Computer Science 2 Notes

Giacomo Ellero

Semester 2, 2023/2024

Contents

1 Introduction

1.1 Asymptotic notation

Definition

If $\exists C \in \mathbb{R}^+$ and $N \in \mathbb{N}$ such that for two sequences $a_n, b_n > 0$ we have that $a_n \leq Cb_n$ for all $n \geq N$, then we write $a_n = O(b_n)$ and we read "a_n is big O of b_n".

These sequences describe the time it takes for an algorithm to solve a certain problem of size n.

Example

Let $a_n = n^2 + 2n + 1$. We will prove that $a_n = O(n^2)$.

Proof. We have that

$$\begin{aligned} a_n &= n^2 + 2n + 1 \\ &\leq n^2 + 2n^2 + n^2 \\ &= 4n^2 \end{aligned}$$

$\square$

As we can see just need to show that C exists, we don't need to find the best one.

Usually we don't use limits to prove that a sequence is big O of another, it is usually more convenient to proceed by inequalities.

Note: Logarithms

When we have sequences with logarithms we don't need to specify the basis, as the logarithm is a constant factor of another logarithm by the change of basis formula.

1.1.1 Operations and other notation

Let $a_n = O(c_n)$ and $b_n = O(d_n)$, then we have that

- $a_n + b_n = O(\max\{c_n, d_n\})$
- $a_n \cdot b_n = O(c_n \cdot d_n)$

We also define the "opposite" of big O notation, the Ω notation: a_n is $\Omega(b_n)$ if $a_n \geq Cb_n$ for all $n \geq N$.

Moreover, if $a_n = O(b_n)$ and $a_n = \Omega(b_n)$ (for some different C and N) then we write $a_n = \Theta(b_n)$.

1.2 Randomness

In some algorithms we will need to use randomness to solve the problem, therefore we need a language to talk about it.

When we run an experiment we can define

- an outcome ω_i
- the value of the outcome x_i (similar to a bet)
- the probability of the outcome p_i

We can also define the expected result $E(X)$ of the experiment as

$$E(X) = \sum_{i=1}^{n} x_i p_i$$

For more about probability see the notes of the probability course.

1.3 IEEE-754

This is not in the syllabus but it's useful to know.

IEEE-754 is a standard for representing floating point numbers in computers. We will discuss in particular the 32-bits representation but larger or smaller formats also exist.

We subdivide the 32 bits in 3 parts: 1 bit for the **sign**, 8 bits for the **exponent**, and 23 bits for the **fraction** or mantissa.

0								8																							31
1	0	0	1	0	0	1	0	1	0	0	1	1	0	1	1	0	1	0	1	1	0	1	0	0	1	1	1	1	0	1	1
Sign	Exponent								Fraction																						

Figure 1: IEEE-754 32-bits representation

The number is interpreted according to the following formula:

$$n = (-1)^s \cdot 2^{e-127} \cdot (1 + f)$$

Basically the number is represented in scientific notation, in base 2.

Note that when converting the fraction part to decimal the powers of 2 are negative and decreasing (i.e.: bit at 9 is 2^{-1}, bit at 10 is 2^{-2}, etc).

2 Basic algorithms

2.1 Introductory statements

2.1.1 How to approach a algorithm problem

We usually proceed by following these steps:

1. Find an algorithm
2. Prove that the algorithm is correct
3. Calculate the time complexity of the algorithm
   (a) Can we do better?

We will use the following steps to approach some classic problems such as addition and multiplication of binary numbers.

2.1.2 Shifts

We refer to shifts as the operation of moving all the bits of a number to the left or to the right by a certain amount of positions. This operation mathematically corresponds to multiplying or dividing the number by a power of the base.

$$x \ll n = x \cdot b^n$$

The $\ll$ symbol denotes a left shift by n positions, while the $\gg$ symbol denotes a right shift by n positions.

In our mathematical world shifts are $O(n)$, in the real world they are usually $O(1)$ because the computer can perform multiple shifts at the same time.

2.1.3 Useful facts

We will start by stating the following facts without providing a proof (prove them if you're curious but in class we skipped it).

- If $c \in \mathbb{R}^+$, then $g(n) = 1 + c + c^2 + \cdots + c^n$ is $\Theta(1)$ if $c < 1$, $\Theta(n)$ if $c = 1$, and $\Theta(c^n)$ if $c > 1$.
- In any base $b \geq 2$ the sum of any 3 single-digit numbers is at most 2 digits long.
- $\forall n \in \mathbb{N}$ and any base b there exists a power of b in $[n, bn]$.

Note: $\Theta(1)$

When we say that a function is $\Theta(1)$ we mean that it is bounded from above by a constant C (since it is big O of 1) and bounded from below by a constant c (since it is $\Omega(1)$).

This usually means that $a_n \neq 0$ for all n.

2.2 Addition

Problem: Let x, y be binary numbers of n bits. We want to compute $x + y$.

We proceed by using the normal addition algorithm: this is a well know algorithm that we know is correct.

At each step we are adding at most 3 numbers (2 bits and a carry). We know that the result of each step is at most 2 bits long: one bit goes for the result and the other goes for the carry. This ensures that at the next step we will also be adding at most 3 numbers, hence each step is performed in a finite amount of time.

Since we are performing n steps, the time complexity of this algorithm is $O(n)$.

Can we do better? No, we can't, since we need to read all the bits of the input and this operation is already $O(n)$.

2.3 Multiplication

Problem: Let x, y be binary numbers of n bits. We want to compute $x \cdot y$.

Again, we proceed using the normal multiplication algorithm that we know is correct. The algorithm has the following parts:

1. Write the multiplications of x by each bit of y
2. Sum all the results

Part 1 Let (s_n) be the sequence of the shifted values of x and p_n be the sequence of the partial products.

$$\begin{aligned} s_0 &= x \\ s_n &= s_{n-1} \ll 1 \\ p_n &= \begin{cases} 0 & \text{if } \text{y[n]} = 0 \\ s_n & \text{if } \text{y[n]} = 1 \end{cases} \end{aligned}$$

Choosing the right p_n is $O(1)$, but computing s_n is $O(n)$ hence this part is $O(n)$.

Note that we are "storing" the result of the previous shifts, so if we want to compute s_5, for example, we don't need to compute s_4 again. Without this optimization the time complexity of this part would be $O(n^2)$.

Part 2 In this step we are computing $\sum_{i=0}^{n} p_n$. These are sums of numbers of at most $2n$ bits, hence this part is $O(n)$.

Conclusion The time complexity of the multiplication algorithm is $O(n) \cdot O(n) = O(n^2)$.

2.3.1 Egyptian Multiplication

This is an ancient algorithm that is used to multiply two numbers. It works as follows:

1. Write the two numbers in two columns
2. Divide the first number by 2, floor the result and write it underneath it in the same column
3. Multiply the second number by 2 and write the result underneath it in the same column
4. When the first number is 1, sum all the numbers in the second column if the corresponding number in the first column is odd

We can easily implement this algorithm in Python as follows:

```python
def egyptian_multiplication(x, y):
    result = 0
    while x >= 1:        # This loop runs n times since x has n bits
        if x % 2 == 1:   # We always consider the worst case, hence this is always tr
            result += y  # Sums are O(n)
        x = x >> 2       # Shifts are O(n)
        y = y << 2       # Shifts are O(n)
    return result
```

The complexity of the algorithm is $(O(n) + O(n) + O(n)) \cdot O(n) = O(n^2)$.

2.3.2 Other multiplication algorithms

Another algorithm we can use to implement multiplication works by dividing each number in half and recursively computing the result.

Let x be a binary number of n bits, then x_{up} and x_{low} are both binary numbers of $\frac{n}{2}$ bits such that $x = 2^{\frac{n}{2}} x_{\text{up}} + x_{\text{low}}$.

We can use this fact to write the two numbers as

$$\begin{aligned} x &= 2^{\frac{n}{2}} x_{\text{up}} + y_{\text{low}} \\ y &= 2^{\frac{n}{2}} y_{\text{up}} + y_{\text{low}} \end{aligned}$$

Where x_{up} and y_{up} are the upper halves of x and y and x_{low} and y_{low} are the lower halves of x and y.

Then we compute the result as

$$x \cdot y = 2^n x_{\text{up}} y_{\text{up}} + 2^{\frac{n}{2}} (x_{\text{up}} y_{\text{low}} + x_{\text{low}} y_{\text{up}}) + x_{\text{low}} y_{\text{low}}$$

Each time we are performing 4 multiplication of half the size and adding them together. The time this algorithm takes is

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

However if we keep expanding the recursion we will see that eventually we get to $T(1)$ which is $O(1)$ and we are left with n-many $O(n)$ terms, hence the time complexity of this algorithm is also $O(n^2)$.

3 Divide and Conquer

We will now explore a class of algorithm that split the problem in smaller sub-problems which are easier to compute and then combine the results.

The last algorithm we discussed is a very simple example of a divide and conquer algorithm, now we will dive into more complex ones.

3.1 Karatsuba's algorithm

This is a multiplication algorithm that is based on the last algorithm we discussed with some variations.

The algorithm takes inspiration from the multiplication of two complex numbers:

$$\begin{aligned} (a + bi)(c + di) &= (ac - bd) + (ad + bc)i \\ \implies bc + ad &= (a + b)(c + d) - ac - bd \end{aligned}$$

Applying this to the multiplication of two binary numbers we get

$$x \cdot y = x_{\text{up}} y_{\text{up}} 2^n + 2^{\frac{n}{2}} ((x_{\text{up}} + x_{\text{low}})(y_{\text{up}} + y_{\text{low}}) - x_{\text{up}} y_{\text{up}} - x_{\text{low}} y_{\text{low}}) + x_{\text{low}} y_{\text{low}}$$

By doing this we are performing only 3 unique multiplications of half the size and 4 additions. Since we saw how additions are faster than multiplications we can expect this algorithm to be faster than the previous one.

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

We can arrange the work in nodes of a tree:

level	nodes	work per node	total work
0	1	Cn	Cn
1	3	$\frac{n}{2}C$	$\frac{3}{2}Cn$
2	9	$\frac{n}{4}C$	$\frac{9}{4}Cn$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	3^k	$\frac{n}{2^k}C$	$\left(\frac{3}{2}\right)^k Cn$

We have that the total work is the sum of the total work at each level of the tree:

$$Cn\left(1+\frac{3}{2}+\left(\frac{3}{2}\right)^2+\cdots+\left(\frac{3}{2}\right)^k\right)$$

Compared to the "old" algorithm that has a total work that looks like

$$Cn\left(1+2+2^2+\cdots+2^k\right)$$

Through some algebra we can show that this sum is $3Cn^{\log_2 3}$, hence proving that the algorithm is $O(n^{\log_2 3}) \sim O(n^{1.58})$.

3.2 The master theorem

This theorem helps us to compute the time complexity of divide and conquer algorithms. The proof of the theorem can be quite confusing and, despite being written here, it is not as important as understanding the statement.

Statement: If $a > 0, b > 1, d \geq 0$, n is a power of b, and $T(n)$ is defined by induction as

$$\begin{aligned} T(1) &= 1 \\ T(n) &= aT\left(\frac{n}{b}\right) + O(n^d) \end{aligned}$$

Where

- a is the number of subproblems
- b is the factor by which the problem size is reduced at each step
- d is the exponent in the work done at each level

Then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Proof. We can write a table with the work to be done at each size of the problem

level	# of problems	size of each problem	total work
0	1	n	Cn^d
1	a	$\frac{n}{b}$	$Ca\left(\frac{n}{b}\right)^d$
2	a^2	$\frac{n}{b^2}$	$Ca^2\left(\frac{n}{b^2}\right)^d$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	a^k	$\frac{n}{b^k}$	$Ca^k\left(\frac{n}{b^k}\right)^d$

with $k = \log_b n$.

To find the total work we need to sum all the total work at each step. With some algebra we can show that the total work is

$$Cn^d\left(1+\left(\frac{a}{b^d}\right)+\left(\frac{a}{b^d}\right)^2+\cdots+\left(\frac{a}{b^d}\right)^k\right)$$

We can consider the following 3 cases:

- If $\frac{a}{b^d}=1$, then $d=\log_b a$ and the sum becomes ??? and its complexity is $O(n^d \log n)$
- If $\frac{a}{b^d}<1$, then $d>\log_b a$ and the sum becomes ??? and its complexity is $O(n^d)$
- If $\frac{a}{b^d}>1$, then $d<\log_b a$ and the sum becomes ??? and its complexity is $O(n^{\log_b a})$

$\square$

4 Introduction to sorting algorithms

4.1 Comparison based sorting

Every comparison-based sorting algorithm must do $\Omega(n\log n)$ comparison to sort n distinct elements in the worst case.

Statement: For any comparison-based algorithm A, for all $n \geq 2$ there exists an input of size n such that A makes at least $\log_2(n!)=\Omega(n\log n)$ comparisons.

The $n!$ comes from the fact that the factorial represents all the possible permutations of n numbers, hence, when we are sorting, we are just finding the "correct" permutation that sorts the input.

The $\log_2$ comes from the fact that each comparison we are making a choice, which the end gives two outcomes, each one with half the options available for the next permutation. Since we are looking for the worst outcome we have that we stop after $\log_2$ of the number we started with.

4.2 Sorting without comparisons

This is kinda surprising but it is actually possible if we have some other information about the input we are sorting.

For example, if we know that the input is a sequence of integers in a certain range we can use the **counting sort** algorithm. This algorithm works by counting the occurrences of each number in the input and then writing them in the correct order. This is a very fast algorithm compared to other generic sorting and takes $O(n)$ time.

Other examples of sorting algorithms that don't use comparisons are **radix sort** and **bucket sort**.

5 Finding the median

In general we can sort the list and find the median like that, which would be $O(n\log n)$. It turns out that we can do better than that.

We will solve the more general problem that looks like this:

- *Input*: A list of numbers S; an integer k
- *Output*: The k-th smallest element of S

In particular if $k=\frac{|S|}{2}$ we have that k is the median.

5.1 Description of the algorithm

To do so we will use a recursion based algorithm:

1. Select an element V from S at random
2. We create 3 sub-lists:
   (a) $S_L = x \in S : x < V$
   (b) $S_V = x \in S : x = V$
   (c) $S_R = x \in S : x > V$
3. We now call the algorithm recursively with the following parameters:
   (a) If $k \leq |S_L|$ then we call the algorithm with S_L and k
   (b) If $|S_L| < k \leq |S_L| + |S_V|$ then we return V
   (c) If $k > |S_L| + |S_V|$ then we call the algorithm with S_R and $k - |S_L| - |S_V|$

5.2 The choice of V

To calculate the time complexity of this algorithm we first have to choose a good V. This is a hard task, because the "best" V would be the median, but that's exactly what we are trying to find; on the other hand the "worst" V would be the minimum or the maximum of the input, which would reduce the problem size by only 1.

Since we cannot choose a good V we will choose a random V: the odds of choosing the best or the worst cases are both extremely unlikely, hence we need to define a "good enough" V.

We consider a good enough V if it lies between $\frac{1}{4}$ and $\frac{3}{4}$ of the input, hence we can expect that the size of the sub-problems will be at most $\frac{3}{4}$ of the original problem.

This choice is arbitrarily, we could have chosen any other fraction and the time complexity, as we will see later, would have been the same. We chose these fractions because in this way the input is divided in 2 "bad" cases of size $\frac{1}{4}$ and a "good" case of size $\frac{1}{2}$, hence the size of the "bad" part is the same as the size of the "good" part.

5.3 Time complexity

We can now compute the time complexity

$$T(n) = \underbrace{T\left(\frac{3}{4}n\right)}_{\text{expected reduction}} + \underbrace{O(n)}_{\text{comparisons with } V} + \underbrace{\gamma O(n)}_{\text{finding a good } V}$$

Note that γ is the byproduct of the random choice of V, we don't know how many times we will have to choose V to find a good one.

Although we cannot be sure about the exact time this algorithm takes, we can compute the **expected** time.

We saw before how the probability of choosing a good V is $\frac{1}{2}$. We can calculate the expected time as follows: we pick a random V and

1. If V is good we are done
2. If V is bad we need to repeat the algorithm

Hence we have that $E = 1 + \frac{1}{2}E$, thus $E = 2$. We have

$$\begin{aligned} E(T(n)) &= E\left(T\left(\frac{3}{4}n\right)\right) + O(n) + 2O(n) \\ &= O(n) + O(n) + 2O(n) \\ &= O(n) \end{aligned}$$

Note: Other solutions

This algorithm is presented to illustrate that algorithms that make use of random variables exist and can be useful; in the case of this specific problem there exist better algorithms that can solve the problem in a deterministic way, such as the **median of medians** algorithm.

6 Introduction to Graphs

This week we will be working on graphs, this is referenced in chapter 3 of the book.

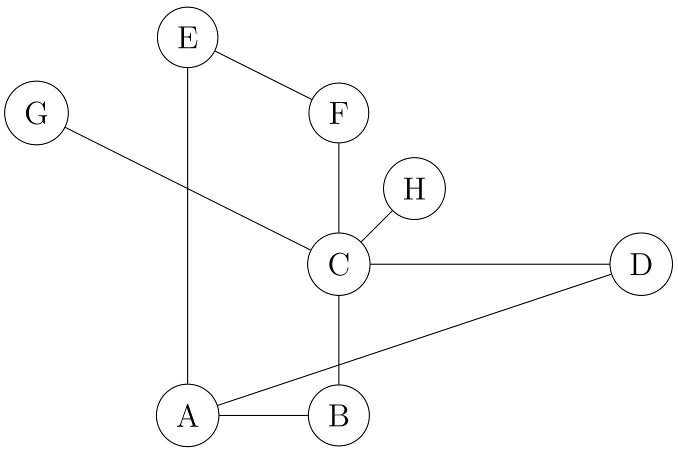

Figure 2: figure
An example of a graph

Graphs can be useful to represent a lot of different things, for example relationships, networks, maps, tournaments, etc.

6.1 Definitions

Definition

A **graphs** $G = (V, E)$ is a pair of finite sets where V is the set of **vertices** (or nodes) and E is the set of **edges**.

E is a subset of $V \times V$.

The elements of E are pairs of vertices that represent the connections between the vertices.

These are some other useful definitions we will use:

Adjacent vertices If $e = (u, v) \in E$ then we say that u and v are adjacent.

Incident edges If $e = (u, v) \in E$ then we say that e is incident to u and v.

Neighbour A vertex joined by an edge to another vertex is called a neighbor.

Directed and undirected graphs a graph is *directed* if the edges have a direction, otherwise it is *undirected*. In an undirected graph the edges are unordered pairs $\{u, v\}$, in a directed graph they are ordered pairs (u, v).

Loop if a vertex is connected to itself we say that it is a loop.

Simple graph a graph is simple if it has no loops and any pair of vertices is connected by at most one edge.

Degree of a vertex the degree of a vertex v, $d(v)$ is the number of edges incident to it.

Walk a walk on a graph is an alternating series of vertices and edges in which every edge is incident to the two vertices immediately preceding and following it.

Path a path is a walk in which no vertex is repeated.

Cycle a cycle is a path in which the first and last vertices are the same.

Acyclic a graph is acyclic if it has no cycles.

Connectedness we have two definition depending if the graph is directed or undirected:

- An undirected graph is connected if every vertex is reachable from all other vertices.
- A directed graph is *strongly connected* if every vertex is reachable from all other vertices. We use the word *strongly* because some nodes may have edges connecting to it but not going out of it.

We call each connected part of the graph a **connected component** of G. In *undirected graphs* this is an equivalence relation on the set of vertices.

Bipartite graph a bipartite graph is an undirected graph $G = (V, E)$ where V can be partitioned into two sets V_1, V_2 such that every edge $(u, v) \in E$ is such that $u \in V_1$ and $v \in V_2$. This means that there are no edges between vertices in the same set.

Tree a tree is a connected acyclic undirected graph, or a connected forest. In every tree it is possible to identify a node a the root, every node can be seen as the root of the tree. We call descendant of a node v every node that is reachable from v going downwards and ancestor every node that is reachable from v going upwards.

Forest a forest is an acyclic undirected graph, or a union of trees.

6.2 Memory representation

6.2.1 Adjacency matrix

Definition (adjacency matrix): Let $G = (V, E)$ be a graph with $|V| = n$.

An **adjacency matrix** is a $|V| \times |V|$ matrix A where

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{if } (i, j) \notin E \end{cases}$$

We want to represent the following graph using an adjacency matrix:

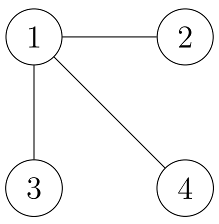

Figure 3: An example graph

We can represent this graph using the following adjacency matrix:

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{array}$$

Figure 4: The adjacency matrix of the graph in (3)

Using this representation we can easily check if two vertices are connected ($O(1)$) but we need a lot of space to store it ($O(|V|^2)$).

Adjacency matrices can be useful can be useful since we can perform linear algebra magic on them (e.g.: compute the k power, or find the eigenvalues) to find some properties of the graph.

6.2.2 Adjacency list

This is a more space efficient way to represent a graph, especially if the graph is sparse (meaning it has few edges and a lot of nodes).

Definition (adjacency list): Let $G = (V, E)$ be a graph with $|V| = n$.

An **adjacency list** is a list of n lists where the i-th list contains all the vertices adjacent to the i-th vertex.

Example: The graph in (3) can be represented using the following adjacency list:

Vertex	Adjacent vertices
1	2, 3, 4
2	1
3	1
4	1

Figure 5: The adjacency list of the graph in (3)

This in practice can be implemented using linked lists: in this case we have a space complexity of $O(|V| + |E|)$ while the time complexity to find if a vertex is connected to another one is $O(|E|)$.

(Is really linked lists the best way to implement this? What about other data structures?)

6.2.3 Summerizing

We can use the following table to summarize the two representations:

	space	$(u, v) \in E$	all neighbors
matrix	$O(\lvert V\rvert^2)$	$O(1)$	$O(\lvert V\rvert)$
list	$O(\lvert V\rvert + \lvert E\rvert)$	$O(d(v))$	$O(d(v))$

Table 1: Comparison of graph representations

7 Depth First Search (DFS)

In this section we will look at algorithms to explore a graph.

7.1 Definition

We setup the algorithm like this:

```python
# These are the variables we will be using:
clock = 1
visited = {}
pre = {}
post = {}

def setup(G):
    clock = 1
    for v in G.vertices():
        visited[v] = False
        pre[v] = None  # time of first visit of `v`
        post[v] = None # time of last visit of `v`
```

The actual algorithm is as follows:

```python
def explore(G, v):
    visited[v] = True
    pre[v] = clock
    clock += 1
    for (u, v) in v.edges():
        if not visited[u]:
            explore(G, u)
    post[v] = clock
    clock += 1
```

Then we call `explore` on any vertex of G and let the function traverse the graph.

7.2 Proof of correctness

We can now state the algorithm formally:

Statement (DFS): Let $G = (V, E)$ be a graph and $v \in V$. The algorithm `explore(G, v)` visits all the vertices of G that are reachable from v.

This is equivalent to

Statement (DFS): v can reach $u \iff$ `visited[u]` is `True` after completing `explore(G, v)`.

Proof. We prove each implication separately:

- $\Longleftarrow$ Since we started from `visited` containing only `False` values, the only way `visited[u]` can be `True` is if u was part of the exploration of some other node w. If $W = v$ we are done, otherwise we can repeat the argument for w. We know this procedure will terminate since the graph is finite.

- $\Longrightarrow$ We prove this by contradiction. Suppose that u is reachable from v and that there is a run of `explore` that misses u. Since u is reachable from v there is a path from v to u. Let w be the first vertex on this path that is not visited by the run of `explore` and let z be the vertex that precedes w. This is a contradiction, by the way the algorithm is defined, since w is a neighbor of z and it is not visited we visit it.

□

7.2.1 More than one connected component

By the definition of the algorithm we have that it doesn't work if we have a graph with more than 1 connected component.

We can complete the algorithm to take care of this case too:

```python
def DFS(G):
    # Setup
    clock = 1
    visited = {}
    pre = {}
    post = {}

    for v in G.vertices():
        visited[v] = False
        pre[v] = None
        post[v] = None

    # Run the algorithm for all nodes.
    # This will actually call `explore` just once
    # per connected component.
    for v in G.vertices():
        if not visited[v]:
            explore(G, v)
```

7.3 Time complexity

We have that the setup part is $O(|V|)$, since we are looping through all nodes.

For the actual `explore` part we have at least $O(|V|)$ since for each node we have to set `visited` to `True`, but we have to take in account the time to visit all the edges of the graph: each edge is visited exactly twice.

Combining the two parts we have that the time complexity of the algorithm is $O(|V| + |E|)$.

7.4 About the pre and post variables

Remark (pre and post): For every node v we have that `pre[v]` $<$ `post[v]`.

Proof. This is obvious from the definition of the algorithm. □

Theorem (pre and post intervals): For every node v and u, the intervals $[\text{pre}(v), \text{post}(v)]$ and $[\text{pre}(u), \text{post}(u)]$ are either disjoint or one is contained in the other.

Proof. We have two cases:

1. $\text{pre}(v) < \text{post}(u)$, hence $\text{pre}(u) < \text{pre}(v) < \text{post}(u)$. This means that v is a descendant of u, thus the exploration of v must have ended before the exploration of u. We have that $\text{pre}(u) < \text{pre}(v) < \text{pre}(u) < \text{post}(v)$, hence the intervals are contained in one another.
2. $\text{pre}(u) > \text{post}(v)$, hence $\text{pre}(u) < \text{post}(u) < \text{pre}(v) < \text{post}(v)$ and the intervals are disjoint.

□

7.5 Counting connected components

We can modify the algorithm to count the number of connected components in the graph.

```python
def DFS(G):
    clock = 1
    visited = {}
    pre = {}
    post = {}
    cc = {}

    for v in G.vertices():
        visited[v] = False
        pre[v] = None
        post[v] = None
        cc[v] = None

    connected_component = 0

    for v in G.vertices():
        if not visited[v]:
            connected_component += 1
            explore(G, v)


def explore(G, v):
    cc[v] = connected_component
    # The rest of the algorithm is the same
```

In this way, at the end of the procedure the variable `connected_component` will contain the number of connected components in the graph and the variable `cc` will contain the connected component of each node.

8 DFS on directed graphs

8.1 DFS tree

We can introduce a new variable, initialized as

```
    parent = {}
    for v in G.vertices():
        parent[v] = None
```

When exploring the graph we set this variable to the node that called the current one. In this way we can build a tree that represents the exploration of the graph, this tree is called the **DFS tree**.

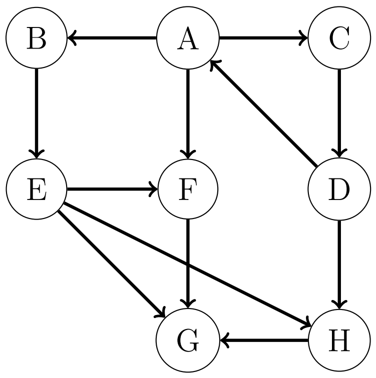

Figure 6: An example of a directed graph

By looking at the graph in (6) and performing a DFS in our head we can see that, starting from A, we can reach all the other nodes. We also note that this is not true for every node, for example G cannot reach any other node.

If we construct the DFS tree of this graph starting from A and breaking ties in alphabetical order we get the following tree:

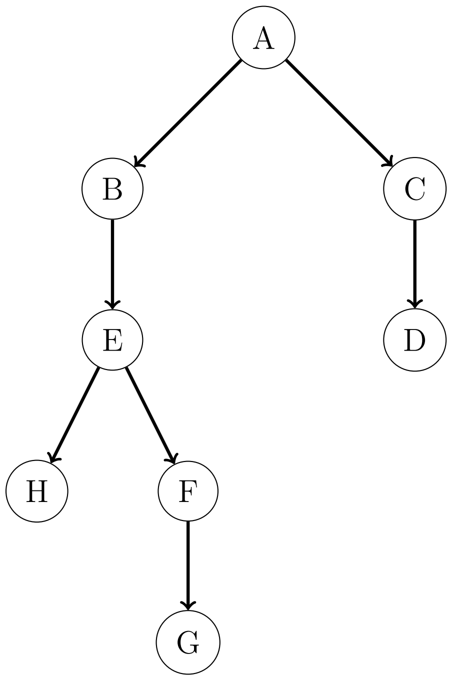

Figure 7: The DFS tree of the graph in (6)

We can define the following for DFS trees:

Forward edge an edge of the original graph that goes from a node to one of its descendants in the DFS tree.

Back edge an edge of the original graph that goes from a node to one of its ancestors in the DFS tree.

Cross edge an edge of the original graph that goes from a node to a node that is neither an ancestor nor a descendant.

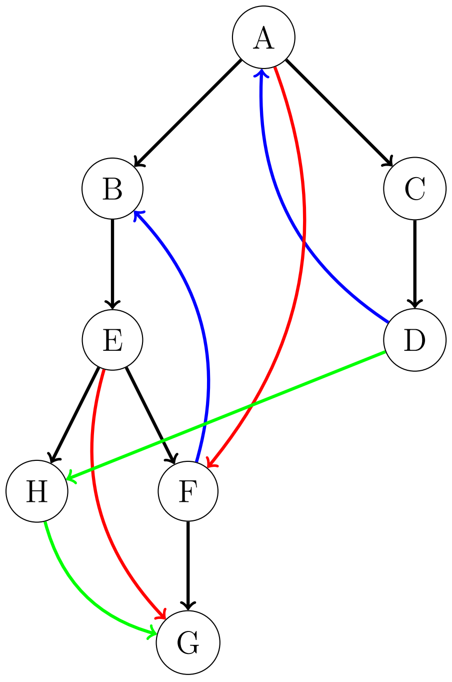

Figure 8: The DFS tree of the graph in (6) with forward (red), back (blue), and cross edges (green)

We note that if we consider the pre and post variable for each node as an “interval”, we can see that nodes connected with a forward edge have the interval of the children contained into the interval of the parent, nodes connected with a back edge have the interval of the parent contained in the interval of the children, and nodes connected with a cross edge have disjoint intervals.

8.2 Theorems about edges

8.2.1 Edges in undirected graphs

<u>**Statement**</u>: In a DFS of an undirected graph, every edge is either a tree edge or a back edge.

Proof. Let $\{u, v\}$ be an edge of G and suppose that $\text{pre}(u) < \text{pre}(v)$, hence u is discovered before v.

We have two cases:

- If the first time the search explores $\{u, v\}$ is in the direction from u to v then v is undiscovered until that time. Then the edge $\{u, v\}$ is a tree edge.
- If the first time the search explores $\{u, v\}$ is in the direction from v to u then v is already discovered. Then the edge $\{u, v\}$ is a back edge.

□

8.2.2 Cycles and back edges

<u>**Statement**</u>: A directed graph has a cycle $\iff$ a DFS of the graph has at least one back edge.

Proof. We prove each implication separately:

$\Longleftarrow$ If (u, v) is a back edge then there is a cycle consisting of the edge (u, v) and the path from v to u in the DFS tree. We know such path exists since to have a back edge we have to get to v with u already being discovered.

$\Longrightarrow$ If G has a cycle, then we can write it as $v_0 \to v_1 \to \cdots \to v_k \to v_0$. Let v_i be the first vertex in the cycle that is discovered. When we enter the cycle all the other nodes of the cycle are undiscovered since v_i is the first one by definition. Since DFS visits all the "legal" edges the algorithm will visit all edges of the cycle, hence eventually it will reach the edge (v_{i-1}, v_i), which is a back edge.

□

9 Topological sort

Definition (topological sort): If $G = (V, E)$ is a directed graph we say that an ordered list L of the vertices in V is a topological sort of G if for every edge $(u, v) \in E$, the vertex u comes before v in L.

Example: Consider the graph in Figure (6). A topological sort of this graph is $[A, C, D, B, E, F, H, G]$. Note that topological sorts are not unique.

9.1 Cycles and topological sorts

9.1.1 Theorem: cycle implies no topological sort

Statement: If a directed graph has a cycle then it has no topological sorts.

Proof. By contradiction, we assume that G has a topological sort L. Let $v_0, v_1, \dots, v_k$ be the cycle in G. Every node v_i in the cycle has a predecessor u, such that $(v_i, u) \in E$. Let $\overline{v}$ be the first node in the cycle that is in L. Since $\overline{u}$ is a predecessor of $\overline{v}$, $\overline{u}$ comes before $\overline{v}$ in L, but this is a contradiction since $\overline{v}$ comes before $\overline{u}$ in the cycle. □

9.1.2 Lemma: incoming edges and cycles

Statement: If $G = (V, E)$ is a directed graph in which every vertex has at least one incoming edge, then G has at least one cycle.

Proof. Take any vertex v in V and call it v_{n+1}, where $n = |V|$.

Since every vertex has at least one incoming edge, there exists a vertex v_n such that $(x, v_{n+1}) \in E$.

We repeat this process until we reach a vertex v_1, which will have an outgoing edge to some other vertex $\overline{v}$ that we have already visited. □

9.1.3 Theorem: no cycle implies topological sort

Statement: If a directed graph G doesn't contain a cycle, then it has at least one topological sort.

Proof. We will prove this by constructing an algorithm that will find a topological sort of G.

```
def find_topological_sort(G):
    L = []
    V = G.vertices()
    E = G.edges()
    while len(V) > 0:
        v = V.get_vertex_with_no_incoming_edges()
        L.append(v)
        V.remove(v)
        E.remove_all_edges_from(v)
    return L
```

The idea is that if a vertex has no incoming edges then it is correct to append it to the list.

This algorithm will work since we will always be able to get a vertex with no incoming edges since, if we couldn't, we would have a cycle (by the lemma above).

□

9.2 Topological sorts from DFS

9.2.1 Theorem: DFS and topological sorts in DAGs

Consider the following the following algorithm:

```
visited = {}
L = []

def explore(G,v):
    visited[v] = True
    for (u, v) in v.edges():
        if not visited[u]:
            explore(G, u)
    L.append(v)
```

We want to prove that the list L will be a topological sort of the graph G by the end of the algorithm.

Now we formally state and prove this.

Statement: $G = (V, E)$ is a directed acyclic graph (DAG). If we run the algorithm `DFS(G)`, then for every edge $(u, v) \in E$, the call to `explore(G, u)` will end after the call to `explore(G, v)`.

Proof. We look at the state of the algorithm when the edge (u, v) is explored. The other vertices can be in one of the following states:

1. not yet visited
2. visited and in `L`
3. visited but not in `L` because the exploration of the vertex has not ended yet

We note that when we explore u, v will be in state 1 or 2, not 3. This is because if v is in state 3 then there exist a path from v to u and we have a cycle, which is a contradiction.

Now we consider the two possible cases separately:

1. Then we are about to explore from u. v is a neighbor of u and it is unexplored yet so we will explore it. The exploration of v will end before the exploration of u.
2. Then the exploration of v has already ended. This means that the exploration of u will end after the exploration of v.

In both cases we have our result. □

Note that this theorem only works for DAGs, if we have a graph with cycles then the list L may not be a topological sort.

10 Strongly connected components (SCC)

10.0.1 Reverse of a graph

Definition (reverse): Given a directed graph $G = (V, E)$, the reverse of G is the graph $G^R = (V, E^R)$ where $(u, v) \in E^R \iff (v, u) \in E$.

10.1 Theorems about SCCs

10.1.1 Theorem: component graphs are DAGs

Definition (component graph): Let G be a directed graph, then the component graph G' is obtained by grouping all the vertices of a connected components of G.

Statement: If G is a directed graph and G' is its component graph, then G' is a directed acyclic graph.

Proof. We prove this by contradiction.

Suppose that G' has a cycle $C_1, \dots, C_n$. This means that it is possible for any $v \in C_i$ to reach any $u \in C_{i+1}$. This means that the two components should be merged, which is a contradiction. □

10.1.2 Theorem: strongly connected components in the reverse

Statement: Let $G = (V, E)$ be a directed graph and $G^R = (V, E^R)$ its reverse. Then the connected components of G^R are the strongly connected components of G.

Proof. If we consider G^R we have that the connected components stay connected in the reverse, and they don't get connected to other components since G' is a DAG.

Note to reader: I don't think this is a formal enough proof. If you come up with a better one please let me know. □

10.1.3 Theorem: post numbers in SCCs

Statement: If C, C' are two strongly connected components (SCC) of a directed graph G and there is an edge from a node in C to a node in C', then the highest `post` number in C is greater than the highest `post` number in C'.

Proof. If DFS visits the component C before C' then all of C' will be visited before we exit C.

If DFS visits C' before C then C will not be visited until we exit C'. □

10.1.4 Definitions: source and sink

Definition:

- Source is a vertex with no incoming edges.
- Sink is a vertex with no outgoing edges.

Note that the source will have the highest `post` number.

Reversing the graph will transform the sources into sinks and vice versa.

10.2 Finding SCCs

To find SCCs we can use the following algorithm:

```
# Consider an explore function that
# returns a list of the vertices explored

def find_SCCs(G):
    V = G.vertices()
    SCCs = {}
    while len(V) > 0:
        G_R = G.reverse()
        explored_nodes_of_reverse = explore(G_R, G_R.random_vertex())
        sink = explored_nodes_of_reverse.get_vertex_with_highest_post()

        # These nodes are all part of the same SCC
        explored_nodes = explore(G, sink)
        SCCs.add(explored_nodes)
        V.remove_all(explored_nodes)

return SCCs
```

11 Breadth First Search (BFS)

For this class we had to hand in an assignment, you can find my solution in `cs_assignment_1.pdf`.

11.1 Introduction

The problem we are going to look at now is how to get from one vertex to another in a graph passing through the fewest number of edges.

It's easy to see that DFS is not the right tool for this problem. We need to find a different algorithm.

Definition (shortest path): Let $G = (V, E)$ and $s \in V$ be the start vertex. The shortest path distance $d(v)$ from s to $v \in V$ is the minimum number of edges in any path from s to v. If there is no path from s to v then $d(v) = \infty$.

11.2 Definition of BFS

This is the algorithm we will use to find the shortest path.

It works in the following way:

```
def BFS(G, s):
    distance = {}
    for v in G.vertices():
        distance[v] = float('inf')

    distance[s] = 0
    queue = Queue()
    queue.enqueue(s)

    while not queue.is_empty():
        v = queue.dequeue()
        for u in v.neighbors():
            if distance[u] == float('inf'):
                distance[u] = distance[v] + 1
                queue.enqueue(u)
```

You can watch the algorithm in action here: `https://www.cs.usfca.edu/~galles/visualization/BFS.html`.

11.2.1 Tools: queues

For this algorithm we use a `Queue` data structure.

Queues are similar to arrays and have the following operations:

- `enqueue(x)`: add an element to the end of the queue
- `dequeue()`: remove the first element of the queue and returns it

Sometimes `enqueue` is called `inject` and `dequeue` is called `eject`.

We say that queues are *FIFO* (First In First Out) data structures.

We will look at the implementation of a queue later.

11.3 Proof of correctness

Theorem (correctness of BFS): Let $G = (V, E)$ be a graph and $s \in V$. Assume the range of distances d from s to all other vertices is $[0, M]$. For each $t \in [0, M]$ there is an iteration of the algorithm such that:

1. all the vertices $v \in$ `queue` have $d[v] = t$
2. all the vertices u that have been removed from `queue` have $d[u]$ equal to the correct distance from s to u
3. For all vertices u that have been removed from `queue` we have that $d[u] \leq t - 1$

Note that since the conditions (1), (2), and (3) have to be true for any t we call them *invariants.*

Proof. We will prove this by induction on t.

Base case $t = 0$. This is trivially true since the only vertex in the queue is s and $d[s] = 0$.

Inductive step We assume that the statements are true for $t = \tau$.

At this time, some vertices have been already removed from `queue`, others are still in it, and some have yet to be explored.

At this point of time we know the following facts:

- By (3), all vertices removed from `queue` have $d[u] \leq \tau - 1$
- By (2) the distances are correct.
- By (1), all vertices in `queue` have $d[v] = \tau$.

At the next round $\tau + 1$, BFS will go through as many iterations as the number of vertices in `queue`: the algorithm will remove those vertices. Note that these vertices vertices have all distance τ and their distance is no longer going to change. This implies that (2) and (3) are true.

Moreover, all vertices not yet seen that are neighbor of the ones that were in `queue` at τ have their distances set to $\tau + 1$, thus (1) is true.

□

11.4 Time complexity

We split the algorithm in parts:

- The setup part of the algorithm is $O(|V|)$.
- We check whether the queue is empty $|V|+1$ times, hence $O(|V|)$.
- We insert a vertex in the queue based on its neighbors, hence $O(|E|)$.
- We enqueue and dequeue every vertex in the graph at most once, hence $O(|V|)$.

Combining all the steps we get a time complexity of $O(|V|+|E|)$.

Note that we need to take in account the time for queue operations, which depend on the implementation of the queue itself.

11.4.1 Time complexity of queue operations - basic

For the sake of this course its enough to know that queue both `enqueue` and `dequeue` are $O(1)$.

Moreover queues are usually implemented using a linked list, but if you know already the maximum size of the queue you can also use an array which is slightly more efficient.

11.5 Implementation of a queue - optional

Warning: Advanced stuff ahead

This is not part of the course, but if you are curious you can read on.

The following code snippets are written in C.

11.5.1 Linked list implementation

We will implement a queue using a linked list as underlying data structure.

```
struct node {
    int data;
    struct node *next;
};

struct queue {
    struct node *head;
    struct node *tail;
};

struct queue *create_queue() {
    struct queue *q = malloc(sizeof(struct queue));
    q->head = NULL;
    q->tail = NULL;
    return q;
}

void enqueue(struct queue *q, int x) {
    struct node *n = malloc(sizeof(struct node));
```

```
        n->data = x;
        n->next = NULL;
        if (q->head == NULL) {
            q->head = n;
            q->tail = n;
        } else {
            q->tail->next = n;
            q->tail = n;
        }
    }

    int dequeue(struct queue *q) {
        int x = q->head->data;
        struct node *n = q->head;
        q->head = q->head->next;
        free(n);
        return x;
    }
```

We see that each of the operations do not depend on the length of the queue, hence they are both $O(1)$.

Note that in this implementation we have to `malloc` each node (which is a slow operation) and dereference a pointer (which can also be slow).

11.5.2 Array implementation

If we know the maximum size of the queue we can use an array to implement it, which will make the operations even faster.

The idea is that we keep a pointer of the first and last element of the queue and we use an array to store the elements.

When we enqueue an element we add it to the end of the array and we increment the pointer to the last element. When we dequeue an element we return the first element of the array and we increment the pointer to the first element.

If we run out of space at the end of the array we can just cycle back to the beginning since, by knowing the maximum size of the queue, we are sure that the first elements are not going to be used anymore at the time we have to cycle back to the start.

```
    struct queue {
        int *data;
        off_t head;
        off_t tail;
        size_t size;
    };

    struct queue *create_queue(size_t size) {
        struct queue *q = malloc(sizeof(struct queue));
        q->data = malloc(size * sizeof(int));
        q->head = 0;
        q->tail = 0;
        q->size = size;
        return q;
```

```
}

void enqueue(struct queue *q, int x) {
    q->data[q->tail] = x;
    q->tail = (q->tail + 1) % q->size;
}

int dequeue(struct queue *q) {
    if (q->head == q->tail) {
        // The queue is empty
        return -1;
    }

    int x = q->data[q->head];
    q->head = (q->head + 1) % q->size;
    return x;
}
```

We see that the operations are still $O(1)$ but this time we `malloc` only once and we don't have to dereference any pointers.

12 Dijkstra's algorithm

We now consider a graph $G = (V, E)$ and a function $l(u, v) > 0$ that assigns a length to each edge $(u, v) \in E$.

Today's problem is to find the shortest path from a vertex $s \in V$ to any other vertex taking into account the length of the edges.

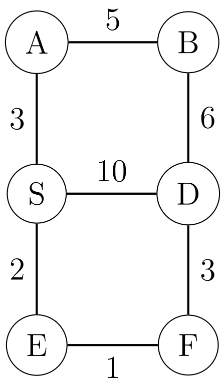

Figure 9: An example of a graph with weighted edges

We have seen that BFS does a similar job, but it would work only if all the edges had the same length. We need to find a different algorithm.

12.0.1 Naive approach

We can try to convert the problem into a shortest path problem with unweighted edges, which we know already how to solve.

For example, we could add new vertices in between each edge, one for each unit of length of the edge, and then connect them with edges of length 1.

It is easy to show that this approach would work and it would convert the problem into one we already solved but it is also highly inefficient.

12.0.2 Tools: priority queues

For this algorithm we will need a new data structure: the priority queue.

A priority queue works more or less like a normal queue but each element of the queue also has a priority.

When we dequeue an element we get the one with the highest priority first, no matter when it was enqueued.

An example of priority queue in real life is the triage in a hospital: the patients are not treated in the order they arrived but in the order of the severity of their condition.

12.1 Definition

Let $G(V, E)$, $s \in V$, and $l(u, v) > 0 \forall (u, v) \in E$. Let $d(v)$ be the length of the shortest path from s to v.

```python
def Dijkstra(G, s):
    d = {}

    # Setup
    for v in G.vertices():
        d[v] = float('inf')
    d[s] = 0

    # Insert all vertices in the priority queue
    Q = PriorityQueue(G.vertices())
    while not Q.is_empty():

        # Get the vertex with the smallest distance
        u = Q.dequeue()
        for v in u.neighbors():
            # We check if we can reach v in a shorter path
            if d[v] > d[u] + l(u, v):
                # If we can, we update the distance of v
                d[v] = d[u] + l(u, v)

                # Save the newfound distance of v in the queue
                # by updating its priority
                Q.update_priority(v, d[v])
```

12.1.1 Example

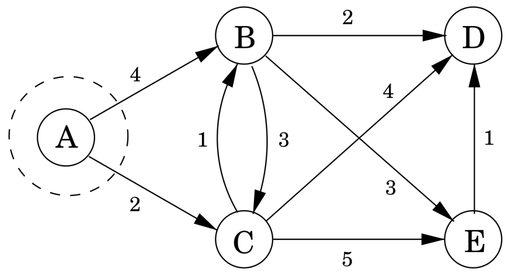

Figure 10: An example of a directed graph with weighted edges

In the graph in Figure (10) we would start from A which is the start vertex and we apply the algorithm.

We can make a table of the distances at each step of the algorithm:

Extracted vertex	A	B	C	D	E
Setup	0	∞	∞	∞	∞
A	0	**4**	**2**	∞	∞
C	0	**3**	2	**6**	**7**
B	0	3	2	**5**	**6**
D	0	3	2	5	6
E	0	3	2	5	6

Table 2: The distances at each step of the algorithm

In the Table (2) we have highlighted the vertices that have been updated.

At each step we extract the vertex with the smallest distance from the queue and we update the distances of its neighbors if we can reach them in a shorter path.

12.1.2 Negative edges

Sometimes having a negative length edge can be useful, therefore we need to adjust the algorithm to take them into account.

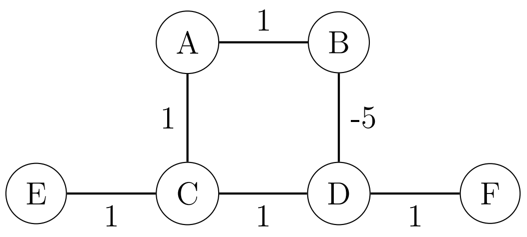

Figure 11: An example of a graph with negative edges

In the graph in Figure (11) we have a cycle containing a negative edge, which means that we can go around the cycle as many times as we want and we will always decrease the distance.

This is a problem which is not solvable with Dijkstra's algorithm, a graph with **negative cycles** is not solvable with this algorithm.

Moreover, even if there was no negative cycle but still some negative edges, the algorithm would still not work correctly because it wouldn't be able to find the shortest path.

Despite this flaw we can still use Dijkstra's algorithm in most cases, as long as there are no negative edges.

12.2 Time complexity

We analyze each step of the algorithm:

- The setup part of the algorithm is done once
- The check to determine whether the queue is empty is done $|V|$ times
- Finding the right vertex to dequeue is done $|V|$ times
- Updating the priority of the neighbors of the dequeued vertex is done $|E|$ times

Then, we have to take into account the time complexity of the priority queue operations which vary depending on the implementation.

12.2.1 With a linked list

This is the simplest implementation and the original one that Dijkstra used.

Operation	**Time complexity**
Init	$O(n)$
Check empty	$O(1)$
Find min and remove	$O(n)$
Update priority	$O(1)$

Table 3: Time complexity of the operations of a priority queue implemented with a linked list

Note that these operations have these time complexities only if applied to this specific algorithm, in general they can be different.

12.2.2 With a binary heap

First we need to define a *full binary tree*.

A full binary tree is a rooted tree where each note has two children (left and right) or no children at all, except possibly on the bottom layer.

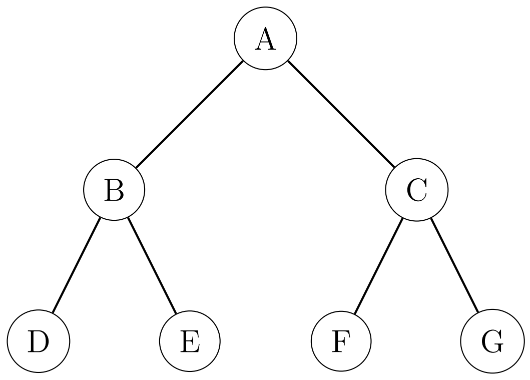

Figure 12: An example of a full binary tree

If the number of nodes is different than $2^h - 1$ for some h then we start filling the tree from the left and we say that the tree is not full.

We can navigate the tree as follows:

- From a parent of index i we can get to the left child at index $2i$ and to the right child at index $2i + 1$.
- From a child at index i we can get to the parent at index $\lfloor i/2 \rfloor$.

Therefore this data structure can be implemented using an array and navigated using the above formulas.

We can now introduce a minimum binary heap or **min-heap**: this is a binary heap where the value of every node is less than or equal to the value of its children.

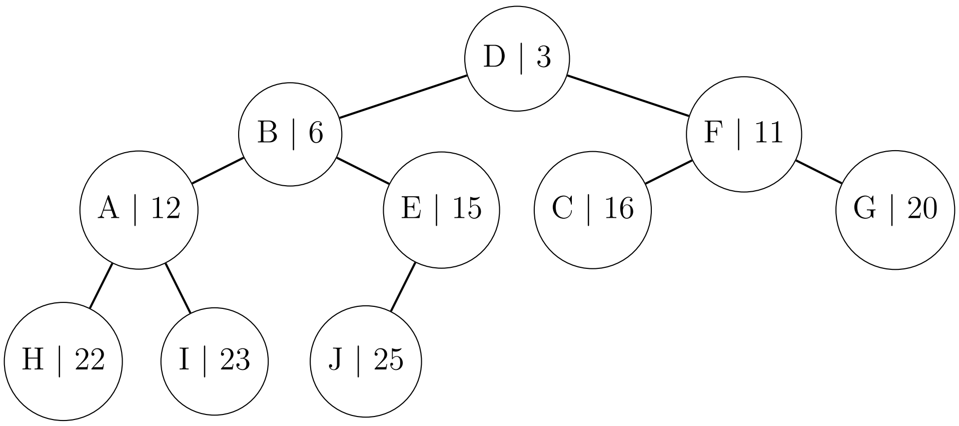

Figure 13: An example of a min-heap

where the letter is the value of the node and the number is the priority.

We can now compile a table with the time complexity of each operation of a min-heap.

Operation	**Time complexity**	**Comment**
Init	$O(n \log n)$	We need to sort the list first
Check empty	$O(1)$	We can just check if the array is empty
Remove minimum	$O(\log n)$	We remove the root and heapify the tree
Update priority	$O(\log n)$	We need to heapify the tree

Table 4: Time complexity of the operations of a min-heap

Heapifying the tree means to restore the property of the min-heap after we removed the root or updated the priority of a node.

- When we remove the root we swap it with the last node of the tree, then we swap the new root with its smallest child and we keep doing this until the tree is a min-heap again. In this process we are doing $O(\log n)$ operations.
- When we update a priority we just need to swap the node with its parent until the tree is a min-heap again, which is also $O(\log n)$.

12.2.3 Summarizing

Operation	**Times**	**L.L**	**L.L tot.**	**B.H.**	**B.H. tot.**
Init	1	$O(\lvert V\rvert)$	$O(\lvert V\rvert)$	$O(\lvert V\rvert \log \lvert V\rvert)$	$O(\lvert V\rvert \log \lvert V\rvert)$
Check empty	$\lvert V\rvert$	$O(1)$	$O(\lvert V\rvert)$	$O(1)$	$O(\lvert V\rvert)$
Find min and remove	$\lvert V\rvert$	$O(\lvert V\rvert)$	$O(\lvert V\rvert^2)$	$O(\log \lvert V\rvert)$	$O(\lvert V\rvert \log \lvert V\rvert)$
Update priority	$\lvert E\rvert$	$O(1)$	$O(\lvert E\rvert)$	$O(\log \lvert V\rvert)$	$O(\lvert E\rvert \log \lvert V\rvert)$

Table 5: Time complexity of the operations of Dijkstra's algorithm with a linked list (L.L) and a binary heap (B.H.)

Summing up the time complexities we find

- The linked list implementation has a total time complexity of $O(|V| + |V|^2 + |E|)$
- The binary heap implementation has a total time complexity of $O(|V| \log |V| + |E| \log |V|)$.

12.3 Proof of correctness

This is a very long and complex proof, it will not be asked during the exam, it was done in class to show how to approach proofs like this.

To make the proof easier we will use the following definitions:

- At the end of each while loop, the nodes that have been removed from Q are *black*, and the nodes still in Q are *white*.
- An *all-black* path from s to v is a path where each node, except possibly v, is black.

Statement: the following invariants hold:

i. All the black nodes v have a $d(v)$ equal to the length of the shortest path from s to v or ∞ if such path does not exist.

ii. All nodes v have $d(v)$ equal to the length of the shortest *all-black* path from s to v or ∞ if such path does not exist.

Moreover, at the end of the execution, all nodes are black, and the invariants imply that $d(v)$ is the length of the shortest path from s to v for all $v \in V$, that is, the algorithm is correct.

Proof. Before we start we make the following observations:

- At all the times $\forall v \in V$ if $d(v)$ is not ∞ then there is an all-black path from s to v of length $d(v)$.
- $\forall v \in V$, $d(v)$ is the only decreased from step to step.
- If at some point $d(v)$ equals to the length of the shortest path from s to v then $d(v)$ will never change again.

These are all obvious just by looking at the code.

Then, we proceed by induction on the number of iterations of the while loop t.

Base case At the beginning of the algorithm, s is the only black node and $d(s) = 0$.

Indeed the shortest path from s to s is of length 0, therefore (i) is true. Moreover, for all $v \in V$ if $v = s$ then $d(v) = 0$ and (ii) is true; if $v \neq s$ then either it is a neighbor of s and $d(v) = l(s, v)$ or $d(v) = \infty$ which is again correct.

Inductive step 1 In this step we assume that (i), (ii) are true at t and we want to show that (i) at $t+1$. Let v be the vertex that is removed from the queue during the $t+1$-th iteration. We need to argue that $d(v)$ is the length of the shortest path from s to v.

Notice that the algorithm does not change $d(v)$ after v is removed from the queue, therefore we just need to show that at time t $d(v)$ is the length of the shortest path from s to v.

We argue by contradiction: suppose that at time t there exists a path from s to v of length $L < d(v)$. By (ii) this path must contain a white vertex, let u be the first white vertex along such path. Then, the length of this path up to u is $\geq d(u)$ since (ii) guarantees that $d(u)$ is the shortest path from s to u. Moreover the length of this path up to u is $\leq L$ since u is a vertex along that path. Combining all these facts we get

$$d(u) \leq L < d(v)$$

which is a contradiction of the algorithm itself, since v was chosen to be the white vertex with the smallest $d(.)$ at the end of the t-th iteration.

Inductive step 2 In this step we assume that (i), (ii) are true at t and (i) is true at $t+1$ (as we have just shown) and we want to show that (ii) is true at $t+1$. Let v be the vertex that is removed from the queue during the $t+1$-th iteration and denote $d_t(\cdot)$ the value of $d(\cdot)$ at time t and $d_{t+1}(\cdot)$ the value of $d(\cdot)$ at time $t+1$.

At time t every vertex x is in one of the following states:

1. x is black
2. $x = v$ (and therefore is white)
3. x is white and $x \neq v$

If we are in case (1) then $d_t(x)$ was the length of the shortest path from s to x at time t (because of (i)) and $d_t(x) = d_{t+1}(x)$. Then, by (ii) at t, there is an *all-black* path from s to x of length $d_t(x)$ and this path is still valid at time $t+1$ since the distance is the same. Then (ii) is true at $t+1$.

If we are in case (2) the proof is very similar to the case of (1). This is because we also have that $d_t(v) = d_{t+1}(v)$ and, since v is turning black at this iteration, by (i) at $t+1$ $d_{t+1}(v)$ is the length of the shortest path from s to v. Then by (ii) at time t there is an *all-black* path from s to v of length $d_t(v)$ and this path is still valid at time $t+1$ since the distance is the same. Then (ii) is true at $t+1$.

If we are in case (3) we will argue by contradiction and suppose that there is an *all-black* path P from s to x of length $L < d_{t+1}(x)$. We have 3 cases:

a. P does not contain v. But then

$$L < d_{t+1}(x) \leq d_t(x)$$

because if there is an all-black path at $t+1$ and $x \neq v$ then there is also one at t. But this is a contradiction since, because of (ii) at t, $d_t(x)$ is the length of the shortest all-black path from s to x. Therefore such path P cannot exist, and (ii) is true at $t+1$.

b. P contains v as the last intermediate vertex before x. Then, the length of the path from s to v is

$$L_v = L - l(v, x)$$

But $d_t(v) \leq L_v$ since, because of (ii) at t, $d_t(v)$ was already the length of the shortest path from s to v, and we get that

$$d_t(v) + l(v, x) \leq L$$

Now the algorithm updates $d_{t+1}(x)$ to some value $L' \leq d_t(v) + l(v, x)$. Combining the assumptions with all the statements we get

$$d_t(v) + l(v, x) \leq L < d_{t+1} \leq d_t(v) + l(v, x)$$

which is a contradiction.

c. The path from s to x contains v as an intermediate vertex but not as the last one. Let y be the last intermediate vertex before x. So that P is made of a path from s to v, a path from v to y, and the edge (y, x).

   We have that y must be black (since we are considering *all-black* paths), therefore y must have been taken out of the queue at some time $t' \leq t$. We have

$$d_{t'}(y) + l(y, x) \leq L$$

   since, by (ii) at t', $d_{t'}(y)$ is the length of the shortest *all-black* from s to y. Moreover, by the definition of algorithm, distances can only get shorter, therefore

$$d_{t+1}(x) \leq d_{t'}(x) \leq d_{t'}(y) + l(y, x) \leq L$$

   but by assumption $L < d_{t+1}(x)$, which is a contradiction.

□

13 Kruskal's algorithm

13.1 Minimum spanning tree

Definition (minimum spanning tree): Given an undirected connected graph $G = (V, E)$ and a function $c : E \to \mathbb{N}^+$ that assigns a cost to each edge. A minimum (weight) spanning tree is a subset $S \subseteq E$ whose total cost is as small as possible that still connects all the vertices.

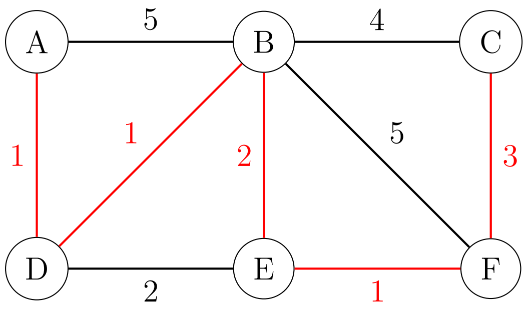

Figure 14: An example of a graph and (in red) one of its minimum spanning tree

13.2 Definition

This algorithm is used to find the minimum spanning tree of a graph.

Let $G(V, E)$ be a connected undirected graph and $c : E \to \mathbb{N}^+$ be a function that assigns a cost to each edge.

The algorithm works as follows:

```
def Kruskal(G, c):
    # A set of edges
    S = {}

    # Sort the edges by increasing cost
    sorted_edges = sort_edges(G, c)

    for e in sorted_edges:
        (u, v) = e

        # If `e` connects two previously disconnected components
        if not S.connects(u, v):
            S.add_edge(e)

    return S
```

Basically the algorithm starts from a graph without any edges and adds the edges one by one, always choosing the one with the smallest cost that connects a node that wasn't already connected.

Note that the check `if not S.connects(u, v)` is equivalent to saying "if adding this edge doesn't create a cycle". We will prove this later.

13.2.1 Example

Consider the graph in Figure (14).

The algorithm will connect the vertices starting from the cheapest ones.

Edge	(A, D)	(B, D)	(E, F)	(B, E)	(D, E)	(C, F)	(B, C)	(A, B)	(B, F)
Cost	1	1	1	2	2	3	4	5	5

Table 6: The edges of the graph in Figure (14) sorted by cost

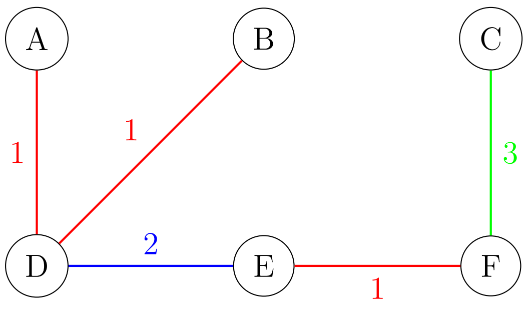

Figure 15: The minimum spanning tree of the graph in Figure (14). Red edges are added first, then blue, then green.

This type of algorithm is called a *greedy algorithm* because it always chooses the best option at each step. This is not always the best solution to the problem but in this case it gets the job done.

13.3 Theorems and proofs

13.3.1 Theorem: solutions are trees

Theorem: Any optimal solution to the problem of finding a minimum spanning tree is a tree (i.e. undirected, connected, and acyclic).

Proof. The graph we start with is already undirected and connected, we just need to show that the graph is acyclic.

We note that if a connected undirected graph contains a cycle, then any one of the edges in the cycle can be removed without disconnecting the graph.

We will argue by contradiction that minimum spanning trees contain a cycle. Then, as we saw, we can remove an edge from the cycle and the graph will still be connected while the total cost of the edges will be smaller (since $c > 0$).

This contradicts the assumption that the original tree was a minimum spanning tree, therefore we have that minimum spanning trees are acyclic. □

13.3.2 Theorem: the result set is connected

Theorem: The set of edges S is connected.

Proof. We will argue by contradiction that the set of edges S is divided in two or more connected components.

Since the original graph is connected, there must exist some edge e that connects two of these components. Then the algorithm (at some point) will add e to S and the two components will be connected.

We can repeat the same process for all the other connected components until we are left with just one. □

Remark: If we don't assume that the original graph is connected, then S will have as many connected components as the original graph.

13.3.3 Proof of correctness and optimality

We not only want to prove that the algorithm works, but also that it is the best solution possible.

This is quite a tricky thing to prove and we will proceed by saying that there exist some optimal solutions (that we don't know), but each step of our algorithm is also a step in some optimal solution. We can formalize this as follows:

Theorem: At every step, there is an optimal solution T^* that includes all the edges in S at that step.

Proof. We will proceed by induction on the number of iterations of the for loop t.

Temp assumption We will temporary assume that all the costs are distinct. This is just to make the proof easier, we will remove this assumption at the end of the proof.

Base case The base case is trivial, since at the beginning `S` is empty and there is an optimal solution that includes all the edges in S.

Inductive step Suppose at some iteration t the invariant is true, therefore there exists an optimal solution T^* that includes all the edges in $S \subseteq T^*$ at that step. Let $e = (u, v)$ be the edge added at the $t + 1$-th iteration. We now want to show that $S \cup \{e\} \subseteq T^*$.

If $e \in T^*$ there is nothing to prove. If $e \notin T^*$ then consider $T^* \cup \{e\}$: this graph must contain a cycle C. We will now prove that this cannot happen since it is a contradiction.

In C there must be a vertex that is in T^* and is not in S because otherwise the algorithm would not have added e because we already have a path that connects u and v. Moreover, let $u \in S$, then $v \notin S$ because otherwise the algorithm would not have added e.

This means that on the path $P \subset T^*$ that connects u and v there is an edge $e' = (x, y) \in C, T^*$ and $e' \neq e$ that connects $x \in S$ and $y \notin S$. We have that $c(e) < c(e') \ \ (*)$ because of the way we sorted the edges at the beginning of the algorithm. But this implies that $T^* \cup \{e\} \setminus \{e'\}$ is a better solution than T^*, which is a contradiction.

Thus e must be in T^* and the invariant is true at $t + 1$.

Removing temp assumption Note that in the inequality $(*)$ we used the fact that all the costs are distinct. If we didn't make this assumption we would have that $c(e) \leq c(e')$.

This is still fine because it just implies that there are more than one optimal solutions to the problem (for reference in our case T^* and $T^* \cup \{e\} \setminus \{e'\}$) but our algorithm is always one of them.

□

Remark: If at the end of the algorithm the above theorem is true, then the algorithm is optimal.

Proof. This is true by the definition of the algorithm itself, if $S \subsetneq T^*$ then the cost of T^* would be higher than the cost of S, but this is a contradiction of the fact that T^* is a minimum spanning tree. □

14 Prim's algorithm

This is another greedy algorithm that can be used to find the best path from one vertex to another.

This is very similar to Dijkstra's algorithm, but we mainly focus on the weight of the edges and not on the distance from the starting vertex.

14.1 Definition

Let $G(V, E)$ be a connected undirected graph, $w : E \to \mathbb{N}^+$ be a function that assigns a weight to each edge, and $s \in V$ be the starting vertex.

The algorithm works as follows:

```
def Prim(G, w, s):
    c = {} # Cost of the shorter path from s to v
    P = {} # Predecessors

    # Setup
    for v in G.vertices():
        c[v] = float('inf')
        P[v] = None

    c[s] = 0
    Q = PriorityQueue(G.vertices(), c)
    while not Q.is_empty():
        u = Q.dequeue()
```

```
            for v in u.neighbors():
                # In this algorithm we check the weight of the edge
                # not the distance from the starting vertex
                if c[v] > w(u, v):
                    c[v] = w(u, v)
                    P[v] = u # Set the predecessor of v to be u
                    Q.update_priority(v, c[v])
        return P
```

From the array of predecessors `P` we can reconstruct the minimum spanning tree of G.

14.2 Time complexity

The time complexity of this algorithm is the same as Dijkstra's algorithm, since the only difference is that we are checking the weight of the edges and not the distance from the starting vertex.

Therefore we have a complexity of $O(|V|\log|V| + |E|\log|V|)$.

15 Huffman coding

15.1 Introduction

15.1.1 Problem

Let Ω be a set of symbols (like the alphabet) and define a string to be a sequence of symbols from Ω.

Given a string we want to encode it as a sequence of bits such that

- The sequence can be efficiently decoded
- The sequence is as short as possible

15.1.2 Example

An example of encoding for the English alphabet is ASCII, which uses 8 bits for each character. This is a very simple encoding which is very fast to decode but it is not very efficient in terms of space and only supports 128 fixed-length characters.

(As a side note, nowadays the preferred encoding for text is UTF-8, which is a superset of ASCII and can store any Unicode character.)

Storing an ASCII string is not very space efficient, for example the letter `Q` (in Italian at least) has to be followed by a `U` or another `Q` and it is not a frequent letter. ASCII doesn't know about this, it just assigns 8 bits to each character, but we can exploit patterns in the strings we want to store to make the encoding more efficient.

15.1.3 Theorem: probability of a good encoding

Theorem: Let Ω be a set of symbols with $|\Omega| = 2^k$, and x be a string of length n over Ω chosen with a uniform probability from Ω^n. Then

$$P(\text{length(E(x))} \leq \text{kn} - \alpha) \leq 2^{-\alpha+1}$$

where $E(x)$ is the encoding of x.

Proof. There are 2^{kn} possible strings of length n over Ω. Each string x is chosen with a probability of 2^{-kn}. Let the length of some string be $t = kn - \alpha$.

The maximum number of strings that is possible to encode such that length(E(x)) $\leq$ t can't be more than the number of strings of length n over Ω because otherwise we would have two strings that are encoded in the same way, which is not possible (pigeonhole principle).

The number of strings of length less than or equal to t is

$$\sum_{i=0}^{t} 2^i = 2^{t+1} - 1 < 2^{t+1}$$

Then the probability that a string's encoding is less than or equal to t is

$$P(\cdot) \leq \frac{2^{t+1}}{2^{kn}} = \frac{2^{(kn-\alpha)+1}}{2^{kn}} = 2^{-\alpha+1}$$

□

This theorem says in a world of total randomness we will never have a good encoding because, for example, not all sequences of characters form a word.

15.1.4 Example

Suppose we create an encoding in which some characters are encoded with less bits than others.

Character	**Encoding**
A	0
B	01
C	11

Table 7: An example of encoding

If we need to decode a string such as `0111111` we encounter an issue: we don't know how to group characters, for example the first characters could either be a `B` or a `AC`. We would need first to decode the whole sequence and start decoding from the end, which is not efficient.

This problem arises because the encoding is not *prefix-free*, that is, no encoding of a character is a prefix of another encoding.

Character	**Encoding**
A	01
B	100
C	00
D	101
E	11

Table 8: An example of prefix-free encoding

15.2 The Huffman algorithm

Huffman found a method to create a prefix-free encodings.

We build a binary tree where the root is the empty string and each child is adds a 0 or a 1 to the parent's string. Then we choose an encoding for each character of Ω only using the leaves of the tree.

15.2.1 Example

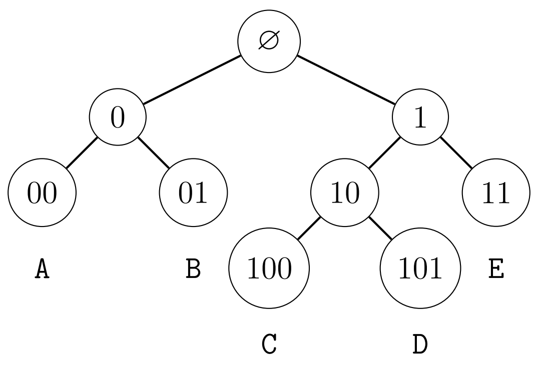

Figure 16: An example of a Huffman tree

This is just an example but we see that the encoding of each character is prefix-free.

Now we need to choose how to assign the encodings to the characters, ideally we want the most frequent characters to have the shortest encodings, that is to be closer to the root of the tree.

15.2.2 The algorithm

The idea of the algorithm is to start from the leaves, which are the characters of Ω, and merge the two nodes with the smallest frequency until we are left with just one node.

An implementation could look like this

```
class GraphNode:
    def __init__(self, frequency, character=None, children=[]):
        self.frequency = frequency
        self.character = character
        self.children = children

# frequency_map is an array of pairs (character, frequency)
def huffman_tree(frequency_map):
    (graph_nodes, frequency) = map(
        lambda x: (GraphNode(frequency=x[1], character=x[0], children=[]), x[1]),
        frequency_map,
    )

    Q = PriorityQueue(graph_nodes, frequency)

    while Q.size() > 1:
        u = Q.dequeue()
        v = Q.dequeue()
        w = GraphNode(children=[u, v], frequency=u.frequency + v.frequency)

        Q.enqueue(w)

    return Q.dequeue()
```

16 Set cover problem

16.1 Problem

Let Ω be a finite set and $S_1, S_2, \dots, S_n$ be subsets of Ω.

We want to find a finite sequence (i_n) up to k such that the union of $S_{i_1}, S_{i_2}, \dots, S_{i_k}$ is equal to Ω.

There is no known algorithm that can solve this problem efficiently, the only correct solution we know requires exponential time.

Although the optimal solution has not been achieved yet we can use a greedy algorithm to find a good-enough approximation.

16.2 The greedy algorithm

```
def set_cover_greedy(sets, omega):
    U = omega # The set of elements that are not yet covered
    C = Set() # The set of sets that will be returned

    while not U.is_empty():
        S = max(sets, key=lambda x: x.intersection(U).size())
        C = C.union(S)
        U = U.difference(S)

    return C
```

16.2.1 How bad is the approximation?

Suppose that the optimal solution is k^*. We want to find, in the worst case, how many sets the greedy algorithm will return as a function of k^*.

At least one of the subsets in k^* must be of size at least $|\Omega|/k^*$ (by the pigeonhole principle). Hence the greedy algorithm will choose at least a set of size $|\Omega|/k^*$.