Concise Caching of Driving Instructions

Jeppe Rishede Thomsen Man Lung Yiu
Hong Kong Polytechnic University
{csjrthomsen,csmlyiu}@comp.polyu.edu.hk

Christian S. Jensen
Aalborg University
csj@cs.aau.dk

ABSTRACT

Online driving direction services offer fundamental functionality to mobile users, and such services see substantial and increasing loads as mobile access continues to proliferate. Cache servers can be deployed in order to reduce the resulting network traffic. We define so-called concise shortest paths that are equivalent to driving instructions. A concise shortest path occupies much less space than a shortest path; yet it provides sufficient navigation information to mobile users. Then we propose techniques that enable the caching of concise shortest paths in order to improve the cache hit ratio.

Interestingly, the use of concise shortest paths in caching has two opposite effects on the cache hit ratio. The cache can accommodate a larger number of concise paths, but each individual concise path contains fewer nodes and so may answer fewer shortest path queries. The challenge is to strike a balance between these two effects in order to maximize the overall cache hit ratio. In this paper, we revisit two classes of caching methods and develop effective caching techniques for concise paths. Empirical results on real trajectory-induced workloads confirm the effectiveness of the proposed techniques.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

General Terms

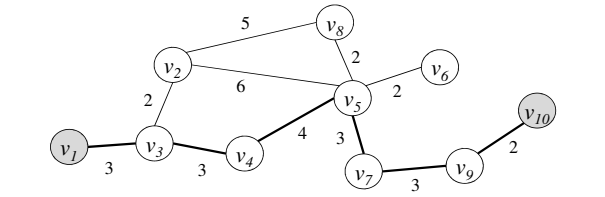
Algorithms, Performance

Keywords

road networks, shortest paths, caching

1. INTRODUCTION

Navigation services offer step-by-step navigation, which provide a driver with *driving instructions*[1] based on the driver's current GPS location [7]. We illustrate such driving instructions in Figure 1. Suppose that the driver starts at node v_1 and the target is node v_{10}. First, a detailed model uses point-by-point instructions, which essentially form the shortest path: $SP_{1,10} = \langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10}\rangle$. In contrast, a concise model uses turn-by-turn instructions that are necessary for navigation, e.g., $\langle v_1, \underline{\Uparrow} v_5, \underline{\overset{\uparrow^{90}}{\frown}} \underline{\Uparrow} v_{10}\rangle$. These instructions combine a sequence of checkpoints (e.g., v_1, v_5, v_{10}) and turn instructions (e.g., $\underline{\Uparrow}$, $\underline{\overset{\uparrow^{90}}{\frown}}$). After leaving v_1, the user should continue (indicated by $\underline{\Uparrow}$) until reaching a checkpoint (v_5). Finally, the user should make a right turn ($\underline{\overset{\uparrow^{90}}{\frown}}$) and then continue ($\underline{\Uparrow}$) until reaching the target (v_{10}).



Shortest path (point-by-point):	$\langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10}\rangle$
Driving instructions (turn-by-turn):	$\langle v_1, \underline{\Uparrow} v_5, \underline{\overset{\uparrow^{90}}{\frown}} \underline{\Uparrow} v_{10}\rangle$
Concise shortest path:	$\langle v_1, v_5, v_7, v_{10}\rangle$
Generic concise shortest path:	$\langle v_1, \mathbf{v_4}, v_5, v_7, v_{10}\rangle$

Figure 1: A road network, with a query from v_1 to v_{10}

Mobile users are increasingly using online driving direction services (rather than offline navigation software) as they are free of charge and do not require purchase and installation of up-to-date map data on mobile devices. Examples of driving direction APIs include Google Directions [1] and MapQuest Directions [2]. They can provide users with both shortest paths and turn-by-turn driving instructions.

The above services receive intensive workloads from mobile users on a daily basis. They can deploy web cache servers in order to reduce the network traffic [3, 19]. In our application context, we propose to cache driving instructions at web cache servers. For the sake of cache management, we represent concise driving instructions (with checkpoints and turn instructions) according to a unified format called *concise shortest path*. By replacing turn instructions by nodes, we obtain the following concise shortest path for our example: $CP_{1,10} = \langle v_1, v_5, v_7, v_{10}\rangle$. The default driving instruction is to go straight until reaching the next node in the concise shortest path (e.g., from v_1 to v_5). Upon reaching such a node (v_5), the driver checks whether the next such node (v_7) is adjacent; if yes, the user makes a turn there. We elaborate on the construction of and navigation using such paths in Section 4. Figure 2 shows examples of a shortest path SP (crosses) and the corresponding concise shortest path CP (dots).

The use of concise shortest paths in caching has two opposite ef-

[1]These instructions can be spoken or given visually.

SIGSPATIAL'14, November 04 - 07 2014, Dallas/Fort Worth, TX, USA

http://dx.doi.org/10.1145/2666310.2666401.

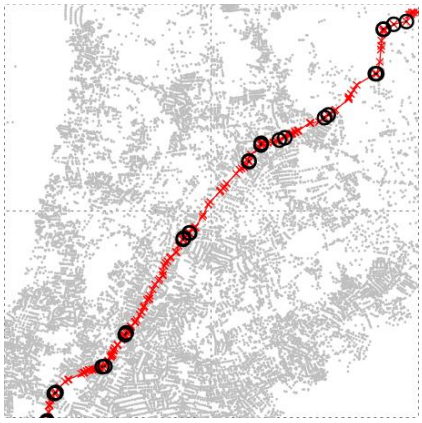


Figure 2: Shortest path (crosses) and concise shortest path (circles), in New York

fects on the cache hit ratio. Clearly, the cache can accommodate a larger number of concise paths than corresponding complete paths, suggesting a higher cache hit ratio. However, a concise path contains fewer nodes than its corresponding complete path and can answer fewer shortest path queries than the complete path. Consider the concise shortest path $\langle v_1, v_5, v_7, v_{10}\rangle$ in Figure 1. By the optimal subpath property [8], this path can answer any query whose end node lies on the path. Thus, this path can answer $Q_{5,10}$ (the query from v_5 to v_{10}) but not $Q_{4,10}$ (the query from v_4 to v_{10}).

Intuitively, if a query (say, $Q_{4,10}$) is frequent, then it is desirable to include its query nodes into a concise path. We use the term *generic concise shortest path* for a shortest path that extends a concise shortest path to include such frequent nodes. An example is the path $\langle v_1, \mathbf{v_4}, v_5, v_7, v_{10}\rangle$ as shown in Figure 1. Although such a path is slightly less compact than CP, it can answer one more frequent query (e.g., $Q_{4,10}$) than CP. In general, there are $2^{|SP|-|CP|}$ possible instances of generic shortest paths having the same start and end nodes. Which of them should be cached? Our key challenge is to select generic shortest paths such that the overall cache hit ratio is maximized.

Our contributions are:

- We propose the notion of a generic concise shortest path that enables a trade-off between the path size and the number of queries answerable by the path.
- We present a statistics-driven model for selecting generic concise paths in a static caching setting.
- We develop an adaptive technique for determining generic concise paths in a dynamic caching setting.

The outline of the paper is as follows. Section 2 reviews related work, and Section 3 describes the problem setting. Next, we introduce concise shortest paths in Section 4, and we present caching techniques for them in Sections 5 and 6. Then, we evaluate the performance of our proposed techniques in Section 7. Finally, Section 8 offers conclusions.

2. RELATED WORK

Compact representations of shortest paths.

We proceed to review existing work on compressing a shortest path, i.e., representing it with few nodes.

The K-skip shortest path [21] is a path that contains at least one out of every K consecutive nodes in the corresponding shortest path. In Figure 1, the 3-skip shortest path of $SP_{1,10}$ is: $SKIP_{1,10} = \langle v_1, v_5, v_{10}\rangle$. Unfortunately, unlike concise shortest paths, K-skip shortest paths are lossy and may provide ambiguous driving instructions to the user. In practice, during driving, only a small portion of the road network is within the driver's eyesight. For example, when the user reaches v_3, the above 3-skip path $SKIP_{1,10}$ does not contain sufficient information to decide whether to go to v_2 or to v_4.

Batz et al. [6] develop a more effective compression method for shortest paths by using a shortest path index. However, this method requires the introduction of a shortest path index at the client side in order to decode the compressed path correctly. Otherwise, the client cannot obtain unambiguous driving instructions.

Semantic caching.

Caching has been studied extensively for database systems [17] and web search engines [16]. In the traditional caching model, each query requests a specific data item, and the cache only supports exact matches. On the other hand, in the semantic caching model [12, 14, 22, 24], each cached query result (or result set) is associated with a validity range, and it can be used to answer and refine any query that intersects its validity range. Semantic caching has been studied for spatial queries [12,14,24] and for shortest path queries [22].

In previous work [22], we have studied the semantic caching of shortest paths. By the optimal subpath property [8], a shortest path SP can answer any shortest path query $Q_{s,t}$ whose source s and target t both fall into SP. For example, in Figure 1, the shortest path $SP_{1,10} = \langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10}\rangle$ can answer the shortest path query $Q_{4,10}$ as path $SP_{1,10}$ contains both v_4 and v_{10}. We have also developed a caching policy for shortest paths. However, we have not studied concise shortest paths. As we will detail in Section 4.3, we propose the notion of a generic concise shortest path that enables a trade-off between its size and the number of queries it can answer. This characteristic enables new caching techniques that require balancing between the lengths of paths and the hit ratios of paths.

3. PROBLEM SETTING

We first provide definitions for our problem, and then we introduce our objectives. Table 1 provides an overview of the notations used throughout the paper.

Symbol	Meaning
$G(V, E, W)$	a graph
v_i	a node in the node set V
(v_i, v_j)	an edge in the edge set E
$Q_{s,t}$	a shortest path query from node v_s to node v_t
$SP_{s,t}$	a shortest path result of $Q_{s,t}$
$\|SP_{s,t}\|$	the size of $SP_{s,t}$ (in number of nodes)
CP	a concise shortest path
GCP	a generic concise shortest path
Ψ	the cache

Table 1: Table of symbols

DEFINITION 1 (GRAPH). *Let $G(V, E, W)$ be a directed spatial graph with a set V of nodes and a set $E \subseteq V \times V$ of edges. Each node $v_i \in V$ models a road junction and has lat-long coordinates. Each edge $(v_i, v_j) \in E$ models a road segment, and $W{:}E \rightarrow R$ assigns a positive weight to each edge.*

DEFINITION 2 (QUERY AND RESULT). *A shortest path query, denoted by $Q_{s,t}$, consists of a source and a target node, v_s and v_t. The result of $Q_{s,t}$, denoted by $SP_{s,t}$, is a sequence of nodes from v_s to v_t such that it has the smallest sum of edge weights (among all paths from v_s to v_t in G).*

By the optimal subpath property [8], any subpath of a shortest path SP is also a shortest path. Thus, a query $Q_{s,t}$ can be answered by SP if $v_s, v_t \in SP$ and v_s appears before v_t in SP. We denote this event by: $Q_{s,t} \subset SP$. We proceed to define the concept of cache hit.

DEFINITION 3 (CACHE HIT). *Let a cache Ψ be a set of shortest paths. A query $Q_{s,t}$ obtains a hit from Ψ if there exists some $SP \in \Psi$ such that $v_s, v_t \in SP$ and v_s appears before v_t in SP. We denote this event by: $Q_{s,t} \subset \Psi$.*

DEFINITION 4 (CACHE SIZE). *Let $|P|$ denote the size (number of nodes) of a path. The total size of the cache is defined as: $||\Psi|| = \sum_{P \in \Psi} |P|$.*

We illustrate the above definitions with the example in Figure 1. Suppose that a cache contains two shortest paths: $\Psi = \{SP_{1,10}, SP_{2,6}\}$, where $SP_{1,10} = \langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10}\rangle$ and $SP_{2,6} = \langle v_2, v_5, v_6\rangle$. The cache size is: $||\Psi|| = |SP_{1,10}| + |SP_{2,6}| = 7+3 = 10$. Consider the queries $Q_{3,7}$ and $Q_{1,2}$. Query $Q_{3,7}$ obtains a cache hit because the cached path $SP_{1,10}$ contains both query nodes v_3 and v_7. On the other hand, the query $Q_{1,2}$ does not obtain a cache hit.

We adopt the standard client-server architecture shown in Figure 3 as the setting. A cache server is placed in-between the mobile clients and an online shortest path service (e.g., Google Directions [1]). Upon receiving a shortest path query $Q_{s,t}$, the cache server checks whether there is a cache hit. If yes then it returns the result. Otherwise, it forwards $Q_{s,t}$ to the online shortest path service and eventually returns the result path $P_{s,t}$ computed by the service.

We assume that the dominant cost is that of the network traffic between the cache server and the shortest path service. Thus, our objective is to **maximize the hit ratio of the cache server** (subject to a given cache capacity) [16, 17, 19]. While the shortest path service may deploy a shortest path index [4, 11, 20, 26] for its own performance reasons, the introduction of such an index does not improve the cache hit ratio and is orthogonal to our work.

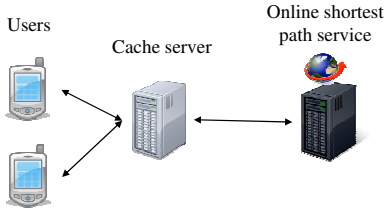


Figure 3: Client server architecture

Unlike previous work [22], this paper utilizes *concise shortest paths* to improve the hit ratio of caching. We elaborate on concise shortest paths and their operations in Section 4, and we present caching methods for them in Sections 5 and 6.

As a remark, two issues are orthogonal to this paper. First, we reuse existing data structures for the caching of shortest paths [22] to reduce the CPU overhead for maintaining cache structures. Second, we assume that the edge weights $W(v_i, v_j)$ maintained at the online shortest path service are independent of time. This is the case for edge weights that model travel distances (e.g., in MapQuest). If this is not the case, we may associate each cache item (shortest path) with a expiry time [9].

4. CONCISE SHORTEST PATHS

For navigation purposes, it suffices for mobile users to know driving instructions (e.g., going straight, making turns) instead of complete shortest paths. As discussed in the introduction, the notion of a concise shortest path is equivalent to driving instructions.

We first provide definitions for concise shortest paths. Then, we propose a server-side algorithm for converting a shortest path into a concise shortest path. Also, we present a client-side algorithm that enables a user to navigate using a concise shortest path. Finally, we introduce generic concise shortest paths.

4.1 Definition and Examples

We consider three possible driving instructions for mobile users, as illustrated in Figure 4. Suppose that the user is reaching a current node v_c from a previous node v_p. In each case in the figure, the shortest path is indicated by bold arrows.

- Case I: The driving instruction is `'continue'` to the next node v_n as this is the only option.
- Case II: There is more than one option (e.g., travel to v_{n1} or to v_{n2}). These options can be distinguished by their deviation angles from the previous travel direction (see Definition 5). In this figure, the deviation angles of (v_c, v_{n1}) and (v_c, v_{n2}) are $\delta_{p,c,n1}$ and $\delta_{p,c,n2}$, respectively. If the correct choice is the one with the smallest deviation angle, like in this figure, the driving instruction is `'go straight, by the smallest angle'`.
- Case III: The correct choice is not the option with the smallest deviation angle. In this case, we must provide an explicit driving instruction: `'turn clockwise/anticlockwise by X`^{o}`'`, where X^o is the deviation angle $\delta_{p,c,n}$ (see Definition 5).

The above driving instructions are not explicitly stored in a concise shortest path. In Section 4.2, we provide a client-side algorithm to reconstruct driving instructions from a concise shortest path, during navigation.

DEFINITION 5 (DEVIATION ANGLE). *Given two adjacent edges (v_p, v_c) and (v_c, v_n), the deviation angle between them is:*

$$\delta_{p,c,n} = |180^o - \measuredangle v_p v_c v_n| \quad (1)$$

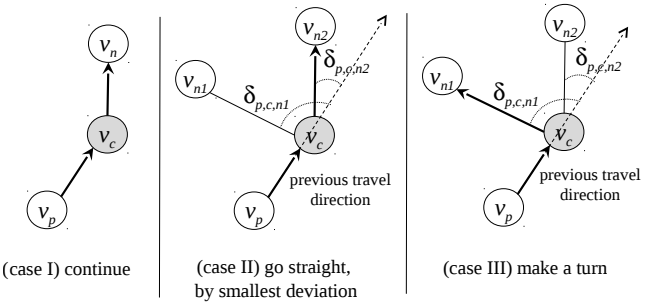
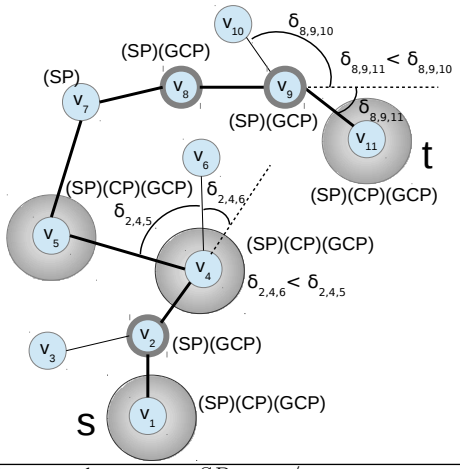


Figure 4: Possible driving instructions; the shortest path is indicated by bold arrows

Based on the above driving instructions, we define a concise shortest path as follows.

DEFINITION 6 (CONCISE SHORTEST PATH). *Let a shortest path $SP_{s,t} = \langle v_{\rho(1)}, v_{\rho(2)}, \cdots, v_{\rho(l)} \rangle$ be given, where $v_{\rho(i)}$ is the i-th node in $SP_{s,t}$, $s = \rho(1)$, and $t = \rho(l)$. A concise shortest path $CP_{s,t}$ is a subsequence of $SP_{s,t}$ such that: (i) $CP_{s,t}$ contains v_s and v_t, and (ii) $CP_{s,t}$ contains $v_{\rho(i)}, v_{\rho(i+1)}$ if nodes $v_{\rho(i-1)}, v_{\rho(i)}$ and, $v_{\rho(i+1)}$ satisfy case III.*

We illustrate a concise shortest path in Figure 5. Let the source and target nodes be v_1 and v_{11}, respectively. The shortest path $SP_{1,11} = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11} \rangle$ is shown with bold edges. According to Definition 6, the concise shortest path is: $CP_{1,11} = \langle v_1, v_4, v_5, v_{11} \rangle$. First, it must contain both the source and target nodes (v_1 and v_{11}). Since $\langle v_2, \mathbf{v_4}, v_5 \rangle$ matches case III, v_4 and the next node (v_5) are in $CP_{1,11}$. The remaining nodes either match case I (e.g., v_7, v_8) or case II (e.g., v_2, v_9), so they are not in $CP_{1,11}$. Observe that a concise path describes the driving instructions for $SP_{s,t}$ unambiguously. Directions are given exactly when it is necessary to change the travel direction.



Shortest path: $SP_{1,11} = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11} \rangle$
Concise shortest path: $CP_{1,11} = \langle v_1, v_4, v_5, v_{11} \rangle$
Generic concise shortest path: $GCP_{1,11} = \langle v_1, \mathbf{v_2}, v_4, v_5, v_{11} \rangle$

Figure 5: Concise shortest path

Concise shortest paths have the advantage that they consume less space while still offering sufficient information for driving.

4.2 Operations

We study caching methods for concise shortest paths in Sections 5 and 6. Given a shortest path SP, the cache server can execute Algorithm 1 to extract a concise shortest path from SP.

First, we specify the initial travel direction (in Lines 3–6). We add the two first nodes of SP to CP if the first node has multiple branches, i.e., $deg(v_{\rho(1)}) > 1$. Otherwise, there is only one option so it suffices to add the first node of SP to CP.

For each subsequent node $v_{\rho(i)}$ on SP, we compute the deviation angle to each adjacent node and then take the smallest deviation angle as δ^*. If the next node $v_{\rho(i+1)}$ on SP does not have the smallest deviation angle, we specify the travel direction by adding the current and the next nodes $(v_{\rho(i)}, v_{\rho(i+1)})$ of SP to SP. Finally, we add the last node of SP to CP.

Upon receiving a concise shortest path CP from the cache server, the client can apply Algorithm 2 to navigate along CP. Note that this algorithm also works correctly for extended versions of CP, which we discuss below.

Algorithm 1 Server:ExtractConcise(Shortest path SP)

```
1:  let v_ρ(i) be the i-th node in SP
2:  CP ← ⟨⟩                                          ▷ the concise path
3:  if deg(v_ρ(1)) > 1 then
4:      append v_ρ(1), v_ρ(2) to CP
5:  else
6:      append v_ρ(1) to CP
7:  for i from 2 to |SP| − 1 do
8:      if deg(v_ρ(i)) > 2 then
9:          δ* ← δ_ρ(i−1),ρ(i),ρ(i+1)                 ▷ deviation angle
10:         for each node v_j adjacent to v_ρ(i) do
11:             if v_j ≠ v_ρ(i−1) and v_j ≠ v_ρ(i+1) then
12:                 δ* ← min{δ*, δ_ρ(i−1),ρ(i),j}
13:         if δ* < δ_ρ(i−1),ρ(i),ρ(i+1) then          ▷ include nodes
14:             append v_ρ(i) to CP if it is not in CP
15:             append v_ρ(i+1) to CP
16: append v_ρ(|SP|) to CP if it is not in CP          ▷ last node
17: Return CP
```

Algorithm 2 Client:NavigateConcise(Concise path CP)

```
1:  let (v_prev, v_cur) be the edge being traversed by the client
2:  while v_cur ≠ v_t do                               ▷ target not reached
3:      if v_cur is not in CP then                     ▷ go straight
4:          δ* ← 180^o                                 ▷ the smallest deviation angle
5:          for each node v_j adjacent to v_cur do
6:              if v_j ≠ v_prev then
7:                  if δ* > δ_prev,cur,j then
8:                      δ* ← δ_prev,cur,j
9:                      v_next ← v_j                   ▷ keep the best option
10:         display "continue to v_next"
11:     else                                           ▷ follow CP
12:         let v_next be the next node of v_cur in CP
13:         display "turn to v_next"
14:     wait until the client reaches v_next
15:     (v_prev, v_cur) ← (v_cur, v_next)
```

4.3 Generic Concise Shortest Paths

Although a concise shortest path occupies less space than a corresponding (complete) shortest path, it covers fewer nodes and thus it may answer a smaller number of queries. For example, suppose that the query $Q_{2,11}$ is a frequently-used query in Figure 5. Observe that the shortest path $SP_{1,11}$ can answer $Q_{2,11}$, while the concise path $CP_{1,11}$ cannot do so as it does not contain v_2.

To address this issue, we consider generic versions of concise shortest paths that include additional nodes:

DEFINITION 7 (GENERIC CONCISE SHORTEST PATH). *A sequence $GCP_{s,t}$ is said to be a generic concise shortest path (from node v_s to node v_t) if it is a subsequence of $SP_{s,t}$ and a supersequence of $CP_{s,t}$.*

As an example, consider the map in Figure 5. Table 2 illustrates all generic concise shortest paths from source v_1 to target v_{11}. Note that the number of generic concise shortest paths is $2^{|SP_{s,t}|-|CP_{s,t}|} = 2^{8-4} = 16$. In the next section, we discuss how to select a generic concise shortest path in order to maximize the cache hit ratio.

5. STATIC CACHING SETTING

Static caching focuses on caching the most popular data items [5, 16, 18]. It utilizes a query log that records past queries in order

$CP_{1,11}$	$\langle v_1, v_4, v_5, v_{11}\rangle$
other generic concise paths	$\langle v_1, v_2, v_4, v_5, v_{11}\rangle$ $\langle v_1, v_4, v_5, v_7, v_{11}\rangle$
	$\langle v_1, v_4, v_5, v_8, v_{11}\rangle$ $\langle v_1, v_4, v_5, v_9, v_{11}\rangle$
	$\langle v_1, v_2, v_4, v_5, v_7, v_{11}\rangle$ $\langle v_1, v_2, v_4, v_5, v_8, v_{11}\rangle$
	$\langle v_1, v_2, v_4, v_5, v_9, v_{11}\rangle$ $\langle v_1, v_4, v_5, v_7, v_8, v_{11}\rangle$
	$\langle v_1, v_4, v_5, v_7, v_9, v_{11}\rangle$ $\langle v_1, v_4, v_5, v_8, v_9, v_{11}\rangle$
	$\langle v_1, v_2, v_4, v_5, v_7, v_8, v_{11}\rangle$ $\langle v_1, v_2, v_4, v_5, v_7, v_9, v_{11}\rangle$
	$\langle v_1, v_2, v_4, v_5, v_8, v_9, v_{11}\rangle$ $\langle v_1, v_4, v_5, v_7, v_8, v_9, v_{11}\rangle$
$SP_{1,11}$	$\langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11}\rangle$

Table 2: Generic concise shortest paths (from v_1 to v_{11})

to determine the access frequencies of data items. Static caching incurs low overhead at runtime; however, it may not adapt well to a varying query distribution.

We present a caching method for generic concise shortest paths that applies an existing static caching policy for shortest paths [22]. The frequencies of queries in the query log are used to define a 'benefit' score that captures the 'importance' of a path in answering queries. Our main challenge is to find generic concise paths that have high 'benefit' scores.

In Section 5.1, we give a benefit score model for generic concise paths. Then, in Section 5.2, we present a method for computing a generic concise path with a high benefit score. Finally, we propose an efficient implementation in Section 5.3.

5.1 Benefit Model

We adopt the benefit model presented in previous work [22] to quantify the importance of a path in answering queries. First, we provide definitions of query log and query frequency.

DEFINITION 8 (QUERY LOG AND FREQUENCY). *A query log $\mathcal{QL}$ is a collection of shortest path queries that have been issued by users in the past.*
The query frequency $\chi_{s,t}$ of a query $Q_{s,t}$ is defined as the number of occurrences of $Q_{s,t}$ in the query log $\mathcal{QL}$.

$$\chi_{s,t} = |\{Q_{s,t} \in \mathcal{QL}\}| \tag{2}$$

Table 3 illustrates the values of $\chi_{s,t}$ derived from an example query log for the map in Figure 5. For simplicity, since we assume $\chi_{s,t} = \chi_{t,s}$.

$\chi_{s,t}$	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}
v_1	/	0	**10**	**50**	0	0	0	**16**	0	0	0
v_2		/	0	0	0	0	**35**	0	0	0	**12**
v_3			/	0	**20**	0	0	0	0	0	0
v_4				/	**2**	0	0	0	0	0	**20**
v_5					/	0	**5**	0	0	**18**	0
v_6						/	0	0	0	0	0
v_7							/	0	0	0	0
v_8								/	0	0	**30**
v_9									/	0	0
v_{10}										/	0
v_{11}											/

Table 3: Example of $\chi_{s,t}$ values for the graph, with $\chi_{s,t} = \chi_{t,s}$

Given a generic concise shortest path P, we define its *benefit* $\Upsilon(P)$ as:

$$\Upsilon(P) = \sum_{v_i, v_j \in P,\ Q_{i,j} \subset P} \chi_{i,j} \tag{3}$$

The benefit is the number of queries (in $\mathcal{QL}$) that can be answered by P. As a remark, a simple (nested-loop) implementation for computing $\Upsilon(P)$ would take $O(|P|^2)$ time.

Then, we define the benefit of a cache Ψ as follows:

$$\Upsilon(\Psi) = \sum_{v_i, v_j \in V,\ Q_{i,j} \subset \Psi} \chi_{i,j} \tag{4}$$

As an example, consider the cache content in Table 4 and the map in Figure 5. The path $CP_{1,11}$ has benefit: $\Upsilon(CP_{1,11}) = \chi_{1,4} + \chi_{1,5} + \chi_{1,11} + \chi_{4,5} + \chi_{4,11} + \chi_{5,11} = 50+0+0+2+20+0 = 72$. The benefit of the cache Ψ is: $\Upsilon(\Psi) = \chi_{1,4} + \chi_{4,5} + \chi_{4,11} + \chi_{3,5} = 50 + 2 + 20 + 20 = 92$. Observe that Equation 4 avoids duplicate counting. For example, the term $\chi_{4,5}$ appears only once in the equation of $\Upsilon(\Psi)$, despite that the query $Q_{4,5}$ can be answered by paths $CP_{1,11}$, $CP_{3,8}$, and $CP_{4,9}$.

	concise path CP
$CP_{1,11}$	$\langle v_1, v_4, v_5, v_{11}\rangle$
$CP_{3,8}$	$\langle v_3, v_4, v_5, v_8\rangle$
$CP_{4,9}$	$\langle v_4, v_5, v_9\rangle$

Table 4: A cache Ψ of concise shortest paths

With the above benefit model, we formulate our static caching problem as follows:

PROBLEM 1 (STATIC CACHING PROBLEM). *Select a set of generic concise shortest paths $\Psi = \{P_{s,t}\}$ such that: (i) the benefit $\Upsilon(\Psi)$ is maximized, and (ii) the cache size $||\Psi||$ is bounded by a given cache capacity value.*

In previous work [22], we require each $P_{s,t}$ to be a shortest path. Here, we consider a much larger search space and allow each $P_{s,t}$ to be a generic concise shortest path.

5.2 Benefit-Based Generic Concise Path

Given a shortest path $SP_{s,t}$, we want to construct a generic concise path $GCP_{s,t}$ such that it has a high benefit and a small size.

Given the paths present in the cache Ψ, we define the *marginal benefit* of a path P with respect to Ψ as:

$$\begin{aligned} \Upsilon_m(P \setminus \Psi) &= \Upsilon(\Psi \cup \{P\}) - \Upsilon(\Psi) \\ &= \sum_{v_i, v_j \in P,\ Q_{i,j} \in P,\ Q_{i,j} \not\subset \Psi} \chi_{i,j} \end{aligned} \tag{5}$$

This notion captures the total frequencies of queries (in $\mathcal{QL}$) that can be answered by P but not by paths in Ψ. Intuitively, it is desirable to cache a path P if the benefit $\Upsilon_m(P \setminus \Psi)$ is high and the size $|P|$ is small. Thus, we define the *normalized benefit* of a path P as: $\overline{\Upsilon_m}(P \setminus \Psi) = \frac{\Upsilon_m(P \setminus \Psi)}{|P|}$.

A brute-force approach is to enumerate all possible generic concise shortest paths (like in Table 2) and then find the one with the highest $\overline{\Upsilon_m}$ value. However, as discussed in Section 4.3, the number of possible generic concise shortest paths is exponential in the path size.

In the following, we present a greedy heuristic solution to solve this problem in polynomial time. Algorithm 3 is the pseudo-code of this solution. It takes a shortest path $SP_{s,t}$ and its corresponding concise path $CP_{s,t}$ as input. We denote BP as the path with the highest $\overline{\Upsilon_m}$ value found so far. It is initialized to $CP_{s,t}$. The set S contains the possible nodes that can be added to BP. In each iteration, we compute the benefit of the path $BP \cup \{v_c\}$ for each v_c (Lines 5–9), and find the node (say v_{best}) with the highest benefit. If the normalized benefit of $BP \cup \{v_{best}\}$ is higher than that of BP, we add v_{best} to BP and repeat the loop. Otherwise, the algorithm terminates.

We illustrate the working of algorithm 3 in Table 5. Here, we are given a shortest path $SP_{1,11} = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11}\rangle$ and

Algorithm 3 Static-Greedy (Shortest path SP, Concise shortest path CP, Cache Ψ)

1: $BP \leftarrow CP$ ▷ the best path found so far
2: $S \leftarrow$ the set of nodes in $SP - CP$
3: $\gamma_{cur} \leftarrow \Upsilon_m(BP \setminus \Psi)$
4: **while** $S \neq \emptyset$ **do**
5: $\quad \gamma_{best} \leftarrow 0$ ▷ the best score in this iteration
6: $\quad$ **for** each $v_c \in S$ **do**
7: $\quad\quad \gamma_c \leftarrow \Upsilon_m((BP \cup \{v_c\}) \setminus \Psi)$
8: $\quad\quad$ **if** $\gamma_c > \gamma_{best}$ **then**
9: $\quad\quad\quad \gamma_{best} \leftarrow \gamma_c$; $v_{best} \leftarrow v_c$
10: $\quad$ **if** $\frac{\gamma_{cur}}{|BP|} < \frac{\gamma_{best}}{|BP|+1}$ **then**
11: $\quad\quad$ remove v_{best} from S
12: $\quad\quad$ insert v_{best} into BP (by the order in SP)
13: $\quad\quad \gamma_{cur} \leftarrow \gamma_{best}$
14: $\quad$ **else**
15: $\quad\quad$ Return BP

its corresponding concise shortest path $CP_{1,11} = \langle v_1, v_4, v_5, v_{11}\rangle$. Note that Table 5a shows only the rows and columns whose nodes fall into path $SP_{1,11}$. Table 5b, shows the running steps of the algorithm. First, we initialize BP to $CP_{1,11}$ and the set S to $\{v_2, v_7, v_8, v_9\}$. Those $\chi_{s,t}$ entries that contribute to the benefit of BP are shaded light-gray (see Table 5a). In iteration 1, we add the best node v_8 to BP because its normalized benefit is higher than that of BP. Those $\chi_{s,t}$ entries that contribute to the additional benefit of BP are shaded medium-gray. In iteration 2, even for the best node v_2, the normalized benefit of $BP \cup \{v_2\}$ is smaller than that of BP. Thus, the algorithm terminates and returns the path $\langle v_1, v_4, v_5, v_8, v_{11}\rangle$.

$\chi_{s,t}$	v_1	v_2	v_4	v_5	v_7	v_8	v_9	v_{11}
v_1	/	0	50	0	0	16	0	0
v_2		/	0	0	35	0	0	12
v_4			/	2	0	0	0	20
v_5				/	5	0	0	0
v_7					/	0	0	0
v_8						/	0	30
v_9							/	0
v_{11}								/

(a) relevant entries of $\chi_{s,t}$

Iteration	Steps	Path
Initialization	$CP = \langle v_1, v_4, v_5, v_{11}\rangle$ $S = \{v_2, v_7, v_8, v_9\}$	$\langle v_1, v_4, v_5, v_{11}\rangle$ $\overline{\Upsilon_m}$=72/4=**18**
(1)	v_2: (72+12)/5 v_7: (72+5)/5 v_8: (72+16+30)/5 v_9: (72+0)/5 Highest: 23.6 > 18	**Add** v_8 $\langle v_1, v_4, v_5, v_8, v_{11}\rangle$ $\overline{\Upsilon_m}$=118/5=**23.6**
(2)	v_2: (118+12)/6 v_7: (118+5)/6 v_9: (118+0)/6 Highest: 21.67 < 23.6	**STOP** $\langle v_1, v_4, v_5, v_8, v_{11}\rangle$

(b) running steps

Table 5: Finding the best generic concise shortest path, for $SP_{1,11} = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_9, v_{11}\rangle$ and $CP_{1,11} = \langle v_1, v_4, v_5, v_{11}\rangle$

As a remark, the above algorithm may not always return the optimal result (i.e., the generic concise path with the highest normalized benefit). Suppose that we add the nodes v_2, v_7, v_8 to $CP_{1,11}$ to form a generic concise path: $P^* = \langle v_1, v_2, v_4, v_5, v_7, v_8, v_{11}\rangle$. According to Table 5a, the normalized benefit of P^* is: $(118 + 35 + 12 + 5)/7 = 24.28$, which is higher than the algorithm's result (23.6).

Time Complexity Analysis.

Let n be the number of nodes in SP. Note that both the sizes of S and BP are upper-bounded by n.

In the while-loop (Lines 4–15), we remove a node v_{best} from S in each iteration, so it has at most n iterations. The for loop (Lines 6–9) has at most n iterations because S contains at most n nodes. In Line 7, each call to $\Upsilon_m((BP \cup \{v_c\}) \setminus \Psi)$ takes $O(n^2)$ time as the path size is at most n. By combining the above, the total running time of the while-loop is $O(n^4)$.

Before the while-loop, Lines 1–2 take $O(n)$ time and Line 3 takes $O(n^2)$ time. Thus, the time complexity of the algorithm is $O(n^4)$.

The computational cost is high, even though the typical path size n is in the order of hundreds on real road networks.

5.3 Efficient Implementation

We proceed to present a more efficient implementation of Algorithm 3. The idea is to identify shared expressions in the calculation and compute such expressions only once, regardless of the path size n ($n = |SP|$).

Recall that it takes $O(n^2)$ time to compute $\Upsilon_m(P \setminus \Psi)$, where P is a subsequence of SP. Consider the scenario that we need the updated γ value after adding a node v_c into P. Fortunately, we can apply Equation 6 to derive $\Upsilon_m((P \cup \{v_c\}) \setminus \Psi)$ from $\Upsilon_m(P \setminus \Psi)$ incrementally. This derivation requires only $O(n)$ time to compute $\sum_{v_j \in P,\, Q_{c,j} \not\subset \Psi} \chi_{c,j}$ because we have one node for v_c and at most n nodes in P.

$$
\begin{aligned}
&\Upsilon_m((P \cup \{v_c\}) \setminus \Psi) \qquad (6)\\
&= \sum_{v_i, v_j \in P \cup \{v_c\},\, Q_{i,j} \not\subset \Psi} \chi_{i,j}\\
&= \sum_{v_i \in P \cup \{v_c\}} \; \sum_{v_j \in P \cup \{v_c\},\, Q_{i,j} \not\subset \Psi} \chi_{i,j}\\
&= \left(\sum_{v_i \in P} \; \sum_{v_j \in P,\, Q_{i,j} \not\subset \Psi} + \sum_{v_i \in P,\, Q_{i,c} \not\subset \Psi} + \sum_{v_j \in P,\, Q_{c,j} \not\subset \Psi} \right) \chi_{i,j}\\
&= \Upsilon_m(P \setminus \Psi) + 2 \cdot \sum_{v_j \in P,\, Q_{c,j} \not\subset \Psi} \chi_{c,j}
\end{aligned}
$$

We propose an efficient implementation in Algorithm 4. For each node v_c in S, we maintain its benefit $\Upsilon_m((P \cup \{v_c\}) \setminus \Psi)$ in an attribute $v_c.\gamma$. We first compute $\Upsilon_m(BP \setminus \Psi)$ at Line 3, then derive $v_c.\gamma$ for each node $v_c \in S$ incrementally.

The while-loop (Lines 6–21) repeats until S becomes empty or the benefit of BP cannot be improved further. In each iteration, we simply find the node (v_{best}) with the highest γ. If the normalized benefit $\overline{\Upsilon_m}$ of $BP \cup \{v_{best}\}$ is better than that of BP, we insert v_{best} into BP. Also, we need to update the value $v_c.\gamma$ for each remaining node $v_c \in S$ (Lines 13–17). By using the idea in Equation 6, we can derive a shared expression for updates (Line 13). We can compute it once before using it to update each $v_c.\gamma$.

Time Complexity Analysis.

We proceed as in the time complexity analysis in Section 5.2. Let n be the number of nodes in SP. Again, both the sizes of S and BP are upper-bounded by n.

Algorithm 4 Static-FastGreedy (Shortest path SP, Concise shortest path CP, Cache Ψ)

1: $BP \leftarrow CP$ ▷ the result
2: $S \leftarrow$ the set of nodes in $SP - CP$
3: $\gamma_{cur} \leftarrow \Upsilon_m(BP \setminus \Psi)$
4: **for** each $v_c \in S$ **do**
5: $v_c.\gamma \leftarrow \gamma_{cur} + 2 \cdot \sum_{j \in BP,\, Q_{c,j} \not\subset \Psi} \chi_{c,j}$
6: **while** $S \neq \emptyset$ **do**
7: $\gamma_{best} \leftarrow 0$
8: **for** each $v_c \in S$ **do** ▷ find the best v_c
9: **if** $v_c.\gamma > \gamma_{best}$ **then**
10: $\gamma_{best} \leftarrow v_c.\gamma$; $v_{best} \leftarrow v_c$
11: **if** $\frac{\gamma_{cur}}{|BP|} < \frac{\gamma_{best}}{|BP|+1}$ **then** ▷ compare $\overline{\Upsilon_m}$
12: remove v_{best} from S
13: $\Delta\gamma \leftarrow 2 \cdot \sum_{j \in BP,\, Q_{best,j} \not\subset \Psi} \chi_{best,j}$ ▷ shared part
14: **for** each $v_c \in S$ **do** ▷ update $v_c.\gamma$
15: $v_c.\gamma \leftarrow v_c.\gamma + \Delta\gamma$
16: **if** $Q_{best,c} \not\subset \Psi$ **then**
17: $v_c.\gamma \leftarrow v_c.\gamma + 2 \cdot \chi_{best,c}$
18: insert v_{best} into BP (by the order in SP)
19: $\gamma_{cur} \leftarrow \gamma_{best}$
20: **else**
21: Return BP

The while-loop (Lines 6–21) has at most n iterations. Within the while-loop, the first for loop (Lines 8–10) takes $O(n)$ time, the summation (Line 13) takes $O(n)$ time, and the second for loop (Lines 14–17) takes $O(n)$ time. Thus, the time complexity of the while-loop is $O(n^2)$.

Before the while-loop, Lines 1–2 take $O(n)$ time, and Lines 3–4 take $O(n^2)$ time. Thus, the time complexity of the algorithm is $O(n^2)$.

6. DYNAMIC CACHING SETTING

We have examined the static caching approach in Section 5. In this section, we adopt the *dynamic caching* [10, 15, 16] approach, which intends to cache the most recently accessed data items. When a cache miss occurs and the cache is full, these policies decide which data item to evict from the cache. For instance, the Least-Recently-Used (LRU) policy evicts the least recently used item in the cache. These policies allow the cache to adapt dynamically to the query workload. However, they incur runtime overhead in maintaining the cache.

In this section, we present a dynamic caching method for generic concise shortest paths. It adopts the LRU policy. When we obtain a shortest path SP (after a cache miss), our main problem is how to compute a short generic shortest path that will be able to answer many queries (in the future). To tackle this problem, we employ two lightweight data structures for maintaining query statistics: (i) a sliding window $\mathbb{W}$ that keeps the most recent $\mathbb{W}_{size}$ queries, and (ii) an array μ that records the frequencies of the query nodes in $\mathbb{W}$.

Algorithm 5 shows the pseudo code for our dynamic caching method. It is invoked when there is a cache miss for a query $Q_{s,t}$. First, we calculate the shortest path SP for $Q_{s,t}$, and then compute a concise shortest path GCP for it. Then, for each node v_i in SP, we add it into the DGCP if v_i is in CP or its frequency in μ is non-zero. Next, we remove the oldest query from the sliding window $\mathbb{W}$ and insert the newest query into $\mathbb{W}$. Then, we update the frequency array μ accordingly. Finally, we apply the LRU policy to update the cache Ψ with the generic concise shortest path.

Algorithm 5 Dynamic-GCP (Query $Q_{s,t}$, Cache Ψ)

Global structures:
$\mathbb{W}$: a sliding window for the most recent $\mathbb{W}_{size}$ queries
μ: the frequencies of query nodes in $\mathbb{W}$
1: $SP \leftarrow$ calculate the shortest path from $Q_{s,t}$
2: $CP \leftarrow$ calculate the concise shortest path of SP
3: **for** each $v_i \in SP$ **do** ▷ calculate dynamic GCP path
4: **if** $v_i \in CP$ or $\mu_i > 0$ **then**
5: add v_i to $DGCP$
6: **if** $\mathbb{W}$ is full **then** ▷ remove old query
7: dequeue $Q_{s',t'}$ from $\mathbb{W}$
8: $\mu_{s'} \leftarrow \mu_{s'} - 1$; $\mu_{t'} \leftarrow \mu_{t'} - 1$
9: enqueue $Q_{s,t}$ into $\mathbb{W}$ ▷ insert new query
10: $\mu_s \leftarrow \mu_s + 1$; $\mu_t \leftarrow \mu_t + 1$
11: apply LRU policy to update Ψ by $DGCP$ ▷ update cache

Query	Shortest Path
$Q_{8,9}$	$\langle v_8, v_5, v_7, v_9 \rangle$
$Q_{1,4}$	$\langle v_1, v_3, v_4 \rangle$
$Q_{1,10}$	$\langle v_1, v_3, v_4, v_5, v_7, v_9, v_{10} \rangle$
$Q_{2,10}$	$\langle v_2, v_5, v_7, v_9, v_{10} \rangle$

(a) shortest paths of queries

Time t	Query	ΔL	μ	DGCP Path
0	N.A.	N.A.	1,3,5,6,7,10: 1	N.A.
1	(8,9)	$-(1,6)$ $+(8,9)$	1,3,5,6,7,10: 1	$\langle v_8, v_5, \mathbf{v_7}, v_9 \rangle$
2	(1,4)	$-(3,5)$ $+(1,4)$	3,5,7,8,9,10: 1	$\langle v_1, \mathbf{v_3}, v_4 \rangle$
3	(1,10)	$-(7,10)$ $+(1,10)$	1,4,7,8,9,10: 1	$\langle v_1, v_5, v_7, \mathbf{v_9}, v_{10} \rangle$
4	(2,10)	$-(8,9)$ $+(2,10)$	1: 2; 4,8,9,10: 1	$\langle v_2, v_5, v_7, \mathbf{v_9}, v_{10} \rangle$

(b) running steps

Table 6: Running steps of dynamic caching, $L_{size} = 3$, for the road network in Figure 1

We proceed to illustrate the running steps of the algorithm. Table 6a shows a list of queries and their shortest paths. Table 6b depicts the running steps. ΔL shows which query will be added and removed from $\mathbb{W}$ at each timestamp. Array μ shows the frequency of query nodes in the sliding window. In this example, the extra node(s) are added to the CP path (shown in bold) in order to obtain the DGCP path. At time $t = 1$, we receive the query (8,9) and calculate its DGCP path. Since calculation of DGCP paths happen before updating μ, the content of μ does not change from the initial state. Then, we remove (1,6) and add (8,9) to $\mathbb{W}$. At time $t = 2$, the query from $t = 1$ affects the content of μ. Window $\mathbb{W}$ is updated as before. Similarly, the running steps at $t = 3$ and $t = 4$ are shown in the table.

7. EXPERIMENTAL STUDY

Section 7.1 covers the methods considered in the experiments. Section 7.2 covers the experimental settings. Section 7.3 covers experiments on real trajectory induced workload, while Section 7.4 covers the experiments on synthetic workloads.

7.1 Methods Considered

We evaluate the hit ratios of our caching methods and competitors on a machine running Debian. Table 7 lists all the methods used in our experiments. In the static caching setting, all methods use an existing caching policy for shortest paths [22]. In the

dynamic caching setting, all methods use LRU for cache replacement.

Our proposed methods are: (i) CP, caching concise paths as defined in Section 4, (ii) GCP, the static caching method for generic concise paths in Section 5, and (iii) DGCP, the dynamic caching method for generic concise paths in Section 6.

As discussed in related work, K-Skip [21] is a lossy shortest path compression method, so we do not compare with it. We compare our methods with two competitors: (i) full shortest path (SP), and (ii) concise path skip (CP-Skip). CP-Skip is a variant of K-Skip that includes all nodes of CP and every k-th node of SP. Since CP-Skip contains CP, it is a lossless shortest path method.

Method	Static caching policy	Dynamic caching policy
CP	[22]	LRU
GCP	[22]	N.A.
DGCP	N.A.	LRU
SP	[22]	LRU
CP-Skip	[22]	LRU

Table 7: Methods used in experiments

7.2 Experimental Setting

Datasets and Workloads

Table 8 shows information on the road networks used in the experiments. Although a real query log for online direction services [23] exist, service providers do not make their query logs publicly available. Thus, we generate query logs that each contains 300,000 queries.

We use real trajectories from the Aalborg [13] and Beijing [25] road networks. From each trajectory, we extract the start and end locations as the source and target nodes of a shortest path query in a query log. Since the trajectory datasets for Aalborg and Beijing are small (4.3k and 13k trajectories, respectively), we enlarge the trajectory dataset (to 300,000 trajectories) by sampling trajectories in the log and deviating their coordinates within a given radius (default value: 0.5 km).

For each of the two larger road networks (Colorado and New York), we generate a query log as follows. In the real world, drivers start from a dense region (e.g., a residential region) and travel towards another dense region (e.g., an industrial region). To simulate such behavior, we randomly choose a set of cluster centers with a given radius to form a set of clusters. Next, we randomly pick a pair of nodes from any 2 clusters to form a shortest path. By default, we use 10 clusters, and the radius is 8 km.

Following an existing experimental methodology [22], we divide the query log into two equal parts: (i) a query log $\mathcal{QL}$, for extracting query statistics and training the cache, and (ii) a query workload $\mathcal{WL}$ used for measuring the hit ratios of the methods.

Road network	Map size (km^2)	Road network
Aalborg	60x60	$download.cloudmade.com$ 129k nodes, 137k edges
Beijing	60x60	$download.cloudmade.com$ 76k nodes, 85k edges
Colorado	1000x1000	$www.dis.uniroma1.it/challenge9$ 435k nodes, 1,057k edges
New York	6000x4000	$www.dis.uniroma1.it/challenge9$ 264k nodes, 733k edges

Table 8: Characteristics of datasets

Parameters and Default Values

Table 9 lists the parameters and their default values. The first two entries show the default radiuses for the different road networks. Since different road networks have different sizes, we use 0.5 km as the default radius for the Aalborg and Beijing road networks, and we use 8 km as the default radius for the Colorado and New York road networks. The *number of clusters* denotes the number of clusters used for generating the synthetic workload (for Colorado and New York). The *cache capacity (ratio)* expresses the cache space as a percentage of the space needed for storing the entire road network. Thus, the absolute cache space for the different road networks are different. The last two are parameters for specific to CP-Skip and DGCP. *Path size ratio* is used by CP-Skip to define the ratio of the CP-Skip path length to the full shortest path length. *Window size* is used by DGCP to define the size of the sliding window.

Parameter	Default value
Radius (for Aalborg, Beijing)	0.5 km
Radius (for Colorado, New York)	8 km
Number of clusters (for Colorado, New York)	10
Cache capacity (ratio)	50%
Path size ratio (for CP-Skip)	50%
Window size (for DGCP)	1000 queries

Table 9: Default parameters

7.3 Real Trajectory Induced Workloads

We proceed to report on experiments using real trajectory induced workloads (on the Aalborg and Beijing road networks).

First, we examine the average lengths (i.e., number of nodes) of paths in the cache for the different methods, see Table 10. Observe that full shortest paths (SP) are much longer than concise shortest paths (CP). Generic concise shortest paths (GCP/DGCP) are only slightly longer than concise shortest paths (CP). This means that SP paths contain many intermediate nodes that do not intersect query nodes in the workload. Note that the average path lengths of the methods (e.g., SP and CP) are similar in the static and dynamic settings.

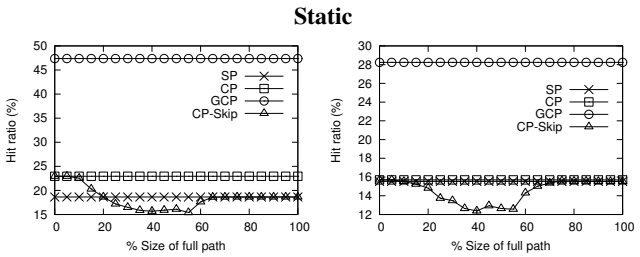
Road Network	Static			Dynamic		
	SP	CP	GCP	SP	CP	DGCP
Aalborg	338.5	38.7	48.1	315.6	37.3	41.3
Beijing	166.1	26.4	35.6	129.8	23.1	26.1
Colorado	874.8	98.3	115.7	887.3	96.9	106.1
New York	501.9	71.3	73.3	514.7	70.4	72.0

Table 10: Average path length (in nodes)

Next, we study the effect of the path size ratio on the performance of CP-Skip, see Figure 6. The other methods (SP, CP, GCP) are included for reference only; they do not take the path size ratio as a parameter. Clearly, GCP outperforms CP-Skip in all cases. The results for dynamic caching are similar, so we omit them.

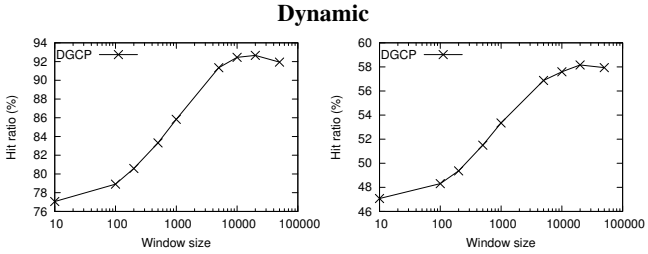
Next, we investigate the effect of the window size parameter on the hit ratio of the DGCP method (in the dynamic caching setting). Figure 7 plots the hit ratio of DGCP with respect to the window size. The hit ratio increases until the window size reaches 10,000. Observe that we can achieve a high hit ratio when the window size is sufficiently large.

Figure 8 shows the hit ratios of the methods as a function of the cache capacity, Again, GCP outperforms the other methods significantly, in both the static and dynamic settings. Since CP-Skip and



(a) Aalborg data (b) Beijing data

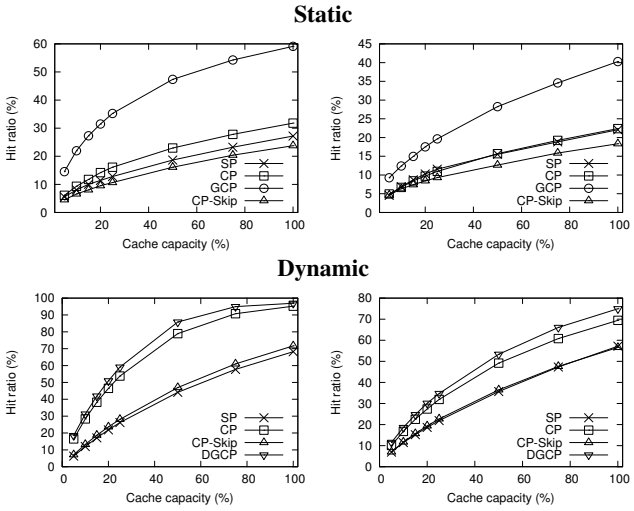
Figure 6: Hit ratio vs. path size ratio



(a) Aalborg data (b) Beijing data

Figure 7: Hit ratio vs. window size

SP have similar hit ratios, we exclude CP-Skip from subsequent experiments.



(a) Aalborg (b) Beijing data

Figure 8: Hit ratio vs. cache capacity

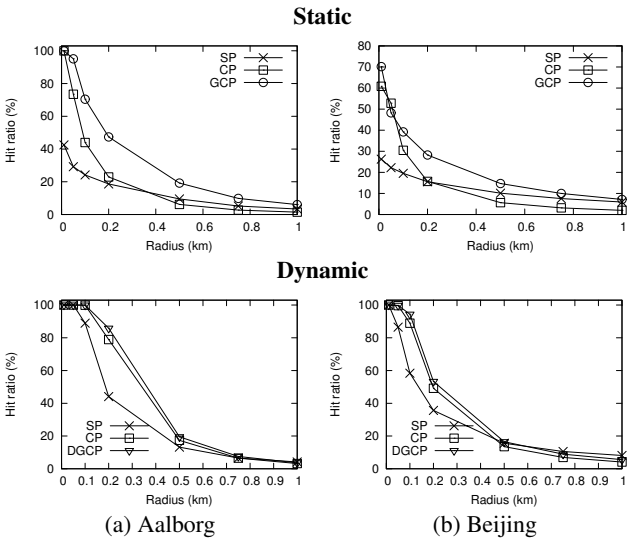
Figure 9 shows the hit ratios of the methods for various cluster radius values. Observe that a smaller radius leads to a higher hit ratio. This happens because smaller clusters result in fewer possible unique queries.

7.4 Synthetic Workload

We proceed to report on experiments using synethetic workload (on the Colorado and New York road networks).

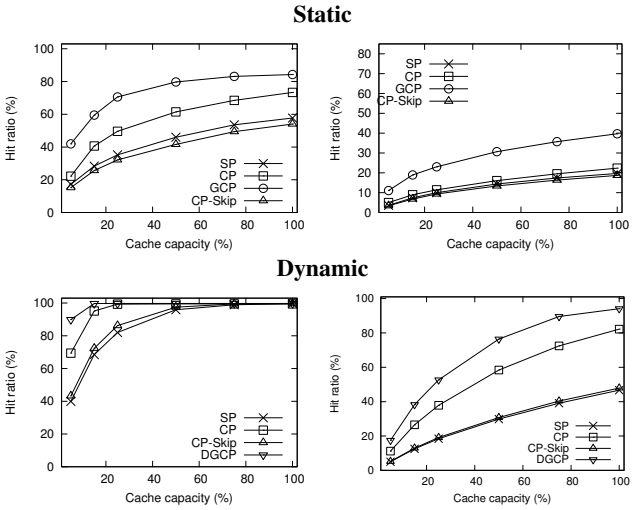
We plot the average path length of the methods, as shown in Table 10. In general, the results are similar to those for the real trajectory induced workloads.

Figure 10 shows the hit ratios of the methods with respect to the



(a) Aalborg (b) Beijing

Figure 9: Hit ratio vs. cluster radius



(a) Colorado (b) New York

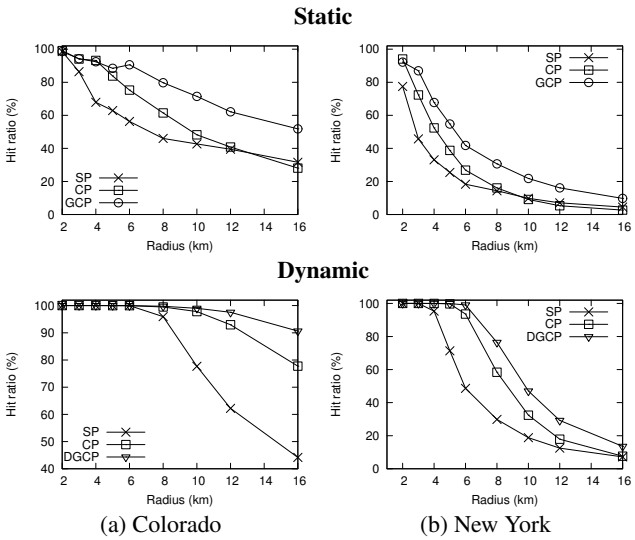
Figure 10: Hit ratio vs. cache capacity

cache capacty. In the static (dynamic) caching setting, GCP/DGCP can achieve a hit ratio of 84.28% (99.75%) and 39.70% (94.13%) for Colorado and New York, respectively. Observe that, in the dynamic caching setting, we can achieve a very high cache hit ratio with a small cache capacity. In the remaining experiments, we do not include the results for CP-Skip as CP-Skip and SP have similar hit ratios.

Figure 11 shows the impact of the cluster radius on the hit ratios of the methods. Observe that a smaller radius leads to a higher hit ratio. This is expected as a smaller radius implies that there are fewer unique query pairs in the workload.

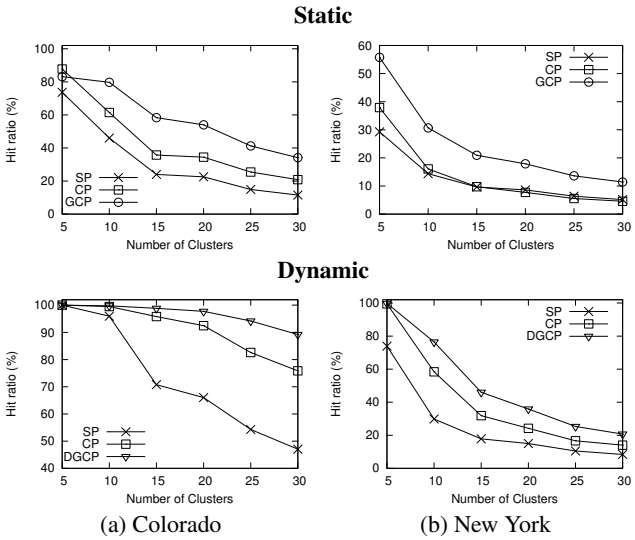
In the static caching setting, GCP always outperforms its competitors, and CP is the second best, followed by CP-Skip and SP in the last tier. In the dynamic caching setting, DGCP clearly outperforms its competitors.

Figure 12 shows the impact of the number of clusters on the cache hit ratio. GCP/DGCP consistently outperforms the competitors, with a lead to the next-best competitor by at least 20%.



(a) Colorado (b) New York

Figure 11: Hit ratio vs. cluster radius



(a) Colorado (b) New York

Figure 12: Hit ratio vs. number of clusters

8. CONCLUSIONS

In this paper, we exploit concise shortest paths to boost the cache hit ratio of shortest path queries in cache servers. First, we propose the notion of a generic concise shortest path that enables a trade-off between the path size and the number of queries that can be answered by the path. Then we develop static and dynamic caching techniques for generic concise shortest paths.

Experimental results show that the hit ratios of our best methods (GCP and DGCP) are 10%–40% higher than that of competitors. Our methods are more robust with respect to the cache capacity.

As for the future work, we will study the maintenance of cache content with respect to dynamic traffic updates.

Acknowledgements

This work was supported by ICRG grant G-YN38 from the Hong Kong Polytechnic University and by a grant from the Obel Family Foundation.

9. REFERENCES

[1] Google Directions API. `https://developers.google.com/maps/documentation/directions/`.

[2] MapQuest Open Directions API. `http://developer.mapquest.com/web/products/open/directions-service`.

[3] Web cache softwares. `http://en.wikipedia.org/wiki/Web_cache`.

[4] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, pages 782–793, 2010.

[5] R. A. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri. The impact of caching on search engines. In *SIGIR*, pages 183–190, 2007.

[6] G. V. Batz, R. Geisberger, D. Luxen, P. Sanders, and R. Zubkov. Efficient route compression for hybrid route planning. In *MedAlg*, pages 93–107, 2012.

[7] K. Button and D. Hensher. *Handbook of Transport Systems and Traffic Control*. Handbooks in Transportation Research Series. Elsevier Science, 2001.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[9] A. Dingle and T. Pártl. Web cache coherence. *Computer Networks*, 28(7-11):907–920, 1996.

[10] Q. Gan and T. Suel. Improved techniques for result caching in web search engines. In *WWW*, pages 431–440, 2009.

[11] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[12] H. Hu, J. Xu, W. S. Wong, B. Zheng, D. L. Lee, and W.-C. Lee. Proactive caching for spatial queries in mobile environments. In *ICDE*, pages 403–414, 2005.

[13] C. S. Jensen, H. Lahrmann, S. Pakalnis, and J. Runge. The INFATI data. *arXiv preprint cs/0410001*, 2004.

[14] K. Lee, W.-C. Lee, B. Zheng, and J. Xu. Caching complementary space for location-based services. In *EDBT*, pages 1020–1038. 2006.

[15] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. In *WWW*, pages 257–266, 2005.

[16] E. P. Markatos. On caching search engine query results. *Computer Communications*, 24(2):137–143, 2001.

[17] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-k page replacement algorithm for database disk buffering. In *SIGMOD*, pages 297–306, 1993.

[18] R. Ozcan, I. S. Altingövde, and Ö. Ulusoy. Static query result caching revisited. In *WWW*, pages 1169–1170, 2008.

[19] S. Podlipnig and L. Böszörményi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, 2003.

[20] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.

[21] Y. Tao, C. Sheng, and J. Pei. On k-skip shortest paths. In *SIGMOD*, pages 421–432, 2011.

[22] J. R. Thomsen, M. L. Yiu, and C. S. Jensen. Effective caching of shortest paths for location-based services. In *SIGMOD*, pages 313–324, 2012.

[23] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Y. Halevy. Hyper-local, directions-based ranking of places. *PVLDB*, 4(5):290–301, 2011.

[24] B. Zheng and D. L. Lee. Semantic caching in location-dependent query processing. In *SSTD*, pages 97–116, 2001.

[25] Y. Zheng, L. Zhang, X. Xie, and W.-Y. Ma. Mining interesting locations and travel sequences from GPS trajectories. In *WWW*, pages 791–800, 2009.

[26] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *SIGMOD*, pages 857–868, 2013.