Summary of Algorithm Design (2006)
Jon Kleinberg, Eva Tardos

** Obviously, I don't take any credit for the material in this file. This is just for my own reference.*

Chapter 2: Basics of Algorithm Analysis

Suppose f and T are positive functions.

$$O(f(n)) = \{T(n) | \exists c > 0 \text{ and } n_0 > 0;\ \forall n \geq n_0\ T(n) \leq c{:}f(n)\}$$
$$\Omega(f(n)) = \{T(n) | \exists c > 0 \text{ and } n_0 > 0;\ \forall n \geq n_0\ T(n) \geq c{:}f(n)\}$$
$$o(f(n)) = \{T(n) | \forall c > 0\ \exists n_0 > 0;\ \forall n \geq n_0\ T(n) \leq c{:}f(n)\}$$
$$!(f(n)) = \{T(n) | \forall c > 0\ \exists n_0 > 0;\ \forall n \geq n_0\ T(n) \geq c{:}f(n)\}$$
$$(f(n)) = \{T(n) | T(n) \in O(f(n)) \text{ and } T(n) \in \Omega(f(n))\}$$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$

$$\begin{array}{rcl} c = 0 & \Rightarrow & f \in o(g) \\ c = \infty & \Rightarrow & f \in w(g) \\ 0 < c < \infty & \Rightarrow & f \in (g) \end{array}$$

$$\log_a n = \frac{\log_b n}{\log_b a} \qquad \ln(n!) \approx n \ln n - n + O(\ln n)$$

Examples:
$O(log(n))$: Binary Search
$O(n)$: Merging two sorted list
$O(n\log n)$: Sorting a list
$O(n^2)$: $\binom{n}{2}$
$O(2^n)$: NP-Complete, it is the number of subsets of an n-element set.
$O(n!)$: It is the number of ways to match up n items with n other items.

Chapter 3: Graphs

A **binary tree** is a tree in which each node has at most two children.
A **binary search tree** is a binary tree in which for each node, its left descendants are less than or equal to the current node, which is less than the right descendants.
A **balanced tree** is "balanced" enough to ensure $O(log(n))$ times for `insert` and `find`, but it's not necessarily as balanced as it could be.
A **complete binary tree** is a binary tree which is totally filled, other than possibly the rightmost elements on the last level.
A **full binary tree** is a binary tree in which every node has either zero or two children.
A **perfect binary tree** is one that is both full and complete.
A **min-heap** is a complete binary tree where each node is smaller than its children (so the root is the minimum of the tree).

BFS and DFS algorithms run in time $O(m+n)$ if the graph is given by the adjacency list representation.

A **topological ordering** of G is an ordering of its nodes as $v_1; \ldots; v_n$ so that for every edge $(v_i; v_j)$, we have $i < j$. In other words, all edges point "forward" in the ordering.

Theorem 1 *G has a topological ordering iff it is a DAG.*

3.1 Tree Decomposition

A **tree decomposition** (TD) of $G = (V; E)$ consists of

- A tree T (different node set)
- A set $B_t \subseteq V$ for each $t \in T$ (called "*bag*") that satisfy the following 3 conditions:
  1. Node Coverage: Every node of G belongs to at least one bag B_t.
  2. Edge Coverage: $\forall e \in E; \exists B_t$ containing both ends of e.
  3. Coherence: Let $t_1; t_2; t_3$ be 3 nodes of T s.t. t_2 lies on the path from t_1 to t_3. Then if a node v of G belongs to both B_{t_1} and B_{t_3}, it also belongs to B_{t_2}.

Let T be a TD of G. *Width* of T is defined as $(\max_{t \in T} |B_t|) - 1$. *Tree-width* of G is defined as minimum width of any TD of G.
A TD is *nonredundant* iff there is no edge $(x; y)$ in the tree s.t. $B_x \subseteq B_y$. We can always turn a redundant TD to a nonredundant one without changing the tree-width.

Theorem 2 *Any nonredundant TD of an n-node graph has at most n bags.*

Theorem 3 *Every graph with tree-width k has a node of degree at most k.*

Theorem 4 *An n-node graph has tree-width n-1 iff it is a clique*[1] .

Chapter 4: Greedy Algorithms

Greedy algorithms make the locally optimal choice at each stage with the hope of finding a global optimum.

Using the heap-based priority queue Dijkstra's Algorithm for finding the shortest path in a weighted graph runs in $O(m \log n)$.

Using the heap-based priority queue Prim's Algorithm for finding the minimum spanning tree runs in $O(m \log n)$.

Chapter 5: Divide and Conquer

Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. [CLRS]
It is usually used to reduce a polynomial running time down to a faster running time.

Master Theorem:

$$T(n) \leq \begin{cases} aT(n/b) + cn^x & n > 1 \\ d & n = 1 \end{cases}$$

$$a > b^x \quad \Rightarrow \quad T(n) \in (n^{\log_b a})$$

$$a = b^x \quad \Rightarrow \quad T(n) \in (n^x \log_b n)$$

$$a < b^x \quad \Rightarrow \quad T(n) \in (n^x)$$

[1] A set of nodes is a *clique* if every two nodes are adjacent.

In the first case the total running time is dominated by the work done on constant-size subproblems at the bottom of the recursion. In the the third case it's dominated by the top level. The second case represents a "knife-edge"; the amount of work done at each level is exactly the same.

Mergesort:

$$\begin{aligned} T(n) &= T(divide) + T(sub_1) + T(sub_2) + T(combine) \\ &= \underbrace{2T(n/2)}_{recursion} + \underbrace{cn}_{divide+combine} \end{aligned}$$

Chapter 6: Dynamic Programming

In contrast to divide-and-conquer (which applies on disjoint subprblems), dynamic programming applies when the subproblems overlap (i.e. when subproblems share subsubproblems). [CLRS]
Memoization: A DP algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.[CLRS]

Dijkstra's Algorithm (Greedy, $O(m\log n)$): Finding shortest path from one node to all nodes.
Bellman-Ford's Algorithm (DP, (mn)): Finding shortest path from one node to all nodes, where negative edges are allowed.
Floyd-Warshall's Algorithm (DP, (n^3)): Finding shortest path between all pairs of nodes, where negative edges are allowed.

Bellman-Ford's and Floyd-Warshall's algorithms can be used to detect negative cycles in graphs.

Chapter 7: Network Flow

A **flow network** is a directed graph $G = (V, E)$ with the following features. Associated with each edge e is a nonnegative capacity c_e. There is a single source node s and a single sink node t.

Assumptions: 1) No edge enters the source s and no edge leaves the sink t. 2) There is at least one edge incident to each node. 3) All capacities are integers.

An **s-t flow** is a function f that maps each edge e to a nonnegative real number, $f : E \to \mathbb{R}^+$; the value $f(e)$ intuitively represents the amount of flow carried by edge e. A flow f must satisfy the following two properties:
(i) (*Capacity conditions*) For each $e \in E$, we have $0 \leq f(e) \leq c_e$.
(ii) (*Conservation conditions*) For each node v other than s and t, we have

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e) \qquad (f^{out}(v) = f^{in}(v))$$

The *value* of a flow f , denoted $v(f)$, is defined to be the amount of flow generated at the source: $v(f) = f^{out}(s)$.

Ford-Fulkerson Max-Flow Algorithm

Theorem 5 *If all capacities in the flow network are integers then the Ford-Fulkerson Algorithm can be implemented to run in $O(mC)$ time ($C = \sum_{e \text{ out of } s} c_e$).*

An **s-t cut** is a partition $(A; B)$ of the vertex set V, so that $s \in A$ and $t \in B$. The capacity of a cut $(A; B)$, denoted $c(A; B)$, is simply the sum of the capacities of all edges out of A.

Theorem 6 *If f is an s-t flow such that there is no s-t (augmenting) path in the residual graph, then there is an s-t cut $(A^*; B^*)$ for which $v(f) = c(A^*; B^*)$. Consequently, f has the maximum value of any flow, and $(A; B)$ has the minimum capacity of any $s - t$ cut.*

Given a flow f of maximum value, we can compute an s-t cut of minimum capacity in $O(m)$ time as follows: We construct the residual graph G_f , and perform breadth-first or depth-first search to determine the set A^* of all nodes that s can reach. We then define $B^* = V A^*$.

Scaling Max-Flow Algorithm

Augmentation increases the value of the maximum flow by the bottleneck capacity of the selected path; so if we choose paths with large bottleneck capacity, we will be making a lot of progress. A natural idea is to select the path that has the largest bottleneck capacity. Having to find such paths can slow down each individual iteration by quite a bit. We avoid this slowdown by not worrying about selecting the path that has exactly the largest bottleneck capacity. Instead, we will maintain a scaling parameter Δ, and we will look for paths that have bottleneck capacity of at least Δ.

Theorem 7 *The **Scaling Max-Flow** Algorithm in a graph with m edges and integer capacities finds a maximum flow in at most $2m(1 + \lceil \log_2 C \rceil)$ augmentations. It can be implemented to run in at most $O(m^2 \log_2 C)$ time.*

Pre ow-Push Max-Flow Algorithm

A **pre ow** is a quasi-flow where in place of the conservation conditions, we require only inequalities: Each node other than s must have at least as much flow entering as leaving. The Preflow-Push Algorithm will maintain a preflow and work on converting the preflow into a flow.

Theorem 8 *The running time of the Preflow-Push Algorithm, implemented using a special data structures, is $O(mn)$ plus $O(1)$ for each nonsaturating push operation. In particular, the generic Preflow-Push Algorithm runs in $O(n^2 m)$ time, while the version where we always select the node at maximum height runs in $O(n^3)$ time.*

The Ford-Fulkerson Algorithm can be used to find a maximum matching in a bipartite graph in $O(mn)$ time.

Chapter 8: NP-Completeness

To prove problem X is NP-Complete first show that it is in NP. Then reduce an NP-complete problem Y to X ($Y \leq_P X$) i.e.: Start with an instance of Y. Transform it to an instance of X in polynomial time. Discuss the connection between the two instances.

Chapter 11: Approximation Algorithms

Standard form for **Linear Programming (LP)**, as an optimization problem:

> Given an $m \times n$ matrix A, and vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, find a vector $x \in \mathbb{R}^n$ to solve the following optimization problem:
>
> $$\begin{aligned} \min \quad & c^t x \\ \text{s.t.} \quad & x \geq 0; \\ & Ax \geq b \end{aligned}$$

The decision version of LP is in $NP \cap co\text{-}NP$. The most widely used algorithm for LP is the *simplex method*. It works very well in practice yet its worst-case running time is known to be exponential; it is simply that this exponential behavior shows up in practice only very rarely. But in general LP problems can be solved in polynomial time, and very efficient algorithms exist in practice.

Decision version of **Subset Sum** (a special case of the **Knapsack Problem**) is as follows:

> Given natural numbers $w_1, \ldots, w_n$, and a target number W, is there a subset of $\{w_1, \ldots, w_n\}$ that adds up to precisely W?

We have already seen an algorithm to solve this problem; why are we now including it on our list of computationally hard problems? Using dynamic programming we can solve this problem in time $O(nW)$, which is reasonable when W is small, but becomes hopelessly impractical as W (and the numbers w_i) grow large. Consider, for example, an instance with 100 numbers, each of which is 100 bits long. Then the input is only $100 \times 100 = 10{,}000$ digits, but W is now roughly 2^{100} (i.e. W is also roughly 100 bits long but its value can be as high as 2^{100}).

Vahid Vaezian (vvaezian [at] sfu.ca), February 3, 2018