2018 IEEE International Conference on Systems, Man, and Cybernetics

Memory-restricted Routing With Tiled Map Data

Thomas Bläsius*, Jan Eube†, Thomas Feldtkeller†, Tobias Friedrich*, Martin S. Krejca*
J. A. Gregor Lagodzinski*, Ralf Rothenberger*, Julius Severin†, Fabian Sommer†, Justin Trautmann†

***Abstract*—Modern routing algorithms reduce query time by depending heavily on preprocessed data. The recently developed *Navigation Data Standard (NDS)* enforces a separation between algorithms and map data, rendering preprocessing inapplicable. Furthermore, map data is partitioned into tiles with respect to their geographic coordinates. With the limited memory found in portable devices, the number of tiles loaded becomes the major factor for run time.**

We study routing under these restrictions and present new algorithms as well as empirical evaluations. Our results show that, on average, the most efficient algorithm presented uses more than 20 times fewer tile loads than a normal A*.

I. Introduction

Due to the importance of routing applications in real-life situations, finding shortest paths in graphs has been subject to extensive research [1]–[5], [8]–[10], [12]–[15]. Thus, it is not surprising that Dijkstra's algorithm [5] for finding shortest paths is arguably the most famous among all graph algorithms. Although no algorithm with better asymptotic run time is known, domain-specific knowledge can help to speed up shortest path computations in industrial applications. One improvement is to use estimates for the distances (for example, geographic distances in road networks) in order to make the search target-directed [13]. This modification of Dijkstra's algorithm is called A* and reduces the search space by preferring the exploration of vertices presumably closer to the target.

More recent speedup techniques are typically based on the fact that the number of shortest path queries highly predominates the number of graph changes. This allows for a two-phase algorithm: the preprocessing phase computes auxiliary information about the input graph, and the query phase answers shortest path queries, profiting from the the auxiliary data computed before. The research on such preprocessing techniques gathered pace with the public release of large road networks in 2005 as part of the 9th DIMACS Implementation Challenge. With the goal of achieving better query times, a plethora of speedup techniques has been developed. Examples are target-directed techniques such as ALT [10] or Arc Flags [14], [15], which aim for improved estimations of the actual distances or exclude the use of certain edges leading in the wrong direction. Other strategies such as REACH [12] and Contraction Hierarchies [8] exploit the hierarchical nature of road networks by pruning the search at unimportant vertices or by introducing shortcuts. Techniques based on lookup tables such as Transit Node Routing [1] or PHAST [3] yield extremely low query times but require a large amount of auxiliary data.

Besides improving query times, recent research was also impelled by the goal to handle additional challenges appearing in real-world routing scenarios. Examples are the inclusion of turn costs [9] and routing with time-dependent metrics [2]. In order to support real-time traffic updates, the algorithm CRP starts with a metric-independent preprocessing step [4]. Afterward, a new metric (for example, depending on the current traffic) can be incorporated in less than a second.

While preprocessing greatly improves the query times of routing algorithms, it also requires sufficient memory for computing and storing the preprocessed data. Further, the information necessary in order to compute this data needs to be available in the first place. This is, for example, not the case with the *Navigation Data Standard (NDS)*, which was jointly developed by major automobile manufacturers[1] and suppliers[2] and requires a separation between algorithm and data. From a business point of view, this is highly desirable for car manufacturers, as it enables them to obtain the routing software and the underlying graph data from different sources. From an algorithmic point of view, this implies that the routing algorithm has no prior knowledge about the map data. Thus, it has to treat each routing query as a new and unknown shortest path instance, prohibiting any kind of preprocessing.

We consider this scenario under the constraint of restricted memory, as found in mobile devices or built-in navigation systems. Moreover, we assume that the data is subdivided into tiles, as is the case with NDS. This subdivision of the map has the advantage that map changes can be distributed to the user by updating only a few tiles. However, in devices with limited memory, this implies that the same tile is potentially loaded multiple times if the shortest path algorithm explores vertices from the same tile in different stages of the search. In this scenario, the number of tile loads becomes an important factor.

In order to account for these restrictions, we develop routing algorithms without preprocessing and evaluate them with respect to their number of tile loads. We note that this complexity measure is similar to the *parallel disk model* by Vitter and Shriver [18], which focuses on I/O-efficiency and allows to load a block of data with a single I/O-operation. The main difference in our setting is that the tiles prescribe the partitioning of the data into blocks, which cannot be changed in the course of the algorithm. Further, Sanders et al. [17]

* Algorithm Engineering Group, Hasso Plattner Institute, University of Potsdam, Potsdam, Germany, firstname.lastname@hpi.de
† Hasso Plattner Institute, University of Potsdam, Potsdam, Germany, firstname.lastname@student.hpi.uni-potsdam.de

[1]BMW, Daimler, Hyundai, Nissan, Renault, VW, Volvo
[2]For example, Garmin, HERE, and TomTom.

2577-1655/18/$31.00 ©2018 IEEE
DOI 10.1109/SMC.2018.00567

and Goldberg and Werneck [11] also consider routing with restricted memory and propose efficient algorithms. However, their scenarios still allow preprocessing, whereas ours does not.

Contribution and Outline: In Section II, we describe the limitations for routing algorithms motivated by NDS and briefly introduce basic concepts related to Dijkstra's algorithm and A*, which form the basis of our algorithms. Different strategies in order to reduce the number of tile loads are presented in Section III. We extensively evaluate our algorithms in Section IV. Compared to a simple A* algorithm, we decrease the number of tile loads on a reasonably small device by a factor of 20. A discussion and outlook in Section V concludes the paper.

II. Preliminaries

In this section, we first describe the limited information that a routing algorithm has access to in our scenario. We then explain the basic concept of routing with no preprocessing.

A. Modeling Tiled Map Data

We consider road networks whose sets of nodes are partitioned into disjoint regions with respect to certain longitudes and latitudes. We call a set of nodes that belong to the same region a *tile*. Similar to NDS, we assume that the tiles have an extent of $\frac{360^\circ}{2^{14}}$ in each dimension, yielding squares sized about $6\,\mathrm{km}^2$ in equatorial regions and rectangles of about $3.7\,\mathrm{km}^2$ in Europe.

We model such a road network as a directed edge-weighted graph. Each node of the graph stores its geographic coordinates as well as all of its outgoing and incoming edges. Moreover, each node has a unique ID, referencing the tile containing the node. An edge stores a positive weight and its two incident nodes' IDs. Note that an edge does not store the coordinates of its incident nodes.

B. Basic Shortest Path Algorithms

Dijkstra's algorithm [5] forms the basis of many algorithms for finding shortest paths in graphs with non-negative weights. Given a source node s and a target node t, it iteratively computes the shortest paths to intermediate nodes until t is reached. The algorithm conducts by using labels for every node, which indicate the length of a shortest path from s to that respective node found so far. The nodes are stored in a priority queue that orders its elements by their label. Initially, this queue only contains s.

In each iteration, a node v with highest priority (smallest label) is removed from the queue and *expanded*. That is, for all nodes adjacent to v, it is checked whether a path via v is shorter than the shortest path known so far for that node. If so, such a node's label is updated and the node is added to the queue if not already present.

The correctness of Dijkstra's algorithm follows from its property that a node's label is final once it is removed from the queue – that is, the shortest path from s has been found, and the node will never be reintroduced into the queue again. This property is known as *label-setting*. Thus, if t is removed from the queue, the algorithm terminates and its complexity is determined by the number of intermediate nodes explored.

In order to make Dijkstra's algorithm target-directed, one can reorder the queue such that nodes presumably closer to t get expanded first. Thus, the queue is not necessarily ordered by shortest distance from s anymore, and nodes removed from the queue may be reintroduced later on if a shorter path is found. This property is known as *label-correcting*. Note that a label-correcting algorithm still terminates on a finite graph, as each node can only be reintroduced into the queue a finite number of times. Such an algorithm terminates once the shortest known path from s to every vertex in the queue is longer than the currently best path to t.

A beneficial way to reorder the queue is to introduce a heuristic for every node that estimates the (non-negative) distance from this node to t. A node's label is then the sum of its shortest path found so far and its estimate. This feature is prominently displayed in the algorithm A* [13]. Note that A* may expand far more nodes than Dijkstra's algorithm if the heuristic is bad, since many nodes may be reintroduced into the queue. However, if the heuristic is *admissible* – that is, the actual path length of a node to t is never overestimated –, Dijkstra's original termination criterion of removing t from the queue becomes sufficient (as the estimate of t has to be 0).

A* with an admissible heuristic is not necessarily label-setting, as intermediate nodes can still be added to the queue multiple times. In order to make A* label-setting, the heuristic h has to be monotone. That is, for any two adjacent nodes v_1 and v_2, the triangle inequality $h(v_1) \leq h(v_2) + d(v_1, v_2)$ needs to hold, where $d(v_1, v_2)$ denotes the weight of the edge from v_1 to v_2.

A typical estimate for $h(v)$ is the geodesic distance between v and t. If the objective is travel time, $h(v)$ can be determined by dividing the geodesic distance between v and t by a maximum speed value of $130\,\mathrm{km/h}$. We use this term in both cases, noting that the constant factor introduced for the travel time does not change admissibility or monotonicity.

III. Algorithmic Framework

Dijkstra's algorithm provides a good baseline for our scenario, since we cannot preprocess our data. However, we have the additional constraint of limited memory (the *cache*), which restricts the number of loaded tiles. Hence, we introduce the following approaches:

1. Using a heuristic based on the nodes' coordinates. Inspired by A*, we study heuristics that estimate the distance from a node to the target yielded by the coordinates of both nodes.

2. Prioritizing tiles in the cache. Some nodes in the queue may belong to tiles that are cached while others belong to tiles that are currently not in the cache. It serves the intuition to give nodes in cached tiles a higher priority than other ones.

3. Loading new tiles that are close by. Over time, the nodes in the queue may become geographically scattered. In order to achieve a more compact search space, we choose a

tile that is close to the currently cached nodes whenever a new one has to be loaded.

4. Replacing tiles in the cache. When loading a new tile, we may have to remove tiles from the cache. Hence, we discuss two caching strategies.

5. Combining different strategies. Combining the strategies mentioned above arbitrarily does not always make sense. We describe the different possible combinations and introduce a naming scheme (see Figure 2) used in Section IV.

A. Estimating the Distance to the Target

In order to reduce the size of the search space, nodes that are supposedly closer to the target t can be expanded first. A* does so by using a heuristic that estimates the distance of each node to t. A common admissible and monotone heuristic used for A* is the geodesic distance between a node v and t. However, there is a major obstacle in our scenario: in order to calculate the geodesic distance between v and t, their respective coordinates have to be accessed. Hence, if the tile T that v belongs to is currently not cached, it has to be loaded, resulting in an expensive calculation of the estimate of v.

We circumvent this problem by introducing a new heuristic involving a case distinction between T being currently cached or not. We call cached tiles and their nodes *accessible*.

First, note that it is sufficient to apply the heuristic for nodes in the queue. Assume the node v is inserted into the queue due to the expansion of its neighbor u. We call u the *parent* of v. If v is accessible, we compute the geodesic distance to t. Otherwise, we use the estimated distance between u and t and assume v to lie on this geodetic line. Since we know the distance between u and v, we subtract this value from the estimate of u without loading T. Refer to Figure 1a for a visual representation of this case.

This new heuristic h' is formalized as below. Here, h denotes the geodesic distance of a node to t and d denotes the weight of an edge.

$$h'(v) = \begin{cases} h(v) & \text{if } v \text{ is accessible,} \\ \max\{0, h(u) - d(u,v)\} & \text{else.} \end{cases}$$

Since h is admissible, the same holds for h'. However, it is not necessarily monotone, since the estimate of a node v highly depends on its current parent if T is not cached at that point in time. Furthermore, the estimate of v is updated every time the parent node is replaced. Thus, applying h' results only in a label-correcting algorithm. Nevertheless, our results show that the labels of the explored nodes get updated very rarely (the number of node explorations increases by less than $0.1\,\%$ in comparison to A*), suggesting that the algorithm is almost label-setting. One possible explanation for this behavior is that, given a node v and its parent u, the new estimate $h'(v)$ can differ from $h(v)$ by at most $2d(u,v)$, which is typically not very large (in particular compared to $h(v)$). Thus, $|h'(v) - h(v)|$ will be rather small, which results in the algorithm being almost label-setting.

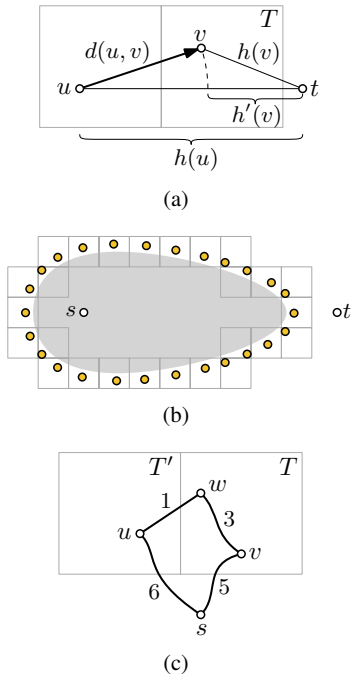

Fig. 1: **(a)** Illustration of how to compute an alternative heuristic $h'(v)$ depending on $d(u,v)$ and $h(v)$ but not on the distance $h(v)$ from v to the target t. **(b)** A possible state of a search from s to t with the already explored search space (gray region) and the vertices currently in the queue (yellow vertices). If all vertices in the queue have the same label, all displayed tiles have to be loaded before one tile is considered a second time. **(c)** An example where the tile-exhaustive search has to reload the tile T. The vertex u outside of T is expanded after v, and the shortest path from s to w has to go through u. If T' is not present in the cache, T may has to be discarded before T' is loaded.

B. Prioritizing Tiles in the Cache

Over time, the *search front* (the vertices currently in the queue) can become geographically large even with the target-directed A* search. Moreover, the search space typically expands rather uniformly. See Figure 1b for an exaggerated example. With respect to the cache efficiency, this has the effect that many tiles are relevant at the same time: after expanding the search space in one tile T, the algorithm potentially expands the search space in all other tiles along the search front before coming back to T. If T has been replaced from the cache by then, this results in an expensive reload.

In order to mitigate this effect, we use a strategy that has already been employed successfully by Edelkamp and Schrdl [6], [7] and decreases cache misses: as long as the queue contains accessible vertices, we choose an accessible vertex with the smallest label for the next expansion step. If no vertex in the queue is accessible, we have to load a new

tile and thus simply choose a vertex with the smallest label. We call an algorithm *tile-exhaustive* if it uses this strategy.

Note that this drastically changes the order in which the vertices are expanded, yielding the following two issues: first, expanding vertices only because they are accessible could lead the search in the wrong direction, increasing the search space. Second, expanding a vertex v with a high label while the queue contains vertices with much smaller labels increases the chance that the shortest path to v has not been found yet. In this case, v needs to be reexpanded later. This increases the run time and might even lead to an additional tile load (in case the tile of v has been discarded by then). In the following, we shortly discuss why these are not major obstacles for realistic road networks. Our results in Section IV support these observations.

Concerning the first issue, we assume that new tiles are loaded only if a vertex in the tile is expanded. This ensures that the tile includes at least one vertex that would have been expanded even without reordering the queue. All other vertices in the tile are geographically close. Thus, it is likely that they would have been expanded not much later, which indicates that expanding the whole tile immediately does not increase the search space by a large amount. If above assumption does not hold, this argument does not hold anymore. For instance, because we load a tile only to access vertex coordinates for estimating their distance to the target. Hence, we always use the estimation proposed in Section III-A when applying A* together with the prioritization of tiles in the cache. For more details on the combinations of different techniques, see Section III-E.

Concerning the second issue, we can argue similarly: assume the tile T is loaded to expand the vertex v. Then v has the minimum label among all vertices in the queue. Therefore, if we assume our base algorithm to be label-setting, we know the shortest path from the source s to v. It is then reasonable to assume that other vertices in T, which are geographically close to v, either have their final label or obtain it by exploring T. If this is the case, we never have to reexplore T. In fact, the situation in which we have to reload T is rather special. See Figure 1c for an example with specific path lengths. Our results in Section IV show that this does not happen too often.

C. Loading New Tiles That Are Geographically Close

The discussion from the previous section indicates that it is desirable to have geographically close tiles in the cache. For instance, consider the situation in Figure 1c: if both tiles T and T' are in the cache, then we find a shortest path to w by only exploring accessible tiles. Thus, the tile T does not have to be loaded multiple times. In this section, we propose a strategy for loading new tiles into the cache that encourages geographical compactness.

Recall from the previous section that a new tile is loaded only if the queue contains no accessible vertices when employing a tile-exhaustive strategy. The default behaviour in this situation is to load the tile containing a vertex with the smallest label. In order to achieve a certain geographical closeness of tiles in cache, we slightly decrease the labels of vertices that are close to accessible tiles. More precisely, let T be a tile that is not in the cache but adjacent to an accessible tile. Let v be a vertex in T with the smallest label. Then we give a boost to v by decreasing its label, which increases the chance that T is loaded next. For this purpose, we multiply the label of v with a constant $0 < \ell < 1$. Furthermore, if T is no longer adjacent to an accessible tile, the priority boost of v no longer applies.

The exact impact of this strategy heavily depends on the exact value we choose for ℓ. If ℓ is close to 1, almost nothing changes. If ℓ is close to 0, certain vertices are explored much earlier than normally, which increases the odds that they have to be explored multiple times, possibly even after the tile is removed from cache.

Our experiments indicate that ℓ close to 1 yields the desired geographic compactness of accessible tiles while the negative effects are negligible.

D. Cache Replacement Strategies

Once the cache is full and a new tile has to be loaded, it is necessary to determine a tile to be replaced. A straightforward approach is to employ the general-purpose caching strategy *LRU (least recently used)*, which replaces the tile that was not accessed for the longest time. In this section, we introduce a domain-specific caching strategy, exploiting the accessible additional information about the tiles.

Considering a label-setting algorithm, the best candidate tiles for removal are those whose nodes have all been expanded. Such tiles will never have to be reloaded again, since their nodes' labels are final. Also for a label-correcting algorithm that is almost label-setting, as discussed in Section III-A, it is reasonable to remove such tiles.

In contrast to that, tiles that still contain unexplored nodes should not be removed, since these tiles will very likely be needed soon. Even when we utilize a tile-exhaustive algorithm as described in Section III-B, it is possible that a new tile has to be loaded even though there is an accessible tile containing unexplored nodes. This situation occurs if the graph induced by that tile is not connected. A typical instance in which this happens is a highway that cannot be accessed from smaller streets next to it without leaving the tile, as the closest entrance ramp lies in an inaccessible, different tile. However, note that the smaller streets next to the highway are probably not much farther away from the source s than the highway. Thus, chances are high that this tile will be reexplored soon. Hence, it makes sense to preferably leave tiles with unexplored nodes in the cache. If there is a tie (that is, there are multiple or no completely explored tiles), we apply LRU as a tiebreaker.

E. Combinations of Approaches

In this section, we discuss which combinations of the previously mentioned properties make sense. Additionally, we introduce a naming scheme to identify the different combinations. This scheme will also be used throughout the remainder of the paper, in particular in the evaluation in Section IV. The different combinations as well as the naming scheme are illustrated in Figure 2.

The two algorithms forming the basis of our other routing strategies are Dijkstra's Algorithm (abbreviated with DA) and A* (which itself is based on DA). When using the heuristic introduced in Section III-A, we call the resulting algorithm A$^+$. We can make these algorithms (DA, A*, and A$^+$) tile-exhaustive by prioritizing the exploration of accessible tiles, as described in Section III-B. We indicate that an algorithm is tile-exhaustive by adding the suffix TE.

Note that Figure 2 includes the tile-exhaustive variants of DA (DATE) and of A$^+$ (A$^+$TE), but not of A* (A*TE). Although it is theoretically possible to make A* tile-exhaustive, applying A*TE can be very inefficient, since the geodesic distance heuristic employed by A* depends on the node's coordinates. In order to know those coordinates, the node's tile has to be loaded. Therefore, when expanding a node, all of its neighbors' tiles have to be loaded to calculate their geodesic distances to t. These tiles then reside in the cache and are exhaustively explored by the tile-exhaustive strategy. When the algorithm expands nodes on the borders of these tiles, they might again have neighbors in other tiles. This can result in a cascade where A*TE unnecessarily explores a huge portion of the graph. We also observed this behavior in preliminary experiments. Thus, we disregard A*TE entirely in the remainder of the paper.

As discussed in Section III-C, it is beneficial for tile-exhaustive algorithms if the accessible tiles are geographically close. We call an algorithm that prefers loading new tiles close to accessible tiles *local*. In the abbreviations used in Figure 2 and in Section IV, we denote local algorithms with the prefix L. As explained in Section III-C, the local strategy is only meant to be used together with the tile-exhaustive strategy. Its main purpose is to always have a compact search front which can be explored exhaustively, thus rendering reloads of tiles more unlikely. Therefore, we consider the local strategy only together with the tile-exhaustive strategy, yielding the two variants LDATE and LA$^+$TE.

If the algorithm uses the domain-specific cache replacement strategy of removing tiles with only explored nodes before tiles with unexplored nodes discussed in Section III-D, we denote it with the suffix R (for replacement strategy). Note that the cache replacement strategy can be used independently of all other improvements. However, if the algorithm is not tile-exhaustive, then the cache rarely contains tiles with only explored nodes. Thus, the replacement strategy often degenerates to LRU. Therefore, we only consider it together with the tile-exhaustive strategies. Moreover, as the domain-specific replacement strategy has only a small effect on the performance, we omit the less interesting variants that are not local (that is, we omit A$^+$TER and DATER and consider only LA$^+$TER and LDATER).

IV. Experiments

The two main questions we want to answer with our experiments are the following: how well do we reduce the number of tiles loaded by employing our strategies from Section III, and do these strategies increase the overall run time indicated by the number of accessed nodes?

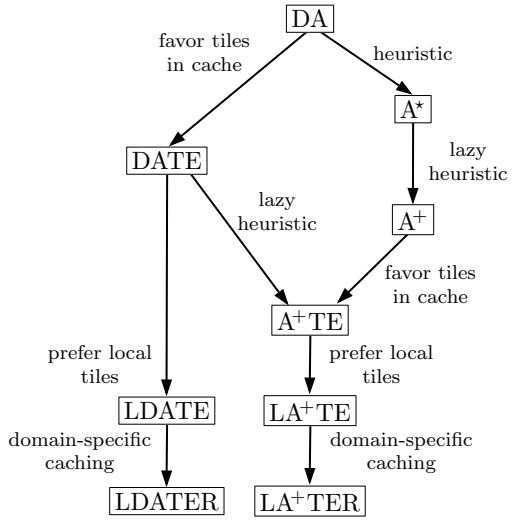

Fig. 2: Overview of the combinations of approaches.

One factor that might influence the overall run time of our algorithm is computing the actual solution after computing the cost of a shortest path. This can be done by basically going backward through the solution in a way akin to dynamic programming: if the destination t has distance $\mathrm{dist}(s,t)$ from s, then t must have a neighbor t' such that $\mathrm{dist}(s,t') + d(t',t) = \mathrm{dist}(s,t)$, where $\mathrm{dist}(s,t')$ is the distance from s to t' and $d(t',t)$ is the length of the edge (t',t). Then we know that the shortest path from s to t includes the edge (t',t), and it remains to reconstruct the shortest path from s to t', which can be done analogously. Concerning our setting, this clearly leads to additional tile loads. However, the number of tiles loaded by constructing the actual path is always the same, independently of our routing strategy. We thus do not think it makes sense to include this constant overhead into our experiments.

A. Setup

We evaluate our algorithms on the PTV road graph[3] subdivided into tiles as explained in Section II. For our experiments, we sampled 10,000 pairs of source and target nodes and computed a shortest path with each algorithm shown in Figure 2. For the local strategy, we used $\ell = 0.99$ as a boosting factor, since this value produced good results in preliminary experiments. Most of our experiments use a cache size of 500 tiles in order to model highly restricted memory. While, this might seem rather small for a state-of-the-art cache, only a small fraction of a device's memory can be used by the routing algorithm, as other processes need memory, too. Further, car manufacturers are known for saving every cent they can, which leads to surprisingly small main memories of on-board devices. However, we also ran experiments on vastly varying cache sizes (between 10 and 10,000 tiles). The corresponding results are reported in Figure 3.

[3]The PTV graph can be obtained for research purposes at i11www.iti.uni-karlsruhe.de/resources/roadgraphs.php.

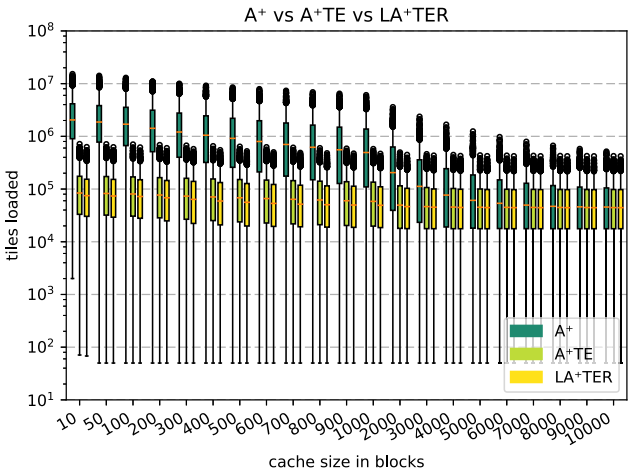

Fig. 3: Number of tiles loaded compared among different cache sizes for A^+, $A^\star$TE, and LA^+TER (from left to right).

We present our results as box plots. The boxes depict the interquartile range (middle $50\,\%$), where a line denotes the median. We let the whiskers extend 1.5 times the interquartile range beyond that. Outliers are shown individually.

B. Experimental Results

For the number of tiles loaded compared among all algorithms, see Figure 4a. One can observe that the $A^\star$ or A^+ variants perform significantly better than their DA counterparts, which is mainly due to a smaller search space. Moreover, the tile-exhaustive strategy significantly decreases the number of tile loads. It yields the strongest improvement among all proposed strategies, reducing the number of tiles loaded by a factor of roughly 14 when applied to A^+ and by a factor of 11.9 when applied to DA. Comparing the number of loaded tiles to the number of distinct tiles, each apportioned by the path lengths, we observe that this is already close to optimal. Note that the number of distinct tiles loaded is a lower bound for the number of tiles loaded unless we are able to reduce the search space somehow. Figure 4c implies that A^+ still has potential in this regard (on average, each tile is loaded 21.87 times), whereas Figure 4d shows that the tile-exhaustive strategy rarely has to load the same tile multiple times (each tile is loaded 1.56 times on average). Although our other strategies have a much lesser impact, they reduce this factor further down to 1.43 and 1.3 for LA^+TE and LA^+TER, respectively.

Refer to Figures 4e and 4f in order to investigate more closely how preferring local tiles and using our domain-specific caching strategy decreases the number of tile loads. Preferring local tiles has a minor impact for short roads, but the impact is increasing for longer roads. This can be reasoned about as follows: for long-distance queries, the search space as well as the search front become larger. Therefore, the accessible tiles are more likely to be scattered along the search front, and preferring local tiles has an impact. Contrary, independent of the path length, the domain-specific caching strategy slightly reduces tile loads.

Concerning our second question of our strategies' impact on the number of processed nodes, refer to Figure 4b. The $A^\star$ variants naturally load fewer nodes than their DA counterparts, due to their smaller search space. Furthermore, our strategies for decreasing tile loads only yield a minor impact on the number of processed nodes. For instance, LA^+TER processes about $15.5\,\%$ more nodes than $A^\star$ (which is label-setting). In other words, LA^+TER processes on average each node at most 1.155 times. Thus, it is almost label-setting.

Overall, combining all of our different strategies (LA^+TER) decreases the number of tile loads by a factor of 34.93, compared to Dijkstra's algorithm, and by a factor of 20.77, compared to $A^\star$. Additionally, the number of processed nodes is increased by only 1.155, compared to $A^\star$. This significant decrease in tile loads strongly outweighs the small increase in the number of processed nodes. Our results for a cache size of 500 tiles qualitatively carry over to other cache sizes as well, as is evident from Figure 3. Again, we see that the tile-exhaustive strategy has the most impact, which is even bigger for smaller cache sizes.

V. Conclusion and Outlook

Due to the separation between data and algorithm motivated by NDS, many state-of-the-art algorithms are not applicable. Moreover, the tiled data in addition with memory restrictions yield new challenges for computing shortest paths; in particular, the number of tile loads becomes the major factor for run time. In this paper, we introduced and evaluated several strategies for handling these challenges. Without increasing the number of processed nodes significantly, the presented algorithms were able to reduce the number of tile loads by a factor of 34.93, compared to Dijkstra's algorithm, and by a factor of 20.77, compared to $A^\star$.

There are three different ways to further improve our results: first, improving the strategies for cache replacement and loading new tiles. Second, reducing the search space, for example, with a bidirectional search. Third, setting up scenarios more similar to realistic situations. In the following, we discuss these three possibilities in greater detail.

In order to improve the strategy of loading new tiles such that accessible tiles are geographically close to each other, one could consider a range of influence extending beyond the direct neighbors. For the domain-specific caching, we only used the information whether or not all nodes of a tile are already expanded. Additionally, we also have domain-specific information such as the geographic positions and the current labels. These could help improve the cache replacement strategy. However, the possible improvements are rather limited, as LA^+TER already rarely reloads tiles. This directly leads to the second way for improvement: reducing the search space.

One technique in order to reduce the search space is bidirectional search [16]. We note that all improvements presented in our paper can easily be applied to bidirectional algorithms. However, the setting of memory-restricted routing

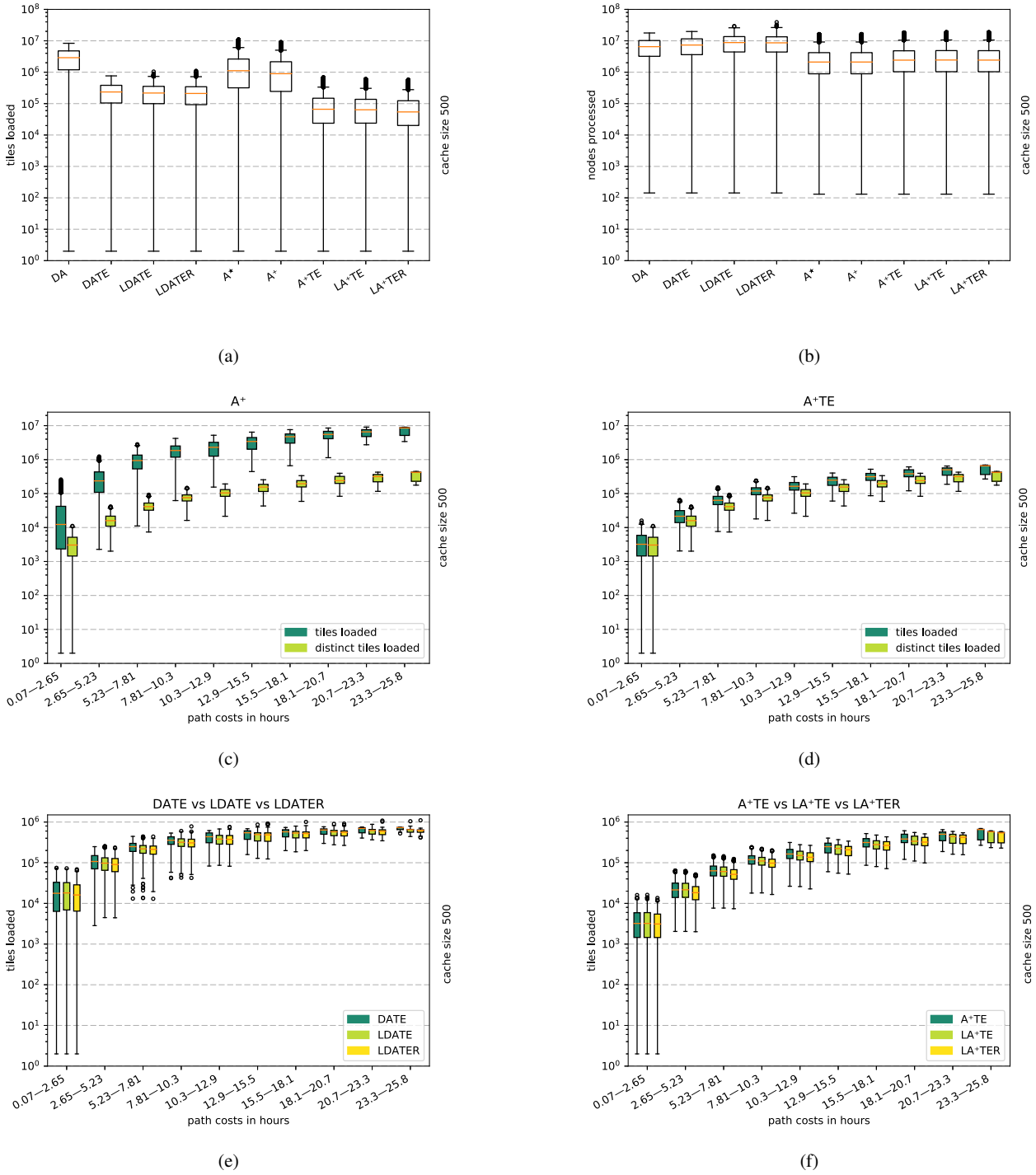

Fig. 4: Experimental performance measures for different algorithms. **(a)** Tiles loaded for all algorithms. **(b)** Nodes processed for all algorithms. **(c)** Tiles loaded compared to search space in tiles for A$^+$, by different route lengths. **(d)** Tiles loaded compared to search space in tiles for A$^+$TE, by different route lengths. **(e)** Tiles loaded for DATE, LDATE, and LDATER for routes of different length. **(f)** Tiles loaded for A$^+$TE, LA$^+$TE, and LA$^+$TER for routes of different length.

on tiled data yields new problems for bidirectional search. The major difficulty results from both searches competing for cache when running alternatingly. Another reason why we did not consider bidirectional search is that routing data is typically time-dependent. Thus, it would be necessary to know the arrival time in advance in order to perform the backward search.

Although the considered setting is already rather close to actual industrial instances, there are certain aspects we idealized. For example, we assumed each tile to occupy the same amount of space in the cache, whereas real-world data is potentially more heterogeneous. For instance, one could consider that removing a large tile, on the one hand, frees more space in the cache and, on the other hand, it is more expensive to be reloaded. Another aspect potentially yielding new challenges when employing NDS is the graph data structure at hand: in NDS maps, graphs are considered undirected, and potential restrictions on the routing direction are stored as additional information for each edge. Furthermore, this information as well as the length of the edges is stored only in the tile containing one of the end vertices (independent of the possible routing directions). Hence, for edges crossing the boundary of a tile, one might have to reload a tile only to access this information.

REFERENCES

[1] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. In *Proc. of 9th ALENEX*, pages 46–59, 2007.

[2] G. Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *Journal of Experimental Algorithmics*, 18:1.1–1.43, 2013.

[3] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. In *Proc. of 25th IPDPS*, pages 921–931, 2011.

[4] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51:1–26, 2015.

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

[6] Stefan Edelkamp. Improving the cache-efficiency of shortest path search. In *Proc. of 40th KI*, pages 99–113, 2017.

[7] Stefan Edelkamp and Stefan Schrödl. Localizing a*. In *Proc. of. 17th AAAI*, pages 885–890, 2000.

[8] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and simpler hierarchical routing in road networks. In *Proc. of 7th WEA*, pages 319–333, 2008.

[9] Robert Geisberger and Christian Vetter. Efficient routing in road networks with turn costs. In *Proc. of 10th SEA*, pages 100–111, 2011.

[10] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proc. of 16th SODA*, pages 156–165, 2005.

[11] Andrew V. Goldberg and Renato Fonseca F. Werneck. Computing point-to-point shortest paths from external memory. In *Proc. of 7th ALENEX*, pages 26–40, 2005. URL: http://www.siam.org/meetings/alenex05/papers/03agoldberg.pdf.

[12] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proc. of 6th ALENEX*, pages 100–111, 2004.

[13] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[14] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of shortest path and constrained shortest path computation. In *Proc. of 4th WEA*, pages 126–138, 2005.

[15] Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*, pages 219–230, 2004.

[16] Ira S. Pohl. *Bi-directional and Heuristic Search in Path Problems*. PhD thesis, Stanford University, USA, 1969.

[17] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile route planning. In *Proc. of 16th ESA*, pages 732–743, 2008. doi:10.1007/978-3-540-87744-8_61.

[18] Jeffrey S. Vitter and Elizabeth A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12:110–147, 1994.