Into Algorithms

Ola Dahl

2024-05-05

Table of contents

Preface

This is a book about algorithms. By algorithms, I mean algorithms in the computer science sense, such as algorithms for traversing a graph, computing prime factors of a number, or finding the best path through a maze.

The book reflects my experience from learning about algorithms. I have practiced, mostly using Kattis, but also participated in on-line courses and purchased, and partly read, on-line articles and books.

I did my first Kattis problem in September 2019. Starting from a hello world style problem, I continued, with problems having difficulty level set to easy.

I was confident that I would succeed. I regarded myself as a programmer, with several years of professional and private experience of software development.

After having tried a few problems with difficulty level set to easy, I moved on to a problem with medium difficulty.

It was a problem about a graph, and about figuring out the shortest path to a goal from different places in the graph, and then deciding to take, for a given node, a path that was not a shortest path.

I made several attempts, and all of them failed.

I had heard of Dijkstra's algorithm, but never implemented it myself. I had also heard of Dynamic Programming, and even practised it during my Master thesis work. I had no clue, however, if these two topics had anything in common.

After many tries, and solving many easier problem in between, I solved the problem. This was in August 2020.

I realized that my confidence about programming was not on par with my skills for solving Kattis problems. I had a long way to go, in order to solve more problems on medium level, and I didn't even dare to try any problems with difficulty level set to hard.

I wanted to learn more. I signed up for, and completed, an excellent course series on Data Structures and Algorithms. It was not for free, but it was a very rewarding investment. The teachers were excellent, the lectures were pedagogical, and the labs were challenging.

I continued with Kattis, and managed to solve also some problems with somewhat higher difficulty level.

I had an ongoing project where I wanted to make books with layers. The idea was, and still is, to have a book with several layers, with common content, and with content specific for each layer.

For a book about programming, it makes sense, I think, to have layers that correspond to programming languages.

Out of this came the decision to write down some of my experiences of working with algorithms in the context of solving Kattis problems, as a layered book.

It is a work in progress, and I will continue to update it, here on GitHub, as time goes on.

It is written using Quarto.

Chapter 1

Introduction

There are many books on algorithms. Why would I try to write yet another one?

One reason is selfishness. When you write, you learn.

Another is that there might be others, in situations similar to mine (software developers that need to sharpen their algorithm skills), that might have some use of what I write.

Chapter 2

Sources of inspiration

I have already mentioned Kattis, which is where I started by journey into algorithms. In the beginning of that journey I discovered Sannemo (2020), which I think contains very good material about competetive programming. It also contains recommended Kattis problems to solve.

Another book about competitive programming is Laaksonen (2020). It contains good material, and it gives a recommendation to use the CSES problem set for practice, while reading the book.

A pair of books, with *The Lower Bound of Programming Contests in the 2020s* as their subtitle, and a web page with information about the books, is Halim and Halim (2018) and Halim and Halim (2020).

Another resource that I would like to mention is Algorithms for Competitive Programming, with descriptions of many algorithms, and code examples.

2.1 Outline

We start of with graphs, in Chapter 3.

Chapter 3

Graphs

We describe graphs, and some algorithms for graphs.

3.1 Notations and Terminology

A graph is defined by a set of vertices, also referred to as nodes, and a set of edges.

We use the notation V for the set of vertices and the notation E for the set of edges.

A graph g is defined as

$$g = (V, E)$$

An edge is e is defined by a pair of vertices, as

$$e = (v_1, v_2)$$

A directed graph is a graph where the vertices in each edge e are interpreted as a ordered pair (v_1, v_2), so that the edge starts at v_1 and ends at v_2.

An undirected graph is where the vertices in each edge e do not have any defined order. Their role is to indicate to which vertices the edge is connected, without any connotations of direction.

A weighted graph is a graph where each edge e is associated with a numeric value w, referred to as the weight of the edge.

3.2 Graph traversal

We can traverse a graph by starting from a node v_s, and then find a way to visit all nodes that can be reached from v_s.

If the graph is connected, in the sense that all its nodes can be reached from v_s, then the graph traversal will visit all the nodes.

If the graph is not connected, we can, after the traversal is done, pick a node that is not yet visited. We can start a new traversal from that node. When this is done we can check if all nodes are visited. If this is not the case, we can repeat the process, by picking a new node that is not yet visited and start a new traversal from that node.

In the end, we will have visited all nodes.

3.2.1 DFS

When doing a graph traversal, we must, at a given node v, determine which node to visit next. Naturally, we must select one of the neighbors of v.

If we select a neigbor of v, and then select one of the neighbors of the selected neigbor, and proceed like this until we come to a node where where there either are no neigbors to select, or we have already visited all neigbors, we get what is known as depth first search, abbreviated as DFS.

While proceeding, we keep track of nodes where we have selected neighbors, and when we are done with a node, we return to the node from which the node we are done with was selectd. We kan keep track of this using a stack.

If we implement the DFS as a recursive function, the call stack for this function will serve as such a stack.

Pseudocode for a DFS algorithm is shown in Algorithm 3.1 .

Algorithm 3.1 DFS

Require: a graph and a vector *visited*, with all elements initialized to false

```
1: procedure DFS(v, visited)
2:     visited[v] = true
3:     for v_nb in neighbors[v] do
4:         if not visited[v_nb] then
5:             DFS(v_nb, visited)
6:         end if
7:     end for
8: end procedure
```

As can be seen on line 3 in Algorithm 3.1 , we have a really cool notation for our pseudocode.

Here is an implementation of DFS in C++.

Chapter 4

Summary

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

Chapter 5

Extra

This content will only appear in the advanced version.

```cpp
#include <vector>

template <typename Graph, bool verbose = false, bool save_path = false>
class DFS
{
    Graph g;
    std::vector<int> path;

    void dfs(int v, std::vector<bool> &visited)
    {
        if constexpr (verbose)
        {
            std::cout << "visiting v: " << v << std::endl;
        }
        visited[v] = true;
        if constexpr (save_path)
        {
            path.push_back(v);
        }
        for (auto v_nb : g.get_nb_vec(v))
        {
            if (!visited[v_nb])
            {
                dfs(v_nb, visited);
            }
        }
    }
```

```
public:
    explicit DFS(const Graph &g) : g(g)
    {
    }

    void traverse(int v_start)
    {
        std::vector<bool> visited(g.get_n_nodes(), false);
        dfs(v_start, visited);
    }

    const std::vector<int> &get_path() const
    {
        return path;
    }

    void print_path() const
    {
        for (auto v : path)
        {
            std::cout << v << " ";
        }
        std::cout << std::endl;
    }
};
```

ok, let's try something else:

```
#include <vector>

template <typename Graph, bool verbose = false, bool save_path = false>
class DFS
{
    Graph g;
    std::vector<int> path;

    void dfs(int v, std::vector<bool> &visited)
    {
        if constexpr (verbose)
        {
            std::cout << "visiting v: " << v << std::endl;
        }
        visited[v] = true;
        if constexpr (save_path)
        {
```

```
            path.push_back(v);
        }
        for (auto v_nb : g.get_nb_vec(v))
        {
            if (!visited[v_nb])
            {
                dfs(v_nb, visited);
            }
        }
    }

public:
    explicit DFS(const Graph &g) : g(g)
    {
    }

    void traverse(int v_start)
    {
        std::vector<bool> visited(g.get_n_nodes(), false);
        dfs(v_start, visited);
    }

    const std::vector<int> &get_path() const
    {
        return path;
    }

    void print_path() const
    {
        for (auto v : path)
        {
            std::cout << v << " ";
        }
        std::cout << std::endl;
    }
};
```

{#lst-code .r lst-cap="Some R code"}

Let's make some pseudo code, as

Cool, right?

Algorithm 5.1 Quicksort

```
1: procedure QUICKSORT(A, p, r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
6:     end if
7: end procedure
8: procedure PARTITION(A, p, r)
9:     x = A[r]
10:    i = p − 1
11:    for j = p to r − 1 do
12:        if A[j] < x then
13:            i = i + 1
14:            exchange A[i] with A[j]
15:        end if
16:        exchange A[i] with A[r]
17:    end for
18: end procedure
```

References

Halim, Steven, and Felix Halim. 2018. *Competitive Programming 4 - Book 1.* https://cpbook.net.

———. 2020. *Competitive Programming 4 - Book 2.* https://cpbook.net.

Laaksonen, Antti. 2020. *Guide to Competitive Programming - Learning and Improving Algorithms Through Contests.* 2nd ed. Springer.

Sannemo, Johan. 2020. *Algorithmic Problem Solving.* https://jsannemo.se/aps.pdf.