Most Diverse Near-Shortest Paths

Christian Häcker
Johannes Gutenberg University Mainz, Germany
chaecker@students.uni-mainz.de

Panagiotis Bouros
Johannes Gutenberg University Mainz, Germany
bouros@uni-mainz.de

Theodoros Chondrogiannis
University of Konstanz, Germany
theodoros.chondrogiannis@uni.kn

Ernst Althaus
Johannes Gutenberg University Mainz, Germany
ernst.althaus@uni-mainz.de

ABSTRACT

Computing the shortest path in a road network is a fundamental problem that has attracted lots of attention. However, in many real-world scenarios, determining solely the shortest path is not enough as users want to have additional, alternative ways of reaching their destination. In this paper, we investigate a novel variant of alternative routing, termed the k-Most Diverse Near-Shortest Paths (kMDNSP). In contrast to previous work, kMDNSP aims at maximizing the diversity of the recommended paths, while bounding their length based on a user-defined constraint. Our theoretical analysis proves the NP-hardness of the problem at hand. To compute an exact solution to kMDNSP, we present an algorithm which iterates over all paths that abide by the length constraint and generates k-subsets of them as candidate results. Furthermore, in order to achieve scalability, we also design three heuristic algorithms that trade the diversity of the result for performance. Our experimental analysis compares all proposed algorithms in terms of their runtime and the quality of the recommended paths.

CCS CONCEPTS

• **Information systems** → **Geographic information systems**;
• **Mathematics of computing** → Graph algorithms.

KEYWORDS

Alternative routing, Route planning, Path similarity, Near-shortest paths, Path diversification

ACM Reference Format:
Christian Häcker, Panagiotis Bouros, Theodoros Chondrogiannis, and Ernst Althaus. 2021. Most Diverse Near-Shortest Paths. In *29th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '21), November 2–5, 2021, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3474717.3483955

1 INTRODUCTION

Shortest path computation is a fundamental problem in road networks where the length of a path captures, e.g., the overall covered distance or the travel time. In many real-life scenarios though, recommending solely the shortest path is not enough. Contemporary route planning and navigation services recommend multiple and *diverse* paths that might be longer than the shortest path, but have other desirable properties, e.g., less traffic congestion. Another scenario arises in emergency situations such as natural disasters or terrorist attacks, where dissimilar evacuation plans need to be determined. To this end, various approaches have been proposed in the past to determine a set of alternative or diverse paths. The majority of these approaches consider two key factors, namely the *dissimilarity* and the *length* of the recommended paths. The length is used to define a cost for the set of recommended paths that has to be minimized [11, 12, 24], while dissimilarity is treated as a constraint under a user-defined threshold, e.g., the recommended paths must be at least 50% pairwise dissimilar. For this purpose, a number of (dis)similarity measures can be considered [24].

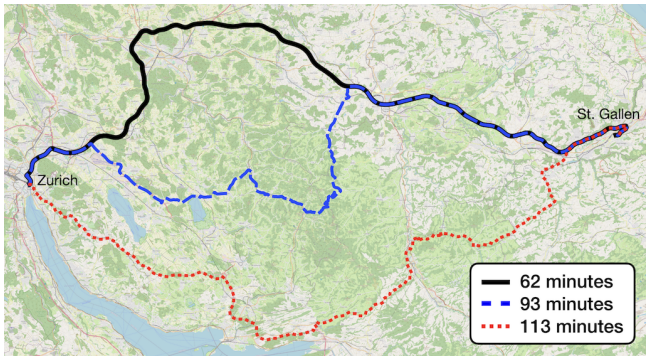

Figure 1: Motivation example

We observe two important shortcomings in existing works. First, expecting the user to properly define a dissimilarity threshold is in most cases counter intuitive. Even if the users are aware of the path similarity measure used, it would be hard to understand the semantics of a 50% threshold for example, or the difference to a 40% one. Second, to abide by the dissimilarity constraint, a method might end up recommending paths that are too long. For instance, Figure 1 shows three paths between Zurich and St. Gallen in Switzerland and their length in terms of travel time. Assume that only two of these paths will be displayed to the user. In scenarios such as routing electric vehicles, where the recommended paths need to abide by energy consumption constraints, a result containing the black (solid line) and the blue path (dashed line) is preferred over a result containing the black and the red path (dotted line). While the black and the red path are clearly dissimilar, the red path is too long in comparison to the other alternatives.

SIGSPATIAL '21, November 2–5, 2021, Beijing, China

ACM ISBN 978-1-4503-8664-7/21/11.
https://doi.org/10.1145/3474717.3483955

In view of the aforementioned shortcomings, we introduce in this paper a novel variant of alternative routing, termed the *k-Most Diverse Near-Shortest Paths* (*k*MDNSP) problem. First, we consider path length under a user-defined constraint, i.e., the user requests the length of the recommended paths not to exceed a given threshold. Such a length constraint allows us to guarantee the quality of the result, and can be easily defined from a user perspective, e.g., with respect to the shortest path. This path-finding task is known as the *near-shortest* path problem [6, 7]. Second, we consider path dissimilarity as part of the optimization objective, i.e., to recommend the set of paths with the highest diversity, defined as the lowest pairwise dissimilarity among the recommended paths. Note that our problem bears some resemblance to the well-studied problem of result diversification [15, 29–31]. However, typical result diversification methods are not applicable in the context of routing problems. The key difference is that these methods expect the entire space of the objects to be given in advance. For routing problems, precomputing and storing all possible paths between every pair of nodes is unrealistic, even for small networks.

The contributions of this work can be summarized as follows:

- We introduce the problem of identifying the *k-Most Diverse Near-Shortest Paths* (*k*MDNSP) as a novel instance of alternative routing (Section 3).
- We conduct a theoretical analysis to prove the hardness of the problem at hand; specifically, for $k = 2$, *k*MDNSP is *weakly NP*-hard, while for arbitrary values of k, the problem is *strongly NP*-hard (Section 3).
- We investigate the exact computation of *k*MDNSP. We present the *EXACT* algorithm which first computes the set of all near-shortest paths, and then generates k-subsets of these paths using a binomial tree [20] while pruning unpromising subsets (Section 4).
- As *EXACT* is not applicable to real-world networks due to the prohibitively large number of near-shortest paths, we design the *SSVP* and *PENALTY* heuristic algorithms which build upon the concepts of *simple single-via* paths and *penalty-based* routing, respectively, to reduce the number of examined paths (Sections 5.1– 5.2).
- Furthermore, we devise an additional heuristic approach that *incrementally* constructs the result set, completely avoiding the generation of k-subset candidates performed by both *SSVP* and *PENALTY* (Section 5.3).

In Section 7, we present the results of our extensive experimental analysis on real-world road networks. Our tests showed that the computation of *k*MDNSP using *EXACT* is not practical even for small networks, while *DIRECT* and *PENALTY* are able to scale to large networks, but with different properties. *DIRECT* trades the result quality for performance while *PENALTY* aims at recommending the most diverse set of paths, possible. Finally, concluding remarks and directions for future work are given in Section 8.

2 RELATED WORK

In what follows, we overview existing works that tackle alternative routing from different angles. Note that a recent qualitative comparison of alternative routing definitions can be found in [23].

Liu et al. [24] introduced the *k-Dissimilar Paths with Minimum Collective Length* (*k*DPwML) problem, which was further investigated in [11]. A *k*DPwML query computes the set of k sufficiently dissimilar paths w.r.t. a similarity threshold θ, that exhibits the lowest collective path length among all sets of k sufficiently dissimilar paths. Another formal definition of alternative routing is the *k-Shortest Path with Limited Overlap* (*k*SPwLO) problem [9]. In contrast to the *k*DPwML which aims at minimizing the collective length of the result paths, a *k*SPwLO query aims at computing dissimilar paths while minimizing the length of each subsequent result. Since both problems are hard, i.e., *k*DPwML is strongly *NP*-hard and *k*SPwLO is weakly *NP*-hard, various heuristic algorithms have been proposed [10–12]. In contrast to our work, both *k*DPwML and *k*SPwLO aim at optimizing either the collective or the individual length of the alternative paths, and treat path similarity as a constraint, i.e., a user-defined threshold is required.

To the best of our knowledge, the only existing work that treats path similarity as an optimization criterion is by Cheng et al. [8]. Specifically, this work tackles the problem of finding *k-Diversified Shortest Paths*, i.e., a set of k simple paths from a source to a target node such that (1) the collective length of the paths in the result set is minimum, and (2) the similarity of the paths is minimum. However, this problem definition comes with two important shortcomings. First, since both path length and similarity are used as optimization criteria, it is possible that the recommended paths are either too long (minimizing similarity) or too similar (minimizing length). Second, similar to other approaches that involve multi-criteria optimization [13, 21, 26], the *k-Diversified Shortest Paths* problem may not have a unique solution. Besides incurring a high computational cost, the final result is determined in a post-processing phase where the trade-off between the total length of the paths and the diversity of the result is considered.

Apart from the above approaches, various methods adopt an iterative approach to compute alternative paths. For instance, penalty-based methods [3, 8, 18, 28] first introduce a penalty on the weights of the edges and then compute the paths by repeatedly running a shortest path algorithm on the input road network. More specifically, before each run, the weights of the edges on already computed paths are increased by a fixed value (penalty) thus prioritizing the expansion of edges that are not on those paths. In Section 5, we discuss how a penalty-based approach can be used for *k*MDNSP. In the same context, Jeong et al. [17] presented an approach to compute alternative paths by imposing a limit on both the length and the similarity of paths. At each round, the proposed algorithm alters the last path added to the tentative result set to obtain a set of candidate paths, and adds to the result the most dissimilar path to the already found ones. Peng at al. [27] proposed a similar method to compute hop-constrained dissimilar paths in web graphs.

Another approach for alternative routing is to first compute a large set of candidate paths and then evaluate these paths with respect to some predefined objective criteria to determine the final result. The Plateau method [1, 22] builds two shortest path trees, one from the source and one from the target, and looks for paths that appear in both trees simultaneously, termed plateaus. A similar idea is formally captured by the alternative graph, i.e., a subgraph of the original network that contains many promising alternative paths [4]. Abraham et al. [2] introduced the notion of single-via

paths. Given a source s and a target t, the single-via path of a node n is defined by the concatenation of the shortest path from s to n and the shortest path from n to t. The authors also propose to evaluate each single-via path against a set of user-defined constraints, i.e., length, local optimality and stretch. In Section 5.1, we present how kMDNSP can benefit from an extension to the single-via paths [11].

Last, there also exist methods that, in contrast to our work, utilize additional information about the network to recommend paths that can be seen as alternative routes. For instance, pareto-optimal paths or the route skyline [13, 21, 26] can be directly seen as alternative routes or can be further examined in a post-processing phase to provide the final result.

3 NOTATION AND PROBLEM DEFINITION

We model a road network $G = (N, E)$ as a directed weighted graph with a set of nodes N and a set of edges $E \subseteq N \times N$. Every edge $e = (n_i, n_j) \in E$ is assigned a positive weight $w(e)$ or $w(n_i, n_j)$, which captures the cost of moving from node n_i to node n_j. A (simple) path $p(s \rightsquigarrow t)$ from a source node s to a target t is a connected and cycle-free sequence of edges $\langle (s, n_i), \ldots, (n_j, t) \rangle$. The length $\ell(p)$ of a path p is the sum of the weights of all contained edges, i.e.,

$$\ell(p) = \sum_{\forall (n_i, n_j) \in p} w(n_i, n_j) \tag{1}$$

The *shortest* path $p_s(s \rightsquigarrow t)$ has the lowest length among all paths from node s to t. Further, a path $p(s \rightsquigarrow t)$ is called *near-shortest* if its length is within a $1 + \epsilon$ factor of the $p_s(s \rightsquigarrow t)$ shortest path length, i.e., $\ell(p) \leq (1 + \epsilon) \cdot \ell(p_s)$, where $\epsilon \geq 0$ is a user-defined parameter.

Let p, p' be two paths from node s to t. We define their dissimilarity based on the Jaccard coefficient (similar to [11, 24]), i.e.,

$$Dis(p, p') = 1 - \frac{\sum_{\forall (n_i, n_j) \in p \cap p'} w(n_i, n_j)}{\sum_{\forall (n_i, n_j) \in p \cup p'} w(n_i, n_j)} \tag{2}$$

We also define the *diversity* of a set of paths P as the lowest pairwise dissimilarity among the contained paths, i.e.,

$$Div(P) = \min_{\forall p, p' \in P} Dis(p, p') \tag{3}$$

We now formally define the problem of finding the *most diverse near-shortest paths*.

Problem 1 (kMDNSP). *Given a road network $G = (N, E)$, a source s and a target t, both in N, a requested number of paths k, and a length constraint threshold $\epsilon \geq 0$, find the $P_{k\text{MDNSP}}$ set of k paths from s to t, such that:*

(A) all paths in $P_{k\text{MDNSP}}$ are near-shortest, with respect to the shortest path $p_s(s \rightsquigarrow t)$, i.e.,

$$\forall p \in P_{kMDNSP} : \ell(p) \leq (1 + \epsilon) \cdot \ell(p_s)$$

(B) $P_{k\text{MDNSP}}$ has the highest diversity among every subset of k paths P_A that satisfy Condition A, i.e.,

$$P_{kMDNSP} = \arg\max_{\forall P \subseteq P_A} \{Div(P)\}, \ with \ |P| = k$$

Example 3.1. Consider the road network G in Figure 2 and the query kMDNSP(G, s, t, k=3, ϵ=0.7). The $p_s(s \rightsquigarrow t) = \langle (s, n_2), (n_2, t) \rangle$ shortest path has a length of 35; hence, the length of a recommended near-shortest path cannot exceed the $(1 + \epsilon) \cdot \ell(p_s) = 59$ threshold. Besides p_s, the paths that abide by this constraint are $p_1 = \langle (s, n_2), (n_2, n_4), (n_4, t) \rangle$ with $\ell(p_1) = 40$, $p_2 = \langle (s, n_1), (n_1, n_3), (n_3, t) \rangle$ with $\ell(p_2) = 46$, $p_3 = \langle (s, n_1), (n_1, n_3), (n_3, n_4), (n_4, t) \rangle$ with $\ell(p_3) = 46$ and $p_4 = \langle (s, n_1), (n_1, n_2), (n_2, t) \rangle$ with $\ell(p_4) = 55$. We apply Formulas 2 and 3 to compute the diversity of all distinct sets that contain $k = 3$ out of these paths. The answer to the kMDNSP query is $P_{k\text{MDNSP}} = \{p_1, p_2, p_4\}$ with $Div(P_{k\text{MDNSP}}) = \min\{Dis(p_1, p_2), Dis(p_1, p_4), Dis(p_2, p_4)\} = 0.89$.

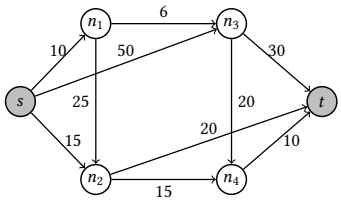

Figure 2: Running example

Finally, we elaborate on the complexity of the kMDNSP problem.

Theorem 3.2. *The kMDNSP problem is* weakly *NP-hard for $k = 2$ and* strongly *NP-hard if k is part of the input.*

Proof. To prove the first part of the theorem, we make a reduction from the Partition-Problem, a known weakly NP-complete problem. For the second part of the theorem, we make a reduction from the Disjoint-Path-Problem, a known strongly NP-hard problem. The full proof is available in the appendix. □

4 AN EXACT APPROACH

A naïve approach for kMDNSP would first construct *all* possible paths from source s to target t and filter out those that violate Condition A in Problem 1. Then, it would examine *all* possible k-subsets of near-shortest paths to find the one that satisfies Condition B. Such an approach is clearly impractical. In view of this, we present an exact approach which directly computes the set of near-shortest paths and efficiently generates only promising k-subsets.

4.1 The *EXACT* Algorithm

Algorithm 1 illustrates a high-level pseudocode of *EXACT*. The algorithm invokes the `GetNearShortestPaths` function to compute the set of all near-shortest paths P_{NSP} from source s to target t with respect to threshold ϵ (cf. Section 4.2). Between Lines 3 to 7, *EXACT* iterates through the contents of P_{NSP}; let p be the current near-shortest path. The first step is to compute the dissimilarities of p to the rest of the paths in P_{NSP} (Line 4). Then, the algorithm examines the k-subsets of P_{NSP} that contain p as candidate solutions to Problem 1. Their diversity is then compared to the diversity of current $P_{k\text{MDNSP}}$ and the result set is updated, if necessary (Lines 5-7). We elaborate on the computation of these k-subsets in Section 4.3. Finally, the result set $P_{k\text{MDNSP}}$ is returned in Line 8.

4.2 Computing Near-Shortest Paths

As *EXACT* does not examine the near-shortest paths in any particular order, we build upon the path enumeration method from [6, 7] for their computation. Function 1 details the pseudocode of `GetNearShortestPaths`. The key idea is to traverse the network

ALGORITHM 1: *EXACT*

Inputs : road network $G = (N, E)$, source node s, target node t, number of results k, threshold ϵ
Variables : set of near-shortest paths P_{NSP}
Output : set $P_{k\text{MDNSP}}$

1 $P_{k\text{MDNSP}} \leftarrow \emptyset$;
2 $P_{\text{NSP}} \leftarrow$ `GetNearShortestPaths`(G, s, t, ϵ);
3 **foreach** p *in* P_{NSP} **do**
4 **compute** $Dis(p, p'), \forall p' \in P_{\text{NSP}}$; ▷ dissimilarities of p
5 **foreach** $P \subseteq P_{\text{NSP}}$ **with** $p \in P$ **and** $|P| = k$ **do**
6 **if** $Div(P) > Div(P_{k\text{MDNSP}})$ **then**
7 $P_{k\text{MDNSP}} \leftarrow P$; ▷ update result set
8 **return** $P_{k\text{MDNSP}}$;

FUNCTION 1: `GetNearShortestPaths`

Inputs : road network $G = (N, E)$, source node s, target node t, threshold ϵ
Variables : current path as stack S, shortest path tree $T_{N \rightsquigarrow t}$ from all nodes in N to t, maximum allowed path length $\mathcal{L}_{max}$
Output : set P_{NSP} of all near-shortest paths from s to t

1 $T_{N \rightsquigarrow t} \leftarrow$ `ComputeShortestPathsRev`(G, t); ▷ all sp's to t
2 $p_s(s \rightsquigarrow t) \leftarrow$ `GetShortestPath`$(T_{N \rightsquigarrow t}, s)$;
3 $\mathcal{L}_{max} \leftarrow (1 + \epsilon) \cdot \ell(p_s(s \rightsquigarrow t))$; ▷ set length constraint
4 $P_{\text{NSP}} \leftarrow \{p_s\}$; ▷ initialize result set with p_s
5 S.push(s); ▷ initialize current path
6 **mark** s as *visited*; ▷ to avoid cycles
7 **while** S *is not empty* **do**
8 $n \leftarrow S$.top(); ▷ last node in current path
9 $(n, n') \leftarrow$ `GetNextEdge`(G, n);
10 **if** $(n, n') \neq$ *null* **then**
11 $p_s(n' \rightsquigarrow t) \leftarrow$ `GetShortestPath`$(T_{N \rightsquigarrow t}, n')$;
12 **if** n' is *unvisited* **and** $\ell(S) + w(n, n') + \ell(p_s(n' \rightsquigarrow t)) \leq \mathcal{L}_{max}$ **then**
13 **if** $n' = t$ **then** ▷ new near-shortest path found
14 **add** $S \bigcup \{t\}$ to P_{NSP};
15 **else**
16 S.push(n'); ▷ extend current path
17 **mark** n' as *visited*;
18 **else**
19 S.pop(n); ▷ all edges of n examined
20 **mark** n as *unvisited*; ▷ reset n's status
21 **return** P_{NSP};

in a depth-first fashion, filtering out paths that violate the near-shortest path constraint. At each stage, a single path is maintained by the function inside stack S, while every node in this path is marked as *visited* (or *unvisited*) to avoid cycles.

At each iteration in Lines 7–20, `GetNearShortestPaths` retrieves the last node n in the current path (i.e., the top of S) and considers its next unused outgoing edge to extend the current path (Line 9). If such an edge does not exist, i.e., n does not have an outgoing edge or all its edges are already considered, the node is removed from S (Line 19). In addition, the status of n is reset (Line 20), so that the node and all its outgoing edges can be reused in future iterations. Otherwise, assume that edge (n, n') is used to extend the current path. Function `GetNearShortestPaths` checks two conditions in Line 12 to complete this extension. First, to avoid cycles, node n' must be marked as *unvisited*, i.e., S should not currently contain the node. Second, the length of new path to be created should not exceed the maximum allowed length as defined based on the ϵ threshold, i.e., $\mathcal{L}_{max} = (1 + \epsilon) \cdot \ell(p_s(s \rightsquigarrow t))$. Following from previous work [7, 12], we further enhance this pruning by estimating a lower bound for the length of every path which extends $S \bigcup \{n'\}$ to reach target t. For this purpose, `GetNearShortestPaths` executes in Line 1 a shortest path algorithm (e.g., Dijkstra's algorithm [14]) that traverses the network starting from node t considering the direction of the edges reversed. The shortest path algorithm computes all shortest paths $p_s(n \rightsquigarrow t)$ for every node $n \in N$ (including $p_s(s \rightsquigarrow t)$) and stores them in the shortest path tree $T_{N \rightsquigarrow t}$. With $T_{N \rightsquigarrow t}$, `GetNearShortestPaths` checks in Line 12 if the length of the *shortest* extension of the new path $S \bigcup \{n'\}$ to target t would be a near-shortest path. If both conditions in Line 12 are met, the function either constructs the new path by pushing n' to stack S (Lines 16–17) or appends $S \bigcup \{t\}$ to P_{NSP} when n' is in fact target t. Otherwise, all possible extensions of $S \bigcup \{n'\}$ to the target would also violate the near-shortest path constraint, and thus, `GetNearShortestPaths` ignores edge (n, n').

4.3 Generating Candidate k-Subsets

To generate candidate k-Subsets of paths, we employ a dynamic programming scheme called "filling a rucksack" (or Algorithm F for simplicity), from [20]. Algorithm F builds upon the concept of the binomial tree and constructs path sets by reusing already generated smaller subsets. More specifically, a binomial tree T of height k is incrementally built to represent all subsets $P \subseteq P_{\text{NSP}}$ of cardinality $|P| \leq k$. For every near-shortest path p examined by *EXACT* (Algorithm 1, Line 3), Algorithm F extends all existing subsets in T of cardinality up to $k - 1$ by adding the new path. Essentially, a new branch is attached to the root of the tree using an edge labeled by p, and the subtree representing subsets of cardinality up to $k - 1$ is copied under this new branch.

We enhance the above expansion process using the following pruning idea. Intuitively, when a new path p is added to a subset P, the diversity of the set can only decrease, i.e., $Div(P \bigcup \{p\}) \leq Div(P)$; this monotonicity directly follows from Formula 3 and the definition of path set diversity. Therefore, when we copy the subtree that represents all subsets of cardinality up to $k - 1$ before adding p, we use the diversity $Div(P_{k\text{MDNSP}})$ of the current result set to prune unpromising subsets, i.e., those with a diversity below or equal to $Div(P_{k\text{MDNSP}})$. Finally, to amplify the effect of this pruning, we consider the subsets in the binomial tree in decreasing order of their cardinality, i.e., starting with the $(k - 1)$-subsets. This examination order allows us to faster improve the $Div(P_{k\text{MDNSP}})$ bound and discard unpromising subsets earlier.

Example 4.1. Consider Figure 2. Without loss of generality, assume that *EXACT* examines the near-shortest paths in the following order: $p_2 = \langle (s, n_1), (n_1, n_3), (n_3, t) \rangle$, $p_4 = \langle (s, n_1), (n_1, n_2), (n_2, t) \rangle$, $p_3 = \langle (s, n_1), (n_1, n_3), (n_3, n_4), (n_4, t) \rangle$, $p_s = \langle (s, n_2), (n_2, t) \rangle$, and $p_1 = \langle (s, n_2), (n_2, n_4), (n_4, t) \rangle$. Figure 3 illustrates two stages in the binomial tree construction; under each path subset, we show its diversity. Note that the height of both trees is limited to $k = 3$. The tree at the top is constructed after examining the first three near-shortest paths. Subset $\{p_3, p_4, p_2\}$ contains exactly k paths and thus, acts as the current (tentative) solution with $Div(P_{k\text{MDNSP}}) = 0.79$.

In the next step, path p_s is examined, which produces the binomial tree at the bottom. The bold lines indicate the new edges

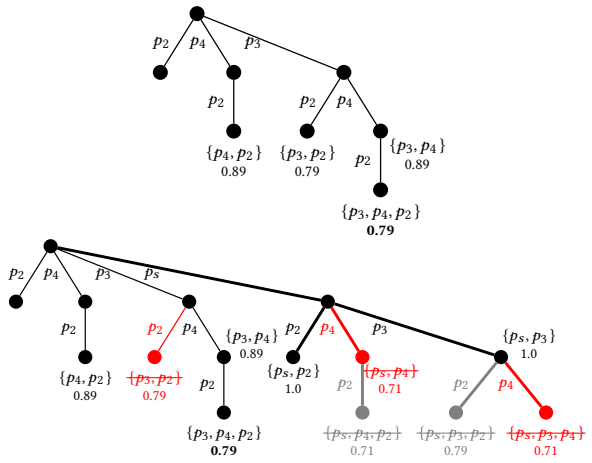

Figure 3: Binomial trees for subsets of paths with cardinality up to $k = 3$ in our running example from Figure 2.

to be added for constructing subsets that include p_s. In addition, we color in red every pruned edge/subset, while a light gray color indicates an edge/subset never constructed. We elaborate on these cases. Subset $\{p_s, p_3, p_4\}$ is constructed but pruned as its diversity 0.71 does not exceed $Div(P_{k\text{MDNSP}}) = 0.79$. In the same manner, subset $\{p_s, p_4\}$ with diversity equal to 0.71 is pruned from further extension, which also means that its $\{p_s, p_4, p_2\}$ extension is never constructed. Last, $\{p_3, p_2\}$ is pruned during the copy process because its diversity does not exceed $Div(P_{k\text{MDNSP}})$, and so the $\{p_s, p_3, p_2\}$ subset is never constructed.

4.4 Complexity Analysis

Finally, we show that the worst-case running time of *EXACT* grows more than exponentially. For the first phase, the algorithm enumerates all possible near-shortest paths. The total number of simple paths from source s to target t in a complete network G is $|P_{\text{NSP}}| = \sum_{r=0}^{|N|-2} \left(\binom{|N|}{r} \cdot r! \right)$. Notice that $(|N|-2)! \leq |P_{\text{NSP}}| \leq 2^{|N|-1}(|N|-2)!$ holds in this case. In the worst case, all paths in G are near-shortest, i.e., if the weight of the edges adjacent to source s is high, e.g. W, while the rest of the edges are short, e.g. 1. Under this premise, the shortest path has length W and the longest $W + |N| - 1$. For the second phase, *EXACT* to enumerate in the worst case all subsets containing up to k near-shortest paths, i.e., $\binom{|P_{\text{NSP}}|}{k}$ subsets, where $|P_{\text{NSP}}|$ is the number of near-shortest paths. Overall, we have a worst-case running time between $((|N|-2)!)^k$ and $(2^{|N|-1} \cdot (|N|-2)!)^k$, resulting in a $2^{\Theta(k|N|\log|N|)}$ complexity, using $n! \in 2^{\Theta(n \log n)}$.

5 HEURISTIC APPROACHES

Our complexity analysis in Section 4.4 showed that the cost of *EXACT* is prohibitively high due to the large number of near-shortest paths; in real-world networks, we expect this number to be exponential to the size of the network. In view of this, we next present three heuristic algorithms which trade the quality of the result set (i.e., the degree of diversity) for performance.

FUNCTION 2: `GetNearShortestSSVPaths`

Inputs : road network $G = (N, E)$, source node s, target node t, threshold ϵ
Variables : shortest path tree $T_{s \rightsquigarrow N}$ from s to all nodes in N, shortest path tree $T_{N \rightsquigarrow t}$ from all nodes in N to t, maximum allowed path length $\mathcal{L}_{max}$
Output : set $P_{\text{NS-SSVP}}$ of near-shortest simple single-via paths from s to t

```
1  T_{s⇝N} ← ComputeShortestPaths(G, s);                    ▷ all sp's from s
2  T_{N⇝t} ← ComputeShortestPathsRev(G, t);                 ▷ all sp's to t
3  p_s(s⇝t) ← GetShortestPath(T_{s⇝N}, t);
4  L_max = (1 + ε) · ℓ(p_s(s⇝t));                           ▷ set length constraint
5  foreach n ∈ N \ {s, t} do
6      if n not in p_s then
7          p_s(s⇝n) ← GetShortestPath(T_{s⇝N}, n);
8          p_s(n⇝t) ← GetShortestPath(T_{N⇝t}, n);
9          p_sv(n) ← p_s(s⇝n) ∘ p_s(n⇝t);                 ▷ n's SVP
10         if p_sv(n) is simple then
11             if ℓ(p_sv(n)) ≤ L_max then
12                 add p_sv(n) to P_NS-SSVP;               ▷ p_ssv(n) found
13         else
14             G_1 ← G without the nodes and the edges in p_s(s⇝n);
15             G_2 ← G without the nodes and the edges in p_s(n⇝t);
16             p_1 ← p_s(s⇝n) ∘ ComputeShortestPath(G_1, n, t);
17             p_2 ← ComputeShortestPath(G_2, s, n) ∘ p_s(n⇝t);
18             if ℓ(p_1) ≤ L_max then
19                 add p_1 to P_NS-SSVP;
20             if ℓ(p_2) ≤ L_max then
21                 add p_2 to P_NS-SSVP;
22 return P_NS-SSVP;
```

5.1 The *SSVP* Algorithm

Our first heuristic algorithm builds upon the concept of the *simple single-via paths* (simple SVP or SSVP, for short) introduced in [11], extending the idea of *single-via paths* (SVP) from [2, 25]. In brief, for each network node $n \notin p_s(s \rightsquigarrow t)$, the SSVP $p_{ssv}(n)$ is identical to SVP $p_{sv}(n)$, if the latter is simple. Otherwise, $p_{ssv}(n)$ can be constructed by concatenating either $p_s(s \rightsquigarrow n)$ with the shortest path from n to t that visits no nodes in $p_s(s \rightsquigarrow n)$, or the shortest path from s to n that visits no nodes in $p_s(n \rightsquigarrow t)$ with $p_s(n \rightsquigarrow t)$.

With SSVP's, we can accelerate the construction of the $P_{k\text{MDNSP}}$ paths at the expense of not computing the exact solution to Problem 1. The *SSVP* heuristic algorithm captures this idea. Essentially, the algorithm operates almost identically to *EXACT*. However, instead of computing all near-shortest paths from source node s to target t (set P_{NSP}), *SSVP* computes all near-shortest simple single-via paths that connect s to t (set $P_{\text{NS-SSVP}}$). Note that by definition, we have $|P_{\text{NS-SSVP}}| << |P_{\text{NSP}}|$ and so the search space of *SSVP* is significantly smaller compared to *EXACT*. As a result, the overall cost of generating candidate k-subsets is also reduced.

Finally, we discuss the `GetNearShortestSSVPaths` function that replaces `GetNearShortestPaths` in Algorithm 1 to construct the $P_{\text{NS-SSVP}}$ set. Function 2 illustrates its pseudocode. In Lines 1–2, the function computes two shortest path trees. Tree $T_{N \rightsquigarrow t}$ models the shortest paths from all network nodes N to target t (similar to Function 1, Line 1) while $T_{s \rightsquigarrow N}$, the paths from source s to all nodes in N. At this point, the $p_s(s \rightsquigarrow t)$ shortest path is retrieved from either of the trees, and the length constraint is set based on $\mathcal{L}_{max}$ (Lines 3–4). Then, in Lines 5–21, `GetNearShortestSSVPaths` examines the SVP for all nodes $n \in N \setminus \{s, t\}$ not contained in the $p_s(s \rightsquigarrow t)$ shortest

path (Line 6). Every such $p_{sv}(n)$ path is efficiently constructed by combining the $T_{s\rightsquigarrow N}$ and $T_{N\rightsquigarrow t}$ trees (Lines 7–9). If $p_{sv}(n)$ is simple and its length does not violate the near-shortest path constraint then a new near-shortest SSVP $p_{ssv}(n) = p_{sv}(n)$ is found (Lines 10–12). However, if $p_{sv}(n)$ is not simple, `GetNearShortestSSVPaths` needs to construct $p_{ssv}(n)$ (Lines 14–21) as described in the beginning of the subsection. For this purpose, `GetNearShortestSSVPaths` first virtually constructs the necessary reduced networks $G1, G2 \subseteq G$ (Lines 14–17) and then constructs near-shortest SSVP's p_1, p_2 by combining $T_{s\rightsquigarrow N}$ or $T_{N\rightsquigarrow t}$ with a shortest path computed on the G_1 or G_2 reduced network, respectively (Lines 18–19). To increase the diversity of $P_{\text{NS-SSVP}}$, we use both p_1 and p_2, as long as they abide by the near-shortest path constraint.

Example 5.1. In our running example and the kMDNSP(G, s, t, $k = 3, \epsilon = 0.7$) query, Function 2 constructs three distinct near-shortest SSV paths, compared to the five computed by Function 1 in *EXACT*. The first path is $p_s = \langle (s, n_2), (n_2, t) \rangle$ as the shortest path is by definition an SSVP. In addition, the function will compute an SSVP for each of the n_1, n_3, n_4 nodes which are not contained in p_s. Specifically, we have $p_{ssv}(n_1) = p_s(s\rightsquigarrow n_1) \circ p_s(n_1\rightsquigarrow t) = p_2$, $p_{ssv}(n_3) = p_s(s\rightsquigarrow n_3) \circ p_s(n_3\rightsquigarrow t) = p_2$ and $p_{ssv}(n_4) = p_s(s\rightsquigarrow n_4) \circ p_s(n_4\rightsquigarrow t) = p_1$. Overall, we have $P_{\text{NS-SSVP}} = \{p_s, p_1, p_2\}$. Finally, since $|P_{\text{NS-SSVP}}| = k$, *SSVP* returns $P_{\text{NS-SSVP}}$ as the final result.

5.2 The *PENALTY* Algorithm

Our second heuristic algorithm builds upon the *Iterative Penalty Method* (IPM) [18, 28]. Similar to IPM, our method consists of two components: a function that computes shortest paths (but, extended to check the near-shortest path constraint), and a mechanism that penalizes edges on previously computed paths.

Designing an effective penalty mechanism is a challenging task. On the one hand, a large penalty would typically achieve high path dissimilarities but might prevent the computation of enough near-shortest paths. On the other hand, a small penalty would allow the computation of many near-shortest paths, but would also slow down the query evaluation. In this work, similar to the common practice, we adopt a multiplicative type of penalty. The penalized weight of an edge e is $w'(e) = f \cdot w(e)$, where f is the multiplicative factor of the penalty. However, in contrast to previous works, the value of f is dynamically adjusted to assist the computation of at least k dissimilar near-shortest paths as fast as possible. Assuming $\epsilon \leq 1$, we define the following heuristic based on our tests:

$$f = 2 - m \cdot \frac{2 - (1 + \epsilon)}{2} = 2 - m \cdot \frac{1 - \epsilon}{2} \tag{4}$$

where $m \geq 0$ controls the penalty magnitude. Note that the penalty is always applied on the original edge weights and so $f > 1$ should hold. Essentially, the semantics of our penalty mechanism are the following. The value of m is initially set to 0 but increases by 1 whenever our modified shortest path method fails to return a path. Under this premise, f is initialized to 2 (doubling the edge weights is a common approach in bibliography [3]) but gradually decreases to allow the computation of more near-shortest paths until $f \leq 1$.

Algorithm 2 illustrates the pseudocode of *PENALTY*. Initially, the algorithm calls the `GetNextNearShortestPath` function to compute shortest path $p_s(s\rightsquigarrow t)$. To this end, the function operates on the original non-penalized edge weights. Having computed

ALGORITHM 2: *PENALTY*

Inputs : road network $G = (N, E)$, source node s, target node t, number of results k, threshold ϵ
Variables : set of near-shortest paths P_{NSP}, maximum allowed path length $\mathcal{L}_{max}$, multiplicative factor f, magnitude factor m, modified weight $w'(e)$ for an edge e
Output : set $P_{k\text{MDNSP}}$

1 $P_{k\text{MDNSP}} \leftarrow \emptyset$, $P_{\text{NSP}} \leftarrow \emptyset$, $\mathcal{L}_{max} \leftarrow \infty$; ▷ initialization
2 **foreach** *edge* $e \in E$ **do**
3 $w'(e) \leftarrow w(e)$;
4 $p \leftarrow$ `GetNextNearShortestPath`$(G, s, t, \mathcal{L}_{max})$; ▷ $p = p_s(s\rightsquigarrow t)$
5 $P_{\text{NSP}} \leftarrow P_{\text{NSP}} \bigcup \{p\}$;
6 $\mathcal{L}_{max} \leftarrow (1 + \epsilon) \cdot \ell(p))$; ▷ set length constraint
7 $m \leftarrow 0, f \leftarrow 2$; ▷ initialize penalty factor (Formula 4)
8 **while** $f > 1$ **do**
9 **foreach** $p' \in P_{NSP}$ **do**
10 **foreach** *edge* $e \in p'$ **do**
11 $w'(e) \leftarrow w(e) \cdot f$; ▷ recalculate penalties
12 $p \leftarrow$ `GetNextNearShortestPath`$(G, s, t, \mathcal{L}_{max})$;
13 **if** $p \neq null$ **and** $p \notin P_{NSP}$ **then**
14 $P_{\text{NSP}} \leftarrow P_{\text{NSP}} \bigcup \{p\}$;
15 **compute** $Dis(p, p')$, $\forall p' \in P_{\text{NSP}}$; ▷ p dissimilarities
16 **foreach** $P \subseteq P_{\text{NSP}}$ **with** $p \in P$ **and** $|P| = k$ **do**
17 **if** $Div(P) > Div(P_{k\text{MDNSP}})$ **then**
18 $P_{k\text{MDNSP}} \leftarrow P$; ▷ update result set
19 **else**
20 $m \leftarrow m + 1$, **update** f; ▷ decrease penalty factor (Formula 4)
21 **return** $P_{k\text{MDNSP}}$;

$\ell(p_s)$, *PENALTY* sets the maximum allowed length $\mathcal{L}_{max}$ of near-shortest paths (Line 6). Furthermore in Line 7, the penalty factors are initialized to $m = 0$ and $f = 2$, according to Formula 4. Then, in Lines 8–19, *PENALTY* iteratively examines its near-shortest paths. At each iteration, the algorithm first applies the penalty (Lines 9–11) and then calls `GetNextNearShortestPath` to construct a new near-shortest path, using the penalized edge weights w'. If the function successfully returns a path p (never constructed before), the path is added to P_{NSP} and *PENALTY* applies Algorithm *F* to generate candidate k-subsets P that include p, similar to *EXACT* and *SSVP* (Lines 13–17). The diversity of each subset P is computed using the original weights w. Otherwise, if `GetNextNearShortestPath` fails to produce a new near-shortest path, m is increased by 1 and the penalty factor f is updated (decreased) before the next iteration (Line 19). This iterative process terminates when *PENALTY* cannot further decrease f, i.e., when $f \leq 1$. At this point, a penalty can no longer be enforced, and so, the algorithm terminates returning current $P_{k\text{MDNSP}}$ as the result.

We now briefly discuss `GetNextNearShortestPath`. The function essentially extends a traditional shortest path method (e.g., Dijkstra's algorithm) with two extra features. First, for each node n in the network, two types of distances from source node s are maintained (corresponding to $\ell(p(s\rightsquigarrow n))$); the original distance and the modified distance that is computed using the penalized weights. `GetNextNearShortestPath` visits nodes by their modified distance from s, which enables the function to compute a path dissimilar to the ones already computed. The original distance is used to check the near-shortest path constraint. Intuitively, to guarantee that only near-shortest paths are computed, only nodes within $\mathcal{L}_{max}$ original distance from the source are visited.

Example 5.2. We demonstrate *PENALTY* using our running example and kMDNSP($G, s, t, k = 3, \epsilon = 0.7$). The first path constructed by `GetNextNearShortestPath` and added to P_{NSP} is shortest path $p_s = \langle (s, n_2), (n_2, t) \rangle$. The edges in p_s are penalized by doubling their weight ($f = 2$), i.e., $w'(s, n_2) = 2 \cdot w(s, n_2) = 30$ and $w'(n_2, t) = 2 \cdot w(n_2, t) = 40$. After applying the above penalties, `GetNextNearShortestPath` is called again returning $p_2 = \langle (s, n_1), (n_1, n_3), (n_3, t) \rangle$. Path p_2 is added to P_{NSP} and its edges are penalized, again by doubling their weight, i.e., $w'(s, n_1) = 20$, $w'(n_1, n_3) = 12$, $w'(n_3, t) = 60$. Afterwards, *PENALTY* calls once again the `GetNextNearShortestPath` function which returns $p_1 = \langle (s, n_2), (n_2, n_4), (n_4, t) \rangle$. At this point, the tentative result is $P_{k\text{MDNSP}} = P_{\text{NSP}} = \{p_s, p_2, p_1\}$, with $Div(P_{k\text{MDNSP}}) = 0.75$. In the next iteration, `GetNextNearShortestPath` fails to return a new near-shortest path. As a result, *PENALTY* increases m to 1 and updates $f = 2 - m \cdot \frac{1-\epsilon}{2} = 1.85$. `GetNextNearShortestPath` is called and fails again. In fact, the function continues failing until $m = 7$, which means that $f = 0.95 < 1$. At this point, *PENALTY* terminates and returns $P_{k\text{MDNSP}} = \{p_s, p_2, p_1\}$.

ALGORITHM 3: *DIRECT*

Inputs : road network $G = (N, E)$, source node s, target node t, threshold ϵ
Variables : shortest path tree $T_{N \rightsquigarrow t}$ from all nodes in N to t, maximum allowed path length $\mathcal{L}_{max}$
Output : set $P_{k\text{MDNSP}}$

```
1  T_{N⇝t} ← ComputeShortestPathsRev(G, t);                ▷ all sp's to t
2  p_s(s⇝t) ← GetShortestPath(T_{N⇝t}, s);
3  L_max ← (1 + ε) · ℓ(p_s(s⇝t));                          ▷ set length constraint
4  P_kMDNSP ← {p_s(s⇝t)};                                  ▷ initialize result set
5  P_NSP ← ∅;
6  while |P_kMDNSP| < k do
7      p_last ← the last path added to P_kMDNSP;
8      P_dev ← GetDeviatingSubPaths(G, p_last);
9      foreach p ∈ P_dev do
10         n ← last node in p;
11         p ← p ∘ GetShortestPath(T_{N⇝t}, n);
12         if p is simple and ℓ(p) ≤ L_max then
13             P_NSP ← P_NSP ⋃ {p};
14     p_res ← arg max_{p∈(P_NSP \ P_kMDNSP), p'∈P_kMDNSP} Dis(p, p');
15     P_kMDNSP ← P_kMDNSP ⋃ {p_res};                      ▷ update result set
16 return P_kMDNSP;
```

5.3 The *DIRECT* Algorithm

Despite operating on a smaller search space compared to both *EXACT* and *SSVP*, the *PENALTY* algorithm still resorts to Algorithm *F* in order to compute k-subsets of paths. Our last heuristic algorithm called *DIRECT*, takes on a different approach that is inspired by the method proposed in [17]. Intuitively, the $P_{k\text{MDNSP}}$ result set (initially containing the $p_s(s \rightsquigarrow t)$ shortest path) is built incrementally in $k - 1$ rounds. At each round, $P_{k\text{MDNSP}}$ is updated by appending the near-shortest path with the highest dissimilarity to the previously added paths. As such, *DIRECT* eliminates the need to generate candidate k-subsets.

Algorithm 3 illustrates the pseudocode of *DIRECT*. The algorithm maintains all constructed near-shortest paths inside set P_{NSP}. At each iteration of the while loop in Line 6, *DIRECT* first uses the last path added to $P_{k\text{MDNSP}}$, denoted by $p_{last}(s \rightsquigarrow t)$, in order to construct the set P_{dev} of deviating subpaths from source node s (Lines 7–8). Each subpath $p(s \rightsquigarrow n) \in P_{dev}$ is then extended to reach the target t by the shortest path $p_s(n \rightsquigarrow t)$. To efficiently retrieve the $p_s(n \rightsquigarrow t)$ paths, *DIRECT* utilizes the shortest path tree $T_{N \rightsquigarrow t}$ computed in Line 1. If the now extended p is a simple path that abides by the near-shortest path constraint (conditions in Line 12), the path is appended to P_{NSP}. As the last step, $P_{k\text{MDNSP}}$ is updated in Lines 14–15 by appending the path $p_{res} \in P_{\text{NSP}} \setminus P_{k\text{MDNSP}}$ with the maximum dissimilarity to the paths in $P_{k\text{MDNSP}}$. *DIRECT* terminates after exactly k near-shortest paths are added to $P_{k\text{MDNSP}}$.

Last, we briefly discuss the construction of the deviating subpaths (Line 8). To this end, we iterate over all subpaths $p(s \rightsquigarrow n)$ of p_{last}. For each subpath, we consider all outgoing edges (n, n') of node n excluding the one in p_{last}, and define every deviating subpath $p(s \rightsquigarrow n')$. We further enhance the diversity of the constructed paths by using the outgoing edges of each n' node, i.e., edges (n', n''), to produced deviating subpaths $p(s \rightsquigarrow n'')$ as well. This "double-deviation" strategy increases the cardinality of P_{dev} which helps *DIRECT* to construct at least the requested number k of recommended paths.

Example 5.3. We illustrate *DIRECT* using our running example in Figure 2 and kMDNSP($G, s, t, k = 3, \epsilon = 0.7$); recall that the near-shortest path constraint is based on $\mathcal{L}_{max} = 59$. As the first step, the shortest path $p_s = \langle (s, n_2), (n_2, t) \rangle$ is added to the $P_{k\text{MDNSP}}$ result set. With p_s as p_{last}, the algorithm generates P_{dev}. First, subpath $p(s \rightsquigarrow n_2)$ is considered. By examining the outgoing edges of n_2, excluding (n_2, t) as it lies on p_{last}, subpath $\langle (s, n_2), (n_2, n_4) \rangle$ is constructed and added to P_{dev}. Next, we examine the outgoing edges of n_4 resulting in the construction of subpath $\langle (s, n_2), (n_2, n_4), (n_4, t) \rangle = p_1$ that is also added to P_{dev}. Subsequently, we consider subpath $p(s \rightsquigarrow s)$ of p_{last}. The examination of the outgoing edges of s, excluding (s, n_2), and the outgoing edges of the neighbors of s, excluding n_2, results in the construction of (sub)paths $\langle (s, n_1) \rangle$, $\langle (s, n_1), (n_1, n_2) \rangle$, $\langle (s, n_1), (n_1, n_3) \rangle$ $\langle (s, n_3) \rangle$, $\langle (s, n_3), (n_3, n_4) \rangle$ and $\langle (s, n_3), (n_3, t) \rangle$. The last two $\langle (s, n_3), (n_3, n_4) \rangle$ and $\langle (s, n_3), (n_3, t) \rangle$ are pruned as their length exceed $\mathcal{L}_{max}$. All subpaths in P_{dev} are then extended to reach target t (if necessary), resulting in paths $p_2 = \langle (s, n_1), (n_1, n_3), (n_3, t) \rangle$, $p_4 = \langle (s, n_1), (n_1, n_2), (n_2, t) \rangle$, $\langle (s, n_1), (n_1, n_3), (n_3, t) \rangle$, and $\langle (s, n_3), (n_3, t) \rangle$. At this point, all extended paths whose length exceeds $\mathcal{L}_{max}$ are discarded. Hence, P_{NSP} contains all distinct extended paths that abide by the near-shortest path constraint, i.e., $P_{\text{NSP}} = \{p_s, p_2, p_4, p_1\}$. As $Dis(p_s, p_2) = 1 > Dis(p_s, p_1) = 0.82 > Dis(p_s, p_4) = 0.71$, the algorithm adds p_2 to the result set. *DIRECT* continues in the same manner using p_2 as p_{last} to finally add $p_3 = \langle (s, n_1), (n_1, n_3), (n_3, n_4), (n_4, t) \rangle$ to $P_{k\text{MDNSP}}$. At this point, the algorithm terminates returning $P_{k\text{MDNSP}} = \{p_s, p_2, p_3\}$ with $Div(P_{k\text{MDNSP}}) = 0.79$.

6 EXTENSIONS

In the following, we also discuss possible extensions to kMDNSP.

Path (dis)similarity. Various measures have been proposed in the literature to compute the similarity of two paths (cf. [24]). Choosing the best fitting similarity measure heavily depends on the application and hence is out of the scope of our work. Without loss of generality in Section 3, we used the Jaccard coefficient, but all

Table 1: Road networks tested

road network	# of nodes	# of edges	topology
Adlershof	349	979	City-center
Oldenburg	6,105	14,058	City-center
Porto Alegre	63,751	187,364	Grid-based
Milan	187,537	525,296	Ring-based
Chicago	386,533	1,121,620	Grid-based
Florida	1,070,376	2,712,798	State

the algorithms and techniques presented in Sections 4 and 5 will operate under any arbitrary similarity measure.

Path set diversity. In Section 3, we defined the diversity $Div(P)$ for a set of paths P as the lowest pairwise dissimilarity among the contained paths. Supporting other diversity definitions [31] is an interesting direction for future work. Nevertheless, as a proof of the generality of our analysis, we briefly discuss in the following, an alternative definition to Formula 3 where the diversity is calculated based on the collective dissimilarity of the contained paths:

$$Div(P) = \sum_{\forall p,p' \in P} Dis(p, p') \tag{5}$$

Our methods can be easily applied in this setup, with little (resp. for *EXACT*, *SSVP*, *PENALTY*) to no change (resp. for *DIRECT*). In practice, we only need to adjust the pruning technique in Section 4.3 used by Algorithm F to discard unpromising subsets. Assume that $P_{k\text{MDNSP}}$ contains the current solution to the kMDNSP problem. In contrast to the diversity definition in Formula 3, $Div(P)$ of a subset P under Formula 5 does not decrease after adding a new path. However, Algorithm F can still prune unpromising subsets by computing an upper bound $\overline{Div(P)}$ for their diversity; intuitively, only subsets P with $\overline{Div(P)} > Div(P_{k\text{MDNSP}})$ should be extended. To clarify, consider $k = 3$ and a subset with two paths, $P = \{p_1, p_2\}$. The highest possible diversity for P after adding a third path cannot exceed $\overline{Div(P)} = Dis(p_1, p_2) + 2$, assuming the new path p_3 is disjoint to both p_1 and p_2, i.e., $Dis(p_1, p_3) = Dis(p_2, p_3) = 1$.

7 EXPERIMENTAL ANALYSIS

Our analysis was conducted on a machine with two AMD EPYC 7351 16-Core processors, 512GiB 2666Mhz DDR4 memory, running GNU/Linux 5.4.0-66. All four presented algorithms were implemented in C++, compiled using GNU G++ 9. We experimented with six publicly available real-world road networks [5, 19]. We selected networks with different characteristics and topologies. Table 1 summarizes the characteristics of our tested datasets.

To assess the performance of the algorithms, we measured their average runtime and the average number of near-shortest paths they examine. For this purpose, we ran 1000 queries of randomly selected source-target pairs, while varying the number k of requested paths in $\{2, 3, 4, 5\}$ and the near-shortest path threshold ϵ in $\{0.01, 0.05, 0.1, 0.2, 0.3\}$. In each test, we varied one of the parameters and set the other to its default value, i.e., $k = 3$ and $\epsilon = 0.1$. Note that we enforced a 2 minute timeout for the methods. To assess the quality of the results, we also report the diversity of the result sets, excluding the queries where *EXACT* and *SSVP* timed out. [1]

[1]No timeouts occurred for *PENALTY* and *DIRECT*.

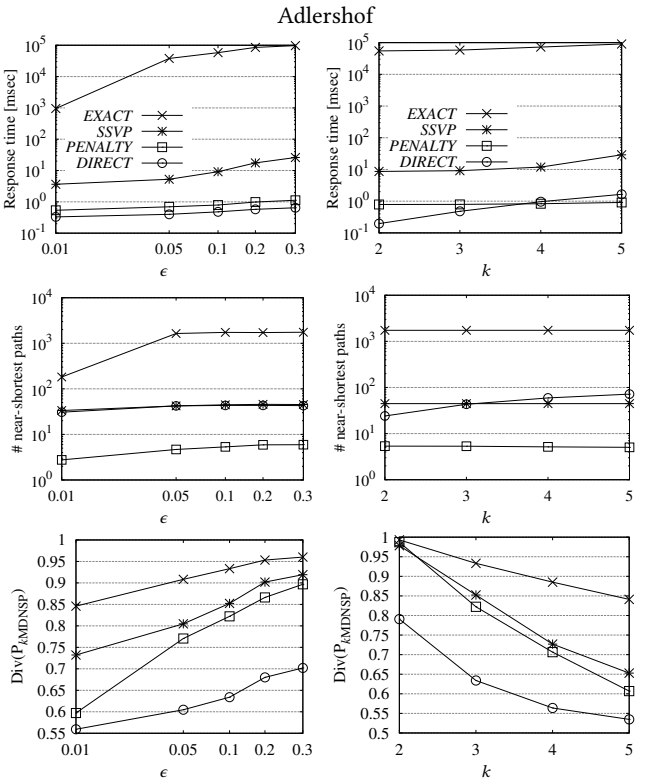

Figure 4: Comparison of all algorithms

7.1 Exact Computation

We first investigate how practical is the exact computation of kMDNSP using *EXACT*. For this purpose, we experimented with the smallest network in Table 1, i.e., Adlershof. In Figure 4, we observe that even for a network with only a few hundreds of nodes, *EXACT* is already several orders of magnitude slower than the heuristic algorithms. In fact, *EXACT* timed out on 50% of the queries on average; specifically for $\epsilon > 0.1$ or $k > 3$, the majority of the queries timed out. The poor performance of *EXACT* is explained by the large number of near-shortest paths it examines. As Figure 4 shows, *EXACT* considers significantly more paths (in the order of thousands) compared to its heuristic competitors, which results in high costs not only for computing and storing these paths in memory, but also for generating the candidate k-subsets.

Figure 4 also reports the quality of the results produced by each algorithm in Aldershof. As expected, *EXACT* is able to report the best $P_{k\text{MDNSP}}$ set of alternative paths, i.e., the set with the highest diversity $Div(P_{k\text{MDNSP}})$. Nevertheless, we also observe that the result diversity produced by the *SSVP* heuristic method is only 10% lower than *EXACT* on average, while taking at least two order of magnitudes less time to answer a query. Overall, our experiments in Adlershof show the limitations of *EXACT*. The computation of kMDNSP using *EXACT* is clearly impractical for real-world networks. As such, for the rest of our analysis we consider only the heuristic algorithms presented in Section 5.

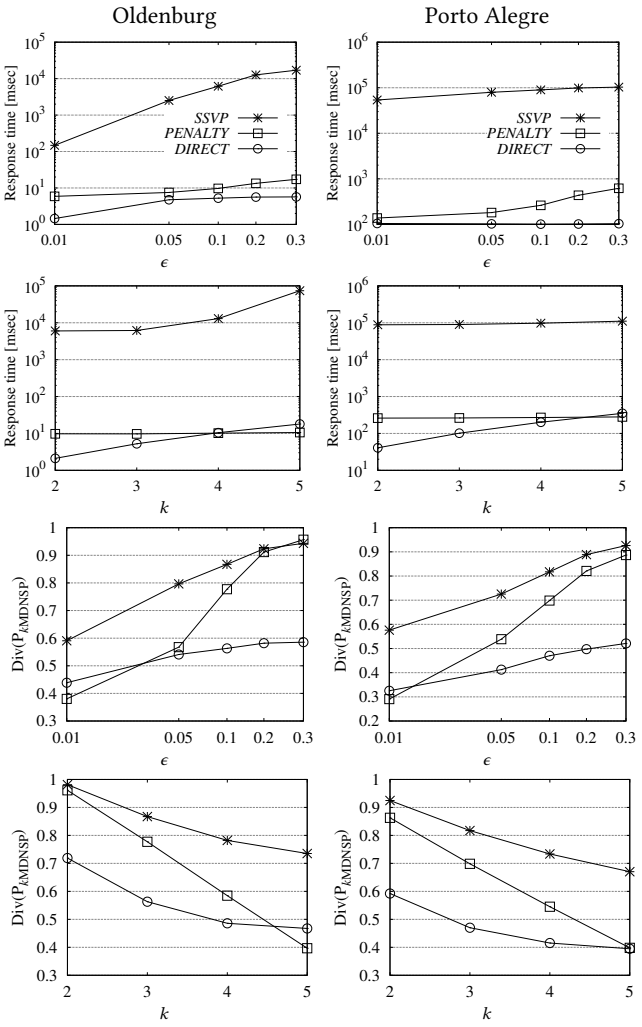

Figure 5: Comparison of heuristics: the case of networks with less than 100k nodes

7.2 Heuristics-based Computation

Figure 5 reports the results for two networks, i.e., Oldenburg and Porto Alegre, with up to 100k nodes. We observe that both *PENALTY* and *DIRECT* always outperform *SSVP* by a wide margin (in some cases, even by three orders of magnitude). The reasons for this performance gap is that *SSVP* not only constructs more near-shortest paths than *PENALTY* but also requires a large amount of calls to Dijkstra's algorithm (Function 2, Lines 16–17). Moreover, we observe that *SSVP* is also severely affected by ϵ due to the increase in the number of near-shortest paths to be computed and examined. Notice how its runtime rises by two orders of magnitude when varying ϵ from 0.01 to 0.3.

While being the slowest algorithm, *SSVP* delivers the best set of alternative paths, followed by *PENALTY*, while *DIRECT* ranks last in almost all cases. All three algorithms are strongly affected by both ϵ and k. More specifically, as ϵ increases, more near-shortest paths are computed enabling the methods to identify better combinations (i.e., with higher diversity) as results. Notice that *PENALTY* benefits the most from the increase in ϵ. This is because larger values of ϵ lead to larger decreases of the multiplicative factor f (see Formula 4). As a result, *PENALTY* is able to almost match the result diversity achieved by *SSVP*, for $\epsilon \geq 0.1$. In contrast, the result quality drops for all algorithms with an increasing k. Essentially, the algorithms do not compute enough near-shortest paths to both cover the extra spots in $P_{k\text{MDNSP}}$ and maximise its diversity at the same time. Note that for *SSVP* and *PENALTY*, the number of near-shortest paths does not increase with k, but only with ϵ. *DIRECT* computes more paths by executing more rounds (*DIRECT* constructs $P_{k\text{MDNSP}}$ in exactly $k-1$ rounds) but this increase is not large enough.

Our results in Figure 5 unveil the limitations of *SSVP*. Essentially, the algorithm scales poorly with both ϵ, k, and the size of the network. In fact for Porto Alegre, *SSVP* timed out in the majority of the queries. As such, we exclude *SSVP* from our experiments on the three largest road networks.

Finally, Figures 6 and 7 report the runtime and result quality of *DIRECT* and *PENALTY* on Milan, Chicago, and Florida. First of all, we observe that both algorithms are able to handle queries in large networks as no timeouts occurred. However, it is also clear that *DIRECT* is always the most efficient method. The key difference is on how the algorithms scale with the test parameters. *PENALTY* scales worse with ϵ because it has to compute significantly more near-shortest paths as ϵ increases, while *DIRECT* scales worse with k as it has to execute more rounds (i.e., $k-1$). With regard to the quality of the results, *PENALTY* delivers in all cases significantly more diverse alternative paths, as it examines a larger number of near-shortest paths.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a novel instance of alternative routing termed the kMDNSP problem. The goal is to recommend the set of k near-shortest paths (based on a user-defined length threshold) with the highest diversity, defined as the lowest pairwise dissimilarity among the recommended paths. Our tests showed that computing kMDNSP with the *EXACT* algorithm is impractical for real-world networks, so we proposed three heuristic algorithms. Our iterative heuristic and penalty-based methods are able to scale to large networks while offering different trade-offs between result quality and performance. For the future, we plan to study kMDNSP under alternative definitions of path diversity and investigate other evaluation approaches, e.g., using flow algorithms.

ACKNOWLEDGMENTS

This work is partially supported by Grant No. CH 2464/1-1 of the Deutsche Forschungsgemeinschaft (DFG).

REFERENCES

[1] 2005. Choice Routing. Cambridge Vehicle Information Technology Ltd.. http://www.camvit.com
[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2013. Alternative routes in road networks. *ACM J. Exp. Algorithmics* 18 (2013).
[3] Vedat Akgün, Erhan Erkut, and Rajan Batta. 2000. On finding dissimilar paths. *Eur. J. Oper. Res.* 121, 2 (2000), 232–246.
[4] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. 2011. Alternative Route Graphs in Road Networks. In *ICST TAPAS*. 21–32.
[5] Thomas Brinkhoff. 2002. A Framework for Generating Network-Based Moving Objects. *GeoInformatica* 6, 2 (2002), 153–180.

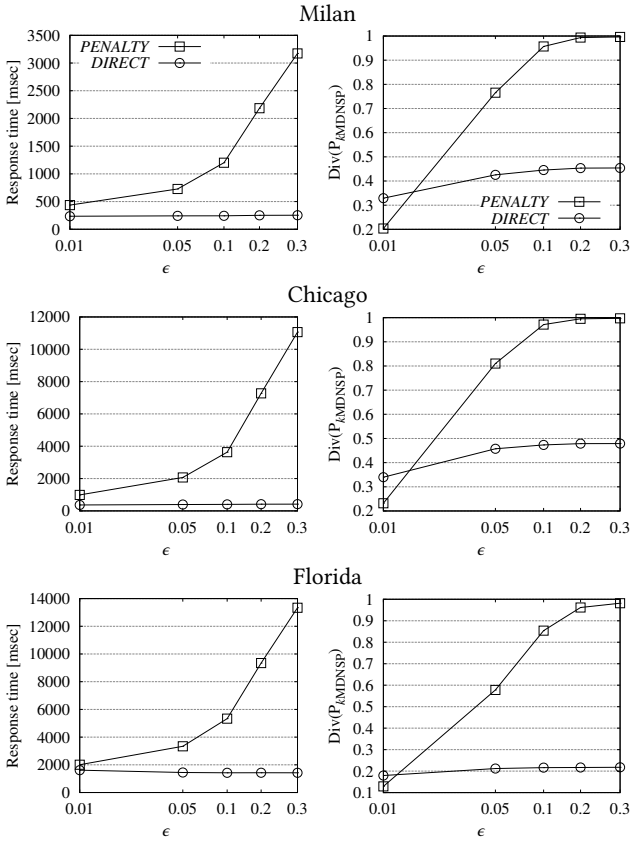

Figure 6: Large scale analysis: vary-ϵ

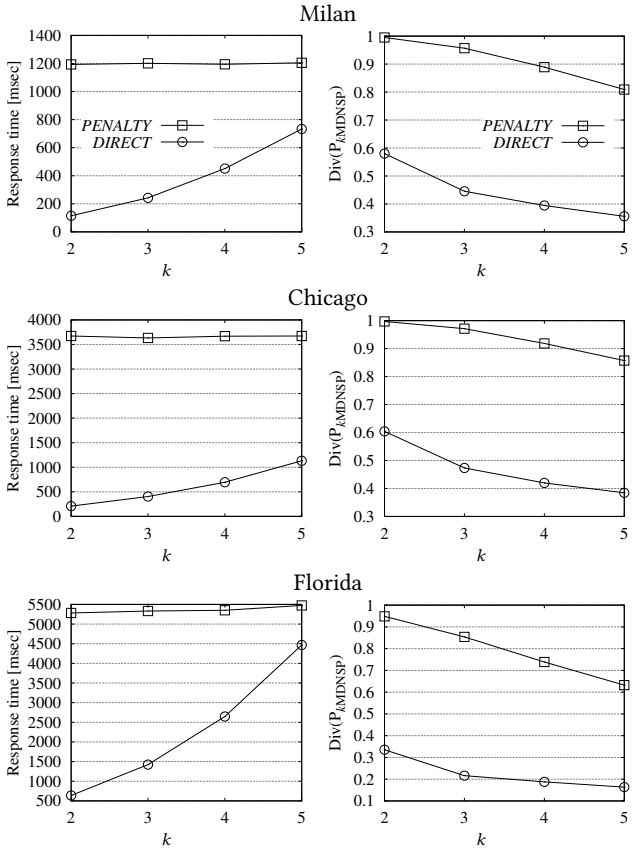

Figure 7: Large scale analysis: vary-k

[6] Thomas H. Byers and Michael S. Waterman. 1984. Determining All Optimal and Near-Optimal Solutions when Solving Shortest Path Problems by Dynamic Programming. *Oper. Res.* 32, 6 (1984), 1381–1384.

[7] W. Matthew Carlyle and R. Kevin Wood. 2005. Near-shortest and K-shortest simple paths. *Networks* 46, 2 (2005), 98–109.

[8] Dan Cheng, Olga Gkountouna, Andreas Züfle, Dieter Pfoser, and Carola Wenk. 2019. Shortest-Path Diversification through Network Penalization: A Washington DC Area Case Study. In *IWCTS@SIGSPATIAL*. 10:1–10:10.

[9] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. 2015. Alternative Routing: K-shortest Paths with Limited Overlap. In *ACM SIGSPATIAL GIS*. 68:1–68:4.

[10] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. 2017. Exact and Approximate Algorithms for Finding k-Shortest Paths with Limited Overlap. In *EDBT*. 414–425.

[11] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. 2018. Finding k-dissimilar paths with minimum collective length. In *ACM SIGSPATIAL GIS*. 404–407.

[12] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. 2020. Finding k-shortest paths with limited overlap. *VLDB J.* 29, 5 (2020), 1023–1047.

[13] Daniel Delling and Dorothea Wagner. 2009. Pareto Paths with SHARC. In *SEA*. 125–136.

[14] E W Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (1959), 269–271.

[15] Marina Drosou and Evaggelia Pitoura. 2012. DisC diversity: result diversification based on dissimilarity and coverage. *Proc. VLDB Endow.* 6, 1 (2012), 13–24.

[16] M. R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.*

[17] Yeon-Jeong Jeong, Tschangho John Kim, Chang-Ho Park, and Dong-Kyu Kim. 2009. A Dissimilar Alternative Paths-search Algorithm for Navigation Services: A Heuristic Approach. *KSCE Journal of Civil Engineering* 14, 1 (2009), 41–49.

[18] P E Johnson, D S Joy, D B Clarke, and J M Jacobi. 1993. *HIGHWAY 3.1: An enhanced HIGHWAY routing model: Program description, methodology, and revised user's manual.* Technical Report. U.S. Dept. of Energy, OSTI.

[19] Alireza Karduni, Amirhassan Kermanshah, and Sybil Derrible. 2016. A Protocol to Convert Spatial Polyline Data to Network Formats and Applications to World Urban Road Networks. *Scientific Data* 3, 160046 (2016).

[20] Donald E. Knuth. 2005. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating All Combinations and Partitions.* Addison-Wesley Professional.

[21] Hans-Peter Kriegel, Matthias Renz, and Matthias Schubert. 2010. Route skyline queries: A multi-preference path planning approach. In *IEEE ICDE*. 261–272.

[22] Lingxiao Li, Muhammad Aamir Cheema, Mohammed Eunus Ali, Hua Lu, and David Taniar. 2020. Continuously Monitoring Alternative Shortest Paths on Road Networks. *Proc. VLDB Endow.* 13, 11 (2020), 2243–2255.

[23] Lingxiao Li, Muhammad Aamir Cheema, Hua Lu, Mohammed Eunus Ali, and Adel Nadjaran Toosi. 2021. Comparing Alternative Route Planning Techniques: A Comparative User Study on Melbourne, Dhaka and Copenhagen Road Networks. *IEEE TKDE* preprint (2021).

[24] Huiping Liu, Cheqing Jin, Bin Yang, and Aoying Zhou. 2018. Finding Top-k Shortest Paths with Diversity. *IEEE TKDE* 30, 3 (2018), 488–502.

[25] Dennis Luxen and Dennis Schieferdecker. 2014. Candidate Sets for Alternative Routes in Road Networks. *ACM J. Exp. Algorithmics* 19, 1 (2014).

[26] Kyriakos Mouratidis, Yimin Lin, and Man lu Yiu. 2010. Preference Queries in Large Multi-cost Transportation Networks. In *IEEE ICDE*. 533–544.

[27] You Peng, Xuemin Lin, Ying Zhang, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2021. Efficient Hop-constrained st Simple Path Enumeration. *The VLDB Journal* (2021), 1–25.

[28] Nagui M. Rouphail, S. Ranji Ranjithan, Wael El Dessouki, Timothy Smith, and E. Downey Brill. 1995. A Decision Support System for Dynamic Pre-Trip Route Planning. In *AATTE*. 325–329.

[29] Behnaz Saboonchi, Pierre Hansen, and Sylvain Perron. 2014. MaxMinMin p-dispersion problem: A variable neighborhood search approach. *Comput. Oper. Res.* 52 (2014), 251–259.

[30] Marcos R Vieira, Humberto L Razente, Maria CN Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Traina, and Vassilis J Tsotras. 2011. On query result diversification. In *IEEE ICDE*. 1163–1174.

[31] Kaiping Zheng, Hongzhi Wang, Zhixin Qi, Jianzhong Li, and Hong Gao. 2017. A survey of query result diversification. *Knowl. Inf. Syst.* 51, 1 (2017), 1–36.

A PROOF OF THEOREM 3.2

We prove both parts of the theorem by reducing an NP-complete problem to the problem of deciding whether there are k disjoint path of length at most $1+\epsilon$ times the shortest path, i.e. deciding whether there is a set P of k path with $Dis(P) = 1$ for the constructed instance.

For the first part, we make a reduction from the weakly NP-complete Partition-Problem, i.e. we are given natural numbers $a_1, \ldots, a_r$ and ask whether there is a subset $I \subseteq \{1, \ldots, r\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$. The hardness of the problem was already shown in [16] and it is listed as problem [SP12].

Given an instance $\{a_1, \ldots, a_r\}$ of the Partition-Problem, we construct the graph $G = (N, E)$ with $N = \{s, n_0, n_1, \ldots, n_r\}$ and the following edges:

- two edges e_i and f_i from n_{i-1} to n_i, the first of weight 0, the second of weight a_i and
- two edges e_0 and f_0 from s to n_0, both of weight $W := \sum_{i \in \{1,\ldots,r\}} a_i$.

Let furthermore $t = n_r$, $k = 2$ and $\epsilon = 0.5$. We show that this instance has k edge-disjoint near-shortest path, iff the Partition-instance has a solution (see Figure 8 for an illustration).

Clearly, the construction can be done in polynomial time. The shortest path has length W (one edge e_0 or f_0 of weight W and then the edges e_i, $i \geq 1$ of weight 0). Assume that there are two disjoint path p_1 and p_2 of length at most $3W/2$. Each of them has the form $(s, n_0, n_1, \ldots, n_r)$, i.e. for each $0 \leq i \leq r$ the paths p_1 and p_2 contain one of e_i or f_i. Hence, the paths have to use first an edge of weight W and the weight of the remaining edges have to be at most $W/2$. Let $I \subseteq \{1, \ldots, r\}$ be the set of indices i such that p_1 uses f_i. Then $w(p_1) = W + \sum_{i \in I} w(f_i) = W + \sum_{i \in I} a_i$. Similarly, $w(p_2) = W + \sum_{i \notin I} w(f_i) = W + \sum_{i \notin I} a_i$. As $W = \sum_{i \in \{1,\ldots,r\}} a_i$, we have $\sum_{i \in I} a_i = W/2 = \sum_{i \notin I} a_i$.

On the other hand, if I solves the PARTITION-Problem, i.e. $\sum_{i \in I} a_i = W/2$, we can easily construct the paths p_1 using the edges e_0, f_i for $i \in I$ and e_i for $i \notin I$ and p_2 using f_0, f_i for $i \notin I$ and e_i for $i \in I$. The paths are edge-disjoint and both have weight $3W/2$.

For the second part, we make a reduction from the strongly NP-hard Disjoint-Path-Problem, i.e. given a graph $G = (N, E)$ and pairs (s_i, t_i) for $1 \leq i \leq k$, find k edge-disjoint path $p_1, \ldots p_k$ such that p_i is a path from s_i to t_i. This problem is listed as [ND40] of the NP-complete problems in [16].

Given an instance $G = (N, E)$ and $(s_i, t_i)_{i \in \{1,\ldots,k\}}$ of the Disjoint-Path-Problem, we construct the graph $G = (N \cup \{s, t\}, E \cup \{(s, s_i) \mid 1 \leq i \leq k\} \cup \{(t_i, t) \mid 1 \leq i \leq k\}$. The edges in E get weight 0, the edges (s, s_i) weight $k + i$ and the edges (t_i, t) weight $2k - i$. Let W be the weight of the shortest $s - t$ path in this graph and let $\epsilon = 3k/W - 1$. Hence, the length of the near-shortest path is at most $(1 + \epsilon) \cdot W = 3k$. Clearly, the construction can be done in polynomial time. We show that this instance has a solution, iff the Disjoint-Path-instance that a solution (see Figure 9 for an illustration).

Assume that there are edge-disjoint path $p_1, \ldots, p_k$ such that $\ell(p_i) \leq 3k$ for all i. Let p_i be the path starting with (s, s_i). As the union of these path contain all k edges leaving s and all k edges entering t, the total weight of the paths is $\sum_{i=1}^{k} w(s, s_i) + w(t_i, t) = \sum_{i=1}^{k} k + i + 2k - i = 3k^2$. As the average weight of the path is equal to their maximal allowed weight, the weight of each path p_i has to be $3k$. Hence, the path p_i starting with (s, s_i) has to end with (t_i, s). If we remove the first and last edges from the paths p_i, we obtain a solution of the Disjoint-Path-instance.

On the other hand, if $p_1, \ldots, p_k$ is a solution of the Disjoint-Path-instance, we can add (s, s_i) and (t_i, s) to p_i and get k edge-disjoint path between s and t, all of length $3k$.

We want to mention that we can easily modify the construction such that the edge lengths are different from 0 and satisfy the triangle inequality. In the first case, we simply add W to each edge length, increasing the shortest path length to $(r+1)W$ and the two paths of a partition would have length $(r + 3/2)W$ each. Hence, we should chose $\epsilon = (r + 3/2)/(r + 1) - 1$. For the second case, we chose all edges of the given graph having length 1 and multiply $|N|$ to the weight of each edge added in the construction. In the choice of ϵ, we have to take into account that a path can have between 1 and $|N| - 1$ edges of G. As we multiplied the weights of the edges adjacent to s and t by $|N|$, we ensured that the edges adjacent to s and t of a path dominate its total length.

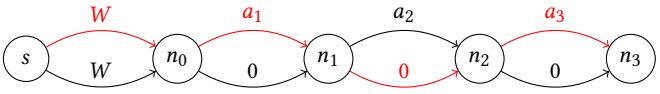

Figure 8: The graph that would be constructed for the instance of Partition with the items of weights a_1, a_2 and a_3. The red path would correspond to choosing items 1 and 3 and has length $W + a_1 + a_3$.

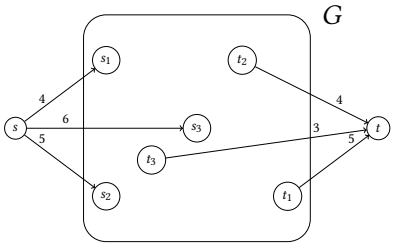

Figure 9: The graph constructed for an instance of the Disjoint-Path-Problem with three pairs of nodes. In order to construct three edge-disjoint path in the constructed graph of length at most 9, the path starting with (s, s_i) has to end with (t_i, t). Hence, we have to find disjoint path between the pairs s_i and t_i.