PHAST: Hardware-Accelerated Shortest Path Trees

Daniel Delling Andrew V. Goldberg Andreas Nowatzyk Renato F. Werneck
Microsoft Research Silicon Valley, 1065 La Avenida, Mountain View, CA 94043, USA

September 2010

Technical Report
MSR-TR-2010-125

We present a novel algorithm to solve the nonnegative single-source shortest path problem on road networks and other graphs with low highway dimension. After a quick preprocessing phase, we can compute all distances from a given source in the graph with essentially a linear sweep over all vertices. Because this sweep is independent of the source, we are able to reorder vertices in advance to exploit locality. Moreover, our algorithm takes advantage of features of modern CPU architectures, such as SSE and multi-core. Compared to Dijkstra's algorithm, our method makes fewer operations, has better locality, and is better able to exploit parallelism at multi-core and instruction levels. We gain additional speedup when implementing our algorithm on a GPU, where our algorithm is up to three orders of magnitude faster than Dijkstra's algorithm on a high-end CPU. This makes applications based on all-pairs shortest-paths practical for continental-sized road networks. The applications include, for example, computing graph diameter, exact arc flags, and centrality measures such as exact reaches or betweenness.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
`http://www.research.microsoft.com`

1 Introduction

In recent years, performance gains in computer systems have come mainly from increased parallelism. As a result, exploiting the full potential of a modern computer has become increasingly difficult. Applications must not only work on multiple cores, but also access memory efficiently, taking into account issues such as data locality. Parallel algorithms are often unavailable or involve a compromise, performing more operations than the best sequential algorithm for the same problem. In this paper we introduce a compromise-free algorithm for the single-source shortest path problem on road networks. Our algorithm performs fewer operations than existing ones, while taking advantage of locality, multi-core and instruction-level parallelism.

The *single-source shortest path problem* is a classical optimization problem. Given a graph $G = (V, A)$, a length $\ell(a)$ assigned to each arc $a \in A$, and a source vertex s, the goal is to find shortest paths from s to all other vertices in the graph. Algorithms for this problem have been studied since the 1950's. The *non-negative single-source shortest path problem* (NSSP), in which $\ell(a) \geq 0$, is a special case that comes up in several important applications. It can be solved more efficiently than the general case with Dijkstra's algorithm [6, 11]. When implemented with the appropriate priority queues [15], its running time is within a factor of three of breadth-first search (BFS), a simple linear-time traversal of the graph. This indicates that any significant practical improvements in performance must take advantage of better locality and parallelism. Both are hard to achieve based on Dijkstra's algorithm [24, 25].

Motivated by web-based map services and autonomous navigation systems, the problem of finding shortest paths in road networks has received a great deal of attention recently; see e.g. [7, 8] for overviews. However, most research focused on accelerating point-to-point queries, in which both a source s and a target t are known. Up to now, however, Dijkstra's algorithm was still the fastest known solution to the NSSP problem.

We present a new algorithm for the NSSP problem that works well for certain classes of graphs, including road networks. We call it *parallel hardware-accelerated shortest path trees (PHAST)*. Building on previous work on point-to-point algorithms, PHAST uses *contraction hierarchies* [14] to essentially reduce the NSSP problem to a traversal of a shallow, acyclic graph. This allows us to take advantage of modern computer architectures and get a significant improvement in performance.

The PHAST algorithm requires a preprocessing phase, whose cost needs a moderate number of shortest path computations to be amortized. Moreover, PHAST only works well on certain classes of graphs. Fortunately, however, road networks are among them. Several important practical applications require multiple shortest path computations on road networks, such as preprocessing for route planning (see, e.g., [19, 23, 17, 18]) or the computation of certain centrality measures, like betweenness [2, 13]. For these applications, PHAST is extremely efficient. On continental-sized road networks, a purely sequential version of our algorithm is two orders of magnitude faster than the best previous solution. Moreover, PHAST scales well on multi-core machines. On a standard 4-core workstation, one can compute all-pairs shortest paths in a few days instead of several months.

Another development in modern computers is the availability of very powerful, highly-parallel, and relatively cheap graphics processing units (GPUs). They have a large number of specialized processors and a highly optimized memory system. Although aimed primarily at computer graphics applications, GPUs have increasingly been used to accelerate general-purpose computa-

tions [27]. In this paper, we propose an efficient GPU implementation of PHAST. Note that this is nontrivial, since GPUs are geared towards computation on very regular data objects, unlike actual road networks. Still, our implementation achieves significant speedups even compared to the CPU implementation of PHAST itself. On a standard workstation equipped with a high-end consumer graphics card, we gain another order of magnitude over CPU-based PHAST. This reduces the computation of all-pairs shortest paths on a continental-sized road network to about half a day, making applications requiring such computations practical.

This paper is organized as follows. Section 2 reviews Dijkstra's algorithm and the point-to-point algorithm PHAST builds upon, contraction hierarchies [14]. Section 3 describes the basic PHAST algorithm. Section 4 shows how to improve locality to obtain a faster single-core version of the algorithm. Section 5 shows how the algorithm can be parallelized in different ways, leading to even greater speedups on multi-core setups. Section 6 describes a typical GPU architecture and a GPU implementation of PHAST. Section 7 presents how to extend PHAST to compute the auxiliary data needed for some applications. Section 8 reports detailed experimental results. Final remarks are made in Section 9.

2 Background

2.1 Dijkstra's Algorithm

We now briefly review the NSSP algorithm proposed by Dijkstra [11] and independently by Dantzig [6]. For every vertex v, the algorithm maintains the length $d(v)$ of the shortest path from the source s to v found so far, as well as the predecessor (*parent*) $p(v)$ of v on the path. Initially $d(s) = 0$, $d(v) = \infty$ for all other vertices, and $p(v) = \mathit{null}$ for all v. The algorithm maintains a priority queue of *unscanned* vertices with finite d values. At each step, it removes from the queue a vertex v with minimum $d(v)$ value and *scans* it: for every arc $(v, w) \in A$ with $d(v) + \ell(v, w) < d(w)$, it sets $d(w) = d(v) + \ell(v, w)$ and $p(w) = v$. The algorithm terminates when the queue becomes empty.

Efficient implementations of this algorithm rely on fast priority queues. On graphs with n vertices and m arcs, an implementation of the algorithm using binary heaps runs in $O(m \log n)$ time. One can do better, e.g., using k-heaps [20] or Fibonacci heaps [12], the latter giving an $O(m + n \log n)$ bound. If arc lengths are integers in $[0 \dots C]$, bucket-based implementations of Dijkstra's algorithm work well. The first such implementation, due to Dial [10], gives an $O(m+nC)$ bound. There have been numerous improvements, including some that are very robust in practice. In particular, multi-level buckets [9] and smart queues [15] run in $O(m + n \log C)$ worst-case time.

Smart queues actually run in linear time if arc lengths have a uniform distribution [15]. In fact, experimental results show that, when vertex IDs are randomly permuted, an implementation of NSSP using smart queues is usually within a factor of two of breadth-first search (BFS), and never more than three, even on especially-built bad examples.

For concreteness, throughout this paper we will illustrate the algorithms we discuss with their performance on one well-known benchmark instance representing the road network of Western Europe [8], with 18 million vertices and 42 million arcs. (More detailed experiments, including additional instances, will be presented in Section 8.) If vertex IDs are assigned at random, the smart queue algorithm takes 9.2 seconds on an Intel Core-i7 920 clocked at 2.67 GHz, and BFS takes 7.0 seconds. The performance of both algorithms improve if one reorders the vertices such

that neighboring vertices tend to have similar IDs, since this reduces the number of cache misses during the computation. Interestingly, reordering the vertices by a simple depth first search (DFS) like procedure (explained in detail in Section 8) already gives good results: Dijkstra's algorithm takes 3.3 seconds, and BFS takes 2.2. We tested several other layouts but were unable to obtain significantly better performance. Therefore, unless otherwise noted, in the remainder of this paper we use the DFS layout when reporting running times.

2.2 Contraction Hierarchies

We now discuss the *contraction hierarchies* (CH) algorithm, proposed by Geisberger et al. [14] to speed up point-to-point shortest path computations on road networks. It has two phases. The preprocessing phase takes only the graph as input, and produces some auxiliary data. The query phase takes the source s and target t as inputs, and uses the auxiliary data to compute the shortest path from s to t.

The preprocessing phase of CH picks a permutation of the vertices and *shortcuts* them in this order. The *shortcut operation* deletes a vertex v from the graph (temporarily) and adds arcs between its neighbors to maintain the shortest path information. More precisely, for any pair $\{u, w\}$ of neighbors of v such that $(u, v) \cdot (v, w)$ is the only shortest path in between u and w in the current graph, we add a *shortcut* (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$. The output of this routine is the set A^+ of shortcut arcs and the position of each vertex v in the order (denoted by $rank(v)$). Although any order gives a correct algorithm, query times and the size of A^+ may vary. In practice, the best results are obtained by on-line heuristics that select the next vertex to shortcut based, among other factors, on the number of arcs added and removed from the graph in each step [14].

The query phase of CH runs a bidirectional version of Dijkstra's algorithm on the graph $G^+ = (V, A \cup A^+)$, with one crucial modification: both searches only look at *upward* arcs, those leading to neighbors with higher rank. More precisely, let $A^\uparrow = \{(v, w) \in A \cup A^+ : rank(v) < rank(w)\}$ and $A^\downarrow = \{(v, w) \in A \cup A^+ : rank(v) > rank(w)\}$. During queries, the forward search is restricted to $G^\uparrow = (V, A^\uparrow)$, and the reverse search to $G^\downarrow = (V, A^\downarrow)$. Each vertex v maintains estimates $d_s(v)$ and $d_t(v)$ on distances from s (found by the forward search) and to t (found by the reverse search). These values can be infinity. The algorithm keeps track of the vertex u minimizing $\mu = d_s(u) + d_t(u)$, and each search can stop as soon as the minimum value in its priority queue is at least as large as μ.

Consider the maximum-rank vertex u on the shortest s-t path. As shown in [14], u minimizes $d_s(u) + d_t(u)$ and the shortest path from s to t is given by the concatenation of the s–u and u–t paths. Furthermore, the forward search finds the shortest path from s to u (which belongs to $G^\uparrow$), and the backward search finds the shortest path from u to t (which belongs to $G^\downarrow$).

This simple algorithm is surprisingly efficient on road networks. On the European road network, random s–t queries visit fewer than 500 vertices (out of 18 million) on average and take a fraction of a millisecond on a standard workstation. Preprocessing takes only about 10 minutes and adds fewer shortcuts than there are original arcs.

Note that the forward search can easily be made target-independent by running Dijkstra's algorithm in $G^\uparrow$ from s until the priority queue is empty. Even with this loose stopping criterion, the *upward search* only visits about 500 vertices on average. Also note that the distance label $d_s(u)$ of a scanned vertex u does not necessarily represent the actual distance from s to u—it

may be only an upper bound. We would have to run a backward search from u to find the actual shortest path from s to u.

From a theoretical point of view, CH works well in networks with low *highway dimension* [1]. Roughly speaking, these are graphs in which one can find a very small set of "important" vertices to hit all "long" shortest paths.

3 Basic PHAST Algorithm

We are now ready to discuss a basic version of PHAST, our new algorithm for the NSSP problem. It has two phases, preprocessing and (multiple) NSSP computations. The algorithm is efficient only if there are sufficiently many NSSP computations to amortize the preprocessing cost.

The preprocessing phase of PHAST just runs a standard CH preprocessing, which gives us a set of shortcuts A^+ and a vertex ordering. This is enough for correctness. We discuss improvements in the next section.

A PHAST query initially sets $d(v) = \infty$ for all $v \neq s$, and $d(s) = 0$. It then executes the actual search in two subphases. First, it performs a simple forward CH search: it runs Dijkstra's algorithm from s in $G^{\uparrow}$, stopping when the priority queue becomes empty. This sets the distance labels $d(v)$ of all vertices visited by the search. The second subphase scans all vertices in $G^{\downarrow}$ in descending rank order.[1] To scan v, we examine each incoming arc $(u, v) \in A^{\downarrow}$; if $d(v) > d(u) + \ell(u, v)$, we set $d(v) = d(u) + \ell(u, v)$.

Theorem 3.1 *PHAST computes correct distance labels from s.*

Proof. We have to prove that, for every vertex v, $d(v)$ eventually represents the distance from s to v in G (or, equivalently, in G^+). Consider one such v in particular, and let w be the maximum-rank vertex on the shortest path from s to v in G^+. The first phase of PHAST is a forward CH query, and is therefore guaranteed to find the shortest s–w path and to set $d(w)$ to its correct value (as shown in [14]). By construction, G^+ contains a shortest path from w to v in which vertices appear in descending rank order. The second phase scans the arcs in this order, which means $d(v)$ is computed correctly. ■

In this and the following few sections, our discussion will focus on the computation of distance labels only, and not the actual shortest path trees. Section 7 shows how to compute parent pointers and other auxiliary data in a straightforward manner.

On our benchmark instance, PHAST performs a single-source shortest path computation in about 2.2 seconds, which is the same as BFS and lower than the 3.3 seconds needed for Dijkstra's algorithm.

4 Improvements

In this section we describe how the performance of PHAST can be significantly improved by taking into account the features of modern computer architectures. We focus on its second phase (the linear sweep), since the time spent on the forward CH search is negligible: less than 0.05 ms

[1] For correctness, any reverse topological order will do.

on our benchmark instance. In Section 4.1, we show how to improve locality when computing a single tree. Then, Section 4.2 discusses how building several trees simultaneously not only improves locality even further, but also enables the use of special instruction sets provided by modern CPUs. Finally, Section 4.3 explains how a careful initialization routine can speed up the computation.

4.1 Reordering Vertices

To explain how one can improve locality and decrease the number of cache misses, we first need to address data representation. For best locality, $G^{\uparrow}$ and $G^{\downarrow}$ are represented separately, since each phase of our algorithm works on a different graph.

Vertices have sequential IDs from 0 to $n-1$. We represent $G^{\uparrow}$ using a standard cache-efficient representation based on a pair of arrays. One array, *arclist*, is a list of arcs sorted by tail ID, i.e., arc $(u,\cdot)$ appears before $(w,\cdot)$ if $u < w$. This ensures that the outgoing arcs from vertex v are stored consecutively in memory. Each arc (v,w) is represented as a two-field structure containing the ID of the head vertex (w) and the length of the arc. The other array, *first*, is indexed by vertex IDs; *first*$[v]$ denotes the position in *arclist* of the first outgoing arc from v. To traverse the adjacency list, we just follow *arclist* until we hit *first*$[v+1]$. We keep a sentinel at *first*$[n]$ to avoid special cases.

The representation of $G^{\downarrow}$ is identical, except for the fact that *arclist* represents incoming instead of outgoing arcs. This means that *arclist* is sorted by head ID, and the structure representing an arc contains the ID of its tail (not head). Distance labels are maintained as a separate array, indexed by vertex IDs.

Given this representation, a simple reordering of the vertices leads to improved memory locality during the second phase of PHAST, which works on $G^{\downarrow}$.

To determine a good new order, we first assign *levels* to vertices. Levels can be computed as we shortcut vertices during preprocessing, as follows. Initialize all levels to zero; when shortcutting a vertex u, we set $L(v) = \max\{L(v), L(u)+1\}$ for each current neighbor v of u, i.e., for each v such that $(u,v) \in A^{\uparrow}$ or $(v,u) \in A^{\downarrow}$. By construction, we have the following lemma.

Lemma 4.1 *If $(v,w) \in A^{\downarrow}$, then $L(v) > L(w)$.*

This means that the second phase of PHAST can process vertices in descending order of level: vertices on level i are only visited after all vertices on levels greater than i have been processed. This order respects the topological order of $G^{\downarrow}$.

Within the same level, we can scan the vertices in any order. In particular, by processing vertices within a level in increasing order of IDs,[2] we maintain some locality and decrease the running time of PHAST from 2.2 to 0.8 seconds.

We can obtain additional speedup by actually reordering the vertices. We assign lower IDs to vertices at higher levels; within each level, we keep the DFS order. Now PHAST will be correct with a simple linear sweep in increasing order of IDs. It can access vertices, arcs, and head distance labels sequentially, with perfect locality. The only non-sequential access is to the distance labels of the arc tails (recall that, when scanning v, we must look at the distance labels of its neighbors). Keeping the DFS relative order within levels helps to reduce the number of the associated cache misses.

[2] As mentioned in Section 2.1, we assign IDs according to a DFS order, which has a fair amount of locality.

As most data access is now sequential, we get a substantial speedup. With reordering, one NSSP computation is reduced from 0.8 seconds to 195 milliseconds, which is about 17 times faster than Dijkstra's algorithm. We note that the notion of reordering vertices to improve locality has been applied before to hierarchical route planning techniques [16], albeit in a more ad hoc manner.

4.2 Computing Multiple Trees

Reordering ensures that the only possible non-sequential accesses during the second stage of the algorithm happen when reading distance labels of arc tails. More precisely, when processing vertex v, we must look at all incoming arcs (u, v). The arcs themselves are arranged sequentially in memory, but the IDs of their tail vertices are not sequential.

We can improve locality by running multiple NSSP computations simultaneously. To grow trees from k sources $(s_0, s_1, \ldots, s_{k-1})$ at once, we maintain k distance labels for each vertex $(d_0, d_1, \ldots, d_{k-1})$. These are maintained as a single array of length kn, laid out so that the k distances associated with v are consecutive in memory.

The query algorithm first performs (sequentially) k forward CH searches, one for each source, and sets the appropriate distance labels of all vertices reached. As we have seen, the second phase of PHAST processes vertices in the same order regardless of source, so we can process all k sources during the same pass. We do so as follows. To process each incoming arc (u, v) into a vertex v, we first retrieve its length and the ID of u. Then, for each tree i (for $0 \leq i < k$), we compute $d_i(u) + \ell(u, v)$ and update $d_i(v)$ if the new value is an improvement. For a fixed v, all $d_i(v)$ values are consecutive in memory and are processed sequentially, which leads to better locality and fewer cache misses.

Increasing k leads to better locality, but only up to a point: storing more distance labels tend evict other, potentially useful, data from the processor caches. Another drawback is increased memory consumption, because we need to keep an array with kn distance labels.

Still, for small values of k we can achieve significant speedups with a relatively small memory overhead. Setting $k = 16$ reduces the average running time per tree from 195.3 to 113.3 milliseconds on our benchmark instance.

SSE Instructions. We use 32-bit distance labels. Current x86-CPUs have special 128-bit SSE registers that can hold four 32-bit integers and allow basic operations, such as addition and minimum, to be executed in parallel. We can use these registers during our sweep through the vertices to compute k trees simultaneously, k being a multiple of 4. For simplicity, assume $k = 4$. When processing an arc (u, v), we load all four distance labels of u into an SSE register, and four copies of $\ell(u, v)$ into another. With a single SSE instruction, we compute the packed sum of these registers. Finally, we build the (packed) minimum of the resulting register with the four distance labels of v, loaded into yet another SSE register. Note that computing the minimum of integers is only supported by version 4.1 of SSE or higher.

For $k = 16$, using SSE instructions reduces the average run-time per tree from 113.3 to 43.0 milliseconds, for an additional factor of 2.6 speedup. In total, this algorithm is 77 times faster than Dijkstra's algorithm on one core.

4.3 Initialization

PHAST (like Dijkstra's algorithm) assumes that all distance labels are set to ∞ during initialization. This requires a linear sweep over all distance labels, which takes about 10 milliseconds. This is negligible for Dijkstra's algorithm, but represents a significant time penalty for PHAST. To avoid this, we mark vertices visited during the CH search with a single bit. During the linear sweep, when scanning a vertex v, we check for this bit: If it is not set, we know $d(v) = \infty$, otherwise we know that v has been scanned during the upward search and has a valid (though not necessarily correct) value. After scanning v we unmark the vertex for the next shortest path tree computation. The results we have reported so far already include this implicit initialization.

5 Exploiting Parallelism

We now consider how to use parallelism to speed up PHAST on a multi-core CPU. For computations that require shortest path trees from several sources, the obvious approach for parallelization is to assign different sources to each core. Since the computations of the trees are independent from one another, we observe excellent speedups. Running on four cores, without SSE, the average running time per tree ($k = 1$) decreases from 195.3 to 49.2 ms, a speedup of 3.97. (Recall that k indicates the number of sources per linear sweep.) Setting k to 16 (again without SSE), the running time drops from 113.3 to 28.5 ms per tree, a speedup of 3.98.

However, we can also parallelize a single tree computation. On our benchmark instance, the number of the vertex levels is around 140, orders of magnitude smaller than the number of vertices. Moreover, low levels contain many more vertices than upper levels. Half of the vertices are in level 0, for example. This allows us to process vertices of the same level in parallel if multiple cores are available. We partition vertices in a level into (roughly) equal-sized blocks and assign each block to a thread (core). When all threads terminate, we start processing the next level. Blocks and their assignment to threads can be computed during preprocessing. Running on 4 cores, we can reduce a single NSSP computation from 195.3 to 51.1 milliseconds on the same machine, a factor of 3.82 speedup. Note that this type of parallelization is the key to our GPU implementation of PHAST, explained in the next section.

6 GPU Implementation

Our improved implementation of PHAST is memory bandwidth limited. One way to overcome this limitation is to use a modern graphics card. The NVIDIA GTX 480 (Fermi) we use in our tests has a higher memory bandwidth (177 GB/s) than a high-end Intel Xeon CPU (32 GB/s). Although clock frequencies tend to be lower on GPUs (less than 1 GHz) than CPUs (higher than 3 GHz), the former can compensate by running many threads in parallel. The NVIDIA GTX 480 has 15 independent cores, each capable of executing 32 threads (called a *warp*) in parallel. It follows a Single Instruction Multiple Threads (SIMT) model, which uses predicated execution to preserve the appearance of normal thread execution at the expense of inefficiencies when the control-flow diverges. Moreover, barrel processing is used to hide DRAM latency. For maximal efficiency, all threads of a warp must access memory in certain, hardware-dependent ways. Accessing 32 consecutive integers of an array, for example, is efficient. Another constraint of GPU-based computations is that communication between main and GPU memory is rather

slow. Fortunately, off-the-shelf GPUs nowadays have enough on-board RAM (1.5 GB in our case) to hold all the data we need.

Our GPU-based variant of PHAST, called *GPHAST*, satisfies all the constraints mentioned above. In a nutshell, GPHAST outsources the linear sweep to the GPU and the CPU remains responsible for computing the upward CH trees. During initialization, we copy both $G^{\downarrow}$ and the array of distance labels to the GPU. To compute a tree from s, we first run the CH search on the CPU and copy the search space (with less than 2 KB) to the GPU. As in the single-tree parallel version of PHAST, we then process each level in parallel. The CPU starts, for each level i, a *kernel* on the GPU, which is a (large) collection of threads that all execute the same code and that are scheduled by the GPU hardware. Note that each thread is responsible for exactly one vertex. With this approach, the overall access to the GPU memory within a warp is efficient in the sense that DRAM bandwidth utilization is minimized. Caching on the GPU is only moderately effective due to limited data reuse.

On an NVIDIA GTX 480, installed on the machine used in our previous experiments, a single tree can be computed in 9.7 ms. This represents a speedup of 339 over Dijkstra's algorithm, 20 over the sequential variant of PHAST, and 5.3 over the 4-core CPU version of PHAST. Note that GPHAST uses a single core from the CPU.

We also tested reordering vertices by degree to make each warp work on vertices with the same degree. However, this has a strong negative effect on the locality of the distance labels of the tails of the incoming arcs. Hence, we keep the same ordering of vertices as for our CPU implementation of PHAST.

Multiple Trees. If the GPU has enough memory to hold additional distance labels, GPHAST also benefits from computing many trees in parallel. When computing k trees at once, the CPU first computes the k CH upward trees and copies all k search spaces to the GPU. Again, the CPU activates a GPU kernel for each level. Each thread is still responsible for writing exactly one distance label. We assign threads to warps such that threads within a warp work on the same vertices. This allows more threads within a warp to follow the same instruction flow, since they work on the same part of the graph.

For $k = 16$, GPHAST needs 2.9 ms per shortest path tree. That is more than 1000 times faster than sequential Dijkstra's algorithm, 67 times faster than sequential PHAST, and 6.7 times faster than computing 64 trees on the CPU in parallel (16 sources per sweep, one sweep per core). GPHAST can compute all-pairs shortest paths (i.e., n trees) in about 14 hours on a standard workstation. On the same machine, n executions of Dijkstra's algorithm would take 230 days, even if we compute 4 trees in parallel (one on each core).

7 Computing Auxiliary Information

Our discussion so far has assumed that PHAST computes only distances from a root s to all vertices in the graph. We now discuss how it can be extended to compute the actual shortest path trees (i.e., parent pointers) and show how PHAST can be used for several concrete applications.

7.1 Building Trees

One can easily change PHAST to compute parent pointers in G^+. When scanning v during the linear sweep phase, it suffices to remember the arc (u, v) responsible for $d(v)$. Note that some of the parent pointers in this case will be shortcuts (not original arcs). For many applications, paths in G^+ are sufficient and even desirable [4].

If the actual shortest path tree in G is required, it can be easily obtained with one additional pass through the arc list of G. During this pass, for every original arc $(u, v) \in G$, we check whether the identity $d(v) = d(u) + \ell(u, v)$ holds; if it does, we make u the parent of v. As long as all original arc lengths are strictly positive, this leads to a shortest path tree in the original graph.

In some applications, one might need to compute all distance labels, but the full description of a single s–t path. In such cases, a path in G^+ can be expanded into the corresponding path in G in time proportional to the number of arcs on it [14].

7.2 Applications

With the tree construction procedure at hand, we can now give practical applications of PHAST: computing exact diameters and centrality measures on continental-sized road networks, and as well as faster preprocessing for point-to-point route planning techniques. The applications require extra bookkeeping or an additional traversals of the arc list to compute some auxiliary information. As we shall see, the modifications are easy and the computational overhead is relatively small.

Diameter. The diameter of a graph G is defined by the longest shortest path in G. Its exact value can be computed by building n shortest path trees. PHAST can easily do it by making each core keep track of the maximum label it encounters. The maximum of these values is the diameter. To use GPHAST, we maintain an additional array of size n to keep track of the maximum value assigned to each vertex over all n shortest path computations. In the end, we do one sweep over all vertices to collect the maximum. This is somewhat memory-consuming, but it keeps the memory accesses within the warps efficient.

Arc Flags. *Arc flags* [22, 23] are used to speed up the computation of point-to-point shortest paths. A preprocessing algorithm first computes a partition $\mathcal{C}$ of V into *cells*, then attaches a *label* to each arc a. A label contains, for each cell $C \in \mathcal{C}$, a Boolean flag $F_C(a)$ which is `true` if there is a shortest path starting with a to at least one vertex in C. During queries, a modified variant of Dijkstra's algorithm only considers arcs for which the flag of the target cell is set to `true`. This approach can easily be made bidirectional and is very efficient, with speedups of more than three orders of magnitude over a bidirectional version of Dijkstra's algorithm [19].

The main drawback of this approach is its preprocessing time. While a good partition can be computed in a few minutes [21, 26, 28], computing the flags requires building a shortest path tree from each boundary vertex, i.e., each vertex with an incident arc from another cell. In a typical setup, one has to compute about 40 000 shortest path trees resulting in preprocessing times of about 12 hours (on four cores). Instead of running Dijkstra's algorithm, however, we can run GPHAST with tree reconstruction, reducing the time to set flags to about 3 minutes.

Centrality Measures. PHAST can also be used to compute the exact *reach* [18] of a vertex v. The reach of v is defined as the maximum, over all shortest s–t paths containing v, of $\min\{dist(s,v), dist(v,t)\}$. This notion is very useful to accelerate the computation of point-to-point shortest paths. The best known method to calculate exact reaches for all vertices within a graph requires computing all n shortest path trees, which we can do much faster with PHAST. Fast heuristics [16] compute reach upper bounds only and are fairly complicated.

Another centrality measure based on shortest paths is *betweenness* [2, 13]. It is defined as $c_B(v) = \sum_{s \neq v \neq t \in V} \sigma_{st}(v)/\sigma_{st}$, where σ_{st} is the number of shortest paths between two vertices s and t, and $\sigma_{st}(v)$ is the number of shortest s–t paths on which v lies on. Computing exact betweenness relies on n shortest path tree computations [5]. Again, replacing Dijkstra's algorithm by PHAST makes exact betweenness tractable for continent-size road networks.

Note that these and other applications often require traversing the resulting shortest path tree in bottom-up fashion. This can be easily done in a cache-efficient way by scanning vertices in reverse level order.

8 Experimental Results

8.1 Experimental Setup

We implemented our CPU-based PHAST with all optimizations from Section 4 in C++ and compiled it with Microsoft Visual C++ 2010. For parallelization, we use OpenMP. CH queries use a binary heap as priority queue; we tested other data structures, but their impact on performance is negligible because the queue size is small.

As already mentioned, we run most of our evaluation on an Intel Core-i7 920 running Windows 7 Enterprise. It has four cores clocked at 2.67 GHz and 12 GB of DDR3-1066 RAM. Our standard benchmark instance is the European road network, with 18 million vertices and 42 million arcs, made available by PTV AG [29] for the 9th DIMACS Implementation Challenge [8]. The length of each arc represents the travel time between its endpoints.

PHAST builds upon contraction hierarchies. We implemented a parallelized version of the CH preprocessing routine, as discussed in [14]. For improved efficiency, we use a slightly different priority function for ordering vertices. The priority of a vertex u is given by $2 \cdot ED(u) + CN(u) + H(u) + 5 \cdot L(u)$, where $ED(u)$ is the difference between the number of arcs added and removed (if u were contracted), $CN(u)$ is the number of contracted neighbors, $H(u)$ is the total number of arcs represented by all shortcuts added, and $L(u)$ is the level u would be assigned to. With this priority term, preprocessing takes about five minutes (on four cores) and generates upward and downward graphs with 33.8 million arcs each. The number of levels is 140, with half the vertices assigned to the lowest level, as shown in Figure 1. Note that the lowest 20 levels contain all but 100 000 vertices, while all but 1 000 vertices are assigned to the lowest 66 levels. We stress that the priority term has limited influence on the performance of PHAST. It works well with any function that produces a "good" contraction hierarchy (leading to fast point-to-point queries), such as those tested by Geisberger et al. [14].

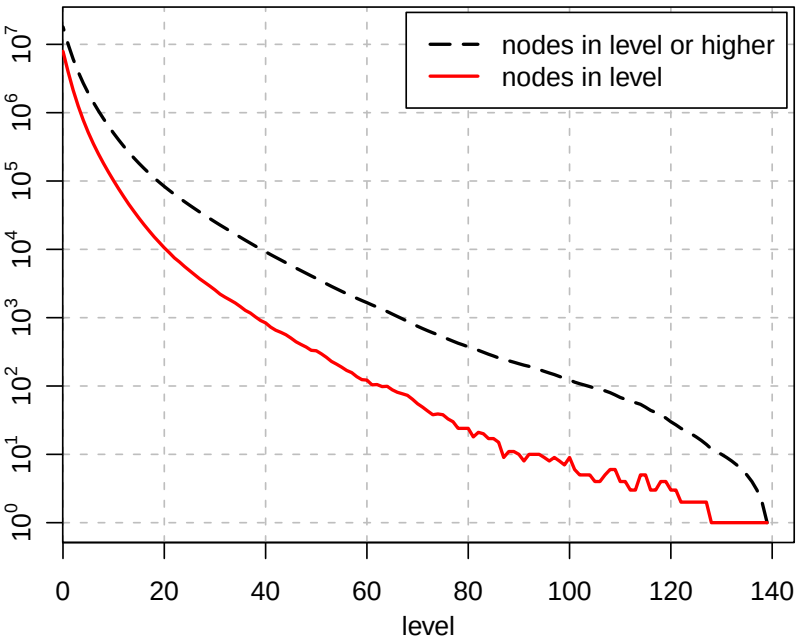

Figure 1: vertices per level

8.2 Single Tree

We now evaluate the performance of Dijkstra's algorithm and PHAST when computing a single tree. We tested different priority queues (for Dijkstra's algorithm) and different graph layouts (for both algorithms). We start with a *random* layout, in which vertex IDs are assigned randomly, to see what happens with poor locality. We also consider the *original* layout (as given in the input graph as downloaded); it has some spatial locality. Finally, we consider a *DFS* layout, with IDs given by the order vertices are discovered during a DFS from a random vertex. The resulting figures are given in Table 1. For reference we also include the running time of a simple BFS.

We observe that both the layout and the priority queue have an impact on Dijkstra's algorithm. It is four times slower when using the binary heap and the random layout than when using a bucket-based data structure and the DFS layout. For single-core applications, the smart queue implementation [15] (based on multi-level buckets) is robust and memory-efficient. In our setup, however, Dial's implementation [10], based on single-level buckets, is comparable on a single core and scales better on multiple cores. For the remainder of this paper, all numbers given for Dijkstra's algorithm executions refer to the Dial's implementation with the DFS layout.

The impact of the layout on PHAST is even more significant. By starting from the DFS ordering and then ordering by level, the average execution time for one source improves from 1479 to 195 ms, a speedup of 7.6. For all combinations tested, PHAST is always faster than Dijkstra's algorithm. The speedup is about a factor of 17 for sequential PHAST, while this number increases to 77 if we use four cores to scan the vertices within one level in parallel, as explained in Section 5.

To evaluate the overhead of PHAST, we also ran a lower bound test. To test the memory bandwidth of the system, we sequentially and independently read from all arrays (*first*, *arclist*, and the distance array) and then write a value to each entry of the distance array. On our test machine, this takes 76.4 ms; PHAST is only 2.5 times slower than this.

Table 1: Performance of various algorithms on the European road network. Three graph layouts are considered: random, as given in the input, and DFS-based.

		TIME PER TREE [MS]		
algorithm	details	random	input	DFS
Dijkstra	binary heap	12683	6562	5804
	Dial	8996	4087	3355
	smart queue	9298	4157	3290
BFS	—	6978	2844	2212
PHAST	original ordering	1479	803	783
	reordered by level	460	201	195
	reordered + four cores	153	54	51

Note that this lower bound merely iterates through the arc list in a single loop. In contrast, most algorithms (including PHAST) loop through the vertices, and for each vertex loop through its (few) incident arcs. Although both variants visit the same arcs in the same order, the second method has an inner loop with a very small (but varying) number of iterations, thus making it harder to be sped up by the branch predictor. Indeed, it takes 171 ms to traverse the graph exactly as PHAST does, but storing at $d(v)$ the sum of the lengths of all arcs into v. This is only 24 ms less than PHAST, which suggests that reducing the number of cache misses (from reading $d(u)$) even further by additional reordering is unlikely to improve the performance of PHAST significantly.

8.3 Multiple Trees

Next, we evaluate the performance of PHAST when computing many trees simultaneously. We vary both k (the number of trees per linear sweep) and the number of cores we use. We also evaluate the impact of SSE instructions. The results are given in Table 2.

Table 2: Average running times per tree when computing multiple trees in parallel. We consider the impact of using SSE instructions, varying the number of cores, and increasing the number of sources per sweep (k). The numbers in parentheses refer to the execution times when SSE is activated; for the remaining entries we did not use SSE.

sources/	TIME PER TREE [MS]		
sweep	1 core	2 cores	4 cores
1	195.3	97.4	49.2
4	140.7 (78.1)	70.4 (41.2)	35.2 (26.2)
8	121.5 (59.3)	60.8 (32.0)	30.5 (21.7)
16	113.3 (43.0)	56.8 (25.1)	28.5 (19.5)

Without SSE, we observe perfect speedup when using four cores instead of one, for all values of k. With SSE, however, we obtain smaller speedups when running on multiple cores. The more cores we use, and the higher k we pick, the more data we have to process in one sweep. Still, using all optimizations (SSE, multi-core) helps: the algorithm is 10 times faster with $k = 16$ on four cores than with $k = 1$ on one core.

For many cores and high values of k, memory bandwidth becomes the main bottleneck of PHAST. This is confirmed by executing our lower bound test on all four cores in parallel. For $k = 16$ and four cores, it takes 13.3 ms per "tree" to traverse all arrays in optimal (sequential) order. This is more than two thirds of the 19.5 ms needed by PHAST. This indicates that PHAST is approaching a barrier. This observation was the main motivation for our implementation of PHAST on a GPU, whose performance we discuss next.

8.4 GPHAST

To evaluate GPHAST, we implemented our algorithm from Section 6 using CUDA SDK 3.1 and compiled it with Microsoft Visual C++ 2008[3]. We conducted our experiments on an NVIDIA GTX 480 installed in our benchmark machine. The GPU is clocked at 701 MHz and has 1.5 GB of DDR5 RAM. Table 3 reports the performance when computing up to 16 trees simultaneously.

As the table shows, the performance of GPHAST is excellent. A single tree can be built in only 9.7 ms. When computing 16 trees in parallel, the running time per tree is reduced to a mere 2.9 ms. This is a speedup of more than three orders of magnitude over plain Dijkstra's algorithm. On average, we only need 161 picoseconds per distance label, which is roughly half a CPU clock cycle. On four cores, CH preprocessing takes 317 seconds, but this cost is amortized away after only 278 trees are computed if one uses GPHAST instead of Dijkstra's algorithm (also on four cores).

Table 3: Performance and GPU memory utilization of GPHAST in milliseconds per tree, depending on k.

trees / sweep	memory [MB]	time [ms]
1	395	9.70
2	464	4.97
4	605	3.85
8	886	3.22
16	1448	2.90

8.5 Hardware Impact

In this section we study the performance of PHAST on different computer architectures. Although GPHAST is clearly faster than PHAST, GPU applications are still very limited, and general-purpose servers usually do not have high-performance GPUs. Table 4 gives an overview of the five machines we tested. It should be noted that M2-1 and M2-4 are older machines (about 5 and 3 years old, respectively), whereas M1-4 (our default machine) is a recent commodity workstation. M2-6 and M4-12 are modern servers costing an order of magnitude more than M1-4. Note that M4-12 has many more cores than M2-6, but has worse sequential performance due to a lower clock rate. With the exception of M1-4, all machines have more than one NUMA node (local memory bank). For these machines, access to local memory is faster than to memory assigned to a different NUMA node. M4-12 has more NUMA nodes (eight) than CPUs (four). M4-12 and M2-6 run Windows Server 2008R2, M2-4 runs Windows Server 2008, and M2-1 runs Windows Server 2003.

We tested the sequential and parallel performance of Dijkstra's algorithm and PHAST on these machines. By default, the operating system moves threads from core to core during execution, which can have a significant adverse effect on memory-bound applications such as PHAST. Hence, we also ran our experiments with each thread pinned to a specific core. This ensures that the relevant distance arrays are always stored in the local memory banks, and results in improved

[3]CUDA SDK 3.1 is not compatible with Microsoft Visual C++ 2010 at the time of writing.

Table 4: Specifications of the machines tested. Column $|P|$ indicates the number of CPUs, whereas $|c|$ refers to the total number of physical cores in the machine. *NUMA nodes* refers to how many local memory banks the system has. The given memory bandwidth refers to the (theoretical) speed with which a core can access its local memory.

	CPU					memory				
name	brand	type	clock [GHz]	$\|P\|$	$\|c\|$	type	size [GB]	clock [MHz]	bandwidth [GB/s]	NUMA nodes
M2-1	AMD	Opteron 250	2.40	2	2	DDR	16	133	6.4	2
M2-4	AMD	Opteron 2350	2.00	2	8	DDR2	64	266	12.8	2
M4-12	AMD	Opteron 6168	1.90	4	48	DDR3	128	667	42.7	8
M1-4	Intel	Core-i7 920	2.67	1	4	DDR3	12	533	25.6	1
M2-6	Intel	Xeon X5680	3.33	2	12	DDR3	96	667	32.0	2

locality at all levels of memory hierarchy. For the same reason, on multi-socket systems, we also copy the graph to each local memory bank explicitly (when running in pinned mode). Table 5 shows the results. Running single-threaded, PHAST outperforms Dijkstra's algorithm by a factor of approximately 19, regardless of the machine. This factor increases slightly (to 21) when we compute one tree per core. The reason for this is that cores share the memory controller(s) of a CPU. Because PHAST has fewer cache misses, it benefits more than Dijkstra's algorithm from the availability of multiple cores.

The impact of pinning threads and copying the graph to local memory banks is significant. Without pinning, no algorithm performs well when run in parallel on a machine with more than one NUMA node. On M4-12, which has four CPUs (and eight NUMA nodes), we observe speedups of less than 6 (the number of cores of a single NUMA node) when using all 48 cores, confirming that non-local memory access is inefficient. However, if the data is properly placed in memory, the algorithms scale much better. On M4-12, using 48 cores instead of a single one makes PHAST 34 times faster. Unsurprisingly, pinning is not very helpful on M1-4, which has a single CPU.

Computing 16 trees per core within each sweep gives us another factor of 2, independent of the machine. When using all cores, PHAST is consistently about 40 times faster than Dijkstra's algorithm.

Table 5: Impact of different computer architectures on Dijkstra's algorithm and PHAST. When running multiple threads, we examine the effect of pinning each thread to a specific core or keeping it unpinned (free). In each case, we show the average running time per tree in milliseconds.

	Dijkstra [ms]			PHAST [ms]				
	single	1 tree/core		single	1 tree/core		16 trees/core	
machine	thread	free	pinned	thread	free	pinned	free	pinned
M2-1	6073.5	3967.3	3499.3	315.4	184.4	158.3	99.5	85.0
M2-4	6497.5	1499.0	1232.2	330.5	104.0	56.6	49.0	31.4
M4-12	5183.1	417.3	168.5	272.7	49.1	8.0	18.4	4.0
M1-4	3355.7	1143.2	1103.5	195.3	50.2	49.2	19.6	19.5
M2-6	2321.7	413.7	288.8	134.9	21.2	14.5	14.5	7.2

Table 6: Dijkstra's algorithm, PHAST, and GPHAST comparison. Column *memory used* indicates how much main memory is occupied during the construction of the trees (for GPHAST, we also use 1.5 GB of GPU memory).

			per tree		n trees	
		memory	time	energy	time	energy
algorithm	device	used [GB]	[ms]	[J]	[d:h:m]	[MJ]
Dijkstra	M1-4	2.8	1103.52	179.87	230:00:41	3237.73
	M2-6	8.2	288.81	95.88	60:04:51	1725.93
	M4-12	31.8	168.49	125.86	35:02:55	2265.52
PHAST	M1-4	5.8	19.47	3.17	4:01:24	57.12
	M2-6	15.6	7.20	2.39	1:12:13	43.03
	M4-12	61.8	4.03	3.01	0:20:09	54.19
GPHAST	GTX 480	2.2	2.90	1.13	0:14:33	20.36

8.6 Comparison: Dijkstra vs. PHAST vs. GPHAST

Next, we compare Dijkstra's algorithm with PHAST and GPHAST. Table 6 reports the best running times (per tree) for all algorithms, as well as how long it would take to solve the all-pairs shortest-paths problem. We also report how much energy these computations require (for GPU computations, this includes the entire system, including the CPU).

On the most powerful machine we tested, M4-12, the CPU-based variant is almost as fast as GPHAST, but the energy consumption under full work load is much higher for M4-12 (747 watts) than for M1-4 with installed GPU (390 watts). Together with the fact that GPHAST still is faster than PHAST on M4-12, the energy consumption per tree is about 2.5 times worse for M4-12. M1-4 without GPU (completely removed from the system) consumes 163 watts and is about as energy-efficient as M4-12. Interestingly, M2-6 (332 watts) does a better job than the other machines in terms of energy per tree. Still, GPHAST is more than two times more efficient than M2-6.

The graphics card itself costs half as much as the M1-4 machine on which it is installed, and the machine supports two cards. With two cards, GPHAST would be twice as fast, computing all-pairs shortest paths in about 7 hours (1.45 ms per tree), at a fifth of the cost of M4-12 or M2-6. In fact, one could even buy some very cheap machines equipped with 2 GPUs each. Since the linear sweep is by far the bottleneck of GPHAST, we can safely assume that the all-pairs shortest-paths computation scales perfectly with the number of GPUs.

8.7 Other Inputs

Up to now, we have only tested one input, the European road network with travel times. Here, we evaluate the performance of our algorithm if applied to travel distances instead of travel times. CH preprocessing takes about 41 minutes on this input, generating upwards and downwards graphs with 410 levels and 38.8 millions arcs each. Moreover, we evaluate the road network of the US (generated from TIGER/Line data [30]), also made available for the 9th DIMACS Implementation Challenge [8]. It has 24 million vertices and 58.3 million arcs. For travel times (distances), the CH preprocessing takes 10 (28) minutes, and the search graphs have 50.6 (53.7) million arcs and 101 (285) levels.

Table 7: Performance of Dijkstra's algorithm, PHAST, and GPHAST on other inputs.

		Europe		USA	
algorithm	device	time	distance	time	distance
Dijkstra	M1-4	1103.52	618.18	1910.67	1432.35
	M2-6	288.81	177.58	380.40	280.17
	M4-12	168.49	108.58	229.00	167.77
PHAST	M1-4	19.47	23.01	28.22	29.85
	M2-6	7.20	8.27	10.42	10.71
	M4-12	4.03	5.03	6.18	6.58
GPHAST	GTX 480	2.90	4.75	4.49	5.69

Table 7 presents the results. All algorithms are slower on the US graph, which has about 6 million more vertices than Europe. More interestingly, switching from travel times to distances has a positive effect on Dijkstra's algorithm (there are fewer *decrease-key* operations), but makes PHAST slower (it has more arcs to scan). However, the differences are relatively small. PHAST is always much faster than Dijkstra's algorithm, and GPHAST yields the best performance on all inputs.

8.8 Arc-Flags

Our last set of experiments deals with the computation of arc flags (as described in Section 7). The purpose of this test is to show that additional information (besides the distance labels) can indeed be computed efficiently. As input, we again use the road network of Western Europe with travel times. First, we use SCOTCH [28] to create a partition of the graph into 128 cells and 20240 boundary vertices in total. This takes less than a minute. Next, we remove the so-called *1-shell* (attached trees) of the graph, which has roughly 6 million vertices. Optimal flags for arcs within these trees can be set easily [3, 19]. This step takes 10 seconds. We then start the computation of the remaining arc flags. We compute for each boundary vertex two shortest path trees with GPHAST (one forward, one backward) and set the corresponding flags accordingly. We do this by copying G to the GPU, together with an array with 32-bit integers representing 32 flags for each arc. Since we need to compute 128 flags per arc, we copy this array to the main memory after computing 32 flags and reinitialize it. We do so due to memory constraints on the GPU; we set $k = 8$ for the same reason. Overall, with this approach the GPU memory is almost fully loaded.

The last step, tree construction and setting of arc flags, takes 201 seconds. This is 4.95 ms per boundary vertex (and direction) on average, of which 2.45 ms are spent computing the 12 million distance labels. Hence, reconstructing the parent pointers and setting the flags takes exactly as much time as computing the tree. This is expected, since we have to look at almost the same amount of data as during tree construction. On four cores we reduce the overall time for computing flags from 12.3 hours with Dijkstra's algorithm to 10 minutes (including partitioning and the CH preprocessing).

9 Concluding Remarks

We presented PHAST, a new algorithm for computing shortest path trees in graphs with low highway dimension, such as road networks. Not only is it faster than the best existing sequential algorithm, but it also exploits parallel features of modern computer architectures, such as SSE instructions, multiple cores, and GPUs. The GPU implementation can compute a shortest path tree about three orders of magnitude faster than Dijkstra's algorithm. This makes many applications on road networks, such as the exact computation of centrality measures, practical.

A previously studied parallel algorithm for the NSSP problem is Δ-*stepping* [25]. It performs more sequential operations than Dijkstra's algorithm, its parallel implementation requires fine-grained synchronization, and the amount of parallelism in Δ-stepping is less than that in PHAST. Thus, for a large number of NSSP computations, PHAST is a better choice. For a small number of computations (e.g., a single computation) PHAST is not competitive because of the preprocessing. However, it is not clear if on road networks Δ-stepping is superior to Dijkstra's algorithm in practice. The only study of Δ-stepping on road networks [24] has been done on a Cray MTA-2, which has an unusual architecture and requires a large amount of parallelism for good performance. The authors conclude that continent-size road networks do not have sufficient parallelism. It would be interesting to see how the Δ-stepping algorithm performs on a more conventional multi-core system or a small cluster. Another interesting project is an MTA-2 implementation of PHAST.

Future research includes studying the applicability of PHAST to other networks. A possible approach would be to stop the construction of the contraction hierarchy as soon as the average degree of the remaining vertices exceeds some value. PHAST then has to explore more vertices during the CH upwards search and would only sweep over those vertices that were contracted during preprocessing.

Finally, it would be interesting to study which kinds of map services are enabled by the ability to compute full shortest path trees instantaneously, and by the fact that preprocessing based on all-pairs shortest paths is now feasible.

References

[1] I. Abraham, A. Fiat, A. Goldberg, and R. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *Proc. 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2005.

[2] J. M. Anthonisse. The rush in a directed graph. Technical Report BN 9/71, Stichting Mathematisch Centrum, 2e Boerhaavestraat 49 Amsterdam, October 1971.

[3] R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009. Special Section on Selected Papers from ALENEX 2008.

[4] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

[5] U. Brandes. A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[6] G. B. Dantzig. *Linear Programming and Extensions*. Princeton Univ. Press, Princeton, NJ, 1962.

[7] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 117–139. Springer, 2009.

[8] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[9] E. V. Denardo and B. L. Fox. Shortest-Route Methods: 1. Reaching, Pruning, and Buckets. *Oper. Res.*, 27:161–186, 1979.

[10] R. B. Dial. Algorithm 360: Shortest Path Forest with Topological Ordering. *Comm. ACM*, 12:632–633, 1969.

[11] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numer. Math.*, 1:269–271, 1959.

[12] M. L. Fredman and R. E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. Assoc. Comput. Mach.*, 34:596–615, 1987.

[13] L. C. Freeman. A Set of Measures of Centrality Based Upon Betweeness. *Sociometry*, 40:35–41, 1977.

[14] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[15] A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM J. Comput.*, 37:1637–1655, 2008.

[16] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 93–139. American Mathematical Society, 2009.

[17] A. V. Goldberg and R. F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proc. 7th International Workshop on Algorithm Engineering and Experiments*, pages 26–40. SIAM, 2005.

[18] R. Gutman. Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proc. 6th International Workshop on Algorithm Engineering and Experiments*, pages 100–111, 2004.

[19] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 41–72. American Mathematical Society, 2009.

[20] D. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networs. *J. Assoc. Comput. Mach.*, 24:1–13, 1977.

[21] G. Karypis and G. Kumar. A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[22] E. Köhler, R. H. Möhring, and H. Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, volume 3503 of *Lecture Notes in Computer Science*, pages 126–138. Springer, 2005.

[23] U. Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *IfGIprints 22, Institut fuer Geoinformatik, Universitaet Muenster (ISBN 3-936616-22-1)*, pages 219–230, 2004.

[24] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. Parallel Shortest Path Algorithms for Solving Large-Scale Instances. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 249–290. American Mathematical Society, 2009.

[25] U. Meyer and P. Sanders. Δ-Stepping: A Parallelizable Shortest Path Algorithm. *J. Algorithms*, 49:114–152, 2003.

[26] V. Osipov and P. Sanders. n-Level Graph Partitioning. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, Lecture Notes in Computer Science, pages 278–289. Springer, 2010.

[27] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[28] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, Lecture Notes in Computer Science, pages 493–498. Springer, 1996.

[29] PTV AG - Planung Transport Verkehr. `http://www.ptv.de`.

[30] D. US Census Bureau, Washington. UA Census 2000 TIGER/Line files. http://www.census.gov/geo/www/tiger/tigerua/ua.tgr2k.html, 2002.