University of Bergen
Department of Informatics

Title of your master thesis

Author: Steinar Simonnes

Supervisor: Pål Grønås Drange

UNIVERSITETET I BERGEN

Det matematisk-naturvitenskapelige fakultet

May, 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`
Friday 31st May, 2024

Contents

List of Figures

List of Tables

Code Listings

Chapter 1

Introduction

TODO: talk about Shortest Path, how cool of a problem it is, and how we have solved it in a bunch of different ways already.

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

Chapter 2

Preliminaries

2.1 Graphs

In algorithms, we often use graphs as an abstract structure to represent the fundamental problem behind an algorithms 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 Lumphur, 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.

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

Definition 2.1.2 (Weighted graph)**.** A *weighted graph* $G := (V, E, from, to, w)$ is a graph, where $w : E \to \mathbb{R}$ is the *weight* of each edge. If a graph is not weighted, we often treat it like it is weighted with all edges having unit weight, a weight of 1. An algorithm intended for weighted graphs will therefore often work on unweighted graph 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.

TODO figures for graphs

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, commonly denoted as either $N(u)$ or $G[u]$, is defined as the vertices in G that are reachable from u using just a single edge: $N(u) := \{to(e) \mid e \in E,\ from(e) == u\}$.

Definition 2.1.5 (Walk)**.** A *walk* $P := [p_1, p_2, ..., p_k]$ in a graph G is a sequence of edges where each edge ends where the next one starts: $\forall i \in \{1, 2, .., k-1\} : to(p_i) = from(p_{i+1})$. If $s := from(p_1)$ and $t := to(p_k)$, we say that P is an *s-t-walk* in G.

Definition 2.1.6 (Path)**.** A *path* $P := [p_1, p_2, ..., p_k]$ in a graph G is a walk with the extra requirement that each vertex is used more than once: $\forall i, j \in \{1, 2, .., k\} : j \neq i + 1 \rightarrow to(p_i) \neq from(p_j)$. If $s := from(p_1)$ and $t := to(p_k)$, 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 2.1.7 (Cycle)**.** A *cycle* in a graph is a walk that ends in the same vertex it started. If it does not repeat any vertices other than the start and end vertex, then we call is a *simple cycle*.

Definition 2.1.8 (The cost of a path (/walk))**.** Let $P := [p_1, p_2, ..., p_k]$ be a path (/walk) in a weighted graph G. The *cost* of P is defined as the sum of its edges: $\sum_{i=1}^{k} w(p_i)$. In a collection of paths (/walks), we say that the *shortest* path (/walk) is the *cheapest* one, the one with the lowest cost. Likewise for the longest and most expensive path (/walk). Note that in some literature, *shortest* may instead mean *fewest edges*, and the term *length* could mean both the number of edges or the cost of a path. For that ambiguous reason, we will from now on avoid the word *length*, and *shortest* will always mean *cheapest*.

Now we are ready to define the underlying problem of the example we started with. 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 therefore represent whatever we want them to.

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

Definition 2.1.9 (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 and keep this property.

Definition 2.1.10 (Connected graph)**.** We say that an undirected graph is a *connected* graph if it has only one component.

Definition 2.1.11 (Cut)**.** Let $G = (V, E)$ be an connected and undirected graph. A *cut* $\mathbb{C} \subseteq E$ of G is a subset of edges such that $(V, E \setminus \mathbb{C})$ is an unconnected graph of exactly two components. If $s, t \in V$ end up in separate components after the cut, we denote $\mathbb{C}$ as an *s-t-cut* in G.

2.2 Planarity

A fascinating class of graphs that we will focus on in this thesis 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.

2.2.1 Planar embeddings

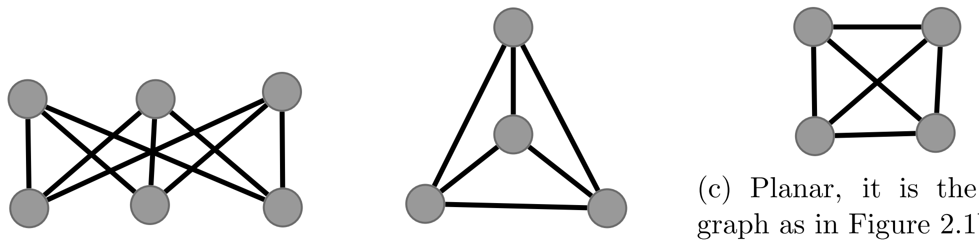
Definition 2.2.1 (Embedding)**.** Let $G = (V, E)$ be a graph. An *embedding* of G is a drawing of G in the plane, with points representing vertices and curves representing edges between their endpoints' respective points, such that none of the edges intersect each other other than in their endpoints.

Definition 2.2.2 (Planar graph)**.** We say that a graph G is a *planar graph* if there exists a planar embedding of G.

Definition 2.2.3 (Straight-line embedding)**.** Let $G = (V, E)$ 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 2.1b and Figure 2.1c for an example of a planar graph. Figure 2.1b also shows a planar embedding. Figure 2.1a 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 an edge does not cross any other edges it can be as curved as we want.

It is generally difficult to verify whether a given graph is a planar graph, 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 2.1b, and can be drawn as such.

Figure 2.1: Examples of planar and non-planar graphs

2.2.2 Duality

Definition 2.2.4 (Face)**.** TODO Dette er kanskje ikke helt sant, nå blir ikke utsiden en region, siden resten av grafen er inni. Let G be a planar graph, with an embedding in the plane. A *face* of G in this embedding is a region bounded by edges and vertices not containing any other vertices or edges.

TODO dette er en fin definisjon, men da må vi også definere hva 'induced' betyr. Let G be a planar graph. A *face* of G is a region in the plane bounded by an induced cycle of G.

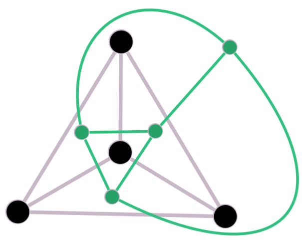
Definition 2.2.5 (Duality of planar graphs)**.** Let G be a planar graph. The *dual graph* of G, or simply the *dual* of G, is the graph $G^{dual} := (F, E^{dual})$, where

- F is the set vertices, representing faces in G
- D^{dual} is the set of edges, where two faces have an edge between them if they are adjacent in G. That is, There is an edge separating them in G. Each edge in G also has its own equivalent in E^{dual}.

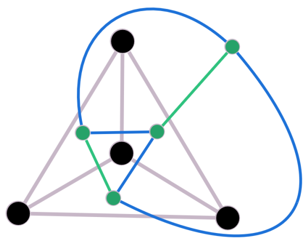
The dual graph is always planar, and the dual of the dual is the original graph. See Figure 2.2a for an example of a dual graph.

Fact 2.2.1 (A cycle in a dual graph is a cut in the original graph)**.** Let $G = (V, E)$ be a connected and planar graph, let G^{dual} be its dual graph, and let C be a simple cycle in G^{dual}. Then C will always correspond to a cut of G. If we define C' as the edges that correspond to edges in C, or equivalently as the the edges that cross edges in C, then $(V, E \setminus C')$ is an unconnected graph of exactly two components.

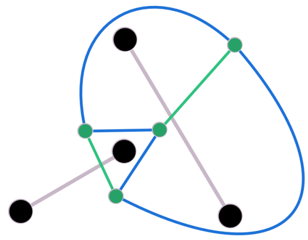
See Figure 2.2 for an example. In 2.2b we have found a simple cycle in the dual graph, and if we delete all of edges in the original graph that cross the cycle we end up with the disconnected graph in 2.2c.



(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 2.2: A simple cycle in a dual graph always corresponds to a cut in the original graph.

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.

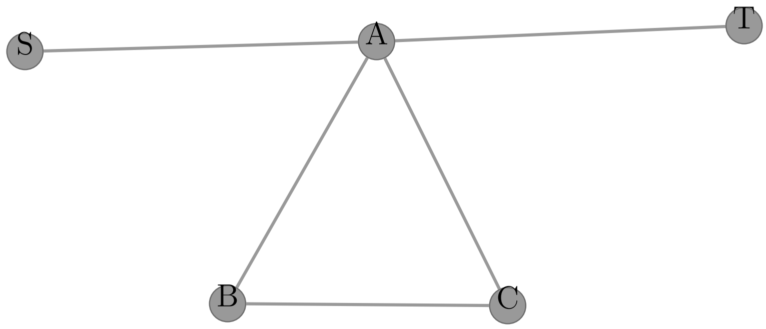


Figure 3.1: No odd s-t-path exist, but we still have many odd s-t-walks

3.3 Analysis

Consider 3.1. There are no odd paths from s to t, yet we have an infinite amount of odd walks. 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, the algorithm would search a twice, once for each parity, and the resulting walk is not a path. Therefore this algorithm cannot be used to solve Shortest Odd Path.

3.3.1 Correctness

TODO. Is this necessary?

3.3.2 Complexity

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. 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 running time of $O(m)$, and removing all of them takes a total of $O(m \cdot log\ m)$.

In total, the algorithm runs in $O(m \cdot log\ m)$.

TODO: er dette i det hele tatt riktig? Det føles feil. Wikipedia gir at Dijkstra kjører på $O((n + m) \cdot log\ n)$.

3.3.3 Benchmarking

TODO

3.3.4 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 $H \sqsupset G$, 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 4.1, then Figure 4.2 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.

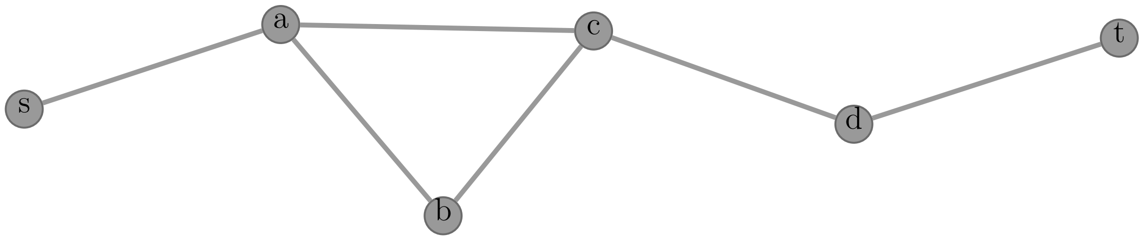


Figure 4.1: Our input graph G, for Shortest Odd Path

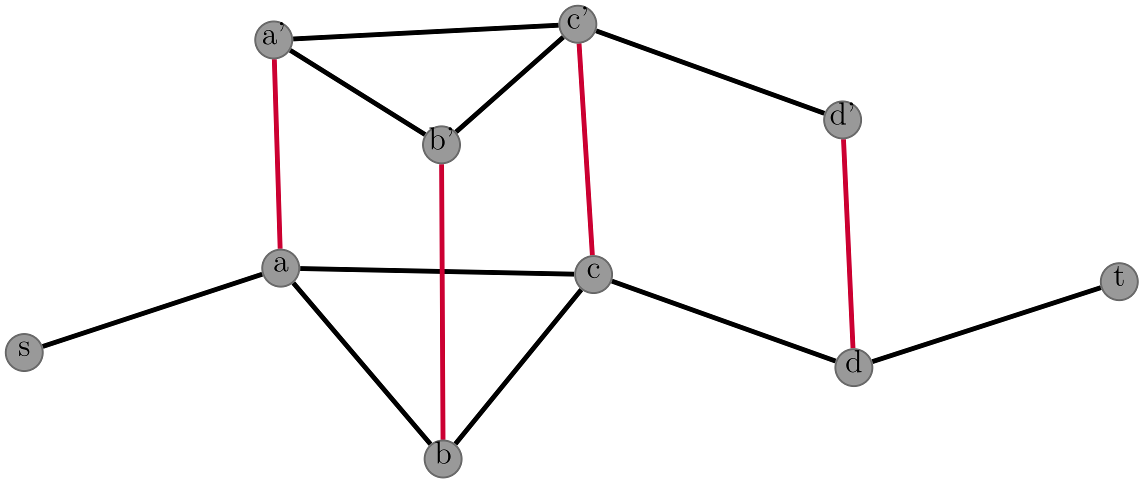


Figure 4.2: The mirror graph H of G, with mirror edges marked in red and mirror vertices labeled with an $'$

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 4.1, then H and F could look like Figure 4.2. 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.

TODO 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.

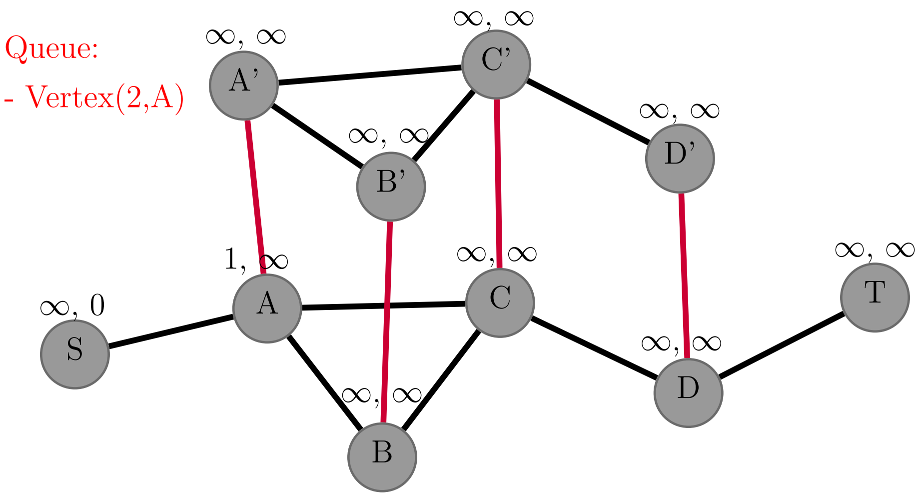
4.1.2 The idea for our Shortest Alternating Path algorithm

We will explain the general idea of our algorithm by following an example, and solve for the graph in figure 4.1. First we construct the mirror graph like explained in 4.1.1, to produce the graph in figure 4.2. Then we initialize an empty priority queue of vertices and edges to be scanned. For each vertex $u \in V(H)$, we denote

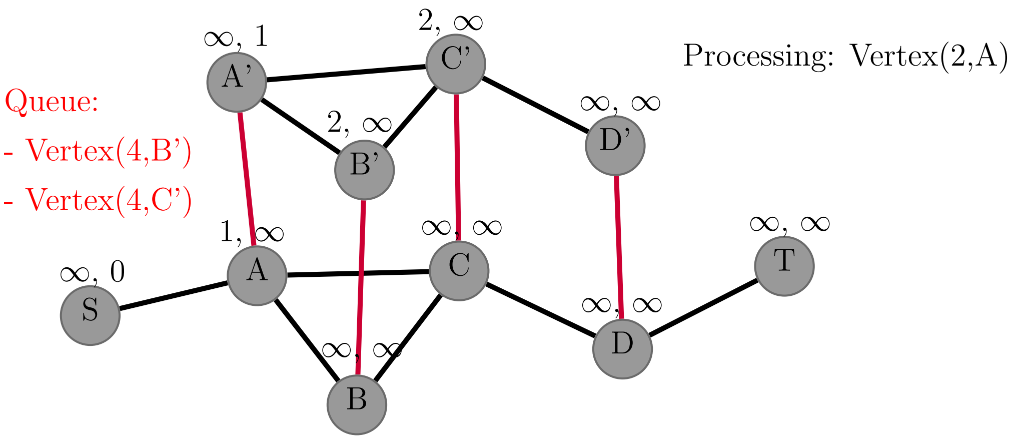
- $d_u^+ :=$ the length of the shortest alternating s-u-path ending on a mirror edge
- $d_u^- :=$ the length of the shortest alternating s-u-path ending on a non-mirror edge
- $pred_u :=$ the last edge used to find u's most recent value for d_u^-

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 $2 \cdot weight((s, u))$.

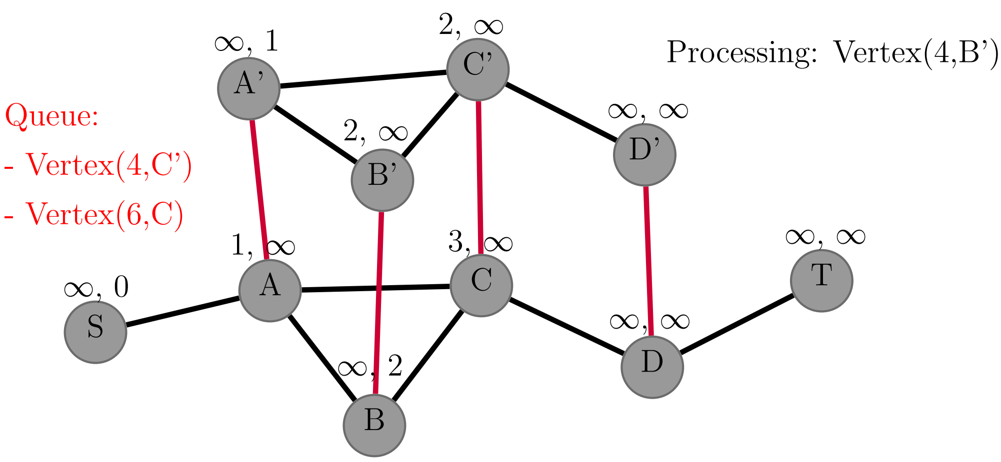
We visualize it like in the diagram below. Each vertex u has its values for d_u^- and d_u^+ to its top left and top right, respectively.



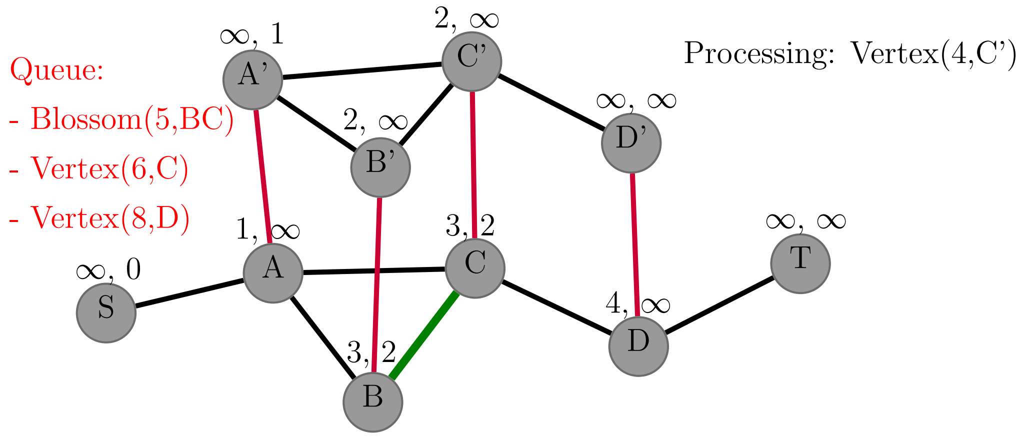
The first and only vertex in the queue is A. We pop it, set $d^+_{A'} := d^-_A$, and 'scan' A'. By that, we mean to look at each neighbour $e \in N(A')$, and see if our new value $d^+_{A'} + weight(e)$ is better than the previous value $d^-_{to(edge)}$. That is the case for both B' and C', so we update 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.



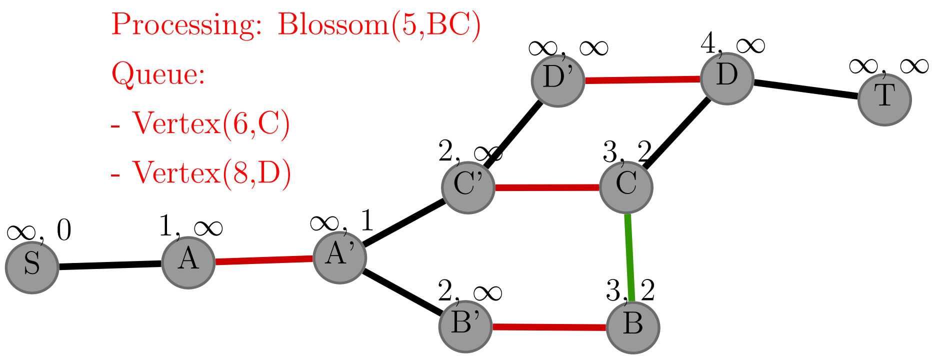
The next vertex in the queue is B', so we set $d^+_B := d^-_{B'}$ and scan B':



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 $d_C^+ + d_B^+ + weight((B, C))$.



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].

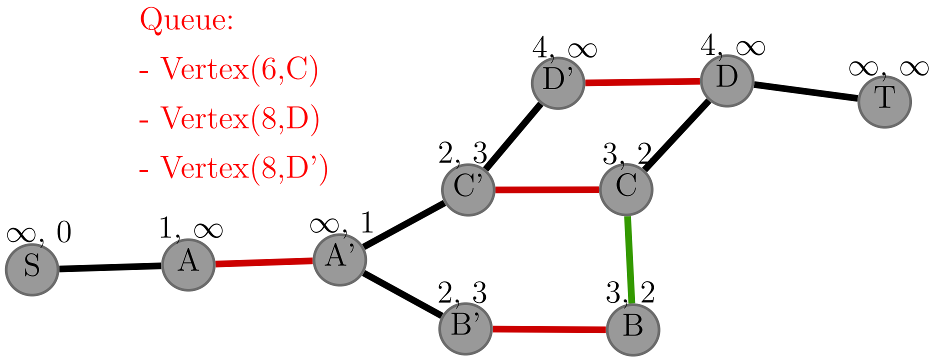


The first reason why we care about this blossom is because now we can immedeately set the final, optimal d^- and d^+ for all the vertices in the blossom. That is because we now have two alternating paths to each vertex, one goes around the cycle while the other takes the shortcut. One of these ends up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

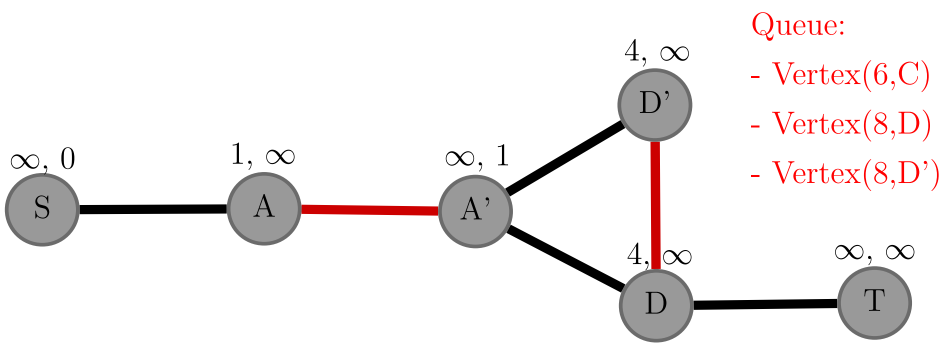
- If $d_u^+ = \infty$, we set $d_u^+ = d_{mirror(u)}^-$.
- If we can improve d_u^- by coming from its neighbour in the blossom, we do so.

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^-.



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.



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

4.2.1 Initialization and the main control loop

Code Listing 4.1: Main

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

    while ! control() {}

    if d_minus[t] == ∞ {
        // The graph is a no-instance, no odd s-t-paths exist
        return None;
    }

    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 Some(d_minus[t], path);
}
```

Code Listing 4.2: 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;
    }
    d_plus[s] = 0;
    completed[s] = true;

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

Code Listing 4.3: Control, the main loop

```
fn control() -> bool {
    while ! pq.is_empty() {
        match pq.top() {
            Vertex(_, u) => {
                if completed[u] {
                    pq.pop();
                }
```

```
                else {
                    break;
                }
            },
            Blossom(_, edge) => {
                if base_of(from(edge)) == base_of(to(edge)) {
                    pq.pop();
                }
                else {
                    break;
                }
            }
        }
    }

    if pq.is_empty() {
        // No odd s-t-paths in G exist :(
        return true;
    }
    match pq.pop() {
        Vertex(delta, l) => {
            if l == t {
                // We have found a shortest odd s-t-path has been
                    ↪ found :)
                return true;
            }
            grow(l);
        }
        Blossom(delta, edge) => {
            blossom(e);
        }
    }
    return false;
}
```

4.2.2 Scanning vertices

Code Listing 4.4: Grow

```
fn grow(l) {
    d_plus[mirror(l)] = d_minus[l];
    scan(mirror(l));
}
```

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;
                pq.push(Vertex(new_dist_v, v));
            }
        }
        else if d_plus[v] < ∞ and base_of(u) != base_of(v) {
            pq.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;
            }
        }
    }
}
```

4.2.3 Computing blossoms

Code Listing 4.6: Blossom

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

    List<int> to_scan1 = set_blossom_values(p1);
    List<int> 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 4.7: Backtrack blossom

```
fn backtrack_blossom(edge) {
    // TODO
}
```

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

4.2.4 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(b, u) {
    basis[u] = b;
    for v in 0..n {
        if basis[v] == u {
            basis[v] = b;
        }
    }
}
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 b:

Code Listing 4.11: Observer basis

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

Now we only go through the vertices that have u as their base, in time linear to the count of vertices that need to be updated.

The second version is to use a structure resembling union-find, where each disjoint set and its representative is a blossom and its base. To update the base of u we simply set the new base and do nothing else. When we require the base of a vertex we recursively query its representative's base and contract the path along the way in the style of union-find.

Code Listing 4.12: UF-like basis

```
fn set_base(b, u) {
    basis[u] = b;
}
fn get_base(u) -> int {
    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.

TODO gjør eksperimenter.

4.3 Notes on implementing the psuedocode

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 improving adjustments, and we will discuss these here.

First of all, the original algorithm used the idea of building up an 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. The problem we had with this was that only mirror edges were ever added to the tree, so the tree wouldn't be connected, and the notation was confusing and overly complex for such 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 its two d^+'s and its edge weight without dividing by two. Again do we find this simpler, and we no longer have to convert integer weights to floating points just to prioritize them correctly.

Thirdly, union-find. TODO.

TODO: tree vs completed, sum types, easier comparison and better queue priority to avoid floats, union-find,

4.5 Analysis

4.5.1 Complexity

4.5.2 Benchmarking methodology

4.5.3 Results

4.5.4 Discussion

Chapter 5

Network Diversion

5.1 Intuition

5.1.1 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.

SBP: 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 SBP. 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 simple path.

Instead we create a new graph H, by subdividing all edges in G *except* b, like seen in figure TODO. 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.

5.1.2 Extending the idea to multiple edges

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. TODO fact check. 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. And as it turns out, that is exactly what we need to solve Network Diversion.

TODO: Intuition for ND

5.2 Psuedocode

Code Listing 5.1: Main

```
network_diversion(PlanarGraph graph, s, t, Edge diversion) ->
    ↪ Option<(int, List<Edge>)> {
    let `path` be any s-t-path that does not use the diversion edge;
}
```

5.3 Analysis

5.3.1 Complexity

5.3.2 Benchmarking methodology

5.3.3 Results

5.3.4 Discussion

Chapter 6

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.

[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.