Project Number 288094

eCOMPASS

eCO-friendly urban Multi-modal route PlANning Services for mobile uSers

STREP
Funded by EC, INFSO-G4(ICT for Transport) under FP7

eCOMPASS – TR – 003

A New Dynamic Graph Data Structure for Large-Scale Transportation Networks

Georgia Mali, Panagiotis Michail, Christos Zaroliagis

July 2012

A New Dynamic Graph Data Structure for Large-Scale Transportation Networks *

Georgia Mali[1,2], Panagiotis Michail[1,2], and Christos Zaroliagis[1,2]

[1] Computer Technology Institute & Press "Diophantus", N. Kazantzaki Str., Patras University Campus, 26504 Patras, Greece
[2] Dept of Computer Engineering & Informatics, University of Patras, 26500 Patras, Greece
Email: {mali,michai,zaro}@ceid.upatras.gr

5 July 2012

Abstract. We present a new graph data structure specifically suited for large-scale dynamic transportation networks. Our graph structure provides simultaneously three unique features: compactness, agility and dynamicity. All previously known graph structures were lacking support in at least one of the aforementioned features and/or could not be efficiently applied in large-scale dynamic transportation networks. We demonstrate the practicality of our new graph structure by conducting an experimental study for shortest route planning in large-scale European road networks with a few dozen millions of nodes and edges. Using classical shortest path routing algorithms, we can easily achieve point-to-point query times in the order of milliseconds, while our new graph structure can be updated in just a few microseconds after a node or edge insertion or deletion.

1 Introduction

In recent years we observe a tremendous amount of research for efficient route planning in road and other public transportation networks. For instance, we are witnessing extremely fast algorithms that answer point-to-point shortest path queries in a few milliseconds (in certain cases even less) in large-scale road networks with a few dozen millions of nodes and edges after a certain preprocessing phase; see e.g., [3, 7, 11, 13, 14]. These algorithms are clever extensions and/or variations of the classical Dijkstra's algorithm [8] – which turns out to be rather slow when applied to large-scale networks – and hence are usually referred to as *speed-up techniques* (over Dijkstra's algorithm).

Speed-up techniques employ not only heuristics to improve the search space of Dijkstra's algorithm, but also optimizations in the way they are implemented. To the best of our knowledge, the graph structures used are variations of the adjacency list graph representation and provide control on the storage of the graph elements. These graph structures are not only *compact*, in the sense that they store nodes and edges in adjacent memory addresses, but also support arbitrary *reordering* of nodes and edges to increase the locality of the main for-loop in Dijkstra's algorithm. The latter, known as *internal node reordering*, has played a crucial role in achieving the extremely fast running times for point-to-point shortest path queries in large-scale networks [3, 7, 11, 13, 14]. This optimization effectively improves the locality of the nodes by rearranging them within memory, hence improving cache efficiency and running times of the algorithms.

These graph structures are very efficient when the graph remains static but suffer badly when updates occur. The reason is that, in order to keep their compactness and locality, an update must shift a great amount of elements in memory. Therefore, such representations are used mostly in static scenarios, where the underlying network does not change, an assumption that may not always be realistic in practice. For instance, road networks face sudden unforeseen changes like an accident or a natural disaster, frequent small changes like traffic jams, even rare changes like the construction of a new road. Also, edge weights reflecting travel times may change within a day depending on the specific hour of the day considered. In such cases, the graph representing the network must be updated and any preprocessed data must be recomputed. Hence, it is essential that any graph structure used in such applications can support efficient insertions and deletions of nodes and edges.

* This work was supported by the EU FP7/2007-2013 (DG INFSO.G4-ICT for Transport), under grant agreement no. 288094 (project eCOMPASS). This work was done while the third author was visiting the Karlsruhe Institute of Technology.

In summary, what is needed for efficient routing in dynamic large-scale transportation networks is a graph data structure that supports the following features.

1. *Compactness*: ability to efficiently access consecutive nodes and edges, a requirement of all speed-up techniques based on Dijkstra's algorithm.
2. *Agility*: ability to change and reconfigure its internal layout in order to improve the locality of the elements, according to a given algorithm.
3. *Dynamicity*: ability to efficiently insert or delete nodes and edges.

According to the above features, a choice of a graph structure suitable for the aforementioned applications must be made. An obvious choice is an adjacency list representation, implemented with linked lists of adjacent nodes, because of its simplicity and dynamic nature. Even though it is inherently dynamic, in a way that it supports insertions and deletions of nodes and edges in $O(1)$ time, it provides no guarantee on the actual layout of the graph in memory (handled by the system's memory manager). Therefore, it does have dynamicity but it has neither compactness nor agility.

A very interesting variant of the adjacency list, extensively used in several speed-up techniques (see e.g., [3]), is the *forward star* graph representation [1, 2], which stores the adjacency list in an array, acting as a dedicated memory space for the graph. The nodes and edges can be laid out in memory in a way that is optimal for the respective algorithms, occupying consecutive memory addresses which can then be scanned with maximum efficiency. This is very fast when considering a static graph, but when an update is needed, the time for inserting or deleting elements is prohibitive because large shifts of elements must take place. Thus, a forward star representation offers compactness and agility, and therefore ultra fast query times, but does not offer dynamicity.

Motivated by the efficiency of the forward star representation in the static scenario, we present a new data structure for directed graphs which supports all the aforementioned features. In particular:

- Scanning of consecutive nodes or edges is optimal (up to a constant factor) in terms of time and memory transfers, and therefore comparable to the maximum efficiency of the forward star representation (compactness).
- Nodes and edges can be reordered within allocated memory in order to increase any algorithm's locality of reference, and therefore efficiency. Any speed-up technique can give its desired node ordering as input to our graph structure (agility).
- Inserting or deleting edges and nodes compares favourably with the performance of the adjacency list representation implemented as a linked list, and therefore it is fast enough for all practical applications (dynamicity).

To assess the practicality of our new graph structure, we conducted a series of experiments on shortest path routing algorithms on large-scale European road networks. Our goal was to merely show the performance gain of using our graph structure compared to the adjacency list or the forward star representation on classical shortest path routing algorithms, rather than to beat the running times of the best speed-up techniques. Even with the classical shortest path routing algorithms, our graph structure can answer shortest path queries in milliseconds and handle updates of the graph layout, like insertions or deletions of both nodes and edges, in just a few microseconds. Most importantly, existing speed-up techniques can switch their underlying graph structure to our new graph structure, thus keeping their stelar query performance while getting efficient update times almost for free.

Our graph data structure stands as a good compromise between two extremes, the adjacency list representation which offers optimal dynamicity and the forward star representation which offers optimal compactness and agility. It is just 2% slower than the forward star representation in query time but over a *million* times faster in update time. On the other hand, it is slower than the adjacency list in update time, but close to 30% faster in query times.

Note that our graph structure is not a speed-up technique on its own but can increase the efficiency of any speed-up technique implemented on top of it. Existing techniques operating on static graph layouts can benefit from its dynamic nature to extend their focus to dynamic scenarios as well, without sacrificing performance. To the best of our knowledge, our approach to deal with dynamic large-scale transportation networks is the first one that concerns the dynamization of the

graph structure per se. In contrast, the so far known approaches to deal with such dynamic scenarios [17, 18] operate on a static graph layout and are only concerned with the development of dynamic algorithms that update the preprocessed data. All those techniques can also be implemented on top of our graph structure and hence benefit from its performance.

Our analysis of the new graph structure is based on the *cache-oblivious model* [10, 15] which accounts memory transfers in memory blocks of unknown size B, as these transfers are the dominating operation w.r.t. time. We consider the two-level memory hierarchy model where the memory hierarchy consists of a first-level fast memory (cache) and an arbitrarily large second-level slow memory (main memory) partitioned into blocks of (unknown) size B. The data from the main memory to the cache and vice versa are transferred in blocks (one block at a time).

This paper is organized as follows. In Section 2, we review some preliminary concepts that will be used throughout the paper. In Section 3, we present the new graph structure, along with its theoretical analysis and comparison to other graph structures. In Section 4, we present our experimental study on real-world data. We conclude in Section 5.

2 Preliminaries

Let $G = (V, E)$ be a directed graph with node set V, edge set E, $n = |V|$, and $m = |E|$. All graphs throughout this paper are considered directed, unless mentioned otherwise. Also, there is a weight function $wt : E \to \mathbb{R}_0^+$ associated with E.

2.1 Graph Representations

There are multiple data structures for graph representations. The specific type of data structure used, depends heavily on the characteristics of the input graph and the performance requirements of each specific application.

The most commonly used data structure for representing a graph is the *adjacency list representation*. The *adjacency list* $A(u)$ of a vertex u in a graph $G = (V, E)$ is defined as the set $A(u) = \{w : (u, w) \in E\}$. The *adjacency list representation* of a graph G stores the adjacency list of every vertex in G as a linked list. There is an additional linked list of length n for storing the heads of the lists. The adjacency list representation takes $O(n + m)$ space. Any additional information attached to the nodes is stored at the respective heads of the lists. Accordingly, any additional information attached to an edge (u, w), such as $wt(u, w)$, is stored along with w at the adjacency list of u. Figure 2 shows[1] the adjacency list representation of the graph in Figure 1.

A variant of the adjacency list representation is the *forward star representation* [1, 2]. In this representation, the node list is implemented as an array, and all adjacency lists are appended to a single edge array sorted by their source node. Unique non-overlapping *adjacency segments* of the edge array contain the adjacency list of each node. Each node points to the segment in the edge array containing its adjacent edges. The additional information attached to nodes or edges is stored in the same way as in the adjacency list. The forward star representation of the graph in Figure 1 is shown in Figure 3.

2.2 Packed-memory Array

A packed-memory array [4] maintains N ordered elements in an array of size $P = cN$, where $c > 1$ is a constant. The cells of the array either contain an element x or are considered empty. Hence, the array contains N ordered elements and $(c-1)N$ empty cells called *holes*. The goal of a packed-memory array is to provide a mechanism to keep the holes in the array uniformly distributed, in order to support efficiently insertions, deletions and scans of (consecutive) elements. This is accomplished by keeping intervals within the array such that a constant fraction of each interval contains holes. When an interval of the array becomes too full or too empty, its elements are spread out evenly within a larger interval by keeping their relative order. This process is called a *rebalance* of the (larger) interval. Note that during a rebalance an element may be moved to a different cell within an interval. We shall refer to this as the *move* of an element to another cell.

[1] For simplicity, we do not show any additional information associated with nodes or edges.

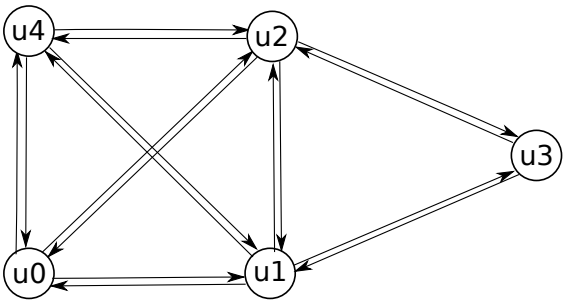


Fig. 1: A directed graph with 5 nodes and 16 edges.

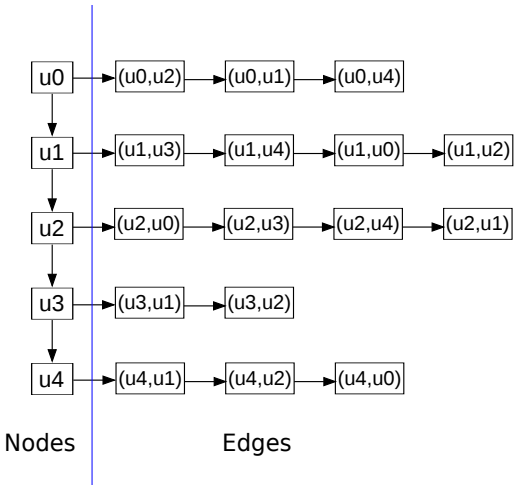


Fig. 2: Adjacency list representation.

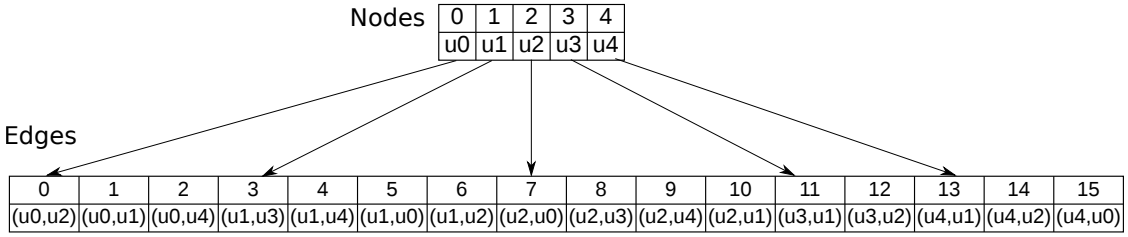


Fig. 3: Forward star representation.

The array is initially divided into $\Theta(P/\log P)$ segments of size $\Theta(\log P)$ such that the number of segments is a power of 2. A perfect binary tree is built iteratively on top of these array segments, assigning each leaf of the tree to one segment of the array. Each tree vertex y is associated with an array interval corresponding to a collection of contiguous array segments assigned to y's descendant leaves. The root vertex is associated with the entire array. The depth of the root vertex is defined as 0 and the depth of the leaves is equal to $d = \log \Theta(P/\log P) = \Theta(\log P)$.

The total number of cells contained in the collection of array segments associated with an internal tree vertex u is called the *capacity* of u, while the actual number of (non-empty) elements in the respective cells is called the *cardinality* of u. The ratio between u's cardinality and its capacity is called the *density* of u. Clearly, $0 \leq density(u) \leq 1$.

Strict density thresholds are imposed on the vertices of the tree. For arbitrary constants ρ_d, ρ_0, τ_0, τ_d, such that $0 < \rho_d < \rho_0 < \tau_0 < \tau_d = 1$, a vertex's *upper density threshold* is defined as $\tau_k = \tau_0 + \frac{\tau_d - \tau_0}{d}k$, where k is the vertex depth, and its *lower density threshold* as $\rho_k = \rho_0 + \frac{\rho_0 - \rho_d}{d}k$. Consequently, $0 < \rho_d < \rho_{d-1} < ... < \rho_0 < \tau_0 < \tau_1 < ... < \tau_d = 1$. A tree vertex u is *within thresholds* if $\rho_k \leq density(u) \leq \tau_k$. An example of a packed-memory array is shown in Figure 4, where the upper field of each internal vertex shows the cardinality (left number) and the capacity (right number) of the array segment it is associated with, while the lower field shows the lower and upper density thresholds.

When inserting or deleting an element that belongs to a specific segment in the array, the leaf vertex u associated with the segment is checked for whether it remains within thresholds after the operation or not. If it does remain within thresholds, the segment is rebalanced, and the vertex's cardinality and density are updated. If the operation causes the vertex to exceed any of its thresholds, an ancestor v of u that is within thresholds is sought. If such a vertex exists, then its associated interval (containing all elements in the collection of segments comprising the interval) is rebalanced. Note that whenever an interval associated with a tree vertex v is rebalanced, its descendant vertices

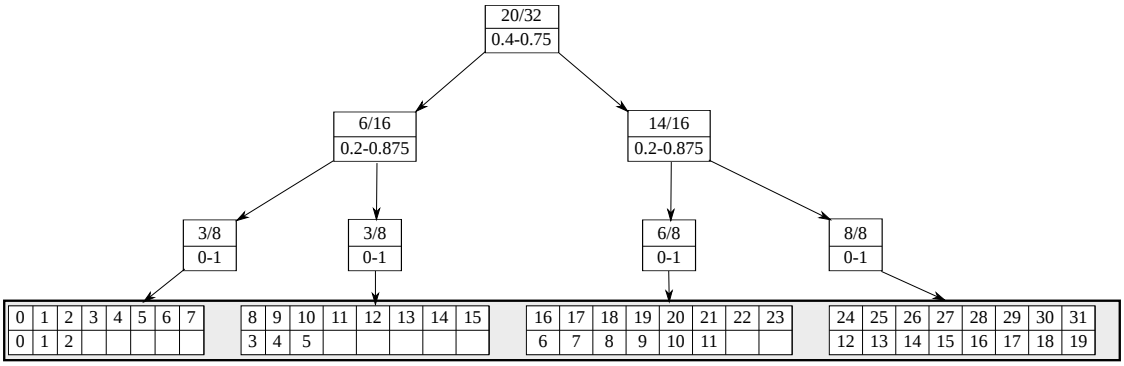


Fig. 4: Packed-Memory Array on the ordered set $[0, 19]$.

are not only within their own density thresholds but also within the density thresholds of v. Finally, if an ancestor within thresholds does not exist, the array is re-allocated to a new space, double or half in size accordingly, and the thresholds are recomputed. The following results are shown in [4].

Theorem 1. *[4] The packed-memory array structure maintains N elements in an array of size cN, for any desired constant $c > 1$, supporting insertions and deletions in $O(\log^2 N)$ amortized time and $O(1 + \frac{\log^2 N}{B})$ amortized memory transfers, as well as scanning of S consecutive elements in $O(S)$ time and $O(1 + S/B)$ memory transfers.*

3 The Packed-memory Graph

3.1 Structure

Our graph structure consists of three packed-memory arrays, one for the nodes and two for the edges of the graph (viewed as either outgoing or incoming) with pointers associating them. The two edge arrays are copies of each other, with the edges sorted as outgoing or incoming in each case. Therefore, the description and analysis in the following will consider only the outgoing edge array. The structure and analysis is identical for the incoming edge array. A graphical representation of our new graph structure, for the example graph of Figure 1, is shown in Figure 5.

Let $P_n = 2^k$, where k is such that $2^{k-1} < n \leq 2^k$. The nodes are stored in a packed-memory array of size P_n with *node density* $d_n = \frac{n}{P_n}$. Therefore, the packed-memory node array has size $P_n = c_n n$ where $c_n = 1/d_n$. Each node is stored in a separate cell of the packed-memory node array along with any information associated with it. The nodes are stored with a specific arbitrary order $u_0, u_1, ..., u_{n-2}, u_{n-1}$ which is called *internal node ordering* of the graph. As we shall see in Section 3.3, this ordering has a great impact on the performance of the algorithms implemented on top of our new graph structure.

Let $P_m = 2^l$, where l is such that $2^{l-1} < m \leq 2^l$. The edges are also stored in a packed-memory array of size P_m with *edge density* $d_m = \frac{m}{P_m}$. Therefore, the packed-memory edge array has size $P_m = c_m m$ where $c_m = 1/d_m$. Each edge is stored in a separate cell of the packed-memory edge array along with any information associated with it, such as the edge weight. The edges are laid out in a specific order, which is defined by their source node. More specifically, we define a partition $C = \{E_{u_0}, E_{u_1}, ..., E_{u_{n-2}}, E_{u_{n-1}}\}$ of the edges of the graph according to their source nodes, where $E_{u_i} = \{e \in E | source(e) = u_i\}$, $E_{u_i} \cap E_{u_j} = \emptyset$, $\forall i, j$, $i \neq j$, and $E_{u_0} \cup E_{u_1} \cup ... \cup E_{u_{n-2}} \cup E_{u_{n-1}} = E$. That is, each edge e belongs to only one set of the partition and there is a one-to-one mapping of nodes to their corresponding outgoing edge sets.

The sets E_{u_i}, $0 \leq i < n$, are then stored consecutively in the packed-memory edge array in the same order as the one dictated by the internal node ordering in the packed-memory node array. Thus, all outgoing edges E_{u_i} of a node u_i are stored in a unique range of cells of the packed-memory edge array without any other outgoing edges stored between them. This range is denoted by R_{u_i} and its length is $O(|E_{u_i}|)$ due to the properties of the packed-memory edge array.

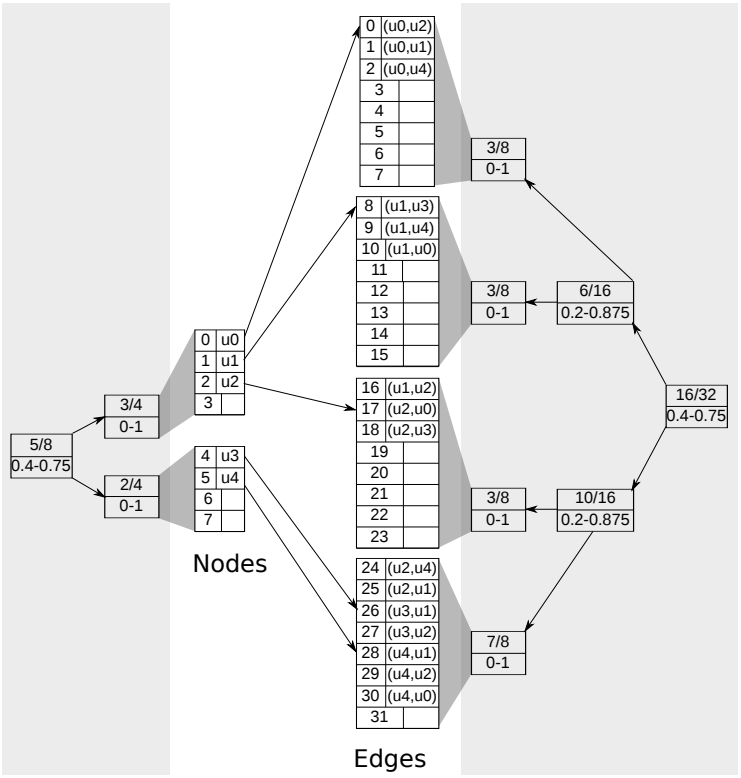


Fig. 5: Packed-memory Graph representation

Every node u_i stores a pointer to the start and to the end of R_{u_i} in the edge array. The end of R_{u_i} is at the same location as the start of $R_{u_{i+1}}$, since the outgoing edge sets have the same ordering as the nodes. If a node u_i has no outgoing edges, both of its pointers point to the start of $R_{u_{i+1}}$. Hence, given a node u_i, *determining* R_{u_i} takes $O(1)$ time.

3.2 Operations

Our new graph structure supports the following operations.

Scanning Edges. In order to scan the outgoing edges of a node u, the range, R_u, including them is determined by the pointers stored in the node. Then this range is sequentially scanned returning every outgoing edge of u.

Inserting Nodes. In order to insert a node u_i between two existing nodes u_j, u_{j+1}, we identify the actual cell that should contain u_i and execute an insert operation in the packed-memory node array. Clearly, this insertion changes the internal node ordering. The insert operation may result in a rebalance of some interval of the packed-memory node array, and some nodes being moved into different cells. For each node that is moved, its edges are updated with the new position of the node. The outgoing edge pointers of the newly created node u_i (which has not yet any adjacent edges) point to the start of the range $R_{u_{j+1}}$.

Deleting Nodes. In order to delete a node u_i between two existing nodes u_{i-1}, u_{i+1}, we first have to delete all of its outgoing edges, a process that is described in the next paragraph. Then, we identify the actual node array cell that should be cleared and execute a delete operation in the packed-memory node array. The delete operation may also result in a rebalance as before, so, for each node that is moved, its edges are updated with the new position of the node.

Inserting or Deleting Edges. When a new edge (u_i, u_j) is inserted (deleted), we proceed as follows. First, node u_i and its outgoing edge range R_{u_i} are identified. Then, the cell to insert to (delete from) within this range is selected and an insert (delete) operation in this cell of the packed-memory edge array is executed. This insert (delete) operation may cause a rebalance in an interval of the packed-memory edge array, causing some edges to be moved to different cells. As a result, the ranges of other nodes are changed too.

When a range R_{u_k} changes, the non-zero ranges R_{u_x} and R_{u_y}, $x < k < y$, adjacent to it change too. Note that x may not be equal to $k-1$ and y may not be equal to $k+1$, since there may be ranges with zero length adjacent to R_{u_k}. In order for R_{u_x}, R_{u_y} and the pointers towards them to be updated, the next and previous nodes of u_k with outgoing edges need to be identified. Let the maximum time required to identify these nodes be denoted by T_{up}. We describe later how to implement this operation efficiently and thus specifying T_{up}.

3.3 Internal Node Reordering

One of the most important operations supported by our graph structure is the internal node reordering. In many cases, there are algorithms that have some information beforehand about the sequence of accesses of the elements of the graph, and this can be exploited in speeding-up their performance. For example, if an algorithm needs to access all nodes in a topologically sorted order, a performance speed-up can be gained if these nodes have been already laid out in memory in that specific order. In this way, the cache misses get reduced, the memory transfers during the scanning of the nodes is optimal, and the algorithm has a much lower running time. In another example, some algorithms (e.g., [11]) use hierarchical decomposition techniques in order to obtain a small sub-graph which is considered more important than the rest of the graph. The nodes of this sub-graph are accessed much more frequently than the rest of the nodes. Therefore, improving the locality of those important nodes in memory can give a performance boost in the algorithm.

Our graph structure can internally change the relative position of the nodes and the edges, effectively changing their internal ordering. It does so, by removing an element from its original position and reinserting it to arbitrary new position. We call this operation an *internal relocation* of an element. In fact, a relocation is nothing more than a deletion and reinsertion of an element, two operations that have efficient running times and memory accesses. In order to relocate internally an edge (u_i, u_j), it is sufficient to delete the edge and then reinsert it in an arbitrary cell within the range R_{u_i}. In order to relocate internally a node u_i it suffices to delete the node and its adjacent edges E_{u_i} from the graph and then reinsert u_i and E_{u_i} in another arbitrary position.

3.4 Analysis

In this section we give the bounds of the operations supported by our graph structure. We start with the operation of scanning nodes and edges.

Lemma 1. *The packed-memory graph supports scanning of S consecutive nodes or S consecutive edges in $O(S)$ time and $O(1 + S/B)$ memory transfers.*

Proof. Scanning S consecutive nodes or S consecutive edges is equivalent to scanning S consecutive elements in a packed-memory array. Hence, the lemma follows from Theorem 1. □

The next lemma provides bounds for accessing a node's outgoing edges, a core subroutine in many algorithms especially of those based on Dijkstra's algorithm.

Lemma 2. *The packed-memory graph supports the scanning of all outgoing edges E_{u_i} of a node u_i in $O(|E_{u_i}|)$ time and $O(1 + |E_{u_i}|/B)$ memory transfers.*

Proof. In order to scan all outgoing edges E_{u_i} of a node u_i, we initially determine the range R_{u_i} in constant time by following the respective pointers from u_i. This range has size $O(|E_{u_i}|)$ and lies in consecutive cells in the packed-memory edge array. Now, the lemma follows from Theorem 1. □

We turn now to the dynamic operations in our graph structure. At first, we need to analyze the performance of inserting or deleting edges between existing nodes in the graph.

Lemma 3. *The packed-memory graph supports inserting (deleting) an edge in $O(T_{up}\log^2 m)$ amortized time and $O(1+\frac{T_{up}\log^2 m}{B})$ amortized memory transfers.*

Proof. In order to insert (delete) an edge, the actual operation is executed on the packed-memory edge array, which stores m elements. From Theorem 1, this takes $O(\log^2 m)$ amortized time and $O(1+\frac{\log^2 m}{B})$ amortized memory transfers.

The insert (delete) operation may cause a rebalance in an interval of the packed-memory edge array, and its respective ranges. For any existing edge that is moved to a different cell due to the rebalance, an update to the ranges may be needed. Any such update takes at most T_{up} time. Hence, overall any insert (delete) operation takes $O(T_{up}\log^2 m)$ amortized time and $O(1+\frac{T_{up}\log^2 m}{B})$ amortized memory transfers. □

In order to insert a node, our graph structure might need to move several nodes to different cells in the packed-memory node array and update all of their adjacent edges. The number of adjacent edges of any node is bounded by Δ which is the maximum degree of a node in the graph. Note that in large-scale transportation networks Δ is typically bounded by a small constant; for instance, in road networks $\Delta \leq 4$. The following lemma describes the performance of our graph structure when inserting a node to the graph.

Lemma 4. *The packed-memory graph supports inserting a node in $O(\Delta\log^2 n)$ amortized time and $O(1+\frac{\Delta\log^2 n}{B})$ amortized memory transfers, where Δ is the maximum node degree of the graph.*

Proof. In order to insert a node u_i in the graph, we need to insert it into the packed-memory node array. Such an insertion may cause some nodes to be moved to different cells. All adjacent edges of these nodes must be updated with the new position of the node. These edges lie in $O(\Delta)$ consecutive cells of the packed-memory edge array. Creating the pointers of u_i takes constant time, since they are set equal to the start pointer of u_{i+1}. Now, the lemma follows from Theorem 1 and Lemma 2. □

We now study the complexity of deleting a node from the graph. Clearly, a node cannot be deleted unless all of its adjacent edges are deleted. Thus, the process of deleting a node is complemented by the process of deleting all of its adjacent edges.

Lemma 5. *The packed-memory graph supports deleting a node in $O(\Delta T_{up}\log^2 m+\Delta\log^2 n)$ amortized time and $O(1+\frac{\Delta T_{up}\log^2 m+\Delta\log^2 n}{B})$ amortized memory transfers, where Δ is the maximum node degree of the graph.*

Proof. In order to delete a node u from the graph, first we need to delete all adjacent edges of u. Since the node has $O(\Delta)$ adjacent edges, we get from Lemma 3 that deleting all adjacent edges takes $O(\Delta T_{up}\log^2 m)$ amortized time and $O(1+\frac{\Delta T_{up}\log^2 m}{B})$ amortized memory transfers.

Then, we need to delete the node from the packed-memory node array which, from Theorem 1, takes $O(\log^2 n)$ amortized time and $O(1+\frac{\log^2 n}{B})$ amortized memory transfers. Such a deletion may cause some nodes to be moved to different cells and their adjacent edges have to be updated as before. These edges lie in $O(\Delta)$ consecutive cells of the packed-memory edge array. From Theorem 1 and Lemma 2, this takes $O(\Delta\log^2 n)$ amortized time and $O(1+\frac{\Delta\log^2 n}{B})$ amortized memory transfers.

Therefore, the deletion of a node u takes a total of $O(\Delta T_{up}\log^2 m+\Delta\log^2 n)$ amortized time and requires $O(1+\frac{\Delta T_{up}\log^2 m+\Delta\log^2 n}{B})$ amortized memory transfers. □

Finally, it remains to specify the actual time T_{up} needed for identifying the range of a node. As described before, this is the time needed to identify the immediately next and previous nodes of a given node u_i that have adjacent edges. The naive approach would be starting two linear searches from u_i, one towards the start of the node array and one towards its end that would end as soon as a node with adjacent edges is found in each direction. However, this may take up to $O(n)$ time which is not efficient.

An alternative would be to use a segment tree [5],[6, Section 10.3] over the packed-memory node array. This kind of tree can answer range queries in $O(\log n)$ time. Therefore, it can efficiently

identify the first nodes to the left and to the right of u_i that have outgoing edges in $O(\log n)$ time and thus $T_{up} = O(\log n)$.

A crucial observation, however, is that when the graph has no isolated nodes then both u_{i-1} and u_{i+1} have adjacent edges and can be identified in $O(1)$ time. In this case, the search routine finishes after just one step resulting in $T_{up} = O(1)$. Since we are interested in transportation networks which usually have no isolated nodes, we can safely use linear search to identify these nodes, a process that rarely needs to scan more than just a few adjacent cells. Therefore, for all practical purposes $T_{up} = O(1)$, which is actually verified by our experimental study.

3.5 Comparison with other Graph Structures

In this section, we compare our new graph structure with other graph structures on the basis of the three performance features set in the Introduction, namely compactness, agility, and dynamicity.

	Adjacency List	**Forward Star**	**Packed-memory Graph**
Space	$O(m+n)$	$O(m+n)$	$O(c_m m + c_n n)$
Time			
Scanning S edges	$O(S)$	$O(S)$	$O(S)$
Inserting/Deleting an edge	$O(1)$	$O(m)$	$O(\log^2 m)$
Inserting a node	$O(1)$	$O(n)$	$O(\Delta \log^2 n)$
Deleting a node u	$O(\Delta)$	$O(\Delta m + n)$	$O(\Delta \log^2 m + \Delta \log^2 n)$
Memory Transfers			
Scanning S edges	$O(S)$	$O(1 + S/B)$	$O(1 + S/B)$
Inserting/Deleting an edge	$O(1)$	$O(1 + m/B)$	$O(1 + \frac{\log^2 m}{B})$
Inserting a node	$O(1)$	$O(1 + n/B)$	$O(1 + \frac{\Delta \log^2 n}{B})$
Deleting a node u	$O(\Delta)$	$O(1 + \frac{\Delta m + n}{B})$	$O(1 + \frac{\Delta \log^2 m + \Delta \log^2 n}{B})$

Table 1: Comparison of space, running time and memory transfer complexities on all three graph data structures. B denotes the cache block size, while Δ denotes the maximum node degree.

An adjacency list representation implemented with linked lists seems like a reasonable candidate for a graph data structure. It supports optimal insertions/deletions of nodes and the scanning of the edges is fast enough to be used in practice. However, since there is no guarantee for the memory allocation scheme, the nodes and edges are most probably scattered in memory, resulting in many cache misses and less efficiency during scan operations, especially for large-scale networks. Finally, an adjacency list representation provides no support for any (re-)ordering of the nodes and edges in arbitrary relative positions in memory, since the allocated memory positions are not necessarily ordered. Therefore, an adjacency list representation implemented with linked lists favours no algorithm that can exploit any insight in memory accesses.

On the other hand, a forward star representation is optimal during the scan operations. Due to its layout, S consecutive edges are stored in at most $1 + \frac{S}{B}$ memory blocks. Hence, the least amount of blocks is transferred into the cache memory during a scan operation. Moreover, its elements can be reordered in-line in a way that will favour the memory accesses of any algorithm. However, an insertion (deletion) of a node or an edge in an arbitrary position must shift all subsequent elements in the array in order to make space for the new element. Clearly, this has $O(n)$ and $O(m)$ time complexity for nodes and edges, respectively, which is prohibitive for large-scale networks. Hence, the forward star representation cannot support fast enough insertions and deletions of elements.

A packed-memory graph representation is effective in all three features. It keeps the elements in the same way as the forward star representation, with only one difference: it keeps slightly larger arrays complemented with empty elements uniformly distributed within the array. Therefore, it accomplishes a performance within a factor of the forward star representation's performance in scanning consecutive elements. Also, it supports fast enough insertions and deletions of elements to be used in practical applications. Finally, the elements can be efficiently reordered in order to favour the memory accesses of any algorithm.

A summary of the complexity of the main operations in the aforementioned three graph structures is shown in Table 1. The above discussion and Table 1 show that our new graph structure is a good compromise, at the expense of a small space overhead, between the graph structures that accomplish two extreme complexities. It is a viable choice in dynamic scenarios, contrary to the forward star representation, without sacrificing any time or memory transfers complexity, and is clearly superior to the adjacency list representation in the most frequent operation of scanning consecutive edges, a core subroutine of all shortest path routing techniques.

4 Experiments

To assess the practicality of our new graph structure, we conducted a series of experiments on shortest path routing algorithms on real world large-scale transportation networks (European road networks) in dynamic scenarios. We considered mixed sequences of operations consisting of point-to-point shortest path queries and updates. For the point-to-point shortest path queries we used Dijkstra's algorithm and two classical speed-up techniques, the Bidirectional and the A^* variants of Dijkstra's algorithm. We used these speed-up techniques since our goal is to merely show the performance gain of our graph structure over the adjacency list and forward star representations, rather than beat the running times of the fastest possible algorithms.

Setup. All experiments were conducted on an Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz with a cache size of 6144Kb and 4Gb of RAM. Our applications were compiled by GCC version 4.4.3 with optimization level 3.

Data. The road networks for our experiments were acquired from [9] and consist of the road networks of Italy and Germany. The provided graphs are strongly connected and undirected. Hence, we consider each edge as bidirectional. Edge weights represent travel distances.

Algorithms. As already mentioned, the algorithms used are classical shortest path routing algorithms, in particular, the plain Dijkstra's algorithm, the Bidirectional Dijkstra's algorithm (B-Dijkstra), and the A^* variant of Dijkstra's algorithm, using Euclidean distances. Each of these algorithms was implemented once, supporting interchangeable graph representations in order to minimize the impact of factors other than the performance of the underlying data structures. All three graph structures were implemented having as target to keep only the essential core parts different, such as the accessing of a node or an edge. Thus, the only factor differentiating the experiments is the efficiency of accessing and reading, as well as inserting or deleting nodes and edges in each graph structure.

4.1 Results

Static performance

Query Times. We have executed 10000 random queries on each of the road networks, using all the above algorithms, on all three graph structures. The results are reported in Tables 2 and 3.

The results clearly show a speed-up of the packed-memory graph (PMG) over the adjacency list (AL) in all of our selected algorithms. Moreover, it is very close to the performance of the forward star (FS) representation, the difference being the additional holes added in the arrays. Losing less than 2% in query efficiency over the forward star representation, our graph structure has a clear dynamic potential, as shown in the next section.

The results also show that there is a trade-off between the node and edge density of the graph and its performance. The less holes (larger density) the packed-memory arrays have, the better the performance of the graph in queries. This is to be expected since the size of the graph gets larger with the addition of the holes. The advantage is that we can fine tune the density of the graph according to the application. If the application expects many updates, we will configure the graph to have smaller density in order to efficiently accommodate new nodes and edges. In less dynamic cases, we might raise the density to a much higher level, or even compress the packed-memory graph by removing all holes and continue statically.

		AL	FS	PMG	Speedup PMG vs AL
Italy	Size (Mb)	731.68	731.68	896	-22.46%
$n = 6686493$	Edge Scan (ms)	2002.84	987.88	1271.42	36.52%
$m = 14027956$	Dijkstra (ms)	1359.83	954.11	970.19	28.65%
Density	A^* Dijkstra (ms)	663.13	522.92	532.16	19.75%
$d_n = 80\%$ $d_m = 84\%$	B-Dijkstra (ms)	1058.95	730.67	743.06	29.83%

Table 2: Running times on the road network of Italy.

		AL	FS	PMG	Speedup PMG vs AL
Germany	Size (Mb)	1277.36	1277.36	1792	-40.29%
$n = 11548845$	Edge Scan (ms)	3821.61	1889.69	2404.77	37.07%
$m = 24738362$	Dijkstra (ms)	2629.46	1892.87	1938.62	26.27%
Density	A^* Dijkstra (ms)	846.75	666.85	680.38	19.65%
$d_n = 69\%$ $d_m = 74\%$	B-Dijkstra (ms)	1822.59	1287.61	1318.45	27.66%

Table 3: Running times on the road network of Germany.

Locality of reference. In order to measure the locality of reference we have monitored the main memory accesses during the execution of 10000 queries of Dijkstra's algorithm on our road networks. We call two consecutive reads/writes in memory a *hop* and its distance *hop size*. Clearly, the larger the hop size, the worse the locality of the algorithm. We have recorded each hop during consecutive node accesses and consecutive edge accesses and have plotted the distributions of their size in Figure 6. The results confirm our theoretical analysis and explain the running times in the previous section.

It is evident that the hop size is much smaller using the forward star or packed-memory graph representation, as expected. The edge hops are smaller in both these cases because the outgoing edges of a node are accessed all at once since they lie in consecutive memory addresses. The difference in the node hops is not so large since the pattern of accessing the nodes during the queries of the plain Dijkstra is not matched to the internal ordering of the nodes in the graph structure.

Moreover, hops larger than the block size will yield a block transfer in any case, hence the size of the hop over the block threshold is of no significant importance. However, it is obvious that in the forward star or the packed-memory graph representation it is much more likely for a hop to be smaller than the block size, therefore avoiding a cache miss and increasing efficiency. This is clearly the reason why the packed-memory graph and forward star representations are so much faster than the adjacency list in the static scenario.

Dynamic performance

Individual operations. In order to measure the performance of our graph structure in dynamic scenarios, we have compared it with the optimal performance of the adjacency list representation. Our dynamic operations include the random insertion and deletion of nodes and edges, and the internal relocation of nodes in random arbitrary relative positions. Since in an adjacency list representation there is no actual node ordering in memory, changing the relative order of the nodes is of no use. However, its performance can serve as a reference to the performance of our relocating routine. The results can be seen in Tables 4a and 4b.

The performance of the forward star representation is not reported in these experiments because its update operations were rather inefficient (close to a *million* times slower than the update performance of the other two graph structures).

Clearly, the adjacency list representation supports extremely fast update operations since it needs $O(1)$ time to allocate (deallocate) space and update its pointers. The performance of our graph structure is about an order of magnitude slower than the performance of the adjacency list representation. However, its update operations are still extremely fast to be used in practice. Hence, unless there is a need for excessively more update operations than queries, our graph structure should outperform the adjacency list. Our next experiment elaborates on this matter.

Random sequence of operations. In order to compare the adjacency list representation and the packed-memory graph representation in a typical dynamic scenario, we have compared their performance on sequences of random, uniformly distributed, mixed operations. These sequences contain either random queries using the A^* Dijkstra's algorithm or random updates in the graph, namely edge insertions and deletions.

All sequences contain the same amount of shortest path queries (1000 queries) with varying order of magnitude of updates in the range $[10^4, 10^8]$. To have the same basis of comparison, the same sequence of shortest path queries is used in all experiments.

The update operations are chosen at random between edge insertions and edge deletions. When inserting an edge, we do not consider this edge during the shortest path queries, since we do not want insertions to have any effect on them. In a deletion operation, we select at random a previously inserted edge to remove. We remove no original edges, since altering the original graph structure between shortest path queries would yield non-comparable results. The experimental results are reported in Figure 7, where the horizontal axis represents the ratio of the number of updates and the number of queries, while the vertical axis represents the total time for executing the sequence of operations.

The experiments verify our previous findings. While the running times of the queries dominate the running times of the updates, the packed-memory graph representation maintains a constant speed-up over the adjacency list representation. Our experiments show that the number of updates should be at least 50000 times more than the number of (A^* Dijkstra) queries in order for the packed-memory graph to be inferior than the adjacency list, a situation that is rather uncommon for the application scenario we consider.

Clearly, there are much faster speed-up techniques on Dijkstra's algorithm than its A^* variant, but even their running times dominate the running times of our updates. Since updates are much more rare in practical scenarios than shortest path queries, we can safely suggest the usage of our graph structure even for the fastest existing techniques.

5 Conclusion and Future Work

We have presented a new graph structure that is very efficient for routing in dynamic large-scale transportation networks. It builds upon the advantages of basic data structures, and at the same time remedies the drawbacks of existing graph structures.

We look forward to see the effects of our graph structure on other speed-up techniques, especially those that employ specific node orderings and hierarchical decomposition. We expect that the gain will be much larger in such techniques, since there will exist a natural matching between node importance in the algorithm's realm and node ordering within actual memory.

Finally, we are very interested in future implementations of the new graph structure on systems that will make even better use of its advantages. We firmly believe that its usage on slower memory hierarchies, like these of hand-held devices will improve the overall performance of the respective routing algorithms.

Acknowledgements. We would like to thank Daniel Delling for many fruitful and motivating discussions, and Kostas Tsichlas for introducing us to the cache-oblivious data structures.

Italy	Adjacency List	PMG
Insert / Delete nodes (μs)	0.31	2.74
Insert / Delete edges (μs)	0.71	7.01
Internal node relocations (μs)	2.41	29.61

(a) Italy

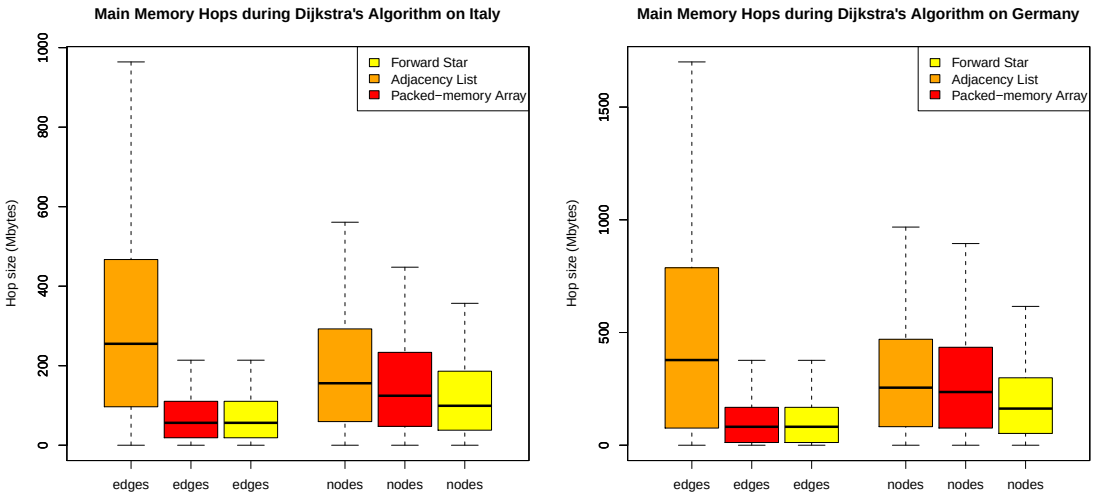
Germany	Adjacency List	PMG
Insert / Delete nodes (μs)	0.34	3.18
Insert / Delete edges (μs)	0.75	7.31
Internal node relocations (μs)	2.59	24.15

(b) Germany

Table 4: Dynamic operations

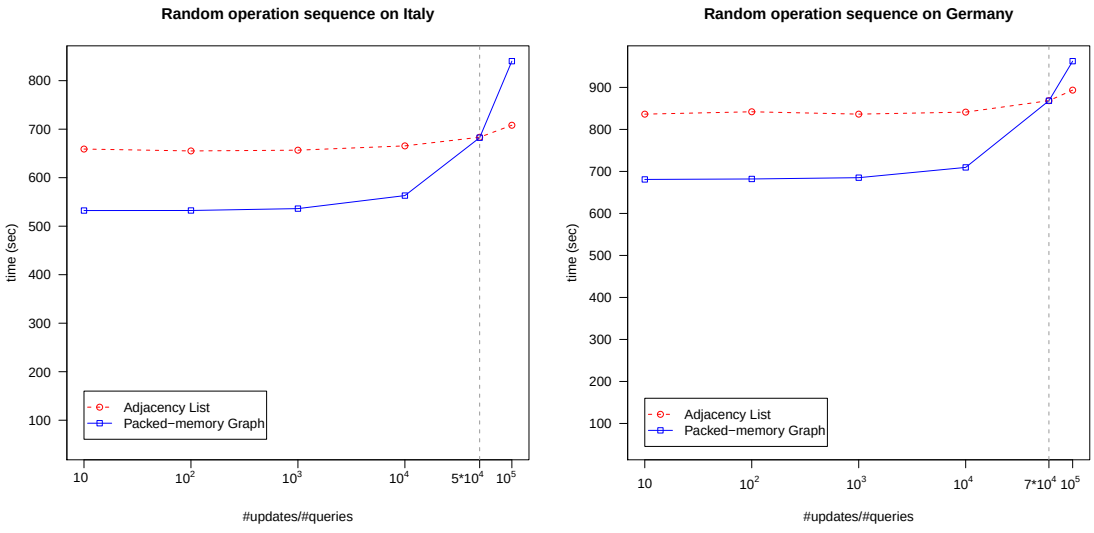
References

1. Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993) "Network Flows: Theory, Algorithms and Applications". Englewood Cliffs, NJ: Prentice Hall.



(a) (b)

Fig. 6: Memory hops during Dijkstra's algorithm on the road networks of Italy and Germany.



(a) (b)

Fig. 7: Running times on mixed sequences of operations consisting of 1000 queries and updates of varying length in $[10^4, 10^8]$.

2. ARRIVAL Deliverable D3.6, "Improved Algorithms for Robust and Online Timetabling and for Timetable Information Updating". ARRIVAL Project, March 2009, http://arrival.cti.gr/uploads/3rd_year/ARRIVAL-Del-D3.6.pdf.
3. Bauer, R., Delling, D.: "SHARC: Fast and robust unidirectional routing". ACM Journal of Experimental Algorithmics 14: (2009)
4. Bender, M. A., Demaine, E., Farach-Colton, M.: "Cache-Oblivious B-Trees". SIAM Journal on Computing, 35(2):341-358, 2005.
5. Bentley, J. L.: "Solutions to Klee's rectangle problem". Technical Report, Carnegie-Mellon University, Pittsburgh, 1977.
6. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: "Computational Geometry: Algorithms and applications". 3rd edition, 2008.
7. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.: "PHAST: Hardware-Accelerated Shortest Path Trees". In 25th International Parallel and Distributed Processing Symposium (IPDPS'11), IEEE, 2011
8. Dijkstra, E.K.: "A note on two problems in connexion with graphs". Numerische Mathematik 1 (1959) 269-271
9. 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, http://www.cc.gatech.edu/dimacs10/.
10. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: "Cache-oblivious algorithms". In Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS 99), p.285-297. 1999
11. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In Catherine C. McGeoch, editor, Proceedings of the 7th Workshop on Experimental Algorithms (WEA08), volume 5038 of Lecture Notes in Computer Science, pages 319333. Springer, June 2008.
12. Goldberg, A.V., Harrelson, C.: "Computing the Shortest Path: A^* Search Meets Graph Theory". In Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '05), 156-165
13. Goldberg, A. V., Kaplan, H., and Werneck, R. F. 2007. "Better Landmarks Within Reach". In Proceedings of the 6th Workshop on Experimental Algorithms (WEA07), C. Demetrescu, Ed. Lecture Notes in Computer Science, vol. 4525. Springer, 3851.
14. Goldberg, A. V., Kaplan, H., and Werneck, R. F. 2009. "Reach for A^*: Shortest Path Algorithms with Preprocessing". In The Shortest Path Problem: Ninth DIMACS Implementation Challenge, ser. DIMACS Book, C. Demetresku, A.V. Goldberg and D.S. Johnson, Eds. American Mathematical Society, 2009, vol.74, pp. 93-139
15. Prokop, H.: "Cache-oblivious algorithms". MSc Thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.
16. Sanders, P., Schultes, D.: "Engineering Highway Hierarchies". In 14th European Symposium on Algorithms (ESA), LNCS 4168, pp. 804-816. Springer, 2006.
17. Schultes, D., Sanders, P.: "Dynamic highway-node routing". In Proceedings of the 6th international conference on Experimental algorithms (WEA'07), 2007, pp. 66-79.
18. Wagner, D., Willhalm, T., Zaroliagis, C.: "Geometric Containers for Efficient Shortest Path Computation", ACM Journal of Experimental Algorithmics, Vol.10 (2005), No.1.3, pp.1-30.