s and t.

Proof. Firstly, we show the *oracle construction time* of *SE-Oracle-FastFly-Adapt*. The oracle construction time of the original oracle in [44, 45] is $O(nu + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$, where u is the on-the-fly shortest path query time. In *SE-Oracle-FastFly-Adapt*, we use algorithm *FastFly* for the shortest path query on C, which has the shortest path query time $O(N \log N)$ according to Theorem 4.1. We substitute u with $N \log N$. Thus, the oracle construction time of *SE-Oracle-FastFly-Adapt* is $O(nN \log N + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$.

Fig. 127. Baseline comparisons (effect of ϵ on EP_c point cloud dataset for A2P query)

Fig. 128. Baseline comparisons (effect of n on EP_c point cloud dataset for A2P query)

Secondly, we show the *oracle size* of *SE-Oracle-FastFly-Adapt*. The proof of the oracle size of *SE-Oracle-FastFly-Adapt* is in [44, 45]. Thus, the oracle size of *SE-Oracle-FastFly-Adapt* is $O(\frac{nh}{\epsilon^{2\beta}})$.

Thirdly, we show the *shortest path query time* of *SE-Oracle-FastFly-Adapt*. The proof of the shortest path query time of *SE-Oracle-FastFly-Adapt* is in [44, 45]. Thus, the shortest path query time of *SE-Oracle-FastFly-Adapt* is $O(h^2)$.

Fourthly, we show the *error bound* of *SE-Oracle-FastFly-Adapt*. The on-the-fly shortest path query algorithm in *SE-Oracle-FastFly-Adapt* is algorithm *FastFly*, which returns the exact shortest path passing on C according to Theorem 4.1, the error of *SE-Oracle-FastFly-Adapt* is due to the oracle part. The proof of the error bound of the oracle part regarding *SE-Oracle-FastFly-Adapt* is in [44, 45]. So, we
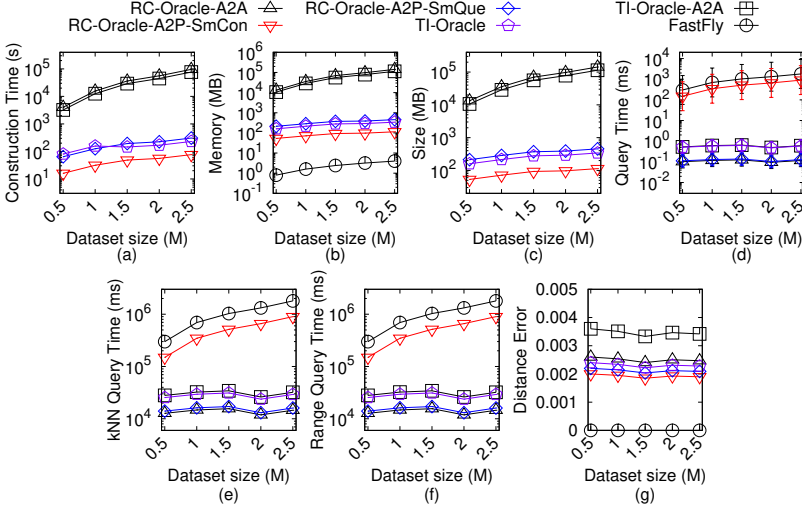
Fig. 129. Baseline comparisons (effect of N on EP_c point cloud dataset for A2P query)
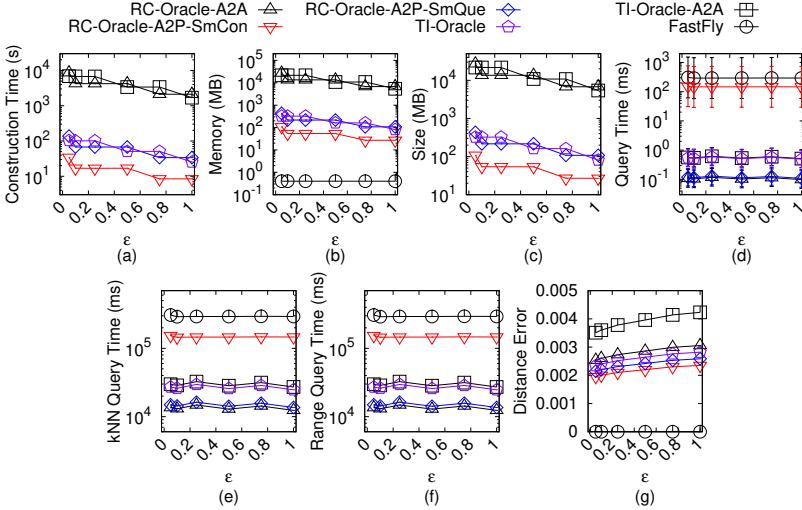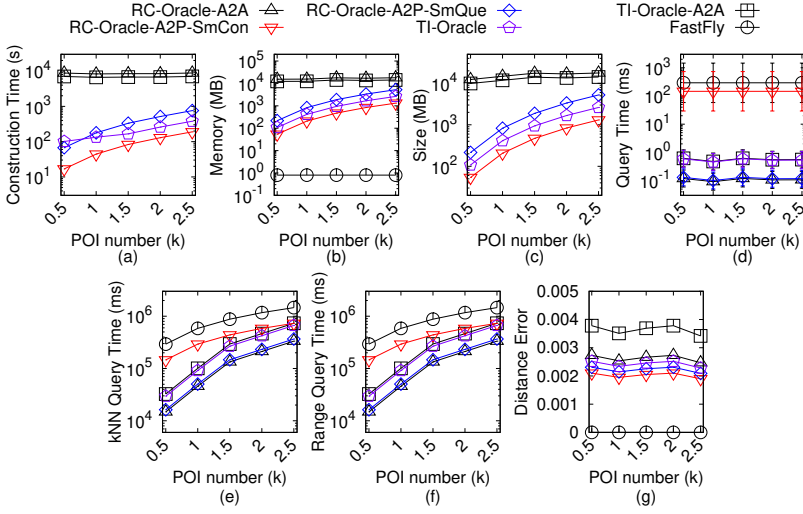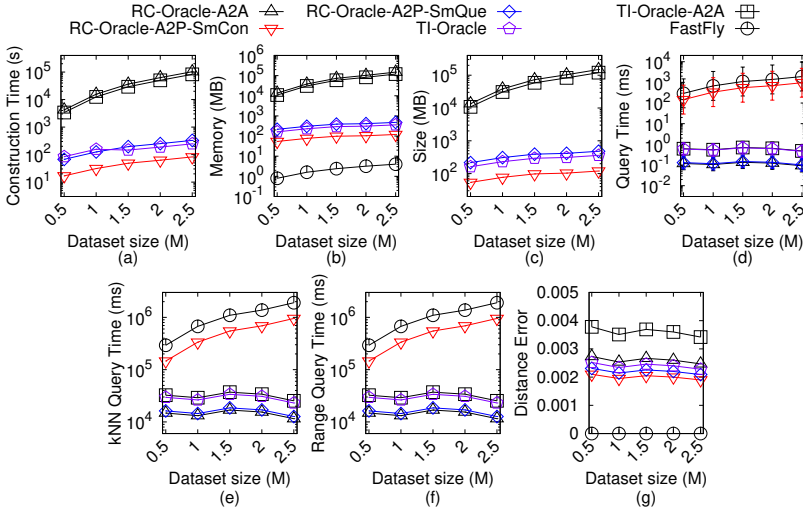

Fig. 130. Baseline comparisons (effect of ϵ on GF_c point cloud dataset for A2P query)

obtain that *SE-Oracle-FastFly-Adapt* always has $(1-\epsilon)|\Pi^*(s,t|C)| \leq |\Pi_{SE\text{-}Oracle\text{-}FastFly\text{-}Adapt}(s,t|C)| \leq$ $(1+\epsilon)|\Pi^*(s,t|C)|$ for any pair of POIs s and t in P. $\hfill\square$

THEOREM E.6. *The oracle construction time, oracle size and shortest path query time of SE-Oracle-Adapt-A2A are* $O\left(\frac{N^3}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}+\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log+\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log\left(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\right)\right)$, $O\left(\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log\frac{1}{\epsilon}\right)$ *and* $O(h^2)$, *respectively. Consider a point cloud C and a TIN T converted from C. Compared with* $\Pi^*(s,t|T)$, *SE-Oracle-Adapt-A2A always has* $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt\text{-}A2A}(s,t|T)| \leq$ $(1+\epsilon)|\Pi^*(s,t|T)|$ *for any pair of points s and t on T, where $\Pi_{SE\text{-}Oracle\text{-}Adapt\text{-}A2A}(s,t|T)$ is the shortest surface path of SE-Oracle-Adapt-A2A passing on T between s and t. Compared with $\Pi^*(s,t|C)$, SE-Oracle-Adapt-A2A returns the approximate shortest path passing on C.*

Fig. 131. Baseline comparisons (effect of n on GF_c point cloud dataset for A2P query)

Fig. 132. Baseline comparisons (effect of N on GF_c point cloud dataset for A2P query)

PROOF. Firstly, we show the *oracle construction time* of *SE-Oracle-Adapt-A2A*. We need to place POIs on faces of T using the method in [44, 45] to adapt *SE-Oracle-Adapt* to *SE-Oracle-Adapt-A2A*, and there are $O(\frac{1}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon})$ POIs per face. Since there are $O(N)$ faces, there are total $O(\frac{N}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon})$ POIs. We substitute n of *SE-Oracle-Adapt* in Theorem E.4 with $\frac{N}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon}$. But since *SE-Oracle-Adapt-A2A* first needs to convert C to T, it needs an additional $O(N)$ time for this step. Thus, the oracle construction time of *SE-Oracle-Adapt-A2A* is $O(N + \frac{N^3}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon} + \frac{Nh}{\sin\theta \cdot \sqrt{\epsilon} \cdot \epsilon^{2\beta}} \log + \frac{Nh}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon} \log(\frac{N}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon})) = O(\frac{N^3}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon} + \frac{Nh}{\sin\theta \cdot \sqrt{\epsilon} \cdot \epsilon^{2\beta}} \log + \frac{Nh}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon} \log(\frac{N}{\sin\theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon}))$.
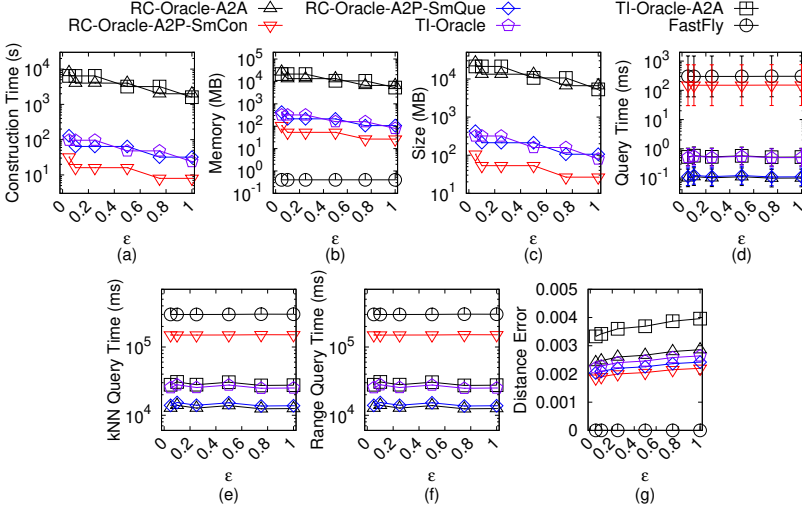
Fig. 133. Baseline comparisons (effect of ϵ on LM_c point cloud dataset for A2P query)

Fig. 134. Baseline comparisons (effect of n on LM_c point cloud dataset for A2P query)

Secondly, we show the *oracle size* of *SE-Oracle-Adapt-A2A*. We substitute n of *SE-Oracle-Adapt* in Theorem E.4 with $\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}$. Thus, the oracle size of *SE-Oracle-Adapt-A2A* is $O(\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log\frac{1}{\epsilon})$.

Thirdly, we show the *shortest path query time* of *SE-Oracle-Adapt-A2A*. The shortest path query time of *SE-Oracle-Adapt-A2A* is the same as *SE-Oracle-Adapt*, which is $O(h^2)$.

Fourthly, we show the *error bound* of *SE-Oracle-Adapt-A2A*. *SE-Oracle-Adapt* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any pair of POIs s and t in P. According to the POIs placed in [44, 45], we obtain that *SE-Oracle-Adapt-A2A* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{SE\text{-}Oracle\text{-}Adapt\text{-}A2A}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for any pair of points s and t on C. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on C, *SE-Oracle-Adapt-A2A* returns the approximate shortest path passing on C.   □

Fig. 135. Baseline comparisons (effect of N on LM_c point cloud dataset for A2P query)

Fig. 136. Baseline comparisons (effect of ϵ on RM_c point cloud dataset for A2P query)

THEOREM E.7. *The oracle construction time, oracle size and shortest path query time of SE-Oracle-FastFly-Adapt-A2A are* $O(\frac{N^2 \log N}{\sin \theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon} + \frac{Nh}{\sin \theta \cdot \sqrt{\epsilon} \cdot \epsilon^{2\beta}} \log + \frac{Nh}{\sin \theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon} \log(\frac{N}{\sin \theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon}))$, $O(\frac{Nh}{\sin \theta \cdot \sqrt{\epsilon} \cdot \epsilon^{2\beta}} \log \frac{1}{\epsilon})$ *and* $O(h^2)$, *respectively. Consider a point cloud C. SE-Oracle-FastFly-Adapt-A2A always has* $(1-\epsilon)|\Pi^*(s,t|C)| \leq |\Pi_{SE\text{-}Oracle\text{-}FastFly\text{-}Adapt\text{-}A2A}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ *for any pair of points s and t on C, where* $\Pi_{SE\text{-}Oracle\text{-}FastFly\text{-}Adapt\text{-}A2A}(s,t|C)$ *is the shortest path of SE-Oracle-FastFly-Adapt-A2A passing on C between s and t.*

PROOF. Firstly, we show the *oracle construction time* of *SE-Oracle-FastFly-Adapt-A2A*. The oracle construction time of *SE-Oracle-Adapt-A2A* is $O(\frac{uN}{\sin \theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon} +$

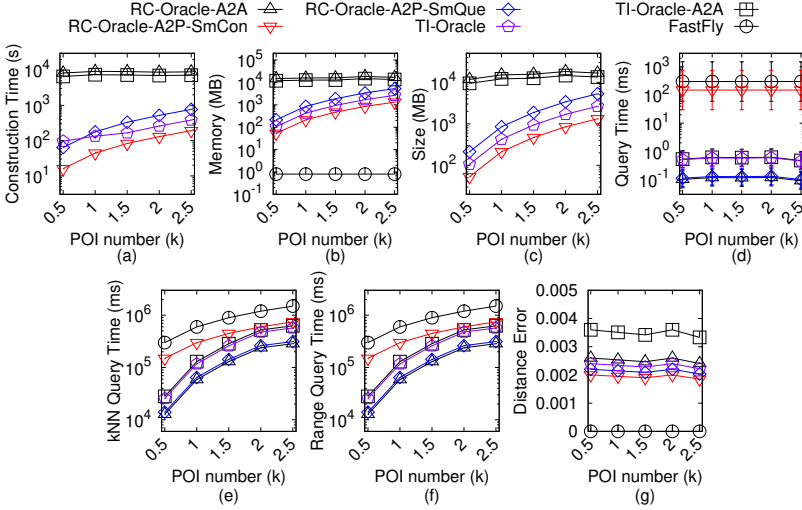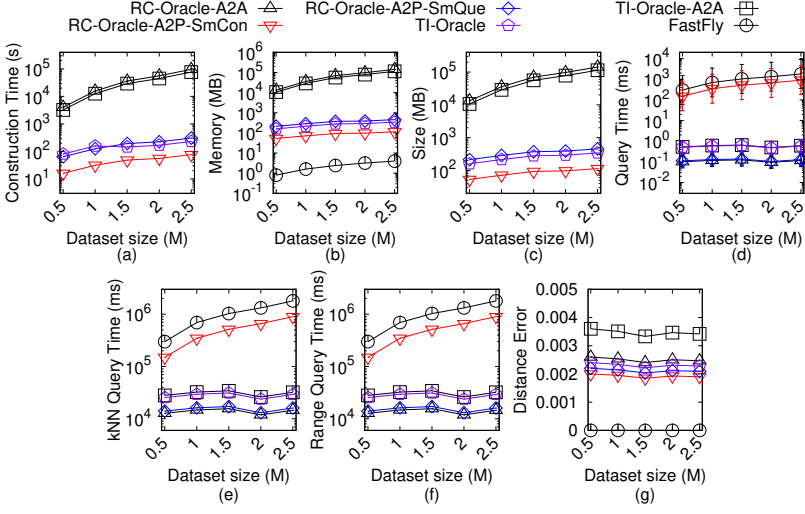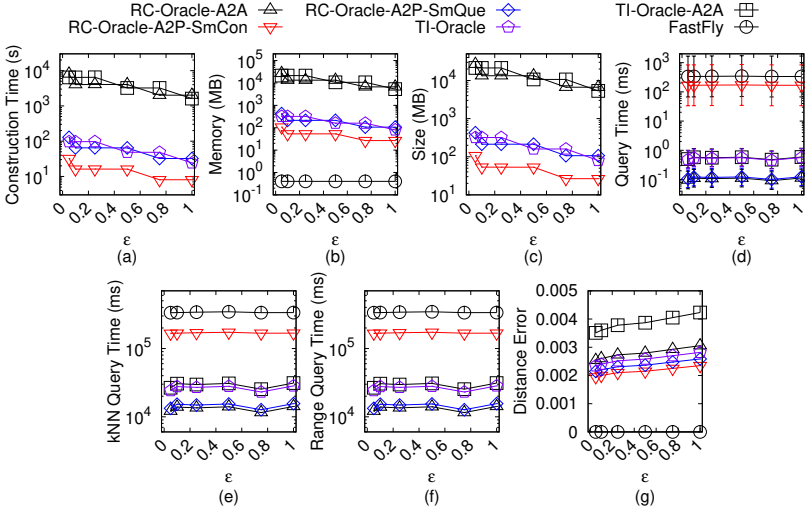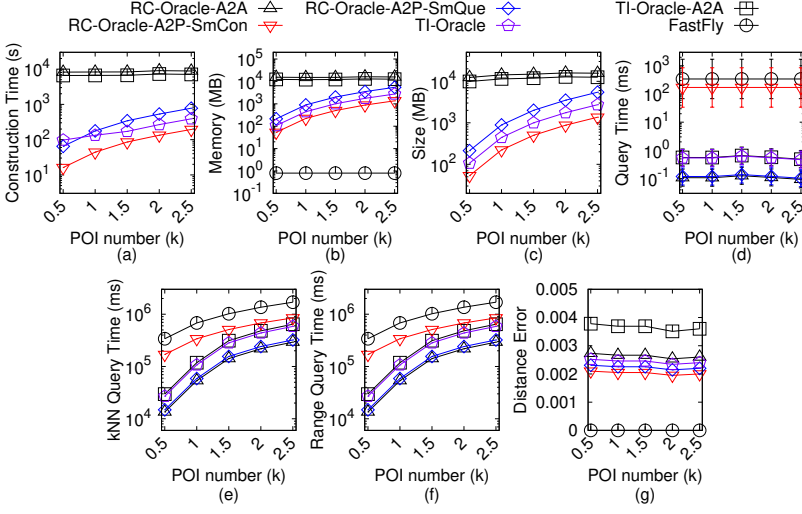Fig. 137. Baseline comparisons (effect of n on RM_c point cloud dataset for A2P query)
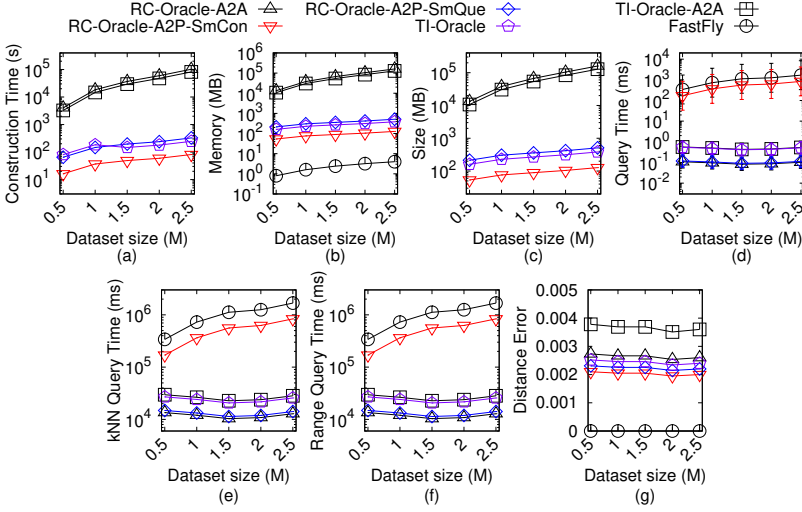
Fig. 138. Baseline comparisons (effect of N on RM_c point cloud dataset for A2P query)

$\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$, where u is the on-the-fly shortest path query time. In *SE-Oracle-FastFly-Adapt-A2A*, we use algorithm *FastFly* for the shortest path query on C, which has the shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute u with $N\log N$. Thus, the oracle construction time of *SE-Oracle-FastFly-Adapt-A2A* is $O(\frac{N^2\log N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log + \frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}\log(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}))$.

Secondly, we show the *oracle size* of *SE-Oracle-FastFly-Adapt-A2A*. The proof of the oracle size of *SE-Oracle-FastFly-Adapt-A2A* is the same as that of *SE-Oracle-Adapt-A2A*. Thus, the oracle size of *SE-Oracle-FastFly-Adapt-A2A* is $O(\frac{Nh}{\sin\theta\cdot\sqrt{\epsilon}\cdot\epsilon^{2\beta}}\log\frac{1}{\epsilon})$.
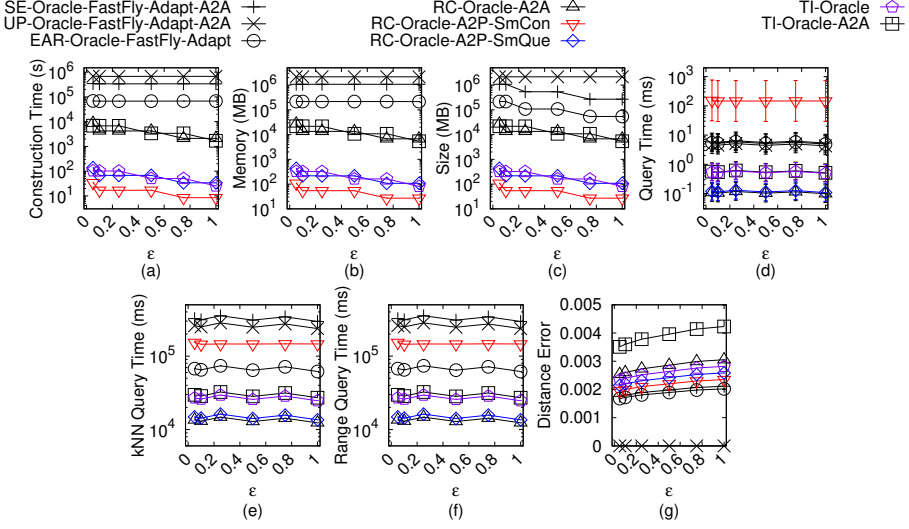
Fig. 139. Ablation study on BH_c point cloud dataset for A2P query
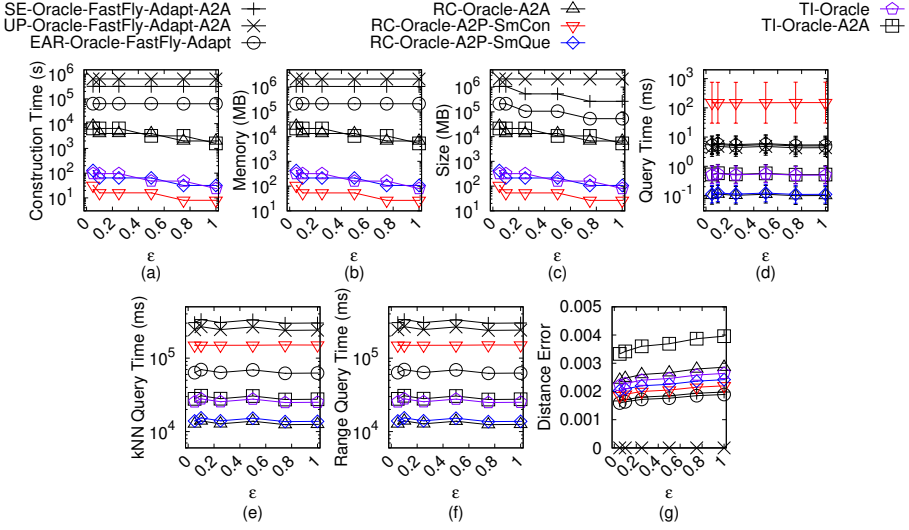
Fig. 140. Ablation study on EP_c point cloud dataset for A2P query

Thirdly, we show the *shortest path query time* of *SE-Oracle-FastFly-Adapt-A2A*. The shortest path query time of *SE-Oracle-FastFly-Adapt-A2A* is the same as *SE-Oracle-FastFly-Adapt*, which is $O(h^2)$.

Fourthly, we show the *error bound* of *SE-Oracle-FastFly-Adapt-A2A*. The on-the-fly shortest path query algorithm in *SE-Oracle-FastFly-Adapt-A2A* is algorithm *FastFly*, which returns the exact shortest path passing on C according to Theorem 4.1, the error of *SE-Oracle-FastFly-Adapt-A2A* is due to the oracle part. The proof of the error bound of the oracle part regarding *SE-Oracle-FastFly-Adapt-A2A* is the same as that of *SE-Oracle-Adapt-A2A*. So, we obtain that *SE-Oracle-FastFly-Adapt-A2A* always has $(1 - \epsilon)|\Pi^*(s, t|C)| \le |\Pi_{SE\text{-}Oracle\text{-}FastFly\text{-}Adapt\text{-}A2A}(s, t|C)| \le (1 + \epsilon)|\Pi^*(s, t|C)|$ for any pair of points s and t on C. □

Fig. 141. Ablation study on GF_c point cloud dataset for A2P query
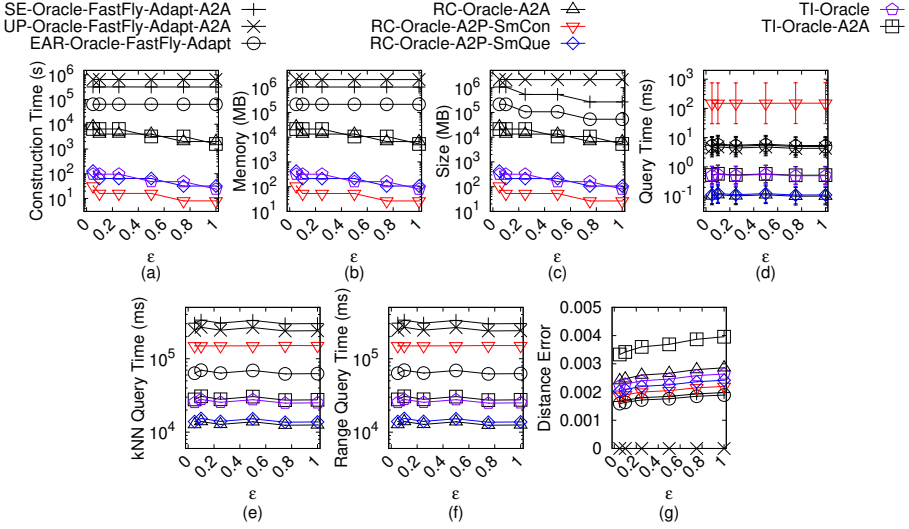
Fig. 142. Ablation study on LM_c point cloud dataset for A2P query

THEOREM E.8. *The oracle construction time, oracle size and shortest path query time of UP-Oracle-Adapt are $O(nN^2 + n^2)$, $O(n^2)$ and $O(1)$, respectively. Consider a point cloud C and a TIN T converted from C. Compared with $\Pi^*(s, t|T)$, UP-Oracle-Adapt returns the exact shortest path passing on T. Compared with $\Pi^*(s, t|C)$, UP-Oracle-Adapt returns the approximate shortest path passing on C.*

PROOF. Firstly, we show the *oracle construction time* of *UP-Oracle-Adapt*. The oracle construction time of the original oracle is $O(nu)$ since it needs to calculate the pairwise shortest paths, where u is the on-the-fly shortest path query time. In *UP-Oracle-Adapt*, we use algorithm *DIO* for the shortest path query on T, which has shortest path query time $O(N^2)$ according to Theorem E.1. So, we substitute u with N^2. It then needs $O(n^2)$ time to store the pairwise P2P shortest paths
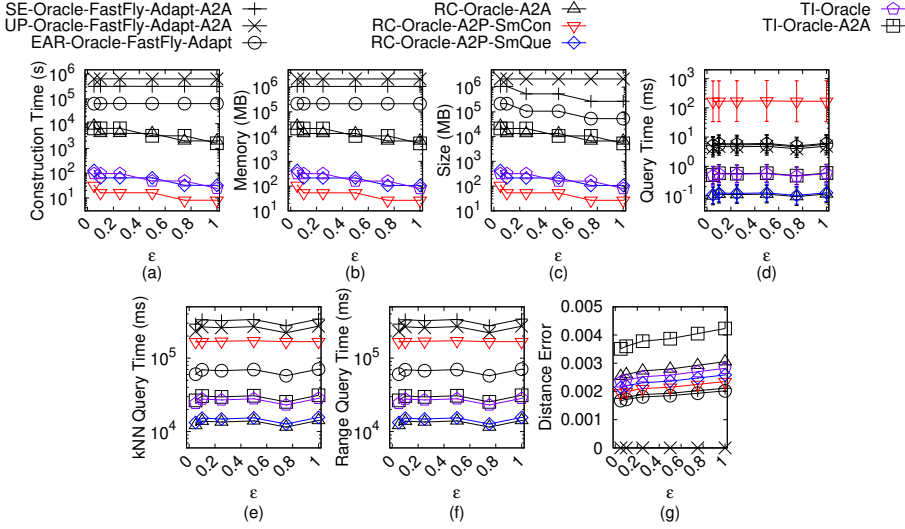
Fig. 143. Ablation study on RM_c point cloud dataset for A2P query
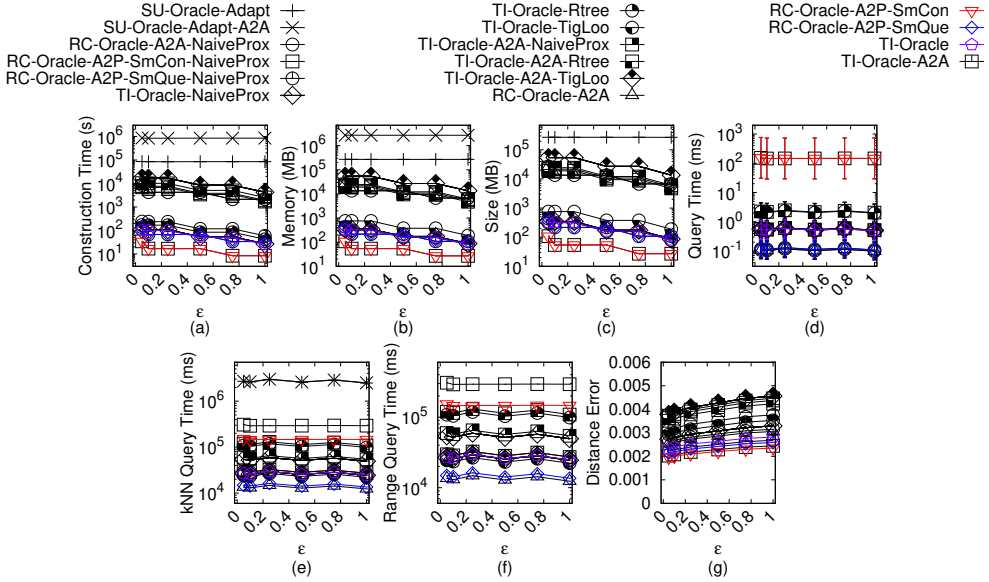
Fig. 144. Comparisons with other proximity queries oracles and variation oracles on BH_c point cloud dataset for A2P query

passing on T into a hash table. But since *UP-Oracle-Adapt* first needs to convert C to T, it needs an additional $O(N)$ time for this step. Thus, the oracle construction time of *UP-Oracle-Adapt* is $O(N + nN^2 + n^2) = O(nN^2 + n^2)$.

Secondly, we show the *oracle size* of *UP-Oracle-Adapt*. It needs to store the pairwise shortest paths and there are n POIs. Thus, the oracle size of *UP-Oracle-Adapt* is $O(n^2)$.
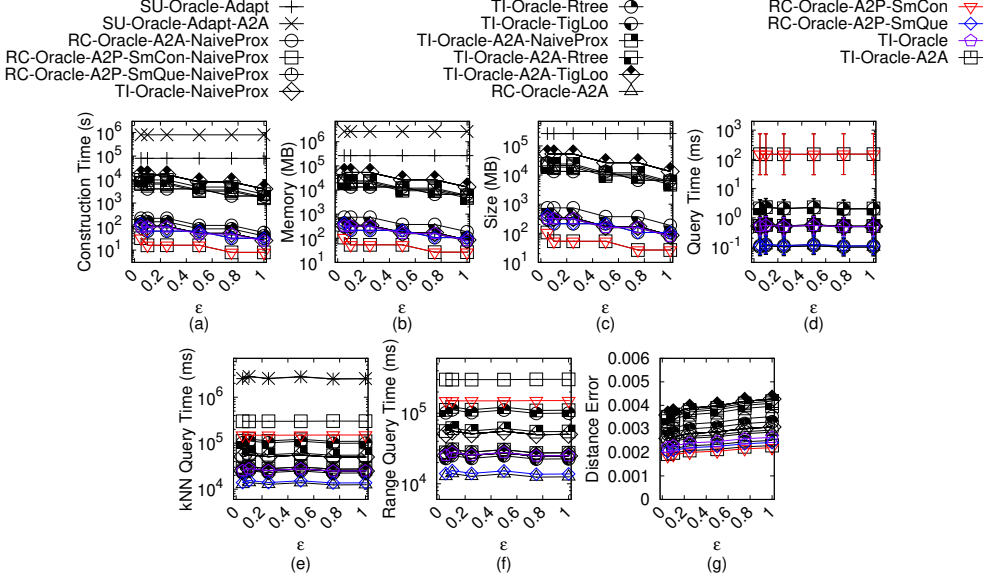
Fig. 145. Comparisons with other proximity queries oracles and variation oracles on EP_c point cloud dataset for A2P query
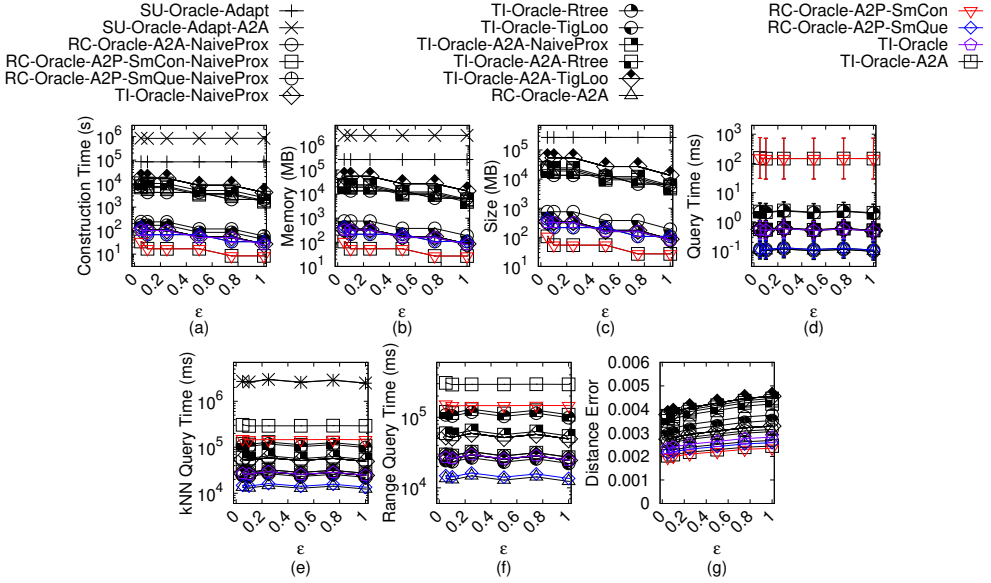
Fig. 146. Comparisons with other proximity queries oracles and variation oracles on GF_c point cloud dataset for A2P query

Thirdly, we show the *shortest path query time* of *UP-Oracle-Adapt*. It can directly return the shortest path since it stores the pairwise shortest paths. Thus, the shortest path query time of *UP-Oracle-Adapt* is $O(1)$.

1973
1974
1975
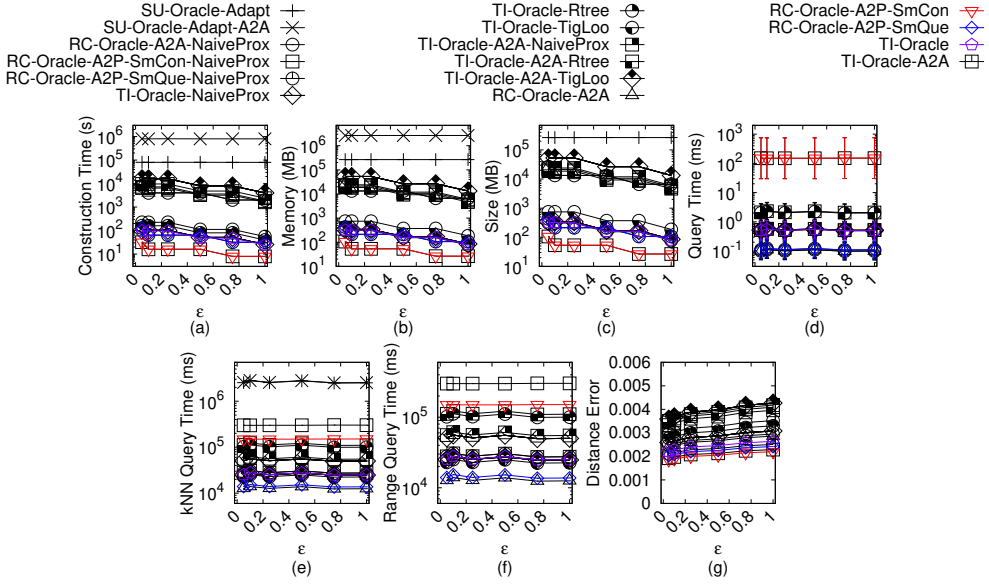
Fig. 147. Comparisons with other proximity queries oracles and variation oracles on LM_c point cloud dataset for A2P query
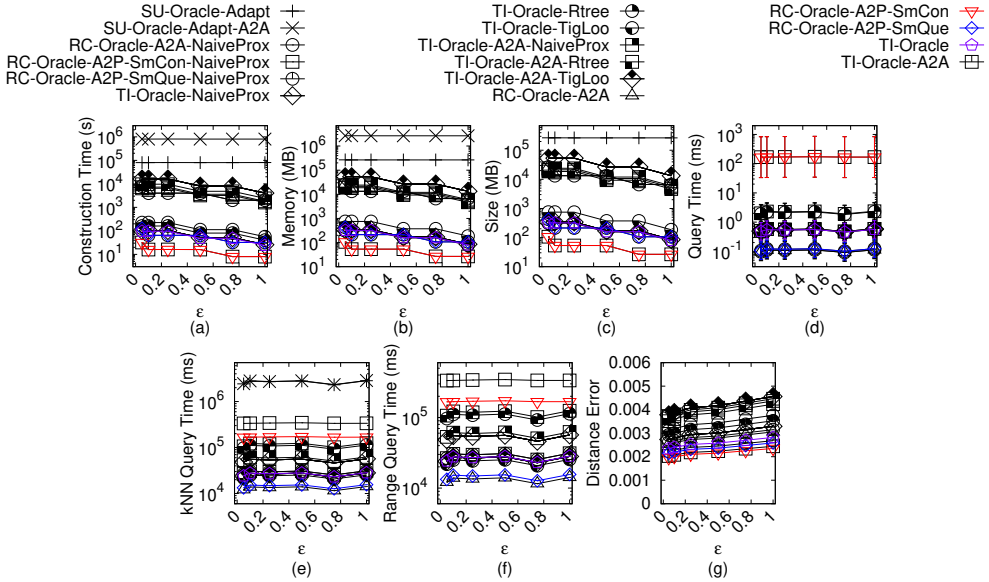
Fig. 148. Comparisons with other proximity queries oracles and variation oracles on RM_c point cloud dataset for A2P query

Fourthly, we show the *error bound* of *UP-Oracle-Adapt*. The on-the-fly shortest path query algorithm in *UP-Oracle-Adapt* is algorithm *DIO*, which returns the exact surface shortest path passing on T according to Theorem E.1, the error of *UP-Oracle-Adapt* is due to the oracle part. Compared with $\Pi^*(s, t|T)$, since *UP-Oracle-Adapt* stores the pairwise shortest paths passing on T,

Table 5. Summary of all notation

Notation	Meaning
C	The point cloud with a set of points
N	The number of points of C
L	The maximum side length of C
$N(p)$	A set of neighbor points of p
P	The set of POI
n	The number of vertices of P
G	The point cloud graph of C
$G.V / G.E$	The set of vertices and edges of G
$d_E(p, p')$	The Euclidean distance between point p and p'
$\Pi^*(s, t\|C)$	The exact shortest path passing on C between s and t
$\|\Pi^*(s, t\|C)\|$	The distance of $\Pi^*(s, t\|C)$
ϵ	The error parameter
T	The *TIN* converted from C
$\Pi^*(s, t\|T)$	The exact shortest surface path passing on T between s and t
$\Pi_W(s, t\|T)$	The shortest network path passing on T between s and t
$\Pi_E(s, t\|T)$	The shortest path passing on the edges of T between s and t where these edges belongs to the faces that $\Pi^*(s, t\|T)$ passes
n'	The number of target objects in *kNN* or range query
$\Pi_A(s, t\|C)$	The shortest path passing on C between s and t returned by oracle A, where $A \in \{$*RC-Oracle, RC-Oracle-A2P-SmCon, RC-Oracle-A2P-SmQue, RC-Oracle-A2A, TI-Oracle, TI-Oracle-A2A*$\}$
M_{path}	The path map table
M_{end}	The endpoint map table
P_{remain}	A set of remaining POIs of P on C that we have not used algorithm *FastFly* to calculate the exact shortest path passing on C with $p_i \in P_{remain}$ as source
$D(q)$	A set of endpoints (which can be points on C or POIs in P) that we need to use algorithm *FastFly* to calculate the exact shortest path passing on C from q to $p_i \in D(q)$ as destinations
P'	A set of POIs that are in P and not in $D(q)$
B	A set of boundary points
M_{cont}	The containing point map table
M_{boun}	The boundary point map table
M_{belo}	The belonging point map table
$M_{intra\text{-}path}$	The intra-path map table
$M_{inter\text{-}path}$	The inter-path map table
$M_{inter\text{-}end}$	The inter-endpoint map table
B_1 / B_2	A set of boundary points of the partition cell that the source / destination belongs to
h	The height of the compressed partition tree
β	The largest capacity dimension
θ	The minimum inner angle of any face in T
l_{max} / l_{min}	The length of the longest / shortest edge of T
λ	The number of highway nodes in a minimum square
ξ	The square root of the number of boxes
m	The number of Steiner points per face

UP-Oracle-Adapt returns the exact shortest path passing on T. Compared with $\Pi^*(s, t\|C)$, since we   1980

Table 6. Comparison of algorithms (support the shortest path query) on a point cloud

Algorithm	Oracle construction time		Oracle size		Shortest path query time		Error	Query type
Oracle-based algorithm								
SE-Oracle-Adapt [44, 45]	$O(nN^2 + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$	Large	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small	P2P
SE-Oracle-FastFly -Adapt [44, 45]	$O(nN \log N + \frac{nh}{\epsilon^{2\beta}} + nh \log n)$	Medium	$O(\frac{nh}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small	P2P
SE-Oracle-Adapt -A2A [44, 45]	$O(\eta N^2 + \frac{\eta h}{\epsilon^{2\beta}} + \eta h \log \eta)$	Large	$O(\frac{\eta h}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small	P2P, A2P, A2A
SE-Oracle-FastFly -Adapt-A2A [44, 45]	$O(\eta N \log N + \frac{\eta h}{\epsilon^{2\beta}} + \eta h \log \eta)$	Medium	$O(\frac{\eta h}{\epsilon^{2\beta}})$	Medium	$O(h^2)$	Small	Small	P2P, A2P, A2A
UP-Oracle-Adapt [54]	$O(nN^2 + n^2)$	Large	$O(n^2)$	Large	$O(1)$	Small	Small	P2P
UP-Oracle-FastFly -Adapt [54]	$O(nN \log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Small	Small	P2P
UP-Oracle-Adapt -A2A [54]	$O(\eta N^2 + \eta^2)$	Large	$O(\eta^2)$	Medium	$O(1)$	Small	Small	P2P, A2P, A2A
UP-Oracle-FastFly -Adapt-A2A [54]	$O(\eta N \log N + \eta^2)$	Medium	$O(\eta^2)$	Medium	$O(1)$	Small	Small	P2P, A2P, A2A
EAR-Oracle-Adapt [25]	$O(\lambda \xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$	Large	$O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda \xi \log(\lambda \xi))$	Medium	Small	P2P, A2P, A2A
EAR-Oracle-FastFly -Adapt [25]	$O(\lambda \xi m N \log N + \frac{N \log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$	Medium	$O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$	Large	$O(\lambda \xi \log(\lambda \xi))$	Medium	Small	P2P, A2P, A2A
RC-Oracle-Naive	$O(nN \log N + n^2)$	Medium	$O(n^2)$	Large	$O(1)$	Small	Small	P2P
RC-Oracle-Naive-A2A	$O(N^2 \log N)$	Medium	$O(N^2)$	Large	$O(1)$	Small	Small	P2P, A2P, A2A
RC-Oracle	$O(\frac{N \log N}{\epsilon} + n \log n)$	Small	$O(\frac{n}{\epsilon})$	Small	$O(1)$	Small	Small	P2P
RC-Oracle-A2P-SmCon	$O(\frac{N \log N}{\epsilon} + n \log n)$	Small	$O(\frac{n}{\epsilon})$	Small	$O(N \log N)$	Medium	Small	P2P, A2P
RC-Oracle-A2P-SmQue	$O(\frac{N \log N}{\epsilon} + n \log n)$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(1)$	Small	Small	P2P, A2P
RC-Oracle-A2A	$O(\frac{N \log N}{\epsilon})$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(1)$	Small	Small	P2P, A2P, A2A
TI-Oracle	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(1)$	Small	Small	P2P, A2P
TI-Oracle-A2A	$O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(1)$	Small	Small	P2P, A2P, A2A
On-the-fly algorithm								
DIO-Adapt [15, 46]	-	N/A	-	N/A	$O(N^2)$	Large	Small	P2P, A2P, A2A
ESP-Adapt [26, 51]	-	N/A	-	N/A	$O(\gamma N \log(\gamma N))$	Large	Small	P2P, A2P, A2A
DIJ-Adapt [27]	-	N/A	-	N/A	$O(N \log N)$	Medium	Medium	P2P, A2P, A2A
FastFly	-	N/A	-	N/A	$O(N \log N)$	Medium	No error	P2P, A2P, A2A

Remark: $n << N$, h is the height of the compressed partition tree, β is the largest capacity dimension [44, 45], $\eta = \frac{N}{\sin \theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon}$, θ is the minimum inner angle of any face in T, λ is the number of highway nodes in a minimum square, ξ is the square root of the number of boxes, m is the number of Steiner points per face, $\gamma = \frac{l_{max}}{\epsilon l_{min} \sqrt{1 - \cos \theta}}$, l_{max} (resp. l_{min}) is the length of the longest (resp. shortest) edge of T.

regard $\Pi^*(s, t|C)$ as the exact shortest path passing on C, *UP-Oracle-Adapt* returns the approximate shortest path passing on C. □

THEOREM E.9. *The oracle construction time, oracle size and shortest path query time of UP-Oracle-FastFly-Adapt are $O(nN \log N + n^2)$, $O(n^2)$ and $O(1)$, respectively. Consider a point cloud C. UP-Oracle-FastFly-Adapt returns the exact shortest path passing on C.*

PROOF. Firstly, we show the *oracle construction time* of *UP-Oracle-FastFly-Adapt*. The oracle construction time of the original oracle is $O(nu)$ since it needs to calculate the pairwise shortest paths, where u is the on-the-fly shortest path query time. In *UP-Oracle-FastFly-Adapt*, we use algorithm *FastFly* for the shortest path query on C, which has the shortest path query time $O(N \log N)$ according to Theorem 4.1. We substitute u with $N \log N$. It then needs $O(n^2)$ time to store the

Table 7. Comparison of other proximity queries oracles and their variation oracles on a point cloud

Algorithm	Oracle construction time		Oracle size		kNN query time		Error	Query type
SU-Oracle-Adapt [40]	$O(n'N^2 \log N)$	Large	$O(n'N)$	Medium	$O(n'N^2 \log N)$	Large	No error	P2P, A2P
SU-Oracle-Adapt-A2A [40]	$O(N^2\sqrt{N} \log N)$	Large	$O(N\sqrt{N})$	Medium	$O(N^2\sqrt{N} \log N)$	Large	No error	A2A
RC-Oracle-NaiveProx	$O(\frac{N \log N}{\epsilon} + n \log n)$	Small	$O(\frac{n}{\epsilon})$	Small	$O(n')$	Small	Small	P2P
RC-Oracle-A2P-SmCon -NaiveProx	$O(\frac{N \log N}{\epsilon} + n \log n)$	Small	$O(\frac{n}{\epsilon})$	Small	$O(N \log N + n')$	Medium	Small	P2P, A2P
RC-Oracle-A2P-SmQue -NaiveProx	$O(\frac{N \log N}{\epsilon} + n \log n)$	Small	$O(\frac{n}{\epsilon})$	Small	$O(n')$	Small	Small	P2P, A2P
RC-Oracle-A2A-NaiveProx	$O(\frac{N \log N}{\epsilon})$	Small	$O(\frac{N}{\epsilon})$	Small	$O(n')$	Small	Small	P2P, A2P, A2A
RC-Oracle	$O(\frac{N \log N}{\epsilon} + n \log n)$	**Small**	$O(\frac{n}{\epsilon})$	**Small**	$O(n')$	**Small**	**Small**	**P2P**
RC-Oracle-A2P-SmCon	$O(\frac{N \log N}{\epsilon} + n \log n)$	**Small**	$O(\frac{n}{\epsilon})$	**Small**	$O(N \log N + n')$	Medium	Small	**P2P, A2P**
RC-Oracle-A2P-SmQue	$O(\frac{N \log N}{\epsilon} + n \log n)$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(n')$	**Small**	**Small**	**P2P, A2P**
RC-Oracle-A2A	$O(\frac{N \log N}{\epsilon})$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(n')$	**Small**	**Small**	**P2P, A2P, A2A**
TI-Oracle-NaiveProx	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P
TI-Oracle-Rtree	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P
TI-Oracle-TigLoo	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P
TI-Oracle-A2A-NaiveProx	$O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P, A2A
TI-Oracle-A2A-Rtree	$O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P, A2A
TI-Oracle-A2A-TigLoo	$O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$	Small	$O(\frac{N}{\epsilon})$	Medium	$O(n')$	Small	Small	P2P, A2P, A2A
TI-Oracle	$O(\frac{N \log N}{\epsilon} + Nn + n \log n)$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(n')$	**Small**	**Small**	**P2P, A2P**
TI-Oracle-A2A	$O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$	**Small**	$O(\frac{N}{\epsilon})$	**Medium**	$O(n')$	**Small**	**Small**	**P2P, A2P, A2A**

Remark: n' is the number of target objects.

pairwise P2P shortest paths passing on C into a hash table. Thus, the oracle construction time of *UP-Oracle-FastFly-Adapt* is $O(nN \log N + n^2)$.

Secondly, we show the *oracle size* of *UP-Oracle-FastFly-Adapt*. It needs to store the pairwise shortest paths and there are n POIs. Thus, the oracle size of *UP-Oracle-FastFly-Adapt* is $O(n^2)$.

Thirdly, we show the *shortest path query time* of *UP-Oracle-FastFly-Adapt*. It can directly return the shortest path since it stores the pairwise shortest paths. Thus, the shortest path query time of *UP-Oracle-FastFly-Adapt* is $O(1)$.

Fourthly, we show the *error bound* of *UP-Oracle-FastFly-Adapt*. The on-the-fly shortest path query algorithm in *UP-Oracle-FastFly-Adapt* is algorithm *FastFly*, which returns the exact shortest path passing on C according to Theorem 4.1, the error of *UP-Oracle-FastFly-Adapt* is due to the oracle part. Since *UP-Oracle-FastFly-Adapt* stores the pairwise shortest paths passing on C, we obtain that *UP-Oracle-FastFly-Adapt* returns the exact shortest path passing on C. □

THEOREM E.10. *The oracle construction time, oracle size and shortest path query time of UP-Oracle-Adapt-A2A are $O(\frac{N^3}{\sin \theta \cdot \sqrt{\epsilon}} \log \frac{1}{\epsilon} + \frac{N^2}{\sin^2 \theta \cdot \epsilon} \log^2 \frac{1}{\epsilon})$, $O(\frac{N^2}{\sin^2 \theta \cdot \epsilon} \log^2 \frac{1}{\epsilon})$ and $O(1)$, respectively. Consider a point cloud C and a TIN T converted from C. Compared with $\Pi^*(s, t|T)$, UP-Oracle-Adapt-A2A always has $(1-\epsilon)|\Pi^*(s, t|T)| \leq |\Pi_{UP-Oracle-Adapt-A2A}(s, t|T)| \leq (1+\epsilon)|\Pi^*(s, t|T)|$ for any pair of points s and t on T, where $\Pi_{UP-Oracle-Adapt-A2A}(s, t|T)$ is the shortest surface path of UP-Oracle-Adapt-A2A passing on T between s and t. Compared with $\Pi^*(s, t|C)$, UP-Oracle-Adapt-A2A returns the approximate shortest path passing on C.*

Proof. Firstly, we show the *oracle construction time* of *UP-Oracle-Adapt-A2A*. We need to place POIs on faces of T using the method in [44, 45] to adapt *UP-Oracle-Adapt* to *UP-Oracle-Adapt-A2A*, and there are $O(\frac{1}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon})$ POIs per face. Since there are $O(N)$ faces, there are total $O(\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon})$ POIs. We substitute n of *UP-Oracle-Adapt* in Theorem E.8 with $\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}$. It then needs $O(\frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon})$ time to store the pairwise P2P shortest paths passing on T into a hash table. But since *SE-Oracle-Adapt-A2A* first needs to convert C to T, it needs an additional $O(N)$ time for this step. Thus, the oracle construction time of *UP-Oracle-Adapt-A2A* is $O(N + \frac{N^3}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon}) = O(\frac{N^3}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon})$.

Secondly, we show the *oracle size* of *UP-Oracle-Adapt-A2A*. We substitute n of *UP-Oracle-Adapt* in Theorem E.8 with $\frac{N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon}$. Thus, the oracle size of *UP-Oracle-Adapt-A2A* is $O(\frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon})$.

Thirdly, we show the *shortest path query time* of *UP-Oracle-Adapt-A2A*. The shortest path query time of *UP-Oracle-Adapt-A2A* is the same as *UP-Oracle-Adapt*, which is $O(1)$.

Fourthly, we show the *error bound* of *UP-Oracle-Adapt-A2A*. Since *UP-Oracle-Adapt* returns the exact shortest path passing on T, according to the POIs placed in [44, 45], we obtain that *UP-Oracle-Adapt-A2A* always has $(1-\epsilon)|\Pi^*(s,t|T)| \leq |\Pi_{UP\text{-}Oracle\text{-}Adapt\text{-}A2A}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T)|$ for any pair of points s and t on C. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on C, *UP-Oracle-Adapt-A2A* returns the approximate shortest path passing on C. □

THEOREM E.11. *The oracle construction time, oracle size and shortest path query time of UP-Oracle-FastFly-Adapt-A2A are $O(\frac{N^2\log N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon})$, $O(\frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon})$ and $O(1)$, respectively. Consider a point cloud C. UP-Oracle-FastFly-Adapt-A2A always has $(1-\epsilon)|\Pi^*(s,t|C)| \leq |\Pi_{UP\text{-}Oracle\text{-}FastFly\text{-}Adapt\text{-}A2A}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any pair of points s and t on C, where $\Pi_{UP\text{-}Oracle\text{-}FastFly\text{-}Adapt\text{-}A2A}(s,t|C)$ is the shortest path of UP-Oracle-FastFly-Adapt-A2A passing on C between s and t.*

Proof. Firstly, we show the *oracle construction time* of *UP-Oracle-FastFly-Adapt-A2A*. The oracle construction time of *UP-Oracle-Adapt-A2A* is $O(\frac{uN}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon})$, where u is the on-the-fly shortest path query time. In *UP-Oracle-FastFly-Adapt-A2A*, we use algorithm *FastFly* for the shortest path query on C, which has the shortest path query time $O(N\log N)$ according to Theorem 4.1. We substitute u with $N\log N$. It then needs $O(\frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon})$ time to store the pairwise P2P shortest paths passing on C into a hash table. Thus, the oracle construction time of *UP-Oracle-FastFly-Adapt-A2A* is $O(\frac{N^2\log N}{\sin\theta\cdot\sqrt{\epsilon}}\log\frac{1}{\epsilon} + \frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon})$.

Secondly, we show the *oracle size* of *UP-Oracle-FastFly-Adapt-A2A*. The proof of the oracle size of *UP-Oracle-FastFly-Adapt-A2A* is the same as that of *UP-Oracle-Adapt-A2A*. Thus, the oracle size of *UP-Oracle-FastFly-Adapt-A2A* is $O(\frac{N^2}{\sin^2\theta\cdot\epsilon}\log^2\frac{1}{\epsilon})$.

Thirdly, we show the *shortest path query time* of *UP-Oracle-FastFly-Adapt-A2A*. The shortest path query time of *UP-Oracle-FastFly-Adapt-A2A* is the same as *UP-Oracle-FastFly-Adapt*, which is $O(1)$.

Fourthly, we show the *error bound* of *UP-Oracle-FastFly-Adapt-A2A*. The on-the-fly shortest path query algorithm in *UP-Oracle-FastFly-Adapt-A2A* is algorithm *FastFly*, which returns the exact shortest path passing on C according to Theorem 4.1, the error of *UP-Oracle-FastFly-Adapt-A2A* is due to the oracle part. The proof of the error bound of the oracle part regarding *UP-Oracle-FastFly-Adapt-A2A* is the same as that of *UP-Oracle-Adapt-A2A*. So, we obtain that *UP-Oracle-FastFly-Adapt-A2A* always has $(1-\epsilon)|\Pi^*(s,t|C)| \leq |\Pi_{UP\text{-}Oracle\text{-}FastFly\text{-}Adapt\text{-}A2A}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C)|$ for any pair of points s and t on C. □

Theorem E.12. *The oracle construction time, oracle size and shortest path query time of EAR-Oracle-Adapt are* $O(\lambda \xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$, $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ *and* $O(\lambda \xi \log(\lambda \xi))$, *respectively. Consider a point cloud* C *and a TIN* T *converted from* C. *Compared with* $\Pi^*(s,t|T)$, *EAR-Oracle-Adapt always has* $|\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T) + 2\delta|$ *for any pair of points* s *and* t *on* T, *where* $\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)$ *is the shortest surface path of EAR-Oracle-Adapt passing on* T *between* s *and* t *and* δ *is an error parameter* [25]. *Compared with* $\Pi^*(s,t|C)$, *EAR-Oracle-Adapt returns the approximate shortest path passing on* C.

Proof. Firstly, we show the *oracle construction time* of *EAR-Oracle-Adapt*. The oracle construction time of the original oracle in [25] is $O(\lambda \xi m u + \frac{u}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$, where u is the on-the-fly shortest path query time. In *EAR-Oracle-Adapt*, we use algorithm *DIO* for the shortest path query on T, which has the shortest path query time $O(N^2)$ according to Theorem E.1. So, we substitute u with N^2. But since *EAR-Oracle-Adapt* first needs to convert C to T, it needs an additional $O(N)$ time for this step. Thus, the oracle construction time of *EAR-Oracle-Adapt* is $O(N + \lambda \xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N) = O(\lambda \xi m N^2 + \frac{N^2}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$.

Secondly, we show the *oracle size* of *EAR-Oracle-Adapt*. The proof of the oracle size of *EAR-Oracle-Adapt* is $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ in [25]. Thus, the oracle size of *EAR-Oracle-Adapt* is $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$.

Thirdly, we show the *shortest path query time* of *EAR-Oracle-Adapt*. The proof of the shortest path query time of *EAR-Oracle-Adapt* is in [25]. Thus, the shortest path query time of *EAR-Oracle-Adapt* is $O(\lambda \xi \log(\lambda \xi))$.

Fourthly, we show the *error bound* of *EAR-Oracle-Adapt*. The on-the-fly shortest path query algorithm in *EAR-Oracle-Adapt* is algorithm *DIO*, which returns the exact surface shortest path passing on T according to Theorem E.1, so the error of *EAR-Oracle-Adapt* is due to the oracle part. Compared with $\Pi^*(s,t|T)$, the proof of the error bound of the oracle part regarding *EAR-Oracle-Adapt* is in [25]. Since T is converted from C, we obtain that *EAR-Oracle-Adapt* always has $|\Pi_{EAR\text{-}Oracle\text{-}Adapt}(s,t|T)| \leq (1+\epsilon)|\Pi^*(s,t|T) + 2\delta|$ for any pair of points s and t on T. Compared with $\Pi^*(s,t|C)$, since we regard $\Pi^*(s,t|C)$ as the exact shortest path passing on C, *EAR-Oracle-Adapt* returns the approximate shortest path passing on C. □

Theorem E.13. *The oracle construction time, oracle size and shortest path query time of EAR-Oracle-FastFly-Adapt are* $O(\lambda \xi m N \log N + \frac{N \log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$, $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$ *and* $O(\lambda \xi \log(\lambda \xi))$, *respectively. Consider a point cloud* C. *EAR-Oracle-FastFly-Adapt always has* $|\Pi_{EAR\text{-}Oracle\text{-}FastFly\text{-}Adapt}(s,t|C)| \leq (1+\epsilon)|\Pi^*(s,t|C) + 2\delta|$ *for any pair of points* s *and* t *on* C, *where* $\Pi_{EAR\text{-}Oracle\text{-}FastFly\text{-}Adapt}(s,t|C)$ *is the shortest path of EAR-Oracle-FastFly-Adapt passing on* C *between* s *and* t *and* δ *is an error parameter* [25].

Proof. Firstly, we show the *oracle construction time* of *EAR-Oracle-FastFly-Adapt*. The oracle construction time of the original oracle in [25] is $O(\lambda \xi m u + \frac{u}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$, where u is the on-the-fly shortest path query time. In *EAR-Oracle-FastFly-Adapt*, we use algorithm *FastFly* for the shortest path query on C, which has the shortest path query time $O(N \log N)$ according to Theorem 4.1. We substitute u with $N \log N$. Thus, the oracle construction time of *EAR-Oracle-FastFly-Adapt* is $O(\lambda \xi m N \log N + \frac{N \log N}{\epsilon^{2\beta}} + \frac{Nh}{\epsilon^{2\beta}} + Nh \log N)$.

Secondly, we show the *oracle size* of *EAR-Oracle-FastFly-Adapt*. The proof of the oracle size of *EAR-Oracle-FastFly-Adapt* is in [25]. Thus, the oracle size of *EAR-Oracle-FastFly-Adapt* is $O(\frac{\lambda m N}{\xi} + \frac{Nh}{\epsilon^{2\beta}})$.

Thirdly, we show the *shortest path query time* of *EAR-Oracle-FastFly-Adapt*. The proof of the shortest path query time of *EAR-Oracle-FastFly-Adapt* is in [25]. Thus, the shortest path query time of *EAR-Oracle-FastFly-Adapt* is $O(\lambda \xi \log(\lambda \xi))$.

Fourthly, we show the *error bound* of *EAR-Oracle-FastFly-Adapt*. The on-the-fly shortest path query algorithm in *EAR-Oracle-FastFly-Adapt* is algorithm *FastFly*, which returns the exact shortest path passing on C according to Theorem 4.1, the error of *EAR-Oracle-FastFly-Adapt* is due to the oracle part. The proof of the error bound of the oracle part regarding *EAR-Oracle-FastFly-Adapt* is in [25]. So, we obtain that *EAR-Oracle-FastFly-Adapt* always has $|\Pi_{EAR\text{-}Oracle\text{-}FastFly\text{-}Adapt}(s, t|C)| \leq (1 + \epsilon)|\Pi^*(s, t|C) + 2\delta|$ for any pair of points s and t in C. □

THEOREM E.14. *The oracle construction time, oracle size and shortest path query time of RC-Oracle-Naive are $O(nN \log N + n^2)$, $O(n^2)$ and $O(1)$, respectively. RC-Oracle-Naive returns the exact shortest path passing on a point cloud.*

PROOF. Firstly, we show the *oracle construction time* of *RC-Oracle-Naive*. Since there are total n POIs, *RC-Oracle-Naive* first needs $O(nm)$ time to calculate the shortest path passing on the point cloud from each POI to all other remaining POIs using on-the-fly shortest path query algorithm, where m is the on-the-fly shortest path query time. It then needs $O(n^2)$ time to store the pairwise P2P shortest paths passing on the point cloud into a hash table. In *RC-Oracle-Naive*, we use algorithm *FastFly* for the point cloud shortest path query, which has the shortest path query time $O(N \log N)$ according to Theorem 4.1. We substitute m with $N \log N$. Thus, the oracle construction time of *RC-Oracle-Naive* is $O(nN \log N + n^2)$.

Secondly, we show the *oracle size* of *RC-Oracle-Naive*. *RC-Oracle-Naive* stores $O(n^2)$ pairwise P2P shortest paths passing on the point cloud. Thus, the oracle size of *RC-Oracle-Naive* is $O(n^2)$.

Thirdly, we show the *shortest path query time* of *RC-Oracle-Naive*. *RC-Oracle-Naive* has a hash table to store the pairwise P2P shortest paths passing on the point cloud. Thus, the shortest path query time of *RC-Oracle-Naive* is $O(1)$.

Fourthly, we show the *error bound* of *RC-Oracle-Naive*. The on-the-fly shortest path query algorithm in *RC-Oracle-Naive* is algorithm *FastFly*, which returns the exact shortest path passing on the point cloud according to Theorem 4.1. Since *RC-Oracle-Naive* also computes the pairwise P2P exact shortest paths passing on the point cloud, so it returns the exact shortest path passing on the point cloud. □

THEOREM E.15. *The oracle construction time, oracle size and shortest path query time of RC-Oracle-Naive-A2A are $O(N^2 \log N)$, $O(N^2)$ and $O(1)$, respectively. RC-Oracle-Naive returns the exact shortest path passing on a point cloud.*

PROOF. Firstly, we show the *oracle construction time* of *RC-Oracle-Naive-A2A*. Since we create POIs that have the same coordinate values as all points on the point cloud, we substitute n to N in the oracle construction time of *RC-Oracle-Naive*. Thus, the oracle construction time of *RC-Oracle-Naive-A2A* is $O(N^2 \log N)$.

Secondly, we show the *oracle size* of *RC-Oracle-Naive-A2A*. Since we create POIs that have the same coordinate values as all points on the point cloud, we substitute n to N in the oracle size of *RC-Oracle-Naive*. Thus, the oracle construction time of *RC-Oracle-Naive-A2A* is $O(N^2)$.

Thirdly, we show the *shortest path query time* of *RC-Oracle-Naive-A2A*. The shortest path query time of *RC-Oracle-Naive-A2A* is the same as that of *RC-Oracle-Naive*. Thus, the shortest path query time of *RC-Oracle-Naive-A2A* is $O(1)$.

Fourthly, we show the *error bound* of *RC-Oracle-Naive-A2A*. The error bound of *RC-Oracle-Naive-A2A* is the same as that of *RC-Oracle-Naive*, so *RC-Oracle-Naive-A2A* returns the exact shortest path passing on the point cloud. □

THEOREM E.16. *The oracle construction time, oracle size and kNN query time of SU-Oracle-Adapt are $O(n'N^2 \log N)$, $O(n'N)$ and $O(n'N^2 \log N)$, respectively. SU-Oracle-Adapt returns the exact kNN result.*

PROOF. The proof of the oracle construction time, oracle size, *kNN* query time and error analysis of *SU-Oracle-Adapt* is in [40]. □

THEOREM E.17. *The oracle construction time, oracle size and kNN query time of SU-Oracle-Adapt-A2A are $O(N^2 \sqrt{N} \log N)$, $O(N \sqrt{N})$ and $O(N^2 \sqrt{N} \log N)$, respectively. SU-Oracle-Adapt returns the exact kNN result.*

PROOF. Since we select $\sqrt{N}$ points as POIs, we need to change n' to $\sqrt{N}$ in the oracle construction time, oracle size and *kNN* query time of *SU-Oracle-Adapt*, to obtain the theoretical analysis for *SU-Oracle-Adapt-A2A*. □

THEOREM E.18. *The oracle construction time, oracle size and kNN query time of RC-Oracle-NaiveProx are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$ and $O(n')$, respectively. The error rate of the kNN query result by using RC-Oracle-NaiveProx is $(1 + \epsilon)$.*

PROOF. The theoretical analysis proof of *RC-Oracle-NaiveProx* is the same as *RC-Oracle*. But, *RC-Oracle* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.19. *The oracle construction time, oracle size and kNN query time of RC-Oracle-A2P-SmCon-NaiveProx are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$ and $O(N \log N + n')$, respectively. The error rate of the kNN query result by using RC-Oracle-A2P-SmCon-NaiveProx is $(1 + \epsilon)$.*

PROOF. The theoretical analysis proof of *RC-Oracle-A2P-SmCon-NaiveProx* is the same as *RC-Oracle-A2P-SmCon*. But, *RC-Oracle-A2P-SmCon* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.20. *The oracle construction time, oracle size and kNN query time of RC-Oracle-A2P-SmQue-NaiveProx are $O(\frac{N \log N}{\epsilon} + n \log n)$, $O(\frac{n}{\epsilon})$ and $O(n')$, respectively. The error rate of the kNN query result by using RC-Oracle-A2P-SmQue-NaiveProx is $(1 + \epsilon)$.*

PROOF. The theoretical analysis proof of *RC-Oracle-A2P-SmQue-NaiveProx* is the same as *RC-Oracle-A2P-SmQue*. But, *RC-Oracle-A2P-SmQue* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.21. *The oracle construction time, oracle size and kNN query time of RC-Oracle-A2A-NaiveProx are $O(\frac{N \log N}{\epsilon})$, $O(\frac{N}{\epsilon})$ and $O(n')$, respectively. The error rate of the kNN query result by using RC-Oracle-A2A-NaiveProx is $(1 + \epsilon)$.*

PROOF. The theoretical analysis proof of *RC-Oracle-A2A-NaiveProx* is the same as *RC-Oracle-A2A*. But, *RC-Oracle-A2A* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.22. *The oracle construction time, oracle size and kNN query time of TI-Oracle-NaiveProx are $O(\frac{N \log N}{\epsilon} + Nn + n \log n)$, $O(\frac{N}{\epsilon})$ and $O(n')$, respectively. The error rate of the kNN query result by using TI-Oracle-NaiveProx is $(1 + \epsilon)$.*

PROOF. The theoretical analysis proof of *TI-Oracle-NaiveProx* is the same as *TI-Oracle*. But, *TI-Oracle* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.23. *The oracle construction time, oracle size and kNN query time of TI-Oracle-Rtree are* $O(\frac{N \log N}{\epsilon} + Nn + n \log n)$, $O(\frac{N}{\epsilon})$ *and* $O(n')$, *respectively. The error rate of the kNN query result by using TI-Oracle-Rtree is* $(1 + \epsilon)$.

PROOF. The oracle construction time, oracle size and error rate of the *kNN* query result of *TI-Oracle-Rtree* is the same as *TI-Oracle*. For the *kNN* query time, since *TI-Oracle-Rtree* first uses R-tree to find the location of a query point in $O(\log n')$ time, and use the similar method as *TI-Oracle* to find the final result in $O(n')$ time, so the *kNN* query time of *TI-Oracle-Rtree* is also $O(\log n' + n') = O(n')$. So, *TI-Oracle* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.24. *The oracle construction time, oracle size and kNN query time of TI-Oracle-TigLoo are* $O(\frac{N \log N}{\epsilon} + Nn + n \log n)$, $O(\frac{N}{\epsilon})$ *and* $O(n')$, *respectively. The error rate of the kNN query result by using TI-Oracle-TigLoo is* $(1 + \epsilon)$.

PROOF. The oracle size and error rate of the *kNN* query result of *TI-Oracle-TigLoo* is the same as *TI-Oracle*. For the oracle construction time, *TI-Oracle-TigLoo* involves the construction of tight and loose surface oracles, but the result is the same as *TI-Oracle*. *TI-Oracle* performs better in terms of oracle construction time in the experimental result. For the *kNN* query time, although *TI-Oracle-TigLoo* can gradually expand in *kNN* queries to slightly reduce *kNN* query time, its result is still the same as *TI-Oracle*. □

THEOREM E.25. *The oracle construction time, oracle size and kNN query time of TI-Oracle-A2A-NaiveProx are* $O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$, $O(\frac{N}{\epsilon})$ *and* $O(n')$, *respectively. The error rate of the kNN query result by using TI-Oracle-A2A-NaiveProx is* $(1 + \epsilon)$.

PROOF. The theoretical analysis proof of *TI-Oracle-A2A-NaiveProx* is the same as *TI-Oracle-A2A*. But, *TI-Oracle-A2A* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.26. *The oracle construction time, oracle size and kNN query time of TI-Oracle-A2A-Rtree are* $O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$, $O(\frac{N}{\epsilon})$ *and* $O(n')$, *respectively. The error rate of the kNN query result by using TI-Oracle-A2A-Rtree is* $(1 + \epsilon)$.

PROOF. The oracle construction time, oracle size and error rate of the *kNN* query result of *TI-Oracle-A2A-Rtree* is the same as *TI-Oracle*. For the *kNN* query time, since *TI-Oracle-A2A-Rtree* first uses R-tree to find the location of a query point in $O(\log n')$ time, and use the similar method as *TI-Oracle* to find the final result in $O(n')$ time, so the *kNN* query time of *TI-Oracle-A2A-Rtree* is also $O(\log n' + n') = O(n')$. So, *TI-Oracle-A2A* performs better in terms of *kNN* query time in the experimental result. □

THEOREM E.27. *The oracle construction time, oracle size and kNN query time of TI-Oracle-A2A-TigLoo are* $O(\frac{N \log N}{\epsilon} + N\sqrt{N} + \sqrt{N} \log \sqrt{N})$, $O(\frac{N}{\epsilon})$ *and* $O(n')$, *respectively. The error rate of the kNN query result by using TI-Oracle-A2A-TigLoo is* $(1 + \epsilon)$.

PROOF. The oracle size and error rate of the *kNN* query result of *TI-Oracle-A2A-TigLoo* is the same as *TI-Oracle-A2A*. For the oracle construction time, *TI-Oracle-A2A-TigLoo* involves the construction of tight and loose surface oracles, but the result is the same as *TI-Oracle-A2A*. *TI-Oracle-A2A* performs better in terms of oracle construction time in the experimental result. For the *kNN* query time, although *TI-Oracle-A2A-TigLoo* can gradually expand in *kNN* queries to slightly reduce *kNN* query time, its result is still the same as *TI-Oracle-A2A*. □