Mobile Route Planning*

Peter Sanders, Dominik Schultes, and Christian Vetter

Universität Karlsruhe (TH), 76128 Karlsruhe, Germany, {sanders,schultes}@ira.uka.de, veaac@gmx.de

Abstract. We provide an implementation of an exact route planning algorithm on a mobile device that answers shortest-path queries in a road network of a whole continent instantaneously, i.e., with a delay of about 100 ms, which is virtually not observable for a human user. We exploit spatial and hierarchical locality properties to design a significantly compressed external-memory graph representation, which can be traversed efficiently and which occupies only a few hundred megabytes for road networks with up to 34 million nodes. Next to the accuracy, the computational speed, and the low space requirements, the simplicity of our approach is a fourth argument that suggests an application of our implementation in car navigation systems.

1 Introduction

In recent years, there has been a lot of work on route planning algorithms, particularly for road networks, aiming for fast query times and accurate results. The various real-world applications of such algorithms can be classified according to their respective platform into *server* applications (e.g., providing driving directions via the internet, optimising logistic processes) and *mobile* applications (in particular car navigation systems). On the one hand, many approaches have been evaluated successfully with respect to the server scenario—the fastest variant of transit-node routing [1] computes shortest-path distances in the Western European road network in less than two microseconds on average. On the other hand, there have been only few results on efficient implementations of route planning techniques on mobile devices like a PDA. In this paper, we want to close this gap.

The arising challenges are mainly due to the memory hierarchy of typical PDAs, which consists of a limited amount of fast main memory and a larger amount of comparatively slow flash memory, which has similar properties as a hard disk regarding read access. In order to obtain an efficient implementation, we have to arrange the data into blocks, respecting the locality of the data. Then, reading at a single blow a whole block that contains a high percentage of relevant data is much more efficient then reading single data items at random. Furthermore, compression techniques can be used to increase the amount of data that fits into a single block and, consequently, decrease the number of required block accesses.

Our Contributions

We present an efficient and practically useful implementation of a fast and exact route planning algorithm for road networks on a *mobile device*. For this purpose, we select contraction hierarchies [2] as our method of choice—we review the most relevant concepts in Section 2. In Section 3, we design an external-memory graph representation that takes advantage of the locality inherent in the data to compress the graph and to reduce the number of required I/O operations—which are the bottleneck of our application. The graph is divided into several blocks, each containing a subset of the nodes and the corresponding edges. We put particular efforts in exploiting the fact that the edges in one block only lead to nodes in a small subset of all blocks; many edges even lead to nodes in the same block.

* Partially supported by DFG grant SA 933/1-3.

By this means, our 'mobile' implementation achieves a considerable improvement compared to the original implementation of contraction hierarchies (CH) [2], which, in turn, is already considerably better than the bidirectional variant of Dijkstra's algorithm, as summarised in the following table, which refers to experiments on an European road network with about 18 million nodes.[1]

	only length		complete path	
	time [ms]	space [MB]	time [ms]	space [MB]
bidir. Dijkstra	298 209	408	298 209	408
CH [2]	394.9	350	3 025.6	560
CH mobile	59.4	140	96.6	275

All details on our experimental setting can be found in Section 4, followed by much more experimental results in Section 5. We conclude our paper in Section 6, where we also discuss related work, draw further comparisons, and outline possible future work.

2 Contraction Hierarchies

We decided to use contraction hierarchies [2] for our mobile implementation due to its simplicity, its outstanding low memory requirements, and its hierarchical properties that can be exploited to improve the locality of the accessed data. In this section, we give an account of the most important concepts of contraction hierarchies.

Preprocessing. In a first step, the nodes of a given graph $G = (V, E)$ (with $n := |V|$) are ordered by 'importance'[2]—we obtain a bijection $\ell : V \to \{1, 2, 3, \ldots, n\}$, where n represents the highest importance. In a second step, we first set $G' := G$ and then, while $V' \neq \emptyset$, we *contract* the node v with the lowest importance in V', i.e., we remove v and all incident edges from G'; since we want to preserve the lengths of all shortest paths containing a subpath of the form $\langle u, v, u' \rangle$, we add a so-called *shortcut* edge (u, u') (whose weight corresponds to the length of the path $\langle u, v, u' \rangle$) to E' whenever it is required. In a third step, we build the so-called *search graph* $G^* = (V, E^*)$. We define $\hat{E}$ to contain all edges from E and all shortcut edges that have been added at some point during the second step. Then, $E_\uparrow := \{(u, v, w) \in \hat{E} \mid \ell(u) < \ell(v)\}$, $E_\downarrow := \{(u, v, w) \in \hat{E} \mid \ell(u) > \ell(v)\}$, and $\overline{E_\downarrow} := \{(v, u, w) \mid (u, v, w) \in E_\downarrow\}$.[3] Finally, $E^* := E_\uparrow \cup \overline{E_\downarrow}$. Furthermore, we introduce a forward and a backward flag such that for any edge $e \in E^*$, $f(e) =$ true iff $e \in E_\uparrow$ and $b(e) =$ true iff $e \in \overline{E_\downarrow}$. Note that G^* is a directed acyclic graph.

Query. We perform two normal Dijkstra searches in G^*, one from the source using only edges where the forward flag is set and one from the target using only edges where the backward flag is set. Forward and backward search are interleaved, we keep track of a tentative shortest-path length and abort the forward/backward search process not until all keys in the respective priority queue are greater than the tentative shortest-path length. To further reduce the search space size, we employ the *stall-on-demand* technique [3, 4, 2].

[1] Running times refer to the query type 'cold' (Section 4.4), where the cache is cleared after each random query. Since the priority queue of Dijkstra's algorithm would not fit in the main memory, we ran Dijkstra only on the Dutch subnetwork of Europe and linearly extrapolated the obtained query times.

[2] [2] explains in detail how this can be done.

[3] Here, we write an edge from u to v with weight w as triple (u, v, w) since we want to treat two edges (u, v) with different weights as two discrete edges when defining the set E^*.

In order to determine not only the shortest-path *length*, but also a full description, the shortcut edges have to be unpacked to obtain the represented subpaths in the original graph. A simple recursive unpacking routine can be used provided that we have stored the middle node v of each shortcut (u, u') that represents the path $\langle u, v, u' \rangle$.

3 External-Memory Graph Representation

Locality. Reading data from external memory is the bottleneck of our application. To get a good performance, we want to arrange the data into blocks and access them blockwise. Obviously, the arrangement should be done in such a way that accessing a single data item from one block typically implies that a lot of data items in the same block have to be accessed in the near future. In other words, we have to exploit locality properties of the data.

The node order of the real-world road networks that we have obtained already respects *spatial* locality, i.e., the nodes are ordered somehow by spatial proximity. However, we can do better. We consider the reverse search graph $\overline{G^*} = (V, \overline{E^*})$, where $\overline{E^*} := \{(v, u, w) \mid (u, v, w) \in E^*\}$, which is an acyclic graph (as G^*), and compute a simple topological order, using a modified depth-first search (DFS) where a node v is visited not until all nodes u with $(u, v) \in \overline{E^*}$ have been visited. We will see in Appendix A.2 that this order already greatly improves the locality and is almost as good as a more sophisticated order that can be obtained using a much more expensive technique. One reason for the success of this method is presumably the small depth of the search graph.

We can further stress the *hierarchical* locality: nodes of a similar high importance should be close to each other since – due to the nature of the query algorithm – they are likely to be visited in tandem. For a fixed *next-layer fraction* f, we divide the nodes into two groups: the first group contains the $(1 - f) \cdot |V|$ nodes of smaller importance, the second group the $f \cdot |V|$ nodes of higher importance. Within each group, we keep the topological order obtained by our modified DFS. We recurse in the second group until all nodes fit into a single block. This *hierarchical reordering* step is a slightly generalised version of a technique used in [5]. It is important to note that the resulting order is still a topological order.

Note that a good node order has not only the obvious advantage that a loaded block contains a lot of relevant data, but also can be exploited to compress the data effectively: in particular, the difference of target and source ID of an edge is typically quite small; often the target node even belongs to the same block.

Main Data Structure. The starting point for our compact graph data structure is an *adjacency array* representation: All edges (u, v) are kept in a single array, grouped by the source node u. Each edge stores only the target v and its weight. In addition, there is a node array that stores for each node u the index of the first edge (u, v) in the edge array. The end of the edge group of node u is implicitly given by the start of the edge group of u's successor in the node array. We want to divide this graph data structure into blocks. In order to decrease the number of required block accesses, we decided not to store node and edge data separately, but to put node and the associated edge data in a common block—as illustrated in Fig. 1.

When encoding the target v of an edge (u, v), we want to exploit the existing locality, i.e., in many cases the difference of the IDs of u and v is quite small and, in particular, u and v often belong to the same block. Therefore, we distinguish between *internal* and *external* edges: internal edges lead to a node within the same block, external edges lead to a node in a different block. We use a flag to mark whether an edge is an internal or an external one. In case of an internal edge, it is sufficient to just store the node index within the same block, which

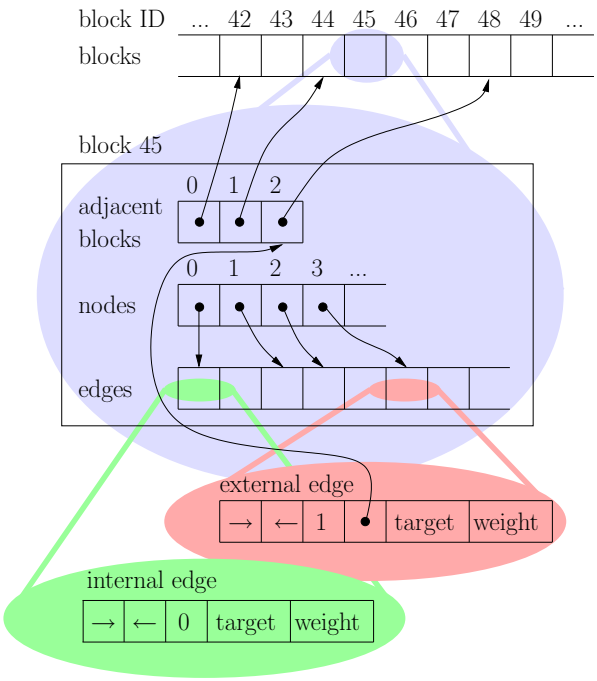

Fig. 1. External-memory graph data structure (without path unpacking information). Each edge stores three flags: a forward flag (→), a backward flag (←), and a flag that indicates whether it is an external edge leading to a node in a different block.

requires only a few bits. In case of an external edge, we need the block ID of the target and the node index within the designated block. We introduce an additional indirection to reduce the number of bits needed to encode the ID of the adjacent block: It can be expected that the number of blocks adjacent to a given block B is rather small, i.e., there are only a few different blocks that contain all the nodes that are adjacent to nodes in B. Thus, it pays to explicitly store the IDs of all adjacent blocks in an array in B. Then, an external edge need not store the full block ID, but it is sufficient to just store the comparatively small block index within the adjacent-blocks array. Again, this is illustrated in Fig. 1.

Building the Graph Representation. We pursue the following goals: the graph data structure should occupy as little memory as possible, and accessing the data should be fast. We make the following design choices: each block has the same constant size; each block contains a subset of the nodes and all incident edges[4]; the node range of each block is consecutive and maximal[5]; all three 'logical' arrays (adjacent blocks, nodes, edges) are stored in a single byte array one after the other, the starting index of each logical array is stored in the header of the block; within each block, we use the minimal number of bits to store the respective attributes[6].

In general, building the blocks is not trivial due to a cyclic dependency: On the one hand, the distribution of the nodes into blocks depends on the required memory for each edge—in particular, an internal edge typically occupies less memory than an external edge. In other words, a block can accommodate more internal than external edges. On the other hand, the distinction whether an edge is internal or external depends on the distribution of the nodes:

[4] We currently do not deal with the very exceptional case that the degree of a single node is so large that its edges do not fit in one block.
[5] Since nodes have got different degrees, this implies that different blocks can contain different numbers of nodes.
[6] For example, if a block has 42 different adjacent blocks, then each external edge (u, v) in this block uses 6 bits to address the adjacent block that contains v.

if the target node of an edge fits into the same block, we have an internal edge; otherwise, we have an external edge.

Fortunately, we can exploit the fact that we sorted the nodes topologically. When we process the nodes from the most important one to the least important one, all edges (u, v) point to nodes v that have already been processed. This implies that we already know whether (u, v) is an internal or external edge and, in case of an external edge, we also know the number of nodes in the corresponding block B so that we can choose the minimal number of bits required to encode the index of node v within the block B. This way, we can easily calculate the memory requirements of the current edge. If all edges of the current node u fit into the current block, the node and its incident edges are added. Otherwise, a new block is started. Note that when we consider to add another node and its edges, we have to account not only for the memory directly used by these additional objects, but also for a potential memory increase of the other nodes and edges in the same block: for example, whenever the number of edges in the block exceeds the next power of two, all nodes in the block need an additional bit to store the index of the first outgoing edge.

Since most edge weights in our real-world road networks are rather small and only comparatively few edges (in particular, some ferries and shortcuts that leave very important nodes) are quite long, we use one bit to distinguish between a long and a short edge; depending on the state of this bit, we use more or less bits to store the weight.

Storing the Graph Representation. The blocks representing the graph are stored in external memory. In main memory, we manage a cache that can hold a subset of the blocks. We employ a simple least-recently used (LRU) strategy. In the external-memory graph data structure, a node u is identified by its block ID $B(u)$ and the node index $i(u)$ within the block. We need a mapping from the node ID u used in the original graph to the tuple $(B(u), i(u))$. Such a mapping is realised in a simple array, stored in external memory.

We want to access the external memory *read-only* in order to improve the overall performance and in order to preserve the flash memory, which can get unusable after too many write operations. Therefore, we clearly separate the read-only graph data structures from some volatile data structures, in particular the forward and the backward priority queue. We use a hash map to manage pointers from reached nodes to the corresponding entries in the priority queues. Since the search spaces of contraction hierarchies are so small (a few hundred nodes out of several million nodes in the graph), it is no problem to keep these data structures in main memory. Note that in [6], a similar distinction between read-only and volatile data structures has been used.

Path Unpacking Data Structures. The above data structures are sufficient to support queries that determine the shortest-path *length*. On a mobile device, however, we usually are also interested in a complete description of the shortest path (e.g., in order to generate driving directions). First of all, since we have changed the node order, we need to store for each node its original ID so that we can perform the reverse mapping. Furthermore, we need the functionality to unpack shortcut edges. To support a simple recursive unpacking routine, we store the ID of the middle node of each shortcut (as already mentioned in Section 2). We distinguish between internal and external shortcuts (u, u'), where the middle node v belongs to the same block as u or not. For an internal shortcut, the middle node can be stored as an index within the block, for an external shortcut, we have to specify the block $B(v)$ and the index within $B(v)$.

To accelerate the path unpacking, we can explicitly store pre-unpacked paths as sequences of original node IDs (cp. [7]). Looking up the edges (v, u) and (v, u') in case of an *external* shortcut

(u, u') with middle node v might require an expensive additional block read. Therefore, it is reasonable to completely pre-unpack all external shortcuts and to store the corresponding node sequences in some additional data blocks. Instead of the middle node, we store the starting index within these additional data blocks. We can exploit the fact that an external shortcut can contain other external shortcuts. We do not have to store these contained shortcuts explicitly, but it is sufficient to just note the correct starting position and a direction flag since contained shortcuts might be filed in the reverse direction. To actually implement this idea, which is new compared to [7], we use a top-down approach. We consider external shortcuts in a descending order of importance. A shortcut is unpacked only if it is not contained in an already unpacked shortcut.

4 Experimental Setting

4.1 Implementation

The implementation of both the preprocessing routines and the query algorithm has been done in C++ using the Standard Template Library. We distinguish between three different ways to access the external memory of the mobile device, as illustrated in Fig. 2. The *unbuffered* access

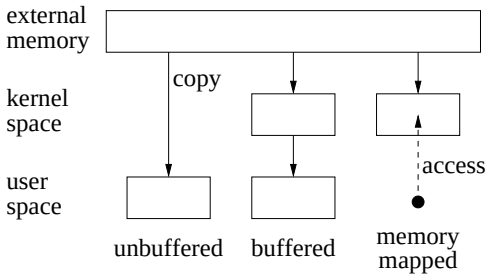

Fig. 2. Different ways to access the external memory.

allows to copy a data block directly from the external memory to the user space where it can be accessed by the application. In case of the *buffered* access, an additional copy of the data block is kept in the kernel space. When accessing the same block at a later point in time, the block can be re-read from the kernel space instead of performing an expensive read operation from the external memory—provided that the block is still available in the kernel space. The third alternative is to use *memory mapped files*: when reading data that is not available in the kernel space, the data is loaded into the kernel space; the application accesses the data in the kernel space without making an explicit copy in the user space.

We would rather like to use the unbuffered variant to access the data blocks that contain the graph data structure because it is the fastest option (cp. Appendix A.1). However, a bug in the kernel of the installed operating system forces us to use the buffered access method. Since we want to have explicit control over the caching mechanism, we still use our own cache in the user space and we instruct the operating system to clear its cache in the kernel space after each query. Note that the time to clear the system cache is not included in our figures since this is not considered to be a part of the actual task; furthermore, this step would not be required if the unbuffered access worked.

In addition to the actual graph data structure, we have to access the mapping from original node IDs to the tuple consisting of block ID and index within the block. This is done using memory mapped files. Note that only two accesses are needed per query, one to lookup the source node and one for the target.

4.2 Environment

Experiments have been done on a Nokia N800 Internet Tablet equipped with 128 MB of RAM and a Texas Instruments OMAP 2420 microprocessor, which features an ARM11 processor running at 330 MHz. We use a SanDisk Extreme III SD flash memory card with a capacity of 2 GB; the manufacturer states a sequential reading speed of 20 MB/s. The operating system is the Linux-based Maemo 3.2 in the form of Internet Tablet OS 2007. The program was compiled by the GNU C++ compiler 4.2.1 using optimisation level 3.

Preprocessing has been done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GB main memory and 2 × 1 MB L2 cache, running SuSE Linux 10.3 (kernel 2.6.22). The program was compiled by the GNU C++ compiler 4.2.1 using optimisation level 3.

4.3 Test Instances

Most experiments have been done on a road network of Europe[7], which has been made available for scientific use by the company PTV AG. For each edge, its length and one out of 13 road categories (e.g., motorway, national road, regional road, urban street) is provided.

In addition, we perform some experiments on a publicly available version of the US road network (without Alaska and Hawaii) that was obtained from the DIMACS Challenge homepage [8] and on a new version[8] of the European road network ("New Europe") that was provided for scientific use by the company ORTEC. In all cases, we use a travel time metric.

Our starting point are precomputed contraction hierarchies [2]. Preprocessing takes 31 min, 32 min, and 58 min for Europe, USA, and New Europe, respectively.

4.4 Query Types

We distinguish between four different query types:

1. *'cold'*: Perform 1 000 random queries; after each query, clear the cache[9]. This way, we can determine the time that is needed for the first query when the program is started since in this scenario the cache is empty.
2. *'warm'*: Perform 1 000 random queries to warm up the cache; then, perform a different set of 1 000 random queries without clearing the cache; determine the average time only of the latter 1 000 queries. This way, we can determine the average query time for the scenario that the device has been in use for a while.
3. *'recompute'*: Select 100 random target nodes $t_1, \ldots, t_{100}$ and for each target t_i, 101 random source nodes $s_{i,0}, \ldots, s_{i,100}$. For each target t_i and each $j, 1 \leq j \leq 100$, perform one query from $s_{i,0}$ to t_i without measuring the running time and one query from $s_{i,j}$ to t_i performing time measurements, and clear the cache. This way, we can determine the time needed to recompute the shortest path to the same target in case that only the source node changes—which can happen if the driver does not follow the driving directions.
4. *'w/o I/O'*: Select 100 random source-target pairs. For each pair, repeat the same query 101 times; ignore the first iteration when measuring the running time. This way, we obtain a benchmark for the actual processing speed of the device when no I/O operations are performed.

[7] Austria, Belgium, Denmark, France, Germany, Italy, Luxembourg, the Netherlands, Norway, Portugal, Spain, Sweden, Switzerland, and the UK

[8] In addition to the old version, the Czech Republic, Finland, Hungary, Ireland, Poland, and Slovakia.

[9] In Section 4.1, we have already mentioned that we always clear the *system* cache after each query. Note that in this section, we are talking about our *own* cache.

For practical scenarios, the first and the third query type are most relevant; for comparisons to related work, the second query type is interesting.[10]

5 Experimental Results

In most practical scenarios, we want to have the opportunity to not only determine the shortest-path length, but also (parts of) the actual shortest path. However, a full description of the computed path is not always required. Therefore, unless otherwise stated, our experiments refer to the case that the path-unpacking data structures exist, but are not used. We deal with other scenarios as well, in particular in Section 5.5. Note that the query times always include the time needed to map the original source and target IDs to the corresponding block IDs and node indices, while figures on the memory consumption do not include the space needed for the mapping.

5.1 Different Block Sizes

On the one hand—as Fig. 5 in Appendix A.1 indicates—choosing a larger block size usually increases the reading speed. On the other hand, loading larger blocks often means that more irrelevant data is read. Hence, we expect that the best choice is a medium-sized block. This is confirmed by Fig. 3 (a), where we used different block sizes, a next-layer fraction of 1/16, and the first query type ('cold'). The total memory consumption increases for increasing block sizes since the range of the various values (like adjacent-block indices) increases so that more bits are needed. If the block size gets too small, we observe more external shortcuts so that the path-unpacking data structures occupy more space. For all further experiments, we use a block size of 4 KB.

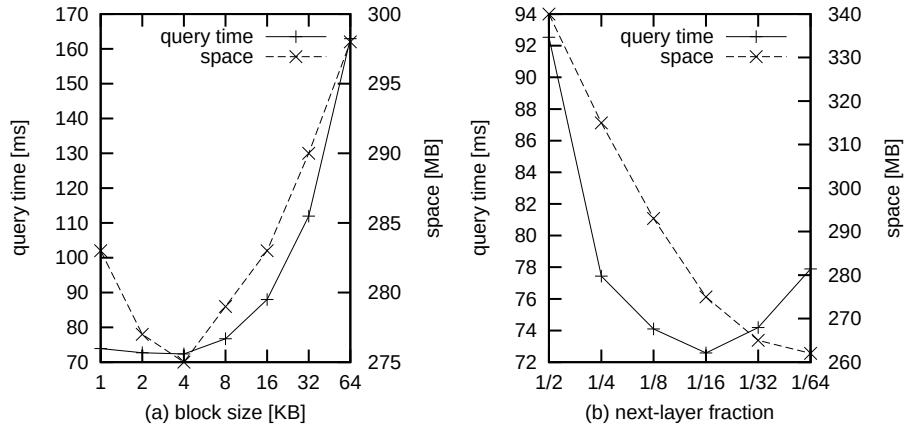

Fig. 3. Average query time and total space consumption depending on the chosen block size (a) and the next-layer fraction f (b).

[10] It might be the case that in practical scenarios the second type is even more relevant than the first one: if there is some idle time between switching on the device and starting the query algorithm, data blocks that contain the more important nodes (which are used in most queries) could be prefetched so that the observed runtime behaviour would be closer to the 'warm' scenario than to the 'cold' scenario.

5.2 Different Hierarchical Localities

In Fig. 3 (b), we investigate the effect of choosing different next-layer fractions, using the first query type ('cold'). The best compromise between spatial and hierarchical locality is obtained for $f = 1/16$, which we use as default value for all further experiments. The memory consumption decreases for a decreasing value of f due to a smaller number of external edges/shortcuts. Note that omitting the hierarchical reordering yields a query time of 96.8 ms and a space consumption of 264 MB.

5.3 Different Cache Sizes

In one test series (Fig. 4), we applied the second query type ('warm') to different cache sizes. We obtained the expected behaviour that the average query time decreases with an increasing cache size. Interestingly, even very small cache sizes are sufficient to arrive at quite good query times. Note that, on average, a cache size of 160 KB is already sufficient to accommodate the blocks needed for a single query in the European road network. We observe virtually no improvement when using a cache size of 64 MB instead of 32 MB. This is due to the fact that the 2 000 random queries that we perform (cp. Section 4.4) do not completely fill the 32 MB-cache. Nevertheless, for the remaining experiments, we just use the maximum cache size of 64 MB.

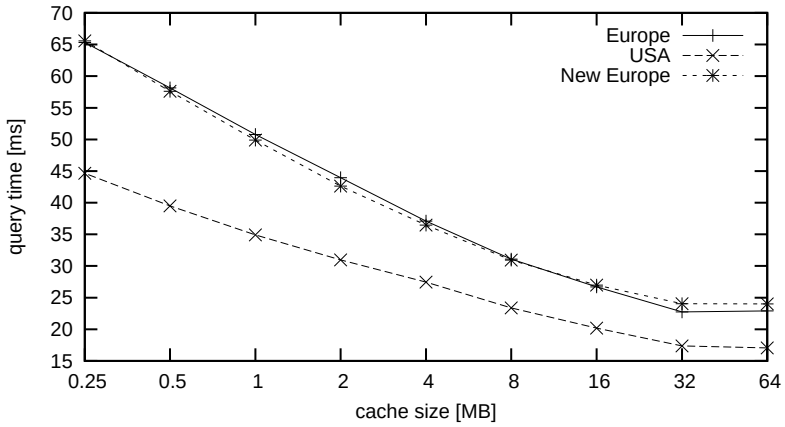

Fig. 4. Query performance for the second query type ('warm'), depending on the cache size.

5.4 Main Results

Table 1 gives an overview of the external-memory graph representation. Building the blocks is very fast and can be done in about 2–4 minutes. Although the given memory consumption already covers everything that is needed to obtain very fast query times (including path unpacking), we need 30% *less* space than the original graph would occupy in a standard adjacency-array representation in case of Europe.

The results for the four query types introduced in Section 4.4 are represented in Tab. 2. On average, a random query has to access 39 blocks in case of the European road network. When the cache has been warmed-up, most blocks (in particular the ones that contain very important nodes) reside in the cache so that on average less than four blocks have to be fetched from external memory. This yields a very good query time of 23 ms. Recomputing the optimal

Table 1. Building the graph representation. We give the number of nodes, the number of edges in the original graph and in the search graph, the number of graph-data blocks (without counting the blocks that contain pre-unpacked paths), the average number of adjacent blocks per block, the numbers of internal edges, internal shortcuts and external shortcuts as percentage of the total number of edges, the time needed to pre-unpack the external shortcuts and to build the external-memory graph representation (provided that the search graph is already given), and the total memory consumption including pre-unpacked paths.

	$\lvert V\rvert$ [$\times 10^6$]	$\lvert E\rvert$ [$\times 10^6$]	$\lvert E^*\rvert$ [$\times 10^6$]	#blocks	#adj. blocks	int. edges	int. shcs.	ext. shcs.	time [s]	space [MB]
Europe	18.0	42.2	36.9	52 107	9.1	70.6%	32.2%	7.7%	123	275
USA	23.9	57.7	49.4	80 099	8.4	69.2%	33.7%	8.0%	186	400
New Europe	33.7	75.1	65.7	103 371	8.3	70.3%	32.7%	7.5%	210	548

path using the same target, but a different source node can be done in 34 ms. As expected, the bottleneck of our application are the accesses to the external memory: if all blocks had been preloaded, a shortest-path computation would take only about 8 ms instead of the 72 ms that include the I/O operations.

Table 2. Query performance for four different query types.

		cold		warm		recompute		w/o I/O
	settled nodes	blocks read	time [ms]	blocks read	time [ms]	blocks read	time [ms]	time [ms]
Europe	280	39.2	72.4	3.6	22.9	7.9	34.1	8.4
USA	223	30.1	56.5	4.4	17.1	6.1	28.0	6.1
New Europe	351	44.5	84.2	4.6	24.0	8.5	39.9	12.3

5.5 Path Unpacking

In Tab. 3, we compare five different variants of path (not-)unpacking, using the first query type ('cold') in each case. First (a), we store no path data at all. This makes the query very fast since more nodes fit into a single block. However, with this variant, we can only compute the shortest-path length. For all other variants, we also store the middle nodes of the shortcuts in the data blocks. This slows down the query even if we do not use the additional data (b). After having computed the shortest-path length, getting the very first edge of the path (which is useful to generate the very first driving direction) is almost for free (c). Computing the complete path takes considerably longer if we do not use pre-unpacked path data (d). Pre-unpacked paths (e) somewhat increase the memory requirements, but greatly improve the running times. Note that almost half of the pre-unpacked paths are contained in other pre-unpacked paths so that they require no additional space. Further experimental results on path unpacking can be found in Appendix A.3.

Table 3. Comparison between different variants of path unpacking.

	Europe		USA		New Europe	
	time [ms]	space [MB]	time [ms]	space [MB]	time [ms]	space [MB]
(a) no path data	59.4	140	47.2	213	66.8	257
(b) only length	72.4	203	56.5	312	84.2	403
(c) first edge	72.5	203	56.7	312	84.4	403
(d) complete path	458.3	203	932.8	312	698.9	403
(e) compl. path (fast)	96.6	275	90.6	400	117.8	548

6 Discussion

As far as we know, we provide the first implementation of an *exact* route planning algorithm on a *mobile device* that answers queries in a road network of a whole continent *instantaneously*, i.e., with a delay that is virtually not observable for a human user. Furthermore, our graph representation is comparatively *small* (only a few hundred megabytes) and the employed query algorithm is quite *simple*, which suggests an application of our implementation in car navigation systems.

6.1 Related Work

There is an abundance of shortest-path speedup techniques, in particular for road networks. For a broad overview, we refer to [9, 4]. In general, we can distinguish between goal-directed and hierarchical approaches.

Goal-Directed Approaches (e.g., [10–13, 6]) direct the search towards the target t by preferring edges that shorten the distance to t and by excluding edges that cannot possibly belong to a shortest path to t—such decisions are usually made by relying on preprocessed data. For a purely goal-directed approach, it is difficult to get an efficient external-memory implementation since no hierarchical locality (cp. Section 3) can be exploited. In spite of the large memory requirements, Goldberg and Werneck [6] successfully implemented the ALT algorithm on a Pocket PC. Their largest road network (North America, 29 883 886 nodes) occupies 3 735 MB and a random query takes 329 s. Using similar hardware[11], a slightly larger graph ("New Europe"), and a slightly smaller cache size (8 MB instead of 10 MB), our graph representation requires only 548 MB (about 1/7 of the space needed by [6]) and our random queries (including path unpacking) take 42 ms (more than 7 500 times faster) when our cache has been warmed up and 118 ms (more than 2 500 times faster) when our cache is initially empty.

Hierarchical Approaches (e.g., [14, 5, 15, 16, 3, 4]) exploit the hierarchical structure of the given network. In a preprocessing step, a hierarchical representation is extracted, which can be used to accelerate all subsequent queries. Although hierarchical approaches usually can take advantage of the hierarchical locality, not all of them are equally suitable for an external-memory implementation, in particular due to sometimes large memory requirements. The RE algorithm [14, 5] has been implemented on a mobile device, yielding query times of "a few seconds including path computation and search animation" and requiring "2–3 GB for USA/Europe" [17].

Commercial Systems. We made a few experiments with a commercial car navigation system, a recent TomTom One XL[12], computing routes from Karlsruhe to 13 different European capital cities. We observe an average query time of 59 s to compute the route, not including the time needed to compose the driving directions. Obviously, this is far from being a system that provides instantaneous responses.[13] Furthermore, to the best of our knowledge, current commercial systems do *not* compute exact routes.

[11] We use a more recent version of the ARM architecture, but with a slightly slower clock rate (330 MHz instead of 400 MHz); in [6], random reads of 512-byte blocks from flash memory can be done with a speed of 366 KB/s, compared to 326 KB/s on our device.

[12] AK9SQ CSBUS, ARM9 processor clocked at 266 MHz, application version 6.593, OS version 1731, 29 MB RAM, road network of Western Europe version 675.1409

[13] Note that such a commercial product is slowed down due to various reasons (e.g., some time is spent to refresh the display in order to update a progress bar), which are neglected in our test environment. Therefore, a direct quantitative comparison is not possible.

More Related Work. Compact graph representations have been studied earlier. In [18], the nodes of the graph are rearranged according to the in-order of a separator tree that results from recursively removing edges to separate the graph into components. By this means, the difference between the IDs of adjacent nodes gets small so that applying suitable encoding schemes yields a better compression rate than we achieve. However, the study in [18] does not take into account additional edge attributes like the edge weight and, more importantly, it does not refer to the external-memory model, which is crucial for our application.

There has been considerable theoretical work on external-memory graph representations and external-memory shortest paths (e.g. [19–22]). Indeed, although road networks (let alone our hierarchical networks) are not planar, the basic ideas in [20] lead to a similar approach to blocking as we use it. Also, the redundant representation proposed in [20], which adds a neighbourhood of all nodes to a block, might be an interesting approach to further refinements. However, the worst case bounds obtained are usually quite pessimistic and there are only few implementations: the closest one we are aware of [23] only works for undirected graphs with unit edge weights and does not exploit the kind of locality properties we are dealing with.

6.2 Future Work

Increasing the compression rate seems possible, in particular by using more sophisticated techniques, e.g. from [18]. However, we have to bear the decoding speed in mind: it might be counterproductive to use techniques that are very complicated.

One particularly relevant scenario is the case that the driver deviates from the computed route (third query type in Section 4.4). The recomputation could be accelerated by explicitly storing and reusing the backward search space.

Acknowledgements

We would like to thank Robert Geisberger for providing precomputed contraction hierarchies [2] for various networks.

References

1. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In: 7th Workshop on Experimental Algorithms (WEA). Volume 5038 of LNCS., Springer (2008) to appear.
2. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: 7th Workshop on Experimental Algorithms (WEA). Volume 5038 of LNCS., Springer (2008) to appear.
3. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: 6th Workshop on Experimental Algorithms (WEA). Volume 4525 of LNCS., Springer (2007) 66–79
4. Schultes, D.: Route Planning in Road Networks. PhD thesis, Universität Karlsruhe (TH) (2008)
5. Goldberg, A.V., Kaplan, H., Werneck, R.F.: Better landmarks within reach. In: 6th Workshop on Experimental Algorithms (WEA). Volume 4525 of LNCS., Springer (2007) 38–51
6. Goldberg, A.V., Werneck, R.F.: Computing point-to-point shortest paths from external memory. In: Workshop on Algorithm Engineering and Experiments (ALENEX). (2005) 26–40
7. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge [8]. (2006)
8. 9th DIMACS Implementation Challenge: Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/` (2006)
9. Sanders, P., Schultes, D.: Engineering fast route planning algorithms. In: 6th Workshop on Experimental Algorithms (WEA). Volume 4525 of LNCS., Springer (2007) 23–36
10. Lauther, U.: An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In: Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung. Volume 22., IfGI prints, Institut für Geoinformatik, Münster (2004) 219–230
11. Köhler, E., Möhring, R.H., Schilling, H.: Fast point-to-point shortest path computations with arc-flags. In: 9th DIMACS Implementation Challenge [8]. (2006)

12. Hilger, M.: Accelerating point-to-point shortest path computations in large scale networks. Diploma Thesis, Technische Universität Berlin (2007)
13. Goldberg, A.V., Harrelson, C.: Computing the shortest path: A^* meets graph theory. In: 16th ACM-SIAM Symposium on Discrete Algorithms. (2005) 156–165
14. Gutman, R.: Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In: Workshop on Algorithm Engineering and Experiments (ALENEX). (2004) 100–111
15. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: 14th European Symposium on Algorithms (ESA). Volume 4168 of LNCS., Springer (2006) 804–816
16. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks with transit nodes. Science **316**(5824) (2007) 566
17. Goldberg, A.: personal communication (2008)
18. Blandford, D.K., Blelloch, G.E., Kash, I.A.: An experimental analysis of a compact graph representation. In: Workshop on Algorithm Engineering and Experiments (ALENEX). (2004)
19. Nodine, M.H., Goodrich, M.T., Vitter, J.S.: Blocking for external graph searching. Algorithmica **16**(2) (1996) 181–214
20. Agarwal, P.K., Arge, L.A., Murali, T.M., Varadarajan, K., Vitter, J.: I/O-efficient algorithms for contour-line extraction and planar graph blocking. In: 9th ACM-SIAM Symposium on Discrete Algorithms. (1998) 117–126
21. Hutchinson, D., Maheshwari, A., Zeh, N.: An external memory data structure for shortest path queries. In: 5th ACM-SIAM Computing and Combinatorics Conference. Volume 1627 of LNCS., Springer (1999) 51–60
22. Meyer, U., Zeh, N.: I/O-efficient undirected shortest paths with unbounded edge lengths. In: 14th European Symposium on Algorithms (ESA). Volume 4168 of LNCS., Springer (2006) 540–551
23. Ajwani, D., Dementiev, R., Meyer, U.: A computational study of external-memory BFS algorithms. In: ACM-SIAM Symposium on Discrete Algorithms. (2007) 601–610
24. Ajwani, D., Malinger, I., Meyer, U., Toledo, S.: Characterizing the performance of flash memory storage devices and its impact on algorithm design. In: 7th Workshop on Experimental Algorithms (WEA). Volume 5038 of LNCS., Springer (2008) to appear.
25. Karypis Lab: METIS - Family of Multilevel Partitioning Algorithms (2008)
26. R Development Core Team: R: A Language and Environment for Statistical Computing. `http://www.r-project.org` (2004)

A Further Experiments

A.1 Accessing Flash Memory

Figure 5 shows for the three different access methods introduced in Section 4.1 the reading speed from flash memory depending on the chosen block size. The unbuffered access method is clearly the fastest; up to 64 KB block size, its speed considerably increases with an increasing block size, then we observe a saturation. The kink at 256 KB in case of the buffered access is presumably due to some prefetching strategy of the operation system, which seemingly loads more data than necessary for certain block sizes. Even for sequential reads, we stay below a

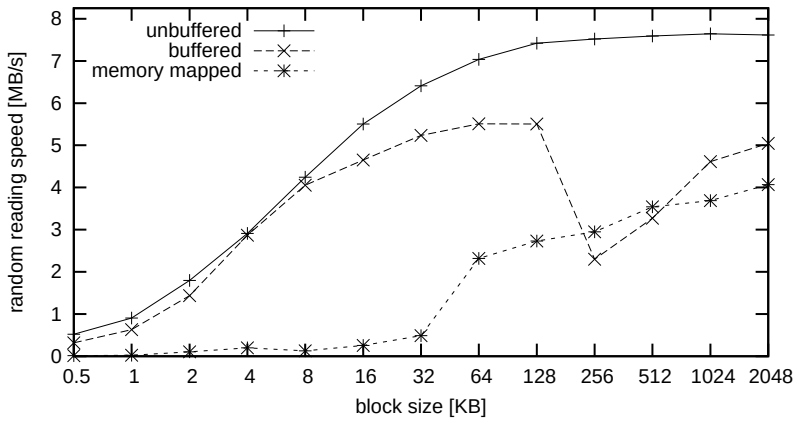

Fig. 5. Reading from flash memory.

speed of 8 MB/s and do not get close to the 20 MB/s as promised by the manufacturer of the flash memory card. Probably, this is due to limitations of our mobile device.

Note that in [24], Ajwani et al. present an extensive study on the performance of flash memory storage devices. For random reads, their results qualitatively correspond to the behaviour we found in the unbuffered case.

A.2 Node Ordering

As already mentioned in Section 3, a good node order has two advantages: first, due to a better locality, we have to access less blocks during a query; second, the effectiveness of our compression scheme improves (which, in turn, has a positive effect on the query performance as well since more data fits into a single block). In order to be able to investigate both benefits separately, we performed a test series where we considered different node orderings: in each case, we put the same constant number of edges in each block. This way, only the first advantage is effective. Table 4 summarises the results of this test series. We observe that our simple topological order combined with the hierarchical reordering (which we use in our experiments in Section 5) provides a significant improvement compared to the original order. A topological order that is applied on top of an order based on graph partitioning using METIS [25] provides hardly any better results so that it does not pay to invest the additional effort.

Table 4. Average number of accessed blocks for random queries considering three different node orders. Numbers in parentheses refer to the case that we also apply the hierarchical reordering step. For 'New Europe', METIS crashed so that we cannot provide results.

	original	topological	METIS
Europe	289 (106)	118 (102)	116 (102)
USA	263 (103)	102 (83)	102 (83)
New Europe	405 (165)	155 (140)	– (–)

A.3 Local Queries

For the first query type ('cold') and the European road network, Fig. 6 shows the query time distribution (including path unpacking) as a function of *Dijkstra rank*—the number of iterations Dijkstra's algorithm would need to solve this instance. The distributions are represented as box-and-whisker plots [26]: each box spreads from the lower to the upper quartile and contains the median, the whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually. We observe that for short- and medium-ranged queries, which are likely to dominate real-world application scenarios, we achieve very good query times, while even long-range queries are usually answered in less than 150 ms.

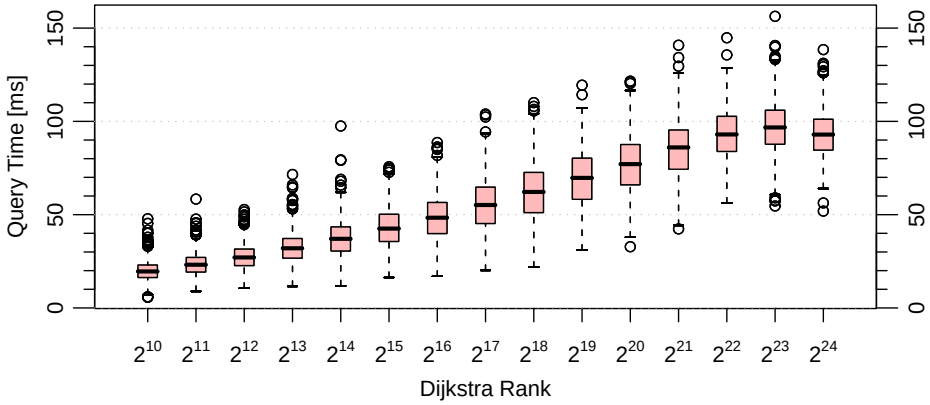

Fig. 6. Local queries.