Summary

Advanced Algorithms and Datastructures 2020-2021

(`WBCS009-05`)

Bjorn Pijnacker

Contents

About this summary

This summary is based on the book *Introduction to Algorithms* edition 3, written by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Any references made to 'the book' in the text reference this book. All the summary contents are derived from the material for the Advanced Algorithms and Datastructures course (`WBCS009-05`) given in 2b 2020-2021.

Chapter 1

Introduction

An algorithm is any well-defined computational procedure that takes some value as input and produces some value as output. An algorithm thus transforms the input into the output.

An algorithm can be formally defined in many ways. If we have a formal definition we can obtain an instance of the problem, that is, any input (that satisfies the posed constraints) needed to compute a solution to the problem. An algorithm is said to be correct when, for every problem instance, it halts with the correct solution.

Along with algorithms we often find data structures. Data structures are a way to store and organize data to facilitate access and modifications in a way that some algorithm or problem might require.

1.1 Insertion sort

Insertion sort solves the sorting problem where we have a sequence of numbers which we want to permute to have some order provided by the keys. Insertion sort does this by removing elements from one list and placing them in a sorted position in a second list, hence the name insertion sort.

Insertion sort can be implemented in-place, that is, it does not need extra memory besides the input to sort the array. The pseudocode for insertion sort can been seen in Algorithm 1.

Algorithm 1: Insertion sort

```
1 Function Insertion-Sort(A)
      Input: An array $A$ of length $n$ comparable elements
      Output: The same array with a sorted order
2     for $j = 2$ to $A.length$ do
3         $key = A[j]$
          /* Insert $A[j]$ into the sorted sequence $A[i \,..\, j-1]$ */
4         $i = j - 1$
5         while $i > 0$ and $A[i] > key$ do
6             $A[i+1] = A[i]$
7             $i = i - 1$
8         $A[i+1] = key$
```

1.2 Analyzing algorithms

If we have an algorithm we often wish to properly analyze its running time and space complexities. With this we mean the relationship between input size and the scaling of these properties. As such, we express these in terms of the input size. We will go more into algorithm analysis in the next chapter.

Chapter 2

Divide & Conquer

There is a wide range of algorithm design techniques. For Algorithm 1, we used an incremental approach. We can also use an alternative approach, named "divide-and-conquer". The runtime complexity of these types of algorithms is often much easier to find, as we will see later in this chapter.

2.1 Merge sort

To show a sorting algorithm that uses the divide and conquer approach, we consider merge sort. This algorithm first divides the n-element sequence to be sorted into two $n/2$-size subsequences, it then sorts these recursively and merges them to produce the sorted answer. The pseudocode shown in Algorithm 2 implements this idea.

When we have such an divide and conquer algorithm we want to be able to analyze it. We can often describe the running time of such a recursive algorithm by a recurrence equation. We can then use mathematical tools to solve the recurrence and provide bounds for the algorithm performance.

For merge sort we see the following recurrence equation $T(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

We see that if the size of the array $n = 1$ then the running time is constant. If it is not, then we see the running time is equal to twice the running time of half the problem, plus an operation of order $\Theta(n)$. The first term here is the recurrence, and the second is the merging operation. We will see later that this evaluates to a running time of $\Theta(n \lg n)$[1].

2.2 Growth of functions

The notations we use to describe asymptotic running time of an algorithm are defined in terms of functions whose domains are the set $\mathbb{N}$. We use the asymptotic notation of $\Theta(\cdot)$ to give the tight bound for a function. A tight bound means that a function is both upper bounded and lower bounded by the bound, within a constant multiplier. We also have $O(\cdot)$ which talks about an asymptotic upper bound and $\Omega(\cdot)$ which is the asymptotic lower bound.

[1]Here $\lg(\cdot)$ is a shorthand for $\log_2(\cdot)$.

Since these functions talk about asymptotic bounds, we forego constants and lower-value terms of polynomials in the bounds. For example: if the runtime complexity of some function is $T(n) = 3n^2 + 2n$ we say it is of order $\Theta(n^2)$, since these functions grow in the same manner. This also makes it easier to compare the orders of different functions, since irrelevant terms are removed.

Algorithm 2: Merge sort

```
Function MERGE($A, p, q, r$)
    /* Merges sorted subarrays A[p..q] and A[q+1..r] to be a single sorted
       array                                                            */
    Input: An array $A$ with sorted subsequences $A[p\,..\,q]$ and $Aq + 1\,..\,r$
    Output: The same array with the sorted subarrays merged into a single sorted
            subarray
    $n_1 = q - p + 1$
    $n_2 = r - q$
    Define $L[1\,..\,n_1 + 1]$ and $R[1\,..\,n_2 + 1]$ as new arrays
    for $i = 1$ to $n_1$ do
        $L[i] = A[p + i - 1]$
    for $j = 1$ to $n_2$ do
        $R[j] = A[q + j]$
    $L[n_1 + 1] = \infty$
    $R[n_2 + 1] = \infty$
    $i = 1$
    $j = 1$
    for $k = p$ to $r$ do
        if $L[i] \leq R[j]$ then
            $A[k] = L[i]$
            $i = i + 1$
        else
            $A[k] = R[j]$
            $j = j + 1$
Function MERGE-SORT($A, p, r$)
    /* Recursively sorts the array A[p..q]                              */
    Input: An array $A$ of length $q - p$ comparable elements
    Output: Array $A$ in sorted manner
    if $p < r$ then
        $q = \lfloor (p + r)/2 \rfloor$
        MERGE-SORT($A, p, q$)
        MERGE-SORT($A, q + 1, r$)
        MERGE($A, p, q, r$)
```

2.3 Divide-and-conquer

In the divide-and-conquer technique we solve a problem in three distinct steps:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

We will discuss three methods of solving these type of recurrences, them being the substitution method, the recursion-tree method, and the master theorem method.

2.3.1 Solving a divide-and-conquer problem

The substitution method

The substitution method comprises two steps: (1) guess the form of the solution; (2) use mathematical induction to find the constants and show that the solution works. The most difficult part of this is guessing a good solution. There aren't many ways to do this except from experience and creativity.

The recursion-tree method

This method is a way of coming up with a good guess for the substitution method. In the tree, each nodes represents the cost of a single subproblem. We finally get to the leaves that all have a cost not dependent on subproblems. We can sum up the costs in the recursion tree sloppily to come to a good guess. If we do it very carefully however, it can be used as a direct proof. See the book for a demonstration of how to create and sum a recursion tree.

The master method

The master method provides a method for solving recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$ and $b > 1$ and $f(n)$ is an asymptotically positive function. The master theorem is the following:

Theorem 1 *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. *If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.*

2. *If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{log_b a} \lg n)$.*

3. *If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

For examples and an explanation of this theorem, see the book.

Chapter 3

Heaps

3.1 Heapsort

Heapsort is yet another sorting algorithm. Heapsort has the same time complexity as merge-sort: $O(n \lg n)$ but sorts in-place, like insertion sort. Heapsort uses a datastructure by the name of a "heap" to manage its data, which allows it to be as efficient as it is.

A heap data structure is an array object that can be interpreted as a nearly complete binary tree. Each node of the tree corresponds to one element in the array. By definition, we say the children of some element with index i are the elements with index $2i$ and $2i + 1$. As such, the parent of any node with index j is the element with index $\lfloor j/2 \rfloor$.

There are two kinds of binary heaps: max-heaps and min-heaps. The values in the nodes satisfy some heap property, this being the max-heap property and min-heap property. The max-heap property is that for every node i other than the root it holds that $A[\lfloor i/2 \rfloor] \geq A[i]$: a parent is always larger or equal than its children.

For heap sort we use the max-heap. For a priority queue, a min-heap can be used. We now present some procedures to implement the max-heap. This requires multiple functions:

Max-Heapify makes sure the heap property is maintained. Of the node i, it compares it with its children and finds which one is the biggest. If this is i, then we are done. If it is not then we swap i with the bigger child and continue calling MAX-HEAPIFY on the switched child. This function has running time complexity $O(\lg n)$.

Build-Max-Heap simply calls MAX-HEAPIFY on all non-leaf nodes, starting at the end of the array. This makes it so all nodes are considered and the heap property exists in the end. This function has running time complexity $O(n \lg n)$.

Heap-Sort sorts the array by continuously grabbing the root–thus largest value–node, and exchanging it with the last node. It then reduces the heap size by one, making the maximum value node not part of the heap anymore. It calls MAX-HEAPIFY on the switched node to restore the max-heap property. Doing this for all the nodes causes the array not part of the heap anymore to be sorted. At the end our heap will have size 1, containing only the smallest element still. This function has running time complexity $O(n \lg n)$.

The implementations of these functions is shown in Algorithm 3.

Algorithm 3: Heap sort

```
1  Function MAX-HEAPIFY(A, i)
       Input: A representing the heap array and i representing the index of the node that
              violates the max-heap property
       Output: The heap with proper max-heap property adherence
2      $l = 2i$
3      $r = 2i + 1$
4      if $l \leq A.heap\text{-}size$ and $A[l] > A[i]$ then
5          $largest = l$
6      else
7          $largest = i$
8      if $r \leq A.heap\text{-}size$ and $A[r] > A[largest]$ then
9          $largest = r$
10     if $largest \neq i$ then
11         exchange $A[i]$ with $A[largest]$
12         MAX-HEAPIFY($A$, $largest$)
13 Function BUILD-MAX-HEAP(A)
       Input: An array A
       Output: The same array reordered to satisfy the max-heap property
14     $A.heap\text{-}size = A.length$
15     for $i = \lfloor A.length/2 \rfloor$ downto 1 do
16         MAX-HEAPIFY($A$, $i$)
17 Function HEAP-SORT(A)
18     BUILD-MAX-HEAP($A$)
19     for $i = \lfloor A.length/2 \rfloor$ downto 2 do
20         exchange $A[1]$ with $A[i]$
21         $A.heap\text{-}size = A.heap\text{-}size - 1$
22         MAX-HEAPIFY($A$, 1)
```

We can also use a heap for a priority queue. One way would be to add functions that allow us to do so in the max-heap, but easier is to use a min-heap, then use the following method outlined in Algorithm 4 to obtain the first in line.

Algorithm 4: Heaps for priority queue

```
1 Function DEQUEUE($A$)
      Input: $A$ representing the heap array for a min-heap
      Output: The smallest element in the min-heap and this element removed from $A$
2     elem = $A[1]$
3     exchange $A[1]$ with $A[A.heap\text{-}size]$
4     $A.heap\text{-}size = A.heap\text{-}size - 1$
5     MAX-HEAPIFY($A$, $1$)
6     return elem
```

3.2 Linear-time Sorting

We've seen that there is multiple methods to sort in $O(n \lg n)$ time complexity, but in some cases we can sort in average linear time by foregoing the comparison between multiple elements. Any comparison-based sorting algorithm can only sort with a lower bound of $\Omega(n \lg n)$ for n elements, as is proven in the book.

3.2.1 Counting Sort

Counting sort provides a speedup on conventional sorting algorithms by having some extra information and constraints. In this case, we sort a dataset with only integers in the range $[0, k)$, where k is an upper bound known a priori. Counting sort determines for each element x the number of elements smaller than x. If x has 17 smaller elements than it, we know that x belongs in position 18 of the output. This is shown in Algorithm 5

Algorithm 5: Counting sort

```
1 Function COUNTING-SORT($A$, $B$, $k$)
      Input: $A$ representing the array to be sorted, $B$ representing the array for the output
             to be placed in, $k$ representing the upper bound for the integers in $A$
      Output: The array $B$ with all elements from $A$ in a sorted order
2     let $C[0..k]$ be a new array containing all zeros
3     for $j = 1$ to $A.length$ do
4         $C[A[j]] = C[A[j]] + 1$
      /* C[i] now contains the number of elements equal to i */
5     for $i = 1$ to $k$ do
6         $C[i] = C[i] + C[i-1]$
      /* C[i] now contains the number of elements less than or equal to i */
7     for $j = A.length$ downto 1 do
8         $B[C[A[j]]] = A[j]$
9         $C[A[j]] = C[A[j]] - 1$
```

3.2.2 Radix Sort

Radix sort sorts integers by the least significant digit first. It then sorts the first to least significant digit and so on, until all numbers have been sorted. Since there is a known upper bound for the values a digit can take (such as $[0,9]$ or $[a,z]$) we can sort this in linear time. Since the number of digits is a constant, this doesn't factor in to the calculation. This means that radix sort runs in $O(n)$.

The sorting algorithm used inside of radix sort must be stable, meaning that elements with the same key in the input array appear in the same order in the output array. In this case that key is whatever digit radix sort is currently sorting on.

3.2.3 Bucket sort

Bucket sort assumes that the input is drawn from a uniform distribution and runs on average in $O(n)$ with a worst case of $O(n^2)$. Bucket sort also requires some prior information about the data to sort, namely that all the values are in $[1,0)$. It sorts them by dividing the interval into n equally sized buckets and placing elements in these buckets one-by-one. On average, we expect just a few elements per bucket. We simply go over all the buckets and list the elements in order. If there are multiple elements in a bucket we first sort these with insertion sort, which is very quick for a small number of elements. The pseudocode for bucket sort is presented in Algorithm 6.

Algorithm 6: Bucket sort

```
Function BUCKET-SORT($A$)
    Input: Array $A$ to be sorted with values in $[0,1)$
    Output: Array $A$ sorted
    $n = A.length$
    let $B[0 \mathinner{..} n-1]$ be a new array of empty lists
    for $i = 1$ to $n$ do
        insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
    for $i = 0$ to $n-1$ do
        sort list $B[i]$ with insertion sort
    return the concatenation of all lists $B[0], B[1], \ldots, B[n-1]$ in that order
```

Chapter 4

Dynamic Programming

Dynamic programming is a programming paradigm similar to divide and conquer as it solves problems by combining the solutions of subproblems. Dynamic programming and divide and conquer differ when there are overlapping subproblems. A dynamic programming algorithm saves the subproblem solutions and reuses them later, avoiding doing the same thing multiple times. Dynamic programming is typically applied to optimization problems.

When developing a dynamic programming algorithm we follow a sequence of four steps: (1) Characterize the structure of an optimal solution; (2) recursively define the value of an optimal solution; (3) compute the value of an optimal solution, typically in a bottom-up fashion; (4) construct an optimal solution from computed information.

The first step shown is the characterization of the structure of an optimal solution. A common pattern we follow in discovering optimal substructure is the following: (1) We show that a solution to the problem consists of making a choice which leaves a subproblem to be solved; (2) We suppose that for a given problem we are given the choice that leads to an optimal solution. We do not care how this choice is made; (3) Given this choice we determine which subproblems ensue and how to best characterize the resulting space of subproblems; (4) We show that the solutions to the subproblems must be optimal.

For more examples and explanations regarding dynamic programming, please see the book chapter 15.1, 15.3, and 15.4. All of this is quite abstract hard to summarize, so I don't want to explain it here. Scanning over the sections in question should be good enough for a general understanding.

Chapter 5

Median, Order Statistics, and String Matching

5.1 Medians and Order Statistics

The ith order statistic is defined as the ith smallest element of a sequence of orderable items. The minimum is the first order statistic ($i = 1$) and the maximum is the nth order statistic ($i = n$). The upper median is the $i = \lfloor (n+1)/2 \rfloor$th order statistic and the lower median occurs at $i = \lceil (n+1)/2 \rceil$. When we refer to 'the median' we mean the lower median.

We can solve the problem of selecting any ith order statistic easily in $O(n \lg n)$ time, by simply sorting the list of elements and indexing at i. We can do better though, as we will show in this section.

To find the minimum and maximum we need at least $n - 1$ comparisons. We can find both at once using at most $3\lfloor n/2 \rfloor$ comparisons, however, which is better than $2(n-1)$, by processing elements in pairs. We compare the elements from the input with each other. Then we compare the smaller with the current minimum and the larger with the current maximum, giving three comparisons for every two elements.

5.1.1 Selection in expected linear time

Using the algorithm Randomized-Select we can solve the selection problem in expected linear time. This algorithm is modeled after the quicksort algorithm. We partition the input array recursively and then, as opposed to quicksort, work on just a single side of the partition. The algorithm is presented in Algorithm 7.

5.1.2 Selection in worst-case linear time

We can do better than Algorithm 7 however by following these steps: (1) Divide the n elements into $\lfloor n/5 \rfloor$ groups of 5 elements each and one group made up of the remaining $n \bmod 5$ elements; (2) Find the median of each of the groups by insertion-sorting the elements of each groups and picking the median; (3) Use SELECT to find the median of the previously found medians; (4) Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition such that x is the kth smallest element and there are $n - k$ elements on the high side of the

Algorithm 7: Randomized selection

```
1  Function RANDOMIZED-SELECT($A, p, r, i$)
       Input: An array $A$ of which we want to find the $i$th order statistic and $p$ and $r$
              indicating the subarray in which to search
       Output: The $i$th order statistic of $A$
2      if $p == r$ then
3          return $A[p]$
4      $q =$ RANDOMIZED-PARTITION($A, p, q-1, i$)
5      $k = q - p + 1$
6      if $i == k$ then
           /* The pivot value is the answer                                   */
7          return $A[q]$
8      else if $i < k$ then
9          return RANDOMIZED-SELECT($A, p, q-1, i$)
10     else
11         return RANDOMIZED-SELECT($A, q+1, r, i-k$)
12 Function RANDOMIZED-PARTITION($A, p, r$)
13     $i =$ RANDOM($p, r$)
14     exchange $A[i]$ and $A[r]$
15     return PARTITION($A, p, r$)
16 Function PARTITION($A, p, r$)
17     $x = A[r]$
18     $i = p - 1$
19     for $j = p$ to $r - 1$ do
20         if $A[j] \leq x$ then
21             $i = i + 1$
22             exchange $A[i]$ and $A[j]$
23     exchange $A[i+1]$ and $A[r]$
24     return $i + 1$
```

partition; (5) If $i = k$ then return. Otherwise use SELECT recursively to find the ith smallest element on the low side if $i < k$ or the $i - k$th on the high side if $i > k$.

5.2 String Matching

The problem of string matching is as follows: Assume that the text is an array $T[1\,..\,n]$ of length n and that the pattern is an array $P[1\,..\,m]$ of length $m \leq n$. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. We say that a pattern P occurs with shift s in text T if $0 \leq s \leq n - m$ and $T[s+1\,..\,m] = P[1\,..\,m]$. If P occurs with shift s in T then we call s a valid shift, otherwise we call it an invalid shift. The string-matching problem is that of finding all valid shifts with which a given pattern P occurs in text T.

Some important terminology that is used for strings is the following: We denote by Σ^* the set of all finite-length strings that can be formed from an alphabet Σ. The zero-length string ε also belongs to Σ^*. The length of a string is denoted by $|x|$. The concatenation of two strings x and y is written as xy and has length $|x| + |y|$. If we have some string P we write P_k to mean the k-character prefix $P[1\,..\,k]$.

We say that a string w is a prefix of x, denoted $w \sqsubset x$ if $x = wy$ for some $y \in \Sigma^*$. Note that y can be ε. The same applies for suffix, which we denote as $w \sqsupset x$.

5.2.1 String matching algorithms

The easiest way to do string matching is called the naive string-matching algorithm. It finds all the valid shifts by using a loop that checks $P[1\,..\,m] = T[s+1\,..\,s+m]$ for each of the possible values of s. This is clearly not an optimal procedure, since it's running-time complexity is $\Theta(n^2)$.

A better method is one that uses a finite automaton to do the matching. This works as follows:

A finite automaton M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where Q is a finite set of states, $q_0 \in Q$ is the start state, $A \subseteq Q$ is the set of accepting states, Σ is our input alphabet, and δ is a function $Q \times \Sigma \to Q$ called the transition function of M. The automaton begins in state q_0 and it reads the characters of the input string one at a time. If the automaton is in some state $q \in Q$ and reads a, it moves from state q to state $\delta(q, a)$. Whenever $q \in A$, the machine M accepts the string read so far.

A finite automaton induces a function ϕ called the final-state function $\Sigma^* \to Q$ such that $\phi(w)$ is the state M ends up in after scanning the string w. Thus M accepts a string w iff $\phi(w) \in A$. We define ϕ as

$$\begin{aligned} \phi(\varepsilon) &= q_0, \\ \phi(wa) &= \delta(\phi(w), a) \qquad\qquad \text{for } w \in \Sigma^*, a \in \Sigma \end{aligned}$$

When we want to use a finite automaton for matching a string on a pattern P, we first need to construct the automaton for P. The most important part of this is computing δ, which requires first computing a different function σ. The function $\sigma : \Sigma^* \to \{0, 1, \ldots, m\}$ is the suffix function corresponding to P such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

Using this suffix function we can define our transition function:

$$\delta(q, a) = \sigma(P_q a)$$

To see why this is accurate, please read page 998 in the book, seeing as I don't actually get it at all. An algorithm that computes the transition function δ is shown in Algorithm 8.

Algorithm 8: Computing the transition function

```
1  Function COMPUTE-TRANSITION-FUNCTION($P$, $\Sigma$)
       Input: The pattern $P$ and the alphabet $\Sigma$ corresponding to $P$
       Output: The transition function $\delta : Q \times \Sigma \to Q$ corresponding to $P$ and $\Sigma$ for a
               finite automaton to use to do string matching on $P$
2      $m = P.length$
3      for $q = 0$ to $m$ do
4          foreach $a \in \Sigma$ do
5              $k = \min(m + 1, q + 2)$
6              repeat
7                  $k = k - 1$
8              until $P_k \sqsupset Pqa$
9              $\delta(q, a) = k$
10     return $\delta$
```

The running time of our function COMPUTE-TRANSITION-FUNCTION is $O(m^3|\Sigma|)$. Much faster procedures exist to compute δ, making for a much faster preprocessing time for the matching procedure, but we will not focus on these. Instead we consider the Knuth-Morris-Pratt algorithm, which avoids computing δ altogether and instead uses an auxiliary function π which is computed in $\Theta(m)$ and stored in $\pi[1\,..\,m]$. This array π allows computing δ "on the fly". For any state q, $\pi[q]$ allows is to compute $\delta(q, a)$ in a manner that does not depend on a, saving us a lot of processing time and storage space.

The prefix function π for some pattern encapsulates knowledge about how the pattern matches against shifts of itself. This allows us to avoid useless shifts in the naive pattern-matching algorithm and allows us to avoid precomputing the full function δ for a string-matching automaton. Given a pattern $P[1\,..\,m]$, the prefix function $\pi : \{1, 2, \dots, m\} \to \{0, 1, \dots, m\}$ for P is

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q. An algorithm to compute this is shown in Algorithm 9.

Algorithm 9: Computing the prefix function

```
Function COMPUTE-PREFIX-FUNCTION($P$, $\Sigma$)
    Input: The pattern $P$
    Output: The prefix function $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m\}$ corresponding to $P$
    $m = P.length$
    let $\pi[1..m]$ be a new array
    $\pi[1] = 0$
    $k = 0$
    for $q = 2$ to $m$ do
        while $k > 0$ and $P[k+1] \neq P[q]$ do
            $k = \pi[k]$
        if $P[k+1] == P[q]$ then
            k = k + 1
        $\pi[q] = k$
    return $\pi$
```

Note: In this section I forego writing the exact matching algorithms I discuss. The actual matching isn't as much the focus of this section as is the transition function computation. To see the actual algorithms, refer to the book chapter 32.

Chapter 6

P vs NP

6.1 Formal-language framework

To formally talk about problem classes we can use formal-language theory. We review this here. An alphabet Σ is a finite set of symbols. A language L over Σ is any set of strings made up from symbols in Σ. We denote the empty string by ε, the empty language by $\varnothing$, and the language of all strings over Σ by Σ^*.

Operations we can perform on languages include things such as union and intersection. We also define the complement of $L = \Sigma^* - L$ and the concatenation $L = L_1 L_2$ as $L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}$. The closure of a language L is $L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$ where L^k is the language obtained by concatenating L with itself k times.

From the point of language theory any set of instances for any decision problem Q is the set Σ^* where $\Sigma = \{0, 1\}$. We view Q as a language over Σ where $L = \{x \in \Sigma^* : Q(x) = 1\}$. We can now express the relation between decision problems and the algorithms that solve them. We say that an algorithm A accepts an answer $x \in 0, 1^*$ if given input x the algorithm's output $A(x) = 1$. The language accepted by A is the set of strings $L = \{x \in 0, 1^* : A(x) = 1\}$, that is, the set of strings that the algorithm accepts. We call a language L decided by some algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A.

6.2 Polynomial time

A language L is accepted in polynomial time by an algorithm A if it is accepted by A and in addition there exists a constant k such that for any length-n string $x \in L$ algorithm A accepts x in time $O(n^k)$. A language L is decided in polynomial time by an algorithm A if there exists a constant k' such that for any length-n string $x \in 0, 1^*$ the algorithm correctly decides whether $x \in L$ in time $O(n^{k'})$.

Some problems are hard to solve but easy to verify. For example, if we wish to find a hamiltonian cycle in an undirected graph, we cannot do so in polynomial time. However, if we have a proposed solution to the problem, we can verify this in polynomial time. If we define the formal language

$$\text{HAM-CYCLE} = \{\langle G \rangle : G \text{ is a hamiltonian graph}\}$$

we can try to see how an algorithm might try to decide this language. We can check all permutations of edges and see whether this is a hamiltonian cycle or not. Using a adjacency matrix, we have $m!$ possible permutations and therefore a running-time of $\Omega(m!)$, and there is no k such that this is in $O(n^k)$. Clearly, this problem is not part of the class P.

6.3 The complexity class NP

The complexity class NP consists of all the languages that can be verified by a polynomial-time algorithm. By definition, if $L \in$ P then $L \in$ NP, since being able to solve a problem in polynomial time means it can also be verified in polynomial time. It is unknown whether P $=$ NP, but most researchers assume it is not.

6.4 NP-completeness and reducibility

The class of "NP-complete" problems has the property if *any* problem in the class turns out to be solvable in polynomial time, then *every* problem in NP has a polynomial-time solution. No polynomial-time algorithm has ever been discovered for any NP-complete problem.

The NP-complete problems are the "hardest" languages in NP. We will define in this section how to compare the relative hardness of languages using polynomial-time reducibility. We will also formally define the NP-complete class of languages.

6.4.1 Reducibility

A problem Q can be reduced to another problem Q' if any instance of Q can be easily rephrased as an instance of Q', the solution to which provides a solution to the instance of Q. We say that a language L_1 is polynomial-time reducible to a language L_2, written $L_1 \leq_p L_2$, if there exists some polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in L_1$ iff $f(x) \in L_2$.

Polynomial-time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. We can define the set of NP-complete languages now: A language $L \subseteq \{0,1\}^*$ is NP-complete if (1) $L \in$ NP, and (2) $L' \leq_p L$ for every $L' \in$ NP. If our language L satisfies property 2 but not property 1 then we say that L is NP-hard.

Chapter 7

Greedy Algorithms

Greedy algorithms are algorithms that make a choice at the moment, that is, it makes the locally optimal choice in hopes that this will lead to a globally optimal solution. Not for every problem does there exist a greedy algorithm that can solve the problem with a globally optimal solution, but for many problems there does exist one. If we can find these we can solve the problem much faster than we could with, for example, dynamic programming.

When trying to come up with a greedy solution to a problem there is a set of steps we can follow: (1) Determine the optimal substructure of the problem; (2) Develop a recursive solution; (3) Show that if we make a greedy choice then only one subproblem remains; (4) Prove that it is always safe to make the greedy choice; (5) Develop a recursive algorithm that implements the greedy strategy; (6) Convert the recursive algorithm to an iterative algorithm.

As we can see there is two properties that we must prove for a greedy strategy. These are the greedy-choice property and the optimal substructure property.

Greedy-choice property is a property that says that making the locally optimal choice leads to a globally optimal solution. The choice may depend on previous choices made so far, but it may not depend on any future choices or solutions to subproblems. Generally we prove this by examining a globally optimal solution to some subproblem. We then show how to modify the solution to substitute the greedy choice for some other choice, resulting in a similar but smaller subproblem.

Optimal substructure. A problem has optimal substructure when an optimal solution to a problem contains within it optimal solutions to subproblems. This is key in greedy algorithms, as it is in dynamic programming as well. To prove that this is the case for some problem we can simply show that an optimal solution to the subproblem combined with the already-made greedy choice yields an optimal solution to the original problem. This implicitly uses induction on the subproblems to prove the property at every step.

7.1 Huffman codes

Huffman codes compress data very effectively. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs to build an optimal way of representing each character as a binary string. We consider here the problem of designing such a binary character code. We can do this either by giving each character a fixed-length code, or we can use variable-length codes, which generally yield much better results.

We consider here prefix codes: codes in which no codeword is also the prefix of another codeword. This can always achieve optimal data compression among any character code, though we don't prove this here. This allows us to use a tree-like method of decoding.

Huffman invented a greedy algorithm that constructs an optimal prefix called the Huffman code. This algorithm is shown in Algorithm 10. The algorithm builds a tree T corresponding to the optimal code in a bottom-up manner.

Algorithm 10: Huffman's greedy algorithm

```
Function HUFFMAN($C$)
    Input: The set $C$ of $n$ characters which each character $c \in C$ having an attribute
           $c.freq$ giving its frequency in the to-be-encoded text
    Output: The tree $T$ corresponding to the optimal code in a bottom-up manner
    $n = |C|$
    $Q = C$          /* Q is a min-priority-queue keyed on char frequency */
    for $i = 1$ to $n - 1$ do
        allocate a new node $z$
        $z.left = x = $ EXTRACT-MIN($Q$)
        $z.right = y = $ EXTRACT-MIN($Q$)
        $z.freq = x.freq + y.freq$
        INSERT($Q, z$)
    return EXTRACT-MIN($Q$)          /* Return the root of the tree T */
```

Chapter 8

Hashing

A hash table is a datastructure that allows the efficient implementation of a dictionary, with the operations INSERT, SEARCH, and DELETE. Searching for an element can take Θn in the worst case, but on average it is $O(1)$. There are even ways to have $O(1)$ searching in the worst-case, if we impose some constraints on the data to be stored.

8.1 Direct-address tables

Direct addressing is a simple technique to use when the universe of keys U is relatively small. We use an array for this, or direct-address table, denoted by $T[0\,..\,m-1]$ in which each slot corresponds to a key $k \in U$. If the set contains no element with key k then $T[k] = \text{NIL}$. The dictionary operations are very easy to implement and are shown in Algorithm 11.

Algorithm 11: Dictionary operations in direct-address tables

```
1 Function DIRECT-ADDRESS-SEARCH($T, k$)
2     return $T[k]$
3 Function DIRECT-ADDRESS-INSERT($T, x$)
4     $T[x.key] = x$
5 Function DIRECT-ADDRESS-DELETE($T, x$)
6     $T[x.key] = \text{NIL}$
```

The downside of direct addressing is obvious: if U is large, storing T of size $|U|$ may not be practical, especially if the set K of keys $k \in U$ that are actually stored is much smaller than U, meaning we waste a ton of memory. This is why a hash table is often a better solution.

8.2 Hash tables

When direct addressing, an element with key k is stored in $T[k]$. With hashing, we store the element in $T[h(k)]$, where h is our hash function. We say that an element with key k hashes to slot $h(k)$. The hash function reduces the range of array indices and with that the size of the array T. Since $h : U \to \{0, 1, \ldots, m\}$, $|T|$ is reduced from $|U|$ to m.

One downside of this method is that a collision may occur, where multiple keys can hash to the same slot. There are multiple methods of resolving the conflict created by this collision:

Collision resolution by chaining. By making every slot in T a linked list we can store multiple elements in one slot. This means we need to change the functions for searching, inserting, and deleting, as shown in Algorithm 12

Algorithm 12: Dictionary operations in hash tables with chaining to avoid collisions

```
1 Function DIRECT-ADDRESS-SEARCH($T, k$)
2     search for an element with key $k$ in list $T[h(k)]$
3 Function DIRECT-ADDRESS-INSERT($T, x$)
4     insert $x$ at the head of list $T[h(x.key)]$
5 Function DIRECT-ADDRESS-DELETE($T, x$)
6     delete $x$ from the list $T[h(x.key)]$
```

Taking this approach to collision avoidance has an impact on the time complexity of our dictionary. Hypothetically, if our hash function were to return the same slot for every key, we would have all elements in our list, and so our retrieval would take $\Theta(n)$. We can improve the average case to close to $O(1)$ by choosing a good hash function, which we'll talk about later.

Collision resolution by open addressing. In open addressing, all elements occupy the table itself. This means we have a max number of elements we can store. To avoid collision in open addressing, we use a technique called probing. Our hash function takes two parameters in this case, the key as before, and a number indicating the number of tries so far. This means that we try a different location in the hash table after colliding with some other element previously, until we have found a location to store our element.

There are three methods of probing we consider here: linear probing, quadratic probing, and double hashing.

Linear probing uses the hash function $h(k,i) = (h'(k) + i) \bmod m$ so that we just try the next element continuously until we find a location where our item fits. This approach is simple, but not good for the average search time, since it clusters elements in one location in our hash table.

Quadratic probing uses a hash function of the form $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$. This has an offset that is not linear and thus it distributes the items better over the array, taking a bigger offset the bigger i gets.

Double hashing uses a hash function of the form $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$ where both h_1 and h_2 are auxiliary hash functions. This leads to the best results for searching times. There is one constraint that is that $h_2(k)$ must be relatively prime to m to ensure the entire table is searched. The easiest way to do this is to ensure m is a power of 2 and $h_2(k)$ always returns an odd number.

8.3 Hash keys

The most important property for hash function to perform well is that it has uniform hashing, that is, every slot in the range $[1\,..\,m]$ has an approximately equal probability of being chosen. Sadly we don't often know the distribution of the keys, and thus a good approach is ensure that our hash function produces wildly different result for similar keys. A few approaches for hash keys are the following:

The division method maps a key k into one of the m slots by taking the remainder of k divided by m. This gives

$$h(k) = k \bmod m$$

With this method we try to avoid certain values of m, such as powers of two or numbers close to a power of two. Generally, we want a prime that's not too close to a power of two.

The multiplication method multiplies the key k by a constant between 0 and 1 then extracts the fractional part and multiples it by m. This results in

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

With this method, the value of m isn't critical. Typically we choose it to be some power of two since this allows for easy implementation of the function on computers

Universal hashing can yield good performance in general, no matter what keys are used. The function used is

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$$

where p is a prime large enough that k is in the range $[0\,..\,p-1]$. This gives us a family of functions to use that hashes universally. The family is formally defined as

$$\mathcal{H}_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}$$

where $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ and $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$.

Chapter 9

Binary Search Trees

9.1 Binary Search Trees

The search tree is a data structure in which basic operations take time proportional to the height of the tree. For a complete binary tree with n nodes this is $O(\lg n)$, which means we can improve a lot over linear data structures like arrays when searching for a specific item.

A binary search tree is a binary tree. We represent this as a linked data structure where each node has some key, some data and pointers to its children and its parent. The keys in a binary search tree are always stored according to the binary-search-tree property:

> Let x be a node in a binary search tree. If y is a node in the left subtree of x then $y.key \leq x.key$. If y is a node in the right subtree of x then $y.key \geq x.key$.

This property allows us to print the key in sorted order by doing an inorder tree walk.

The property also makes searching in a binary tree very easy. If the key we're searching for is smaller than the element we're currently look at then we search the left subtree, it it's larger we search the right subtree, and if it's equal to the current element we have found what we our looking for and we can terminate. Similarly, we can easily find the minimum and maximum elements by going all the way left or all the way right, respectively.

To insert and delete values from a binary tree we use the procedures shown in Algorithm 13. How the three functions work is explained below:

TREE-INSERT inserts an element in the tree by searching the tree until it finds a node with no child where it can insert itself appropriately. Inserting keeps the binary-search-tree property intact.

TRANSPLANT replaces one subtree as a child of its parent with another subtree. This replaces the subtree rooted at u with the subtree rooted at node v, node u's parent becomes node v's parent and u's parent ends up having v as its appropriate child.

TREE-DELETE deletes a node from the tree. It uses TRANSPLANT to move around subtrees and it has if cases for the different situations that can happen. For a more in-depth explanation, refer to the book.

Algorithm 13: Inserting and deleting from a binary search tree

```
Function TREE-INSERT($T, z$)
    Input: A tree $T$ and a node $z$ to add to $T$
    $y = $ NIL
    $x = T.root$
    while $x \neq$ NIL do
        $y = x$
        if $z.key < x.key$ then
            $x = x.left$
        else
            $x = x.right$
    $z.p = y$
    if $y ==$ NIL then
        $T.root = z$ /* Tree T was empty */
    else if $z.key < y.key$ then
        $y.left = z$
    else
        $y.right = z$
Function TRANSPLANT($T, u, v$)
    if $u.p ==$ NIL then
        $T.root = v$
    else if $u == u.p.left$ then
        $u.p.left = v$
    else
        $u.p.right = v$
    if $v \neq$ NIL then
        $v.p = u.p$
Function TREE-DELETE($T, z$)
    if $z.left ==$ NIL then
        TRANSPLANT($T, z, z.right$)
    else if $z.right ==$ NIL then
        TRANSPLANT($T, z, z.left$)
    else
        $y = $ TREE-MINIMUM($z.right$)
        if $y.p \neq z$ then
            TRANSPLANT($T, y, y.right$)
            $y.right = z.right$
            $y.right.p = y$
        TRANSPLANT($T, z, y$)
        $y.left = z.left$
        $y.left.p = y$
```

9.2 Red-Black Trees

A red-black tree is a search tree with a color per node: red or black. By constraining the node colors on any simple path from root to leaf, red-black trees ensure that no path is more than twice as long as another, so that the tree is approximately balanced. A red black tree satisfies the following red-black properties: (1) Every node is either red or black; (2) The root is black; (3) Every leaf (NIL) is black; (4) If a node is red then both its children are black; (5) For each nodes all simple paths from the node to descendant leaves contain the same number of black nodes.

The operations to insert and delete from a red-black tree take $O(\lg n)$ time. Since they modify the tree, the red-black properties may be violated. To restore these properties we have to change the color of some nodes and also change the pointer structure. Changing the pointer structure is done through rotation which is a local operation that reserves the binary-search-tree property. We have left rotation and right rotation, which are the opposites of each other. See item 9.2 for a visual example of tree rotation.

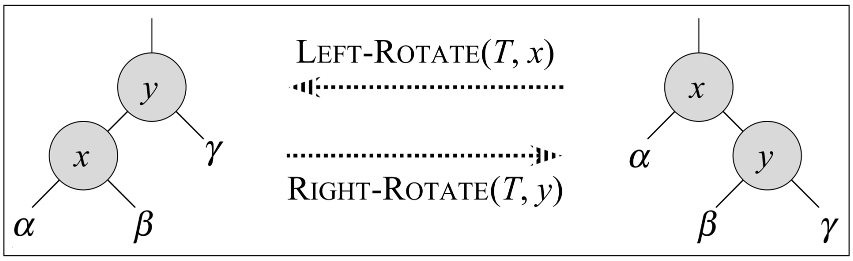

Figure 9.1: Tree rotation

To insert into a node z a red-black tree T we insert as if T were an ordinary binary search tree and color z red. We then call a procedure names RB-INSERT-FIXUP to repair our red-black-tree properties. The pseudocode for RB-INSERT-FIXUP is shown in Algorithm 14. For an exact explanation of how this works as it does, please refer to the book section 13.3. It is 5 pages of explanation that I can't properly summarize here. The same holds for deletion, which is even more complicated than inserting.

Algorithm 14: Fixing up a red-black tree after inserting

```
Function RB-INSERT-FIXUP(T, z)
    while z.p.color == RED do
        if z.p == z.p.p.left then
            y = z.p.p.right
            if y.color == RED then
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else
                if z == z.p.right then
                    z = z.p
                    LEFT-ROTATE(T, z)
                z.p.color = BLACK
                z.p.p.color = RED
                RIGHT-ROTATE(T, z.p.p)
        else
            y = z.p.p.left
            if y.color == RED then
                z.p.color = BLACK
                y.color = BLACK
                z.p.p.color = RED
                z = z.p.p
            else
                if z == z.p.left then
                    z = z.p
                    RIGHT-ROTATE(T, z)
                z.p.color = BLACK
                z.p.p.color = RED
                LEFT-ROTATE(T, z.p.p)
```

Chapter 10

Graphs

Note: In this chapter I will not put any pseudocode as I did before; instead I will try to explain the concepts as much as possible on a more intuitive level. The pseudocode for all these algorithms is quite large and this chapter would become unwieldily large were I to put it all in.

10.1 Graph Representation

We have two ways to represent a graph $G = (V, E)$: as a collection of adjacency lists or as an adjacency matrix. The adjacency-list representation is well suited for sparse graphs, since there is just a single entry for each edge. It has memory complexity $\Theta(V + E)$. Storing a dense graph works better in an adjacency matrix, which has memory complexity $\Theta(V^2)$. If memory isn't a concern and we need fast edge weight access, then the adjacency matrix is also the best option, since for the matrix edge weight access time are $O(1)$, and for the adjacency list this is sometimes more.

10.2 Elementary Graph Algorithms

10.2.1 Breadth-first search

Breadth-First search (BFS) is one of the simplest graph-searching algorithms. It can be used to find the optimal path from one node to another in an unweighted graph.

Given a graph $G = (V, E)$ and some source vertex s, BFS systematically explores the graph to find the vertices reachable from s. It also produces a breadth-first tree with root s that contains all the reachable vertices. From this we can derive the shortest path from s to any reachable vertex.

BFS is named as such because it expands the frontier between discovered and undiscovered vertices in a breadth-first manner. It discovers all the vertices at distance k before discovering any at $k + 1$, hence why it finds the optimal paths.

To search, BFS keeps track of a node's color: black, gray, or white. A white vertex hasn't been discovered yet; a gray vertex has been discovered but has undiscovered neighbors; and a black vertex is discovered and has all gray and/or black neighbors. BFS also keeps track of a vertex's predecessor, this being the node that it was discovered from. This allows us to reconstruct the path from the source s to any discovered node by traversing backwards from that node to s, following the predecessor trail.

Breath-first search uses a queue datastructure to keep track of vertices. When a node is discovered, it is placed in the queue. When all neighbors of the current node have been placed in the queue, the first node in the queue is popped and we try to discover all its neighbors. When the queue is empty we have searched all connected vertices to s. This is pretty trivial to implement so I will not put the pseudocode here.

10.2.2 Depth-first search

Depth-first search (DFS) prioritizes deeper searching whenever possible, as opposed to BFS's breadth-first approach. Depth-first search explores edges out of the most recently discovered vertex v that still has unexplored neighbors. Once all of v's edges have been explored DFS backtracks to explore other edges.

The predecessor subgraph of DFS is also different from that of BFS. The predecessor subgraph of DFS forms a so-called forest, since the search may repeat from multiple sources as long as disconnected components exist. The pseudocode for DFS is given in the book.

DFS assigns every vertex a discovery time d and finishing time f. The discovery time is the first time DFS visits the vertex, and the finishing time is when DFS returns out of this vertex since all of its edges have been explored fully. When not discovered the vertex is white; when it is in between d and f it is gray; and when it is done, so f is known, the vertex is colored black.

Topological sort using DFS

We can use depth-first search to do a so-called topological sort of a directed acyclic graph (dag). Such a dag often is used as a dependency graph, so certain vertices can only be visited (or can only 'happen', if they signify actions) after other vertices have been visited. If we call DFS for each vertex v as to compute finishing times f, we can put each vertex on a stack as the vertex finished. If we the pop the entire stack, we get our topological sort.

Strongly connected components

Two vertices in a graph are strongly connected if you can go from either node to the other. To compute the strongly connected components of a graph we do the following: (1) Call DFS(G) to compute finishing times f for each vertex v; (2) Compute G^T, that is defined as $G^T = (V, E^T)$ where $E^T = \{(u,v) : (v,u) \in E\}$; (3) Call DFS($G^T$) and consider the vertices in order of decreasing f; (4) Output the vertices of each tree in the depth-first forest formed in the previous step ass separate strongly connected components.

10.3 Minimum Spanning Trees

A minimum spanning tree of a graph is a tree that spans over the graph with minimal edge costs. In this case spanning means that it reaches all vertices.

To grow a spanning tree on an undirected graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ we consider two algorithms with a greedy approach. Both algorithms' strategy is captured by the following generic method, which grows the minimum spanning tree one edge at a time.

We maintain the loop invariant that prior to each iteration, A is a subset of some minimum spanning tree, where A is a set of edges. At each step, we determine an edge (u, v) that we can add to A without violating this invariant, in the sense $A \cup \{(u, v)\}$ is also a subset of a minimal spanning tree. We call such an edge a safe edge. We keep adding safe edges to this set until we A forms a minimal spanning tree. The tricky part here is finding the safe edge. The book goes on to formally define this, but finding the safe edge is where the two algorithms differ, so we will go directly to discussing the algorithms.

10.3.1 Kruskal's algorithm

Kruskal's algorithm finds a safe edge to add by seeing what the edge with the least weight is that connects two trees in the graph. This algorithm creates $|V|$ trees each containing one vertex. It then connects these trees together by continuously choosing the cheapest edge to do so with. It never connects an edge that is connected to the same tree on either side.

The running time of Kruskal's algorithm depends highly on what datastructure you use to implement the multiple disjoint sets. Since this is a chapter not covered in this course, we will ignore the time-complexity calculation for this algorithm.

10.3.2 Prim's algorithm

Prim's algorithm chooses which edge to add to the growing tree by selecting the globally cheapest edge that connects the tree to a node that is not yet in the tree. To store these edges we can simply use a min-priority queue that is keyed on the edge weight. When the algorithm terminates we have an empty queue and we have the minimum spanning tree A of G.

Using a min-heap to implement this algorithm yields a time complexity of $O(E \lg V)$. Using a Fibonacci heap (which we haven't covered) we improve this to be $O(E + V \lg V)$.

10.4 Single-Source Shortest Paths

If we wish to find the shortest path from some source to some destination we can do so in a couple ways, which we discuss in this section. We first define some things: We have a weighted, directed graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$. The weight $w(p)$ of some path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum

$$w(p) = \sum_i = 1^k w(v_{i-1}, v_i)$$

We define the shortest-path weight $\delta(u,v)$ by

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

Optimal paths have the optimal substructure that any path within it is also the shortest path between the respective vertices that that path is between. Many algorithms that find the shortest path rely on this, since this optimal substructure is an indication that dynamic programming or the greedy paradigm may apply.

Some instances of the single-source shortest-paths problem may include negative edges. If our graph has negative-weight cycles in it, the problem is not well-defined, since any solution including is improved by including the cycle one more time, until we get to a solution with weight $-\infty$. If we have a negative-weight cycle on the path from s to v, we, as such, define $\delta(s,v) = -\infty$. Positive cycles also can't be contained in a shortest path, since excluding it is always possible *and* leads to a better path than the one containing the cycle.

The algorithms in this section use the technique of relaxation. For each vertex $v \in V$ we maintain an attribute d that is the upper bound on the weight of the shortest path from s to v. We call this the shortest-path estimate. This value is initialized as ∞. The process of relaxing an edge (u,v) consists of testing whether we can improve the shortest path to v found so far by going through u, and if so update $v.d$ and $v.\pi$[1].

10.4.1 The Bellman-Ford algorithm

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. It returns a boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is not, the algorithm produces the shortest paths and their weights.

The algorithm initializes the d and π attributes first, as named above. It them loops over each edge $|V| - 1$ times and relaxes it. It does this one edge at a time, $|V| - 1$ times, not one edge $|V| - 1$ times, then the next edge, etc. At the end we check for a negative-weight cycle and return FALSE if one is found. We return TRUE otherwise.

10.4.2 Single-source shortest paths in dags

By relaxing the edges of a weighted dag according to the topological sort of its vertices we can compute the shortest paths from a single source in $\Theta(V + E)$ time. In a dag a shortest path is always well defined, since by definition no negative-weight cycles can exist.

10.4.3 Dijkstra's algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on w weighted directed graph for the case in which all edges are non-negative.

[1]Recall that π is the predecessor; used to reconstruct the path later on

Dijkstra's algorithm maintains a set of S vertices whose final shortest-path from s have already been determined. It repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S and relaxes all edges leaving u. It uses a min-priority queue to select these edges from $V - S$.

With some implementation details and constraints, Dijkstra's algorithm can run with time complexity $O(V \lg V + E)$. For the exact analysis, refer to the book chapter 24.3.

10.4.4 Floyd-Warshall algorithm

The Floyd-Warshall algorithm uses a different approach to solve the all-pairs shortest-path problem on a directed graph $G = (V, E)$. The resulting algorithm runs in $\Theta(V^3)$ time. We assume no negative-weight cycles exist.

The Floyd-Warshall algorithm relies on the observation that a shortest path from i to j is the concatenation of the shortest paths from i to k and from k to j. If we let $d_{ij}^{(k)}$ be the weight of the shortest path from vertex i to j for which all intermediate vertices are in the set $\{1, 2, \ldots, k\}$, then we can define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\} & \text{if } k \geq 1 \end{cases}$$

The matrix $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer: $d_{ij}^{(n)} = \delta(i, j)$ for all $i, j \in V$.

There is multiple methods of reconstructing the final path from this. Since I personally don't fully understand them, refer to the book for this. I think the actual underlying understanding above is the most important part anyway.

10.5 Maximum Flow

If we interpret a graphs edge weights as capacities we can come up with a problem of maximum flow. How much can flow from one node to another within a unit of time? We define a flow network formally as: Let $G = (V, E)$ be a flow network with capacity function c. Let s be the source and t be the sink. A flow in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the following two properties: (1) For all $u, v \in V$ we require $0 \leq f(u, v) \leq c(u, v)$ (flow is non-negative and there is at most so much flow as there is capacity); (2) For all $u \in V - \{s, t\}$ we require $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$.

In the maximum flow problem, we wish to have a single source and sink. If this is not the case, we can add a supersource and supersink with infinite capacity edges to the existing sinks and sources. We can use these to solve the problem with.

10.5.1 Ford-Fulkerson method

The Ford-Fulkerson method solves the problem of maximum flow. This method iteratively increases the value of flow. We start with $f(u, v) = 0$ for all edges. At each iteration we increase the flow value in G by finding an "augmenting path" in the associated residual network G_f.

Given a flow network G and a flow f, the residual network G_f is the flow network with capacities that represent how we can change the flow on edges of G. An edge of the flow network can admit some amount of additional flow. If that value is positive, we place it in G_f. We define the residual capacity of some edge by

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases}$$

The residual network $G_f = (V, E_f)$ is now given by

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$$

Given a flow network G and a flow f an augmenting path p is a simple path from s to t in the residual network G_f. This augmenting path may increase the flow on an edge (u,v) by up to $c_f(p)$ without violating the capacity constraint in the original flow network G. The augmenting capacity $c_f(p)$ is given by

$$c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$$

Now that we have these concepts, we can solve the maximum flow problem. A pseudocode implementation is given in Algorithm 15.

Algorithm 15: Ford-Fulkerson method for the maximum-flow problem

```
Function FORD-FULKERSON($G$, $s$, $t$)
    Input: A flow network $G$ where the $f$ component for each edge is 0; a source $s$; and
           a sink $t$
    Output: The maximum flow between $s$ and $t$ in $G$
    while there exists a path $p$ from $s$ to $t$ in $G_f$ do
        $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$
        foreach $(u,v) \in p$ do
            if $(u,v) \in G.E$ then
                $(u,v).f = (u,v).f + c_f(p)$
            else
                $(v,u).f = (v,u).f - c_f(p)$
```