Data Structures for Locally Distributed Routing

Nicolai Waniek*, *Student Member, IEEE*, Edvarts Berzs, and Jörg Conradt, *Senior Member, IEEE*

Abstract—Routing and path finding are fundamental for databases as well as information networks. The classical algorithm to compute shortest paths is Dijkstra's algorithm, which is highly suited when the computation is performed sequentially and access to global knowledge is granted. However, without significant workarounds the computational overhead is infeasible if large quantities of data have to be distributed across several hosts. When the data is represented in the form of a graph, many researchers suggest to distribute data according to a clustering that is based on global edge information.
In this paper we propose to take inspiration from neurobiology. Namely we are motivated by spatial navigation in the rodent hippocampus, and cluster nodes by assigning each node a spatial coordinate which is independent of edge weights. Furthermore, similar to other work, we suggest to apply hierarchical contraction on this data with location-based clustering.
The focus of the presented work is not to develop methods that run faster than state-of-the-art techniques, but analyze and characterize algorithms and data structures that we think are simpler to distribute. Results demonstrate that our approach is feasible for many practical data sets and it admits a simple approach to parallelize path and distance queries.

1 Introduction

Shortest path computation and routing are essential parts in many advanced technological systems. Unconsciously we use these kinds of algorithms on a daily basis, for instance when searching for the shortest path on a planned road trip. Hence it does not come as a surprise that, over the last few decades, many scientists have contributed to the optimization of the runtime and memory complexity of such algorithms, thereby significantly reducing retrieval times for shortest path queries in large knowledge and information systems. Databases, navigation systems, intelligent transportation systems, or computer networks, to name just a few areas, benefit from their findings. Furthermore, recent developments have provided significant speed ups of relevant algorithms for global road-networks [1], [2], [3], [4], [5], [6].

Another area that benefits from improved routing algorithms is robotics: autonomous agents that operate in an unmapped environment have to solve the problem of simultaneous localization and mapping (SLAM, e.g. [7], [8], [9], [10], [11]), which accumulates tremendous amounts of spatial knowledge over time. However, they should not only map an environment and compute their own location, but eventually be able to find a shortest path to a location of interest. This scenario can easily be extended to multiple robots in several ways. For instance, robots can cooperatively map an environment and build a common database of spatial knowledge [12], [13], [14]. Here we present work towards another idea: distributing the shortest path computation onto multiple participating agents.

1.1 Related Work

The most fundamental algorithm for finding arbitrary shortest paths in graphs with edges E, vertices V, and non-negative weights is Dijkstra's algorithm. Currently, the best runtime complexity of $\mathcal{O}((|E| + |V| \log |V|)$ can be achieved by an implementation with Fibonacci heaps. In practice, however, such a complexity can pose issues for shortest path queries on huge graphs with millions of nodes and edges. Additionally and, for our work more importantly, Dijkstra's algorithm does not natively support parallel execution except for it's bi-directional version in which computations start at source and target node simultaneously.

Many heuristic graph preprocessing techniques are known that aim at speeding up the shortest path queries. These can be classified in goal-directed and hierarchical methods [5], [15], [16], [17], [18]. The goal-directed methods mostly rely on the input graph to represent Euclidean distances between the nodes, more specifically, most goal-directed methods rely on some geometric characteristics of the graph, like triangle inequality or spatial locality. Hierarchical methods, however, do not in general require such limitations on the input graph and were thus inspiration for our work.

Sophisticated methods for hierarchical clustering of data to speed up shortest path computations include Hierarchical Encoded Path Views (HEPV, [19], [20]), High Performance Multi-Level Routing, or HiTi [21], to name just a few. Yet another approach to optimize computational times is shown in [22], which extends multi-level graphs to incorporate knowledge about transit times in road networks. Similar to the approach in [18], it benefits mainly from precomputing the all-pair shortest distances of some vertices that have been identified to be important. The consequence is a speed-up of the retrieval time at the cost of storing the additional information of the all-pair shortest path information. A more recent development is presented in [23], which suggests to recursively build a tree by partitioning the input graph into sub-graphs. Transition information from one sub-graph to the next is subsequently stored alongside the tree in form of transition matrices. The resulting data structure, dubbed G-Tree, supports efficient shortest path and k-nearest neighbor (kNN) queries.

Manuscript created November 18, 2016. This work was supported by EU FET project GRIDMAP 600725. The first two authors contributed equally. Asterisk indicates corresponding author.
N. Waniek, E. Berzs, and J. Conradt are with the Neuroscientific System Theory Group (NST), Technical University Munich, Arcisstraße 21, 80333 Munich, Germany. (e-mail: nicolai.waniek@tum.de).

Most if not all of the hierarchical methods mentioned so far benefit drastically from specialized ways in which the input graph is partitioned. For a general overview of hierarchical methods, we refer to [24], [25]. For recent advances in graph partitioning we refer to the excellent survey presented in [26].

1.2 Motivation and Biological Inspiration

The methods presented above often require global knowledge during the preprocessing steps, for instance to achieve an optimal or at least near-optimal graph partition, or detailed knowledge about individual vertices in the graph. Although they achieve remarkable speed-ups in computation time, they don't lend themselves easily to distribution across a huge number of hosts that can participate in the computation. As hardware and network infrastructure continues to become smaller and cheaper we see this as a major drawback. As a consequence, we looked at natural systems that process spatial information in parallel and distributed as a source of inspiration.

The discoveries of place and grid cells in the rodent Hippocampal Formation and Entorhinal Cortex were a significant step towards understanding the principles behind spatial navigation ind animals. In short, grid cells, a specific type of neuron [27], are assumed to present a metric coordinate space computed from ego-motion information and geometrical cues of the environment [28], [29], [30]. On the other hand, place cells represent individual locations, may store additional trajectory information, and presumably link each place to its grid cell code. The assembly of all place cells can thus be thought to represent a topological graph of places and trajectories with assigned edge weights in a possibly non-metric space, whereas the grid cell assembly uniquely identifies single locations in a metric space with high accuracy [31], [32]. Following this, the brain conveys the impression that metric information of individual locations (vertex position) is separated from transition information (edge weights). Another remarkable curiosity is that grid cell responses appear in several discrete scales [33], which provably leads to optimal coding and representation of the input space [31], [32], [34]. The basic principles of grid cells were successfully tested in or used for robotic scenarios involving navigation and mapping [11], [35], [36], [37].

In search of a simple data distribution scheme we asked if it is feasible to use a hierarchical structure which lends its characteristics from grid cells. This means that the cell size ratio between two neighboring hierarchy levels is constant over the whole hierarchy. Furthermore, each hierarchy level should be split into several clusters or cells which cover possibly overlapping areas of input, and we seek to increase sparseness from lower to higher levels. The approach should ignore peculiarities about specific vertices and treat all input equally. Here we present our results towards a data structure that we see suitable for this task and which will allow us to easily distribute the data onto several hosts, while retaining high efficiency for computing shortest paths.

Our work is significantly different from existing approaches, because our technique to distribute data is independent of edge weights between nodes and treats all nodes alike. In the work presented here we augment nodes with spatial coordinates. We are aware that data association and coordinate generation are a major issue in many applications, especially robotics. Nevertheless we introduce coordinates to examine our proof of principle. Overall, our approach can be seen as a mixture of goal-directed as well as hierarchical methods. Additionally, unlike many other approaches that admit near-optimal solutions, we are only interested in *optimal* shortest paths, that is, we search only such paths between source and target nodes for which no shorter path exists. Because of the spatial coordinates, we can efficiently assign each node to a cluster, regardless of the shape of the graph of which the node is a part. This provides a way to easily distribute the nodes and compute shortest paths *in parallel*, which will be important for future work: we conjecture that our method will allow for efficient computation in a locally distributed setting, where each cluster can be assigned to a different host.

2 Preliminary Definitions and Requirements

In this section we formally define a Sparse Layered Graph (SLG) structure according to our method and introduce further definitions, requirements, and possible constraints. We will use the same symbols as in the definitions throughout the algorithm specifications. We build upon the work of and try to stay as close as possible to the definitions given in [5], [16], [23], [25], nevertheless we include all required definitions for the sake of completeness.

If not specified otherwise, $|\cdot|$ will denote the cardinality of a set. To distinguish between elements of a single layer and elements between layers, we will keep layer indices in subscripts and all others in superscripts. For instance, $v_i^j \in V$ is the j-th element of V at the i-th layer.

2.1 Single layer definitions

Definition 1. *Let $G = (V, E)$ be an undirected planar graph, where V is the set of vertices and E the set of edges $E = \{e(u, v) :$ there exists an edge between $u, v \in V\}$. Each edge $e(u, v) \in E$ is associated with a weight denoted as $w(u, v) \geq 0$. Furthermore each node $v \in V$ is associated with a coordinate $x(v) \in \mathbb{R}^2$, where all coordinates $x(v)$ are assumed to be uniformly distributed in the coordinate space $\mathbb{R}^2$.*

A typical real-world occurrence of such a setup is road maps, where each place can be identified by a unique world coordinate and edge weights $w(u, v)$ represent the time to travel from u to v [16]. For the material presented here we assume that we have a method available to uniquely assign a coordinate $x(v)$ to any vertex v. We will use the terms *vertex* and *node* interchangeably.

Definition 2. *Let $G = (V, E)$ be a graph as defined above. We define a covering C of G as a collection of subsets*

$$C = \{c^j \subseteq V : j \in J\},$$

such that

$$V = \bigcup_{j \in J} c^j ,$$

where $J = \{1, 2, \ldots, N\}$ is an index set and $N = |C|$. In addition, each subset c^j is accompanied by a coordinate $\zeta^j =$

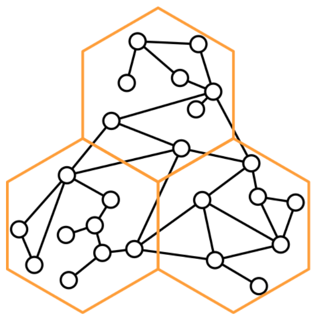

Figure 1. Graph, Cells, and Border Nodes. An input graph (left hand side, small circles and connections) is covered by a set of cells (orange hexagons); we extract all border nodes from these cells for further use in the Transition Graph (right hand side, filled small circles).

$\zeta(c^j) \in R^2$, called it's cell center, in concordance to the coordinates $x(v)$ defined above.

In other words, each vertex $v \in V$ is contained in at least one subset c^j, as depicted in Figure 1. In contrast to graph partitions that are frequently used in other work, it is not guaranteed that $c^j \cap c^k = \emptyset$, $\forall c^j, c^k \in C$. We use the terms cell and cluster for c^j interchangeably, and usually omit the super-script j if readability can be increased and j can be inferred from context.

Note that the cell center ζ^j not necessarily is the center of mass of the contained vertices $v \in c^j$. In fact, and without loss of generality, we use centers ζ^j such that all cell centers form a hexagonal tesselation of V as shown on the left hand side of Figure 1. However, other distributions of cell centers are possible, e.g. a non-uniform Voronoi Tesselation, but these are omitted as they do not yield significant new insights. In the majority of cases, the problem of finding the optimal graph partitioning is at least NP-Hard [38] and thus for large data sets it is feasible to search for suboptimal graph partitions, especially since in most cases slight deviations from the optimum do not severely impair the performance of any algorithms that are applied to this partition.

To accelerate shortest path queries, we introduce an additional data structure called Transition Graph that only considers nodes $v \in B$, where B is a border node set.

Definition 3. *A border node set $B = \{v^b : v^b \in V\} \subseteq V$ is the set of nodes u^b, v^b for which the following holds:*

$$\exists j,k \in J, j \neq k, \exists e(u^b, v^b) \in E : (u^b \in c^j) \wedge (v^b \in c^k)$$

$B^c = \{v^b : v^b \in c\}$ is the border node set of cell c. For simplicity we identify $B^j := B^{c^j}$ for the border node set of cell c^j. Furthermore we denote with

$$F = \{e(u,v) : u,v \in B \wedge e(u,v) \in E\}$$

the set of all edges between pairs of border nodes.

Hence the set B captures all nodes that indicate transitions from one cell to another. Or in other words, each node that has an edge leaving its cell will be considered a border node. The right hand side of Figure 1 shows a visualization of the border nodes, where each node that belongs to B^c is shaded in blue.

2.2 Multi-layer graph

We can now easily extend the fundamental definitions above to expand to multiple layers by adding an index i which denotes the i-th layer. The indices form a strict total order $I = (0, \ldots, i, \ldots, L-1)$, where L is the number of layers. For instance, $G_i = (V_i, E_i)$ describes the graph G_i of the i-th layer, which has vertex set V_i and edge set E_i. On the other hand, c_i^j is the j-th cluster in layer i. Finally, we need to distinguish between two different graphs on layer i: the graphs induced by the Sparse Layer Graph and Transition Graph construction described below are denoted by $\mathbf{S}_i$ and $\mathbf{T}_i$, respectively. Their node and edge sets are $(V_i^{\mathbf{S}}, E_i^{\mathbf{S}})$ and $(V_i^{\mathbf{T}}, E_i^{\mathbf{T}})$.

We note the following properties and observations that we will use throughout the algorithm descriptions and complexity analysis below:

- The maximum ratio of nodes that are contracted to total number of nodes in previous layer
$$\alpha_i = \frac{|V_i^{contracted}|}{|V_{i-1}|} \; \forall i \in I \setminus 0$$
- Similarly, the ratio of cells between two consecutive layers is denoted as
$$\gamma_i = \frac{N_i}{N_{i-1}} \; \forall i \in I \setminus 0$$
- If $i < j$, then $|V_i| \leq |V_j|$. However, there is no such relation between $|E_i|$ and $|E_j|$
- The number of cells in the lowest layer N_0^C is a parameter that can be chosen freely. Based on our experiments, a reasonable choice is $N_0^C = \sqrt{|V|}$.
- The ratio of border nodes to all nodes in the layer i is
$$\beta_i = \frac{|B_i|}{|V_i|} \; \forall i \in I$$
- The ratio of number of nodes in $\mathbf{T}_i$ to number of nodes in $\mathbf{S}_i$:
$$\lambda_i = \frac{|V_i^{\mathbf{T}}|}{|V_i^{\mathbf{S}}|}$$

3 Construction Algorithms

Using the definitions and symbols from above, we can now proceed to the description of the iterative method of building the Sparse Layered Graph $\mathbf{S}$ from an input graph $\mathcal{G}$ as defined in Definition1, presented in Algorithm 1.

3.1 Sparse Layered Graph Construction

The Sparse Layered Graph (SLG) contains a total of L layers, which are numbered consecutively as $(0, \ldots, L-1)$, and each layer $\mathbf{S}_i$ consists of a covering C_i.

Definition 4. *We define a Sparse Layered Graph* $\mathbf{S}$ *as*

$$\mathbf{S} = \bigcup_{i \in I} \mathbf{S}_i\,, \qquad \mathbf{S}_i = \bigcup_{j \in J} \mathbf{S}_i^j\,,$$

where $\mathbf{S}_i$ represents the sparse layered graph structure in layer i, which itself is a cover of all subgraphs induced by the covering C_i. Consequently, $\mathbf{S}_i^j := \mathbf{S}_i^{c^j}$ is the subgraph induced by cell c_i^j.

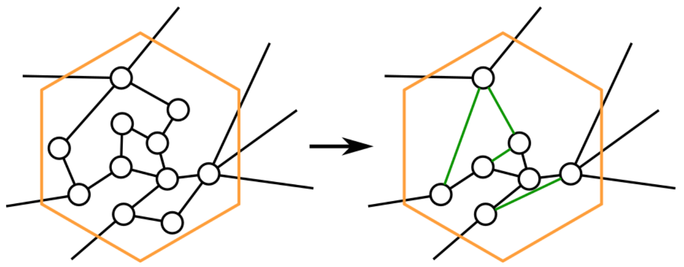

Figure 2. Edge Contraction. The contraction process removes a certain number of nodes with minimum degree within each cell and inserts new edges between the remaining nodes (indicated in green) where necessary. As a consequence the number of nodes in the layer is reduced, whereas the number of edges may increase depending on the structure of the input graph $\mathcal{G}$.

$\mathbf{S}_0$ consists of a covering of the original graph $\mathcal{G}$, according to Definition 2. Each of the following $j = 1, \ldots, L-1$ layers is constructed by taking the set of remaining nodes and edges from the previous layer $j-1$, assigning these to the cells c_i^j with centers ζ_i^j of the new layer j, and finally performing a *node contraction* step to prune the graph. A visualization of the contraction process is given in Figure 2, and of the SLG and its construction in Figure 3.

Definition 5. *A node v is said to be* contracted *in the current layer, if the following conditions hold*

1) *v is removed from the current layer*
2) *all edges with an end-point in v are removed*
3) *all neighbors u, w of v are directly connected with each other by novel edges. The weights of these new edges are calculated according to $d(u,w)^{\text{contracted}} = d(u,v) + d(v,w)$. If the edge $(u,w) \in E_i$ and if $d(u,w)^{\text{contracted}} < d(u,w)$, then we will flag this edge as contracted and store only $d(u,w)^{\text{contracted}}$. During path querying, we will have to expand contracted edges.*

We note that the contraction of nodes within a cell can be performed independently of and thus in parallel to other cells on the current layer. The effect of the node contraction for one cell is shown in Figure 2. Thus, this step of the algorithm can be easily parallelized. However, since each layer depends on the result from previous layers (see Figure 3), the layers cannot be processed in parallel and have to be handled in a serial manner. Note, that, by construction, each consecutive layer will have *at most* as many nodes as the lower layers, and in most cases strictly less nodes than previous layers.

3.2 Transition Graph Construction

Once a layer $\mathbf{S}_i$ is generated, its Transition Graph (TG) $\mathbf{T}_i$ can be derived from it. Formally, $\mathbf{T} = \bigcup_{i \in I} \mathbf{T}_i$, where each $\mathbf{T}_i$ consists of the TGs of all cells c_i^j, namely $\mathbf{T}_i = \bigcup_{j \in J} \mathbf{T}_i^j$. The TG construction algorithm 2 builds an additional sparse graph structure for each layer $\mathbf{S}_i$. Unlike the SLG construction algorithm, each layer can be handled simultaneously, thus admitting parallel execution.

Initially, all edges that have their head and tail nodes in different cells are extracted (see Definition 3 and Figure 1). Then, each cell is processed separately by considering only the nodes in the respective cell. For all such nodes belonging to a certain cell, the All-Pairs Shortest Path (APSP) distances are calculated and edges with weights corresponding to the APSP results are added to the Transition Graph. With this method we acquire a graph with considerably reduced number of nodes compared to Sparse Layered Graph. The algorithm is presented in Algorithm 2 and depicted in Figure 4.

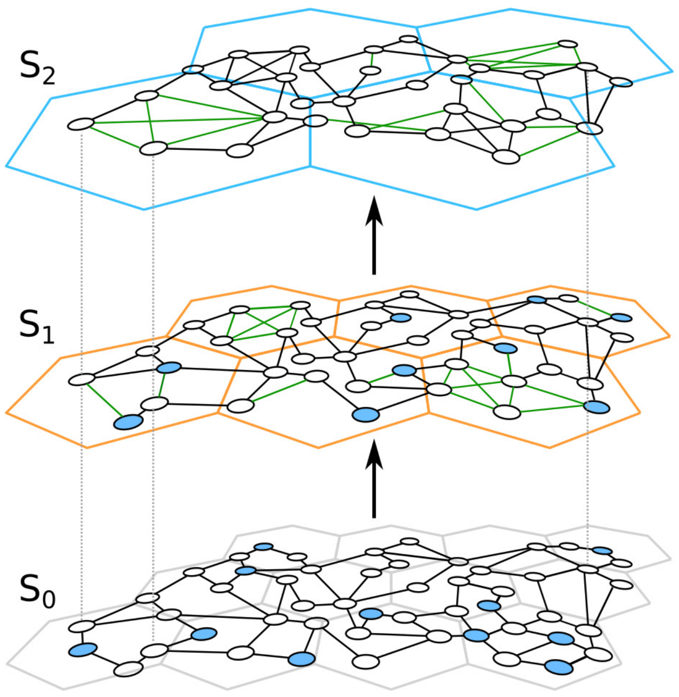

Figure 3. Sparse Layered Graph and Layer Construction. The SLG $\mathbf{S}$ consists of several layers $\mathbf{S}_i$ (three layers shown here), each layer inducing its own subgraph. Additionally, each layer $\mathbf{S}_i$ consists of many cells c_i^j with their own subgraph $\mathbf{S}_i^j$. The first (bottom) layer is formed by covering the input Graph $\mathcal{G}$ with a certain number of cells (gray hexagons, bottom row). Every following layer is constructed by taking all nodes and edges from the previous layer, assigning the nodes to their corresponding cells (orange hexagons, middle row) and performing edge contraction in each cell (green edges, compare to the input layer). The next layer (top row) receives the remaining nodes and edges as input and operates on cells with a fixed size increment (blue hexagons).

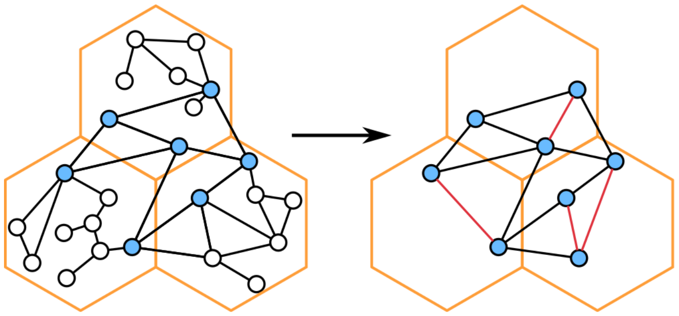

Figure 4. Transition Graph Construction. The Transition Graph for the graph presented in Figure 1 (left hand side) is constructed by selecting only the border nodes of each cell, and adding the all-pair shortest path information within each cell (red lines on right hand side) if necessary.

4 Queries

At the moment, our data structures support two kinds of queries: *getShortestDistance(s,t)* and *getShortestPath(s,t)*, which return the shortest distance and shortest path between source

Algorithm 1 Generation of the Sparse Layered Graph

```
Initialize: $i = 1, \mathbf{S}_0 = \mathcal{G}$
for all $v \in V_0^{\mathbf{S}}$ do                        ▷ clustering on layer 0
    Assign each $v$ to closest cell $c^j$
end for
while $i \le L - 1$ do                    ▷ construction of layers $i > 0$
    $\mathbf{S}_i = \mathbf{S}_{i-1}$
    for all $v \in V_i^{\mathbf{S}}$ do                    ▷ clustering on layer $i$
        Assign each $v$ to closest cell $c^j$
    end for
    for all $c^j$, $j \in J$ do          ▷ node contraction in each cell
        for all contractable nodes $v$ with $deg(v) \ge 2$ do
            for all neighbor pairs $(u, w)$ of $v$ do
                $d^{contracted} \leftarrow d(u, v) + d(v, w)$
                if $(u, w) \in E_i^{\mathbf{S}} \wedge d^{contracted} < d(u, w)$ then
                    $d(u, w) \leftarrow d^{contracted}$
                    flag edge $(u, w)$ as contracted
                else if $(u, w) \notin E_i^{\mathbf{S}}$ then
                    add new edge $(u, w)$ to $\mathbf{S}_i$
                    $d(u, w) \leftarrow d^{contracted}$
                    flag edge $(u, w)$ as contracted
                end if
                remove $v$ from $\mathbf{S}_i$
                mark all neighbors of $v$ as non-contractable
            end for
        end for
    end for
    $i \leftarrow i + 1$
end while
```

Algorithm 2 Generation of the Transition Graph

```
for all $i \in I$ do
    Initialize: $\mathbf{T}_i \leftarrow (B_i, F_i)$                ▷ see Definition 3
    for all cells $c^j$, $j \in J$ do             ▷ all-pair shortest-path
        Initialize: $\mathbf{T}_i^j \leftarrow (\emptyset, \emptyset)$
        for all $u, v \in B_i^j$ do
            $w(u, v) \leftarrow \min d(u, w)$ in $\mathbf{S}_i^j$
            $\mathbf{T}_i^j = \mathbf{T}_i^j \cup e(u, v)$                ▷ insert (new) edge
            flag edge $e(u, v)$ as an APSP-edge
        end for
        $\mathbf{T}_i = \mathbf{T}_i \cup \mathbf{T}_i^j$
    end for
end for
```

node s and target node t, respectively. Their shorthand notation is `getDist` and `getPath`. In practice with serial execution, the shortest distance can be found approximately twice as fast as the shortest path. This fact has some practical advantages, for example, in path planning: often we are interested in the length of a path between a source node and some points of interest, whereas the exact path is of secondary or later interest.

4.1 Shortest Distance Query

The `getDist` algorithm presented in Algorithm 3 first finds the lowest level $\mathbf{S}_k$ in $\mathbf{S}$, where the source and target nodes are both present, i.e. where they are not contracted. $\mathbf{T}_k$ is used as a subgraph and the graphs from source and target cells in $\mathbf{S}_k$ are appended to it. Then, the shortest path from s to t in this temporary graph is calculated. Due to the construction of $\mathbf{S}$ and $\mathbf{T}$ it is guaranteed that this path has the same distance as the full path from s to t in the input graph $\mathcal{G}$. Note that this calculation is done on a single machine at the moment and is thus performed sequentially. For obvious reasons, operating in layer k is unlikely to be optimal in all cases. For instance, if both nodes are contracted on the lowest layer, all operations will take place on this layer. The suggested algorithm is only to demonstrate proof-of-principle as it allows performance improvements at several spots.

Algorithm 3 Shortest distance query: `getDist`

```
Input source and target nodes $s, t \in V$
Output $dist$ - length of shortest path between $s$ and $t$
$k \leftarrow$ lowest common level of $s$ and $t$
$[c_s, c_t] \leftarrow$ cells on level $k$ which include $s$ and $t$
$\mathbf{Q} = \mathbf{T}_k \cup \mathbf{S}_k{}^{c_s} \cup \mathbf{S}_k{}^{c_t}$                 ▷ merge sub-graphs
$P \leftarrow$ shortest path in $\mathbf{Q}$          ▷ for instance with Dijkstra
return $|P|$
```

4.2 Shortest Path Query

Path retrieval with `getPath` is very similar to `getDist`, in fact first step is identical: we find the shortest path from s to t in the temporary Graph $\mathbf{Q}$ constructed during distance computation. However due to the construction of $\mathbf{T}$, the path that we acquire may not yet include all vertices of the shortest path in $\mathcal{G}$. Resolving the partial path to get the full shortest path in $\mathcal{G}$ is presented in Algorithm 4 and an example is shown in Figure 5.

We note that the shortest path from s to t in $\mathbf{S}_k$ will *necessarily* include all vertices on the corresponding path in $\mathbf{Q}$, and in fact in the same order. More precisely, an ordered set of vertices from the shortest path from s to t in $\mathbf{S}_k$ is a superset of the ordered set of vertices on the shortest path in $\mathbf{Q}$. Furthermore, the set of vertices on a shortest path in a layer $j < k$ is a superset of the vertices on layer k due to the sparsification during the node contraction.

We can now follow the shortest path in $\mathbf{Q}$ and resolve it to the final path in $\mathcal{G}$. This process can be seen to consist of two phases depends on the type of edge which was determined during construction of SLG and TG. An edge on the shortest path in $\mathbf{Q}$ for a specific layer k can be one of three types: a regular edge, a contracted edge, or an APSP edge. Considering APSP edges, we know that they were added during the TG construction, hence we can simply query the SLG cells that maintain the head and tail of such edges (in parallel) to retrieve the shortest edge-expansion. Applied to all APSP edges, the process yields a shortest path P^k on layer k with only regular and contracted edges. This concludes the first phase.

During the second phase, we will expand any other edge and finally retrieve the full path P^0 in $\mathcal{G}$. As regular edges are immediate leftovers from the original graph $\mathcal{G}$, they cannot be resolved any further. However, all contracted edges need to be expanded to their non-contracted counterparts. Given a contracted edge $e(u, v)$ between two nodes u, v on layer k, we drop to layer $k - 1$ to expand it. Due to the construction of the SLG it is guaranteed that there is either a direct edge

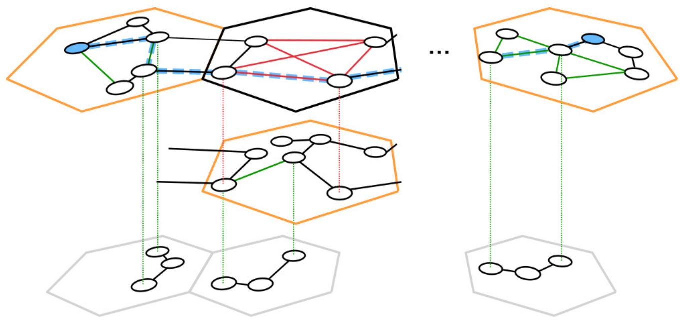

Figure 5. Shortest Path Query example. The goal is to find the shortest path, indicated by the dashed blue line between the two blue nodes in the topmost row. First, the highest level on which both nodes are not contracted is identified and their respective cells (top row, orange) of the SLG are combined with the TG to find the shortest path using only intermediate TG cells (in black). Then, all-pair shortest path edges (red edge in top row) of the shortest path in the TG are resolved to their respective edges in SLG cells (middle row). Finally, each contracted edge (green edges on shortest path) are resolved using lower level representations (bottom row). This process is invoked recursively until all remaining contracted edges are resolved.

on layer $k-1$ connecting the two nodes, or a neighbor z that resides in between. If both u, v belong to the same cell c on layer $k-1$, the path expansion can thus be handled by c itself. In some cases, u and v belong to two separate cells. However, the discrete layer structure with a suitable scaling of cluster sizes and the regularity of the cluster centers guarantees that only direct neighbors need to be accessed to expand the path. Recursively applying edge expansion with ultimately lead to the shortest path P^0 in $\mathcal{G}$.

Although this approach may not yield the best performance with respect to execution time, it shows the capability for parallelization or distribution of the processes in future work. Each of the cells can potentially be moved to a separate distinct host, for instance using peer to peer network technology. In this case, the shortest path may be distinct from the shortest transmission routing and only necessary SLG cells have to be queried for detailed path information. The distribution of the TG is not as straightforward, though, and will require more work in the future.

Algorithm 4 Shortest path query: `getPath`

```
  Input source and target nodes $s, t \in V$
  Output $P^0$ - shortest path between $s$ and $t$
  $k \leftarrow$ lowest common level of $s$ and $t$
  $[c_s, c_t] \leftarrow$ cells on level $k$ which include $s$ and $t$
  $\mathbf{Q} = \mathbf{T}_k \cup \mathbf{S}_k{}^{c_s} \cup \mathbf{S}_k{}^{c_t}$          ▷ merge sub-graphs
  $P_Q \leftarrow$ shortest path in $\mathbf{Q}$        ▷ for instance with Dijkstra
  $P^k \leftarrow P_Q$ with expanded APSP edges              ▷ 1. phase
  $P^0 \leftarrow P^k$ with recursively expanded edges       ▷ 2. phase
  return $P^0$
```

5 Complexity Analysis

In the following section we will analyze the run-time complexity of our algorithms. We are interested if the algorithms presented in Sections 3 and 4 introduce any severe theoretical issues that have to be addressed. As our focus is not an improvement with respect to execution times but a simple distribution scheme we wish to stay within Dijkstra execution bounds for any operation. Further, we want to understand which impact edge and node count have during construction of the data structures.

The construction consists of building two data structures, namely a Sparse Layered Graph $\mathbf{S}$ and a Transition Graph $\mathbf{T}$, which we will analyze separately. We will give lower and upper bound estimates of the worst case run-time for the construction on a single computer in a sequential manner, and show why this analysis strongly depends on the graph $\mathcal{G}$. Afterwards we will quickly discuss the average run-time complexity of the construction when using suitable data structures to hold the edge and node information. Then we will address the average case query times and, finally, give an outlook to the complexity in a distributed setting. We wish to remind the reader that update operations on a regular array require, in theory, $\mathcal{O}(N)$ operations both in the average and the worst case. In contrast, the average update operations for a hash table are close to $\mathcal{O}(1)$, while the worst-case complexity still resides in $\mathcal{O}(N)$. During the analysis we will use the set operators to identify relationships between complexity classes. Furthermore, we will usually omit the cardinality operator $|\cdot|$ when it is clear from context to improve legibility. For example, $\mathcal{O}(c \cdot V)$ denotes the set of functions that run in linear time with constant c on the number of nodes in V, hence reside in $\mathcal{O}(V)$ which may be expressed as $\mathcal{O}(c \cdot V) \subseteq \mathcal{O}(V)$.

5.1 Construction of the Sparse Layered Graph

As all necessary operations to construct $\mathbf{S}$ are the same in each cell and layer, we will reduce the nomenclature of all required variables to make the analysis more accessible. Hence, we will start by denoting the set of vertices as V and the set of edges as E without any sub- or superscript indexes. As soon as we extend the formal analysis to multiple cells or layers, we will re-introduce those indexes.

5.1.0.1 Dependency on the topology of $\mathcal{G}$: A naive implementation of Algorithm 1 might use at least two tables. One to store edge information, another to store node information. Each of these tables will be sorted before contracting nodes, allowing binary search to locate specific elements in logarithmic time. For instance, a table containing the edge information $e = (u, w) \in E$ will be sorted on both node indexes, such that a specific edge can be found in $\mathcal{O}(\log E)$ time. Likewise, the table of nodes will be sorted. Conclusively, the run time complexity to pre-sort the data within a cell will take

$$\mathcal{O}(E \log E + V \log V) \tag{1}$$

time. However, for any non-degenerate case of $\mathcal{G}$, we can assume that $V \leq E$, for which Equation 1 simplifies to

$$\begin{aligned}\mathcal{O}(E \log E + V \log V) &\subseteq \mathcal{O}(2 \cdot (E \log E)) \\ &\subseteq \mathcal{O}(E \log E)\,.\end{aligned} \tag{2}$$

Real-world graphs or networks, however, often show the tendency to be scale-free. This means that most nodes in $\mathcal{G}$ will be connected to only few other nodes, and only rarely a node will be connected to many other cells. Essentially, the distribution of node-degrees $\deg(v), \forall\, v \in V$ follows a power-law in realistic scenarios. Accordingly, we suggest to store information differently as soon as a scale-free network can be expected.

For scale-free networks, Algorithm 1 uses an array to store node information and an adjacency list that represents the edge information for each node. The array of nodes can be sorted in $\mathcal{O}(V \log V)$ time, and each of the adjacency lists will require at most $\mathcal{O}(\deg_m(V) \log(\deg_m(V))$ time, where $\deg_m(V) := \max\{\deg(V\}$. Due to the assumption that $1 \leq \deg(v) \ll V$ holds in most real-world graphs, we can give the amortized run-time complexity of

$$\begin{aligned} &\mathcal{O}(V \log V + V \cdot \deg_m(V) \cdot \log(\deg_m(V))) \\ \overset{amortized}{\subseteq} \; &\mathcal{O}(V \log V + V \log V) \subseteq \mathcal{O}(2 \cdot V \log V) \end{aligned} \tag{3}$$

to sort the data. Furthermore, the assumption allows us to expect an amortized retrieval time of

$$\mathcal{O}(1 + \log V) \subseteq \mathcal{O}(\log V) \tag{4}$$

for adjacency information for any node.

Consequently, it is important to consider which type of data will be stored in $\mathbf{S}$. Although the following analysis is mostly based on the assumption to store scale-free networks, this will not have a severe impact on the run-time complexity of the node contraction.

5.1.0.2 Node Contraction: We first consider the contraction of a single node $v \in V$, given any neighbors $u, w \in V$. For any such pair, we will remove the edges from each neighbor to v and either update an existing or introduce a novel edge. This requires to perform at most $\mathcal{O}(\deg(v))$ operations. With Equation 4 we can combine the estimate to yield

$$\mathcal{O}(\log V + \deg(v)) \subseteq \mathcal{O}(\log V + E) \tag{5}$$

worst case time requirement to contract a single node v. However, for scale-free networks $\deg(V)$ will usually be a small number $1 \leq \deg(V) \ll V \leq E$ for which we can expect a run-time behavior which is bounded from the bottom by

$$\mathcal{O}(\log V + \deg(v)) \overset{amortized}{\supseteq} \mathcal{O}(\log V + 1)\,. \tag{6}$$

To understand the worst case bounds of the final algorithm, we will consider the two extremal cases that may occur during contraction of $v \in V$. The first case is when $u, w \in V$ are the only neighbors of v, by which the lower bound time complexity is governed by Equation 5 for any node v. As we flag all neighbors u, w of v as *non-contractable*, we will perform this operation at most $\frac{V}{2}$ many times, which yields the overall lower bound of

$$\mathcal{O}(\frac{V}{2}E) \subseteq \mathcal{O}(V \cdot E) \overset{V \leq E}{\subseteq} \mathcal{O}(E^2) \tag{7}$$

to contract all nodes $v \in V$ within a single cell. However, for the amortized case presented in Equation 6 we may achieve

$$\mathcal{O}(V \cdot (\log V + 1)) \overset{amortized}{\subseteq} \mathcal{O}(V \cdot \log V + V) \tag{8}$$

logarithmic run time complexity to contract all nodes in many cases.

The second extreme case to consider is if v is connected to all other nodes $u, w \in V$, but $\nexists e(u, w) \in E$ for any u, w. In other words, the graph exposes a star topology and $deg(v) = E$. Contracting v will thus require to introduce $\frac{E(E-1)}{2}$ many new edges, while removing E old edges. This leads to an upper bound estimate with a cubic run-time of

$$\mathcal{O}(E + E\frac{E(E-1)}{2}) \subseteq \mathcal{O}(E^3)\,, \tag{9}$$

due to the $\mathcal{O}(E)$ time requirement to update an existing array in a naive way. Obviously, this can be improved to

$$\mathcal{O}(E + \frac{E(E-1)}{2}) \subseteq \mathcal{O}(E^2) \tag{10}$$

run-time requirements with the help of suitable data structure and updating edge information only once. Equation10 represents the worst upper bound for the amortized example as well. In fact, all other nodes $u \in V, u \neq v$ will be marked as non-contractable and won't be considered anymore, Equation 10 therefore is a tight upper bound.

5.1.0.3 Multiple Cells and Layers: We will now proceed to extend the analysis to include multiple cells and layers, and give estimates about the naive worst case run-time complexity Ω and for the amortized worst-case complexity $\Omega_{amortized}$.

To understand the simplification, we require the following observations. Equations 7 and 10 show that the run-time complexity can be approached by $\mathcal{O}(E^2)$ both from bottom and top. Hence we can focus on additional steps that are required to form a single layer. In an sequential implementation, Algorithm 1 sorts all existing cells to improve access times. Obviously, this can be achieved in $\mathcal{O}(N_i \log N_i)$ time for N_i cells on layer i. Furthermore, each node $v \in V_i$ gets assigned to its closest cell. This is achievable in $\mathcal{O}(V_i \log N_i)$ time given a suitable data structure, e.g. a kd-tree. In addition, we assumed L to be a small number, $N_i \ll V_i$ and thus $N_i \ll E_i$, and that $N_i \leq N_{i+1}$ holds. We will use $E := max_i E_i$ and $V := max_i V_i$ in the remainder of this paragraph.

Given the preconditions, we can now finally assess the worst-case run-time complexity, which is given by

$$\begin{aligned} \Omega = \mathcal{O}(&N_0 \log N_0 + V_0 \log N_0 + \\ &\sum_{i=1}^{L-1} \left(N_i \log N_i + V_i \log N_i + N_i \cdot E_i^2\right)) \\ \subseteq\; &\mathcal{O}(L \cdot E^2) \subseteq \mathcal{O}(E^2)\,. \end{aligned} \tag{11}$$

The first two terms describe sorting cells and assigning nodes to cells on the bottom layer, on which no contraction will be performed. Conclusively, the run-time complexity is completely governed by the maximal number of edges in $\mathbf{S}$, because N_i and L quickly become negligible for a large E.

For the amortized analysis, we can argue similarly but, using Equation 6 and 8, achieve a lower and upper bound of

$$\mathcal{O}(V \log V + E) \subseteq \Omega_{amortized} \subseteq \mathcal{O}(E^2)\,. \tag{12}$$

Hence, the lower bound run-time will be mostly governed by the number of nodes. For real-world data we would expect to be close to the lower bound of the amortized run-time complexity in many cases.

5.1.0.4 A short discussion about the average case: Until now we have mostly focused on the worst case time complexity of our algorithm. With an improved data structure such as a hash table, we could reduce the effective and average run-time of the implementation. A hash table allows to insert and search in $\mathcal{O}(1)$ time, essentially reducing Equations 1 – 4 to $\mathcal{O}(1)$ as well. Furthermore the upper bound presented in Equation 9 will directly collapse to Equation 10. The impact is less dramatic in the theoretical assessment of the other parts, though, as they remain as derived above. However, we suggest to use hash tables to store all relevant data if run-time performance is an important criteria.

5.2 Building of the Transition Graph

Following the description of Algorithm 2, the TG construction operates on each cell of $\mathbf{S}$. Although the TG construction can be nested within the SLG construction, we will treat the TG construction separately here. Hence, we will first analyze some cell c on a layer i for which we define the node and edge set as V_i^c and E_i^c, respectively.

With an appropriate data structure, for instance a lookup table, $\mathbf{S}_i^c$ can be extracted in $O(1)$ time. Similarly, each border node can be identified in amortized $O(1)$ time. The computation of the all-pair shortest path is performed by running Dijkstra's algorithm on each border node. Therefore the run-time complexity to generate one cell of the TG is

$$B_i^c(E_i^c + V_i^c \log V_i^c) \tag{13}$$

The assumption of scale-free networks with uniformly distributed nodes leads to the approximations of

$$V_i^c \approx \frac{V_i^{\mathbf{S}}}{C_i}, \qquad E_i^c \approx \frac{E_i^{\mathbf{S}}}{C_i}, \qquad B_i^c \le B_i \tag{14}$$

for all cells. This leads to the overall run-time complexity of

$$\begin{aligned} &\mathcal{O}(\sum_{i=0}^{L-1} C_i B_i \Big(\frac{E_i^{\mathbf{S}}}{C_i} + \frac{V_i^{\mathbf{S}}}{C_i} \log \frac{V_i^{\mathbf{S}}}{C_i} \Big)) \\ &= \mathcal{O}(\sum_{i=0}^{L-1} B_i \Big(E_i^{\mathbf{S}} + V_i^{\mathbf{S}} \log \frac{V_i^{\mathbf{S}}}{C_i} \Big)) \end{aligned} \tag{15}$$

to construct the TG for all cells and layers. Due to $B_i \ll V_i$, and with $E := \max E_i^{\mathbf{S}}$ and $V := \max V_i^{\mathbf{S}}$ this can be amortized to

$$\begin{aligned} &\mathcal{O}(\sum_{i=0}^{L-1} E_i^{\mathbf{S}} + V_i^{\mathbf{S}} \log \frac{V_i^{\mathbf{S}}}{C_i}) \\ \subseteq\ &\mathcal{O}(\sum_{i=0}^{L-1} E_i^{\mathbf{S}} + V_i^{\mathbf{S}} \log V_i^{\mathbf{S}}) \\ \subseteq\ &\mathcal{O}(L \cdot (E + V \log V)) \,. \end{aligned} \tag{16}$$

Clearly, the TG construction is dominated by the maximal number of vertices in a cell and not the overall number of vertices in $\mathbf{S}$.

5.3 About Query Time Complexity

The distance and path queries are certainly sub-optimal in their presented form. For instance, they only locate the highest layer k on which the source and target nodes are still available and will not climb to higher layers for increased speed. Because future work will improve significantly on this, we will only assess and finally discuss the run-time complexity of the queries in their current version as presented in Algorithms 3 and 4. This is necessary to evaluate if the proposed techniques deem feasible for their future purpose in distributed scenarios, or if there are significant disadvantages inherent to the data structures.

Given a source s, a target t, and only a single layer $k = 0$, the worst case for distance estimation appears when all of the cells of this layer have to be traversed. The complexity follows from computing a shortest path $P_{\mathbf{Q}}$ in $\mathbf{Q}$. According to Algorithm 3, $\mathbf{Q}$ is defined as the union of T_k with the subgraphs induced by c_k^s and c_k^t, the two cells of $\mathbf{S}_k$ in which s and t reside. Computing Dijkstra's algorithm form s to t in Q is theoretically at least as fast or faster than computing the shortest path in $\mathbf{S}_k$, because $\mathbf{T}_k$ is, on average, sparser than $\mathbf{S}_k$ due to construction. However the specific way the union to form $\mathbf{Q}$ is implemented in a practical implementation may yield significant penalties. Furthermore if the TG consists of only one or two cells, the distance computation will inflate to path queries within c^s and c^t.

Path queries require additional computations to expand edges. Due to construction of $\mathbf{S}$ and $\mathbf{T}$, an edge will be either regular, contracted, or an APSP edge. First, any APSP edge on the shortest path $P_{\mathbf{Q}}$ will be expanded, generating a shortest path $P_{\mathbf{S}}^k$ in $\mathbf{S}_k$. The edge expansion is performed by shortest-path computation within the corresponding cell in $\mathbf{S}_k$. Given that $P_{\mathbf{Q}}$ spans $\zeta^{\mathbf{Q}}$ many cells in $\mathbf{Q}$, for which $0 \le \zeta^{\mathbf{Q}} \le C_k - 1$ holds, this leads to

$$\mathcal{O}(\zeta^{\mathbf{Q}} \Big(\frac{E_k^{\mathbf{S}}}{C_k} + \frac{V_k^{\mathbf{S}}}{C_k} \log \frac{V_k^{\mathbf{S}}}{C_k})) \,. \tag{17}$$

Empirical data suggests that $\zeta^{\mathbf{Q}} \approx \frac{C_k}{2}$, for which the average run-time complexity of Equation 17 then simplifies to

$$\mathcal{O}(E_k^{\mathbf{S}} + V_k^{\mathbf{S}} \log \frac{V_k^{\mathbf{S}}}{C_k}) \subseteq \mathcal{O}(E_k^{\mathbf{S}} + V_k^{\mathbf{S}} \log V_k^{\mathbf{S}}) \,. \tag{18}$$

Finally, we need to resolve remaining contracted edges in $P_{\mathbf{S}}^k$ to retrieve the expanded shortest path $P_{\mathbf{S}}^0$. Recursively expanding contracted edge will require at most $2^{k-1} \cdot (P_{\mathbf{S}}^k - 1)$ computations in lower-level cells. More specifically, we will expand each contracted edge into two novel contracted edges on every lower layer during the worst case scenario until we reach layer 0.

We note that the overall-worst (or nightmare) case $\zeta^{\mathbf{Q}} = C_k - 1$ exists, though. A properly prepared input graph $\mathcal{G}$ which resembles a space-filling curve could be contracted in such a way that the resulting (sparse) graph again resembles a space-filling curve. Consequently, each cell on each layer needs to be queried. Fortunately, such a path usually won't appear in real-world data.

We conclude that there exist three cases for data retrieval. For the expected average case our technique will expose speed-ups although this was not the original focus of our work. In the worst case, our method will show only marginal theoretical degradations (if at all) despite the increased complexity. During the nightmare case, queries on our data structure will be significantly slower than existing methods due to the curse of recursive invocation of shortest-path computation. Conclusively our methods are suitable for future purposes as they do not introduce computational disadvantages in the average case.

5.4 Short Note on Complexity in a Distributed Setting

To extend the complexity analysis to investigate the theoretical speed-up or slow-down of our technique in a distributed setting, it would be necessary to consider employing a certain number, ideally close to the number of cells, of independent processing units and addressing the number of messages that have to be sent into account. Furthermore, the construction of the SLG itself is iterative as one layer needs to be formed after another which imposes a certain delay of the construction. However, we expect that updates to a single cell in a specific layer after an initial SLG was already formed will require to update only few cells that propagate through the hierarchy. As the real number of messages that have to be passed depends on the employed distribution scheme, we will postpone the complexity analysis of a distributed implementation to future work.

6 Experimental Characterization

To experimentally characterize our method we developed prototypes in MATLAB and C++. We used the implementations for evaluating the accuracy and correctness of the algorithms as well as to test various parameter settings, for instance the initial cell size or relative cell size ratio between consecutive layers. We generated artificial input data in form of planar graphs with 200000 vertices that were uniformly distributed. We examined graphs with different levels of connectivity.

All queries returned the correct result in the sense that all queried distances or paths from a source to a target node corresponded to results obtained by Dijkstra's algorithm executed on the original graph $\mathcal{G}$.

We then analyzed the impact of several cell size ratios. Remember that the ratio γ_j is defined as $\gamma_j = \frac{|C_j|}{|C_{j-1}|} \leq 1$. We expected γ_j not to have any significant impact on the average number of nodes or edges in the SLG. This somewhat counter-intuitive statement can be explained by the node contraction step, which will only stop at border or non-contractable nodes. Border nodes themselves will only be a small fraction of the overall number of nodes in most graphs and especially in the artificially generated data. That our assumption was indeed correct is shown in Figures 6 and 8. The increase in the number of edges for higher layers in the SLG can be explained by the contraction process as well. During the contraction, potentially non-connected vertices will receive novel edges. After several iterations, the graph on a resulting layer will consist of many strongly connected nodes.

The assumption that the cell size ratio has no impact is not true for the TG, though. Larger cells in the SLG mean that the ratio between border nodes and regular nodes is smaller. This leads to smaller numbers of nodes in the TG for smaller γ_j. This behavior is depicted in Figures 7 and 9. The explanation for increased numbers of edges for some cell ratios is the same as for the SLG.

A very similar behavior could be observed for the impact of the number of initial cells and, therefore, initial cell size on layer 0 (no data shown). Given uniformly distributed input, the number of nodes in the SLG will drop quickly during the contraction in the first layer as most contractable nodes will be removed. However, subsequent steps will only be able to contract roughly the same number of nodes regardless of

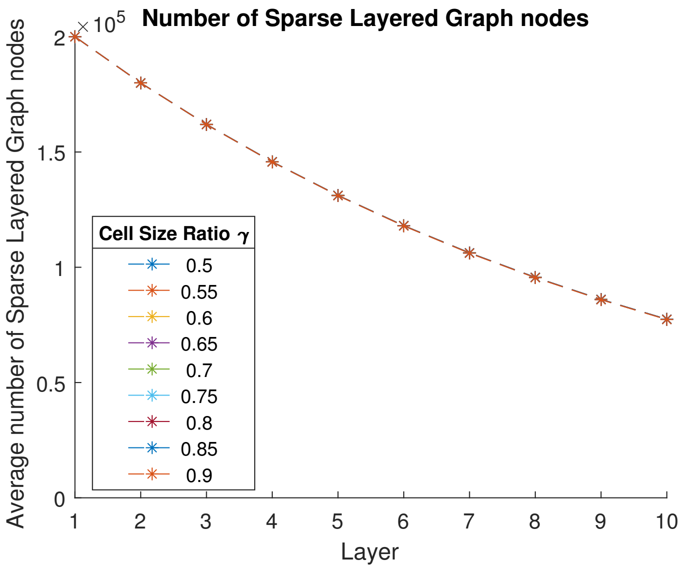

Figure 6. Average number of nodes in the Sparse Layered Graph for several cell size ratios. The average number of nodes in the SLG does not depend on the exact cell size ratio.

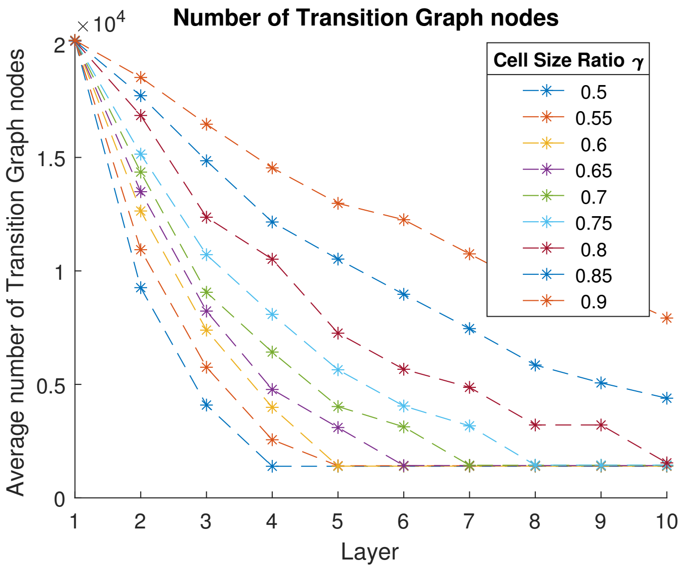

Figure 7. Average number of nodes in the Transition Graph for several cell size ratios. In contrast to the SLG, the average number of nodes in the TG depends on the exact cell size ratio.

the initial cell size with border cells being the only exception. Likewise for the cell size ratio, the number of nodes and edges in the TG will however be influenced by the number of initial cells. The reason is the same as stated for the cell size ratio.

With respect to the execution speed, our prototype is currently slightly slower than Dijkstra's algorithm on a corresponding layer, but still within real-time bounds and time requirements for path planning in robotics scenarios (data not shown). There is an obvious reasons: the implementation focus was not on execution speed but on algorithmic correctness. Therefore we allowed the usage of sub-optimal containers to hold parts of the data structure to be able to examine the algorithms in detail. Furthermore,

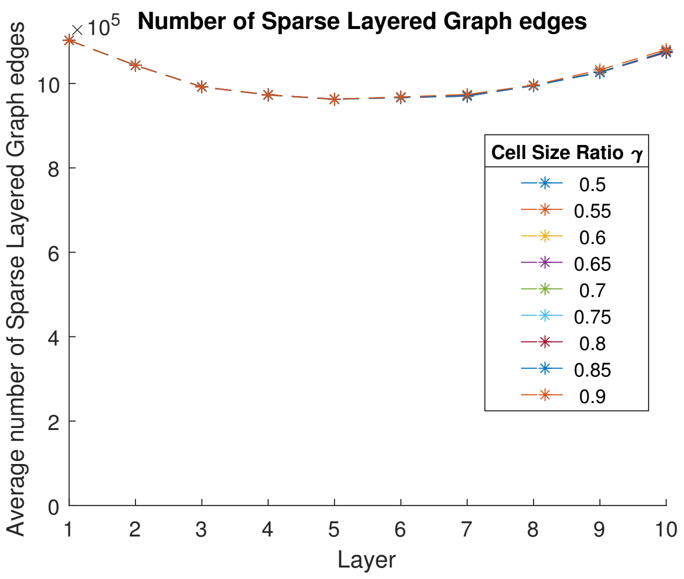

Figure 8. Average number of edges in the Sparse Layered Graph for several cell size ratios. As expected, the number of edges does not depend on the cell size ratio. The increase of number of edges in higher layers can be explained by the contraction process, which leads to graphs which are stronger connected on higher layers.

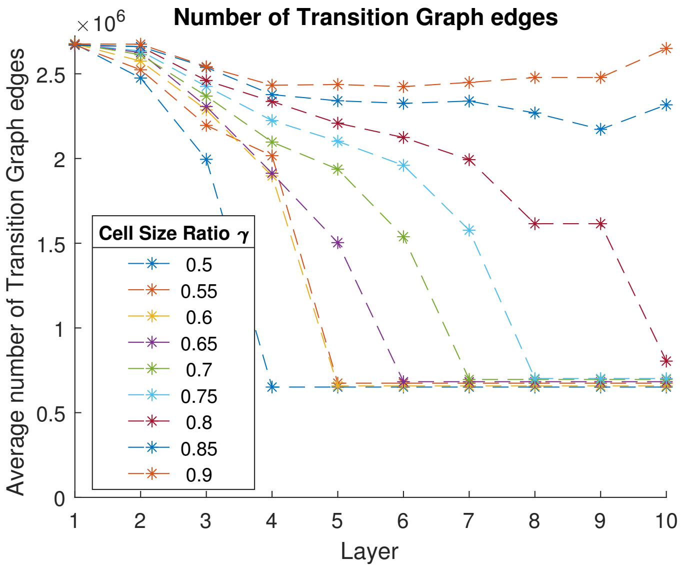

Figure 9. Average number of edges in the Transition Graph for several cell size ratios. Likewise the number of nodes, the number edges strongly depends on the cell size ratio.

the implementation currently requires a cumbersome data lookup during the `getDist` computation to merge data from **S** and **T**. We expect speed improvements as soon as these issues are addressed.

7 Conclusion & Future Work

In this work we presented our results towards distributed routing. Our goal was not a run-time improvement of existing methods, but to find, analyze, and characterize a solution that has the potential for locally distributed computing. This means that most operations should be performed on a small subset of the overall data that can be distributed. We developed data structures that are strongly inspired by biological neural networks that are essential for spatial navigation in rodents. We analyzed the run-time complexity of the algorithms that construct said data structures and operate on the stored data. The results show that the construction and query algorithms reside in good or acceptable run-time complexity. The discrete layer structure and the regular clustering of the data will make it comparatively easy to distribute it across several participating hosts.

Future work will therefore concentrate on improved query mechanisms and a fully distributed implementation. The construction mechanism and data retrieval of our data structure have remarkable similarities to peer-to-peer network technology. Consequently, we will analyze if our data structure is implementable as a so called overlay to existing peer-to-peer network protocols or, if not, which extensions are required. Meanwhile, we seek to find a smart way to distribute the Transition Graph, which in its current description is non-trivial.

Another major focus of future work will entail dynamic updates to the data. Ultimately, the data structures should be employed in distributed robotics scenarios with massive numbers of agents, all of which may generate new data. We think that, due to the regularity of the cluster arrangement, novel data will have to be processed only locally in a few cells and huge changes to the data can be avoided.

Acknowledgment

We would like to thank Cristian Axenie and Marcello Mulas for many valuable discussions about the data structure and remarks on the manuscript.

References

[1] R. Agrawal and H. V. Jagadish, “Efficient search in very large databases,” in *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB ’88, (San Francisco, CA, USA), pp. 407–418, Morgan Kaufmann Publishers Inc., 1988.

[2] M. A. W. Houtsma, P. M. G. Apers, and S. Ceri, “Distributed transitive closure computations: The disconnection set approach,” in *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB ’90, (San Francisco, CA, USA), pp. 335–346, Morgan Kaufmann Publishers Inc., 1990.

[3] M. J. Egenhofer, “What’s special about spatial? database requirements for vehicle navigation in geographic space (extended abstract).,” in *SIGMOD Conference* (P. Buneman and S. Jajodia, eds.), pp. 398–402, ACM Press, 1993.

[4] N. Jing, Y.-W. Huang, and E. A. Rundensteiner, “Hierarchical optimization of optimal path finding for transportation applications,” in *Proceedings of the Fifth International Conference on Information and Knowledge Management*, CIKM ’96, (New York, NY, USA), pp. 261–268, ACM, 1996.

[5] N. Jing, Y. W. Huang, and E. A. Rundensteiner, “Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation,” *Knowledge and Data Engineering*, vol. 10, no. 3, pp. 409–432, 1998.

[6] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, “Route planning in transportation networks,” 2015.

[7] R. Smith, M. Self, and P. Cheeseman, “Estimating uncertain spatial relationships in robotics,” in *Autonomous robot vehicles*, pp. 167–193, Springer, 1990.

[8] H. Durrant-Whyte and T. Bailey, “Simultaneous localization and mapping: part i,” *IEEE Robotics Automation Magazine*, vol. 13, pp. 99–110, June 2006.

[9] S. Thrun and M. Montemerlo, "The graph slam algorithm with applications to large-scale mapping of urban structures," *The International Journal of Robotics Research*, vol. 25, no. 5-6, pp. 403–429, 2006.
[10] G. Grisetti, G. D. Tipaldi, C. Stachniss, W. Burgard, and D. Nardi, "Fast and accurate slam with rao–blackwellized particle filters," *Robotics and Autonomous Systems*, vol. 55, no. 1, pp. 30–38, 2007.
[11] M. Milford and G. Wyeth, "Persistent navigation and mapping using a biologically inspired slam system," *The International Journal of Robotics Research*, vol. 29, no. 9, pp. 1131–1153, 2010.
[12] S. Thrun and Y. Liu, "Multi-robot slam with sparse extended information filers," in *Robotics Research. The Eleventh International Symposium*, pp. 254–266, Springer, 2005.
[13] A. Howard, "Multi-robot simultaneous localization and mapping using particle filters," *The International Journal of Robotics Research*, vol. 25, no. 12, pp. 1243–1256, 2006.
[14] N. Waniek, J. Biedermann, and J. Conradt, "Cooperative slam on small mobile robots," in *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 1810–1815, Dec 2015.
[15] C. Knoblock, "Search Reduction in Hierarchical Problem Solving," in *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)",*, vol. 2, (Anaheim, California, USA), pp. 686–691, 1991.
[16] F. Schulz, D. Wagner, and C. Zaroliagis, *Using Multi-level Graphs for Timetable Information in Railway Systems*, pp. 43–59. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002.
[17] M. Holzer, F. Schulz, and D. Wagner, "Engineering multilevel overlay graphs for shortest-path queries," *J. Exp. Algorithmics*, vol. 13, pp. 5:2.5–5:2.26, Feb. 2009.
[18] J. Maue, P. Sanders, and D. Matijevic, "Goal-directed shortest-path queries using precomputed cluster distances," *Journal of Experimental Algorithmics (JEA)*, vol. 14, p. 2, 2009.
[19] Y.-W. Huang, Y. wu Huangy, N. Jing, and E. A. Rundensteiner, "Hierarchical path views: A model based on fragmentation and transportation road types," 1995.
[20] Y.-W. Huang, N. Jing, and E. Rundensteiner, "A hierarchical path view model for path finding in intelligent transportation systems," *GeoInformatica*, vol. 1, no. 2, pp. 125–159, 1997.
[21] S. Jung and S. Pramanik, "An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps," *IEEE Transactions on Knowledge and Data Engineering*, vol. 14, no. 5, pp. 1029–1046, 2002.
[22] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, (Philadelphia, PA, USA), pp. 46–59, Society for Industrial and Applied Mathematics, 2007.
[23] R. Zhong, G. Li, K. L. Tan, L. Zhou, and Z. Gong, "G-tree: An efficient and scalable index for spatial search on road networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, pp. 2175–2189, Aug 2015.
[24] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner, "High-performance multi-level graphs," in *IN: 9TH DIMACS IMPLEMENTATION CHALLENGE*, pp. 52–65, 2006.
[25] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner, "High-performance multi-level routing," 2008.
[26] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, "Recent advances in graph partitioning," *CoRR*, vol. abs/1311.3144, 2013.
[27] T. Hafting, M. Fyhn, S. Molden, M.-B. Moser, and E. I. Moser, "Microstructure of a spatial map in the entorhinal cortex," *Nature*, vol. 436, pp. 801–806, Aug 2005.
[28] E. I. Moser and M. B. Moser, "A metric for space," *Hippocampus*, vol. 18, no. 12, pp. 1142–1156, 2008.
[29] H. Sanders, C. Rennó-Costa, M. Idiart, and J. Lisman, "Grid cells and place cells: An integrated view of their navigational and memory function," *Trends in Neurosciences*, vol. 38, no. 12, pp. 763 – 775, 2015.
[30] M. B. Moser, D. C. Rowland, and E. I. Moser, "Place cells, grid cells, and memory," *Cold Spring Harb Perspect Biol*, vol. 7, p. a021808, Feb 2015.
[31] X.-x. Wei, J. Prentice, and V. Balasubramanian, "A principle of economy predicts the functional architecture of grid cells," *eLife*, 2015.
[32] M. Stemmler, A. Mathis, and A. V. M. Herz, "Connecting multiple spatial scales to decode the population activity of grid cells," *Science Advances*, vol. 1, no. 11, 2015.
[33] H. Stensola, T. Stensola, T. Solstad, K. Froland, M.-B. Moser, and E. I. Moser, "The entorhinal grid map is discretized," *Nature*, vol. 492, pp. 72–78, Dec. 2012.
[34] S. Sreenivasan and I. Fiete, "Grid cells generate an analog error-correcting code for singularly precise neural computation," *Nat Neurosci*, vol. 14, pp. 1330–1337, Oct 2011. Article.
[35] U. M. Erdem and M. E. Hasselmo, "A goal-directed spatial navigation model using forward trajectory planning based on grid cells," *Eur J Neurosci*, vol. 35, pp. 916–931, Mar 2012. 22393918[pmid].
[36] U. M. Erdem, M. J. Milford, and M. E. Hasselmo, "A hierarchical model of goal directed navigation selects trajectories in a visual environment," *Neurobiology of Learning and Memory*, vol. 117, pp. 109 – 121, 2015. Memory and decision making.
[37] M. Mulas, N. Waniek, and J. Conradt, "Hebbian plasticity realigns grid cell activity with external sensory cues in continuous attractor models," *Front Comput Neurosci*, vol. 10, p. 13, Feb 2016. 26924979[pmid].
[38] S. Fortunato, "Community detection in graphs," *Physics reports*, vol. 486, no. 3, pp. 75–174, 2010.

Nicolai Waniek received the Diploma in computer science from Ulm University, Ulm, Germany, in 2012. He is currently working toward a Ph.D. degree in the Neuroscientific System Theory Group at Technische Universität München, Germany.

He is interested in self-organization and self-construction of dynamic systems in computational neuroscience and robotics, parallel and distributed algorithms, and computer vision.

Edvarts Berzs acquired his Master of Science in Electrical Engineering and Information Technology from Technical University Munich (TUM), Germany. He is currently working in the industry on LTE-Advanced topics.

He is interested in distributed algorithms, mobile communication networks, LTE-A, 5G and competitive programming.

Jörg Conradt is W1 Professor at the Technische Universität München in the Department of Electrical and Computer Engineering. His research group is affiliated with TUM's Competence Center on Neuroengineering and the Munich Bernstein Center for Computational Neuroscience. He holds an M.S. Degree in Computer Science/Robotics from the University of Southern California, a Diploma in Computer Engineering from TU Berlin, and a Ph.D. in Physics/Neuroscience from ETH Zurich. His research investigates key principles by which information processing in brains works, and applies those to real-world interacting technical systems: http://www.nst.ei.tum.de.