2017 IEEE 18th International Conference on Mobile Data Management

Towards Optimal Free-of-Charge Trip Planning in Bike-Sharing Systems

Jian Xu*, Jianliang Xu†, Guanjie Cao*, Haitao Xu*, Ming Xu*, Ning Zheng*
*Hangzhou Dianzi University
Hangzhou, China
Email:{jian.xu, 13055109, xuhaitao, mxu, nzheng}@hdu.edu.cn
†Hong Kong Baptist University
Hong Kong, China
Email:xujl@comp.hkbu.edu.hk

***Abstract*—Most bike-sharing service providers offer a free ride for a short time period. In this paper, we study how to find the optimal route that is free of rental cost and minimizes the trip distance from one location to another within a bike-sharing system, in which the utilization of bike stations dynamically changes over time. We use a time-dependent dynamic graph to model the network of bike stations. In the graph, each vertex represents a bike station and is associated with a vertex-usage function. The difficulty of this problem is mainly attributed to the fluctuation of the usage function because a fully utilized station cannot accept returned bikes. Efficiency is another challenge, as we must explore all possible paths between the source and the destination. To address these challenges, we propose techniques to find a solution optimized for efficiency. First, to reduce the search space, we construct a station network graph on top of a road network. Next, we employ a paths-tree to identify all the paths with lengths less than the user-preferred maximum detour distance and we select an optimal path toward the destination. We present the design details of our algorithms and we analyze the algorithms' correctness and complexity. To demonstrate the feasibility of our methods, we also report the results from the extensive experiments we conducted.**

***Keywords*-bike-sharing; trip planning**

I. Introduction

Services with the magic word "free" attached are often welcomed enthusiastically by consumers. Public bike-sharing services are becoming more and more popular in recent years. Most bike-sharing systems (BSS) in operation provide a short period of free-ride time, which usually amounts to 30 minutes or one hour, to encourage people to travel by bike.

A BSS is composed of many bike stations, each of which allowing a limited number of bikes to be parked. The bikes are picked up from one station and then returned —either to the same or to another station. A station is a dock with slots to secure bikes and also includes a kiosk with a computer connected to a back-end database. A slot is a locking point at a bike station.

Assume that the time period of a free ride is T_{free}. A user with a trip longer than T_{free} may return the bike to a free slot at a station along the route within T_{free}, select another bike (or the same one) from the station, and then continue riding. In this way, the user can avoid paying usage fees. Take for example, the bike-sharing program REDBike in Hangzhou, famous for its bright red bikes. For each day in October 2015, there were about 412,900 check-outs. It is interesting to observe that among these check-outs, about 23,700 bikes were picked up immediately after a return. That is, the user returned a bike and then borrowed another instantly. Finding a feasible route based on real-time information and future availability of slots, offers us an interesting topic for research.

But we face two challenges. First, it is complicated to find a vacant station on the planned route due to the rental dynamics. Second, it is hard to find a free-of-charge trip path efficiently.

In this paper, we study how to help users plan a free-of-charge trip. We investigate how the station usage data from REDBike can be used to predict future station utilization. We also show how a network of stations is used to construct a route from the source to the destination. In particular, the main contributions of this paper are:

- exploring the details of constructing a station graph from a road network, which is used as the basis for searching a route;
- demonstrating the potential of utilizing historical bike-sharing data to make predictions;
- proposing an algorithm that constructs a unique paths-tree to facilitate the heuristic searching process and returns a free-of-charge path;
- analyzing and evaluating our algorithms on both the synthesized and real data from REDBike to validate their correctness and feasibility.

The rest of the paper is structured as follows. Section II describes the model and problem. Section III describes the construction of our graph of the station network. Section IV presents the prediction methods. Section V provides a detailed introduction of proposed algorithms. Section VI presents the results of our experiment. Section VII follows with related works. Section VIII concludes this paper.

2375-0324/17 $31.00 © 2017 IEEE
DOI 10.1109/MDM.2017.22

II. Preliminaries

In this work, road networks are modeled by a connected and directed planar graph *G(V, E)*, where V is the set of vertices and E is the set of edges, and a weight can be assigned to each edge representing the length between two neighboring vertices. The bicycle stations are embedded in the road networks and they can be located in the edges. If the distances to the two end vertices of an edge are known, it is straightforward to derive the network distance to any point on this edge. Thus, for the sake of simplicity, we assume that all bicycle stations are in vertices. We denote a set of vertices that represents bike stations with V_b, and we have $V_b \subset V$.

Definition 1 (Station's utilization): Suppose a station $u_i \in V_b$, has a fixed capacity k_i, which corresponds to a maximum number of slots, and means that k_i bikes can be parked at this station. The number of occupied slots (that is, those not available for returning bikes) at station u_i at time slice t is denoted by $X_i(t) \in [0, k_i]$. If $X_i(t) < k_i$ at time t, then we say the station is available for returning bikes. If $X_i(t) = k_i$, then we say the station is full.

Stations range in size from 20 to 30 slots in REDBike system. Therefore, a station's number of occupied slots is normalized by dividing it by that station's total number of slots. We use the function $util(t)$ to specify a station's utilization. For example, when $util(t)$=1, the station has no available slot.

Definition 2 (Free-of-charge path): Suppose for vertices $u, w \in V_b$, there is a path that compresses the ordinary road vertices and is represented by $p = \langle u, v_0, v_1, ..., v_k, w \rangle$ from u to w, where $v_i \in V_b$, for all $0 \leq i \leq k$. For any two successive vertices v_i, v_j on the path, where $v_i.util(t) \neq 1$ and $v_j.util(t') \neq 1$ (t, t' are the times the user arrives at vertices v_i, v_j respectively), if $t' - t < T_{free}$, we say path p is free of charge.

In other words, assuming an average biking speed $Speed$ and denoting the shortest distance between v_i and v_j with d_{ij}, if $Speed \times T_{free} < d_{ij}$, the user cannot reach v_j from v_i within time T_{free}. So path p is not free of charge.

The length $d(p)$ of a free-of-charge path $p = \langle v_0, v_1, ..., v_k \rangle$ is the sum of the shortest distance of its constituent vertices:

$$d(p) = \sum_{i=1}^{k} d(v_{i-1}, v_i) \tag{1}$$

We define the length of an optimal free-of-charge path $\rho(u, v)$ from u to v by

$$\rho(u, v) = \begin{cases} min\{d(p) : u \rightarrow v\} & \\ \infty & \text{if no free-of-charge path.} \end{cases} \tag{2}$$

Definition 3 (Optimal free-of-charge path): An optimal free-of-charge path from vertex u to v is defined as any free-of-charge paths p with length $d(p) = \rho(u, v)$.

In this paper, we focus on finding an optimal free-of-charge path within the maximum detour distance.

III. Map Generalization

Given a road graph $G(V, E)$, we can adopt Dijkstra's algorithm to find a shortest path for the given vertices u and v. On a bicycle trip, the source and destination vertices are always bicycle stations. We are interested only in bicycle stations that have been passed. It also can be observed that the number of bicycle stations is much smaller than the number of vertices representing intersections. Thus if we can construct a station graph $G'(V', E')$, and prove that the route derived from this station graph is identical to the path from the original road network, then we can reduce the running time of Dijkstra's algorithm from $O(|V|^2)$ to $O(|V'|^2)$, where $|V| \gg |V'|$.

Definition 4: (Neighboring stations). Given two vertices u and w, where $u, w \in V_b$, if there is a shortest path $p = \langle u, v_0, v_1, ..., v_k, w \rangle$ from u to w, and $v_i \in \{V - V_b\}$, for any $0 \leq i \leq k$, we say u and w are neighboring station vertices.

Definition 5: (Station graph). A station graph is a connected and directed planar graph $G'(V', E')$. All vertices in the graph are bicycle stations. If u and v are neighboring, then edge $(u, v) \in E'$. If u and v are not neighboring, then there is no edge between u and v in the network. $G'(V', E')$ is a weighted graph, and each edge has an associated weight. The weight of edge (u, v) is the length of a shortest path p between u and v, and is denoted as $d(u, v)$.

Constructing the station graph involves finding all neighboring stations in a graph. We must find, for every pair of vertices u, v, a shortest (least-weight) path from u to v, if they are neighboring according to Definition 4, where the length of a path is the sum of the weights of its constituent edges in G. With this result, we can draw a graph of the station network.

The problem of neighboring stations can be solved by running a single-source shortest-path algorithm $|V'|$ times, once for each station as the source.We observed that the station graph constructed on top of a road network is sparse. That is to say, in general, a station neighbors only several other stations. Running a complete Dijkstra's algorithm for every pair of stations is not necessary. In this paper, we introduce Algorithm 1, an improved Dijkstra's algorithm, which —while searching for a shortest path to its destination —stops searching when it comes across a station.

Algorithm 2 computes the all-neighboring-stations problem by leveraging Algorithm 1 as a subroutine. Algorithm 2 assumes implicitly that the graph is stored in adjacency lists. The algorithm returns the usual $|V'| \times |V'|$ matrix $D = \{d_{ij}\}$, where d_{ij} represents the length of a shortest path.

Algorithm 1 Neighboring-Stations(G, ω, s)

```
Input:
    A graph $G$, a weight function $w$ and a vertex $s$;
Output:
    Neighboring stations of $s$;

 1: $s.d = 0$        //Initialization
 2: for each vertex $v \in G.V$ do
 3:     $v.via\_s =$ False
 4:     if $v \in G.Adj[s]$ then
 5:         $v.d = w(s, v)$ //$Adj[u]$ lists adjacent vertices
 6:         $v.\pi = s$
 7:     else
 8:         $v.d = \infty$
 9:         $v.\pi =$ NIL
10:     end if
11: end for
12: $S = \{s\}$
13: $Q = G.V - \{s\}$ //Searching neighbors
14: while $\exists v \in Q$, $v.d \neq \infty$ and $v.via\_s =$ False do
15:     $u =$ Extract-min($Q$)
16:     $S = S \cup \{u\}$
17:     for each vertex $v \in G.Adj[u]$ do
18:         Relax-and-Mark($u, v, \omega$)
            //If passing through a station, set $v.via\_s$ as True.
19:     end for
20: end while
21: return Neighboring stations of $s$;
```

Algorithm 2 Station-Network(G, ω)

```
Input:
    Graph $G$, weight function $\omega$;
Output:
    Neighboring stations stored in matrices $D$;

1: for each vertex $u \in G.V_s$ do
2:     Neighboring-Stations($G, \omega, u$)
3:     for each vertex $v \in G.V_s$ do
4:         $d.d_{uv} = \delta(u, v)$
5:         $d.path_{uv} =$ Print-Path($G, u, v$)
6:     end for
7: end for
8: return $D$;
```

IV. Station Usage Prediction

We focus next on predicting station usage, that is, the definition of function $util(t)$. In particular, we are interested in predicting a station is full or not.

As mentioned in Section II, the analysis presented in this paper is carried out using normalized occupied slots (NOS), unless specified otherwise.

A prediction window (PW) is a period of time t_{PW} that specifies how far into the future to predict. In this paper, we define a long-term prediction as t_{PW} is greater than or equal to one hour, and we define a short-term prediction as t_{PW} is less than one hour.

We are primarily concerned with weekday occupation (Monday to Friday, excluding holidays) of each station. We separate each day into five-minute time slices. Supposing the operation time of a BSS is from 6:00 to 21:00, then there are 180 time slices in total. A D_{usage} is calculated by averaging station data in every time slices. If we use $D_{usage}[t]$ to denote the value of NOS at t time, then $D_{usage}[t+1]$ means the value of NOS is five minutes after t. In this manner, each station obtains a D_{usage}, a curve of its NOS.

Long-term prediction

Obviously, the $D_{usage}[t]$ value will fluctuate from day to day due to different traffic situations, varying weather conditions, and other reasons. Because of this fluctuation, a given $D_{usage}[t]$ value may be atypical. In order to conduct the long-term prediction, or to estimate a typical $D_{usage}[t]$, it is natural to take some sort of average of the values. In this paper, we maintain an average, called Estimated $D_{usage}[t]$ or $E_{usage}[t]$, of the $D_{usage}[t]$ values. Upon the closing of each day, the system obtains a new $D_{usage}[t]$, and we update $E_{usage}[t]$ according to the following formula:

$$E_{usage}[t] = (1-\alpha)E_{usage}[t] + \alpha \times D_{usage}[t] \quad (3)$$

The new value of $E_{usage}[t]$ is a weighted combination of the previous value of $E_{usage}[t]$ and the new value for $D_{usage}[t]$. Such an average is called an exponential weighted moving average (EWMA), because the weight of a given $D_{usage}[t]$ decays exponentially fast as the updates proceed. The larger the value of α, the higher that recent data is weighted. The value of α is obtained from experiments.

Because t_{PW} is greater than or equal to one hour, we simply take the $E_{usage}[t]$ of that station as the prediction.

Short-term prediction

Because the current occupation of each station is known to the predictor, we make a short-term prediction using the current NOS and the long-term predictor. We call the model a historic trend, as seen in the prediction method [1]. The model first computes the difference of E_{usage} at t and $t + t_{PW}$, and then adds it to the current NOS. We define

$$E_{diff}(t, t_{PW}) = E_{usage}[t + t_{PW}] - E_{usage}[t] \quad (4)$$

For a station v_i, with input of:(1) the current time t, (2) the prediction window t_{PW}. (3) the current occupation of each station $v_i.NOS(t)$, (4) the estimated day usage $v_i.E_{usage}(t)$, we make following prediction

$$v_i.util(t+t_{PW}) = \begin{cases} v_i.E_{usage}[t + t_{PW}] \text{ if } t_{PW} \geq 1 \text{ hour} \\ v_i.NOS(t) + v_i.E_{diff}(t, t_{PW}) \end{cases} \quad (5)$$

Though the prediction model introduced here is simple, it is important to note that there are many other prediction methods available [1]–[5]. We do not conduct comparative studies of these methods, as that is beyond the scope of this paper. In a real implementation of our whole solution, practitioners can choose an appropriate prediction according to their specific needs.

V. Least-cost path planning

Recall from Sections III and IV that we have obtained $G' = (V', E')$ and function $util(t)$. We have the following definition:

Definition 6: (Time-dependent dynamic graph). A time-dependent dynamic graph is a station graph $G'(V', E')$, where U is a set of positive-valued functions in [0,1]. For every vertex (station) v_i, there is a utilization function $v_i.util(t) \in U$, where t is a time variable in a time domain. A utilization function $v_i.util(t)$ specifies normalized occupied slots at time t.

In this section, an optimal free-of-charge path is referred to as a least-cost path (LCP). And we concentrate on finding a LCP from source to destination in a time-dependent dynamic graph within the preferred detour M_{detour}. We propose two solutions to find such a path. First we introduce a baseline solution called the k-Shortest-LCP which is based on the k-shortest path algorithm. Next, we propose a second solution, a paths-tree heuristic searching algorithm, Heuristic-LCP for short.

In this paper, for the sake of simplicity, we assume that a user is skilled, and can collect and return a bike smoothly. In this way, the time cost in picking up and returning a bike can be ignored in comparison to the time spent on traveling.

A. After Getting a Path: Refinement

In order to simplify the query process, a post-processing step is performed on both solutions. With a path $p = \langle v_0, v_1, v_2, ..., v_k \rangle$ and a period of free-ride time T_{free}, the goal of path-refinement is to find a series of stations in which each station must have free slots and allow a user to borrow a bike. The shortest distance between each successive pair of stations in the series is less than $T_{free} \times Speed$, where $Speed$ is the average speed of a user will take.

The procedure of Path-Refinement consists mainly of a loop. The algorithm examines each station after the last rental in p, to see whether the distance between the stations is greater than $T_{free} \times Speed$, and also to check the station's usage upon its arrival, to ensure there are free slots for a user to return the bike. Upon termination, the algorithm returns Null if there is no such path, or computes the free-of-charge path from the source node v_0 to its destination v_k.

That is, given a path with $(k + 1)$ nodes, to find the free-of-charge path from the source to the destination, the total number of nodes we need to examine over all iterations is k.

B. k-Shortest-LCP

A straight-forward solution for LCP problem is to find a shortest path, and test whether it is free of charge. If the path is not for free, we try the second shortest path. The k-shortest path or the ranking of k-shortest paths is a classical network-optimization problem. In the algorithms available for solving the loopless k-shortest paths problem, the computational upper bound of Yen [6]'s algorithm increases linearly with the value of k. Consequently, in general, Yen's algorithm is more efficient than the others.

The k-Shortest-LCP first finds a k-shortest path from source to destination, then performs path-refinement using Path-Refinement that decides whether the path is free of charge and then refines the path. We begin trying from $k = 1, 2, ..., i$ and do not stop until the difference of the length of the k-shortest path and a shortest path is beyond the user-preferred detour distance.

The k-shortest path algorithm adopted in this paper is from [7], a new implementation of Yen's ranking loopless paths algorithm.

C. Heuristic-LCP

Because there are so many possible paths between the source and the destination, as we saw in the last subsection, searching paths one by one is generally not the best choice. While the k-Shortest-LCP algorithm follows a find-and-test strategy, the performance of the algorithm deteriorates seriously in certain cases. For example, if a free-of-charge path is the longest one satisfying a user's maximum detour, then the algorithm stops only at the last minute. So we need another strategy — one that will find all possible paths, then select the shortest and most feasible one among them.

The Heuristic-LCP algorithm finds all paths first, then constructs a paths-tree, then prunes, and finally gets the path we want.

1) Strategy of the Algorithm:

Step 1 Searching a shortest path

The difference between Heuristic-LCP algorithm and the shortest-path algorithm is that the path it finds is not always the shortest one. The Heuristic-LCP strategy uses Dijkstra's algorithm to first find a shortest path. After finding a shortest path, we run path-refinement to check whether this path is free of charge. If it is, the algorithm is done. If the path is not free of charge, the length of the path serves as a benchmark during further searching of all paths to the destination.

Step 2 Listing all paths

This step lists all paths between the source and the destination within a specified maximum detour.

Because the aim here is to list all possible paths without considering the free-ride time limit, a simple solution is to find a path by running a depth-first search. A depth-first search within the graph explores the edges of the most recently discovered vertex v that still has unexplored edges leaving it. Whenever the search discovers a vertex v during a scan for the adjacency list of an already discovered vertex u, it updates $v.d$, and also sets v's predecessor attribute π to u. Once all of u's edges have been explored, the search backtracks to its predecessor.

Once it updates its distance attribute, the algorithm checks the difference between the benchmark and $v.d$ to see if it is out of the user-specified detour M_{detour}. That is, the algorithm excludes the path for which the length is beyond a certain detour distance. We find only loopless paths.

In this step, we also examine whether a vertex is the destination. If it is, then the step stores the corresponding path in a set P of all possible paths.

Step 3 Constructing a paths-tree S

Now we have all paths between a pair of given source and destination, and the lengths of these paths are within the user's detour limit. Let set $P = \{p_1, p_2, ..., p_m\}$ be the set of these paths. Path $p_i = \langle s, v_0, v_1, ..., v_k, d\rangle$ is one of them.

We set tree S as initially empty. Then we examine every path p_i in P.

First, we check the first edge $\langle s, v_0\rangle$, to see whether it belongs to S. Then we look at edge $\langle v_0, v_1\rangle$, to check the subpath of $p_i\langle s, v_0, v_1\rangle$ belongs to S. This process repeats until it has discovered an edge $\langle v_{i-1}, v_i\rangle$, and its corresponding subpath of $p_i\langle s, v_0, v_1, ..., v_i\rangle$ does not belong to S.

After finding the first vertex v_i, the subpath of p_i from s to v_i does not belong to S, we merge the subpath of p_i from v_i to v_k to S.

The entire process loops until all paths in P are added to paths-tree S.

Step 4 Pruning S

In this step, we perform pruning according to the maximum free-ride time.

From the root of S, we start to examine each vertex along the path. Suppose we have a vertex u, we first test its usage $u.util(t)$. If the usage of u is equal to 1, then that means there are no slots free for returning the bike to this station. We update the user's remaining free-ride time to vertex u. If the usage of u is less than 1, then that means there are free slots available for transfer. We update the maximum free-ride time to vertex u. Then the examination continues.

This examination is performed in a depth-first search manner once again. Whenever we meet a vertex for which the remaining free-ride time is less than zero, we stop and prune the branch connected with v.The pruning process ends when it has explored the whole tree. At last, we get S'.

Step 5 Extracting the optimal path

Because each path p in S' has its distance attribute $p.d$, we can visit every leaf of S' and return the path with the least distance. Finally, we run path-refinement to recommend the series of transfer stations along the optimal path towards the destination.

Algorithm 3 Heuristic-LCP(G, s, d, t, M_{detour})

```
Input:
    Station network $G$, source $s$ and destination $d$, time $t$, maximum detour $M_{detour}$;
Output:
    Path from $s$ to $d$;

$path$ = Dijkstra($G, s, d$)
$p$ = Path-Refinement($t, path$)
if $p$!= Null then
    return $p$
end if
$s'$ = NewNode();   $s'.node_G = s$ //Initialization
Insert($S, s'$);   Enqueue($Q, s'$)
while $Q \neq \emptyset$ do
    $u'$ =Dequeue($Q$)
    if $u'.node_G = d$ then
        $p$ = Path-Refinement($t$,ExtractPath($S, s', u'$))
        if $p$!= Null then
            return $p$
        end if
    end if
    $Parent$ =ParentS($u'$)
    for each vertex $v \in G.Adj[u'.node_G]$ do
        if $v \in Parent$ then
            $Continue$
        end if
        $v'$ =NewNode();   $v'.node_G = v$
        $v'.\pi = u'$
        $v'.d = u'.d + G.\omega(u'.node_G, v)$
        if $(v'.d - path.d) < M_{detour}$ then
            Update $v'.rem$ according $v'.node_G.util(t + v'.d/Speed)$,
            If it's less than zero, then $Continue$
            Insert($S, v'$);   Enqueue($Q, v'$)
        end if
    end for
end while
return Null
```

2) The Heuristic-LCP Algorithm:

The strategy introduced in the last subsection is workable. But the strategy finds all paths within the user-specified detour distance, possibly including paths that do not satisfy the free-ride time limit. Finding these paths and inserting them into S are not desirable operations.

We need to develop a solution that is simplified and that also includes the necessary operations of steps 2, 3 and 4.

The procedure Heuristic-LCP (Algorithm 3) works as follows. Lines 1-5 compute a shortest path, and then test whether it is free of charge. If it is not, the distance serves as a benchmark in later steps.

In our implementation, we used a paths-tree S to store

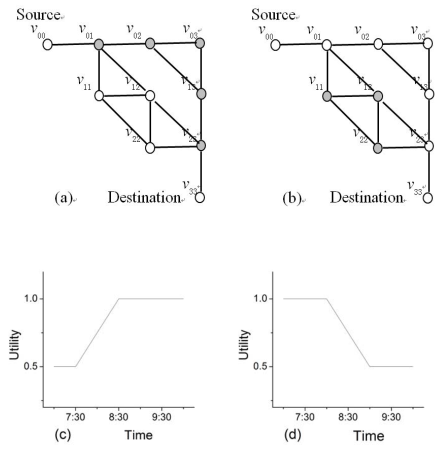

Figure 1. Time dependent dynamic graph and functions.

vertices explored during the searching procedure. We also used a min-priority queue Q of vertices in S to assist during searching, keyed by their d values, that is, their distance from source. Lines 6-7 create a node, correspond its attribute $node_G$ with its original vertex in graph G, and insert it into tree S and queue Q.

Each time through the while loop of lines 9-29, lines 9-15 extract a vertex u' from Q, check whether it is the destination, and construct a vertex set $Parent$ of its parent nodes in S. Then lines 17-29 go through each of its adjacent vertices in G. For each adjacent vertex v, if it does not belong to parent nodes in S, it is copied. Attributes of the new vertex are updated. That is, set attribute $node_G$ to v, set π to u' and update d by adding $u'.d$ with $\omega(u'.node_G, v)$. If the distance between s and v is less than the detour limit, we examine its utility, and update the remaining free time of a user at vertex v'. If a vertex's remaining free time is less than zero, the loop continues. Then finally, lines 27 insert new vertex v' into S and Q. Lines 9-29 obviously combine steps 2-4 introduced in last subsection together.

The algorithm returns a path when the destination has been found and returns Null otherwise.

3) Running Example:

In this subsection, we examine an example to acquire a deeper understanding of the algorithm.

Consider a query $LCP(v_s, v_d, t, M_{detour})$, over G (Figure 1). Figure 1(a) shows a graph with ten stations' vertices before 8:00AM. Figure 1(b) shows the same graph after 8:30AM. The $util$ function for vertices v_{11}, v_{12}, and v_{22}, is shown in Figure 1(c). The $util$ function for vertices $v_{01} - v_{03}$, v_{13}, and v_{23}, is shown in Figure 1(d). The inputs for Algorithm 3 are G, $v_s = v_{00}, v_d = v_{33}$ and t=8:00 . It is easy to find that the shortest path between v_{00} and v_{33} is $\langle v_{00}, v_{01}, v_{12}, v_{23}, v_{33}\rangle$. The length of this path is 4.82. Suppose $T_{free} \times Speed = 2$, that is the free-of-charge distance for a user with speed $Speed$. The subpath $\langle v_{01}, v_{12}, v_{23}\rangle$ of a shortest path does not ever satisfy this condition. So this shortest path is not free of charge.

The algorithm now begins to list all paths between v_{00} and v_{33}. We get the following paths:

$p_1 = \langle v_{00}, v_{01}, v_{02}, v_{03}, v_{13}, v_{23}, v_{33}\rangle, d = 6$
$p_2 = \langle v_{00}, v_{01}, v_{02}, v_{13}, v_{23}, v_{33}\rangle, d = 5.41$
$p_3 = \langle v_{00}, v_{01}, v_{12}, v_{11}, v_{22}, v_{23}, v_{33}\rangle, d = 6.82$
$p_4 = \langle v_{00}, v_{01}, v_{12}, v_{23}, v_{33}\rangle, d = 4.82$
$p_5 = \langle v_{00}, v_{01}, v_{12}, v_{22}, v_{23}, v_{33}\rangle, d = 5.41$
$p_6 = \langle v_{00}, v_{01}, v_{11}, v_{12}, v_{22}, v_{23}, v_{33}\rangle, d = 6$
$p_7 = \langle v_{00}, v_{01}, v_{11}, v_{22}, v_{23}, v_{33}\rangle, d = 5.41$

Let us set M_detour to 1. In this case, the paths of length longer than 5.82 will be excluded as candidates. Under this condition, because the lengths of paths p_1, p_3 and p_6 are 6, 6.82 and 6 respectively, they are not included in the resulted paths-tree S.

Because the algorithm takes t=8:00 as input, as Figure 1 indicates, stations v_{11}, v_{12}, v_{22} turn out to be status full at 8:30. When a user arrives at v_{11}, v_{12} and v_{22}, there is no free slot available to return the bike. Paths p_4, p_5, p_7 are infeasible. Finally, we select the shortest one: p_2. For t=8:00, the algorithm finds a route $\langle v_{00}, v_{01}, v_{02}, v_{13}, v_{23}, v_{33}\rangle$. After refinement, the path turns into $\langle v_{00}, v_{02}, v_{13}, v_{33}\rangle$.

4) Correctness:

Lemma 1: Given a weighted, directed graph $G(V, E)$, and providing an optimal path from source s to destination d, which satisfies M_{detour} limit, before termination of Algorithm 3, there is always a vertex u in Q with the following property:

1. u is on an optimal path to d
2. Algorithm 3 has found a path to d

Proof: At the beginning of lines 7-23, s is in Q, s is on an optimal path to d, and the algorithm has found the path.

Suppose path $p = \langle s, v_1, v_2, ..., v_k, d\rangle$ is an optimal path towards d.

We assume the lemma is true when vertices before $v_i (0 < i < k)$ have been expanded. And we need to prove it is true when we are expanding v_i.

When v_i is selected for expansion, one more of its successors (one not appearing in its predecessors) will be put in Q, and at least one of them, v'_{i+1} will be on an optimal path p to d. (because path p is hypothesized to pass through v_i). Thus v'_{i+1} is added to Q. Algorithm 6 has found an optimal path to d. So, we can let v'_{i+1} be the $(i + 1)$ vertex on p.

Now we have proved all steps prior to the termination of Algorithm 3, and we have found an optimal path p in Q. ■

Theorem 1: Given a weighted, directed graph $G(V, E)$, and providing an optimal path from source s to destination d, which satisfies M_{detour} limit, Algorithm 3 will terminate with an optimal path.

Proof: First, Algorithm 3 must terminate. If it does not terminate, it continues to expand the vertex in Q forever, and increase the d attribute of each vertex in Q infinitely, until finally it exceeds the limit of M_{detour}, and thus cannot be queued within Q.

Second, the algorithm will return an optimal path. Since there is an optimal path, Q will include that path. If Algorithm 3 terminates at line 7 (when Q is empty), but returns without an optimal path, then it is in conflict with Lemma 5.1. This completes the proof of the theorem. ■

5) Analysis:

Algorithm 3 runs Dijkstra's algorithm first. It takes a total time $O(|V|^2)$, and achieves a total running time *O(*$(|V|lg|V|+|E|)$ if the graph is sufficiently sparse. Next the algorithm runs path-refinement, for which the running time depends on $\overline{|p|}$, the average length of the found paths.

In the worst case, we can suppose M_{detour} and free-ride time are unlimited. Algorithm 3 will list all paths between the source and the destination. As a result, Algorithm 6 requires the generation and storage of a paths-tree, the size of which is exponential in the depth of the deepest destination node. In more precise terms, Algorithm 3 has $O(n)$ time complexity, where n is the number of nodes generated. In other words, Algorithm 3 has $O(B^d)$ time complexity, when B is an effective branching factor for the paths-tree and d is the depth of the deepest destination node.

Though it has $O(B^d)$ time complexity, we can do nothing about d. In a real implementation of Algorithm 3, we can obtain a small B by offering the user limited choices of M_{detour}. That is, we can direct the searching process sharply toward the destination with a small M_{detour} limit.

VI. Experiments

Road network: We exported the map of Hangzhou from OpenStreetMap (http://www.openstreetmap.org). We extracted a road network from the original map for experimental purpose in this paper.

Bike usage data: The staff of REDBike system kindly provided us with use of their records from April 29 to May 24, 2014, and all stations' geo-locations. In total, we accessed 6.2 million rental records from 1,480 stations in Hangzhou.

Other settings: All algorithms are implemented in Java and tested on a Windows platform with two Intel Xeon E7-4807 CPUs and 32 GB memory.

A. Map Generalization

Using Algorithm 1-2, we removed some road nodes, and aggregated road segments into edges between adjacent vertices according to our definition. Generalization preserves the relationships between all bike stations and simplifies the original map.

The road network of Hangzhou contains 114,639 nodes and 152,046 segments. In a generalized station graph, the number of nodes is 1,480 and the number of edges is 10,735. The storage ratio of nodes between the generalized map and the original map is 0.01. The storage ratio of edges decreases to 0.07.

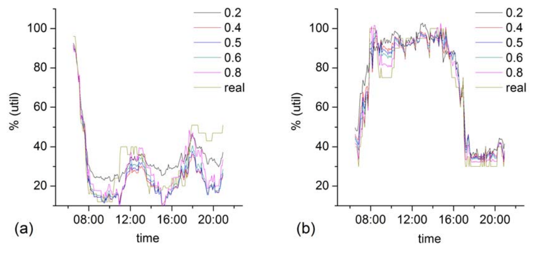

Figure 2. Influence of different α

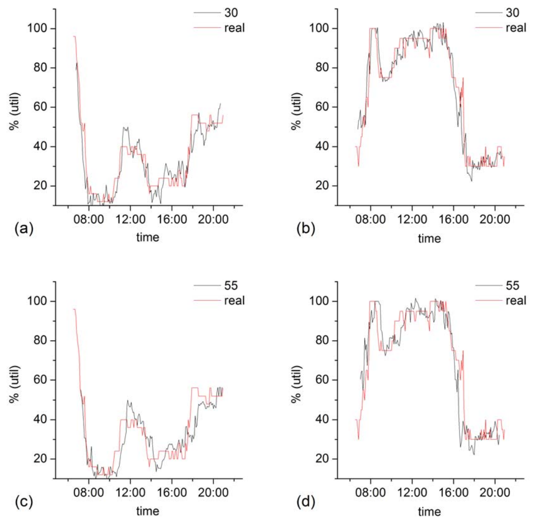

Figure 3. 30-minutes and 55-minutes prediction

B. Prediction Evaluation

The predictor used two weeks of historic data, from May 5 to 16, 2014 to build the model. We use the next five weekdays, from May 19 to 24, 2014 to evaluate the model. From 6:00 AM on May 19, the model was fed the current time and the real records at that moment in five-minute increments. The prediction error was computed as the normalized absolute difference between the prediction and the real observation at that time.

Figure 2(a) and 2(b) compare the accuracy of the EWMA under different α values in uptown and downtown stations.

We randomly chose 10 stations in uptown and downtown, and conducted EWMA prediction. The results show an α value that is too large or too small causes more errors. Subsequently, in later experiments, we chose an α value equal to 0.5. The prediction error for this setting is about 12% in both the downtown and uptown cases, which is close to results reported by other researchers.

For the short-term prediction, as shown in Figures 3, the historic trend model performs better than EWMA does. But once the window was enlarged, the results deteriorated

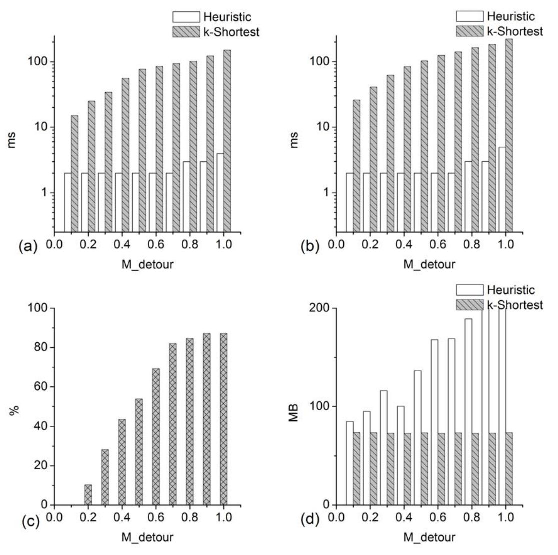

Figure 4. Influence of different M_{detour}

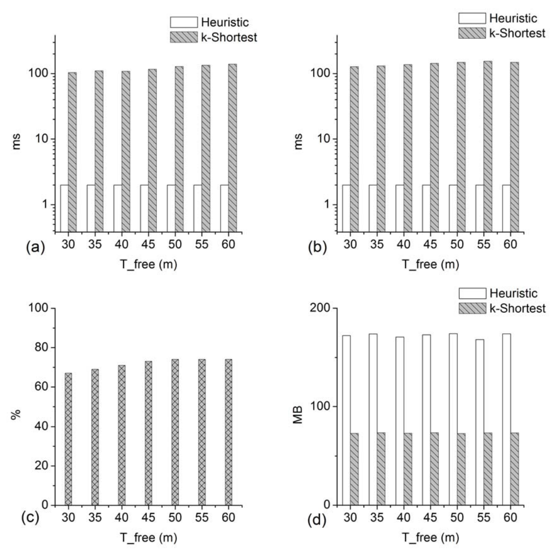

Figure 5. Influence of different T_{free}

quickly. We decided to make 60 minutes the cut-off point between long-term and short-term prediction.

C. Influence of Different M_{detour}

In this experiment, we set $Speed$ to 8 km/hour, set T_{free} to 30 minutes, and 80% of stations were underutilized. We ran Algorithm 3 using different M_{detour} values. We randomly chose 20 sources and 20 destinations. The running time, success rate and memory used are shown in Figure 4.

Figure 4(a) shows the processing time of the queries that were successful. Because the M_{detour} increased during each experiment, the search range got larger. That means with a larger M_{detour} value, the algorithm searches more vertices. As expected, the running time increased while M_{detour} increases. We noticed, however, that the running time of Heuristic-LCP increased less than the k-Shortest-LCP. Figure 4(b) shows the processing time of the queries which were unsuccessful. Because an unsuccessful query explores more vertices, the running times in Figure 4(b) are little more than those in Figure 4(a). In both figures, the time cost of Heuristic-LCP is far less than that of the k-Shortest. This demonstrates that the strategy adopted by Heuristic-LCP is more practicable than the find-and-test strategy adopted by the k-Shortest-LCP.

Figure 4(c) shows the success rate of different M_{detour} values. It shows that increasing M_{detour} values increase the success rate of finding a free-of-charge path in both algorithms. Figure 4(d) shows the maximum memory cost in both algorithms. Though they use the same generalized map, the memory cost of Heuristic-LCP increases dramatically while M_{detour} increases. Heuristic-LCP uses a paths-tree during searching, which requires additional memory to follow the optimal path towards the destination. We expect to decrease the memory cost of Heuristic-LCP in the future by implementing it with a more optimal data structure. The memory cost of the k-Shortest-LCP is less than that of Heuristic-LCP, as shown in Figure 4(d).

D. Influence of Different T_{free}

In this experiment, we explored the influence on performance of different T_{free} values. We set M_{detour} to 0.6, and ran the algorithm. Figure 5(a) and 5(b) show the processing time of the queries that were successful and unsuccessful. We plotted the success rate and memory cost in Figure 5(c) and 5(d). It seems there is little difference between different T_{free} values ranging from 30 to 60 minutes. We believe the reason behind this result is that the distance between most of the adjacent stations is less than a user can cycle within the free-ride time. This is true, as we know that the average distance of adjacent stations is about 1,000 meters in our map, far less than a user can cycle in 30 minutes.

The success rate increases slightly as shown in Figure 5(c), and keeps stable after certain T_{free} values. The memory cost shown in Figure 5(d) is similar to that in the last subsection.

E. Evaluation with a Real Dataset

In a typical weekday, 20-40% of stations in the downtown area are full, and most of the stations in uptown are not full or they have free slots. We set the function $util(t)$ with history records from May 20, 2014, and conducted experiments with different distances between sources and destinations. T_{free} was set to 30 minutes, M_{detour} was set to 0.6 and t was set to 8:00-9:00.

In both successful and unsuccessful cases as shown in Figure 6(a) and 6(b), the further the distance between the

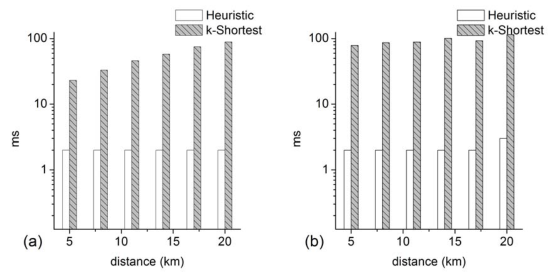

Figure 6. Evaluation in a real environment

origin and the destination, the greater the execution time of the k-Shortest-LCP is. But for Heuristic-LCP, it has a consistent performance. As we did in Subsections C and D, we plotted the running time in a log10 manner in order to show two numbers with a great gap. So, the difference between the different route distances in Figure 6(b) is not obvious as it is in Figure 6(a). Another reason for this is that the algorithm must search all possible routes within the M_{detour} limit. The success rate is about 70%, the same as we saw, in the last subsection.

VII. Related Work

Bike-sharing systems have been deployed in more and more cities in recent years, starting from Europe and quickly spreading to the entire world [8]. Due to the popularity of bike-sharing, the management of bike-sharing systems has been a hot topic within the research community. Research in this area has mainly focused on system operation [3], the analysis of cyclic mobility pattern [1], [2], and short-term prediction of bike availability.

Froehlich et al. [1] implemented four simple predictive models to predict the availability of bicycles at each station. In [2], an Auto-Regressive Moving Average (ARMA) model — taking into account the recent history, the current station, and its closest surrounding stations — was used to predict bicycle availability. Vogel et al. [3] conducted time-series analysis to forecast bike demand. Li et al. [4] considered that bike traffic is impacted by multiple complex factors and proposed a hierarchical prediction model. Gast et al. [5] introduced a predictive model of availability in BSS based on queuing theory.

Classic routing approaches [9] were proved to be effective in finding a shortest path between origin and destination. Speed-up techniques [10] also have been developed. When coming to the routing problem, we always consider graph $G = (V, E)$, with arc functions $\omega : E \to R$, representing the distance between two vertices on ends, the travel time on the arc, or the energy consumption of battery-operated Electric Vehicles(EVs). The distance of an arc is fixed, but travel time and energy consumption are dynamic due to congestion and road environment.

The need to find shortest paths over a large graph, where the weights of each edge dynamically change overtime, is becoming more critical due to increasing traffic problems. Dehne et al. [11] introduced an algorithm for time-dependent networks where the availability of links and the cost of using a link is given by a non-decreasing linear function. Wu et al. [12] explored reachability and time-based path queries in a temporal graph. Wang et al. [13] presented a labeling strategy, an efficient indexing technique for route planning on a timetable graph.

When function is connected with energy consumption on an edge, it leads to a charging stop problem for battery-operated electric vehicles [14], [15].

In this paper, we consider $G' = (V', E')$, with usage functions $U : V' \to [0, 1]$, representing usage of each vertex in V'. And the traveler is subject to a maximum free-ride time. It is a specialised routing problem in a time-dependent dynamic graph. As a result, it is different from the routing problems introduced above.

VIII. Conclusion

In this paper, we studied how to perform a query $LCP(v_s, v_d, t, M_{detour})$ in a time-dependent dynamic graph $G' = (V', E')$. We proposed two algorithms: k-Shortest-LCP and Heuristic-LCP to find the optimal path. We conducted extensive studies with synthesized and real datasets, and we confirmed that Heuristic-LCP — which constructs a unique paths-tree to facilitate the searching process — can acquire query results more efficiently.

A future work we would like to explore might be a comparative study of different prediction methods of station usage, and the deployment of our algorithms in an actual application environment.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (No. 61572165), the Natural Science Foundation of Zhejiang Province (No. LZ15F 020003) and the Public Projects of Zhejiang Province (No. 2013C33082). Jianliang Xu's work is supported by Hong Kong Research Grants Council (No. 12201615 and 12244916).

References

[1] J. Froehlich, J. Neumann, and N. Oliver, "Sensing and predicting the pulse of the city through shared bicycling," in *Proceedings of the 21st IJCAI*, 2009.

[2] A. Kaltenbrunner, R. Meza, J. Grivolla, J. Codina, and R. Banches, "Urban cycles and mobility patterns: Exploring and predicting trends in a bicycle-based public transport system," *Pervasive and Mobile Computing*, vol. 6, no. 4, pp. 455–466, 2010.

[3] P. Vogel and D. C. Mattfeld, "Strategic and operational planning of bike-sharing systems by data mining: A case study," in *Proceedings of the Second ICCL*, ser. ICCL'11, Berlin, Heidelberg, 2011, pp. 127–141.

[4] Y. Li, Y. Zheng, H. Zhang, and L. Chen, "Traffic prediction in a bike-sharing system," in *Proceedings of the 23rd SIGSPATIAL*, ser. GIS '15, 2015, pp. 33:1–33:10.

[5] N. Gast, G. Massonnet, D. Reijsbergen, and M. Tribastone, "Probabilistic forecasts of bike-sharing systems for journey planning," in *The 24th ACM CIKM*, 2015.

[6] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[7] E. Q. Martins and M. M. Pascoal, "A new implementation of yens ranking loopless paths algorithm," *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, vol. 1, no. 2, pp. 121–133, 2003.

[8] P. DeMaio, "Bike-sharing: History, impacts, models of provision, and future," *Journal of Public Transportation*, vol. 12, no. 4, pp. 41–56, 2009.

[9] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[10] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, "Route planning in transportation networks," *ArXiv eprints*, 2015.

[11] F. Dehne, M. T. Omran, and J.-R. Sack, "Shortest paths in time-dependent fifo networks," *Algorithmica*, vol. 62, no. 1-2, pp. 416–435, 2012.

[12] H. Wu, Y. Huang, J. Cheng, J. Li, and Y. Ke, "Reachability and time-based path queries in temporal graphs," in *Proceedings of the 32rd ICDE*, 2016.

[13] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *Proceedings of the 2015 ACM SIGMOD*. ACM, 2015, pp. 967–982.

[14] M. Baum, J. Dibbelt, A. Gemsa, D. Wagner, and T. Zündorf, "Shortest feasible paths with charging stops for battery electric vehicles," in *Proceedings of the 23rd SIGSPATIAL*, ser. GIS '15, 2015, pp. 44:1–44:10.

[15] S. Storandt, "Quick and energy-efficient routes: computing constrained shortest paths for electric vehicles," in *Proceedings of the 5th SIGSPATIAL International Workshop on Computational Transportation Science*. ACM, 2012, pp. 20–25.