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 – 005

A New Dynamic Graph Structure for Large-Scale Transportation Networks

Georgia Mali, Panagiotis Michail, Andreas Paraskevopoulos, and Christos Zaroliagis

October 2012

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

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

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

9 November 2012

Abstract. We present a new dynamic graph structure specifically suited for large-scale transportation networks. Our graph structure provides simultaneously three unique features: compactness, agility and dynamicity. All previously known graph structures could not support efficiently (or could not support at all) at least one of the aforementioned features, especially in large-scale transportation networks. We demonstrate the practicality and superiority of our new graph structure by conducting an experimental study for shortest route planning in large-scale European and USA road networks with a few dozen millions of nodes and edges. Our approach is the first one that concerns the dynamic maintenance of a large-scale graph with ordered elements using a contiguous part of memory, and which allows an arbitrary online reordering of its elements without violating its contiguity.

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, 8, 13, 15, 16]. These algorithms are clever extensions and/or variations of the classical Dijkstra's algorithm [9] – 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. 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 offline *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, 8, 13, 15, 16]. This optimization effectively improves the locality of the nodes by offline arranging 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 may suffer badly when updates occur, since an update must shift a great amount of elements in memory in order to keep compactness and locality. Updates either can occur explicitly (reflecting, for instance, changes in a road network varying from traffic jams and planned constructions to unforseen disruptions, etc), or may constitute an inherent part of an algorithm. The latter is evident in many state-of-the-art approaches, e.g., [3, 8, 13, 15, 16], which perform a mandatory preprocessing phase that introduces many "shortcuts" to the graph that in turn involve repetitively deleting and inserting edges, often intermixed with limited-scope executions of Dijkstra's algorithm. Hence, it is essential that any graph structure used in such cases can support efficient insertions and deletions of nodes and edges (dynamic graph structure).

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

⋆ This work was supported by the EU FP7/2007-2013 (DG CONNECT.H5-Smart Cities & Sustainability), under grant agreement no. 288094 (project eCOMPASS). This work was done while the last author was visiting the Karlsruhe Institute of Technology.

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 dynamic 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. For this reason, a dynamic version was developed [20] that offers constant time insertions and deletions of edges, at the expense of slower query times and limited agility. The main idea is to move a node's adjacent edges to the end of the edge array in order to insert new edges adjacent to the node.

Motivated by the efficiency of the (static and dynamic) forward star representation, we present a new graph structure for directed graphs, called *Packed-Memory Graph*, 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 static forward star representation (compactness).
- Nodes and edges can be reordered *online* 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 the dynamic forward star representation, 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 and USA road networks. To this end, we implemented the Bidirectional and the A^* variants of Dijkstra's algorithm, as well as all the ALT-based algorithms presented in [14]. Our goal was to merely show the performance gain of using our graph structure compared to the adjacency list or the dynamic forward star representation on shortest path routing algorithms, rather than beating the running times of the best speed-up techniques.

Our experiments showed that our graph structure: (i) can answer shortest path queries in milliseconds and handle updates of the graph layout, like insertions or deletions of both nodes and edges, in microseconds; (ii) outperforms the adjacency list and dynamic forward star graph structures in query time (frequent operations), while being a little slower in update time (rare operations). Hence, our graph structure is expected to be more efficient in practical applications with intermixed operations of queries and updates. In fact, our experiments with random and realistic such sequences showed the superiority of the packed-memory graph structure over the adjacency list and dynamic forward star graph structures, unless the number of updates compared to queries is excessively large.

Note that our graph structure is neither a speed-up technique, nor a dynamic algorithm. It can, however, increase the efficiency of any speed-up technique or dynamic algorithm implemented on top of it. Existing speed-up techniques can switch their underlying static graph structure to our new dynamic graph structure, thus keeping their stellar query performance while getting efficient update times almost for free. As our experiments show, this can be beneficial for the mandatory preprocessing phase of such techniques.

Similarly, dynamic approaches (e.g., [21, 22]) that operate on a static graph layout and which are only concerned with the development of dynamic algorithms that update the preprocessed data can also be implemented on top of our graph structure and hence benefit from its performance.

To the best of our knowledge, our approach is the first one that concerns the dynamic maintenance of a large-scale graph with ordered elements (i) using a *contiguous* part of memory, and (ii) allowing an arbitrary online reordering of its elements *without* violating its contiguity.

Our graph structure builds upon the cache-oblivious packed-memory array [4]. The main idea is to keep the graph's elements intermixed with empty elements so that insertions can be easily accommodated, while ensuring that deletions do not leave large chunks of memory empty. This is achieved by monitoring and reconfiguring the density of elements within certain parts of memory *without* destroying their relative order. It thus accomplishes two seemingly contradictory goals: maintaining all graph elements sorted and close to each other in memory, and efficiently inserting new elements in-between, resulting in a fully dynamic graph structure with great locality of references.

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 comparison to other graph structures. In Section 4, we present our experimental study on road networks. 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.

Graph Representations. There are multiple data structures for graph representations and their use depends heavily on the characteristics of the input graph and the performance requirements of each specific application. We assume the reader is familiar with the *adjacency list representation* of a graph. Each node keeps a list of references to its adjacent nodes (or edges), and all nodes are stored in a node list. Figure 2 shows[1] the adjacency list representation of the graph in Figure 1.

A variant of the adjacency list 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.

The main drawback of the forward star representation is that in order to insert an edge at a certain adjacency segment, all edges after the segment must be shifted to the right. Clearly, this is an $O(m)$ operation. For this reason, a dynamic version [20] of the forward star representation was developed where each adjacency segment has a size equal to a power of 2, containing the edges and some empty cells at the end. When inserting an edge, if there are empty cells in the respective segment, the new edge is inserted in one of them. Otherwise, the whole segment is moved to the end of the edge array, and its size is doubled. Deletions are handled in a virtual manner, just emptying the respective cells rather than deallocating reserved memory.

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 efficient 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, breaching its so-called *density thresholds*, 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. The density thresholds

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

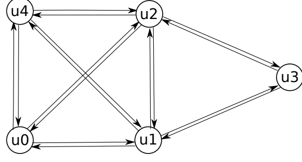


Fig. 1: A directed graph with 5 nodes and 16 edges.

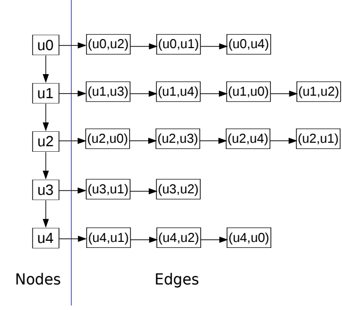


Fig. 2: Adjacency list representation.

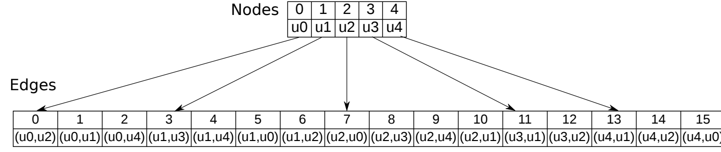


Fig. 3: Forward star representation.

and the rebalance ranges are monitored by a complete binary tree on top of the array. A full description of the packed-memory array is presented in the Appendix (Section A) and an example of a packed-memory array is shown in Figure 4.

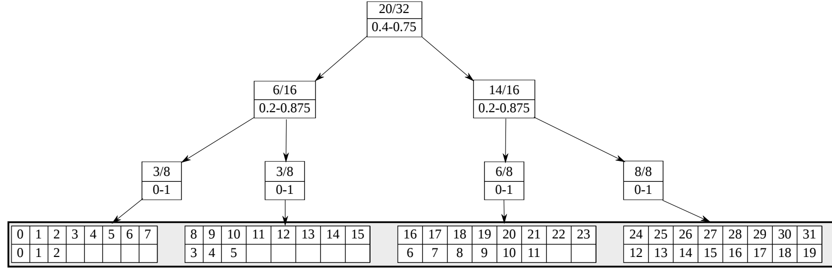


Fig. 4: Packed-Memory Array on the ordered set $[0, 19]$.

3 The Packed-Memory Graph (PMG)

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. This ordering may have a great impact on the performance of the algorithms implemented on top of our new graph structure.

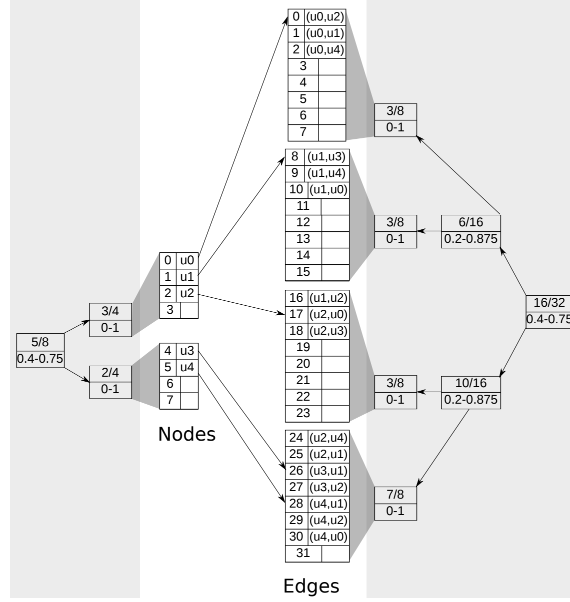


Fig. 5: Packed-memory Graph representation

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.

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. Their asymptotic bounds are given in the Appendix (Section B).

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}. In the Appendix (Section B) we describe how to implement this operation efficiently and explain that for all practical purposes $T_{up} = O(1)$.

Internal Node Reordering. Due to its design, our graph structure has the ability to internally reorder its nodes, which can be an attractive property. 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.

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.

3.3 Comparison with other dynamic 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. The asymptotic bounds of the main operations are shown in Table 1. The analysis and proofs on the complexities of PMG operations can be found in the Appendix (Section B).

	Adjacency List	**Dynamic Forward Star**	**Packed-memory Graph**
Space	$O(m+n)$	$O(m+n)$	$O(m+n)$
Time			
Scanning S edges	$O(S)$	$O(S)$	$O(S)$
Inserting/Deleting an edge	$O(1)$	$O(1)$	$O(\log^2 m)$
Inserting a node	$O(1)$	$O(1)$	$O(\Delta \log^2 n)$
Deleting a node u (with adjacent edges)	$O(\Delta)$	$O(n+\Delta)$	$O(\Delta \log^2 m + \Delta \log^2 n)$
Internal relocation of a node u (with adjacent edges)	not supported	not supported	$O(\Delta \log^2 m + \Delta \log^2 n)$
Memory Transfers			
Scanning S edges	$O(S)$	$O(1+S/B)$	$O(1+S/B)$
Inserting/Deleting an edge	$O(1)$	$O(1)$	$O(1+\frac{\log^2 m}{B})$
Inserting a node	$O(1)$	$O(1)$	$O(1+\frac{\Delta \log^2 n}{B})$
Deleting a node u (with adjacent edges)	$O(\Delta)$	$O(1+\frac{n+\Delta}{B})$	$O(1+\frac{\Delta \log^2 m+\Delta \log^2 n}{B})$
Internal relocation of a node u (with adjacent edges)	not supported	not supported	$O(1+\frac{\Delta \log^2 m+\Delta \log^2 n}{B})$

Table 1: Space, time and memory transfer complexities (the latter in the cache-oblivious model [12]). B: cache block size; Δ: maximum node degree (typically $O(1)$ in large-scale transportation networks).

An adjacency list representation implemented with linked lists seems like a reasonable candidate for a graph 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 (inherently) 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 dynamic forward star representation succeeds in storing the adjacent edges of each node in consecutive memory locations. Hence, the least amount of blocks is transferred into the cache memory during a scan operation of a node's adjacent edges. Moreover, it can efficiently insert or delete new edges in constant time. When inserting an edge adjacent to a node u, if there is space in the adjacency segment of u, it gets directly inserted there. Otherwise, the adjacency segment is moved to the end of the edge array, and its size is doubled. Therefore, it is clear that edge insertions can be executed in $O(1)$ amortized time and memory transfers. The edge deletion scheme consists of just emptying the respective memory cells, without any sophisticated rearranging operations taking place. Clearly, this also takes constant time. However, due to the particular update scheme, a specific adjacency segment ordering cannot be guaranteed. Moving an adjacency segment to the end of the edge array clearly destroys any locality of references between edges with different endpoints, resulting in slower query operation. Since the nodes are stored in an array, inserting a new node u at the end of the array is performed in constant time, while deleting u must shift all elements after u and therefore takes $O(n)$ time in addition to the time needed to delete all edges adjacent to u. Finally, the dynamic forward star as it is designed cannot support internal relocation of a node u (with adjacent edges). Simply inserting some new adjacent edges causes u to be moved again to the end of the array, rendering the previous relocation attempt futile.

A packed-memory graph representation, due to its memory management scheme, achieves great locality of references at the expense of a small update time overhead (but fast enough to be used in practical applications). No update operation can implicitly alter the relative order of the nodes and edges, therefore relative elements are always stored in memory as close as possible. Finally, the elements can be efficiently reordered in order to favour the memory accesses of any algorithm.

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 and USA road networks) in static and dynamic scenarios. For the former, we use the query performance of the static forward star (FS) graph structure as a reference point, since FS stores all edges adjacent to a node in consecutive memory locations. For the latter, we considered mixed sequences of operations consisting of point-to-point shortest path queries and updates. Our goal was to merely show the performance gain of our graph structure (PMG) over the adjacency list (ADJ) and dynamic forward star (DynFS) representations, rather than beating the running times of the currently fastest shortest path routing algorithms. Since we are studying general-purpose dynamic graph structures, we do not use FS in the dynamic scenario, because it (naturally) fails in such a scenario (as it was indeed confirmed by experiments we conducted).

All experiments were conducted on an Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz with a cache size of 6144Kb and 8Gb of RAM. Our implementations were compiled by GCC version 4.4.3 with optimization level 3. The road networks for our experiments were acquired from [10, 11] and consist of the road networks of Italy, Germany, Western USA and Central USA. The provided graphs are connected and undirected. Hence, we consider each edge as bidirectional. Edge weights represent travel distances. Experiments conducted with travel times exhibited similar relative performance and are reported in the Appendix (Section C).

4.1 Algorithms

In order to assess the performance of our graph structure we decided to implement the full set of shortest path algorithms considered in [14]. These algorithms are based on the well known Dijkstra's algorithm,

and its Bidirectional and A^* [17] variants. Recall that Dijkstra's algorithm grows a full shortest path tree rooted at a source node s keeping a partition of the node set V into a set of nodes with permanent distances (maintained implicitly), and a set of nodes with tentative distances maintained explicitly in a priority queue Q, where the priority of a node v is its tentative distance $d(s, v)$ from s. In each iteration, the node u with minimum tentative distance $d(s, u)$ is deleted from Q, its adjacent vertices get their tentative distances updated, and u thus becomes *settled*.

Given a source node s and a target node t, the A^* (or Dijkstra's goal-directed) variant [17] is similar to Dijkstra's algorithm with the difference that the priority of a node in Q is modified according to a heuristic function $h_t : V \to \mathbb{R}$ which gives a lower bound estimate $h_t(u)$ for the cost of a path from a node u to a target node t. By adding this heuristic function to the priority of each node, the search is pulled faster towards the target. The tighter the lower bound is, the faster the target is reached. The only requirement is that the h_t function must be monotone: $h_t(u) \leq wt(u, v) + h_t(v), \forall(u, v) \in E$. One such heuristic function is the Euclidean distance between two nodes, which is clearly a lower bound of their distance in Euclidean graphs.

The key contribution in [14] is a highly effective heuristic function for the A^* algorithm using the triangle inequality theorem and precomputed distances to a few important nodes, the so-called *landmarks*. The resulting algorithm is called ALT. Its preprocessing involves computing and storing the shortest path distances between all nodes and each of the selected landmarks. Then, during a query, the algorithm computes the lower bounds in constant time using the precomputed shortest distances with the triangle inequality. The efficiency of ALT depends on the initial selection of landmarks in the graph and the number of landmarks that are used during the query. In order to have good query times, peripheral landmarks as far away from each other as possible must be selected.

The A^* algorithm can also be used in a bidirectional manner. The forward search from s is the same as in the unidirectional variant, while the backward search from t operates on the inverse graph $\overleftarrow{G} = (V, \overleftarrow{E})$, $\overleftarrow{E} = \{(v, u) | (u, v) \in E\}$ using the heuristic function h_s instead of h_t.

In [14], two approaches are used as the quitting criteria of the bidirectional A^* algorithm. In the *symmetric approach* the search stops when one of the searches is about to settle a node v for which $d(s, v) + h_t(v)$ (forward search) or $d(v, t) + h_s(v)$ (backward search) is larger than the shortest path distance found so far. In the *consistent approach*, the heuristic functions H_s and H_t are used instead of h_s and h_t, such that $H_s(u) + H_t(u) = c$, where c is a constant. These are defined as either *average heuristic functions* $H_t(u) = -H_s(u) = \frac{h_t(v) - h_s(v)}{2}$ or *max heuristic functions* $H_t(v) = \max\{h_t(v), h_s(t) - h_s(v) + b\}$ and $H_s(v) = \min\{h_s(v), h_t(s) - h_t(v) + b\}$, where b is a constant.

In summary, our performance evaluation consists of the following algorithms:

- D: Dijkstra's algorithm.
- B: The bidirectional variant of Dijkstra's algorithm.
- AE: A^* search with Euclidean lower bounds.
- BEM: The bidirectional AE algorithm with the max heuristic function.
- BEA: The bidirectional AE algorithm with the average heuristic function.
- AL: Regular ALT algorithm.
- BLS: The symmetric bidirectional ALT algorithm.
- BLM: The consistent bidirectional ALT algorithm with the max heuristic function.
- BLA: The consistent bidirectional ALT algorithm with the average heuristic function.

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 graph structures. All three graph structures (ADJ, DynFS, PMG) were implemented under an abstraction layer, 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. Clearly, the abstraction layer yields a small performance penalty, however, it provides the opportunity to compare all graph structures in a fair, uniform manner.

4.2 Results

Static performance. We start by analyzing and comparing the real-time performance of the aforementioned algorithms on a static scenario, i.e., consisting only of shortest path queries. Following [14], we used

two performance indicators: (a) the shortest path computation time (machine-dependent), and (b) the *efficiency* defined as the number of nodes on the shortest path divided by the number of the settled nodes by the algorithm (machine-independent).

In each experiment, we considered 10000 shortest path queries. For each query, the source s and the destination t were selected uniformly at random among all nodes. In our experiments with the ALT-based algorithms (AL,BLS,BLM,BLA) we used at most 16 landmarks, as it is the case in [14]. Note that the query performance of these algorithms can be increased by adding more well-chosen landmarks.

Our experimental results on the road networks of Germany and Central USA are shown in Tables 2 and 3 (the results on the other road networks are reported in the Appendix, Section C). For each algorithm and each graph structure, we report the average running times in ms and their standard deviation (in parentheses). We also report the space consumption for each graph structure. As expected, our experiments

Germany	$n = 11{,}548{,}845$ $m = 24{,}738{,}362$				
	Time(ms)				Efficiency(%)
	ADJ (2.365Gb)	**DynFS** (2.542Gb)	**PMG** (3.328Gb)	**FS** (2.365Gb)	
D	2,870.44 (1666.22)	2,669.67 (1,511.12)	2,007.51 (1,175.17)	2,005.93 (1,152.05)	0.10 (0.14)
B	1,965.98 (1282.14)	1,768.85 (1,136.14)	1,375.55 (909.70)	1,361.06 (884.04)	0.17 (0.32)
AE	709.60 (620.53)	599.94 (514.33)	513.58 (456.45)	509.28 (441.46)	0.61 (1.04)
BEM	798.41 (710.95)	704.26 (616.54)	591.77 (534.19)	579.36 (513.39)	0.66 (1.11)
BEA	697.06 (630.75)	609.92 (538.30)	511.14 (468.54)	502.96 (449.66)	0.76 (1.37)
AL	139.80 (142.96)	118.99 (121.98)	110.57 (114.01)	98.61 (102.34)	5.81 (12.50)
BLS	160.99 (175.77)	138.05 (150.86)	125.42 (137.26)	111.60 (122.47)	5.48 (9.66)
BLM	126.92 (170.90)	112.05 (152.07)	102.76 (140.79)	93.28 (128.76)	8.81 (10.39)
BLA	94.11 (126.93)	85.47 (115.48)	76.30 (103.79)	68.73 (94.03)	12.23 (15.89)

Table 2: Running times in the road network of Germany. Standard deviations are indicated in parentheses.

Central USA	$n = 14{,}081{,}816$ $m = 34{,}292{,}496$					
	Time(ms)				Efficiency(%)	Efficiency(%)
	ADJ (3.054Gb)	**DynFS** (3.269Gb)	**PMG** (3.297Gb)	**FS** (3.054Gb)		[14]
D	4,474.65 (2,616.76)	3,418.06 (2,037.16)	2,864.33 (1,699.87)	2,766.28 (1,666.86)	0.07 (0.11)	0.09
B	3,012.38 (1,942.74)	2,303.91 (1,506.18)	2,048.01 (1,346.31)	1,963.67 (1,302.10)	0.11 (0.19)	0.14
AE	2,055.29 (1,449.96)	1,522.24 (1,080.02)	1,414.35 (1,011.49)	1,370.49 (1,158.41)	0.17 (0.11)	0.10
BEM	1,881.84 (1,283.16)	1,448.47 (992.24)	1,274.99 (1,051.98)	1,236.90 (1,019.23)	0.19 (0.12)	–
BEA	1,853.45 (1,276.14)	1,429.97 (989.39)	1,237.39(1,017.03)	1,208.97 (992.65)	0.20 (0.12)	0.14
AL	223.26 (223.75)	173.62 (176.78)	161.93 (162.383)	151.34 (146.69)	3.75 (9.42)	1.87
BLS	257.47 (278.42)	205.35 (224.89)	176.84 (188.33)	172.46 (183.39)	3.67 (7.38)	2.02
BLM	211.50 (285.76)	172.11 (236.39)	161.27 (224.779)	157.41 (213.92)	7.43 (10.33)	3.27
BLA	146.51 (188.625)	116.88 (151.434)	108.401 (142.124)	104.88 (135.35)	10.02 (15.57)	3.87

Table 3: Running times in the road network of Central USA. Standard deviations are indicated in parentheses.

confirm the ones in [14] regarding the relative performance of the evaluated algorithms, with BLA being the algorithm with the highest performance (for an explanation, see Appendix, Section C).

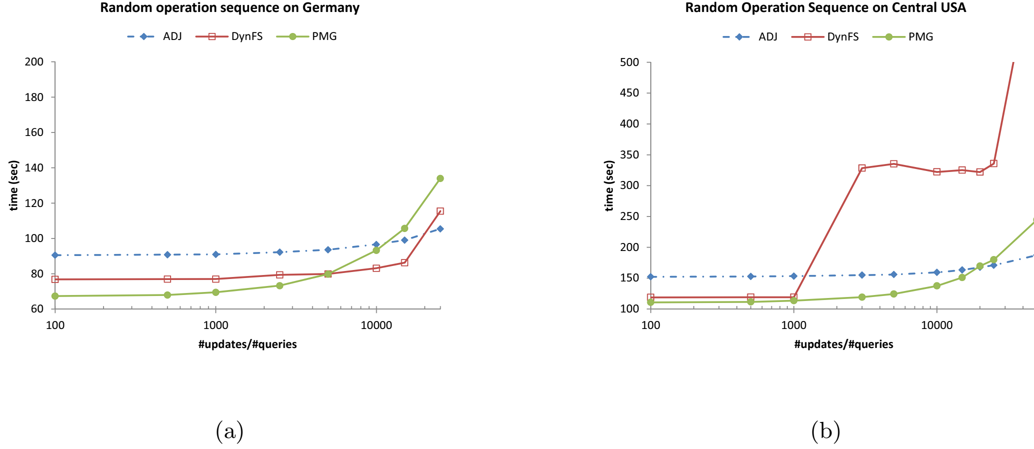
The major outcome of our experimental study is that there is a clear speed-up of PMG over ADJ and DynFS in all selected algorithms. This is due to the fact that, at the expense of a small space overhead, the packed-memory graph achieves greater locality of references, less cache misses, and hence, better performance during query operations. In our experiments, PMG is roughly 25% faster than ADJ and 10% faster than DynFS. In addition, its performance is very close to the optimal performance of the static forward star (FS), taking into consideration that PMG is a dynamic structure. It is roughly 9% slower in Germany (sparser PMG) and roughly 3% slower in the larger graph of Central USA (denser PMG). Note

that the denser the PMG is, the smaller size it has, the more it resembles an FS and the more it matches its query performance. Hence, if we take FS as a reference point for query performance in static scenarios, then PMG is the dynamic structure closest to it.

The space overhead of PMG is such because we chose its node and edge densities in a way that the sizes of the node and edge arrays are equal to the next power of 2 of the space needed. Note that the node and edge densities can be fine-tuned according to our expectation of future updates and the particular application. For example, the PMG size in the German network could be further reduced after fine-tuning. However, this is not our prime concern in this experimental study.

Note also that our PMG implementations achieve larger efficiency compared to those reported in [14] for the same algorithms, which were applied on a roughly 70% smaller part of the Central USA road network (consisting of 4,130,777 nodes and 9,802,953 edges). We can increase further the time performance of the algorithms by extending the number of landmarks at the expense of some more memory consumption. For instance, using 22 landmarks on a PMG loaded with the Central USA road network, BLA queries are reduced to an average of 101.51ms.

Dynamic performance. We report results on random and realistic sequences of operations using the three dynamic graph structures (ADJ, DynFS, PMG). The performance of FS 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 graph structures). Experimental results with individual dynamic operations and internal node relocations are reported in the Appendix (Section C).



(a) (b)

Fig. 6: Running times on mixed sequences of operations consisting of 1000 queries and updates of varying length in $[10^5, 5 \times 10^7]$.

Random sequence of operations. We have compared the performance of the graph structures on sequences of random, uniformly distributed, mixed operations as a typical dynamic scenario. These sequences contain either random shortest path queries (BLA) 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 $[10^5, 5 \times 10^7]$. 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 6, where the horizontal axis (log-scale) 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 speed-up over the adjacency list and dynamic forward star representations. In the road network of Germany, the number of updates should be at least 5000 times more than the number of (BLA) queries in order for the packed-memory graph to be inferior to the dynamic forward star, and at least 10000 more to be inferior to the adjacency list representation. Accordingly, in the road network of Central USA, the ratio for the adjacency list is 20000, while the dynamic forward star exhibits a very slow behaviour after a certain point, due to its greedy allocation scheme. DynFS manages to have good running times in Germany, which is smaller in size, and hence more memory space is available for allocation. However, the running times of DynFS are hindered by the blow-up in its size when it operates on a larger graph, which is apparent with the Central USA road network. Finally, note that even 5000 times more updates than queries is rather uncommon for the application scenario we consider.

Realistic sequence of operations. In order to compare the graph structures in a typical practical scenario, we have measured running times for the preprocessing stage of Contraction Hierarchies(CH) [13]. The CH preprocessing stage iteratively removes nodes and their adjacent edges from the graph, and shortcuts their neighbouring nodes if the removed edges represent shortest paths (this is examined by performing consecutive shortest path queries). We have recorded the sequence of operations (shortest path queries, edge insertions, node and edge deletions) executed by the preprocessing routine of the CH code [6], and given it as input to our three data structures. This is a well suited realistic example, since it concerns one of the most successful techniques for route planning requiring a vast amount of shortest path queries and insertions of edges. Note that CH treats deletions of nodes and edges virtually, hence they are not included in our sequence of operations. We used the aggressive variant [6] of the CH code, since it provides the most efficient hierarchy. The results are shown in Table 4.

Germany	Hierarchy Construction (sec)
ADJ	32,194.4
DynFS	19,938.2
PMG	17,087.7

Table 4: CH preprocessing on Germany with 3,615,855 shortest path queries and 21,342,855 edge insertions.

On all three graph structures, insertions can be processed so fast that have a minimum effect on the total running time of this sequence, even though they dominate (by a factor of 6) the shortest path queries. On the other hand, shortest path queries are the slowest operations on this sequence, and since our graph structure has the best performance on them, it can easily outperform the two other graph structures. Our experiments also show that the use of PMG could improve the particular stage of the CH preprocessing by at least 14%.

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.

References

1. Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993) "Network Flows: Theory, Algorithms and Applications". Englewood Cliffs, NJ: Prentice Hall.
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. Contraction Hierarchies source code, http://algo2.iti.kit.edu/routeplanning.php
7. de Berg, M., Cheong, O., van Kreveld, M., Overmars, M.: "Computational Geometry: Algorithms and applications". 3rd edition, 2008.
8. 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
9. Dijkstra, E.K.: "A note on two problems in connexion with graphs". Numerische Mathematik 1 (1959) 269-271
10. 9th DIMACS Implementation Challenge – Shortest Paths, http://www.dis.uniroma1.it/challenge9/index.shtml.
11. 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, http://www.cc.gatech.edu/dimacs10/.
12. 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
13. 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 (WEA 08), volume 5038 of Lecture Notes in Computer Science, pages 319333. Springer, June 2008.
14. 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
15. Goldberg, A. V., Kaplan, H., and Werneck, R. F. 2007. "Better Landmarks Within Reach". In Proceedings of the 6th Workshop on Experimental Algorithms (WEA 07), C. Demetrescu, Ed. Lecture Notes in Computer Science, vol. 4525. Springer, 3851.
16. 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. Demetrescu, A.V. Goldberg and D.S. Johnson, Eds. American Mathematical Society, 2009, vol.74, pp. 93-139
17. Hart, P. E., Nilsson, N. J., Raphael, B., "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics, 4:4(1968).
18. Prokop, H.: "Cache-oblivious algorithms". MSc Thesis, Massachusetts Institute of Technology, Cambridge, MA, 1999.
19. Sanders, P., Schultes, D.: "Engineering Highway Hierarchies". In 14th European Symposium on Algorithms (ESA), LNCS 4168, pp. 804-816. Springer, 2006.
20. Schultes, D.: "Route Planning in Road Networks". PhD Dissertation, University of Karlsruhe, 2008.
21. Schultes, D., Sanders, P.: "Dynamic highway-node routing". In Proceedings of the 6th international conference on Experimental algorithms (WEA'07), 2007, pp. 66-79.
22. 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.

APPENDIX

All bounds regarding memory transfers throughout the paper concern the *cache-oblivious model* [12, 18], which accounts memory transfers in memory blocks of unknown size B. In particular, there is a two-level memory hierarchy consisting 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).

A 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.

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 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.*

B Complexity Analysis of PMG Operations

In this section we give the asymptotic 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. □

Regarding the complexity of relocating internally a node, we can treat this operation as a sequence of update operations in the graph.

Lemma 6. *The packed-memory graph supports relocating 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 relocate a node u_i to an arbitrary relative position between two other nodes u_j and u_{j+1}, we also need to relocate the edge range R_{u_i} in-between the edge ranges R_{u_j} and $R_{u_{j+1}}$.

First, we delete the node and all of its adjacent edges 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 using Lemma 5. Then, we reinsert the node in the desired position in $O(\Delta \log^2 n)$ amortized time and $O(1 + \frac{\Delta \log^2 n}{B})$ using Lemma 4. Finally, we reinsert all of its $O(\Delta)$ adjacent edges in $O(\Delta T_{up} \log^2 m)$ amortized time and $O(1 + \frac{\Delta T_{up} \log^2 m}{B})$ amortized memory transfers using Lemma 3. Therefore, an internal relocation of a node u takes a total of $O(\Delta \log^2 n + \Delta T_{up} \log^2 m)$ amortized time and $O(1 + \frac{\Delta \log^2 n + \Delta T_{up} \log^2 m}{B})$ 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],[7, 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.

C Further Experimental Results

Static performance. In this section we present more experiments on static operations using all three dynamic graph structures. Experiments with the FS graph structure are not shown since they are similar to the ones already reported. Tables 5 and 6 show the experimental results on the road networks of Italy and Western USA using travel distances as edge weights. As expected, our experiments confirm the ones in [14] regarding the relative performance of the evaluated algorithms. Algorithms D and B have the worst time and efficiency. The A^*-based algorithms (AE,BEM,BEA,AL,BLS,BLM,BLA) perform far better since they use a goal directed orientation. Amongst them, the efficiency depends on the accuracy of the heuristic function that is used. The algorithms using the Euclidean distance heuristic function (AE,BEM,BEA) do not perform as well as the ones using ALT(AL,BLS,BLM,BLA), since the ALT algorithms generally provide more accurate and tighter lower bounds.

Italy	$n = 6{,}686{,}493$ $m = 14{,}027{,}956$			
	Time(ms)			Efficiency(%)
	ADJ (1.356Gb)	**DynFS** (1.458Gb)	**PMG** (1.664Gb)	
D	1,517.67 (871.47)	1,236.76 (694.37)	1,019.23 (587.98)	0.21 (0.45)
B	1,165.93 (769.07)	988.90 (642.15)	780.54 (517.21)	0.35 (0.85)
AE	549.92 (523.31)	471.06 (436.61)	392.11 (378.18)	0.92 (1.63)
BEM	516.08 (493.12)	461.82 (435.61)	372.03 (360.68)	1.10 (1.85)
BEA	462.08 (442.37)	415.66 (397.57)	330.48 (320.98)	1.22 (1.97)
AL	124.20 (129.39)	115.49 (117.96)	95.65 (100.80)	6.60 (13.01)
BLS	127.85 (134.28)	119.87 (133.46)	97.04 (102.62)	6.66 (10.90)
BLM	93.17 (121.65)	86.95 (123.45)	73.00 (96.71)	12.20 (13.89)
BLA	73.78 (89.13)	68.95 (96.13)	57.66 (70.47)	14.92 (17.74)

Table 5: Running times in the road network of Italy. Standard deviations are indicated in parentheses.

Western USA	$n = 6{,}262{,}104$ $m = 15{,}248{,}146$				
	Time(ms)			Efficiency(%)	Efficiency(%)
	ADJ (1.360Gb)	**DynFS** (1.456Gb)	**PMG** (1.664Gb)		[14]
D	1,782.44 (1,021.61)	1,401.3 (805.07)	1,097.06 (638.37)	0.11 (0.08)	0.15
B	1,328.11 (852.90)	1,042.56 (671.69)	841.16 (549.22)	0.17 (0.15)	0.21
AE	1,087.93 (809.58)	797.07 (596.75)	695.34 (527.97)	0.22 (0.37)	0.16
BEM	993.84 (717.88)	766.53 (554.30)	665.95 (488.49)	0.27 (0.51)	–
BEA	982.59 (714.07)	760.89 (554.19)	654.35 (483.65)	0.28 (0.51)	0.21
AL	109.63 (119.06)	91.43 (100.15)	73.33 (80.99)	4.93 (10.84)	2.69
BLS	114.83 (126.94)	93.76 (103.83)	78.18 (86.94)	5.02 (8.65)	3.47
BLM	110.65 (167.98)	91.14 (138.72)	80.49 (123.98)	9.78 (12.60)	5.63
BLA	78.27 (106.60)	64.08 (87.46)	55.63 (76.57)	12.00 (17.19)	7.21

Table 6: Running times in the road network of Western USA. Standard deviations are indicated in parentheses.

As we mentioned earlier, we follow different schemes on the A^*-based bidirectional variants regardless of their heuristic function. In the symmetric scheme, the lower bounds are tighter than in the consistent scheme, but the two searches cannot stop immediately when they meet. This leads to a larger search space, and higher query times (BLS). It seems that stopping the search early is more important than keeping the bound intact, hence the consistent approach is more efficient (BEM and BEA vs AE, BLM and BLA vs BLS). Using the consistent approach, the average heuristic (BLA) has better performance than the max heuristic (BLM). Depending on b, the max heuristic (BEM, BLM) cannot always give tight lower bounds, resulting in slower performance. This is confirmed by the difference in the standard deviation of running times between them (BEM vs BEA, and BLM vs BLA). Overall, BLA is the algorithm with the highest performance.

We now turn to experimental results using travel times as edge weights. In Tables Tables 7 and 8, we report the experiments on the two largest road networks, Western and Central USA. We observe a similar relative performance among the implemented algorithms.

Individual dynamic 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. The performance for internal node relocations in the PMG structure is not reported since it is the aggregate of deleting a node (and its edges) and reinserting it in an arbitrary position. The results can be seen in Tables 9a and 9b.

Clearly, the adjacency list and the dynamic forward star representations support extremely fast update operations since they need $O(1)$ time to allocate (deallocate) space and update their pointers. The only exception is deleting a node on a dynamic forward star representation, which is slower, confirming its theoretical time complexity of Table 1. The performance of our graph structure is about an order of magnitude slower than the performance of the other two structures. However, its update operations are still extremely fast to be used in practice. Hence, unless there is a need for excessively more update

Western USA	n = 6,262,104 m = 15,248,146				
	Time(ms)			Efficiency(%)	Efficiency(%)
	ADJ (1.360Gb)	**DynFS** (1.456Gb)	**PMG** (1.664Gb)		[14]
D	1863.95 (1066.23)	1432.72 (822.74)	1186.87 (689.66)	0.12 (0.10)	0.2
B	1301.15 (870.66)	988.35 (664.07)	849.93 (580.93)	0.21 (0.24)	0.2
AE	994.97 (795.22)	738.39 (594.42)	693.86 (571.24)	0.28 (0.27)	0.2
BEM	848.51 (697.58)	633.22 (523.72)	616.45 (521.37)	0.41 (0.48)	–
BEA	826.70 (686.48)	615.23 (513.65)	595.24 (508.91)	0.42 (0.51)	0.2
AL	155.61 (154.95)	125.60 (126.14)	106.82 (107.47)	3.71 (8.46)	2.8
BLS	143.66 (153.99)	115.12 (124.46)	99.69 (108.88)	3.94 (6.81)	3.1
BLM	88.72 (134.71)	72.11 (110.28)	67.50 (105.05)	13.12 (15.57)	4.9
BLA	61.31 (80.80)	49.41 (65.40)	45.08 (60.40)	15.69 (19.85)	6.0

Table 7: Running times in the road network of Western USA using travel times as cost function. Standard deviations are indicated in parentheses.

Central USA	n = 14,081,816 m = 34,292,496				
	Time(ms)			Efficiency(%)	Efficiency(%)
	ADJ (3.054Gb)	**DynFS** (3.269Gb)	**PMG** (3.297Gb)		[14]
D	4758.52 (2837.64)	3518.57 (2099.37)	3182.41 (1910.40)	0.06 (0.09)	0.1
B	2895.94 (1953.81)	2153.86 (1458.77)	2026.16 (1397.67)	0.13 (0.28)	0.1
AE	2342.58 (1962.78)	1697.76 (1440.96)	1626.50 (1412.69)	0.16 (0.20)	0.1
BEM	1736.55 (1437.87)	1282.41 (1068.57)	1269.35 (1077.21)	0.25 (0.41)	–
BEA	1701.69 (1412.59)	1254.65 (1048.73)	1216.97 (1043.65)	0.25 (0.43)	0.2
AL	291.74 (275.98)	226.72 (216.57)	193.99 (187.36)	2.35 (7.20)	2.1
BLS	300.96 (303.43)	235.74 (239.63)	202.54 (208.31)	2.65 (6.21)	2.2
BLM	185.26 (270.61)	149.68 (221.07)	130.60 (194.65)	8.38 (11.85)	3.9
BLA	116.07 (146.35)	91.76 (116.72)	85.72 (110.47)	10.82 (15.91)	5.5

Table 8: Running times in the road network of Central USA using travel times as cost function. Standard deviations are indicated in parentheses.

operations than queries, our graph structure should outperform both the adjacency list and the dynamic forward star representation.

Germany	**ADJ**	**DynFS**	**PMG**
Insert node	0.20	0.12	3.33
Delete node (with adjacent edges)	1.78	4.73	8.62
Insert edge	0.74	0.75	7.31
Delete edge	0.79	0.80	7.11

(a) Germany

Central USA	**ADJ**	**DynFS**	**PMG**
Insert node	0.29	0.19	6.05
Delete node (with adjacent edges)	1.77	21.49	39.88
Insert edge	1.06	1.08	4.68
Delete edge	1.03	1.05	34.21

(b) Central USA

Table 9: Time (μs) for 10,000 single dynamic operations