Shortest Path Cache

October 14, 2011

Jeppe Rishede Thomsen
Department of Computing
Hong Kong Polytechnic University
Kowloon, Hong Kong
csjrthomsen@comp.polyu.edu.hk

Man Lung Yiu
Department of Computing
Hong Kong Polytechnic University
Kowloon, Hong Kong
csmlyiu@comp.polyu.edu.hk

ABSTRACT

short description of what we do, why this is an interesting/challenging problem, and how well we solve the problem (key results, mention theoretical guaranties.)

1. INTRODUCTION

Introduce problem and a running example to use throughout the paper. clarify why we need to solve this problem.

Challenges:

- The number of possible Shortest Path (SP) candidates to add to the cache are $|V|^2$. $|V|$ is the number of vertices in a road network.
- Searching the cache must be faster than calculating a SP.
- Needs to use small amount of space to achieve a good cache hit ratio.
- How to determine the value of a path and all its possible sub-paths.

2. RELATED WORK

Introduce related work, group them by similar work and tell what they do and why their approches can not solve our problem.

3. PROBLEM

In our setting we assume users own an Internet enabled mobile device with positioning capabilities. Users issue SP queries to an online service provider. Users want the response time, on the return of their SP result, to be comparable to that of an offline application. We use the terms query, incoming query, and user query interchangeably.

The SP service provider needs to provide a fast service to its users. The service provider also want to save cost on hardware, such as CPU and HDD space. The SP provider therefor wants to return as many SP results as possible, using the least amount of computation and space.

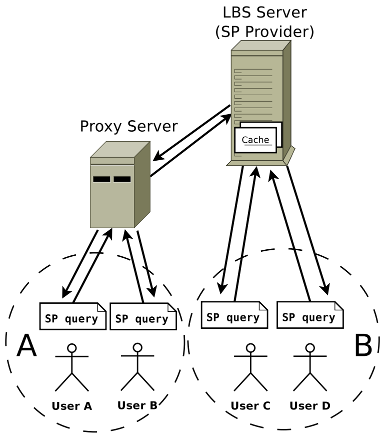


Figure 1: Overall scenario showing where the cache could be placed and where users may submit SP queries.

Calculating a SP will, regardless of the algorithm used, always be an expensive calculation [**?**]. Using a SP cache at the SP service provider can reduce the CPU cycles used in order to return a SP result. Doing so would at the same time also increase the response time of the SP service, as saving CPU cycles not only allows for more SPs to be computed on the same hardware, but also allows for returning a cached result much faster than it would be possible if the SP had to be calculated first.

A SP has the property of Optimal Subpath (OSS) (see lemma 3.1) which means that any SP in the cache can answer any SP query where both origin and target node are on the SP. Calculating which SPs, and their sub-paths, provide the most benefit will obviously be necessary for optimal utilization of the cache. A static cache, which is populated in an offline phase (fig. 3E), is used. A static cache will impose minimal overhead to query processing. Section 4 explains why we choose a static cache.

The properties of OSS(Lemma 3.1) states that all sub-paths on a SP are also SPs. Using SP Q1 in table 3 we would also be able to a query from v_3 to v_5, or v_4 to v_6 (See fig. 2), as these nodes are part of Q1. The same of cause also holds for any SP Q1-Q6 from table 3.

3.1 Architecture

We propose a system with a static SP cache implemented in front of an existing SP service (See fig. 3) such that if the cache can answer a query then the result can be returned immediately.

When the system receives a SP query from a user (fig. 3A) the system first checks if the cache (fig. 3B) is able to answer the query.

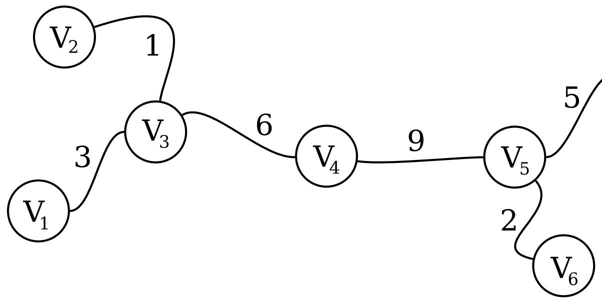


Figure 2: Simple graph representation of a map.

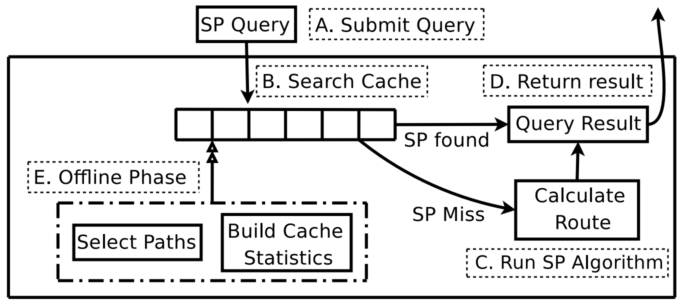


Figure 3: Buffer placement in SP service providers system.

If the cache contains the query answer it is immediately returned (fig. 3D), else the SP algorithm is called (fig. 3C) and the SP result returned (fig. 3D).

3.2 Optimization Intend

As the main problem with expensive SP calculations is time needed, we use reduction in *CPU Execution Time (CET)* as the main optimization target and measurement to evaluate our success. At a SP service provider, the time spent to return a SP result is essentially composed of two tasks: calculating the SP and the overhead of query processing.

We will address 3 subgoals:

1. Reduce the gross CET used to calculate SPs.
2. Reduce the gross CET spent on overhead in query processing.
3. Determine the value of a sub-path in the cache.

We want reduce the gross CET of SP calculations - goal 1 - as it is the most CPU intensive task at a SP service, and therefor also the most time consuming. By using a cache we expect to see a negavite correlation between cache hit rate and CET used on SP calculation.

In a SP service there will always be a some overhead associated with SP query processing. Introducing a cache in front of the existing SP service (fig. 3B) will undeniably add more overhead as the cache needs to be queried for all queries submitted to the SP service, regardless of whether it is able to answer the query or not. In order for the cache to be useful, the overhead introduced needs to be minimized (goal 2). We always have to make sure that the savings achieved by adding a cache to the system is greater than the overhead introduced.

The fact that the system will be using a static cache, and SPs exhibit the OSS property, makes goal 3 - determining the value of a SP subpath - very important. The ability to do goal 3 well will have a direct influence on goal 1 and 2. If we fill the cache with useless SPs we will end up calculating a SP for all queries, as well as the overhead from checking the cache for each query. Solving goal 3 well is the most direct way to solve goal 1.

Abbreviation	Meaning
SP	Shortest Path
$SP_{s,t}$	SP: $\{v_s, v_{s+1}, \ldots, v_t\}$
LRU	Least Recently Used
FIFO	First In First Out
SPS	SP subpath Sharing
CET	CPU Execution Time
OSS	Optimal Subpath

Table 1: Table of Notation

The OSS property states that every sub-path of a SP is also a SP. i.e SP Q1 in table 3 consists of: $\{SP_{1,3}, SP_{1,4}, SP_{1,5}, SP_{1,6}, SP_{3,4}, SP_{3,5}, SP_{3,6}, SP_{4,5}, SP_{4,6}, SP_{5,6}\}$, each one being a SP.

LEMMA 3.1. *If a path $SP_{s,t} : v_s, v_{s+1}, \ldots, v_t$ is a SP, then $\forall (v_k, v_l) | v_k \in SP_{s,t} \wedge v_l \in SP_{s,t}$ there is a SP $SP_{k,l}$ with start-/end-node in v_k, v_l, following a sub-path of $SP_{s,t}$*

4. CHALLENGES OF SP CACHING

In this section we qualify the differences between static (no change in cache content at service runtime) and dynamic caching (Cache content is changed at service runtime) of SPs. We also introduce our state of the art competitor and explain why it is not an adequate solution for the SP caching problem.

Using a dynamic cache[1] and calculating the utility of each path is very expensive. If a dynamic cache is used, and we want to ensure it always keeps the most useful paths in the cache, it will be very expensive to calculate the utility of a new query with respect how much it overlaps with existing SPs (i.e. how many vertices it shares with an existing SP) and how likely it will be able to answer a query in the future, thus adding a substantial overhead to query processing. As the utility of a SP is so expensive to calculate, while the SP service is running, it violates goal 2 in section 3.2.

Using LRU as the cache replacement policy in a dynamic cache ensures that only minimal overhead is added by using a dynamic cache. When a new query is submitted LRU evicts the least recently used SP and keeps the most recently used SPs in the cache. LRU, however, has several shortcomings:

- It has no way to determine the usefulness of inserting a path (i.e. no scoring function), which violates goal 3 (Sec. 3.2). Because LRU does not have a scoring function then, even if a path SP is valuable (covers many potential queries), if a sequence of consecutive queries, which SP can not cover, is submitted, then P will be evicted.
- LRU also does not have any way to optimize utilization of the cache space available, possibly wasting a lot of space.
- If no additional structure is added then querying the cache may require a scan of all paths in the cache to examine the cache can answer a query or not.
- LRU does not consider utilization of the space in the cache.

[1]Every time a new query is submitted we consider evicting an old item from the cache and inserting the new query, using a replacement policy e.g. Least Recently Used (LRU)

Symbol	Meaning
$Q_{s,t}$	SP query from s to t
$\chi_{s,t}$	The frequency of a SP
Ψ	The Cache
$\lvert SP\rvert$	Length of a SP
$\lvert\Psi\rvert$	Number of vertices in the cache
$\Phi(sp)$	The set of all sub-paths in a SP sp
$\Phi^c(\Psi)$	The set of all unique (sub-)paths in Ψ
$\Gamma(\Psi)$	Calculates the total utility of the content in the cache
$G(\mathbf{V},\mathbf{E})$	Graph representation of the Map
$\mathbf{V}$	The set of vertices in the Map
WL	Historical workload

Table 2: Table of Symbols

To give an example of LRUs shortcomings we assume a cache of size 10 (i.e. has space for 10 vertices), using LRU, and running on the map from figure 2. Queries Q1-Q6 from table 3 are used, Q1 first and Q6 last.

The answer to Q1 and Q2 (table 3) are added to the cache as both results have a length of 5 and they can both fit.

Q3 results in a cache hit with Q1. Q1 can answer Q3 because both start and target node (1 and 4) are on the cached path of Q1. This is a property is given by OSS (Lemma 3.1).

When Q4 is submitted Q2 will be evicted and Q4 inserted. Q2 will be evicted from the cache because no item in the cache can answer Q4, and Q2 is the least recently used.

Q5, which *could* have been answered by Q2, now results in the eviction of Q1 and insertion of Q5 because the cache no longer contains any item able to answer Q5 and Q1 is now the least recently used item.

Q6 is not covered by Q3 or Q5, the current elements in the cache, resulting in a cache miss. Because there are 7 nodes in the cache and Q6:$\{v_3, v_4, v_5, v_6\}$ has 4 nodes, Q3 will be evicted (since it was inserted first) and Q6 inserted.

The end result is that 30% of the cache space is wasted and out of the 6 queries only we only resulted in a 1 cache hit. If Q1 and Q2 had been kept in the cache we could have had 3 cache hits. However, keeping Q1 and Q2 also demonstrates LRUs problem with overlapping paths. Q1 and Q2 are identical except for one node, which waste a lot of space on duplicated nodes. Had we represented each node only once (addressed in section 5.5) we would be able to fit all nodes in the map in the cache.

5. CONTRIBUTION

Intro to section
Show benefits of a static cache over a dynamic cache.
List all advantages of a static cache solution
(*Static cache* solves goal 2. has zero maintenance cost after filling the cache.)

explain what will be introduced, which sub-problems are considered, and which subsections presents what.

5.1 Benefit model

We will define our goals more formally, introducing the benefit equations we aim to minimize.

A query is a pair of vertices ids (v_s, v_t), denoted $Q_{s,t}$ and the SP returned from such query we denote $Q_{s,t}$. In order to evaluate goal 3 we calculate the utility of the content in the cache, denoted $\Gamma(\Psi)$. To calculate $\Gamma(\Psi)$ we find the set of unique sub-paths from all SPs in the cache (Ψ) and sum up the frequency, $\chi_{s,t}$, of each unique path. χ, a table of utility computed from existing historical data. An example of different entries, $\chi_{s,t}$, is given in table 5

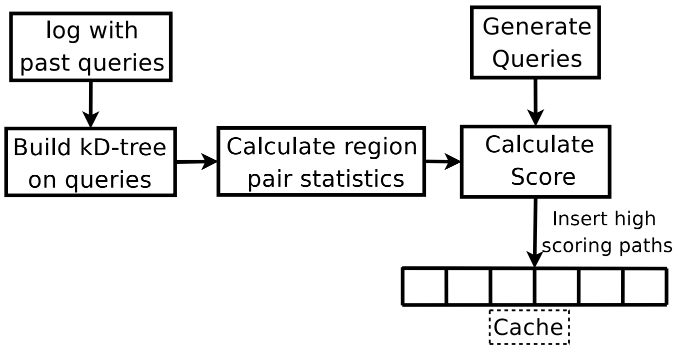


Figure 4: Insertion of cache elements in offline phase.

Query	$Q_{s,t}$	$SP_{s,t}$
Q1	$Q_{1,6}$	$\{v_1, v_3, v_4, v_5, v_6\}$
Q2	$Q_{2,6}$	$\{v_2, v_3, v_4, v_5, v_6\}$
Q3	$Q_{1,4}$	$\{v_1, v_3, v_4\}$
Q4	$Q_{4,7}$	$\{v_4, v_5, v_7\}$
Q5	$Q_{2,5}$	$\{v_2, v_3, v_4, v_5\}$
Q6	$Q_{3,6}$	$\{v_3, v_4, v_5, v_6\}$

Table 3: Example Queries

In order to evaluate our method we try to maximize $\Gamma(\Psi)$ (Eq. 2).

$$\Phi^c(\Psi) = \bigcup_{\forall SP \in \Psi} \Phi(SP) \tag{1}$$

$$\Gamma(\Psi) = \sum_{SP_{s,t} \in \Phi^c(\Psi)} \chi_{s,t} \tag{2}$$

$$\Phi(sp) \leftarrow \{SP_{s,t} \mid s \in sp, t \in sp, s \neq t\} \tag{3}$$

When we want to evaluate the utility of Ψ we first calculate $\Phi(SP)$ for each SP. Table 4 shows the set of sub-paths from the SPs for each query Q1-Q6 from table 3 (we assume all queries fit into the cache). Once we have all the sets of sub-paths from the cache content, we use equation 1 to union them together and obtain the set of unique sub-paths in the cache, $\Phi^c(\Psi)$ (see table 4). After we have obtained $\Phi^c(\Psi)$, we use equation 2 to calculate $\Gamma(\Psi)$, the total benefit we expect to have with the content in Ψ. With $\Phi^c(\Psi)$ obtained, using $\chi_{s,t}$ values from table 5, $\Gamma(\Psi)$ would be 6

$$\begin{aligned}
\Phi(SP_{1,6}) &= \{SP_{1,3}, SP_{1,4}, SP_{1,5}, SP_{1,6}, SP_{3,4}, SP_{3,5},\\
&\quad SP_{3,6}, SP_{4,5}, SP_{4,6}, SP_{5,6}\}\\
\Phi(SP_{2,6}) &= \{SP_{2,3}, SP_{2,4}, SP_{2,5}, SP_{2,6}, SP_{3,4}, SP_{3,5},\\
&\quad SP_{3,6}, SP_{4,5}, SP_{4,6}, SP_{5,6}\}\\
\Phi(SP_{1,4}) &= \{SP_{1,3}, SP_{1,4}, SP_{3,4}\}\\
\Phi(SP_{4,7}) &= \{SP_{4,5}, SP_{4,7}, SP_{5,7}\}\\
\Phi(SP_{2,5}) &= \{SP_{2,3}, SP_{2,4}, SP_{2,5}, SP_{3,4}, SP_{3,5}, SP_{4,5}\}\\
\Phi(SP_{3,6}) &= \{SP_{3,4}, SP_{3,5}, SP_{3,6}, SP_{4,5}, SP_{4,6}, SP_{5,6}\}\\
\hline
\Phi^c(\Psi) &= \{SP_{1,3}, SP_{1,4}, SP_{1,5}, SP_{1,6}, SP_{3,4}, SP_{3,5},\\
&\quad SP_{3,6}, SP_{4,5}, SP_{4,6}, SP_{5,6}, SP_{2,3}, SP_{2,4}, SP_{2,5},\\
&\quad SP_{2,6}, SP_{4,5}, SP_{4,7}, SP_{5,7}\}
\end{aligned}$$

Table 4: Φ (Eq. 3) results for queries in table 3

$\chi^s/_t$	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	X	0	0	1	0	1	0
v_2	0	X	0	0	1	1	0
v_3	0	0	X	0	0	1	0
v_4	1	0	0	X	0	0	1
v_5	0	1	0	0	X	0	0
v_6	1	1	1	0	0	X	0
v_7	0	0	0	1	0	0	X

Table 5: $\chi_{s,t}$ values for $\Phi^c(\Psi)$ in table 4

i.e. $\chi_{1,6} + \chi_{2,6} + \chi_{1,4} + \cdots + \chi_{3,6} = 6$, finding a value in table 5 for all SPs of $\Phi^c(\Psi)$ (Tab. 4).

When filling the cache, during the offline phase, potential SP $_{s,t}$ items are scored based the $\chi_{s,t}$-value found using the SP start and end vertices. The score will be zero if the path is already covered by a path, or sub-path, already present in the cache. If we assume $SP_{1,6}$(Q1) is already in the cache and we are considering to insert either $SP_{2,6}$(Q2) or $SP_{4,7}$(Q4) next, then we first find the entry in table 5 for each query, and check neither is already covered by the content of Ψ, i.e not covered by Q1. The score of Q2 would be 2 and for Q4 it would be 1. In this case we would then insert Q2 as it represents the largest expected benefit if inserted into Ψ.

5.2 Hardness Analysis

Theoretical analysis showing how hard the problem is to solve for SP caching.
show it is NP-Hard

5.3 Greedy algorithm

We will here introduce PScache (Alg. 1), the base idea of our greedy algorithm. First we will explain what each line of PScache does, where after we will give an example to show how PScache works on example input.

PScache takes 4 arguments: $G(V, E)$, the graph representation of a map. $\boldsymbol{\Psi}$, the cache to be used (it is possible to imagine there could be more than one cache in case the SP provider serve two disjoint areas like i.e. Europe and Japan). $\mathcal{B}$, the cache budget, states how many nodes Ψ can contain before it is full. χ specifies the statistical information used by PScache. In PScache, line 1, we first initialize H as a max-heap. Line 2-3 fills H (utility,SP) pairs, with utility calculated using equation 5. Equation 5 calculates the score of a SP, sp, based on the sub-paths of sp which are not already present in the cache (Eq. 4).

$$U_{sp} \leftarrow \Phi(sp) \setminus \Phi^c(\Psi) \tag{4}$$

$$\begin{aligned} S(\chi, SP_{b,e}, \Psi) = \quad & \chi_{s,t} | SP_{b,e} \in U_{SP_{b,e}}, \neq t \\ & \chi_{s,t} \in \chi, s = b, t = e, s \end{aligned} \tag{5}$$

Line 4 of PScache starts looping to fill up the cache, terminating when either the cache is full ($|\Psi| \leq \mathcal{B}$), when the highest scoring SP has an utility value of 0 ($key_{max} \neq 0$) (According to lemma 5.1 we can not benefit from SPs with utility equal to zero), or if there are no SP candidates left to add to Ψ.

In line 5 we assign the pair (utility, SP) the the highest utility score to $\langle key_{max}, SP_{max} \rangle$ and remove it from H, the max-heap.

In line 6 we update the utility value, key_{max}, to know its true utility in case it has changed due to an SP insertion.

Line 7-11 Implements a compare-and-update/insert loop, avoiding recalculation of all elements in H every round of the loop in line 4. This is necessary because the basis for calculating utility in equation 5 is all possible paths, except those paths, and their sub-paths, already in the cache (Ψ). This means that every time we add a SP to the cache, the utility of some other candidates are likely to be reduced. The way it works is by comparing key_{max} and key of the top element of H, H.TopKey, (line 7), if key_{max} > H.TopKey we add its SP to the cache immediately (line 7-9), else we recalculate key_{max} and add $\langle key_{max}, SP_{max} \rangle$ back into H (line 10-11). This will in the worst case mean we have to recalculate all elements in H after a SP insertion in Ψ, the cache. However, it is likely that the new (utility,SP) pair with the highest utility score is still close to the top of H, meaning we save a lot of calculations/time. This works since utility scores can only decrease or stay the same, but never increase.

LEMMA 5.1. *The utility of a SP will, when added to the cache, Ψ, always increase the utility of Ψ by exactly the utility of the SP:* $S(\chi, SP, \Psi) = \Gamma(\Psi \cup \{SP\}) - \Gamma(\Psi)$

PROOF. When calculating the utility of a SP sp: $S(\chi, sp, \Psi)$ only the set of sub-paths in sp disjoint from $\Phi^c(\Psi)$ is consider as basis for the calculation.

As there is no overlap between SPs forming the basis for the utility of Ψ and the set of SPs forming the basis for the utility of sp, then any positive utility value of sp can only be added to the Ψ utility value. $\square$

To show an example of how PScache (Alg. 1) works we assume an empty cache, using $\mathcal{B} = 15$, table 4, 5, and the queries from table 3. For clarity we limit our example to show only how it works on the queries Q1-Q6 from table 3. We use table to show the utility of each query after each round of the PScache (Alg. 1, line 4-11)

In the first round we calculate a score for all possible queries (Alg. 1, line 2-3). Since the cache is empty, a SP score consist of its frequency from table 5, as well as the frequency of its sub-paths. For Q1 this would give a utility of 0+1+0+1+0+0+1+0+0+0 = 3 i.e. the frequency of all the sub-paths from $\Phi(SP_{1,6})$ (Tab. 4). The utility for Q2-Q6, as well as all other possible queries, is calculated in the same manner. Only the example queries are shown in table 6, ... is used to symbolize the remaining possible queries. From round 1 (Tab. 6) we can see that Q1 and Q2 are equal candidates for the first item in the cache. In line 4-8 query result $SP_{1,6}$ (from Q1) is chosen, as indicated by the $|\underline{\overline{42}}|$ in round 1 (Tab. 6) . How to break ties, as with Q1 and Q2, would depend on the individual max-heap implementation.

In the second round of PScache (Alg. 1, line 4-11) the cache is now no longer empty and we will run line 4-10 several times to find which SP now has the highest utility. When we first run line 6 we find Q2 now has a utility of 2 and in line 7 we see the top item in H has an utility 1 (expected utility, since has not been recomputed yet), so in line 11 we push back the SP (SP_{max}) together with its updated utility value of 10. Second time we now pop Q6, since it now has the highest utility value of 32. However, after line 6 it now has a utility of 0, so it will be reinserted into H with its new utility (line 11). Third time around we now pop Q5. After line 6 has a utility of 5, which is still smaller than Q4 in the heap, so we reinsert it (line 11). On the fourth turn we pop Q4 with a utility of 15, and after line 6 an actual utility of 11. However, the top of H also has a utility of 11, but since the utility of Q3 can never get larger, only stay the same or become smaller, we don't have to worry about it and we can goto line 7-9 and insert $SP_{4,7}$ (Q4). The second round almost had us recalculate all the items in H (worst case), however, we were still able to avoid calculating the actual utility of Q3 before we were certain we should insert the SP of Q4.

Query \ Round	1	2	3	4
Q3: $(SP_{1,4})$	11	11	0	0
Q4: $(SP_{4,7})$	15	$\boxed{11}$	-	-
...	...	...	...	...

Table 6: Utility for Q1-Q6 during 4 rounds of PScache. Boxed in values indicate which items are added to the cache in each round.

Algorithm 1: PScache($G(V,E), \Psi, \mathcal{B}, \chi$)

```
input :
  $G(V,E)$: Graph representation of Map
  $\Psi$: The cache
  $\mathcal{B}$: Cache budget
  //H Contains the utility score of all possible SPs in $\mathbf{G}(\mathbf{V},\mathbf{E})$.
    The SP is the value v and the utility is the key k, (k, v). The
    heap is sorted on the key.
1 H Initialize Max-Heap
  //Initialially fill H
2 forall $SP_{s,t} | s \in V, t \in V, s \neq t$ do
3     H.push(S($\chi, SP_{s,t}, \Psi$), $SP_{s,t}$)
  //Fill cache
4 while $|\Psi| \leq \mathcal{B}$ AND $SP_{ms}.k \neq 0$ OR $|H| > 0$ do
      //Assign (utility,SP) pair with the highest utility to $SP_{ms}$
5     $\langle key_{max}, SP_{max} \rangle \leftarrow$ H.pop()
      //Update utility, as previous SP insertion has changed it
6     $key_{max} = S(\chi, SP_{max}, \Psi)$ //H.TopKey() looks at the
        top (k,v) pair without removing it from the heap
7     if $key_{max} \geq H.TopKey$ then
8         if $(\mathcal{B} - |\Psi|) \geq |SP_{max}|$ then
9             $\Psi.insert(SP_{max})$
10    else
11        H.push(S($\chi, SP_{max}, \Psi$), $SP_{max}$)
```

Round three works as already seen in round 2. We first consider Q3, which has an actual utility of 0, then we consider Q2, whose utility is unchanged after line 6, so we insert it in round 3.

In round 4 Q5 still has a utility of 7, but after recalculation at line 6 it becomes 0 and the algorithm stops. We never insert a SP with a utility of 0, since any query it can answer, the cache, Ψ, can already answer.

5.4 Statistics

We have previously introduced the notion of utility and shown how we can use the frequency of SPs (Tab. 5) on a map to calculate the utility of an individual SP.

We will make it clear why assuming a uniform distribution of SPs is an insufficient strategy to base utility calculations on.

Algorithm 2 gives a high level overview of how our algorithms works and we will reference it in the following sections when we solve a new problem part. We will make it clear how we extract the frequency information, the statistics, of SPs on a map (Alg. 2,PART1). We will show how we use these statistics in order to generate SP candidates for insertion to the cache, Ψ (Alg. 2,PART2).

The extraction of statistics extraction and selection of candidate paths is part E of figure 3.

5.4.1 Statistics Extraction

Algorithm 2: Build Cache Statistics & Generate Candidates.

```
input : $G(V,E), \Psi, \mathcal{B}, WL$
  //PART1
1 WL - historical workload. i.e start-/end-points of trajectories
2 Build regions on G(V,E)
3 Build the table R - holds the vertex sets for each region
4 use WL with R to build $\chi$ (statistics), using regions.
  //PART2
5 forall $(R_s, R_t) | (R_s, R_t) \in \chi$ do
6     Max-Heap H $\leftarrow$ generate-candidate($\chi, WL$)
7 while Utility(Max-candidate $\in H$) $\neq 0$ AND $|\Psi| < \mathcal{B}$ do
8     Add Max-candidate to $\Psi$
      //empty H and generate a new set of candidates.
9     H.clear
10    forall $(R_s, R_t) | (R_s, R_t) \in \chi$ do
11        Max-Heap H $\leftarrow$ generate-candidate($\chi, WL$)
```

The purpose of statistics extraction is to build a frequency table like table 5. While we have not yet introduced regions, this essentially corresponds to line 4 in PART1 of algorithm 2.

There are two extremes when considering what to build table 5 from. In the first extreme we can assume a uniform distribution of all possible queries ($\chi_{s,t} = \frac{1}{v^2}$), meaning that the SPs chosen by PScache for the cache will be evenly distributed over the entire map. Due to the number of possible paths on a map, in the order of $\frac{v^2}{2}$ possible paths on the number of vertices in a map, a uniform distribution will pick too many low quality candidates (SPs very few users will submit queries for in the future) for the cache.

The second extreme is to build table 5 from historical information about user queries, resulting in PScache only adding paths to Ψ which has been observed in the past (Eq. 7). We extracted our set of start- and end-points pairs from the historical workload i.e. previously entered SP queries. We denote this set Ω. We use Ω to build table 5 with $v_1, v_2, \ldots, v_n$ on both column and row, n being the number of vertices in the map, $G(V,E)$. We first initialize all values in the table to zero and then for each pair $(s,t) \in \Omega$ we increment the count in cell s,t and t,s by one.

If historical queries are used we can avoid many low quality SPs. We do however need to be sure we have enough historical data since candidates for the cache can only be chosen from the exact member of the set of historical workload. If the historical dataset is not large enough we risk overfitting the cache content to the historical dataset, thereby including low quality SPs too. Between the two option (Uniformly destribution vs. historical workload), using historical information is the best candidate. We will use historical workload in the remainder of this section.

Using historical queries from table 5 we collect all the start-/end-pairs into Ω ($\{(1,6),(2,6),(1,7),(2,7)\}$). We then initialize table 7 with all zeros and then add all the pairs from Ω into table 7

Once we have inserted all pairs from Ω into table 7 we have finished extracting our statistics, the result of which is table 7.

When we generate all possible SP candidates, or generate SP candidates from the same data source as the statistics were generated from, there is a high chance of overfitting the SPs to the statistics. By trying to identify popular regions, in stead of specific paths, we can work at a higher abstraction level and more easily capture the tendencies in the historical workload. A tendency could e.g. be that most people drive from a residential area to an industrial area twice a day for work. If we only look at the raw data we can only

$\chi^{s}/_{t}$	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	X	0	0	0	0	1	1
v_2	0	X	0	0	0	1	1
v_3	0	0	X	0	0	0	0
v_4	0	0	0	X	0	0	0
v_5	0	0	0	0	X	0	0
v_6	1	1	0	0	0	X	0
v_7	1	1	0	0	0	0	X

Table 7: $\chi_{s,t}$ values for Ω pairs from table 8 queries.

$$\begin{aligned} Q_{1,6} &= \{v_1, v_3, v_4, v_5, v_6\} \\ Q_{2,6} &= \{v_2, v_3, v_4, v_5, v_6\} \\ Q_{1,7} &= \{v_1, v_3, v_4, v_5, v_7\} \\ Q_{2,7} &= \{v_2, v_3, v_4, v_5, v_7\} \end{aligned}$$

Table 8: Historical workload (WL)

$\chi^{R_i}/_{R_j}$	R_1	R_2	R_3	R_4
R_1	0	0	1	1
R_2	0	0	1	1
R_3	1	1	0	0
R_4	1	1	0	0

Table 9: χ region table produced using query workload from table 8.

see that there are many people sharing some set of edges in their routes, but since people usually live in the own house, or different apartment buildings, their starting and end points will be different and we won't capture that they all are going from (generally) the same place to the same area. Having a higher level understanding of popular regions on the map allows us to insert more paths into the cache which serves as connections between two popular regions.

By dividing the map into partitions we can record the statistics for these partitions in stead of the individual SPs. Using the example queries in table 8 we can easily see that the 4 queries are almost identical, differing by at most 2 vertices. By using our regular statistics method we only know that v_3, v_4, v_5 are very popular, but if we use the map/graph in figure 5 and perform the statistic on the regions in stead, then we get something like table 9. We still check the frequency for each pair in a SP, but we now increment a region cell in stead of a vertex cell in 9.

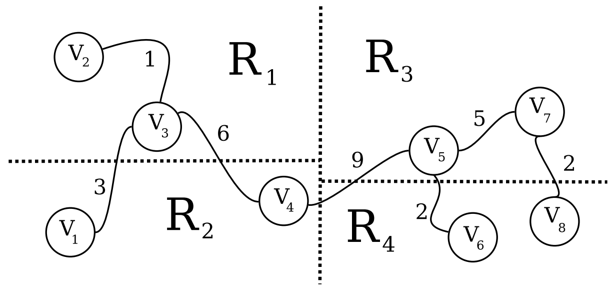


Figure 5: Map Partitioned into 4 areas.

Using the queries in table 8 with figure 5 we get the region-to-vertex set relationship in table 10. Using table 10, or equivalently equation 6, we can transform the start-/end-pairs into region pairs. Doing this we can build the region version of table 5 (Tab. 9). To generate candidates we generate one possible Ψ candidate from

R_1 :	$\{v_2, v_3\}$
R_2 :	$\{v_1, v_4\}$
R_3 :	$\{v_5, v_7\}$
R_4 :	$\{v_6, v_8\}$

Table 10: Regional vertex sets

Algorithm 3: PScache($G(V, E), \Psi, \mathcal{B}, \chi, WL$) – Generating candidates from historical data (Add WL parameter and replaces line 2 & 3 in algorithm 1)

```
  //Initialially fill H
1 forall $SP_{s,t}|(s, t) \in WL, s \in V, t \in V, s \neq t$ do
2     H.push(S($\chi, SP_{s,t}, \Psi$), $SP_{s,t}$)
```

each positive entry in table 9 by randomly choosing a start-/end-node from each regions pairs vertex sets i.e. one vertex from each region (Tab. 10). Using regions does not change PScache, regions are only used in the choosing of candidates in line 2 & 3. This completes our implementation of line 2-4 of algorithm 2

$$\chi_{R_i,R_j} = |R_i| \times |R_j| \times \chi_{s,t} \tag{6}$$

$$\chi s, t = \frac{\chi_{R_i,R_j}}{|R_i| \times |R_j|} \tag{7}$$

For partitioning the map we use kD-tree, an existing partition method.

5.4.2 *Candidate Generation*

We will explain how and why we chose to implement line 5-6 of algorithm 2. Once we have crated the χ-table we need candidate SPs for insertion into the cache. There are two ways we can generate these candidate. The first way being to generate candidates directly from the map $G(V, E)$, and the second method being based on additional historical workload to produce the candidates.

We can generate candidates for Ψ by finding all possible SP in $G(V, E)$ as described in algorithm 1. Generating all SP possible on $G(V, E)$ is quite expensive when considering the possible number of candidates vs. the size of $\mathcal{B}$ which would usually be much smaller. One straight forward optimization to this would be to use only those SP which has a non-zero entry in table 5, this could be very effective if there are few positive values. Another method we can use is sampling, where we just generate a subset of possible candidates. By choosing nodes evenly distributed on the map we can make sure the subset of SPs picked, still covers the map.

Another way to generate candidates for the cache is to base the candidate generation on historical workload. We modify algorithm 1 to take one extra parameter, WL, meaning the historical workload. We replace the requirement in line 2 of algorithm 1 to use all possible SPs, with a requirement to only calculated the scores on the set of SPs from WL. The changes to algorithm 1 are presented in algorithm 3.

To generate SP candidates for Ψ by finding all possible SP on a map $G(V, E)$ is a relatively simple matter, all we have to do is execute a SP query for all possible combinations of vertices in V, leading to $\frac{|V|^2}{2}$ Candidates.

To restrict the candidate generation to only include those who has a non-zero entry in table 5 we will exclude any pair of vertices which has an entry of zero in table 5. We generate candidates as before, but only consider any pair with a non-zero entry in table 5.

If we use sampling we can simply pick random pairs of vertices until we have enough candidates. We assume a uniform random function choosing vertices uniformly distributed on the map, so the sampling method still produce a representative set of candidates covering the map.

Regardless of how the candidates are generated in algorithm 1, it does not affect the main part of the algorithm, only line 2-3. We will be using both sampling from randomly generated candidates as well as split workload. This concludes our implementation of line 5-6 of algorithm 2

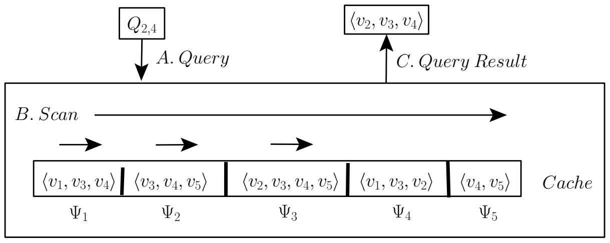
5.5 Cache Representations and Cache Concepts

The cache storage representation, and the manner in which we search it for cache hits, is crucial to the performance of a cache. We will present the basic, naive, approach as well as several ideas for improvement.

5.5.1 Simple array of paths

The simplest way to represent a cache is a simple array of paths (Fig. 6). This representation is expensive to search since we need to search each individual path in the cache to check for a cache hit. The search procedure is akin to a nested for-loop, every time we want to check if an item is in the cache. We will use this representation as a baseline for comparing optimizations to handle goal 2 (section 3.2).

In the example shown in Figure 6 we submit the query $Q_{2,4}$ to the cache (Fig. 6A). To search the cache we first scan Ψ_1 (Fig 6B) to see if both v_2 and v_4 are in the cached SP result $\text{SP}_{1,3}$. The result is empty so we scan Ψ_2, which again does not contain v_2 and v_4, meaning we did not find a cache hit. When we scan Ψ_3 we find both v_2 and v_4, so we stop searching and return the result $\langle v_2, v_3 v_4 \rangle$ from Ψ_3 (Fig 6C). Had the result of $Q_{2,4}$ not been in the cache we would have scanned all elements $\Psi_1 - \Psi_6$ before calling a SP algorithm to calculate the result.



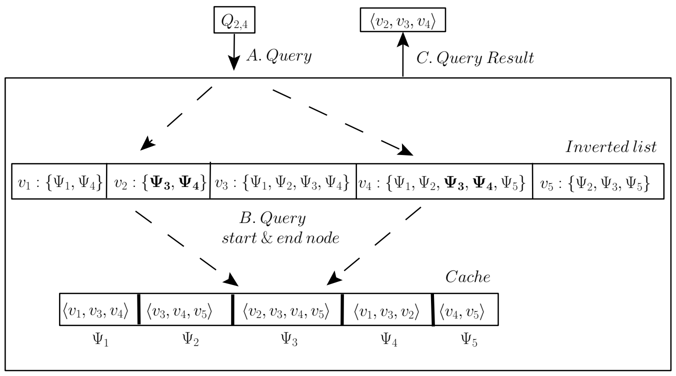
$Q_{2,4}$
A. Query
$\langle v_2, v_3, v_4 \rangle$
C. Query Result
B. Scan
$\langle v_1, v_3, v_4 \rangle$
$\langle v_3, v_4, v_5 \rangle$
$\langle v_2, v_3, v_4, v_5 \rangle$
$\langle v_1, v_3, v_2 \rangle$
$\langle v_4, v_5 \rangle$
Cache
Ψ_1
Ψ_2
Ψ_3
Ψ_4
Ψ_5

Figure 6: Simple array of paths.

5.5.2 Simple array of paths inverted list

If we add an inverted list, holding for each vertex a list of paths which includes the vertex in its path (Fig. 7), we can reduce the query time notably compared to searching the cache directly. We consider adding an inverted list as it is an effective means to reduce CET spent on query execution and thereby fulfill goal 2 from section 3.2.

In the example shown in figure 6 the start- and end-point of the query $Q_{2,4}$ (Fig. 7A) are used to retrieve the list of paths that vertex v_2 and v_4 are included in (Fig. 7B). If the intersection of the two lists are non-empty we have a cache hit. In figure 7 there are two candidates which can provide the answer, namely Ψ_3 and Ψ_4. If we have more than one candidate we simply choose the first candidate and return the query answer from that item (Fig. 7C).



$Q_{2,4}$
A. Query
$\langle v_2, v_3, v_4 \rangle$
C. Query Result
Inverted list
$v_1 : \{\Psi_1, \Psi_4\}$
$v_2 : \{\boldsymbol{\Psi_3}, \boldsymbol{\Psi_4}\}$
$v_3 : \{\Psi_1, \Psi_2, \Psi_3, \Psi_4\}$
$v_4 : \{\Psi_1, \Psi_2, \boldsymbol{\Psi_3}, \boldsymbol{\Psi_4}, \Psi_5\}$
$v_5 : \{\Psi_2, \Psi_3, \Psi_5\}$
B. Query start & end node
Cache
$\langle v_1, v_3, v_4 \rangle$
$\langle v_3, v_4, v_5 \rangle$
$\langle v_2, v_3, v_4, v_5 \rangle$
$\langle v_1, v_3, v_2 \rangle$
$\langle v_4, v_5 \rangle$
Ψ_1
Ψ_2
Ψ_3
Ψ_4
Ψ_5

Figure 7: Simple array of paths, accessed by inverted list.

5.5.3 Graph representation & Sharing sub-paths

An alternative way to represent the cache of SPs is as a graph (Fig. 8). The advantage of representing the cache as a graph is that each vertex is only represented once. We maintain a list of path ids with each vertex, with an entry for each cache item containing the vertex. To know whether the cache can answer a query is still easy, simply check whether the intersection of the SP id set from query's start- and end-vertex is non empty. Finding the actual answer does however take a little more work, as we need to traverse the graph along the SP query answer to know which vertices are in the list. This searches $\sum_{v_0}^{v_{|SP|}} v_i * |e_i|$ vertices, where $v_i \in V$ is a vertex on the SP answer and $|e_i| \in E$ is the edge degree of v_i.

Figure 8 illustrates how the query $Q_{1,5}$ is queried on the graph representation of the cache. First we check if the SP id set intersection of v_1 and v_5 is non-empty (Fig. 8B). Since there is a SP (path 1) in the cache which can answer $Q_{1,5}$, we will do a search for all vertices between v_1 and v_5 which also includes path 1 in their SP id set (Fig. 8C). The edges taken when doing this search is shown with bold arrows. v_2 is also examined since it is connected to the actual path, but is ignored afterwards as v_2 does not have SP id 1 in its set. Note that since both v_1 and v_5 has id 1 in their SP id set, then there will always be a path between them. Once we have traversed the graph and found the answer to $Q_{1,5}$ we return the result (Fig. 8D).

As the graph representation is very compact and only represents each vertex once, it allows for a high degree of sub-path sharing, which directly translates into more space available for more paths in the cache. As such the goal of this representation is to reduce the overall CET used on SP calculation (Goal 1 in section 3.2) by increasing the number of paths in the cache so fewer queries may need to be calculated.

The cache query time is slightly worse than when using an array with an inverted list. The graph representation overall fulfills goal 1 by allowing more paths in the cache, but it does unfortunately also impose some overhead to the query answer time, as each answer needs to be found in the graph (Fig. 8C) once it has been confirmed (Fig. 8B).

An optimization to the graph representation is to not directly represent each path id in the path id set of each vertex. We can do this by collapsing each continues sequence of path ids, which then again allow for even more ids to be added to the cache. Table 11 shows how the id sets for each vertex in figure 8 can be collapsed and what the savings are in terms of how many more path ids we have space for.

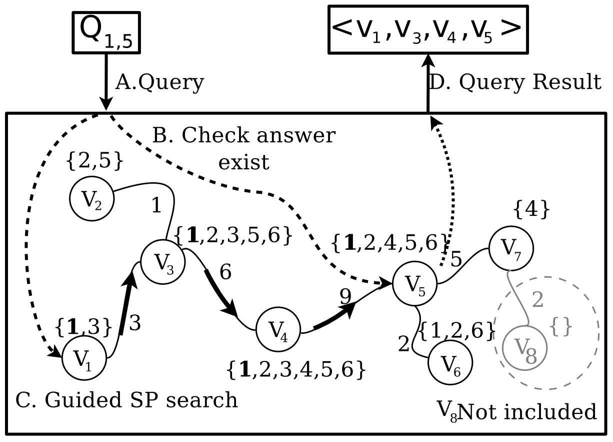


Figure 8: Graph representation of SP cache. Inserted paths from table 3

Vertex	Compressed list	Num. path ids saved
v_1	$\langle 1, 3\rangle$	0
v_2	$\langle 2, 5\rangle$	0
v_3	$\langle 1-3, 5-6\rangle$	1
v_4	$\langle 1-6\rangle$	4
v_5	$\langle 1-2, 4-6\rangle$	1
v_6	$\langle 1-2, 6\rangle$	0
v_7	$\langle 4\rangle$	0

Table 11: Compressed list content and the space saved by using them.

6. EXPERIMENTS

Introduce what data is used and how we generate synthetic data Introduce standard test parameters for the tests to follow.

Parameter	**Meaning / used for**	**Standard value**
Mapfile	The map which the test is performed on	
NumQueries	Number of SP queries in test	
QuerySet	which dataset is used to provide queries	
TrainSet	For generating region statistics, which dataset is used	
CacheSize	Size of cache in bits	
cacheType	Type of cache representation (list/graph)	
kD-tree	Hight of the kD-tree	
avgLenght	Average length of a shortest path	

Write experiments to examine performance of goal 1 & 2 Test ideas (several ideas may be combined, like item 1 can be done on all datasets from item 2):

- increase kd-tree hight from 0-10
- different maps
- compare cache type performance
- compare with baseline methods.
- vary the cache size
- vary number of queries.

7. CONCLUSION AND FUTURE WORK

summation of each goal from the problem setting for each goal argue we solved it well (recap key results and partial conclusions from applicable sections).

8. REFERENCES