Optimizing Landmark-Based Routing and Preprocessing

Alexandros Efentakis
Research Center "Athena"
Artemidos 6, Marousi 15125, Greece
efentakis@imis.athena-innovation.gr

Dieter Pfoser*
Department of Geography and GeoInformation Science
George Mason University
4400 University Drive, MS 6C3 Fairfax
VA 22030-4444
dpfoser@gmu.edu

ABSTRACT

Many acceleration techniques exist for the single-pair shortest path problem on road networks. Most of them have been significantly improved over the years to achieve faster preprocessing times and superior performance. In this spirit, our current work significantly improves the classic ALT (A* + Landmarks + Triangle equality) algorithm. By carefully optimizing both preprocessing and query phases, we managed to effectively minimize preprocessing time to a few seconds, making the ALT algorithm also suitable for dynamic scenarios, i.e., road networks with changing edge weights due to traffic updates. We also accelerated the query phase for both unidirectional and bidirectional versions of the ALT algorithm, providing fast enough query times (including full-path unpacking) suitable for real-time services and continental road networks.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms; H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Algorithms

Keywords

Shortest-path computation, ALT algorithm, Road Networks

1. INTRODUCTION

Over the years there has been a great deal of research in finding point-to-point shortest paths in road networks. Although the classic Dijkstra algorithm [13] solves the single pair shortest path (SPSP) problem of finding an exact shortest path of length $d(s,t)$ between a given source s and target t in a graph $G = (V, E, l)$, it still requires a few seconds in continental-sized road networks. Faster alternative algorithms use a two-stage approach: preprocessing requires a few minutes (or hours) and produces a (linear) amount of additional data that is used to accelerate shortest path queries.

Existing methods for solving the SPSP problem in road networks may be classified to three major categories (see [9] for an overview). *Hierarchical Approaches* such as Transit Node Routing (TNR) [4], Contraction Hierarchies (CH) [17] or Hub-based Labeling algorithm (HL) [1] exploit the inherent hierarchical structure of the given road network and build a *search Graph* which includes *shortcuts*, i.e., additional edges connecting important nodes (those participating in many SP queries). In contrast, *goal direction techniques* such as ALT [18] and Arc-flags [24, 26] direct the search towards the target by preferring edges that shorten the distance to the goal node and ignoring edges that cannot possibly belong to the shortest path based on their preprocessed data. A third category is based on *graph separators* such as HiTi [22] and Customizable Route Planning (CRP) [8]. 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.

Many of those acceleration techniques have been significantly improved over the years. State-of-the-art methods such as TNR and HL originally required extensive preprocessing time of more than a few hours. Later works [2], [3] improved those preprocessing times to just a few minutes. Unfortunately those preprocessing times are still not fast enough for real-time mapping services, where edge weights change frequently due to traffic updates (typically every 15 - 30 minutes). Additionally, since they are based on Contraction Hierarchies (CH), whenever edge weights change new shortcuts must be added and others must be removed from the search graph, altering the search graph's entire structure, making path unpacking harder and slower to implement. Additionally, CH is very sensitive to the metric used, making the preprocessing significantly slower for a) travel distances b) turn restrictions [8]. That is why mapping services such as Bing Maps, prefer to use CRP which might be orders of magnitude slower than HL or TNR, but has faster preprocessing times (few secs) and uses always the same shortcuts regardless of the metric used ([11], [8]). To sum up, the above mentioned methods, although provided superior shortest-path computation speeds are not suited for a dynamic navigation scenario in which the edge weights of the road network graph change based on actual traffic conditions.

This work aims at significantly improving the performance of the ALT (A* + Landmarks + Triangle equality) algorithm [18] and making it suitable for a dynamic navigation scenario. We focus our attention on the ALT algorithm, since it has none of the aforementioned disadvantages of hierarchical methods, i.e., (i) it is very robust with respect to the metric used [18] (ii) it requires no path

*On leave from Research Center "Athena", Greece.

IWCTS '13, November 05-08 2013, Orlando, FL, USA
 ACM 978-1-4503-2527-1/13/11 $15.00
http://dx.doi.org/10.1145/2533828.2533838.

unpacking (producing the actual road network path of the shortest route) (iii) its storage requirements and auxiliary data structure size depend solely on the number of landmarks (and not on the utilized metric) and most importantly (iv) the typical landmarks-based preprocessing may also be used for estimating the graph distance between any two vertices. For that reason, it has been used in other contexts outside road networks (such as social networks) in cases when the actual distance between two nodes can be sufficiently replaced by the close estimation provided by the typical landmarks preprocessing ([27],[32]).

Our specific contributions are to improve (i) ALT's preprocessing time and (ii) its shortest-path query phase performance. By proposing a novel, simple, yet efficient landmark selection strategy and exploiting several optimization strategies, we managed *to lower the preprocessing time from several minutes [10] to a few seconds*. Moreover, we also improved ALT's *query phase* and *tripled unidirectional ALT performance* while also *improving bidirectional performance by 44%*. Although we did not alter the actual algorithm (including memory requirements and time complexity) our efforts significantly broadened ALT's entire scope, since: a) its preprocessing is now fast enough for supporting dynamic road networks with frequent traffic updates b) ALT algorithm is now fast enough to support real time SP queries for global scale mapping services. The efficiency and performance of our approach is already demonstrated in the live system [14] of the SimpleFleet [31] project that uses live-traffic information updated every 5 minutes.

In addition, we also filled a gap in ALT's bibliography, since to the best of our knowledge there was no previous work examining its performance for varying number of landmarks and the travel distances metric. By documenting those experiments we get an additional insight of the performance characteristics of the algorithm.

The outline of this work is as follows. Section 2 describes previous work in relation to the ALT algorithm. Section 3 describes our scientific contribution beyond the current state-of-the-art in terms of ALT's preprocessing and performance. Experiments establishing the superiority of our approach are provided in in Section 4. Finally, Section 5 gives conclusions and directions for future work.

2. RELATED WORK

In the discussion on related work that follows, we are dealing with directed weighted graphs $G(V, E, l)$, where V is a finite set of vertices, $E \subseteq VxV$ are the edges of the graph and l is a positive weight function $E \rightarrow R^+$. Typically, weight l represents the travel time required to traverse the edge. In other cases, l may refer to the length of the edge in meters (for travel distances metric). The reverse graph $\bar{G} = (V, E)$ is the graph obtained from G by substituting each edge $(u, v) \in E$ by (v, u).

2.1 The ALT algorithm

The concept of landmarks within the context of the single-pair shortest-path problem was officially introduced in [18]. In this work, a small set of vertices called landmarks is chosen and for each vertex, the authors precompute distances to and from every landmark. Given a set $S \subseteq V$ of landmarks and distances $d(L_i, v)$, $d(v, L_i)$ for all nodes $v \in V$ and landmarks $L_i \in S$, the following triangle inequalities hold: $d(u, v) + d(v, L_i) \geq d(u, L_i)$ and $d(L_i, u) + d(u, v) \geq d(L_i, v)$. Therefore, the function $\pi_f = max_{L_i} max\{d(u, L_i) - d(v, L_i), d(L_i, v) - d(L_i, u)\}$, where $0 \leq i \leq |S| - 1$, provides a lower bound for the graph distance $d(u, v)$.

ALT is a bidirectional variant of the classic A* algorithm [20] using the aforementioned lower bounds. Since the combination of A* and bidirectional search is not trivial, correctness can only be guaranteed if π_f (the heuristic function for the forward search) and π_r (the heuristic function for the backward search) are consistent. This means $\pi_f(u, v)$ in G must be equal to $\pi_r(v, u)$ in the reverse graph. ALT typically uses the average potential function [21] defined as $p_f(v) = (\pi_f(v) - \pi_r(v))/2$ for the forward and $p_r(v) = (\pi_r(v) - \pi_f(v))/2 = -p_f(v)$ for the backward search.

The original implementation of ALT uses for each SP computation, only a subset of *h active landmarks*, which are those that provide the best lower bounds on the $s - t$ distance. Later works [19] update the set of active landmarks dynamically during the query phase. The computation starts using the initially best landmarks and as the algorithm progresses additional landmarks (which may provide better lower bounds) are brought into the active set. After every active landmark update, the potential functions change and therefore the priority queues must also be updated. Additionally the algorithm can no longer terminate as soon as the two opposite searches meet. Instead the ALT algorithm may safely terminate only when the sum of minimum keys in the forward and the backward queue exceeds $\mu + p_f(s)$, where μ represents the tentative shortest path length.

Preprocessing. The *preprocessing stage* for ALT is divided in two phases, the landmarks selection process and the computation of distances of all other graph vertices from and to the landmarks. As far as the landmark selection process is concerned, many alternative strategies have been suggested in [18] and [19]. As Delling et al. suggest in [10], "*no technique picks landmarks that universally yield the smallest search space for random queries*" (although some methods, such as the Avoid and maxCover [19] typically perform better).

In the next section we are going to describe the various optimizations and changes we did for the acceleration of both the preprocessing and query phases of the ALT algorithm.

3. SUPERCHARGING ALT

In this section we are going to describe in detail the various optimizations and techniques we used during the preprocessing and SP query phases of the ALT algorithm, in order to dramatically reduce preprocessing time and achieve superior performance, compared to previous approaches.

3.1 Preprocessing

As noted earlier, the preprocessing stage for the ALT algorithm is divided in two phases, the landmarks selection process and the computation of distances of all other graph vertices from and to the landmarks. Most of the preprocessing time is dominated by the landmark selection process, which usually is done by sophisticated algorithms such as Avoid and MaxCover [18]. Unfortunately for continental road network graphs and $|S| > 16$, the MaxCover heuristic is not longer applicable due to its high memory requirements [10]. Therefore it is obvious we need a simpler and faster landmark selection strategy that will also provide similar performance.

3.1.1 Landmark Selection

We propose a novel and extremely simple strategy for selecting landmarks. We partition the graph (using a partitioning tool) into cells and from each cell we select the four corner-most vertices (top, bottom, left, right according to their coordinates) as landmarks. So, if we are going to use 32 landmarks we partition the graph into 8 cells and get the 4 corner-most vertices per cell. If for example, the top node coincides with the leftmost node for a particular cell, we take the second best in one of those directions. Our new landmark selection strategy will be denoted hereafter as the *partition - corners* method.

On a side note, the partitioning of the graph should not considered part of the actual preprocessing, since it is metric independent and happens only once even for dynamic scenarios (as previous works such as CRP suggest [8]). After partitioning the graph and efficiently storing the cell of each node, the selection of landmarks actually takes less than 1-2 sec, since it only requires a linear sweep in the vertex information vector of size |V|.

At first glance, our *partition - corners* selection strategy seems naive but it has many important advantages: i) It is extremely fast ii) It ensures that landmarks are uniformly distributed within the graph iii) The acquired landmarks may accelerate even local SP queries (between nodes belonging to the same cell). Still, since there is no quality guarantee for the selected landmarks, during the SP query phase we do not use the *Active landmarks* optimization (see Section 2.1) used in earlier works. This way, all available landmarks participate in every SP query, which compensates for their supposed "lower" quality (see 3.2 for details).

In terms of partitioning tools, we used the state-of-the-art partitioning tool Buffoon / KaFFPa [30], which was kindly provided to us by its authors. Buffoon / KaFFPa creates far better quality partitions (fewer border nodes) than its predecessor METIS [23] which was used many times before, in the context of SP computation ([15] and [16]). Still, the actual quality of the partitioning plays very little role in our landmark selection process, since we are not interested in minimizing the number of border nodes. Therefore our approach will work with any partitioning tool.

3.1.2 Landmark Distances Calculation

During the second phase of the ALT algorithm preprocessing, we need to calculate distances of all graph vertices to and from the landmarks. Keep in mind that is very important to accelerate this particular preprocessing phase, since in order to adapt ALT to a dynamic scenario (where edge weights change frequently due to traffic updates), this second phase has to run at every batched traffic update. On the contrary, the landmark selection phase has to run only once even for dynamic graphs, since all previous approaches [10] assume using static landmarks, i.e., they do not reposition landmarks if the graph weights are altered.

In order to calculate distances of all graph vertices to and from the landmarks, we need to run two Dijktra algorithms from each landmark, one that runs in the forward graph and one that runs in the reverse graph, for a total of $2|S|$ Dijkstra searches. Since each Dijkstra search is independent from the others, this process may be easily parallelized. Still, this is not good enough if we want to provide preprocessing times of less than a minute. We also need to accelerate each of those individual Dijkstra searches. For that purpose, we applied the following four optimizations:

Dijkstra Heaps. A Dijkstra implementation with a heap structure that only supports Insert and Delete-Min operations (without a Decrease-Key operation), hereafter referred to as *Dijkstra - NoDec*), performs more heap operations and is theoretically inferior to the asymptotic running time of Dijkstra implementations with decrease-key (denoted as *Dijkstra - Dec*). However, previous works have shown that such streamlined heaps are likely to be more efficient. In fact [6] has shown all *Dijkstra - NoDec* implementations when run for various graphs (including road networks) are at least 1.4 times faster than their Dijkstra-Dec counterparts. This improved performance was also evident in our experiments.

Priority Queue Optimization. Instead of using binary heaps for our priority queue implementation, we used the *aligned 4-ary heap* which is a highly-optimized heap for cache memory implemented by Sanders [28]. This array-based heap aligns its data to cache blocks, which in turn reduces the number of cache-misses when accessing any data item. The Sanders implementation we used supports Insert and Delete-Min operations in $O(log4N)$ time and block transfers each. We were able to use such an implementation, only after we employed the previous *Dijkstra - NoDec* optimization.

Although buckets-based priority queues are shown to perform even better for the Dijkstra algorithm [6],[7], since we are going to run many Dijkstra searches in parallel, we did not want to use such memory intensive data structures whose efficiency and size needed depends on the smallest and largest edge weight of the graph. Our experiments have shown that indeed our *aligned 4-ary heap* Dijkstra - NoDec implementation is very fast and memory efficient at the same time and scales pretty well for multicore processors.

Node Reordering. Delling et al. report [7] that Dijkstra's performance improves significantly, if we reorder the vertices so that neighboring vertices have similar IDs, in order to reduce the number of cache misses during computation. Based on [29], they also show that a simple depth first search layout, i.e., reordering the vertices according to a simple depth first search (DFS) improves Dijkstra's speed by 2.8 times. Following those observations, we initially reordered vertices ID according to a DFS layout and then we reordered vertices again, using the partition obtained during the landmark selection process, so that nodes within the same cell are assigned consecutive nodeIDs, as suggested by [16].

Keep in mind that the finalized nodes ordering (DFS layout + partition) not only lowers preprocessing times but it additionally accelerates the SP query phase of the ALT algorithm, as evidenced by our experiments (see Sec. 4).

Graph Data Management. Typically when we want to run bidirectional SP queries on a graph, we use a compact modified adjacency array [25] representation of both forward and reverse graphs, which stores two additional bits per edge in order to separate incoming from outgoing edges per vertex. Although storing the forward and reverse graphs together as a single adjacency array representation is very memory efficient, our experiments have shown that storing forward and reverse graphs separately is significantly faster during preprocessing. Consequently, since forward and reverse graphs are stored separately, during preprocessing we first run all the Dijkstra algorithms from each landmark in the forward graph and once we are done we run the same Dijkstra searches in the reverse graph. That way the parallel threads only operate on one of the two adjacency arrays graph structures, which makes the entire process significantly faster.

Storing the two graphs separately, also accelerates the SP query phase of the ALT algorithm, especially for the unidirectional ALT which runs only in the forward graph. Although storing separately the forward and reverse graphs requires almost double the main memory, the corresponding graph data structure will always be much smaller than the main memory required for storing the landmarks distances. Therefore, it has no impact on the scalability of the ALT algorithm for larger networks.

Conclusively, our experiments (see Sec. 4) showed that by using those four optimizations described earlier, for 32 landmarks and the benchmark continental road network of Western Europe, even our *sequential* calculation of vertex distances from and to landmarks is 3 times faster than previously best published landmarks paper [10] in terms of preprocessing. If we parallelize the process, it takes merely 30sec. on a commodity workstation, which makes landmarks competitive in terms of preprocessing with the fastest (in terms of preprocessing) CRP acceleration technique. As a result, the ALT algorithm may now be used in dynamic road networks with frequent traffic updates as well.

3.2 Shortest-Path Querying

All of the previous preprocessing optimizations (namely: Avoiding decrease-key operations, the aligned 4-ary heap, nodes reordering and storing forward and reverse graphs separately) have also a positive impact on the SP query phase of the ALT algorithm. Still we can do even better. So, we applied 3 additional optimizations to the SP query phase of the ALT algorithm:

Active Landmarks Purging. Previous landmark approaches [19] used the active landmarks optimization (see Sec. 2.1), i.e., they use a subset of the available landmarks during the query phase, which is updated dynamically during the search. This optimization has the disadvantage of requiring to dynamically update the priority queues during the search and that the ALT algorithm cannot terminate as soon as the two opposing searches meet. We dropped this optimization entirely and during the search we get lower bounds based on all available landmarks. This lowers the number of settled nodes (especially for the unidirectional version of ALT), without imposing an unbearable burden on the computation cost, since typical workstations nowadays are more powerful than those of previous years ago.

Keep in mind that by using all available landmarks we more than compensate for their supposed "lower" quality, due to our simple *partition - corners* strategy (see Sec. 3.1.1). Still, since we use all available landmarks for calculating lower bounds, we need to significantly accelerate the process, which can be done with our next two optimizations:

Landmark Distance Records. Similar to [10], we store landmarks distances in a 32-bit vector of size $2 \cdot |S| \cdot |V|$. Distance of node with nodeID $i \in [0, |V| - 1]$ from landmark number $j \in [0, |S| - 1]$ is stored at position $2 \cdot |S| \cdot i + 2 \cdot j$ and the distance of node i to the landmark j is stored in the next position $(2 \cdot |S| \cdot i + 2 \cdot j + 1)$. But what we do entirely differently, is that we store landmarks distances *from landmarks to nodes* negated (as negatives), because this is how they are going to be used during estimation of lower bounds (for the forward search).

In the case of a bidirectional search, at its beginning we cache *the opposite of landmark distances of start and goal node* in two separate vectors of size $2|S|$ (denoted hereafter as the *oppositeStartNodeVector* and *oppositeGoalNodeVector*). As a result, at each node expansion $\pi_f = max(nodeVector + oppositeGoalNodeVector)$ and $\pi_r = -min(nodeVector + oppositeStartNodeVector)$. By storing the opposite of landmarks distances from landmarks in the aforementioned 32-bit vector, we avoid unnecessary additive inversions during the calculation of lower bounds, which makes calculation faster and prepares the ground for our next optimization:

SSE Instructions Current x86-CPUs have special 128-bit SSE registers that hold four 32-bit integers and allow basic operations, such as addition, minimum and maximum to be executed in parallel. By using these 128-bit registers we can significantly accelerate the computation of the lower bounds $\pi_f = max(nodeVector + oppositeGoalNodeVector)$ and $\pi_r = -min(nodeVector + oppositeStartNodeVector)$ computation. This optimization alone gives a solid 10-20% improvement.

Although [7] has used SSE instructions for accelerating SP computation from multiple sources, to the best of our knowledge we are the first that utilize this optimization within a single source SP computation. Moreover, latest Intel Haswell processors already possess 256-bit registers (512-bits registers are in the works) and as a result, this optimization will be even more efficient in the near future.

By all those optimizations, with bidirectional ALT we can achieve SP query times with 48 landmarks, better than those previously reported for 64 landmarks. Moreover, we managed to triple unidirectional ALT performance, as will be shown in the next section.

4. EXPERIMENTS

The experimentation that follows, assesses the performance of our optimizations for the preprocessing and query phases of unidirectional and bidirectional versions of the ALT algorithm for varying number of landmarks.

Experiments were performed on a workstation with a four-core Intel Core i7 processor clocked at 3.4GHz and 32Gb of main memory, running Ubuntu 12.10 64bit. Our code was written in C++ and was compiled with GCC 4.7 and using optimization level 3. We used OpenMP for parallelization. Although the preprocessing stage used all 4 cores (with hyperthreading), SP queries used only one core for accurate benchmarking. We used the strongly connected component of the European road network with 18 million nodes and 42 million arcs made available by PTV AG for the 9th DIMACS Implementation Challenge [12]. Both nodeIDs and edge weights are 32-bit integers. We experimented with both travel times and travel distances.

4.1 Travel times

We compare our approach with the previously best (in terms of efficiency and performance) published ALT paper [10]. During their experiments, they used a slower workstation than ours (dual AMD Opteron 252 at 2.6 GHz with 16 Gb of RAM). Their codebase was also in C++ and was compiled with GCC 4.1, using optimization level 3. In contrast to our approach, no parallelization was used for preprocessing. Their experiments were based on the same benchmark European road network. The results (as well as ours) are based on 10,000 random s-t queries. For an accurate comparison we present their originally recorded times and the same times divided by a of factor of 1.31 (difference between our processors' clock speeds). Their experiments were done for 8, 16, 32, 64 landmarks. We experimented with 8-64 landmarks at steps of 8.

Results are presented in Table 1. In the PREPOCESSING section of the table, the column "time" refers to total preprocessing time (landmark selection + calculating landmark distances) and the column "dist" refers to the time required only for calculating landmark distances. For [10], the numbers in parentheses represent the simulated times, which are the quotients of the original times divided by 1.31.

Results for preprocessing clearly show the inferior performance of previous methods. We use a simpler landmark selection strategy that requires merely 1-2s instead of previous time-consuming and complicated strategies. In addition, our approach is superior even for the preprocessing time required for updating the landmarks distances. Even if we divide the simulated times of [10] by 5.6 (the typical parallel speedup encountered on our 4-core processor with hyperthreading), our approach is still consistently 3 times faster. It is therefore obvious that our various optimizations for preprocessing have really paid off and as a result, the improved preprocessing time always remains consistently below 1min.

In terms of unidirectional queries, we see that unidirectional ALT is now 3 times faster but also settles fewer nodes. This fact is a clear indication that the active landmarks optimization does not work for unidirectional queries and as a result, dropping it was the right choice. By using all the available landmarks, we can easily achieve query times of less than 72ms and the unidirectional ALT scales better when we increase the number of landmarks.

In the case of bidirectional queries, for $|S| \leq 16$, the lower quality of our selected landmarks comes into play and our method settles more nodes and is slightly slower than [10]. On the contrary, for $|S| \geq 24$, results are entirely different. Our method, due to its aggressive optimizations, is consistently faster by 4-5ms from the simulated query times of [10], which constitutes an average im-

Table 1: Performance of ALT for varying number of landmarks, in comparison to [10] (travel times)

	PREPROCESSING				QUERY UNIDIR.				QUERY BIDIR			
	[10]	OURS	[10]	OURS	[10]	OURS	[10]	OURS	[10]	OURS	[10]	OURS
ALGO	time (s)	time (s)	dist (s)	dist (s)	# settled nodes	# settled nodes	time (ms)	time (ms)	# settled nodes	# settled nodes	time (ms)	time (ms)
ALT-8	1566(1205)	8	168(128)	7	1,019.843	1,140,887	391.6(301)	175.3	163,776	465,503	127.8(98.3)	115.7
ALT-16	5112(3932)	16	330(254)	15	815,639	804,663	327,6(252)	124.0	74,669	248,247	53.6(41.2)	60.9
ALT-24		23		22		677,446		110.2		134,315		35.3
ALT-32	1626(1251)	31	666(512)	30	683,566	506,805	301.4(232)	85.9	40,945	74,423	29.4(22.4)	17.5
ALT-40		38		37		449,259		81.0		53,410		15.2
ALT-48		46		45		430,389		78.4		48,499		12.4
ALT-56		55		53		400,483		71.7		38,140		10.8
ALT-64	4092(3148)	60	1326(1020)	58	604,698	385,322	288.5(221)	70.6	25,324	36,607	19.4(14.8)	10.3

provement of 22-44%. By using 48 landmarks, we get even more improved query times than those previously achieved with 64 landmarks (simulated times). For $|S| \geq 48$ we are able to achieve query times $\leq 12ms$, which means that bidirectional ALT is now capable of handling real-time SP queries, since contrary to hierarchical methods, it does not require extra time for returning full paths (path unpacking).

4.2 Travel distances

We also repeated the same experiments, using travel distances for the same road network. This effort was necessary in order to cover a significant gap in the ALT's large bibliography, since (to the best of our knowledge) there is not some previous work demonstrating the performance of ALT algorithm for travel distances, $|S| > 16$ and varying number of landmarks. As a result, in Table 2 we simply present our results. We use the exact same landmarks as before.

Table 2: Performance of ALT for varying number of landmarks and travel distances metric

	PREPROCESS.		QUERY UNIDIR.		QUERY BIDIR	
ALGO	time (s)	dist (s)	# settled nodes	time (ms)	# settled nodes	time (ms)
ALT-8	6	5	1,176,419	170.8	1,165,631	228.1
ALT-16	14	13	682,947	101.1	604,168	127.7
ALT-24	20	19	511,786	91.3	317,227	88.7
ALT-32	26	25	348,060	59.7	160,836	45.1
ALT-40	33	32	319,109	53.0	142,400	39.1
ALT-48	38	37	294,548	48.7	123,952	32.8
ALT-56	44	43	278,579	44.8	112,515	30.0
ALT-64	51	48	264,516	44.5	101,957	29.1

Results for travel distances are exactly what we expected. Preprocessing is *15-22% faster*, since the individual Dijkstra searches typically perform better for travel distances. After a node is encountered for the first time, it is less frequent for further expansions to improve its cost. That is after all one of the advantages of the ALT algorithm in comparison to hierarchical methods, i.e., its preprocessing is faster for travel distances, whereas methods such as CH require 7 times more preprocessing time for the same metric [8] when compared to the travel times metric.

We also see that unidirectional ALT is now almost competitive with bidirectional ALT, both in SP query times and number of settled nodes. This was something to be expected, since previous works [18] has recorded the quite similar efficiency of both methods when travel distances were used. The interesting fact though is, that unidirectional ALT is now faster for travel distances than travel times similarly to plain Dijkstra. To the best of our knowledge, we are the first to pinpoint this very interesting fact.

Moreover, since ALT is more robust with respect to different metrics [5], switching to travel distances only makes bidirectional ALT 2-3 times slower. In the same scenario, hierarchical methods become at least one order of magnitude slower. Because of its robustness, the ALT algorithm has been successfully used for other kinds of graphs, such as social networks ([27],[32]), where most hierarchical, road network oriented methods would fail.

5. CONCLUSION AND FUTURE WORK

In this work we have significantly improved the classic ALT algorithm, both in terms of preprocessing time and shortest-path query performance. Our improvements were considerable in that we lowered preprocessing times to $< 1min$ (a total of 40-52 times improvement) in comparison to previous published works. We also tripled unidirectional ALT SP query performance and improved bidirectional ALT performance up to 44%. Our efforts significantly altered the ALT's scope since (i) its preprocessing is now fast enough for *supporting dynamic road networks with frequent traffic updates* and (ii) the ALT algorithm may now support *real time SP queries* for global scale mapping services.

As shown by previous works, for real-world services we do not always use the fastest algorithm but the most practical one. The ALT algorithm already has several excellent qualities. Robustness to the metric used, the ability to return full paths, robustness to the graph density and stable auxiliary data memory size for all metrics. Through our efforts, the ALT algorithm has now, and what was missing, *practical preprocessing times and fast enough performance for real-world mapping services*. The efficiency and performance of our approach is already demonstrated in a live system [14] addressing fleet management needs. Given this effort, the ALT algorithm is now ready for practical use.

We can give the following directions for future work. Now that ALT has been significantly improved, it would be easy to combine it with other fast preprocessing methods for road networks, like CRP, to further boost SP query performance, without a significant increase in preprocessing times. Moreover, since ALT has been used in other contexts outside road networks, it would be interesting to show how our method performs for other kind of graphs as well. But most of all, we hope to encourage more researchers to add the ALT algorithm to their practical applications.

Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme "SimpleFleet" (`http://www.simplefleet.eu`, grant agreement No. FP7-ICT-2011-SME-DCL-296423).

6. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 230–241, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *Proceedings of the 20th Annual European conference on Algorithms*, ESA'12, pages 24–35, 2012.

[3] J. Arz, D. Luxen, and P. Sanders. Transit node routing reconsidered. In V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 55–66. Springer Berlin Heidelberg, 2013.

[4] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, Apr. 2007.

[5] 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. *J. Exp. Algorithmics*, 15:2.3:2.1–2.3:2.31, March 2010.

[6] M. Chen, R. A. Chowdhury, V. Ramachandran, D. L. Roche, and L. Tong. Priority queues and dijkstraś algorithm, 2007.

[7] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast: Hardware-accelerated shortest path trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 921–931, Washington, DC, USA, 2011.

[8] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *Proceedings of the 10th international conference on Experimental algorithms*, SEA'11, pages 376–387, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *ALGORITHMICS OF LARGE AND COMPLEX NETWORKS. LECTURE NOTES IN COMPUTER SCIENCE*. Springer, 2009.

[10] D. Delling and D. Wagner. Landmark-based routing in dynamic graphs. In *Proceedings of the 6th international conference on Experimental algorithms*, WEA'07, pages 52–65, Berlin, Heidelberg, 2007. Springer-Verlag.

[11] D. Delling and R. Werneck. Faster customization of road networks. In V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 30–42. Springer Berlin Heidelberg, 2013.

[12] C. Demetrescu, A. V. Goldberg, and D. Johnson. *The shortest path problem. Ninth DIMACS implementation challenge, Piscataway, NJ, USA, November 13–14, 2006. Proceedings.* DIMACS Book 74. AMS , 2009.

[13] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[14] A. Efentakis, S. Brakasoulas, N. Grivas, G. Lamprianidis, K. Patroumpas, and D. Pfoser. Towards a Flexible and Scalable Fleet Management Service. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, 2013. To appear.

[15] A. Efentakis, D. Pfoser, and A. Voisard. Efficient data management in support of shortest-path computation. In *Proceedings of the 4th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, CTS '11, pages 28–33, New York, NY, USA, 2011. ACM.

[16] A. Efentakis, D. Theodorakis, and D. Pfoser. Crowdsourcing computing resources for shortest-path computation. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, pages 434–437, New York, NY, USA, 2012. ACM.

[17] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *16th ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 2004.

[19] A. V. Goldberg and R. F. F. Werneck. Computing point-to-point shortest paths from external memory. In *Algorithm Engineering and Experimentation*, 2005.

[20] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[21] T. Ikeda, M. Y. Hsu, H. Imai, S. Nishimura, H. S. Moura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by ai search techniques. 1994.

[22] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14:1029–1046, 2002.

[23] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, 1998.

[24] E. Köhler, R. H. Möhring, and H. Schilling. Fast point-to-point shortest path computations with arc-flags. In *IN: 9TH DIMACS IMPLEMENTATION CHALLENGE*, 2006.

[25] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, Berlin, 2008.

[26] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *J. Exp. Algorithmics*, 11, February 2007.

[27] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 867–876, New York, NY, USA, 2009. ACM.

[28] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5:312–327, 1999.

[29] P. Sanders, D. Schultes, and C. Vetter. Mobile route planning. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 732–743, Berlin, Heidelberg, 2008. Springer-Verlag.

[30] P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experimentation (ALENEX'12)*, pages 16–29, 2012.

[31] SimpleFleet. Democratizing fleet management [online]. `http://www.simplefleet.eu`, 2013.

[32] K. Tretyakov, A. Armas-Cervantes, L. García-Bañuelos, J. Vilo, and M. Dumas. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In *Proc. 20th CIKM conf.*, pages 1785–1794, 2011.