Fall 2010

February 25, 2011

Jeppe Rishede Thomsen
Department of Computing
Hong Kong Polytechnic University

Abstract

We develop an a caching method for shortest path queries.

1. Introduction

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..

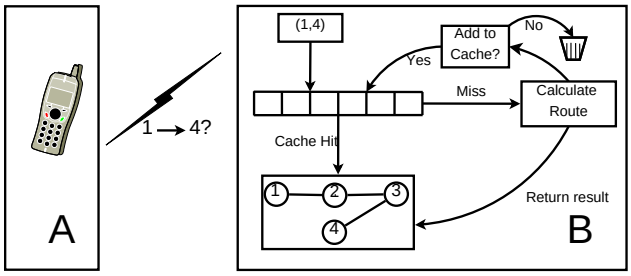

Figure 1. Advanced graph

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

3.2. Architecture

The system architecture of what we propose is shown in fig. 1

To enable us to better argue about the real performance, as well as the strong and weak points of our ideas, we use a gaussian distribution model for the queries together with a graph with 6100/7035 vertices/edges.

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

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.

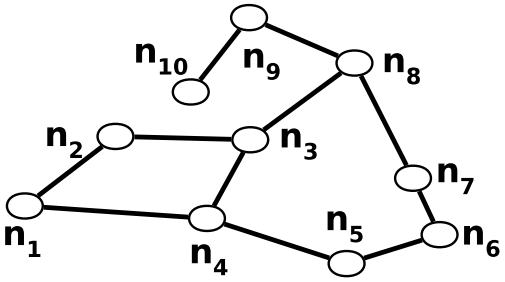

Figure 2. Small map

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

5.1. Solution Proposals

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.

5.1.1. 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 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.

Optimal Substructure Cache (OSC). Optimal Substructure Cache (OSC) uses 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

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($C(l)$)

the cache which have a lower score than some new candidate, then the item with the lowest score will be replaced.1

TODO: ***write argument why to use + or * when calculating score.***

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 the SP in a cache item and multiplying it on the previously calculated sum we get the full score.

Scoring Mechanism. 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***

5.1.2. Shortest Path (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.

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.

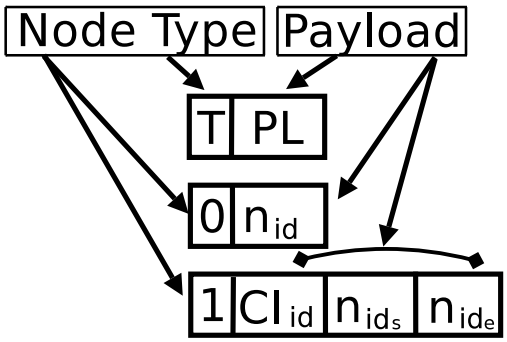

Figure 3. Node types

how does it work? what goal does it fulfill? how does it fulfill the goal?

5.1.3. 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.

5.1.4. Cache structures. TODO: ***flesh out, add figures to clarify the difference***

- array, no utilization of optimal substructure property of SP items
- array, utilizing optimal substructure property of SP items
- 2D array, to more quickly (and cheaper) identify cache hits (expensive to maintain)

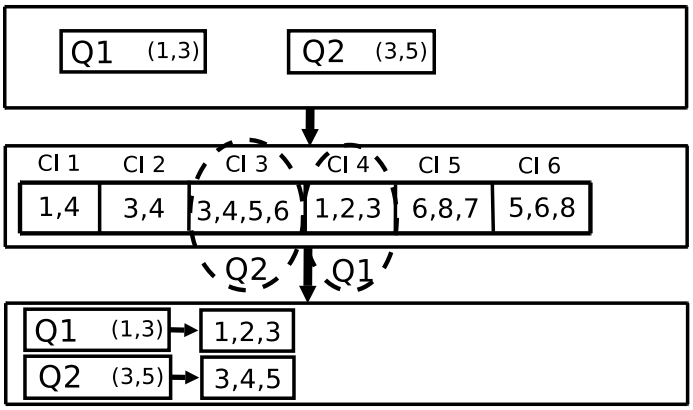

Figure 4. Direct cache access

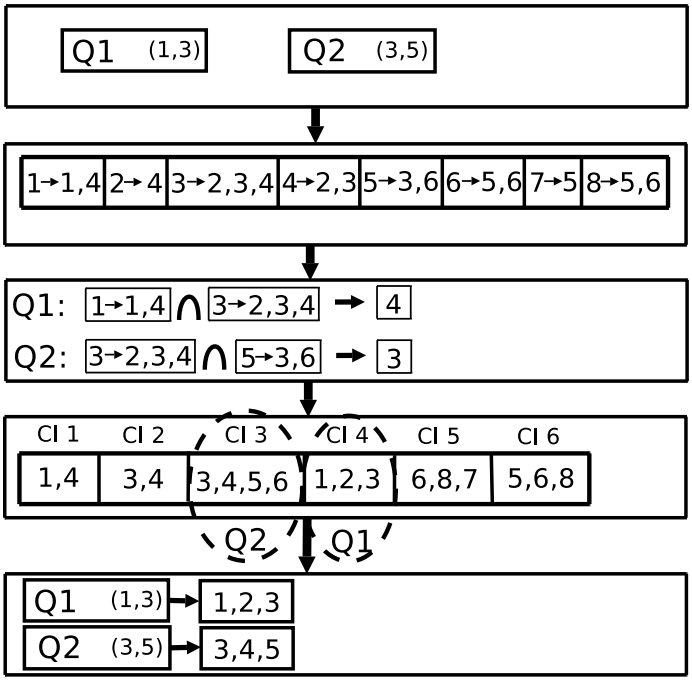

Figure 5. Indirect cache access