Faster Customization of Road Networks

Daniel Delling and Renato F. Werneck

Microsoft Research Silicon Valley
{dadellin,renatow}@microsoft.com

Abstract. A wide variety of algorithms can answer exact shortest-path queries in real time on continental road networks, but they typically require significant preprocessing effort. Recently, the customizable route planning (CRP) approach has reduced the time to process a new cost function to a fraction of a minute. We reduce customization time even further, by an order of magnitude. This makes it worthwhile even when a single query is to be run, enabling a host of new applications.

1 Introduction

Computing driving directions in road networks is a fundamental problem of practical importance. Although it can be solved in almost-linear time by Dijkstra's shortest-path algorithm [14], this is not fast enough for interactive queries on continental road networks. This has motivated a wide variety of recent algorithms [2, 5, 7, 8, 18, 21, 22] that rely on a (relatively slow) preprocessing stage to enable much faster queries. Different algorithms offer distinct tradeoffs between preprocessing time, space requirements, and query times. Directly or indirectly, they exploit the fact that road networks have a strong hierarchy when minimizing driving times in free-flow traffic [3]. When other cost functions are optimized, however, their performance can be much worse. Moreover, these approaches should only be used when there are enough queries to amortize the preprocessing cost. This is not true in many practical situations, such as when the cost function changes very frequently (to consider traffic, for example), or when users can choose from several (possibly uncommon) cost functions.

In contrast, the recently proposed *customizable route planning* (CRP) algorithm [11] is lightweight and robust to changes in the cost function (metric). It works in *three* stages. The first, *metric-independent preprocessing*, uses graph partitioning to define the topology of a multilevel overlay graph [23], which is the same regardless of the cost function. The second stage, *customization*, uses the metric to compute the actual costs of the overlay arcs. Finally, the *query stage* uses the output of the first two stages to compute shortest paths in real time (milliseconds). The first stage may take a few minutes (or even hours), but only needs to be run (or updated) when new road segments are built. Metric changes (which are much more frequent) require running only customization, which takes less than a minute. Since it does not rely on strong hierarchies, CRP is robust to metric changes. Unlike most other methods, it can also handle turn costs (and

restrictions) quite naturally, with little effect on performance and space usage. It is thus ideal for a real-world routing engine, and is indeed in use by Bing Maps.

This paper shows how to make customization even faster, enabling a wide range of new applications. To compute overlay arc costs, we propose replacing Dijkstra's algorithm by other approaches, such as contraction and Bellman-Ford. Although they may even *increase* the number of operations (such as arc scans) performed, careful application of algorithm engineering techniques leads to better locality and enables parallelism at instruction and core levels. Remarkably, our new customization routine takes less time (sequentially) than running Dijkstra's algorithm once. Unlike in any other method, *a single query* is enough to amortize the customization cost, making it ideal for highly dynamic applications.

2 Preliminaries

Basics. A road network is usually modeled as a directed graph $G = (V, A)$ with $n = |V|$ vertices and $m = |A|$ arcs. Each vertex $v \in V$ represents an intersection, and each arc $(v, w) \in A$ a road segment. A *metric* (or *cost function*) $\ell : A \to \mathcal{N}$ maps each arc to a positive *length* (or *cost*). In the *point-to-point shortest path problem*, our goal is to compute the minimum-length path in the graph between a *source* s and a *target* t. It can be solved by Dijkstra's algorithm [14], which processes vertices in increasing order of distance from s and stops when t is processed. It runs in essentially linear time in theory and in practice [20].

In this paper, we focus on a more realistic model for road networks, which takes turn costs (and restrictions) into account. We think of each vertex v as having one *entry point* for each of its incoming arcs, and one *exit point* for each outgoing arc. We extend the concept of *metric* by also associating a *turn table* T_v to each vertex v; $T_v[i, j]$ specifies the cost of turning from the i-th incoming arc to the j-th outgoing arc. (The order around each vertex is arbitrary but fixed.)

We can run Dijkstra's algorithm on this *turn-aware* graph by associating distance labels to entry points instead of vertices [11, 19]. An alternative approach (often used in practice) is to operate on an *expanded graph* G', where each vertex corresponds to an entry point in G, and each arc represents the concatenation of a turn and an arc in G. This allows standard (non-turn-aware) algorithms to be used, but roughly triples the graph size. In contrast, the turn-aware representation is almost as compact as the simplified one (with no turns at all), since common turn tables can be shared among vertices.

Customizable Route Planning. The *customizable route planning* (CRP) [11] algorithm is a speedup technique that computes shortest paths in three stages: (metric-independent) preprocessing, customization, and queries.

The *preprocessing* stage defines a multilevel overlay [23] of the graph and builds auxiliary data structures. A *partition* of V is a family $\mathcal{C} = \{C_0, \ldots, C_k\}$ of sets $C_i \subseteq V$ such that each $v \in V$ is in exactly one *cell* C_i. A *multilevel partition* of V of L *levels* is a family of partitions $\{\mathcal{C}^1, \ldots, \mathcal{C}^L\}$, where l denotes the level of a partition $\mathcal{C}^l$. Let U^l be the size of the biggest cell on level l. We

deal only with *nested* multilevel partitions: for each $l \leq L$ and each cell $C_i^l \in \mathcal{C}^l$, there exists a cell $C_j^{l+1} \in \mathcal{C}^{l+1}$ with $C_i^l \subseteq C_j^{l+1}$; we say C_i^l is a *subcell* of C_j^{l+1}. (We assume $\mathcal{C}^0$ consists of singletons and $\mathcal{C}^{L+1} = \{V\}$.) A *boundary* arc on level l is an arc with endpoints in different level-l cells; its endpoints are *boundary vertices*. A boundary arc on level l is also a boundary arc on all levels below.

The preprocessing phase of CRP uses PUNCH [12], a graph-partitioning heuristic tailored to road networks, to create a multilevel partition. Given an unweighted graph and a bound U, PUNCH splits the graph into cells with at most U vertices while minimizing the number of arcs between cells. To find a multilevel partition, one calls PUNCH repeatedly in top-down fashion: after partitioning the full graph, one partitions each subcell independently.

Besides partitioning, the CRP preprocessing phase sets up the topology of the overlay graph. Consider a cell C on any level, as in Fig. 1. Every incoming boundary arc (u, v) (i.e, with $u \notin C$ and $v \in C$) defines an *entry point* for C, and every outgoing arc (v, w) (with $v \in C$ and $w \notin C$) defines an *exit point* for C. The *overlay* for cell C is simply a complete bipartite graph with directed *shortcuts* (gray lines in the figure) between each entry point (filled circle) and each exit point (hollow circle) of C. The overlay of a level l consists of the union of all cell overlays, together with all boundary arcs (black arrows) on this level.

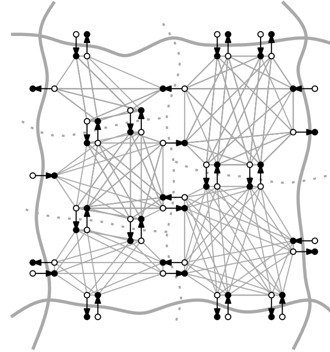

Fig. 1. Overlays of five cells.

During the *customization stage*, CRP computes the *lengths* of the shortcuts on the overlay. A shortcut (p, q) within a cell C represents the shortest path (restricted to C) between p and q. Lengths are computed bottom-up, starting at level one. To process a cell, CRP runs Dijkstra's algorithm from each entry point p to find the lengths of all shortcuts starting at p. Processing level-one cells requires running Dijkstra on the original graph, taking turn costs into account. Higher-level cells can use the overlay graph for the level below, which is much smaller and has no explicit turns (turn costs are incorporated into shortcuts).

An s–t CRP *query* runs bidirectional Dijkstra on the overlay graph, but only entering cells containing either s or t. To retrieve the arcs corresponding to each shortcut in the resulting path, one runs bidirectional Dijkstra within the appropriate cell. We accelerate unpacking using an LRU cache to store the unpacking information of a level-i shortcut as a sequence of level-$(i-1)$ shortcuts.

3 Our Approach

Separating preprocessing from metric customization allows CRP to incorporate a new cost function on a continental road network in less than 30 seconds [11] (sequentially) on a modern server. This is enough for real-time traffic, but too slow to enable on-line personalized cost functions. Accelerating customization even further requires speeding up its basic operation: computing the lengths of

the shortcuts within each cell. To do so, we propose different strategies to replace Dijkstra's algorithm.

One strategy is *contraction*, the basic building block of the *contraction hierarchies* (CH) algorithm [18] and an element of many other speedup techniques [2, 5, 7, 8, 21]. Instead of computing shortest paths explicitly, we eliminate internal vertices from a cell one by one, adding new arcs as needed to preserve distances; the arcs that eventually remain are the desired shortcuts. For efficiency, not only do we precompute the order in which vertices are contracted, but also abstract away the graph itself. During customization, we simply simulate the actual contraction by following a (precomputed) series of *instructions* describing the basic operations (memory reads and writes) the contraction routine would perform.

Contraction works well on the first overlay level, since it operates directly on the underlying graph, which is sparse. Density quickly increases during contraction, however, making it expensive as cell sizes increase. On higher levels, we compute shortest paths explicitly (as before), but make each computation more efficient. We replace Dijkstra with lightweight algorithms that work better on small graphs, and apply techniques to reduce the size of the search graph.

The next two sections describe each strategy in more detail, including how they can be engineered for better performance in practice.

4 Contraction

The *contraction* approach is based on the *shortcut* operation [18]. To shortcut a vertex v, one removes it from the graph and adds new arcs as needed to preserve shortest paths. For each incoming arc (u, v) and outgoing arc (v, w), one creates a *shortcut* arc (u, w) with $\ell(u, w) = \ell(u, v) + \ell(v, w)$. A shortcut is only added if it represents the only shortest path between its endpoints in the remaining graph (without v), which can be tested by running a *witness search* (local Dijkstra) between its endpoints. CH [18] uses contraction as follows. During preprocessing, it heuristically sorts all vertices in increasing order of importance and shortcuts them in this order; the order and the shortcuts are then used to speed up queries.

We propose using contraction during customization. To process a cell, we can simply contract its internal vertices while preserving its boundary vertices. The arcs (shortcuts) in the final graph are exactly the ones we want. To deal with turn costs appropriately, we run contraction on the expanded graph.

The performance of contraction strongly depends on the cost function. With travel times in free-flow traffic (the most common case), it works very well. Even for continental instances, sparsity is preserved during the contraction process, and the number of arcs less than doubles [18]. Unfortunately, other metrics often need more shortcuts, which leads to denser graphs and makes finding the contraction order much more expensive. Even if a good order is given, simply performing the contraction can still be quite costly [18].

Within the CRP framework, we can deal with these issues by exploiting the separation between metric-independent preprocessing and customization. During preprocessing, we compute a unique contraction order to be used by all

metrics. Unlike previous approaches [18], to ensure this order works well even in the worst case, we simply assume that every potential shortcut will be added. Accordingly, we do not perform witness searches during customization. For maximum efficiency, we precompute a sequence of *microinstructions* to describe the entire contraction process in terms of basic operations. We detail each of these improvements next.

Contraction Order. Computing a contraction order that minimizes the number of shortcuts added is NP-hard [6]. In practice, one uses on-line heuristics that pick the next vertex to contract based on a priority function that depends on local properties of the graph [18]. A typical criterion is the difference between the number of arcs added and removed if a vertex v were contracted. We tested similar greedy priority functions to evaluate each vertex v, taking into account parameters such as the number $ia(v)$ of incoming arcs, the number $oa(v)$ of outgoing arcs, and the number $sc(v)$ of shortcuts created or updated if v is contracted (this may be less than $ia(v) \cdot oa(v)$, since self-loops are never needed). We found that picking vertices v that minimize $h(v) = 100sc(v) - ia(v) - oa(v)$ works well. This essentially minimizes the number of shortcuts added, using the current degree as a tiebreaker (the precise coefficients are not important).

This approach gives reasonable orders, but one can do even better by taking the graph topology into account. There exist natural orders that lead to a provably small number of shortcuts for graphs with small separators [10, 25] or treewidth [10]. It suffices to find a small separator for the entire graph, recursively contract the two resulting components, then contract the separating vertices themselves. For graphs with $O(\sqrt{n})$-separators (such as planar graphs), such nested dissection leads to $O(n \log n)$ shortcuts. Although real-world road networks are far from planar, they have even smaller separators [12].

This suggests using partitions to guide the contraction order. We create additional *guidance levels* during the preprocessing step, extending our standard CRP multilevel partition downward (to even smaller cells). We subdivide each level-1 cell (of maximum size U) into nested subcells of maximum size U/σ^i, for $i = 1, 2, \ldots$ (until cells become too small). Here $\sigma > 1$ is the *guidance step.* For each internal vertex v in a level-1 cell, let $g(v)$ be the smallest i such that v is a boundary vertex on the guidance level with cell size U/σ^i. We use the same contraction order as before, but delay vertices according to $g(\cdot)$. If $g(v) > g(w)$, v is contracted before w; within each guidance level, we use $h(v)$.

Microinstructions. While the contraction order is determined during the metric-independent phase of CRP, we can only *execute* the contraction (follow the order) during customization, once we know the arc lengths. Even with the order given, this execution is expensive [18]. To contract v, we must retrieve the costs (and endpoints) of its incident arcs, then process each potential shortcut (u, w) by either inserting it or updating its current value. This requires data structures supporting arc insertions and deletions, and even checking if a shortcut already exists gets costlier as degrees increase. Each fundamental operation, however, is rather simple: we read the costs of two arcs, add them up, compare the result with

the cost of a third arc, and update it if needed. The entire contraction routine can therefore be fully specified by a sequence of triples (a, b, c). Each element in the triple is a memory position holding an arc (or shortcut) length. We must read the values in a and b and write the sum to c if there is an improvement.

Since the sequence of operations is the same for any cost function, we use the metric-independent preprocessing stage to set up, for each cell, an *instruction array* describing the contraction as a list of triples. Each element of a triple represents an offset in a separate *memory array*, which stores the costs of all arcs (temporary or otherwise) touched during the contraction. The preprocessing stage outputs the entire instruction array as well as the size of the memory array.

During customization, entries in the memory array representing input arcs (or shortcuts) are initialized with their costs; the remaining entries (new shortcuts) are set to ∞. We then execute the instructions one by one, and eventually copy the output values (lengths of shortcuts from entry to exit points in the cell) to the overlay graph. With this approach, the graph itself is abstracted away during customization. We do not keep track of arc endpoints, and there is no notion of vertices at all. The code just manipulates numbers (which happen to represent arc lengths). This is cheaper (and simpler) than operating on an actual graph.

Although the space required by the instruction array is metric-independent (shared by all cost functions), it can be quite large. We can keep it manageable by representing each triple with as few bits as necessary to address the memory array. In addition, we use a single *macroinstruction* to represent the contraction of a vertex v whenever the resulting number of writes exceeds an *unrolling threshold* τ. This instruction explicitly lists the addresses of v's c_{in} incoming and c_{out} outgoing arcs, followed by the corresponding $c_{in} \cdot c_{out}$ write positions. The customization phase must explicitly loop over all incoming and outgoing positions, which is slightly slower than reading tuples but saves space.

5 Graph Searches

Although contraction could be used to process the entire hierarchy, it is not as effective at higher levels as it is at level-one cells, since the graphs within each higher-level cell are much denser. In such cases, it is cheaper to actually run graph searches. This section therefore proposes search-based techniques to accelerate higher levels of the hierarchy. Each leads to improvements on its own, and they can be combined in the final algorithm.

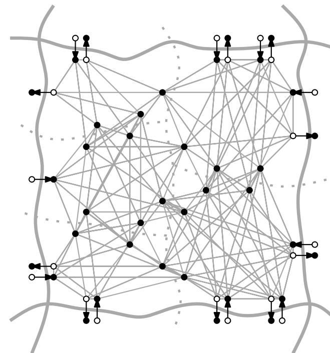

Fig. 2. Pruned overlay.

Pruning the Search Graph. To process a cell C, we must compute the distances between its entry and exit points. As shown in Fig. 1, the graph G_C on which we operate within C is the union of subcell overlays (complete bipartite graphs) with some boundary arcs between them. Instead of searching

G_C directly, we first contract its internal exit points (see Fig. 2). Since each such vertex has out-degree one (its outgoing arc is a boundary arc within C), this reduces the number of vertices and edges in the search graph. Note that C's own exit points must be preserved (they are the targets of our searches), but they do not need to be scanned (they have no outgoing arcs).

Improving Locality. Conceptually, to process a cell C we could operate on the full overlay graph, but restricting the searches to vertices inside C. For efficiency, we actually copy the relevant subgraph to a separate memory location, run our searches on it, then copy the results back. This simplifies the searches (there are no special cases), allows us to use sequential local IDs, and improves locality.

Alternative Algorithms. We can further accelerate customization by replacing Dijkstra's algorithm with Bellman-Ford [9, 16]. It starts by setting the distance label of the source to 0, and all others to ∞. Each round then scans each vertex once, updating the distance label of its neighbors appropriately. For better performance, we only *active* vertices (i.e., those whose distance improved since the previous round) and stop when there is no active vertex left. While Bellman-Ford cannot scan fewer vertices than Dijkstra, its simplicity and better locality make it competitive. The number of rounds is bounded by the maximum number of arcs on any shortest path, which is small for reasonable metrics but linear in the worst case. One could therefore switch to Dijkstra's algorithm whenever the number of Bellman-Ford rounds reaches a given (constant) threshold.

For completeness, we also tested the Floyd-Warshall algorithm [15]. It computes shortest paths among *all* vertices in the graph, and we just extract the relevant distances. Its running time is cubic, but with its tight inner loop and good locality, it could be competitive with Bellman-Ford on denser graphs.

Multiple-source executions. Multiple runs of Dijkstra's algorithm (from different sources) can be accelerated if combined into a single execution [22, 26]. We apply this idea to Bellman-Ford. Let k be the number of simultaneous executions, from sources $s_1, \ldots, s_k$. For each vertex v, we keep k distance labels: $d_1(v), \ldots, d_k(v)$. All $d_i(s_i)$ values are initialized to zero (each s_i is the source of its own search), and all remaining $d_i(\cdot)$ values to ∞. All k sources s_i are initially marked as active. When Bellman-Ford scans an arc (v, w), we try to update all k distance labels of w at once: for each i, we set $d_i(w) \leftarrow \min\{d_i(w), d_i(v) + \ell(v, w)\}$. If any such distance label actually improves, we mark w as active. This simultaneous execution needs as many rounds as the worst of the k sources, but, by storing the k distances associated with a vertex contiguously in memory, locality is much better. In addition, it enables instruction-level parallelism [26], as discussed next.

Parallelism. Modern CPUs have extended instruction sets with SIMD (single instruction, multiple data) operations, which work on several pieces of data at once. In particular, the SSE instructions available in x86 CPUs can manipulate special 128-bit registers, allowing basic operations (such as additions and comparisons) on four 32-bit words in parallel.

Consider the simultaneous execution of Bellman-Ford from $k = 4$ sources, as above. When scanning v, we first store v's four distance labels in one SSE register. To process an arc (v, w), we store four copies of $\ell(v, w)$ into another register and use a single SSE instruction to add both registers. With an SSE comparison, we check if these tentative distances are smaller than the current distance labels for w (themselves loaded into an SSE register). If so, we take the minimum of both registers (in a single instruction) and mark w as active.

In addition to using SIMD instructions, we can use core-level parallelism by assigning cells to distinct cores. (We also do this for level-1 cells with microinstructions.) In addition, we parallelize the highest overlay levels (where there are few cells per core) by further splitting the sources in each cell into sets of similar size, and allocating them to separate cores (each accessing the entire cell).

6 Experiments

We implemented our algorithm in C++ (using OpenMP for parallelization) and compiled it using Microsoft Visual Studio 2010. Our test machine runs Windows Server 2008 R2 and has 96 GiB of DDR3-1333 RAM and two 6-core Intel Xeon X5680 3.33 GHz CPUs, each with 6×64 KB of L1, 6×256 KB of L2, and 12 MB of shared L3 cache. Unless otherwise mentioned, we run our experiments on a benchmark instance representing the road network of (Western) Europe, made available by PTV AG for the 9th DIMACS Implementation Challenge [13]. The original instance has $n = 18 \cdot 10^6$ vertices, $m = 42 \cdot 10^6$ arcs, and travel times as the cost function. Following Delling et al. [11], we augment it by setting turn costs to 100 s for U-turns (and zero otherwise).

We first evaluate the effectiveness of microinstructions. Each point in Fig. 3 represents the total (sequential) customization time up to a certain cell size, using only microinstructions. Each curve reflects a different guidance step in the contraction order; smaller steps mean heavier use of partition information. Microinstructions use 32-bit addresses, and the unrolling threshold τ is 10.

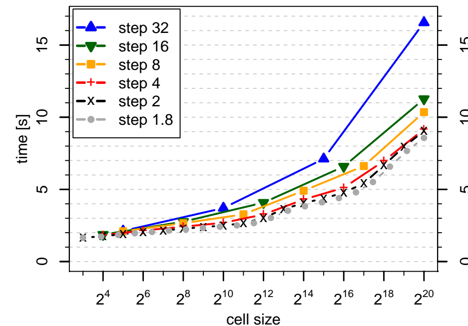

Fig. 3. Customization time up to given cell sizes for various guidance steps.

It takes less than 2 s to run the microinstructions on the entire graph if the maximum cell size is 16, and less than 3 s for cells of size 256. Significantly larger cells are much more costly. As expected, smaller guidance steps lead to better contraction orders (smaller instruction arrays), but the effect is not overwhelming: step 4 is about as fast as step 1.8, which is roughly equivalent to nested dissection. Given these results, for the remainder of this paper we only use microinstructions to customize cells of size up to 256 (using 128, 64, 32, 16 and 8 as guidance levels), with 16 (in-

Table 1. Time (in milliseconds) spent on each overlay level for different algorithms. The total time includes 1.99 s to process the lowest overlay level (2^8) with microinstructions.

method	2^{11}	2^{14}	2^{17}	2^{20}	total
Dijkstra	1071	710	587	423	4783
4-Dijkstra	1245	850	717	541	5372
4-Dijkstra(SSE)	1036	627	439	327	4425
16-Dijkstra	1184	822	676	486	5161
16-Dijkstra(SSE)	1117	669	464	366	4608
Bellman-Ford	1164	840	753	589	5343
4-Bellman-Ford	930	723	710	603	4962
4-Bellman-Ford(SSE)	693	473	399	295	3852
16-Bellman-Ford	994	797	766	646	5197
16-Bellman-Ford(SSE)	512	335	291	230	3360
Floyd-Warshall	7802	7025	7414	6162	30403

stead of 32) bits for addresses and $\tau = 50$. To compute all shortcut lengths up to this level, customization takes about 1.99 s to follow the 833 million (write) instructions (which use about 3.1 GB of RAM). In contrast, running Dijkstra-based customization would take 15.23 seconds (even using a so-called phantom level [11]), an order of magnitude slower.

We now consider higher levels. Our second experiment uses 5 levels, with maximum cell sizes 2^8, 2^{11}, 2^{14}, 2^{17}, and 2^{20}. Table 1 reports the total time (on a single core) to compute the shortcut lengths on each of the top 4 levels, as well as the total customization time (including the 1.99 seconds for the lowest level). It shows that individual executions of Dijkstra's algorithm are slightly faster than Bellman-Ford, and Floyd-Warshall is not competitive. Computing distances from 4 boundary vertices at once (prefix "4-" in the table) helps Bellman-Ford, but hurts Dijkstra (which needs more scans). SSE instructions help both algorithms. Due to better locality, the fastest approach is Bellman-Ford with 16 simultaneous searches, which takes 1289 ms to process the top 4 levels, less than half the time taken by plain Dijkstra (2764 ms). We therefore pick 16-Bellman-Ford with SSE as our default approach for the top 4 levels (2^{11}, 2^{14}, 2^{17}, and 2^{20}).

Table 2 compares this default version of CRP with the original implementation of CRP [11] and with CH [18], which has the fastest preprocessing time among state-of-the-art two-phase algorithms. We include two versions of CH: a standard implementation operates on the expanded graph, and a compact version uses explicit turn tables. The latter, due to Geisberger and Vetter [19], was run on a machine comparable to ours, a 2.6 GHz dual 8-core Intel Xeon E5-2670 with 64 GiB of DDR3-1600 RAM. We test our default instance as well as a version with cheaper U-turns (1 s instead of 100 s). For CH, the *customization time* we report corresponds to the entire preprocessing; when a metric changes, CH finds a new order and a different set of shortcuts. The CH space includes shortcuts only, and not original arcs. In every case, (random) query times (and scans) are for finding the distance only. Queries are sequential and customization uses all available cores (12 or 16).

Table 2. Comparison between CRP and two different CH variants on Europe.

	U-TURN: 1 s				U-TURN: 100 s			
	CUSTOMIZING		QUERIES		CUSTOMIZING		QUERIES	
algorithm	time [s]	space [MiB]	nmb. scans	time [ms]	time [s]	space [MiB]	nmb. scans	time [ms]
CH compact [19]	410.90	219.42	624	0.27	1753.84	641.95	1998	2.27
CH expanded	1090.52	1442.88	386	0.17	1392.41	1519.48	404	0.19
CRP original [11]	2.10	61.72	3556	1.92	2.44	61.72	3805	1.96
CRP	0.35	70.49	2702	1.60	0.35	70.49	3009	1.64

CH has faster queries (and is more robust to metric changes) on the expanded than on the compact graph, since it can use a more fine-grained contraction order. CRP queries are slower, but still fast enough for interactive applications. More importantly, our CRP customization takes only 0.35 seconds (on 12 cores) and is at least *three orders of magnitude* faster than CH, with much lower metric-dependent space requirements. It is also up to seven times faster than the original CRP customization.

Fig. 3 suggests that a precomputed metric-independent order could lead to faster CH customization times. Indeed, running our microinstruction-based customization up to cells of size $n/2$ (about 9 million) with guidance step 2 takes about 10.7 seconds (sequentially). Using the same same order for CH would lead to comparable customization times. The number of shortcuts generated (231 M) is only twice the number of original arcs in the expanded graph (116 M), but still significant considering that CH (without witness searches) must keep all of them for every metric. CH queries should be comparable to CRP, which performs only 1272 scans and takes 0.89 ms on average in the resulting 21-level setup.

Our last experiment considers other benchmark instances. From the 9th DIMACS Challenge [13], we take PTV Europe and TIGER USA, each with two cost functions: driving times (with 100 s U-turn costs) and distances. We also consider OpenStreetMap (OSM) data (v. 121812) representing major landmasses and with realistic turn restrictions. Finally, we test the instances used by Bing Maps, which build on Navteq data and include actual turn costs and restrictions; the proprietary "default" metric correlates well with driving times.

For each instance, Table 3 shows the average number of scans and running time (over 100 random queries) of turn-aware Dijkstra, followed by the metric-independent CRP preprocessing time and the amount of metric-independent data generated. It then reports the customization time and the amount of metric-dependent data produced, followed by average statistics about queries (over 100 000 runs): number of scans, time to get the length of the shortest path, and time to get a full description of the path (length and underlying arcs). Queries are sequential and use a (prewarmed) LRU cache for 2^{18} shortcuts; preprocessing and customization run on 12 cores. We use the default CRP settings in every case, with a sixth overlay level (cell size 2^{23}) for the two largest instances.

The table shows that CRP is indeed robust, enabling consistently fast customization and queries. It is slowest for OSM instances, which are very large

Table 3. Performance of CRP on various benchmark instances.

				Dijkstra		CRP						
				QUERIES		PREPRO		CUSTOM		QUERIES		
		n	cost	scans	time	time	space	time	space	nmb.	dist	path
source	input	[×10^6]	func	[×10^6]	[ms]	[s]	[MiB]	[ms]	[MiB]	scans	[ms]	[ms]
PTV	Europe	18.0	dist	9.1	3069	796	4151	351	70.5	2916	1.86	2.43
	Europe	18.0	time	15.2	6093	796	4151	347	70.5	3009	1.64	1.81
TIGER	US	23.9	dist	12.1	4790	617	6649	677	111.1	3088	1.84	2.78
	US	23.9	time	13.2	6124	617	6649	664	111.1	2964	1.60	1.89
OSM	Australia	4.9	time	3.4	919	79	531	44	4.6	1108	0.27	0.40
	S. America	11.4	time	9.2	2549	222	2520	256	20.4	1238	0.32	0.61
	N. America	162.7	time	115.8	70542	2752	18675	1202	199.1	2994	1.60	3.63
	Old World	189.4	time	127.0	77121	3650	21538	1234	195.4	2588	1.49	4.20
Bing	N. America	30.3	dflt	28.3	11684	936	8125	762	136.6	3395	1.60	1.91
	Europe	47.9	dflt	37.0	17750	1445	7872	602	120.7	3679	2.10	2.52

because (unlike other inputs) they use vertices to represent both intersections and geometry. Even so, customization takes about a second, and queries take under 2 milliseconds. Preprocessing time is dominated by partitioning. While the amount of metric-dependent data is relatively small, instruction arrays can be quite large. Metric-independent space usage could be reduced using smaller τ or limiting microinstructions to smaller cells (than 256). Curiously, finding the length of a path takes similar time on most instances, but describing the path takes longer on OSM data. For every instance, customization is at least one order of magnitude faster than a single Dijkstra search, and would still be faster *even if run sequentially.* The main reason is the poor locality of Dijkstra's algorithm, whose working set is spread throughout the graph.

7 Final Remarks

We have significantly reduced the time needed to process a cost function to enable interactive queries on road networks. Our customization is an order of magnitude faster than the best previous method [11], and takes less time than a single Dijkstra search. The ability to incorporate a new metric in fractions of a second enables a host of new applications. For example, it allows personalized cost functions: we could store a compact description of the preferences of each user, and run customization on the fly whenever the user accesses the system. Cost functions can even be tuned interactively (in less than a second). Our approach can also be helpful in applications [4, 17, 24] that repeatedly compute shortest paths on the same underlying graph with a changing cost function. Most importantly, very fast customization has the potential to enable applications that have not even been considered so far.

Acknowledgements. We thank D. Luxen for routable OSM instances [1], C. Vetter for making his code [19] available, and T. Pajor for running it.

References

1. Project OSRM, 2012. http://project-osrm.org/.
2. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *Proc. ESA*, LNCS 7501, pp. 24–35. Springer, 2012.
3. I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *SODA*, pp. 782–793, 2010.
4. R. Bader, J. Dees, R. Geisberger, and P. Sanders. Alternative Route Graphs in Road Networks. In *Proc. TAPAS*, LNCS 6595, pp. 242–253. Springer, 2011.
5. H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.
6. R. Bauer, T. Columbus, B. Katz, M. Krug, and D. Wagner. Preprocessing Speed-Up Techniques is Hard. In *Proc. CIAC*, LNCS 6078, pp. 359–370. Springer, 2010.
7. R. Bauer and D. Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM JEA*, 14(2.4):1–29, 2009.
8. 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.
9. R. Bellman. On a routing problem. *Q. Appl. Math*, 16(1):87–90, 1958.
10. T. Columbus. Search space size in contraction hierarchies. Master's thesis, Karlsruhe Institute of Technology, 2012.
11. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *Proc. SEA*, LNCS 6630, pp. 376–387. Springer, 2011.
12. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *Proc. IPDPS*, pp. 1135–1146. IEEE Computer Society, 2011.
13. C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors. *The Shortest Path Problem: 9th DIMACS Implementation Challenge*, DIMACS Book 74. AMS, 2009.
14. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
15. R. W. Floyd. Algorithm 97 (shortest paths). *Comm. ACM*, 5(6):345, 1962.
16. L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton U. Press, 1962.
17. S. Funke and S. Storandt. Polynomial-time Construction of Contraction Hierarchies for Multi-criteria Queries. In *Proc. ALENEX*, pp. 41–54. SIAM, 2013.
18. R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Sci.*, 46(3):388–404, 2012.
19. R. Geisberger and C. Vetter. Efficient Routing in Road Networks with Turn Costs. In *Proc. SEA*, LNCS 6630, pp. 100–111. Springer, 2011.
20. A. V. Goldberg. A Practical Shortest Path Algorithm with Linear Expected Time. *SIAM J. Comp.*, 37:1637–1655, 2008.
21. A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Demetrescu et al. [13], pp. 93–139.
22. M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [13], pp. 41–72.
23. M. Holzer, F. Schulz, and D. Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM JEA*, 13(2.5):1–26, 2008.
24. D. Luxen and P. Sanders. Hierarchy Decomposition and Faster User Equilibria on Road Networks. In *Proc. SEA*, LNCS 6630, pp. 242–253. Springer, 2011.
25. N. Milosavljevic. On optimal preprocessing for contraction hierarchies. In *Proc. SIGSPATIAL IWCTS*, pp. 6:1–6:6, 2012.
26. H. Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *Proc. IPDPS*, pp. 1–10, 2010.