Efficient Data Management in Support of Shortest-Path Computation

Alexandros Efentakis
Institute for the Management of Information Systems
G. Mpakou 17
11526 Athens, Greece
efentakis@imis.athena-innovation.gr

Dieter Pfoser
Institute for the Management of Information Systems
G. Mpakou 17
11526 Athens, Greece
pfoser@imis.athena-innovation.gr

Agnès Voisard
Free University Berlin
Takustr. 9
14195 Berlin-Dahlem, Germany
Agnes.Voisard@fu-berlin.de

ABSTRACT

While many efficient proposals exist for solving the single-pair shortest-path problem, a solution that sees the algorithmic solution in combination with efficient data management has received considerably smaller attention. This work proposes a data management approach for efficient shortest path computation that exploits road network hierarchies and allow us to minimize the portion of the network that is kept in main memory.

The proposed approach is insensitive to network changes as it does not rely on any pre-computation, but only on given road network properties. In that we specifically target large road networks that exhibit a high degree of change (e.g., OpenStreetMap). Extensive experimental evaluation shows that the presented solution is both efficient and scalable and provides competitive shortest-path computation performance without requiring a preprocessing stage for the road network graph.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms; H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Algorithms

Keywords

Shortest Path computation, HBA*, Cell Manager, OpenStreetMap

1. INTRODUCTION

Although many previous publications introduced fast algorithms for shortest-path (SP) computation ([4, 9, 6, 7, 12, 13, 2, 5, 14]), most authors assume that the entire road network graph resides in main memory. Additionally, many preprocessing algorithms, such as the ALT [6], Highway Hierarchies [17] or Arc-flags [13] require the storage of additional information related to the algorithm (landmarks, shortcuts, arc-flags) to complement the original road network graph. Others like Contraction Hierarchies [5] compact the original road network for SP computation but still need additional information to output the actual shortest path. For a broad overview of shortest path speed-up techniques up to 2008, one can refer to [18].

On the other hand, through crowd-sourcing, road network graphs are evolving rapidly (150,000 new ways are added to Open Street Map [OSM] data per day) and therefore preprocessing algorithms have to recompute their data structures frequently to keep providing accurate results. Additionally, edge weights may change over time to represent fluctuations in traffic conditions. These weights are common referred to as speed profiles. Therefore different versions of the same road network are required for SP computation depending on the time of the day and vastly increasing the memory size required to store the road network graph.

Compared to the literature devoted to engineering algorithms for shortest paths, little attention has been paid to engineering an efficient storage mechanism, i.e., the road network graph is not completely loaded into main memory but is instead fetched from secondary storage. Using a naive mechanism, which fetches nodes as required during SP computation is not an option since a Dijkstra search within a city may require many thousands node expansions. Performing so many requests within a fraction of a second is far too challenging for any database or file system. Consequently, road network data should be fetched in tiles, to minimize the number of queries to secondary storage.

Such an efficient storage manager for SP algorithms will be invaluable in the traditional *routing server* scenario, with a single computer (or a cluster thereof) serving routing requests from many clients. As more road network data through crowd-sourcing is freely available, it will be difficult to store the entire dataset (world) in main memory. Here, the algorithm's data structures are kept in main memory and road network data is fetched on demand. The need for handling many parallel requests points us towards a database-backed mechanism that can efficiently handle the number of requests necessary to load road network data into main memory.

The main contribution of this paper is to propose an efficient storage manager to support SP computation in connection with large road network datasets stored on disk. Although this mechanism is *routing algorithm neutral*, i.e., it would work with any traditional routing algorithm (Dijkstra, A^*), it will be significantly faster and more efficient when combined with a hierarchical algorithm, such as HBA^* [14]. The objective of this work is not to showcase a new SP algorithm that outperforms existing solutions, but to introduce an effective data management mechanism in combination with an hierarchical routing algorithm like HBA^* to minimize the portion of the road network that is kept in main memory. Since the road network graph is used as is, no SP algorithm-specific preprocessing is required and therefore our solution can be used even with dynamic speed profiles, i.e., a dynamic weight database of the road network graph that changes over time [15].

2. THE HBA* ALGORITHM

In this section we will describe some basic concepts and we will present the basic properties of HBA* algorithm, a hierarchical bidirected A* variant initially presented at [14].

2.1 The Single Pair Shortest-Path Problem

A road network is modeled as a directed graph $G = (V, E)$, whose vertices/nodes V represent intersections and edges E represent links between intersections. Additionally, a real-valued weight function $w : E \rightarrow \mathbf{R}$ is given, mapping edges to weights, with weights typically corresponding to travel times. In this paper we will deal with the *single-pair shortest path problem* (SPSP) of finding a shortest path between a source vertex s and a target vertex t of the road network directed graph.

The SPSP problem on a graph with non-negative edge weights $w(u, v) \geq 0$ can be solved by applying Dijkstra's algorithm [4]. By exploiting knowledge about the structure of the graph, the A* algorithm [9] selects the next node u to be expanded by using the cost $d(u)$ of a shortest path from s to u (Dijkstra) in combination with the *estimated cost to the goal*, $h(u, t)$. A* is guaranteed to find the optimal solution provided h never *overestimates* the real cost of reaching the target.

In [14] the HBA* algorithm was introduced to efficiently solve the SPSP problem by exploiting *road network hierarchies* to achieve faster computation times and efficient memory usage. HBA* is a variant of a bi-directed A* algorithm for SP computation in a hierarchical road network. What follows is a brief discussion of road network properties and a short description of the algorithm.

2.2 Hierarchical Road Networks

Roadmap data available from vendors usually provides road category information for each road (edge). A typical example of road categories on a road network, may include categories such as "Freeway" or "Local Road". In the road networks used (Section 4.1), low numbers were assigned to higher road categories, i.e., the highest road category was "1: Freeway" and the lowest was "13: Other Road" for the OSM road network.

The road category information gives rise to interpret the network as a *hierarchical road network*: Level L_i of the road network consists of all road segments of road categories $j \leq i$, including all nodes incident to those segments. Let $G = (V, E)$ be the whole road network with vertex/node set V and road segment/edge set E, and let $L_i = (V_i, E_i)$. Then $V_i \subseteq V_{i+1}$ and $E_i \subseteq E_{i+1}$ for all i, and $V = \cup_i V_i$ and $E = \cup_i E_i$.

The significance of hierarchical road networks is evident on how roads of varying importance would typically be used in a *routing task by a person*. First, he searches for major roads connecting the two areas of interest and then he finds access roads to those major roads. The basic question is how we can mimic such route finding behaviour in SP algorithms.

2.3 HBA* algorithm's details

HBA* algorithm emulates this typical route finding behaviour by alternating between running an A* algorithm from s to t (forward search), as well as an A* algorithm from t to s (reverse search) in the reverse graph. Each of those searches utilizes *hierarchical jumping* (HJ), a technique that favours the use of higher category roads to reduce the overall search space and improve the running time of SP computation.

On hierarchical jumping we split roads in two different *cell levels*. The first *upper* cell level (UCL) includes only roads of higher importance (highways, major roads etc) and the second *lower* cell level (LCL) includes the entire road network. If one of the opposite A* searches expands a node and the edge leading to this particular node belongs to UCL, it ignores all outgoing edges (smaller roads) that do not belong to UCL. Therefore only nodes being reachable by same or higher category edges are visited during node expansion. Additionally, if one of the two A* searches is on UCL and the opposite A* search is not, it freezes until the opposite search reaches that same level. That partially ensures that the two searches meet at the UCL i.e., at a higher category road. By adjusting which road categories are assigned to UCL, we can manipulate the quality of results that HBA* produces in contrast to the number of nodes expanded. Therefore, if all available road categories were assigned on UCL, the HBA* algorithm would fall back into a standard bi-directed A* search.

If π_f(u) is the Euclidean distance of node u from target node t divided by the maximum speed of the road network (travel time metric) and π_r(u) is the Euclidean distance of node u from start node s divided by the same maximum speed, π_f and π_r give lower bounds for forward and reverse search respectively. Similar to [10], HBA* uses $p_f(u) = \frac{\pi_f(u) - \pi_r(u)}{2}$ as the potential function for the forward search and $p_r(u) = -p_f(u)$ for the reverse one, which are *feasible* and *consistent* and therefore provide optimal results in a bi-directed A* search.

Analogous to [6] the HBA* algorithm maintains the length μ of the shortest path seen so far. Initially, $\mu = \infty$. When an edge (u, w) is scanned by the forward search and w has already been scanned in the reverse direction, we know the length $d_s(u)$ of path $s - u$ and length $d_t(w)$ w-t path respectively. If $\mu > d_s u + w(u, w) + d_t w$, we have found a shorter path than those found before, so we update μ and its path accordingly. Similar updates are done during the reverse search. The HBA* algorithm terminates when the search in one direction expands a vertex that has already been scanned in the opposite direction. By using the aforementioned techniques, the HBA* algorithm mimics human driving behaviour, i.e., when given the choice, it selects higher category roads to reach a destination.

Since we effectively reduce the overall search space with HJ, the algorithm's performance in terms of memory consumption and computation speed is dramatically improved. On a typical route, during the middle portion of the search, the number of nodes expanded is considerably reduced. This is further evident when examining the number of nodes in a per-category basis of a road network. More than 50% of nodes on typical road networks respectively belong to minor roads! Thus, by eliminating this portion of the road network the SP search is considerably accelerated.

Still, hierarchical jumping in SP computation may eliminate candidate solutions and provide suboptimal results. To address this issue, the concept of *initialization buffer* was introduced in [14]. Assuming that we want to compute a SP from node s to t, the initialization buffer $I(\epsilon)$ around both s and t prevents the use of HJ for all vertices $u \in I(\epsilon) : \{dist(u, t) < \epsilon \vee dist(u, s) < \epsilon\}$. In [14] a simple Euclidean distance measure was used to quantify the initialization buffer. Although this approach was feasible, for travel time metric graphs it is better to directly use the cost $d(u)$ to reach this node from origin of search.

What needed to be established was the optimal initialization buffer ϵ in terms of cost $d(u)$ rather than distance. Extensive experimentation (omitted due to space limitation) for three road networks showed that increasing ϵ leads to a logarithmic increase in the quality of HBA* results (thus decreasing the gap between HBA* and optimal Dijkstra results) and results in a linear increase in the number of nodes expanded. The experiments also showed that, $\epsilon = 300s$ is a good compromise for HBA* to (a) find almost optimal results (less than 0.30% worse than optimal) and (b) still expand less than 40% of the nodes bi-directed Dijkstra does. Note that 300s is also a logical time span to define *neighbourhood searches*. Consequently, 300s initialization buffer was used in all experiments of Section 5.

3. HIERARCHICAL DATA MANAGEMENT

The objective of this work is to develop an efficient storage manager for hierarchical road network data for SP algorithms. This section focuses on the design of respective data management techniques that exploit the hierarchical structure of road networks.

In the *routing server* scenario, a server receives routing requests from numerous clients. The server is assumed to have enough main memory (MM) for the routing algorithm's data structures but road network data is fetched on demand. We propose an effective storage manager that fetches road network data packed in *cells* from secondary storage for use with SP routing algorithms. A database was also chosen as the underlying storage mechanism to experiment with varying cell schemas and provide support for parallel requests. In this context, we refer to this storage manager as the *Cell Manager* (CM). In a nutshell, we model the road network graph as a set of hierarchical cells. In connection with the HBA* algorithm and its HJ mechanism, hierarchical cells reduce the portion of the road network that is kept in main memory.

3.1 Hierarchical Partitioning

Hierarchical jumping as used by the HBA* can be conceptualized as shown in Figure 1. Here, the same road network is shown at different *levels* of abstraction. The network on top includes only major roads, while the network at the bottom includes all available roads. On a typical car route, a driver initially moves on minor roads, then moves to major roads and essentially chooses to ignore lower category roads until getting close to the destination.

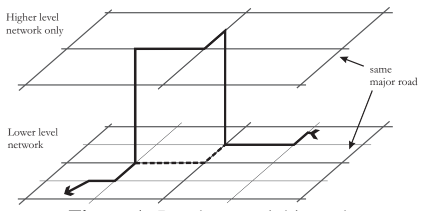


Figure 1: Road network hierarchy

In order to exploit this behaviour in data management, we partition the road network graph, both, with respect to space and hierarchy. The road network is partitioned into a regular number of cells. Each cell should typically contain the same number of nodes and edges. *Hierarchical partitioning* refers to considering edges of a certain category (highways, neighbourhood roads etc.) for a specific *spatial partitioning* resulting into two separate *cell levels*. A cell belonging to the Upper Cell Level (UCL) contains high capacity roads (low category numbers). A cell of the Lower Cell Level (LCL) contains all available roads.

The mapping of road categories (those defined by the road network data vendor) to UCL may change and depends on the specific road network (size and roads distribution per road category). In our experimentation we use commercial road networks (Athens and Vienna) for which road categories 1 to 5 are mapped to the UCL, while for the OSM road network of Germany road categories 1 to 7 belong to the UCL. This flexibility is necessary, since different road network vendors use a different categorization for road networks.

3.2 Spatial Partitioning

Apart from the hierarchical decomposition of the road graph we need to consider the spatial partitioning of the graph, i.e., how many nodes a cell contains. Too many nodes per cell mean unnecessary data is moved through the network, too few and the number of requests to the CM increase. The authors in [19] suggest medium-sized cells as the best choice. In our case, experimentation showed that we obtain best results for each cell containing roughly hundred nodes.

Having established the best choice number of nodes per cell v, we need to enforce this choice for both cell levels. The number of cells per cell level $c(l)$, with $l = \{l \in L; L = \{UCL, LCL\}\}$ is calculated by the formula $c(l) = \frac{n(l)}{v}$ where $n(l)$ is the total number of nodes assigned to cell level l. Consequently, $c(l)$ is related to the cell level's available nodes.

Having established some soft limits for $c(l)$, we have to resolve how nodes are assigned to cells. We can either use a *regular rectangular grid* where all cells of the same cell level have the exact same size for a particular road network or use a tool such as *METIS* [11] for partitioning the road network graph for each cell level. METIS is a set of serial programs for partitioning graphs and reducing communication volume between cells (i.e., minimizing cross edges).

Since HBA* is exploiting hierarchies will mostly request UCL nodes. Provided that UCL cells are larger (since they cover a bigger geographical area) but contain only a subset of all available nodes and edges (since they include only major roads and their intersections), the total number of cells loaded will be significantly lower when compared to Dijkstra's algorithm. Figure 1 illustrates the cells being loaded for a typical route. All cells contain roughly the same number of nodes and edges. Larger cells belong to the UCL and cover only high-capacity roads. During the middle portion of the search (HJ), only the UCL portion of the network is evaluated and only UCL cells are retrieved, while for the beginning and the end of the search small cells of the LCL are fetched.

The obvious advantage of the METIS partitioning is that truly all cells almost contain the same number of nodes contrary to the simple rectangular grid where certain cells are empty and others are overpopulated, including up to 1000 nodes, instead of the desired range of 100 to 200 nodes per cell. The partitioning schemas and their performance is described in Section 4. Partitioning the road network by either method requires minimal effort (the METIS splitting even for the larger road network of Germany takes less than 30s on an average workstation) and time since it only takes into account the nodes' position (rectangular grid) or connections (METIS). In that sense, it does not depend on the underlying SP algorithm.

3.3 Cell Manager

The Cell Manager (CM) implements a storage manager for a road network based on the aforementioned spatio-hierarchical partitioning model. CM manages the creation and management of the main memory graph data structure for the SP algorithm based on node requests and by retrieving the respective cells from secondary storage. CM maintains a least-recently used (LRU) cache of cells that can hold a limited subset of road network graph data (according to MM limitations). The LRU policy aligns itself well with the SP algorithm, allowing us to discard "aged" and not frequently used cells safely. The architecture of the Cell Manager is shown in Figure 2. The LRU cache of cells may only store a subset of all available cells of both cell levels combined. Additionally, each

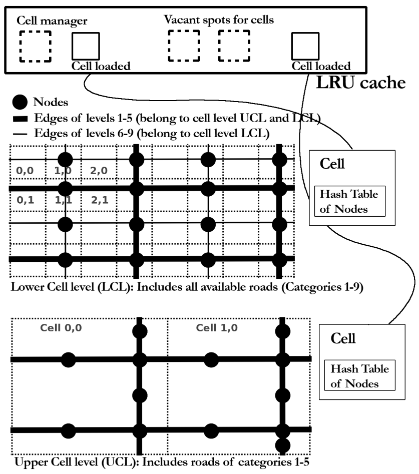

Cell manager
Vacant spots for cells
Cell loaded
Cell loaded
LRU cache
Nodes
Edges of levels 1-5 (belong to cell level UCL and LCL)
Edges of levels 6-9 (belong to cell level LCL)
0,0
1,0
2,0
0,1
1,1
2,1
Cell
Hash Table of Nodes
Lower Cell level (LCL): Includes all available roads (Categories 1-9)
Cell 0,0
Cell 1,0
Cell
Hash Table of Nodes
Upper Cell level (UCL): Includes roads of categories 1-5

Figure 2: Hierarchical partitioning of road network

node contains information about its neighbor nodes and the cell they belong to (mainly for cross edges).

With HBA*, cells belonging to UCL are never unloaded, since they are frequently used. As soon as the SP algorithm expands additional nodes, CM implicitly loads all the necessary cells (if they are not present in the cache). The CM knows the neighbouring cells of a node. Thus, should a node outside the current cell be expanded and provided its cell is not present in CM's LRU cache, the cell is retrieved from secondary storage. It is easy to use CM with traditional routing algorithms like Dijkstra or A* . Here however, only the LCL (including all edges) is used and we cannot exploit CM's hierarchical features. In that sense, CM is *routing algorithm neutral*.

With respect to the database schema, all cells of both cell levels are stored in two separate db tables and are indexed by cell IDs for fast retrieval. Each cell is stored in the database as a set of records, one for each node. For each node, edge information is stored as a set of neighboring nodes. For space reasons this data is not stored as separate attributes but as a single compact string for each edge. CM is also *database neutral*, since it does not use any vendor specific data types and therefore can be implemented on any standard RDBMS. On our current implementation PostgreSQL[16] was used due to its overall enterprise spatial capabilities.

In conclusion, CM provides a storage manager for road networks based on spatio-hierarchical partitioning and implementing a LRU buffering strategy that fits to the road network traversal of the hierarchical shortest-path algorithms.

4. EXPERIMENTAL SETTING

Our experimental evaluation compares the performance of bi-directed Dijkstra and the HBA* algorithm using the Cell Manager as the storage manager. The comparison will be in terms of (i) loaded cells, (ii) total nof nodes loaded in MM and (iii) computation time. Two separate tiling schemas, rectangular grid and METIS partitioning, will also be compared in for efficiency and speed.

Experiments have been conducted on a Intel Core 2 Duo CPU clocked at 3.00 GHz with 8Gb main memory, running Ubuntu 10.10 64bit (kernel 2.6.35 − 28). The HBA* algorithm and the Cell Manager have been implemented in Java and 64bit PostgreSQL 9.0.3.

4.1 Road Networks

We used two commercial city size (Athens and Vienna) road networks and one crowdsourced country size network (Germany). The rationale behind this was to assess the performance of the storage manager and SP algorithm for commercial vs. user-generated networks as well as small-scale to large-scale networks. The two commercial networks [20] cover the greater metropolitan areas of Athens, Greece and Vienna, Austria and comprise of nine categories, with 1 corresponding to highways up to 9 for dirt roads.

	Athens	Vienna	Germany
Road categories	1-9	1-9	1-13
Total nof nodes	140,633	55,954	3,554,665
Total nof edges	206,428	74,783	4,375,777

Table 1: Available road categories and sizes of road networks graphs

	Road categories assigned at UCL	Nodes at UCL	Nodes at UCL / Total nodes
Athens	1-5	20,101	14.3%
Vienna	1-5	17,199	30.7%
Germany	1-7	695,251	19.6%

Table 2: Road categories assigned to cell levels and nodes distribution

The country-scale network covering Germany was derived from OSM data provided by Cloudmade [3]. Using a user-generated road network that frequently gets updated plays to the strengths of the HBA* algorithm when used in connection with the Cell Manager, since this approach does not rely on preprocessing the road network graph.

For all three road networks a *travel time metric* was used by assigning typical speeds to each road category. Table 1 summarizes the properties of the three road networks used.

4.2 Cell Partitioning

Two cell levels were used for all road networks . The Upper Cell Level (UCL) included major roads (road categories 1-5 for Athens, Vienna and road categories 1-7 for Germany). The Lower Cell Level (LCL) includes the entire road network graph. Assigning road categories to cell level UCL conformed to the actual meaning of road categories in the respective road networks, i.e., road categories 1-7 for Germany are basically the same roads as road categories 1-5 for Athens or Vienna). The tiling schemas used for each road network aim at having roughly 100 - 200 nodes in a single cell (cf. Table 3).

Since the rectangular grid ignores graph structure or density of nodes, certain cells are overpopulated (>1000 nodes per cell) and more than 20% of the cells at each cell level are empty . In contrast, with METIS partitioning the number of nodes per cell was almost constant. That is why the METIS partitioning performed better in every experiment we conducted.

The proposed cell hierarchy is directly linked to the operation of the HBA* algorithm and road network properties. UCL nodes comprise only 14,3%, 30,7% and 19.6% of the total number of nodes for Athens, Vienna and Germany respectively (Table 2). Since HBA* uses higher category roads, it will mostly use UCL (outside the initialization buffers area) and reads fewer cells from secondary storage. This behavior gives HBA* the scalability and speed in memory constrained environments for handling large road networks. Additionally in Vienna, where UCL nodes comprise 30% of the network, HBA* in almost all cases finds optimal results (Table 4).

4.3 Shortest-path Queries

Two different SP query types were used in the experimentation (cf [19]). In the *cold query* case for the 1000 random queries that

	Part. schema UCL	Part. schema LCL	Avg. nodes per cell UCL	Avg. nodes per cell LCL
Athens	13 x 13	39 x 39	118	92
Vienna	13 x 13	26 x 26	101	82
Germany	64 x 64	128 x 128	169	216

Table 3: Partitioning statistics

	Quality of Results (%)	Nodes Expanded (%)	Nodes Expanded
Athens	0.26%	38.93%	4,999
Vienna	0.07%	32.52%	1,708
Germany	0.29%	3.72%	11,188

Table 4: Algorithm comparison for the three road networks graphs

	Cells Loaded	Nodes loaded in MM	Computation time (%)	Computation time (ms)
Athens				
HBA* Regular	8.2%	113.9%	56.2%	56
HBA* Metis	45.51%	50.1%	41.6%	44
Vienna				
HBA* Regular	19.2%	86.2%	46.8%	21
HBA* Metis	46.6%	53.14%	43.3%	20
Germany				
HBA* Regular	5.3%	9.5%	3.9%	97
HBA* Metis	7.4%	7.1%	3.5%	91

Table 5: Results for rectangular grid and METIS (Cold SP queries)

are executed, the LRU cache (implemented by the CM) is cleared after each query. This is an efficient way to determine the cost of the first query in an un-initialized system. For *warm queries* the LRU cache is not cleared. This determines the "true" average query time for the routing server scenario.

In secondary storage experiments for mobile devices ([8],[19]), the cold query experiments were the most important ones (the device will not perform thousands of SP computations). In our case, since we emulate the *routing server* scenario, computation time of warm query experiments is more important, since the same server will satisfy thousands of SP requests.

5. EXPERIMENTAL RESULTS

What follows below are experiments contrasting the performance of the HBA* algorithm with that of bi-directed Dijkstra, when using the Cell Manager. In contrast to related work (cf. [19] and [5]), where outputting the complete shortest path is considered a separate task independent from the actual SP calculation, this task is included in all times reported in our experiments.

5.1 SP Quality

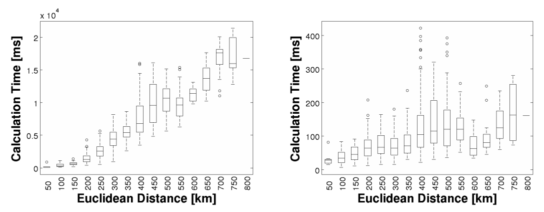
The first set of experiments contrasts the performance of HBA* SP computation with bi-directed Dijkstra in terms of quality of results and number of nodes expanded. These sizes are independent of the CM's tiling schema and depend only on the actual algorithms. Bi-directed Dijkstra is used as the benchmark in all experiments and relative percent measurements relate to the performance on this algorithm.

Although HBA* due to HJ does not guarantee optimal results, in reality the results it produces are close to identical to the optimal Dijkstra results. Table 4 shows that for Germany it produces on average 0.3% worse results. Even better numbers apply to Athens and Vienna (0.26% and 0.07% respectively). On the other hand, in Germany, HBA* expands only 3.7% of the nodes that bi-directed Dijkstra expands, by utilizing less than 20% of the road network.

5.2 Cold SP queries

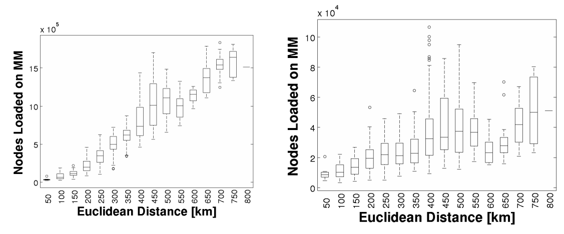
In order to assess the performance for cold queries, we compare the rectangular grid and the METIS partitioning, in terms of cells and nodes loaded. The former represents the number of requests to the CM, whereas the latter stands for the data fetching into memory. In addition, we also assess the computation time in each case. Bi-directed Dijkstra experiments were conducted using METIS partitioning. The results for 1000 random SP queries conducted for each of the three networks are presented in Table 5.

Results show that the combination of HBA* and CM provides average computation times of less than 100ms for Germany. This is achieved with no preprocessing and the entire road network graph stored on disk. On average for a SP query for Germany 27,000 nodes are loaded, which is roughly 0.8% of the total road network graph. Bi-directed Dijkstra on the other hand had an average computation time of 5.5s for Germany (*60 times slower*) and had to load on average 15% of the total road network.



(a) Bidirected Dijktra (b) HBA* (METIS)

Figure 3: Cold SP queries - computation times [Germany]



(a) Bidirected Dijktra (b) HBA* (METIS)

Figure 4: Cold SP queries - nodes loaded [Germany]

Another obvious result is that *METIS partitioning is more effective* and results in better computation times, e.g., by 5-12 ms for all networks. The METIS effectiveness is credited to the fact that all cells have about the same size and therefore the nof nodes fetched is minimal. For Germany METIS partitioning loads on average 27,127 nodes per search, whereas the regular grid feches an average of 35,380 nodes per search.

Fluctuations in computation times and fetched number of nodes in relation to Euclidean distance between origin and destination of search for Germany are presented in the box and whisker plots of Figure 4. The figures show experiments for bi-directed Dijkstra and HBA* with METIS partitioning. Athens and Vienna results are also similar.

Results show that the nof nodes loaded in MM follow similar pattern to computation times, i.e., the nodes retrieved from secondary storage are the main contributing factor to the performance of CM and SP algorithm. We also see that the larger the graph, the greater the advantage of our approach. This is due to the use of mainly UCL in path computation. The closer the origin and the destination are, the smaller is the advantage of a hierarchical SP algorithm.

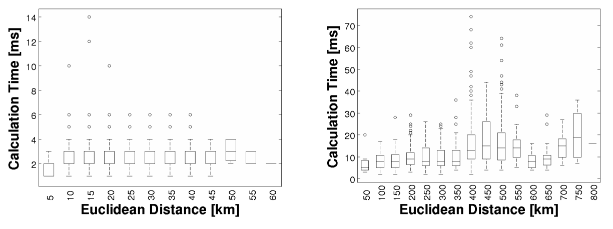
5.3 Warm SP queries

The previous section established that the combination of CM and HBA* is efficient for all road network graphs when all graph data is stored on secondary storage. Still, since we emulate the routing server scenario, those results are not the typical case. A routing server performs countless SP computations and therefore most popular cells are already loaded in MM. For these warm query experiments, we performed 1000 SP random queries to "warm up" the CM's LRU cache and then performed another set of 1000 SP queries, for which we then recorded the computation times. METIS partitioning was used in all experiments.

Table 6 gives average running times. This table should be com-

	Computation time %	Computation time (ms)
Athens	40.5%	6
Vienna	40.2%	2
Germany	3.4%	13

Table 6: Warm SP queries - Computation times



(a) Vienna (b) Germany

Figure 5: Warm SP queries - computation times

pared to Table 5. The avg. running time for Germany has been reduced from 91ms to 13ms, i.e., a speedup of 7. Similar speedups of 10 and 8 apply to Vienna and Athens, respectively. These speedups can be largely attributed to the reduced number of accesses to the CM.

The computation times as function of the Euclidean distance between origin and destination are shown in Figure 5 for Germany. We see that the combination of CM and HBA* for warm queries provides computation times similar to typical main memory SP algorithms. This is expected since HBA* utilizes mainly UCL, which contains a small fraction of total nodes and therefore is expected to fit in MM. Additionally, our approach requires no preprocessing and uses no pre-computed routes (and therefore requires no unpacking routines). This makes is suitable for dynamic networks in which either the edge weights change or the network graph itself (OSM). This creates a significant advantage for this approach over novel SP algorithms, like Contraction Hierarchies [5] or Transit Node Routing [2].

6. RELATED WORK

The idea of partitioning a graph for SP computation is not new. Maue et al. [12] partitioned the graph into clusters and precomputed distances between border nodes of clusters in order to prune their modified Dijkstra's algorithm. Möhring et al. [13] also divided the graph into partitions and gathered information on whether each edge is on a shortest path into a given region. For each edge this information is stored in a arc-flag vector to be used in their Dijkstra computation to avoid exploring unnecessary paths.

Both approaches considered METIS as a clustering algorithm and results are similar to ours in the sense that METIS yields the highest speed factors. Unfortunately, both those approaches require extensive preprocessing time. In [13] the preprocessing time for smaller road networks than our Germany OSM network ranged from 2 to 16 hours. In [12] preprocessing time for their Germany network (similar to ours), was 9 hours. But even then, SP computation in [12] for similar sized road networks took on average 62ms (5 times slower than our warm queries average time).

Other attempts on using secondary storage for road network data [8], [19] experimented with running routing requests on mobile devices. Goldberg and Werneck [8] implemented the ALT algorithm [6] on a Pocket PC and achieved on the largest road network used (North America, 29.9×10^6 nodes) an average running time of 329s. The preprocessing time required was 208 minutes. Sanders et al [19] implemented a mobile implementation of Contraction Hierarchies [5] on a Nokia N800 device and achieved on the European road network (18×10^6 nodes) an average running time of 458ms for calculating the complete shortest path. Using pre-unpacked paths, the computation time for Europe improved to 97ms. The preprocessing time for mobile contraction hierarchies was 31 minutes for the European road network.

All previous secondary storage attempts arranged the data in blocks and accessed them blockwise similar to our cell partitioning. They also used the same LRU caching policy. Additionally, in order to assign nodes to blocks, they exploited the locality properties of the data, meaning that their nodes were ordered by spatial proximity (nodes with similar IDs should be nearby). Like the present approach, Sanders et al. [19] divided nodes in two groups, one group containing more important nodes and the other containing the rest of the nodes.

The main differences between previous and the present approach are that here the road category information already present in the road dataset is used to distinguish important and unimportant nodes (and therefore no preprocessing was required). Additionally, we did not use any particular node ordering (in order to avoid any preprocessing time) for both METIS and the rectangular grid.

Using the road network graph in its original format has other obvious advantages as well. For one, no special path unpacking routines are required to output the shortest path. For example, Sanders et al. [19] need to explicitly store pre-unpacked paths as sequences of original node IDs. Additionally, when using dynamic weights (time-dependent routing), several versions of the road network (not just edge weights but new shortcuts and new pre-unpacked paths) need to be computed for all previous preprocessing algorithms.

7. CONCLUSION

This work introduced the *Cell Manager* (CM) as an efficient, hierarchical storage manager and companion to hierarchical SP algorithms. In this work, we specifically investigated the HBA* algorithm with the CM. The CM uses a spatio-hierarchical tiling schema to partition the network into a set of tiles according to spatial distribution and road network hierarchies. Two space partitioning algorithms, a regular grid partitioning and the METIS [11] were used.

Our extensive experimentation with two city road networks and the crowd-sourced OSM road network of Germany showed that the CM facilitates fast computation times, efficient memory usage, minimal number of queries and almost optimal results when used in connection with the HBA* shortest path algorithm. The METIS tiling schema has also proved to be more effective than the regular grid, both in terms of resulting computation time and fetched road network size. Overall, HBA* and the aforementioned Cell Manager provide an efficient and scalable solution for serving multiple routing requests in a routing server scenario. Since the road network graph does not require any SP specific preprocessing and is used in its original form, both the CM and HBA* can also be used with dynamic networks and speed profiles.

Our ongoing and future work is as follows. Although the Cell Manager was implemented using a database, it could also be implemented using embedded DBs or files. Therefore an implementation of CM and HBA* on a mobile device may be extremely beneficial. On the other hand, since HBA* due to its hierarchical nature provides slightly sub-optimal results we need to establish how this error can be quantified. The best approach would be to find an error estimate based on road network graph properties such as the *highway dimension* [1]. By predicting the error of HBA*, we may be able to minimize its limited sub-optimality by successfully tweaking its parameters.

8. REFERENCES

[1] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 782–793, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[2] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[3] CloudMade. Cloudmade downloads [online]. `http://downloads.cloudmade.com/`, 2011.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2004.

[7] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A* : Efficient point-to-point shortest path algorithms. In *Workshop on Algorithm Engineering and Experiments*, pages 129–143, 2006.

[8] A. V. Goldberg and R. F. F. Werneck. Computing point-to-point shortest paths from external memory. In *Algorithm Engineering and Experimentation*, pages 26–40, 2005.

[9] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[10] T. Ikeda, M. Y. Hsu, H. Imai, S. Nishimura, H. S. Moura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by ai search techniques. 1994.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[12] J. Maue, P. Sanders, and D. Matijevic. Goal directed shortest path queries using precomputed cluster distances. In *5th Workshop on Experimental Algorithms (WEA), Number 4007 IN LNCS*, pages 316–328. Springer, 2006.

[13] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *J. Exp. Algorithmics*, 11, February 2007.

[14] D. Pfoser, A. Efentakis, A. Voisard, and C. Wenk. A new perspective on efficient and dependable vehicle routing. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 388–391, New York, NY, USA, 2009. ACM.

[15] D. Pfoser, N. Tryfona, and A. Voisard. Dynamic travel time maps - enabling efficient navigation. In *Proceedings of the 18th International Conference on Scientific and Statistical Database Management*, pages 369–378, Washington, DC, USA, 2006. IEEE Computer Society.

[16] PostgreSQL. The world's most advanced open source database [online]. `http://www.postgresql.org/`, 2011.

[17] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms*, pages 568–579, 2005.

[18] P. Sanders and D. Schultes. Engineering fast route planning algorithms. In *Proceedings of the 6th international conference on Experimental algorithms*, WEA'07, pages 23–36, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] P. Sanders, D. Schultes, and C. Vetter. Mobile route planning. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 732–743, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Teleatlas. Tele atlas multinet shapefile 4.3.1 format specifications, 2005.