译文及原文

三号黑体居中，上下各空一行

**Customizable Route Planning**

Daniel Delling1, Andrew V. Goldberg1, Thomas Pajor2*\**, and Renato F. Werneck1

1 Microsoft Research Silicon Valley

*{*dadellin,goldberg,renatow*}*@microsoft.com

2 Karlsruhe Institute of Technology

[pajor@kit.edu](mailto:pajor@kit.edu)

**Abstract.** We present an algorithm to compute shortest paths on con- tinental road networks with arbitrary metrics (cost functions). The ap- proach supports turn costs, enables real-time queries, and can incorpo- rate 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.

# Introduction

The past decade has seen a great deal of research on finding point-to-point short- est 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 fo- cused 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 per- form 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 han- dling 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 met- ric. 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

女 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 hier- archical 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 (Sec- tion 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 sig- nificant 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 partic- ular 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.

# 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 rout- ing problem* [11], one must answer queries on linear combinations of a small set of metrics (typically two) known in advance. Queries in the *constrained rout- ing 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 rerun- ning 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 de- termine 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 *hi- erarchy* 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 prepro- cessing, 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 uncom- petitive 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.

# 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* (ex- cluding 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*; Sec- tion 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*, *Cs*, and *Ct*. (Here *Cv* 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 rou- tinely finds solutions with half as many boundary edges (or fewer), compared to the general-purpose partitioners (such as METIS [20]) commonly used by previ- ous 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, *. . .*, *vk}* is the set of boundary vertices of a cell, let *Ti*

be the shortest path tree (restricted to the cell) rooted at *vi*, and let *T* l be

*i*

the subtree of *Ti* consisting of the vertices with descendants in *B*. We take the

union *C* = *∪k*

*i*=1

*T* l 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

*i*

the effort spent on each level is balanced, which happens for *U ≈* 215.

To analyze preprocessing times (not depicted in the plots), take *U* = 215 (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

+

x



+

+

+

x



+



+

+

x

x

+

+

+

x





+

x



x





x



x

x

 

x x

+

* cliques

x red. cliques

+ skeleton CH−skeleton

+

x



+

x



+







+

x







 +

x

+

x

x

+

x

+

x

x

x

x

* none

x

x ALT

+ PCD



x+



+

+

+

+

+

x

+

+

+

x

+

x



x



x



x







x x

+

+



x 



x

+

cliques

red. cliques skeleton CH−skeleton

x x 

+

x

●+

x



●+

x



+

x



 

+

x

x

+

x

+

x

x

x

x

+

* none

x ALT

+ PCD







x

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* = 220−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* = 215 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 re- duction is relatively fast, we use reduced cliques as the default overlay.

Goal-direction. For even faster queries, we can apply more sophisticated tech- niques (than bidirectional Dijkstra) to search the overlay graph. While in prin- ciple 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 indepen- dent preprocessing (*k ≈ n/U* ). During metric customization, we run Dijkstra’s algorithm *k* times on the overlay graph to compute a *k × k* matrix with the dis- tances 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 ver- tices 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 con- sumption 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 millisec- ond). PCD, however, needs much smaller cells, and thus more space and cus- tomization time (about a minute for *U* = 214) 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].

query time [ms]

query time [ms]



U = 218

x 17

1

+

U1 = 2

U = 216

1

U1 = 2

15

U = 214

13

1

U1 = 2



x

+





x

+



x







x

+x

x

x

+

+

+

+

x



U = 218

x 17

1

+

U1 = 2

U = 216

1

U1 = 2

15

U = 214

13

1

U1 = 2

+





x









x

x

+

x

x

+x

+

+

+



+

1 2

3

4

5

1

2

3

4

5

metric−dependent space [MB]20 metric−dependent space [MB]

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 = 27+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 *Hi* as before: it includes all boundary arcs, plus an overlay linking the boundary vertices within a cell. Note that *Hi* can be computed using only *Hi*−1. We use PUNCH to create multilevel partitions, in top-down fashion.

An *s*–*t* query runs bidirectional Dijkstra on a restricted graph *Gst*. An arc (*v, w*) from *Hi* will be in *Gst* 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 = 211 and *U*1 = 216). 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 implemen- tation 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 ver- tices 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 |
| time space vertex time |  | time space vertex time |
| algorithm [cell sizes] | [s] [MB] scans [ms] |  | [s] [MB] scans [ms] |
| CALT [211:216] | 21.3 37.1 5292 0.92 |  | 17.2 48.9 5739 1.26 |
| MLD-1 [214] | 4.9 10.1 45420 5.81 |  | 4.8 10.1 47417 6.12 |
| MLD-2 [212:218] | 5.0 18.8 12683 1.82 |  | 5.0 18.8 13071 1.83 |
| MLD-3 [210:215:220] | 5.2 32.7 6099 0.91 |  | 5.1 32.7 6344 0.98 |
| MLD-4 [28:212:216:220] | 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 acceler- ates 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 im- prove 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.

# 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 intersec- tion becomes a single vertex with some associated information. If a vertex *u* has *p* incoming and *q* outgoing arcs, we associate a *p × q turn table Tu* to it, where *Tu*[*i, j*] represents the turn from the *i*th incoming arc into the *j*th 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 *bound- ary 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 |
| algorithm | customizing queries  time space vertex time [s] [MB] scans [ms] |  | customizing queries  time space vertex time [s] [MB] scans [ms] |

MLD-1 [214] 5.9 10.5 44832 9.96 7.5 10.5 62746 12.43

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| MLD-2 [212:218] | 6.3 | 19.2 | 12413 3.07 | 8.4 | 19.2 | 16849 | 3.55 |
| MLD-3 [210:215:220] | 7.3 | 33.5 | 5812 1.56 | 9.2 | 33.5 | 6896 | 1.88 |
| MLD-4 [28:212:216:220] | 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 /*= *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 com- pact 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.

# Conclusion

Recent advances in graph partitioning motivated us to reexamine the separator- based multilevel approach to the shortest path problem. With careful engineer- ing, 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 sepa- rating 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.

**自定义路径规划**

Daniel Delling, Andrew V. Goldberg,

Thomas Pajor, and Renato F. Werneck

硅谷微软研究院

*{*dadellin,goldberg,renatow*}*@microsoft.com

卡尔斯鲁厄理工学院

[pajor@kit.edu](mailto:pajor@kit.edu)

**摘要**

我们提出了一种算法，来计算关于任意度量值（成本函数）的大陆公路网络的最短路径。这种算法支持转换成本，实现实时查询，并且可以在几秒钟内，以足够快的速度支持实时交通更新和个性化的优化功能，来纳入一个新的度量值。由于度量特定的数据的总量是地图本身的一小部分，所以这使我们能够同时维护在存储器几个度量。

1. **介绍**

我们提出了一种算法，来计算关于任意度量值（成本函数）的大陆公路网络的最短路径。这种算法支持转换成本，实现实时查询，并且可以在几秒钟内，以足够快的速度支持实时交通更新和个性化的优化功能，来纳入一个新的度量值。由于度量特定的数据的总量是地图本身的一小部分，所以这使我们能够同时维护在存储器几个度量。

在过去十年时间，在解决公路网路点到点最短路径问题上，有大量的研究。尽管Dijkstra算法以很小的资源开销解决问题，且时间复杂度几乎是线性的，但是在大陆形状大小的地图上仍然需要几分钟的时间。实现这个算法需要使用一个两步骤:处理需要几分钟(甚至几小时)和生产(线性)的辅助数据，然后使用它来执行查询。以前的研究大多数集中在最自然的度量标准，行驶时间。然而真实世界的系统通常也支持自然度量值，如最短距离，步行，骑自行车，避免掉头，避免喜欢高速公路，或避免左转弯。

我们考虑的可定制的路线规划问题，其目标是在有着任意度量值的公路网络上执行时时查询。这种算法可以在两种情况下使用：他们可能会立刻保持几个活跃的度量值（去响应任何查询），或新的度量可以动态生成。具有这些特性的系统具有明显的吸引力。它支持实时路况更新和其他动态场景，可以轻松定制通过处理标准度量值的任意组合，甚至可以提供个性化的行车路线（例如，对于身高和体重有限制的卡车）。为了实现这样的系统，我们需要一种算法，它允许实时查询，具有快速定制（几秒钟）的功能，并保持对每个数据非常少的百RIC。最重要的是，它必须是足够强大：全部三个属性必须适用于任何度量值。但是没有现成的算法满足这些要求。

为了实现这些目标，我们对道路网络两个特性之间加以区分。它的拓扑结构是一系列的每个道路路段或转弯的静态属性，例如物理长度，道路类别，限速的，以及转向类型。它的度量编码了穿越路段或者转向的实际成本。它通常可以以一种函数的方式紧凑地描述，并映射（在恒定的时间内）边缘/转向的属性成本。我们假设当度量值可能会经常变化，甚至共存的时候，拓扑是被度量值和少量的改变共享的。

为了利用这种分层，我们考虑定制的路线规划有三个阶段的算法。第一，度量无关的处理，可以相对缓慢，因为它不经常运行。只需要图形的拓扑作为输入，并可能产生相当数量（与输入大小）的辅助数据。第二阶段，度量定制，对于每个度量标准运行一次，并且必须更快（几秒钟），并产生少量的这个原始图形的小部分的数据。最后，查询阶段使用前两个阶段的输出，并且必须是对实时应用有足够快的响应。

在第2节，我们通过对适用于这种算法的适用性分析来探索设计空间。我们注意到，具有较强的分层组件，在多数情况下，最快的方法出现了过于敏感以至于不能够度量变化的状况。我们专注于基于分离器的算法，它更健壮，但却在最近的研究中一直被忽视，因为公布的结果使它们看起来似乎没有竞争力：相比数千或数百万与其它方法通过迪杰斯特拉观察到的最高加速比均低于60。

在第3节中，回顾了之前的算法，并且彻底地重造了一种基于分离器的算法。通过应用现有的加速技术，还有图形分割学的最新进展，以及一些工程学的进步，我们可以用比现有加速技术少得多的定制的时间（几秒钟）和空间（几十兆字节），在大约1毫秒内，响应大陆道路网络的查询。

我们的论文的另一个贡献是妥善处理了转弯成本（在第4节）。它已经被广泛地认为任何算法可以很容易地移植扩展来有效地处理这些问题，但我们注意到，一些算法中实际上有显著的性能损失，特别是在如果转向表示空间效率的时候。与此相反，我们可以在对性能的影响不大的情况下，自然处理转向。

我们强调我们的算法并不意味着在任何特定度量值下是响应最快的。对于“好”的度量值，我们的查询比最佳的分层算法有点慢。然而，我们的查询是健壮的，它适于任意包括那些分级算法不能适用在内的度量值。我们的算法能快速处理新的度量值，并通过使特殊的度量值信息足够小，来在内存中立刻保存几个度量值。

1. **前沿技术**

现在，出现了各种不同的前沿的研究，以一种特殊的方法来处理多度量值的路径规划问题。SHARC处理方法可以被优化至一次处理多个（已知）度量值。在灵活的路径选择问题，我们必须响应询问在事先已知一小部分度量值（通常是两个）的线性组合。在响应约束路径问题的时候必须避免边缘的整个类。在多标准优化时，就必须找到多个度量值之间的帕累托最优路径。

在初始度量值良好的情况下，ALT和CH算法可以适应一些微小的变化，而不需要运行处理。所有这些方法必须预先知道基本度量值，并有只有在性能度量值较少，准确，而且彼此相似的情况下，才能有好的结果。在实践中，甚至到了度量值看似很小的变化（如高昂的掉头费用）实施的一些方法是不切实际的。与此相反，当我们在处理的过程中出现了这样的度量值，我们则要忽略。

我们现在讨论的现有点至点算法的性能，以确定他们如何适应我们的设计目标。一些最成功的现有的方法，如达不到基于路径，收缩层级（CH），SHARC，中间节点的路径，和依赖于路网与出行时间的强劲层次的次结构。在图中，遥远区域之间的最快路径趋向于使用相同的主通路。

对于有强健层次的度量值，比如旅行时间，CH算法拥有许多我们想要的特性。在处理过程中，CH算法按照其重要性的升序排列这些顶点，并且以这种阶简化它们（为了简化v，我们暂时把它从图中删除，然后根据需要添加距离弧线，以保持其相邻点之间的距离不变）。查询运行双向Dijkstra算法，但只沿着弧线方向或简化方向来获得更重要的顶点。如果一个度量值变化幅度不大，则它可以保持排序不变，并在一分钟左右重新计算的简化。不幸的是，工作与一种度量值下的排列阶，有可能在另一种完全不同的度量值不太适用（如旅行时间和距离，或堵车）。此外，当度量值有着不太明显的层次结构时，查询就会变得比平时更加缓慢。更为关键的是，一旦出现了糟糕的度量值，处理阶段将会变得与实际不符（在空间和时间上），如第4节将展示。与此相反，基于目标方向的一些技术，如PCD，ALT和弧标志，对于任意度量值产生了相同数量的辅助数据。查询虽然是不稳健的，但是：对于糟糕的度量值，他们最糟可以像Dijkstra算法一样慢。即使是旅行时间，PCD和ALT算法跟其他算法相比没有竞争力。

第三种方法是基于图形分割。在处理过程中，人们通过计算图的多层次分区，去创建一系列相互连接的覆盖图形。因为查询的进行，它开始于最低级（地方），从而移动到更高级（全球）的水平。虽然这些技术超前于基于层次的方法，但它的查询时间在实践中被公认为不具有竞争力，并且他们也没有在大陆规模的道路网络进行测试。（例外情况是最近的扩展变体，通过在处理过程中添加许多额外的边缘来获取大量的查询次数，但也同时花费了空间和时间）。因为预处理和查询时间基本上相对于度量值独立的，而基于分离器的算法是最天然的适用于解决我们的问题。

1. **我们的算法**

首先，我们将描述一个基本的算法，再考虑几种技术，使其更加实用，用实验结果来指导我们的设计。我们的代码是用C++（使用OpenMP共享并行化），并使用微软的编译环境Visual C++2010。我们使用四个堆栈作为优先级的队列。实验是在一台商用工作站上运行的，它的使用了英特尔酷睿I7处理器920(四核，2.67GHz时钟频率，以及型号为DDR3-1066的6GB的内存)，它运行着WindowsServer2008R2的服务器系统。我们的标准基准测试实例，是欧洲公路网，拥有1800万的顶点4200万弧，由PTV AG第九届DIMACS Implementation Challenge提供。顶点ID和弧的权值都是32位整数。

我们必须最大限度地减少度量值的既定时间，以及相关的空间（不包括原图），和查询时间，同时保持独立于度量值的时间和空间合理。我们使用均匀随机挑选了10000组s-t数据查询评估我们的算法。我们专注于寻找最短路径权值;第4节展示了如何检索实际路径。尽管我们的报告结果是旅行时间和旅行距离，但通过设计我们的算法，可以使任何度量值运行稳健。

**基础算法。**我们独立于度量值的预处理阶段，把图分割成相连接的小区，它们各自有着最多的U（输入参数）顶点，尽可能少的使用边界圆弧（在不同小区的端点弧）。

度量值的定制阶段生成含有所有边界顶点（那些与另一个小区的至少一个邻居）和G的边界圆弧它还包含一个集合为每个单元C的曲线图H：每对边界顶点（V，W）在C，我们创建了一个弧（v，W），其成本是相同的v和w之间的最短路径（限于C）（或无限如果w是不可达从v）中。我们通过从每个边界顶点运行Dijkstra算法这样做。注意，H是一个覆盖的图：H中的任意两个顶点之间的距离是一样的在G。

最后，执行s和t之间的查询，我们在由H，Cs和CT组合的图形上，运行Dijkstra双向算法。（这里CV表示的G通过在含有小区顶点诱导的子图）。如已经提到的，这是基于分离器的方法的基本策略。

特别是，HiTi使用基于边缘的分离器和派系来表示每个小区。不幸的是，HiTi科技尚未在大道路网测试;实验限于小网格和概念的原始证明，并没有出现用现代算法工程技术来进行了优化的报告。

我们对HiTi和类似的算法第一个改进是使用PUNCH去分解图。最近开发的处理道路网络，它经常发现的解决方案与一半的边界边缘（或更少），相较于一般用途的分解方案使用者（如METIS）通过前一常用算法。更好的分区减少了定制的时间和空间，从而促使更快的查询。在我们的实验中，使用了冲头的相对长的运行，需要大约一个小时。如果我们使用基本版本的PUNCH算法，我们的研究结果也不会改变太多，它仅仅只有5％左右的性能损耗。

我们通过使用并行化系统：查询工作在两片CPU上同时进行向前搜索和向后搜索，并且定制使用所有的四个CPU(每一个单元独立运行)。

**稀疏化。**在叠加图采用全部的资源似乎是浪费，特别是对于良好的度量值。在作出其度量独立的拓扑的权值方面，我们考虑各种技术来减少叠加图。

第一种方法是减少边缘，从而消除了集合的弧不在最短路径上。计算所有派系之后，我们从H中的每个顶点v运行Dijkstra算法，直到扫描了V（在H）的所有邻居才立刻停止。请注意，这些搜索通常很快，因为它们只访问叠加图。

更激进方法是保存一些内部单元的顶点。如果B ={V1，V2。 。 ，VK}是一系列的小区边界顶点，令Ti为以vi为根节点的最短路径树(仅限于单元)，其次令 成为Ti的子树，它由B中的子孙树的顶点组成。我们用这些子树的组合，然后通过两个甚至更少的邻结点简化所有的内部顶点。请注意，这个框架图是技术上不重叠，但它保留了所有边界顶点之间的距离，这正是我们需要的。

最后，我们尝试了一个轻量级的收缩计划。从框架图形开始，我们尽可能的简化较低度的内部顶点，当没有了不用增加多于一个的边缘数目这样的操作可以进行时的，操作停止。 图1（左）比较旅行时间和行程距离所有四个覆盖（组团，减少的组团，骨架，CH-骨骼）。每个小区涉及总查询时间和U型不同的值（小区大小）无关的数据的量。勿庸置疑，所有叠加层需要更多的空间作为细胞数目的增加（即，当U减小）。然而，当花费在每一个等级的工作平衡时，查询时间被最小化，此时U。

为了分析预处理时间（在图中未标出），取U =（含旅行时间）为例。发现全派系只需要40.8s，但边缘减少（45.8s）或建造骨架图（45.1s）都几乎一样方便。在79.4s的情况下CH-骨架明显更加昂贵，但仍然实用。大多数方法得到更快的当U变小：全派系需要不到5秒与U =256。唯一的例外是CH-骨架：当U是非常小的，所有骨骼的总大小是相当大的，而且处理它们只需要几分钟。

然而，在查询时间和度量值依赖空间方面，CH-骨架支配纯骨架图。减少边缘的数目（从1.2M具有降低的派系到0.8M与骨架，为U =行驶时间）可能不足以抵消顶点（从34K至280K）的数量增加，向其中Dijkstra-基于算法是更敏感。这也解释了为什么减少派系得到最快的查询，以饱满的派系也不甘落后。

除了全派系，当我们从旅行时间切换到距离（以不太明显的层次结构）时，所有叠加层有更糟糕的表现。因为边缘降低是比较快的，我们使用降低派系作为默认叠加。

**目标方向。**为了更快的查询，我们可以应用更复杂的技术（比双向Dijkstra算法）来搜索叠加图。虽然原则上可以使用任何方法，我们的模型限制了我们对那些与度量值独立的预处理次数。我们测试了PCD和ALT。

与我们的基本算法使用PCD（预计算的距离集群），我们做到以下几点。令k预处理度量值独立的过程中发现的细胞（K≈N / U）的数量。期间度量值定制，我们对叠加图形运行Dijkstra算法k次来计算第k×K矩阵的所有单元之间的距离。查询然后使用矩阵通过修剪是远离的最短路径顶点引导双向搜索。需要注意的是，不同于“纯”PCD，我们的定制和查询过程中使用的叠加图。

另一种技术是核心ALT（CALT）。查询开始与限制在源小区和目标小区的双向的Dijkstra搜索。那么他们的边界顶点作为一个ALT出发点（A \*搜索/标/三角不等式）的叠加图查询。该ALT预处理进行Dijkstra O（L）次选择L顶点作为标志，并存储这些标志性建筑，并在覆盖所有顶点之间的距离。查询使用这些距离和三角不等式来引导朝着目标搜索。这种基于处理器的并发处理方法需要挑附近的覆盖顶点作为代理为源或目标得到他们的距离范围。因此，查询使用四个CPU处理器：两个选择代理，另外两个进行实际双向搜索。

图1（右）表面了基本算法的查询时间和度量依赖空间消耗，CALT（带32避免标），和PCD，具有降低的派系作为叠加的曲线图。随着空间有所增加，两者的目标方向的技术产生显著更快的查询（大约一毫秒OND）。 PCD，但是，需要更小的单元，因此更多的空间和定制时间（约为U =分钟），比ALT（小于3秒）。这两种方法都超过出行距离出行时间更为有效。

**多级别。**为了加快查询，我们可以使用覆盖图的多个级别，基于分区的方法的常用技术，其中包括HiTi。我们需要的G嵌套分区，其中每一个边界边缘在i级也是在第i级的边界的边缘 - 1，对于i>1，0级分区是原始图形，每个顶点作为小区。对于第i级分割，我们如之前创建的曲线图，Hi：它包括所有边界圆弧，以及一个覆盖联的小区内的边界顶点。需要注意的是，Hi可以仅使用高保真1来计算。我们以自上而下的方式使用PUNCH建立多层次的分区。

一个s-t查询, 使用双向Dijkstra算法运行在一个受限图GST上。如果既v和w都在同一个小区并且s或t在级i + 1，一个Hi弧（V，W）将存在在GST中。目标的方向仍然可以在顶层中使用。

图2表明了多级算法的两个重叠的水平上的顶层的性能（具有降低的派系）和ALT。我们报告多次查询和度量值依赖空间U0和U1，在底部和顶部的水平，最大单元尺寸的多个值。与图1比较，表明采用两个级别使得对于相同的空间快得多的查询。对于出行的时间，查询需要1毫秒，约40 MB（与U0=和U1=）。这需要16 s到计算底部覆盖层，5秒来计算顶部覆盖层，只有0.5秒处理标志性建筑。当使用60 MB时，查询可能只需0.5毫秒。

**流线型的实现。**虽然稀疏化技术节省空间和目标方向加速查询，改进适中，并提出在预处理时间，实施复杂性为代价，和度量值独立的（覆盖拓扑仅仅是度量值独立的全派系）。此外，简单的集合执行时间和空间需求可通过代表分区作为矩阵的每个单元，使得性能差更小的提高。该矩阵包含其进入和退出的顶点之间的32位距离（这些都分别至少有一个传入或传出边界圆弧，顶点;大多数边界顶点都是）。我们还需要阵列与相应的顶点编号的行（和列）关联，但这些小的和由所有指标共享。

因此，我们创建了一个基于矩阵的流线型实现，由于邻接集合的实施大约快了两倍。它不使用边缘降低，因为它不再节省空间，减慢定制，它的有效性取决于度量值。（跳过无限矩阵元素将使查询仅稍快一些）。同样，我们从精简表示排除CALT，因为它的查询是复杂并具有高方差。

定制时间通常通过建立的最低水平的覆盖支配，因为它的原理是直接工作在底层图形（更高的水平上的水平以下的小得多派系工作）。正如我们观察到的，较小的小区往往会导致更快的预处理。因此，作为一种优化，精简实现包括假体等级（与U =32），以加速定制，但引发它离开查询，保持空间使用不受影响。对于MLD-1和MLD-2，我们使用U =256，以及第二个假体水平。

表1比较了我们的简化的多级实现（称为MLD，具有最多4个级别）与CALT的原来的2级实现。对于每一个算法，我们提出在每个级别的单元尺寸范围。 （因为CALT加速顶层，它使用不同的单元尺寸比MLD-2）。我们还考虑CH的两个版本：第一个（经济）减少预处理时间，第二个（大方）快捷键的数量。关于CH，我们报告存储快捷方式（每弧8个字节，但不包括原始图形）所需的总空间。对于所有的算法，预处理采用四核和查询使用至少两个。

我们不重排CH预处理之后的顶点（按照惯例，以IM-证明查询所在地），因为这样可以防止不同的指标从共享相同的图形。即便如此，由于旅行时间，CH查询是一个数量级比我们的算法快。对于出行距离，MLD-3和MLD-4比CH快，但幅度不大。出于实用的目的，所有的变体具有足够快的查询。

我们的方法的主要吸引力是有效率的度量值定制。我们需要少得多的空间，例如：MLD-2需要约20 MB，这是原始图形和幅度小于CH的阶的不到5％。最值得注意的是，定制倍小。我们只需要5秒，应对新的度量值，这是速度不够快，使个性化的行车路线。这是两个数量级比CH快，即使是一个乖巧的指标。假体水平在这里帮助：没有他们，MLD-1将需要大约20秒。

注意，CH定制可以更快，如果处理阶是固定的预先。经济变量可以重建的距离在54号的出行时间和178 S上的层次（按阶）（仍慢于我们的方法）。

不幸的是，使用阶为一个度量来重建另一个如果它们是非常相似的唯一有效的。还要注意，可以通过仅存储层次的上部，在查询时间的费用节省空间。

表1显示，我们可以轻松应对实时路况：如果所有的边缘成本变化（由于交通更新），我们只能在5秒后处理新的查询。我们也可以支持本地更新非常有效。如果单个边缘成本的变化，我们必须重新计算每个级别最多只有一个单元格，MLD-4需要不到一毫秒这样做。这是另一个理由不使用边缘减少或CALT：用任何一种技术，在一个单元格的变化可能传播超越它。