Planar Network Diversion

Matthias Bentert
University of Bergen, Norway

Pål Grønås Drange
University of Bergen, Norway

Fedor V. Fomin
University of Bergen, Norway

Steinar Simonnes
University of Bergen, Norway

Abstract

Network Diversion is a graph problem that has been extensively studied in both the network-analysis and operations-research communities as a measure of how robust a network is against adversarial disruption. In Network Diversion we want to enforce all s-t-paths through a specific edge b by removing edges from G. This problem is especially well motivated in transportation networks, which are often assumed to be planar. Motivated by this and recent theoretical advances for Network Diversion on planar input graphs, we develop a fast $O(n \log n)$ time algorithm and present a practical implementation of this algorithm that is able to solve instances with millions of vertices in a matter of seconds.

2012 ACM Subject Classification Mathematics of computing → Paths and connectivity problems; Mathematics of computing → Graphs and surfaces; Mathematics of computing → Graph algorithms; Theory of computation → Streaming, sublinear and near linear time algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Minimal cuts, Bridges, Network interdiction, Algorithm engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2025.6

Supplementary Material *Software (Source Code)*: `https://doi.org/10.5281/zenodo.15387001` [3]

Funding *Matthias Bentert*: Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 819416).
Fedor V. Fomin: Supported by the Research Council of Norway, Beyond Worst-Case Analysis in Algorithms (grant agreement no. 314528).

Acknowledgements We sincerely thank Petr Golovach and Tuukka Korhonen for their insightful discussions in the early stages of this work, which helped shape the development of the algorithm presented in this paper.

1 Introduction

The Network Diversion problem is a variant of the classic Minimum Cut problem, which models network vulnerability to adversarial disruption. In this problem, we are given two vertices s and t and an edge b in a graph. The task is to identify at most k edges such that, after their removal, every s-t path is forced to pass through b, while ensuring that at least one such path remains. Equivalently, the problem can be reformulated as finding a minimal s-t cut of size at most $k + 1$ that includes the edge b. Although Network Diversion may initially appear to be a minor variation of Minimum Cut, the two problems differ significantly in terms of computational complexity. While Minimum Cut is solvable in polynomial time, Network Diversion is NP-hard on directed graphs. For undirected graphs, however, it remains an open question whether Network Diversion is NP-hard or admits a polynomial-time algorithm.

In the context of transportation networks, which are often modeled as planar, undirected, and weighted graphs, the ability to compute diverse s-t-cuts is crucial for assessing network resilience and planning infrastructure. Unlike traditional minimum-cut computations, the Network Diversion framework allows exploring different minimal cuts by enforcing the inclusion of a specific edge b. We show empirically that this allows us to find a set of minimal cuts that is quite diverse. Given that planar graphs have $O(n)$ edges, we can efficiently enumerate a diverse set of potential s-t-cuts in $O(n^2 \log n)$ time, providing a tractable means to identify critical bottlenecks. (See Figure 6 in the appendix for an illustration.) Moreover, when edge weights represent failure probabilities, the problem of determining whether a given edge b is likely to become an s-t-bridge reduces to a minimum-weight cut problem in a transformed graph with logarithmic weights. This formulation allows for efficient computation of the probability that all alternative paths fail, making it a valuable tool for transportation planning, security analysis, and network robustness evaluation.

Network Diversion has garnered attention from both the operations-research and network-analysis communities [4, 7, 11, 18, 19, 20]. It measures the vulnerability of a network against flow-manipulation by sabotaging connections in the network and is formally defined as follows.

Planar Network Diversion

Input:	An undirected planar graph $G = (V, E)$, an edge-weight function denoting the destruction cost $w\colon E \to \mathbb{R}_{\geq 0}$, two vertices s and t, an edge b, and a budget $k \in \mathbb{R}_{\geq 0}$.
Problem:	Does there exist a set $F \subseteq E$ of edges of total weight at most k such that b is an s-t-bridge in $G - F$?

In particular, if there is a solution to the problem with small k, then the network is vulnerable as an adversary can damage only few edges to divert all flow or traffic between s and t in the network to a single target connection b (which can be chosen to be particularly susceptible within the network). Suppose an adversary wants to attack a communication network by forcing data to be rerouted through specific links. The problem is to determine the smallest number of edges that need to be removed to successfully carry out this attack, providing a measure of the network's vulnerability.

The problem is particularly relevant to transportation networks and has been extensively studied in the context of planar graphs. Cullenbine et al. [5] provided a polynomial-time algorithm for Network Diversion on planar graphs under the condition that both terminals, s and t, are located on the outer face [6]. They raised the question of whether this algorithm could be extended to arbitrary planar graphs, which has been answered in the affirmative by Bentert et al. [2]. We give a new algorithm here which is deterministic, conceptually simpler, and faster.

1.1 Our Results

We present the first deterministic algorithm for Network Diversion on weighted planar graphs that runs in truly polynomial time. Specifically, it achieves a running time in $O(n \log n)$ with only small hidden constants, making it highly efficient in practice.

Our algorithm is, as previous algorithms, based on the correspondence between cuts and cycles in dual graphs. Specifically, we use this relationship to find a path from the face on the left-hand side of edge b to its right-hand side, which is completed to a cycle by adding the dual edge $b^\star$. This path ensures that vertex s is positioned on one side of the cycle and vertex t

on the opposite side, achieved by identifying an odd-length path [8, 17] in a transformed version of the dual graph. Using methods to find the least expensive odd-length paths in weighted graphs, we efficiently determine the most cost-effective minimal cut that includes b. This approach has potential implications for designing other types of cuts in planar graphs, applicable even when edges are labeled with elements from various groups [14, 16].

As part of our algorithm, we implement Derigs' algorithm to find the least expensive odd-length s-t-path and conduct experiments with two different data structures. We believe this implementation to be of independent interest.

Cut–Uncut Problems. Network Diversion belongs to the broader category of *cut–uncut problems* [12, 13]. There, the goal is to separate specific terminals while maintaining connectivity between designated terminal pairs. These problems are notoriously hard and usually NP-hard. In the context of Network Diversion, we are able to leverage the small number of terminals and the planarity of the input graph to derive a polynomial-time algorithm. In a recent paper, Bentert et al. [2] showed that Two–Sets Cut–Uncut is fixed-parameter tractable in the size of S and T on planar graphs. In this problem, we are given a planar graph, and two sets S and T of vertices, and the goal is to find a small set of edges that separates all of S from all of T, while maintaining connectivity within S and T. This result immediately implies a polynomial-time algorithm for Network Diversion on planar input graphs as shown next. Let $b = \{b_s, b_t\}$. Consider a solution F and the graph $G' = G - (F \cup \{b\})$. Note that exactly one endpoint of b is in the same connected component of G' as s and the other endpoint is in the connected component of t. We can guess which endpoint is in the connected component of s (let us assume without loss of generality b_s) and then solve the instance of Two–Sets Cut–Uncut with $S = \{s, b_s\}$ and $T = \{t, b_t\}$. The algorithm by Bentert et al. [2] is a randomized Monte Carlo algorithm and potentially requires a large number of repetitions, making it computationally demanding. Furthermore, it does not support weighted graphs, hence we would need to simulate weights by introducing parallel edges proportional to each weight (assuming only integral weights), potentially leading to an exponential increase in edges and thus invalidating polynomial-time guarantees. We therefore believe that their algorithm is less competitive. We overcome both of these issues and present a simple, deterministic algorithm that can handle edge weights. Our algorithm is implemented in Rust and available in our repository [3].

1.2 Background

In 1993, Phillips [19] introduced Network Inhibition as a flow-interdiction problem as follows. The input consists of a graph G, two vertices s and t, a budget k, edge capacities c and *edge destruction costs* d (i.e., each edge e has a capacity c_e and a cost d_e for destroying e). One can now assign each edge e a value α_e. This means that one pays αd_e to reduce the capacity of e to $(1 - \alpha)c_e$. The task is to assign numbers in such a way that the maximum flow between s and t is minimized (with respect to the new edge capacities) while the cost of all modifications does not exceed the budget k. Phillips proved NP-completeness on subcubic graphs, and weak NP-completeness for series-parallel graphs, planar graphs, bandwidth-3 graphs, and more [19]. They complemented their findings by providing polynomial-time algorithms for outerplanar graphs and an FPTAS for planar graphs.

That problem was further studied by Wood [20], who gave several other NP-completeness results and implemented the problem as an ILP and considered different LP relaxations. We refer to the doctoral thesis of Kallemyn [15] for a thorough survey on the subject of modeling network interdiction tasks.

Curet [7] introduced the problem Network Diversion on directed graphs. They showed that the problem is NP-complete and implemented an ILP for solving the problem on real-world instances. Cintron-Arias et al. [4] used the Lagrangean relaxation for integer programming to implement ILP-based heuristic algorithms for real-world networks.

A decade later, Cullenbine, Wood, and Newman [6] studied the problem on undirected graphs. They showed how to solve Network Diversion on so-called s-t-planar graphs in polynomial time in both the directed and undirected setting. An *s-t-planar graph* is a planar graph where s and t belong to a common face. Their algorithm is based on the following observation: If we find an s-t-path P in G that contains[1] b, then a cycle in the dual, containing $b^\star$ separates s from t if and only if P is *crossed* an odd number of times. They refer to P as a *reference path*. We use the same idea and explain it in detail in the next section.

▶ **Proposition 1** (Cullenbine et al. [6]). *A solution to Network Diversion on a planar graph G corresponds directly to a minimum-weight, simple, odd-parity cycle $E_C^\star \supseteq \{b^*\}$ in the dual graph of G, where parity is measured with respect to a simple (s,t)-reference path in G that contains b.*

They also note the obstacle for solving Network Diversion on planar graphs.

> *"It is easy to identify a reference path as required by the proposition if one exists. And then, using shortest-path techniques, it is easy to find a minimum-weight odd-parity cycle $E_S^\star$ in $G^\star$ such that $b^\star \in E_S^\star$. Unfortunately, such an approach does not lead to a general, efficient method for solving Network Diversion on undirected planar graphs, because $E_S^\star$ may not be simple, and because minimality of the corresponding (s,t)-cut demands a simple cycle."*

We will demonstrate in Section 3 how we circumvent this obstacle. We note that the question of whether Network Diversion on general undirected graphs is polynomial-time solvable or NP-hard remains open.

Cullenbine et al. [6] also ran experiments on both generated and real-world data sets using a Mixed-Integer-Linear-Programming (MILP) approach. We compare our implementation to theirs and find that our algorithm is faster by orders of magnitude.

2 Preliminaries and Notation

Usually, when working with dual graphs, one needs to consider multigraphs. However, we will only be computing simple paths in the dual graph and hence we can keep only the lowest-cost edge of a set of parallel edges. For the same reason, we can ignore any self-loops in the dual graph and only consider simple weighted graphs. We refer to the textbooks by Agnarsson and Greenlaw [1] and by Diestel [9] for an introduction to general graph theory, planar graphs, and other graph theoretic notation.

We refer to a planar graph with a specific embedding as a *plane graph*, and whenever we talk about *dual graphs*, we mean duals of plane graphs. The dual is defined as usual, and for a plane graph G, we refer to the dual graph as $G^\star$. Since every edge $e \in E(G)$ corresponds

[1] We can find an s-t-path containing b using an algorithm for two-disjoint paths, or using the Odd Path algorithm (see Section 2.2).

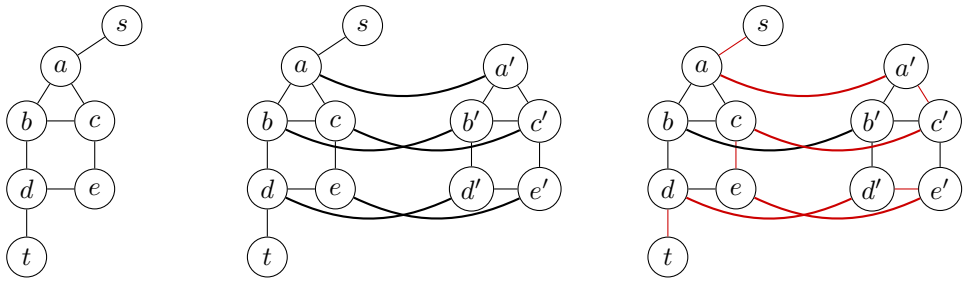


Figure 1 Odd s-t-path using alternating paths. The thick edges are in M.

to a dual edge in $G^\star$, we will refer to the dual edge of e as $e^\star$. Similarly, if $S \subseteq E$, then we write $S^\star$ for $\{e^\star \mid e \in S\}$. For weighted graphs, we let $w(e^\star) = w(e)$.

A set of edges C is called a *cut* if there exists a set $S \subseteq V(G)$ such that C contains exactly those edges with exactly one endpoint in S. A cut $C = \text{cut}(S)$ is an s-t-cut if $s \in S$ and $t \notin S$ or vice versa. A cut C is *minimal* if no proper subset of C is a cut. The *weight* of a cut (or a path/cycle) is the sum of the weights of its edges. In this work, we only consider *minimal non-empty cuts*.

2.1 Derigs' Shortest Odd Path

The main subroutine of our algorithm is an algorithm that finds a shortest odd s-t-path. Here, *shortest* means lowest cost, and *odd* means that we need an odd number of edges. There are several algorithms solving this problem, and in graphs without weights, there is an $O(m)$ algorithm due to Lapaugh and Papadimitriou [17][2]. We note in passing that on directed graphs, checking the existence of an odd path is NP-complete [17].

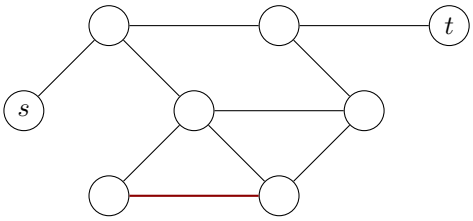
For the weighted version, we use Derigs' algorithm for shortest odd path [8], which is a Dijkstra-like algorithm, but with an additional blossom step. Both Derigs' and Lapaugh and Padadimitriou's algorithms rely on finding augmenting paths in a transformed graph.

▶ **Theorem 2** (Derigs' algorithm [8]). *Given a weighted graph G and two vertices s and t, we can in $O(m \log n)$ time find a cheapest* odd *s-t-path, or correctly conclude that none exist.*

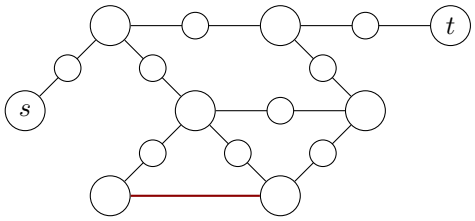
The main idea behind Derigs' algorithm is the following. Let G be an input graph, with two designated vertices s and t and suppose we want to find an *odd* s-t-path. We create a new graph G' consisting of two copies of G, and we let v' denote the copy of v, and refer to v' as the mirror vertex of v. Now, add all edges v, v' to G' and finally delete s' and t' from G'. Create a matching $M = \{(v, v') \mid v \in V(G) \setminus \{s, t\}\}$. Note that neither s nor t are in any edges of M.

Given a graph and a matching, an *alternating path* is a path with an odd number of edges in the graph in which every edge of even index is in the matching and the other edges are not in the matching. It is not hard to see that G has an odd s-t-path if and only if G' has an alternating s-t-path with respect to M. Derigs [8] shows how to do this step in $O(m \log n)$ time.

[2] This algorithm finds a shortest even path, but finding an odd path is simple by adding a pendant vertex s' to s and search for an even s'-t-path instead.



(a) An instance of Shortest Detour Path with the detour edge marked in red.



(b) Instance of Shortest Odd Path: All edges but the detour edge are subdivided.

Figure 2 Shortest Detour Path reduced to Shortest Odd Path by subdividing all edges except the detour. Notice that in the new graph, G', we have $n' = n + m - 1$ and $m' = 2m - 1$.

2.2 An Application of Shortest Odd Path

Before explaining the algorithm for Network Diversion, which builds on shortest odd path, we give another useful application of shortest odd path, that we call Detour Path. This problem is defined as follows.

Detour Path

Input:	An edge-weighted graph G, two vertices s and t, and an edge b.
Problem:	Among all s-t-path that use b, find a shortest one (or conclude that no such path exists).

In Detour Path, we are given an edge-weighted graph G, two vertices s and t, and an edge b, and we are asked to find a shortest path from s to t that uses b. Since we are asking for a path rather than a walk, we need to find two disjoint paths – one from s to one of the endpoints of b and one from t to the other endpoint of b. This can be solved by running two instances of Min-Sum Two Disjoint Paths. However, we can also solve the problem faster by using shortest odd path as follows. On input (G, s, t, b), create the graph G' by subdividing every edge of G except b. If e_1, e_2 are the subdivision edges that replace e, let $w(e_1) = w(e_2) = \frac{1}{2}w(e)$. See Figure 2 for an illustration. Now, it is clear that in G', any path between vertices in the original graph G is odd if and only if the path contains b. Moreover, the length of paths are preserved. The entire procedure can be performed in time $O((n + m) \log n)$.

3 Network Diversion on Planar Graphs

In this section, we present our algorithm for Network Diversion on planar input graphs. We start with a collection of relevant results from the known literature.

First, we use the following strong connection between cuts in planar graphs and cycles in their dual graph.

▶ **Theorem 3** ([9, Proposition 4.6.1]). *Minimal cuts in the graph correspond to simple cycles in the dual graph.*

Second, it was previously observed (for example by Bentert et al. [2]) that a minimal cut separates two vertices if and only if the corresponding cycle crosses a path between those two vertices an odd number of times.

▶ **Observation 4.** *Let P be an s-t-path in G. A cycle $C^\star$ in the dual graph corresponds to a minimal s-t-cut if and only if it crosses P an odd number of times. Moreover, if $C^\star$ is a minimum-cost cycle satisfying the above criteria, then C is a minimum-weight minimal s-t-cut.*

We use this observation in the following reformulation.

▶ **Corollary 5.** *Let E_P be the edges of any s-t-path, and $E_P^\star$ the corresponding dual edges. Any cycle $C^\star$ in the dual graph corresponds to an s-t-cut in G if and only if $C^\star$ contains an odd number of edges from $E_P^\star$.*

Finally, using Derigs' algorithm (Theorem 2), we obtain the following result as a corollary.

▶ **Corollary 6.** *Let G be a graph, $F \subseteq E(G)$ be a subset of edges of G, and s and t be two vertices. We can find in $O(m \log n)$ time an s-t-path that uses an odd number of edges from F, or correctly conclude that none exist.*

Proof. Let G' be the graph that results from subdividing every edge in G that does not appear in F. If e_1, e_2 are the subdivision edges that replace e, let $w_1 = \lceil w/2 \rceil$ and $w_2 = \lfloor w/2 \rfloor$. Now, the length of any path must be congruent (mod 2) to the number of edges it contains in F. ◀

Using the well-known fact that for $n \geq 3$, we have that $m \leq 3n - 6$ for planar graphs [9], we obtain the following.

▶ Remark 7. Since, $m \in O(n)$ we can in time $O(n \log n)$ find an s-t-path that uses an odd number of edges from F, or correctly conclude that none exist.

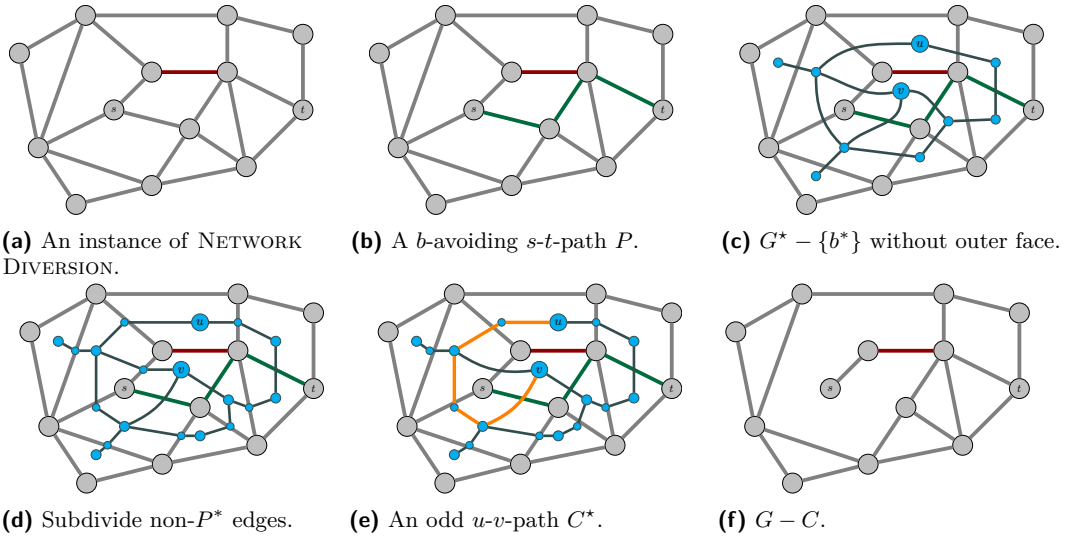
3.1 Algorithm for Planar Network Diversion

Our algorithm for Network Diversion on planar graphs can be explained relatively succinctly, and is illustrated in Figure 3. The algorithm proceeds in five steps described below, and the running time follows from the previous results (Corollary 6 and Remark 7).

Computing the dual. Computing the dual graph of a given plane graph is folklore wisdom, but we include a high-level description for completeness. The input to the algorithm is a plane graph, two vertices and an edge. In our implementation, all input graphs are given as straight-line embeddings (which always exist). However, we only use the fact that every vertex has an ordered list of edges incident to it (ordered in counter-clockwise fashion). Computing the dual of a plane graph G then simply proceeds as follows. Let $e = uv$ be an edge, and let f_{uv} and f_{vu} be the faces one each side of e. We find these faces by traversing directed edges, starting with uv, and following to the *next* edge incident on v after e. When we return to uv, we have computed f_{uv}, and we can traverse the edge the other direction, vu to obtain f_{vu}.

This creates a map from each edge to its two (or one) neighboring faces, and we thus have the vertices and the edges of the dual graph. In total, we visit each edge exactly twice (once in each orientation), and thus it takes linear $O(n)$ time to compute the dual. We note that *bridges* are only incident on one face.

We can finally describe our algorithm. To this end, let $G = (V, E)$ be an edge-weighted graph and let two vertices s and t and an edge b be given. Let $b^\star = \{u, v\}$ be the dual of b. We assume without loss of generality that G is connected.



(a) An instance of Network Diversion. **(b)** A b-avoiding s-t-path P. **(c)** $G^\star - \{b^*\}$ without outer face.

(d) Subdivide non-P^* edges. **(e)** An odd u-v-path $C^\star$. **(f)** $G - C$.

Figure 3 An illustration of the different steps in our algorithm for Network Diversion on planar graphs. We assume here that all edges have the same weight. Steps (a) – (c) are directly from our algorithm and Step (d) shows the reduction from finding a shortest path that crosses the computed path an odd number of times to finding a shortest odd path given by Corollary 6. Step (e) shows the solution of running Derigs' algorithm and Step (f) shows the solution to the input instance we computed.

1. Compute a path P between s and t in $G - \{b\}$ or conclude that b is already a bridge between s and t in G.
2. Compute the dual graph $G^\star$ of G.
3. Find a shortest u-v-path $C^\star$ in $G^\star - \{b^\star\}$ that uses an odd number of edges from $E_P^\star$ using Derigs' algorithm, or conclude that none exist.
4. Return C.

Step 3 is where we overcome the issue raised by Cullenbine et al. [6]. We compute a shortest odd path guaranteed to be *simple* in $G^\star$, which necessarily corresponds to a minimal cut in G.

▶ **Theorem 8.** *Given a plane graph G, s, t, and b, we can compute the minimum diversion set or correctly conclude that none exist in $O(n \log n)$ time.*

Proof. If s and t do not belong to the same connected component of G, then we can safely output no. If there is no s-t-path in $G - \{b\}$, then b is already an s-t-bridge and we can safely output yes. Otherwise, we find an s-t-path P that does not use b. What remains is to show that a minimum minimal s-t-cut containing b corresponds to a shortest cycle using $b^\star$ that uses an odd number of edges from $P^\star$. The cost aspect follows from the correspondence between length of cycles in the dual and cost of minimal cuts in the primal graph, so the only thing remaining is to show that this cycle is indeed an s-t-cut in the primal. By Corollary 5, given a cut C, and a u-v-path S, u and v are on different sides of the cut if and only if S uses an odd number of edges from C, or equivalently, C uses an odd number of edges from S. Hence, a shortest cycle containing $b^\star$ and using an odd number of edges from $P^\star$ indeed corresponds to an s-t-cut in the primal. It follows that the algorithm outlined above is correct. For the running time, note that computing the dual $G^\star$, finding any s-t-path, and subdividing edges, all take $O(n + m) = O(n)$ time. Running Derigs' algorithm takes $O(n \log n)$ time by Remark 7. ◀

Table 1 Comparison of our two implementations for computing a shortest odd path on different graphs. The first column shows the name of the data set (the first seven datasets are real world data, the next bottom four are randomly generated Delaunay graphs of different sizes). The first columns n and m show the number of vertices and edges in the respective graph and the last two columns denote the running times of the naïve implementation and the implementation using the union–find data structure.

	n	m	naïve	union–find
Power	1723	2394	2 ms	2 ms
Oldenburg	6106	7035	6 ms	5 ms
San Joaquin	18263	23874	22 ms	18 ms
Cali Road	21048	21693	21 ms	18 ms
Musae Git	37700	289003	125 ms	123 ms
SF Road N	174956	223001	225 ms	191 ms
Ca Citeseer	227321	814137	629 ms	608 ms
Delaunay 50k	50000	149961	106 ms	96 ms
Delaunay 100k	100000	299959	219 ms	205 ms
Delaunay 150k	150000	449965	346 ms	315 ms
Delaunay 200k	200000	599961	476 ms	449 ms

4 Implementation and Experimental Setup

To the best of our knowledge, we provide the first implementation of Derigs' shortest odd path algorithm. We believe that this implementation might be of independent interest. It can handle edge-weighted graphs, and solves graphs of one million edges in less than a second.

In addition, we implement the algorithm for Network Diversion on planar input graphs. We note here that in our specific implementation, we only work with straight-line embeddings of planar graphs, but this can easily be generalized and replaced with any library that can, given a plane graph, return a mapping between vertices and faces in the input graph to faces and vertices, respectively, of the dual graph. Given a plane graph, computing the dual, and finding an s-t-path that does not use a specific edge b are both simple $O(n)$ algorithms, and the bulk of the running time of the algorithm is finding a shortest odd path using Derigs' algorithm, which runs in time $O(n \log n)$.

We want to point out one particular detail about our implementation. We use Edmond's Blossom algorithm [10] for computing matchings (or, strictly speaking, *alternating paths*). For the following technical discussion, we assume familiarity with this algorithm. When we have found and computed a blossom, we shrink it into a pseudonode by setting the base of all its vertices to the base of the blossom. Whenever we consider a potential blossom edge, we see if the two vertices have the same base, and if so, then they are in fact already in the same pseudonode and the edge can be discarded. Whenever we set u to have the base β, we also have to see if any other vertices have u as their base and set their bases to β as well. Derigs does not specify which data structure to use to update these bases efficiently and the naïve way would search through all vertices in the graph in linear time, which would not actually give the claimed running time. Now we go through only the vertices that have u as their base, in time linear to the count of vertices that need to be updated. The second version is to use a structure resembling union–find, where each disjoint set and its representative is a blossom and its base. To update the base of u we simply set the new base and do nothing else. When we require the base of a vertex, we recursively query its representative's base and contract the path along the way in the style of union–find.

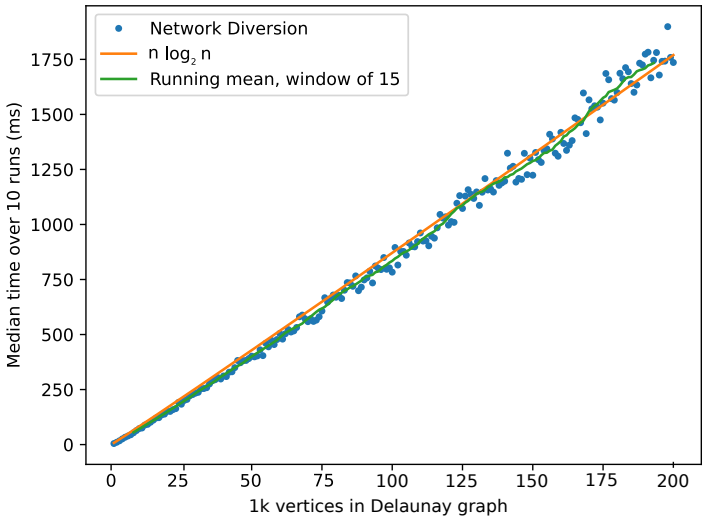


Figure 4 Running times of our implementation for Network Diversion on Delaunay graphs of different sizes. The x-axis shows the number of vertices in 1000s and the blue dots show the running time in milliseconds. The green line shows a running mean running time over the previous 15 instances. The orange line shows the line $^{n \log_2 n}/_{100}$ ms.

All of our experiments were performed on a laptop with an Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz with 192KiB, 1.5 MiB, and 9 MiB L1, L2, and L3 cache, respectively, and 15G RAM, running on GNU/Linux Ubuntu 20.04.6 LTS. We use Rust edition 2021 compiled with `rustc 1.81.0-nightly`.

We tested our implementation of Derigs' algorithm for finding a shortest odd path on both real-world (non-planar) data and on randomly generated Delaunay graphs. A Delaunay graph is the graph formed by the edges of the Delaunay triangulation of a set of randomly drawn points in the plane, where vertices represent the points and edges connect pairs of points that form the sides of the triangles in the triangulation. Delaunay graphs are planar and triangulated, meaning that every face (except the outer) is a triangle.

For Network Diversion on planar graphs, we implemented and compared our algorihm to the previous implementation by Cullenbine et al. [6]. They used both real-world and synthetic data, where they generated weighted grid graphs. Unfortunately, we do not have access to the real-world data sets they used or the exact MILP formulation they used. Instead, we generated random planar graphs similar to the kind of graphs that Cullenbine et al. generated. In particular, we generated randomly weighted grid graphs of different sizes and chose s, t, and b uniformly at random. We also tested our implementation on the generated Delaunay graphs. We ran each experiment 100 times and took the average running time. Since our algorithm does not in any way use the fact that the given graph is a grid graph or a Delaunay graph and the running times for similar-sized graphs are comparable, it is safe to assume that the running times scale with the number of vertices, faces, and edges.

5 Experimental Results

In this section, we discuss our experimental findings. First, we tested our implementation for finding a *shortest odd path* both with a naïve implementation of Edmond's Blossom algorithm and with the union–find data structure discussed in Section 4. The results are depicted in Table 1 and the implementation with union–find turned out to be faster by roughly 10% on average. It sped up the algorithm especially on sparser graphs but we also mention that it occasionally was slower when the number of edges approached $\binom{n}{2}$. However, since planar graphs are sparse, we have chosen union–find for the remainder of the experiments.

Next, we tested our implementation for Network Diversion on randomly generated Delaunay graphs. They are planar by construction, and additionally, they provide diverse structures, capturing a wide variety of graph topologies. They are also relatively dense, which allows us to test the algorithm's performance on challenging instances. The results are shown in Figure 4. We found that for a graph with n vertices, $^{n \log n}/_{100}$ milliseconds was a very precise estimate for the median running time.

Finally, we compared our running time on randomly generated grid graphs to the running times reported by Cullenbine et al. [6] on similarly generated grid graphs. Our running times are reported in Table 2 and the results are compared to the results by Cullenbine et al. in Figure 5. The grids were generated by first creating an $N \times N$ grid, then uniformly at random selecting s, t, and b, and add random weights to all edges. This matches the method of Cullenbine et al. [6], with the exception that they always choose s and t to be on the outer face. Running our algorithm with s and t chosen to be on the outer face performs at average slightly faster than t_{50} reported in Table 2. On grids of size larger than 100×100, Cullenbine et al. reported timeouts and for grids of size at most 100×100, our algorithm is roughly 1000 times faster. For reference, a line $cn \log n$ – for some constant c that is chosen to fit our curve – is also shown. We note that we only provide the running times as Cullenbine et al. reported, and that they use AMPL/CPLEX 12.1 on a Linux machine with four 2.27–GHz processors, whereas ours is run on a laptop on a single 2.80–GHz core.

In their experiments, the grids of size 100×100 ran in roughly 50 seconds[3] and larger grids timed out. Our algorithm solved grids of size 100×100 in less than 0.05 seconds which is roughly 100 times faster than the MILP. Our algorithm was able to solve grids of size 2000×2000 in less than 30 seconds.

6 Conclusion

We developed a simple yet efficient $O(n \log n)$-time algorithm for Network Diversion for planar graphs. We implemented the algorithm in Rust and it performs well, being able to solve instances with millions of nodes in less than 30 seconds.

This is the first deterministic algorithm for Planar Network Diversion and the only algorithm that handles edge weights. The algorithm is significantly simpler than the previous known algorithm by Bentert et al. [2].

The complexity of Network Diversion on general graphs is still open, and is a very interesting problem. Indeed, our technique does not even apply to graphs of genus one. However, we conjecture that the problem remains polynomial-time solvable on these graphs. Another interesting question is parameterization by *crossing number* as many road and transportation networks are not planar due to the existence of bridges and underground tunnels. However, it is often safe to assume that these graphs can be drawn in the plane with few crossing edges. Is it possible to solve Network Diversion in time $f(c) \cdot n^{O(1)}$ for some function f, where c is the crossing number of the given graph?

[3] They report $\mu = 50.1$ s, $\sigma = 39.6$ s for 10 runs. Our algorithm runs in time $\mu = 0.0341$ s, $\sigma = 0.0061$ s.

Table 2 Running times in seconds for solving Network Diversion on weighted grid graphs. The weights on the edges are floats chosen uniformly at random between 0 and 1000. The times are the 25%, 50%, and 75% quartiles, respectively.

	n	m	t_{25}	t_{50}	t_{75}
Grid 10×10	100	180	0.00	0.00	0.00
Grid 20×20	400	760	0.00	0.00	0.00
Grid 30×30	900	1740	0.00	0.00	0.00
Grid 40×40	1600	3120	0.00	0.00	0.00
Grid 50×50	2500	4900	0.01	0.01	0.01
Grid 60×60	3600	7080	0.01	0.01	0.01
Grid 70×70	4900	9660	0.01	0.02	0.02
Grid 80×80	6400	12 640	0.02	0.02	0.02
Grid 90×90	8100	16 020	0.03	0.03	0.03
Grid 100×100	10 000	19 800	0.04	0.04	0.04
Grid 200×200	40 000	79 600	0.16	0.16	0.17
Grid 300×300	90 000	179 400	0.35	0.38	0.40
Grid 400×400	160 000	319 200	0.66	0.74	0.77
Grid 500×500	250 000	499 000	0.90	1.23	1.28
Grid 600×600	360 000	718 800	1.62	1.84	1.99
Grid 700×700	490 000	978 600	2.42	2.55	2.69
Grid 800×800	640 000	1 278 400	3.19	3.50	3.67
Grid 900×900	810 000	1 618 200	4.06	4.71	4.92
Grid 1000×1000	1 000 000	1 998 000	5.58	5.81	6.06
Grid 2000×2000	4 000 000	7 996 000	25.96	28.88	29.74

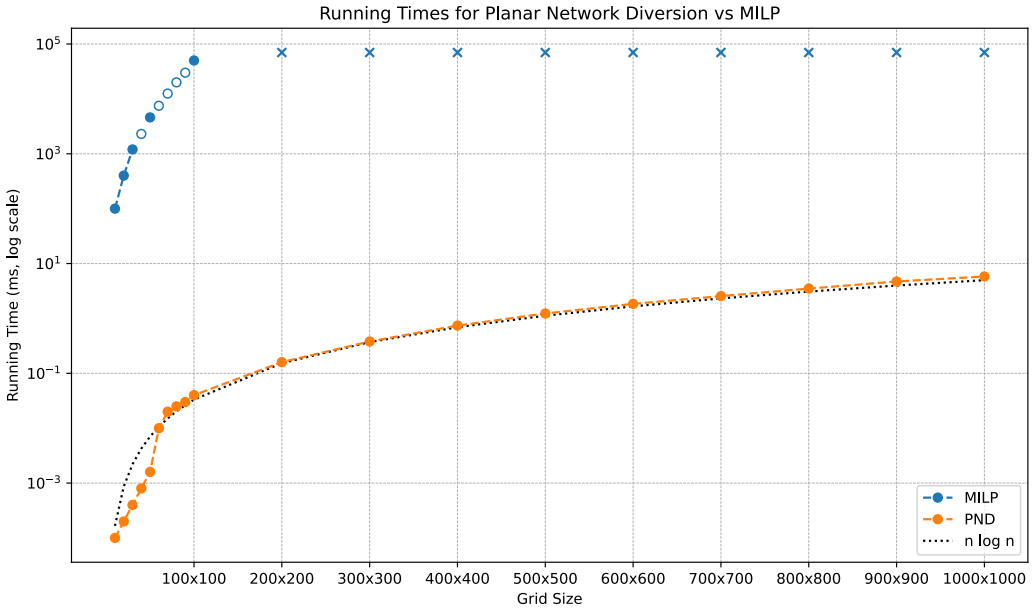


Figure 5 Comparison of the previous MILP algorithm by Cullenbine et al. [6] (blue, with filled circles for the values they provide and empty circles for interpolated values, and crosses for timeouts) and ours (orange) on grids of different sizes.

References

1. Geir Agnarsson and Raymond Greenlaw. *Graph theory: Modeling, applications, and algorithms*. Prentice-Hall, Inc., 2006.
2. Matthias Bentert, Pål Grønås Drange, Fedor V. Fomin, Petr A. Golovach, and Tuukka Korhonen. Two-sets cut-uncut in planar graphs. In *Proceedings of the 51st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 22:1–22:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ICALP.2024.22.
3. Matthias Bentert, Pål Grønås Drange, Fedor V. Fomin, and Steinar Simonnes. Planar network diversion source code, 2025. doi:10.5281/zenodo.15387001.
4. Ariel Cintron-Arias, Norman Curet, Lisa Denogean, Robert Ellis, Corey Gonzalez, Shobha Oruganti, and Patrick Quillen. A network diversion vulnerability problem. *IMA Preprints Series, University of Minnesota Twin Cities*, 2001. URL: https://hdl.handle.net/11299/3553.
5. Christopher Cullenbine, R. Kevin Wood, and Alexandra Newman. New results for the directed network diversion problem. In *Proceedings of the 10th Cologne-Twente Workshop on Graphs and Combinatorial Optimization (CTW)*, pages 134–137, 2011. URL: http://ctw2011.dia.uniroma3.it/ctw_proceedings.pdf#page=146.
6. Christopher A. Cullenbine, R. Kevin Wood, and Alexandra M. Newman. Theoretical and computational advances for network diversion. *Networks*, 62(3):225–242, 2013. doi:10.1002/net.21514.
7. Norman D. Curet. The Network Diversion Problem. *Military Operations Research*, 6(2):35–44, 2001. doi:10.5711/morj.6.2.35.
8. Ulrich Derigs. An efficient Dijkstra-like labeling method for computing shortest odd/even paths. *Information Processing Letters*, 21(5):253–258, 1985. doi:10.1016/0020-0190(85)90094-8.
9. Reinhard Diestel. *Graph Theory*. Springer, 2016.
10. Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965. doi:10.4153/CJM-1965-045-4.
11. Ozgur Erken. *A branch-and-bound algorithm for the network diversion problem*. PhD thesis, Naval Postgraduate School, 2002.
12. Chris Gray, Frank Kammer, Maarten Löffler, and Rodrigo I. Silveira. Removing local extrema from imprecise terrains. *Computational Geometry: Theory and Applications*, 45(7):334–349, 2012. doi:10.1016/j.comgeo.2012.02.002.
13. Pim van 't Hof, Daniël Paulusma, and Gerhard J. Woeginger. Partitioning graphs into connected parts. *Theoretical Computer Science*, 410(47-49):4834–4843, 2009. doi:10.1016/j.tcs.2009.06.028.
14. Yoichi Iwata and Yutaro Yamaguchi. Finding a shortest non-zero path in group-labeled graphs. *Combinatorica*, 42(S2):1253–1282, 2022. doi:10.1007/s00493-021-4736-x.
15. Benjamin S. Kallemyn. *Modeling Network Interdiction Tasks*. PhD thesis, Air Force Institute of Technology, 2015.
16. Yusuke Kobayashi and Sho Toyooka. Finding a shortest non-zero path in group-labeled graphs via permanent computation. *Algorithmica*, 77(4):1128–1142, 2017. doi:10.1007/s00453-016-0142-y.
17. Andrea S. Lapaugh and Christos H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14(4):507–513, 1984. doi:10.1002/net.3230140403.
18. Chungmok Lee, Donghyun Cho, and Sungsoo Park. A combinatorial Benders decomposition algorithm for the directed multiflow network diversion problem. *Military Operations Research*, 24(1):23–40, 2019. URL: https://www.jstor.org/stable/26609633.
19. Cynthia A. Phillips. The network inhibition problem. In *Proceedings of the 25th annual ACM Symposium on Theory of Computing (STOC)*, pages 776–785, 1993. doi:10.1145/167088.167286.
20. R. Kevin Wood. Deterministic network interdiction. *Mathematical and Computer Modelling*, 17:1–18, 1993. doi:10.1016/0895-7177(93)90236-R.

A Illustration of diverse cuts

Diverse minimal s-t-cuts computed on a Delaunay graph with 35 vertices and 91 edges with fixed s and t. The weights on the edges were chosen to be inverse proportional $1/\ell$ to its length. For each edge $e \in E(G)$, we run Network Diversion with input (G, s, t, e). Several instances yield the same solution, resulting in significantly fewer distinct solutions (34) than the 91 edges in the graph, which one might initially expect to correspond to the number of computed cuts. Most cuts were found twice with one cut being found nine times. Here, we have plotted all unique solutions, sorted by cost, with the cheapest being the true minimum s-t-cut. Total computation time was around 0.5 seconds. (Total elapsed time was 469 ms, with each computation taking 5.162 ± 2.79 ms, incl. call via `subprocess.run` from Python.)

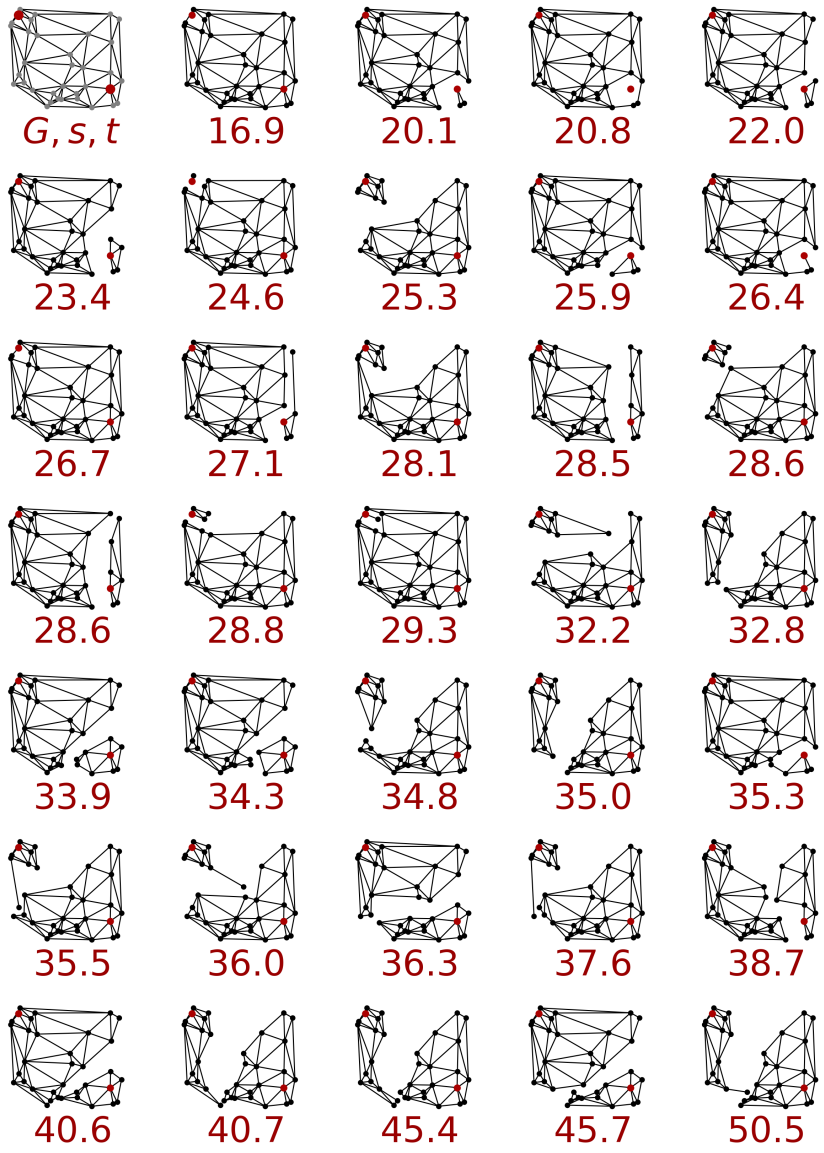


Figure 6 Diverse minimal s-t-cuts, with associated costs, computed on a Delaunay graph with 35 vertices and 91 edges.