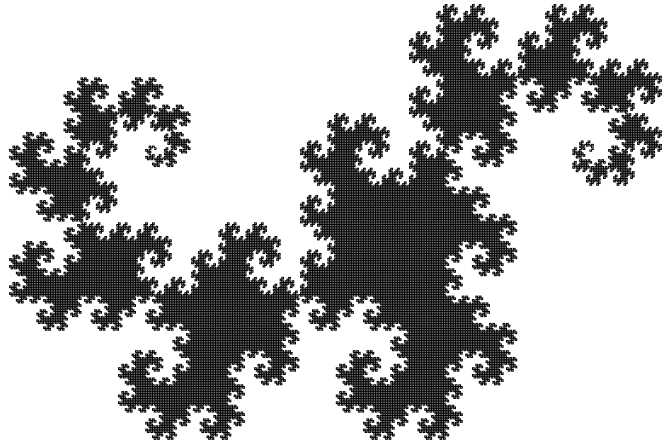

CONTENTS

Preface

The goal of this book is to serve as a preparation guide for coding competitions. I should mention that this is a *personalized* reference book for myself and my team. As a consequence, more attention is given over to writing for Java and C++.

Some competition environments allow texts with a certain criterea. ICPC World Finals rules that a team may bring a **Team Reference Document**, a document containing up to 25 pages of reference materials, single-sided, letter or A4 size. This book does not follow the requisite, but we advise you to look at our appendix section.

We have included an two appendixes. Appendix 1 is devoted to common data structures we encountered using Java and C++. Appendix 2 includes common mathematical formulas for reference.

Chapter 1

Behind the Big O

1.1 Reductions

FILL

Chapter 2

Trees

Chapter 3

Graphs

Graphs are a fundamental data structure in the world of programming. Formally a **graph** G consists of a finite nonempty set V of objects called **vertices** and a set E called **edges**. G is an ordered pair of sets V and E.

$$G = (V, E)$$

Graphs are often used to represent physical entities (a network of roads, relationship between people). Common graph problems include shortest paths, number of minimum cuts, strongly connected components ....

3.1 Breadth-First Search

Consider you're playing an all-time classic, LoZ: Ocarina of Time. Being the completionist that you are, you want to open every single chest in all of the dungeons. How would you traverse the dungeon?

Often labeled as a 'cautionary search', the plan of **Breadth-First Search** is to uniformally explore the nodes of a graph outwards. In BFS, we pick a starting node n_1 and visit all of n_1's neighbors before visiting their neighbors. Using this algorithm allows us to visit all rooms of a dungeon without roaming too deep.

The main perk of BFS over other search algorithms is it can compute shortest paths in unweighed graphs.

3.2 Depth-First Search

If BFS is the cautious and tentative exploration strategy, then **Depth-First Search** (DFS) is its more aggressive cousin. DFS explores aggressively, delving deeper into the graph and backtracks only when necessary.

The implementation of DFS is similar to BFS, but instead of using a queue, we use a stack. Another method is to use recursion.

```
1       void search(Node root){
2         if(root == null) return;
3         visit(root);
4         root.visited = true;
5         for each (Node n in root.adjacent) {
6           if (n.visited == false) {
7             search(n);
8           }
9         }
10      }
```

Running time: $O(V + E)$

Why use DFS over BFS? DFS has its own catalog of applications that can't be replicated using BFS.

- Computing a topological ordering of directed acyclic graphs[1]
- Computing strongly connected components of directed graphs

Topological Sort

Khoa is really interested in artificial intelligence, so much so that before he graduates, he wants to take the course offered by the computer science department, CS 4242 (X). However, before he can enroll in CS 4242, he must have completed its prerequisites.

The prerequisites for CS 4242 are CS 3304 (W) and MATH 3332 (V). CS 3304 and MATH 3332 also have a prerequisite, MATH 1190 (S).

The sequence of which Khoa can take his classes can be arranged via topological sort. There are two sequences which Khoa can take.

A **topological ordering** of a directed graph G is a labeling f of G's nodes such that:

[1] Note: G has a directed cycle $\implies$ No topological ordering

1. the $f(v)$'s are the set $1, 2, \ldots, n$
2. $(u, v) \in G \Rightarrow f(u) < f(v)$

As long as our structure is a directed acyclic graph (DAG), we can compute a topological ordering. Every DAG has a sink vertex. Remove the sink vertex, and the remaining graph is a DAG. (Unless $|v| < 1$) The pseudocode for a topological sort is as follows:

```
1       DFS(graph G, start vertex s) {
2         visit(s);
3         for each edge (s,v) {
4           if(v.visited == false)
5             DFS(G, v);
6         }
7         set f(s) = current_label
8         current_label--;
9       }
10
11      topologic_sort(graph G) {
12        mark all nodes unexplored;
13        current_label = n;
14        for each vertex $v \in G$ {
15          if(v.visited == false)
16            DFS(G, v);
17        }
18      }
```

Running time: $O(V + E)$
Memory space: $O(V + E)$

Proof. Take any edge, (u, v). We will show that $f(u) < f(v)$.

Case 1: u is visited by DFS before v
u will recursively call v, which will continue DFS. Eventually v will exhaust its adjacent neighbors and become a sink vertex before u. $f(u) < f(v)$

Case 2: v is visited by DFS before u
v makes recursive DFS calls. u will never be visited during these recursively calls, otherwise the graph would contain a directed cycle. As a result, v will become a sink vertex before u is visited. $f(u) < f(v)$ □

Strongly Connected Components

The **strongly connected components** of a directed graph G are defined as the equivalence classes of the relation $u \sim v \Leftrightarrow \exists$ path(u, v) and path(v, u). SCCs can be computed using **Kosaraju's Two Pass algorithm** .

Kosaraju's Two Pass Algorithm

Theorem 3.2.1. *Can compute SCC in* $O(m + n)$

Chapter 4

Weighted Graphs

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

[2] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891921, 1905.

[3] Knuth: Computers and Typesetting, `http://www-cs-faculty.stanford.edu/~uno/abcde.html`

Appendix I: Data Structures

Appendix II: Mathematical Formulae

Index