# 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 little attention.

This work proposes a data management approach for efficient shortest path computation that exploits road network hierarchies. Hierarchies allow us to minimize the portion of the network that is kept in main memory. This approach is insensitive to changes to the network 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, Relational Databases

## 1. INTRODUCTION

Although many previous publications introduced fast algorithms for shortest-path (SP) computation ([7, 14, 28, 33, 13, 9, 10, 19, 21, 30, 3, 8, 26]), most authors assume that the entire road network graph resides in main memory. Additionally, many preprocessing algorithms, such as the ALT [9], Highway Hierarchies [30] or Arc-flags [21] require the storage of additional information related to the algorithm (Landmark information, shortcuts, arc-flags) to complement the original road network graph. Others like Contraction Hierarchies [8] 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 [31].

On the other hand, through crowdsourcing road network graphs are evolving rapidly (150.000 new ways are added to Open Street Map data on average per day [25]) 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 simple mechanism, which fetches nodes as required during SP computation is not an option since a bi-directed Dijkstra search within a city limits may require hundred of thousands node expansions. Performing so many requests within a reasonable time (fractions of a second) is far too challenging for any database or file system. Consequently, road network data should be fetched in "batches", i.e., 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 (such as OpenStreetMap) 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 shortest-path 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 like Dijkstra or $A^*$ , it will be significantly faster and more efficient when combined with a hierarchical algorithm, such as $HBA^*$ [26]. 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 [27].

The remainder of this work is structured as follows. Section 2 briefly surveys SP computation and specifically the HBA$^*$ algorithm. Exploiting hierarchical networks, our data management approach, the Cell Manager is detailed in Section 3. Section 4 describes the environment and data used in the experimentation and Section 5 presents the results when comparing the proposed data management approach used in connection with HBA$^*$ and bi-directed Dijkstra SP computation. Section 6 surveys related work and Section 7 gives conclusions and directions for future work.

## 2. SHORTEST-PATH COMPUTATION AND THE HBA* ALGORITHM

In this section we will describe some basic concepts used at the course of this paper. We will also present the basic properties of HBA$^*$ algorithm, a hierarchical bidirected A* variant initially presented at [26] and propose additional ways to fine-tune its performance.

### 2.1 The Single Pair Shortest-Path Problem

A road network is modelled 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 $w : E \to \mathbf{R}$ is given, mapping edges to weights, with weights typically corresponding to travel times. Given a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ in $G$, the weight of the path is the sum of the weights of its constituent edges $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. The weight $\delta(s, t)$ of a shortest path between vertices $s$ and $t$ is defined as:

$$\delta(s,t) = \left\{ \begin{array}{l} min\{w(p) : \ p \text{ a path from } s \text{ to } t\} \\ \infty \ (if\ no\ path\ from\ s\ to\ t\ exists) \end{array} \right\} \quad (1)$$

A shortest path from $s$ to $t$ is any simple path $p$ with $w(p) = \delta(s, t)$ [6].

Assume a directed graph with non-negative edge weights $w(u, v) \geq 0$ is given. The *single-pair shortest path problem* (SPSP) of finding a shortest path between a source vertex $s$ and a target vertex $t$ can always be solved by applying the greedy Dijkstra's algorithm [7]. Shortest path search can be improved by exploiting knowledge about the structure of the underlying graph. The A$^*$ algorithm [14] (also known as heuristic search) 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 [26] 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 to compute a shortest path from nodes $s$ to $t$ 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 segment/edge. A typical example

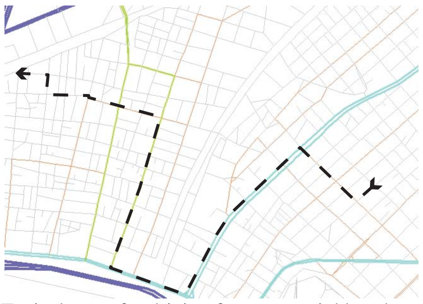


**Figure 1:** Typical route for driving from one neighbourhood to another

of road categories on a road network, may include categories such as "Freeway", "Major Road" or "Local Road". In the road networks used in this paper, low numbers were assigned to higher road categories, i.e., the highest road category was "1: Freeway" and the lowest was "9: Other Road" for the commercial road networks used. The road networks used in all experiments, are described in Section 4.1

This 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 $E = \cup_i E_i$.

To best understand the significance of hierarchical road networks, consider the typical example of how roads of varying importance would typically be used in a *routing task by a human*. Figure 1 gives an example of driving from one neighbourhood in Athens (Kallithea) to another (Moschato). The route length is 3.45km. The route consists of links of varying categories shown in Table 2.

The route starts at level 6 (gray) and continues on levels 5 (purple), 3 (blue), 4 (green), 5 (purple) and arrives at level 6 (gray). Such a route represents a typical behaviour of a driver searching for a route between two locations: First, he searches for a major road connecting the two areas of interest, and then he finds access roads to those major roads [4].

The basic question that needs to be addressed is how we can mimic such route finding behaviour in SP algorithms.

### 2.3 The HBA* algorithm

HBA$^*$ algorithm emulates the typical route finding behaviour by alternating between running an A$^*$ algorithm from $s$ to $t$ (the "forward search"), as well as an A$^*$ algorithm from $t$ to $s$ (the "reverse search") in the reverse graph (the graph with each edge reversed). Each of those searches utilizes *hierarchical jumping*, a technique that favours the use of the higher-category roads to reduce the overall search space and to significantly improve the running time of shortest-path computation.

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

If $\pi_f$(u) is the Euclidean distance of node u from target node $t$ divided by the maximum speed of the road network (for travel time metric) and $\pi_r$(u) is the Euclidean distance of node u from start node $s$ divided by the maximum speed of the road network, $\pi_f$ and $\pi_r$ give lower bounds for forward and reverse search respectively. Similar to [15], HBA* uses $p_f(u) = \frac{\pi_f(u)-\pi_r(u)}{2}$ as the potential function for the forward search and $p_r(u) = \frac{\pi_r(u)-\pi_f(u)}{2} = -p_f(u)$ for the reverse one. Although $p_f$ and $p_r$ usually do not give lower bounds as good as $\pi_f$ and $\pi_r$, they are *feasible* and *consistent* and therefore when used in a standard bi-directed A* search provide optimal results.

Analogous to [9] the HBA* algorithm maintains the length $\mu$ 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 $\mu$ 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 behavior (Figure 1), i.e., when given the choice, it selects higher category roads to reach a destination.

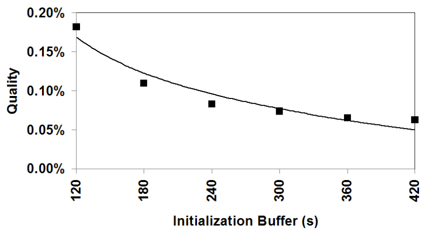
The advantage is that since we effectively reduce the overall size of the road network with hierarchical jumping, fewer nodes need to be subsequently expanded and the performance of the algorithm in terms of memory consumption and computation speed is dramatically improved. On a route similar to Figure 1, during the middle portion of the search, the number of nodes that have to be evaluated is dramatically reduced. This is further evident when examining the number of nodes in a per-category basis of a road network. 70% and 50% of all nodes in the case of Athens and Vienna, respectively belong to the lowest category (local road of minor importance)! Thus, by eliminating this portion of the road network the performance of a SP search will be considerably improved.

### 2.4 Tuning HBA* Behavior

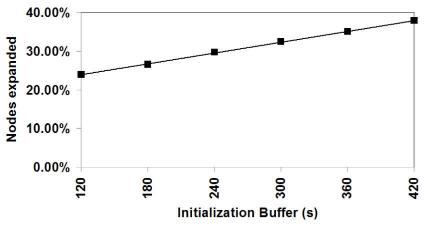
An important issue when using hierarchical jumping in SP computation is that it may eliminate candidate solutions and provide suboptimal results. To address this potential issue, the concept of *initialization buffer* was introduced in [26]. Assuming that we want to compute a SP from node $s$ to $t$, the initialization buffer $I(\epsilon)$ around both $s$ and $t$ prevents the use of hierarchical jumping for all vertices $u \in I(\epsilon) : \{dist(u, t) < \epsilon \vee dist(u, s) < \epsilon\}$.

In [26] a simple Euclidean distance measure is used to quantify the initialization buffer. Although this approach is feasible, it is not optimal, since for every node expanded by the forward search, the algorithm needs to maintain the euclidean distance from $s$ to the expanded node $u$. The same applies to every node $u$ expanded by the reverse search with respect to target node $t$.

However, HBA* "knows" for each node it expands the cost $d(u)$ to reach this node from origin of search. Hence, a better alternative is to use $d(u)$ directly to determine whether $u$ is within the initialization buffer area. What needs to be established is the optimal initialization buffer $\epsilon$ in terms of cost (travel time) rather than distance.



(a) Quality of HBA* results for various initialization buffer sizes (Vienna)



(b) Nodes expanded difference between bidirected Dijkstra and HBA* for various initialization buffer diameters (Vienna)

**Figure 2:** Buffer initialization diameters experiments for Vienna

To identify the optimal $\epsilon$, we compared the quality of bi-directed Dijkstra and HBA* results for 1000 SP computations for the road networks of Athens, Greece and Vienna, Austria, and varying $\epsilon$ ranging from 120 to 420 seconds of travel time. The measured parameters are (a) the quality of results $Q$ and (b) the relative number of expanded nodes. $Q$ is computed as follows.

$$Q = \frac{\delta_{HBA^*}(s,t) - \delta_{Dijkstra}(s,t)}{\delta_{Dijkstra}(s,t)} \tag{2}$$

The results for Vienna (results for Athens were almost identical and therefore are omitted) presented in Figure 2 show that an increase of $\epsilon$ leads to a logarithmic increase in the quality of HBA* results (thus decreasing the gap between the optimal results found by bi-directed Dijkstra and HBA* algorithm). It also results in a linear increase in the number of nodes expanded by HBA* . Hence, $\epsilon = 300s$ is a good compromise for HBA* to (a) find almost optimal results and (b) still expand less than 40% of the nodes bi-directed Dijkstra does. Note that 300s is also a logical time span for *neighbourhood searches*, i.e., routes for which a driver would exploit all available roads rather than use hierarchical jumping. Consequently, 300s is the initialization buffer used in all experiments of Section 5.

## 3. DATA MANAGEMENT AND HIERARCHICAL NETWORKS

The objective of this work is to develop an efficient storage manager for hierarchical road network data for use with SP algorithms. After the discussion of the HBA* algorithm, this section focusses on the design of respective data management techniques that exploit the structure of hierarchical networks.

In a traditional *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 with road network data being fetched on demand. The need for many parallel requests pointed us towards a RDBMS based mechanism. In this work, 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 as the underlying storage mechanism was also chosen to experiment with varying cell schemas and size. In this context, we refer to this storage manager as the *Cell Manager* (CM).

In a nutshell, we model the road network graph by means of a hierarchical partitioning schema as a set of hierarchical cells. In connection with the HBA* algorithm and its hierarchical jumping mechanism, hierarchical cells reduce the portion of the road network that is kept in memory, i.e., why load lower category edges when the algorithm has already "jumped" to higher categories.

## 3.1 Hierarchical Partitioning

Hierarchical jumping as used by the HBA* can be conceptualized as shown in Figure 3. Here, the same road network is shown at different *levels* of abstraction. The network on top includes only higher category roads, while the network at the bottom includes also lower category roads. Following the route example from Section 2.2, after initially moving on lower category roads, when moving to higher categories one essentially chooses to ignore lower category roads until getting close to the destination.

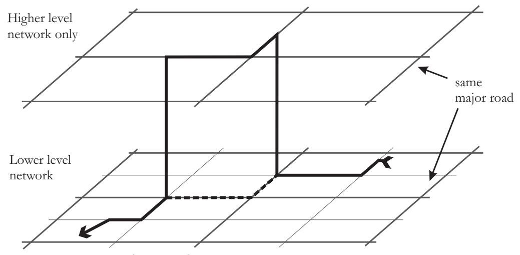


**Figure 3:** Road network hierarchy

The advantage is that since we effectively reduce the overall size of the road network with each hierarchical jump, fewer nodes need to be subsequently explored and the performance of the algorithm in terms of memory consumption and computation speed is dramatically improved.

To be able to exploit this behavior 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, cf. Section 2.2). 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 and its 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 network 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 [32] 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 $\nu$, 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 as follows.

$$c(l) = \frac{n(l)}{\nu} \tag{3}$$

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. Additionally, for UCL the total number of cells depends on the road categories assigned to this level.

Having established some soft limits for $c(l)$, we have to resolve how nodes are assigned to cells. Here we have two choices. Either, we use a *regular rectangular grid* where all cells of the same cell level have the exact same size for a particular road network or we use a tool such as *METIS* [17] for partitioning the road network graph for each cell level. METIS is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices. The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes [18]. METIS was originally intended for parallel processing where partitions should have close to equal size and small boundaries to reduce communication volume between cells (i.e., minimizing cross edges between cells).

To compare the two partitioning schemas, we also added another prerequisite. The extent of a UCL cell is to be uniformly divided into the same number of cells at the lower cell level, i.e., $c(UCL) = k^2 \times c(LCL)$.

In the case of the Athens network, UCL, by including categories 1-5, contains 20,101 nodes. With $\nu = 100$, $c(UCL) = 200 \approx 14 \times 14$. LCL, containing all nodes, contains 140,633 nodes. Here $c(LCL) = 1406 \approx 37 \times 37$. By approximation on both cell levels, UCL uses a rectangular grid of $13x13$ and LCL a grid of $39x39$. In this way each cell on both levels contains about 100 nodes and a UCL cell covers exactly 9 LCL cells.

A UCL cell may cover a larger geographical area, but still occupy the same memory size than a LCL cell since it has less detail. HBA* in 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 A* or Dijkstra algorithms. Figure 4 illustrates the cells being loaded for a search similar to that of Figure 1. 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 (jumping to UCL), only the UCL portion of the network is evaluated and only larger cells are retrieved, while for the beginning and the end of the search small cells of the LCL are fetched.

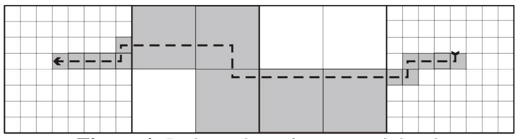


**Figure 4:** Paths and varying network levels

Using METIS we split the Athens road network graph of roads as follows, $c(UCL) = 169 \approx 13 \times 13$ and $c(LCL) = 1521 \approx 39 \times 39$. As a remark, the METIS splitting even for the larger road network of Germany takes less than 30s on an average workstation.

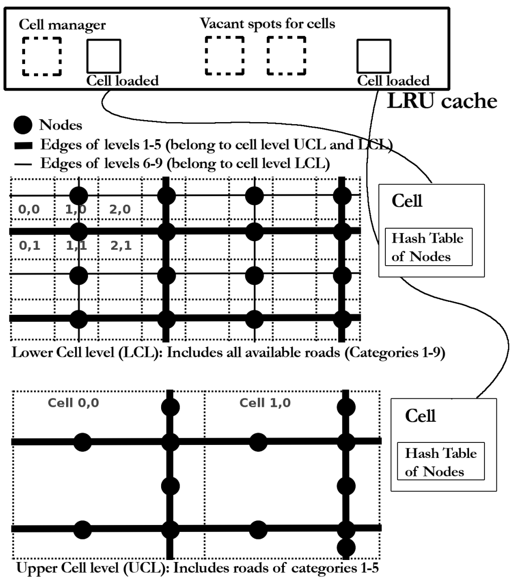

Cell manager
Vacant spots for cells
Cell loaded
Cell loaded
LRU cache
Nodes
Edges of levels 1-5 (belong to cell level UCL and LCL)
Edges of levels 6-9 (belong to cell level LCL)
0,0
1,0
2,0
0,1
1,1
2,1
Cell
Hash Table of Nodes
Lower Cell level (LCL): Includes all available roads (Categories 1-9)
Cell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

**Figure 5:** Hierarchical partitioning of road network

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 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 a 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 simple 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 5. The LRU cache of cells may only store a subset of all available cells of both cell levels combined. Each cell is stored in MM as a hash table of nodes. Additionally, each node contains information about its neighbor nodes and the cell they belong to (for cross edges between cells).

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 neighboring cells of a node. Thus, should a node outside the current cell be expanded and provided it is not present in CM's LRU cache of cells, it is retrieved from secondary storage.

It is easy to use CM with traditional routing algorithms like Dijkstra or A$^*$ (and their bi-directed versions as well). Here however, LCL including all the network edges is used and we are not able to exploit CM's hierarchical features. In that sense, the Cell Manager is *routing algorithm neutral*.

With respect to the database schema, the cells of the two cell

| Bidir | Road Cat. | X | Y | Weight | Neigh NodeID | Cell ID UCL | Cell ID LCL |
|---|---|---|---|---|---|---|---|
| 1 | 5 | 641901 | 5480423 | 42.7 | 1576795 | 12 | 150 |
| **Text representation in LCL:** 112641901548042342.7,1576795,12,150 | | | | | | | |
| **Text representation in UCL:** 1641901548042342.7,1576795,12 | | | | | | | |

**Table 1:** Text Representation of same neighbour node in db for different cell levels

| | Athens (Teleatlas) | Vienna (Teleatlas) | Germany (OSM) |
|---|---|---|---|
| Available 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 2:** Available road categories and sizes of road networks graphs

levels are stored in two separate database tables and are indexed by cell IDs for fast retrieval of the appropriate cell. 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. Table 1 shows the data that is kept, namely, the direction of the edge, edge category, $x$ and $y$ coordinates of the neighboring node (in meters in the appropriate projection), neighboring node ID, and cell ID of the neighboring node in the case of cross edges, i.e., if the neighboring node belongs to a different cell than the one examined. For space reasons this data is stored by means of separate attributes but as a single string for each edge.

Also, for LCL data, should the edge belong to UCL categories, we also store the UCL cell ID. When the HBA$^*$ algorithm jumps to the higher cell level, it needs to know which cell it must retrieve.

The Cell Manager is *database neutral*, since it does not use any vendor specific data types and therefore can be implemented on any standard RDBMS. We have experimented with MySQL [22] and its MyISAM storage engine and PostgreSQL [29]. Although MySQL and PostgreSQL showed similar performance, the lack of foreign key support on MySQL's MyISAM storage engine and the overall enterprise and advanced spatial capabilities of PostgreSQL make PostgreSQL a better choice in the long run and therefore all computation times reported later in this paper are achieved by using PostgreSQL.

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.

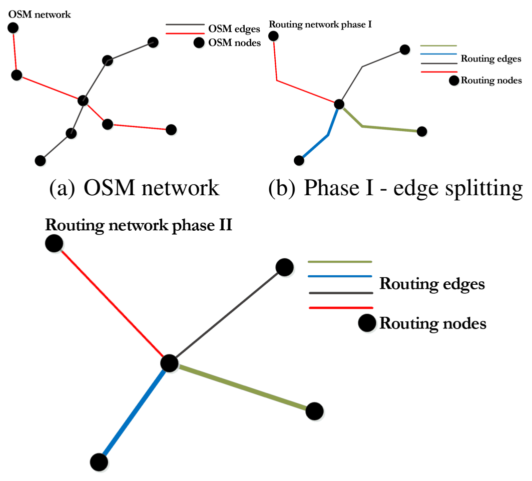
## 4. EXPERIMENTAL SETTING

The experimental evaluation of Section 5 will compare the performance of bi-directed Dijkstra and the HBA$^*$ algorithm using the Cell Manager as the storage manager for road networks. The comparison will be in terms of (i) loaded cells (= total nof requests to the storage manager), (ii) total nof nodes loaded (= nof loaded cells $\times$ nodes per cell) and (iii) computation time. Two separate tiling schemas, rectangular grid and METIS partitioning, will also be compared in terms of efficiency and speed.

This section details the datasets and overall parameters used in the experimentation.

## 4.1 Road Networks

The road networks used in the experimentation were two of city size (Athens and Vienna) and one of country size (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.



(a) OSM network (b) Phase I - edge splitting

(c) Phase II - simplification of edges

**Figure 6:** OSM road network preprocessing

The two city-scale networks [34] cover the greater metropolitan areas of Athens, Greece and Vienna, Austria. The portion of those road networks have a respective extent of roughly 25 × 25 km in each case. The networks comprise nine categories, with 1 corresponding to highways up to 9 for dirt roads.

The country-scale network covering Germany is derived from OpenStreetMap data [24] provided by Cloudmade [5]. Constructing a routing network from OSM data was a multi-part process involving (i) using only the edges (and nodes) that belong to certain road categories [23] and eliminating all other information (e.g., buildings) [1] and (ii) preprocessing OSM data for use for routing. This latter step involves (ii.a) splitting long linestrings, which are used to represents edges in OSM, into separate routing edges and assigning new edge IDs to each new routing edge (cf. Phase I in Figure 6), (ii.b) eliminating OSM nodes that do not constitute routing edges (Phase II), and (ii.c) assigned a weight to each edge (original length / road category speed). The conversion of the OSM Network to a routing network was done using a custom Java tool developed for this task.

Overall, 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 separate road categories. Like the approach used in [9], we used the largest strongly connected component of the available road network graphs. Strong connectivity of the road networks was checked using JGraphT [16]. Table 2 summarizes the road network properties that were used in our SP experiments.

## 4.2 Cell Partitioning

For all road networks two cell levels were used. The Upper Cell Level (UCL) includes 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 and Vienna). The tiling schemas used for each road network aim at having roughly 100 - 200 nodes in a single cell (cf. Table 4).

Since the rectangular grid uses only the bounding box of the graph and all other properties like the graph's structure or density

| | 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 3:** Road categories assigned to cell levels and nodes distribution

| | 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 4:** Partitioning statistics

of nodes are ignored, we have certain cells that are overpopulated (more than 1000 nodes per cell) and others that are empty (Table 5). In contrast, with METIS partitioning the number of nodes per cell is rather constant (Table 6). We will see that the METIS partitioning performed better in every experiment we conducted for all three road network graphs.

The proposed cell hierarchy is directly linked to the operation of the HBA$^*$ algorithm and respective road network properties. As shown in Table 3, UCL nodes comprise only 14,3%, 30,7% and 19.6% of the total number of nodes for Athens, Vienna and Germany respectively. Since HBA$^*$ aims for using higher category roads, it will mostly use the upper cell level (once outside the initialization buffers area) and is expected to read fewer cells from secondary storage. This behavior will give HBA$^*$ the scalability and speed in memory constrained environments for handling large road networks. Another important aspect is that for Vienna, where UCL nodes comprise 30% of the network, HBA$^*$ in almost all cases finds optimal results by only utilizing this 30% of the road network (Table 7).

## 4.3 Shortest-path Queries

Two different SP query types were used in the experimentation (cf [32]). In the *cold query* case for the 1000 random queries that are executed, the LRU cache (implemented by the Cell Manager) 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 ([12],[32]), the cold query experiments are the most important ones, since the device is not expected to 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. Still, cold query results are conducted for reasons of a fair comparison to related approaches.

## 4.4 Performance Environment and Implementation

Experiments have been conducted on a Intel Core 2 Duo CPU clocked at 3.00 GHz with 8Gb main memory and 6144 KB L2 Cache, running Ubuntu 10.10 64bit (kernel 2.6.35 − 28). The HBA$^*$ algorithm and the Cell Manager have been implemented in Java.

| | MAX Nodes per cell UCL | Empty cells UCL | MAX Nodes per cell LCL | Empty cells LCL |
|---|---|---|---|---|
| **Athens** | 2862 | 89 (53%) | 3012 | 1056 (69%) |
| **Vienna** | 1085 | 37 (22%) | 1443 | 192 (28%) |
| **Germany** | 4694 | 885 (22%) | 3608 | 3768 (23%) |

**Table 5:** Rectangular grid statistics

| | MAX Nodes per cell UCL | MIN Nodes per cell UCL | MAX Nodes per cell LCL | MIN Nodes per cell LCL |
|---|---|---|---|---|
| **Athens** | 126 | 111 | 102 | 84 |
| **Vienna** | 108 | 95 | 94 | 72 |
| **Germany** | 174 | 158 | 231 | 173 |

**Table 6:** METIS 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 7:** Algorithm comparison for the three road networks graphs

64-bit PostgreSQL 9.0.3 was used for the implementation of the Cell Manager.

Some implementation details and data structures used in the Cell Manager are as follows. Since the CM may need to unload a cell, scanned nodes for forward and reverse search are kept in two separate hash tables. The forward and reverse priority queues were implemented using the standard heap implementation provided by Java, augmented by two hash tables. Considering that we used standard Java data structures, using more effective data structures might improve the overall performance. Still, querying the database is the bottleneck in the performance of the Cell Manager and for cold query experiments (initially empty cache) the times recorded are mainly due to database speeds.

## 5. EXPERIMENTAL RESULTS

What follows below are experiments contrasting the performance of the HBA$^*$ algorithm with that of bi-directed Dijkstra SP computation when using the Cell Manager.

In contrast to related work (cf. [32] and [8]), where outputting the complete shortest path is considered a separate task independent from the actual SP calculation, this task is included in the costs reported in our experiments. While in other approaches this involves path-unpacking data structures, the CM directly supports this task as part of the SP computation.

### 5.1 SP Quality

The first set of experiments contrasts the performance of HBA$^*$ SP computation to that of bi-directed Dijkstra in terms of quality of results and number of nodes totally expanded. These sizes are independent of the CM's tiling schema and depend only on the actual algorithm. 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 hierarchical jumping does not guarantee optimal results, in reality the results it produces are close to identical to the optimal bi-directed Dijkstra. Table 7 shows that for Germany it produces on average 0.3% worse results. Even better numbers apply to Athens and Vienna with 0.26% and 0.07% worse results, respectively.

On the other hand, Table 7 shows that in larger networks, HBA$^*$ expands only 3.7% (11,888 nodes on average per search) of the nodes that bi-directed Dijkstra expands and utilizes less than 20% of the road network (cf. Table 3).

### 5.2 Cold SP queries

The biggest challenge for a secondary storage based data management is how it performs for cold queries, i.e., experiments where the cache is initially empty. To assess the performance, we compare the two tiling schemas, the rectangular grid and the METIS partitioning, in terms of cells and nodes loaded. The former represents

| | 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 8:** Results for rectangular grid and METIS (Cold SP queries)

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 network are presented in Table 8.

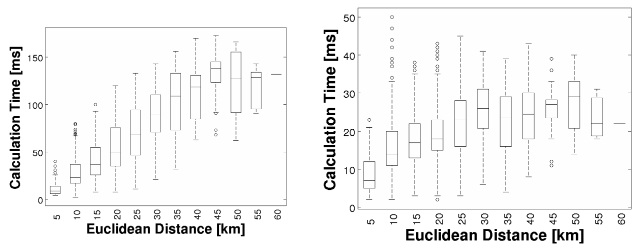
The results show that the combination of HBA$^*$ and CM provides average computation times of less than 100ms even for large networks. Again, this is achieved with no preprocessing and the entire road network graph stored on disk. On average for a SP query for Germany 27,000 nodes are loaded, which is roughly 0.8% of the total road network graph.

*Bi-directed Dijkstra* on the other hand had a average computation time of 5.5s for Germany (which is *by a factor of 60 slower than* HBA*) and had to load on average of 515,000 nodes (corresponding to 15% of the total road network graph). Note that, since no unloads occurred in the Dijkstra experiments so to allow for a fair comparison to HBA$^*$ , these computation times will be significantly worse in an actual server-based routing scenario.

Another obvious result is that *METIS partitioning is more effective that an regular partitioning grid* and results in better computation times, e.g., 5-12 ms for all networks. The METIS effectiveness is also evident in that for a small road network such as Athens, Dijkstra plus METIS loads less nodes than HBA$^*$ and regular grid does (cf. 113.9% in Table 8). The METIS effectiveness is credited to the fact that all cells have about the same size and therefore the minimal number of nodes is fetched. A regular grid has certain cells that are overpopulated and therefore many unnecessary nodes (which are not utilized during the search) are fetched. Results showed for Germany that METIS loads on average 27,127 nodes per search, whereas using the regular grid partitioning an average of 35,380 nodes per search (8000 more nodes per search) is fetched. Additionally, this margin grows even bigger in the worst case, where METIS loads almost 14,000 fewer nodes than the regular grid for Germany.

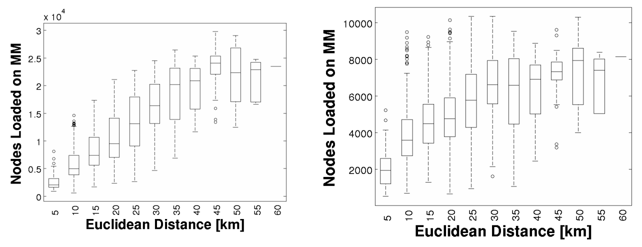
Fluctuations in computation times and fetched number of nodes loaded in relation to Euclidean distance between the origin and the destination of search (indicator for SP result size) are presented in the box and whisker plots of Figures 7 - 10. The figures show experiments for bi-directed Dijkstra and HBA$^*$ with METIS partitioning. Each box spreads from the lower to upper quartile and contains the *median*, the *whiskers extend to the minimum and maximum value* and outliers plotted separately. The results for Athens are very similar to that of Vienna and are therefore omitted.

The figures clearly show that the number of nodes loaded in MM follow similar pattern to computation times. This means that the



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

**Figure 7:** Cold SP queries - computation times [Vienna]



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

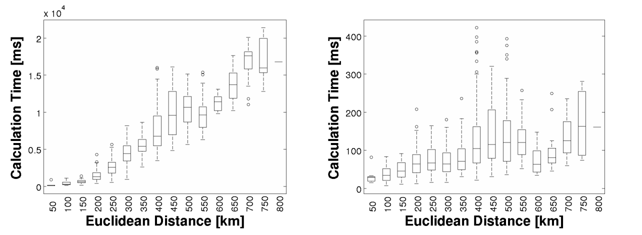
**Figure 8:** Cold SP queries - nodes loaded [Vienna]

nodes retrieved from secondary storage are of course the main contributing factor to the performance of SP algorithms in combination with a storage manager. We also see that the performance indicators for HBA* and CM do not degrade with Euclidean distance beyond a certain point. Hence, the larger the graph, the greater the advantage of this approach. This is of course due to the use of the UCL in path computation. The closer the origin and the destination are, the smaller is the advantage of a hierarchical SP algorithm. HBA* in the most disadvantageous query case of Germany loads 15 times fewer nodes and provides 50 times better computation times than bi-directed Dijkstra.

## 5.3 Warm SP queries

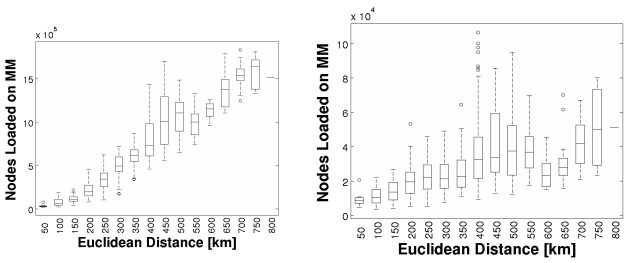
The previous section established that the combination of CM and HBA* is efficient even for large road network graphs and all graph data stored on secondary storage. Still, since we emulate the routing server scenario, those results will not be the typical case. A routing server will perform countless SP computations and therefore most popular cells will be already loaded in MM. For this 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. The METIS partitioning was used in all experiments.

Table 9 gives average running times. This table should be compared to Table 8. For example, the avg. running time for Germany has been reduced from 91ms to, now, 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. We also see that the margin between bidirected Dijkstra and HBA* computation times for warm queries has very slightly increased for all three road networks (i.e., HBA* performs even better on warm queries).



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

**Figure 9:** Cold SP queries - computation times [Germany]



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

**Figure 10:** Cold SP queries - nodes loaded [Germany]

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

**Table 9:** Warm SP queries - Computation times

The computation times as function of the Euclidean distance between origin and destination for Vienna and Germany are shown in Figure 11.

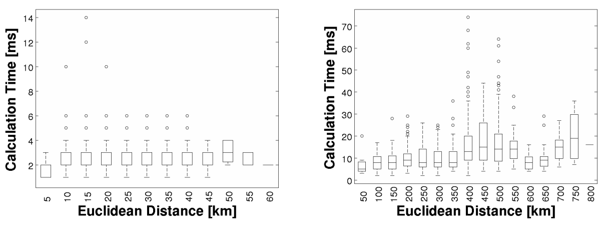
We see that the combination of CM and HBA* for warm queries provides computation times similar to typical main memory SP algorithms. This is expected since HBA* utilizes only the upper cell level, which contains only a fraction of the total nodes of the road network graph and therefore is expected to fit in main memory. Using a different RDBMS and by adding a second layer of caching on the database layer (e.g., by using MemCached [20]) could possibly result in a even faster SP computation.

## 5.4 Summary

HBA* in combination with the Cell Manager provide fast computation times that, in the case of warm queries, are comparable to SP algorithms utilizing main-memory data structures for road network graphs. This 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 (cf. user-generated road networks). This creates a significant advantage for this approach over novel SP algorithms, like Contraction Hierarchies [8] and Transit Node Routing [3].

## 6. RELATED WORK

The idea of partitioning a graph for shortest path computation is not new. Maue et al. [19] partitioned the graph into clusters and



(a) Vienna (b) Germany

**Figure 11:** Warm SP queries - computation times

precomputed distances between border nodes of clusters in order to prune their modified Dijkstra's algorithm with lower and upper bounds. Möhring et al. [21] also divided the graph into partitions and gathered information for each edge, on whether this edge is on a shortest path into a given region. For each edge this information is stored in a vector (arc-flag vector). Arc-flags are 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 [21] the preprocessing time for a 3 times smaller road network ($10^6$ nodes) than the Germany OSM network we used ranged from 2 to 16 hours. In [19] preprocessing time for their Germany network, which was similar in size to ours, was 9 hours. But even with this preprocessing, SP computation in [19] for similar sized road networks took on average 62ms, which is 5 times slower than our warm queries average time (Section 5.3).

Other attempts on using secondary storage for road network data [12], [32] experimented with running routing requests on mobile devices. Goldberg and Werneck [12] implemented the ALT algorithm [9] on a Pocket PC and achieved on the largest road network used (North America, $29.9 \times 10^6$ nodes) an average running time of 329s. The preprocessing time required was 208 minutes for the aforementioned network. Sanders et al [32] implemented a mobile implementation of Contraction Hierarchies [8] on a Nokia N800 device and achieved on the European road network ($18 \times 10^6$ 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. According to [32], the Rearch algorithm [11],[13] was also implemented on a mobile device yielding query times of a few seconds.

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). The real road networks that were used have already similar node ordering implemented (the road networks we used, did not). Still, in [32], in order to improve spatial proximity of nodes, a modified depth-first search on the reverse graph to compute a new improved node topological order was implemented. Such an approach also requires to store for each node its original ID, so to be able to perform the reverse mapping. Like the present approach, Sanders et al. [32] 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 obvious advantages as well. For one, no special path unpacking routines are required to not only compute but also output the shortest path. For example, Sanders et al. [32] need to explicitly store pre-unpacked paths as sequences of original node IDs. In not doing so, the average SP computation time is 4 times longer (from 97 ms to 458 ms). 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 [17] were used.

Our extensive experimentation with two commercial road networks (Athens, Vienna) and the crowdsourced OpenStreetMap (OSM) road network of Germany showed that the CM facilitates fast computation times, efficient memory usage, a 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.

In addition, we improved the performance and speed of the initialization buffer concept introduced in [26] as a means to tune the efficiency of the HBA$^*$ algorithm. 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, i.e., a dynamic weight database commonly referred to as speed profiles and a evolving road network graph such as obtained by crowdsourcing efforts.

Our ongoing and future work is as follows. Although the Cell Manager was implemented using a database, it could also be implemented using embedded or object DBs, or native filesystems. 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* [2]. By predicting the error of HBA*, we may be able to minimize its limited sub-optimality by successfully tweaking its parameters.

## 8. REFERENCES

[1] Osmosis [online]. `http://wiki.openstreetmap.org/wiki/Osmosis`, 2011.

[2] 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.

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

[4] P. H. Bovy and E. Stern. Route choice: Wayfinding in transport networks. *Transportation Research Part A: Policy and Practice*, 27(4):338–339, 1993.

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

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

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

[8] 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.

[9] 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.

[10] 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.

[11] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Better landmarks within reach. In *the 9TH DIMACS Implementation Challenge: Shortest Paths*, 2007.

[12] 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.

[13] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Algorithm Engineering and Experimentation*, pages 100–111, 2004.

[14] 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.

[15] 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.

[16] JGraphT. A free java graph library [online]. `http://jgrapht.sourceforge.net/`, 2011.

[17] 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.

[18] K. Lab. Metis - family of multilevel partitioning algorithms [online]. `http://glaros.dtc.umn.edu/gkhome/views/metis`, 2011.

[19] 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.

[20] Memcached. A distributed memory object caching system [online]. `http://memcached.org/`, 2011.

[21] 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.

[22] MySQL. Reference manual. storage engines [online]. `http://dev.mysql.com/doc/refman/5.0/en/storage-engines.html`, 2011.

[23] OpenStreetMap. Map features [online]. `http://wiki.openstreetmap.org/wiki/Map_Features`, 2011.

[24] OpenStreetMap. Openstreetmap [online]. `http://www.openstreetmap.org/`, 2011.

[25] OpenStreetMap. Stats - openstreetmap wiki [online]. `http://wiki.openstreetmap.org/wiki/Stats#OpenStreetMap_Statistics_Available`, 2011.

[26] 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.

[27] 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.

[28] I. Pohl. Bi-directional search. *Machine Intelligence*, 6, 1971.

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

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

[31] 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.

[32] 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.

[33] L. Sint and D. de Champeaux. An improved bidirectional heuristic search algorithm. *J. ACM*, 24:177–191, April 1977.

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