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 distance queries in a road network of a whole continent instantaneously, i.e., with a delay of 20–200 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 highway-node routing [2, 3] 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. We give an algorithm that efficiently builds the external-memory graph data structure. Then, applying the query algorithm of highway-node routing is rather straightforward.

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

After stating our experimental setting in Section 4, we give the results of our extensive experimental study in Section 5. We demonstrate that we can smoothly handle huge road networks with up to 33.7 million nodes and 75.1 million edges. Our data structures occupy only about two third of the space an uncompressed representation of the original graph would need; for the Western European road network, our compression rate is even close to a factor of two. Query times are below 200 ms even in the case that the program is run for the first time so that the cache is empty. Recomputing an optimal path in case that the driver has deviated from the proposed route can be done in about 40 ms when considering our largest network. We conclude our paper in Section 6, where we also discuss related work, draw comparisons, and outline possible future work.

2 Highway-Node Routing

We decided to use highway-node routing 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 highway-node routing, while we refer to [2, 3] for details.

Multi-Level Overlay Graph. An overlay graph of a given graph consists of a subset of the nodes and an edge set that has the property that shortest path distances are preserved. More formally, for a given graph G_ℓ and a node set $V_{\ell+1}$, the graph $G_{\ell+1} = (V_{\ell+1}, E_{\ell+1})$ is an *overlay graph* of G_ℓ iff for all $(u, v) \in V_{\ell+1} \times V_{\ell+1}$, we have $d_{\ell+1}(u, v) = d_\ell(u, v)$, where $d_\ell(u, v)$ denotes the distance from u to v in G_ℓ.

The overlay graph definition can be applied iteratively to define a multi-level hierarchy. For given *highway-node sets* $V =: V_0 \supseteq V_1 \supseteq \ldots \supseteq V_L$, we define the *multi-level overlay graph* $\mathcal{G} = (G_0, G_1, \ldots, G_L)$ in the following way: $G_0 := G$ and for each $\ell \geq 0$, $G_{\ell+1}$ is the overlay graph of G_ℓ. The *level* $\ell(u)$ of a node $u \in V$ is $\max\{\ell \mid u \in V_\ell\}$.

Node Classification. Originally [2], a node classification $V \to \{0, \ldots, L\}$ provided by the highway hierarchies approach [4] was used to define the highway-node sets leading to a multi-level overlay graph with about ten levels. In a more recent variant [5], a node classification is obtained by iteratively contracting the 'least important' node, yielding a hierarchy with $|V|$ levels.

Query. We define the *forward search graph* $\overrightarrow{\mathcal{G}} := (V, \{(u, v) \mid (u, v) \in E_{\ell(u)}\})$ and, analogously, the *backward search graph* $\overleftarrow{\mathcal{G}} := (V, \{(u, v) \mid (v, u) \in E_{\ell(u)}\})$. We perform two *normal* Dijkstra searches in $\overrightarrow{\mathcal{G}}$ and in $\overleftarrow{\mathcal{G}}$. 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. Note that we could not guarantee the correctness if we aborted the entire query as soon as both search scopes meet for the first time. To further reduce the search space size, we employ the *stall-on-demand* technique introduced in [2].

3 External-Memory Graph Representation

3.1 Locality

Reading data from the 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.

In our case, we even have two different types of locality at hand, which we call *spatial* and *hierarchical* locality: First, the nodes can be ordered by spatial proximity[1]. Second, highway-node routing is a hierarchical approach, where the query never moves downwards in the hierarchy. Therefore, it makes sense to also consider the level when ordering the nodes.

The following arrangement, which is a slightly generalised version of the ordering used in [6], takes into account both the spatial and the hierarchical locality. For a fixed *next-layer fraction* f, we divide the nodes into two groups: the first group contains the $(1-f)\cdot|V|$ nodes in the lower levels, the second group the $f\cdot|V|$ nodes in the higher levels. Within each group, we keep the original, spatial order. We recurse in the second group until all nodes fit into a single block.

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.

3.2 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 have to represent the forward and the backward search graph in order to allow forward and backward search, respectively. To avoid some redundancies, we store each edge only once and use a forward and a backward flag to mark whether the edge belongs to the forward or backward search graph or even to both.

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 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.

[1] In fact, the real-world road networks that we have obtained are already ordered fairly well.

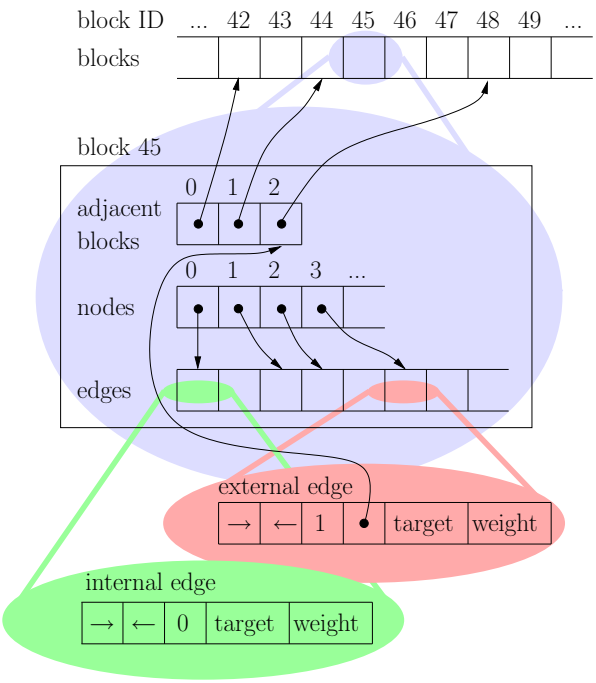

Fig. 1. External-memory graph data structure. Each edge stores three flags: a forward flag (→), a backward flag (←), and a flag that indicates whether it is an external edge leading to a node in a different block.

3.3 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 do not require the encoding procedure to be very efficient since the data structures can be built on a fast machine. In order to allow a fast decoding procedure, we make the following design choices: each block has the same constant size; each block should be encoded in the same way; each block contains a subset of the nodes and all incident edges[2]; the node range of each block is consecutive and maximal[3]; 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; each adjacent block ID, each node, and each edge is byte-aligned; the different attributes of an edge do not need to be byte-aligned[4].

Under these constraints, we want to minimise the required memory consumption. This 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: if the target node of an edge fits into the same block, we have an internal edge; otherwise, we have an external edge.

We have four unknowns: the total number b of blocks, the maximal number a of adjacent blocks, the maximal number n of nodes in a block, and the maximal number m of edges in a block. Note that the encoding of an edge weight does not depend on the distribution of the

[2] 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.
[3] Since nodes have got different degrees, this implies that different blocks can contain different numbers of nodes.
[4] For example, three 5-bit attributes can be stored in just two bytes, wasting only one bit.

nodes; consequently, we can determine a good encoding scheme in advance; in the following, we use w to denote the number of bits occupied by an edge weight.

Storing an adjacent block ID takes $\lceil \log_2(b) \rceil_8$ bits, where we use $\lceil x \rceil_8$ to denote the smallest number $\geq x$ that is a multiple of eight. Storing a node takes $\lceil \log_2(m) \rceil_8$ bits, an internal edge occupies $I := \lceil 3 + \lceil \log_2(n) \rceil + w \rceil_8$ bits, and an external edge $X := \lceil 3 + \lceil \log_2(a) \rceil + \lceil \log_2(n) \rceil + w \rceil_8$ bits.

In order to determine good values for b, a, n, and m, we use the following procedure:

1. *guess* values for b, a, n, and m;
2. first pass: determine the distribution of the nodes into blocks;
3. second pass: actually build the blocks and check for overflows;
4. if required, *adapt* values for b, a, n, and m, and repeat from 2.

A good strategy is to first 'guess' sufficiently large values and then try to reduce the values one after the other as far as possible. While the second pass is rather straightforward, the first pass is a little more tricky. We need a subroutine that places as many nodes as possible in the current block, starting from a given node with ID f. We manage a variable A containing the available memory in the current block; initially A is set to the block size minus the space needed for the header and minus $a \cdot \lceil \log_2(b) \rceil_8$ to store the adjacent block IDs. In order to check whether a node u can be added, we determine the required memory R and test for $R \leq A$. In the positive case, we subtract R from A and continue with the next node. The required memory R is $\lceil \log_2(m) \rceil_8$ plus the space needed for the edges. To determine the memory needed by an edge (u, v), we have to decide whether it leads to a different block or not. If $f \leq v \leq u$, it is definitely an internal edge. Thus, we add I to R. Otherwise, we assume that it is an external edge and add X to R. However, if $v > u$, there is still the chance that (u, v) will turn out to be an internal edge if v is added to the same block at a later point in time. In order to deal with this case, we increase a counter value $c(v)$ that counts the number of potentially internal edges that lead to v. Later, when the node v is considered to be added to the current block, we *subtract* $c(v) \cdot (X - I)$ from R because adding v implies that each of the $c(v)$ edges that have been counted as external edges earlier will occupy only the space of an internal edge. This way, the required memory can even get negative in some exceptional cases so that adding the node *increases* the available memory.

Further Compression Techniques. Since most edge weights in our real-world road networks are rather small and only comparatively few edges (in particular, some ferries and high-level overlay edges) 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.

We could use further compression techniques: for example, for an internal edge, we could store not the target index, but the *difference* of target and source index, which presumably can be done using less bits. However, we have not implemented this yet since our current design requires that edges are byte-aligned so that saving a few bits would only be useful if we could fall below the next byte limit.

3.4 Further Data Structures

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 $\beta(u)$ 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 $(\beta(u), i(u))$. 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 highway-node routing 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 [7], a similar distinction between read-only and volatile data structures has been used.

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

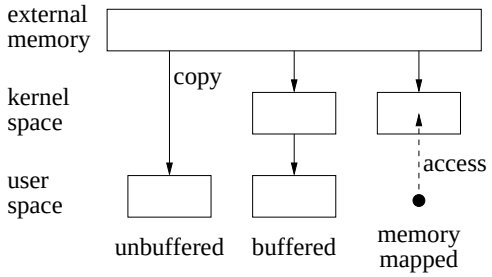

Fig. 2. Different ways to access the external memory.

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). 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[5], 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[6] 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 the highway-node routing search graphs that have been precomputed using contraction hierarchies [5]. Preprocessing takes 29 min, 28 min, and 38 min for Europe[7], 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[8]. 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.

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

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

[7] The preprocessing of the European search graph that we used for our experiments requires a comparatively expensive pre-preprocessing step in order to approximate the betweenness centrality of the nodes. The time of this pre-preprocessing step is not included in the 29 minutes. In the future, we can switch to a new version of the search graph, which came up later: it does not require the betweenness calculation and the average search space size increases by only 2.5%.

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

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, perform one query from $s_{i,0}$ to t_i without measuring the running time and, then, 100 queries from $s_{i,1}, \ldots, s_{i,100}$ to t_i performing time measurements; finally, 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.

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

5 Experimental Results

In all our experiments, we give the average query times needed to determine the shortest-path length, without outputting a complete description of the shortest path. For a remark on the removal of this limitation, we refer to Section 6.2. Note that the query times 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 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 decreases for increasing block sizes since the adjacent block IDs occupy less space. For all further experiments, we use a block size of 8 KB.

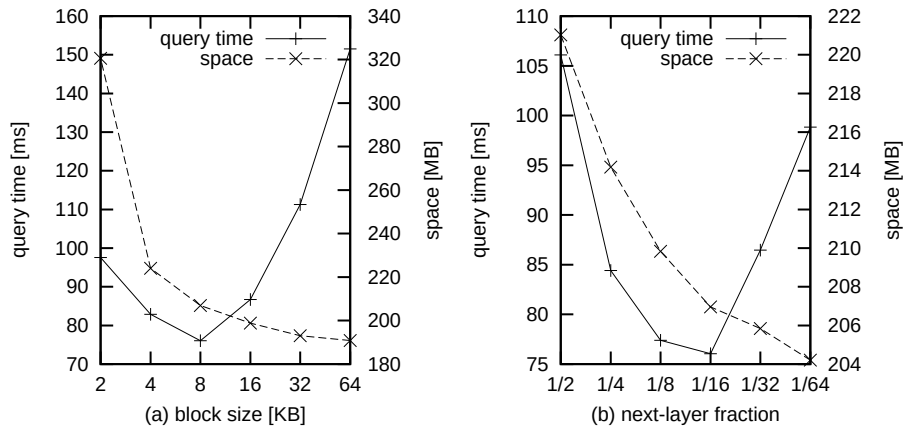

Fig. 3. Average query time and total space consumption depending on the chosen block size (a) and the next-layer fraction f (b).

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 slightly decreases for a decreasing value of f due to a smaller number of external edges. Note that just using the original spatial ordering yields a query time of 447 ms and a space consumption of 203 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 a very small cache size of only 1 MB is sufficient to arrive at quite good query times. Note that, on average, a cache size of 256 KB is almost sufficient to accomodate the blocks needed for a single query in the European road network. For the remaining experiments, we use the maximum cache size of 64 MB.

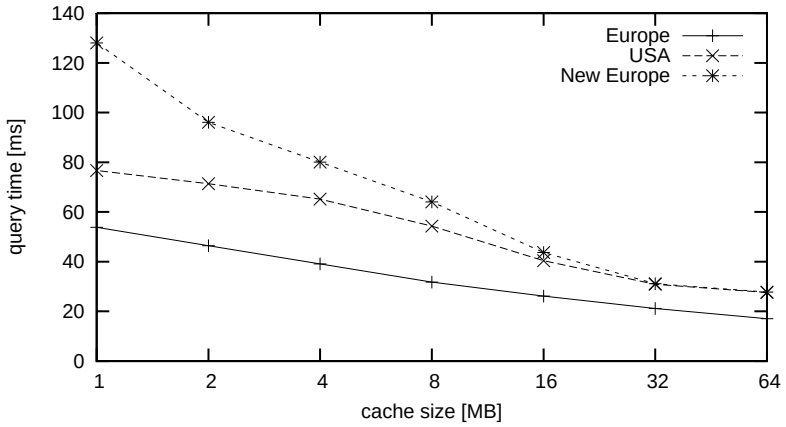

Fig. 4. Query performance for the second query type ('warm'), depending on the cache size.

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. The total memory consumption of the external-memory representation of the European road network is only 53% of the size that the original graph occupies in a standard adjacency-array representation. Note that we determined the optimal settings of the block size and the next-layer fraction for the European road network and then applied the same settings to the USA and New Europe. A more careful tuning of these settings might further reduce the space consumption of the latter networks.

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 32.2 blocks in case of the European road network. When the cache has been warmed-up, most blocks (in particular the ones that contain higher-level nodes) reside in the cache so that only about 3 blocks have to be fetched from external memory. This yields a very good query time of 17 ms. Recomputing the optimal path using the same target, but a different source node can be done in 22 ms. As expected, the bottleneck

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 b of blocks, the maximal number a of adjacent blocks, the maximal number n of nodes in a block, the maximal number m of edges in a block, the memory consumption I of an internal edge (with a large edge weight), the memory consumption X of an external edge (with a large edge weight), the time needed to build the external-memory graph representation provided that the search graph is already given, and the total memory consumption.

	$\lvert V\rvert$ [$\times 10^6$]	$\lvert E_{\text{orig}}\rvert$ [$\times 10^6$]	$\lvert E_{\text{search}}\rvert$ [$\times 10^6$]	b	a	n	m	I [B]	X [B]	time [s]	space [MB]
Europe	18.0	42.2	37.7	26491	183	2786	1700	4 (5)	5 (6)	121	207
USA	23.9	57.7	50.9	44153	337	970	1375	4 (6)	5 (7)	205	345
New Europe	33.7	75.1	68.1	59528	673	2693	1370	4 (6)	5 (7)	249	465

of our application are the accesses to the external memory: if all blocks had been preloaded, a shortest-path computation would take only about 7 ms instead of the 76 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	343	32.2	76.0	2.9	17.1	7.3	22.0	6.7
USA	268	54.2	124.9	7.8	27.6	12.4	31.9	5.7
New Europe	418	86.4	197.1	5.2	27.8	14.6	41.3	10.5

6 Discussion

To our knowledge, 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, 3]. In general, we can distinguish between goal-directed and hierarchical approaches.

Goal-Directed Approaches (e.g., [10–13, 7]) 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 3.1) can be exploited. In spite of the large memory requirements, Goldberg and Werneck [7] 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[9], a slightly larger graph ("New Eu-

[9] We use a more recent version of the ARM architecture, but with a slightly slower clock rate (330 MHz instead of 400 MHz); in [7], 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.

rope"), and a slightly smaller cache size (8 MB instead of 10 MB), our graph representation requires only 465 MB (a factor 8 less) and our random queries (without path unpacking) take 64 ms (more than 5 000 times faster) when our cache has been warmed up and 197 ms (more than 1 500 times faster) when our cache is initially empty.

Hierarchical Approaches (e.g., [14, 6, 4, 15, 2, 3]) 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, 6] 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" [16].

Commercial Systems. We made a few experiments with a commercial car navigation system, a recent TomTom One XL[10], 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—as our implementation does.[11]

More Related Work. Ajwani et al. [17] 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 Appendix A (Fig. 5, unbuffered).

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 neigbourhood 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 implementation 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

Currently, we are working on an efficient implementation of a path unpacking routine, which probably will be ready for the final version of this paper. A mix of the algorithms presented in [24, 5] seems to be promising. The main objective is to determine the very first edge of the computed route instantaneously so that the first driving direction can be generated.

[10] AK9SQ CSBUS, application version 6.593, OS version 1731, 29 MB RAM, road network of Western Europe version 675.1409

[11] Furthermore, to the best of our knowledge, current commercial systems do *not* compute exact routes.

Increasing the compression rate seems possible, in particular by using more sophisticated techniques, e.g. from [18], and by applying an individual encoding scheme to each block. 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 the backward search space.

Acknowledgements

We would like to thank Robert Geisberger for providing precomputed contraction hierarchies [5] 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 (2008) submitted to WEA'08.
2. Schultes, D., Sanders, P.: Dynamic highway-node routing. In: 6th Workshop on Experimental Algorithms (WEA). Volume 4525 of LNCS., Springer (2007) 66–79
3. Schultes, D.: Route Planning in Road Networks. PhD thesis (2007) submitted.
4. Sanders, P., Schultes, D.: Engineering highway hierarchies. In: 14th European Symposium on Algorithms (ESA). Volume 4168 of LNCS., Springer (2006) 804–816
5. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks (2008) submitted to WEA'08.
6. 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
7. 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
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. Bast, H., Funke, S., Sanders, P., Schultes, D.: Fast routing in road networks with transit nodes. Science **316**(5824) (2007) 566
16. Goldberg, A.: personal communication (2008)
17. Ajwani, D., Malinger, I., Meyer, U., Toledo, S.: Characterizing the performance of flash memory storage devices and its impact on algorithm design. Technical Report MPI-I-2008-1-001, Max Planck Institut für Informatik, Saarbrücken (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. Delling, D., Sanders, P., Schultes, D., Wagner, D.: Highway hierarchies star. In: 9th DIMACS Implementation Challenge [8]. (2006)

A 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 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.

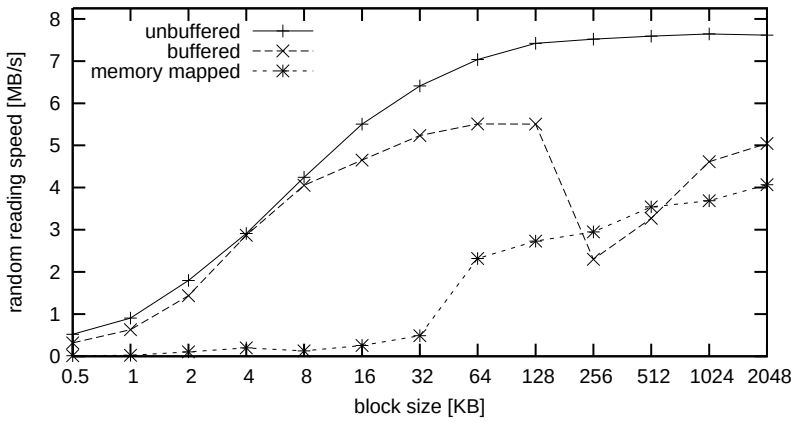

Fig. 5. Reading from flash memory.