Customizable Route Planning

Daniel Delling[1], Andrew V. Goldberg[1],
Thomas Pajor[2⋆], and Renato F. Werneck[1]

[1] Microsoft Research Silicon Valley
{dadellin,goldberg,renatow}@microsoft.com
[2] Karlsruhe Institute of Technology
pajor@kit.edu

Abstract. We present an algorithm to compute shortest paths on continental road networks with arbitrary metrics (cost functions). The approach supports turn costs, enables real-time queries, and can incorporate a new metric in a few seconds—fast enough to support real-time traffic updates and personalized optimization functions. The amount of metric-specific data is a small fraction of the graph itself, which allows us to maintain several metrics in memory simultaneously.

1 Introduction

The past decade has seen a great deal of research on finding point-to-point shortest paths on road networks [7]. Although Dijkstra's algorithm [10] runs in almost linear time with very little overhead, it still takes a few seconds on continental-sized graphs. Practical algorithms use a two-stage approach: *preprocessing* takes a few minutes (or even hours) and produces a (linear) amount of auxiliary data, which is then used to perform *queries* in real time. Most previous research focused on the most natural metric, driving times. Real-world systems, however, often support other natural metrics as well, such as shortest distance, walking, biking, avoid U-turns, avoid/prefer freeways, or avoid left turns.

We consider the *customizable route planning* problem, whose goal is to perform real-time queries on road networks with *arbitrary metrics*. Such algorithms can be used in two scenarios: they may keep several active metrics at once (to answer queries for any of them), or new metrics can be generated on the fly. A system with these properties has obvious attractions. It supports real-time traffic updates and other dynamic scenarios, allows easy customization by handling any combination of standard metrics, and can even provide personalized driving directions (for example, for a truck with height and weight restrictions). To implement such a system, we need an algorithm that allows real-time queries, has fast customization (a few seconds), and keeps very little data for each metric. Most importantly, it must be *robust*: all three properties must hold for *any metric*. No existing algorithm meets these requirements.

To achieve these goals, we distinguish between two features of road networks. The *topology* is a set of static properties of each road segment or turn, such as

⋆ This work was done while the third author was at Microsoft Research Silicon Valley.

physical length, road category, speed limits, and turn types. The *metric* encodes the actual cost of traversing a road segment or taking a turn. It can often be described compactly, as a function that maps (in constant time) the properties of an edge/turn into a cost. We assume the topology is shared by the metrics and rarely changes, while metrics may change quite often and even coexist.

To exploit this separation, we consider algorithms for customizable route planning with *three stages*. The first, *metric-independent preprocessing*, may be relatively slow, since it is run infrequently. It takes only the graph topology as input, and may produce a fair amount of auxiliary data (comparable to the input size). The second stage, *metric customization*, is run once for each metric, and must be much quicker (a few seconds) and produce little data—a small fraction of the original graph. Finally, the *query stage* uses the outputs of the first two stages and must be fast enough for real-time applications.

In Section 2 we explore the design space by analyzing the applicability of existing algorithms to this setting. We note that methods with a strong hierarchical component, the fastest in many situations, are too sensitive to metric changes. We focus on separator-based methods, which are more robust but have often been neglected in recent research, since published results made them seem uncompetitive: the highest speedups over Dijkstra observed were lower than 60 [17], compared to thousands or millions with other methods.

Section 3 revisits and thoroughly reengineers a separator-based algorithm. By applying existing acceleration techniques, recent advances in graph partitioning, and some engineering effort, we can answer queries on continental road networks in about a millisecond, with much less customization time (a few seconds) and space (a few tens of megabytes) than existing acceleration techniques.

Another contribution of our paper is a careful treatment of turn costs (Section 4). It has been widely believed that any algorithm can be easily augmented to handle these efficiently, but we note that some methods actually have a significant performance penalty, especially if turns are represented space-efficiently. In contrast, we can handle turns naturally, with little effect on performance.

We stress that our algorithms are not meant to be the fastest on any particular metric. For "nice" metrics, our queries are somewhat slower than the best hierarchical methods. However, our queries are robust and suitable for real-time applications with arbitrary metrics, including those for which the hierarchical methods fail. Our method can quickly process new metrics, and the metric-specific information is small enough to keep several metrics in memory at once.

2 Previous Techniques

There has been previous work on variants of the route planning problem that deal with multiple metrics in a nontrivial way. The preprocessing of SHARC [3] can be modified to handle multiple (known) metrics at once. In the *flexible routing problem* [11], one must answer queries on linear combinations of a small set of metrics (typically two) known in advance. Queries in the *constrained routing problem* [23] must avoid entire classes of edges. In multi-criteria optimiza-

tion [8], one must find Pareto-optimal paths among multiple metrics. ALT [14] and CH [12] can adapt to small changes in a benign base metric without rerunning preprocessing in full. All these approaches must know the base metrics in advance, and for good performance the metrics must be few, well-behaved, and similar to one another. In practice, even seemingly small changes to the metric (such as higher U-turn costs) render some approaches impractical. In contrast, we must process metrics as they come, and assume nothing about them.

We now discuss the properties of existing point-to-point algorithms to determine how well they fit our design goals. Some of the most successful existing methods—such as reach-based routing [15], contraction hierarchies (CH) [12], SHARC [3], transit node routing [2], and hub labels [1]—rely on the strong *hierarchy* of road networks with travel times: the fastest paths between faraway regions of the graph tend to use the same major roads.

For metrics with strong hierarchies, such as travel times, CH has many of the features we want. During preprocessing, CH heuristically sorts the vertices in increasing order of importance, and *shortcuts* them in this order. (To *shortcut* v, we temporarily remove it from the graph and add arcs as necessary to preserve the distances between its neighbors.) Queries run bidirectional Dijkstra, but only follow arcs or shortcuts to more important vertices. If a metric changes only slightly, one can keep the order and recompute the shortcuts in about a minute [12]. Unfortunately, an order that works for one metric may not work for a substantially different one (e.g., travel times and distances, or a major traffic jam). Furthermore, queries are much slower on metrics with less-pronounced hierarchies [4]. More crucially, the preprocessing stage can become impractical (in terms of space and time) for bad metrics, as Section 4 will show.

In contrast, techniques based on *goal direction*, such as PCD [21], ALT [14], and arc flags [16], produce the same amount of auxiliary data for any metric. Queries are not robust, however: they can be as slow as Dijkstra for bad metrics. Even for travel times, PCD and ALT are not competitive with other methods.

A third approach is based on *graph separators* [17–19, 25]. During preprocessing, one computes a multilevel partition of the graph to create a series of interconnected overlay graphs. A query starts at the lowest (local) level and moves to higher (global) levels as it progresses. These techniques predate hierarchy-based methods, but their query times are widely regarded as uncompetitive in practice, and they have not been tested on continental-sized road networks. (The exceptions are recent extended variants [6, 22] that achieve great query times by adding many more edges during preprocessing, which is costly in time and space.) Because preprocessing and query times are essentially metric-independent, separator-based methods are the most natural fit for our problem.

3 Our Approach

We will first describe a basic algorithm, then consider several techniques to make it more practical, using experimental results to guide our design. Our code is written in C++ (with OpenMP for parallelization) and compiled with Microsoft

Visual C++ 2010. We use 4-heaps as priority queues. Experiments were run on a commodity workstation with an Intel Core-i7 920 (four cores clocked at 2.67 GHz and 6 GB of DDR3-1066 RAM) running Windows Server 2008 R2. Our standard benchmark instance is the European road network, with 18 million vertices and 42 million arcs, made available by PTV AG for the 9th DIMACS Implementation Challenge [9]. Vertex IDs and arc costs are both 32-bit integers.

We must minimize *metric customization time*, *metric-dependent space* (excluding the original graph), and *query time*, while keep metric-independent time and space reasonable. We evaluate our algorithms on 10 000 s–t queries with s and t picked uniformly at random. We focus on finding shortest path *costs*; Section 4 shows how to retrieve the actual paths. We report results for travel times and travel distances, but *by design* our algorithms work well for any metric.

Basic Algorithm. Our *metric-independent preprocessing* stage partitions the graph into connected cells with at most U (an input parameter) vertices each, with as few boundary arcs (arcs with endpoints in different cells) as possible.

The *metric customization* stage builds a graph H containing all boundary vertices (those with at least one neighbor in another cell) and boundary arcs of G. It also contains a *clique* for each cell C: for every pair (v, w) of boundary vertices in C, we create an arc (v, w) whose cost is the same as the shortest path (restricted to C) between v and w (or infinite if w is not reachable from v). We do so by running Dijkstra from each boundary vertex. Note that H is an *overlay* [24]: the distance between any two vertices in H is the same as in G.

Finally, to perform a *query* between s and t, we run a bidirectional version of Dijkstra's algorithm on the graph consisting of the union of H, C_s, and C_t. (Here C_v denotes the subgraph of G induced by the vertices in the cell containing v.)

As already mentioned, this is the basic strategy of separator-based methods. In particular, HiTi [19] uses edge-based separators and cliques to represent each cell. Unfortunately, HiTi has not been tested on large road networks; experiments were limited to small grids, and the original proof of concept does not appear to have been optimized using modern algorithm engineering techniques.

Our first improvement over HiTi and similar algorithms is to use PUNCH [5] to partition the graph. Recently developed to deal with road networks, it routinely finds solutions with half as many boundary edges (or fewer), compared to the general-purpose partitioners (such as METIS [20]) commonly used by previous algorithms. Better partitions reduce customization time and space, leading to faster queries. For our experiments, we used relatively long runs of PUNCH, taking about an hour. Our results would not change much if we used the basic version of PUNCH, which is only about 5% worse but runs in mere minutes.

We use parallelism: queries run forward and reverse searches on two CPU cores, and customization uses all four (each cell is processed independently).

Sparsification. Using full cliques in the overlay graph may seem wasteful, particularly for well-behaved metrics. At the cost of making its topology metric-dependent, we consider various techniques to reduce the overlay graph.

The first approach is *edge reduction* [24], which eliminates clique arcs that are not shortest paths. After computing all cliques, we run Dijkstra from each vertex v in H, stopping as soon as all neighbors of v (in H) are scanned. Note that these searches are usually quick, since they only visit the overlay.

A more aggressive technique is to preserve some internal cell vertices [6, 17, 25]. If $B = \{v_1, v_2, \ldots, v_k\}$ is the set of boundary vertices of a cell, let T_i be the shortest path tree (restricted to the cell) rooted at v_i, and let T_i' be the subtree of T_i consisting of the vertices with descendants in B. We take the union $C = \cup_{i=1}^{k} T_i'$ of these subtrees, and shortcut all internal vertices with two neighbors or fewer. Note that this *skeleton graph* is technically not an overlay, but it preserves distances between all *boundary* vertices, which is what we need.

Finally, we tried a lightweight *contraction* scheme. Starting from the skeleton graph, we greedily shortcut low-degree internal vertices, stopping when no such operation is possible without increasing the number of edges by more than one.

Fig. 1 (left) compares all four overlays (cliques, reduced cliques, skeleton, and CH-skeleton) on travel times and travel distances. Each plot relates the total query time and the amount of metric-independent data for different values of U (the cell size). Unsurprisingly, all overlays need more space as the number of cells increases (i.e., as U decreases). Query times, however, are minimized when the effort spent on each level is balanced, which happens for $U \approx 2^{15}$.

To analyze preprocessing times (not depicted in the plots), take $U = 2^{15}$ (with travel times) as an example. Finding full cliques takes only 40.8 s, but edge reduction (45.8 s) or building the skeleton graph (45.1 s) are almost as

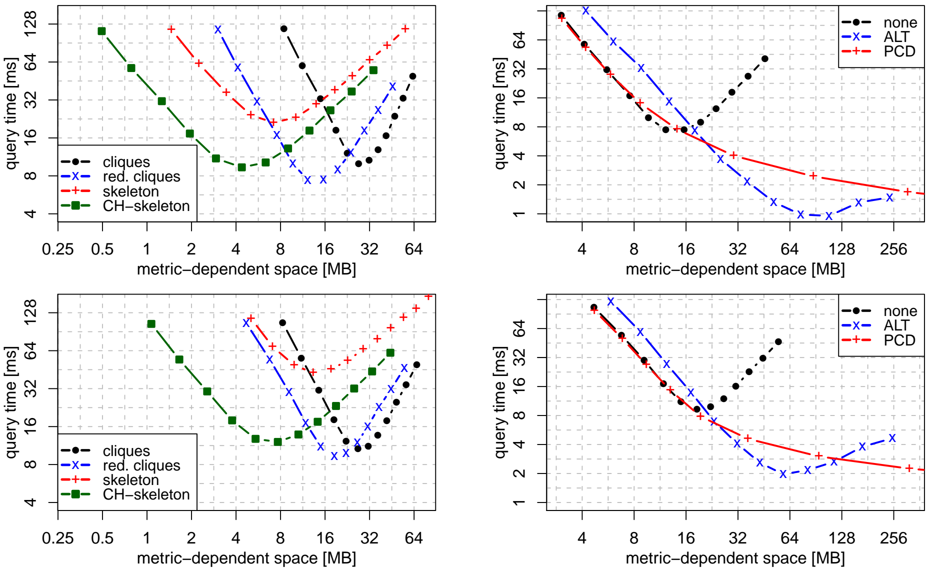

Fig. 1. Effect of sparsification (left) and goal direction (right) for travel times (top) and distances (bottom). The i-th point from the left indicates $U = 2^{20-i}$.

cheap. CH-skeleton, at 79.4 s, is significantly more expensive, but still practical. Most methods get faster as U gets smaller: full cliques take less than 5 s with $U = 256$. The exception is CH-skeleton: when U is very small, the combined size of all skeletons is quite large, and processing them takes minutes.

In terms of query times and metric-dependent space, however, CH-skeleton dominates pure skeleton graphs. Decreasing the number of edges (from 1.2M with reduced cliques to 0.8M with skeletons, for $U = 2^{15}$ with travel times) may not be enough to offset an increase in the number of vertices (from 34K to 280K), to which Dijkstra-based algorithms are more sensitive. This also explains why reduced cliques yield the fastest queries, with full cliques not far behind.

All overlays have worse performance when we switch from travel times to distances (with less pronounced hierarchies), except full cliques. Since edge reduction is relatively fast, we use reduced cliques as the default overlay.

Goal-direction. For even faster queries, we can apply more sophisticated techniques (than bidirectional Dijkstra) to search the overlay graph. While in principle any method could be used, our model restricts us to those with metric-independent preprocessing times. We tested PCD and ALT.

To use PCD (Precomputed Cluster Distances) [21] with our basic algorithm, we do the following. Let k be the number of cells found during the metric independent preprocessing ($k \approx n/U$). During metric customization, we run Dijkstra's algorithm k times on the overlay graph to compute a $k \times k$ matrix with the distances between all cells. Queries then use the matrix to guide the bidirectional search by pruning vertices that are far from the shortest path. Note that, unlike "pure" PCD, we use the overlay graph during customization and queries.

Another technique is *core ALT* (CALT) [4]. Queries start with bidirectional Dijkstra searches restricted to the source and target cells. Their boundary vertices are then used as starting points for an ALT (A* search/ landmarks/triangle inequality) query on the overlay graph. The ALT preprocessing runs Dijkstra $O(L)$ times to pick L vertices as landmarks, and stores distances between these landmarks and all vertices in the overlay. Queries use these distances and the triangle inequality to guide the search towards the goal. A complication of core-based approaches [15, 4] is the need to pick nearby overlay vertices as *proxies* for the source or target to get their distance bounds. Hence, queries use four CPU cores: two pick the proxies, while two conduct the actual bidirectional search.

Fig. 1 (right) shows the query times and the metric-dependent space consumption for the basic algorithm, CALT (with 32 *avoid* landmarks [15]), and PCD, with reduced cliques as overlay graphs. With some increase in space, both goal-direction techniques yield significantly faster queries (around one millisecond). PCD, however, needs much smaller cells, and thus more space and customization time (about a minute for $U = 2^{14}$) than ALT (less than 3 s). Both methods are more effective for travel times than travel distances.

Multiple Levels. To accelerate queries, we can use multiple levels of overlay graphs, a common technique for partition-based approaches, including HiTi [19].

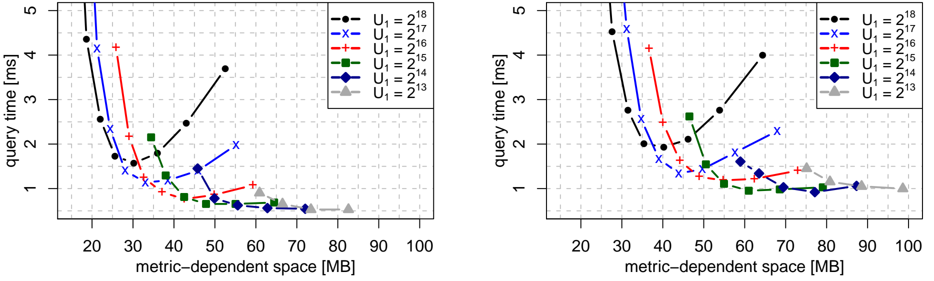

Fig. 2. Performance of 2-level CALT with travel times (left) and distances (right). For each line, U_1 is fixed and U_0 varies; the i-th point from the right indicates $U_0 = 2^{7+i}$.

We need *nested partitions* of G, in which every boundary edge at level i is also a boundary edge at level $i-1$, for $i > 1$. The level-0 partition is the original graph, with each vertex as a cell. For the i-th level partition, we create a graph H_i as before: it includes all boundary arcs, plus an overlay linking the boundary vertices within a cell. Note that H_i can be computed using only H_{i-1}. We use PUNCH to create multilevel partitions, in top-down fashion.

An s–t query runs bidirectional Dijkstra on a restricted graph G_{st}. An arc (v, w) from H_i will be in G_{st} if both v and w are in the same cell as s or t at level $i + 1$. Goal-direction can still be used on the top level.

Fig. 2 shows the performance of the multilevel algorithm with two overlay levels (with reduced cliques) and ALT on the top level. We report query times and metric-dependent space for multiple values of U_0 and U_1, the maximum cell sizes on the bottom and top levels. A comparison with Fig. 1 reveals that using two levels enables much faster queries for the same space. For travel times, a query takes 1 ms with about 40 MB (with $U_0 = 2^{11}$ and $U_1 = 2^{16}$). Here it takes 16 s to compute the bottom overlay, 5 s to compute the top overlay, and only 0.5 s to process landmarks. With 60 MB, queries take as little as 0.5 ms.

Streamlined Implementation. Although sparsification techniques save space and goal direction accelerates queries, the improvements are moderate and come at the expense of preprocessing time, implementation complexity, and metric-independence (the overlay topology is only metric-independent with full cliques). Furthermore, the time and space requirements of the simple clique implementation can be improved by representing each cell of the partition as a *matrix*, making the performance difference even smaller. The matrix contains the 32-bit distances among its entry and exit vertices (these are the vertices with at least one incoming or outgoing boundary arc, respectively; most boundary vertices are both). We also need arrays to associate rows (and columns) with the corresponding vertex IDs, but these are small and shared by all metrics.

We thus created a matrix-based *streamlined implementation* that is about twice as fast as the adjacency-based clique implementation. It does not use edge reduction, since it no longer saves space, slows down customization, and its effectiveness depends on the metric. (Skipping infinite matrix entries would make

Table 1. Performance of various algorithms for travel times and distances.

	travel times				travel distances			
	CUSTOMIZING		QUERIES		CUSTOMIZING		QUERIES	
algorithm [cell sizes]	time [s]	space [MB]	vertex scans	time [ms]	time [s]	space [MB]	vertex scans	time [ms]
CALT [2^{11}:2^{16}]	21.3	37.1	5292	0.92	17.2	48.9	5739	1.26
MLD-1 [2^{14}]	4.9	10.1	45420	5.81	4.8	10.1	47417	6.12
MLD-2 [2^{12}:2^{18}]	5.0	18.8	12683	1.82	5.0	18.8	13071	1.83
MLD-3 [2^{10}:2^{15}:2^{20}]	5.2	32.7	6099	0.91	5.1	32.7	6344	0.98
MLD-4 [2^{8}:2^{12}:2^{16}:2^{20}]	4.7	59.5	3828	0.72	4.7	59.5	4033	0.79
CH economical	178.4	151.3	383	0.12	1256.9	182.5	1382	1.33
CH generous	355.6	122.8	376	0.10	1987.4	165.8	1354	1.29

queries only slightly faster.) Similarly, we excluded CALT from the streamlined representation, since its queries are complicated and have high variance [4].

Customization times are typically dominated by building the overlay of the lowest level, since it works on the underlying graph directly (higher levels work on the much smaller cliques of the level below). As we have observed, smaller cells tend to lead to faster preprocessing. Therefore, as an optimization, the streamlined implementation includes a *phantom level* (with $U = 32$) to accelerate customization, but throws it away for queries, keeping space usage unaffected. For MLD-1 and MLD-2, we use a second phantom level with $U = 256$ as well.

Table 1 compares our streamlined multilevel implementation (called MLD, with up to 4 levels) with the original 2-level implementation of CALT. For each algorithm, we report the cell size bounds in each level. (Because CALT accelerates the top level, it uses different cell sizes than MLD-2.) We also consider two versions of CH: the first (*economical*) minimizes preprocessing times, and the second (*generous*) the number of shortcuts. For CH, we report the total space required to store the shortcuts (8 bytes per arc, excluding the original graph). For all algorithms, preprocessing uses four cores and queries use at least two.

We do not permute vertices after CH preprocessing (as is customary to improve query locality), since this prevents different metrics from sharing the same graph. Even so, with travel times, CH queries are one order of magnitude faster than our algorithm. For travel distances, MLD-3 and MLD-4 are faster than CH, but only slightly. For practical purposes, all variants have fast enough queries.

The main attraction of our approach is efficient metric customization. We require much less space: for example, MLD-2 needs about 20 MB, which is less than 5% of the original graph and an order of magnitude less than CH. Most notably, customization times are small. We need only 5 seconds to deal with a new metric, which is fast enough to enable personalized driving directions. This is two orders of magnitude faster than CH, even for a well-behaved metric. Phantom levels help here: without them, MLD-1 would need about 20 s.

Note that CH customization can be faster if the processing order is fixed in advance [12]. The economical variant can rebuild the hierarchy (sequentially) in 54 s for travel times and 178 s for distances (still slower than our method).

Unfortunately, using the order for one metric to rebuild another is only efficient if they are very similar [11]. Also note that one can save space by storing only the upper part of the hierarchy [7], at the expense of query times.

Table 1 shows that we can easily deal with real-time traffic: if all edge costs change (due to a traffic update), we can handle new queries after only 5 seconds. We can also support *local updates* quite efficiently. If a single edge cost changes, we must recompute at most one cell on each level, and MLD-4 takes less than a millisecond to do so. This is another reason for not using edge reduction or CALT: with either technique, changes in one cell may propagate beyond it.

4 Turns

So far, we have considered a simplified (but standard [7]) representation of road networks, with each intersection corresponding to a single vertex. This is not very realistic, since it does not account for turn costs (or restrictions, a special case). Of course, any algorithm can handle turns simply by working on an expanded graph. A traditional [7] representation is *arc-based*: each vertex represents one *exit point* of an intersection, and each arc is a road segment followed by a turn.

This is wasteful. We propose a *compact representation* in which each intersection becomes a single vertex with some associated information. If a vertex u has p incoming and q outgoing arcs, we associate a $p \times q$ *turn table* T_u to it, where $T_u[i, j]$ represents the turn from the ith incoming arc into the jth outgoing arc.[3] In addition, we store with each arc (v, w) its *tail order* (its position among v's outgoing arcs) and its *head order* (its position among w's incoming arcs). These orders may be arbitrary. Since degrees are small, 4 bits for each suffice.

In practice, many vertices share the same turn table. The total number of such *intersection types* is modest—in the thousands rather than millions. For example, many degree-four vertices in the United States have four-way stop signs. Each distinct turn table is thus stored only once, and each vertex keeps a pointer to the appropriate type, with little overhead.

Dijkstra's algorithm, however, becomes more complicated. In particular, it may now visit each vertex (intersection) multiple times, once for each entry point. It essentially simulates an execution on the arc-based expanded representation, which increases its running time on Europe from 3 s to about 12 s. With a *stalling* technique, we can reduce the time to around 7 s. When scanning one entry point of an intersection, we can set bounds for its other entry points, which are not scanned unless their own distance labels are smaller than the bounds. These bounds depend on the turn table, and can be computed during customization.

To support the compact representation, MLD needs two minor changes. First, it uses turn-aware Dijkstra on the lowest level (but not on higher ones). Second, matrices in each cell now represent paths between incoming and outgoing *boundary arcs* (and not boundary vertices, as before). The difference is subtle. With turns, the distance from a boundary vertex v to an exit point depends on whether

[3] In our customizable setting, each entry should represent just a turn type (such as "left turn with stop sign"), since its cost may vary with different metrics.

Table 2. Performance of various algorithms on Europe with varying U-turn costs.

	U-turn: 1 s				U-turn: 100 s			
	CUSTOMIZING		QUERIES		CUSTOMIZING		QUERIES	
algorithm	time [s]	space [MB]	vertex scans	time [ms]	time [s]	space [MB]	vertex scans	time [ms]
MLD-1 [2^{14}]	5.9	10.5	44832	9.96	7.5	10.5	62746	12.43
MLD-2 [2^{12}:2^{18}]	6.3	19.2	12413	3.07	8.4	19.2	16849	3.55
MLD-3 [2^{10}:2^{15}:2^{20}]	7.3	33.5	5812	1.56	9.2	33.5	6896	1.88
MLD-4 [2^{8}:2^{12}:2^{16}:2^{20}]	5.8	61.7	3556	1.18	7.5	61.7	3813	1.28
CH expanded	3407.4	880.6	550	0.18	5799.2	931.1	597	0.21
CH compact	846.0	132.5	905	0.19	23774.8	304.0	5585	2.11

we enter the cell from arc (u, v) or arc (w, v), so each arc needs its own entry in the matrix. Since most boundary vertices have only one incoming (and outgoing) boundary arc, the matrices are only slightly larger.

We are not aware of publicly-available realistic turn data, so we augment our standard benchmark instance. For every vertex v, we add a turn between each incoming and each outgoing arc. A turn from (u, v) to (v, w) is either a *U-turn* (if $u = w$) or a *standard turn* (if $u \neq w$), and each of these two types has a cost. We have not tried to further distinguish between turn types, since any automated method would not reflect real-life turns. However, adding U-turn costs is enough to reproduce the key issue we found on realistic (proprietary) data.

Table 2 compares some algorithms on Europe augmented with turns. We consider two metrics, with U-turn costs set to 1 s or 100 s. The metrics are otherwise identical: arc costs represent travel times and standard turns have zero cost. We tested four variants of MLD (with one to four levels) and two versions of CH (generous): *CH expanded* is the standard algorithm run on the arc-based expanded graph, while *CH compact* is modified to run on the compact representation. Column *vertex scans* counts the number of heap extractions.

Small U-turn costs do not change the shortest path structure of the graph much. Indeed, CH compact still works quite well: preprocessing is only three times slower (than reported in Table 1), the number of shortcuts created is about the same, and queries take marginally longer. Using higher U-turn costs (as in a system that avoids U-turns), however, makes preprocessing much less practical. Customization takes more than 6 hours, and space more than doubles. Intuitively, nontrivial U-turn costs are harder to handle because they increase the importance of certain vertices; for example, driving around the block may become a shortest path. Query times also increase, but are still practical. (Note that recent independent work [13] shows that additional tuning can make compact CH somewhat more resilient: changing U-turn costs from zero to 100 s increases customization time by a factor of only two. Unfortunately, forbidding U-turns altogether still slows it down by an extra factor of 6.)

With the expanded representation, CH preprocessing is much costlier when U-turns are cheap (since it runs on a larger graph), but is much less sensitive to an increase in the U-turn cost; queries are much faster as well. The difference

in behavior is justified. While the compact representation forces CH to assign the same "importance" (order) to different entry points of an intersection, the expanded representation lets it separate them appropriately.

MLD is much less sensitive to turn costs. Compared to Table 1, we observe that preprocessing space is essentially the same (as expected). Preprocessing and query times increase slightly, mainly due to the lower level: high U-turn costs decrease the effectiveness of the stalling technique on the turn-enhanced graph.

In the most realistic setting, with nontrivial U-turn costs, customization takes less than 10 seconds on our commodity workstation. This is more than enough to handle frequent traffic updates, for example. If even more speed is required, one could simply use more cores: speedups are almost perfect. On a server with two 6-core Xeon 5680 CPUs running at 3.33 GHz, MLD-4 takes only 2.4 seconds, which is faster than just running sequential Dijkstra on this input.

Path Unpacking. So far, we have reported the time to compute only the distance between two points. Following the parent pointers of the meeting vertex of forward and backward searches, we may obtain a path containing shortcuts. To unpack a level-i shortcut, we run bidirectional Dijkstra on level $i-1$ (and recurse as necessary). Using all 4 cores, unpacking less than doubles query times, with no additional customization space. (In contrast, standard CH unpacking stores the "middle" vertex of every shortcut, increasing the metric-dependent space by 50%.) For even faster unpacking, one can store a bit with each arc at level i indicating whether it appears in a shortcut at level $i+1$. This makes unpacking 4 times faster for MLD-2, but has little effect on MLD-3 and MLD-4.

5 Conclusion

Recent advances in graph partitioning motivated us to reexamine the separator-based multilevel approach to the shortest path problem. With careful engineering, we drastically improved query speedups relative to Dijkstra from less than 60 [17] to more than 3000. With turn costs, the speedup increases even more, to 7000. This makes real-time queries possible. Furthermore, by explicitly separating metric customization from graph partitioning, we enable new metrics to be processed in a few seconds. The result is a flexible and practical solution to many real-life variants of the problem. It should be straightforward to adapt it to augmented scenarios, such as mobile or time-dependent implementations. (In particular, a unidirectional version of MLD is also practical.) Since partitions have a direct effect on performance, we would like to improve them further, perhaps by explicitly taking the size of the overlay graph into account.

Acknowledgements. We thank Ittai Abraham and Ilya Razenshteyn for their valuable input, and Christian Vetter for sharing his CH results with us.

References

1. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. *SEA'11*. Springer, LNCS, 2011.

2. H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. *ALENEX'07*, pp. 46–59. SIAM, 2007.
3. R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM JEA* 14(2.4):1–29, 2009.
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 JEA* 15(2.3):1–31, 2010.
5. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. To appear in *IPDPS'11*. IEEE, 2011.
6. D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Routing. In Demetrescu et al. [9], pp. 73–92.
7. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. *Algorithmics of Large and Complex Networks*, LNCS 5515, 2009.
8. D. Delling and D. Wagner. Pareto Paths with SHARC. *SEA'09*, pp. 125–136. Springer, LNCS 5526, 2009.
9. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, eds. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge.* DIMACS Book 74. AMS, 2009.
10. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* 1:269–271, 1959.
11. R. Geisberger, M. Kobitzsch, and P. Sanders. Route Planning with Flexible Objective Functions. *ALENEX'10*, pp. 124–137. SIAM, 2010.
12. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. *WEA'08*, pp. 319–333. Springer, LNCS 5038, 2008.
13. R. Geisberger and C. Vetter. Efficient Routing in Road Networks with Turn Costs. *SEA'11*. Springer, LNCS, 2011.
14. A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A^* Search Meets Graph Theory. *SODA'05*, pp. 156–165, 2005.
15. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Demetrescu et al. [9], pp. 93–139.
16. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [9], pp. 41–72.
17. M. Holzer, F. Schulz, and D. Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM JEA* 13(2.5):1–26, 2008.
18. Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Effective Graph Clustering for Path Queries in Digital Maps. *CIKM'96*, pp. 215–222. ACM Press, 1996.
19. S. Jung and S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE TKDE* 14(5):1029–1046, 2002.
20. G. Karypis and G. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. on Scientific Comp.* 20(1):359–392, 1999.
21. J. Maue, P. Sanders, and D. Matijevic. Goal-Directed Shortest-Path Queries Using Precomputed Cluster Distances. *ACM JEA* 14:3.2:1–3.2:27, 2009.
22. L. F. Muller and M. Zachariasen. Fast and Compact Oracles for Approximate Distances in Planar Graphs. *ESA'07*, pp. 657–668. Springer, LNCS 4698, 2007.
23. M. Rice and V. J. Tsotras. Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. *Proc. VLDB Endowment*, vol. 4, no. 2, 2010.
24. F. Schulz, D. Wagner, and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM JEA* 5(12):1–23, 2000.
25. F. Schulz, D. Wagner, and C. Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. *ALENEX'02*. Springer, LNCS 2409, 2002.