Data Sci. Eng. (2017) 2:71–93
DOI 10.1007/s41019-016-0030-0

Investigating TSP Heuristics for Location-Based Services

Weihuang Huang[1] · Jeffrey Xu Yu[1]

Received: 10 October 2016 / Accepted: 10 December 2016 / Published online: 4 February 2017

Abstract Travel planning is one of the important issues in the location-based services (*LBS*). Traveling salesman problem (*TSP*) is to find the optimal tour that traverses points exactly once in the minimum total distance. Given the hardness of *TSP* (NP-hard), *TSP* query for a given set of points, *Q*, is not widely studied for online *LBS*, and the nearest-neighbor heuristic is the only heuristic adapted to find *TSP*-like tours with additional constraints for *LBS*. The questions to ask are: Is the nearest-neighbor the best in terms of accuracy? Which heuristics among many should we use to process *TSP* queries online for *LBS*? In the literature, *TSPLIB* benchmarks are designed for special cases where the number of points used is large, and the existing synthetic datasets are based on uniform/normal distributions. Both do not reflect the real datasets used in real applications. Therefore, the best heuristics suggested by the *TSPLIB* and the existing benchmarks need to be reconsidered for *LBS* setting. In this work, we investigate 22 heuristics and show that the best heuristics in terms of accuracy for *LBS* are not the ones suggested by the existing work, and identify several heuristics by extensive performance studies over real datasets, *TSPLIB* benchmarks, the existing synthetic datasets and our new synthetic datasets. Among many issues, we also show that it is possible to get high-quality *TSP* by precomputing/indexing, even though it is hard to prove by theorem.

✉ Weihuang Huang
whhuang@se.cuhk.edu.hk

Jeffrey Xu Yu
yu@se.cuhk.edu.hk

[1] The Chinese University of Hong Kong, Hong Kong, China

Keywords TSP · Heuristics · Sketch

1 Introduction

Location-based services (*LBS*) attract great attention from both research and industry communities, and various queries have been studied. In Refs. [7, 26] discover useful information from trajectory data generated in daily life. In Refs. [17, 24] optimize query processing on location-based social networks. In Refs. [12, 18, 25] combine *LBS* with traditional keyword search. Travel planning has also been studied, and becomes an important issue in location-based services (*LBS*), which are to find tours among points of interest (*POI*), where *POI*s are with latitude and longitude in a two-dimensional space or in a road network. In Refs. [9, 26] study on how to find trajectories from an existing trajectory set. There are works that try to construct routes satisfying certain requirements. In [8] constructs the most popular routes between two given points. In [35] defines different queries as finding the earliest arrival, latest departure and shortest duration paths on the transportation networks. Some recent work study finding the shortest tour connecting two *POI*s [34, 38] and searching the optimal meeting point for a set of *POI*s, which are to minimize the sum of distances from these *POI*s to the meeting point [36, 37].

As an important issue in travel planning, traveling salesman problem (*TSP*) has been extensively studied, which finds a tour that traverses all the points exactly once with the minimum overall distance, for a given set of points, and is known as NP-hard problem. The hardness is mainly due to two reasons. First, given n points, there are $n!$ possible routes to traverse, in order to find the one with minimum overall distance. Second, the local optimum property does not hold. The state-of-the-art exact *TSP* solution, Concorde, is based on linear programming (http://www.math.uwaterloo.ca/tsp/concorde.html). By randomization, Arora in [1] finds $(1+\frac{1}{c})$-approximate answer in $O(n(\log n)^{O(c)})$ time, for every fixed $c > 1$, which is known as the best theoretical result, but is difficult to implement. In the literature, numerous works have been proposed to study *TSP* [19]. A large number of heuristics are proposed to find a high-quality tour within reasonable time. In Refs. [5, 31] and the most recent [21] summarize and test many representative heuristics and compare them in both effectiveness and efficiency.

Given the hardness of *TSP*, *TSP* query is not well studied in database community. Recently, there are *TSP*-like problems being studied for *LBS*, which are with constraints to reduce the search space [6, 23, 33], and find a tour by adding nearest neighbors one by one in a manner of expanding the partial result found. In other words, the work reported [6, 23, 33] only use one heuristic, namely the nearest neighbor, among many possible heuristics. The questions that arise are as follows. Is the nearest neighbor the best in terms of accuracy? What are the other methods and which one should we use to process *TSP* if there are many? This issue is important, since it opens ways for us to explore different ways to deal *TSP* in *LBS* for real large datasets with different properties.

There are several attempts to study different heuristics. First, [21] studies heuristics for *TSP* queries that travel more than 1000 points. However, in many real applications, the number of points can vary in a large range. For *LBS*, the number of points can be much smaller than that number. Second, the *TSPLIB* benchmark (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95) studies about 150 difficult cases, which is not sufficient to understand the heuristics in real datasets. Third, there are synthetic datasets [20, 21], but they do not reflect all real datasets. Figure 1 shows 4 datasets. Figure 1a shows a dataset with 3038 points in *TSPLIB*. Figure 1b shows a dataset containing 3000 points that follow normal distribution generated [21]. Figure 1c shows 3000 randomly sampled *POI*s in a real dataset in New York (NY). Figure 1d shows 3000 randomly sampled check-ins in Los Angeles (LA) from the location-based social network, *Gowalla* (https://en.wikipedia.org/wiki/Gowalla). Fourth, there are no performance studies to study all heuristics. In this work, we study 22 heuristics for *TSP* queries.

The main contributions of this work are summarized below. First, we study 22 *TSP* construction heuristics. The reason to study such heuristics is due to the efficiency requirement in *LBS*, since construction heuristics [21] are efficient to find *TSP* without any further refinement. Second, we propose new synthetic datasets to understand *TSP* in the real *LBS* setting. Third, we conduct extensive performance studies over the selected real datasets, *TSP* benchmarks, the existing synthetic datasets, and our new synthetic datasets. Fourth, we conclude that both the nearest-neighbor-based heuristics that are widely used in *LBS* and the best heuristics in *TSPLIB* for difficult setting are not the best to be used in *LBS*. We identify several that can achieve high accuracy efficiently. Among many issues, we also show that it is possible to get high-quality *TSP* by precomputing/indexing, even though it is hard to prove by theorem.

The rest of the paper is organized as follows. Section 2 discusses the preliminaries and gives the problem statement. We introduce all the 22 construction heuristics in Sect. 3. In Sect. 4, we discuss our new synthetic datasets generation in detail, and we report our finding over the 22 heuristics using real datasets, the selected 20 *TSPLIB* benchmarks, the existing synthetic datasets, and new synthetic datasets. We conclude this work in Sect. 5.

2 Preliminaries

Consider a set of points V in a two-dimensional space, where the distance between two points u and v in V is the Euclidean distance, denoted as $d(u, v)$.

$$d(u,v) = \sqrt{(u.x - v.x)^2 + (u.y - v.y)^2} \tag{1}$$

We denote the two x and y coordinates of a point u as $u.x$ and $u.y$.

An edge-weighted complete undirected graph $G = (V, E)$ can be constructed for the set of given points. Here, V is the set of nodes for the same set of points, and E is a set of edges for every pair of nodes in V where an edge weight for an edge (u, v) is the distance between u and v, $d(u, v)$.

Let Q be a subset of nodes of size $n = |Q|$ in V. A Hamilton path over Q is a simple path, $(v_1, v_2, v_3, \ldots, v_{n-1}, v_n)$, that visits every node exactly once, where (v_i, v_{i+1}) is an edge in the graph G. A Hamilton circuit over Q is a simple cycle over all nodes in G. Both Hamilton path and Hamilton circuit can be regarded as a permutation of nodes (or points) in Q. Here, a permutation π over Q is a one-to-one mapping. In other words, a node can only appear at a specific position in a permutation. Below, we use π_i to indicate a specific node v in Q at the ith position. We indicate a permutation over Q as $T = (\pi_1, \pi_2, \ldots, \pi_n)$. Given a permutation T over Q, the

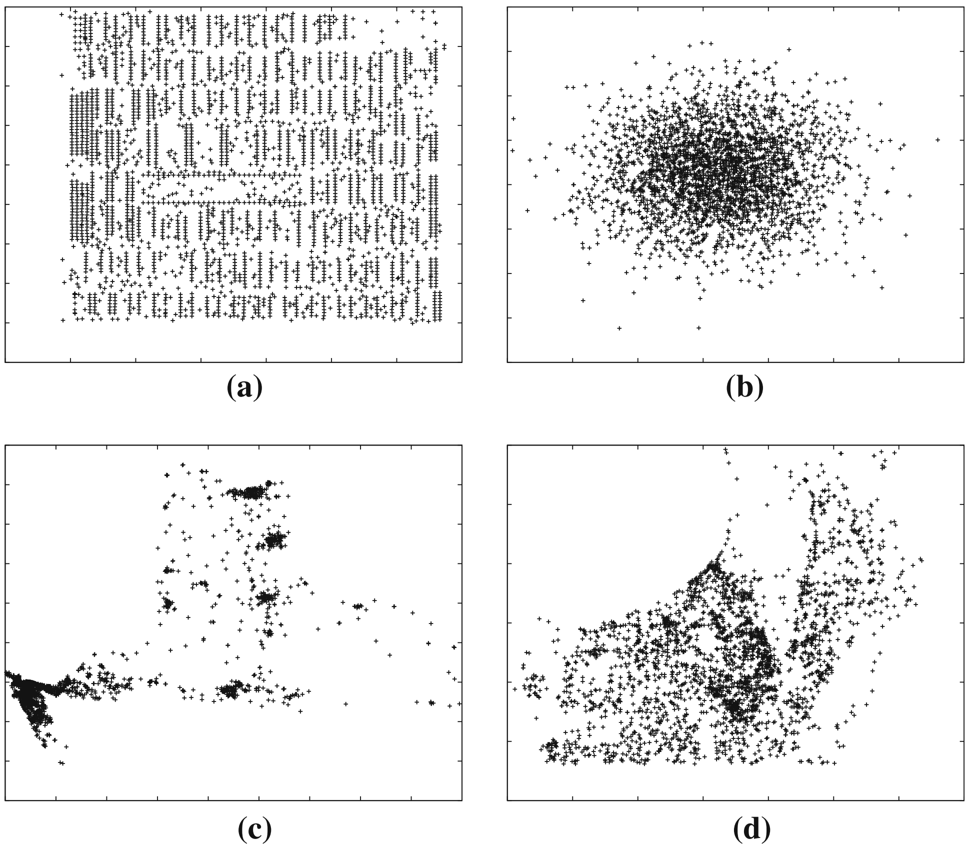


Fig. 1 Different datasets. **a** *TSPLIB* (pcb3038). **b** Normal. **c** New York (*POI*). **d** Los Angeles (check-ins)

distance of a Hamilton path by T is defined as $d(T) = \sum_{1 \le i < n} d(\pi_i, \pi_{i+1})$, and the distance of a Hamilton circuit by T is defined as $d(T) = \sum_{1 \le i < n} d(\pi_i, \pi_{i+1}) + d(\pi_n, \pi_1)$. Let $\mathcal{T}$ be the set of all possible paths (circuits) for Q. The size of $\mathcal{T}$ is $|\mathcal{T}| = n!$ for Hamilton paths, and the size is $|\mathcal{T}| = \frac{(n-1)!}{2}$ for Hamilton circuits.

In this paper, we focus on Hamilton circuit, and may use "circuit," "tour" and "route" interchangeably, since they are all used in reported studies. Among all possible permutations in $\mathcal{T}$, the optimal Hamilton circuit over Q is the shortest Hamilton circuit, denoted as T^*, such that $d(T^*) = \min_{T \in \mathcal{T}} d(T)$. The problem of finding the optimal Hamilton circuit is known as traveling salesman problem (*TSP*), which is known to be NP-hard. The error-ratio for an approximate T is defined below.

$$\mathsf{eratio}(T) = \frac{d(T) - d(T^*)}{d(T^*)} \tag{2}$$

It is worth mentioning that the *TSP* problem we study in this paper is the symmetric and metric *TSP*. Here, by symmetric it implies $d(u,v) = d(v,u)$, and by metric it implies $\forall u, v, w \in V, d(u,w) + d(w,v) \ge d(u,v)$.

The Problem In this paper, we study *TSP* query to find the shortest Hamilton circuit T^* for a given *TSP* query, Q, which is a set of points, and explore the similarities and differences among 22 heuristics proposed for *TSP* using real datasets and new synthetic datasets in addition to the existing benchmarks and uniform/normal synthetic datasets.

3 The Heuristics

The *TSP* heuristics have been studied. In this work, we focus on tour construction heuristics [21]. By tour construction, it computes a tour (or a circuit) following some rules, and takes the resulting tour by the rules as the final result without further refinement. In [21], construction heuristics are divided into 3 categories: heuristics designed for speed, tour construction by pure augmentation and more complex tour construction. In this work, we cover more heuristics, and divide 22 construction heuristics into 3 new categories, namely (1) space-partitioning-based heuristics, (2) node-based heuristics and (3) edge-based heuristics. Table 2 lists all the 22 heuristics studied, where some are with guarantee of the approximate ratio. Below, we discuss 2 space-partitioning-based heuristics in Sect. 3.1, 4 edge-based heuristics in Sect. 3.3 and 16 node-based heuristics in Sect. 3.2.

3.1 Space-Partitioning-Based Heuristics

The space-partitioning-based methods compute *TSP* for a given set of points Q in three main steps: (1) partition nodes in Q into smaller subsets based on their pairwise Euclidean distances, (2) connect the nodes in the same subset into a Hamilton path and (3) determine the Hamilton circuit for Q by linking all Hamilton paths obtained for all subsets. We discuss two heuristics, namely Strip and Hilbert.

First, Strip computes the minimum bounding rectangle (*MBR*) in two-dimensional space that encloses all the query nodes Q of size $n = |Q|$, and partitions the *MBR* into $\sqrt{\frac{n}{3}}$ equal-width vertical strips. For each vertical strip, Strip sequences all the inside nodes according to y-coordinate by alternately top to bottom and bottom to top. The final circuit is determined by connecting all the sequences computed for all strips. Strip only involves sorting by x-coordinate and y-coordinate.

Algorithm 1: Greedy (Q)

```
Input: $Q$: a TSP query of a set of points
Output: $T$: the TSP for $Q$
begin
    $T \leftarrow \emptyset$; $\mathcal{H} \leftarrow \emptyset$;
    for every $u, v \in Q$ s.t. $u \neq v$ do
        insert $(u, v)$ with $d(u, v)$ into the min-heap $\mathcal{H}$;
    while $|T| < |Q|$ do
        let $(u, v)$ be the edge with min cost deleted from $\mathcal{H}$;
        if $u$ and $v$ are not in the same subtree then
            if $deg(u) \leq 2$ and $deg(v) \leq 2$ then
                insert $(u, v)$ into $T$;
    connect the two nodes in $T$ with degree 1;
    return $T$;
```

Second, the space filling curve is a widely used technique to map multidimensional data into one-dimensional data. The main idea behind the space filling curve is that it keeps the locality information of the original data after mapping such that two near nodes may still be close to each other after mapping. Therefore, visiting query nodes in Q in the order of their appearance along the space filling curve reduces the total length [29]. By space filling curve, it can recursively partition the whole plane into small units, where a unit is labeled with a string of binary digits based on where it is in the hierarchy. For instance, if the entire plane is divided into two units, one is labeled with "0," and the other is labeled with "1." Then it can get 4 units by further dividing each of the unit into another 2 smaller units in the similar manner. Such partitioning stops until there is at most one node at each unit. In this paper, we focus on the Hilbert curve (or Hilbert space filling curve) since it has better locality-preserving behavior.

Both Strip and Hilbert are easy to implement and are efficient. However, they only utilize pairwise distances to reduce the total route length, and neglect the overall distribution of all query nodes, which sacrifices accuracy. To improve the accuracy, in every step, there are several strategies that can be adopted to make the final circuit as short as possible. The state-of-the-art approximate algorithm [1] is based on partition. However, it utilizes dynamic programming to connect inner and inter nodes, which is beyond the scope of this work on simple construction heuristics.

3.2 Edge-Based Heuristics

The edge-based heuristics are based on the minimum spanning tree (*MST*). We discuss the greedy (Greedy) which is known as multiple fragment heuristic, double-*MST* (DMST), the Christofides algorithm (Chris) and the savings algorithm (SV).

First, Greedy is designed based on the Kruskal's algorithm [22] to find the minimum spanning tree for a undirected graph. As shown in Algorithm 1, it inserts every pair of u and v in Q as an edge into a min-heap $\mathcal{H}$ with $d(u, v)$. In the while loop, it picks up the edge (u, v) from $\mathcal{H}$, which is with the minimum distance, and checks if such an edge (u, v) can connect two different subtrees as a larger subtree without a cycle. In addition, it further checks if the tree formed can end up a *TSP* by ensuring that the degree of u/v ($\deg(u)$/$\deg(v)$) is less than or equal to 2. At the end, it connects the two nodes in T with degree 1 to form a circuit.

Second, DMST is an algorithm which traverses the minimum spanning tree T constructed for the edge-weighted undirected graph representation for Q. To obtain the circuit, it keeps the traversal order and skips the nodes which are traversed before.

Third, the Christofides algorithm (Chris) finds the circuit as follows. (1) Given the edge-weighted undirected graph representation $G = (V, E)$ for Q, it finds the minimum spanning tree $T = (V_T, E_T)$. (2) It identifies a subset of V_T, denoted as V_O, which includes all those nodes in V_T that have an odd degree. (3) It then constructs an induced subgraph $G_O(V_O, E_O)$ from G. (4) It finds a minimum weighted perfect matching M from G_O, where a perfect matching M is a set of edges that do not have any common nodes. (5) It constructs a multigraph $G_H = T \cup M$. (6) It then finds an Eulerian circuit in G_H, because every node in G_H has an even degree. (7) Finally, it obtains the Hamilton circuit by removing the repeated nodes from the Eulerian circuit.

Fourth, the savings algorithm (SV) takes a different approach, and does not build a circuit using a minimum spanning tree. SV starts from a randomly selected node as the central node v_c and then builds a pseudo tour, T_P, from the central node v_c to all other nodes in Q. In order to make the tour short, SV looks for shortcuts in the pseudo tour T_P constructed. In every iteration, SV selects a pair of nodes u

and v that connect with v_c in T_P based on Eq. (3), deletes the edge $(u, v_c), (v, v_c)$, and inserts a new edge (u, v) as shortcut. In order to find the shortcut with maximum benefit, it defines a new cost function as:

$$(u, v) = \underset{u,v \in T_P \wedge u \neq v \neq v_c}{\operatorname{argmin}} \{c'(u, v)\} \tag{3}$$

where the cost function c' is given in Eq. (4).

$$c'(u, v) = d(u, v) - d(u, v_c) - d(v, v_c) \tag{4}$$

All the edge-based heuristics aim at finding the edge with the smallest distance directly into the circuit. DMST and Chris have a better approximate ratio than most heuristics studied in this work because a *TSP* circuit becomes a tree if any edge is removed from it, whose total length should be smaller than that of *MST*.

3.3 Node-Based Heuristics

The node-based heuristics construct a circuit by expanding the nodes in Q one by one until all of them are visited. There are three main issues in the heuristics, which are (a) how to initialize an initial node(s) and (b) in every iteration, how to select the next node to expand and where it is for the next node to be inserted. Among the node-based heuristics, we discuss the nearest-neighbor heuristics, the insertion heuristics, the convex hull-based insertion heuristics, the addition heuristics, and the augmented addition heuristics. Algorithm 2 shows the framework of node-based heuristics.

```
Algorithm 2: TSP-N(Q)
   Input: Q: a TSP query of a set of points
   Output: T: the TSP for Q
1  begin
2      T ← init(Q);
3      while T does not contain all nodes in Q do
4          v ← select(Q, T);
5          insert(v, T);
6      return T;
```

The Nearest-Neighbor Heuristics [3] The nearest-neighbor heuristics do not spend time on finding an initial *TSP* by init(Q), and randomly picks one node from Q. In other words, T by init(Q) contains only one node randomly selected. Then, in every iteration in the while loop, it picks up a point from the nodes that have not been selected before, namely from $Q \setminus T$, and inserts it into the end of the current partial path computed in the previous iteration. Assume $T_i = (\pi_1, \pi_2, \ldots, \pi_i)$ is the partial path computed at the ith iteration. There are two ends in T_i, namely π_1 and π_i. Consider the (i+1)th iteration to expand the path by adding one more node. The nearest-neighbor heuristic (NN) selects the nearest-neighbor node to the node at the position π_i from $Q \setminus T_i$, and inserts the node selected at π_{i+1}. On the other hand, the double-ended nearest-neighbor heuristic (DENN) considers the nearest-neighbor node to either of the two end points: π_1 and π_i. Assume the node selected is near to the node at π_1, DENN will insert the newly selected node at π_1 in T_{i+1} and place the node at the jth position (π_j) in T_i at (j+1)th position π_{j+1} in T_{i+1}. Otherwise, if the node selected is near to the node π_i, DENN behaves like NN. Both NN and DENN expand a path. After the while loop, both obtain the *TSP* by adding one edge from the last node to the first node in T. We omit such post-processing from Algorithm 2. In brief, comparing with NN, DENN considers both ends of the current partial path when expanding and selects the one with shorter length. Consequently, DENN consumes longer time than NN to improve the accuracy.

The Insertion Heuristics [31] Like the nearest-neighbor heuristics, the insertion heuristics randomly pick one node from Q by init(Q). Unlike the nearest-neighbor heuristics which expand the current partial path in every iteration, the insertion heuristics enlarge the current partial circuit in every iteration. Let T_i be the partial circuit over nodes of size i such that $\mathsf{T}_i = (\pi_1, \pi_2, \ldots, \pi_i, \pi_1)$. In the ($i$+1)th iteration, the insertion heuristics attempt to add one node into the current circuit by minimizing the increment of the total distance of the circuit. There are two things. One is how to select a node, w, from $Q \setminus \mathsf{T}_i$. The other is how to insert w into T_i to obtain T_{i+1}. We first discuss how to insert a new node into T_i, assuming the node to be inserted next is selected from $Q \setminus \mathsf{T}_i$. We will discuss the node selection next (Table 1).

Consider an insertion of a node w ($\notin \mathsf{T}_i$) between u and v in T_i. Here, for simplicity, we say to insert a node w into an edge (u, v) in T_i, where an edge (u, v) implies that v is next to u in the permutation. In the new circuit to be, T_{i+1}, the edge (u, v) in T_i will be replaced by two edges (u, w) and (w, v). Among all edges in T_i, the edge, (u, v), selected for a given node w is to minimize the incremental cost by Eq. (5).

$$(u, v) = \underset{(u,v) \in \mathsf{T}_i}{\operatorname{argmin}} \{c(u, v, w)\} \tag{5}$$

where $c(u, v, w)$ is a cost function to measure the incremental cost of inserting a node between two nodes (an edge) as given in Eq. (6).

$$c(u, v, w) = d(w, u) + d(w, v) - d(u, v) \tag{6}$$

Next consider how to select the next node. There are 4 ways to select the next node w to be inserted into T_i, namely the random insertion (RI), the nearest insertion (NI), the cheapest insertion (CI) and the furthest insertion (FI) [28]. Here, RI randomly picks one as the next node

Table 1 Expansion order of the node-based heuristics

Node-Based	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
NN	**v_1**	v_{20}	v_2	v_3	v_4	v_{18}	v_{17}	v_{16}	v_{15}	v_{14}	v_{13}	v_{12}	v_{11}	v_{10}	v_9	v_8	v_7	v_6	v_5	v_{19}
DENN	**v_1**	v_{20}	v_2	v_3	v_4	v_{18}	v_{17}	v_{16}	v_{15}	v_{14}	v_{13}	v_{12}	v_{11}	v_{10}	v_9	v_8	v_{19}	v_7	v_6	v_5
NI	**v_1**	v_{20}	v_2	v_3	v_4	v_{18}	v_7	v_6	v_{17}	v_{16}	v_{15}	v_{14}	v_{13}	v_{12}	v_{11}	v_{10}	v_9	v_8	v_{19}	v_5
CI	**v_1**	v_{20}	v_2	v_3	v_{18}	v_4	v_7	v_6	v_{17}	v_{16}	v_{15}	v_{13}	v_{14}	v_{12}	v_{11}	v_{10}	v_9	v_8	v_5	v_{19}
RI	**v_3**	v_6	v_{17}	v_8	v_2	v_9	v_{12}	v_{19}	v_{16}	v_{11}	v_{20}	v_{15}	v_7	v_5	v_{10}	v_{18}	v_{14}	v_4	v_1	v_{13}
FI	**v_1**	v_8	v_5	v_{19}	v_7	v_4	v_{17}	v_{18}	v_{10}	v_3	v_{20}	v_9	v_{16}	v_{15}	v_{12}	v_6	v_{11}	v_2	v_{14}	v_{13}
CHNI	**v_8**	**v_{10}**	**v_{19}**	**v_5**	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}	v_{16}	v_9	v_{17}	v_7	v_6	v_4	v_3	v_2	v_1	v_{20}	v_{18}
CHCI	**v_8**	**v_{10}**	**v_{19}**	**v_5**	v_{20}	v_{12}	v_9	v_{13}	v_{14}	v_{11}	v_6	v_7	v_1	v_2	v_3	v_4	v_{18}	v_{16}	v_{17}	v_{15}
CHRI	**v_8**	**v_{10}**	**v_{19}**	**v_5**	v_1	v_2	v_3	v_4	v_6	v_7	v_9	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}	v_{16}	v_{17}	v_{18}	v_{20}
CHFI	**v_8**	**v_{10}**	**v_{19}**	**v_5**	v_2	v_7	v_{20}	v_{17}	v_{18}	v_4	v_1	v_9	v_{16}	v_{15}	v_{12}	v_6	v_{11}	v_3	v_{14}	v_{13}
NA	**v_1**	v_{20}	v_2	v_3	v_4	v_{18}	v_7	v_6	v_{17}	v_{16}	v_{15}	v_{14}	v_{13}	v_{12}	v_{11}	v_{10}	v_9	v_8	v_{19}	v_5
RA	**v_3**	v_6	v_{17}	v_8	v_2	v_9	v_{12}	v_{19}	v_{16}	v_{11}	v_{20}	v_{15}	v_7	v_5	v_{10}	v_{18}	v_{14}	v_4	v_1	v_{13}
FA	**v_1**	v_8	v_5	v_{19}	v_7	v_4	v_{17}	v_{18}	v_{10}	v_3	v_{20}	v_9	v_{16}	v_{15}	v_{12}	v_6	v_{11}	v_2	v_{14}	v_{13}
NA+	**v_1**	v_{20}	v_2	v_3	v_4	v_{18}	v_7	v_6	v_{17}	v_{16}	v_{15}	v_{14}	v_{13}	v_{12}	v_{11}	v_{10}	v_9	v_8	v_{19}	v_5
RA+	**v_3**	v_6	v_{17}	v_8	v_2	v_9	v_{12}	v_{19}	v_{16}	v_{11}	v_{20}	v_{15}	v_7	v_5	v_{10}	v_{18}	v_{14}	v_4	v_1	v_{13}
FA+	**v_1**	v_8	v_5	v_{19}	v_7	v_4	v_{17}	v_{18}	v_{10}	v_3	v_{20}	v_9	v_{16}	v_{15}	v_{12}	v_6	v_{11}	v_2	v_{14}	v_{13}

Bold values in each row represent the initial node(s) for the corresponding heuristic

w. NI selects the next node w from $Q \setminus \mathsf{T}_i$ that has the smallest distance to a node in T_i (Eq. 7)

$$w = \underset{w \notin \mathsf{T}_i}{\operatorname{argmin}} \{d(w, v), \forall v \in \mathsf{T}_i\} \tag{7}$$

Note that NI inserts the nearest node to the current circuit, instead of the end node as done by the nearest-neighbor heuristics (NN or DENN). CI selects the next node based on the cost function.

$$w = \underset{w \notin \mathsf{T}_i}{\operatorname{argmin}} \{c(u, v, w), \forall (u, v) \in \mathsf{T}_i\} \tag{8}$$

We discuss the similarity between NI and CI in certain cases. Suppose the next node to be inserted is w. Assume v' is the node in T_i under which w is selected by Eq. (7), and assume (u, v) is the edge under which the next node w is selected by CI based on Eq. (8) As proved in [5], (u, v) should satisfy at least one of the 3 conditions: (1) $u = v'$ or $v = v'$, which means one endpoint of this edge in T_i is the nearest neighbor to w; (2) given a circle C centered at w with the radius $1.5 \times d(w, v')$, then either $u \in C$ or $v \in C$, which means one endpoint of this edge is inside a circle centered at w; and (3) for every pair of (v_i, v_j) in T_i, given a circle C_i centered at v_i with radius $1.5 \times d(v_i, v_j)$, then either $w \in C_i$ or $w \in C_j$, which means w is inside the corresponding circle of u or v. Here, if the condition 1 is satisfied, CI selects the same node as NI does.

Unlike the 3 heuristics to select the next node discussed above, FI picks up the next node which is far away from T_i with a higher priority. It chooses the next node following Eq. (9).

$$w = \underset{w \notin \mathsf{T}_i}{\operatorname{argmax}} \{d(w, \mathsf{T}_i)\} \tag{9}$$

where $d(w, \mathsf{T}_i)$ is the smallest distance from a node $w \notin \mathsf{T}_i$ to a node in the current T_i such that $d(w, \mathsf{T}_i) = \min_{v \in T_p} d(v, w)$.

Convex Hull-Based Insertion Heuristics The insertion heuristics pick the start node randomly, which may affect the quality of the result. As pointed out in [27], it is an effective approach to construct an initial circuit for the points Q and then insert the remaining nodes into the initial circuit. It is proved that for the optimal circuit, the nodes lying on the boundary of the convex hull will be visited in their cyclic order [14]. This suggests that the convex hull can serve as a sketch to guide future insertions. The convex hull-based insertions are proposed to find the convex hull of all nodes in Q first using $T \leftarrow init(Q)$ and then compute the circuit using one of the insertion heuristics for the remaining nodes in $Q \setminus T$. Here, we investigate 4 heuristics: convex hull cheapest insertion (CHCI), convex hull nearest insertion (CHNI), convex hull random insertion (CHRI) and convex hull furthest insertion (CHFI). Note that [21] only shows the testing results for CHCI.

The Addition Heuristics [5] The insertion heuristics determine an edge for a node to be inserted among all the edges in T_i. To further reduce the computational cost, for a node w selected from $Q \setminus \mathsf{T}_i$, the addition heuristics pick a node v at π_j on the current circuit T_i and only consider the insertion of w either between two nodes at (π_{j-1}, π_j) or between two nodes at (π_j, π_{j+1}). The edge can be selected with the smallest cost by Eq. (6). We study 3 ways to select the next node w, namely random addition (RA), nearest addition (NA) and furthest addition (FA). We do not study cheapest addition, since the insert position by the cheapest heuristic is decided once the next node is determined. Here, RA selects the next node w from $Q \setminus \mathsf{T}_i$ randomly, and identifies $v \in \mathsf{T}_i$ based on Eq. (10).

$$v = \underset{v \in \mathsf{T}_i}{\operatorname{argmin}}\{d(w, v)\} \tag{10}$$

NA selects a pair of nodes, w and v such that $w \notin \mathsf{T}_i$ and $v \in \mathsf{T}_i$ in a similar way like the nearest insertion, based on Eq. (11).

$$(v, w) = \underset{v \in \mathsf{T}_i, w \notin \mathsf{T}_i}{\operatorname{argmin}}\{d(v, w)\} \tag{11}$$

Here, w is the next node to be selected, and v is the node at π_j position in the current T_i. FA selects the next node w based on Eq. (9), and identifies the insertion position π_j by Eq. (10), and inserts w either between π_{j-1} and π_j or between π_j and π_{j+1} following the minimal incremental cost (Eq. 6).

Augmented Addition Heuristics Consider the insertion heuristics and the addition heuristics. On the one hand, the insertion heuristics explore all edges in T_i to insert a node w between a pair of nodes, u and v, as an edge (u, v). On the other hand, the addition heuristics only consider 2 edges incident to the node v at π_j, when inserting w. Different from insertion/addition heuristics, the augmented additions attempt to explore more than 2 edges up to some extent. Here, like the addition heuristics, they select the node w ($\notin \mathsf{T}_i$) and the insertion position v at π_j in T_i using either RA, NA, and FA. Then, the augmented addition heuristics select an edge with the minimum cost from all edges in a circle centered at w with the radius of $\alpha \cdot r$, where $\alpha \geq 1$ and $r = d(w, v)$ in the two-dimensional space. Here, an edge (u, v) is in the circle if u or v appears in the circle. We denote such augmented addition heuristics as random augmented addition (RA+), nearest augmented addition (NA+) and furthest augmented addition (FA+).

3.4 An Example

Figure 2 shows the optimal *TSP*, *T*, for a *TSP* query Q with 20 points, v_i, for $1 \leq i \leq 20$ sampled from NY. The positions of the 20 points are also given in Fig. 2, which forms 3 clusters, one with 10 points, $\{v_i\}$, for $8 \leq i \leq 17$, one with 2 points, $\{v_7, v_8\}$ and one with 5 points, $\{v_1, v_2, v_3, v_4, v_{20}\}$. There are some points which are at a distance from any of the clusters, such as v_5, v_{18} and v_{19}. Table 2 shows the 22 heuristics in the three categories: space-partitioning-based, edge-based, and node-based. For each heuristic, the approximate ratio is given, if any, which is for a *TSP*, Q,

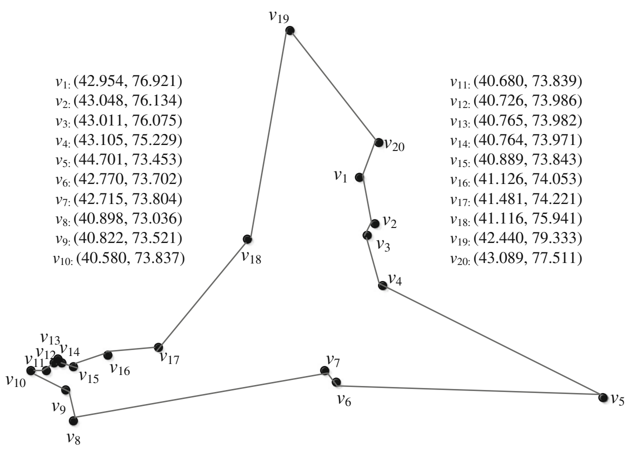

v_{1}: (42.954, 76.921)
v_{2}: (43.048, 76.134)
v_{3}: (43.011, 76.075)
v_{4}: (43.105, 75.229)
v_{5}: (44.701, 73.453)
v_{6}: (42.770, 73.702)
v_{7}: (42.715, 73.804)
v_{8}: (40.898, 73.036)
v_{9}: (40.822, 73.521)
v_{10}: (40.580, 73.837)
v_{11}: (40.680, 73.839)
v_{12}: (40.726, 73.986)
v_{13}: (40.765, 73.982)
v_{14}: (40.764, 73.971)
v_{15}: (40.889, 73.843)
v_{16}: (41.126, 74.053)
v_{17}: (41.481, 74.221)
v_{18}: (41.116, 75.941)
v_{19}: (42.440, 79.333)
v_{20}: (43.089, 77.511)
v_{19}
v_{20}
v_1
v_2
v_3
v_4
v_{18}
v_{17}
v_{16}
v_{15}
v_{14}
v_{13}
v_{12}
v_{11}
v_{10}
v_9
v_8
v_7
v_6
v_5

Fig. 2 A *TSP* Example

with n points. Also, in Table 2, the 5th column shows the eratio, for the *TSP* of 20 points shown in Fig. 2. Here, as default, we select v_1 as the first node to start except for the 4 convex hull-based insertion heuristics, which identify a convex hull with nodes, v_8, v_{10}, v_{19}, and v_4, to start. In particular, we also show how the node-based heuristics select the next node in every iteration in Table 1. In terms of the accuracy, among the 22 heuristics, there are 3 heuristics that get the optimal *TSP*.

4 Performance Studies

We study all 22 heuristics covering a large range of datasets: 4 real datasets, 20 datasets from *TSPLIB* benchmark [30], 2 existing synthetic datasets [21] and new synthetic datasets.

Datasets: The 4 real datasets used to test are shown in Table 3. Here, NY (New York) and BJ (Beijing) are real *POI*s of the two cities. LA (Los Angeles) and HK (Hong Kong) are real check-in data we crawled from the location-based social network in the two cities from Twitter (https://twitter.com/) and Gowalla, respectively.

The 20 datasets selected from the *TSPLIB* benchmark [30] cover 3 major types of *TSPLIB* data: ATT, EUD_2D, and GEO, and are summarized in Table 4, where the 1st and 4th columns are the short names, the 2nd and 5th columns are the names used in the benchmark, and the 3rd and 6th columns are the size of points used in the dataset. There are 10 datasets where $n = |Q|$ is selected between 100 and 1000, denoted as TB_H, and there are 10 datasets where $n = |Q|$ is selected between 1000 and 10000, denoted as TB_T.

We discuss our new synthetic datasets generation in this work. Note that the 2 existing synthetic datasets by uniform distribution and normal distribution [21] are not the

Table 2 Twenty-two heuristics (default start node: v_1)

Category		Heuristics	App. ratio	eratio for Fig. 2
Space-partitioning-based		Hilbert [2]	$\Omega(\sqrt{n})$	1.639
		Strip [29]	$O(\sqrt{n})$	0.129
Edge-based		Greedy [21]	$O(\sqrt{n})$	0.132
		DMST [21]	2	0.116
		Chris [11]	1.5	0.145
		SV [21]	–	0.028
Node-based	Nearest neighbor	NN [31]	$O(\lg n)$	0.158
		DENN [5]	–	0.096
	Insertion	NI [31]	2	0.073
		CI [31]	2	0.049
		RI [31]	$O(\lg n)$	0.011
		FI [28]	1.5	0
	Convex hull-based insertion	CHRI [21]	–	0.004
		CHCI [21]	–	0.001
		CHNI [21]	–	0.006
		CHFI [21]	–	0
	Addition	NA [21]	2	0.085
		RA [21]	–	0.160
		FA [21]	–	0.243
	Augmented addition	NA+ [5]	–	0.073
		RA+ [5]	–	0.083
		FA+ [5]	$O(\sqrt{n})$	0

Table 3 Four real datasets

Dataset	Size	Type	Dataset	Size	Type
NY	653,008	*POI*	LA	411,596	check-in
BJ	115,719	*POI*	HK	20,103	check-in

Table 4 Twenty Selected *TSPLIB* Benchmarks

Abbrv.	Name	n	Abbrv.	Name	n
h1	eil101.tsp	101	t1	u1060.tsp	1060
h2	u159.tsp	159	t2	pcb1173.tsp	1173
h3	rat195.tsp	195	t3	rl1304.tsp	1304
h4	ts225.tsp	225	t4	rl1889.tsp	1889
h5	pr299.tsp	299	t5	u2152.tsp	2152
h6	pcb442.tsp	442	t6	pcb3038.tsp	3038
h7	att532.tsp	532	t7	fnl4461.tsp	4461
h8	u574.tsp	574	t8	rl5915.tsp	5915
h9	gr666.tsp	666	t9	rl5934.tsp	5934
h10	rat783.tsp	783	t10	pla7397.tsp	7397

Table 5 Parameters of new synthetic datasets

Parameter	Values
The size of points (N)	**100,000**
The number of clusters (K)	1, 4, 16, **64**, 256, 1,024, 4,096
Inter-cluster distance (l)	1, 5, **10**, 50, 100
Cluster distribution (α)	0.7, 0.8, 0.9, **1**, 2, 3, 4
TSP Query size (n)	20, 40, 60, 80, **100**, 200, 400, 600, 800

Bold represent default values for the parameters during the experiment

Table 6 Average inter-cluster distance (normalized)

l		K		α	
1	1	0.8	9.84	4	9.25
5	4.97	0.9	9.45	16	9.48
10	10.46	1	10.03	64	10.07
50	52.88	2	9.91	256	10.23
100	107.17	3	10.61	1024	10.4

best for *LBS*, since there is a gap between uniform/normal and the real datasets. In this work, we simulate that people visit n *POI*s in a real dataset of N *POI*s, where a small number of hot *POI*s are visited by many people. In brief, we synthesize large datasets of size N followed by randomly selecting $n = |Q|$ from the large synthetic datasets. A synthetic dataset is generated with $N = 100,000$ points using 3 parameters, namely the number of clusters (K), the inter-cluster distance (l) and the distribution of points in clusters (α). First, the total number of points $N = 100,000$ is used because it is similar to the sizes of the real datasets we use (Table 3). Second, we randomly generate K points, as centers of the clusters, following uniform distribution in a square with size $l \times l$. Let C_i and k_i be the ith cluster and the center of C_i, for $1 \leq i \leq K$. Third, we randomly generate $N_i = \frac{100,000}{K}$ points for each of the K clusters, C_i. The points in a cluster follow Gaussian distribution, which is widely used to model spatial data/events [13, 32] and user mobility [10]. Let σ_i be the variance for C_i with N_i points centered at k_i. The covariance matrix of the Gaussian distribution will be in the form of $[[\sigma_i^2, 0], [0, \sigma_i^2]]$. All σ_i for all clusters C_i $(1 \leq i \leq K)$ follow Pareto distribution (α). The Pareto distribution is used because it is proved in [15] that human mobility patterns follow a power law distribution, which means a small number of places are visited by most people. The corresponding Gaussian distribution will be compact for a small σ, i.e., the small region has high visiting frequency. Table 5 shows the parameters with the default values.

With our synthetic datasets, we can study different settings including uniform and normal. For the normal distribution, it is by setting $K = 1$. where there is only one cluster. For the uniform distribution, it is by setting K as a large value, e.g., $K = N$. We discuss l and α. Recall that l is a parameter to decide the size of plane. When l is large, the average inter-cluster distance will be large, and the overlap between clusters will be small, as shown in Table 6. We show the average inter-cluster distances by varying three parameters. It is nearly in proportionate to l, and the varying of K and α will have no influence on it. For Pareto distribution, the parameter α decides the skewness. As shown Fig. 3, the larger α is, the more skew the distribution will be.

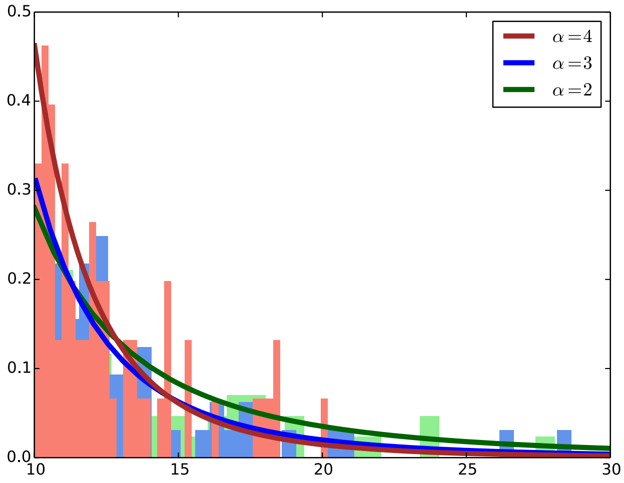
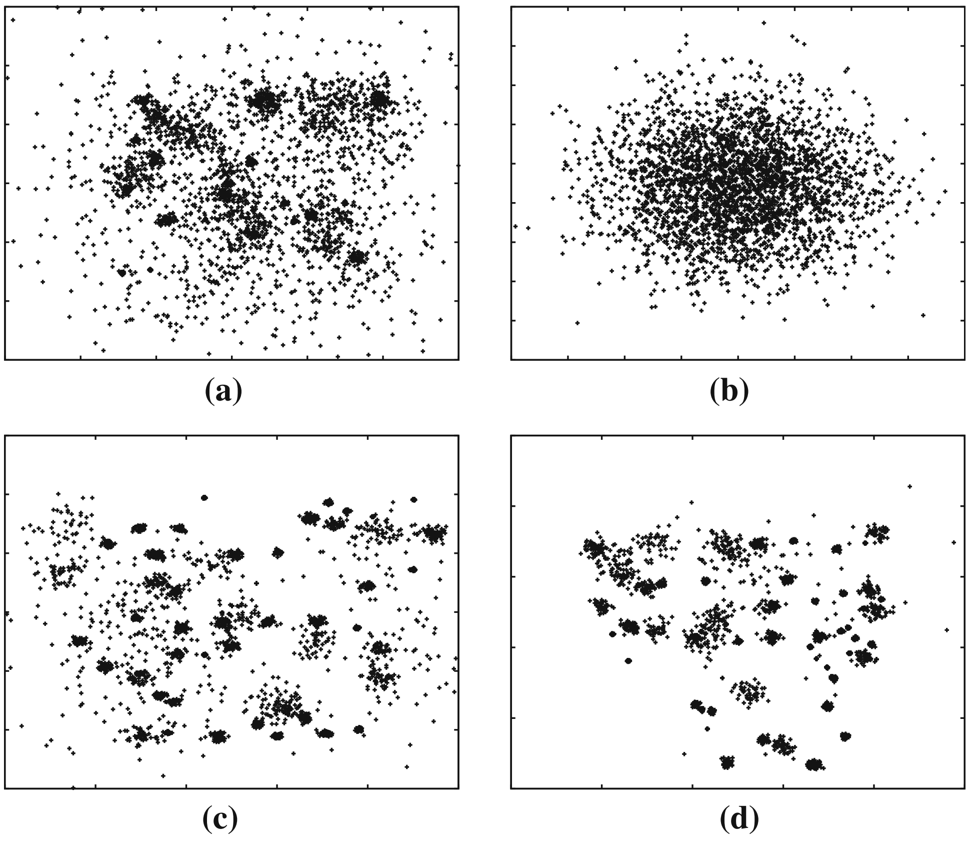


Fig. 3 Pareto distribution

Fig. 4 Data distribution with different parameters.
a $K = 64, l = 10, \alpha = 1$.
b $K = 1, l = 10, \alpha = 1$.
c $K = 64, l = 50, \alpha = 1$.
d $K = 64, l = 10, \alpha = 4$



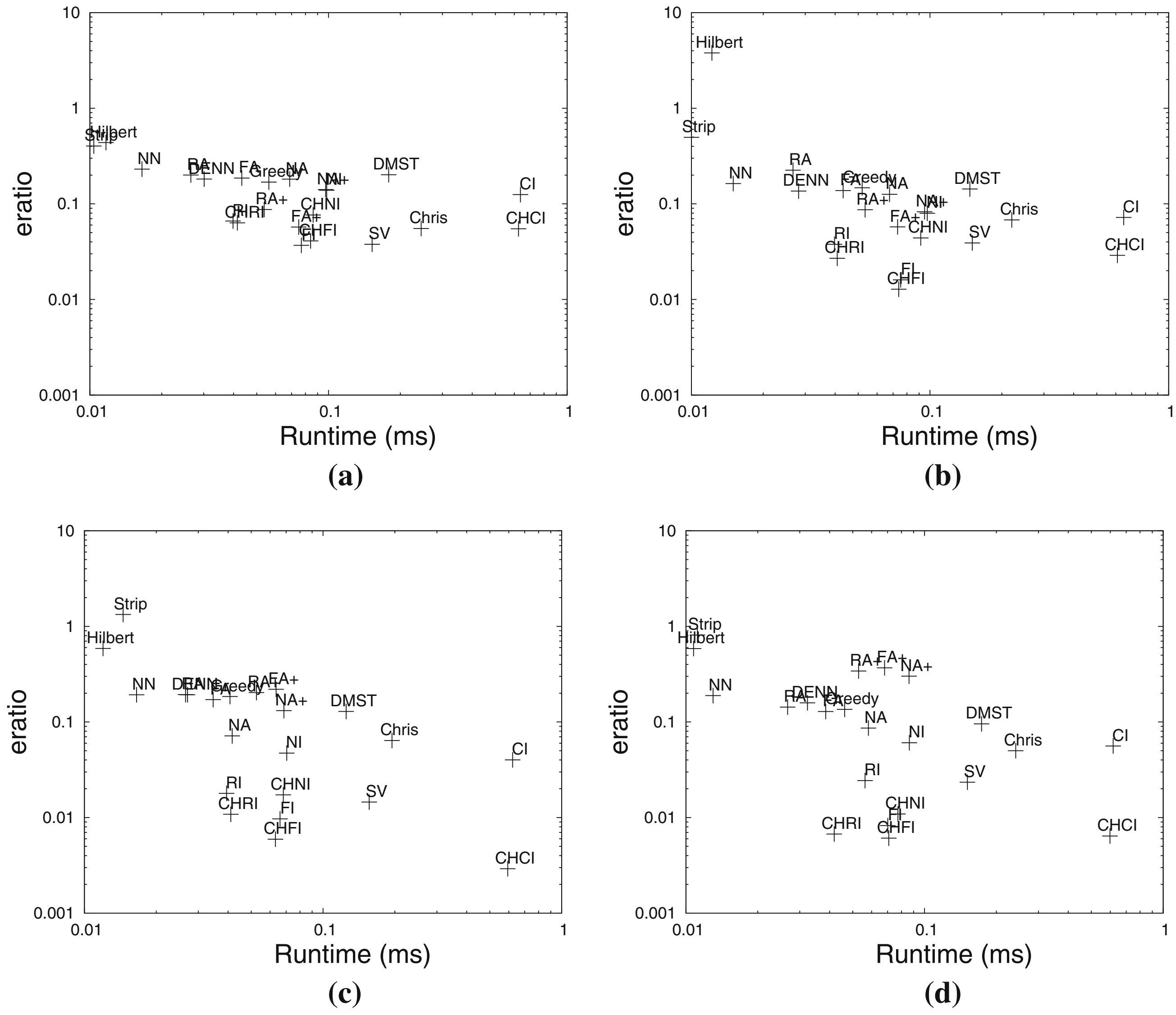


Fig. 5 Accuracy versus Runtime ($|Q| = 60$). **a** BJ. **b** NY. **c** LA. **d** HK

Figure 4 shows the data distributions with different parameters. Each figure contains 3000 points similar to Fig. 1. Figure 4a shows the distribution with default settings, which simulates LA (Fig. 1d). Figure 4b shows the distribution when $K = 1$. It is a normal distribution, similar to the benchmark shown in Fig. 1b. Comparing Fig. 4a, d shows the case when there are more clusters with a smaller variance (larger α). Consequently, it is more skewed. Figure 4d shows how to simulate NY (Fig. 1c). Figure 4c has a larger l.

TSP Queries For real datasets, the *TSP* query size is $n = |Q|$, where the default is 60. For *TSPLIB* datasets, we test it using the same number of points as given in the benchmark. For the 2 existing synthetic datasets, uniform and normal, used in [21], we generate a set of n points for a given size to test. For both real datasets and our synthetic datasets generated using the parameters with $N = 100{,}000$ points, we conduct testing 100 times for a *TSP* query with n points randomly selected, and report the average.

The heuristics We study the 22 heuristics listed in Table 2, which are implemented in C++ following [5, 21], where *KD tree* [4] is used for efficient search. The convex hull for a *TSP* query is implemented by Graham scan [16]. The implementation details can be found in [5]. We have conducted extensive experiments on a PC with two Intel Xeon X5550@2.67GHz CPU and 48GB main memory.

The Measures We measure the heuristics by accuracy and efficiency. The accuracy is based on the error-ratio eratio (Eq. 2), and the efficiency is based on CPU time. We focus on the accuracy, since all the heuristics are fast as reported in [5].

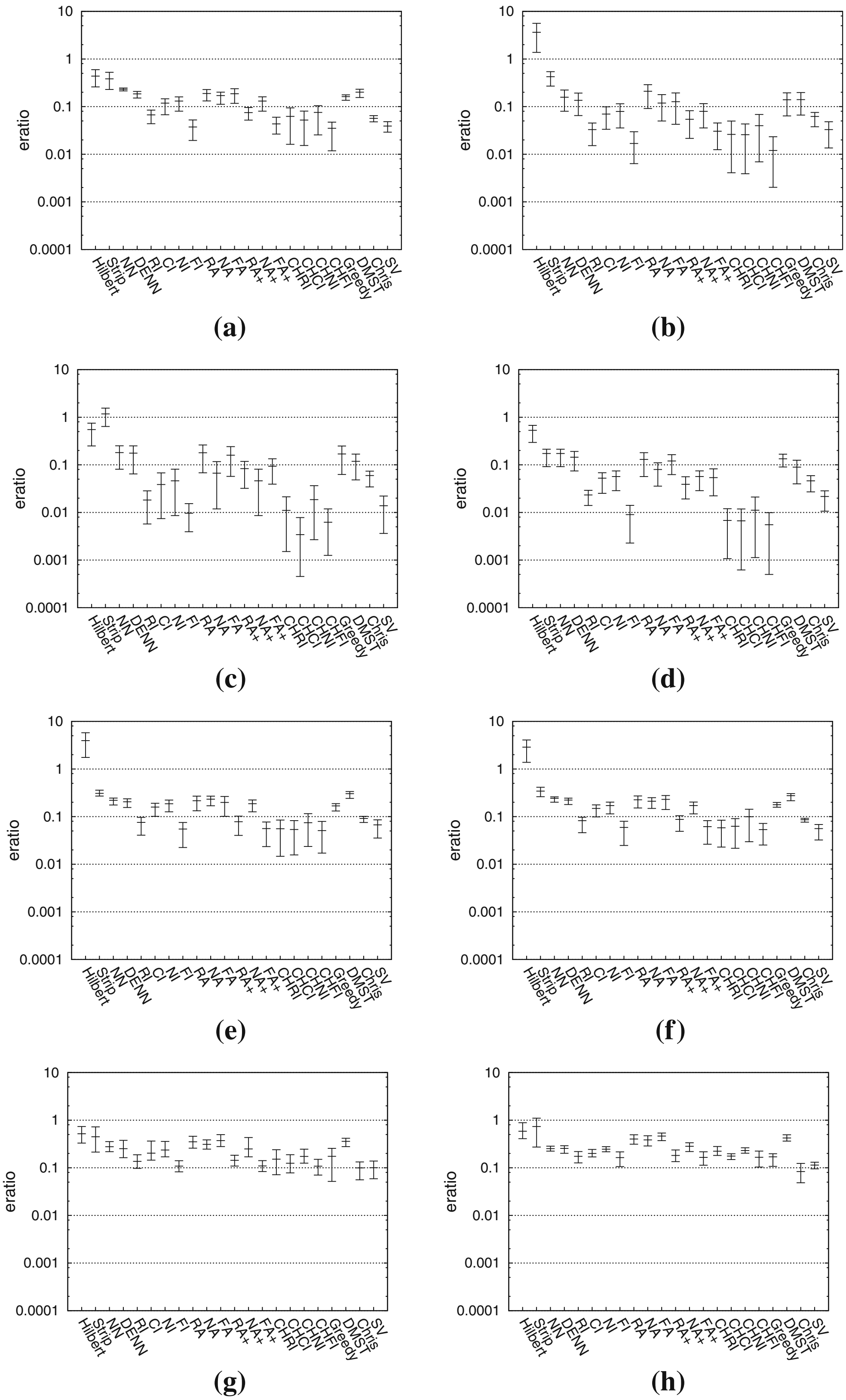


Fig. 6 Accuracy of 22 Heuristics under 26 Datasets ($n = 20, 40, 60, 80, 100$). **a** BJ. **b** NY. **c** LA. **d** HK. **e** Uniform. **f** Normal. **g** TB_H. **h** TB_T

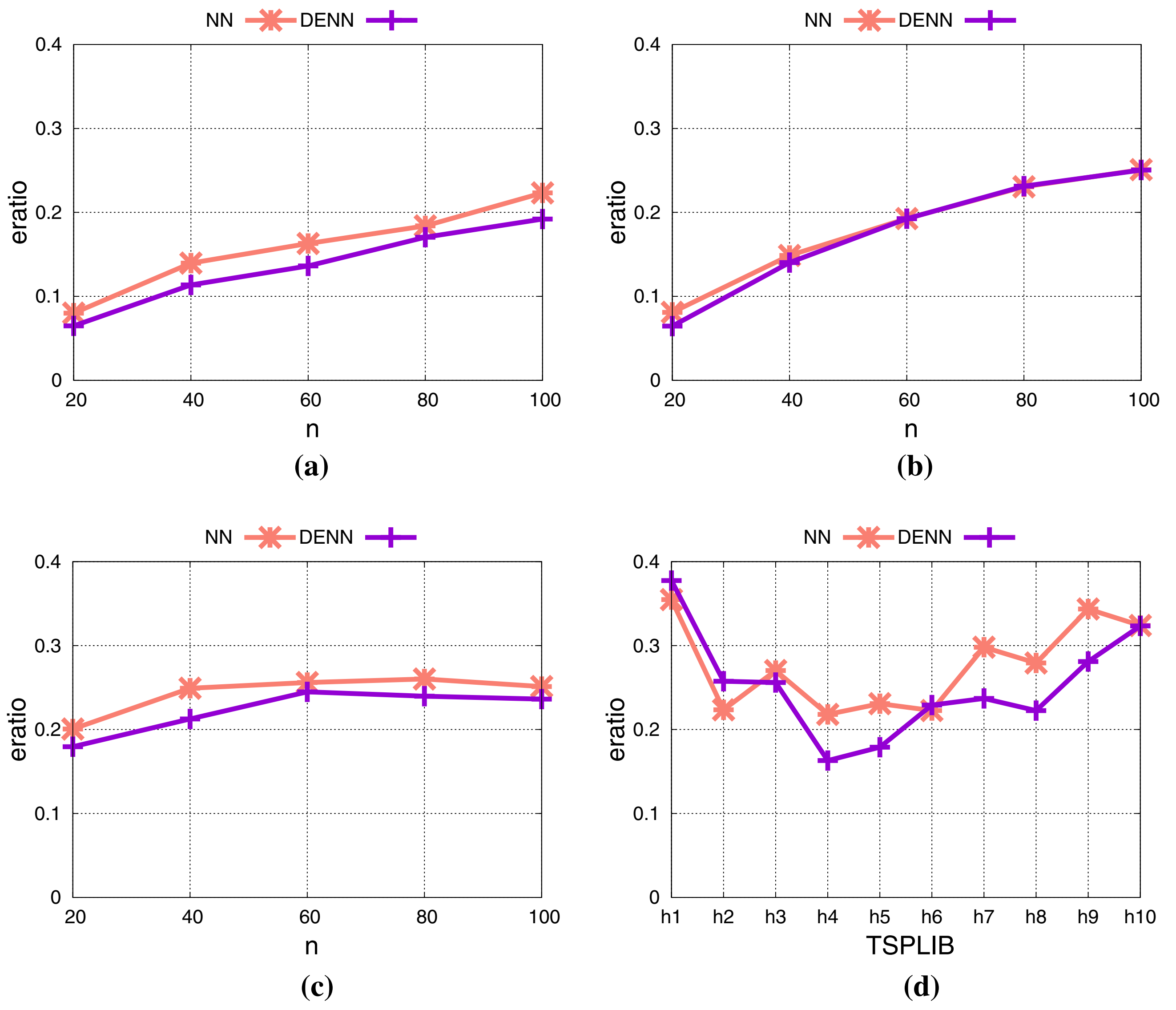


Fig. 7 Nearest-neighbor-based heuristics. **a** NY. **b** LA. **c** Normal. **d** TB_H

Below, we give an overview for the 22 heuristics using real datasets and the existing synthetic datasets in Sect. 4.1. We give details in terms of accuracy in Sect. 4.2, and discuss the issue whether we can find a *TSP* using indexing in Sect. 4.3. Finally, we discuss the heuristics using our new synthetic datasets in Sect. 4.4.

4.1 Accuracy Versus Efficiency

Figure 5 shows CPU time and error-ratio for *TSP* queries of size $n = 60$, for the 4 real datasets. The results shown in Fig. 5 for real datasets highlight the difference from the results conducted in [21] for 9 heuristics with $n = 10{,}000$ using uniform distributed datasets. On the one hand, as shown in [21], SV has the highest accuracy followed by FI, Greedy is better than CHCI, and NN is better than NI. On the other hand, for these 4 real datasets, as shown in Fig. 5, the convex hull-based insertion heuristics achieve near-optimal accuracy, especially for NY, LA, and HK, since BJ is close to the normal distribution due to its urban planning. In other words, the results of BJ share the similarity with those reported in [21].

CPU Time The results of using real datasets are similar to the finding given in [21]. Both Strip and Hilbert are the fastest, as they only need to sort n values. NN and DENN are fast, because the nearest neighbors can be found efficiently using *KD tree*. The convex hull-based insertion heuristics spend time to select the initial convex hull as a sketch in $O(n \lg n)$ time, which is cost-effective since it reduces the number of nodes for insertion. Therefore, the

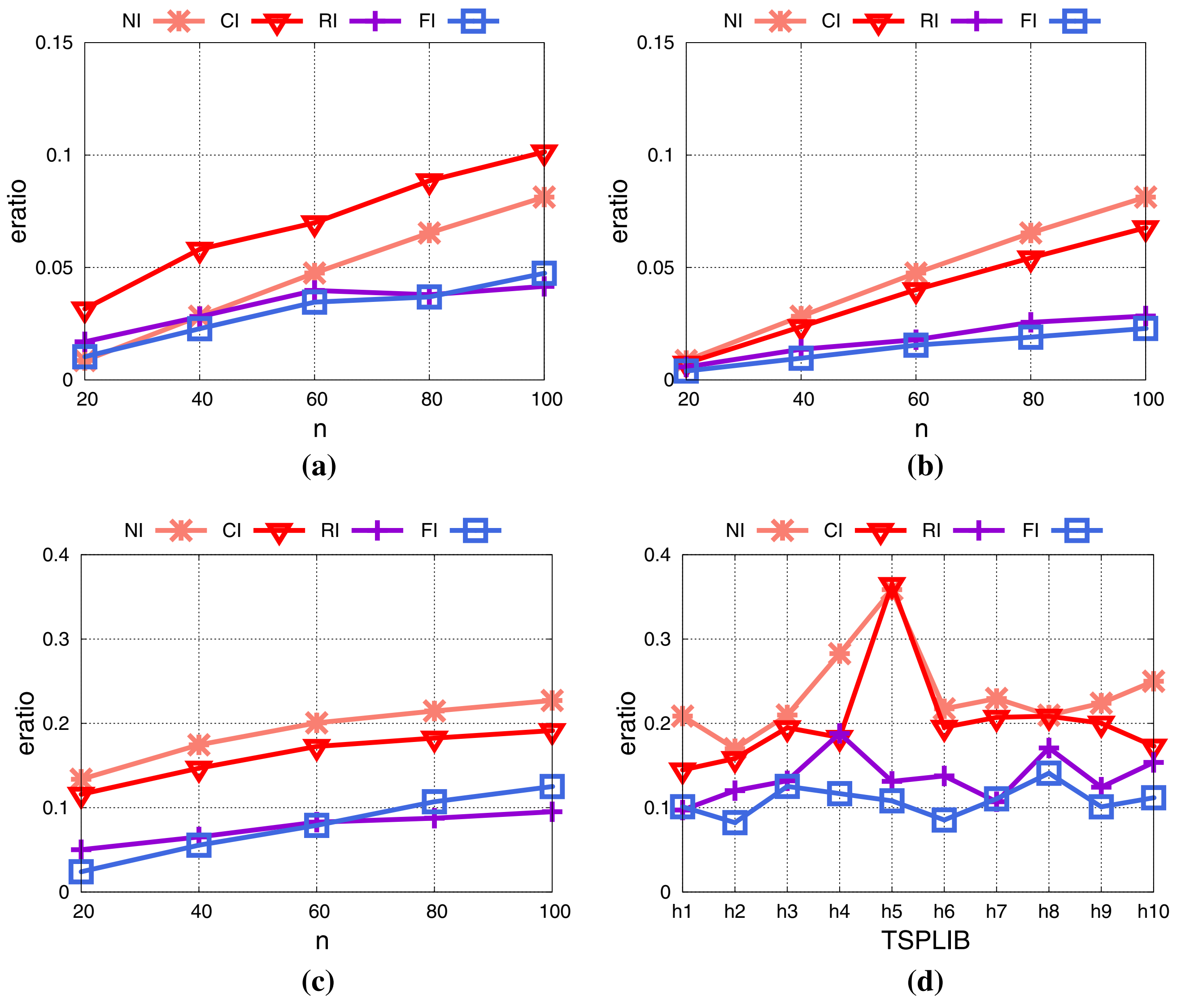


Fig. 8 insertion Heuristics. **a** NY. **b** LA. **c** Normal. **d** TB_H

convex hull-based insertion heuristics can run even faster than the other insertion heuristics which randomly select an initial node to start. For the node-based heuristics, the main CPU cost is to select a node in iterations, and there are 4 expanding orders of nodes in iterations: the random order, the nearest order, the cheapest and the furthest. The random is the fastest, as it picks the next node for insertion randomly. The CPU cost for the nearest and the furthest has only marginal difference, since both can be done using *KD tree*. The cheapest takes the longest time, since it needs to calculate the possible insertion cost for every node and every edge in iterations. For the edge-based heuristics, Greedy is the fastest. Its CPU time is comparable to that by the insertion heuristics. DMST and Chris build an minimum spanning tree before generating the route, and consume more time.

The Error-Ratio In addition to the efficiency (x axis) and the accuracy (y axis) shown in Fig. 5, we further conduct testing for all 22 heuristics over 26 datasets: 4 real datasets (BJ, NY, LA, and HK), 2 synthetic datasets (uniform and normal) and 20 *TSPLIB* benchmarks (10 TB_H and 10 TB_T) for $n = |Q|$ to be selected over 20, 40, 60, 80, and 100. Figure 6 shows the results using candlesticks for the *TSP* queries tested. The differences among the 22 heuristics in terms of accuracy are more obvious than the differences in terms of efficiency, for the datasets tested. Note that the accuracy is related to the distribution, whereas the efficiency is related to the heuristics. As shown in Fig. 6, in terms of accuracy, the range of eratio is between 0.0001 and 10 (y axis). In a short summary, in general, the error-ratio of the 22 heuristics over the real datasets (BJ, NY, LA, HK) is lower than that of the

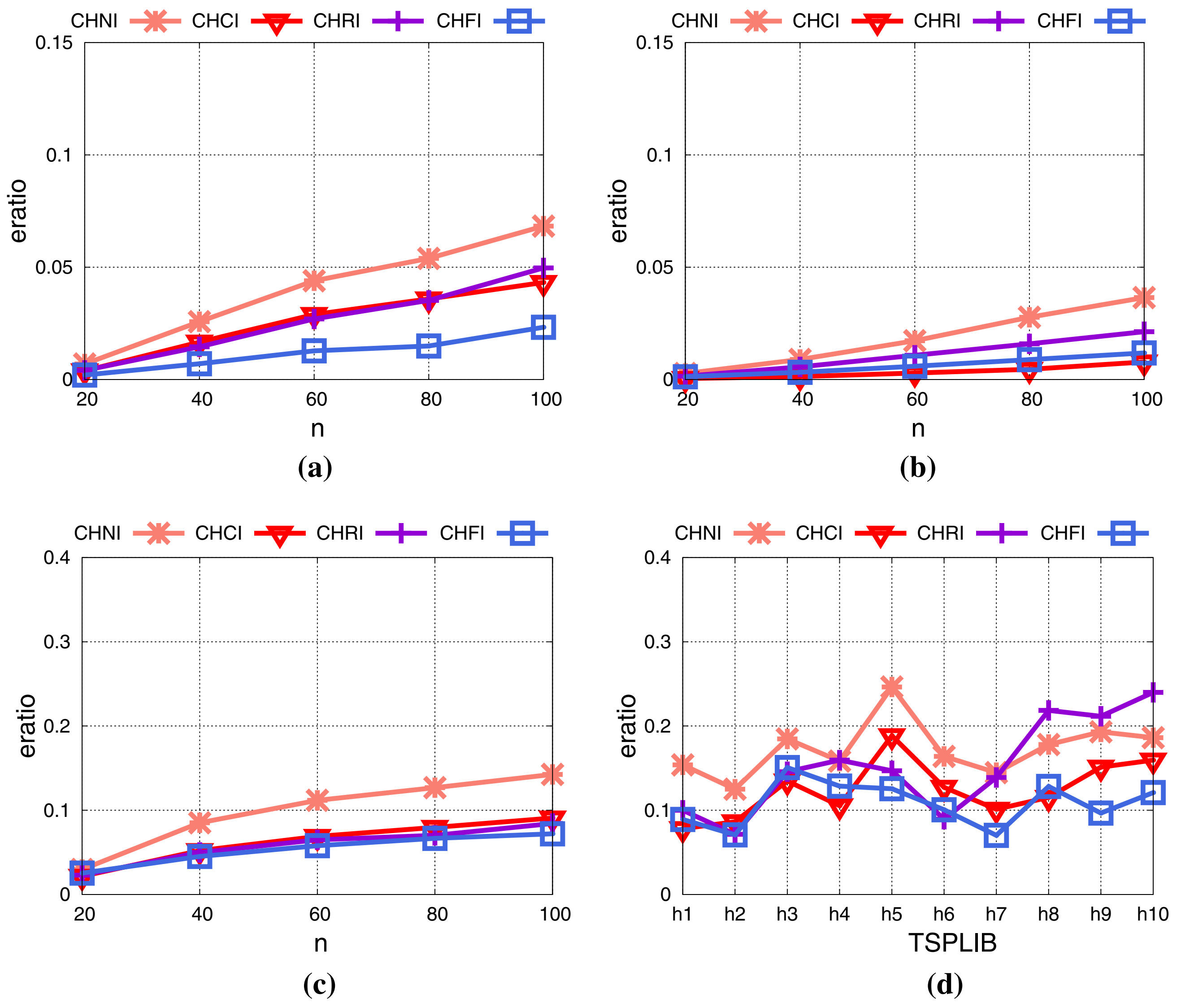


Fig. 9 Convex hull-based insertion heuristics. **a** NY. **b** LA. **c** Normal. **d** TB_H

synthetic datasets by uniform and normal as well as TB, especially for the best cases. Among the 4 real datasets, BJ follows the normal distribution, the error-ratio for the heuristics over BJ is relatively higher. Over NY, LA and HK, most heuristics can generate results with an error-ratio below 0.1 (below 10%). As a comparison, for normal and TB, the error-ratio is larger than 0.1 (10%). Among the 22 heuristics, the convex hull-based insertion heuristics are the best for most cases, whereas the space-partitioning-based heuristics are the worst. The insertion heuristics are better than the augmented addition heuristics which are better than the addition heuristics. For the edge-based heuristics, SV and Chris can be used to obtain accurate answers in some circumstances.

4.2 The Accuracy

In this section, we focus on the accuracy over 2 real datasets, NY and LA, a synthetic dataset by normal, and TB_H benchmarks. The main purpose is to show that the heuristics behave differently in real datasets comparing to normal and TB_H benchmarks. We focus on the node-based and edge-based heuristics, and do not discuss the space-partitioning heuristics since they do not show their advantages in terms of accuracy for *LBS*.

The Nearest-Neighbor Heuristics Figure 7 shows how the accuracy of NN/DENN changes while increasing query size *n*. For the real datasets, the error-ratios increase

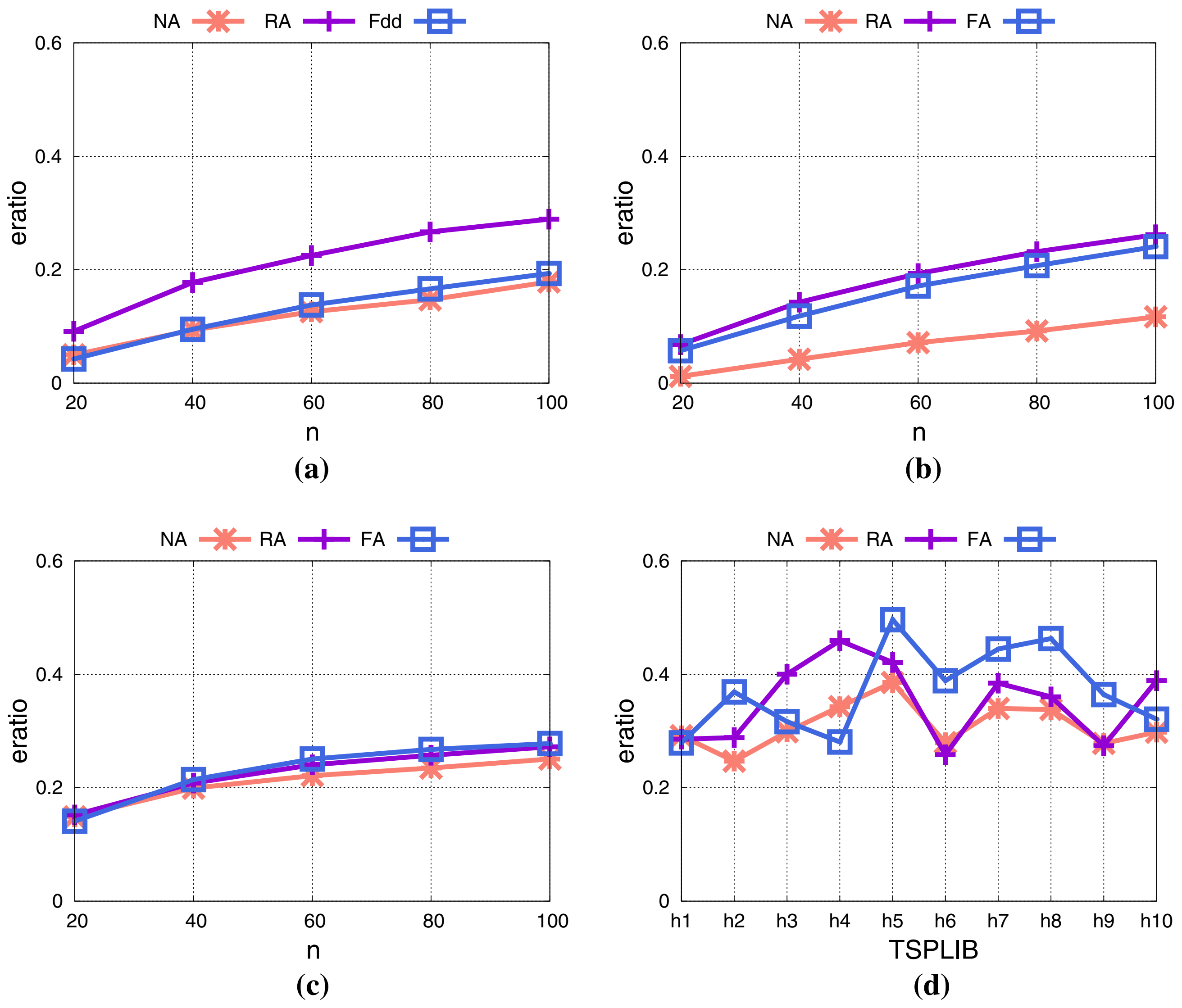


Fig. 10 Addition Heuristics. **a** NY. **b** LA. **c** Normal. **d** TB_H

monotonically. However, it does not hold for non-real datasets, especially for the 10 datasets in TB_H. In TB_H, the first with 101 points has the largest error. This suggests that the accuracy is related to data distribution. In terms of accuracy, DENN is a little better than NN. When n is small, for $n = 20$, the error-ratios for the two over NY and LA are less than 10%, whereas the error-ratio is around 20% for normal, and is even higher than 30% for TB_H. When n becomes larger, the error-ratio for NY and LA increases noticeably, as shown in Fig. 7a, b, which is different from the error-ratios observed from normal.

The Insertion Heuristics Figure 8 shows the error-ratio for the insertion heuristics. Like Fig. 7, the error-ratio increases with query size for the real datasets (NY and LA). The error-ratios for real datasets are much better than that of non-real datasets. The error-ratios range from 0 to 10% for NY and LA, from 0 to 25% for normal, and can be up to 35% for TB_H. Among the 4 insertion heuristics, CI and NI perform in a similar way. It is surprised to notice that RI can perform better than CI and NI in terms of error-ratio. FI performs the best. Both RI and FI have an error-ratio less than 5% for NY, is even less than 3% for LA for all queries tested on average.

The Convex Hull-Based Insertion Heuristics They are the optimal choices for most cases, as shown in Fig. 6. Different from the insertion-based heuristics (RI, NI, CI, and FI) which randomly pick up a node to start, the convex

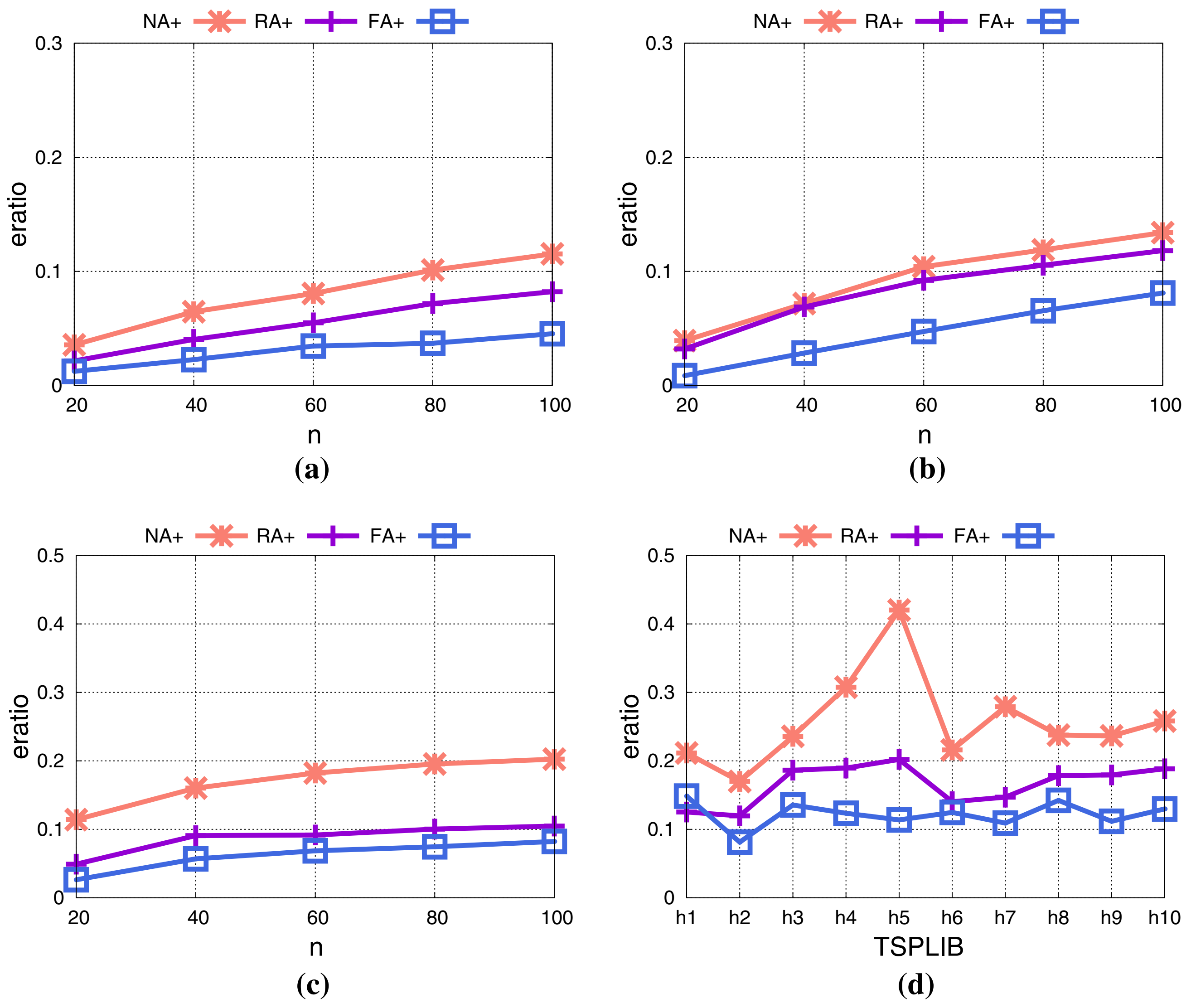


Fig. 11 Augmented addition heuristics. **a** NY. **b** LA. **c** Normal. **d** TB_H

hull-based heuristics find the convex hull as a sketch for a *TSP* query first, and expands the remaining nodes under the guide of the sketch in a similar way as the insertion-based (RI, NI, CI and FI). There are CHRI, CHNI, CHCI, and CHFI. First, comparing Fig. 9 with Fig. 8, the convex hull-based heuristics outperform the corresponding insertion-based heuristics. The error-ratios are noticeably reduced. Second, the reduction on error-ratio by the convex hull changes the order of the heuristics in terms of the accuracy. Consider Figs. 8a and 9a for NY. Without the convex hull, as shown in Fig. 8a, CI performs the worst. Both RI and FI perform in a similar way, and when $n = 100$, RI even outperforms FI. On the other hand, with the convex hull, as shown in Fig. 9a, CHNI (or convex hull plus NI) performs the worst, and CHFI (or convex hull plus FI) outperforms others.

The Addition Heuristics The addition heuristics pick the next node to insert in the same way as the corresponding insertion heuristics. As shown in Table 1 for the *TSP* example in Fig. 2, both FI and FA, and both NI and NA have the same expanding order to select the next node in iterations, respectively. On the other hand, the addition heuristics do not consider all the insertion positions in the current circuit, instead consider only between the two edges that are incident to a node v in the current circuit, where v is the nearest neighbor to the next node selected to insert. As expected, the addition heuristics cannot outperform the corresponding insertion-based heuristics. This is also observed by comparing Fig. 10 with Fig. 8. The advantage of the addition heuristics is the efficiency, since they check much less number of insertion positions on the current circuit in iterations.

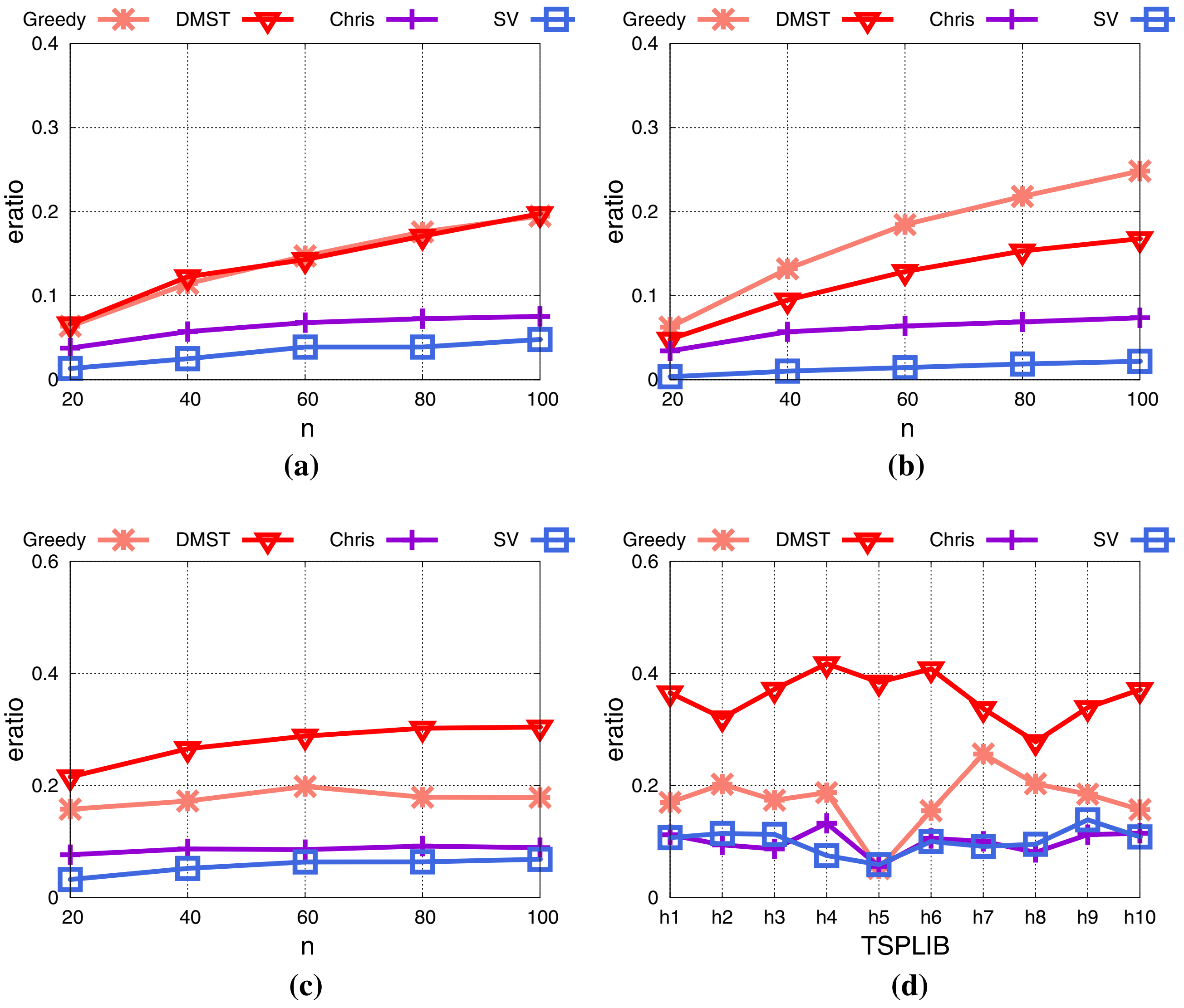


Fig. 12 Edge-based heuristics. **a** NY. **b** LA. **c** Normal. **d** TB_H

It is worth noticing that there are two main things in iterations. One is to select the next node to insert, and the other is to find an insertion position to insert. The following heuristics, RI, CHRI and RA are to select the next node randomly in iterations. As shown in Figs. 8 and 9, RI and CHRI perform well. However, Fig. 10 shows that RA does not perform well, and performs the worst for NY. This suggests that such random heuristics need to explore a certain number of insertion positions on the current circuit, in order to achieve a better accuracy. The same occurs to the furthest heuristics. Both FI and CHFI perform well, but FA does not perform well on the other hand due to the limited number of exploring insertion positions.

The Augmented Addition Heuristics Such heuristics are positioned between the insertion heuristics and the addition heuristics, due to the ways of exploring insertion positions on the current circuit. Figure 11 shows the results for NA+, RA+ and FA+. Comparing Fig. 10 and Figs. 8, 11 shows that, by the limited additional number of insertion positions, RA+ and FA+ outperform NA+, in a similar way as the corresponding RI and FI outperform NI and the corresponding CHRI and CHFI outperform CHNI.

The Edge-Based Heuristics The performance studies done in [21] using $n = 10{,}000$ points under the uniform distribution conclude that SV is the best among all 9 the heuristics tested. FI outperforms Greedy which in turn outperforms CHCI. Different from the results reported in [21] using $n = 10,000$ points under the uniform distribution, as shown in Fig. 6, with many different datasets, SV is not the best, even though SV like other edge-based

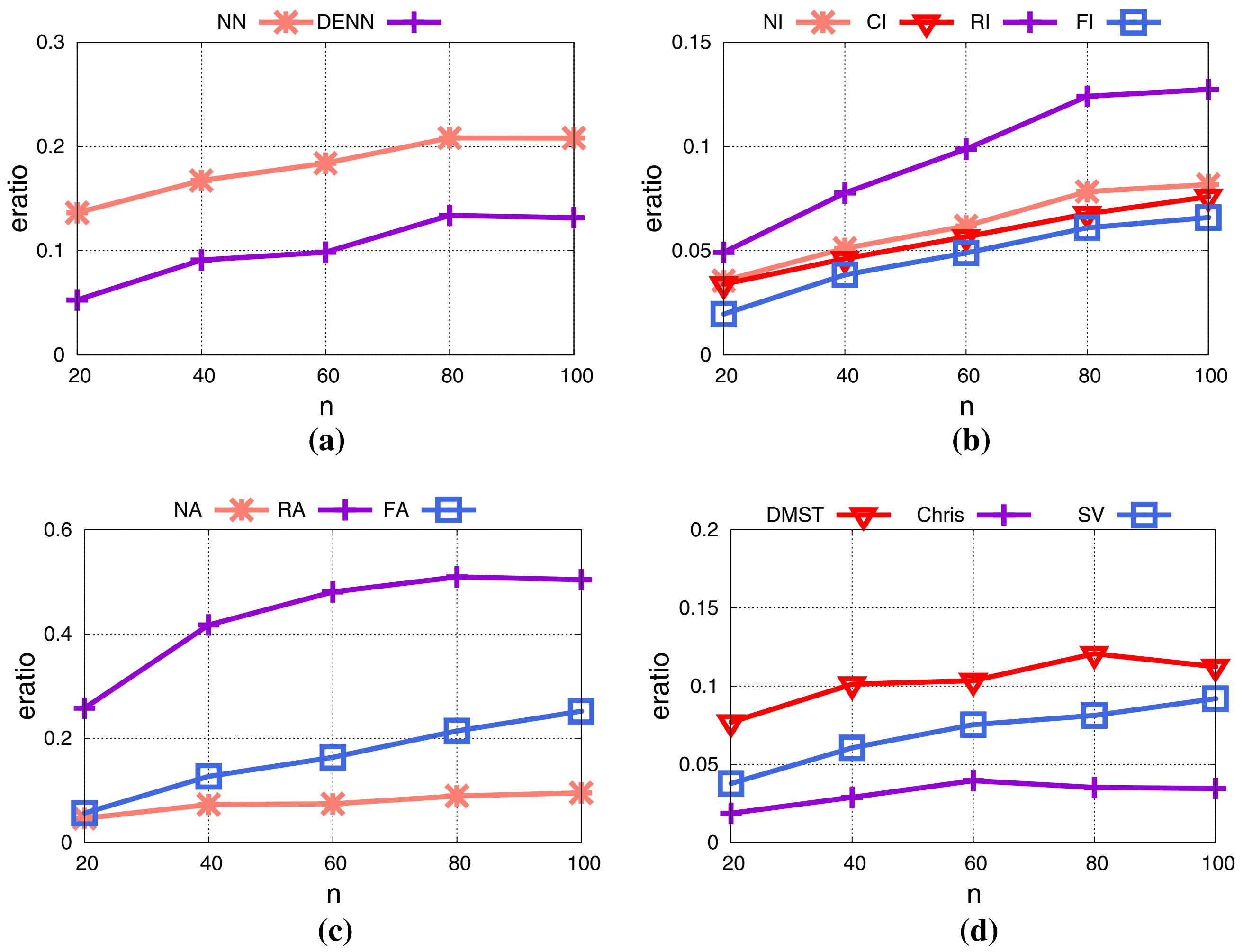


Fig. 13 Effectiveness of Start Nodes (NY). **a** NN/DENN. **b** Insertion. **c** Addition. **d** Edge-Based

heuristics can get better accuracy. Figure 12 shows the details for the edge-based heuristics. SV performs the best. For LA, the error-ratio for SV ranges from 0.2 to 2.3%, which is marginally higher than the error-ratio for CHFI that ranges from 0.1 to 1.2%. Both Chris and DMST are based on *MST*, and Chris outperforms DMST. Note that the approximate ratio is 1.5 for Chris, and is 2 for DMST. As shown in Fig. 5, Chris takes longer CPU time since it needs to find a perfect matching and Euler circuit after generating the minimum spanning tree. Greedy performs well as reported in [21]. However, in our testing settings, Greedy is inferior of Chris in the two synthetic datasets NY and LA, generates similar results with DMST in NY, and is the worst for LA.

The Effectiveness of Start Nodes All heuristics except for the convex hull-based need to select a start node to start expanding randomly, which may have a great impact on the resulting circuit. We study the selection of start nodes using the real dataset NY by varying the number of points $n = 20, 40, 60, 80, 100$. For each n value, for example, $n = 60$, we randomly select 100 sets of points of size n from NY. Assume Q is one of the 100 sets for a given n. We test a certain heuristics by selecting every node in Q as a start node. Given a query size n, we record the difference between the shortest and longest routes for every start node in every of the 100 randomly selected sets, and show the average difference in Fig. 13. It shows how the choice of start node affects the quality. As observed, the error-ratios increase monotonically while the query size increases. This suggests that the random selection of the first node to start is more sensitive when there are more node. As shown in Fig. 13, among the 2 nearest-neighbor heuristics, DENN is more stable than NN, given that DENN is only difference from NN by looking at the two ends of the current path in iterations, instead of only one end. For the insertion heuristics, CI, NI, and FI perform in a similar way, whereas RI is more sensitive to the first node selected to start

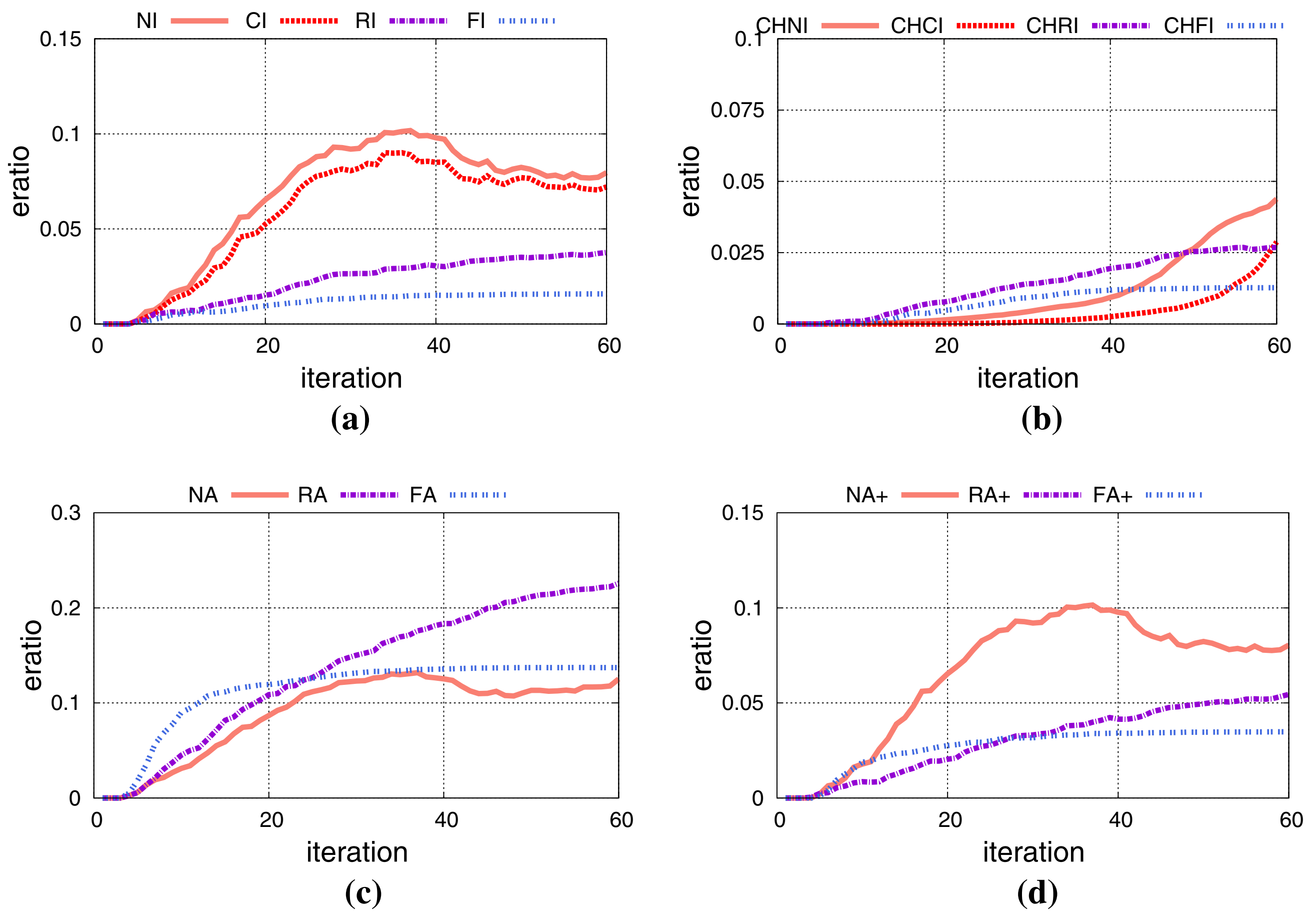


Fig. 14 Error-Ratio in the *i*th Iteration. **a** Insertion. **b** Convex Hull-based Insertion. **c** Addition. **d** Augmented Addition

(Fig. 13b). The sensitive of the random is also shown in Fig. 13c where RA is the worst, and NA and FA outperform RA significantly. For the edge-based heuristics, Fig. 13d shows that it is less sensitive for the edge-based heuristics to select the start node. Chris is the best among all heuristics, in terms of random selection of the start node.

4.3 More on Accuracy

We further analyze the error-ratios for the insertion, the convex hull-based insertion, the addition, and the augmented addition heuristics, and we focus on three issues: (a) the error-ratios in iterations, (b) the error correlation between the intermediate error-ratios and the final error-ratio and (c) the possibility of reoptimization to obtain the $(i{+}1)$th expansion by heuristics given the optimal *TSP* for the first i nodes selected. We conduct testing over the real dataset NY, for $n = 60$. We randomly select 100 sets of points of size $n = 60$ from NY, and report the average. Let T_i be a circuit with i nodes, and let T_i^* be the optimal circuit over the same set of nodes.

The Error-Ratio in the *i* th Iteration The error-ratio $\mathsf{eratio}(\mathsf{T}_i)$ is computed by Eq. (2) for T_i in the *i*th iteration. Figure 14 shows the results. Several observations are made. First, for the random methods (RI, CHRI, RA and RA+), the error-ratios increase in the *i*th iteration when *i* becomes larger. The error-ratio in the *i*th iteration becomes comparatively smaller, if it tries to find an insertion position among more choices, i.e., RI is better than RA+, which is better than RA. Among all the random-based methods, CHRI is best given the convex hull computed. Second, the error-ratios for the addition heuristics are high due to the limited insertion positions in iterations. Third, for the nearest-neighbor methods (NI, CHNI, CHCI), the error-ratio increases in the first iterations and then drop in the late iterations. The reason is that in the first iterations, it takes near-to-far approach, whereas in the late iterations it may insert the next node between two nodes to refine the circuit. For CHCI, given the convex hull computed, it increases, and terminates before it finds the position to drop. Fourth, for the furthest methods (FI, CHFI, FA+, and FA), the error-ratios are small and grow slowly. Among all heuristics, CHFI performs the best.

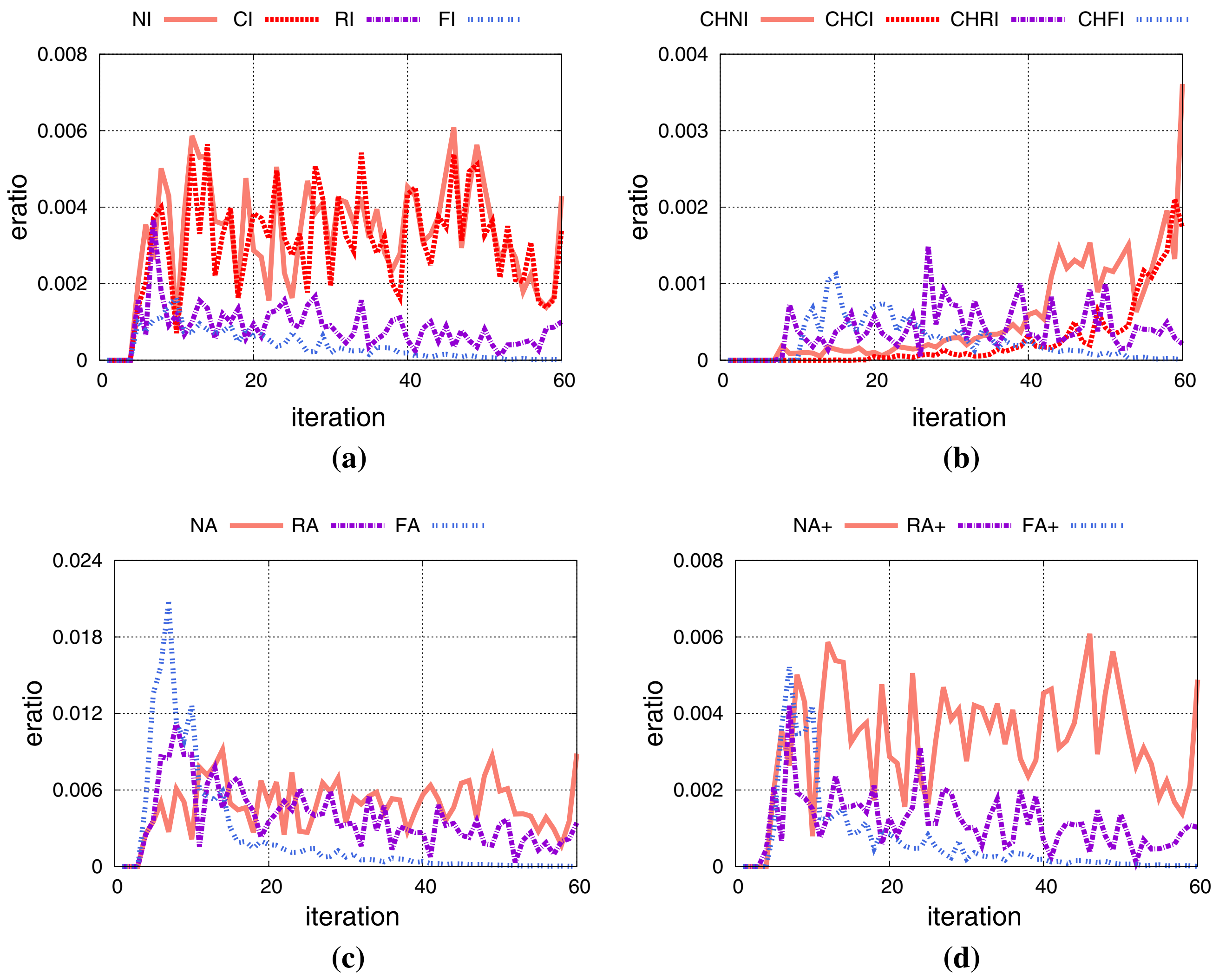


Fig. 15 Reoptimization. **a** Insertion. **b** Convex hull-based insertion. **c** Addition. **d** Augmented addition

The Reoptimization For reoptimization, the rationale behind is whether there is any possibility to answer a *TSP* query based on an indexing to maintain certain length optimal circuit, even though it is known that the local optimal property does not hold. For a given heuristic method to obtain T_{i+1} by expanding a new node from T_i, we consider expanding the new node into the optimal T_i^* over the same set of nodes in T_i generated by the same method. We denote such a circuit obtained as T'_{i+1}, and the error-ratio by $\mathsf{eratio}(\mathsf{T}')$ is the error-ratio introduced in the last iteration only. Figure 15 shows the introduced error-ratio for different heuristics. The introduced errors are very small, which indicates that the reoptimization can generate high-quality solutions in most cases. For the addition heuristics, the error-ratio is higher than the others (Fig. 15c). For the insertion heuristics, FI outperforms the others, and RI also performs well. It is worth noting that when T_i becomes longer, the error-ratios become smaller. When $i > 30$, the error-ratio is below 0.001, and when $i > 40$, the error-ratio is close to zero. In general, the convex hull-based insertion better than the insertion.

4.4 The New Synthetic Datasets

In order to better understand the heuristics in real applications for *LBS*, we study the 22 heuristics in terms of accuracy using the new synthetic datasets proposed in this work.

Figure 16 gives an overview by candlesticks, where each figure is presented by varying the parameter in concern while fixing the other parameters by their default value. In this study, we use $n = 100$ as the default query size. Figure 16a shows that most node-based heuristics are sensitive to the query size, especially of CHCI. The edge-based heuristics are stable with the change of n. The difference between the best case and worst case of Chris is

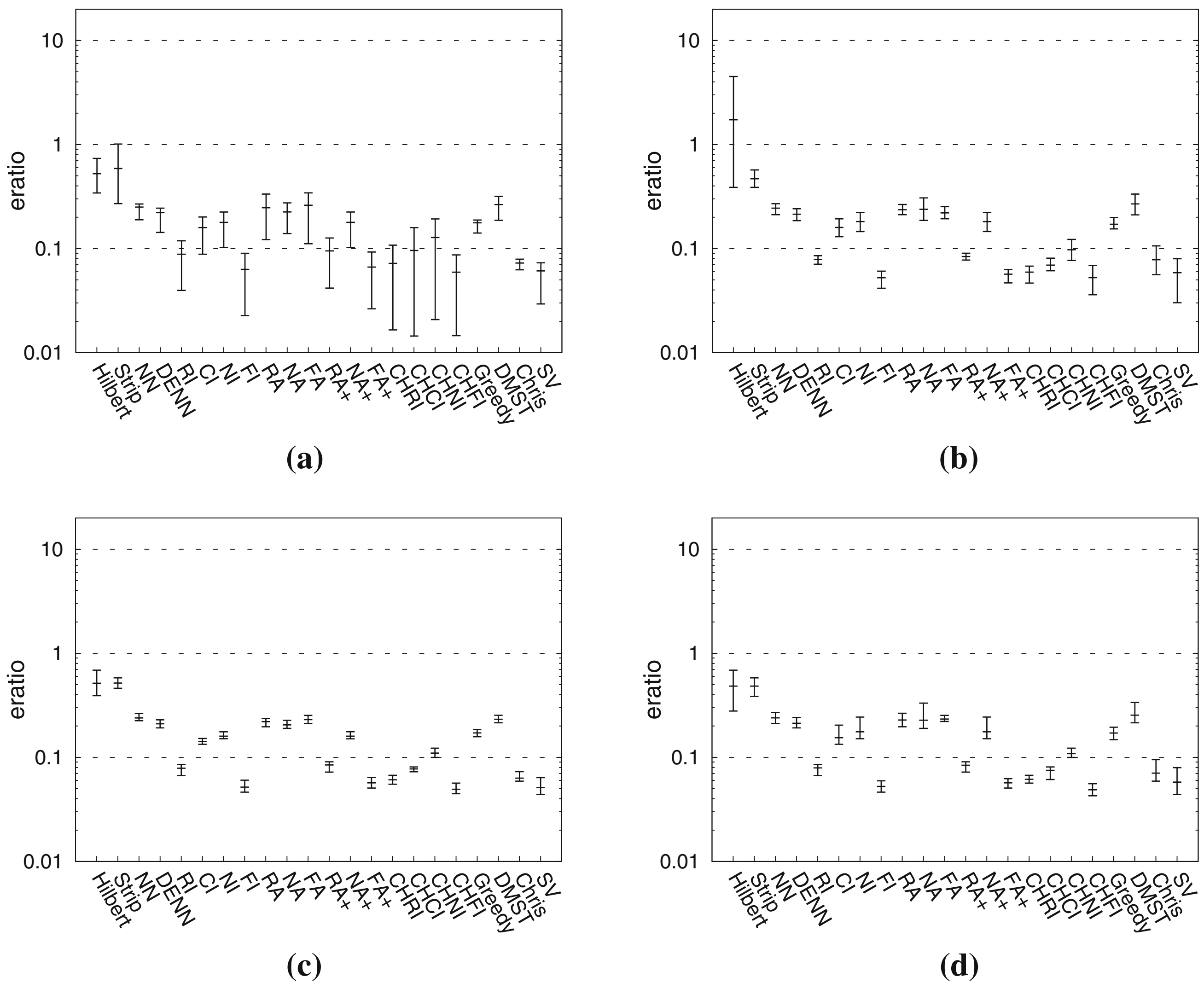


Fig. 16 Impact of parameters: An overview. **a** Query size. **b** Cluster number. **c** Inter-cluster distance. **d** Cluster distribution

just 1.5%. On the other hand, the edge base heuristics are sensitive to the cluster number.

We focus on 6 heuristics in Fig. 17, namely CHFI, CHRI, SV, CHCI, Greedy and DENN, which are selected for the following reasons. CHFI, CHCI and CHRI are the better choices for real datasets tested, SV, CHCI, and Greedy are the better choices as concluded in [21], and DENN is selected as the nearest-neighbor heuristics which is used in *LBS*. Note that DENN outperforms NN in terms of accuracy.

Query Size (n): The error-ratios for all heuristics increase while increasing n. As shown in Fig. 16a, it has higher influence on node-based heuristics (CHCI, CHRI and CHFI). When n is small, CHCI, CHRI and CHFI outperform SV. However, as SV increases slower, SV becomes the best when $n \geq 400$. For Greedy, it looks constant when $n \geq 80$, and will perform well when n is very large. This explains that Greedy outperforms CHCI in [21].

Cluster Number (K): As shown in Fig. 17b, the cluster number has little influence on the heuristics. The dataset follows a normal distribution when $K = 1$ and gradually changes to uniform distribution when K becomes large. The same can be observed in Fig. 6e and f.

Inter-Cluster Distance (l) The inter-cluster distance controls the distance between clusters. Note that the overlap between clusters becomes small when l is large. The error-ratios for CHFI and SV increase slightly while increasing l. For DENN, CHRI, CHCI, and Greedy, the error-ratio increases first and then decrease. SV increases faster than CHFI. When $l = 1$, SV outperforms CHFI. However, CHFI outperforms SV when $l \geq 10$ when the boundaries between clusters are more clear.

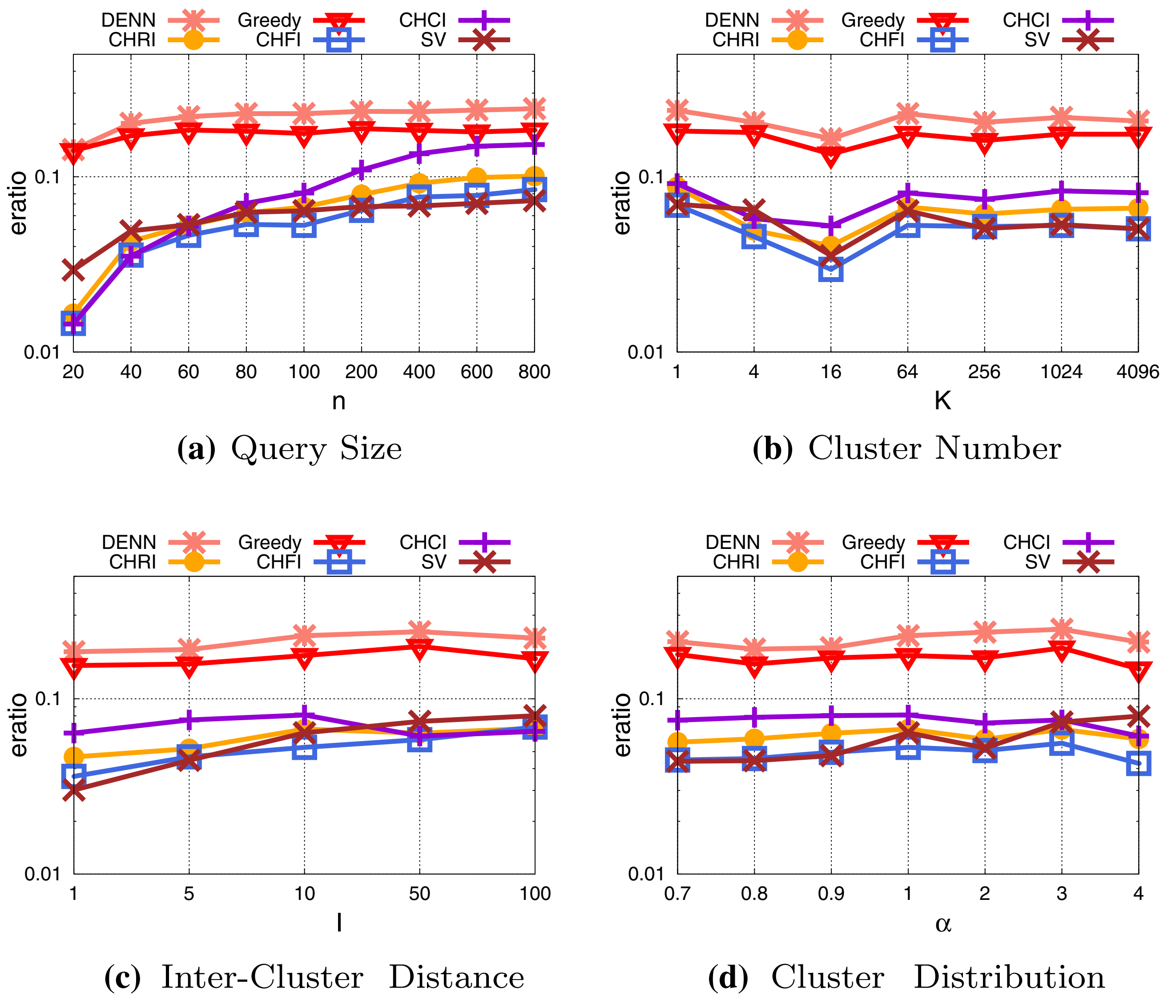


Fig. 17 Impact of Parameters: The Details. **a** Query size. **b** Cluster number. **c** Inter-cluster distance. **d** Cluster distribution

Cluster Distribution (α) It is to control the variances in clusters. When α is small, the data distribution is highly skewed, and most clusters have a small variance. On the other hand, when α is large, more clusters have a large variance and the distribution over all clusters in terms of variance is more uniform. As shown in Fig. 17d, the error-ratio for CHFI increases while increasing α at the beginning and then decreases. The error-ratio for SV increases monotonously. CHFI is more suitable for the skewed case than SV.

As a summary, we conclude the following. First, CHFI works well for real *LBS* applications, in particular, when the query size is relatively small and the query is highly skewed. Second, SV is a better choice when the query size is large and query distribution is uniform as also observed in the existing work. Third, the nearest-neighbor heuristics is currently used in *LBS* for the efficiency, and is not the best for accuracy. Fourth, CHRI can generate the near-optimal answers in many cases by randomly selecting next nodes to insert, and shows that the random heuristics is deserved to be investigated.

5 Conclusion

In this work, we investigate 22 construction heuristics for *TSP* in *LBS* by extensive performance studies over 4 real datasets, 20 datasets from *TSPLIB* benchmark, and 2 existing synthetic datasets. In addition, in order to understand real *LBS* setting, we also conduct extensive testing over the new synthetic datasets proposed in this work to simulate that a small number of hot *POI*s are visited by many people. Different from the existing work, we find that CHFI works well for real *LBS* applications, whereas CHCI get a good answer when the query size is small. Also, CHRI can generate the near-optimal answers in many cases by randomly selecting next points to insert. In addition, for the issue of precomputing/indexing, we find that the quality of the circuit T_{i+1} by expanding a new point by heuristics from the optimal T_i^* is high, which shows that it is deserved to study precomputing/indexing to support *TSP* queries.

Acknowledgements This work was supported by grant of the Research Grants Council of the Hong Kong SAR, China 14209314.

References

1. Arora S (1998) Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. J ACM 45(5):753–782
2. Beardwood J, Halton JH, Hammersley JM (1959) The shortest path through many points. In: Mathematical proceedings of the Cambridge philosophical society. Cambridge University Press, Cambridge, vol 55, pp 299–327
3. Bellmore M, Nemhauser GL (1968) The traveling salesman problem: a survey. Oper Res 16(3):538–558
4. Bentley JL (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517
5. Bentley JL (1992) Fast algorithms for geometric traveling salesman problems. INFORMS J Comput 4(4):387–411
6. Cao X, Chen L, Cong G, Xiao X (2012) Keyword-aware optimal route search. PVLDB 5(11):1136–1147
7. Cao X, Cong G, Jensen CS (2010) Mining significant semantic locations from GPS data. PVLDB 3(1):1009–1020
8. Chen Z, Shen HT, Zhou X (2011) Discovering popular routes from trajectories. In: ICDE, pp 900–911
9. Chen Z, Shen HT, Zhou X, Zheng Y, Xie X (2010) Searching trajectories by locations: an efficiency study. In: SIGMOD, pp 255–266
10. Cho E, Myers SA, Leskovec J (2011) Friendship and mobility: user movement in location-based social networks. In: SIGKDD, pp 1082–1090
11. Christofides N (1976) Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document
12. Cong G, Jensen CS, Wu D (2009) Efficient retrieval of the top-k most relevant spatial web objects. PVLDB 2(1):337–348
13. Cressie N (2015) Statistics for spatial data. Wiley, New York
14. Flood MM (1956) The traveling-salesman problem. Oper Res 4(1):61–75
15. Gonzalez MC, Hidalgo CA, Barabasi A-L (2008) Understanding individual human mobility patterns. Nature 453(7196):779–782
16. Graham RL (1972) An efficient algorithm for determining the convex hull of a finite planar set. Inf Process Lett 1(4):132–133
17. Guo L, Zhang D, Li G, Tan K, Bao Z (2015) Location-aware pub/sub system: When continuous moving queries meet dynamic event streams. In: SIGMOD, pp 843–857
18. Guo T, Cao X, Cong G (2015) Efficient algorithms for answering the m-closest keywords query. In: SIGMOD, pp 405–418
19. Gutin G, Punnen AP (2002) The traveling salesman problem and its variations, vol 12. Springer Science & Business Media, Berlin
20. Johnson DS, McGeoch LA (1997) The traveling salesman problem: A case study in local optimization. Local search in combinatorial optimization 1:215–310
21. Johnson DS, McGeoch LA (2007) Experimental analysis of heuristics for the stsp. In: The traveling salesman problem and its variations. Springer, Berlin, pp 369–443
22. Kruskal JB (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. Proc Am Math Soc 7(1):48–50
23. Li F, Cheng D, Hadjieleftheriou M, Kollios G, Teng S (2005) On trip planning queries in spatial databases. In: SSTD, pp 273–290
24. Li G, Chen S, Feng J, Tan K, Li W (2014) Efficient location-aware influence maximization. In: SIGMOD, pp 87–98
25. Long C, Wong RC, Wang K, Fu AW (2013) Collective spatial keyword queries: a distance owner-driven approach. In: SIGMOD, pp 689–700
26. Luo W, Tan H, Chen L, Ni LM (2013) Finding time period-based most frequent path in big trajectory data. In: SIGMOD, pp 713–724
27. MacGregor JN, Ormerod T (1996) Human performance on the traveling salesman problem. Percept Psychophys 58(4):527–539
28. Nicholson T (1967) A sequential method for discrete optimization problems and its application to the assignment, travelling salesman, and three machine scheduling problems. IMA J Appl Math 3(4):362–375
29. Platzman LK, Bartholdi JJ III (1989) Spacefilling curves and the planar travelling salesman problem. J ACM (JACM) 36(4):719–737
30. Reinelt G (1991) Tsplib-a traveling salesman problem library. ORSA J Comput 3(4):376–384
31. Rosenkrantz DJ, Stearns RE, P. M. L. II. (1977) An analysis of several heuristics for the traveling salesman problem. SIAM J Comput 6(3):563–581
32. Schabenberger O, Gotway CA (2004) Statistical methods for spatial data analysis. CRC Press, Boca Raton
33. Sharifzadeh M, Kolahdouzan MR, Shahabi C (2008) The optimal sequenced route query. VLDB J 17(4):765–787
34. Sommer C (2014) Shortest-path queries in static networks. ACM Comput Surv 46(4):45:1–45:31
35. Wang S, Lin W, Yang Y, Xiao X, Zhou S (2015) Efficient route planning on public transportation networks: A labelling approach. In: SIGMOD, pp 967–982
36. Xu Z, Jacobsen H (2010) Processing proximity relations in road networks. In: SIGMOD, pp 243–254
37. Yan D, Zhao Z, Ng W (2011) Efficient algorithms for finding optimal meeting point on road networks. PVLDB 4(11):968–979
38. Zhu AD, Ma H, Xiao X, Luo S, Tang Y, Zhou S (2013) Shortest path and distance queries on road networks: towards bridging theory and practice. In: SIGMOD, pp 857–868