SALT. A unified framework for all shortest-path query variants on road networks

Alexandros Efentakis[1], Dieter Pfoser[2], and Yannis Vassiliou[3]

[1] Research Center “Athena”
efentakis@imis.athena-innovation.gr
[2] Department of Geography and GeoInformation Science, George Mason University
dpfoser@gmu.edu
[3] National Technical University of Athens, Greece
yv@cs.ntua.gr

Abstract. Although recent scientific literature focuses on multiple shortest-path (SP) problem definitions for road networks, none of the existing solutions can efficiently answer all the different SP query variations. This work proposes SALT, a novel framework that not only efficiently answers most SP queries but also *k*-nearest neighbor queries not tackled by previous methods. Our solution offers excellent query performance and very short preprocessing times, thus making it also a viable option for dynamic, live-traffic road networks and all types of practical use-cases. The proposed SALT framework is a deployable software solution capturing a range of graph-related query problems under one “algorithmic hood”.

Keywords: Shortest-paths, *k*-Nearest Neighbors, *k*NN, SALT framework

1 Introduction

During the last decades, recent scientific literature has produced efficient methods for shortest-path (SP) queries on road networks (cf. [1] for the latest overview). Unfortunately, most aforementioned algorithms are tuned to solving a specific problem efficiently, but are rather inefficient when used in a different context. Contrarily, engineering a framework that efficiently solves multiple shortest-path problems, would be the first step towards the direction of a *grand unified SP toolkit*. To this end, the GRASP algorithms [11], solve most variants of the single-source shortest-path problems on road networks, including *one-to-all* (finding SP distances from a source vertex s to all other vertices), *one-to-many* (computing the SP distances between the source vertex s and a set of target vertices T) and *range queries* (find all vertices reachable from s within a given timespan). GRASP requires minimal preprocessing and provides excellent query performance needed in the context of practical and commercial applications.

Another fundamental problem frequently encountered in location-based services is the *k*NN query, i.e., given a query location and a set of objects on the road network, the *k*NN search finds the *k*-nearest objects to the query location. Unfortunately, even the latest work of [21] is not scalable with the network size, since it requires *several hours* for preprocessing continental road networks. In addition, for a large number of randomly distributed objects, an efficient Dijkstra implementation could answer *k*NN queries by settling a few hundreds nodes and requiring $< 1ms$. Moreover, most previous methods require a *target-selection phase*, i.e., they need to mark the objects location within the underlying index. This phase requires a few seconds, hence having limited appeal for applications involving moving objects (e.g., vehicles). Therefore, it only makes sense to use a complex (non-Dijkstra) kNN processing framework in cases of either rather "small" numbers of objects or objects following skewed distributions (e.g., POIs located near the city center), i.e., for cases in which Dijkstra does not perform well.

The contribution of this work is to provide a unified algorithmic solution that may be used in a *dynamic road network* context, while covering a *wide range of shortest-path problems*, such as (i) single-pair, (ii) one-to-all, (iii) one-to-many, (iv) range and (v) kNN queries. Specifically, we aim at combining the fragmented approaches related to the various shortest-path problem definitions and instead propose a unified framework that tackles all of them. Our proposed **SALT** (graph Separators + ALT) framework requires seconds for preprocessing continental road networks and provides excellent query performance for a wide range of problems. We will show that SALT is (i) $3-4\times$ faster for point-to-point queries when compared to existing methods of similar preprocessing times, (ii) it answers one-to-all, one-to-many and range queries with comparable performance to state-of-the-art approaches, and most importantly, (iii) it may also answer kNN queries in $< 1ms$, for both, static or moving objects. As such, our SALT framework could be a *swiss-army-knife for tackling all shortest-path problem variants*, making it a serious contender for use in commercial applications.

The outline of this work is as follows. Section 2 describes previous related work. Section 3 describes our novel SALT framework and algorithms. Experiments establishing SALT's benefits are provided in Section 4 and Section 5 concludes the paper.

2 Related work

Throughout this work, we use directed weighted graphs $G(V, E, w)$, where V is the set of vertices, $E \subseteq V\text{x}V$ is the set the arcs and w is a positive weight function $E \rightarrow R^+$. The reverse graph $\overline{G} = (V, E)$ is the graph obtained from G by substituting each arc $(u, v) \in E$ by (v, u). A partition of V is a family of sets $C = \{c_0, c_1, \dots c_M\}$, such that each node $u \in V$ is contained in exactly one set c_i. An element of a partition is called a *cell*. A multilevel partition of V is a family of partitions $\{C^0, C^1, \dots C^L\}$ where ℓ denotes the level of a partition C^ℓ. Similar to [4], level 0 refers to the original graph, L is the highest partition level and in this work we use *nested multilevel partitions*, i.e., for each $\ell < L$ and each cell c_i^ℓ there exists a unique cell $c_j^{\ell+1}$ (called the supercell of c_i^ℓ) with $c_i^\ell \subseteq c_j^{\ell+1}$. Accordingly, c_i^ℓ is a subcell of $c_j^{\ell+1}$. In this notation, $c^\ell(v)$ is the cell containing the vertex v on level ℓ. Likewise, the number of cells of the partition C^ℓ is denoted as $|C^\ell|$. For a boundary arc on level ℓ, the tail and head vertices are located in different level-ℓ cells; a boundary vertex on level ℓ is connected with at least one vertex in another level-ℓ cell. Note that for nested multilevel partitions, a boundary vertex/arc at level ℓ is also a boundary vertex/arc for all levels below.

In kNN queries, given a query location s and a set of objects O, the kNN search problem finds k-nearest objects to the query location. Throughout this work, similar to [21], we assume that the query location and the objects are both located at vertices.

The ALT algorithm. In the ALT algorithm [13], a small set of vertices called landmarks is chosen. Then, during preprocessing, we precompute distances to and from every landmark for each vertex. Given a set $S \subseteq V$ of landmarks and distances $d(L_i, v)$, $d(v, L_i)$ for all vertices $v \in V$ and landmarks $L_i \in S$, the following triangle inequalities hold: $d(u, v) + d(v, L_i) \geq d(u, L_i)$ and $d(L_i, u) + d(u, v) \geq d(L_i, v)$. Hence, the function $\pi_f = max_{L_i} max\{d(u, L_i) - d(v, L_i), d(L_i, v) - d(L_i, u)\}$ provides a lower-bound for the graph distance $d(u, v)$. Later works [18], showed that landmarks may also provide upper-bounds on the graph distance between any two vertices. Overall, landmarks may be used to approximate graph distances, according to Eq. 1 and 2. ALT then combines the classic A* algorithm with the aforementioned lower-bounds. For bidirectional search, ALT uses the average potential function, 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.

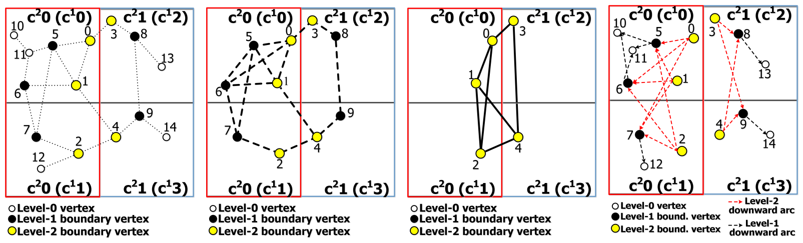
$$d(u, v) \geq max_{L_i} max\{d(u,L_i) - d(v,L_i), d(L_i,v) - d(L_i,u)\} \tag{1}$$

$$d(u, v) \leq min_{L_i}(d(u, L_i) + d(L_i, v)) \tag{2}$$

Graph separators. In Graph Separator (GS) methods, such as CRP [4, 6], a partition C of the graph is computed. Then, the preprocessing phase builds an *overlay graph* H containing all boundary vertices and arcs of G. It also contains a clique for each cell c: for every pair (u, v) of boundary vertices in c, a clique arc (u, v) is created whose cost is the same as the shortest path (restricted to the inner arcs of c) between u and v. For a SP query between s and t, the Dijkstra algorithm must be run on the graph consisting of the union of H, $c^0(s)$ and $c^0(t)$. To further accelerate queries, we may use multiple levels of overlay graphs. Currently, CRP is the most efficient SPSP algorithm in terms of preprocessing time (since the recent Customizable Contraction Hierarchies [9] is only tested on undirected networks) and is thus suitable for dynamic road networks.

SSSP queries. Recently, Efentakis et al. [11] expanded graph separators and proposed GRASP, a novel set of algorithms for handling all variants of single-source shortest-path (SSSP) queries, including one-to-all, one-to-many and range queries. All three algorithms, namely GRASP (one-to-all), isoGRASP (range) and reGRASP (one-to-many) use the exact same data structures and share all the advantages of graph-separator methods, such as very short preprocessing times and excellent parallel query performance. Unfortunately, parallel reGRASP requires a few ms for one-to-many queries on continental road networks and hence is not fast enough for handling kNN queries.

kNN queries There are many works on kNN queries for static objects on road networks. Unfortunately, even the most recent G-tree [21] cannot *scale for continental road networks*, *requiring 16 hours of preprocessing for the full USA network*. Moreover, all index-based approaches require a *target selection phase* to index which tree-nodes contain objects (requiring few seconds) and thus, they cannot be used for moving objects. There is also previous work around kNN queries for moving objects on road networks. However, they are either disk-based [20], have not been tested on continental road networks [14, 17, 20] and cannot address dynamic road networks. Recently, CRP was also expanded [7] to handle kNN queries. Unfortunately, (i) CRP also requires a target selection phase and hence, cannot be applied to moving objects and (ii) it may only perform well for objects near the query location (otherwise the entire upper level of the overlay graph must be traversed). Hence, this solution is also not optimal.



(a) A sample graph G. $|V|$= 15, L=2, $|C^L|$=2

(b) Building the level-1 overlay graph

(c) Building the level-2 overlay graph

(d) Downward arcs and the $G_{GS}\downarrow$ graph

Fig. 1: SALT's GS customization phase. Building the overlay H and the $G_{GS}\downarrow$ graphs

3 The SALT framework

The main contribution of this work is to propose SALT, a unified framework for answering single-pair, single-source (one-to-all, one-to-many and range) and especially kNN queries which are not handled efficiently by existing approaches. The main advantage of SALT is, that the exact same data structures may service all the different type of SP queries on road networks and thus, SALT may be easily integrated into commercial, real-world applications. What follows is a detailed discussion of the SALT framework.

3.1 Preprocessing

SALT's preprocessing consists of two distinct phases, (i) the *graph-separator (GS) phase* and (ii) the *landmarks preprocessing phase*.

The *GS phase* of SALT mimics the preprocessing of GRASP [11] (see Fig. 1). During this phase, we use the Kafpaa/Buffoon [19] partitioning tool to create nested multilevel partitions of the road network graph in a top-down fashion. This initial partitioning phase is *metric independent* and needs to be executed only once, i.e., even in the case of arc-weights changes or for different metrics. Following partitioning, the *customization stage* builds the overlay graph H containing all boundary vertices and arcs of G. The graph H also contains a clique for each cell c: for every pair (u, v) of boundary vertices in c, we create a shortcut arc (u, v) whose cost is the same as the shortest-path (restricted to inner edges of c) between u and v (see Fig. 1(b), 1(c)). Similar to [11], we also calculate the SP distances between all border vertices of level ℓ and all vertices of level $\ell-1$ within each cell c^{ℓ} (see Fig. 1(d)). To differentiate between the two kinds of arcs computed, we will denote as (i) *clique arcs* the added overlay arcs that connect border vertices of the same level ℓ and (ii) *downward arcs* of level ℓ the vertices connecting different levels, i.e., ℓ and $\ell-1$. For added efficiency, *downward arcs* are stored as a separate graph, referred to, as $G_{GS}\downarrow$. Both types of arcs are computed bottom-up and starting at level one. To process a cell, the GS customization stage for SALT executes a Dijkstra algorithm from each boundary vertex of the cell. We also apply the arc-reduction optimization of [12], which reports only distances of boundary vertices that are direct descendants of the root of each executed Dijkstra algorithm.

Although SALT's GS preprocessing phase is similar to GRASP, there are two major differences. (i) In SALT, H and $G_{GS}\downarrow$ have the same number of levels ($L = 6$ in our experiments) with $|C^L| = 16$ (cf. the original GRASP paper with $|C^L| = 128$ and $L = 16$). Using a smaller number of cells at the upper level slightly lowers one-to-all query parallel performance, but *accelerates point-to-point queries and reduces preprocessing time*. Hence, it is a very logical compromise, since our focus is on increased versatility. (ii) Moreover, we have to repeat SALT's GS customization stage twice, one for the forward and one for the reverse graph. This is necessary for the landmarks phase of SALT, but it also allows to answer, both, forward and reverse single-source queries. Thus, at the end of SALT's GS preprocessing we have built two versions of the overlay graphs, H and $G_{GS}\downarrow$, one for forward and one for reverse graph queries, respectively.

The *landmarks preprocessing phase* for SALT extends the preprocessing proposed by [10], which optimized and tailored the ALT algorithm for dynamic road networks. Landmarks are selected by the *partition - corners* landmarks selection strategy, in which we use the cells created by Kafpaa and from each cell we select the four corner-most vertices as landmarks. For SALT, we accelerate the computation of distances of all graph vertices from and to landmarks by executing two sequential GRASP algorithms (forward and reverse) instead of using plain Dijkstra (as in all previous approaches). Moreover, we may perform those $2\times|S|$ GRASP algorithms in parallel. By using these optimizations, the landmarks preprocessing phase of SALT never takes more than $4s$ *for 24 landmarks* and is therefore *at least* $6\times$ *faster than any existing work*.

Thus, at the end of the preprocessing stage of SALT, we have built the overlay graphs H and $G_{GS}\downarrow$ for both forward and reverse searches and calculated distances for all vertices from and to the selected landmarks. For *dynamic road networks*, we only need to repeat the GS customization stage and the computation of distances of all vertices from and to the landmarks. Both phases *require less than* $19s$ for the benchmark road networks we used. This makes SALT suitable for dynamic scenarios, as well.

3.2 Single-pair shortest-path queries

Using SALT's preprocessing data, we can accelerate single-pair SP queries by our SALT-p2p algorithm, that combines CRP (with arc-reduction) with the ALT's adaptation with SIMD instructions of [10]. In CRP, to perform a SP query between s and t, Dijkstra's algorithm must be run on the graph consisting of the union of H, $c^0(s)$ and $c^0(t)$. The difference in SALT-p2p is that, instead of Dijkstra, we use the ALT-SIMD algorithm on the aforementioned graph. Note that both ALT and CRP may also be used in a unidirectional or a bidirectional setting. A similar combination of CALT [2] and CRP was unofficially introduced in [4], which uses the landmark lower-bounds strictly on the upper-level of the GS overlay graph. Thus, *local searches* could not be accelerated. Local search is crucial for kNN queries, since the kNN results for small values of k are usually located close to the query location. In contrast, our *SALT-p2p algorithm*, combining the ALT-SIMD algorithm of [10] and CRP (with arc-reduction), will be much more efficient than stand-alone ALT or CRP. Moreover, since both methods are extremely robust to the metric used [2, 4], their combination will provide excellent performance for both travel times and travel distances.

Theorem 1. *The SALT-p2p algorithm is correct. (Proof omitted for space restrictions)*

3.3 *k*NN queries

SALT's preprocessing data may also be used to answer *k*NN queries. Instead of initiating a *k*NN search from a query location s to objects O, we start a search *from* all the objects at the same time *to* the query location in the reverse graph. Hence, we take advantage of, both, GS and ALT acceleration for guiding the search towards the query location. The SALT-kNN algorithm's query phase is divided in two independent stages. The *Pruning phase* excludes objects that cannot possibly belong to the *k*NN set by using the upper and lower-bounds provided by the landmarks preprocessing data. The *Main phase* executes a unidirectional SALT-p2p algorithm in the reverse graph from all remaining objects at the same time to the query location until the query location is settled. Now we have found the first nearest-neighbor. This process has to be repeated another $k - 1$ times until all *k*NN are discovered. The algorithm is detailed in the following.

Pruning phase. To prune objects that cannot belong to the *k*-nearest neighbors set, we must (i) calculate the *k*-th lowest upper-bound of graph distances between the query location and the objects (cf. Equation 2) and (ii) exclude objects whose distance lower-bounds between them and the query location (cf. Equation 1) exceed the *k*-th lowest upper-bound. To the best of our knowledge, this is the *first work to utilize upper and lower landmark bounds in the context of kNN queries*.

Theorem 2. *SALT-kNN's pruning phase is correct. (Proof omitted for space restrictions)*

For computing the *k*-th lowest upper-bound between the query location and the objects we use a bounded max-heap Q of size k and procedure GETKTHLOWUPBOUND:

```
GETKTHLOWUPBOUND($s, O, landDist$)
1  $Q = emptyMaxHeap$
2  $m = 0$
3  for each $o$ in $O$
4      if $m < k$
5          $Q.push(upperBound(s, o))$
6          $m = m + 1$
7      elseif $(upperBound(s, o) < Q.top())$
8          $Extract - max(Q)$
9          $Q.push(upperBound(s, o))$
10 return $Extract - max(Q)$
```

```
PRUNEPHASE($s, O, landDist$)
1  $O_{small} = \{\}$
2  $kBound = getKthLowBound(s, O, landDist)$
3  for each $o$ in $O$
4      if $lowerBound(s, o) \leq kBound$
5          $O_{small}.add(o)$
6  return $O_{small}$
```

Since the bounded max-heap Q only stores *k*-upper-bound distances, we only need to compare the next objects's upper-bound with the top of the heap. If we have found a lower upper-bound, we remove the top of the heap and add the new upper-bound to Q. At the end of the procedure, the top of the max-heap is the *k*-th lowest upper bound of distances between the query location and the objects.

At the end of the pruning phase (see procedure PRUNINGPHASE), instead of using the objects in O, we only need to check for the *k*-nearest neighbors within the objects in O_{small}. Our experimentation has shown that the pruning phase is very effective, since it efficiently prunes more than 60% of the total number of objects in O.

Main phase. Following the pruning phase, to find the first nearest-neighbor we start by performing a search simultaneously from all objects in O_{small} to the query location s in the reverse graph. To do so, we use the idea of [16]. We add a new vertex T' connected to all objects in O_{small} using zero-weight edges and then perform a unidirectional SALT-p2p algorithm from T' to s in the reverse graph. At the end of this process, we have found the first NN of query location s. Then we eliminate this vertex from O_{small} and repeat for another $k-1$ iterations to retrieve the full *k*NN set (see procedure MainPhase).

Theorem 3. *SALT-kNN's main phase is correct. (Proof omitted for space restrictions)*

```
MainPhase($s, O_{small}, k, \overline{G}$)
1 for $i = 0$ to $k-1$
2     $T' = \text{newVertex}$
3     for each $o \in O_{small}$
4         Conn. $T'$ to $o$ with 0-weight edges
5     $(iNN, iNNdist) = SALT-p2p(T', s, \overline{G})$
6     $O_{small} = O_{small} - iNN$
```

To retrieve not only the SP distance between the query location s and the objects in O_{small} but also the actual *k*NN vertices, we need to maintain for each labeled vertex a reference that points to the originating vertex in the objects' set O_{small}. Thus, when we extract s from the priority queue and terminate the SALT-p2p algorithm at the *i*-th iteration, we know not only the *i*-th SP distance but the *i*-th NN as well. Moreover, for each object o in O_{small}, we need to store the cell ID $c^1(o)$ of the cell this object belongs at the lowest level of the GS hierarchy, to traverse the overlay graph H during each iteration of the SALT-p2p algorithm. Note it is sufficient to store only the $c^1(o)$, since cell IDs for higher levels may be calculated from that.

Although SALT-*k*NN will be very fast for retrieving the first NN object, it will become progressively slower when retrieving the additional $k - 1$ NN, since at each iteration, the SALT-p2p algorithm will start from scratch. To remedy this, at the beginning of the *i*-th iteration, we reload the corresponding priority queue with all vertices labeled during the $i-1$ iteration except those originating from the previous NN vertex found, since most of those labeled vertices were already assigned correct SP distances. Since we use a min-heap priority queue (as all Dijkstra variants), this optimization significantly improves query times and still ensures correctness of the SALT-*k*NN algorithm.

3.4 Summary and Expectations

Although SALT is very efficient for most SP queries, the main phase of SALT-*k*NN could be performed with any valid unidirectional SP algorithm. However, using SALT-p2p has multiple benefits: (i) Its constituent algorithms, ALT and CRP have very fast preprocessing times suitable for *dynamic road networks*. (ii) Unidirectional SALT-p2p provides better performance than bidirectional SALT-p2p, contrary to existing hierarchical methods that may only be used in a bidirectional setting. (iii) SALT-p2p and hence SALT-*k*NN are very *robust to the metric used*. This is an important property for *k*NN queries identifying Points-Of-Interest (POIs) based on walking distance. (iv) SALT-*k*NN's pruning phase is very crucial for a fast implementation. *Only the landmarks preprocessing data could provide this type of functionality*. (v) Lastly, the main phase of the SALT-*k*NN algorithm initially expands vertices closer to the query location s. As such, "unattractive" objects furthest from s (as estimated by the lower-bounds) that cannot be excluded during the pruning phase, do not slow down SALT-*k*NN queries. In fact, experiments showed that finding the first NN is as fast as a plain SALT-p2p query. Hence, it is hard to provide a much better theoretical solution, using standard SP techniques, with fast enough preprocessing times suitable for dynamic road networks.

4 Experiments

The experimentation that follows, assesses the performance of the SALT-p2p and SALT-*k*NN algorithms. For completeness, we also report the performance of sequential and parallel GRASP [11] algorithm within the SALT framework for single-source (one-to-all) queries. Experiments were performed on a workstation with a 4-core i7-4771 processor clocked at 3.5GHz with 32Gb of RAM, running Ubuntu 14.04 64bit. Our code was written in C++ and GCC 4.8 (with OpenMP). Query times are executed on one core and augmented with SSE instructions. We used the European road network (18M vertices / 42M arcs) and the full USA road network (24M vertices / 58M arcs) [8] and experimented with both travel times and travel distances.

For partitioning the graph into nested-multilevel partitions, similarly to [11], we used Buffoon / KaFFPa [19] in a top-down approach. We use a partitioning setup similar to the best recorded CRP results of [3] with total number of overlay levels set to L=6 and $|C^1|$=1048576, $|C^2|$=65536, $|C^3|$=8192, $|C^4|$=1024, $|C^5|$=128 and $|C^6|$=16. We also used 24 landmarks, since adding more landmarks did not offer significant performance benefits for either SALT-p2p or SALT-*k*NN algorithms.

4.1 Preprocessing

Table 1: SALT, GRASP and G-tree preprocessing

	Preprocessing time (s)			
	Travel Times (TT)		**Travel Distances (TD)**	
	EUR	**USA**	**EUR**	**USA**
SALT (GS customiz.)	11.1 (5.5)	14.82 (7.4)	11.3 (5.7)	15.4 (7.7)
SALT (Landmarks)	2.6 (1.3)	3.6 (1.8)	2.7 (1.4)	3.6 (1.8)
SALT (Total)	13.7 (6.9)	18.4 (9.2)	14.0 (7.0)	18.9 (9.5)
GRASP (Orig)	8 (8)	12 (12)	10 (10)	13 (13)
G-tree	(198,479)	(5,736)	(25,918)	(5,001)

In this section we report the preprocessing times for SALT, in comparison to the original GRASP version [11]) and G-tree [21] (G-tree source code was provided by its authors). Note, that contrary to the SALT framework that may simultaneously answer single-pair, single-source (one-to-all, one-to-many, range) and *k*NN queries, GRASP only focuses on single-source queries and G-tree may only be used for undirected networks and *k*NN queries. SALT and GRASP preprocessing times refer to parallel execution and G-tree preprocessing time is sequential. For GRASP and SALT and its graph-separator subphase we only report preprocessing times for the customization stage, similar to [4] and [11], since this is the preprocessing that must be repeated when arc-weights change, for live-traffic road networks. For a fair comparison, for G-tree we do not report the partitioning time required for the building of the G-tree index (which uses METIS [15]) and we only report the preprocessing time for calculating the SP distances inside the respective index structure. Results are presented on Table 1. Numbers inside parentheses represent preprocessing times for undirected versions of the benchmark road networks.

Results show that: (i) G-tree preprocessing times are very disappointing, especially for Europe and travel times, when more than 24h are required for preprocessing, contrary to SALT's preprocessing time *that never exceeds 19s* for all networks and metrics. (ii) In comparison to GRASP, SALT may calculate both forward and reverse graph SSSP queries. If GRASP was to be extended for reverse graph SSSP queries, its preprocessing time would double and hence *it would be* 16−43% *slower than SALT*.

(iii) SALT's preprocessing time *is very robust to the metric used* and preprocessing time is similar for both metrics. (iv) For undirected versions of the road networks (for comparing results to G-tree), SALT's preprocessing time drops in half, both for the GS customization and landmarks phase. Note that although SALT's total preprocessing time is better than any other previous ALT based approach including [10], the GS customization phase could be potentially further accelerated by using the optimizations of [6], namely SIMD instructions or contraction. Furthermore, SALT's memory requirements still remain quite modest, since it requires less than $8.5Gb$ (including the original graph G) for both benchmark road networks and metrics.

4.2 Single-pair / single-source shortest-path queries

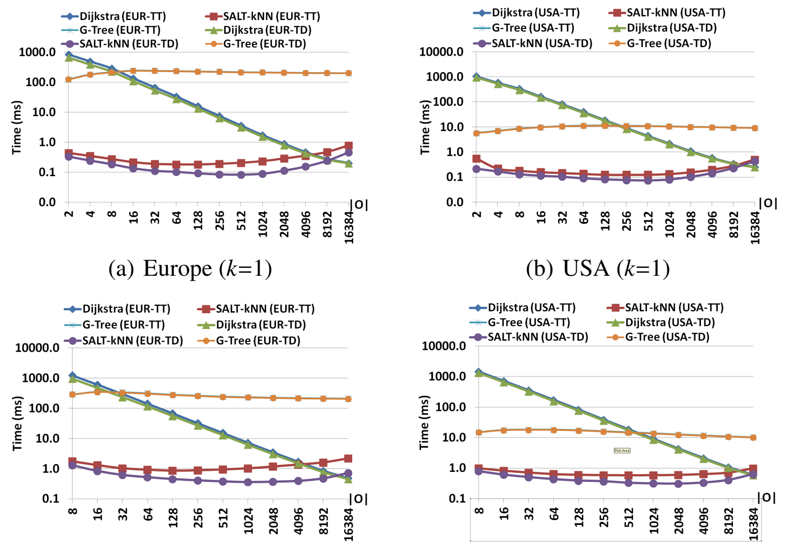
Table 2 compares unidirectional and bidirectional SALT-p2p query performance for single-pair shortest-path (SPSP) queries, compared to its algorithmic components, namely the ALT-SIMD algorithm of [10] and CRP [4] with the arc-reduction of [12], within the SALT framework, for a total of 10,000 queries with the pair of vertices selected uniformly at random. Regarding SSSP queries, we report sequential and parallel performance of GRASP for one-to-all queries within the SALT framework and compare it with the original version of GRASP [11]. For both GRASP versions, the number in parentheses represent sequential times. Results are presented in Table 2.

Results show that: (i) Unidirectional SALT-p2p is always faster than bidirectional SALT-p2p. Thus, to the best of our knowledge, *uniSALT-p2p is the faster unidirectional algorithm for road networks, with preprocessing times of few seconds*. (ii) uniSALT-p2p is $100 - 266\times$ faster than ALT and $3 - 4\times$ faster than CRP. Note that our CRP's query performance is almost identical to the best CRP implementation of [3]. UniSALT-p2p path unpacking (i.e., providing full paths) would also be faster than CRP, since it uses bidirectional ALT instead of bidirectional Dijkstra used by CRP [3]. Moreover, uniSALT-p2p provides comparable performance to recent Customizable Contraction Hierarchies [9] which was only tested on undirected networks. (iii) SALT-p2p is very robust to the metric used. In fact, *uniSALT-p2p is slightly faster for travel distances*.

Table 2: SALT-p2p and GRASP query performance

	SPSP Query times (ms)			
	Travel Times (TT)		**Travel Distances (TD)**	
	EUR	**USA**	**EUR**	**USA**
biALT	103	60	133	89
CRP (+AR)	1.6	1.8	2	2
uniSALT-p2p	0.6	0.6	0.5	0.5
biSALT-p2p	0.9	0.9	0.9	0.9
	SSSP Query times (ms)			
GRASP (Orig)	43 (150)	58 (207)	46 (156)	66 (218)
GRASP (SALT)	50 (169)	65 (224)	53 (175)	68 (228)

For SSSP queries, the GRASP implementation within the SALT framework is $5-12\%$ slower for sequential and $3-16\%$ slower for parallel execution than the original GRASP implementation. Still, it is fast enough for most practical cases and the SALT framework may also execute forward and reverse SSSP queries, which is a considerable advantage. Note, that the slightly less efficient implementation of GRASP within SALT is attributed to the fact that now $|C_L| = 16$ (in comparison to $|C_L| = 128$ in [11]). However, setting $|C_L| = 16$ is the optimal setting for SPSP and kNN queries and thus, we kept the setting that benefits the most frequent type of queries.



(a) Europe (k=1) (b) USA (k=1)

(c) Europe (k=4) (d) USA (k=4)

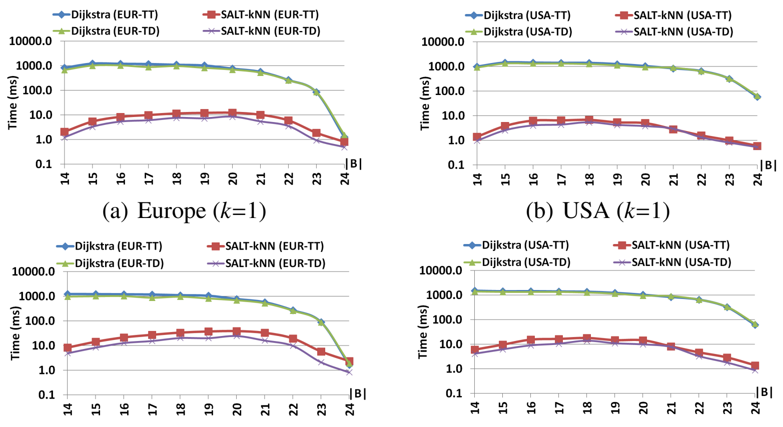
Fig. 2: SALT-kNN Dijkstra and G-tree comparison for varying values of $|O|$.

4.3 kNN queries

Next, we compare SALT-kNN, Dijkstra and G-tree [21] performance for kNN queries. For each experiment, we generate 100 sets of random objects of varying size $|O|$ and for each such set we generate 100 random query locations, for a total of $10,000$ kNN queries per $|O|$. Figure 2 reports average query times for $k = 1$ and $k = 4$. Note, that G-tree requires a *target selection phase*, for each set of objects $|O|$ (requiring $1.9-2.4s$). Thus, contrary to Dijkstra and SALT-kNN, G-tree cannot be used for moving objects.

Results show that SALT-kNN provides stable performance and query times significantly below $1ms$ for k=1. Contrarily, G-tree is almost *two - three orders of magnitude slower* and cannot compete with either SALT-kNN or Dijkstra. Dijkstra starts very slow for small values of $|O|$ but manages to surpass SALT-kNN performance for $|O| > 8192$. These results are similar to [7], where Dijkstra also outperforms online CRP for $k = 10$ and $|O| = 0.01 \times |V|$. Still, since for static points of interest we are usually interested in a specific type of objects (e.g., gas stations) and in the case of moving objects we rarely have such large vehicle fleets (i.e., taxis, trucks) to monitor and we usually aim for kNN queries among the *available* vehicles (a much smaller subset of total vehicles), then the SALT-kNN algorithm is surely to perform better for most practical applications.

After establishing the superiority of SALT-kNN over G-tree, we next evaluate the impact of objects distribution to SALT-kNN and Dijkstra's performance. To this end, similar to [5], we pick a vertex at random and run Dijkstra's algorithm from it until reaching a predetermined number of vertices $|B|$. If B is the set of vertices visited during this search, we pick our objects O as a random subset of B. We keep the number of objects $|O|$ steady at 2^{14} and we experiment with different values of $|B|$ ranging from $2^{14} \dots 2^{24}$, to simulate cases of either: (i) POIs mainly located near the city-center or (ii) vehicle fleets which may service an entire continent but operate mainly on a particular country. Results are presented in Figure 3.



(c) Europe (k=4) (d) USA (k=4)

Fig. 3: SALT-kNN and Dijkstra comparison for $|O| = 2^{14}$ and varying values of $|B|$.

Results show, that SALT-kNN is one - two orders of magnitude faster than Dijkstra when objects are not uniformly located in the road network (as is the typical case, either for static or moving objects). Thus, SALT-kNN guarantees excellent and stable performance, regardless of: (i) the number of objects and (ii) the objects distribution. Moreover, it does not need a target selection phase, such as G-tree or CRP and therefore, it may be used for either static or moving objects. Note, than even without building an index, CRP would still require 10*ms* for the target selection phase and 16,384 objects for the Europe road network [7] and therefore, CRP would *be at least 10 times slower than SALT-kNN for moving objects*.

5 Summary and Conclusions

This work presented SALT, a novel framework for answering shortest-path queries on road networks, including point-to-point, single-source (one-to-all, one-to-many, range) and kNN queries. By combining ideas from the ALT, CRP and GRASP algorithms, the SALT framework efficiently answers point-to-point queries $3-4$ times faster than previous algorithms of similar preprocessing times and answers kNN queries orders of magnitude faster than previous index-based approaches. Moreover, the proposed SALT-kNN algorithm was shown to be especially robust, regardless of the metric used, the number of objects or the distribution of objects in the road network. Hence, the SALT framework presents itself as an excellent solution for most practical use-cases and the best overall solution for real-world applications.

References

1. Bast, Hannah and Delling, Daniel and Goldberg, Andrew and Müller-Hannemann, Matthias and Pajor, Thomas and Sanders, Peter and Wagner, Dorothea and Werneck, Renato. Route Planning in Transportation Networks. Technical report, Microsoft Research, 2014.
2. 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.

3. D. Delling, A. Goldberg, T. Pajor, and R. Werneck. Customizable route planning in road networks. working paper, submitted for publication., 2013.
4. 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.
5. D. Delling, A. V. Goldberg, and R. F. F. Werneck. Faster batched shortest paths in road networks. In *ATMOS*, pages 52–63, 2011.
6. 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.
7. D. Delling and R. F. Werneck. Customizable point-of-interest queries in road networks. *Knowledge and Data Engineering, IEEE Transactions on*, to appear, 2014.
8. 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.
9. J. Dibbelt, B. Strasser, and D. Wagner. Customizable contraction hierarchies. In *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, pages 271–282, 2014.
10. A. Efentakis and D. Pfoser. Optimizing landmark-based routing and preprocessing. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, IWCTS '13, pages 25–30, New York, NY, USA, 2013.
11. A. Efentakis and D. Pfoser. GRASP. Extending graph separators for the single-source shortest-path problem. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 358–370, 2014.
12. 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. ACM, 2012.
13. 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.
14. C. S. Jensen, J. Kolářvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, GIS '03, pages 1–8, New York, NY, USA, 2003. ACM.
15. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, 1998.
16. J. Maue, P. Sanders, and D. Matijevic. Goal directed shortest path queries using precomputed cluster distances. In *5th Workshop on Experimental Algorithms (WEA), Number 4007 IN LNCS*, pages 316–328. Springer, 2006.
17. K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 43–54. VLDB Endowment, 2006.
18. 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.
19. P. Sanders and C. Schulz. Engineering multilevel graph partitioning algorithms. In *Algorithms - ESA 2011*, Lecture Notes in Computer Science, pages 469–480. 2011.
20. H. Wang and R. Zimmermann. Processing of continuous location-based range queries on moving objects in road networks. *Knowledge and Data Engineering, IEEE Transactions on*, 23(7):1065–1078, July 2011.
21. R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In *Proceedings of the 22nd ACM International Conference on Information Knowledge Management*, CIKM '13, pages 39–48, New York, NY, USA, 2013. ACM.