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 10^{th} May, 2024

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Natum mucius vim id. Tota detracto ei sed, id sumo sapientem sed. Vim in nostro latine gloriatur, cetero vocent vim id. Erat sanctus eam te, nec assueverit necessitatibus ex, id delectus fabellas has.

Lorem ipsum dolor sit amet, iisque feugait quo eu, sed vocent commodo aliquid an. Minim suavitate dissentiet te eos. Dicunt eirmod adolescens no sed. Esse nonumy melius an mel, mei ut maiorum luptatum. Eu eum iudico scripta, movet option assueverit mel ex, mea at odio noluisse efficiendi. Ad vidisse atomorum conceptam quo, saepe volumus philosophia eos eu, delenit conceptam no usu.

Vituperata sadipscing deterruisset ei mel, at qui nonumy blandit. Delectus dissentiet et sea, ut rebum regione numquam nam, cum ex augue constituto. Te per nihil semper. Posse voluptatum qui an, aliquando democritum disputando id quo, everti perpetua cu vim. Laudem fabellas mei an, eu reprimique quaerendum usu. Quidam prompta fabellas ne est.

1.1 Background

Lorem ipsum dolor sit amet, cu graecis propriae sea. Eam feugiat docendi an, ei scripta blandit pri. Nonumes delicata reprimique nam ut. Eu suas alterum concludaturque est, ferri mucius sensibus id sed [1].

We can do glossary for acronymes and abriviations also: Software as a Service (SaaS). As you see the first time it is used, the full version is used, but the second time we use SaaS the short form is used. It is also a link to the lookup.

1.1.1 Listings

You can do listings, like in Listing 1.1

Listing 1.1: Look at this cool listing. Find the rest in Appendix A.1

```
$ java -jar myAwesomeCode.jar
```

You can also do language highlighting for instance with Golang: And in line 6 of Listing 1.2 you can see that we can ref to lines in listings.

Listing 1.2: Hello world in Golang

```go
1 package main
2
3 import "fmt"
4
5 func main() {
6     fmt.Println("hello world")
7 }
```

1.1.2 Figures

Example of a centred figure

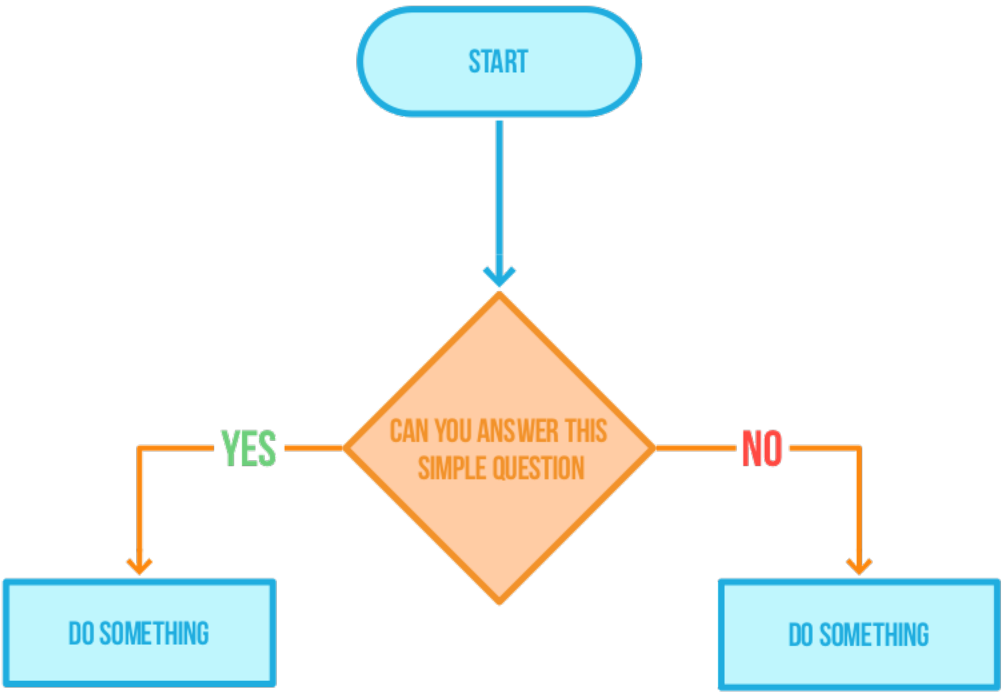

Figure 1.1: Caption for flowchart

Credit: Acme company makes everything `https://acme.com/`

1.1.3 Tables

We can also do tables. Protip: use `https://www.tablesgenerator.com/` for generating tables.

Table 1.1: Caption of table

Title1	Title2	Title3
data1	data2	data3

1.1.4 Git

Git is fun, use it!

Chapter 2

Preliminaries

2.1 Graphs

A *graph* $G = (V, E, from^G, to^G)$ is given by

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

In algorithms, we often use graphs as an abstract structure to represent the fundamental problem behind an algorithms problem without distractions.

TODO: Omskriv, dette er elendig TODO: Hva er det som egentlig er nødvendig å ha med her? Er det noen som noensinne kommer til å lese dette uten å vite hva en graf er? TODO: rettede grafer vs urettede grafer TODO: vektede grafer

2.2 Planarity

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

[1] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-10-2.

Appendix A

Generated code from Protocol buffers

Listing A.1: Source code of something

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