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

Lorem ipsum dolor sit amet, his veri singulis necessitatibus ad. Nec insolens periculis ex. Te pro purto eros error, nec alia graeci placerat cu. Hinc volutpat similique no qui, ad labitur mentitum democritum sea. Sale inimicus te eum.

No eros nemore impedit his, per at salutandi eloquentiam, ea semper euismod meliore sea. Mutat scaevola cotidieque cu mel. Eum an convenire tractatos, ei duo nulla molestie, quis hendrerit et vix. In aliquam intellegam philosophia sea. At quo bonorum adipisci. Eros labitur deleniti ius in, sonet congue ius at, pro suas meis habeo no.

Acknowledgements

Est suavitate gubergren referrentur an, ex mea dolor eloquentiam, novum ludus suscipit in nec. Ea mea essent prompta constituam, has ut novum prodesset vulputate. Ad noster electram pri, nec sint accusamus dissentias at. Est ad laoreet fierent invidunt, ut per assueverit conclusionemque. An electram efficiendi mea.

`Your name`

Tuesday 11th June, 2024

Contents

List of Figures

List of Tables

Code Listings

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 are the applications almost limitless: to find the fastest route home, to find the cheapest airline tickets to Kuala Lumpur, to solve a Rubik's Cube in the fewest moves, to determine the best-case running time of an algorithm, or to move an NPC in a video game.

This thesis, however, is about a curious little variant, called the shortest *odd* path. The problem is the same except that we now only consider paths consisting of an odd number of edges. If you were to step through the graph, and start walking on your right foot, then an odd path is one where you would also end up on your right foot. Now, the applications of this variant are not remotely as obvious. It rarely, if ever, matters whether a path has 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 because many other more useful problems are much easier to solve if we already have an algorithm to determine the shortest odd path in a graph. Consider for example the problem of the shortest *bottleneck* path: find the shortest path from one vertex to another, except that we are also given a 'bottleneck' edge with the requirement that the path has to go through the bottleneck.

example of application of a bottleneck path

then talk about Shortest Odd Path, how strange it is in comparison, and that we care because actually useful problems are easier with such a subprocedure.

then talk about network diversion, and how it is actually a useful problem to solve.

then present the problem statement of the thesis, and give an overview of the contents and such.

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, when you want to find the fastest route to walk to the study hall, or if you want the cheapest combination of flights to take you to Kuala Lumpur, then both questions are really the same problem. If we remove all the details that are unnecessary to solve the problem, 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 this example as well as some other graph problems. Later, Section 2.3 defines the concept of *planar* graphs.

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

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

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

If we are working with multiple graphs at once, say two graphs G and H, then writing just V is ambiguous. In such cases do 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* $G := (V, E, from, to, weight)$ is a graph, where $weight : E \to \mathbb{R}$ 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.

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: $\forall e \in E\ \exists e' \in E : reverse(e) = e'$. If G is not undirected, we say that G is a *directed graph*. Edges in directed graphs are often drawn as arrows, while edges in undirected graphs can be drawn using just a line.

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 using just a single edge: $N(u) := \{to(e) \mid e \in E,\ from(e) = u\}$. In code, it is usually more useful to consider neighbourhoods in terms of edges. We will therefore denote $G[u]$ as the edges in G that start in u: $G[u] := \{e \mid e \in E,\ from(e) = u\}$ We also denote $deg(u) := |N(u)|$ as the size of u's neighbourhood, 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 $from(e) = u$ and $to(e) = v$. 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 (Sparse and dense graphs)**.** Let $G := (V, E, from, to)$ be a graph, let $n := |V|$, and let $m := |E|$. We say that G is a *sparse* graph if $m \in O(n)$. On the contrary, we say that G is a *dense* graph if it is not sparse.

Definition 2.1.7 (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: $\forall i \in \{1, 2, .., k-1\} : to(e_i) = from(e_{i+1})$. If $s := from(e_1)$ and $t := to(e_k)$, we say that P is an *s-t-walk* in G. Another way to denote a walk is to give 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 2.1.8 (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: $\forall i, j \in \{1, 2, .., k\} : j \neq i + 1 \rightarrow to(e_i) \neq from(e_j)$. If $s := from(e_1)$ and $t := to(e_k)$, 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 2.1.9 (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 verex, then we call is a *simple cycle*.

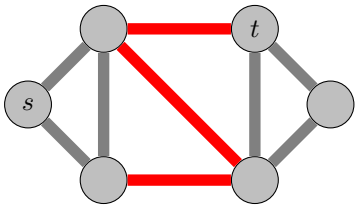
Definition 2.1.10 (The cost of a walk)**.** Let $P := [e_1, e_2, ..., e_k]$ be a walk in a weighted graph G. The *cost* of P is defined as the sum of the weights of its edges: $\sum_{i=1}^{k} weight(e_i)$. In a collection of walks, we say that the *shortest* walk is the *cheapest* one, the one with the 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.

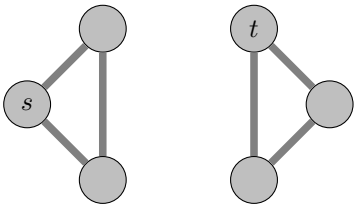
Definition 2.1.11 (Components of a graph)**.** Let G be an undirected graph. A component in G is a set of vertices of G where all vertices in the component have paths to all other vertices in the component. Moreover, this must be a *maximal* set: no other vertices in the graph can be added to the component without giving up this property.

Definition 2.1.12 (Connected graph)**.** We say that an undirected graph is a *connected* graph if it has exactly one component. If it hase more than that we call it a *disconnected* graph, and if it has less it is an *empty* graph.

Definition 2.1.13 (Cut)**.** Let $G = (V, E, from, to)$ be a connected and undirected graph. A *cut* $C \subseteq E$ of G is a subset of edges such that $(V, E \setminus C, from, to)$ 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 example we started with in the previous Section 2.1. Both problems can be represented by an abstract graph, where we want to find the shortest path from one vertex to another. 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.

The problem is called Shortest Path:

Shortest Path
Input: A graph G, two vertices $s, t \in V$
Output: the shortest s-t-path in G

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: the shortest s-t-path in G that uses an odd number of edges

We will also give 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: the shortest s-t-walk in G that uses an odd number of edges

2.2.1 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 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 reader to [Nis88] if they wish to read more.

2.3.1 Planar embeddings

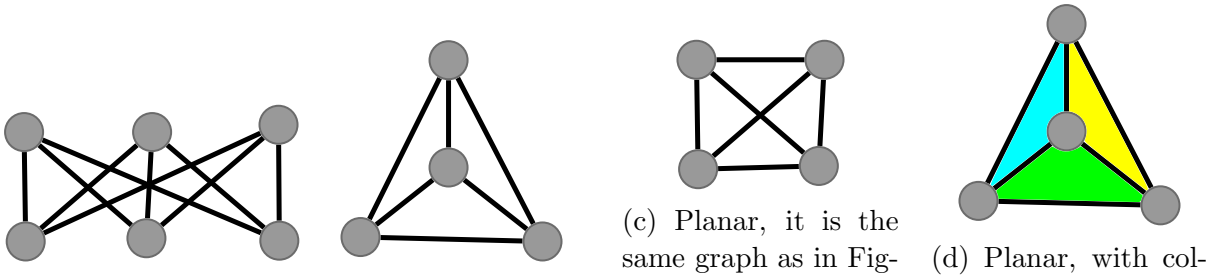
Definition 2.3.1 (Embedding)**.** Let $G = (V, E)$ be a graph. An *embedding* of G is a drawing of G on the plane $\mathbb{R}^2$, with points representing vertices and curves representing edges between their endpoints' respective points, such that none of the edges intersect each other except in their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

Definition 2.3.3 (Straight-line embedding)**.** Let G be a graph. A *straight-line embedding* of G is a planar embedding of 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 and Figure 2.2c for an example of a planar graph. Figure 2.2b also shows a planar embedding. 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 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 difficult to determine whether a given graph is planar, and to compute an appropriate embedding if it is. For all the algorithms we have implemented in this paper, if they take a planar graph as input, we have for simplicity assumed that we are also given a planar embedding of the graph. Furthermore, since all planar graphs also have straight-line embeddings, we have assumed 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.



(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

2.3.2 Duality

The next topic is easy to visualize and understand, but difficult to give formal definitions for. 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 planar graph embedded in the plane. 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 graph may yield different faces. When we refer to a face of 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 planar 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 $left, right : E(G) \to V(G^\star)$ 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

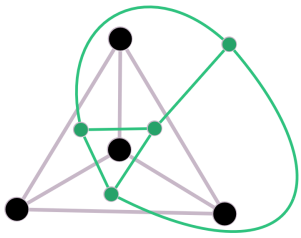
their counterparts: $weight(e^\star) := weight(e)$. See Figure 2.3a for an example of a dual graph.

Note that the dual graph is also planar, and the dual of the dual is the original graph. 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, yet 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.

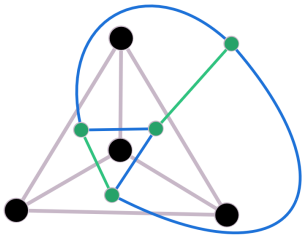
Consider dropping this, or rewriting it

Fact 2.3.1 (A simple cycle in a dual graph is a cut in the original graph)**.** Let $G := (V, E, from, to)$ be a connected planar graph, and let $C^\star$ be a simple cycle in $G^\star$. Then $C^\star$ will always correspond to a cut of 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 $(V, E \setminus C, from, to)$ is an disconnected graph of exactly two components. If the cycle $C^\star$ is not simple, then we still end up with an disconnected graph, but we may have more than just two components.

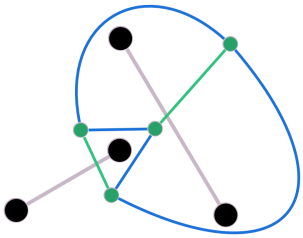
See Figure 2.3 for an example. In Figure 2.3b we have found a simple cycle in the dual graph, and if we delete the corresponding edges in the original graph we end up with the disconnected graph in Figure 2.3c.



(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 $G := (V, E, from, to)$ be a connected planar graph, where $n := |V|, m := |E|$, and f is the number of faces in any embedding of G.
Claim: $n + f - m = 2$.

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 one, 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 $m \leq 3n - 6$.

2.4 Delaunay triangulations

Here we define Delaunay triangulations

Here we define how to make a graph based on Delaunay triangulations.

Here we describe how we benchmark with these graphs.

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, and then be an odd walk instead. 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. Remember, 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, one for the best distance using an odd walk, and one 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 neighbours, and similar when we find the best odd walk.

3.2 Psuedocode

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 psuedocode 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.

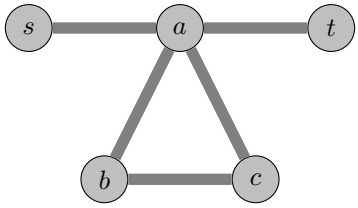


Figure 3.1: No odd s-t-path exist, yet we still have many odd s-t-walks.

3.3 Analysis

Consider Figure 3.1. There are no odd paths from s to t, yet 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 algorithm would 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. However, then a is used twice in the walk, once for each parity, and the resulting walk is not a path. Therefore can this algorithm not be used to solve Shortest Odd Path.

3.3.1 Limitations

The main limitation of the algorithm is that the edges in the input graph must either have 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 can also be directed.

3.3.2 Correctness

Let `(priority, even, u)` be the triple at the front of the queue at any point in the execution of our algorithm. Claim: If `even` is true and `even_done[u]` is false, then `priority` is the cost of the shortest even path from the source to `u`.

Proof. By induction.

The source vertex `s` has a best possible even cost of `0`, and initially we have only `(0, true, s)` in the queue. When that triple is popped the property holds in the base case.

Burde yoinke beviset herfra: https://web.engr.oregonstate.edu/ glencora/wiki/uploads/dijkstra-proof.pdf
Eller herfra: https://community.wvu.edu/ krsubramani/courses/fa05/gaoa/qen/dijk.pdf
Eller fra INF234
Også si det samme når det er odd, kanskje

□

3.3.3 Complexity

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 neighbours 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 neighbours. 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. □

3.3.4 Benchmarking

3.3.5 Discussion

Chapter 4

Shortest Odd Path

4.1 Intuition

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. The algorithm we are about to present is based on Ulrich Derigs' algorithm [Der85], though with some improvements.

4.1.1 Reduction to Shortest Alternating Path

Consider first another related problem:

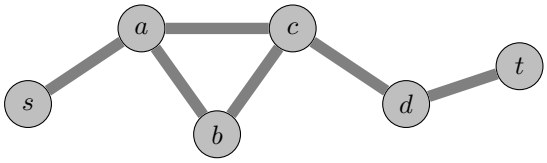
Shortest Alternating Path
Input: A weighted graph $G := (V, E)$, two vertices $s, t \in V$, and a set $F \subseteq E$
Output: the shortest s-t-path in G where every other edge used is in F.

Derig 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*.

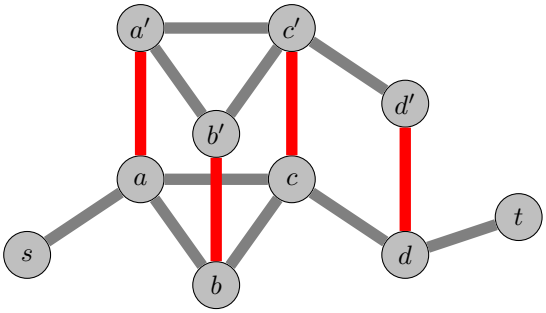
Definition 4.1.1 (Mirror graph)**.** Let $G = (V, E)$ be a graph, and $s, t \in V$ be two vertices. We construct a new graph $H \supset G$, where for each vertex $u \in V \setminus \{s, t\}$ we add a 'mirror' vertex u', and a connecting 'mirror' edge between them. The vertices in $V(H)$ that are also in $V(G)$ are referred to as the 'real' vertices, and the newly added vertices are referred to as the 'mirror' vertices. In addition, for any vertex $u \in V(H) \setminus \{s, t\}$, real or not, we define $mirror(u)$ as u's mirror on the other side. We usually label mirror vertices with an $'$ at the end of the real counterpart's label. For example, if G is the graph in Figure 4.1a, then Figure 4.1b would be its corresponding mirror graph H.

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 H as the mirror graph of G, and let F be the set of mirror edges in H. Now (H, s, t, F) is an instance of Shortest Alternating Path.
3. Let P' be the shortest alternating path of (H, s, t, F), if one exists. If none exist, then we do not have any odd s-t-paths in G either.



(a) The input graph G



(b) The mirror graph H of G, mirror edges marked in red

4. Construct P by filtering out mirror edges from P', and for each edge $(u', v') \in E(H) \setminus (F \cup E(G))$ from the mirror side of H we replace it by the corresponding edge $(u, v) \in E(G)$ 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 H and F could look like Figure 4.1b. One of the two possible alternating paths is $P' := [(s, a), (a, a'), (a', b'), (b', b), (b, c), (c, c'), (c', d'), (d', d), (d, t)]$. When we filter out mirror edges and replace edges from the mirror side with their real counterparts, we end up with $P := [(s, a), (a, b), (b, c), (c, d), (d, t)]$, which is one of the two possible odd paths of G.

dette kan kanskje formuleres bedre?

To see why the reduction works, 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 do not have to go to the other side. Therefore, to reach a neighbour of t, we must have used an even number of mirror edges and an even number of non-mirror edges, and 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 H is the shortest such path, then the corresponding path in G must also be the shortest odd s-t-path in G. The reduction works also in the weighted case, as long as each edge (u', v') on the mirror side get the same weight as their real counterpart, and all the mirror edges get the same (usually 0) weight. The interested reader may see [Der85] for more details on this reduction.

Ball and Derigs [MOB83] have shown how to efficiently solve Shortest Alternating Path. In their algorithms, subgraphs are shrunk into psuedonodes whenever possible, to make the graph smaller. The drawback is that certain psuedonodes 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, with the exception of s and t, a perfect matching of H, and we will therefore never have to expand psuedonodes after shrinking them. The curious reader may visit [MOB83] for more on these algorithms and why our almost-perfect matching is a simpler case.

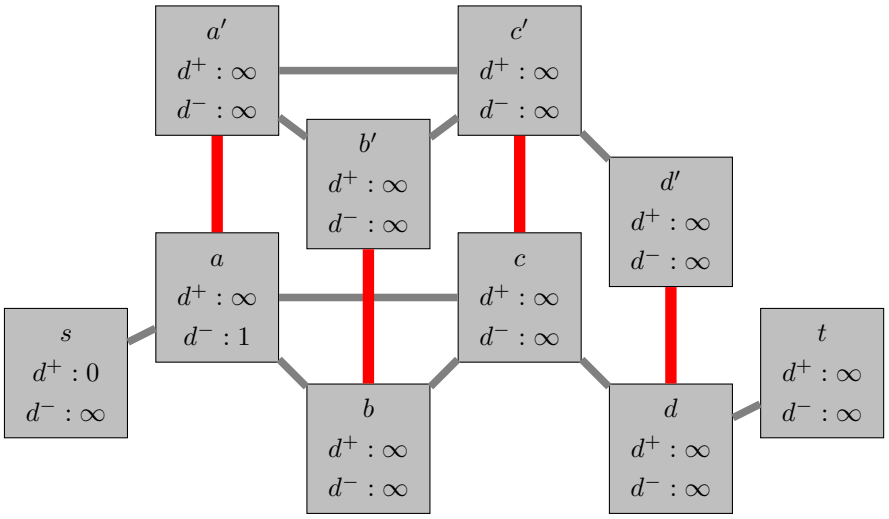
4.1.2 The idea for our Shortest Alternating Path algorithm

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 like explained in 4.1.1, 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 $u \in V(H)$, we denote

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

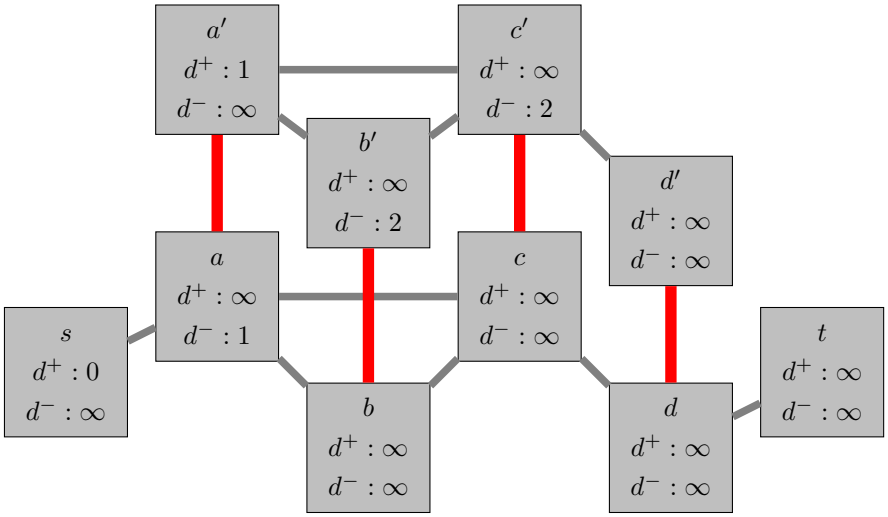
Initally these are either ∞ or undefined, except for the source vertex s, where we can set $d_s^+ := 0$. Then, for each edge $(s,u) \in N(s)$, we can set $d_u^- := weight((s,u))$, $pred_u := (s,u)$, and add u to our priority queue with priority $2 \cdot weight((s,u))$.

We visualize it in the diagram below.



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 neighbour $e \in G[a']$, and see if our new value $d_{a'}^+ + weight(e)$ is better than the previous value $d_{to(edge)}^-$. That is the case for both b' and

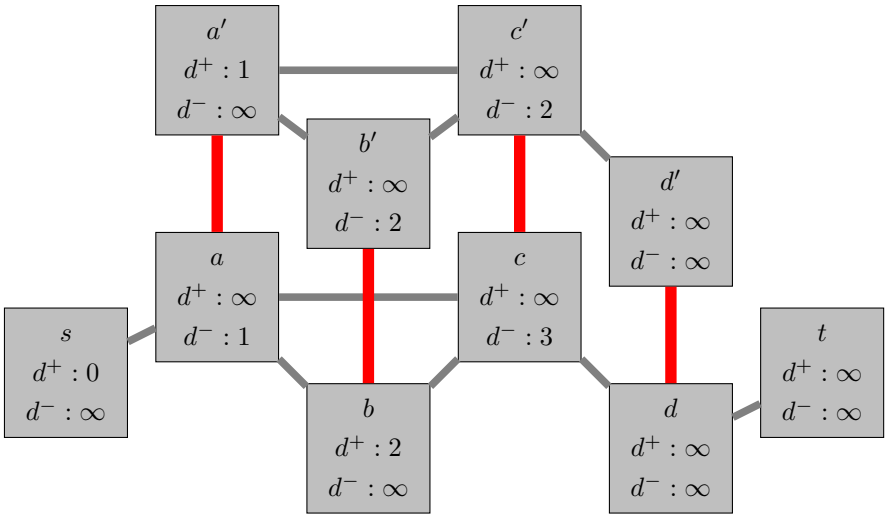
c', so we update their values and add them to the queue. Their priorities in the queue is equal to twice their d^- values, which is $2 \cdot 2 = 4$ for both of them.



Queue:

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

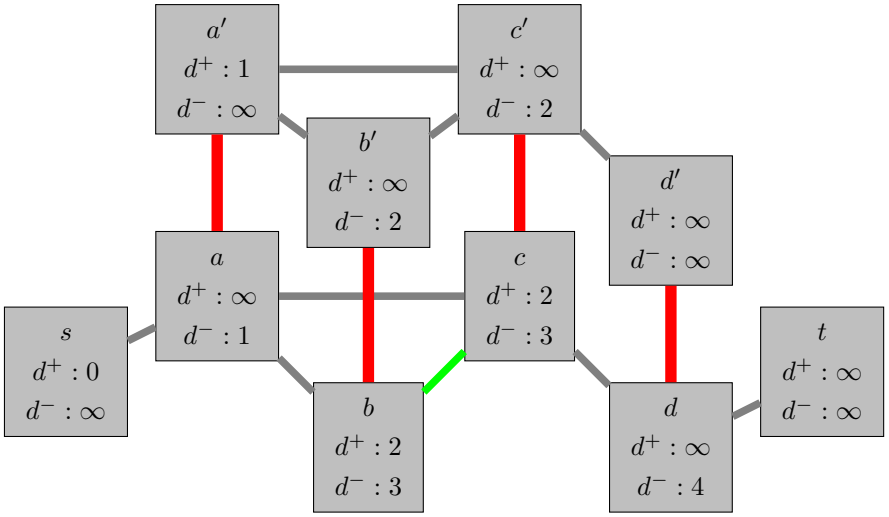
The next vertex in the queue is b', so we set $d_b^+ := d_{b'}^-$ and scan b':



Queue:

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

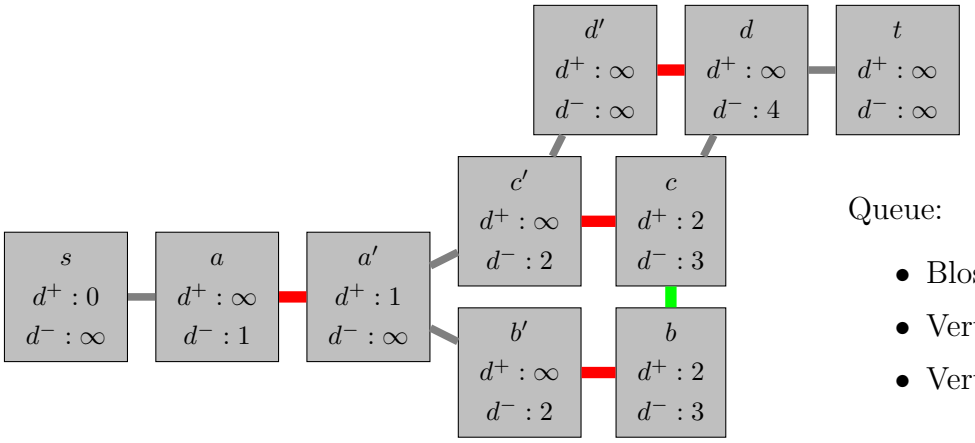
Now c' is the next in the queue, we set $d_c^+ := d^- c'$ and scan c'. This is where the interesting part happens: now we have set both d^+ and d^- for b and c, and that means that we have found an odd cycle in the graph. The edge between them, (b, c), is called the *blossom edge*, and is marked in green.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



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].



Queue:

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

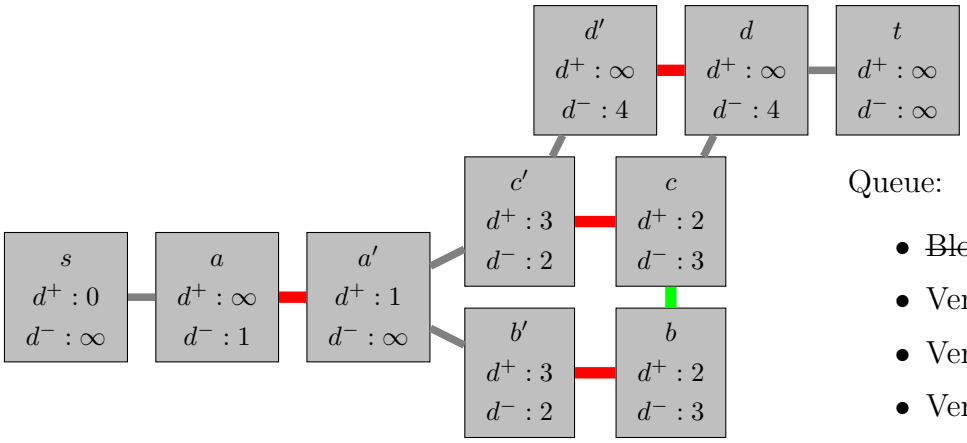
The first reason why we care about this blossom is because now we can immedeately set the final, optimal d^- and d^+ for all the vertices in the blossom. That is because we now have two alternating paths to each vertex, one goes around the cycle while the other takes the shortcut. One of these ends up on a mirror edge, and the other on a normal edge. 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 $d_u^+ = d_{mirror(u)}^-$.

- If we can improve d_u^- by coming from its neighbour in the blossom, we do so.

After all these values have been set, we immedeately 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^-.



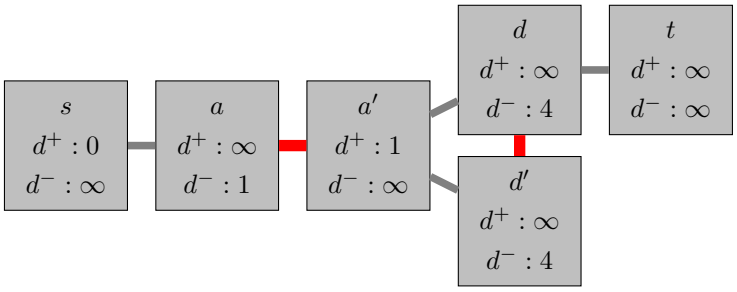
Queue:

- ~~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 it 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 verties 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.

If the reader is familiar with the more general Shortest Alternating Path algorithm [MOB83] or the original blossom algorithm [Edm65], and worry that such psuedonodes 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.

trenger vi si dette? Droppe det? Formulere det annerledes?



Queue:

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

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 4.1.1. 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.

4.2 Psuedocode

Here we give a detailed psuedocode of the algorithm, along with explanations of some of the finer details should the interested reader consider implementing the algorithm on their own. For all 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

link github here

.

4.2.1 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 we never shrunk the blossoms into psuedonodes, 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 from(current_edge) < input_graph.n() {
            path.push(current_edge);
        }
        else {
            path.push(shift_edge_by(current_edge, -input_graph.n()));
        }
    }
    return path;
}
```

4.2.2 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 scan it once, so we discard those that have already been scanned. Each blossom consists of many edges, each of which can be put on the queue, and those too we want to handle only once. If the two endpoints of a 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 discared.

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(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);
            }
        }
    }
}
```

4.2.3 Scanning vertices

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_queue.push(Blossom(d_plus[u] + d_plus[v] +
                ↪ weight(edge)));
            if new_dist_v < d_minus[v] {
                d_minus[v] = new_dist_v;
                pred[v] = e;
            }
        }
    }
```

```
23 }
```

Bør kanskje skrive noe lurt her

4.2.4 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 from $from(e)$ and one from $to(e)$, and backtrack towards the source while alternating normal and mirror 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 this too would run in linear time even if we somehow know beforehand which path needs the fewest edges.

Instead, we iteratively 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 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:

- The paths may not be of equal length, even if the graph is unweighted.
- The paths may not have any edges at all other than the blossom edge.
- 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 psuedonode after computing it.
- 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.
- It is difficult to alternate through mirror and normal edges, while also alternating between computing two separate paths.

Here is our solution. It returns the base and two lists of all the normal edges that make up blossom. That includes the blossom edge itself, which is part of both paths.

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);

            // 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 {
            // *Here we do the same for the other path*
        }
    }
}
```

4.2.5 Computing blossoms

To compute a blossom, we first have to determine its base and its edges, like 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 involved vertices. Afterwards 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 to seperate

functions to improve readability. However, the scans may only be performed after all the vertices in both list 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);
        }
    }
```

4.2.6 Setting the base of blossoms and psuedonodes

When we have found and computed a blossom, we shrink it into a psuedonode 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 they do, they are in fact already in the same psuedonode 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 oberserver pattern, where each vertex u keeps a record of the vertices that have u as its base. Initially $dependents[u] = []$ 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? Sometimes it is. We benchmark both versions and discuss the results in Section 4.4.3.1.

4.3 Improvements on Derigs' algorithm

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

First of all, 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. Firstly, only mirror edges are ever added to the tree, so the disconnected 'tree' would not be a tree at all. Secondly, 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 distance to the source, so even if we added actual edges to the tree it would not be a tree. Lastly, we found that the notation was misleading and overly complex for what really should be a simple concept. We have replaced this with 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`. It does the job.

Secondly, we have chosen to utilize sum types to have one priority queue with both vertices and blossom edges in one. The old algorithm used two priority queues that it always had to query together, which was difficult to read and debug. We find that combining them into one queue simplifies the code greatly. The way we compare their priorities is also different: vertices 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. Again do we find this simpler, and we no longer have to convert integer weights to floating points just to prioritize them correctly.

Thirdly, we came up with a new data structure to keep and update the basis of each vertex. See Section 4.2.6 for a discussion of different structures, and Subsection 4.4.3.1 for an empirical analysis of the improvement.

4.4 Analysis

4.4.1 Limitations

The algorithm does not solve Shortest Odd Path in the absolute general case, but rather has some limitations. These are:

- The input graph must be undirected, otherwise we cannot use blossoms the way we do.
- The edges must have either non-negative weights or no weights at all, otherwise we cannot guarantee that d_u^+ and d_u^- are correct when we scan a vertex u.

Are there more limitations? Should this section go somewhere else?

4.4.2 Complexity

In this section we will give a theoretical analysis of our algorithm's running time.

Proof of the running time

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 H 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 neighbours. 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 complicated. Thanks to the overly complicated code in our `backtrack_blossom` procedure in Section 4.2.4, 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 psuedonode 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 therefore never exceed $O(n)$.

Lastly, we have the main loop, which iteratively pop 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 neighbours. That is a total of $O(m)$ vertices in the queue, for a total cost of $O(m)$. Though it is unlikely, in the extreme case all edges may be put in the queue 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 $O(n+m)+O(n+m)+O(n)+O(m)+O(m \cdot \log m) = O(m \cdot \log m)$. If the graph is simple, then we can simplify it further to $O(m \cdot \log n^2) = O(m \cdot 2 \cdot \log n) = O(m \cdot \log n)$. □

Running times of 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 in Chapter 5 we will use it on planar graphs, where $m \leq 3n - 6$, as shown in Corollary 2.3.2.

We should note, however, that there are also other known polynomial algorithms for Odd Shortest Path. In the same paper that Derigs gave the algorithm of this chapter, he also gave 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 $O(n^2)$ algorithm may be preferable to our $O(m \log n)$ algorithm.

4.4.3 Benchmarking

In this section we will give an empirical analysis of our algorithm's running time.

4.4.3.1 Testing different data structures for the Basis

As shown and discussed in Section 4.2.6, we have implemented 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 6 different real-life graphs, and we show the results below. We ran our shortest odd path algorithm on each graph 40 times, 20 for each structure, and noted the fastest times for each. The vertices with id's 0 and n-1 were chosen as the source and target to paths for, respectively.

reformat

Graph	n	m	Observer	UF	Change
Oldenburg	6105	7035	5.1555 ms	4.6955 ms	-11.277%
San Joaquin County	18263	23874	6.7046 ms	5.7347 ms	-14.212%
Cali Road Network	21048	21693	19.280 ms	19.119 ms	+0.1590%
Musae Github [RAS19]	37700	289003	21.375 ms	19.438 ms	-2.4708%
SF Road Network	174956	223001	93.494 ms	87.570 ms	-6.3364%
Pokec Social Network [LT12]	1632804	30622565	13.225 s	13.774 s	+4.1543%

cite the rest of the graphs

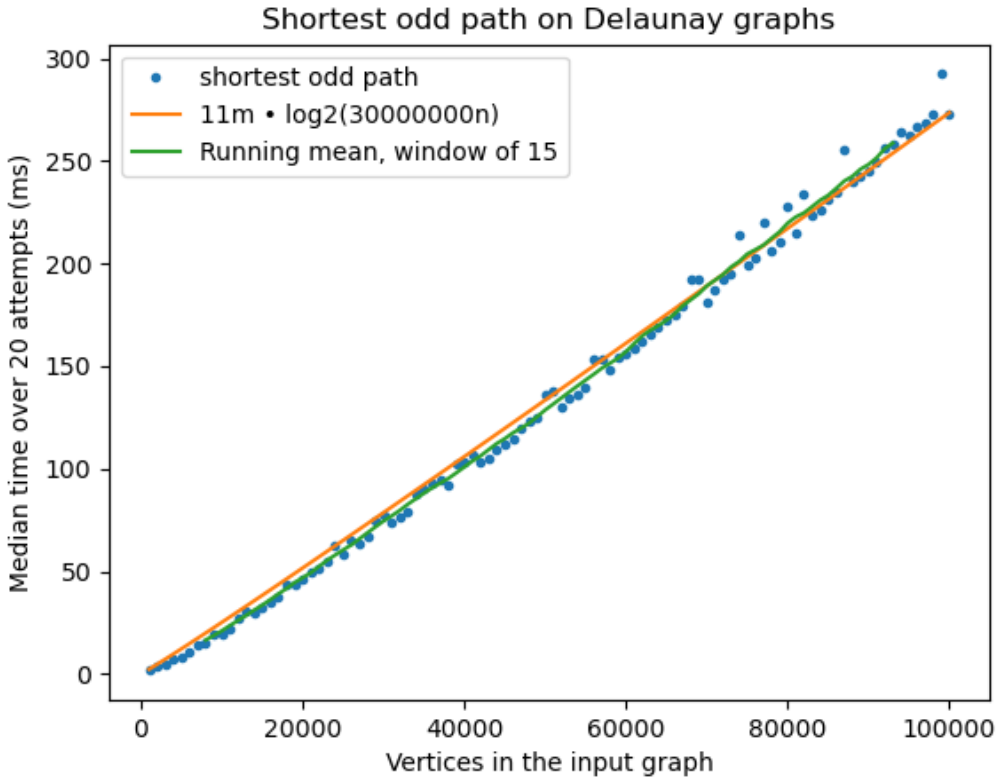
Also test on Delaunay graphs

The results are a little mixed. The UnionFind-based structure spends 4.2% more time on the huge Pokec Social Network graph. However, it also either outperforms or does as well as the Observer-based structure on all the other graphs, at best shaving off 14.2% of the time spent on the San Joaquin County graph. Since the UnionFind-based structure outperforms the Observer-based structure on average, we have chosen to use that one for the remainder of this thesis.

4.4.3.2 Running times on Delaunay graphs

Like explained in Section 2.4, we have generated 101 Delaunay graphs of sizes 1000 to 100 000. For each graph, we have found a sourec and target with the highest possible shortest path between them, and run our Shortest Odd Path algorithm. The algorithm gets 20 attempts on each input graph, and we take the median and plot it below.

The theoretical running time is $O(m \log n)$, and we have tried our best to come up with constants to create a running time function that matches our algorithm. Remember that in Delaunay graphs we have that $m \leq 6n$, so we have set $m := 6n$ in this function.



As we can see, the running times grow almost linearly as the inputs grow larger. The slight upwards curve is barely noticable. This is to be expected with a linearithmic theoretical running time.

4.4.4 Discussion

The focus of this thesis is to create an algorithm that performs well on sparse graphs, especially planar graphs, which is why we consider the running times mainly 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.

write more here

Chapter 5

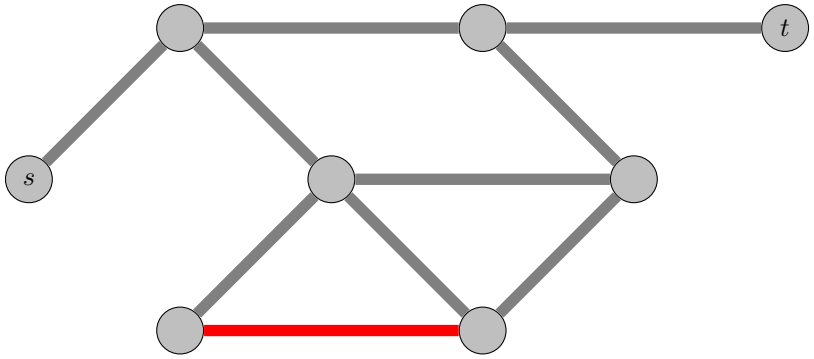
Network Diversion

5.1 Introduction to Network Diversion

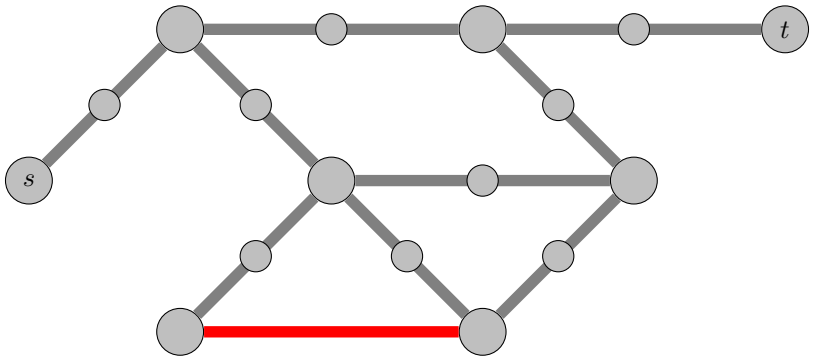
What is Network Diversion? Find out

NETWORK DIVERSION
Input: A weighted graph $G := (V, E, from, to, weight)$, two vertices $s, t \in V$, a *diversion edge* $d \in E$
Output: A *diversion set* $D \subseteq E$ of minimum weight such that all s-t-paths in $(V, E \setminus D, from, to, weight)$ 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, but being able to switch between formulations of the problem makes it easier to solve them.



(a) An instance of Shortest Bottleneck Path, bottleneck marked in red.



(b) All edges except the bottleneck have been subdivided.

Figure 5.1: Shortest Bottleneck Path reduced to Shortest Odd Path by subdividing edges.

5.2 Intuition

5.2.1 Bottleneck Paths

Before we reveal the algorithm for Network Diversion, we will first look at a curious little problem that we call Shortest Bottleneck Path:

Shortest Bottleneck Path
Input: A graph G, two vertices $s, t \in V$, and a 'bottleneck' edge $b \in E$
Output: the shortest s-t-path in G that goes through the bottleneck b

There is no obvious way to solve Shortest Bottleneck Path. One might attempt to find the shortest paths from s to $from(b)$ and from $to(b)$ to t, but those two paths might overlap and reuse the same vertices, and therefore would their concatenation not necessarily be a path but just a walk instead.

Instead we create a new graph H, by subdividing all edges in G *except* b, like seen in figure 5.1. The key point to see here is that any odd s-t-path in H must necessarily go through the bottleneck, 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 bottleneck edge, and from then on we step on all vertices from G using our right leg. If we require that we must end at t on our right leg, then the path must be odd, and any odd path must go through the bottleneck. 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 Bottleneck Path in G.

If we extend the problem to have multiple bottleneck edges, and we have to go through all of them, then our idea will not work. That is good, because otherwise we would have solved the Traveling Salesman Problem in polynomial time and complexity theory as we know it would break down. The problem is that we have no way of knowing whether we have used the marked edges 1, or 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.

Fact check

5.2.2 From a dual cycle to a real diversion

seriøst bedre tittel enn dette

Remember, we want to find the minimum minimal s-t-cut that includes d. We will start with a cycle in the dual graph and add restrictions until we have what we want.

Trenger bedre intro

Let us start off by finding a path in the dual graph, from and to the regions to the left and right of d, Then, we add $d^\star$ to the path, to make it a cycle. As explained in Fact 2.3.1, this simple cycle in the dual graph must correspond to a cut in the original graph. Since it both includes $d^\star$ and does not repeat any other edges or vertices, it corresponds to a minimal cut that uses d.

This cut is not necessarily an s-t-cut in G. To be such a cut, the cycle needs to go 'around' either s or t, but not both. Otherwise s and t would end up in the same component. There is no obvious way to force that, but we will show an as for now unpublished trick. Find first any s-t-path in G that does not use the diversion edge d. Then subdivide all the edges in the dual graph, except those that cross edges in that s-t-path. If we now find an *odd* path from and to the left and right regions of d, then

this must be a path that crosses the s-t-path an odd number of times. Therefore, the s-t-path must have exactly one endpoint on the 'inside' of the cycle, and the other on the 'outside'. That means it cuts off s from t, and is therefore an s-t-cut of G.

If this odd path is also the shortest odd path, then the corresponding cut is the minimum minimal s-t-cut of G that includes d, and the solution to this instance of Network Diversion.

more intuition for ND

5.3 Psuedocode

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)) {
                // There are no s-t-paths that go through d,
                //  and therefore no way to divert the network.
                None => return None;
                Some(cost, odd_path) {
                    diversion = [e for e* in
                        ↪ un_subdivide_edges(odd_path)];
                    return Some(cost, diversion);
                }
            }
        }
    }
}
```

Trenger å se på dette igjen en annen gang, med ferske øyne

5.4 Analysis

5.4.1 Limitations

Whether Network Diversion in the general case can be solved in polynomial time is still an open problem. This algorithm does not solve the general case, but rather the special case where the input graph...

- is planar,
- is undirected,
- has edges of either non-negative weights or no weights at all.

In addition, even though the algorithm as described here will also work on non-simple graphs, we implemented the algorithm with the assumption that the graph is simple. See Chapter 6 for more details.

vurder å skrive dette i kodekapitlet istedet. Eller også ta med at vi trenger en straight-line embedding.

5.4.2 Complexity

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 $O(n') = O(n)$.

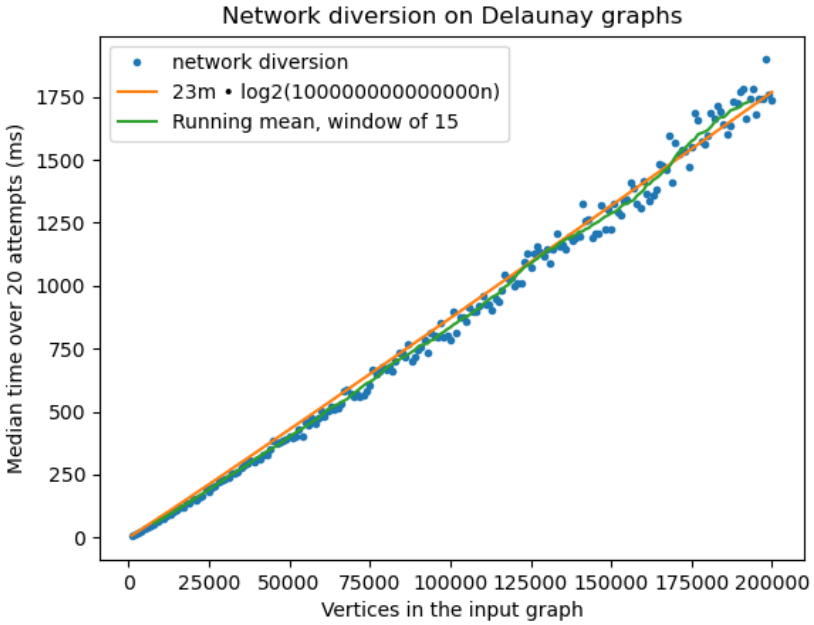
In total, we have a running time of $O(n + m) + O(m \log n) + O(n) = O(m \log n)$. Since G is planar we have that $m \in O(n)$, and we can simplify the complexity to just $O(n \log n)$, which completes the proof. □

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.

5.4.3 Benchmarking

We compare the theoretical and practical running times on Delaunay graphs, like we did in Section 4.4.3.2. For each graph, we have picked a source and target vertex of maximum distance between each other, and picked three edges in the graph as diversion edges. We pick whichever diversion edge leads to the worst running time over 20 attempts, and plot the median over those 20 attempts.

Here too have we tried to create a function out of the theoretical running time of $O(n \log n)$, this time with different constants.



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.

Benchmarking on real graphs

Visualize the diversion set

5.4.4 Discussion

Write some kickass discussion here

Chapter 6

Codebase

Here we write about the codebase. Language, testing, frameworks, design choices, and how the algorithms can be used. Maye also mention Minimum Maximal Matching.

Chapter 7

Conclusion

Bibliography

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

[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.

[LT12] M. Zabovsky L. Takac. Pokec social network, 2012.

[MOB83] Ulrich Derigs Michael O. Ball. An analysis of alternative strategies for implementing matching algorithms. *Networks - An International Journal*, 13(4), 1983.

[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.

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

Appendix A

Generated code from Protocol buffers

Here we paste in the full psuedocode