University of Bergen
Department of Informatics

Diverting Networks with Odd Paths

Author: Steinar Simonnes
Supervisor: Pål Grønås Drange

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2024

Abstract

The problem of Shortest Odd Path is to find a path from one vertex to another in a graph, where the number of edges in the path is odd. Although the problem might seem like a mere curiosity, its utility lies in being applicable to many more useful problems that are easier if we know how to solve Shortest Odd Path.

One of these problems is Network Diversion: given a graph, two vertices, and a marked edge, compute the cheapest set of edges to delete from the graph such that all paths from one vertex to another must pass through the marked edge. Many of its variants are NP-complete, but its complexity on planar graphs remains an open problem.

We implement an efficient algorithm based on [Der85] to solve Shortest Odd Path on undirected graphs, and use that to implement the first-ever efficient algorithm for Network Diversion on planar graphs.

Contents

Chapter 1

Introduction

One of the most well-known, well-studied, and well-understood algorithmic problems is to find the Shortest Path in a graph. The problem is simple: given a graph and two vertices, find the shortest sequence of edges to go from one vertex to the other. Yet, the applications are almost limitless: finding the fastest route home, finding the cheapest airline tickets to Kuala Lumpur, solving a Rubik's Cube in the fewest moves, determining the best-case running time of an algorithm, or moving an enemy in a video game.

This thesis, however, is about a curious little variant called Shortest Odd Path. Here, we consider only paths consisting of an odd number of edges. If you were to step through the graph and start walking with your right foot, then an odd path is one where you would also end up on your right foot. The applications of this variant are not remotely as obvious. It rarely, if ever, matters whether a path has an odd or even length in any of the examples mentioned above, and it is difficult to come up with example problems where it does matter.

The reason we care is that many more useful problems are easier to solve if we already have an algorithm to solve Shortest Odd Path. Consider for example Shortest Detour Path: find the shortest path from one vertex to another, with the additional requierement that we are also given a 'detour' edge that must be included in the path. Imagine doing a road trip through Norway, but for the complete road trip experience you also really want to drive through Norway's longest tunnel, preferably without using the same roads more than once. Coming up with an algorithm for this is not as simple as it sounds, but we will show that it is much easier if we know how to solve Shortest Odd Path.

An even more directly useful problem to solve is Network Diversion: given two vertices and a marked edge in a graph, find the cheapest set of edges to delete from the graph such that all paths from one vertex to another must pass through the marked edge. This one has more immediate practical applications: imagine you are a military commander in a war, you know that the enemy wants to move their troops and supplies, and you are very prepared to ambush them if they cross a certain bridge. Now what is the fastest or cheapest way to destroy bridges to funnel the enemy through the ambush?

Solving Network Diversion efficiently is no simple task, and many of its variants have been proven to be NP-complete. Whether there exists an algorithm to solve Network Diversion in polynomial time in undirected *planar* graphs is for now an open problem, but it turns out the answer is yes: it is possible if we already have an efficient algorithm to solve Shortest Odd Path. This is the topic of our thesis. We develop and implement an efficient algorithm to solve Shortest Odd Path, and then use that to implement the first-ever efficient algorithm for Network Diversion on planar graphs.

Overview of the contents

We start this thesis with some preliminaries, mainly around graph theory, in Chapter 2. Then we warm up our problem-solving skills in Chapter 3, where we solve a much easier variant of Shortest Odd Path, called Shortest Odd Walk. In Chapter 4 we reach the star of the thesis: our algorithm for Shortest Odd Path. With this star leading the way, we can head to Chapter 5 to solve Network Diversion for planar graphs.

Most of the algorithms discussed here have also been implemented and tested in practice [Sim24b], and Chapter 6 presents the codebase. In addition, the source code for this thesis itself can be found at [Sim24a].

The curious reader may explore the chapters and source code in any order they wish, though we would like to suggest a chronological order if nothing else.

Chapter 2

Preliminaries

2.1 Graphs

In the study of algorithms, we often use graphs as an abstract structure to represent the fundamental algorithmic problem without distractions. For example, maybe we want to find the fastest route to walk to the study hall, or perhaps we want the cheapest combination of flights to take you to Kuala Lumpur. Both questions are really the same problem. If we remove all the details that are unnecessary to solve it, like the names of the airports and whether we are walking or flying, then we end up with a graph. This section defines various concepts related to graphs, and Section 2.2 will formalize the underlying problem of both of these examples as well as some other graph problems. Later, Section 2.3 defines the concept of *planar* graphs.

Definition 2.1.1 (Graph)**.** A *graph* $G := (V, E, \text{from}, \text{to})$ is given by

- V, a collection of *vertices*
- E, a collection of *edges*
- $\text{from} : E \to V$, a mapping from each edge to its source vertex
- $\text{to} : E \to V$, a mapping from each edge to its target vertex

For convenience, we also define the function $\text{reverse} : E \to E$. For an edge $e \in E$, $\text{reverse}(e)$ is the edge going in the opposite direction, where $\text{from}(e) = \text{to}(\text{reverse}(e))$ and $\text{to}(e) = \text{from}(\text{reverse}(e))$.

This definition of a graph is a little unusual. A more common definition is to instead define $G := (V, E)$, where the edges are a subset of the cartesian product of the vertices: $E \subseteq V \times V$. A problem with this definition is that we cannot consider graphs with parallel edges, which is something we will need to be able to do later in the thesis.

If we are working with multiple graphs at once, say two graphs G and H, then writing just V is ambiguous. In such cases, we instead denote $V(G)$ and $V(H)$ as G's and H's vertices, respectively. The same goes for $E(G)$ and $E(H)$ for their edges.

Definition 2.1.2 (Weighted graph)**.** A *weighted graph* $G := (V, E, \text{from}, \text{to}, \text{weight})$ is a graph, where $\text{weight} : E \to \mathbb{R}$ is the *weight* of each edge.

If a graph is not weighted, we often treat it as if all edges have unit weight, a weight of 1. Algorithms intended for weighted graphs will therefore often work on unweighted graphs as well. Although weights in the general case can be negative, all the algorithms presented in this thesis are designed for graphs of non-negative weights, and the reader may assume that $\text{weight} \ : E \to \mathbb{R}_{\geq 0}$ unless otherwise stated.

Definition 2.1.3 (Directed and undirected graphs)**.** Let G be a graph. G is said to be an *undirected graph* if each edge has an opposite: $\forall e \in E \ \exists e' \in E : \text{reverse}(e) = e'$. If G is not undirected, we say that G is a *directed graph.*

Most of the algorithms presented in this thesis are designed for undirected graphs. The reader may assume that all graphs in the thesis are undirected unless otherwise stated.

Definition 2.1.4 (Neighbourhood)**.** Let G be a graph, and let $u \in V$ be a vertex in the graph. The *neighbourhood* of u, denoted as $N(u)$, is defined as the vertices in G that are reachable from u by using just a single edge: $N(u) := \{to(e) \mid e \in E, \ \text{from}(e) = u\}$.

In code, it is usually more useful to consider neighborhoods in terms of edges. We will therefore denote $G[u]$ as the edges in G that start in u: $G[u] := \{e \mid e \in E, \ \text{from}(e) = u\}$. We also denote $\deg(u) \ := |N(u)|$ as the size of u's neighborhood, often referred to as the *degree* of u.

Definition 2.1.5 (Simple graph)**.** Let G be a graph. G is said to be a *simple* graph if for each pair of vertices $u, v \in V$, there exists *at most* one edge e such that $\text{from}(e) = u$ and $\text{to}(e) = v$. If two or more edges have the same endpoints, we say the edges are *parallel* to each other, and that the graph has *parallel* edges and is thus not simple.

Definition 2.1.6 (Walk)**.** A *walk* $P \ := [e_1, e_2, ..., e_k]$ in a graph G, for $e_i \in E$, is a sequence of edges where each edge ends where the next one starts: $\forall i \in \{1, 2, .., k-1\} \ : \text{to}(e_i) = \text{from}(e_{i+1})$. If $s := \text{from}(e_1)$ and $t := \text{to}(e_k)$, we say that P is an *s-t-walk* in G.

Another way to denote a walk is to use a sequence of vertices in the order they are visited: $[u_1, u_2, ..., u_n]$, for $u_i \in V$. This works as long as the graph is simple, if there are multiple edges from u_i to u_{i+1}, then the walk is ambiguous.

Definition 2.1.7 (Path)**.** A *path* $P := [e_1, e_2, ..., e_k]$ in a graph G is a walk with the extra requirement that each vertex is used at most once: $\forall i, j \in \{1, 2, .., k\} \ : j \neq i + 1 \rightarrow \text{to}(e_i) \neq \text{from}(e_j)$. If $s := \text{from}(e_1)$ and $t := \text{to}(e_k)$, we say that P is path from s to t, or an *s-t-path* in G.

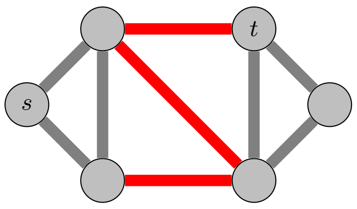
Note that in some literature, a walk is referred to as a path, and a path is referred to as a *simple* path. In this thesis, when we refer to paths they are always simple, meaning that they never repeat any vertices. If any vertices are repeated, we will refer to it as a walk.

Definition 2.1.8 (Cycle)**.** A *cycle* in a graph is a walk that starts and ends in the same vertex. If it does not repeat any vertices except in the last vertex, then we call it a *simple cycle*.

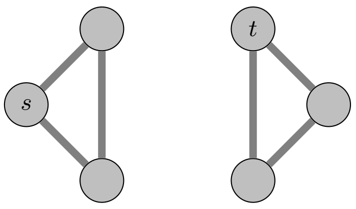
Definition 2.1.9 (The cost of a walk)**.** Let $P := [e_1, e_2, ..., e_k]$ be a walk in a weighted graph G. The *cost* of P is defined as the sum of the weights of its edges: $\sum_{i=1}^{k} \text{weight}(e_i)$. In a collection of walks, we say that the *shortest* walk is the *cheapest* one, the one with the lowest cost. Likewise for the longest and most expensive walk.

Note that in some literature, *shortest* may instead mean *fewest edges*, and the term *length* could refer to both the number of edges and the cost. For that ambiguous reason, we will from now on avoid the word *length*, and *shortest* will always mean *cheapest*. If a graph is unweighted, we pretend that all the edges have a unit weight of 1, and in that case, the cost is the same as the number of edges.

Definition 2.1.10 (Cut)**.** Let $G = (V, E, \text{from}, \text{to})$ be a connected and undirected graph. A *cut* $C \subseteq E$ of G is a subset of edges such that $(V, E \setminus C, \text{from}, \text{to})$ is a disconnected graph of exactly two components. If two vertices $s, t \in V$ end up in separate components after the cut, we denote C as an *s-t-cut* in G. See Figure 2.1 for an example of an s-t-cut.



(a) A connected graph with an s-t-cut marked in red



(b) The disconnected graph of exactly two components, after deleting all the edges in the cut

Figure 2.1: A graph before and after an s-t-cut of edges has been deleted

2.2 Graph problems

Now that we know what a graph is, we are ready to formalize the underlying problem of the examples we started the chapter with. Both problems can be represented by the quest to find the shortest path from one vertex to another in an abstract graph. From a computational perspective, it does not matter whether the edges are roads or flights, or whether the vertices are crossroads or airports. Vertices and edges can represent whatever we want them to. We call the problem Shortest Path, and it is defined as:

Shortest Path
Input: a graph G, two vertices $s, t \in V$
Output: an s-t-path in G of minimum cost

This thesis will focus on a curious variant of the Shortest Path problem, called Shortest Odd Path:

Shortest Odd Path
Input: a graph G, two vertices $s, t \in V$
Output: an s-t-path in G of minimum cost, that uses an odd number of edges

We will also present an algorithm in Chapter 3 for the less restrictive variation called Shortest Odd Walk:

Shortest Odd Walk
Input: a graph G, two vertices $s, t \in V$
Output: an s-t-walk in G of minimum cost, that uses an odd number of edges.

Dijkstra's Algorithm

Both our algorithms for Shortest Odd Path and Shortest Odd Walk borrow ideas from the famous Dijkstra's Algorithm. The algorithm solves Shortest Path on graphs with non-negative weights and handles both directed and undirected graphs. We show it here for reference.

Code Listing 2.1: Dijkstra's Algorithm for Shortest Path

```
fn dijkstras_shortest_path(graph, s, t) {
    for u in V(graph) {
        dist[u] = ∞;
        done[u] = false;
    }
    dist[s] = 0;
    queue = priority_queue((0, s));

    while queue is not empty {
        (dist_u, u) = queue.pop();
        if not done[u] {
            done[u] = true;
            for edge in graph[u] {
                dist_v = dist_u + weight(edge);
                if dist_v < dist[v] {
                    dist[v] = dist_v;
                    queue.push((dist_v, v));
                }
            }
        }
        if done[t] {
            break;
        }
    }

    return dist[t];
}
```

2.3 Planarity

A fascinating and important class of graphs that we will focus on in Chapter 5 are *planar graphs*. We will give the most important definitions and facts about planar graphs here, and refer the curious reader to [Nis88] if they wish to read more.

2.3.1 Planar embeddings

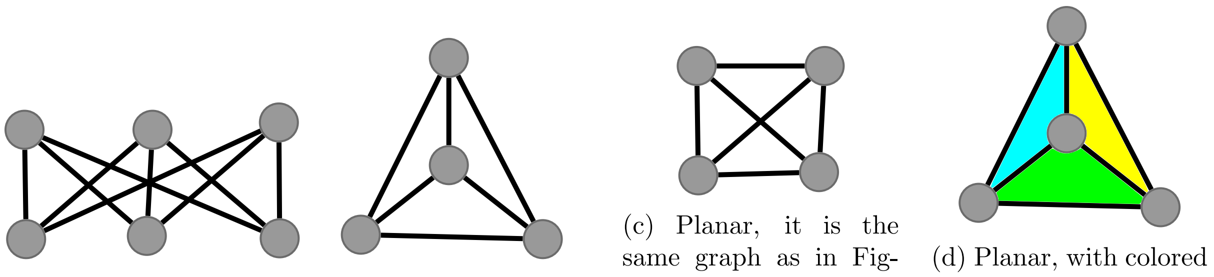
Definition 2.3.1 (Embedding)**.** Let G be a graph. An *embedding* of G is a drawing of G on the plane $\mathbb{R}^2$, with points representing vertices and curves representing edges between their endpoints' respective points, such that none of the edges intersect each other except in their endpoints.

Definition 2.3.2 (Planar graph)**.** We say that a graph G is a *planar graph* if there exists a planar embedding of G. A planar graph along with a specific planar embedding is called a *plane graph*.

Many real-life graphs, especially those based on physical structures, are either planar or almost so. Two edges crossing often entails an inefficiency or extra cost: like having to build a bridge over a road instead of joining the two roads in a crossroad. Many algorithmic problems are much easier to solve in planar graphs, and they are common enough in practical use that the restriction is not too restrictive to be useful.

Definition 2.3.3 (Straight-line embedding)**.** Let G be a graph. A *straight-line embedding* of G is a planar embedding of G where each edge can be drawn as a line segment between its endpoint vertices and still not cross any other edge. In a straight-line embedding, we can forgo the mappings of the edges altogether and consider the mapping of vertices only. Such embeddings always exist: if G is planar then there is a straight-line embedding of G.

See Figure 2.2b, Figure 2.2c and Figure 2.2d for an example of a planar graph. Figure 2.2b and Figure 2.2d also show planar embeddings of the graph. Figure 2.2a shows a graph that is not planar, since no planar embeddings of the graph exist. Note that in all these examples we have


(a) Not planar

(b) Planar

(c) Planar, it is the same graph as in Figure 2.2b

(d) Planar, with colored faces.

Figure 2.2: Examples of planar and non-planar graphs

drawn all the edges as straight line segments, but that is not necessary: as long as a curve does not cross any other curves it can be as curved as we want.

It is generally complicated to determine if a given graph is planar in practice, and to compute an appropriate embedding if it is. In all the algorithms of this thesis that take planar graph as input, we will assume that we are given embeddings of the graphs as well. Furthermore, since all planar graphs also have a straight-line embedding, we will assume that the given embeddings are straight-line embeddings. Our theoretical results hold for planar graphs in general, but in practice, these assumptions make implementing the algorithms less tedious.

2.3.2 Duality

The next topic is easy to visualize and understand, but challenging to formalize. Imagine loading a drawing of a planar graph like the one in Figure 2.2b into an image editing program, and using the fill tool to cover each region in a different color, like in Figure 2.2d. Each such region is called a *face* of the graph, including the region 'outside' the graph called the *outer* face. Two faces are adjacent if they are separated by just a single edge: if we were to delete the edge our fill tool would give both the same color. We will now formalize this concept.

Definition 2.3.4 (Face)**.** Let G be a plane graph. A *face* of G is a region in the embedding bounded by a cycle that contains no other vertices or edges. Equivalently, we can define faces as the connected components that remain in $\mathbb{R}^2$ after we delete all vertices and edges from our embedding.

Note that different embeddings of the same planar graph may yield different faces. When we refer to a face in a graph, it is always in relation to a certain embedding of the graph.

Definition 2.3.5 (Duality of planar graphs)**.** Let G be a plane graph. The *dual graph* of G, denoted as $G^\star$, is the graph where

- The vertices represent faces of G

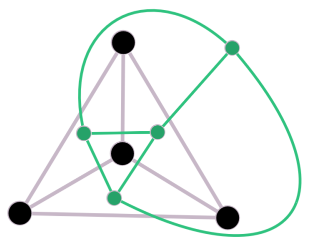
- There is an edge between two faces if they are adjacent in G.

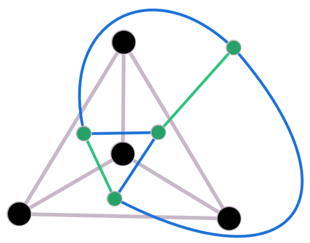
Each edge in $e \in E(G)$ will always have a face on either side, possibly the same face, and thus have a corresponding edge $e^\star \in E(G^\star)$ in the dual graph. We can therefore define two convenience functions $\text{left}, \text{right} : E(G) \to V(G^\star)$ to get the left and right faces of an edge, respectively. If G is weighted, we usually set the weights of $E(G^\star)$ according to their counterparts: $\text{weight}(e^\star) := \text{weight}(e)$. See Figure 2.3a for an example of a dual graph.

Note that the dual graph is also planar, and the dual of the dual is the original graph[1]. We could then for example let e be an edge in the dual graph, and then refer to its real counterpart as $e^\star$. It would not be wrong, but it could possibly lead to confusion. Furthermore, we do not need that fact for any of the results in this thesis. We will therefore give variable names like G, u, and e for the graphs, vertices, and edges that we are "working on", and use variable names like $G^\star$, $u^\star$ and $e^\star$ for their dual equivalents only in intermediary computations before arriving at results for our original graph.

Fact 2.3.1 (Cycle–cut duality)**.** Let $G := (V, E, \text{from}, \text{to})$ be a connected planar graph, and let $C^\star$ be a simple cycle in $G^\star$. Then $C^\star$ will always correspond to a minimal cut in G. If we define $C := \{e \mid e^\star \in E(G^\star)\}$ as the edges in $E(G)$ that correspond to edges in $C^\star$, then $(V, E \setminus C, \text{from}, \text{to})$ is a disconnected graph of exactly two components. If the cycle $C^\star$ is not simple, then we still end up with a disconnected graph, but we may have more than just two components.

See Figure 2.3 for an example. In Figure 2.3b we have found a simple cycle in the dual graph, and if we delete the corresponding edges in the original graph we end up with the disconnected graph in Figure 2.3c.


(a) A planar graph with its dual colored in green


(b) A simple cycle in the dual, colored in blue

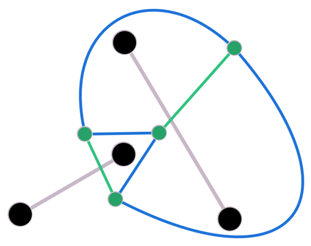

(c) Deleting the edges that cross the dual's cycle cuts the graph in two

Figure 2.3: A simple cycle in a dual graph always corresponds to a cut in the original graph.

Theorem 2.3.1 (The relation between the number of vertices, edges and faces)**.** Let $G := (V, E, \text{from}, \text{to})$ be a connected planar graph, where $n := |V|, m := |E|$, and f is the number of faces in any embedding of G.
Claim: $n + f - m = 2$.

[1]Strictly speaking, there is an embedding of $G^\star$ such that $G^{\star\star} \cong G$.

Proof. Let $H \sqsubseteq G$ be any non-empty connected subgraph that does not have any cycles, and let $n_H := |V(H)|$. Since it does not have any cycles, we have that:

- The outside face must be the only face: $f_H := 1$.
- Each edge must connect a 'new' vertex to the rest of the graph, except the first edge which connects two new vertices. Therefore the number of edges is one less than the number of vertices: $m_H := n_H - 1$.

We now have that $n_H + f_H - m_H = n_H + 1 - (n_H - 1) = 2$, so the equality holds for this subgraph.

Now we can iteratively add either an edge alone or both a vertex and an edge to H until we have G. If we add just an edge, we increase both m_H and f_H by 1 and the equality still holds. If we add a new vertex with a new edge, we increase both n_H and m_h by 1 and the equality still holds.

Therefore, the equality $n + f - m = 2$ holds for any connected planar graph G. □

Corollary 2.3.1. The number of faces is fixed. A graph may have different faces depending on the embedding, but the number of faces is always the same.

Corollary 2.3.2. Since all faces (except possibly the outer face) are bounded by at least three edges, and all edges touch at most two faces, we can show that if $n \geq 3$, then $m \leq 3n - 6$.

Chapter 3

Shortest Odd Walk

Before we start on the main topic of this thesis, we want to discuss a closely related problem:

Shortest Odd Walk
Input: A weighted graph G, two vertices $s, t \in V$
Output: the shortest s-t-walk in G that uses an odd number of edges

The difference is simple: a walk may use the same vertices multiple times, whereas a path can not. A naïve attempt at solving Shortest Odd Path will often accidentally use the same vertices multiple times, which would make it a walk rather than a path. Therefore, we want to present an algorithm to solve Shortest Odd Walk first, and explain why it does not solve Shortest Odd Path.

3.1 Intuition

Our algorithm will take inspiration from Dijkstra's algorithm for Shortest Path, and assume that all the edges have non-negative weights. As seen in Section 2.2, in Dijkstra's algorithm we have an array to keep the tentative best distance to each vertex. In this algorithm we will keep two such arrays. The first is for the best distance using an odd walk, and the second is for the best distance using an even walk. Each vertex can be scanned at most twice: once when we have found the definitive best odd walk and want to find potential improvements to the even walks of its neighbors, and analogously when we find the best even walk.

3.2 Pseudocode

Here is the pseudocode of the algorithm. To see the code implemented in Rust, see the GitHub repository [Sim24b].

Code Listing 3.1: Shortest Odd Walk

```
def shortest_odd_walk(graph, s, t) {
    for u in 0..n {
        even_dist[u] = ∞
        odd_dist[u] = ∞
        even_done[u] = false;
        odd_done[u] = false;
    }
    even_dist[s] = 0

    queue = priority_queue([(0, true, s)]);
    while queue is not empty {
        (dist_u, even, u) = queue.pop()
        if even {
            if even_done[u] continue;
            even_done[u] = true;

            for edge in graph[u] {
                v = to(edge);
                dist_v = dist_u + weight(edge);
                if dist_v < odd_dist[v] {
                    odd_dist[v] = dist_v;
                    queue.push((dist_v, false, v));
                }
            }
        }
        else {
            if odd_done[u] continue;
            odd_done[u] = true;

            for edge in graph[u] {
                v = to(edge);
                dist_v = dist_u + weight(edge);
                if dist_v < even_dist[v] {
                    even_dist[v] = dist_v;
                    queue.push((dist_v, true, v));
                }
            }
        }
        if odd_dist[t] < ∞ {
            return odd_dist[y];
        }
    }
    return None;
}
```

In the pseudocode, we show how to find the best odd walk from the source vertex to the target vertex. If we instead want to find the best odd or even walks to all vertices, we can simply remove the if-clause around the target, and return the arrays instead.

3.3 Analysis

Consider Figure 3.1. There are no odd paths from s to t, but we have an infinite amount of odd walks by utilizing the cycles $[a, b, c]$ or $[a, c, b]$ an odd number of times to offset the parity. Our

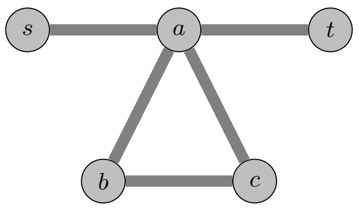


Figure 3.1: No odd s-t-path exist, yet we still have many odd s-t-walks.

algorithm would perhaps first find an odd walk to a, then an even walk to b, then an odd walk to c, then an even walk to a, and lastly an odd walk to t. This is one of the two odd s-t-walks of minimum cost. However, a is visited twice in the walk, once for each parity, and the resulting walk is not a path. Therefore, this algorithm cannot be used to solve Shortest Odd Path.

The main limitation of the algorithm is that the edges in the input graph must have either non-negative weights or no weights at all. Otherwise, we cannot guarantee that `even_dist[u]` and `odd_dist[u]` have their final, correct values when we scan a vertex u. Note that unlike most other algorithms shown in this thesis, this algorithm does not require the input graph to be undirected, it may also be directed.

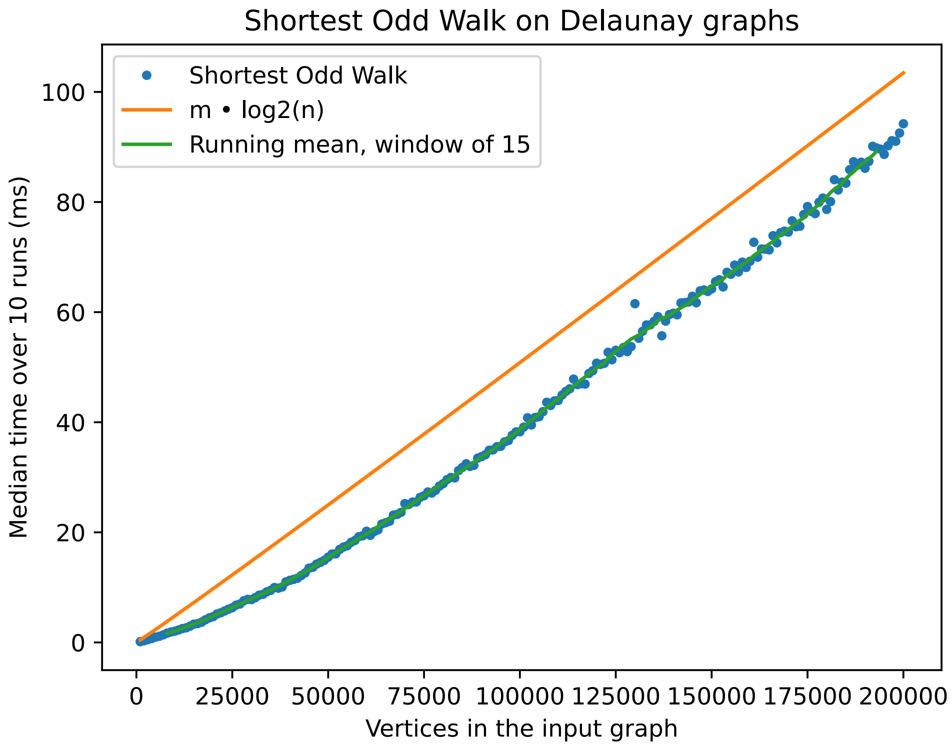
Theorem 3.3.1. Let (G, s, t) be an instance of Shortest Odd Walk, let $n := |V|$ and let $m := |E|$.
Claim: the algorithm runs in time at most $O(m \cdot \log m)$, or $O(m \cdot \log n)$ if the graph is simple.

Proof. Because of our `odd_done` and `even_done` arrays, we can guarantee that each vertex is scanned at most twice, once for each parity. For each scan, we loop through each of the neighbors in linear time and consider putting them in the queue. The total cost of the scans is therefore at most $O(m)$. A vertex may be put into the queue many times before it is scanned, in the worst case once for each of its neighbors. That means that we put vertices in the queue at most $O(m)$ times, for a total cost of $O(m)$, and removing all of them takes a total of $O(m \cdot \log m)$.

The algorithm runs in time at most $O(m) + O(m \cdot \log m) = O(m \cdot \log m)$, which shows the first part of the claim.

If the graph is simple we may simplify the complexity further: $O(m \cdot \log m) \subseteq (m \cdot \log n^2) = O(m \cdot 2 \cdot \log n) = O(m \cdot \log n)$, which shows the second part of the claim. □

To test how well the algorithm scales, we generate 200 Delaunay graphs of sizes 1000, 2000, 3000, and so on until 200k. We explain how these graphs are generated in Section 6.4. For each graph, we have estimated a source and target with the maximum shortest path between them, and run our Shortest Odd Walk algorithm. We take the median running time of 10 runs and plot the results below. We have also tried to find constants to convert the $O(m \cdot \log n)$ theoretical complexity into a comparable function and plotted it next to the real practical results.



As we can see, the algorithm easily solves Shortest Odd Walk on graphs of 200k+ vertices in less than 100ms. There is only a slight upward curve as the inputs grow, which is what we expect from a linearthmic theoretical running time.

We also benchmark the algorithm on seven graphs from real-life scenarios, as seen in the table below. Four of the synthetic Delaunay graphs are added for comparison. Each graph is run 20 times, and we take the average running times and show them in the table below. See Section 6.4 for more details on how the benchmarking is done.

Graph	n	m	Time spent
Power BCSPWR09 [RA15]	1723	2394	$<$ 1ms
Oldenburg [LCH+05]	6106	7035	1ms
San Joaquin County [LCH+05]	18263	23874	3ms
Cali Road Network [LCH+05]	21048	21693	3ms
Musae Github [RAS19]	37700	289003	18ms
SF Road Network [LCH+05]	174956	223001	46ms
Ca Citeseer [RA15]	227321	814137	132ms
Delaunay 50k	50000	149961	15ms
Delaunay 100k	100000	299959	38ms
Delaunay 150k	150000	449965	64ms
Delaunay 200k	200000	599961	92ms

As we can see, the algorithm does very well on all the inputs, and notably has no trouble solving for the Ca Citeseer graph of 227k vertices and 814k edges in less than 150ms. These results are excellent. Despite having a similar complexity to the other algorithms in this thesis, this algorithm is still by far the fastest in practice.

Though we discovered it independently, the algorithm is not particularly groundbreaking or in any way creative. Therefore, we do not expect it to be original. It is, however, quite fast, and we are happy with that. The main reason we include it in this thesis is because of its pedagogical value in introducing our main topic: SHORTEST ODD PATH.

Chapter 4

Shortest Odd Path

Now that we have tried out some algorithms for Shortest Odd Walk, we are finally ready to add the restriction that each vertex is used at most once, and thus solve Shortest Odd Path:

Shortest Odd Path
Input: a weighted graph G, two vertices $s, t \in V$
Output: an s-t-path in G of minimum cost, that uses an odd number of edges

Even though it is deceptively simple to state the problem, solving it in practice is no easy task. In fact, [Tho85] has shown that the problem is NP-complete in directed graphs, even without weights. Undirected graphs are also a challenge: [SS23] have proven that even if the weights are restricted to just $\{-1, 1\}$, and even if we know that they do not form any negative cycles, Shortest Odd Path is still NP-complete in undirected graphs.

Knowing this, we instead direct our focus to a simpler variant: Shortest Odd Path on undirected graphs of non-negative weights. Unlike the other variants, this one *is* possible to solve in polynomial time, as shown by [Tho85], [SS23], [Der85] and others. The algorithm we are about to present is based on Derigs' algorithm [Der85], along with some minor improvements.

4.1 Reduction to Shortest Alternating Path

Consider first another related problem:

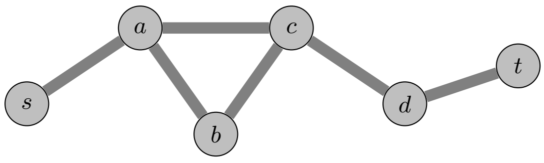
Shortest Alternating Path
Input: a weighted graph $G := (V, E, \text{from}, \text{to}, \text{weight})$, two vertices $s, t \in V$, and a set $F \subseteq E$
Output: an s-t-path in G of minimum cost, where every other edge used is in F

Derigs observed that Shortest Odd Path can be reduced to a special case of Shortest Alternating Path, by constructing what we will refer to as a *mirror graph*.

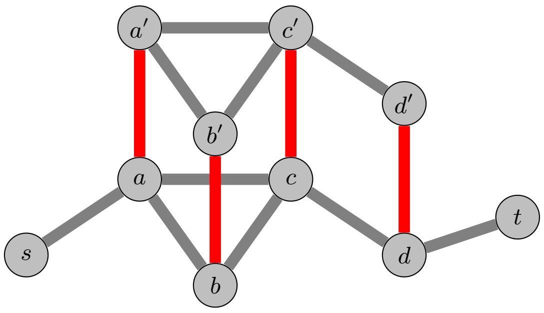
Definition 4.1.1 (Mirror graph)**.** Let G be a graph, and $s, t \in V(G)$ be two vertices. We construct a supergraph $M \sqsupset G$, by adding an extra copy of everything in the graph not directly related to s and t. For each vertex $u \in V(G) \setminus \{s, t\}$, we add a 'mirror' vertex u', and a connecting 'mirror' edge of weight 0 between u and u'. For each edge $e \in E(G) \setminus (G[s] \cup G[t])$ from u to v we add an edge e' of the same weight between the 'mirror' copies of its endpoints, from u' to v'. Now M is the mirror graph of G with respect to s and t.

See Figure 4.1 for an example. If G is the graph in Figure 4.1a, then the mirror graph could look like Figure 4.1b. The part of the mirror graph M that is also present in G is referred to as the 'real' side of the graph, while the new vertices are on the 'mirror' side. Vertices and edges from the mirror side are usually labeled with an $'$. For convenience, we define the function mirror : $V(M) \setminus \{s, t\} \rightarrow V(M) \setminus \{s, t\}$ to go from a vertex to its counterpart on the other side of the mirror. Furthermore, in an abuse of notation, we also define mirror : $E(M) \setminus (G[s] \cup G[t]) \rightarrow E(M) \setminus (G[s] \cup G[t])$ as the same but for edges.

The edges in M that 'cross' the mirror by going between vertices and their counterparts form a matching in M. We refer to these edges as *the edges in the matching*, or sometimes simply as just *the matching*. Note that with the exception of s and t, this is an almost perfect matching in M.



(a) The input graph G



(b) The mirror graph M of G, with the matching marked in red

Figure 4.1: Reduction from Shortest Odd Path to Shortest Alternating Path.

Our reduction from Shortest Odd Path to Shortest Alternating Path follows:

1. Let (G, s, t) be an instance of Shortest Odd Path.
2. Construct M as the mirror graph of G, and let F be the matching in M. Now (M, s, t, F) is an instance of Shortest Alternating Path.
3. Let P' be the shortest alternating path of (M, s, t, F), if one exists. If none exist, then we do not have any odd s-t-paths in G either and are already done.

4. Construct P by filtering out the edges in the matching from P', and for each edge $e' \in E(M)$ from the mirror side of M we replace it by the corresponding edge $\text{mirror}(e') \in E(G)$ from the real side.

5. Now P is the shortest odd s-t-path in G.

For example, if our input G for SHORTEST ODD PATH is Figure 4.1a, then M and F could look like Figure 4.1b. The only alternating s-t-path is $P' := [(s,a),(a,a'),(a',b'),(b',b),(b,c),(c,c'),(c',d'),(d',d),(d,t)]$. When we translate it to a path in G, we end up with $P := [(s,a),(a,b),(b,c),(c,d),(d,t)]$, which is the shortest odd s-t-path in G.

Now that we have two copies of most vertices in M, we run the risk of accidentally using the same vertex multiple times and ending up with a walk rather than a path in G, like with our algorithm in Chapter 3. The key to note here is that F is an (almost) perfect matching, and when we step on a vertex u we have to cross the mirror and step on $\text{mirror}(u)$ next. We will never visit u, go somewhere else, and then later come back to visit $\text{mirror}(u)$. So both copies must be used directly after each other, and when we translate the path in M into a path in G the two copies are effectively merged into just a single step in the path. Therefore, vertices are never repeated and we always end up with a path.

To see why the reduction necessarily yields an *odd* path, simply observe that for each step we take in the graph, we have to go to the other side of the mirror. If we take another step, we get back to the same side again. It is only when we reach the target vertex t that we can stop and not have to go to the other side. Therefore, to reach a neighbor of t, we must have used an even number of edges from the matching and an even number of edges not in the matching. When we take the last step to reach t we have used an odd number of edges and thus found an odd path. If this alternating s-t-path in M is the shortest such path, then the corresponding path in G must also be the *shortest* odd s-t-path in G. The interested reader may see [Der85] for more details on this reduction.

Ball and Derigs [BD83] have shown how to efficiently solve SHORTEST ALTERNATING PATH. In their algorithms, subgraphs are shrunk into pseudonodes whenever possible, to make the graph smaller. The drawback is that certain pseudonodes must later be expanded again, which is the most complicated and expensive part of their algorithms. In our case, however, we have a special case of SHORTEST ALTERNATING PATH. The set F is, except for s and t, a perfect matching of M, and we will therefore never have to expand pseudonodes after shrinking them. The curious reader may visit [BD83] for more on these algorithms and why our almost-perfect matching is a simpler case.

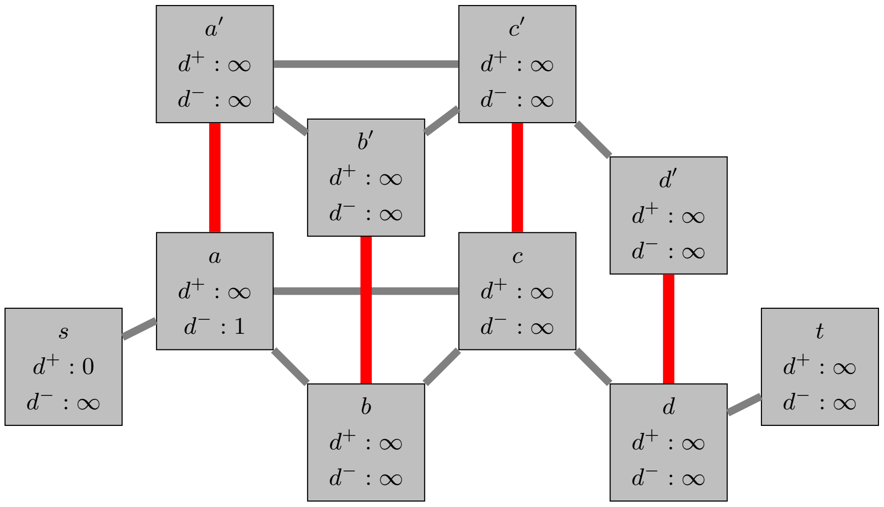
4.2 The idea for our Shortest Alternating Path algorithm

We will explain the general idea of our algorithm by following an example, and solve for the graph in Figure 4.1a. First, we construct the mirror graph as explained in Section 4.1, to produce the graph in Figure 4.1b. Then we initialize an empty priority queue of vertices and edges to be scanned. For each vertex $u \in V(M)$, we denote

- d_u^+ := the length of the shortest alternating s-u-path ending on a matched edge
- d_u^- := the length of the shortest alternating s-u-path ending on a non-matched edge
- pred_u := the last edge used to find u's most recent value for d_u^-

Initially, these are either ∞ or undefined, except for the source vertex s, where we can set $d_s^+ := 0$. Then, for each edge $e \in N(s)$, we can set $d_{\text{to}(e)}^- := \text{weight}(e)$, $pred_{to(e)} := e$, and add $\text{to}(e)$ to our priority queue with priority $2 \cdot \text{weight}(e)$.

We visualize it in the diagram below.

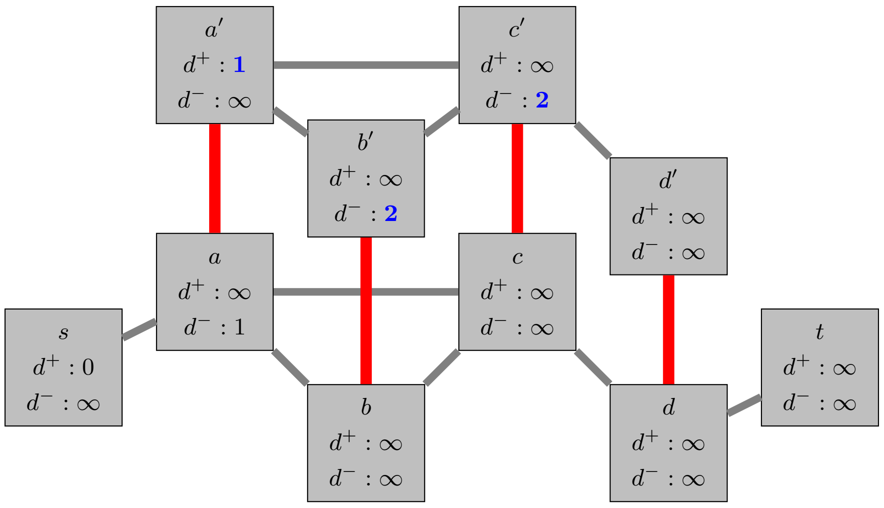


a' $d^+ : \infty$ $d^- : \infty$
c' $d^+ : \infty$ $d^- : \infty$
b' $d^+ : \infty$ $d^- : \infty$
d' $d^+ : \infty$ $d^- : \infty$
a $d^+ : \infty$ $d^- : 1$
c $d^+ : \infty$ $d^- : \infty$
s $d^+ : 0$ $d^- : \infty$
t $d^+ : \infty$ $d^- : \infty$
b $d^+ : \infty$ $d^- : \infty$
d $d^+ : \infty$ $d^- : \infty$

Queue:

- Vertex(2,a)

The first and only vertex in the queue is a. We pop it, set $d_{a'}^+ := d_a^-$, and 'scan' a'. By that, we mean to look at each neighbor $e \in G[a']$, and see if our new value $d_{a'}^+ + \text{weight}(e)$ is better than the previous value $d_{\text{to}(e)}^-$. That is the case for both b' and c', so we update their values and add them to the queue. Their priorities in the queue are equal to twice their d^- values, which is $2 \cdot 2 = 4$ for both of them.

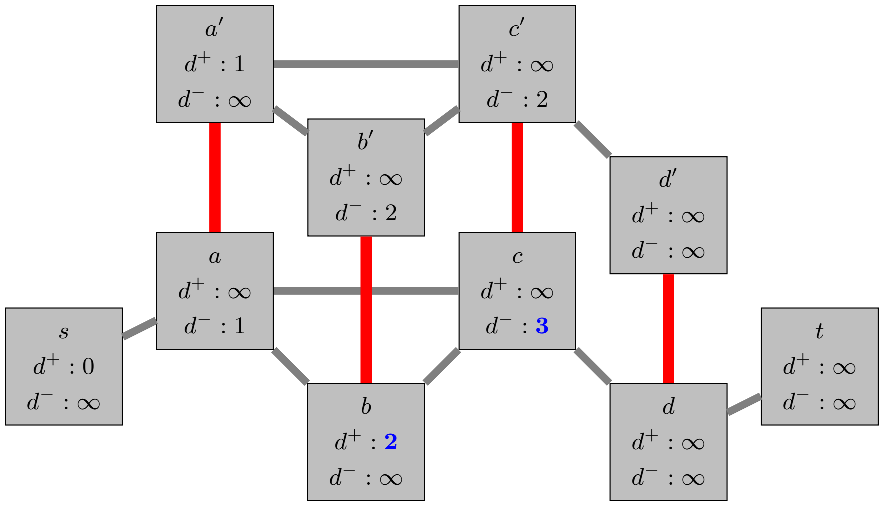


a' $d^+ : 1$ $d^- : \infty$
c' $d^+ : \infty$ $d^- : 2$
b' $d^+ : \infty$ $d^- : 2$
d' $d^+ : \infty$ $d^- : \infty$
a $d^+ : \infty$ $d^- : 1$
c $d^+ : \infty$ $d^- : \infty$
s $d^+ : 0$ $d^- : \infty$
t $d^+ : \infty$ $d^- : \infty$
b $d^+ : \infty$ $d^- : \infty$
d $d^+ : \infty$ $d^- : \infty$

Queue:

- ~~Vertex(2,a)~~
- Vertex(4,b')
- Vertex(4,c')

The next vertex in the queue is b', so we set $d_b^+ := d_{b'}^-$ and scan b':

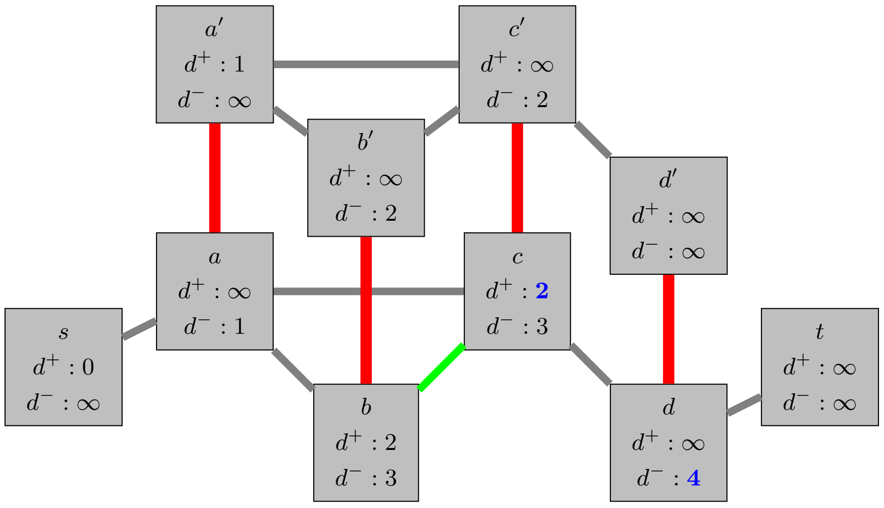


a' $d^+ : 1$ $d^- : \infty$
c' $d^+ : \infty$ $d^- : 2$
b' $d^+ : \infty$ $d^- : 2$
d' $d^+ : \infty$ $d^- : \infty$
a $d^+ : \infty$ $d^- : 1$
c $d^+ : \infty$ $d^- : 3$
s $d^+ : 0$ $d^- : \infty$
t $d^+ : \infty$ $d^- : \infty$
b $d^+ : 2$ $d^- : \infty$
d $d^+ : \infty$ $d^- : \infty$

Queue:

- ~~Vertex(4,b')~~
- Vertex(4,c')
- Vertex(6,c)

Now c' is the next in the queue, we set $d_c^+ := d^-c'$ and scan c'. This is where the interesting part happens: now we have set both d^+ and d^- for b and c, and that means that we have found an odd cycle in the graph. The edge between them, e, is called the *blossom edge*, and is marked in green. We add e to the queue, with the priority $d_c^+ + d_b^+ + \text{weight}(e)$.

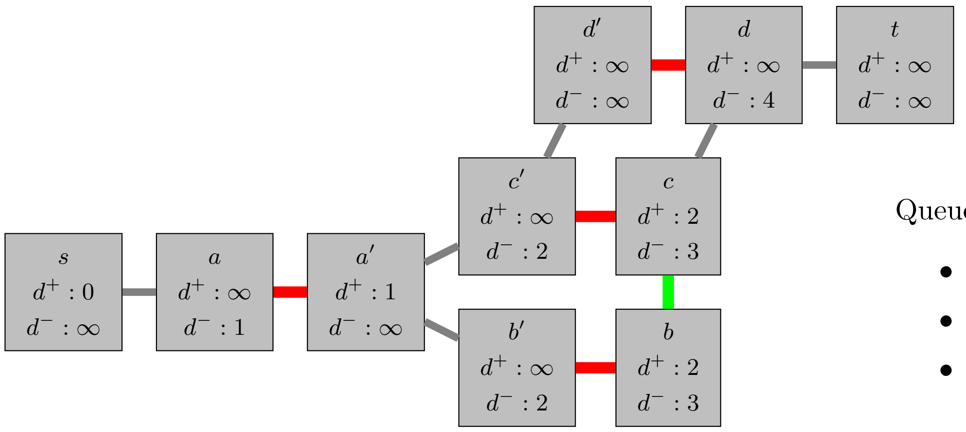


a' $d^+ : 1$ $d^- : \infty$
c' $d^+ : \infty$ $d^- : 2$
b' $d^+ : \infty$ $d^- : 2$
d' $d^+ : \infty$ $d^- : \infty$
a $d^+ : \infty$ $d^- : 1$
c $d^+ : 2$ $d^- : 3$
s $d^+ : 0$ $d^- : \infty$
t $d^+ : \infty$ $d^- : \infty$
b $d^+ : 2$ $d^- : 3$
d $d^+ : \infty$ $d^- : 4$

Queue:

- ~~Vertex(4,c')~~
- Blossom(5,(b,c))
- Vertex(6,c)
- Vertex(6,d)

Next up is to scan this blossom edge, and compute its corresponding odd cycle by backtracking from c and b until they meet at a'. To visualize the cycle, we like to 'stretch out' the graph a little, and draw it like below. Note that some of the edges are omitted for clarity. Now we can see that the cycle consists of $[a', c', c, b, b', a']$. We call the set $\mathbb{B} := \{c', c, b, b'\}$ a *blossom*, and a' the *base* of the blossom, inspired by the famous Blossom algorithm by [Edm65].



d' $d^+ : \infty$ $d^- : \infty$
d $d^+ : \infty$ $d^- : 4$
t $d^+ : \infty$ $d^- : \infty$
c' $d^+ : \infty$ $d^- : 2$
c $d^+ : 2$ $d^- : 3$
s $d^+ : 0$ $d^- : \infty$
a $d^+ : \infty$ $d^- : 1$
a' $d^+ : 1$ $d^- : \infty$
b' $d^+ : \infty$ $d^- : 2$
b $d^+ : 2$ $d^- : 3$

Queue:

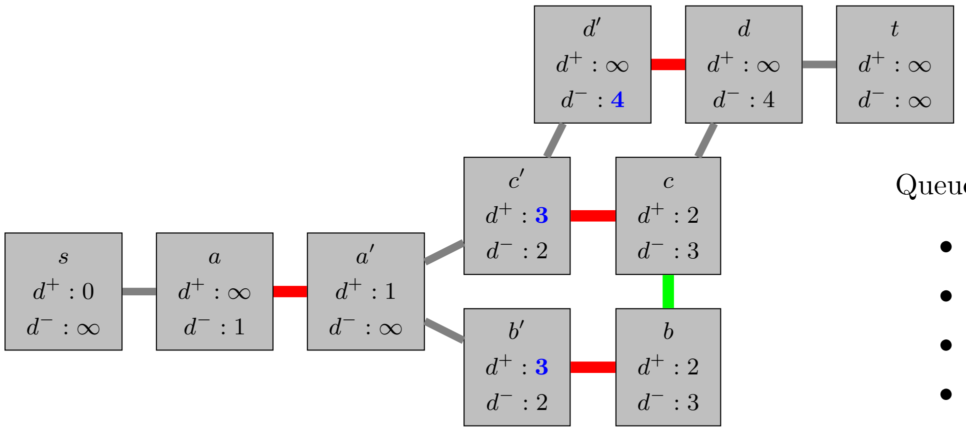
- Blossom(5,(b,c))
- Vertex(6,c)
- Vertex(6,d)

The first reason why we care about this blossom is because now we can immediately set the final, optimal d^- and d^+ for all the vertices in the blossom. That is because we now have two alternating paths to each vertex, one goes around the cycle while the other takes the shortcut. One of these ends up on a matched edge, and the other on a normal edge. Furthermore, both of these are the optimal paths and can be used to set final values for d^+ and d^-. For example, to go from s to c', we can either go through $[s, a, a', c']$ with a cost of d_c^-, or go along $[s, a, a', b', b, c, c']$ with a cost of d_c^+.

More specifically, for each $u \in \mathbb{B}$:

- If $d_u^+ = \infty$, we set $d_u^+ = d^-_{\text{mirror}(u)}$.
- If we can improve d_u^- by coming from its neighbor in the blossom, we do so.

After all these values have been set, we immediately scan all the vertices in $\mathbb{B}$ that just received values for d^+. In this example, we scan c' and b', and discover d'. Unfortunately, since this is a very small blossom we don't have any vertices that receive new values for d^-.

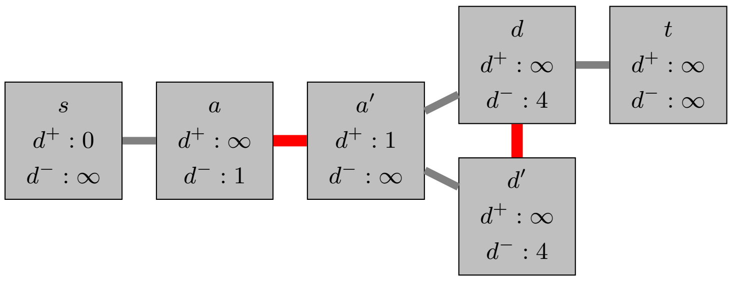


d' $d^+ : \infty$ $d^- : 4$
d $d^+ : \infty$ $d^- : 4$
t $d^+ : \infty$ $d^- : \infty$
c' $d^+ : 3$ $d^- : 2$
c $d^+ : 2$ $d^- : 3$
s $d^+ : 0$ $d^- : \infty$
a $d^+ : \infty$ $d^- : 1$
a' $d^+ : 1$ $d^- : \infty$
b' $d^+ : 3$ $d^- : 2$
b $d^+ : 2$ $d^- : 3$

Queue:

- ~~Blossom(5,(b,c))~~
- Vertex(6,c)
- Vertex(6,d)
- Vertex(8,d')

The second reason we compute the blossom is that we no longer care much about the individual vertices in $\mathbb{B}$, and can shrink them into just the base a'. We will still scan vertices like c from the queue as before, but whenever we are backtracking to compute blossoms we can skip the vertices in $\mathbb{B}$ entirely and go straight to the base a' instead. In this example, in a few iterations, the algorithm will find either (d', a') or (d, a') as a blossom edge, with just $\{d, d'\}$ as its blossom and a' as the base here as well. If we didn't contract the previous blossom, this new blossom would instead consist of $\{c', c, b, b', d, d'\}$, but we are already completely done with most of those vertices and computing all of it again would be a waste. Therefore we shrink them.



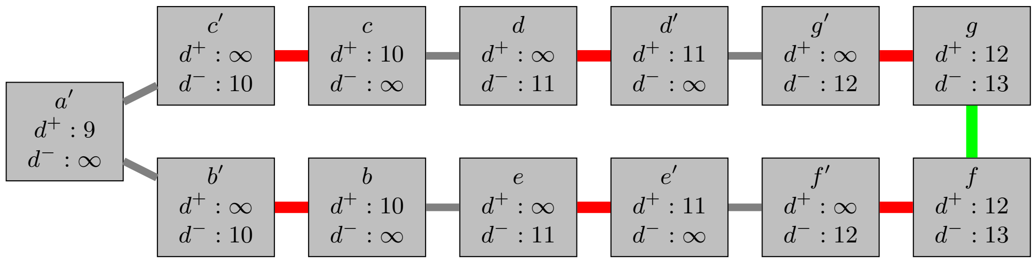
Queue:

- Vertex(6,c)
- Vertex(6,d)
- Vertex(8,d')

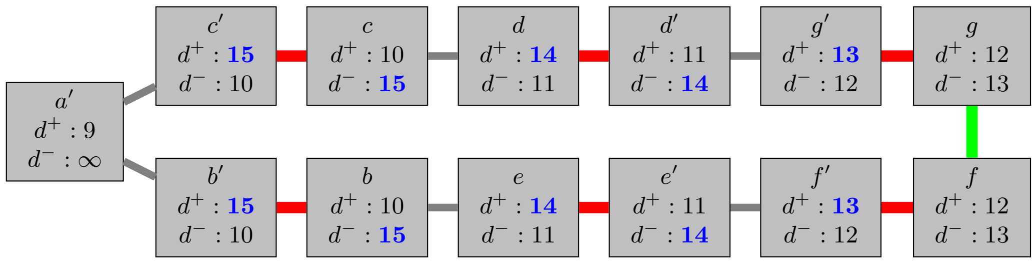
If the concerned reader is familiar with the more general Shortest Alternating Path algorithm [BD83] or the original blossom algorithm [Edm65], and worry that such pseudonodes often have to be expanded again, then remember that in our case the set F is an almost-perfect matching and those cases never happen.

Let us now skip a few steps, until t eventually reaches the front of the queue. At that point, we have that $d_t^- = 5$, and that is also the cost of the shortest odd path in our original input graph. To compute the exact path we can backtrack from t to s and then translate that path as described in Step 4 of our reduction in Section 4.1. We end up with the path $[(s, a), (a, b), (b, c), (c, d), (d, t)]$, which is the shortest odd s-t-path in the graph.

This example had only two small blossoms, none of which illustrated vertices receiving values for d^-. See Section 4.2 for another, larger example of a blossom. Observe how clear the structure is: we always have an odd cycle, where every other edge is in the matching, except for the two edges to the base. In this case every other vertex is missing values for either d^+ or d^-, as is common, though it may happen that a vertex in a blossom already has another tentative value for d^-.



(a) A larger blossom with a' as its base and the blossom edge marked in green.



(b) Every other vertex gets a chance to update their values for d^+ or d^-, here marked in blue.

Figure 4.2: A larger blossom computed.

4.3 Pseudocode

Here we show a detailed pseudocode of the algorithm, along with explanations of some of the finer details should the interested reader consider implementing the algorithm on their own. For the entirety of the code in one piece without comments or alternatives, see Appendix A. And of course, to see the code implemented in Rust, see the GitHub repository [Sim24b].

4.3.1 Initialization

First, we initialize the arrays we need, with appropriate default values for all vertices. The mirror graph is constructed as described in Definition 4.1.1.

Code Listing 4.1: Initialization

```
fn init(input_graph, s, t) {
    graph = create_mirror_graph(input_graph);

    for u in 0..n {
        d_plus[u] = ∞;
        d_minus[u] = ∞;
        pred[u] = null;
        completed[u] = false;
        basis[u] = u;
        in_current_blossom[u] = false;
    }
    d_plus[s] = 0;
    completed[s] = true;

    for edge in graph[s] {
        priority_queue.push(Vertex(weight(edge), to(edge)));
        d_minus[to(edge)] = weight(edge);
        pred[to(edge)] = e;
    }
}
```

The main function ties it all together. The `control` function includes the main loop and does most of the work until there is nothing more to do. Then we can either find the shortest odd path by backtracking or conclude that no odd paths exist.

Code Listing 4.2: Main

```
fn main(input_graph, s, t){
    init(input_graph, s, t);

    control();

    if d_minus[t] == ∞ {
        return None;
    }
    cost = d_minus[t];
    path = backtrack();

    return Some(cost, path);
}
```

Here is how to backtrack once we know we have the shortest odd path to t. Each of the edges from the mirror side must be replaced by their equivalents on the real side of the input graph. Note that unlike when backtracking blossoms, here we do *not* consider the base of the vertices. Here we pretend that we never shrunk the blossoms into pseudonodes so that we find the entire path.

Code Listing 4.3: Backtracking

```
fn backtrack() {
    current_edge = pred[t];
    path = [current_edge];
    while from(current_edge) != s {
        current_edge = pred[mirror(from(current_edge))];
        if current_edge is from the mirror side {
            path.push(mirror(current_edge));
        }
        else {
            path.push(current_edge);
        }
    }
    return path;
}
```

4.3.2 The control loop

This is the main loop of the algorithm. Each iteration of the outer loop will either scan a vertex, handle a blossom edge, or conclude that we are done. Each vertex can be put on the queue many times, but we only want to scan it once, so we discard those that have already been scanned. Likewise, each blossom consists of many edges, each of which can be put on the queue, but we only want to compute each blossom once. If the two endpoints of a potential blossom edge already have the same basis, then we know they have already been computed as part of the same blossom and the edge may safely be discarded.

An important detail that is not clear in the pseudocode is that in the event of a tie in priority, a vertex should always be prioritized before a blossom edge. Otherwise we may get incorrect results. One way to ensure that is to give a specific implementation of how the Vertex and Blossom sum type is ordered.

Code Listing 4.4: Control, the main loop

```
fn control() -> bool {
    loop {
        while ! priority_queue.is_empty() {
            match priority_queue.top() {
                Vertex(_, u) => {
                    if completed[u] {
                        priority_queue.pop();
                    }
                    else {
                        break;
                    }
                },
                Blossom(_, edge) => {
                    if base_of(from(edge)) == base_of(to(edge)) {
                        priority_queue.pop();
                    }
                    else {
                        break;
                    }
                }
            }
        }

        if priority_queue.is_empty() {
            // No odd s-t-paths exist :(
            return;
        }
        match priority_queue.pop() {
            Vertex(_, u) => {
                if u == t {
                    // We have found a shortest odd s-t-path :)
                    return;
                }
                d_plus[u] = d_minus[mirror(u)];
                scan(mirror(u));
            }
            Blossom(_, edge) => {
                blossom(e);
            }
        }
    }
}
```

Code Listing 4.5: Scan

```
fn scan(u) {
    completed[u] = true;
    dist_u = d_plus[u];
    for edge in graph[u] {
        v = to(edge);
        new_dist_v = dist_u + weight(edge);

        if ! completed[v] {
            if new_dist_v < d_minus[v] {
                d_minus[v] = new_dist_v;
                pred[v] = edge;
                priority_queue.push(Vertex(new_dist_v, v));
            }
        }
        else if d_plus[v] < ∞ and base_of(u) != base_of(v) {
            priority = d_plus[u] + d_plus[v] + weight(edge);
            priority_queue.push(Blossom(priority, edge));
            if new_dist_v < d_minus[v] {
                d_minus[v] = new_dist_v;
                pred[v] = e;
            }
        }
    }
}
```

4.3.3 Backtracking a blossom edge

When we compute a blossom edge e, we need to compute the vertices and edges that make up the blossom. We do this by creating two paths, one starting in from(e), and the other starting in to(e). We backtrack both of them towards the source while alternating between matched and non-matched edges until the paths meet up at a common ancestor b. Then we set b as the base, and the two paths make up our blossom.

The naïve way would be to backtrack both paths individually all the way to the source, and only then see where they start to overlap. That would run in time linear to all the vertices in the graph and is a waste of time. A slightly better idea is to backtrack one path all the way to the source, and then backtrack the other only until it reaches a vertex in the other's path. That is better, but even if we somehow know beforehand which path needs the fewest edges this too would run in linear time.

Instead, we alternatingly backtrack both paths at the same time, and mark each vertex when added to a path. Whenever one path reaches a vertex that is already marked by the other, we mark that vertex as the base and delete the vertices in the other path that came after it. Now we can compute the vertices in the blossom in time linear to the number of vertices in the blossom, rather than the entire graph.

Implementing this may sound difficult, tedious, and error-prone, but it is actually way worse. Here are some reasons why this is the most complex part of any algorithm in this thesis, and why any programmer should take particular care when implementing this:

- It is difficult to alternate through matched and non-matched edges, while simultaneously alternating between computing two separate paths.
- We have to consider the basis of each vertex found on the paths rather than the vertex itself because we shrink each blossom into a pseudonode after computing it.
- The paths may not have the same number of edges, even if the graph is unweighted.
- The endpoints we start backtracking from may already be the base if the blossom edge is adjacent to it.

- The blossom edge itself should end up in both, either, or neither of the paths, depending on where it is in relation to the base.
- The paths may not have any edges at all.
- One path may reach the source vertex before the other path has reached b. Then we have to stop backtracking that path and focus on the other.

Here is our solution. The procedure returns the base and two lists of all the non-matched edges that make up the blossom. That usually includes the blossom edge itself, which is part of both paths unless it is adjacent to the base.

Code Listing 4.6: Backtrack blossom

```
fn backtrack_blossom(edge) {
    p1 = [ reverse(edge) ];
    p2 = [ edge ];
    u = get_basis(to(edge));
    v = get_basis(from(edge));
    in_current_blossom[u] = true;
    in_current_blossom[v] = true;

    loop {
        if u != s {
            u = get_basis(mirror(u));
            in_current_blossom[u] = true;
            e = pred[u];
            u = get_basis(from(e));
            p1.push(e);

            // If true, then u is the base
            if in_current_blossom[u] {
                p1.pop();
                in_current_blossom[u] = false;

                // We remove all the edges in p2 after the base
                while p2 is not empty {
                    e = p2.last();
                    v = get_basis(from(e));
                    in_current_blossom[v] = false;
                    p2.pop();
                    if v == u {
                        break;
                    }
                }
                return (u, p1, p2);
            }
        }
        if v != s {
            // *Here we do the same for the other path*
        }
    }
}
```

The last if-statement closely resembles the first, except with other variables, so we have omitted it here for brevity. See Appendix A for the full version.

4.3.4 Computing blossoms

To compute a blossom, we first have to determine its base and its edges, as discussed in the previous section. Then we can use the edges in the paths to potentially improve values for d^+

and d^-, and to set the new base for all the vertices involved. Afterward, we scan all the vertices that we now gave d^+ values.

Two lists of edges can be processed simultaneously without issues, but we treat them separately here to avoid spending time on concatenating them. Setting blossom values and setting the basis can also be done at the same time, but we split it into two separate functions to improve readability. However, the scans may only be performed after all the vertices in both lists have received their new basis.

Code Listing 4.7: Blossom

```
fn blossom(edge) {
    (b, p1, p2) = backtrack_blossom(edge);

    to_scan1 = set_blossom_values(p1);
    to_scan2 = set_blossom_values(p2);

    set_edge_bases(b, p1);
    set_edge_bases(b, p2);

    for u in to_scan1 {
        scan(u);
    }
    for v in to_scan2 {
        scan(v);
    }
}
```

Code Listing 4.8: Set blossom values

```
fn set_blossom_values(path) {
    to_scan = [];

    for edge in path {
        u = from(edge);
        v = to(edge);
        w = weight(edge);
        in_current_cycle[u] = false;
        in_current_cycle[v] = false;

        // We can set a d_minus
        if d_plus[v] + w < d_minus[u] {
            d_minus[u] = d_plus[v] + w;
            pred[u] = reverse(edge);
        }

        int m = mirror(u);
        // We can set a d_plus, and scan it
        if d_minus[u] < d_plus[m] {
            d_plus[m] = d_minus[u];
            to_scan.push(m);
        }
    }

    return to_scan;
}
```

Code Listing 4.9: Set edge bases

```
    fn set_edge_bases(base, path) {
        for edge in path {
            u = from(edge);
            m = mirror(u);
            set_base(base, u);
            set_base(base, m);
        }
    }
```

4.3.5 Setting the base of blossoms and pseudonodes

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 disregarded. Whenever we set u to have the base b, we also have to see if any other vertices have u as their base and set their bases to b as well. Derigs never specified any data structure to update these bases efficiently.

The naïve solution would be to do something like this:

Code Listing 4.10: Näive basis

```
fn set_base(base, u) {
    basis[u] = base;
    for v in 0..n {
        if basis[v] == u {
            basis[v] = base;
        }
    }
}
fn get_base(u) {
    return basis[u];
}
```

This would search through all vertices in the graph in linear time. We have found two potential improvements to this. The first version is to use an Observer pattern, where each vertex u keeps a record of the vertices that have u as its base. Initially $\mathit{dependents}[u] = [\ ]$ for all of them. Then, when we update u's base to *base*:

Code Listing 4.11: Observer basis

```
fn set_base(base, u) {
    basis[u] = base;
    dependents[base].push(u);
    for v in dependents[u] {
        basis[v] = base;
        dependents[base].push(v);
    }
}
```

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 UnionFind, 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 UnionFind.

Code Listing 4.12: UF-like basis

```
fn set_base(base, u) {
    basis[u] = base;
}
fn get_base(u) {
    if u != basis[u] {
        basis[u] = get_base(basis[u]);
    }
    return basis[u];
}
```

Now we can update a base in constant time, with the tradeoff of potentially slower queries. Is this faster? Well, sometimes it is. Contrary to Observer-based version, now we can set the new base of a vertex very quickly, but we spend more time looking up the endpoints of potential blossom edges. It may therefore be preferable in sparse graphs, where the number of edges is small compared to the number of vertices. We benchmark both versions and discuss the results in Section 4.5.2.

4.4 Improvements on Derigs' algorithm

The main idea of our algorithm is the same as the original by Derigs [Der85]. We have, however, made some minor adjustments, and we will discuss these here.

First, the original algorithm used the idea of building up a tree T of alternating edges to mark scanned vertices as done. This is to avoid scanning the same vertex multiple times and to make sure that a blossom edge is only put into the queue after both its vertices have been scanned. The notation $V(T) := V(T) \cup \{k, l\}$ was used to mark k as done. We had multiple problems with this. To begin, only the matched edges are ever added to the tree, so the disconnected 'tree' would not be a tree at all. Furthermore, unlike how for example Dijkstra's Algorithm builds up an implicit tree of scanned vertices, the vertices in our mirror graph are not at all scanned in the order of their distance to the source, so even if we were to add actual edges to the tree it would still not be a tree. Finally, we found that the notation was misleading and overly complex for what really should be a simple concept. We have replaced this with just a boolean array called `completed`, where each vertex u initially has `completed[u] = false` until it has been scanned, at which point we set `completed[u] = true`.

Second, we have chosen to utilize sum types to have one priority queue with both vertices and blossom edges in one. The original algorithm used two priority queues which always had to be queried together, and this was difficult to read, write, and debug. We find that combining them into one queue simplifies the code greatly. The way we set their priorities is also different: vertices now have a priority of *twice* its d^-, so that blossom edges can have a priority of the sum of the d^+'s of its two endpoints and its edge weight *without* dividing by two afterwards.

We find this to be simpler, and we no longer have to convert integer weights to floating points just to prioritize them correctly.

Third, we developed two data structures to store and update the basis of each vertex. Neither improve the theoretical running time, but can be faster in practice depending on the types of inputs. See Section 4.3.5 for a discussion of their differences, and see Section 4.5.2 for an empirical analysis of the improvement.

Fourth, and this is more subjective, we will argue that we have improved the presentation of the algorithm. Derigs' algorithm is both very neat and very useful, but we believe that the way it is presented in the original paper [Der85] is too terse in many places, and overly complex in others. We speak from experience when we say that implementing the algorithm was a challenge, and one of our goals is to provide an easier starting point for programmers who wish to implement the algorithm themselves, to make this neat algorithm more accessible.

4.5 Analysis

Theorem 4.5.1. Let (G, s, t) be an instance of Shortest Odd Path, let $n := |V|$ and let $m := |E|$.
Claim: the algorithm runs in time at most $O(m \cdot \log m)$, or $O(m \cdot \log n)$ of the graph is simple.

Proof. First of all, we construct the mirror graph M with $2n - 2 \in O(n)$ vertices and $2m - \deg(s) - \deg(t) \in O(m)$ edges, in time $O(n + m)$.

With our `completed` array we can guarantee that each vertex is scanned at most once, and the scanning operation just loops through all the neighbors. Therefore, the total cost of all the scans is $O(n + \sum_{u \in V} \deg(u)) = O(n + 2m) = O(n + m)$.

The blossom operation is a little more convoluted. Thanks to the overly complicated code in our `backtrack_blossom` procedure in Section 4.3.3, we can backtrack from a blossom edge and determine the vertices in the blossom in time linear to the size of the blossom. Setting their values for d^+ and d^- can also be done in linear time, and the potential scans have already been accounted for above. The key point here is that we shrink the blossom into a pseudonode afterwards: each vertex can then only be part of such a blossom procedure at most once. Even though we may compute many blossom edges, the total amount of work will never exceed $O(n)$.

Finally, we have the main loop, which iteratively pops vertices and blossom edges from the queue. Each of the $O(n)$ vertices may be put into the priority queue many times, at most once for each of its neighbors. That is a total of $O(m)$ vertices in the queue, for a total cost of $O(m)$. Though it is unlikely, in the worst case all edges may be enqueued as blossom edges as well, again for a total cost of $O(m)$. In total, enqueueing everything costs $O(m)$, and dequeueing everything costs $O(m \cdot \log m)$.

In total, the algorithm runs in time $O(n + m) + O(n + m) + O(n) + O(m) + O(m \cdot \log m) = O(m \cdot \log m)$, which shows the first part of the claim. If the graph is simple, then $O(m \cdot \log m) \subseteq O(m \cdot \log n^2) = O(m \cdot 2 \cdot \log n) = O(m \cdot \log n)$, which shows the second part of the claim. $\square$

4.5.1 Other variants

A running time of $O(m \cdot \log n)$ means that the algorithm generally performs well on sparse graphs. We chose this algorithm with this running time because we will be using it on planar graphs in Chapter 5, where $m \leq 3n - 6$, as shown in Corollary 2.3.2. That means it runs in $O(n \log n)$ on planar graphs.

We should note, however, that there are also other known polynomial algorithms for Shortest Odd Path. In the same paper that Derigs presented the algorithm of this chapter, he also presented another variant [Der85]. The main difference is that we drop the priority queues and use a list instead, and in the control loop instead search through the entire list and pop the element with the lowest priority. That search takes time at most $O(n)$, and can be done at most n times, for a total running time of $O(n^2)$. If the input graphs are dense, then this that may be preferable to our $O(m \log n)$ algorithm.

In the case where the graphs are unweighted, there is an even better algorithm: [LP84] present an algorithm that runs in the lightning fast $O(n + m)$ in undirected unweighted graphs.

All this talk about odd paths might cause the curious reader to ask, what about even paths? The problem of Shortest Even Path is, in fact, equivalent: to find the shortest *even* s-t-path in a graph, simply add a new vertex t' along with an edge from t to t', and then find the shortest odd s-t'-path. The same reduction can of course also be used the other way around, should we prefer to solve Shortest Even Path instead. The reason we choose to focus on odd paths is simple, each time we write the name of the problem we save one entire character.

4.5.2 Benchmarking different data structures for the Basis

As explained in Section 4.3.5, we have developed multiple data structures to keep track of the basis of each vertex. One of them is based on the Observer pattern, the other on the well-known UnionFind structure. We have tested both on seven sparse graphs from the real world, as well as on four synthetic Delaunay graphs. For each graph and for each structure, we have run the algorithm 20 times and noted the average times in the table below. See Section 6.4 for more specific information about how these benchmarks are done.

Graph	n	m	Observer	UF	Change
Power BCSPWR09 [RA15]	1723	2394	1.7ms	1.5ms	-12%
Oldenburg [LCH+05]	6106	7035	5.6ms	4.8ms	-14%
San Joaquin County [LCH+05]	18263	23874	22ms	18ms	-18%
Cali Road Network [LCH+05]	21048	21693	21ms	18ms	-14%
Musae Github [RAS19]	37700	289003	125ms	123ms	-1.5%
SF Road Network [LCH+05]	174956	223001	225ms	191ms	-15%
Ca Citeseer [RA15]	227321	814137	629ms	608ms	-3%
Delaunay 50k	50000	149961	106ms	96ms	-9%
Delaunay 100k	100000	299959	219ms	205ms	-6%
Delaunay 150k	150000	449965	346ms	315ms	-9%
Delaunay 200k	200000	599961	476ms	449ms	-6%

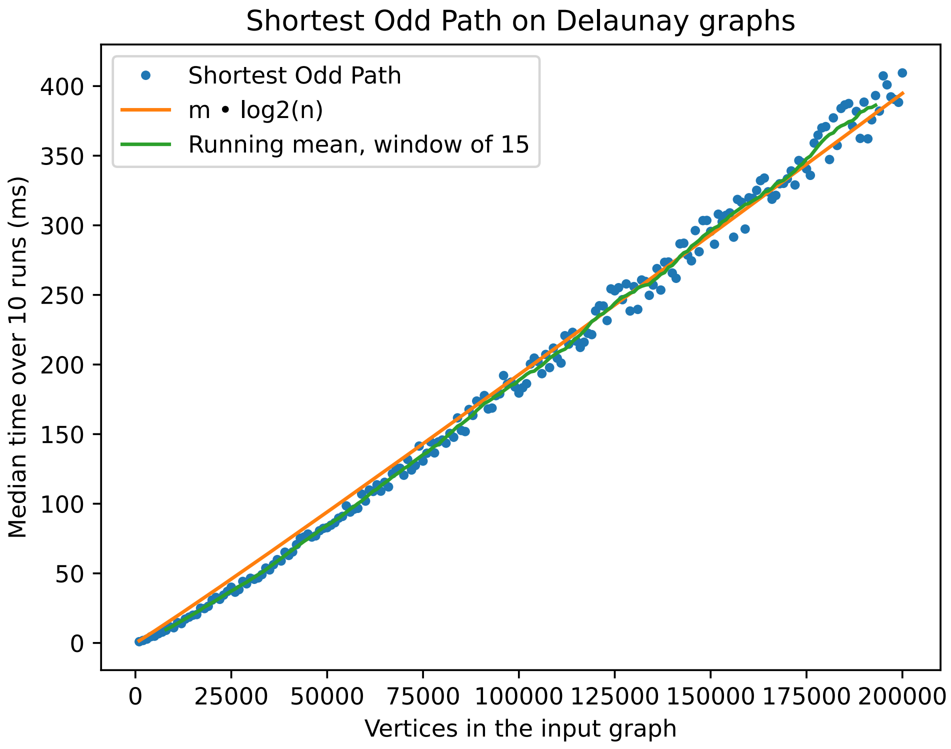
As we can see, the UnionFind-based structure either outperforms or does as well as the Observer-based structure on all the graphs we tested. Most notably, we have an 18% decrease in time spent on the graph of San Joaquin County.

There seems to be a correlation between how well the UnionFind-based structure performs and how sparse the graphs are. This makes sense: the UnionFind-based structure updates the base of vertices very fast and spends more time to lookup bases for each potential blossom edge, while the Observer-based structure does the exact opposite. The focus of this thesis is to create an algorithm that performs well on sparse graphs, especially planar graphs, which is why we mostly test on sparse graphs. If we instead had implemented an algorithm for denser graphs, or just graphs in general, then testing on graphs with more edges would be more appropriate. Therefore, we will use the UnionFind-based structure as the default in our codebase, as well as in the remainder of this thesis, and note to the reader that the other option might be preferable in denser graphs.

4.5.3 Running times on Delaunay graphs

To test how well the algorithm scales, we generate 200 Delaunay graphs of sizes 1000, 2000, 3000, and so on until 200k. We explain how these graphs are generated in Section 6.4. For each graph, we estimate a source and target with the maximum shortest path between them, and run our Shortest Odd Path algorithm. We take the median running time of 10 runs and plot the results below.

The theoretical running time is $O(m \log n)$, and we try our best to come up with constants to create a running time function that matches our algorithm. In Delaunay graphs, we have that $m \leq 3n$, so we have set $m := 3n$ in this function.



As we can see, the running times grow almost linearly as the inputs grow larger. The slight upward curve is barely noticeable, and this is to be expected with a linearithmic theoretical running time. We are very happy with these results. Being able to solve Shortest Odd Path on sparse graphs of 100k vertices in a fifth of a second is exactly the kind of performance we were hoping for.

Now we take this algorithm with us and confidently tackle the next challenge: Network Diversion.

Chapter 5

Network Diversion

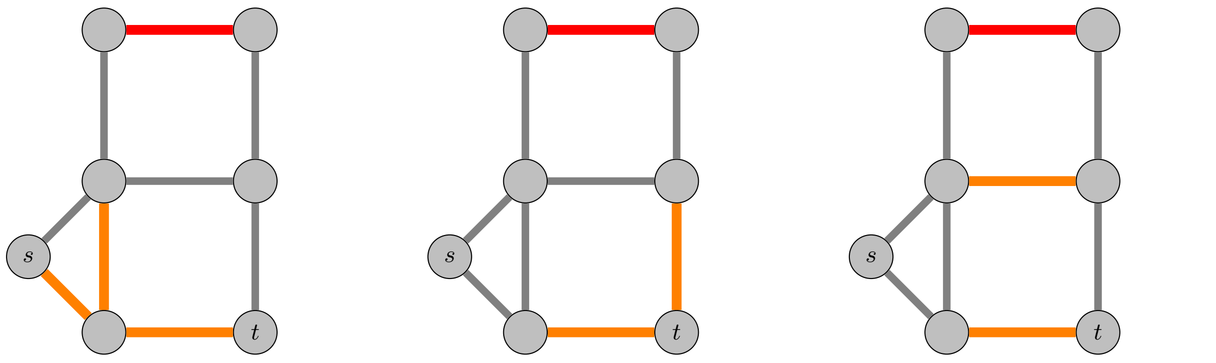
5.1 Introduction to Network Diversion

Now that we have an algorithm for Shortest Odd Path, we will use it to solve a much more useful problem:

Network Diversion
Input: a weighted graph $G := (V, E, \text{from}, \text{to}, \text{weight})$, two vertices $s, t \in V$, and a *diversion edge* $d \in E$
Output: a *diversion set* $D \subseteq E$ of minimum weight such that all s-t-paths in $(V,\ E \setminus D,\ \text{from},\ \text{to},\ \text{weight})$ must go through d

A diversion set may also equivalently be defined as a minimal s-t-cut that includes d. If all edges from the diversion set are deleted except d, then d is the bridge between what would otherwise be two separate components and all s-t-paths must go through d. Network Diversion can then be restated as the quest to find a *minimum* minimal s-t-cut that includes d. Both definitions are equivalent and yield the same optimum results, and being able to switch between formulations of the problem makes it easier to solve them.

See Figure 5.1 for examples. Figure 5.1a and Figure 5.1b show incorrect attempts at diversions, while Figure 5.1c shows a valid diversion.



(a) Invalid, there are still paths that don't go through the diversion edge after deletion

(b) Invalid, deleting the diversion set also blocks all paths that use the diversion edge

(c) Valid, deleting the diversion set funnels all traffic through the diversion

Figure 5.1: Valid and invalid diversions, attempting to force all s-t paths to go through the diversion edge in red by deleting the diversion set in orange.

Unlike with Shortest Odd Path, it is much easier to come up with practical applications of Network Diversion. Consider a communication network of machines that communicate offline, where you, a spy, can intercept all messages between two specific machines. How can you through outages and other diversions force all traffic in the network to go through where you can intercept the messages? Or for a more direct example: consider a network of roads and bridges, the knowledge that the enemy wants to move troops and supplies from one point to another, and a specific bridge where you are especially prepared to ambush them. How can you with the least amount of artillery destroy bridges to force the enemy to move through your ambush?

Initially, finding a minimum minimal s-t-cut that includes a specific edge may seem like yet another variation of the well-known minimum s-t-cut problem, of which we have numerous excellent polynomial-time algorithms. Yet, this is considerably harder to solve correctly. If we just use a normal maximum flow algorithm like Edmonds-Karp [EK72], then we are very likely to end up with a minimum cut that does *not* include the diversion edge, like in Figure 5.1b. If we force the flow algorithm to use the diversion edge, we are likely to end up with a cut that is not minimal, a cut where we might as well drop the diversion edge from the set and still have a cut, like in Figure 5.1b.

In fact, [CWN13] have shown that Network Diversion is NP-complete on directed graphs, even without cycles or weights. Whether Network Diversion can be solved in polynomial time on undirected graphs is still an open problem. [CWN13] have found polynomial-time algorithms for the special case where the input graph is s-t-planar, meaning that the graph can be embedded such that s and t are adjacent to the outside face, but whether there is a polynomial-time algorithm for planar graphs in the general case is still an open problem.

Until now. We will present the first-ever polynomial-time algorithm that solves Network Diversion in undirected planar graphs. It will also work with weighted edges, as long as the weights are non-negative. Many graphs based on physical structures are planar. In the example of roads and bridges, having two roads cross without a crossroad is usually inefficient and more costly, so such networks are very often planar. The costs associated with cutting an edge or blowing up a bridge are usually non-negative, too. So even if we do not solve NETWORK DIVERSION in the most general case, solving it for planar graphs of non-negative edges is not far from it in practice.

5.2 Intuition

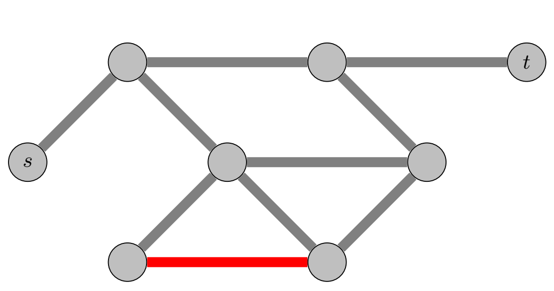
5.2.1 Detour paths

Before we reveal the algorithm for Network Diversion, we will first look at a curious little problem that we call SHORTEST DETOUR PATH. Instead of deleting edges to force all paths to go through a certain edge, we want to purposefully pass through that edge and look for the shortest path that does. Perhaps we are going on a road trip, and we want to stop at a specific gas station along the road to say hi to our friend Mike who works there. And because this is a road trip, we do not want to drive along the same roads multiple times, that would be boring.

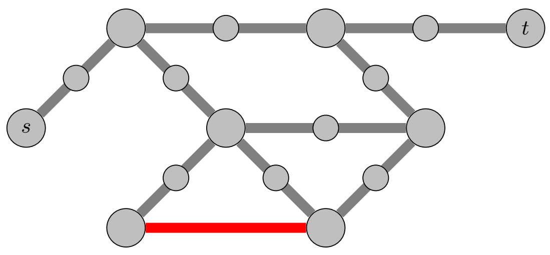
SHORTEST DETOUR PATH
Input: a graph G, two vertices $s, t \in V$, and a 'detour' edge $d \in E$
Output: an s-t-path in G of minimum cost, that goes through the detour d

There is no obvious way to solve SHORTEST DETOUR PATH. One might attempt to concatenate the shortest s-from(d)-path and the shortest to(d)-t-path, but those two paths might overlap and reuse the same vertices, and therefore would their concatenation not necessarily be a path but instead a mere walk. The classical solution is to use a maximum flow algorithm like Edmonds-Karp [EK72] to find those two paths, to enforce that they are vertex-disjoint. It works, but is slower and more complicated compared to what we are about to do.

Instead, we create a new graph H, by subdividing all edges in G *except* d, as seen in Figure 5.2. The key point to see here is that any odd s-t-path in H must necessarily go through the detour, otherwise it would not be odd. We can visualize it by 'stepping through' the edges in H. If we start on our right leg, then in the beginning every time we reach a vertex that is also in G, we reach it by stepping on our left leg. That continues until we use the detour edge, and from then on we step on all vertices from G using our right leg. If we require that we end at t on our right leg, then the path must be odd, and any odd path must go through the detour. Therefore we can simply run our SHORTEST ODD PATH algorithm on H, and if such a path exists we can reverse the subdivision of the edges in the path and the result is the SHORTEST DETOUR PATH in G.



(a) An instance of Shortest Deour Path, the detour marked in red.



(b) All edges except the detour have been subdivided, to create an instance of Shortest Odd Path.

Figure 5.2: Shortest Detour Path reduced to Shortest Odd Path by subdividing all edges except the detour.

If we extend the problem to have multiple detour edges, where we have to go through all of them in any order, then our idea will not work[1]. The problem is that we have no way of knowing whether we have used the marked edges 1, 3, or 5, etc. times, because in all of them we hit vertices from G using our right leg. We can, however, use this idea to find paths that use a certain set of edges an odd amount of times. As it turns out, that is exactly what we need to solve Network Diversion.

5.2.2 From a dual path to a real diversion

Remember, we want to find a minimum minimal s-t-cut in G that includes the diversion edge d.

Instead of looking for a minimal cut in G, let us look for a simple cycle in the dual graph $G^\star$, as is explained to be equivalent in Fact 2.3.1. We can do this by finding a path in $G^\star$ that goes from and to the left and right faces of d, without using $d^\star$ itself, and then adding $d^\star$ at the end to complete the cycle. If the path found is also the shortest such path, then it corresponds to the *minimum* minimal cut in G that uses d, though it is not necessarily an s-t-cut.

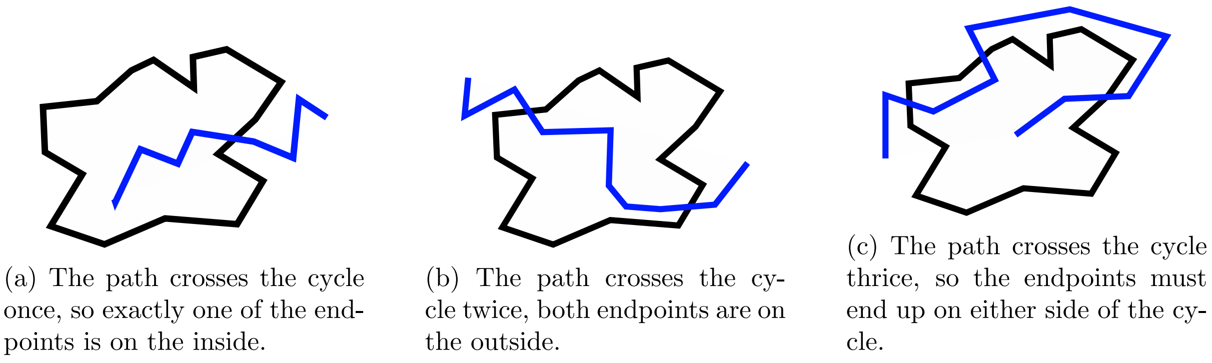
To force s and t to end up in different components after the cut, we need some additional details. First, we find any s-t-path in G, not necessarily the shortest path. Then we subdivide all the edges in the dual graph *except* those who cross edges on the found s-t-path. Now we can look for the shortest *odd* path that goes from and to the left and right faces of d in the subdivided dual graph, and add $d^\star$ at the end to make it a cycle. Like before, this corresponds to a minimum minimal cut in G that uses d, but now it must also cross the edges in the found s-t-path an odd number of times, as explained in Section 5.2.1.

This cycle, and the found s-t-path, can be interpreted as curves in our embedding. The cycle can additionally be interpreted as a Jordan curve, and by the Jordan Curve Theorem, the cycle

[1]This is a good thing, because otherwise we would have solved the Traveling Salesman Problem in polynomial time and complexity theory as we know it would break down.

divides the plane into an 'inside' and an 'outside'. Since the curve of the s-t-path crosses the curve of the cycle an odd number of times, exactly one of its endpoints must be on the inside, as illustrated in Figure 5.3. The endpoints are s and t, meaning that s and t end up in different components after the cut. It follows that this cut is an s-t-cut in G, specifically a minimum minimal s-t-cut in G that uses d.

This is the main idea for our algorithm. It should be noted that we did not come up with this idea ourselves, but have to thank Pål Grønås Drange [Dra24] for his as for now unpublished work on the subject.


(a) The path crosses the cycle once, so exactly one of the endpoints is on the inside.

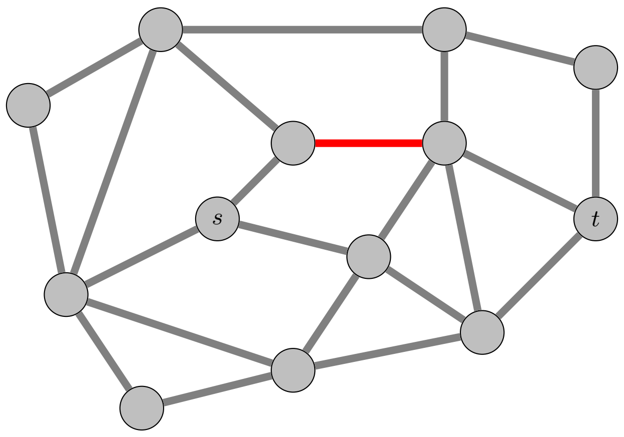
(b) The path crosses the cycle twice, both endpoints are on the outside.

(c) The path crosses the cycle thrice, so the endpoints must end up on either side of the cycle.

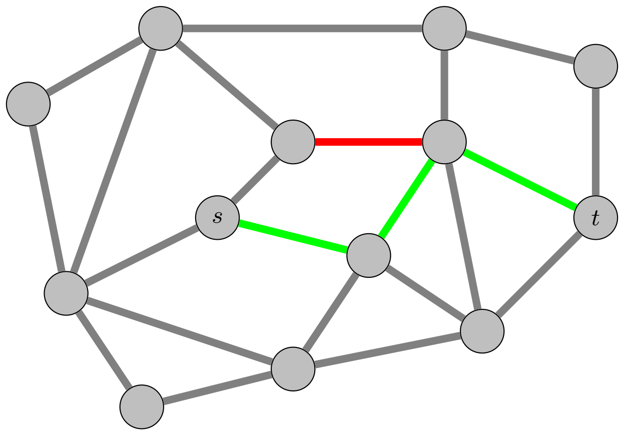
Figure 5.3: The two endpoints of a path end up on different sides of a cycle if and only if it crosses the cycle an odd number of times.

5.2.3 The algorithm

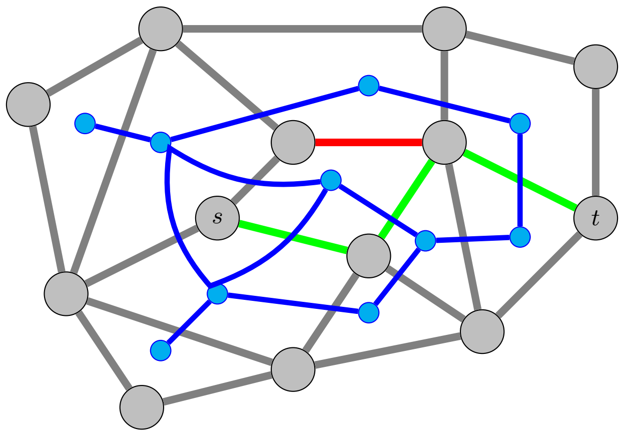
We will explain the algorithm by following an example. We want to find the minimum minimal s-t-cut that includes the diversion edge marked in red.



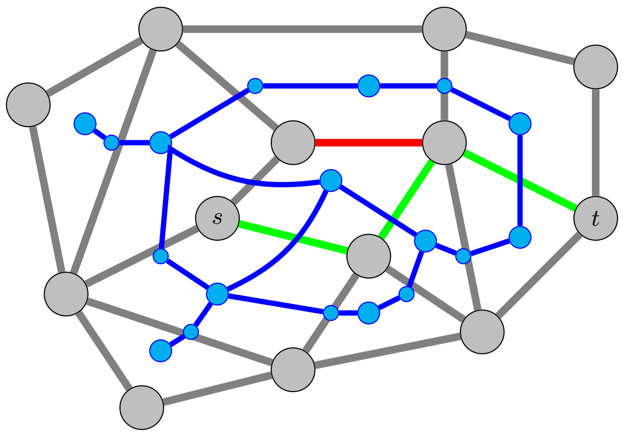
First, we find any s-t-path that does not use the diversion edge. It does not necessarily have to be the shortest path. We have marked such a path in green below.



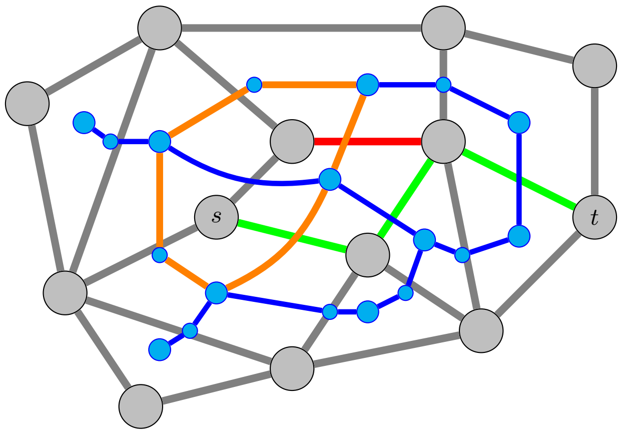
Next up is to compute the dual graph. We delete the dual edge that crosses the diversion edge and color the rest in blue. Note that we have omitted the outside face and its edges in this visualization, otherwise we would have a much too cluttered illustration.



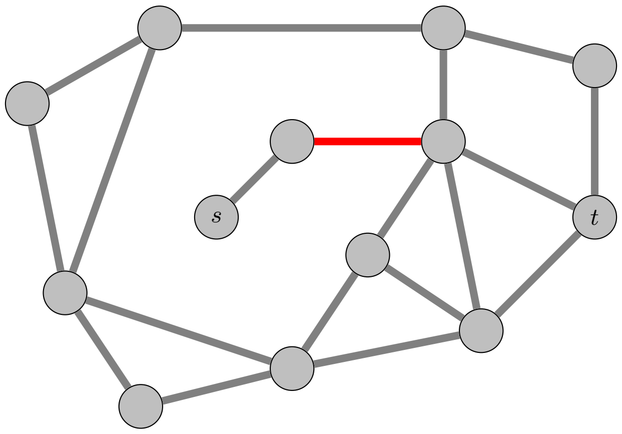
Now we subdivide all the edges in the dual graph except those that cross the path in green.



The last step is to find the shortest odd path in the subdivided dual graph from and to the regions to the left and right of the diversion edge, using our newfound favorite algorithm. If we find such a path, we know that it must cross the s-t-path in green an odd number of times. If we add the dual equivalent of the diversion edge to the path to create a cycle, then we know that this cycle goes around either s or t, but not both. We illustrate the cycle in orange below.



With this, we finally have our diversion set. Simply delete the edges in the original graph that cross the cycle in orange, except for the diversion edge of course. We end up with a graph where all s-t-paths must pass through the diversion edge. The problem is solved.



5.3 Pseudocode

Here comes the pseudocode for our Network Diversion algorithm.

Code Listing 5.1: Main

```
fn network_diversion(graph, s, t, d) {
    graph.delete_edge(d);
    path = shortest_path(graph, s, t);
    graph.add_edge(d);

    match path {
        None => {
            // No s-t-paths exist without d anyway,
            //  so no diversion is needed.
            return Some(0, []);
        }
        Some(p) {
            p* = [ e* for e in p ];
            dual = subdivide_edges_except(graph*, p*);

            match shortest_odd_path(dual, left(d), right(d)) {
                None => {
                    // There are no odd left(d)-right(d)-paths,
                    //  and therefore no way to divert the network.
                    return None;
                }

                Some(cost, odd_path) {
                    diversion = [e for e* in un_subdivide_edges(odd_path)];
                    return Some(cost, diversion);
                }
            }
        }
    }
}
```

5.4 Analysis

Theorem 5.4.1. Let (G, s, t, d) be an instance of Network Diversion, and let $n := |V|$. Claim: our algorithm runs in time $O(n \log n)$.

Proof. We find first a shortest s-t-path in G that does not use d, in time $O(n + m)$.

Then we subdivide all the edges in $G^\star$ except those found in the path, in time $O(n + m)$. This new graph has size $n' \leq 2n \in O(n)$ and $m' \leq 2m \in O(m)$.

Next up is to find an odd path in the subdivided graph, in time $O(m' \log n') = O(m \log n)$.

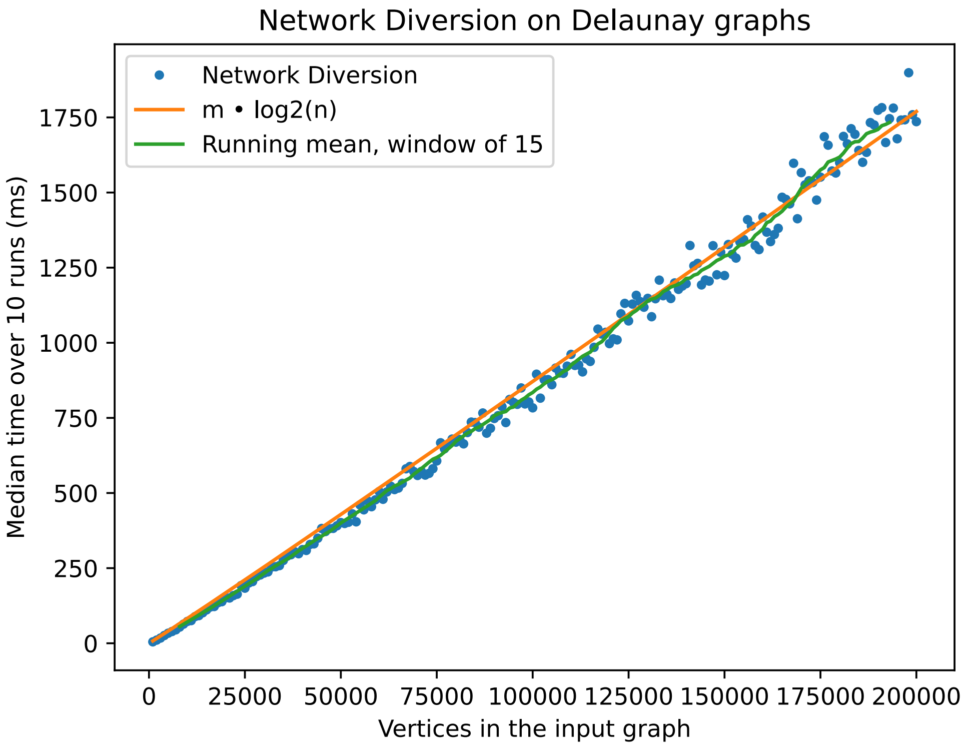
Lastly, if we are interested in the specific set of edges in the diversion and not just the cost, we un-subdivide the odd path in time $O(n') = O(n)$.

In total, we have a running time of $O(n + m) + O(m \log n) + O(n) = O(m \log n)$. Since G is planar we have that $m \in O(n)$, so we can simplify the complexity to just $O(n \log n)$ and complete the proof. □

Note that here we have assumed that the dual graph $G^\star$ has already been computed prior to starting the algorithm. If we have a straight-line embedding of G we can compute $G^\star$ in $O(n+m)$, which would not change the overall running time. However, if we do not have such an embedding the total running time might be considerably more.

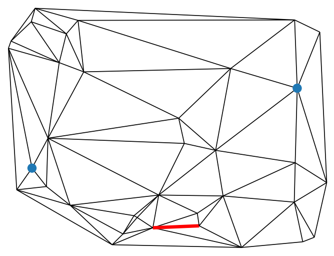
We compare the theoretical and practical running times on Delaunay graphs as we did in Section 4.5.3. For each graph, we have estimated a source and target vertex of maximum distance between each other, and then picked three edges in the graph as diversion edges. We select whichever diversion edge leads to the worst median running time over 10 runs, and plot the results below.

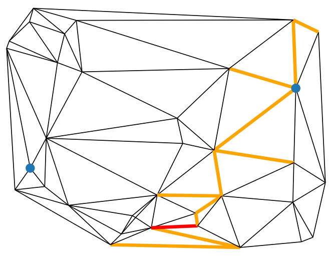
Here too we attempt to create a function out of the theoretical running time of $O(n \log n)$, this time with different constants. We have set $m := 3n$ in the plot since the graphs are planar.



As we can see, the running times grow just barely more than linearly compared to the input size. This is not surprising considering the linearithmic theoretical running time. The algorithm easily solves Network Diversion on planar graphs of 100k vertices in less than a second. Now compare that to the existing algorithms that need more than a second to solve for more than 30 vertices, and it is clear why a polynomial running time matters so much.

See Figure 5.4 for yet another example of what a diversion set may look like, this time on a Delaunay graph of 35 vertices.


(a) s and t are marked in blue, the diversion edge in red


(b) One possible diversion set marked in orange

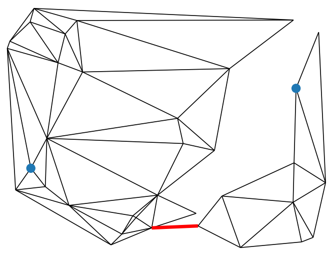

(c) With the diversion set removed, all s-t-paths must go through the diversion edge

Figure 5.4: Example of a solution for NETWORK DIVERSION on a Delaunay graph of 35 vertices

Chapter 6

The Codebase

The crux of this thesis is not about the theory we have presented in the other chapters, but rather about how well these algorithms work in practice, if at all. We have implemented most of the algorithms mentioned in the paper, and in this chapter, we will present the codebase. Of course, we invite the interested reader to explore the repository on their own [Sim24b].

6.1 Functionality

Our library is written in Rust. We chose that language because of its performance, but also because its type system and strict compiler help massively in reducing the developer overhead during development. Being able to fearlessly perform large refactors, and to use sum types rather than just product types to describe the data has had a huge impact on the quality of our code.

With that, we have successfully implemented the following algorithms:

- Shortest Odd Walk on graphs of non-negative weights, as described in Chapter 3.
- Shortest Odd Path on undirected graphs of non-negative weights, as described in Chapter 4.
- Network Diversion on undirected planar graphs of non-negative weights, as described in Chapter 5.
- Shortest Path on unweighted graphs, using a classic breadth-first search. It also runs on weighted graphs, by happily ignoring the weights.
- Shortest Path on graphs of non-negative weights, using Dijkstra's Algorithm as shown in Section 2.2.

- Shortest Detour Path on undirected graphs of non-negative weights, as described in Section 5.2.1.

All of them are generic with respect to edge weights and can handle any type of number-like weights such as `i32`, `u64`, and `f64`. When we subdivide edges in Network Diversion and Shortest Detour Path we use a neat little trick to be able to split a weight into two weights without accidentally rounding down any integers: we set one weight to zero and the other to the original weight. Any path that uses one of those edges will have to use the other afterwards, and their sum will then be the same as the original weight before the split.

6.2 Data structures

To perform these algorithms, we have implemented a few different data structures. The first is a basic structure for undirected graphs, using a vector of vectors of edges to represent the graph with adjacency lists. It is generic in the type of its edge weights, but also in the type of its edges. The 'basic' edge has basic edge methods like `.to()` and `.weight()`, but we also have a struct for planar edges, where we have the methods `.left()` and `.right()`. The methods work as described in Section 2.1 and Section 2.3.

We also have a data structure for planar graphs. They consist of two undirected graphs of planar edges: the 'real' graph and its dual. Each of them has edges that know which faces are to their left and right in the other graph. Even though our algorithm for Network Diversion will work with any planar graphs, parsing them and finding an embedding is rough. We therefore assume that we are given coordinates of each of the vertices and that they form a straight-line embedding. From there we can compute the dual. Whether the given coordinates form true planar embeddings is usually not checked, since the code to verify that runs in the painfully slow $O(m^2)$ and has been disabled by default.

Another limitation is that we can only handle *simple* planar graphs: if we have parallel edges, then the way we compute the dual will not work. If the input graph is not simple, we have to somehow combine the parallel edges until it is. Depending on the use case we have many reasonable strategies for combining them. If the edges represent bridges that are to be blown up with artillery, then the combined edge should probably be the sum of the weights of its components, or the sum of the artillery rounds needed to destroy the edges between those two vertices. If we in another case just need to cut any of the edges between them, then we may want to just take the cheapest one, or perhaps we are forced to take the most expensive one. Rather than making too many assumptions about the use cases, we provide five strategies for combining parallel edges, to cover as many use cases as possible:

- Take the first edge

- Take the last edge
- Keep the highest weight
- Keep the lowest weight
- Sum all weights

We hope that this is enough. If not, then our framework can easily be extended with more strategies.

As discussed in Section 4.3.5 and benchmarked in Section 4.5.2, we provide two data structures to keep track of the basis. They are called ObserverBase and UnionFindBase. Both are implemented using a common trait and can be switched out interchangeably in the SHORTEST ODD PATH algorithm. UnionFindBase usually performs the best on sparse graphs and has been set as the default, but ObserverBase may be preferable in denser graphs. The 'näive' basis we mentioned was too inefficient for anything but a temporary prototype and has long since been deleted.

6.3 Testing

The repository includes a large test suite, to ensure the correctness of everything we have implemented. First of all, each data structure comes with its own set of unit tests. Secondly, we have a total of 17 hand-crafted graphs of various shapes and sizes, and for each of them we have many queries for the different graph problems. We have found the optimal solutions manually and confirmed that the expected answers match the answers provided by the algorithms. Lastly, we have numerous problem-specific assertions in place, like asserting that the output of SHORTEST ODD PATH really is a path, or that the output of NETWORK DIVERSION indeed cuts the graph in two except for the diversion edge.

These tests have been immensely helpful in the development of our algorithms. Whenever we modified anything, we could instantly verify the validity with just the press of a hotkey and its subsequent run of the test suites. Though we do not present many formal proofs for the algorithms in this thesis, we like to think of the tests as informal proofs by empirical analysis.

6.4 Benchmarking

To benchmark our algorithms for SHORTEST ODD WALK and SHORTEST ODD PATH, we have picked seven different graphs from real-life scenarios of various sizes. We focus on sparse graphs, where the number of edges is not too large compared to the vertices. All benchmarks are run on a laptop with 16GB of memory and an i5-1155G7 of 2.5 to 4.5GHz, which we will denote as an *average laptop*. During the runs, we make sure that the power cable is plugged in and that other processes are shut down, for maximum performance.

To compare the theoretical and practical runtimes of our algorithms, we need a collection of graphs of easily scalable sizes. Furthermore, as to also be used for benching Network Diversion, the graphs have to be planar graphs with a built-in planar embedding.

Our solution is this: for a given integer n, generate n random points in the plane, to be the vertices in our graph. Then, using the `scipy` library in Python, compute a Delaunay triangulation of the points. Each of the triangles in the triangulation consists of three points, inbetween of which we add three edges with random weights. Extra care must be taken not to add the same edge multiple times, once for each of the two triangles it is adjacent to. The result is a straight-line embedding of a planar graph of size n, where each face is a triangle in the triangulation, and the dual graph is a Voronoi diagram of the set of points. Though the term is not common, we like to refer to a graph generated like this as a *Delaunay graph*. Now we have a technique to generate straight-line embeddings of graphs of arbitrary size, which is exactly what we need for benchmarking. Another advantage is that these graphs often look quite aesthetically pleasing, as seen in Figure 5.4.

Using this technique, we generate 200 Delaunay graphs of sizes 1000, 2000, 3000, and so on until 200k. Then we use a few heuristic searches to estimate pairs of vertices that are the farthest away from each other, as inputs for our Shortest Odd Walk and Shortest Odd Path algorithms. After that, we select some of the worst diversion edges farthest away from these pairs, as inputs for our Network Diversion. The intention is that each graph gets queries that are the worst or close to the worst possible case so that the size of the problem roughly matches the size of the graph.

The interested reader may visit the GitHub repository [Sim24b] to see the graphs and the Python scripts used to generate them.

Chapter 7

Conclusion

The main topic of this thesis is to solve Shortest Odd Path on undirected graphs of non-negative weights. We have presented a detailed explanation and pseudocode of Derigs' algorithm [Der85], with suggested improvements in both its presentation and performance. After that, we used it to present the first-ever efficient algorithm for Network Diversion on undirected planar graphs with non-negative edges. Although requiring planarity and non-negative weights may seem very restrictive, in practical use many real-world graphs fit the criteria.

We have also presented algorithms to solve some minor problems like Shortest Odd Walk and Shortest Detour Path. We have successfully implemented all of these algorithms in Rust and tested them thoroughly. All algorithms have been benchmarked to show that their theoretical running times match the practical running times. In particular, we show that we can solve Shortest Odd Path and Network Diversion on sparse graphs of 200k vertices in 0.5s and 1.8s, respectively.

Bibliography

[BD83] Michael O. Ball and Ulrich Derigs. An analysis of alternative strategies for implementing matching algorithms. *Networks - An International Journal*, 13(4), 1983.

[CWN13] Christopher A. Cullenbine, R. Kevin Wood, and Alexandra M. Newman. Theoretical and computational advances for network diversion. *Networks - An International Journal*, 62(3), 2013.

[Der85] Ulrich Dergis. An efficient dijkstra-like labeling method for computing shortest odd-/even paths. *Information Processing Letters*, 21(5), 1985.

[Dra24] Pål Grønås Drange. Unpublished theory. Personal communication, 2024.

[Edm65] Jack Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Institute of Standards and Technology*, 69B(1), 1965.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2), 1972.

[LCH$^+$05] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In Claudia Bauzer Medeiros, Max J. Egenhofer, and Elisa Bertino, editors, *Advances in Spatial and Temporal Databases*, pages 273–290, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[LP84] Andrea S. LaPaugh and Christos H. Papadimitriou. The even-path problem for graphs and digraphs. *Networks*, 14:507–513, 1984.

[Nis88] Takao Nishizeki. *Planar Graphs: Theory and Algorithms.* Amsterdam ; New York : North-Holland ; New York, N.Y. : Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., 1988.

[RA15] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[RAS19] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-scale attributed node embedding, 2019.

[Sim24a] Steinar Simonnes. Diverting networks with odd paths. https://github.com/SteinarSi/DivertingNetworksWithOddPaths.git, 2024.

[Sim24b] Steinar Simonnes. Shortest odd path. https://github.com/SteinarSi/ShortestOddPath.git, 2024.

[SS23] Ildikó Schlotter and András Sebő. Odd paths, cycles and t-joins: Connections and algorithms, 2023.

[Tho85] Carsten Thomassen. Even cycles in directed graphs. *European Journal of Combinatorics*, 6(1):85–89, 1985.

Appendix A

The full, uninterrupted pseudocode for Shortest Odd Path

Here is the full pseudocode in one big code block. For explanations and discussion of different variants, see Section 4.3.

Code Listing A.1: Algorithm for Shortest Odd Path

```
fn main(input_graph, s, t){
    init(input_graph, s, t);

    control();

    if d_minus[t] == ∞ {
        return None;
    }
    cost = d_minus[t];
    path = backtrack();

    return Some(cost, path);
}

fn init(input_graph, s, t) {
    graph = create_mirror_graph(input_graph);

    for u in 0..n {
        d_plus[u] = ∞;
        d_minus[u] = ∞;
        pred[u] = null;
        completed[u] = false;
        basis[u] = u;
        in_current_blossom[u] = false;
    }
    d_plus[s] = 0;
    completed[s] = true;

    for edge in graph[s] {
        priority_queue.push(Vertex(weight(edge), to(edge)));
        d_minus[to(edge)] = weight(edge);
        pred[to(edge)] = e;
    }
}

fn backtrack() {
    current_edge = pred[t];
    path = [current_edge];
    while from(current_edge) != s {
        current_edge = pred[mirror(from(current_edge))];
        if current_edge is from the mirror side {
            path.push(mirror(current_edge));
        }
        else {
            path.push(current_edge);
        }
    }
    return path;
}

fn control() -> bool {
    loop {
        while ! priority_queue.is_empty() {
            match priority_queue.top() {
                Vertex(_, u) => {
                    if completed[u] {
                        priority_queue.pop();
                    }
                    else {
                        break;
                    }
                },
                Blossom(_, edge) => {
                    if base_of(from(edge)) == base_of(to(edge)) {
                        priority_queue.pop();
                    }
                    else {
                        break;
                    }
                }
            }
        }

        if priority_queue.is_empty() {
            // No odd s-t-paths exist :(
            return;
        }
        match priority_queue.pop() {
            Vertex(delta, u) => {
                if u == t {
                    // We have found a shortest odd s-t-path :)
                    return;
                }
                d_plus[u] = d_minus[mirror(u)];
                scan(mirror(u));
            }
            Blossom(delta, edge) => {
                blossom(e);
            }
        }
    }
}

fn scan(u) {
    completed[u] = true;
    dist_u = d_plus[u];
    for edge in graph[u] {
        v = to(edge);
        new_dist_v = dist_u + weight(edge);

        if ! completed[v] {
            if new_dist_v < d_minus[v] {
                d_minus[v] = new_dist_v;
                pred[v] = edge;
                priority_queue.push(Vertex(new_dist_v, v));
            }
        }
        else if d_plus[v] < ∞ and base_of(u) != base_of(v) {
            priority = d_plus[u] + d_plus[v] + weight(edge);
            priority_queue.push(Blossom(priority, edge));
            if new_dist_v < d_minus[v] {
                d_minus[v] = new_dist_v;
                pred[v] = e;
            }
        }
    }
}

fn backtrack_blossom(edge) {
    p1 = [ reverse(edge) ];
    p2 = [ edge ];
    u = get_basis(to(edge));
    v = get_basis(from(edge));
    in_current_blossom[u] = true;
    in_current_blossom[v] = true;

    loop {
        if u != s {
            u = get_basis(mirror(u));
            in_current_blossom[u] = true;
            e = pred[u];
            u = get_basis(from(e));
            p1.push(e);

            // Ff true, then u is the base
            if in_current_blossom[u] {
                p1.pop();
                in_current_blossom[u] = false;

                // We remove all the edges in p2 after the base
                while p2 is not empty {
                    e = p2.last();
                    v = get_basis(from(e));
                    in_current_blossom[v] = false;
                    p2.pop();
                    if v == u {
                        break;
                    }
                }
                return (u, p1, p2);
            }
        }
        if v != s {
            v = get_basis(mirror(v));
            in_current_blossom[v] = true;
            e = pred[v];
            v = get_basis(from(e));
            p2.push(e);

            if in_current_blossom[v] {
                p2.pop();
                in_current_blossom[v] = false;

                while p1 is not empty {
                    e = p1.last();
                    u = get_basis(from(e));
                    in_current_blossom[u] = false;
                    p1.pop();
                    if u == v {
                        break;
                    }
                }
                return (v, p1, p2);
            }
        }
    }
}

fn blossom(edge) {
    (b, p1, p2) = backtrack_blossom(edge);

    to_scan1 = set_blossom_values(p1);
    to_scan2 = set_blossom_values(p2);

    set_edge_bases(b, p1);
    set_edge_bases(b, p2);

    for u in to_scan1 {
        scan(u);
    }
    for v in to_scan2 {
        scan(v);
    }
}

fn set_blossom_values(path) {
    to_scan = [];

    for edge in path {
        u = from(edge);
        v = to(edge);
        w = weight(edge);
        in_current_cycle[u] = false;
        in_current_cycle[v] = false;

        // We can set a d_minus
        if d_plus[v] + w < d_minus[u] {
            d_minus[u] = d_plus[v] + w;
            pred[u] = reverse(edge);
        }

        int m = mirror(u);
        // We can set a d_plus, and scan it
        if d_minus[u] < d_plus[m] {
            d_plus[m] = d_minus[u];
            to_scan.push(m);
        }
    }

    return to_scan;
}

fn set_base(base, u) {
    basis[u] = base;
}
fn get_base(u) {
    if u != basis[u] {
        basis[u] = get_base(basis[u]);
    }
    return basis[u];
}
```