Efficient Computation of Jogging Routes*

Andreas Gemsa, Thomas Pajor, Dorothea Wagner, and Tobias Zündorf

Department of Computer Science, Karlsruhe Institute of Technology (KIT)
{gemsa,pajor,dorothea.wagner}@kit.edu, tobias.zuendorf@student.kit.edu

Abstract. We study the problem of computing jogging (running) routes in pedestrian networks: Given source vertex s and length L, it asks for a cycle (containing s) that approximates L while considering niceness criteria such as the surrounding area, shape of the route, and its complexity. Unfortunately, computing such routes is NP-hard, even if the only optimization goal is length. We therefore propose two heuristic solutions: The first incrementally extends the route by joining adjacent faces of the network. The other builds on partial shortest paths and is even able to compute sensible alternative routes. Our experimental study indicates that on realistic inputs we can compute jogging routes of excellent quality fast enough for interactive applications.

1 Introduction

We study the problem of computing *jogging routes* in pedestrian networks. Given a source vertex s (the user's starting point), and a desired length L (in kilometers), the problem asks for a cycle of length (approximately) L that contains the vertex s. A "good" jogging route is, however, not only determined by its length; other criteria are just as important. An ideal route might follow paths through nice areas of the map (e. g., forests, parks, etc.), has rather circular shape, and not too many intersection at which the user is required to turn. A practical algorithm must, therefore, take all of these criteria into account.

Much research focused on efficient methods for the related, but simpler, problem of computing point-to-point (shortest) paths. In fact, a plethora of algorithms exist, many of which are surveyed in [3, 9]. They usually employ sophisticated preprocessing to speed up query performance. In contrast, much less practical work exists for computing cycles. Graphs may contain exponentially many (in the number of vertices) cycles, even if they are planar [1]. If the length of the cycles is restricted by L, they can be enumerated in time $O((n+m)(c+1))$, where c is the number of cycles of length at most L [8]. If one is interested in computing cycles with exactly k edges, the problem can be solved in $O(2^k m)$ expected time [10]. Unfortunately, none of these methods seem practical in our scenario. To the best of our knowledge, no efficient algorithms that quickly compute sensible jogging routes exist.

This work introduces the Jogging Problem. It turns out to be NP-hard, hence, we propose two heuristic approaches. The first, *Greedy Faces* is based

* Partially supported by DFG grant WA 654/16-1.

on building the route by successively joining adjacent faces of the network. The second, *Partial Shortest Paths*, exploits the intuition of constructing equilateral polygons via shortest paths. The latter can be easily parallelized and has the inherent property of providing sensible alternative routes. The result of our algorithms are routes of length within $(1 \pm \varepsilon)L$, but also consider other important criteria that optimize the surrounding area, shape, and route complexity. An experimental study justifies our approaches: Using OpenStreetMap data, we are able to compute jogging routes of excellent quality in under 200 ms time, which is fast enough for interactive applications.

The paper is organized as follows. Section 2 defines variants of the problem and shows NP-hardness. Section 3 introduces our two algorithmic approaches. Section 4 presents experiments, and Section 5 contains concluding remarks.

2 Problems

Before we formally define the considered problems, we need to develop some notation. We model pedestrian networks as *undirected graphs* $G = (V, E)$ with nonnegative integral *edge costs* $\ell\colon E \to \mathbb{Z}_{\geq 0}$. Usually, vertices correspond to intersections and edges to walkable segments. Also, we assume that our graphs admit straight-line embeddings, since vertices have associated latitude/longitude coordinates. For simplicity, our graphs are always connected. A *path* P is a sequence of vertices $P = [u_1, \ldots, u_k]$ for which $u_i u_{i+1} \in E$ must hold. Note that we sometimes just write u_1-u_k-path or P_{u_1,u_k} for short. If the first and last vertices coincide, we call P a *cycle*. The *cost* of a path, denoted by $\ell(P)$, is the sum of its edge-costs. A *shortest path* between two vertices u_1 and u_2 is a u_1-u_2-path with minimum cost. At some places we require intervals around a value $x \in \mathbb{Z}_{\geq 0}$ with error $\varepsilon \in \mathbb{R}_{\geq 0}$. We define them by $I(x, \varepsilon) = [\lfloor (1-\varepsilon)x \rfloor, \lceil (1+\varepsilon)x \rceil]$.

Simple Jogging Problem. The first problem we consider is the Simple Jogging Problem (SJP): We are given a graph G, source vertex $s \in V$, and a targeted cost $L \in \mathbb{Z}_{\geq 0}$ as input. The goal is to compute a cycle P through s with cost $\ell(P) = L$. In practical scenarios, cost usually represent geographical length. It turns out that SJP is NP-hard by reduction from Hamiltonian Cycle. Note that from this, NP-hardness follows for the respective optimization problem, i. e., finding a cycle that *minimizes* $|\ell(P) - L|$.

If we allow running time in the order of L, one can solve SJP by a dynamic program, similarly as it is known for the Subset Sum Problem [5]. The algorithm maintains a boolean matrix $Q\colon V \times \mathbb{Z}_{\geq 0} \to \{0, 1\}$ of size $|V| \times L$, which indicates whether a path to vertex u with cost ℓ exists. Initially, Q is set to all-zero, except for the entry $Q(s, 0)$, which is set to 1. It then considers subsequent cost values ℓ in increasing order (beginning at 0). In each step, the algorithm checks for all edges $uv \in E$ if an existing path can be extended to v with cost ℓ. It does so by looking if $Q(u, \ell - \ell(uv))$ is set to 1, updating $Q(v, \ell)$ accordingly. The algorithm stops as soon as ℓ exceeds the input cost L. Then, the requested jogging route exists iff $Q(s, L) = 1$ holds. The running time of the algorithm is $O(L|E|)$, and thus we conclude that the SJP is weakly NP-hard [5].

Relaxed Jogging Problem. In practice, solely optimizing length (or cost) may result in undesirable routes. Jogging is a recreational activity, therefore, one usually also considers the surrounding area (parks and forests), the shape (preferably edge-disjoint), and the complexity of the route (small number of turns). We argue that the primary goal remains geographical length. However, we allow some (user-specified) slack on the length to take the aforementioned criteria into account. This motivates the Relaxed Jogging Problem (RJP): Given a graph G, a source vertex $s \in V$, input length $L \in \mathbb{Z}_{\geq 0}$, and a parameter $\varepsilon \in [0, 1]$, the goal is to compute a cycle P through s with cost $\ell(P) \in I(L, \varepsilon)$ while optimizing a set of *soft criteria*. We identify three important criteria in the following.

To account for the surrounding area, we introduce *badness* as a mapping on the edges $\mathrm{bad}\colon E \to [0, 1]$. Smaller values indicate "nicer" areas (e. g., parks). Badness values on the edges are provided by the input data. To extend badness to paths, we combine it with the path's length. (Note that we assume costs to represent geographical length for the remainder of the paper.) That is, for a path $P = [u_1, \ldots, u_k]$ its badness is defined by $\mathrm{bad}(P) = \sum \mathrm{bad}(u_i u_{i+1})\ell(u_i u_{i+1})/\ell(P)$. By these means, badness values are scaled by their edge lengths, but are still in the interval $[0, 1]$. This enables comparing paths (wrt. badness) of different lengths.

To optimize edge-disjointness of paths, we consider *sharing*. It counts edges that appear at least twice on P, scaled by their length. Formally, it first accumulates into a set D all indices i, j for which either $u_i u_{i+1} = u_j u_{j+1}$ or $u_i u_{i+1} = u_j u_{j-1}$ hold. (Note that edges are undirected.) The sharing of path P is then $\mathrm{sh}(P) = \sum_{i \in D} \ell(u_i u_{i+1})/\ell(P)$. Sharing values are also in $[0, 1]$.

To evaluate route complexity, we consider *turns*. For two edges a and b, we measure their angle $\measuredangle(a, b)$, and regard them as a turn, iff $\measuredangle(a, b) \notin I(180°, \alpha)$ holds. We usually set α to 15 %.

3 Algorithms

We now introduce our two approaches for the Relaxed Jogging Problem: *Greedy Faces* and *Partial Shortest Paths*. We present each approach in turn, starting with a basic version, then, proposing optimizations along the way.

3.1 Greedy Faces

Assume that we are already given a tentative jogging route (i. e., a cycle in G that contains s). A natural way to extend it, is to attach one of its adjacent "blocks" that lie on the "outer" side of the route. Then, repeat this step, until a route of desired size and shape has been grown. In a planar graph, blocks correspond to faces. But our inputs may contain intersecting edges (such as bridges and tunnels), albeit only few in practice. We, therefore, propose preprocessing G to identify blocks (we still call them faces). These are used by our greedy faces algorithm. Finally, we present smoothening techniques to reduce route complexity in a quick postprocessing step.

Identifying Faces. For our algorithm to work, we must precompute a set F of *faces* in G. We identify each face $f \in F$ with its *enclosing* path P_f. Our preprocessing involves several steps. First, we delete the *1-shell* of G by iteratively removing vertices (and their incident edges) from G that have degree one. The resulting graph is 2-connected and no longer contains dead-end streets (which we want to avoid, anyway). Next, we consider all remaining edges $uv \in E$. For each, we perform a *right-first search*, thereby, constructing an enclosing path P_f for a new face f. More precisely, we run a depth-first search, beginning at uv. Whenever it reaches a vertex x (via an edge a), it identifies the unique edge b that follows a in the (counterclockwise) circular edge ordering at x. (Note that this ordering is always defined for embedded graphs.) It adds b to P_f. If $b = uv$, the algorithm stops, and adds f to F, discarding duplicates. However, since G is not necessarily planar, the edge b might intersect with one of its preceding edges on P_f. In this case, it removes b from P_f, and considers the next edge (after b) in the circular order at x for expansion. While constructing F, the algorithm remembers for each edge a list of its incident faces. It uses them to build a *dual graph* $G^* = (V^*, E^*)$: Vertices correspond to faces (of G), and two faces are connected in G^*, iff they share at least one edge in G. This definition of G^* extends the well-known graph duality for planar graphs, however, as G may not be planar, so may not be G^*. The running time of the preprocessing is dominated by the face-detection step. For every edge it runs a right-first search, each in time $O(|E|)$. Whenever it expands an edge, it must perform intersection tests with up to $O(|V|)$ preceding edges. This results in a total running time of $O(|V||E|^2)$. Note that we expect much better running times in practice: On realistic inputs we may assume faces to have constant size.

Greedy Faces Algorithm. Our greedy faces algorithm, short GF, now uses G^* as input. Its basic idea is to run a (modified) breadth first search (BFS) on G^*. It starts by selecting an arbitrary face $f \in V^*$ that contains the source vertex s, i. e., where $s \in P_f$ holds. It then grows a BFS-tree T (rooted at f), until a stopping condition is met. When it stops, the jogging route P is retrieved by looking at the set of *cut edges* that separate T from $V^* - T$: Their corresponding edges in G constitute a cycle. (Note that this is a well-known property on planar graphs, but carries over to our definition of G^*.) However, to make P a feasible jogging route, we must ensure two properties: The cycle must be (a) simple, and (b) still contain s. We ensure both while growing T. Regarding (a), we know that the corresponding cycle P in G is simple iff the subgraph induced by $V^* - T$ is connected. We check this condition when expanding an edge $fg \in E^*$ during the BFS, discarding fg if adding g to T would disconnect $V^* - T$. Regarding (b), The vertex s is still part of the jogging route as long as at least one incident face of s remains in $V^* - T$. We also perform this check while expanding edges, discarding them whenever necessary. The result of every iteration of the BFS is a potential jogging route P. The algorithm stops as soon as the cost of P exceeds $(1 + \varepsilon)L$. It then returns, among all discovered routes whose length is in $I(L, \varepsilon)$, the one with minimum total badness.

However, up to now, GF does not *optimize* badness. To guide the search towards "nice" areas of the graph, we propose a force-directed approach. Therefore, consider a face f and the *geometric center* $C(f)$ of its enclosing path. Inspired by Newton's law of gravity, we define a force vector $\boldsymbol{\phi}(f,p)$ acting upon a point p of the map by $\boldsymbol{\phi}(f,p) = (\mathrm{bad}(f) - 0.5)\ell(f)/|\boldsymbol{d}|^2 \cdot \boldsymbol{d}/|\boldsymbol{d}|$, where $\boldsymbol{d} = p - C(f)$. Note that, depending on $\mathrm{bad}(f)$, the force is repelling/attracting. Also, the vector $\boldsymbol{\phi}(f,p)$ is directed, and its intensity decreases with the distance squared. Now, the force that acts upon a face g is the sum of the forces over all (other) faces in the graph (toward g). More precisely, $\boldsymbol{\phi}(g) = \sum_{f \in V^*} \boldsymbol{\phi}(f, C(g))$. In practice, we quickly precompute these values restricted to reachable faces (i. e., faces within a radius of $L/2$ from s). The BFS in our algorithm now extends the edge $fg \in E^*$ next, for which g has the highest force in *direction of extension*. More precisely, it extends fg, iff g maximizes the term $\boldsymbol{\phi}(g)\cos(\measuredangle(\boldsymbol{\phi}(g), C(f) - C(P)))$. Note that $C(P)$ is the geometric center of the current (tentative) jogging route P in the algorithm, and $\measuredangle(\cdot,\cdot)$ measures the angle of two vectors. In principle, further criteria can be added to the BFS (e. g., via linear combinations): The *roundness* considers the ratio of the route's perimeter to its area (lower values are better); *convexity* takes the distance between a candidate face and the current route into account (higher values are better). However, preliminary experiments showed that (on realistic inputs) the effect of these criteria is limited. The running time of GF is bounded by the BFS on G^*. In the worst case, it scans $O(|V^*|)$ faces. The next face it expands to can be determined in time $O(|V^*|)$, yielding a total running time of $O(|V^*|^2)$. Finally, recall that our preprocessing removes the 1-shell of G. For the case that the source vertex s is part of the 1-shell, we quickly find the (unique) path P' to the first vertex s' that is not in the 1-shell. We then run our algorithm, but initialized with s' and $L' = L - 2\ell(P)$, simply attaching P' to the route afterward. Also note that routes obtained by GF are optimal with respect to sharing: The only (unavoidable) place it may occur is on P' (in case s is in the 1-shell).

Route Smoothening. By default, GF provides no guarantee on route *complexity* (i. e., on the number of turns). We, therefore, propose reducing it by *smoothing* the route in a postprocessing step. To do so, we first select a small subsequence $P' \subset P$ of the route's vertices. (Note that s must be part of P'.) Then, for each two subsequent vertices $uv \in P'$, we compute a shortest u-v-path (e. g., by Dijkstra's algorithm [4]). Finally, concatenating these paths produces the smoothened route. To also take badness into account, we use a custom metric $\omega\colon E \to \mathbb{Z}_{\geq 0}$, defined by $\omega(a) = \mathrm{bad}(a)\ell(a)$, when computing shortest paths.

It remains to discuss how we choose the subsequence P' from P. We propose three rules. The first, called *equidistant rule* (es), simply selects the k (an input parameter) vertices from P, which are distributed equally regarding their subsequent distances. More precisely, vertex $u \in P$ is selected as the i-th vertex on P' if it minimizes $\ell(P)i/k - \ell(P_{s,u})$ (here, $P_{s,u}$ denotes the subpath of P up to vertex u). Unfortunately, this rule may select vertices at arbitrary (with respect to the route's shape) positions. Therefore, our second rule, called *convex rule* (cs), obtains P' by computing the *convex hull* of P, e. g., by running Graham's Scan

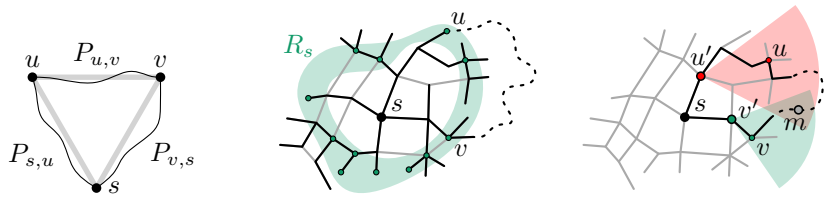

Fig. 1. Left: Intuition of constructing 2-via-routes. Middle: Shortest path tree rooted at s and ring R_s with candidate vertices u, v forming a feasible route (dotted). Right: Selecting middle vertices m that lie "behind" u, v in the shortest path trees of u', v'.

algorithm [6] on P. In case the source vertex s is not part of the convex hull, we must still add it to P': We set its position next to the first vertex of P that is contained in P's convex hull. Finally, the third rule, called *important vertex rule* (ivs), tries to identify k (again, an input parameter) "important" vertices of P: At first, it slices P into k subpaths of equal length. From each, it then selects the vertex u whose incident edges have lowest total badness (i. e., $\prod_{uv \in E} \text{bad}(uv)$ is minimized). This rule follows the intuition that vertices that share many edges of low badness are more likely in "nicer" areas. Note that while smoothening helps to reduce route complexity, its drawback is that the route's length may change arbitrarily. We address this issue by our next approach.

3.2 Partial Shortest Paths

As discussed, GF provides no guarantee on the deviation from the requested route length, if they are smoothened. We, therefore, propose a second approach: It directly computes a set of *via vertices*, connected by shortest paths, but such that the length of the resulting routes is guaranteed to be in $I(L, \varepsilon)$. In the following, we refer to jogging routes that use k via vertices by *k-via-routes*.

2-via-routes. For our basic version, we exploit the intuition of constructing equilateral triangles (see Fig. 1, left), thus, obtaining 2-via-routes. We know that s must be part of the route. Therefore, we choose s as one of the triangle's vertices. It now remains to compute two vertices u, v (and related paths), such that $\ell(P_{s,u}), \ell(P_{u,v}), \ell(P_{v,s}) \in I(L/3, \varepsilon)$. From this, we obtain the required total length of $I(L, \varepsilon)$. To select u and v, we, at first, define a metric on the edges $\omega \colon E \to \mathbb{Z}_{\geq 0}$ that takes the edge's badness into account. As in Section 3.1, we set $\omega(a) = \text{bad}(a)\ell(a)$. We now run a shortest path computation on G from s using this metric with Dijkstra's algorithm [4]. To limit the search, we do not relax edges out of vertices x for whom $\ell(P_{s,x})$ exceeds $(1 + \varepsilon)L/3$. (Note that $\ell(P_{x,s})$ can be stored with x during the algorithm with negligible overhead.) The resulting shortest path tree T_s (rooted at s) accounts for "nice" paths by optimizing ω, and provably contains all feasible candidate vertices u (and v). We refer to this subset of candidate vertices as *ring* around s with distance $I(L/3, \varepsilon)$, in short R_s. We must now find two vertices of the ring that have a connecting

path with length $I(L/3, \varepsilon)$. To do so, we pick a vertex u from the ring R_s, and, compute *its* ring R_u (also with respect to length $I(L/3, \varepsilon)$) by running Dijkstra's algorithm from u, similarly to before. Now, the intersection of R_s with R_u exactly contains the matching vertices v, that is, concatenating $P_{s,u}, P_{u,v}, P_{v,s}$ yields an admissible jogging route (i. e., of length $I(L, \varepsilon)$). See Fig. 1 (middle) for an illustration. The algorithm repeats this step for all vertices in R_s, and selects among all admissible routes it discovers the one minimizing badness. We call this algorithm PSP2 (partial shortest paths with two vias). We remark that distances other than $L/3$ are possible when computing rings. This varies the route's shape, and corresponds to constructing "triangles" with nonuniform side lengths. The running time of PSP2 is dominated by up to $O(|V|)$ shortest path computations, thus, it is bounded by $O(|V|^2 \log |V| + |V||E|)$. Note that we expect much better performance in practice, as the shortest path computations are local.

We now propose two optimizations for PSP2. First, the algorithm can be sped up by a *stopping criterion*. For it to work, it must pick vertices u from R_s in order of increasing value $\omega(P_{s,u})$. Note that this order is automatically provided by Dijkstra's algorithm. It then only needs to consider paths $P_{v,s}$ as third leg of the route, for whom $\omega(P_{v,s}) \geq \omega(P_{s,u})$ holds (all others have been evaluated earlier). By this, the total badness of any route P the algorithm may still find is lower-bound by $\mathrm{bad}_{\mathrm{lb}} = 2\omega(P_{s,u})/(1+\varepsilon)L$. If we keep track of the route P_{opt} minimizing badness, the algorithm may stop as soon as $\mathrm{bad}_{\mathrm{lb}}$ exceeds $\mathrm{bad}(P_{\mathrm{opt}})$—it will provably not find any route with lower badness. Up to now, PSP2 has no guarantee on the sharing of P. In fact, it can be up to 100 % in extreme cases, thus, we propose the following optimization. When the algorithm computes R_u for a vertex $u \in R_s$, we forbid it to relax any edges from $P_{s,u}$. This ensures that $P_{s,u}$ and $P_{u,v}$ are edge-disjoint. To also make $P_{u,v}$ and $P_{v,s}$ edge-disjoint, we disregard routes whose last edges of $P_{u,v}$ and $P_{v,s}$ coincide. Note that we still allow sharing wrt. to the first and last legs of the route (around s).

3-via-routes. Jogging routes obtained by PSP2 follow shortest paths for each of its three legs $P_{s,u}$, $P_{u,v}$, and $P_{v,s}$. However, no such guarantee exists around u and v, which might be undesirable. We now propose an optimized variant of our algorithm, PSP3. It aims to smoothen the route around u and v. Moreover, it uses three via-vertices, which, in general, produces more circular shaped routes.

The algorithm follows the intuition of constructing regular *quadrilaterals*. Taking the source vertex s as one of the quadrilateral's vertices, it must therefore compute vertices u, m, and v, connected by paths $P_{s,u}$, $P_{u,m}$, $P_{m,v}$, and $P_{v,s}$, each with length $I(L/4, \varepsilon)$. We refer to m as *middle vertex*. The algorithm starts, again, by first computing a ring R_s of vertices from s, but now with distance $I(L/4, \varepsilon)$. (It does so by using Dijkstra's algorithm with metric ω.) To smoothen the route around u and v, we do not use u and v directly as sources for the subsequent shortest path computations (like we did with PSP2). Instead, we consider the (tighter) ring R'_s of vertices around s with distance $I(\alpha L/4, \varepsilon)$. Here, the parameter α takes values from $[0.5, 1]$, and controls smoothness around u and v. We obtain the ring R'_s by traversing the shortest path tree from each vertex $u \in R_s$ upward, until the distance condition is met. Moreover, the vertex u

remembers which vertex u' it created in R'_s (this is required later). Next, the algorithm picks vertices u' from R'_s (in any order), and computes, for each, a ring $R_{u'}$ around u'. To account for α, we set the distance of $R_{u'}$ to $I((2-\alpha)L/4, \varepsilon)$. It follows that vertices in $R_{u'}$ have distance $I(L/2, \varepsilon)$ from s, containing potential middle vertices. Having computed all rings, we then consider for each pair of vertices u', v' in R'_s the intersection M of their rings, i. e., $M = R_{u'} \cap R_{v'}$. The algorithm now selects only such middle vertices $m \in M$ that result in smooth paths around u and v. More precisely, a vertex $m \in M$ is selected, iff the *smoothing condition* holds, i. e., the path $P_{u',m}$ contains u and the path $P_{v',m}$ contains v. Intuitively, we are only interested in the part of M that lies "behind" u (resp. v) on the shortest path tree of $R_{u'}$ ($R_{v'}$). See Fig. 1 (right) for an illustration. Each vertex m that fulfills the smoothing condition represents an admissible jogging route by concatenating $P_{s,u}$, $P_{u,m}$, $P_{m,v}$, and $P_{v,s}$. The algorithm returns, among those, the one with minimum badness. With PSP3, the only vertex around which sharing may occur is m (besides s). We avoid it by discarding middle vertices m, for which the last edges of $P_{u',m}$ and $P_{v',m}$ coincide. This can be efficiently checked during the algorithm.

We now propose two optimizations to speed up PSP3. The first avoids the costly computation of set-intersections: Instead of storing (and intersecting) rings $R_{u'}$, the algorithm maintains a vertex-set M_m at each vertex m of the graph. Whenever Dijkstra's algorithm scans a potential middle vertex m, it adds u to M_m (iff the smoothing condition holds). Moreover, it suffices to keep the (at most) two vertices u, v with lowest associated badness values in each set M_m. As a result, managing middle vertices is a constant time operation. The second optimization avoids some calls to Dijkstra's algorithm: If the ring R'_s contains vertices u' and v' for which u' is an ancestor of v' in the shortest path tree, a single Dijkstra run from u' suffices to handle both u' and v'. Including these optimizations, PSP3 essentially runs $O(|V|)$ times Dijkstra's algorithm. Its total running time is thus $O(|V|^2 \log |V| + |V||E|)$, as well as PSP2's.

Bidirectional Search. To allow more flexibility for selecting the middle vertex, we propose the algorithm PSP3-Bi which is an extension of PSP3 using bidirectional search [2]. As PSP3, it starts by computing R_s, and from that, R'_s. However, it now runs (in turn) for each *pair* of vertices u', v' a bidirectional search. Whenever it scans a vertex m that has already been scanned by the opposite direction, it checks (a) whether u (resp. v) are ancestors of m in the forward (resp. backward) shortest path tree, and (b) if the total length of the combined route is in $I(L, \varepsilon)$. If both hold true, it stops, and considers the just-found jogging route as output (it keeps track of the one that minimizes badness). Note that by design, sharing around m cannot occur. Since PSP3-Bi must run a bidirectional search for each pair of vertices in R'_s, its running time is bounded by $O(|V|^3 \log |V| + |V|^2|E|)$.

Parallelization. All PSP-based algorithms can be parallelized quite easily in a shared memory setup: They, first, sequentially compute the ring R_s (resp. R'_s). Subsequent Dijkstra runs may then be distributed among the available processors. Each processor computes its locally optimal route, and the globally optimal route

is selected in a sequential postprocessing step. To avoid race conditions, we use locking as synchronization primitive, whenever necessary.

Alternative Routes. All PSP-based algorithms provide *alternative routes* without significant computational overhead. Instead of just outputting the route with minimum badness, we may output the k best routes. However, these routes tend to be too similar. We, therefore, only consider routes as alternatives that are pairwise different in their via-vertices u and v from R_s (still selecting the k best regarding badness). By these means, we obtain jogging routes that cover different regions of the graph around the source vertex s.

4 Experiments

We implemented all algorithms from Section 3 in C++ compiled with GCC 4.7.1 and flag `-O3`. Experiments were run on one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz with 64 GiB of DDR3-1600 RAM. We focus on the pedestrian network of the greater Karlsruhe region in Germany. We extracted data from a snapshot of the freely available OpenStreetMap[1] (OSM) on 5 August 2012. We only keep walkable street segments and use OSM's `highway` and `landuse` (of the surrounding polygon, if available) tags to define sensible badness values (see [11] for details). The resulting graph has 104 759 vertices and 118 671 edges.

Our first experiment evaluates quality and performance of our algorithms. For each, we ran (the same) 1 000 queries with source vertex s chosen at random. We request routes of 10 km length and ε set to 10 %. Results are summarized in Table 1. We report the average length (in km) of the computed routes, the standard deviation (Std.-Dev.) of their length, their average badness values (Bad.), their average amount of sharing (Sh.), the number of turns on them (No. Trn.), and the average running time of the algorithm on one, and where applicable, also on four and eight processors (Time-x). Sometimes our algorithms may not find any feasible solution. Therefore, we also report their success rates (Succ. Rate).

Algorithms in Table 1 are grouped into blocks. The first evaluates the greedy faces approach from Section 3.1. We observe that GF succeeds in approximating the required route length of 10 km with very little error. However, for 7 % of our queries no solution was found. One reason is that GF is unable to recover from local optima. However, sharing is almost nonexistent with an average value of 0.2 %. This is expected, since by design sharing for GF only occurs around s, iff it lies in a dead-end street. On the downside, route complexity is quite high with 51 turns on average. This justifies our smoothening rules by shortest paths. We set the number of selected vertices to 6 for GF-es and to 9 for GF-ivs. Interestingly, figures are quite similar for all rules: They reduce route complexity by a factor of almost two, which comes with little increase in sharing (up to 6.9 %). Recall that smoothening may arbitrarily change route lengths. Our experiments indicate that the average route length deviates little (it is still 9.5–9.7 km, depending on the specific rule). However, the figure is much less stable: The mean error (Std.-Dev.)

[1] http://openstreetmap.org

Table 1. Solution quality and performance on our Karlsruhe input for both the Greedy Faces (GF) and Partial Shortest Paths (PSP) algorithms. For smoothening, we apply the equidistant rule (es), convex hull rule (cs), and important vertex rule (ivs) to GF.

Algorithm	Length [km]	Std.-Dev.	Bad. [%]	Sh. [%]	No. Trn.	Succ. Rate	Time-1 [ms]	Time-4 [ms]	Time-8 [ms]
GF	9.89	0.58	48.7	0.2	51	93 %	285	—	—
GF-es	9.61	2.07	43.8	6.5	28	93 %	289	—	—
GF-cs	9.73	2.23	43.0	6.9	29	93 %	296	—	—
GF-ivs	9.48	1.98	41.7	6.0	30	93 %	293	—	—
PSP2	9.99	0.58	27.3	52.5	16	98 %	179	84	63
PSP3	10.14	0.41	31.0	23.6	20	98 %	155	78	72
PSP3-Bi	10.06	0.53	33.4	13.9	21	98 %	446	177	140

increases to around 2 km. Regarding running times, GF runs in 285 ms on average, with a mild increase up to 296 ms ($\approx 4\,\%$), if we enable smoothening.

The second block evaluates the PSP approach from Section 3.2 (we set α to 0.6, where applicable). Again, we succeed approximating the required route length of 10 km with little error (≈ 0.5 km on average for all algorithms). Because PSP considers more route combinations than GF, it is more likely to find a feasible solution. This is reflected by the excellent success rate of 98 % (for all PSP algorithms). Regarding badness, PSP finds "nicer" routes (lower average badness) than any of the GF algorithms. However, their sharing (still only possible around s) is much higher. On average, sharing is 52 % for PSP2's, though, we are able to reduce it to 14 % with PSP3-Bi. This is well acceptable in practice. An important advantage of PSP over GF is route complexity: With 16–21 turns on average, this figure is lower than *any* of the GF algorithms, even with applied smoothening. Enabling the stopping criterion decreases running times from 3 579 ms (not reported in the table) to 179 ms, a factor of 20. The fastest algorithm is PSP3 with 155 ms on average. PSP3-Bi is slower by a factor of 2.9. (Recall that it must run a bidirectional search for every *pair* of vertices from R_s; cf. Section 3.2.) Regarding parallelism, we observe speedups of factor 2.1 (PSP2) and 1.9 (PSP3) on four processors over a sequential execution. As expected, with a speedup of 2.5, PSP3-Bi benefits most from parallelization. Increasing the number of processors to eight, improves little. Still, PSP3-Bi benefits most, with a total speedup of 3.1.

We now present two detailed experiments. The first concerns our smoothening rules, the second evaluates variations of the input parameter ε. Each datapoint is based on (the same) 1 000 queries with s selected at random, and L set to 10 km. Fig. 2 shows results of our first experiment. We set ε to 10 %, and vary (on the abscissa) the number of vertices between which the smoothening process computes shortest paths. The left plot reports, for each smoothening rule, how much it affects the length of the routes. We report the average amount (in percent) it changes. The right plot shows the same figure, but for badness. We observe that our routes tend to get shorter after smoothening. This is expected, since we

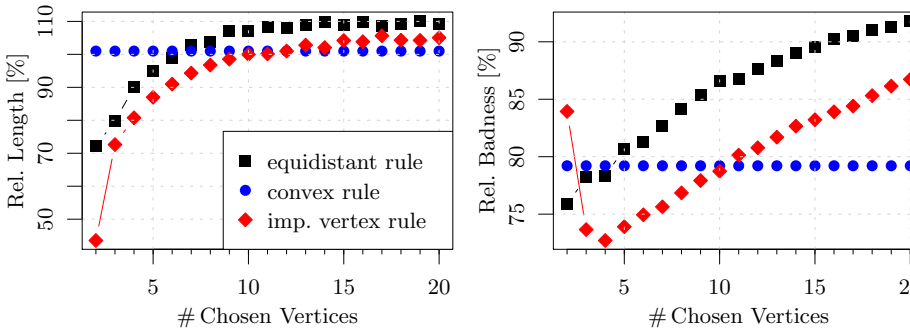

Fig. 2. Evaluating the effect of the smoothening rules on GF. We report the relative amount by which the route's length (left) and badness (right) change while varying the number of vertices the algorithm selects to compute shortest paths (cf. Section 3.1). The legend of the left figure also applies to the right.

rebuild routes using shortest paths. Selecting too few vertices shortens routes severely (to below 50 %). Their length eventually stabilizes above 90 % for six vertices and more. Badness generally improves when using smoothening, but continuously increases with more vertices. Interestingly, the convex rule (which is independent of the number of vertices) seems good regarding both length and badness, which makes it the preferred rule in practice.

Our final experiment evaluates all algorithms for varying input parameter ε. Results are summarized in Fig. 3, which evaluates, for each ε, the average success rate (left plot) and the resulting route's badness (right plot). Note that applying smoothening to GF does not affect the success rate, therefore, we do not enumerate smoothening rules in the left figure. We observe that too much restriction on the allowed length (small ε-values), may result in a low success rate (down to 75 %) and high badness values (more than 50 % for GF). Setting $\varepsilon > 0.07$ already significantly improves the success rate. Unsurprisingly, badness values gradually improve with increasing ε, as this gives the algorithms more room for optimization. Here, a good tradeoff seems setting ε to 0.1. Interestingly, PSP3-Bi's success rate is almost unaffected by ε, even for tiny values below 0.07.

5 Conclusion

In this work, we introduced the NP-hard Jogging Problem. To compute useful jogging routes, we presented two novel algorithmic approaches that solve a relaxed variant of the problem. Besides length, both explicitly optimize two important criteria: Badness (i. e., surrounding area) and sharing (i. e., shape of the route). The methods are based on different intuitions. The first incrementally extends routes by carefully joining adjacent faces of the graph, possibly smoothened by a quick postprocessing step. The second computes sets of alternative routes that resemble equilateral polygons via shortest path computations. Experiments on real-world data reveal that our algorithms are indeed practical: They compute

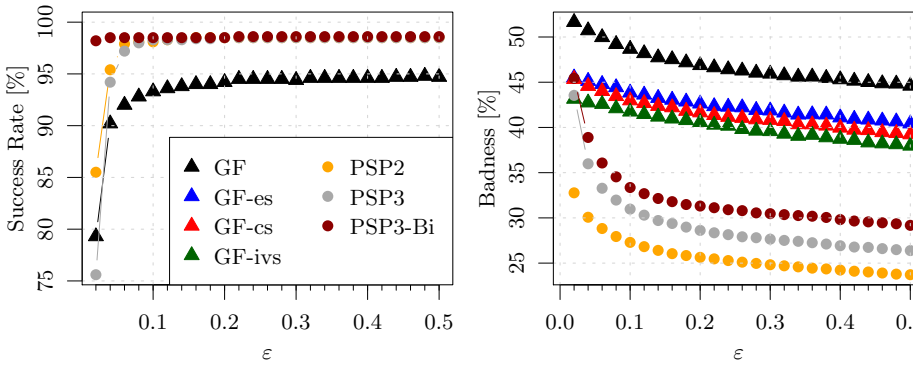

Fig. 3. Evaluating success rate and badness on all algorithms for varying ε. The legend of the left figure also applies to the right. Note that, regarding the greedy faces approach, smoothening does not affect the success rate, hence, we only report it for GF.

jogging routes of excellent quality in under 200 ms time, which is fast enough for interactive applications. Future work includes comparing our algorithms to exact solutions, and better methods for selecting via vertices—either as smoothening rules, or for computing routes directly. Also, providing via vertices (or "areas") as input is an interesting scenario. Finally, we like to accelerate our algorithms further. Especially, PSP may benefit from speedup techniques [3, 9]. This, however, requires adapting them to compute rings instead of point-to-point paths.

References

1. K. Buchin, C. Knauer, K. Kriegel, A. Schulz, and R. Seidel. On the number of cycles in planar graphs. *COCOON*, pp. 97–107, 2007.
2. G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962.
3. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. *Algorithmics of Large and Complex Networks*, pp. 117–139. Springer, Lecture Notes in Computer Science 5515, 2009.
4. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* 1:269–271, 1959.
5. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.
6. R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters* 1(4):132–133, 1972.
7. R. M. Karp. Reducibility among Combinatorial Problems. *Complexity of Computer Computations*, pp. 85–103. Plenum Press, 1972.
8. H. Liu and J. Wang. A new way to enumerate cycles in graph. *AICT and ICIW*, pp. 57-60. IEEE Computer Society, AICT-ICIW '06, 2006.
9. C. Sommer. Shortest-Path Queries in Static Networks, 2012, Submitted. Preprint available at http://www.sommer.jp/spq-survey.htm.
10. R. Yuster and U. Zwick. Color-coding. *Journal of the ACM* 42(4):844–856, 1995.
11. T. Zündorf. Effiziente Berechnung guter Joggingrouten. Bachelor thesis, Karlsruhe Institute of Technology, October 2012.