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 Techn. Institute & Press "Diophantus", Patras University Campus, GR
[2] Dept of Computer Eng. & Informatics, University of Patras, 26504 Patras, Greece
{mali,michai,paraskevop,zaro}@ceid.upatras.gr

Abstract. We present a new dynamic graph structure specifically suited for large-scale transportation networks that provides simultaneously three unique features: compactness, agility and dynamicity. We demonstrate its practicality and superiority by conducting an experimental study for shortest route planning in large-scale European and US 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 memory part, and which allows an arbitrary online reordering of its elements.

1 Introduction

In recent years we observe a tremendous amount of research for efficient route planning in road and other public transportation networks, witnessing extremely fast algorithms that answer point-to-point shortest path queries in a few msecs (in certain cases even less) in road networks with a few dozen millions of nodes and edges after a certain preprocessing phase; see e.g., [3,6,10,12]. These algorithms, known as *speed-up techniques*, are clever extensions/variations of the classical Dijkstra's algorithm. Speed-up techniques employ not only heuristics to improve the query search space, but also optimizations in the way they are implemented. The graph structures used (mostly variations of the adjacency list representation) 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 reference locality. The latter, known as *internal node reordering*, effectively improves node locality by offline arranging nodes within memory, thus improving cache efficiency and query running times.

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

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

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., [3,6,10,12], 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. Thus, it is essential that any graph structure used in such cases can support efficient insertions and deletions of nodes and edges (dynamic graph structure). Hence, for efficient routing in large-scale networks, a graph structure is required supporting the following features.

1. *Compactness*: ability to efficiently access adjacent nodes or 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.

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 [14] 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 adjacent 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 US road networks, implementing the Bidirectional and the A^* variants of Dijkstra's algorithm, as well as all the ALT-based algorithms in [11]. Our goal was to merely show the performance gain of using our 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 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, a fact verified by our experiments. 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. As our experiments show, this can be beneficial for the mandatory preprocessing phase of such techniques.

To the best of our knowledge, our approach is the first one concerning 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. Exposition details and missing proofs due to space limitations, in the rest of the paper, can be found in the full version [13].

2 Preliminaries

Let $G = (V, E)$ be a directed graph with node set V ($n = |V|$), edge set E ($m = |E|$), and edge weight function $wt : E \to \mathbb{R}_0^+$.

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*. A variant of the adjacency list is the *forward star representation* [1,2], in which 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 main drawback of the forward star 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 [14] 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 and the rebalance ranges are monitored by a complete binary tree on top of the array. More details can be found in [13].

3 The Packed-Memory Graph (PMG)

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 is shown in Fig. 1.

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.

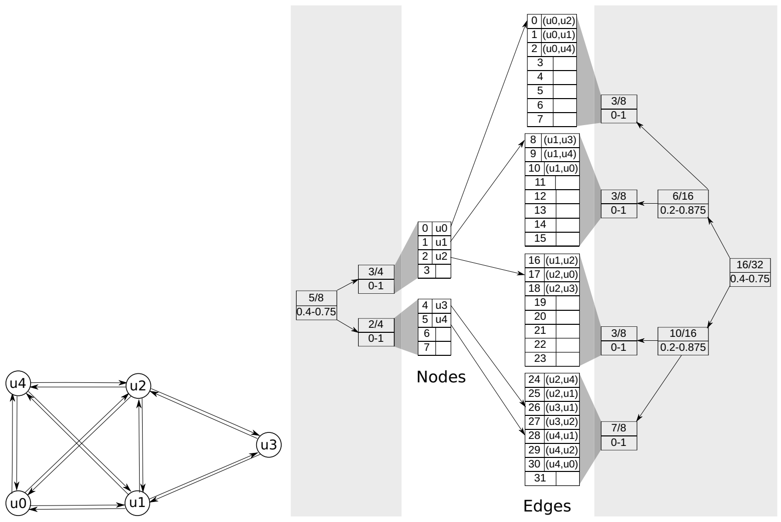


Fig. 1. Packed-memory Graph representation for the graph on the left

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 $E_{u_0} \cup E_{u_1} \cup ... \cup E_{u_{n-2}} \cup E_{u_{n-1}} = E$. The sets E_{u_i}, $0 \leq i < n$, are then stored consecutively in a unique range of cells of the packed-memory edge array in the same order as the one dictated by the internal node ordering in the packed-memory node array. 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 (for clarity, end pointers are not shown in Fig. 1). 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.

Operations. Our new graph structure supports the following operations, whose asymptotic bounds are given in Table 1 and their proofs can be found in [13].

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.

Table 1. Space, time and memory transfer complexities (the latter in the cache-oblivious model [9]). B: cache block size; Δ: maximum node degree (typically $O(1)$ in large-scale transportation networks).

	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 adj. 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 adj. edges)	not supported	not supported	$O(1+\frac{\Delta \log^2 m+\Delta \log^2 n}{B})$

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 [13] 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. 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.

Comparison with Other Dynamic Graph Structures. An adjacency list (ADJ) representation 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.

A dynamic forward star (DynFS) 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 (PMG) 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 practice). No update operation can implicitly alter the relative order of nodes and edges, therefore relative elements are always stored in memory as close as possible. Finally, the elements can be efficiently reordered 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 US road networks) in static and dynamic scenarios. For the former, we use the query performance of the static forward star (FS) 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 point-to-point shortest path queries and updates.

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 [7,8] and consist of the road networks of Italy, Germany, Western US and Central US. Edge weights represent travel distances. Experiments with travel times exhibited similar relative performance.

4.1 Algorithms

We implemented the full set of shortest path algorithms considered in [11]. These algorithms are based on the well known Dijkstra's algorithm, and its Bidirectional and A^* variants. Given a source node s and a target node t, the A^* (or goal-directed) variant modifies the priority of a node according to a monotone 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 t. An example for h_t is the Euclidean distance between two nodes. The A^* algorithm can also be used in a bidirectional manner, using a symmetric heuristic function h_s.

The key contribution in [11] is a highly effective heuristic function for the A^* algorithm using the triangle inequality theorem and precomputed distances to a few important nodes (*landmarks*), resulting in the so-called ALT algorithm. During a query, ALT computes the lower bounds in constant time using the precomputed shortest distances to landmarks with the triangle inequality. The efficiency of ALT depends on the initial selection of landmarks. In [11], 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 $d(s,v)+h_t(v)$ or $d(v,t)+h_s(v)$ 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: Bidirectional variant of Dijkstra's algorithm.
- AE: A^* search with Euclidean lower bounds.
- BEM: Bidirectional AE with the max heuristic function.
- BEA: Bidirectional AE with the average heuristic function.
- AL: Regular ALT algorithm.
- BLS: Symmetric bidirectional ALT.
- BLM: Consistent bidirectional ALT with the max heuristic function.
- BLA: Consistent bidirectional ALT 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 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 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 [11], 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 [11]. The query performance can be increased by adding more well-chosen landmarks.

Our experimental results on the road network of Central US are shown in Table 2 (results on the other road networks are reported in [13]). For each algorithm and each graph structure, we report the average running times in ms. We also report the space consumption for each graph structure. As expected, our experiments confirm the ones in [11] regarding the relative performance of the evaluated algorithms, with BLA being the best (see [13] for an explanation).

The major outcome of our experimental study is that there is a clear speedup 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

Table 2. Running times (ms) in the road network of Central US; Eff ≡ Efficieny

Central US	n = 14,081,816 m = 34,292,496					
	ADJ (3.054Gb)	**DynFS** (3.269Gb)	**PMG** (3.297Gb)	**FS** (3.054Gb)	Eff. (%)	Eff. (%) [11]
D	4,474.65	3,418.06	2,864.33	2,766.28	0.07	0.09
B	3,012.38	2,303.91	2,048.01	1,963.67	0.11	0.14
AE	2,055.29	1,522.24	1,414.35	1,370.49	0.17	0.10
BEM	1,881.84	1,448.47	1,274.99	1,236.90	0.19	–
BEA	1,853.45	1,429.97	1,237.39	1,208.97	0.20	0.14
AL	223.26	173.62	161.93	151.34	3.75	1.87
BLS	257.47	205.35	176.84	172.46	3.67	2.02
BLM	211.50	172.11	161.27	157.41	7.43	3.27
BLA	146.51	116.88	108.401	104.88	10.02	3.87

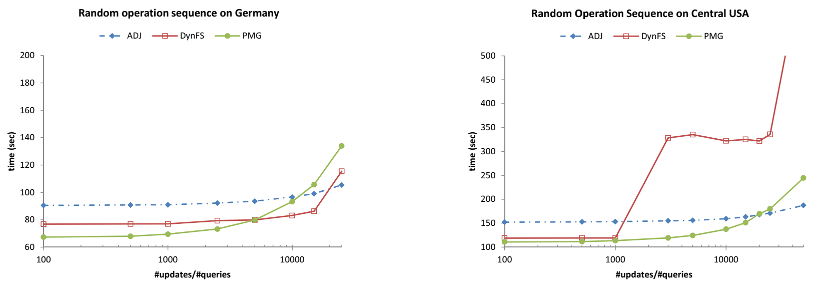
consideration that PMG is a dynamic structure. It is roughly 3% slower in the larger graph of Central US (denser PMG). Note that the denser the PMG is, the smaller size it has, the more it resembles an FS and the more it matches its query performance. Hence, if we take FS as a reference point for query 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. However, this is not our prime concern in this experimental study.

Dynamic Performance. We report results on random and realistic sequences of operations using the three structures ADJ, DynFS, PMG. Further results with dynamic operations and internal node relocations are reported in [13].

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 (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 2, where the horizontal axis (log-scale) represents the ratio of the number of updates and the number of



(a) (b)

Fig. 2. Running times on mixed sequences of operations consisting of 1000 queries and updates of varying length in $[10^5, 5 \times 10^7]$

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, PMG maintains a speed-up over ADJ and DynFS. In the road network of Germany (resp. Central US), the number of updates should be at least 5000 (resp. 20000) times more than the number of (BLA) queries in order for PMG to be inferior to DynFS, and at least 10000 more to be inferior to ADJ. 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 US road network.

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) [10]. 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 [5], 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 [5] of the CH code, since it provides the most efficient hierarchy. The results are shown in Table 3.

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

Table 3. Total time for processing 3,615,855 shortest path queries and 21,342,855 edge insertions, generated during CH preprocessing on Germany

Germany	Total Time (sec)
ADJ	32,194.4
DynFS	19,938.2
PMG	17,087.7

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%.

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., Orlin, J.B.: Network Flows: Theory, Algorithms and Applications. Prentice Hall, Englewood Cliffs (1993)
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. Contraction Hierarchies source code, `http://algo2.iti.kit.edu/routeplanning.php`
6. Delling, D., Goldberg, A.V., Nowatzyk, A., Werneck, R.F.: PHAST: Hardware-Accelerated Shortest Path Trees. In: IPDPS 2011. IEEE (2011)
7. 9th DIMACS Implementation Challenge – Shortest Paths, `http://www.dis.uniroma1.it/challenge9/index.shtml`
8. 10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering, `http://www.cc.gatech.edu/dimacs10/`
9. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th IEEE FOCS 1999, pp. 285–297 (1999)
10. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In: McGeoch, C.C. (ed.) WEA 2008. LNCS, vol. 5038, pp. 319–333. Springer, Heidelberg (2008)
11. Goldberg, A.V., Harrelson, C.: Computing the Shortest Path: A^* Search Meets Graph Theory. In: Proc. SODA, pp. 156–165 (2005)
12. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Reach for A^*: Shortest Path Algorithms with Preprocessing. DIMACS 74, 93–139 (2009)
13. Mali, G., Michail, P., Paraskevopoulos, A., Zaroliagis, C.: A New Dynamic Graph Structure for Large-Scale Transportation Networks, Technical Report eCOMPASS-TR-005, eCOMPASS Project (October 2012), `http://www.ecompass-project.eu/?q=node/135`
14. Schultes, D.: Route Planning in Road Networks. PhD Dissertation, University of Karlsruhe (2008)