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`
Monday 27th May, 2024

Contents

List of Figures

List of Tables

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 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 (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

2.2 Planarity

TODO little introduction here, why do we care

Definition 2.2.1 (Planar graph)**.** Let G be a graph. We say that G is a *planar graph* if it is possible to draw the graph on the plane such that none of the edges intersect each other. Such a drawing is referred to as an *embedding* of G.

TODO figures of planar and non-planar graphs.

Definition 2.2.2 (Duality of planar graphs)**.** Let G be a planar graph, and choose any embedding of G in the plane.

TODO contiue the definition. WTF is a face? What *is* the dual graph?

Chapter 3

Shortest Odd Walk

TODO something something about that basic algorithm for finding a non-simple path

Chapter 4

Shortest Odd Path

4.1 Intuition

TODO

4.2 Psuedocode

Listing 4.1: Main

```
fn main(Graph input_graph, int s, int t) -> Option<(int, List<Edge>)> {
    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;
    }

    Edge current_edge = pred[t];
    List<Edge> 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);
}
```

Listing 4.2: Initialization

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

    for u in 0..n {
        d_plus[u] = ∞;
        d_minus[u] = ∞;
        pred[u] = null;
        completed[u] = false;
    }
    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;
    }
}
```

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, delta)
        }
        Blossom(delta, edge) => {
            blossom(e);
        }
    }
    return false;
}
```

Listing 4.4: Grow

```
fn grow(int l, int delta) {
    int k = mirror(l);
    d_plus[k] = delta;
    scan(k);
}
```

Listing 4.5: Scan

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

Listing 4.6: Blossom

```
fn blossom(Edge edge) {
    (int, List<Edge>, List<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);
    }
}
```

Listing 4.7: Backtrack blossom

```
fn backtrack_blossom(Edge edge) -> (int, List<Edge>, List<Edge>){
    // TODO
}
```

Listing 4.8: Set blossom values

```
fn set_blossom_values(List<Edge> path) -> List<int> {
    List<int> to_scan = [];

    for edge in path {
        int u = from(edge);
        int v = to(edge);
        int 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;
}
```

Listing 4.9: Set edge bases

```
fn set_edge_bases(int b, List<Edge> path) {
    for edge in path {
        let u = from(edge);
        let m = mirror(edge);
        set_base(u, b);
        set_base(m, b);
    }
}
```

Listing 4.10: Basis

```
fn init(Graph input_graph, int s, int t) {
    // omitted
    Graph graph = create_mirror_graph(input_graph, s, t);
    for u in 0..graph.n() {
        basis[u] = u;
    }
    // omitted
}
fn set_base(int b, int u) {
    basis[u] = b;
}
fn get_base(int u) -> int {
    if u != basis[u] {
        basis[u] = get_base(basis[u]);
    }
    return basis[u];
}
```

4.3 Notes on implementing the psuedocode

4.4 Analysis

4.4.1 Complexity

4.4.2 Benchmarking methodology

4.4.3 Results

4.4.4 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

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

Appendix A

Generated code from Protocol buffers

Listing A.1: Source code of something

```
1 System.out.println("Hello Mars");
```