Fall 2010

February 28, 2011

Jeppe Rishede Thomsen
Department of Computing
Hong Kong Polytechnic University

Abstract

We develop an a caching method for shortest path queries.

1. Introduction

1.1. idea for running example

A user issues two queries as seen in fig. **??**, **??**, and 1 . We can then argue, using these figures and the map in fig. 3 about all the optimizations we plan to use. (same queries in all figures, and they corrospond to map)

2. Motivation

Shortest Path (SP) calculation is much slower than retrieval from cache (**TODO:** ***figure out how by how much (possibly for more than one SP algorithm)***)

Users want fast responsetime (and Shortest Path (SP) can do this cheaper with caching [**?**])

SP service providers want to spend less money on hardware (less computation -> less hardware)

it is not feasable to store all possible SPs (even if considering OSS)

Develop a caching scheme useable in domains wiht large set of opssible cacheable items and little or no locality in new items added to system, or suggested added to cache.

No previous work has been done on cacheing SP query results. interesting problem

More users now have a GPS and Web enabled mobile devices which makes SP services more popular, but also require them to serve much larger set of users.

3. Problem

3.1. Problem setting

We assume a setting where owners of mobile, positioning enabled, devices want route planning assistance. We assume users prefer online route planning services over offline solutions. We expect users to use network enabled capable of determining and visualizing users location and route. Users want fast response times from online services, comparable to using an offline application [**?**] Using a cache reduces the computational burden [?] on an online service, providing faster end-user response time [?] by both freeing up computational resources to calculate SP routes, as well as being able to immediately provide the SP result from the cache. We consider only server side caching..

3.2. Architecture

The system architecture of what we propose is shown in fig. 1. The example architecture depicted in fig 1 shows every major step (fig. 1A-E) that our system might execute when a new SP query is submitted to the system (fig. 1A). After a SP query is submitted to the system (fig. 1A), the cache (fig. 1B) is then queried to check if the result is in the cache. If there is a cache hit the system will immidiately return the result from the cache (fig. 1E). If the queried SP result is not in the cache, the system will detect a cache miss and calculate the result (fig. 1C) and immidiately return it to the user (fig. 1E). After returning the calculated result the system will use its cache policy to evaluate if the new result should be added to the cache, possibly expunging some other result from the cache, or discarded (fig. 1D)

TODO: ***add about text: space per edge***

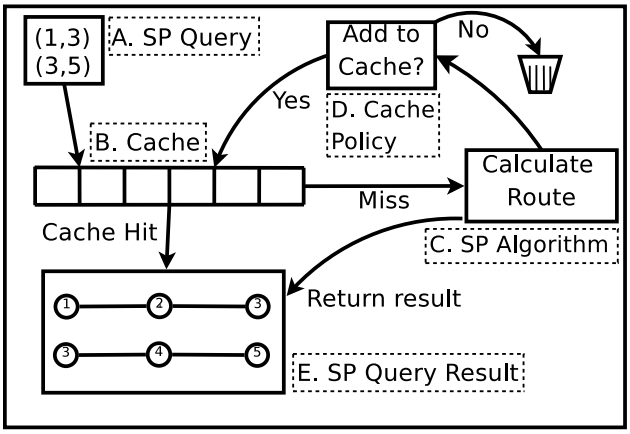

Figure 1. System architecture

Symbol	Meaning
SP	Shortest Path
LRU	Least Recently Used
FIFO	First In First Out
OSC	Set of edges
SPS	SP subpath Sharing

Table 1. Table of symbols and notation

3.3. Optimization goals

The overall goal, and most important measuring point to evaluate our success, is the reduction in CPU time (execution time) used compared to the same workload without any cache or optimizations.

At any SP service provider the CPU time needed is roughly divided into the execution of the SP algorithm and overhead related to query processing and system maintenance.

We will be working on optimizing two sub-goals:

1) Reducing the total time executing the SP algorithm.
2) Reducing the total time spent on overhead.

As stated in sub-goal 1 we want to reduce reduce the time spent executing the SP algorithm. We choose this goal as the SP algorithm is usually the single most CPU intensive task at a SP service [?] and by using a cache we should be able to find a direct correspondence between cache hits and the total CPU time of the SP algorithm.

In a SP service without a cache there should a small amount of overhead related mainly to handling the incoming queries and system maintenance. Sub-goal 2 is about reducing this overhead. We want to reduce the overhead as adding a cache to a SP system may add more overhead related to maintaining the cache integrity.

It is important that the increase in overhead does not grow larger than savings archived from the reduced time the SP algorithm needs to be run.

4. Baseline Competitors

To show the performance gain archived by our proposed solution in section 5 we use 2 baseline competitors. The 2 competitors are chosen as they use simple and well known caching techniques. Both competitors take advantage of the optimal substructure property of the cached shortest path items.

4.1. LRU

LRU - Least Recently Used. This competitor uses the LRU cache replacement policy which gives each cache item a timestamp when it is added to the cache, and updates the timestamp if the cache item contributes to a cache hit. When a SP query does not generate a cache hit, the new SP will trigger a cache replacement. LRU simply throws out the cache item with the lowest timestamp and adds the new cache item with the current timestamp.

4.2. FIFO

FIFO - First In First Out. FIFO replaces, as the name suggests, the oldest cache item when a new SP item is calculated and needs to be added to the cache. FIFO is the simplest of the suggested cache replacement policies.

5. Contribution

The techniques proposed here will each present how, and in what way, they can contribute to fulfilling the goals set out in section 3.3.

All cache replacement techniques presented takes advantage of the optimal substructure property of shortest path, so all items in the SP cache can also answer queries which only need a part of the shortest path.

5.1. Cache Replacement Policies

By using a cache a replacement policy, we try to directly shorten the total execution time by reducing the number of times the SP algorithm has to be executed. The execution time is shortened every time we have a cache hit since the SP algorithm[1] does

1. This can be any SP algorithm, and we a just referring to the general set of SP algorithms

i	item freq	item length	C(l)
1	11	10	100
2	10	15	200
3	2	15	2000

Table 2. SP queries (i), their frequency, length, and cost of calculating result($C(l)$)

not need to run, and SP algorithms usually are quite expensive in terms of the number of calculations and/or comparisons it needs to do to produce a result.

5.1.1. Optimal Substructure Cache (OSC). The basis of OSC is two scoring mechanisms which can both be added or multiplied to each other to assign each cache item a score. OSC does cache replacement based on the calculated score. If a new cache item get a lower score than any of the existing cache items it will not replace any existing cache items and will be discarded, if one or more items exist in the cache which have a lower score than some new candidate, then the item with the lowest score will be replaced (Fig. 1, step D).

5.1.2. Scoring Mechanism. The scoring is done by summing up: the number of times a cache item has formed the basis of a cache hit; the number of times each node in cache item has been the source/target nodes in a query, or member of a query result. By taking the length of a cache item (number of nodes in the SP) and multiplying it on the previously calculated sum we get the full score.

In OSC we use a scoring mechanism to rank the usefulness of cached items. Table 2 shows 3 queries (i) seen at different frequency and with different length. C(l) calculates the number of vertices visited when finding a shortest path with a SP algorithm. Based on these numbers it is then possible to calculate the benefit of adding one of these SP queries to the cache. E.g. Item 1 has higher frequency than item 2, but the saving of putting item 2 in cache is much higher:

item one cost: $11 * 100 = 1100$ and item two cost: $10 * 200 = 2000$. Assuming a cache hit cost 1, then adding item one would save = $1100 - (100 + 10 * 1) = 990$, and item two would save $2000 - (200 + 9 * 1) = 1791$. the equation is: $(total\_cost\_without\_cache) - (cost\_of\_one\_Dijkstra\_run + (total\_runs - 1 * cost\_of\_cache\_hit))$

TODO: ***C(l): modify Dijkstra implementation to return this number***

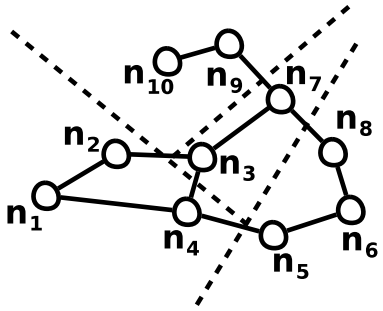

Figure 2. Partitioned Map

5.2. SP subpath Sharing (SPS)

If space consumption is a very high priority then SPS is one option to alleviate the problem. SPS changes the cache structure from $n_1, ..., n_n$ to $CI_{k[s-node]}, CI_{k[e-node]}, n_{e-node+1}, ..., n_n$.[2] I.e. including the start-/end-node of a range from another cache item. The advantage of doing this is a, possibly large, reduction in the space requirements for the cache. Maintaining such a cache would be quite expensive in terms of computation when changes occur, though the space saving would allow for more cache items which again would reduce the total running time. We will later show wether this overhead can really be offset by the additional number of cache items which would fit into the cache.

5.3. Partitioning Map

Partitioning of map into large region to quickly be able to discard items in the cache, leading to less work when trying to find a cache hit. By discarding possible SP cache candidates early we may save some time and computations.

TODO: ***improve fig. 2***

5.4. Cache structures

We will consider 3 basic cache structures, each with optional ways of using or implementing them. The 3 cache structures are briefly:

- A single array. A cache lookup means scanning the array.
- A map on an array, quickly identifying valid candidates for a cache hit.

2. This example only shows how to share something in the front of an item, though adding them after or in the middel does not matter.

- Using one of the two ideas already mentioned, but changing structure of the cache items (SP results) to use SPS, sharing sub-parts, to conserve space.

5.4.1. Array, Direct Access. The simplest way to implement a cache is to just store all cache items in an array. An example of how the cache works is presented in figure 5, part A,D, and E. In figure 5A two SP queries, Q1 & Q2, are submitted. By scanning the array in 5D item 3 and 4 are identified as being able to answer Q1 and Q2. Cache item 3 contains a superset of the set of nodes needed to answer Q2 so in 5E the answers are refined and sent back to the user.

While this cache structure is very simple and there for incurs virtually no overhead in terms of maintaining the structure, then the array needs to be scanned each time a new query is submittet to the system, and the entire array will be scanned if the query can not be answered by the cache. This cache structure has a high overhead because it needs to scan the array, and if the cache is large it is not likely to be an efficient solution or fulfill subgoal 2 very well (see Sec. 3.2).

5.4.2. Array with Map, Indirect Access. By adding a map with an inverted list on top of the array, holding the cache items, we can answer queries in constant time. The inverted list maps node ids to the cache items in which they are present. Figure 5A-E shows how this works. In 5A two queries are submitted. For each query we make a lookup in the map (fig.5B) for the start and end node of the query. In 5C We then take the intersection of the possible cache items containing a shortest path with the start and end nodes of each query. If the were to be empty we would immediately know the cache can not answer the query. In 5D we directly access the cache items identified in 5C and refine them in 5E before sending the answer back to the user.

This cache structure require some additional work to maintain, as the map needs to be updated each time the cache content changes i.e. when cache items are replaced, but otherwise incur very little overhead. This structure is expected to perform quite well and be very good at helping to fulfill subgoal 2 (see Sec. 3.2).

5.4.3. Array with Shared Sub-paths. To support the SPS idea presented before, new node type has been introduced (fig. 4). Previously the cache items only consisted of a list of node ids. To support the idea of SPS two new node types has been introduced, node 0 and 1. Node type 0 is only a node id with a type 0. Node type 1 consist of a cache item id and the id of the start and end node which is shared between the two cache cache items.

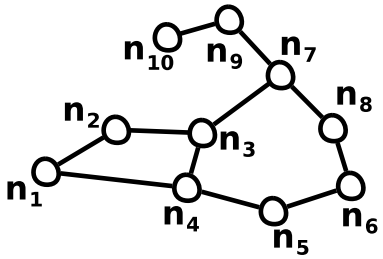

Figure 3. Map

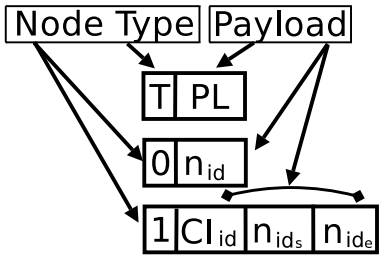

Figure 4. Node types

can be implemented on top of either of the two previously presented cache structures, as it only changes the internal way of storing the cache items. This idea incurs a substantial overhead each time a cache item has to be replaced, as it is needed to check whether the replaced cache item is sharing or being shared with other cache items. This method does however also free up more space, allowing for more cache items and hopefully fewer cache replacements.

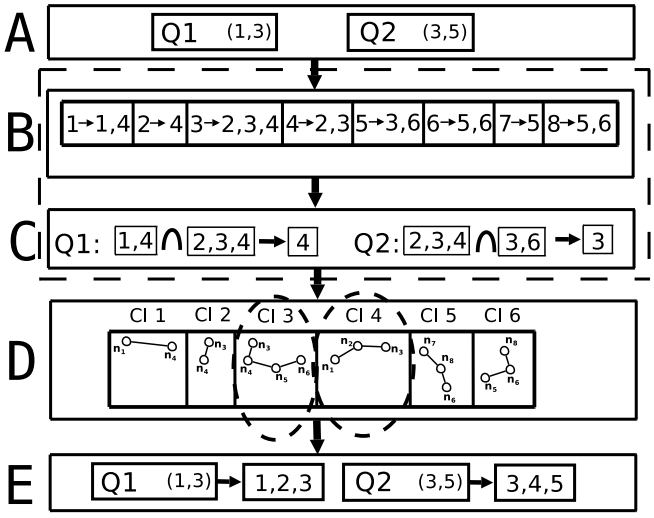

Figure 5. Cache access