Scalability of Customizable Route Planning

Research Project - Master Thesis

Final Report

By Morris Breed
Utrecht University

Supervisors
Dr. E.J. van Leeuwen (UU)
Prof. dr. R.H. Bisseling (UU)
P. Agterberg (Ortec)

July, 2021

Contents

List of Symbols

This list contains some of the symbols used in this report.

C^l	Partition on level l
H	Multilevel overlay graph $\{H_1, H_2, \ldots, H_{L-1}\}$ with top level $L - 1$
B_C^l	Maximum number of boundary points in a cell on level l excluding its subcells
R_C^l	Maximum number of shortcuts of a cell on level l excluding its subcells
B	Total number of boundary points in the overlay graph
B_C^l	Maximum number of boundary points in a cell on level l including its subcells
$C^l(v)$	Cell on level l containing node v
C_i^l	Cell number i on level l
E_C^1	Maximum number of base level edges in a base level cell
K	Split size of the graph, i.e. the number of subcells per cell
k	Number of iterations during PBP-2 query
$l(e)$	Level of the cell of which edge e is a shortcut
L	Split level of the graph
$l_{st}(v)$	Query level of node v
$P(S)$	Collection of source processes during a many-to-many query
$p(s)$	Source process during a one-to-one query
$P(T)$	Collection of target processes during a many-to-many query
$p(t)$	Target process during a one-to-one query
P	Collection of processes
R	Total number of shortcuts in the overlay graph
R_C^l	Maximum number of shortcuts of a cell on level l including its subcells
S	Collection of sources during many-to-many query
S_p	Collection of sources contained in the subgraph of process p
T	Collection of targets during many-to-many query
T_p	Collection of targets contained in the subgraph of process p
V_C^1	Maximum number of nodes in a base level cell

$X_b(p)$ Processed nodes during a backward Dijkstra/CRP search on process p

X_b Processed nodes during a backward Dijkstra/CRP search

$X_f(p)$ Processed nodes during a forward Dijkstra/CRP search on process p

X_f Processed nodes during a forward Dijkstra/CRP search

1 Introduction

Customizable Route Planning *Customizable Route Planning* (CRP) [8] is one of the current state-of-the-art shortest path algorithms. Although Dijkstra's classic algorithm runs in log-linear time with little overhead, computing a shortest path on large graphs still takes multiple seconds. CRP computes shortest paths on continental-sized graphs within a couple of milliseconds.

CRP was speci cally designed for road networks. Other algorithms may achieve slightly faster query times, but CRP distinguishes itself from other shortest path algorithms by being robust to metric changes, being able to incorporate new metrics quickly and by having the ability to simultaneously store data for multiple metrics. Meanwhile, its query times are fast enough for interactive applications. CRP uses two separate preprocessing stages, a metric-independent and a metric-dependent stage, before conducting a Dijkstra-like search in the query stage.

Cloud environment Software applications are increasingly o ered as cloud services these days. The developed software is not installed on on-premise servers but instead hosted on servers of a cloud provider. Users can access the software over the internet, without installing any software locally. The software, or certain components thereof, can be installed on multiple servers, which can enhance the *scalability*: the property of a system to handle a growing amount of work [5]. Another way of handling peaks in work load is to start multiple *instances* of the same server. Companies commonly pay cloud providers per server usage, where there exist cheaper servers (with less memory and computing power) and more expensive ones.

Ortec has decided to o er CRP as a cloud service to their customers. CRP's robustness to metric changes and its ability to handle multiple metrics e ciently played a key role in convincing Ortec to develop software using the algorithm. The most straightforward way of running CRP, or any similar algorithm, in the cloud would be by having one server store all data (e.g. the entire map, preprocessed data). Users then send their queries to this server, which computes and returns the result. Queries arrive non-uniformly in most applications: there will be highs and lows in the number of queries at a certain time. At the moment the number of queries becomes too large for one server, a new instance of that server is started, which would have to load all data as well. This can take quite some time and should be done before demand requires it, leading to a lot of idle time and thus costs.

Project's aim In this research project we focus on the query stage of CRP. Our goal is to nd a distributed memory approach to the query stage of CRP, such that scalability is improved. Instead of using a few large, heavy processes, we aim to design algorithms that employ multiple small interacting processes. We use the word \process" to indicate a (cloud) server or processor throughout this report. Thus, the system becomes more nimble and more scalable in the presence of large numbers of requests. In the resulting approach, query times should stay fast and memory usage and total resource consumption should be kept low.

We present two di erent approaches to solving CRP queries in a distributive fashion: *Partitioning-Based Parallelism* (PBP) and *On-Demand Loading* (ODL). For PBP, we use a partitioning of the graph to distribute subgraphs among di erent processes. These processes communicate with each other to answer queries. For ODL, we store the graph on one process and let a smaller process answer queries by receiving the query's search graph from the global process. For each algorithm, we examine its bene ts and drawbacks; its consequences with respect to correctness, running time, memory use and scalability and whether it can be e ciently extended to an algorithm answering many-to-many queries. Many-to-many queries ask for shortest paths from a set of source nodes to a set of target nodes. In addition, for the PBP approaches, we investigate how we distribute the

graph data among the processes, what communication is needed between the processes in order to answer queries, and whether we can answer queries with source and target on one process's subgraph without communication with other processes. For ODL, we look into the way we store the graph on the external process, so that we can e ciently access the necessary data, what additional data we store on the external process, and what data we keep on the process answering the queries.

Previous work A distributed memory approach to the query stage of CRP has not been developed yet. The introductory paper [8] describes a basic method of parallelizing the metric-dependent preprocessing stage, but this method is not suitable to be extended to the query stage. There has been research into distributed memory versions of Dijkstra's algorithm, but these algorithms usually do not make full use of the advantages of CRP. One of our PBP algorithms is an extension of a parallel Dijkstra algorithm introduced by Tang et al [36]. Research by Hamme [15] describes a method of performing CRP queries on mobile devices, which typically have limited memory capacity. We use this algorithm as the basis of our ODL algorithm.

Results We present two versions of PBP, which we call PBP-1 and PBP-2, and one ODL approach. PBP-1 answers queries using the two processes whose subgraphs contain the source node and the target node. PBP-2 uses multiple label-correcting iterations on all processes to answer queries. For ODL, we load a query's search graph when we receive a query and calculate a shortest path by performing a CRP search on the loaded graph data. For all three algorithms, we designed a version that answers one-to-one queries and a version that answers many-to-many queries. We show theoretical analyses of the algorithms, discuss implementations of them and show results of several experiments we conducted in order to test their performance, scalability and total resource consumption.

Methods of analysis In our theoretical analysis, we focus on the running time of the algorithms and their reliance on the performance of the communication. We conducted analyses using the BSP model [37] and the big-O notation. We investigate the communication frequency and message size, to obtain an indication of how well the algorithms should perform in practice.

During the experiments, our main focus is on the scalability of each approach: how e ciently does the algorithm handle peaks and low points in the number of received requests? In order to test the scalability, we subject the algorithms to di erent types of loads (two arti cial scenarios and two taken from real customer data provided by Ortec) and test the consequences to the query times and the amount of used resources (number of servers, memory usage and work load distribution among processes). Furthermore, we look closely at the communication between the processes, as this is the prominent source of potential performance degradation. We empirically examine the communication frequency and message size and examine if that strokes with our expectations based on our theoretical analyses. Finally, we are interested in two more properties of the algorithms: the in uence of the query distance (the distance between the source and target) on the query times and the impact of the natural cuts, which CRP exploits during preprocessing, on the overall performance. To this end, we perform experiments with varying query distance and perform experiments on arti cial road networks.

The outline of this report is as follows. We will start by treating the existing relevant literature in Section 2. In this section, we discuss CRP's place in the current landscape of routing engines, give a detailed explanation of the algorithm and treat literature regarding distributed memory approaches to CRP and Dijkstra. Furthermore, we discuss the BSP model, which we use during

our theoretical analysis of the algorithms, and the GREREC model that we use to generate random road networks for our experiments. In Section 3, we describe our three di erent algorithms, PBP-1, PBP-2 and ODL and their theoretical analyses. We continue by describing our implementations of the algorithms in Section 4. Finally, in Section 5, we discuss the experiments we conducted and show their results, followed by our nal conclusions.

2 Literature

2.1 Routing algorithms

Routing engines must be fast and must not require too much space to be suitable for interactive applications. Current state-of-the-art routing algorithms have query times of just a few milliseconds. In addition, they have to satisfy the following requirements to be applicable in a realistic setting [3]:

- Query times must be robust to metric changes,
- Incorporating a new metric must be quick,
- Turn costs and turn restrictions must be accounted for, and
- Multiple metrics (cost functions) must be supported.

Users of real-world routing engines may want to use di erent metrics, e.g. shortest distance, avoid toll roads and minimize CO2 emissions. For all these di erent metrics, the query times should be fast. Finally, incorporating a new metric quickly is necessary to have the ability to account for current tra c conditions and to give the user the opportunity to change to a new metric quickly. Turn costs and turn restrictions were often neglected while developing routing engines, because the assumption was widely supported that any algorithm could easily be adjusted to handle these e ciently. However, Delling et al. [8] showed that this is not the case and that most algorithms have a signi cant performance penalty when incorporating turns, especially if the turns are represented space-e ciently. Supporting multiple metrics implies that metric-speci c data structures should be as small as possible, making it feasible to keep data for multiple metrics in memory.

In this section, we sketch the current landscape of routing algorithms. We will describe di erent kinds of techniques that are used in e cient routing engines. First, we brie y recall Dijkstra's famous algorithm, after which we will touch upon more advanced types of techniques: *goal-directed techniques*, *separator-based techniques*, *hierarchical techniques* and *bounded-hop techniques*. We use the work of Bast et al. [3] as the basis for our description of these techniques. The paper chose these particular techniques because, according to the paper, they quickly made real-life impact, as they address problems that need solving before a routing algorithm can be used for large-scale interactive applications.

2.1.1 Dijkstra's algorithm

For computing a shortest path from source node s to target node t, Dijkstra's algorithm maintains a priority queue of nodes v, ordered by their tentative distances $D(v)$ to s. Initially, we set $D(s) = 0$ and $D(v) = \infty$ for all nodes $v \neq s$. The algorithm continues by *scanning* the node u with the minimum tentative distance to s at every iteration: it extracts u from the priority queue and for each outgoing edge (u, v), it checks if $D(v)$ can be improved by using the edge. If so, $D(v)$ is updated and v is added to the queue.

Dijkstra's algorithm possesses the *label-setting property*: once a node v is scanned, $D(v)$ is guaranteed to be optimal, i.e. $D(v) = d(s, v)$. This implies that once we scan target node t, we can stop searching, as we have found a shortest distance from s to t. The counterpart of this property is the *label-correcting property*, which implies that all distances are considered temporary until the nal step of the algorithm. These properties will play an important role in the PBP approach of solving our problem.

The running time of Dijkstra's algorithm on sparse graphs such as road networks is $O(|E| \log(|V|))$ when they are connected, using a Fibonacci heap as priority queue. A natural optimization of the algorithm is to use *bidirectional search*: simultaneously running a forward search from s and a backward search from t. When both searches meet, the tentative distance $D(s, t)$ is updated. We can stop iterating when the minimum key of both searches exceeds the tentative distance, as the distance cannot be improved further in that case, so we have $D(s, t) \leq d(s, t)$. On road networks, using bidirectional search results in visiting roughly half as many nodes as in unidirectional search [3]. We show the bidirectional version of Dijkstra's algorithm in pseudocode in Algorithm 1 (Appendix A).

2.1.2 Other techniques

In every iteration of Dijkstra's algorithm, the search moves to the node with the smallest tentative distance to s. However, this gives no guarantee that we actually move towards our target t. Goal-directed techniques aim at nudging the search into the right direction. One classical example is the *A* search* algorithm [16], which is essentially a modification of Dijkstra's algorithm. It uses a potential function $\pi : V \to \mathbb{R}$ that maps each node v to a lower bound on $d(v, t)$. Instead of using just the tentative distance $D(v)$ from s as key for node v in the priority queue, the algorithm uses $D(v) + \pi(v)$. This causes nodes closer to t to be scanned sooner. Other goal-directed algorithms are *ALT* [14], *Geometric Containers* [38] and *Arc Flags* [23, 17].

Separator-based techniques use preprocessing to compute a separator of the graph and use that separator to construct an *overlay graph*. A subset of the nodes is a *vertex separator* if it decomposes the graph into different cells when it is removed. Similarly, an *arc separator* is a subset of the arcs such that when removed, the graph is split into different cells. An overlay graph is a subgraph of the original graph, in which the original distances are preserved. In a separator-based algorithm, an overlay graph is constructed by adding shortcuts to a separator S: the distances between the nodes or arcs in the separator are calculated in a preprocessing phase, which we call *shortcuts*. The overlay graph is much smaller than the original graph, so using these precomputed distances while answering a query speeds up the process. CRP uses a separator-based approach and computes its overlay graph from an arc separator. We treat the CRP algorithm extensively in the next section. Another example of a separator-based algorithm is *Hierarchical MulTi* (HiTi) [20].

A road network has a natural hierarchy: important roads such as highways appear more frequently in shortest paths than local roads. Hierarchical techniques use this fact to speed up queries. An algorithm that obtains impressive query times using this approach is *Contraction Hierarchies* (CH) [12]. In preprocessing it orders the nodes heuristically by importance. Then, the nodes are *contracted* following this order, starting from the least important node. Contracting a node v means first removing it from the graph. Then, if v was contained in the unique shortest path between two of its neighbors u and w, we add a shortcut between u and w to preserve all shortest path lengths. Queries are solved by performing a bidirectional search on the graph with shortcuts, only using arcs to nodes of a higher rank. On simplified models of road networks, CH outperforms CRP: both preprocessing and queries are faster [3]. In realistic models, we add turn costs and turn restrictions and place some extra demands: multiple cost functions should be supported, query times must be robust to the choice of cost function, and new cost functions should be incorporated quickly. The performance of CH is significantly worse on metrics other than travel times without turn costs and therefore less suited to use on realistic models than CRP [3].

Lastly, we treat bounded-hop techniques. These techniques use preprocessing to calculate distances between nodes and add corresponding shortcuts to the graph. During a query, the destination can be found in fewer *hops*, resulting in faster query times. Precomputing the distances between all

nodes, giving us *single hop paths*, is problematic for large graphs in terms of preprocessing time and especially in terms of storage. The *Hub Labeling* (HL) algorithm [6, 11] uses a two-hop approach. For each node v, the algorithm stores a set of labels, which contain shortest distances from v to certain nodes. These nodes are chosen so that for any pair of vertices u and v, the distance $d(u,v)$ can be computed by only looking at the labels of u and the labels of v. HL achieves the fastest known query times on road networks, but has one big disadvantage: storing all these labels causes its space usage to be significantly higher than that of competing methods [3].

2.2 Customizable Route Planning

CRP was introduced by Delling et al. and meets all requirements for real-world routing engines that we described in Section 2.1 [8]. All other methods fail at least one of them, according to the paper. Hierarchical methods and goal-directed techniques are too sensitive to metric changes: query times are significantly worse for unfavorable metrics. Bounded-hop techniques require too much storage to be applicable in a realistic setting. Separator-based techniques such as CRP only use the topology of the graph for speeding up, so by design they work equally well for any metric, satisfying the first requirement. They were deemed too slow for interactive applications in the past, but combining new concepts and careful engineering led to CRP, which proved to be fast enough and to meet all other aforementioned requirements [8]. In this section, we will describe the algorithm, after which we will explain how it satisfies these requirements.

CRP consists of three stages: metric-independent preprocessing, metric-dependent preprocessing (also called *customization*) and the query stage. In the first stage, the topology of the network is determined: a partition of the graph is computed and then used to build an overlay graph. The second stage takes the cost function and computes the costs of the shortcuts that connect the boundary nodes within each cell of the partition. Then, for each query, the query stage consists of running a multilevel version of bidirectional Dijkstra on the union of the cell containing the source, the cell containing the target and the overlay graph. We will treat each stage separately and in more depth below.

2.2.1 Metric-independent preprocessing

In metric-independent preprocessing, the topological properties of the graph are determined. These are static properties of each road segment or turn, such as length, number of lanes, road category, speed limit or turn type. These properties do not change often in road networks, so this part of preprocessing does not need to run often. Therefore, the running time of this phase is not a priority.

Preprocessing begins with computing a *multilevel partition* of the input graph G. A partition of a graph is a family $C = \{C_0, C_1, \ldots, C_k\}$ of cells, where each node is contained in exactly one cell C_i. A multilevel partition is a family $\{C^0, C^1, \ldots, C^L\}$ of partitions of the graph, where l denotes the level of partition $C^l = \{C_0^l, C_1^l, \ldots, C_{m_l}^l\}$. CRP uses a *nested* multilevel partition, which means that for each cell C_i^l on level $l < L$, there exists a cell C_j^{l+1} on level $l+1$ such that $C_i^l \subseteq C_j^{l+1}$. We then call C_i^l a *subcell* of C_j^{l+1} and C_j^{l+1} a *supercell* of C_i^l. Partition C^0 only contains singletons and we define C^L to consist of one cell, which contains the entire graph.

Overlay graph H corresponding to a graph G with partition C contains all *boundary nodes* (*or boundary points*) and *boundary arcs* of C's cells. Boundary nodes are nodes with at least one neighbor in a different cell and boundary arcs connect two nodes in different cells. Since we use a multilevel partition, the corresponding overlay graph $H = \{H_1, \ldots, H_{L-1}\}$ has multiple levels as well. So, H_l consists of all boundary nodes and boundary arcs of partition C^l. For each cell on each level of the overlay graph, a clique is built: between each pair of boundary nodes a shortcut is

constructed. The costs of these shortcuts are computed in the customization stage. Naturally, we do not compute shortcuts for the singleton graphs in C^0 and the entire graph in C^L. We call the original graph G level 0 or the *base level*. A *base level cell* consists of the nodes and edges of the original graph contained in one level-1 cell.

In the customization phase, the costs of the shortcuts between the boundary points of each cell are computed. During the query stage, for each boundary point we process, we have to check each shortcut to other boundary points for possible improvements. Therefore, customization and query times of CRP heavily depend on the number of boundary arcs in the partition [8]. Graph partitioning algorithm *PUNCH* [9] finds partitions with roughly half as many boundary arcs as general-purpose partitioners do and is therefore a good fit for CRP. It exploits natural cuts that exist in road networks, such as rivers, mountains and highways. PUNCH identifies these cuts by performing local maximum flow computations. Most popular partitioners are faster than PUNCH, but as we mentioned, running time is not our main concern. The customization and query times, however, are important. The customization phase is executed every time we incorporate a new metric. We want to be able to incorporate new metrics quickly and compute queries in interactive applications, so we prioritize minimizing the number of boundary arcs.

At the moment, Ortec uses graph-partitioner *KaHIP* [34]. This algorithm uses the same ideas as PUNCH, as it also uses local maximum flow computations to minimize the number of boundary arcs in the partition. KaHIP takes as input an integer K indicating the *split size* and returns a partition of the graph into K balanced parts. Executing the algorithm recursively onto the parts of the partitioning results in the desired multilevel partition (see Figure 1).

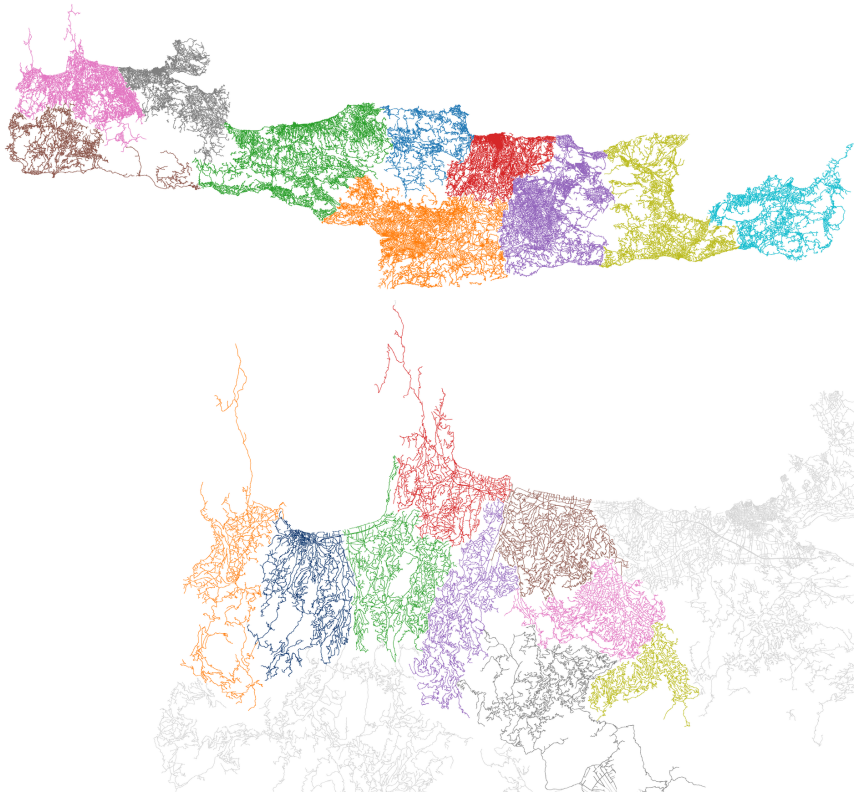

Figure 1: Multilevel partition of the map of Crete. The top picture shows the first partitioning. The bottom picture shows the partitioning of the pink cell (top left) in the second partitioning step.

2.2.2 Customization

With the metric-independent data from the first preprocessing stage and the desired cost function, we can enter the customization stage. This stage is executed every time a new metric is incorporated, so running time is important. In customization, we compute an all-pairs shortest path matrix W_i^l for each cell C_i^l of the partition. We do this bottom-up: we first calculate all distances for cells on the first level by running Dijkstra's algorithm on the graph G from the boundary nodes, before we move on to the higher levels of the partition. For cells on level $l > 1$, we can speed up the search by using overlay graph H_{l-1} instead of G, as this level is already processed. Using the Bellman-Ford algorithm instead of Dijkstra also improves performance [8]. Even faster customization times are achieved by performing Bellman-Ford simultaneously from multiple sources: for Bellman-Ford from k different sources, we maintain distance labels $d_1(v), \ldots, d_k(v)$ for each node v. Every time an arc (v, w) is scanned, we try to improve all labels $d_i(w)$ for $i \leq k$. For this approach we need as many rounds as the worst of all k executions, but by storing the labels of each node contiguously in memory, we improve locality and enable instruction-level parallelism.

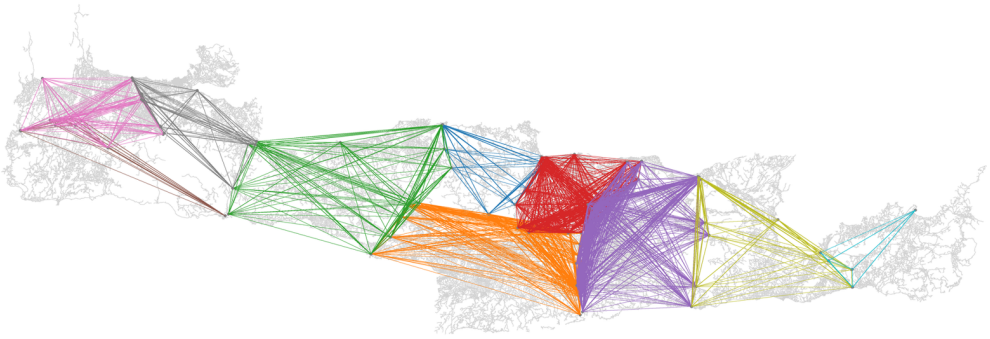

Figure 2: The shortcuts on the highest level of the overlay graph, which are calculated during the metric-dependent preprocessing phase.

2.2.3 Queries

The result of the preprocessing stages is a multilevel overlay graph, for which the shortcut costs between each pair of boundary points for each cell of the (multilevel) partition are already computed. In the query stage, the focus of this project, we use this information to speed up the Dijkstra search.

One-to-one queries One-to-one queries calculate a shortest path between a source s and a target t. A CRP query performs a bidirectional Dijkstra search on overlay graph $H = \{H_1, \ldots, H_{L-1}\}$ and the base level cells containing s and t. These two cells are the only parts of the original graph we use. During the remainder of the query we only use the overlay graph with its shortcuts for which we already computed the distances in the customization phase. The main observation is that we can skip cells of the partition that do not contain s or t and instead use shortcuts. This is done in the following way: each time we scan a node, we check if it is a boundary node on some level of the partition. If so, we scan it using the overlay graph on the *query level* of the node. The query level $l_{st}(v)$ of a node v is the highest level such that v is not in the same cell as s or t. We prove in Proposition 1 that doing so indeed yields a shortest path. We use the following notation in the proof (and in the entire report): $C^l(v)$ denotes the level-l cell containing node v and $l(e)$ denotes the level of the cell of which e is a shortcut.

Proposition 1. [18] Let $H = \{H_1, \ldots, H_{L-1}\}$ be an overlay graph corresponding to a graph G and let s and t be two connected nodes in G. Then there exists a shortest $s - t$ path P consisting of edges $(e_0, e_1, \ldots, e_n)$, where every edge $e = (v, w)$ is a level-$l_{st}(v)$ shortcut in H or a base level edge if $l_{st}(v) = 0$.

Proof. Let $P = (e_0, e_1, \ldots, e_n)$ be a shortest $s - t$ path in overlay graph H. Let $e = (v, w) \in P$ with $e \in H_l$ for some $l = L - 1$.

If $l = l_{st}(v)$, then $C^l(v) \in C^{l+1}(v)$, such that $C^{l+1}(v) = C^{l+1}(s)$ and $C^{l+1}(v) = C^{l+1}(t)$ by definition of $l_{st}(v)$. Therefore, shortest path P at some point has to exit $C^{l+1}(v)$. Let $P^1 = (v_1, \ldots, v, w, \ldots, v_k) \in P$ be the subpath through $C^{l+1}(v)$, where possibly $v_1 = v$ or $v_k = w$. Then (v_1, v_k) is a shortcut of $C^{l+1}(v)$, so we can swap P^1 with (v_1, v_k) in P without changing the path distance.

If $l < l_{st}(v)$, then $C^l(v) = C^l(s)$ or $C^l(v) = C^l(t)$ (or both) by definition of $l_{st}(v)$. Assume without loss of generality that $C^l(v) = C^l(t)$. Then there is a path connecting v and t only using levels lower than l. Let $P^1 = (v, \ldots, t)$ be a shortest such path. Level-l shortcut $e = (v, w)$ is constructed from edges in $C^l(v)$ and connects two level-l boundary points v and w. So, $(v, w, \ldots, t) \in P$ cannot have shorter distance than P^1 because of the triangle inequality. Therefore, swapping $(v, w, \ldots, t)$ with P^1 cannot increase the path distance.

We conclude that there is a shortest $s - t$ path consisting of edges $e = (v, w)$ on query level $l_{st}(v)$.

□

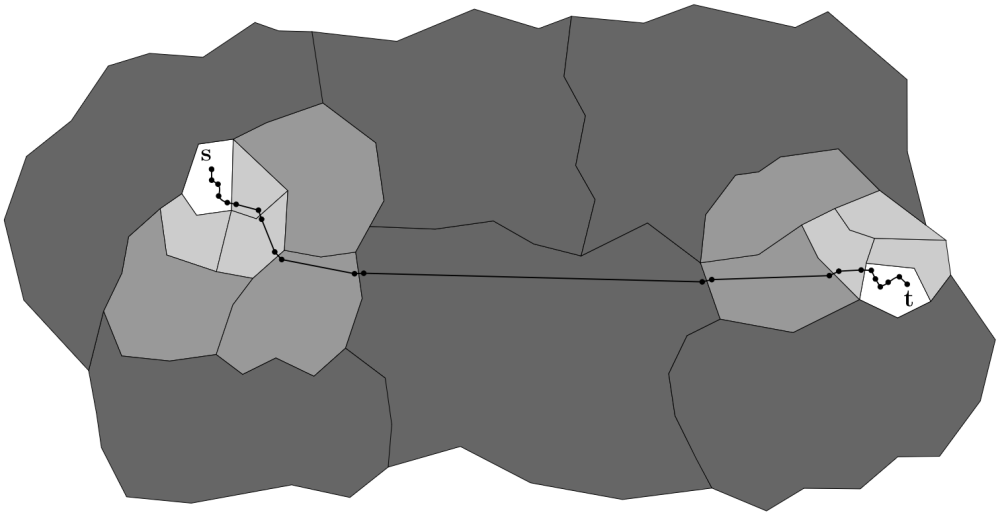

Figure 3: CRP query. Darker coloured cells indicate higher levels of the overlay graph.

When we use higher query levels to scan nodes, we operate on a higher level of the overlay graph, which implies we skip larger parts of the graph. Figure 3 illustrates a CRP query and shows the impact of scanning nodes on their query levels. Just as in regular bidirectional Dijkstra, we update tentative distance $D(s, t)$ whenever the forward and backward searches meet. When both searches only move to nodes on levels of the overlay graph on or above their current level, we get only *up-down paths*: paths $(e_0, e_1, \ldots, e_k, \ldots, e_l)$ where $l(e_i) < l(e_{i+1})$ for $i < k$ and $l(e_i) \geq l(e_{i+1})$ for $k < i < l$. Geisberger et al. [12] showed that if there exists a shortest $s - t$ path, then there also

exists a shortest path that is an up-down path. Their proof relies on the order of importance used in Contraction Hierarchies, but translates to the cell levels in CRP, as we show in Proposition 2.

Proposition 2. If there exists a shortest s t path, then there also exists a shortest s t path that is an up-down path.

Proof. Let P $pe_0;e_1;:::;e_nq$ be a shortest s t path in overlay graph H. Let e_i $pv;wq$ P P with e_i P H_{l_i} be the first local minimum, i.e. $e_{i\ 1}$ P $H_{l_{i\ 1}}$ with $l_{i\ 1}$ ¡ l_i and e_j P H_{l_j} with l_j ¡ l_i for some j ¡ i.

We examine two cases: either $C^{l_i\ 1}pvq$ $C^{l_i\ 1}psq$ or $C^{l_i\ 1}pvq$ $C^{l_i\ 1}ptq$ (case 1), or neither is true (case 2).

For case 1, without loss of generality assume $C^{l_i\ 1}pvq$ $C^{l_i\ 1}psq$. Then, a level $l_{i\ 1}$-shortcut cannot improve a shortest path from s to v (see the proof of Proposition 1). So, there exists a path P^1 only consisting of edges on levels lower than $l_{i\ 1}$ of length at most $dpe_{i\ 1}q$. Therefore, we can swap $pe_0;:::;e_{i\ 1}q$ € P with P^1, so that we use lower levels than $l_{i\ 1}$.

In case 2, a shortest path traverses through $C^{l_i\ 1}pvq$. Let P^1 $pv_1;:::;v;w;:::;v_kq$ € P, where possibly v_1 v or v_k w, be the subpath through $C^{l_i\ 1}pvq$. Then $pv_1;v_kq$ is a shortcut of $C^{l_i\ 1}pvq$, so we can swap P^1 with $pv_1;v_kq$ in P and we still have a shortest path.

So, if a shortest path contains a local minimum we can either decrease the level of the edge before the minimum or increase the level of the minimum itself without increasing the path distance. Therefore, we conclude that if there is a shortest path, there also exists a shortest path that is an up-down path. □

So, letting both the forward and the backward search only traverse to cells on levels equal or above their current levels will still yield a correct answer for our query. This helps to make queries more efficient, as they have to consider fewer nodes in the search. It also guarantees that the meeting point of the forward and backward search will be on the highest level of the overlay graph that is used in the entire query. Only considering up-down paths and scanning nodes on their query level causes the search graph of CRP to be much smaller than for Dijkstra's algorithm, which is illustrated in Figure 4. The query phase of CRP is presented in pseudocode in Algorithm 3 (Appendix A).

The number of boundary points per cell generally increases as we move to higher levels of the overlay graph [1]. Consequently, most calculations are done in the top level of the graph. Exploring the top level in both the forward and backward searches is therefore costly and can be avoided by stopping the search one level below the top level for one of the search directions. In our algorithms, we choose to explore the top level only during the forward search. This optimization is simple, but effective: it can halve the query times for many-to-many queries on large graphs [1].

Many-to-many queries For many-to-many queries, instead of trying to find a shortest path from node s to node t, we are interested in shortest paths from all nodes in some set S to all nodes in some set T. The straightforward approach would be to perform $|S|$ $|T|$ one-to-one queries, but Knopp et al. [22] describe a better way to do this. Their crucial idea is to execute only one forward search for each source and one backward search for each target, resulting in a total of $|S|$ $|T|$ searches. During each backward search, we keep track of the search spaces by using buckets. We associate a bucket $bpvq$ to each node v. This bucket stores pairs $pt;Dpv;tqq$, which represent the tentative distances of shortest paths from v to nodes t P T. These backward searches are performed for every t P T, obtaining distances $pt;Dpv;tqq$ for all nodes v. Then, during the forward searches, we maintain tentative distances $Dps;tq$ for all pairs $ps;tq$ P S T. Every time the forward search

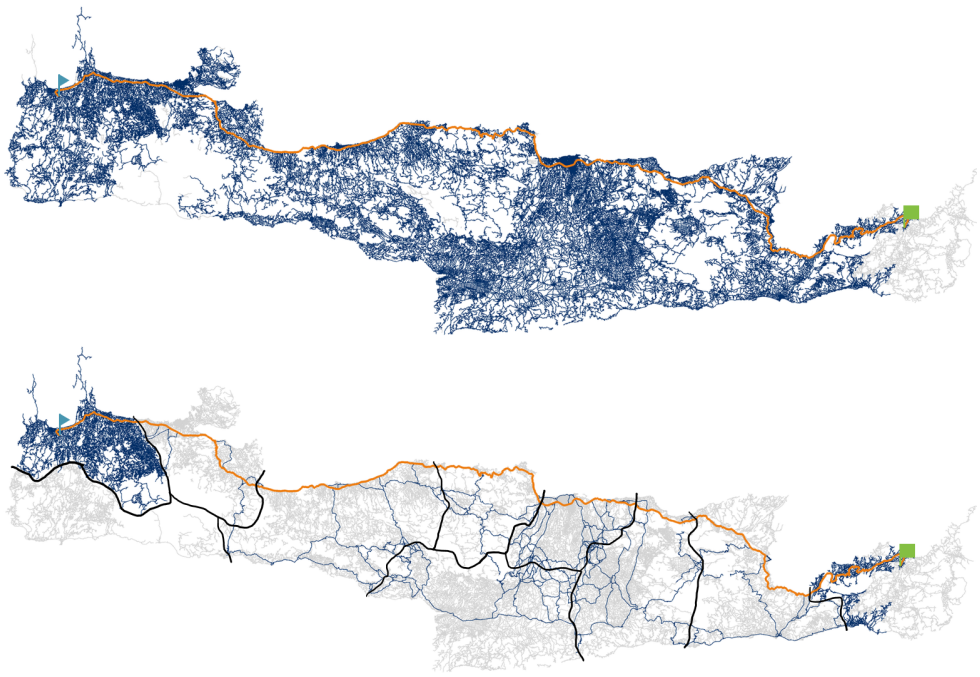

Figure 4: The search graph of Dijkstra's algorithm (top) compared to the search graph of CRP using a single-level partitioning (bottom) for a one-to-one query from the blue flag to the green flag on the map of Crete. The black lines indicate the cell borders of the overlay graph. An edge or shortcut is coloured blue if it is processed during the search.

scans a node v, after which $d(s, v)$ is known, the search goes through all pairs $(t, d(v, t))$ in bucket $b(v)$ and updates $D(s, t)$ whenever $d(s, v) + d(v, t) < D(s, t)$. We use this technique to answer many-to-many CRP queries as shown in Algorithm 4 (Appendix A).

2.2.4 Benefits of CRP

In this section, we explain how CRP satisfies the requirements of real-world routing engines we mentioned in Section 2.1. The robustness of query times to metric changes follows directly from the fact that CRP uses a separator-based technique and therefore does not exploit network characteristics like road hierarchy.

Dividing the preprocessing phase into two different parts is a crucial property of CRP. The metric-independent part has to be run infrequently, because the topological characteristics of a network do not change often. Every time we incorporate a new metric, however, the customization stage has to be performed. The second requirement of routing engines was the ability to incorporate new metrics quickly. The reason that CRP satisfies this requirement is that while the metric-independent part of preprocessing can take minutes or hours, customization times of less than a second can be achieved [8].

CRP uses a relatively small part of the original graph: only the base level cells containing the source(s) and the target(s) are processed. For the rest of the query the overlay graph is used. The costs of the shortcuts in this overlay graph are computed during customization, in which the turn costs and restrictions are taken into account as well. So, CRP only has to add turn costs and restrictions for routing close to the source(s) and the target(s) during the query and therefore has a small penalty for incorporating them, satisfying the third requirement.

The customization phase of CRP is designed such that it is fast and it results in few metric-dependent data. Careful engineering causes metric-dependent space consumption to be much smaller than competing methods, such as Contraction Hierarchies [8]. One example of this engineering is to exploit the fact that many intersections in road networks share the same turn costs. By storing such turn tables only once and storing only a pointer to the corresponding intersections, space consumption is reduced significantly. In particular, the space consumption is small enough (around 70 megabytes per metric for a map of Europe [8]) so that it enables keeping data for multiple metrics in memory, which means CRP also meets the last requirement of routing engines.

2.2.5 Analysis

In this section, we aim to express the running time of the query stage of CRP in terms of graph properties. The running time of shortest path algorithms is commonly expressed in the number of nodes $|V|$ and edges $|E|$ in the graph. We aim to find a more refined expression to better capture the advantages of CRP with respect to Dijkstra's traditional algorithm, and to enable better comparisons with our own suggested approaches. To this end, we introduce the following symbols: V_C^1 for the maximum number of nodes in a base level cell, E_C^1 for the maximum number of edges in a base level cell, and still L for the total number of levels and K for the split size of the partitioning. Additionally, R and B denote the total number of shortcuts and boundary points respectively.

The one-to-one algorithm of CRP can be split into two parts: one (bidirectional) Dijkstra search on two base level cells and one Dijkstra search on the entire multi-level overlay graph. This leads to the following running time:

$$O\left(E_C^1 \log\left(V_C^1\right) + R \log\left(B\right)\right).$$

In Table 2 we show the number of nodes, edges, boundary points and shortcuts in the maps we use during our research, and in Table 3 the averages of these properties for cells on each level of the overlay graph. These tables give an indication of the impact of the speed-up technique of CRP.

The many-to-many version can be analysed similarly. For every target $t \in T$ we perform a unidirectional search with running time $O\left(E_C^1 \log\left(V_C^1\right) + R \log\left(B\right)\right)$. Subsequently, we do the same for every source, with the difference that during the search on the overlay graph, we perform $|T|$ heap operations, since we potentially update all targets in the backward search space. The complete running time therefore becomes:

$$|T| \cdot O\left(E_C^1 \log\left(V_C^1\right) + R \log\left(B\right)\right) + |S| \cdot O\left(E_C^1 \log\left(V_C^1\right) + R \cdot |T| \log\left(B\right)\right).$$

2.3 Distributed memory for CRP and Dijkstra

The customization stage can be parallelized in a relatively straightforward way, as described in the introductory paper of CRP [8]. Computing all-pairs shortest paths for each cell can be done in parallel, as only shortest path distances from each subcell are needed to process a cell. Delling et al. [10] showed that customization can even be done on GPUs. However, extending this approach to queries is not immediate. The query stage performs a variant of Dijkstra's algorithm and is therefore inherently sequential.

One way to make use of distributed memory in the query stage of CRP, would be to parallelize each individual query. Research into parallelizing Dijkstra's algorithm resulted in approaches like those of Crauser et al. [7] or the Δ-stepping algorithm of Meyer et al. [27], where in each iteration nodes are distributed among the processes for a label-correcting step. These types of approaches

need communication between a global process and all other processes at every step. Also, they do not exploit a partition of the graph like we have in CRP, but rather distribute the nodes depending on their tentative distances to the source. Therefore, such an approach does not seem to t our problem.

A di erent approach is presented by Tang et al. [36]. In this approach, the same type of label-correcting steps are performed, but it also exploits a partition of a graph, making it more applicable to our research. They compute a partition of the graph and give each cell of the partition to a di erent process. The processes containing source s and target t start by computing shortest path distances for all nodes in their cells, after which they send the distances to the nodes on the boundary of their cells to their adjacent cells. With this information, these cells can compute tentative distances for their nodes. Then, they do the same: the cells communicate the tentative distances of their boundary nodes to their adjacent cells. This is repeated until all nodes of the graph are settled. Note that with each communication step, we may also need to update distances to nodes v in an already processed cell $C(v)$, as a shortest path to v may go through cells that were untouched at the time $C(v)$ was processed. So, this algorithm uses a label-correcting approach instead of a label-setting one. In the paper, they apply this method of parallelizing to the sequential Dijkstra algorithm that uses two priority queues, which has complexity $O(|V|^2 \ |E|)$ [30]. When the algorithm uses k iterations and the graph is distributed roughly equally among the $|P|$ processes (each process contains $O(\frac{|V|}{|P|})$ nodes and $O(\frac{|E|}{|P|})$ edges), the total running time becomes [36]

$$k \quad O\left(\frac{|V|}{|P|}\right)^2 \quad \frac{|E|}{|P|} \quad k \quad T_{comm};$$

where T_{comm} denotes the cost of one communication step.

The thesis by Hamme [15] describes a completely di erent approach to using multiple processes to solve CRP queries. The thesis describes a method to perform CRP queries on mobile devices. As such devices have little internal memory, the method uses external memory to store the graph and only stores the query's search graph locally. For storing the graph in external memory, the natural structure provided by the partition is used: entire cell graphs are serialized and stored. Locality is improved by storing cells in memory based on the access patterns of CRP queries. In customization and during a query, most cells are visited completely. Only cells on the border of the search space may be visited only partially. Also, because CRP uses a nested partition, adjacent cells on the same level are likely to be required simultaneously. Therefore, cells are stored as a contiguous memory block and the blocks are ordered such that cells with a common supercell are stored consecutively. First, all level-1 cells are stored, ordered by common level-2 supercell, then all level-2 cells follow, ordered by common level-3 supercell and so on. A boundary arc is stored for all cells in the overlay graph where it is a boundary arc. This adds a small redundancy, but enables e cient access to the cells needed for a query.

When a query is performed on the mobile device, the search graph is built in internal memory by loading the needed cells from external memory. The base level cells of the source and target node completely determine which parts of the graph we need, which we will further discuss in Section 3.2. The query duration is the sum of the time it takes to load the needed graph data and the duration of the execution of CRP. It is therefore crucial to the query performance to use as little memory as possible for graph data. Hamme [15] was able to reduce the average data needed for each query on a continental-sized map to approximately 400 kilobytes by storing a skeleton graph per cell instead of a clique.

2.4 BSP Model

For analysing CRP and our suggested approaches, we use the bulk-synchronous parallel (BSP) model [37]. This is a bridging model for designing parallel algorithms. A BSP computer is a distributed-memory computer [4] that consists of a collection of p processors, each having access to their own local memory. The processors are connected to a communication network, such that they can communicate data. A BSP algorithm consists of a series of supersteps that each consist of a computation step and a communication step. In a computation step, many oating-point operations (ops) are executed, while during a communication step, data is transferred from one processor to another. The processors can simultaneously do calculations and send or receive data. Therefore, both the execution time of a computation step and of a communication step are dominated by the maximum among all processes, as the processors that nish sooner will have to wait, resulting in idle time. During BSP algorithms, synchronizations are performed after each calculation or communication step, ensuring that all communication has been resolved (contains operations as making sure all data has arrived, starting communication, etc.).

A BSP computer can be completely described by four parameters: the number of processors p, the computing rate (in ops per second) r, the communication cost per data word g and the global synchronization cost or latency l. The BSP cost of a BSP algorithm predicts the running time of the algorithm and can be expressed in the same four parameters. When, for each step i, we denote the number of ops performed during computation step i by processor j by w_i^j and the number of 8-byte data words sent or received by processor j by h_i^j, the total cost of a BSP algorithm consisting of k_{cal} calculation steps and k_{com} communication steps is given by

$$\sum_{i=0}^{k_{cal}} \max_{0 \le j < p} w_i^j + l + \sum_{i=0}^{k_{com}} \max_{0 \le j < p} h_i^j g + l. \quad (1)$$

We use the BSP model to analyse our algorithms. First, we express their running times in terms of w_i^j, h_i^j, g and l as above. Then, we express the di erent w_i^j and h_i^j using the big-O notation. During our experiments we try to get an indication of the impact of g and l on the total running time of the algorithms.

2.5 GREREC model

We will test our algorithms on two real maps. Recently, research has been conducted into the eld of random road networks. These random road networks provide the ability to control the characteristics of the network, while keeping typical characteristics of real road networks, e.g. they approximate planarity; undirected graphs representing road networks typically have very many nodes of degree 3, many of degree 4, few of degree 5 and very few of degree 6+; and they have heterogeneous edge travel costs [35].

The CRP algorithm exploits the natural obstacles in a map, such as mountains or large waters, to compute partitions of the graph with few boundary edges between cells. To test how our algorithms perform on arbitrary road networks, where such natural obstacles may not exist, we also test them on two random road networks.

In the past, many di erent models were used to approximate road networks. Grid models [2] are too regular to approximate road networks, their average degree, for example, is often too homogeneous. A planar variant of the Erdős{Renyi model was developed [13] and the Growing Random Planar Graph [25] which tries to simulate the e ect of urban sprawls on the road network. These models, however, generate more high degree nodes than real road networks typically contain. A di erent method is used in the square-grid fractal model [21], which uses a hierarchical method.

However, just as the grid models, the resulting graphs are too regular and it has the limitation that the node degree is limited to four.

The Grid model with Random Edges (GRE) [31] was developed with the main idea to randomly introduce the e ects of obstacles and shortcuts in a basic grid model. Using an optimisation algorithm on the six parameters of the model (the length and width of the area, the average lengths of vertical and horizontal lines in the network, the probability of the occurrence of obstacles and the probability of the occurrence of shortcuts), they achieved road networks that reasonably correlated with road networks of 66 main cities in Europe and the USA [35]. In this report, we will use the Grid network with Random Edges and Regional Edge Costs (GREREC) model [35], which is an improvement of the GRE model, to generate the two desired random road networks. GREREC adopts four modi cations compared to the GRE model: it allows the removal of edges at the rim of the network; it allows unconditional removal of vertical and horizontal edges; all types of diagonal shortcuts are allowed; and the edges are directly assigned a random travel cost depending on the position in the network, instead of a cost depending on their geometric length. With the rst three modi cations, the GREREC model relieves some of the restrictions on the network, therefore allowing the generation of a wider variety of network topologies. The rst modi cation does imply we might generate an unconnected graph, but this happens in real road networks as well. The fourth modi cation allows the introduction of road hierarchy e ects in the network. In the GREREC model, edge costs are randomly assigned, depending on their origin node. It makes the assumption that roads in a certain area of a network likely share characteristics such as length and speed limit and therefore assigns the same cost to each outgoing edge for a node. Assigning costs in this way implies that the GREREC model may origin from a grid topology, but it can actually model curved roads or other topology variations by incorporating them in the assigned costs.

The model takes as input length n and width m of the grid, as well as the probability p of keeping edges in the grid and probability q of generating shortcuts and uses the following procedure to generate a network:

1. Generate a grid with length n and width m.
2. Remove each existing edge with probability $1 \quad p$.
3. For each node v_{ij} in the grid for odd i and j, generate the four diagonal shortcuts departing from v_{ij} with probability q.
4. For each node v_{ij} in the grid for even i, generate the four diagonal shortcuts with probability q, unless the diagonal intersects an existing one.
5. Randomly assign a travel cost to the edges, where every edge departing from the same node must have the same cost of travel.

Note that the condition placed in step 4 ensures the planarity of the graph.

To test the similarity of networks created with the GREREC model to real road networks, Sohouenou et al. [35] generated 161 GREREC networks, categorised them into structural pattern groups based on the division of the values of p and q into three equally spaced intervals and computed the following topological indices for each of them. Index : the ratio of the number of cycles to the maximum possible number of cycles ($2 \quad |V| \quad 5$ in planar graphs); index : the ratio of the number of edges to the number of nodes and the index: the ratio of the number of edges to the maximum possible number of links ($3 \quad p|V| \quad 2q$ in planar graphs). The values for

Network group	p	q	α	β	γ	$\langle Degree \rangle$	h^{**}_{Degree}
B1	[0.33, 0.66]	[0, 0.33)	0.291(0.11)	1.51(0.26)	0.538(0.066)	3.02(0.53)	0.969(0.32)
B2	[0.33, 0.66]	[0.33, 0.66]	0.446(0.14)	1.82(0.31)	0.637(0.090)	3.64(0.62)	1.413(0.15)
B3	[0.33, 0.66]	(0.66, 1]	0.543(0.11)	2.01(0.24)	0.699(0.068)	4.02(0.48)	1.753(0.20)
C1	(0.66, 1]	[0, 0.33)	0.476(0.09)	1.88(0.21)	0.655(0.058)	3.77(0.41)	1.021(0.19)
C2	(0.66, 1]	[0.33, 0.66]	0.612(0.11)	2.14(0.26)	0.745(0.070)	4.26(0.51)	1.336(0.15)
C3	(0.66, 1]	(0.66, 1]	0.713(0.11)	2.32(0.29)	0.812(0.067)	4.64(0.58)	1.619(0.30)

mean(standard deviation); ** Degree heterogeneity

Table 1: The topological characteristics of GREREC networks for di erent values of p and q [35].

these indices for each structural pattern group are shown in Table 1. Subsequently, the mentioned network characteristics were compared for the GREREC networks.

This comparison resulted in the following conclusions: GREREC networks tend to have a larger range of networks with higher average degree and higher degree heterogeneity than real networks and the networks in categories B1, B2, C1 and C2 resemble real networks the closest. As we mentioned earlier, the most common node degree in real road networks is 3, followed by 4. Furthermore, degrees higher than 6 are very uncommon. GREREC networks in the categories B2 and B3 had similar degree distributions and were deemed most similar to real road networks by the paper. For our experiments, we used Table 1 together with the network characteristics of our real maps to generate the random road networks. More on this procedure will follow in Section 4.1.1.

3 Proposed algorithms

We will describe our two main approaches to using distributed memory to solve CRP queries, which we call Partitioning-Based Parallelism (PBP) and On-Demand Loading (ODL). For PBP, we discuss two di erent variants: PBP-1 and PBP-2.

3.1 Partitioning-Based Parallelism (PBP)

For the PBP approach, we use one global orchestrator process that orchestrates the query and multiple subprocesses (or just \processes"). Each process has its own part of the map and queries are solved by local computation steps on the subprocesses and communication steps. In preprocessing, we distribute the cells of the partition among the processes. We store each cell of the overlay graph and each base level cell on one of the subprocesses. The orchestrator process maintains the tentative distance $D(s; t_q)$ for each query and facilitates the communication between the subprocesses. It is not realistic to store only one cell on each process, e.g. the map of Western Europe we use in our experiments uses 5 levels. For split size K 10 (meaning 10 subcells per supercell), this would mean 111.110 cells and thus processes. So instead, we use a higher level of the overlay graph to distribute the cells among the di erent processes. In our project, we split the graph on the highest level, thus resulting in K processes. We call the level on which we split the graph the *split level* and denote it by L .

We have two variations of PBP in mind. The rst variation, PBP-1, still solves each individual query sequentially, but the computations are spread over multiple processes such that many queries can be handled more e ciently. PBP-2 parallelizes each individual query: the processes do parallel work and combine their results to compute a shortest path.

3.1.1 PBP-1

For the first PBP approach, besides their own cell of the partitioning, we store the levels $l \geq L$ of the overlay graph on each process. So in our case, we store the highest level of the overlay graph on every process. This duplication will provide a way to make the connection between the different processes and answer queries.

A bidirectional CRP query starts by exploring the base level cells containing s and t. This can be done on source process $p(s)$ and target process $p(t)$, where we denote $p(v)$ for the process containing node v. At first, we will assume $p(s) \neq p(t)$. We will treat the one-to-one and the many-to-many queries separately.

One-to-one queries A PBP-1 query begins with performing a backward search on the subgraph of $p(t)$. This process performs the search, only using its own cells of the partitioning without the common highest overlay graph level, until its priority queue is empty, after which the distances $d(v, t)$ for each boundary node v of its subgraph are known. These distances are sent in a dictionary X_b to source process $p(s)$. The source process then performs a forward CRP search. In the forward search, the common top level of the overlay graph is also used. While scanning nodes v, it checks if a backward distance $d(v, t)$ is present in the received X_b and updates $D(s, t)$ if applicable. Eventually, this search will find a shortest $s - t$ path, as the complete algorithm performs a normal CRP query, with the adjustments that the forward and backward search are performed sequentially and are split over two different processes, which does not affect the found path.

In the case that $p(s) = p(t)$, we can answer a query using one process only. To achieve this, we must make one adjustment to the preprocessing stage. In customization, the shortcut costs are determined by only considering paths within the cell. However, it is possible that a shortest path between two boundary points uses edges of an adjacent cell. In a normal CRP query, in which the whole map is available, one would simply traverse to that adjacent cell and therefore still find a shortest path. When partitioning the map like we do for PBP, it could happen that that adjacent cell is not in the subgraph of the current process, which would mean we may not find a shortest path. Therefore, we adjust the customization process to also take into account paths using edges outside the cell of the two boundary points, which implies that every shortcut cost is a shortest path. Then, when $p(s) = p(t) = p$, we can answer a query by performing the regular CRP algorithm on the subgraph of p.

Many-to-many queries We extend the PBP-1 approach for one-to-one queries to many-to-many queries as follows. Instead of one target process for a one-to-one, we may have multiple target processes that each may contain multiple targets. Let us say that each target process p contains targets $T_p \subseteq T$. The query starts with each of these processes p performing a backward search for each of their targets $t \in T_p$. A target process p fills $X_b(p)$ with entries $d(v, t)$ for their boundary points v and their targets $t \in T_p$. When the priority queues are empty for each target, the process sends $X_b(p)$ to each source process (a process containing at least one source). That way, each source process receives the complete backward search space $X_b = \bigcup_p X_b(p)$. Then, each source process p performs a forward search for each of its sources $S_p \subseteq S$, also using the common highest level of the overlay graph. Source process p finds shortest $s - t$ paths for all $s \in S_p$ and $t \in T$ by checking if $v \in X_b$ when scanning node v. The entire cost array can then be formed by collecting the different parts from each source process.

Algorithms 5 and 6 (Appendix A) show the one-to-one and many-to-many versions of PBP-1. The algorithms are constructed such that all communication goes through the orchestrator process. This process does not perform any calculations, but only "orchestrates" the calculations: it

sends the messages to the right processes and collects and outputs the result. We will discuss the communication in more detail in Section 4.

Analysis We use the BSP model as described in Section 2.4 to analyse the PBP-1 algorithms. Both the one-to-one and the many-to-many algorithm consist of two computation steps and one communication step in between. For the one-to-one algorithm, in the first computation step, the target process p_t is the only active process, while during the second computation step, the source process is the only active process. The BSP cost of the algorithm therefore is, following (1):

$$w_1^{p_t} \quad w_2^{p_s} \quad gh^{1\ 1} \quad 3l: \tag{2}$$

Cost $w_1^{p_t}$ represents the cost of performing a unidirectional CRP search on the subgraph of p_t, $w_2^{p_s}$ represents the cost of a unidirectional CRP search on the subgraph of p_s, including the highest level of the overlay and checking X_B for a possible improvement of $D_s; t$. The number of data words $h^{1\ 1}$ that is sent during the only communication step is equal to the size of the backward search space of a one-to-one query in 8-byte words. In our experiments we will investigate the size of this backward search space empirically. We show a diagram of the BSP cost in Figure 5.

The many-to-many algorithm also consists of two computation steps and one communication step. The message during the communication step has size $h^{1\ 1}$ $|T_p|$, as each target process performs the same search as for a one-to-one query for each of its targets. The BSP cost becomes as follows:

$$\max_{p \in P_T} w_1^p \ |T_p| \quad \max_{p \in P_S} w_2^p \ |S_p| \quad \max_{p \in P_T} gh^{1\ 1} \ |T_p| \quad 3l; \tag{3}$$

where P_S and P_T denote the collections of source processes and target processes respectively. We show a BSP diagram of the algorithm in Figure 6.

To further investigate the theoretical running time of both algorithms, we express w_1^p, w_2^p and $h^{1\ 1}$ in big-O notation. To this end, we let R_C^l denote the maximum number of shortcuts in a level-l cell and B_C^l the maximum number of boundary points in a level-l cell. For both these values we include shortcuts or boundary points of subcells. When we intend to exclude these, we use symbols R_C^l and B_C^l respectively. We show the running times of both the one-to-one and the many-to-many algorithm in Theorem 1.

Theorem 1. The one-to-one and many-to-many algorithms of PBP-1 take $O(E_C^1 \log(V_C^1) \quad R \log(B)) \quad g \quad O(B_C^{L\ 1}) \quad 3l$ and $|T| \quad O(E_C^1 \log(V_C^1) \quad R_C^{L\ 1} \log(B_C^{L\ 1}))$ $|S| \quad O(E_C^1 \log(V_C^1) \quad R \quad |T| \log(B)) \quad g \quad O(|T| \quad B_C^{L\ 1}) \quad 3l$ time, respectively.

Proof. The running time of a unidirectional CRP search using all levels but the top level, belongs to $O(E_C^1 \log(V_C^1) \quad R_C^{L\ 1} \log(B_C^{L\ 1}))$, as we only use shortcuts inside one of the top level cells. When we include the highest level and the check of X_B for a new shortest distance, we perform a search with the same running time (in big-O notation) as the regular CRP algorithm, which is $O(E_C^1 \log(V_C^1) \quad R \log(B))$ as we showed in Section 2.2.5. The latter expression dominates the first. Message size $h^{1\ 1}$ is the size of backward search space X_B in 8-byte data words and thus is of size $O(B_C^{L\ 1})$. The running time of the one-to-one version of PBP-1 therefore can be expressed using a combination of big-O notation and parameters g and l from the BSP model as (following (2)):

$O(E_C^1 \log(V_C^1) \quad R_C^{L\ 1} \log(B_C^{L\ 1}))) \quad O(E_C^1 \log(V_C^1)) \quad R \log(B)) \quad g \quad O(B_C^{L\ 1}) \quad 3l$
$O(E_C^1 \log(V_C^1)) \quad R \log(B)) \quad g \quad O(B_C^{L\ 1}) \quad 3l.$

Similarly, the many-to-many algorithm has running time (following (3))

$$|T| \quad O(E_C^1 \log(V_C^1) \quad R_C^{L\ 1} \log(B_C^{L\ 1}))$$
$$|S| \quad O(E_C^1 \log(V_C^1) \quad R \quad |T| \log(B)) \quad g \quad O(|T| \quad B_C^{L\ 1}) \quad 3l.$$

□

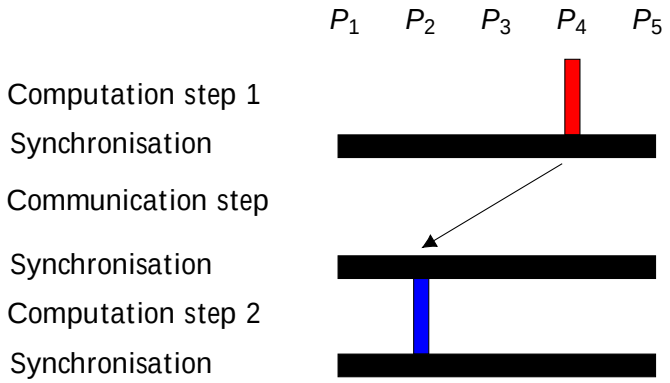

Figure 5: BSP diagram of a one-to-one query of PBP-1 for which $p(s) = p_2$ and $p(t) = p_4$. In computation step 1 (red), a backward search on a process's subgraph is performed. In computation step 2 (blue), a forward search is performed on the union of a process's subgraph and the highest level of the overlay graph. In the communication step, the backward search space of target process P_4 is sent to source process P_2.

3.1.2 PBP-2

Extending the approach to parallelizing Dijkstra of Tang et al. [36] to CRP leads us to PBP-2. We do not store the highest level of the overlay graph on each process, but only give each process one of the K top level cells and its subcells.

One-to-one queries Making the same adjustments to the preprocessing stage as we mentioned for PBP-1, we can again answer a $s \quad t$ query for which $p(s) \quad p(t)$ without communicating to other processes. We therefore assume $p(s) \quad p(t)$ in this description. The algorithm starts by exploring the subgraphs on $p(s)$ and $p(t)$, after which we obtain tentative (forward and backward, respectively) distances for the boundary points of $C^{L\ 1}(s)$ and $C^{L\ 1}(t)$. These boundary points lie on the border of a cell contained in a different process. Processes $p(s)$ and $p(t)$ send the found tentative distances to these adjacent cells, after which their processes add them to their forward or backward priority queue. Then, each process performs multiple label-correcting steps: they perform a bidirectional CRP search on their own subgraph, using these priority queues. When both priority queues are empty, they send the found tentative distances of the boundary points to the corresponding adjacent cells. Every process that receives such distances, we call an \active

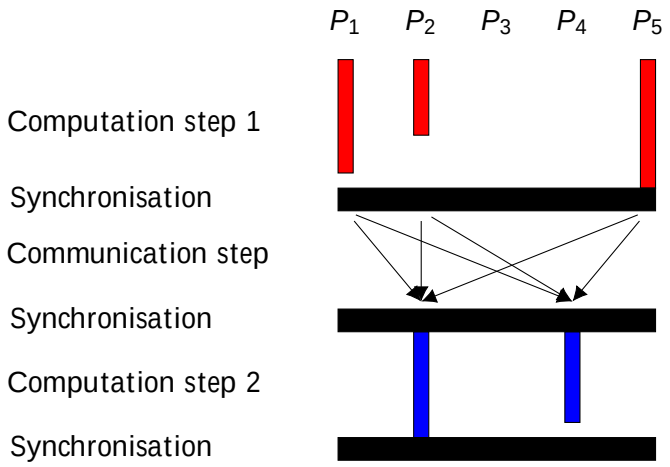

Figure 6: BSP diagram of a many-to-many query of PBP-1 for which processes P_2 and P_4 contain the sources and P_1; P_2 and P_5 contain the targets. In computation step 1 (red), a backward search on a process's subgraph is performed. In computation step 2 (blue), a forward search on the union of a process's subgraph and the highest level of the overlay graph. In the communication step, the backward search spaces of the target processes are sent to the source processes.

process" and updates the distances of their boundary points (if they are an improvement of the current values). It also checks if it contains a node with both a forward and backward distance, in which case it updates its local tentative distance $D(s, t)$, and starts a new label-correcting step. Just as in the parallel Dijkstra algorithm of Tang et al. [36], a shortest distance $d(s, t)$ is found when there passes an iteration without communication. This distance is retrieved by taking the minimum of the local tentative distances of all processes.

Many-to-many queries For the many-to-many algorithm of PBP-2, we start by performing a PBP-1-like search from every target $t \in T$. This means every target process performs a backward search on its own graph, obtaining distances $d(v, t)$ for all local boundary points v, which are stored in dictionary X_B. This dictionary is stored on the process itself and not communicated to other processes. Then we continue by performing a forward search from every source on each source process, resulting in forward distances for all boundary points of the source process's subgraph. After these searches, we start with our label-correcting steps: each process with updated forward distances $D(s, v)$ for some $s \in S$ and local boundary points v (in the first iteration every source process) sends these distances to the processes containing adjacent cells. Every process maintains $|S|$ priority queues, one for each source. When a process receives a forward distance $D(s, v)$, it checks whether it is an improvement of its known tentative distance. If so, it adds it to the priority queue corresponding to s. After processing all received distances, the process performs forward CRP searches with the nodes in the priority queues. Afterwards, it sends its updated tentative distances to adjacent processes and a new label-correcting iteration is started. Each target process p maintains a tentative distance matrix of size $|S| \times |T_p|$, containing distances $D(s, t)$ for $s \in S$ and $t \in T_p$. While scanning v during a forward search, it checks whether it can improve such distance by looking into X_B to see if we already determined a backward distance $D(v, t)$ for v and some $t \in T_p$. When there is no communication left between the processes, each target process contains shortest paths from each source to their local targets. The complete distance matrix can be retrieved by combining these distances for all target processes. In Algorithms 7 and 8 (Appendix A) we present

both the one-to-one and the many-to-many version of PBP-2 in pseudocode.

Analysis When we view the one-to-one PBP-2 algorithm in light of the BSP model, we distinguish two different types computation steps: the first step, in which the initial searches from the source(s) and target(s) are performed and the subsequent label-correcting steps. The total BSP cost for the one-to-one algorithm is given by

$$\max\{w_1^{p[s]}, w_1^{p[t]}\} \cdot g \cdot \max\{h_1^{p[s]}, h_1^{p[t]}\} \cdot 2l + \sum_{i=2}^{k} \left(\max_{p \in P} w_i^p \cdot g \cdot \max_{p \in P} h_i^p \cdot 2l \right), \quad (4)$$

when the algorithm uses k iterations and where we denote w_1^p for the cost of the initial unidirectional forward/backward search from the source/target on process p's subgraph (same as in the analysis of PBP-1), w_i^j for $i \in \{2, \ldots, k\}$ for the cost of updating the distances of the boundary points of process p and performing a bidirectional CRP search on the local subgraph and h_i denotes the maximum sent (boundary point, distance)-pairs (in data words) between processes during communication step i. We show a BSP diagram of the one-to-one algorithm in Figure 7. Note that h_i is expected to be smaller than in PBP-1, since we only send distances for boundary points at the outer boundary of a process's subgraph instead of the entire backward search space. This difference will be studied more extensively in our experiments.

In the many-to-many algorithm, we do not have a communication step after the initial backward searches, leading to the following BSP cost, when using k iterations:

$$\max_{p \in P \cap T} w_1^p \cdot l + \sum_{i=2}^{k} \left(\max_{p \in P} w_i^p \cdot g \cdot \max_{p \in P} h_i^p \cdot 2l \right), \quad (5)$$

where w_i^p for $i \in \{2, \ldots, k\}$ represents the cost of updating boundary point distances and performing a forward label-correcting search, including checking X_B while scanning a node, h_i^p represents the set of updated distances sent by process p and P denotes the collection of all processes. The corresponding BSP diagram is presented in Figure 8.

Again, we want to go a bit further and express w_i^p and h_i^p in big-O notation. This leads to the running times for both PBP-2 algorithms (one-to-one and many-to-many) presented in Theorem 2.

Theorem 2. The one-to-one and many-to-many algorithms of PBP-2 take $O((E_C^1 \log(V_C^1)) + k(R_C^{L-1} \log(B_C^{L-1})) + k \cdot g \cdot O(B_C^{L-1}) + 2l)$ and $k \cdot (O(R_C^{L-1}) \cdot |T| \log(B_C^{L-1})) + g \cdot O(|S| \cdot B_C^{L-1}) + (2k-1) \cdot l$ time, respectively.

Proof. As we showed in the proof of PBP-1's running time, a unidirectional CRP search within one top level cell is $O(E_C^1 \log(V_C^1) + R_C^{L-1} \log(B_C^{L-1}))$. For $i \in \{2, \ldots, k\}$, for both the one-to-one and the many-to-many algorithm, we perform label-correcting steps, which are CRP searches on only the boundary points within one top level cell. So the running time of one such step is $O(R_C^{L-1} \log(B_C^{L-1}))$ for the one-to-one and $O(R_C^{L-1} \cdot |T| \log(B_C^{L-1}))$ for the many-to-many algorithm. Message sizes h_i^p are also of the same order for all $i \in \{1, \ldots, k\}$. Each message can only contain boundary points on the border of a top level cell, so $h_i^p \in O(B_C^{L-1})$ for the one-to-one algorithm and $h_i^p \in O(|S| \cdot B_C^{L-1})$ for the many-to-many version. This results in the following running time for the one-to-one PBP-2 algorithm (following (4)):

$$
\begin{aligned}
& O(E_C^1 \log(V_C^1)) \cdot R_C^{L-1} \log(B_C^{L-1}) \cdot g \cdot O(B_C^{L-1}) \cdot 2l \\
k \cdot & (O(R_C^{L-1} \log(B_C^{L-1})) \cdot g \cdot O(B_C^{L-1}) \cdot 2l) \\
& O(E_C^1 \log(V_C^1)) \cdot (k-1) \cdot O(R_C^{L-1} \log(B_C^{L-1})) \cdot (k-1) \cdot g \cdot O(B_C^{L-1}) \cdot 2l.
\end{aligned}
$$

The running time of the many-to-many algorithm of PBP-2 can be expressed as (following (5))

$$
\begin{aligned}
& O(E_C^1 \log(V_C^1)) \cdot R_C^{L-1} \log(B_C^{L-1}) \cdot l \\
k \cdot & (O(R_C^{L-1} \log(B_C^{L-1})) \cdot |T| \cdot g \cdot O(|S| \cdot B_C^{L-1}) \cdot 2l) \\
& O(E_C^1 \log(V_C^1)) \cdot R_C^{L-1} \log(B_C^{L-1}) \\
k \cdot & (O(R_C^{L-1} \log(B_C^{L-1})) \cdot |T| \cdot g \cdot O(|S| \cdot B_C^{L-1}) \cdot (2k-1) \cdot l.
\end{aligned}
$$

□

Note that the running time of the one-to-one algorithm of PBP-1 is the same running time as in the analysis of Tang et al. [36] we mentioned in Section 2.3 when we swap their Dijkstra algorithm's running time for CRP's running time, since $\frac{B}{|P|} \cdot O(B_C^{L-1})$ and $\frac{R}{|P|} \cdot O(R_C^{L-1})$. In our experiments, besides looking at the impact of l and g on the algorithm's performance, we are interested in the number of iterations k we need to obtain a shortest path. We also take a closer look at how many processes send or receive a message during one iteration.

3.2 On-Demand Loading

In the ODL approach, we use two processes in total: one maintains a database that contains the entire graph in memory and one receives and answers queries by obtaining the necessary graph data from the other process and performing a CRP search. We will call the process storing the graph simply "the database" and the process answering queries the orchestrator process. Our ODL approach follows the same idea as the thesis [15] we treated in the Section 2.3. In the thesis, the search graph is built in memory during the query: every time the query wants to scan a node that is not in memory yet, it fetches the corresponding cell from the database. In our algorithm, we use a different approach: when the query arrives we obtain (possibly a superset of) the entire search graph for the query. We use the fact that a one-to-one query traverses at most $2 \cdot K - 2$ cells per level before switching to a different level. We prove this result in Proposition 3. Another adjustment we make with respect to [15] is that we use a bidirectional implementation of CRP, instead of the unidirectional variant used in the thesis.

Proposition 3. The search graph for a CRP one-to-one query contains nodes of at most $2 \cdot K - 2$ cells per level $l > 0$. In particular, it only contains nodes from cells in $\{C^l \mid C^l = C^l(s) \vee C^l \in C^{l+1}(s)\}$ and $\{C^l \mid C^l = C^l(t) \vee C^l \in C^{l+1}(t)\}$.

Proof. Let a node v be scanned on level $l_{st}(v) = l$. Then, $C^l(v) \in C^{l+1}(s)$ or $C^l(v) \in C^{l+1}(t)$ by definition of $l_{st}(v)$. Furthermore, $C^l(v) = C^l(s)$ and $C^l(v) = C^l(t)$. So, $C^l(v) \in \{C^l \mid C^l \in C^{l+1}(s) \vee C^l \in C^{l+1}(t)\} \cup \{C^l \mid C^l = C^l(s) \vee C^l = C^l(t)\}$. Each cell contains at most K subcells, so the last mentioned set has cardinality at most $2 \cdot K - 2$. □

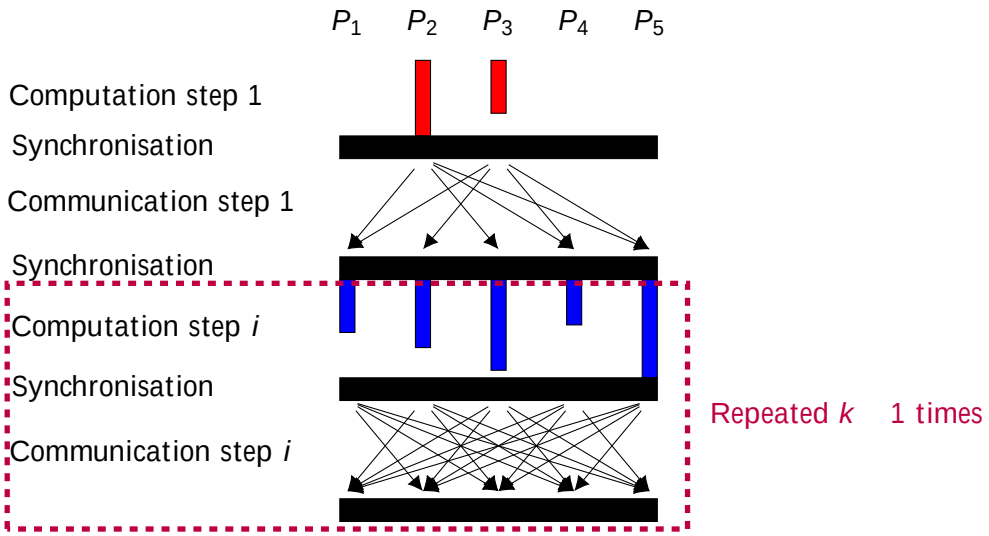

Figure 7: BSP diagram of a one-to-one query for PBP-2 with k 1 label-correcting steps for which p_2 and p_3 contain the source and target. In computation step 1 (red), a backward/forward search is performed on the process's subgraph. In computation step i P t2;:::;ku (blue), the distances of received boundary points are updated and a forward search is performed on the process's subgraph. In all communication steps, the found distances of boundary points that are shared with adjacent processes are sent to those cells.

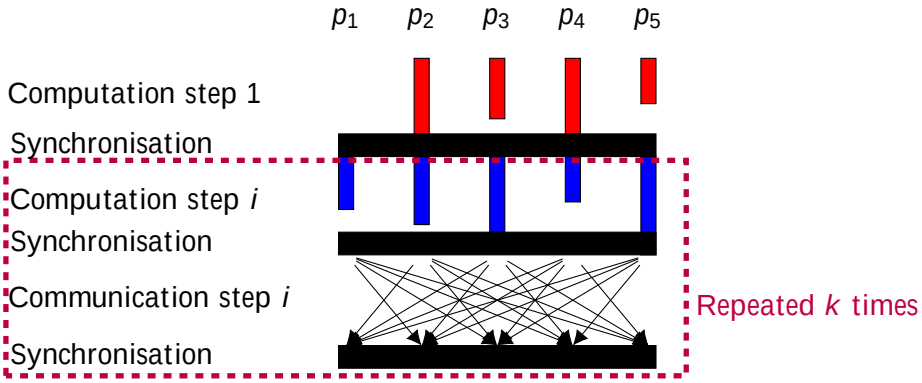

Figure 8: BSP diagram of a many-to-many query for PBP-2 with k 1 label-correcting steps for which $p_2;p_3;p_4$ and p_5 contain the targets. In computation step 1 (red) a backward search is performed on the process's subgraph. In computation step i P t2;:::;ku (blue) the distances of received boundary points are updated and forward searches are performed on the process's subgraph. In all communication steps the found distances of boundary points that are shared with adjacent processes are sent to those cells. Note that not all processes have to be active during an iteration: only the processes that found updates for boundary points on a cell border they share with another process send a message, and only processes of which a neighboring process found any updates of a shared cell border perform a forward search.

Proposition 3 implies that we know exactly which cells of the overlay graph we may need during our query. When a query arrives, we can load these cells and the two base level cells that contain s and t from the database and perform the regular CRP algorithm to answer the query. Since every node v is scanned on query level $l_{st}(v)$ of the overlay graph, we do not need cells on levels l for which $C^l(s) \neq C^l(t)$. Indeed, if s and t are in the same cell, there must exist a shortest path without a level-l shortcut (by the same argument we used to prove Proposition 1). We call the lowest level l for which two nodes v and w share the same cell, i.e. $C^l(v) = C^l(w)$, the *lowest sharing level* $l_0(v, w)$ of two nodes. For an $s \to t$ query, we therefore load the $2 \times K \times 2$ cells as described in Proposition 3 for all levels up to $l_0(s, t)$.

When answering a many-to-many query, we build the search graph by loading the $K \times 1$ cells per source and target as described in the proposition, which results in a search graph of at most $(|S| + |T|)(K \times 1)$ cells per level. Additionally, we load the base level cells for all $s \in S$ and all $t \in T$. Again, we may not need cells of all overlay levels. For each source $s \in S$, we only need cells up to the maximum of all lowest sharing levels with targets $t \in T$: $\max_{t \in T} l_0(s, t)$. For each target t, we only need cells up to level $\max_{s \in S} l_0(s, t)$.

Analysis The analysis of ODL is more straightforward than those of both PBP algorithms. The algorithm consists of two steps: one communication step in which the process with the database in memory sends the necessary graph data to the orchestrator process and one calculation step in which the CRP algorithm is performed on the orchestrator process. The BSP cost for the both the one-to-one and the many-to-many version of ODL can be expressed as

$$w \times gh \times 2l; \tag{6}$$

where w denotes the cost of a CRP search on the loaded search graph and h is the size of the search graph in data words. Only two synchronizations are needed, instead of the three for PBP-1, because we have one calculation step for ODL and two calculation steps for PBP-1. We show the corresponding BSP diagram in Figure 9. In Theorem 3, we show the running times for the one-to-one and many-to-many algorithm of ODL.

Theorem 3. The one-to-one and many-to-many algorithms of ODL take $O(E_C^1 \log(V_C^1)) \times R \log(B) \times g \times O(B) \times R \times E_C^1 \times V_C^1) \times 2l$ and $(|T| \times O(E_C^1 \log(V_C^1)) \times R \log(B)) \times (|S| \times O(E_C^1 \log(V_C^1)) \times R \times |T| \log(B)) \times g \times O(B) \times R \times (|S| \times |T|) \times (E_C^1 \times V_C^1)) \times 2l$ time, respectively.

Proof. For the one-to-one algorithm, the part of the overlay graph we need to load is of size $O(B \times R)$, because when $l_0(s, t) \geq L$, we load $2 \times K \times 2$ cells for each level, including the highest level of the overlay graph. The base level cells both have size $O(E_C^1 \times V_C^1)$. Together they form the message size h that is sent from the database to the orchestrator process. The computational cost w is equal to the cost of the regular CRP algorithm. After the communication step, the orchestrator process has all information it needs to compute a shortest path. This results in the following running time of ODL's one-to-one algorithm (following (6)):

$$O(E_C^1 \log(V_C^1)) \times R \log(B)) \times g \times O(B \times R \times E_C^1 \times V_C^1) \times 2l.$$

The search graph for the many-to-many algorithm also consists of an $O(B \times R)$-sized part of the overlay graph. Additionally, we need $|S| \times |T|$ base level cells, so the total search graph is of size $O(B \times R \times (|S| \times |T|)(E_C^1 \times V_C^1))$. Cost w is again the same as for regular CRP, leading to a total cost of (again, following (6))

$$|T|\,O(E_C^1)\,\log(V_C^1)\;R\;\log(B)\;|S|\,O(E_C^1)\,\log(V_C^1)\;R\;|T|\log(B)$$
$$g\,O(B)\;R\;(|S|\;|T|)\,(E_C^1\;V_C^1)\;2l.$$

□

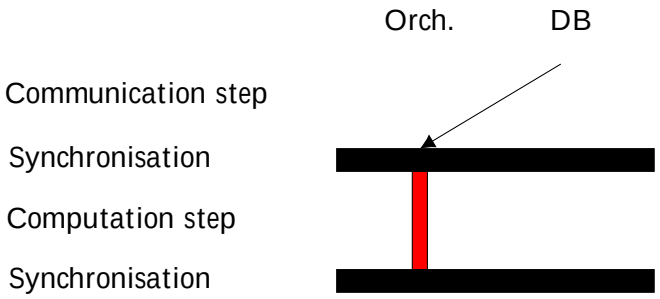

Figure 9: BSP diagram of a one-to-one or many-to-many query for ODL. In the communication step, the database process (DB) sends the search graph to the orchestrator process (Orch.). In the computation step, the orchestrator process performs a CRP search and calculates (a) shortest path(s).

3.3 Discussion of the algorithms

In this section, we discuss and compare the suggested algorithms and their analyses. We will look at the criteria we mentioned in our introduction: performance in terms of running time and memory usage, which includes the communication steps; scalability; the extension to many-to-many queries; the distribution of work among the processes; and the impact of the query distance and of the natural cuts in the network on the performance. We also mentioned the correctness of the algorithm as a point of emphasis in case we would come up with an algorithm that a ects the correctness of CRP. The three algorithms we suggest, however, obtain the same path as CRP, so the correctness is not an issue.

Performance All three algorithms add extra steps to the CRP algorithm while aiming to improve scalability. The PBP-1 algorithm performs CRP spread across two di erent processes and adds one communication step in between. The running time of PBP-1 is therefore of the same order as CRP's running time, with the added cost of size $g\;O(B_C^L\;^1)\;3l$: the cost of the communication step plus three synchronisations. Also important to note is that PBP-1 does not need any communication when the query starts and ends within one process's subgraph. Especially in realistic scenarios, in which most queries are local, this is an important bene t. ODL follows a similar pattern: it performs the same computing steps as CRP, but adds an extra communication step. The added running time for ODL on top of CRP's running time is therefore of size $g\;O(B\;R\;E_C^1\;V_C^1)\;2l$. The running time of PBP-2 relative to CRP is more ambiguous, as it not only depends on network characteristics g and l, but also on the number of iterations k and the number of extra computations it performs due to its label-correcting approach. The added cost due to communication has size $O(k\;g\;B_C^L\;^1\;l)$. Note that this means that the run time of PBP-2 relies a lot more (a factor $O(k)$ on g and l than the run times of both PBP-1 and ODL, since the latter two have only one communication step. However, the message size of PBP-2 is the smallest: $O(B_C^L\;^1)$ versus

$O(B_C^L)$ for PBP-1 and $O(B \cdot R \cdot E_C^1 \cdot V_C^1)$ for ODL. In our experiments, we will examine k and the message sizes for all three algorithms empirically, which will give an indication of the tradeo between the multiple small messages of PBP-2 and the one large message for PBP-1 and ODL.

The memory usage for ODL is expected to be much lower than for the two PBP approaches, since it does not have any graph data in memory continuously. PBP-1 and PBP-2 have a similar amount of graph data in memory: approximately a fraction $\frac{1}{|P|} \cdot \frac{1}{K}$ of the graph. Another factor in the amount of memory that is used by a process is the message sizes during communication. During a query, these messages have to be created, sent, received and processed, which means they are present in the internal memory. In addition, the subprocesses of PBP-2 have to keep some state for every query it is working on during the label-correcting iterations. PBP-1 and ODL have the advantage of only having to send or receive a message once, do corresponding calculations and be able to delete all query-related data from memory. We expect therefore that the subprocesses of PBP-2 will have the highest memory consumption of the three algorithms with PBP-1 not too far behind. ODL will use very little memory compared to the other two. The regular CRP algorithm keeps the entire graph in memory, but does not have the added memory usage due to communication.

Scalability The orchestrator process of ODL does not have to preload any data, which means starting a new instance will be fast. This implies that during peaks in work load, when we may need extra instances to handle all incoming queries, the algorithm can e ciently scale up and during times with few queries it can scale down quickly. We do not scale the database process, i.e. we always have one instance of this process. If we would scale the database, we would have to load the entire graph for each new instance, just as we would do for regular CRP. Therefore, we would lose the scalability advantages of ODL compared to regular CRP. Both the PBP algorithms, as well as the regular CRP algorithm, have to load graph data before they can perform calculations. The startup times for instances of the PBP approaches will be similar, since the only di erence is the top level of the overlay graph that is included for every subprocess of PBP-1 and excluded for PBP-2. Most of these cells of continental-sized graphs can be stored in less than one megabyte, while all graph data together is a couple of gigabytes (see Section 4.1) hence this will not increase the startup time of PBP-1 signi cantly compared to PBP-2. In our experiments we will look at the impact on the starting time of only having to load fraction $\frac{1}{|P|}$ during PBP instead of the entire graph during the regular CRP algorithm.

Many-to-many queries During the execution of the many-to-many algorithm of PBP-1, there is communication between all target processes and all source processes. So, when many of the sources and targets are contained in subgraphs of di erent processes, the number of messages increases. In realistic applications, where most sources and targets are relatively close to each other, we do not have many di erent target and source processes and hence not a big increase in communication frequency. The message size increases with a factor $O(T)$: each target process p sends a message of size $|T_p| \cdot O(B_C^L) = O(|T| \cdot B_C^L)$. For PBP-2, we see the same type of increase in message size: from $O(B_C^L)$ per process p for a one-to-one query to $|S_p| \cdot O(B_C^L) = O(|S| \cdot B_C^L)$ for a many-to-many query. The number of messages will increase as well, as the number of iterations is equal to the maximum among the needed iterations for one-to-one queries between each (s, t)-pair. How much this increase is, will be investigated in our experiments. The message size for ODL increases by a factor $O(|S| \cdot |T|)$ to $O(B \cdot R \cdot |S| \cdot |T| \cdot E_C^1 \cdot V_C^1)$. Other than that, it performs the same steps as the regular CRP algorithm.

Distribution of work The distribution of the work load in the PBP approaches strongly depends on the partition of the graph, especially for PBP-1. If the sources and targets of the queries are randomly chosen from the entire node set, the number of requests per subprocess is proportional to the number of nodes per subgraph. So, if the number of nodes is roughly equal per subgraph, the workload per PBP-1 subprocess will be roughly equal as well. However, this does not happen often in realistic scenarios, as some areas of the graph will more commonly contain sources and/or targets than other areas. In these cases, the load of the processes containing these areas of the graph will increase. This is also an advantage in terms of scalability: we can increase the number of instances of particular processes, instead of having to start another instance with the entire graph in memory as we would have to do with regular CRP. The distribution of work for PBP-2 is spread more evenly among the processes, since a process does not have to contain a source or target to be active during a query. For realistic scenarios, where some subprocesses contain more sources or targets than others, the work load will increase for these processes, but for their neighbour processes as well. This is because the processes containing more sources or targets will also reach the common boundary with their neighbouring processes more often, causing them to send messages to these neighbours. For ODL, the work is not distributed among di erent processes: the orchestrator performs all calculations, while the database stores all graph data.

Impact of query distance For PBP-1, a lot of the performance depends on whether it needs its communication step. When the source and target are contained in the subgraph of one process, the run time will be signi cantly faster. Therefore, for those cases, query times will be similar to those of the regular CRP algorithm, with the addition that we expect a jump in query duration for queries between nodes on di erent processes. We expect the same jump for PBP-2, as we only have to communicate once the source and target are not on the same process. For ODL, we expect a more gradual increase in run time as the query distance increases. We only load cells up to level $l_0(p_s, t_q)$, so when the source and target are in the same cell on low levels, i.e. the query distance is small, we load fewer cells. The larger the query distance, the more cells we need to build our search graphs. Hence, loading these cells from the database will take more time.

Impact of natural cuts The partitioning algorithm used for CRP exploits the natural cuts in road networks to minimize the number of boundary edges between cells. The number of boundary edges will therefore increase, and thus performance will decrease, when these natural cuts do not exist or exist to a lesser extent. We expect the largest impact for the PBP-2 algorithm, as more boundary edges implies paths will be more likely to traverse cell boundaries more often. This means PBP-2 will need more iterations on average. Less convenient partitions will also increase the total amount of graph data, as we have more boundary points and thus shortcuts per cell. Therefore, the ODL algorithm will have to load more data on average, which will decrease performance. The same goes for PBP-1: the one message it has to send, will be larger. We will examine the impact of the partition's quality by testing our algorithms on two maps generated by following the GREREC model we treated in Section 2.5, as well as two real road networks.

4 Implementations

4.1 Maps

In our experiments we use two real maps provided by Ortec: one of The Netherlands, Belgium, Germany and France (which we call the \Europe map" for convenience sake) and one of the 29 most

Eastern states of the United States (the \USA map" from now on). These maps are modelled as directed graphs, but in both maps approximately 95% of all edges has a counterpart in the opposite direction. We use the aforementioned KaHIP algorithm to compute partitions consisting of 5 levels of both graphs with split size K 10. Additionally, the algorithm takes a oat P r0; 1s that controls the allowed *imbalance* in the partitioning, meaning that for all processes p, the constraint

$$|V_p| \leq \frac{(1 + \varepsilon) \cdot |V|}{K}$$

is satis ed. A higher allowed imbalance enables the algorithm to try more di erent cuts, generally leading to fewer boundary edges, at the cost of having larger di erences in cell sizes. For the partitions of the maps in our experiments, we use 0:4. These values for K and were chosen by Ortec after trying out a few di erent setups. For each setup, the performance of CRP queries was determined. The setup was chosen for which the best performance was achieved.

We associate a cell of the overlay graph with a list of boundary points and a cost array-index. The shortcuts in each cell form a clique. For the entire graph, we store an array with the costs of all preprocessed shortcuts, where the costs of all shortcuts of a cell are stored contiguously. When scanning a boundary point of a cell during our search, we use the index to access the part of the cost array we need. As edge costs, we use a linear function of the two static road properties edge length in meters and the needed time in seconds to traverse them (each edge has an associated travel speed): 1.5 times the travel time plus the edge length. The reason for using a combination of both travel time and travel distance is that in applications it is often not preferable to have slightly shorter travel times if that means we have to take a large detour. For example, if driving an extra 20 km would result in a gain in total travel time of a few seconds, we prefer the route with shorter distance and longer travel time. The ratio 1.5:1 is an arbitrary choice made by Ortec, which works well for their applications. We use 44 bytes to represent a boundary point and three integers, i.e. 12 bytes, to represent a cost. These sizes are important to determine our message sizes for the PBP approaches, since they consist of (boundary point, cost)-pairs.

Important for the ODL algorithm is the notion of *partition codes*. We associate a partition code with each node to be able to determine to which cell it belongs on each level of the overlay graph. A partition code consists of four bits per level. To determine in which level l-cell a node is situated, we look at the $4 \cdot (L - l)$ most signi cant bits of its partition code. So the base level cell of node is determined by looking at the entire partition code, while the top level-cell (on level $L - 1$) is encoded by just the rst four bits.

At the end of Section 4.1, we present some properties of the maps and their partitions in Tables 2 and 3. Furthermore, we show the sizes in bytes of the maps we use during our experiments in Table 4, including the subgraph sizes for each PBP process. Lastly, we show the average sizes in bytes of the graph cells in Table 5. This will determine the amount of data we have to load while performing an ODL query.

4.1.1 GREREC

In this section, we explain how we generated the two random road networks we use in our experiments. We follow the GREREC model [35] we treated in Section 2.5. We chose to generate maps of size and with values for indices ; and that are similar to our real maps of Europe and the USA, so that we better isolate the e ects of the (absence of) the natural cuts in the graph on the performance of the algorithms. Our rst arti cial map, which we call \Grerec-1" aims to have similar characteristics as the Europe map, while \Grerec-2" is modelled after the map of the USA.

To generate a grid of similar dimensions to those of our real maps, we drew rectangles around the maps and determined their ratios. Since the number of nodes in the grid is the product of its dimensions, we subsequently multiplied the ratios of the rectangle with a factor based on the total number of nodes of the real map. As an example, we take our Europe map, which consists of 20.321.661 nodes and can be boxed in by a rectangle with ratio 1105 : 1054. Hence, we determined the length n of the grid for Grerec-1 by

$$n \quad \sqrt{\frac{1105}{1054} \; 20{:}321{:}661} \quad 4616$$

and width m by

$$m \quad \sqrt{\frac{1054}{1105} \; 20{:}321{:}661} \quad 4403{:}$$

What remains is determining values for p and q, the probabilities of keeping edges in the grid and of generating shortcuts, respectively. We determined these values by calculating indices , and for the real maps, then computing many networks following the procedure of the GREREC model with varying values for p and q. For each generated map, we calculated , and , as well as the total number of edges, and picked a map with similar values to those of the real map. These indices are intended for undirected graphs, while the maps we use, as we mentioned earlier, are modelled as directed graphs. However, because of the low number of one-directional roads in the graphs, we simply use the underlying undirected graph to calculate the indices. Again, as an example we take the generation of Grerec-1 based on the Europe map: for the latter map we calculated $\langle \; ; \; ; \; \rangle \quad (0{:}09; 1{:}18; 0{:}40)$. Using parameter values $p \quad 0{:}55$ and $q \quad 0{:}1$ generated a map with the same values for these indices. Moreover, the random network contains 47.918.162 (undirected) edges, very close to the number of edges in the underlying graph of the Europe map (45.804.119). We converted each undirected edge of the random network to two directed edges and used this map as Grerec-1 during our experiments. Grerec-2 was generated with input values $n \quad 6513$, $m \quad 3995$, $p \quad 0{:}6$ and $q \quad 0{:}1$, resulting in a map with properties similar to our USA map, for which we calculated $\langle \; ; \; ; \; \rangle \quad (0{:}13; 1{:}25; 0{:}42)$. In Tables 2-5 we present properties of all four maps we use in this project.

	EUROPE	USA	GREREC-1	GREREC-2
Nodes	20.321.661	26.018.211	20.324.248	26.019.435
Edges	91.608.238	123.650.100	95.836.324	130.331.152
Boundary points	1.477.762	1.475.904	744.196	1.235.732
Shortcuts	17.921.501	17.482.309	5.089.656	19.315.832
Size (in GB)	2.54	3.29	2.44	3.43

Table 2: The total number of nodes, edges, boundary points, shortcuts per map and their sizes in gigabytes.

	Europe	Usa	Grerec-1	Grerec-2
	Level 1			
Nodes	236	299	231	319
Edges	1.063	1.422	1.092	1600
Boundary points	17	17	10	17
Shortcuts	98	111	35	95
	Level 2			
Nodes	2.192	2.772	2.170	2.910
Edges	9.882	13.175	10.230	14.575
Boundary points	39	36	21	41
Shortcuts	494	486	137	541
	Level 3			
Nodes	20.843	26.685	20.888	26.550
Edges	93.957	126.821	98.496	132.990
Boundary points	92	80	53	113
Shortcuts	2.818	2.282	860	4.062
	Level 4			
Nodes	203.217	260.182	203.242	260.194
Edges	916.082	1.236.501	958.363	1.303.312
Boundary points	223	165	153	341
Shortcuts	15.359	9.031	6.403	33.245
	Level 5			
Nodes	2.032.166	2.601.821	2.032.425	2.601.944
Edges	9.160.824	12.365.010	9.583.632	13.033.115
Boundary points	464	248	368	843
Shortcuts	58.523	19.198	36.370	188.095

Table 3: The average number of nodes, edges, boundary points and shortcuts per cell on each level of the partitions of the maps. Each boundary point is counted twice: once for the cells on both sides of the boundary it is situated on.

	min	max	avg
	Europe		
PBP-1 subgraph	171	375	287
PBP-2 subgraph	165	369	281
	USA		
PBP-1 subgraph	136	497	361
PBP-2 subgraph	134	494	359
	Grerec-1		
PBP-1 subgraph	142	387	273
PBP-2 subgraph	138	383	270
	Grerec-2		
PBP-1 subgraph	135	539	392
PBP-2 subgraph	115	523	574

Table 4: The minimum, maximum and average sizes of the subgraphs for PBP-1 and PBP-2 in megabytes.

	EUROPE	USA	GREREC-1	GREREC-2
Base level graph	39.1	53.9	67.1	87.2
Level-1 cell	1.75	2.01	0.864	1.70
Level-2 cell	9.2	8.92	2.49	10.2
Level-3 cell	53.6	43.9	17.1	75.7
Level-4 cell	287	168	117	600
Level-5 cell	896	385	711	2844

Table 5: The average size of cells in the overlay graph in kilobytes.

4.2 Azure Functions

Azure Functions is a compute service o ered by Microsoft [28] that we use to implement our algorithms. It allows users to implement software into blocks of code, called \functions". These functions are executed when they are subjected to a certain trigger. Possible triggers include HTTP-requests, timers and uploading a le to storage.

The main bene t of Azure Functions, and also the main reason why it suits our experiments, is that they scale automatically: when the number of requests increases, it provides the function with more resources, i.e. an extra instance of the function, to meet the demand. When the number of requests decreases, the number of instances drops automatically. This allows us to not worry about any scaling rules and just observe how many resources our algorithm needs in each scenario.

One instance of a function is provided with a certain number of virtual processors (vCPUs) and amount of memory, depending on the hosting plan. A vCPU is a physical CPU, in this case belonging to a computer in one of Microsoft's data centers, that is assigned to a particular virtual machine (or Azure Function in this case). Microsoft allocates these CPUs (and the memory), so as a user, we have no control whether our di erent functions have their resources allocated on the same underlying machine, while it could impact the performance of our communication steps. However, it is guaranteed that the functions are hosted on machines in the same data center, which should provide a reasonable upper bound: the latency for communication between computers in one data center should be smaller than 5 ms. This uncertainty regarding the performance of communication is one of our motivations to analyse the communication theoretically as well as empirically.

An Azure Function has a couple of options with respect to hosting. Microsoft o ers three basic hosting plans: a consumption plan, a premium plan and a dedicated plan. The dedicated plan is intended for long-running scenarios and is therefore not applicable for our experiments. The most relevant features for this research of the premium plan are that it o ers more memory than the consumption plan (up to 14 gigabytes versus 1.5 gigabytes) and that it enables the use of pre-warmed instances. Pre-warmed instances are instances that run despite demand not requiring them. When there is a peak in work load, they are readily available to handle part of it. When this happens, a new instance is pre-warmed. So, the pre-warmed instances act as a bu er in case of load peaks. In our experiments, almost all processes exceed 1.5GB memory use, so we use a premium plan for every process. In a premium plan, billing is based on the number of core seconds and the amount of memory allocated across instances.

When working with stateful work ows, Microsoft recommends using *Durable Functions*. This extension of Azure Functions lets the user write stateful orchestrator functions and (stateless) entity functions that are called by an orchestrator function and do actual calculations. Such an orchestrator function works as follows: it runs its code and stores intermediate results from entity functions. Then, it reruns its code multiple times, while checking every time an entity function

should be called, if its result is already in storage. If the result is already in storage, the orchestrator gets it from there; otherwise, it calls the entity function. This behaviour enables a function to keep state.

At first sight, this seemed to suit our application: we want to call multiple processes when performing a query and want to maintain a global state, e.g. the current tentative distance(s), messages to be sent etc. Therefore, we initially implemented our algorithms using Durable Functions. We found out that achieving any of our desired performance objectives was not possible this way: queries could take up to 20 minutes. We suspect the rerunning mechanism causes this problem, which made us decide to switch to regular Azure Functions. These functions are not necessarily designed to maintain state, but when the duration of one execution is short (which it is for answering shortest path queries) and by implementing some basic restarting mechanisms of our own, the functions performed as we expected.

4.3 Redis

Redis (short for \Remote Dictionary server") is an open source, in-memory data structure store, which can be used as a database, cache or message broker [19]. It achieves high performance by storing data in-memory rather than on a disk. When an application uses data from external sources, the latency and throughput (the rate of message delivery over the communication channel) of those sources can become the performance bottleneck, in particular during high work loads. A way to improve performance in such cases is to store data in-memory, physically closer to the application. Redis was created to do this: it uses the main memory of a server to store data types and structures such as strings, hashes, bit arrays, lists, and sets [33]. Users can connect to the server and perform atomic operations on these types, e.g. appending to a string, incrementing the value in a hash, or adding an element to a list.

A Redis database stores key/value pairs. A Redis key must be a string, while the associated value can be of any of the supported data types or structures.

We will use a Redis database to store graph data for our ODL approach. Microsoft offers Azure Redis Cache: a Redis database on a Microsoft server. Different pricing options are offered. For our application we require high networking performance, as we want to load a query's search graph and still answer the query within milliseconds. We therefore use a Redis P4 cache, which promises the best networking performance (bandwidth of 750 megabytes per second) and provides a 53 gigabytes database size.

One advantage of using an in-memory database is that performance is expected to be significantly better than when we would store data on a disk. Among in-memory databases, Redis offers the most high-end features, like supporting more advanced data structures and snapshots (periodically saving the dataset on disk). Furthermore, Redis achieves high performance: we experienced loading times of approximately 9 ms per megabyte with a Redis P4 cache, which should be fast enough to load the query's search graph at the moment the query arrives. A drawback of Redis is its cost: it is considerably more expensive than in-memory alternatives with fewer high-end features like Memcached [26] or than on-disk alternatives like MongoDB [29].

4.4 The algorithms

We will describe our implementations of PBP-1, PBP-2 and ODL in this section. All code is written in C#. The communication messages are in JSON format. For each process, we create one Azure Function with its own hosting plan with four cores and 14 gigabytes of RAM per instance (Azure's EP3 plan). To compare our algorithms to the normal CRP algorithm, we used Ortec's

implementation of CRP. We created one Azure Function, to which we refer as the orchestrator process of regular CRP, which keeps the entire graph in memory, receives queries, performs the CRP algorithm and returns the result.

In our CRP implementation, we calculate queries from a source edge to a target edge, instead of from a source node to a target node. These edges have an associated oat inside the unit interval, to indicate where on the edge the source/target exactly is situated. This enables more precise routing: instead of having to start and nish a query at a node, we can start and nish at any point on an edge. These source and target edges are traversed only partly. When we reach the rst node, the algorithm is just as we described.

PBP-1 Our PBP-1 implementation uses K 1 processes in total: one orchestrator processes and one process per subgraph. The subgraphs are constructed by taking the highest level of the overlay graph and assigning each of the K cells to a di erent process. The subgraph then consists of one top level cell and all its subcells. Additionally, we store the boundary points and shortcut costs of all K top level cells (naturally, without their subcells) on each process as well. The orchestrator only keeps one mapping \EdgeToProcess" in memory. This mapping is built in preprocessing and consists of (edge, process number)-pairs, so that the orchestrator can determine for each edge on which process it is situated. In our pseudocodes (Appendix A), we show the algorithms that compute node-to-node shortest paths. In such a case, we would create a \NodeToProcess" mapping with (node, process number)-pairs. When the orchestrator receives a query, it determines on which process the source and target lie. If both nodes/edges are on one process, the orchestrator sends the query to that process. After performing a bidirectional CRP search, the process returns a shortest path to the orchestrator. If the source and target processes are two di erent processes, the orchestrator sends a message to the target process, which can start its backward search and return its search space to the orchestrator process. The orchestrator then sends the search space to the source process, which conducts the forward search and returns a shortest path. In Figure 10, we show a diagram of the communication during a PBP-1 query.

For many-to-many queries, the orchestrator process uses the aforementioned mapping to determine all source and target processes. It sends a message containing local targets T_p to each target process p. Each target process performs (in parallel) a backward search for each of its targets $t \in T_p$ and sends its search space back to the orchestrator. The orchestrator combines these search spaces to one search space and sends this combined search space to each source process. Each source process p performs (again, in parallel) a forward search for each of its sources $s \in S_p$, calculating shortest distances to each target $t \in T$. It returns these distances, after which the orchestrator can ll the entire $|S| \cdot |T|$ cost array by combining the messages of all source processes.

Note that this implementation di ers a bit from our description of PBP-1 in Section 3.1.1. Instead of having one message from the target process to the source process, we send it twice: once from the target process to the orchestrator and once from the orchestrator to the source process. For many-to-many queries, we send the entire backward search space to all source processes. We will mention possible ideas to not have all communication go through the orchestrator process in Section 6.1.

PBP-2 The PBP-2 implementation also consists of K subprocesses, each containing its own subgraph and one orchestrator process that only stores the EdgeToProcess map. These subgraphs are created during preprocessing just as for PBP-1, but without the duplication of the entire highest level of the overlay graph for every subgraph. The orchestrator receives the query and sends a message to the source and target process, which perform a forward and backward search respectively

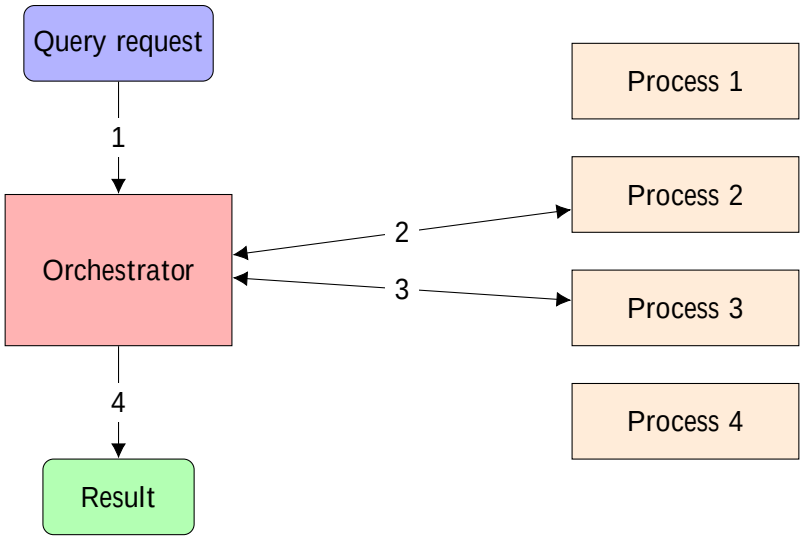

Figure 10: PBP-1 communication for a one-to-one query for which p_{pt_q} process 2 and p_{ps_q} process 3.

on their subgraphs. Both processes ll an array per neighbouring process with (boundary point, cost)-pairs they should send to these processes. The arrays are returned to the orchestrator. Then, for every label-correcting step: the orchestrator collects all returned message arrays and creates the messages that need to be sent to the subprocesses; it sends each message to the corresponding process, which perform their bidirectional label-correcting searches and return the new message arrays and shortest distances $D_{ps;t_q}$ they found on their subgraph; and the orchestrator, if applicable, improves (global) tentative shortest distance $D_{ps;t_q}$. This is repeated until all returned message arrays are empty after an iteration, which indicates we have found a shortest path. In Figure 11 we show a diagram of the communication during a PBP-2 query.

Each process has to maintain a state per query, as it needs the already calculated tentative distances for its boundary points in order to perform a new label-correcting step. We accomplish this by maintaining a data structure in which we store the priority queues per query, which we associate with a query id. Each new query that is received by the orchestrator is assigned a unique id, which is sent with every message regarding that query. Every subprocess can access the priority queue corresponding to the query using this id. When all iterations have passed, i.e. we have determined a shortest path, the orchestrator sends a message to all subprocesses indicating they can delete local data corresponding to the query.

For many-to-many queries, the structure is similar. One di erence is that the message arrays contain (boundary point, cost, target)-triples to indicate which target is associated with the sent cost. Furthermore, we only perform label-correcting steps for the forward search, so after the target processes performed their searches, the label-correcting steps consist of unidirectional CRP searches. The last di erence in implementation is that, instead of updating the tentative distance at each iteration, we wait until the nal iteration is performed, after which every source process returns its $|S_p| \cdot |T|$-sized part of the complete cost array. The orchestrator composes the complete cost array by combining these costs.

Our implementation does not use the assumption that we considered paths outside the cell to compute its shortcuts in preprocessing, so we do need label-correcting steps even when the source and target are contained in one process's subgraph.

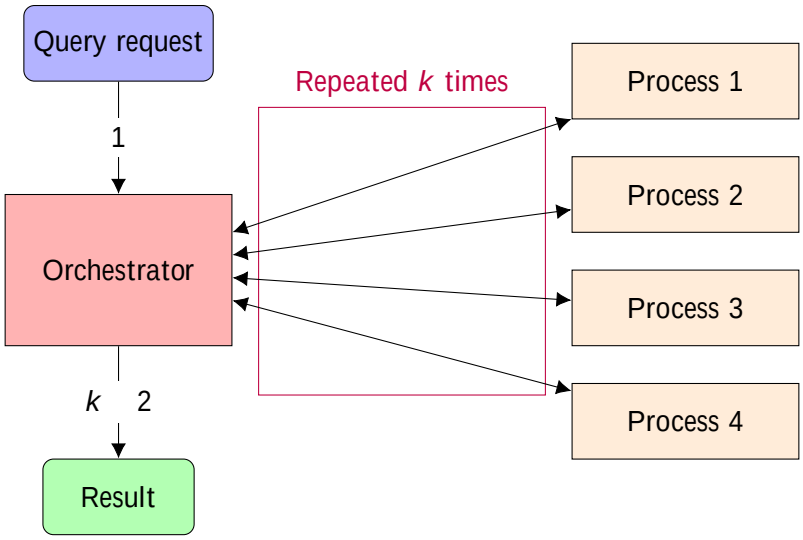

Figure 11: PBP-2 communication for a one-to-one query for which k label-correcting iterations are needed; $p(p(t))$ process 2 and $p(p(s))$ process 3.

ODL In our ODL algorithm, we use an orchestrator process as for the PBP approaches to answer queries. For the database containing all graph data, we use Redis and C# client StackExchange.Redis [24]. In preprocessing, we construct the database as follows. We store all base level cells as Redis values associated with their entire partition code of 4 L bits as its Redis key. Furthermore, we store the cells (the boundary points and their corresponding shortcut costs) by using a string encoding the level l of the cell and the cell's partition code as Redis key. All our Redis values are stored as byte arrays, so we do not use the JSON format as in the PBP approaches.

For each query, we know exactly which cells we need, as we described in Section 3.2. We can determine the partition codes of these cells based on the partition codes of our source(s) and target(s). We assume that we know these partition codes when the query arrives. In applications where this is not the case, we could store a mapping on the orchestrator process containing this information for each node. The Redis keys of the base level cells are simply the entire partition codes of these nodes. The partition codes of cells $C^l(s)$ and $C^l(t)$ are determined by looking at the 4 $(L - l)$ most significant bits. As shown in Proposition 3, we need cells $\{C^l \mid C^l \neq C^l(v) \wedge C^l \in C^{l-1}(v)\}$ for $v \in \{s, t\}$. For each level l, the partition codes of cells $C^l \in C^{l-1}(v)$ share the same first 4 $(L - l + 1)$ bits. So, we can determine which cells we need by taking the first 4 $(L - l + 1)$ bits of the partition codes of both s and t, loading all cells on level l whose partition codes start with the same bits, except the cells containing s or t. The orchestrator does not load any data before a query arrives. It loads the necessary cells from the database as described above, performs the bidirectional CRP algorithm on the composed graph and computes a shortest path. In Figure 12, we show the communication diagram of ODL.

4.5 Queries

During our experiments we tested our algorithms on a few different types of queries. In this section, we will describe how we generated these queries. All requests containing the queries were sent to the Azure Functions of the orchestrator processes of the algorithms by using Postman [32]. We first conducted experiments using random one-to-one and many-to-many queries with $|S| = |T| = 100$. Initially, we generated larger matrix requests as well, e.g. queries with $|S| = |T| = 1000$. However,

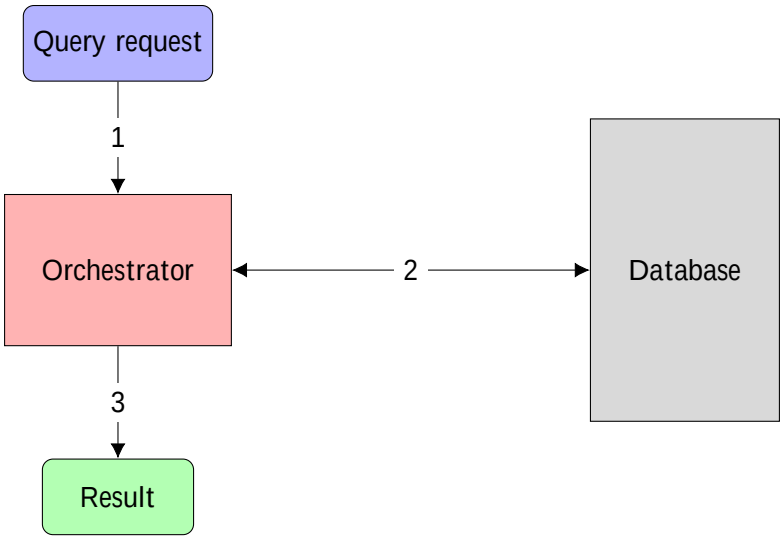

Figure 12: ODL communication for a one-to-one query.

Azure Functions have a maximum size it allows for its JSON messages and during such large requests, some of the PBP messages exceeded this maximum size. This could possibly be solved by using compression on the messages, but we did not investigate this during our project and only performed many-to-many queries with at most 100 sources or targets. The random queries are generated by simply picking two random nodes for a one-to-one or $|S|$ $|T|$ random nodes for a many-to-many request.

To investigate the scalability of each approach, we test the algorithms on three di erent types of loads. First, we look at the scenario that each query is performed consecutively: after a query is sent, we wait until the result is returned by the algorithm, before we send the next request. Second, we divide all queries in equally sized batches, which we perform concurrently. This creates a heavier load on the system, which enables us to investigate how the algorithms handle such scenarios. The third scenario is created using real customer data from Ortec. One of the customers of Ortec which uses their routing engines is a Dutch grocery deliverer and we simulated two hours of their requests to test our algorithms. We collected the source and target locations, as well as the timestamp of each request during two di erent time periods of one hour. This resulted in two sets of queries: \Scenario 1" consisting of 62.088 one-to-one requests and 860 many-to-many requests and \Scenario 2" with 45.106 one-to-one queries and 455 many-to-many queries. Every request has an associated timestamp. All locations were in the Netherlands, so we used our Europe map to answer the queries. Some of the many-to-many queries had source or target sets S or T with more than 100 nodes. For the reasons we explained above, we only took the rst 100 nodes of such sets. Ultimately, this is the goal of our algorithms: they should work well in realistic scenarios. These scenarios have peaks and low points in the number of requests, which can provide insight in the ability of the algorithms to scale up and down during such varying loads. Furthermore, we are interested in their performance during such loads.

To investigate the impact of the query distance on the performance of the algorithms, we generated sets of queries per *Dijkstra rank*. A node v has Dijkstra rank r with respect to source s when it is scanned during the r-th iteration of Dijkstra's algorithm started from s. We generated these queries by picking a random node of the graph as source s, then performing Dijkstra's algorithm from s until we reached the desired Dijkstra rank. We took the node we scanned last as the target node of the query. In realistic applications, it is unlikely that many queries have

sources and targets on opposite sides of continental-sized graphs. Therefore, we focus on Dijkstra ranks that are not too high. The maximum Dijkstra rank we used was 2^{23}. Most queries regarding Dijkstra ranks have their source and targets inside one top level cell, only for the highest two ranks we used (2^{21} and 2^{23}) queries cross top level cell boundaries.

5 Experiments and results

5.1 Setup

In our experiments, we test four algorithms (PBP-1, PBP-2, ODL and regular CRP) on four di erent maps (Europe, USA, Grerec-1 and Grerec-2). For each algorithm and each map, we perform the following sets of queries:

1. 10.000 random one-to-one queries,
2. 50 random 100x100 queries,
3. 1000 queries per Dijkstra rank in $t2^7; 2^9; : : : ; 2^{23}u$,
4. Scenario 1 (62.088 one-to-one queries, 860 many-to-many queries), and
5. Scenario 2 (45.106 one-to-one queries, 455 many-to-many queries).

The rst two types of query sets were performed sequentially and in multiple concurrent batches, as we described in Section 4.5. We used 100 concurrent batches consisting of 100 queries each for the one-to-one queries and 10 batches of 5 queries each for the 100x100 requests. We executed the sequential queries twice: once to log additional information regarding the queries such as message size and function startup time and once to measure performance, so that performance was not decreased by the extra logging. As for the query information, we rst looked at the message sizes of each algorithm. For PBP-1 and PBP-2, we logged the message sizes during the random one-to-one and many-to-many queries. During the PBP-2 queries, we additionally examined the number of iterations per query, as well as the average number of updates per iteration: the number of processes that perform label-correcting calculations on their subgraphs per iteration. These results can be found in Tables 6 and 7. The average message sizes of ODL can be computed using Table 5, the results of which we present in Table 8. During the same queries, we also kept track of the time it took to start a new instance. The average startup times are shown in Table 9. Lastly, we investigated the distribution of work load among the subprocesses for both PBP algorithms during the random queries. We compared them to the number of nodes per subgraph, as we expected (as mentioned in Section 3.3) them to correlate, especially for PBP-1.

To measure performance, we looked at the duration of the queries, the memory consumption per function instance and the total number of instances that were used. During some of the experiments we experienced some failing requests, which were registered as queries with a duration of ve minutes. This caused the average duration to give a distorted representation of the performance, which is why we chose to only consider the succeeded requests when calculating the average query duration. In Figure 13, we show the distribution of query durations per algorithm during the rst experiment we conducted. Each algorithm had similar distributions for the other experiments, except for the experiments with queries per Dijkstra rank, of which we show the query durations in Figure 17. The number of instances remained constant in during the experiments regarding the random queries, which is expected since the load remains constant during these experiments as well. During the realistic load scenarios, the number of used instances did uctuate, which we show

in Figures 18-21. For all experiments, we present the average duration and the standard deviation of the duration for the succeeded queries; the average memory usage and standard deviation of the memory usage per instance and the total number of used instances. We calculated the average and standard deviation for memory usage by recording the memory usage every second and taking the average and standard deviation of that data. The performance results for the random one-to-one queries are shown in Tables 10 and 11, followed by the performance results for the random many-to-many queries in Tables 12 and 13.

We continue with the experiments regarding the queries per Dijkstra rank. We experienced very similar results for every map, so we chose to only present the results of the duration per rank for one map: the USA map. These can be found in Figure 17.

Finally, we performed the experiments using the real customer data from Ortec. In Figure 14 we show the distribution of these queries. In our Results section, we show the same performance measures as for the random queries in Tables 14 and 15. We are particularly interested in their scaling behaviour, i.e. how they react to peaks and low points in the number of received requests, which is why we plotted the number of running instances over time in Figures 18-21, comparing them to the distribution of the received requests. Because of the fact that all queries were situated in the Netherlands, PBP-1 only used two of its ten subprocesses of the Europe map: processes with numbers 3 and 4. This is the reason why we show the number of instances of process 4 in Figure 19.

As we will see in the upcoming results, the PBP-2 did not handle heavy loads well. We experienced multiple failing requests during a many-to-many experiment on the Grerec-2 map (indicated with *), to the point where we were uncomfortable presenting any results. Additionally, the realistic load scenarios were too much to handle for this algorithm: it used an explosive number of instances (couple of tens per subprocess), had very slow query times and many failing requests. We therefore chose to exclude these results as well. In our discussion of the results in Section 5.3, we will delve deeper into the possible reasons for these problems. One ODL experiment failed as well: the many-to-many test on the Grerec-2 map (indicated with **) did not yield results. We expect this is due to the large cell sizes of Grerec-2.

5.2 Results

	Europe	Usa	Grerec-1	Grerec-2
One-to-one	462	332	234	492
Many-to-many	1152	963	652	1651

Table 6: The average message sizes of PBP-1 in kilobytes. For the many-to-many, the average size of the message from a target process to the orchestrator is shown.

	EUROPE	USA	GREREC-1	GREREC-2
	One-to-one			
Iterations	8.4	7.8	7.3	8.3
Updates per iteration	4.7	3.9	4.1	4.0
Message size	12	9	16	60
	Many-to-many			
Iterations	13.2	13.1	16.1	17.9
Updates per iteration	6.7	5.7	5.0	4.7
Message size	84	53	110	397

Table 7: The average number of iterations, the average number of cells that update their boundary points during one iteration and the average message size in kilobytes for PBP-2 queries.

	EUROPE	USA	GREREC-1	GREREC-2
$l_0ps;tq$ 1	79	108	134	174
$l_0ps;tq$ 2	110	144	150	205
$l_0ps;tq$ 3	275	304	195	388
$l_0ps;tq$ 4	124	1096	503	1751
$l_0ps;tq$ 5	5437	4116	2611	12595
$l_0ps;tq$ 6	21606	11059	15462	63693

Table 8: The average size of the preloaded data for an ODL one-to-one query for di erent values of the lowest sharing level $l_0ps;tq$ of source s and target t in kilobytes. These results are computed as follows: for each level used in the query (levels l $l_0ps;tq$), multiply the average cell size from Table 5 by 2 K 2 18, sum them and add the average size of two base level graphs.

	PBP		**ODL**	**RegCrp**
	Orch.	Sub.		
EUROPE	27 (3)	45 (13)	0.54 (0.43)	55 (12)
USA	35 (5)	47 (6)	0.64 (0.35)	57 (9)
GREREC-1	26 (2)	39 (9)	0.97 (0.19)	42 (5)
GREREC-2	36 (4)	48 (5)	0.95 (0.14)	66 (22)

Table 9: Startup times of new instances of processes in seconds. The averages, with standard deviation between brackets, over all conducted experiments are shown. For PBP, both the startup times for the orchestrator process (Orch.) and the subprocesses (Sub.) or shown.

	PBP-1		PBP-2		ODL		RegCRP	
	Seq.	Conc.	Seq.	Conc.	Seq.	Conc.	Seq.	Conc.
	EUROPE							
Duration (ms)	149	186	581	2:1e3	587	2:9e3	34	34
	(65)	(128)	(266)	(2:1e3)	(32)	(939)	(23)	(31)
Memory (GB)	4.2	4.3	2.0	2.3	0.5	3.5	4.4	5.3
	(0.4)	(0.5)	(0.3)	(0.6)	(0.3)	(1.6)	(0.6)	(1.2)
Used instances	1	1	1	7	1	4	1	3
	USA							
Duration (ms)	109	124	450	904	210	1:3e3	20	23
	(76)	(162)	(241)	(578)	(25)	(804)	(10)	(37)
Memory (GB)	6.8	8.5	3.8	3.9	1.2	0.3	6.0	7.1
	(0.7)	(0.9)	(0.2)	(2.0)	(0.3)	(0.4)	(1.0)	(3.6)
Used instances	1	1	1	9	1	3	1	4
	GREREC-1							
Duration (ms)	81	83	499	1:8e3	536	1:4e3	16	17
	(42)	(58)	(223)	(1:8e3)	(116)	(635)	(11)	(23)
Memory (GB)	4.2	8.5	3.5	2.7	1.3	0.9	4.0	5.3
	(0.4)	(0.5)	(0.5)	(1.9)	(0.3)	(0.4)	(1.2)	(2.7)
Used instances	1	1	1	9	1	5	1	3
	GREREC-2							
Duration (ms)	157	246	1:25e3	6:1e3	741	5:9e3	70	93
	(93)	(222)	(467)	(4:0e3)	(77)	(1:7e3)	(32)	(62)
Memory (GB)	7.4	6.6	2.6	4.2	5.3	2.7	8.1	6.7
	(0.5)	(1.3)	(0.1)	(1.3)	(1.9)	(1.9)	(1.2)	(3.9)
Used instances	1	1	1	10	1	8	1	8

Table 10: Performance results of the orchestrator processes of the di erent algorithms during the random one-to-one queries. The 10.000 queries were executed sequentially (Seq.) as well as in 100 concurrent batches of 100 queries (Con.). The average duration of all succeeded queries in milliseconds, with the standard deviation between brackets; average memory usage during the queries in gigabytes per instance, with the standard deviation between brackets; and the total number of used instances are shown.

	PBP-1		PBP-2	
	Seq.	Conc.	Seq.	Conc.
	EUROPE			
Duration (ms)	38.0 (25.8)	50.6 (57.9)	10.4 (12.9)	18.8 (21.0)
Memory (GB)	2.0 (0.4)	1.8 (0.7)	1.1 (0.3)	3.5 (1.2)
Used instances	1	3	1	2
	USA			
Duration (ms)	28.8 (24.4)	32.3 (50.0)	4.34 (3.64)	4.79 (3.97)
Memory (GB)	4.6 (0.7)	3.4 (1.5)	9.8 (0.4)	8.3 (3.3)
Used instances	1	2	1	2
	GREREC-1			
Duration (ms)	22.0 (18.3)	21.3 (27.3)	6.9 (7.4)	20.3 (26.2)
Memory (GB)	4.6 (0.5)	4.3 (1.2)	10.6 (0.9)	7.7 (3.7)
Used instances	1	2	1	5
	GREREC-2			
Duration (ms)	52.2 (33.8)	60.1 (42.8)	19.4 (26.2)	63.7 (71.3)
Memory (GB)	4.5 (0.3)	4.1 (1.2)	10.1 (0.7)	9.5 (3.1)
Used instances	1	2	1	9

Table 11: Performance results of the subprocesses of the PBP algorithms during the random one-to-one queries. The 10.000 queries were executed sequentially (Seq.) as well as in 100 concurrent batches of 100 queries (Con.). The average duration of all succeeded tasks in milliseconds, with the standard deviation between brackets; average memory usage during the tasks in gigabytes, with the standard deviation between brackets; and the total number of used instances are shown.

	PBP-1		PBP-2		ODL		RegCRP	
	Seq.	Conc.	Seq.	Conc.	Seq.	Conc.	Seq.	Conc.
	EUROPE							
Duration (s)	34.5	41.4	38.1	93.0	57.0	92.1	4.45	28.1
	(7.9)	(21.2)	(7.0)	(43.3)	(4.1)	(51.1)	(0.167)	(23.9)
Memory (GB)	4.5	4.1	3.7	4.5	0.4	1.0	8.7	6.5
	(0.7)	(2.1)	(1.3)	(2.6)	(0.1)	(0.5)	(0.8)	(1.8)
Used instances	1	3	1	3	2	2	1	3
	USA							
Duration (s)	27.0	92.2	23.3	70.0	18.9	33.3	2.31	2.53
	(9.5)	(53.0)	(13.6)	(53.5)	(0.353)	(7.7)	(0.143)	(0.218)
Memory (GB)	10.9	6.4	4.4	4.5	0.5	1.3	9.6	10.1
	(1.3)	(3.0)	(1.0)	(2.5)	(0.1)	(0.7)	(0.7)	(1.8)
Used instances	1	3	1	3	1	2	1	3
	GREREC-1							
Duration (s)	20.7	45.1	41.5	99.4	45.2	75.1	1.99	4.58
	(11.7)	(34.2)	(9.9)	(56.5)	(2.26)	(25.4)	(0.075)	(2.51)
Memory (GB)	6.7	4.9	6.3	4.0	1.2	1.1	8.0	7.6
	(0.8)	(2.7)	(0.2)	(3.5)	(0.1)	(0.8)	(1.2)	(1.1)
Used instances	1	3	1	4	1	3	1	2
	GREREC-2							
Duration (s)	43.2	68.4	172		56.5		9.28	21.3
	(1.3)	(26.9)	(13.1)		(0.623)		(0.342)	(19.0)
Memory (GB)	7.5	8.1	7.6	*	0.7	**	7.8	5.9
	(0.6)	(2.7)	(0.9)		(0.2)		(3.1)	(3.4)
Used instances	1	2	1		1		1	4

Table 12: Performance results of the orchestrator processes of the di erent algorithms during the random 100x100 queries. The 50 queries were executed sequentially (Seq.) as well as in 10 concurrent batches of 5 queries (Con.). The average duration of all succeeded queries in seconds, with the standard deviation between brackets; average memory usage during the queries in gigabytes, with the standard deviation between brackets; and the total number of used instances are shown.

	PBP-1		PBP-2	
	Seq.	Conc.	Seq.	Conc.
	EUROPE			
Duration (s)	3.41 (3.65)	6.15 (8.51)	0.108 (0.152)	0.320 (0.483)
Memory (GB)	1.7 (0.8)	3.9 (1.7)	1.5 (0.2)	4.1 (2.1)
Used instances	2	4	1	2
	USA			
Duration (s)	2.86 (3.03)	8.03 (4.27)	0.039 (0.055)	0.264 (0.404)
Memory (GB)	5.7 (0.4)	3.5 (1.9)	5.4 (1.4)	4.0 (2.5)
Used instances	1	3	1	2
	GREREC-1			
Duration (s)	1.68 (1.68)	2.18 (2.37)	0.023 (0.040)	0.203 (0.429)
Memory (GB)	4.4 (0.9)	4.9 (2.7)	10.0 (2.8)	3.7 (2.3)
Used instances	1	2	1	2
	GREREC-2			
Duration (s)	4.65 (4.23)	4.69 (6.06)	0.027 (0.051)	
Memory (GB)	5.0 (0.3)	3.5 (2.0)	5.4 (2.1)	*
Used instances	1	1	1	

Table 13: Performance results of the subprocesses of the PBP algorithms during the random 100x100 queries. The 50 queries were executed sequentially (Seq.) as well as in 10 concurrent batches of 5 queries (Con.). The average duration of all succeeded tasks in seconds, with the standard deviation between brackets; average memory usage during the queries in gigabytes, with the standard deviation between brackets; and the total number of used instances are shown.

	PBP-1	ODL	RegCRP
	Scenario 1		
Duration one-to-one (ms)	27.9 (24.2)	116 (130)	7.24 (13.6)
Duration many-to-many (s)	20.2 (8.2)	121 (63.5)	4.33 (0.619)
Memory (GB)	6.9 (2.0)	1.7 (0.7)	7.4 (4.0)
	Scenario 2		
Duration one-to-one (ms)	30.2 (33.8)	87.1 (88.9)	7.14 (15.0)
Duration many-to-many (s)	22.5 (12.5)	93.3 (60.5)	4.33 (0.530)
Memory (GB)	6.7 (2.7)	1.6 (0.6)	8.1 (3.1)

Table 14: Performance results of the orchestrator processes of the di erent algorithms during the realistic load scenario's. The average duration of all succeeded one-to-one queries in milliseconds, the average duration of all succeeded many-to-many queries in seconds and the average memory usage during the queries in gigabytes are shown. In each column, the standard deviation is shown between brackets.

	PBP-1
	Scenario 1
Duration one-to-one (ms)	16.5 (16.4)
Duration many-to-many (s)	6.78 (5.78)
Memory (GB)	7.7 (2.9)
	Scenario 2
Duration one-to-one (ms)	17.0 (17.8)
Duration many-to-many (s)	5.02 (2.61)
Memory (GB)	7.8 (3.1)

Table 15: Performance results of the subprocesses of PBP-1 during the realistic load scenarios. We only considered the two active processes. The average duration of succeeded executions during all one-to-one queries in milliseconds, the average duration of succeeded executions during all many-to-many queries in seconds and the average memory usage during the queries in gigabytes are shown. The standard deviations are shown between brackets.

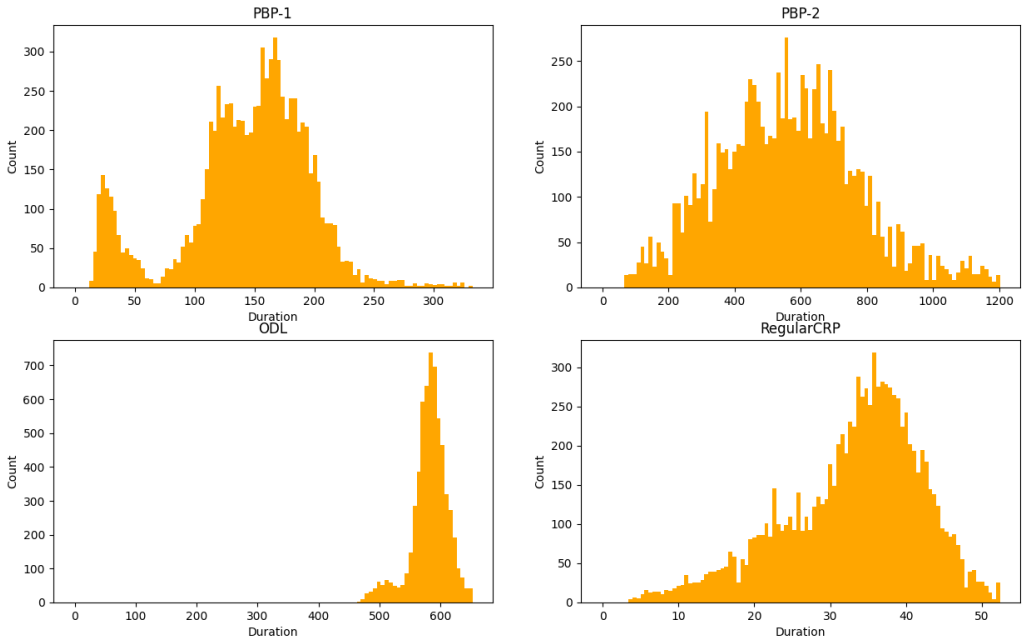

Figure 13: Distribution of run times in milliseconds for the di erent algorithms during the 10.000 one-to-one queries on the Europe map. The horizontal axis is split into 100 buckets and for every bucket the number of queries with a duration within that bucket is shown.

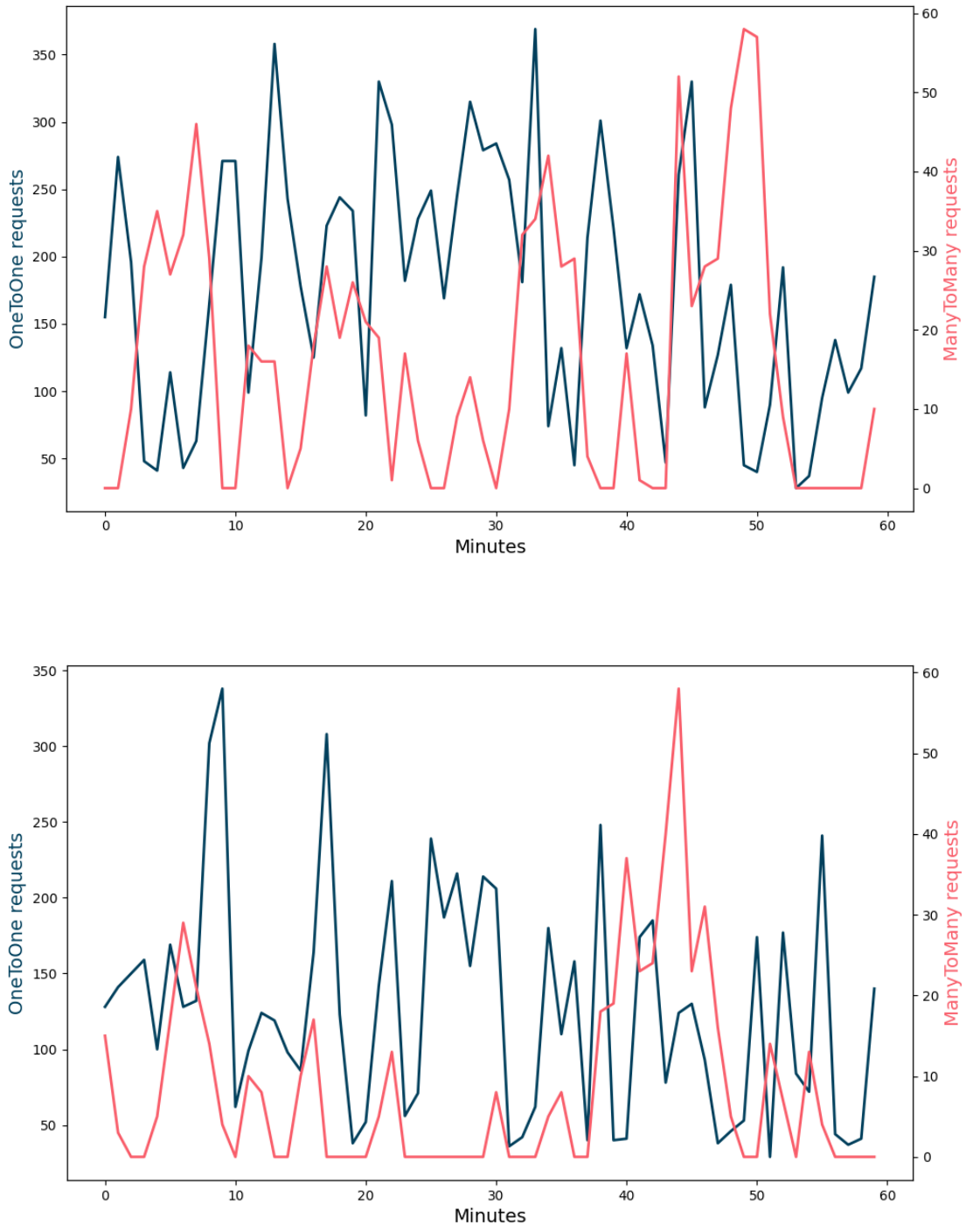

Figure 14: Distribution of queries in the realistic load scenarios 1 (top) and 2 (bottom). The number of received one-to-one requests (blue) and many-to-many requests (pink) are shown.

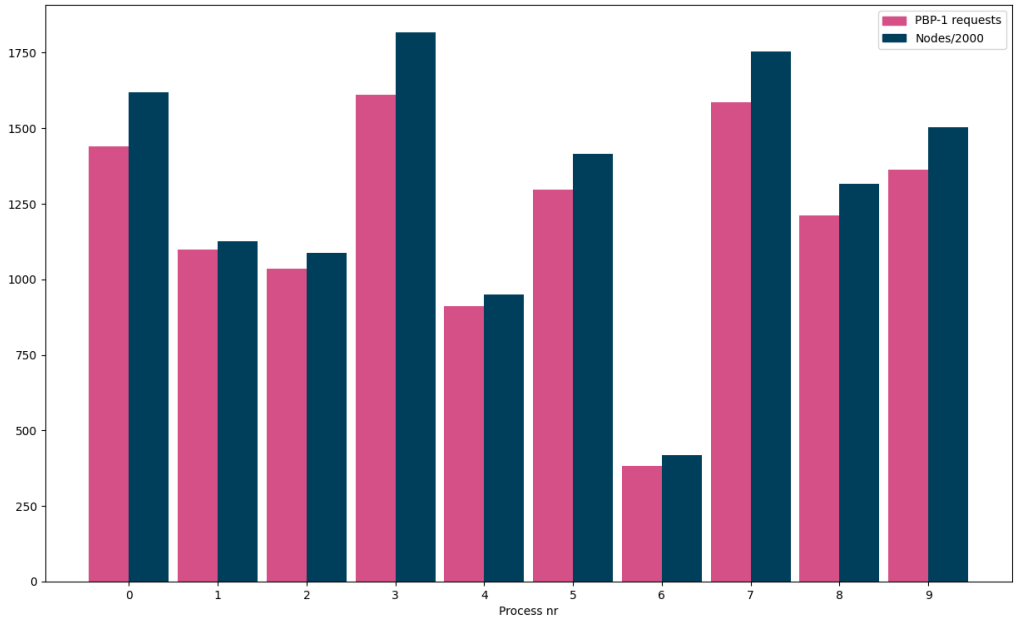

Figure 15: Distribution of work load for PBP-1 during the 10.000 random one-to-one queries on the USA map. We show the number of received requests per subprocess in pink. In blue, the number of nodes (divided by 2000) per process is shown as a reference.

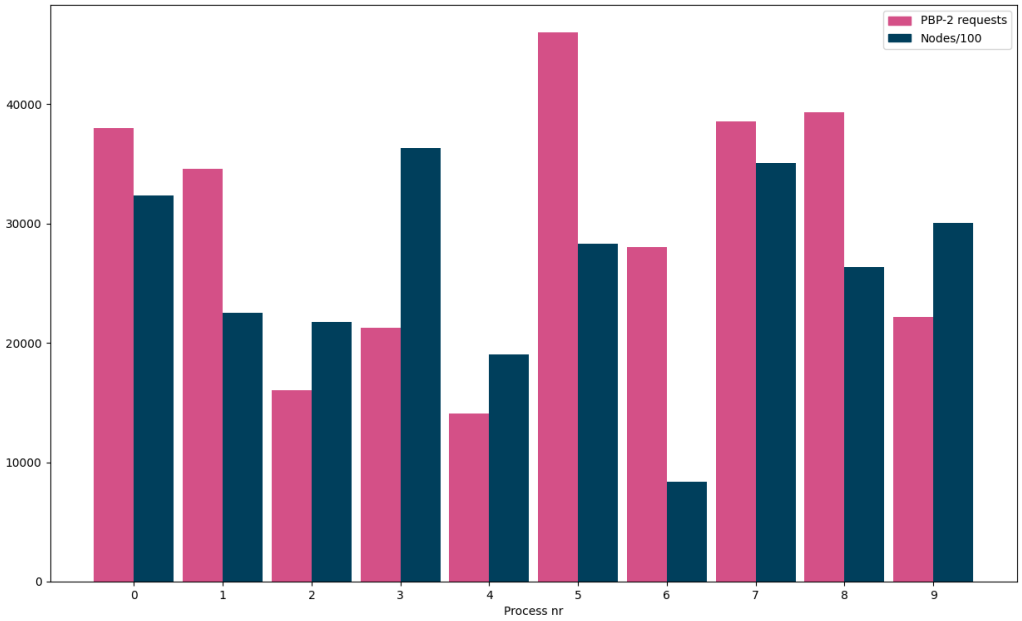

Figure 16: Distribution of work load for PBP-2 during the 10.000 random one-to-one queries on the USA map. We show the number of received requests per subprocess in pink. In blue, the number of nodes (divided by 100) per process is shown as a reference.

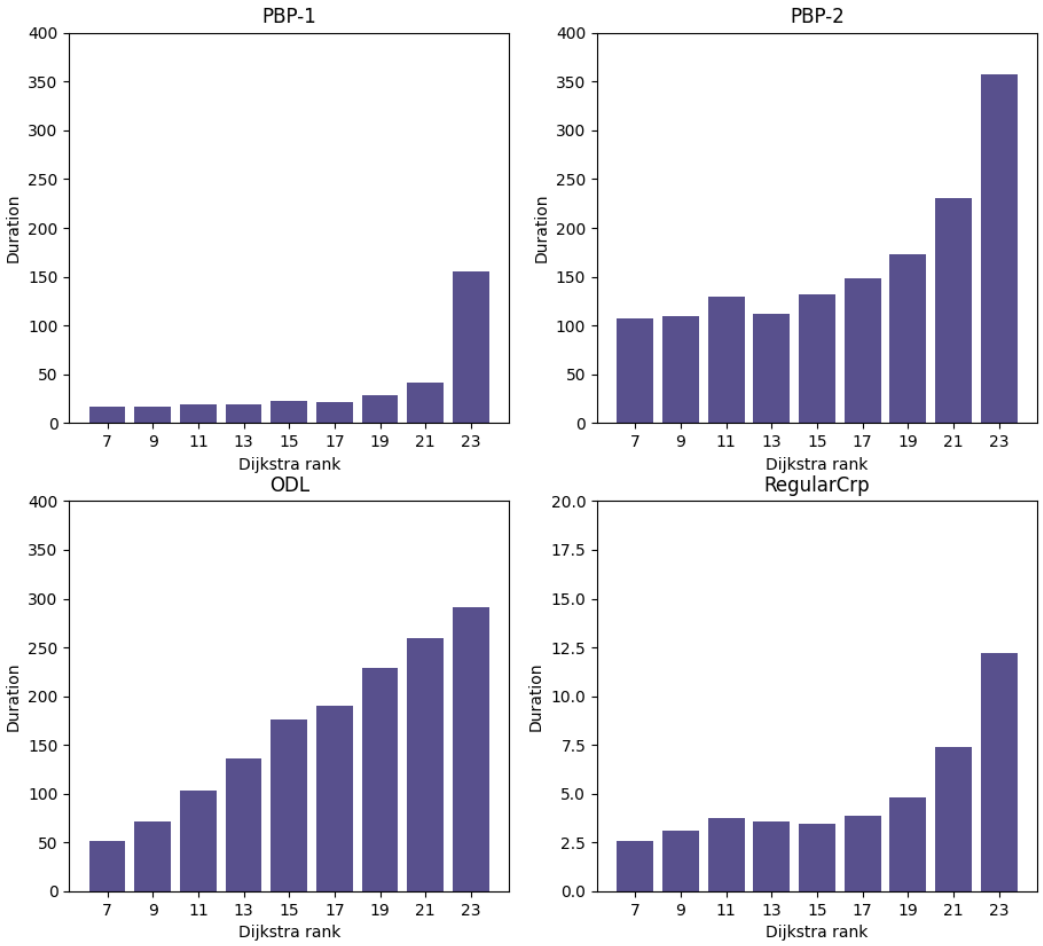

Figure 17: Run time in milliseconds of each algorithm per Dijkstra rank (in powers of two).

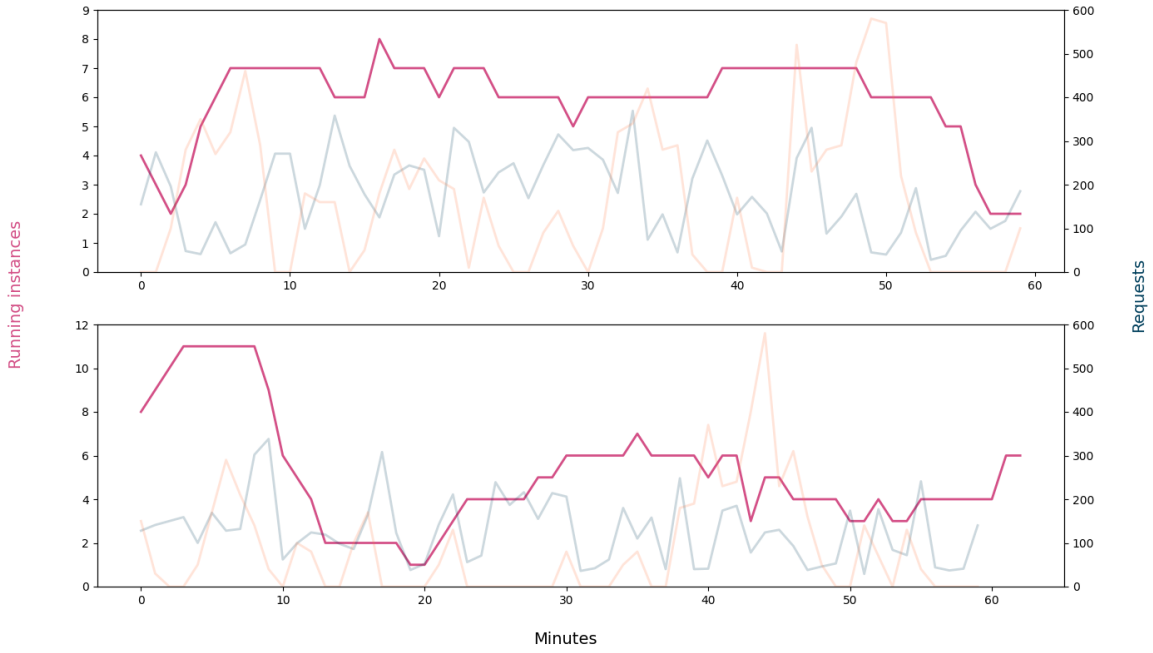

Figure 18: Number of instances during the two realistic load scenarios for the orchestrator process of PBP-1 (pink). As a reference, we show the number of received one-to-one requests (blue) and the number of received many-to-many requests multiplied by 10 (orange).

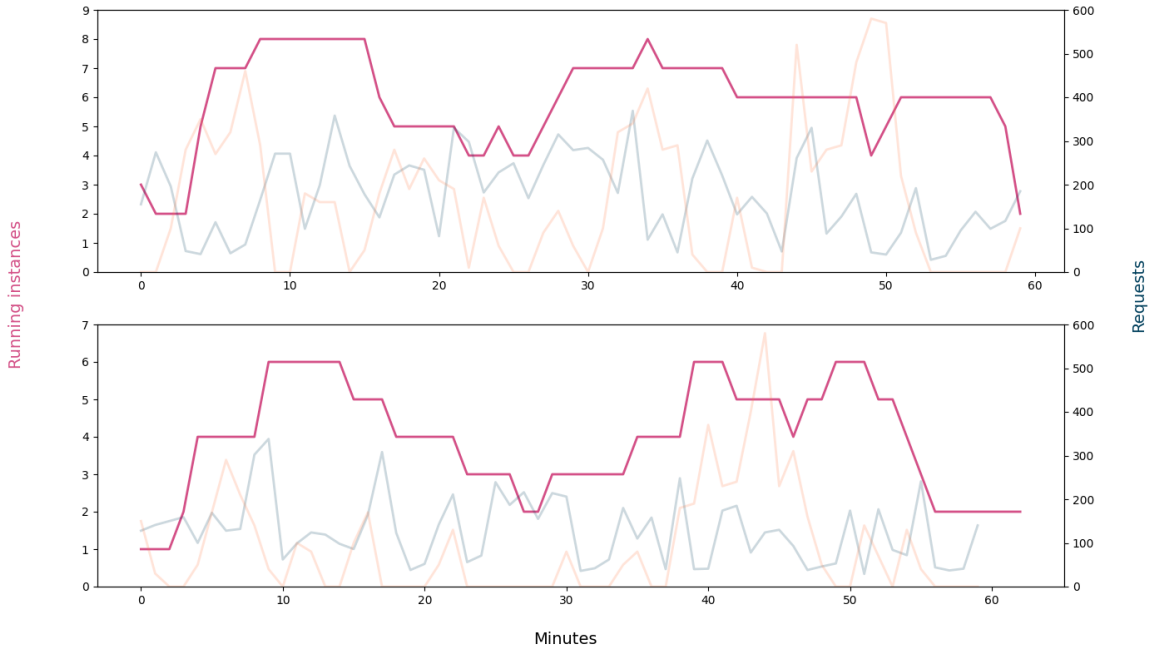

Figure 19: Number of instances during the two realistic load scenarios for process 4 of PBP-1 (pink). As a reference, we show the number of received one-to-one requests (blue) and the number of received many-to-many requests multiplied by 10 (orange).

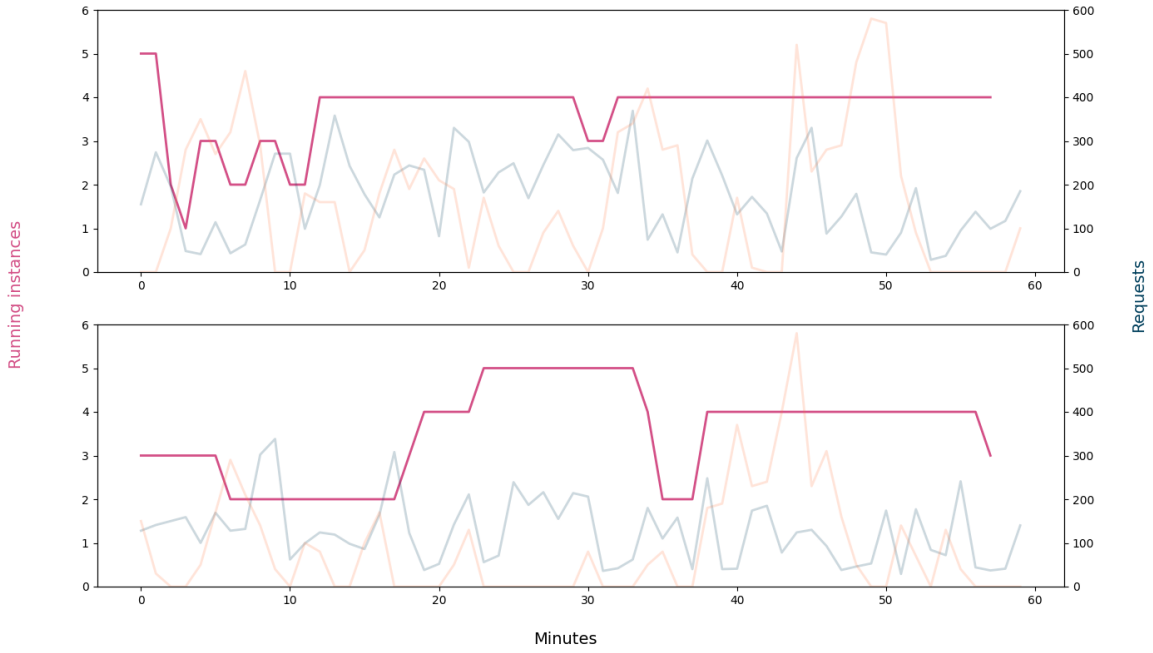

Figure 20: Number of instances during the realistic load scenarios for ODL (pink). As a reference, we show the number of received one-to-one requests (blue) and the number of received many-to-many requests multiplied by 10 (orange).

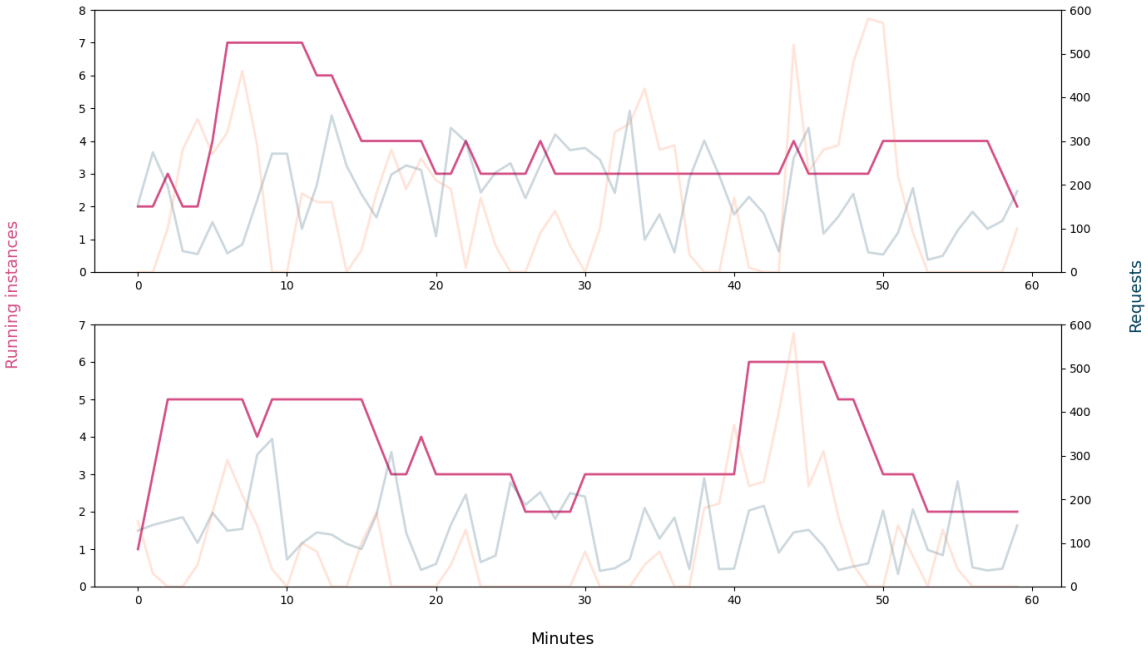

Figure 21: Number of instances during the realistic load scenarios for RegCRP (pink). As a reference, we show the number of received one-to-one requests (blue) and the number of received many-to-many requests multiplied by 10 (orange).

5.3 Discussion of the results

Performance In order to improve scalability, all algorithms experienced lesser performance compared to the regular CRP algorithm, which is in accordance with our theoretical analyses of Section 3. As we saw in those analyses, PBP-1 and ODL both execute the CRP algorithm with one added communication step consisting of one message (or actually two in our PBP-1 implementation), while PBP-2 adds multiple communication steps with multiple messages per step and extra computations for its label-correcting approach. Looking at the results regarding these communication steps during the experiments in Tables 6-8, we see that the message sizes for PBP-1 queries are on average around 500 kilobytes for a one-to-one and around 1 megabyte for the 100x100 queries. The message size for PBP-2 is much smaller than for PBP-1: a factor 8 up to a factor 30. Furthermore, we see it uses on average approximately 8 iterations with 4 active processes (i.e. processes that perform a label-correcting step on their subgraphs) for one-to-one queries and 13-18 iterations with 5-7 active processes during 100x100 queries. ODL loads for long-range queries, in which we need the top-level cells, 10-60 megabytes of graph data, depending on the map. The amount of data decreases fast when we only need lower levels of the overlay graph, since the top level cells by far take up the largest part of the data.

The performance results of the algorithms during our experiments can be viewed in Tables 10-15. The added communication step for PBP-1 resulted in query times that were approximately 4-5 times slower than those of CRP during random one-to-one queries and around 10 times slower for 100x100. The requests handled by subprocesses, where the actual calculations are performed, have durations close to query times of regular CRP. This was expected, as the subprocesses basically perform CRP on their subgraph. This also implies that for queries with source and target on one process's subgraph, PBP-1 achieves query times similar to those of regular CRP, which also explains the peak around the 30 ms mark in Figure 13. Furthermore, we see that the performance does not get much worse during peaks in the number of requests: both for the concurrently sent random queries and during the realistic load scenarios, we did not see a big increase in query times. The memory usage of an instance of the orchestrator process is similar to that of CRP, during the random queries as well as during the realistic load scenarios. This shows that receiving and sending the requests, collecting results and combining search spaces (for many-to-many requests) uses quite a lot of memory, as this process does not keep any graph data in memory. The subprocesses use less data than an instance of regular CRP, which is to be expected as it has a smaller graph in memory.

PBP-2 is signi cantly slower than PBP-1. During not so heavy loads, such as the sequentially executed one-to-one queries, query times are around half a second (except on Grerec-2). The algorithm performs more computation steps than regular CRP: in Tables 7 and 11 we can see that the algorithm uses, for example, an average of 8.4 iterations on the Europe map, and the actual calculations during these iterations take around 10 ms on average, while the complete query times of regular CRP on this map were 34 ms on average. The rest of PBP-2's query time is spent on communication and synchronizing. This suggests that the cost per message and per communication step (communication cost g and latency l in the BSP model) are not negligible and play an important role in the algorithm's run time. Interestingly, during the sequential many-to-many queries, PBP-2's query times approximated the query times of PBP-1 on the real maps. However, when the load increases, things go wrong quickly for PBP-2. During the concurrently executed queries, query times become around three times as bad and even exceed a second. The memory usage of the orchestrator stayed relatively low, probably due to the small message sizes it needs to handle, but memory usage is high for the subprocesses. We expect this to be caused by the fact they have to maintain the state of many queries at once, especially during high loads.

We also expect this to be the main reason PBP-2 needed an explosive number of instances for its subprocesses, already during the concurrent random queries, but especially during the realistic load scenarios.

For ODL, we see big differences in performance during the random queries on the different maps. For the USA map, with the smallest cell sizes (see Table 5), the query times were approximately three times as fast, for both the one-to-one and the 100x100 queries, compared to the Grerec-2 map with the largest cell sizes. This makes sense, as for these random queries the algorithm loads the top level cells relatively often, which are by far the largest in size. The algorithm needs a couple of hundred milliseconds per one-to-one query, significantly more than PBP-1 or regular CRP. This can all be attributed to the time it takes to load the necessary graph data, as that is the only added part compared to regular CRP. Furthermore, ODL's query times deteriorated when the load increased. We expect this to happen due to the database process not scaling as the other processes do: we always have only one instance. Therefore, when the maximum bandwidth of communication with this process is reached, the orchestrator simply has to wait for the data of the previous query to be done loading before it is able to load new data. One advantage we did see for ODL is the memory usage: it rarely exceeded average memory use of 1 gigabyte per instance.

Scalability The three algorithms all needed less time to start a new instance than CRP (see Table 9). The subprocesses of the PBP algorithms, which load approximately a tenth of the entire graph, achieved an average reduction in startup time of approximately 18% compared to the regular CRP implementation. Starting a new instance of ODL is almost instantaneous, as it does not have to preload any data. In that sense, it has the ability to handle high work loads well. However, during the experiments it seems that a lot depends on the database process, as the performance decreased to such an extent during high loads to be unacceptable in most applications. During the realistic load experiments, in which queries had relatively short distances, we saw similar signs: query durations were quite good for one-to-one queries for which relatively little data was needed, but were bad for the many-to-many queries that require more data.

PBP-1 did not need many instances to handle requests during our experiments, even during higher loads, compared to the other algorithms. It has the additional advantage that it can scale certain specific subprocesses that experience high work loads, as we saw during the realistic load scenarios, where only two of the ten subprocesses were active. Regular CRP does not have this flexibility.

PBP-2 seems not to scale well. As we mentioned, we saw performance decreasing drastically during the high loads of our experiments with concurrently sent random queries. This also causes the algorithm to need lots of instances, to the point where it even failed during the realistic scenarios. The algorithm also does not share the advantage with PBP-1 of being able to scale specific parts of the graph, as processes that do not contain source or target are still active during a query.

Distribution of work load The work load for each subprocess of PBP-1 is completely determined by the number of sources or targets its subgraph contains. This can be seen in Figure 15, where the work load per process is proportional to the number of nodes during the random queries. For PBP-2, we expected some correlation, but Figure 16 does not really show this.

During the realistic load scenarios, which only contained sources and targets in the Netherlands, only two subprocesses of PBP-1 were active, while every process of PBP-2 performed label-correcting steps. As we mentioned, this provides PBP-1 with an advantage in terms of scalability: more instances can be started for the processes experiencing the highest loads, while keeping the number of instances for the processes with fewer requests low.

Impact of query distance The impact of the query distance on the query duration per algorithm is shown in Figure 17. These results reflect the behaviour we expected after our analyses, as we described in Section 3.3: for PBP-1, we see a jump in query duration at the point where the queries cross a top level cell boundary, due to the needed communication step for these queries. For PBP-2, we see a similar jump, but query times already rise more on lower ranks and are significantly slower for local queries compared to PBP-1. The ODL query times rise gradually as the query distance increases, which can be explained by the fact that the query times are dominated by the time it takes to load the necessary graph data. The size of this data increases as the query distance increases, as higher level cells are needed.

Impact of natural cuts The two random road networks we generated following the GREREC model differed in many ways. Grerec-1 had many characteristics that were beneficial for the performance of the algorithms: it had by far the fewest total number of boundary points and shortcuts among all four maps (see Table 2); the smallest average subgraph size for PBP (Table 4) and relatively small cell sizes (Table 5). Grerec-2 on the other hand, looking at the same tables, contained the most shortcuts among all maps, the largest PBP subgraphs and by far the largest cell sizes. The performance of all algorithms on these two maps, therefore differed significantly. Message sizes for Grerec-1 were small, while they were the largest for Grerec-2 (Tables 6-8) and startup times for Grerec-1 were the fastest of all maps, slowest for Grerec-2 (Table 9). Furthermore, Grerec-2 was consistently the map with the worst performance results for every algorithm, including regular CRP, while the algorithms performed relatively well on Grerec-1. The performance of PBP-2 and ODL saw a larger downturn than PBP-1 and CRP, we expect due to their reliance on message sizes and due to the fact that the increase in message size was larger for these two algorithms than for PBP-1. The messages during PBP-2 and ODL at least tripled in size on Grerec-2 compared to other maps (Tables 7 and 8). The difference in message size between PBP-1 and PBP-2 may be explained by the fact that we send messages of size $O(B_C^{L-1})$, where $B_C^{L-1} \approx \frac{B}{10}$, for PBP-1 and of size $O(B_C^{L-1})$ during PBP-2, as Grerec-2 on average has more level-5 boundary points per cell (Table 3), but not more total boundary points than the other maps (Table 2). The huge number of shortcuts in higher level cells of Grerec-2 compared to other maps could explain the increased message size for ODL, as it is the only algorithm whose message size is affected by the number of shortcuts. Also, PBP-2 sends a lot more messages than the others, so increased message sizes affects performance more than for other algorithms. These differences in performance indicate that our partitioning algorithm found a convenient partition for Grerec-1, i.e. a partition with few boundary arcs, but it was not able to find such partition for Grerec-2. We also note the impact it had on the performance of regular CRP (query times at least doubled, see Tables 10 and 12), showing the reliance of CRP on this partitioning.

6 Conclusion

In this research, we presented three suggested approaches to using distributed memory for the query stage of CRP in order to improve the scalability: PBP-1, PBP-2 and ODL. Both PBP-1 and ODL show promise to improve scalability, while still being able to achieve performance standards of modern day applications. The method of distributing calculations among the processes of PBP-2 does not seem to scale well, due to its heavy communication between subprocesses and maintaining query states on each subprocess for each query.

PBP-1, with one adjustment to metric-dependent preprocessing, achieves similar query times to CRP when source and target are contained in one process's subgraph. This is especially bene cial during realistic scenarios, in which local queries are typically common. When the query does cross the boundary of such subgraph, queries can still be answered within a couple of hundred milliseconds on maps of continental size. Additionally, we expect we can reduce these query times further with some engineering, which we will discuss in the next section. The time needed to start a new instance is reduced as each process keeps only a subgraph of the entire graph in memory. During our experiments, in which we split the graph in ten parts of roughly equal size, this reduced startup times by approximately 18% compared to the regular CRP implementation. Furthermore, PBP-1 has the advantage of being able to scale speci c processes that contain subgraphs where many queries have their source or target.

PBP-2 does not seem to improve the scalability of CRP. Due to its use of multiple messages during multiple iterations plus its keeping of state per query on each process, the algorithm su ers during heavy work loads, in spite of its small message sizes. During lighter loads it also did not present other clear advantages. Performance could be improved by calculating queries contained in one subgraph using one process only. However, we do not expect PBP-2 to perform or scale better than PBP-1, nor do we see other possibilities to improve the algorithm with further engineering to accomplish that. We therefore conclude that the PBP-2 approach does not suit our desire of a scalable approach to the query stage of CRP.

ODL's ability to instantly start new instances is a big advantage in terms of scalability. The algorithm keeps memory consumption low, as it only needs to keep one query's search graph in memory at once, which enables the use of cheaper servers. However, its performance during our experiments did not satisfy the requirements of interactive applications. The performance relies heavily on the database: for each query using the top level of the overlay graph, the algorithm needs to load multiple megabytes. The bandwidth of the connection to the database, as well as the memory e ciency of the graph data, thus become crucial in order to achieve good performance during heavier loads. Therefore, for ODL to be applicable in realistic scenarios, we either have to nd a way to reduce the amount of necessary graph data per query or to load more graph data in a short amount of time.

This research shows that PBP-1 and ODL are viable methods to improve the scalability of the query stage of CRP. PBP-1 reduces the startup times of new instances and can answer local queries as fast as regular CRP. For longer range queries, still reasonable query times can be achieved. The scalability advantages for ODL are even more signi cant: starting new instances is practically instantaneous. However, satisfying performance standards of real-world applications remains challenging.

6.1 Future work

Future research could be done in several directions. For our implementations, we did not spend a great amount of time on engineering to improve our algorithms as much as possible. Therefore, we expect that there are still areas where improvements could be made. Also, some adjustments could be made to preprocessing or the algorithms itself, which could bene t performance. In this section, we will mention some of these possible improvements which could be explored in future work.

We performed our experiments once and presented their results. We believe that, because of the relatively large size of our query sets and the use of multiple di erent maps, we could con dently present a clear picture of the scalability, performance and resource consumption of the algorithms. However, the algorithms may bene t from extra testing. It may especially be helpful to test algorithms on query scenarios (load distribution, query distance, etc.) speci c to the desired application. This is also the reason why we performed experiments using real customer data of Ortec, to see how the algorithms would perform if Ortec decided to implement these approaches. These realistic scenarios provided great insights. For example, it became clear that ODL's performance was signi cantly better during these scenarios than during random queries.

PBP First, we discuss our choice of distributing the cells among the processes for the PBP algorithms. We opted to use the highest level of the overlay to split the graph: each process received a top level cell and all its subcells, resulting in (for K 10) ten processes. However, one could also choose to use lower levels of the overlay to distribute the cells, or choose a more re ned way of distributing the cells. We believe that in some applications, a smart distribution of cells could improve the scalability. For example, when it is known beforehand that many queries will have their sources and/or targets in a certain area of the map, we could choose to make sure that one process contains such area entirely. This would enable answering many queries without communication to other processes and scaling up only that process during heavy loads.

A di erent area where improvements may be possible is in communication. All communication in our implementations goes through the orchestrator process, which causes messages to be sent twice: once from the sending process to the orchestrator, once from the orchestrator to the receiving process. We therefore could reduce the number of total messages by letting subprocesses communicate directly to each other. One possibility is to let the user send requests directly to the target process, as a query starts on that process. However, in many applications this is not really desirable. An alternative would be to have an orchestrator process still receive the messages, but enabling subprocesses to send and receive messages to and from each other and return the result to the user. So for PBP-1, a query would arrive at the orchestrator, which sends it to the target process. The target process performs its calculations, but instead of sending its search space back to the orchestrator, it sends it directly to the source process. The source process calculates a shortest path and returns the result to the user.

A di erent improvement of PBP would be to be smarter in the choice of which data we send to other processes. In our implementation of PBP-1, we send the entire backward search space from the target process to the source process. However, based on the partition codes of the source and target nodes, we know the query will not use nodes on levels higher than the lowest sharing level $l_{\text{OP}}(s, t_q)$ of the overlay graph (see Section 3.2). In addition, Proposition 3 describes a way to determine the exact cells per level that the search in the opposite direction might use for its search. We could therefore only send the (boundary point, cost)-pairs with boundary points of those cells of levels up to $l_{\text{OP}}(s, t_q)$ to reduce the message size. This ltering works for both PBP-1 and PBP-2.

ODL For ODL, the main reason we could not achieve performance close to the regular CRP implementation is that loading the needed graph data from the database took too long. One way to improve this, is to reduce the amount of graph data needed per query. As we mentioned in Section 4.1, we use cliques to represent the shortcuts between all boundary points of a cell. By using a skeleton graph per cell instead, as is done by Hamme [15], the amount of data per cell would be reduced. The disadvantage of this approach would be that we would need to perform Dijkstra's algorithm on the (very small) skeleton graph each time we want to use a shortcut, which could increase query times.

Increasing the bandwidth or loading speed is another way to achieve better performance. We explained earlier why we do not want to scale the database process as we do for all other processes: we would lose the scalability advantages of ODL. However, we could opt to have a (constant) number of instances of the database process, so graph data could be loaded faster. It remains to be seen if this is a viable option in practice, as services as Redis are quite expensive, but it would increase performance.

Another optimization for some applications would be to keep some parts of the graph in memory continuously. We started ODL queries without any graph data in memory and loaded everything we needed from the database. When we know that certain data is needed for a large number of queries, we could choose to not load this again for every query, but store it on the orchestrator process instead.

A different direction that would be interesting to explore, is that of a combination of the regular CRP implementation, where one process contains the entire graph, and ODL. When starting a new instances of regular CRP, the entire graph has to be loaded. We could opt to load this data in an order determined by the incoming queries. So, when the number of queries increases to the point where we need a new instance, this new instance first loads the graph data it needs for its first received query and keeps that data in memory. Then for the second received query, it loads that search graph, and so on, until it has the entire graph in memory. This way the algorithm can already compute shortest paths, while starting the new instance.

References

[1] Pim Agterberg. Many-to-many customizable route planning with time-dependent driving restrictions. Master's thesis, Utrecht University, 2017.

[2] F. Bai, Narayanan Sadagopan, and A. Helmy. Important: a framework to systematically analyze the impact of mobility on performance of routing protocols for adhoc networks. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*, volume 2, pages 825{835 vol.2, 2003.

[3] Hannah Bast, Daniel Delling, Andrew Goldberg, Dorothea Wagner, and Renato F Werneck. Route planning in transportation networks. *Algorithm Engineering: Selected Results and Surveys*, 9220:19, 2016.

[4] Rob Bisseling. The bulk synchronous parallel model. `https://webspace.science.uu.nl/~bisse101/Book2/psc2_1.1.pdf`, 2020. Parallel Scienti c Computation course slides.

[5] Andre B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd International Workshop on Software and Performance*, WOSP '00, page 195{203, New York, NY, USA, 2000. Association for Computing Machinery.

[6] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338{1355, 2003.

[7] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of Dijkstra's shortest path algorithm. In *International Symposium on Mathematical Foundations of Computer Science*, pages 722{731. Springer, 1998.

[8] Daniel Delling, Andrew V Goldberg, Thomas Pajor, and Renato F Werneck. Customizable Route Planning in road networks. *Transportation Science*, 51(2):566{591, 2017.

[9] Daniel Delling, Andrew V Goldberg, Ilya Razenshteyn, and Renato F Werneck. Graph partitioning with natural cuts. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 1135{1146. IEEE, 2011.

[10] Daniel Delling, Moritz Kobitzsch, and Renato F Werneck. Customizing driving directions with GPUs. In *European Conference on Parallel Processing*, pages 728{739. Springer, 2014.

[11] Cyril Gavoille, David Peleg, Stephane Perennes, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1):85{112, 2004.

[12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using Contraction Hierarchies. *Transportation Science*, 46(3):388{404, 2012.

[13] Stefanie Gerke, Dirk Schlatter, Angelika Steger, and Anusch Taraz. The random planar graph process. *Random Structures & Algorithms*, 32(2):236{261, 2008.

[14] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, volume 5, pages 156{165, 2005.

[15] Janis Hamme. Customizable Route Planning in external memory, 2013. Karlsruhe Institute of Technology, Bachelor's thesis.

[16] 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(2):100{107, 1968.

[17] Moritz Hilger, Ekkehard Köhler, Rolf H Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc- ags. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, 74:41{72, 2009.

[18] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multilevel overlay graphs for shortest-path queries. *Journal of Experimental Algorithmics (JEA)*, 13:2{5, 2009.

[19] IBM Cloud Education. Redis. https://www.ibm.com/cloud/learn/redis/, 2019. Accessed: 09-06-2021.

[20] Sungwon Jung and Sakti Pramanik. An e cient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029{1046, 2002.

[21] Vamsi Kalapala, Vishal Sanwalani, Aaron Clauset, and Cristopher Moore. Scale invariance in road networks. *Physical Review E*, 73(2):026130, 2006.

[22] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing many-to-many shortest paths using Highway Hierarchies. In *Society for Industrial and Applied Mathematics. Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, page 36. Society for Industrial and Applied Mathematics, 2007.

[23] Ulrich Lauther. An extremely fast, exact algorithm for nding shortest paths in static networks with geographical background. *Geoinformation und Mobilität-von der Forschung zur praktischen Anwendung*, 22:219{230, 2004.

[24] Marc Gravell. Stackexchange.Redis. https://stackexchange.github.io/StackExchange.Redis/. Accessed: 09-06-2021.

[25] Adolfo Paolo Masucci, Duncan Smith, Andrew Crooks, and Michael Batty. Random planar graphs and the London street network. *The European Physical Journal B*, 71(2):259{271, 2009.

[26] Memcached. Memcached. https://memcached.org/. Accessed: 09-06-2021.

[27] Ulrich Meyer and Peter Sanders. -stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114{152, 2003.

[28] Microsoft. Azure Functions documentation. https://docs.microsoft.com/en-us/azure/azure-functions/, 2016. Accessed: 09-06-2021.

[29] MongoDB. The database for modern applications. https://www.mongodb.com/. Accessed: 09-06-2021.

[30] Stefano Pallottino. Shortest-path methods: Complexity, interrelations and new propositions. *Networks*, 14(2):257{267, 1984.

[31] Wei Peng, Guohua Dong, Kun Yang, and Jinshu Su. A random road network model and its e ects on topological characteristics of mobile delay-tolerant networks. *IEEE Transactions on mobile Computing*, 13(12):2706{2718, 2013.

[32] Postman. Postman. https://www.postman.com/. Accessed: 09-06-2021.

[33] Redis Labs. Redis. https://redis.io/. Accessed: 09-06-2021.

[34] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933, pages 164{175. Springer, 2013.

[35] Philippe Y.R. Sohouenou, Panayotis Christidis, Aris Christodoulou, Luis A.C. Neves, and Davide Lo Presti. Using a random road graph model to understand road networks robustness to link failures. *International Journal of Critical Infrastructure Protection*, 29:100353, 2020.

[36] Yuxin Tang, Yunquan Zhang, and Hu Chen. A parallel shortest path algorithm based on graph-partitioning and iterative correcting. In *2008 10th IEEE International Conference on High Performance Computing and Communications*, pages 155{161. IEEE, 2008.

[37] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103{111, 1990.

[38] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric containers for e - cient shortest-path computation. *Journal of Experimental Algorithmics (JEA)*, 10:1{3, 2005.

A Algorithms

Algorithm 1: Dijkstra's algorithm (bidirectional)

```
Result: distance d(s,t)
Input : graph G = (V, E), nodes s and t

Create priority queues Q_f, Q_b;
Create dictionaries X_f, X_b;
foreach v ∈ V \ {s, t} do
    Add (v, ∞) to Q_f and Q_b;
Add (s, 0) to Q_f;
Add (t, 0) to Q_b;
μ ← ∞;
while Q_f and Q_b are not empty do
    u ← arg min_{v∈Q_f} D(s, v);
    w ← arg min_{v∈Q_b} D(v, t);
    Remove u from Q_f;
    Remove w from Q_b;
    Add (u, D(s, u)) to X_f;
    Add (w, D(w, t)) to X_b;
    if D(s, u) + D(u, t) ≥ μ and D(w, t) + D(s, w) ≥ μ then
        break;
    if u ∈ X_b and D(s, u) + D(u, t) < μ then
        μ ← D(s, u) + D(u, t)
    if w ∈ X_f and D(s, w) + D(w, t) < μ then
        μ ← D(s, w) + D(w, t)
    foreach (u, v) ∈ E do
        newDistance ← D(s, u) + d(u, v);
        if newDistance < D(s, v) then
            D(s, v) ← newDistance ;
    foreach (v, w) ∈ E do
        newDistance ← d(v, w) + D(w, t);
        if newDistance < D(v, t) then
            D(v, t) ← newDistance ;

Output:
```

Algorithm 1: The bidirectional version of Dijkstra's algorithm for one-to-one queries. The forward priority queue Q_f maintains entries $(v, D(s, v))$ for all nodes v during the forward search, while dictionary X_f keeps track of the nodes that are dequeued and thus have optimal distance. Queue Q_b (with entries $(v, D(v, t))$) and dictionary X_b have the same role for the backward search. Tentative distance $D(s, t)$ is denoted by μ.

Algorithm 2: ExtractNode

```
Result: updated priority queue Q, updated dictionary X, node u, distance D(u) and u's query level
Input : priority queue Q, dictionary X, preprocessing data
u ← arg min_{v∈Q} D(v);
Remove u from Q;
Add (u, D(u)) to X;
level ← l_st(u);
Output: (Q, X, u, D(u), level)
```

Algorithm 2: A function that is used in the subsequent algorithms. The function extracts a node from the priority queue, adds the node's distance to the dictionary of processed nodes and determines the node's query level.

Algorithm 3: CRP query phase (one-to-one)

```
Result: distance d(s,t)
Input : graph G = (V, E), nodes s and t, preprocessing data
Create priority queues Q_f, Q_b;
Create dictionaries X_f, X_b;
foreach v ∈ V ∪ {s, t} do
    Add (v, ∞) to Q_f and Q_b;
Add (s, 0) to Q_f;
Add (t, 0) to Q_b;
μ ← ∞;
while Q_f or Q_b is not empty do
    (Q_f, X_f, u, D(s,u), forwardLevel) = ExtractNode(Q_f, X_f);
    (Q_b, X_b, w, D(w,t), backwardLevel) = ExtractNode(Q_b, X_b);
    if D(s,u) ≠ ∞ and D(w,t) ≠ ∞ then
        break;
    if u ∈ X_b and D(s,u) + D(u,t) < μ then
        μ ← D(s,u) + D(u,t)
    if w ∈ X_f and D(s,w) + D(w,t) < μ then
        μ ← D(s,w) + D(w,t)
    cell ← C^{forwardLevel}(u);
    foreach (u,v) ∈ E_cell do
        newDistance ← D(s,u) + d(u,v);
        if newDistance < D(s,v) then
            D(s,v) ← newDistance ;
    if backwardLevel = L then
        continue;
    cell ← C^{backwardLevel}(w);
    foreach (v,w) ∈ E_cell do
        newDistance ← d(v,w) + D(w,t);
        if newDistance < D(v,t) then
            D(v,t) ← newDistance ;

Output:
```

Algorithm 3: One-to-one algorithm of CRP. We denote E_C for the edges within cell C plus the edges either exiting or entering the cell, depending on the search direction. In the forward search, E_C includes edges with beginpoint in C and endpoint in another cell. During the backward search E_C includes edges with beginpoint in a different cell and an endpoint in C.

Algorithm 4: CRP query phase (many-to-many)

```
Result: distances (s, t) ∀ s ∈ S, t ∈ T
Input : graph G = (V, E), node sets S and T, preprocessing data

Create priority queues Q_f, Q_b;
Create dictionary X_b;
Create distance matrix ;
foreach t ∈ T do
    foreach v ∈ V(t) do
        Add (v, ∞) to Q_b;
    Add (t, 0) to Q_b;
    while Q_b is not empty do
        w ← arg min_{v∈Q_b} D(v, t);
        Remove w from Q_b;
        Add (w, t, D(w, t)) to X_b;
        level ← l_st(w);
        if level = L then
            continue;
        cell ← C^level(w);
        foreach (v, w) ∈ E_cell do
            newDistance ← d(v, w) + D(w, t);
            if newDistance < D(v, t) then
                D(v, t) ← newDistance;
foreach s ∈ S do
    foreach v ∈ V(s) do
        Add (v, ∞) to Q_f;
    Add (s, 0) to Q_f;
    while Q_f is not empty do
        (Q_f, X_f, u, D(s, u), level) = ExtractNode(Q_f, X_f);
        if (u, t) ∈ X_b for some t ∈ T and D(s, u) + D(u, t) < D(s, t) then
            D(s, t) ← D(s, u) + D(u, t)
        cell ← C^level(u);
        foreach (u, v) ∈ E_cell do
            newDistance ← D(s, u) + d(u, v);
            if newDistance < D(s, v) then
                D(s, v) ← newDistance ;

Output:
```

Algorithm 4: Many-to-many algorithm of CRP. Dictionary X_b keeps track of the entire backward search space, containing entries $(v, t, D(v, t))$ for nodes v and targets t. Distance matrix is a matrix of size $|S| \times |T|$ that stores distances $D(s, t)$ for each source s and target t.

Algorithm 5: PBP-1 (one-to-one)

```
Result: distance d(s,t)
Input : graph G ⟨V, E⟩, nodes s and t, preprocessing data

p(s) ← nodeToProcess(s);
p(t) ← nodeToProcess(t);
Send t to p(t);

Create priority queue Q_b and dictionary X_b;
foreach v ∈ V_{p(t)} ∩ {t} do
    Add ⟨v, ∞⟩ to Q_f and Q_b;
Add ⟨t, 0⟩ to Q_b;
while Q_b is not empty do
    (Q_b, X_b, w, D(w, t), level) = ExtractNode(Q_b, X_b);
    if level ≥ L then
        continue;
    cell ← C^level(w);
    foreach ⟨v, w⟩ ∈ E_cell do
        newDistance ← d(v, w) + D(w, t);
        if newDistance < D(v, t) then
            D(v, t) ← newDistance;
Send X_b to orchestrator;
Send X_b to p(s);

Create priority queue Q_f and dictionary X_f;
foreach v ∈ V_{p(s)} ∩ {s} do
    Add ⟨v, ∞⟩ to Q_f and Q_b;
Add ⟨s, 0⟩ to Q_f;
  ← ∞;
while Q_f is not empty do
    (Q_f, X_f, u, D(s, u), level) = ExtractNode(Q_f, X_f);
    if D(s, u) ≥ then
        break;
    if u ∈ X_b and D(s, u) + D(u, t) < then
        ← D(s, u) + D(u, t)
    cell ← C^level(u);
    foreach ⟨u, v⟩ ∈ E_cell do
        newDistance ← D(s, u) + d(u, v);
        if newDistance < D(s, v) then
            D(s, v) ← newDistance;
Send to orchestrator;
Output:
```

Algorithm 5: One-to-one algorithm of PBP-1. The red-boxed and blue-boxed parts are executed on the target and source process respectively. The rest is executed on the orchestrator process. $V_p \subseteq V$ represents the nodes of the subgraph on process p. Mapping *nodeToProcess*, which we created during (metric-independent) preprocessing, maps each node to the process whose subgraph contains the node.

Algorithm 6: PBP-1 (many-to-many)

```
Result: distances d(s, t) @s ∈ S, t ∈ T
Input : graph G = (V, E), node sets S and T, preprocessing data
foreach process p do
    S_p ← {s ∈ S | nodeToProcess(s) = p};
    T_p ← {t ∈ T | nodeToProcess(t) = p};
    Send T_p to p;
foreach process p in parallel do
    foreach t ∈ T_p do
        foreach v ∈ V \ {t} do
            Add (v, ∞) to Q_b;
        Add (t, 0) to Q_b;
        while Q_b is not empty do
            w ← arg min_{v∈Q_b} D(v, t);
            Remove w from Q_b;
            Add (w, D(w, t)) to X_b(p, q);
            level ← l_st(w);
            if level ≥ L then
                continue;
            cell ← C^level(w);
            foreach (v, w) ∈ E_cell do
                newDistance ← d(v, w) + D(w, t);
                if newDistance < D(v, t) then
                    D(v, t) ← newDistance;
    Send X_b(p, q) to orchestrator;
X_b = ⋃_p X_b(p, q);
foreach process p do
    Send S_p and X_b to p;
foreach process p in parallel do
    Create distance matrix (p, q);
    foreach s ∈ S_p do
        foreach v ∈ V \ {s} do
            Add (v, ∞) to Q_f;
        Add (s, 0) to Q_f;
        while Q_f is not empty do
            (Q_f, X_f, u, D(s, u), level) = ExtractNode(Q_f, X_f);
            if (u, t) ∈ X_b for some t ∈ T and D(s, u) + D(u, t) < (p,q)(s, t) then
                (p,q)(s, t) ← D(s, u) + D(u, t)
            cell ← C^level(w);
            foreach (u, v) ∈ E_cell do
                newDistance ← D(s, u) + d(u, v);
                if newDistance < D(s, v) then
                    D(s, v) ← newDistance;
    Send (p,q) to orchestrator;
(p,q) = ⋃_p (p,q);
Output:
```

Algorithm 6: Many-to-many algorithm of PBP-1. Symbol (p,q) denotes the $|S_p| \times |T|$ sized matrix containing distances $D(s, t)$ for sources $s \in S_p$ and targets $t \in T$ and $\cdot$ denotes the entire $|S| \times |T|$ distance matrix.

Algorithm 7: PBP-2 (one-to-one)

```
Result: distance d(s,t)
Input : graph G = (V, E), nodes s and t, preprocessing data
Create message arrays M_{f;p} and M_{b;p} for each process p;
p(s) ← nodeToProcess(s);
p(t) ← nodeToProcess(t);
  ← ∞ ;
Add (s, 0) to M_{f;p(s)} and (t, 0) to M_{b;p(t)};
while ∃_p M_{f;p} ≠ ∅ ∨ M_{b;p} ≠ ∅ is not empty do
    Send M_{f;p} and M_{b;p} to each process p;
    foreach process p in parallel do
        foreach (v, D^1(s, v)) ∈ M_{f;p} do
            if D^1(s, v) < D(s, v) then
                D(s, v) ← D^1(s, v);
                Add (v, D(s, v)) to Q_f;
                if v ∈ X_f then
                    Update (v, D(s, v)) ← (v, D^1(s, v));
        foreach (v, D^1(v, t)) ∈ M_{b;p} do
            if D^1(v, t) < D(v, t) then
                D(v, t) ← D^1(v, t);
                Add (v, D(v, t)) to Q_b;
                if v ∈ X_b then
                    Update (v, D(v, t)) ← (v, D^1(v, t));
        while Q_f or Q_b is not empty do
            (Q_f, X_f, u, D(s, u), forwardLevel) = ExtractNode(Q_f, X_f);
            (Q_b, X_b, w, D(w, t), backwardLevel) = ExtractNode(Q_b, X_b);
            if u ∈ X_b and D(s, u) + D(u, t) < _p then
                _p ← D(s, u) + D(u, t)
            if w ∈ X_f and D(s, w) + D(w, t) < _p then
                _p ← D(s, w) + D(w, t)
            cell ← C^{forwardLevel}(u);
            foreach (u, v) ∈ E_cell do
                newDistance ← D(s, u) + d(u, v);
                if newDistance < D(s, v) then
                    D(s, v) ← newDistance;
                    if v ∈ process q ≠ P then
                        Add (v, D(s, v)) to M_{f;q(p)}
            cell ← C^{backwardLevel}(w);
            foreach (v, w) ∈ E_cell do
                newDistance ← d(v, w) + D(w, t);
                if newDistance < D(v, t) then
                    D(v, t) ← newDistance ;
                    if v ∈ process q ≠ p then
                        Add (v, D(v, t)) to M_{b;q(p)}
        Send _p and M_{f;q(p)} and M_{b;q(p)} for each q ≠ p to the orchestrator.
    foreach processor p do
        M_{f;p} ← ⋃_{q ≠ p} M_{f;p(q)};
        M_{b;p} ← ⋃_{q ≠ p} M_{b;p(q)};
        if _p < then
            _p
Output:
```

Algorithm 7: One-to-one algorithm of PBP-2. The part of the algorithm in the red box is performed on the subprocesses. $M_{f;p}$ denotes the message that has to be sent to process p for a forward label-correcting step. It contains entries $(v, D(s, v))$ with updated distances for boundary points v of process p. $M_{b;p}$ fulfills the same role for the backward search, consisting of entries $(v, D(v, t))$. We denote $M_{f;p(q)}$ for the message that process q sends to process p, so the complete forward message that process p receives is $\bigcup_{q \neq p} M_{f;p(q)}$. Symbol $_p$ represents shortest distance $D(s, t)$ found on process p.

Algorithm 8: PBP-2 (many-to-many)

```
Result: distances d(s, t) @s ∈ S, t ∈ T
Input : graph G = (V, E), node sets S and T, preprocessing data
foreach process p do
    S_p = {s ∈ S | nodeToProcess(s) = p};
    T_p = {t ∈ T | nodeToProcess(t) = p};
    M_{f,p} = {(s, s, 0) | s ∈ S_p};
    Send T_p to p;
foreach process p in parallel do
    foreach t ∈ T_p do
        foreach v ∈ V \ {t} do
            Add (v, ∞) to Q_b;
        Add (t, 0) to Q_b;
        while Q_b is not empty do
            w = arg min_{v∈Q_b} D(v, t);
            Remove w from Q_b;
            Add (w, t, D(w, t)) to X_b(p);
            level = l_st(w);
            if level ≥ L then
                continue;
            cell = C^level(w);
            foreach (v, w) ∈ E_cell do
                newDistance = d(v, w) + D(w, t);
                if newDistance < D(v, t) then
                    D(v, t) = newDistance;
    synchronize;
while ⋃_p M_{f,p} is not empty do
    Send M_{f,p} to each process p;
    foreach process p in parallel do
        foreach s ∈ S do
            foreach (s, v, D^1(s, v)) ∈ M_{f,p} do
                if D^1(s, v) < D(s, v) then
                    D(s, v) = D^1(s, v);
                    Add (v, D(s, v)) to Q_f(s);
                    if v ∈ X_f(s) then
                        Update (v, D(s, v)) = (v, D^1(s, v));
            while Q_f is not empty do
                (Q_f, X_f, u, D(s, u), level) = ExtractNode(Q_f, X_f);
                if (u, t) ∈ X_b(p) for some t ∈ T and D(s, u) + D(u, t) < p(q(s), t) then
                    p(q(s), t) = D(s, u) + D(u, t)
                cell = C^level(u);
                foreach (u, v) ∈ E_cell do
                    newDistance = D(s, u) + d(u, v);
                    if newDistance < D(s, v) then
                        D(s, v) = newDistance;
                        if v ∈ process q ≠ p then
                            Add (s, v, D(s, v)) to M_{f,q}(p)
        Send M_{f,q}(p) for each q ≠ P to the orchestrator;
    synchronize;
    foreach process p do
        M_{f,p} = ⋃_q M_{f,p}(q);
foreach process p do
    Send termination message to p;
    Receive p(p);
synchronize;
Output: p(p);
```

Algorithm 8: Many-to-many algorithm of PBP-2. The red-boxed part is performed on every target process, the blue-boxed part on every subprocess, the rest of the code is performed on the orchestrator process.