Lectures on algorithms

Takunari Miyazaki

February 25, 2026

Contents

§1. Introduction

The notion of *algorithms* is fundamental to all of computer science. Simply put, an algorithm is a finite sequence of precise and simple instructions. Often, we also consider algorithms with respect to these general features (cf. [7, §1.1]):

(i) *Finiteness.* An algorithm is a finite sequence of finite statements. It must also terminate after a finite number, however large, of steps.

(ii) *Definiteness.* Each step of an algorithm must be precisely defined, in rigorous and unambiguous terms.

(iii) *Input.* Zero or more inputs are initially given before execution.

(iv) *Output.* An algorithms generates one or more outputs.

(v) *Effectiveness.* Each operation defined must be sufficiently simple that, in principle, it can be completed exactly by human manually, given sufficient time.

By way of introduction, we consider, as our opening topic, the so-called *stable-matching* problem. Suppose that, in a seminar, there are a number of computer science majors and the same number of engineering majors. We wish to pair every computer science major with a unique engineering major

	1	**2**	**3**
Xavier	Ann	Beth	Chloé
Yusuf	Beth	Ann	Chloé
Zack	Ann	Beth	Chloé

	1	**2**	**3**
Ann	Yusuf	Xavier	Zack
Beth	Xavier	Yusuf	Zack
Chloé	Xavier	Yusuf	Zack

Table 1.1: Preferences of potential partners

for an interdisciplinary project. For this, we assume in addition that every computer science major has ranked all their potential partners according to their preference and so has every engineering major. Our goal is to form a *perfect matching*, everyone having a unique partner, that is *stable*, i.e., no one benefits by breaking up with their matched parter to form a new pair in which both partners mutually prefer each other to their matched partners.

Let A and B be finite sets of the same order $n > 0$. Assume that each member of A has a ranking of preference, with no ties, of all members of B and vice versa. A perfect matching between A and B is a relation $M \subseteq A \times B$ such that, for every $a \in A$, there is unique $b \in B$ such that $(a, b) \in M$ (that is, M induces a one-to-one correspondene between A and B). Given a perfect matching M, we call a pair $(a, b) \in A \times B$ such that $(a, b) \notin M$ *unstable* if a and b mutually prefer each other to their partners matched by M. If there is no unstable pair for such an M, then we call M a *stable matching*.

Example 1.1. Consider three computer science majors, named Xavier, Yusuf and Zack, and three engineering majors, named Ann, Beth and Chloé. We wish to pair every computer science major with a unique engineering major. The students' ranked preferences for their potential partners are given in Table 1.1 (for example, the first row in the left table indicates that Xavier's most preferred partner is Ann, followed by Beth and Chloé).

(a) *Do the three pairs* (Xavier, Chloé), (Yusuf, Beth), (Zack, Ann) *form a stable matching?* No, Xavier and Beth prefer each other over their matched partners (namely, Chloé and Yusuf for Xavier and Beth, respectively). That is, (Xavier, Beth) is an unstable pair.

(b) *How about* (Xavier, Ann), (Yusuf, Beth), (Zack, Chloé)*?* Yes, in this case, it is not difficult to check that no unstable pair exists, so these form a stable matching.

(c) *Is that the only stable matching?* No, (Xavier, Beth), (Yusuf, Ann), (Zack, Chloé) also make up another stable matching.

A natural question is then, in general, given such input, does a stable matching always exist? If so, how can we find one?

Formally, we consider

Stable matching.
Given: *sets A and B of the same order n, each $a \in A$ ranking all $b \in B$ and each $b \in B$ ranking all $a \in A$ with no ties.*
Find: *a stable matching between A and B.*

Consider this so-called *propose-and-reject* approach: members of A propose to members of B, and members of B either accept or reject such proposals, all according to the given preferences. Initially, all $a \in A$ and $b \in B$ are free. To begin, an $a \in A$ asks $b \in B$ who ranks at the top of a's preference. With b available, b accepts a to tentatively form (a, b). Suppose that an $a' \in A$ also ranks the same b at the top and asks b, who has already been paired with a. If b prefers to remains with a, then b rejects a', and a' will ask the next candidate in B. If, however, b prefers a' to a, then b breaks up with a and accepts a' to form (a', b), making a free. Being freed, a will then ask the next candidate in B. This will continue until all are matched. It is important to note that a member of A never asks the same member of B again.

This approach was first proposed and formalized by D. Gale and L. S. Shapley in [4]. We now formalize the *Gale–Shapley algorithm* in:

```
procedure stable-matching($A, B$):
begin
    initialize all $a \in A$ and all $b \in B$ to be free;
    while there is a free $a \in A$, having not asked all $b \in B$, do begin
        select such an $a \in A$;
        $b \leftarrow a$'s first candidate in $B$ whom $a$ has not asked;
        if $b$ is free then
            $b$ accepts $a$ to form $(a, b)$
        else if $b$ is paired with $a' \in A$ but prefers $a$ to $a'$ then
            $b$ accepts $a$ to form $(a, b)$, making $a'$ free
        else
            $b$ declines $a$
    end
end
```

To prove its correctness, we first note two simple facts. (i) Every $a \in A$ asks all $b \in B$ in the decreasing order of a's preference. (ii) Once a $b \in B$ is paired, b never becomes free.

Lemma 1.2. stable-matching(A, B) *finds a perfect matching between A and B.*

Proof. Suppose otherwise. Since every member of A is matched with at most one member of B, where $|A| = |B|$, there must be unmatched members $a \in A$ and $b \in B$. Since b is unmatched, b has never been asked by anyone from A. On the other hand, since a is unmatched, a must have asked everyone in B. This is a contradiction. $\square$

We are now ready to prove

Theorem 1.3. stable-matching(A, B) *finds a stable matching between A and B.*

Proof. By Lemma 1.2, it is sufficient to prove that there is no unstable pair for the perfect matching found. Suppose conversely that there is indeed an unstable pair, say, (a, b) for some $a \in A$ and $b \in B$.

We first note that a must have asked b once, for, otherwise, a prefers a's matched partner to b, implying that (a, b) is not unstable. It then follows that a is not matched with b because b declined a when asked. That is, b prefers b's matched partner to a, implying that (a, b) is not unstable. Thus, we conclude that no unstable pair exists. $\square$

As for runtime, we note that, for each iteration, a member of A always asks a new member of B. Therefore, there are at most n^2 such rounds. This proves

Theorem 1.4. stable-matching(A, B) *terminates after n^2 rounds.* $\square$

As in the last example, there may be multiple stable matchings between A and B. In such cases, which one does the Gale–Shapley algorithm find?

For $a \in A$ and $b \in B$, we call a and b *valid partners* of each other if (a, b) is in some stable matching between A and B. Theorem 1.3 extends to

Theorem 1.5. stable-matching(A, B) *finds a stable matching between A and B such that*

(i) *each $a \in A$ is paired with a's best valid parter in B, and*

(ii) *each $b \in B$ is paired with b's worst valid partner in A.*

Proof. Let S denote the stable matching found by stable-matching(A, B). To prove (i), suppose conversely that S pairs $a \in A$ with someone in B other than a's best valid partner. That is, a was declined by a valid partner in B. We may assume that a is the first such member of A. Let $b \in B$ be a's first valid partner to decline a. Let S' be a stable matching that contains (a, b). When b declined a, suppose that b was paired with $a' \in A$. Suppose that $(a', b') \in S'$ for some $b' \in B$.

Since a is the first member of A to be declined, a' had not been declined by anyone in B when a was declined by b. That is, b was the first member of B whom a' asked when b paired with a', implying that a' prefers b to b'. On the other hand, b declined a when b was paired with a', so b also prefers a' to a. Hence, (a', b) is an unstable pair for S'. From this contradiction, (i) follows.

We leave the proof of (ii) as an exercise (see also [5, (1.8)]). □

§2. Analysis of algorithms

When analyzing efficiency of an algorithm formally, we most often measure the *time* it takes in the *worst case*. We represent time in a function f of the input size n, where $f(n)$ corresponds to the number of "unit operations" executed. We call such a function the *time complexity* of the algorithm.

One might wonder why we focus on the worst case, which might not accurately reflect what happens in practice. An important advantage is that we can assert simple, absolute statements about complexity, without worrying about various instances that might arise. It also turns out that there is no other effective alternative to such worst-case analysis.

To categorize efficiency, we will consider time complexity with respect to broad families of functions, such as the linear and quadratic families. Each such family will be identified by an *asymptotic order of growth*, such as

$$1,\ \log_2 n,\ n,\ n \log_2 n,\ n^2,\ n^3,\ \ldots,\ 2^n,\ n!,\ \ldots$$

Among them, we distinguish families represented by asymptotic orders of the form n^c for a constant $c > 0$. We call time complexity that belongs to such a family *polynomial time*. In fact, this notion of polynomial time formally defines efficiency; that is, *an algorithm is efficient if it has polynomial-time complexity.*

In general, polynomials represent slow growth. Indeed, when the input size n is doubled, an n^c-time algorithm's runtime increases only by the constant factor 2^c for a constant $c > 0$. In the theory of computing, polynomial time has also proven to be a robust model in which to measure and compare algorithm efficiency. Finally, polynomial-time algorithms usually expose inherent, often elegant, structures of the underlying problems.

To focus on the asymptotic order of growth, suppressing inessential details, we will use so-called *asymptotic notation*.

As usual, let $\mathbf{N} = \{0, 1, 2, \ldots\}$, the set of all natural numbers, and $\mathbf{R}^{\geq 0}$ denote the set of all nonnegative real numbers. For simplicity, we will focus on functions mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$. Of fundamental importance is this

Definition. For functions f and g mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$, we say $f(n)$ *is* $O(g(n))$ for $n \in \mathbf{N}$ if there are constants $c > 0$ and $n_0 \geq 0$ such that

$$f(n) \leq cg(n)$$

for all integers $n \geq n_0$.

The purpose is to categorize a given function f under another function g that represents a growth rate, by approximating f from above with g. For this, it is also customary to write $f(n) = O(g(n))$ to mean $f(n)$ is $O(g(n))$, as the O-notation was originally introduced to replace the "$\approx$" sign by the "$=$" sign (see, e.g., [8, Chapter I]). Some authors also use $O(g(n))$ to denote the set of *all functions* $f(n)$ for which there are constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. In that case, we write $f(n) \in O(g(n))$ if $f(n)$ satisfies such a property.

Analogously, for lower and tight bounds, we will use Ω and Θ, respectively, as proposed in [6]. Consider again functions f and g mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$. We say $f(n)$ *is* $\Omega(g(n))$ for $n \in \mathbf{N}$ if there are constants $c > 0$ and $n_0 \geq 0$ such that

$$f(n) \geq cg(n)$$

for all integers $n \geq n_0$. We say $f(n)$ *is* $\Theta(g(n))$ for $n \in \mathbf{N}$ if $f(n)$ is $O(g(n)$ as well as $\Omega(g(n))$, or, equivalently, if there are constants $c, c' > 0$ and $n_0 \geq 0$ such that

$$c'g(n) \leq f(n) \leq cg(n)$$

for all integers $n \geq n_0$.

All three properties are transitive in the following way.

Lemma 2.1. *Let f, g and h be functions mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$.*

(i) *If $f(n)$ is $O(g(n))$, and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.*

(ii) *If $f(n)$ is $\Omega(g(n))$, and $g(n)$ is $\Omega(h(n))$, then $f(n)$ is $\Omega(h(n))$.*

(iii) *If $f(n)$ is $\Theta(g(n))$, and $g(n)$ is $\Theta(h(n))$, then $f(n)$ is $\Theta(h(n))$.*

Proof. To prove (i), suppose that $f(n)$ is $O(g(n))$ and that $g(n)$ is $O(h(n))$. That is, there are constants $c_1 > 0$ and $n_1 \geq 0$ such that $f(n) \leq c_1 g(n)$ for all $n \geq n_1$. Similarly, there are constants $c_2 > 0$ and $n_2 \geq 0$ such that $g(n) \leq c_2 h(n)$ for all $n \geq n_2$. Let $c = c_1 c_2$ and $n_0 = \max(n_1, n_2)$. Evidently, $f(n) \leq ch(n)$ for all $n \geq n_0$, proving (i).

Analogously, (ii) and (iii) follow. □

These properties are also preserved in sums.

Lemma 2.2. *Let f, g and h be functions mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$.*

(i) *If $f(n)$ is $O(h(n))$, and $g(n)$ is $O(h(n))$, then $f(n) + g(n)$ is $O(h(n))$.*

(ii) *If $f(n)$ is $\Omega(h(n))$, and $g(n)$ is $\Omega(h(n))$, then $f(n) + g(n)$ is $\Omega(h(n))$.*

(iii) *If $f(n)$ is $\Theta(h(n))$, and $g(n)$ is $\Theta(h(n))$, then $f(n) + g(n)$ is $\Theta(h(n))$.*

Proof. To prove (i), suppose that $f(n)$ is $O(h(n))$ and that $g(n)$ is $O(h(n))$. That is, there are constants $c_1 > 0$ and $n_1 \geq 0$ such that $f(n) \leq c_1 h(n)$ for all $n \geq n_1$. Similarly, there are constants $c_2 > 0$ and $n_2 \geq 0$ such that $g(n) \leq c_2 h(n)$ for all $n \geq n_2$. Here, if $c = c_1 + c_2$ and $n_0 = \max(n_1, n_2)$, then $f(n) + g(n) \leq ch(n)$ for all $n \geq n_0$, proving (i).

Analogously, (ii) and (iii) follow. □

This lemma does not extend to products, however. Noteworthy is

Proposition 2.3. *For functions f and g mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$, $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$.*

Proof. Suppose that $f(n)$ is $O(g(n)$. That is, there are constants $c > 0$ and $n_0 \geq 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. If $c' = 1/c$, then $g(n) \geq c' f(n)$ for all $n \geq n_0$, proving that $g(n)$ is $\Omega(f(n))$. Analogously, the converse also holds. □

An immediate and yet important consequence of this is

Corollary 2.4. *For functions f and g mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$, $f(n)$ is $\Theta(g(n))$ if and only if $g(n)$ is $\Theta(f(n))$.* □

Given two functions, how do we compare the asymptotic growth rate of one with that of another? For example, if $f(n) = n \log_2 n$ and $g(n) = n^2$, it makes sense to assert that g grows *faster* than f. But how is such "fastness" formally defined?

Let f and g be functions mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$. Formally, we say g grows *asymptotically faster* than f if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

In such a case, we also use the *o-notation* and write $f(n) = o(g(n))$.

Similarly, we may say functions f and g have the same asymptotic growth rate if this limit converges to a positive constant, rather than zero. We note, in particular,

Proposition 2.5. *For functions f and g mapping $\mathbf{N}$ into $\mathbf{R}^{\geq 0}$, if*

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

for a constant $c > 0$, then $f(n)$ is $\Theta(g(n))$.

Proof. Suppose that

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

for a constant $c > 0$. We recall that, by definition, given any $\epsilon > 0$, there is $n_\epsilon \geq 0$ such that

$$\left| \frac{f(n)}{g(n)} - c \right| < \epsilon$$

for all integers $n \geq n_\epsilon$. In particular, there is $n_0 \geq 0$ such that

$$\left| \frac{f(n)}{g(n)} - c \right| < \frac{c}{2}$$

for all integers $n \geq n_0$. It then follows that

$$\frac{c}{2}\, g(n) < f(n) < \frac{3c}{2}\, g(n)$$

for all integers $n \geq n_0$. We conclude that $f(n)$ is $\Theta(g(n))$. □

We conclude this section with an analysis of a well-known algorithm for

Sorting.
Given: *an array A of n comparable elements.*
Find: *all elements of A sorted in the increasing order.*

We recall the bubblesort algorithm formalized in:

```
procedure bubblesort($A$):
begin
    for $i \leftarrow 1$ until $n-1$ do
        for $j \leftarrow 1$ until $n-i$ do
            if $A[j] > A[j+1]$ then
                swap $A[j]$ and $A[j+1]$
end
```

As customary with sorting algorithms, for the runtime analysis, we will focus on the number of element comparisons performed.

Theorem 2.6. bubblesort(A) *sorts n elements in $O(n^2)$ time.*

Proof. The algorithm correctly sorts all elements of A by performing $n-1$ *passes*: For $i = 1, \ldots, n-1$, during the ith pass, the ith largest element is bubbled up to its appropriate entry by sequentially swapping adjacent entries $A[j]$ and $A[j+1]$, if necessary, for $j = 1, \ldots n-i$. In bubblesort(A), the inner for-loop defines a single pass.

For $i = 1, \ldots, n-1$, during the ith pass, $n-i$ element comparisons are performed, as summarized below.

Pass	**Comparisons**
1	$n-1$
2	$n-2$
$\vdots$	$\vdots$
$n-1$	1

The total number of comparisons performed is thus defined by:

$$\begin{aligned} f(n) &= 1 + 2 + \cdots + n - 1 \\ &= \frac{(n-1)n}{2} \end{aligned}$$

Clearly, $f(n) \leq (1/2)n^2$ for all $n \geq 1$. We conclude that bubblesort(A) runs in $O(n^2)$ time. $\square$

§3. Graph algorithms

A *graph* G is defined by a pair (V, E), where V is a finite set of *nodes* (or *vertices*), and E is a finite set of *edges* each of which is defined by a pair of nodes. Unless stated otherwise, graphs will be *undirected*, where all edges are undirected, defined by unordered pairs of nodes, and *simple*, meaning that no multiple edges of the same pair of nodes nor loop edge of a pair of the same node exists. For $v, w \in V$, if $(v, w) \in E$, then we say v is *adjacent* to w (and vice versa), and $e = (v, w)$ is an edge *incident* to v (as well as w). For $v \in V$, the *degree* of v, denoted by $d(v)$, is the number of nodes adjacent to v.

To specify such graphs as input, there are two common types of representations. Consider a graph $G = (V, E)$ and $n = |V|$ and $m = |E|$ as input parameters of G. We also label the nodes as $v_1, \ldots, v_n$.

For the *adjacency-matrix representation*, we use an $n \times n$ 0-1 matrix A such that the entry in the ith row and jth column $A[i, j] = 1$ if $(v_i, v_j) \in E$, or 0 otherwise, for $i = 1, \ldots, n$ and $j = 1, \ldots, n$. As G is undirected, A is *symmetric* along the diagonal, i.e., $A[i, j] = A[j, i]$ for $i = 1, \ldots, n$ and $j = 1, \ldots, n$. As G is simple, the diagonal entries are also all 0, i.e., $A[i, i] = 0$ for $i = 1, \ldots, n$. An important advantage of such a representation is that it only takes $O(1)$ time to examine whether or not there is an edge between any two nodes, by inspecting $A[i, j]$ to determine if $(v_i, v_j) \in E$ for $i = 1, \ldots, n$ and $j = 1, \ldots, n$. However, regardless of the number of edges, it requires $\Omega(n^2)$ space to represent any graph, and it requires $O(n^2)$ time to examine all edges.

For so-called *sparse* graphs, e.g., graphs with $O(n)$ edges, the *adjacency-list representation* is more efficient. For this, we use an array A of length n, where the ith entry $A[i]$ refers to a linked list of all nodes adjacent to $v_i \in V$ for $i = 1, \ldots, n$. As G is undirected, each edge is represented in two lists. To determine whether or not there is $(v_i, v_j) \in E$, it takes $O(d(v_i))$ time to examine the list of length $d(v_i)$ from $A[i]$ for $i = 1, \ldots, n$ and $j = 1, \ldots, n$. An important advantage is that it requires $\Omega(n + m)$ space to represent any graph, and it takes $O(n + m)$ time to examine all edges.

Let $G = (V, E)$ be a graph. A *path* is a sequence of nodes $p = (v_1, \ldots, v_k)$ for $v_1, \ldots, v_k \in V$ such that $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$ (that is, $k \geq 2$). For such a p, we call $k-1$ the *length* of p. We call such a p *simple* if $v_1, \ldots, v_k$ are all distinct. We call such a p a *cycle* if $v_1 = v_k$, $k \geq 3$, and $v_1, \ldots, v_{k-1}$ are all distinct. For $v, w \in V$, we say w is *reachable* from v (and vice versa) if there is a path between v and w in G, and we say the minimum length of such a path the *distance* between v and w in G.

A graph is *connected* if there is a path between every pair of distinct nodes. A connected graph is a *tree* if it has no cycle. This formal definition of trees is characterized in this simple

Lemma 3.1. *For a graph G of n nodes, any two of the following imply the third:*

(i) *G is connected.*

(ii) *G has no cycle.*

(iii) *G has $n-1$ edges.* $\square$

A fundamental question concerning connectivity is

***s*-*t* connectivity.**
Instance: *a graph $G = (V, E)$ and nodes $s, t \in V$.*
Question: *Is there a path between s and t?*

In fact, in practice, it is more useful to consider

***s*-*t* shortest path.**
Given: *a graph $G = (V, E)$ and nodes $s, t \in V$.*
Find: *the length of a shortest path between s and t.*

To solve such problems for a graph $G = (V, E)$, we need to explore all nodes reachable from a given *source* $s \in V$; that is, we need to essentially explore the entire graph in the worst case.

One strategy is to explore outward from s in all directions, one layer at a time. In particular, if G has n nodes, we consider building these layers:

$$\begin{aligned}
L_0 &= \{s\} \\
L_1 &= \{v \in V : v \text{ is adjacent to } s\} \\
L_2 &= \{v \in V - (L_0 \cup L_1) : v \text{ is adjacent to a node in } L_1\} \\
&\vdots \\
L_i &= \{v \in V - (L_0 \cup \cdots \cup L_{i-1}) : v \text{ is adjacent to a node in } L_{i-1}\} \\
&\vdots \\
L_{n-1} &= \{v \in V - (L_0 \cup \cdots \cup L_{n-2}) : v \text{ is adjacent to a node in } L_{n-2}\}
\end{aligned}$$

We will see that L_i consists of all nodes of distance i from s for $i = 1, \ldots, n-1$. Since the furthest node from s is at distance $n-1$, if a node $v \in V$ is reachable from s, then v must be in one of these layers.

We now formalize this approach, called *breadth-first search* (BFS), in:

```
**procedure** BFS($G, s$):
**begin**
    mark $s$ and unmark all other $v \in V$;
    $L_0 \leftarrow \{s\}$;
    $i \leftarrow 0$;
    **while** $L_i \neq \emptyset$ **do begin**
        $L_{i+1} \leftarrow \emptyset$;
        **for** each $v \in L_i$ **do**
            **for** each $e = (v, w)$ for unmarked $w \in V$ **do begin**
                mark $e$;
                mark and insert $w$ into $L_{i+1}$
            **end**;
        $i \leftarrow i + 1$
    **end**
**end**
```

By induction, it is not difficult to prove

Theorem 3.2. *If $L_0, L_1, \ldots, L_{n-1}$ are the layers built by* BFS(G, s)*, then L_i consists of all nodes of distance i from s for $i = 1, \ldots, n-1$; in particular, for given $t \in V$, there is a shortest path of length i from s to t if and only if $t \in L_i$ for an integer i, where $0 \leq i \leq n - 1$.* □

Proof. To begin, we note that L_1 consists of all nodes adjacent to s and thus of distance 1 from s.

As an inductive hypothesis, we now suppose that L_i consists of all nodes of distance i from s for an integer i, where $1 \leq i < n-1$. To build L_{i+1}, the algorithm inserts into L_{i+1} all nodes $w \notin L_j$ for $j = 0, 1, \ldots, i$ such that w is adjacent to a node in L_i. As w is adjacent to a node in L_i, the distance between such a w and s is $\leq i + 1$; in fact, the distance cannot be $< i + 1$ since $w \notin L_j$ for $j = 0, 1, \ldots, i$. Thus, L_{i+1} consists of all nodes of distance $i + 1$ from s. □

The marked edges, also called *discovery edges*, form a tree called a BFS *tree*, and it consists of shortest paths to all reachable nodes from s.

We now analyze its runtime and prove

Theorem 3.3. BFS(G, s) *runs in $O(n + m)$ time for a graph G of n nodes and m edges given by an adjacency list.*

Proof. We first note that, after reading the input, the runtime is proportional to the total number of edges examined to build the layers $L_0, L_1, \ldots, L_{n-1}$. For this, it suffices to count, for $i = 0, 1, \ldots, n-1$, for each $v \in L_i$, the number of edges incident to v, namely, $d(v)$. That is, the total number of edges examined is the sum of $d(v)$ for all $v \in V$.

By the *degree-sum formula* (also called the *handshaking lemma*),

$$\sum_{v \in V} d(v) = 2m$$

(see, e.g., [2, §B.4]); to see this, we may regard that each edge $(v, w) \in E$ for $v, w \in V$ contributes exactly twice in the sum, once in $d(v)$ and once in $d(w)$.

In conclusion, with the time to read the input, the total runtime is then proportional to $n + m + 2m$, which is $O(n + m)$. $\square$

A graph $G = (V, E)$ is *bipartite* if there is a partition of V into subsets X and Y of V such that every edge of G connects a node in X and a node in Y.

As an important application of breath-first search, we next consider

Bipartiteness.
Instance: *a graph $G = (V, E)$.*
Question: *Is G bipartite?*

For this, we will consider a problem of finding a cycle of odd length.

Lemma 3.4. *A bipartite graph contains no cycle of odd length.*

Proof. Consider a cycle of length $2k + 1$ consisting of nodes $v_1, v_2, \ldots, v_{2k+1}$ for an integer $k \geq 1$. To partition the set $\{v_1, v_2, \ldots, v_{2k+1}\}$ into subsets X and Y, without loss of generality, suppose that we put v_1 into X. Then we must put v_{2i} into Y and v_{2i+1} into X for $i = 1, \ldots, k$, resulting in (v_{2k+1}, v_1) connecting two nodes in X. We conclude that no desired bipartition exists for a graph with such a cycle. $\square$

Theorem 3.5. *For a connected graph $G = (V, E)$ of n nodes and $s \in V$, if $L_0, L_1, \ldots, L_{n-1}$ are the layers built by* $\mathrm{BFS}(G, s)$*, then either*

(i) *no edge of G connects two nodes of the same layer, and G is bipartite, or*

(ii) *an edge of G connects two nodes of the same layer, and G contains a cycle of odd length.*

Proof. We first recall that, since L_i consists of the nodes of distance i from s for $i = 1, \ldots, n-1$, every edge of G connects two nodes either in the same layer or in adjacent layers. Thus, if no edge of G connects two nodes of the same layer, then every edge of G connects two nodes in adjacent layers; in particular, if we define, for $\ell = \lfloor (n-1)/2 \rfloor$,

$$X = \bigcup_{j=0}^{\ell} L_{2j} \quad \text{and} \quad Y = \bigcup_{j=0}^{\ell} L_{2j+1},$$

then every edge connects a node in X and a node in Y.

To complete the proof, we suppose that an edge of G connects two nodes of the same layer. In particular, we consider an edge $e = (x, y)$ for $x, y \in L_j$ for an integer j, where $1 \le j \le n-1$. Consider $z \in L_i$ for an integer i, where $0 \le i < j$, such that z is the closest common ancestor of x and y in the BFS tree T resulting from executing BFS(G, s) with s as its root. Consider next a cycle c consisting of the edge e, the unique path p_x from z to x in T and the unique path p_y from z to y in T. Since p_x and p_y both have length $j - i$, with e, the length of c is then $2(j-i)+1$, which is clealy odd. □

An immediate consequence of Theorem 3.5 is then

Corollary 3.6. *A graph is bipartite if and only if it contains no cycle of odd length.* □

In summary, to determine whether a given connected graph $G = (V, E)$ of n nodes is bipartite, it suffices to select $s \in V$ and run BFS(G, s). Using the resulting layers $L_0, L_1, \ldots, L_{n-1}$, we form, for $\ell = \lfloor (n-1)/2 \rfloor$,

$$X = \bigcup_{j=0}^{\ell} L_{2j} \quad \text{and} \quad Y = \bigcup_{j=0}^{\ell} L_{2j+1}.$$

If no edge of G connects two nodes in X or two nodes in Y, then G is bipartite; otherwise, G is not bipartite. This proves

Theorem 3.7. *Whether or not a graph of n nodes and m edges, given by an adjacency list, is bipartite can be determined in $O(n+m)$ time.* □

We now return to the s-t connectivity problem for a graph $G = (V, E)$ and $s, t \in V$. Alternative to breadth-first search is to explore outward from s in one direction recursively until reaching a "dead end" and then backtrack to a node adjacent to an unexplored node to restart it again. We formalize this approach, called *depth-first search* (DFS), for any given $v \in V$ in:

```
**procedure** DFS($G, v$):
**begin**
    mark $v$;
    **for** each $e = (v, w)$ for unmarked $w \in V$ **do begin**
        mark $e$;
        DFS($G, w$)
    **end**
**end**
```

As with breadth-first search, the marked edges, again called *discovery edges*, form a tree called a DFS *tree*. As depth-first search is defined recursively, the resulting DFS tree is, naturally, constructed recursively as well; in particular, we note this simple

Lemma 3.8. *If* DFS(G, v) *resursively invokes* DFS(G, w)*, then, in the* DFS *tree resulting from executing* DFS(G, v)*, all edges marked by* DFS(G, w) *form a subtree with a root w.* □

We call the edges not marked by depth-first search, i.e., not in the DFS tree, the *back edges*, as such edges connect nodes with their ancestors in the DFS tree (see, e.g., [1, §5.2]). For this, we prove this important

Theorem 3.9. *For a graph $G = (V, E)$, if $e = (v, w) \in E$ is not in a* DFS *tree T of G, then one of v and w is an ancestor of the other in T.*

Proof. Without loss of generality, suppose that depth-first search resulting in T marks v before w. When DFS(G, v) examines $e = (v, w)$, it does not mark e because w has already been marked. However, since v is marked upon the invocation of DFS(G, v) but before w is marked, when DFS(G, v) was first invoked, w was yet to be marked. Therefore, w was marked while executing DFS(G, v). By Lemma 3.8, w is a decendant of v in T. □

Clearly, an edge of a tree always connects a node with its ancestor in the tree; that is, Theorem 3.9's statement in fact applies to all edges.

Corollary 3.10. *If T is a* DFS *tree of a graph $G = (V, E)$, then, for every $e = (v, w) \in E$, one of v and w is an ancestor of the other in T.* □

Since the runtime of depth-first search is also proportional to the total number of edges examined, analogous to Theorem 3.3, we note

Theorem 3.11. DFS(G, s) *runs in* $O(n+m)$ *time for a graph* G *of* n *nodes and* m *edges given by an adjacency list.* □

To compare how, in particular, in what order, breath-first and depth-first search explore nodes, it is helpful to consider the following implementations that use well-known data structures, namely, a queue and a stack.

Using a queue, breath-first search may be naturally implemented as in:

```
procedure BFS($G, s$):
begin
    mark $s$ and unmark all other $v \in V$;
    initialize a queue $Q$ to empty;
    $Q$.enqueue($s$);
    while not $Q$.isEmpty() do begin
        $v \leftarrow Q$.dequeue();
        for each $e = (v, w)$ for unmarked $w \in V$ do begin
            mark $e$ and $w$;
            $Q$.enqueue($w$)
        end
    end
end
```

On the other hand, using a stack, depth-first search may be implemented now *nonrecursively* as in:

```
procedure DFS($G, s$):
begin
    mark $s$ and unmark all other $v \in V$;
    initialize a stack $S$ to empty;
    $S$.push($s$);
    while not $S$.isEmpty() do begin
        $v \leftarrow S$.pop();
        if $v$ is unmarked then begin
            mark $v$;
            for each $e = (v, w)$ for $w \in V$ do
                $S$.push($w$)
        end
    end
end
```

It is important to note that, unlike in the original version, in this non-recursive implementation, we do not mark the so-called discovery edges. To construct a DFS tree, we will need to maintain, in addition, a record of the last "parents" of nodes whenever they are pushed into the stack; in particular, whenever a node w is pushed into the stack via an edge (v, w) for $v \in V$, we mark the parent of w as v, overwriting the existing parent in case w has been pushed before. We then construct a DFS tree by tracing such unique parents of nodes.

For the rest of this section, we will focus on directed graphs, i.e., graphs all of whose edges have directions. Formally, a *directed graph* G is defined by a pair (V, E), where V is a finite set of *nodes*, and E is a finite set of *directed edges* defined by ordered pairs of the form (v, w) for $v, w \in V$; in particular, (v, w) now defines an edge *from v to w* (which is different from (w, v), an edge *from w to v*) for $v, w \in V$. As with undirected graphs, directed graphs also admit, while not "symmetric", both adjacency-matrix and adjacency-list representations. The notions of paths, cycles, reachability and distance defined for undirected graphs also naturally extend to directed graphs.

Let $G = (V, E)$ be a directed graph. For $v, w \in V$, we say v and w are *mutually reachable* if there are a path from v to w and a path from w to v in G. We say G is *strongly connected* if every pair of nodes of V are mutually reachable.

A natural question is then

Strong connectivity.
Instance: *a directed graph $G = (V, E)$.*
Question: *Is G strongly connected?*

We first begin with this important

Lemma 3.12. *Let $G = (V, E)$ be a directed graph and $s \in V$. Then G is strongly connected if and only if every $v \in V$, where $v \neq s$, is reachable from s, and s is reachable from every $v \in V$, where $v \neq s$.*

Proof. If G is strongly connected, then the conclusion immediately follows. We suppose that every $v \in V$, where $v \neq s$, is reachable from s, and that s is reachable from every $v \in V$, where $v \neq s$. Consider $x, y \in V$. A path from x to s and a path from s to y gives a path from x to y. Similarly, there is a path from y to x through s. Thus, x and y are mutually reachable. □

Lemma 3.12 then takes us to

Theorem 3.13. *Whether or not a directed graph of n nodes and m edges, given by an adjacency list, is strongly connected can be determined in $O(n+m)$ time.*

Proof. Let $G = (V, E)$ be a given directed graph. To begin, we select $s \in V$. We first run breadth-first search on G from s to determine if all other nodes in G are reachable from s. We also reverse the directions of the edges of G to form $G' = (V, E')$, where $E' = \{(v, w) : (w, v) \in E\}$, and run breadth-first search on G' from s to determine if there are paths to s from all other nodes in G. If both answers are affirmative, then we conclude that G is strongly connected; otherwise, we conclude that G is not strongly connected.

The correctness follows from Lemma 3.12. As for its runtime, it takes $O(n + m)$ time to reverse G's edge directions to form G'. We recall that it also takes $O(n + m)$ time to run breadth-first search. $\square$

If a directed graph has no directed cycles, then we call such a graph a *directed acyclic graph* (DAG). It turns out that DAGs arise naturally in vast areas of applications, especially in computer science (see, e.g., [5, §3.6]). Of particular importance is their useful linear orderings of nodes.

Formally, a *topological ordering* of a directed graph $G = (V, E)$ is a linear ordering $(v_1, \dots, v_n)$ of all nodes of V such that, if $(v_i, v_j) \in E$, then $i < j$ for $i = 1, \dots, n$ and $j = 1, \dots, n$.

We begin with two elementary lemmata followed by a theorem.

Lemma 3.14. *If a directed graph G has a topological ordering, then G is a directed acyclic graph.*

Proof. Let $G = (V, E)$ be a directed graph that has a topological ordering $(v_1, \dots, v_n)$ of all nodes of V. Suppose conversely that G has a cycle, say, c. Let v_i be the node with the least index in c for an integer i, where $1 \le i < n$, and v_j be the node that precedes v_i in c for an integer j, where $1 \le i < j \le n$. Clearly, $(v_j, v_i) \in E$ but $j > i$. This is a contradiction. $\square$

Lemma 3.15. *If G is a directed acyclic graph, then G has a node to which there is no edge.*

Proof. Let $G = (V, E)$ be a directed acyclic graph. Suppose conversely that there is an edge to every $v \in V$. Consider a node $x \in V$ to which there is an edge $(x', x) \in E$ for $x' \in V$. To such an x', there is also an edge $(x'', x') \in E$ for $x'' \in V$. We continue tracing back such edges to form a directed path in the reverse direction until we encounter the same node, say, y, twice. The sequence of nodes between two successive encounters with y form a cycle. This is a contradiction. □

We conclude with

Theorem 3.16. *A directed graph G has a topological ordering if and only if G is a directed acyclic graph.*

Proof. By Lemma 3.14, it suffices to only prove that, if G is a directed acyclic graph, then G has a topological ordering. Consider a directed acyclic graph $G = (V, E)$ of n nodes. We will prove by induction on n.

If $n = 1$, then the only node itself forms a topological ordering.

For $n > 1$, as an inductive hypothesis, suppose that the assertion holds for directed acyclic graphs of $n-1$ nodes. By Lemma 3.15, there is $v \in V$ to which there is no edge in G. Let G' be the graph defined by removing v and all $(v, w) \in E$ for $w \in V$. Since G' is a directed acyclic graph of $n-1$ nodes, by the inductive hypothesis, G' has a topological ordering $t(G')$. Appending v as the first node to $t(G')$ gives a topological ordering of G. □

Based on this proof, we now formalize

```
procedure topological-sort($G$):
begin
    if $|V| = 1$ then return $(v)$
    else begin
        find $v \in V$ to which there is no edge;
        remove $v$ and all $(v, w) \in E$ for $w \in V$ from $G$;
        return $v$ appended, as the first node, to topological-sort($G$)
    end
end
```

It remains to prove

Theorem 3.17. *A topological ordering of a directed acyclic graph of n nodes and m edges, given by an adjacency list, can be found in $O(n + m)$ time.*

Proof. Consider a directed acyclic graph $G = (V, E)$. To achieve $O(n + m)$ time, we maintain, in a queue, the set of all nodes to which there are currently no edges. Meanwhile, for this, as we remove nodes and edges from G, we also need to update, in an array, for each $v \in V$, the current number of incoming edges $(u, v) \in E$ for $u \in U$. Given such an array, we insert $v \in V$ into the queue upon detecting the number of v's incoming edges becoming zero.

For such bookkeeping, before we invoke topological-sort(G), we must initialize both the array and queue, in $O(n+m)$ time. After this initialization, the total runtime of the recursive procedure is then proportional to the total number of edges of G. □

§4. Greedy algorithms

While there is no formal definition, in general, a *greedy algorithm* builds up a solution in small steps, selecting a "locally optimal" option myopically at each step, for an overall optimal solution. As we will see, it is important to note that, in many cases, such a greedy approach does not always guarantee an overall optimal solution; however, when it does, it usually does so with a simple elegant algorithm.

We begin with the following single-resource scheduling problem: We are given a set of *n jobs*, each defined by a pair (s_j, f_j) such that $s_j < f_j$, where s_j and f_j indicate, respectively, the start and finish time of a job j for $j = 1, \dots, n$. The goal is to find the maximum number of jobs in the given set such that no two jobs' intervals overlap.

Formally, we say jobs (s_j, f_j) and $(s_{j'}, f_{j'})$ are *compatible* if $f_j \leq s_{j'}$ or $f_{j'} \leq s_j$. We consider

Interval scheduling.
Given: *a set of jobs J, each specifying the start and finish time.*
Find: *the maximum number of mutually-compatible jobs in J.*

As a greedy solution, it might be tempting to simply select jobs, one at a time, with the earliest start time, shortest interval, or least number of conflicts. However, it is also not difficult to construct simple counterexamples for such heuristics (see, e.g., [5, Figure 4.1]).

In this case, it turns out that the correct strategy is to select jobs one at a time to make the resource free as early as possible. That is, we select a job with the earliest finish time.

We formalize this greedy approach in the next procedure. In the following, to simplify notation, for a job j defined by a pair (s_j, f_j), we also denote s_j and f_j by $s(j)$ and $f(j)$, respectively.

```
procedure interval-scheduling($J$):
begin
    sort and label all $j \in J$ so that $f_1 \leq \cdots \leq f_n$;
    $G \leftarrow \emptyset$;
    for $j \leftarrow 1$ until $n$ do
        if $j$ is compatible with the last job added to $G$ then
            add $j$ to $G$;
    return $G$
end
```

For its correctness, we prove

Theorem 4.1. interval-scheduling(J) *finds an optimal set of mutually-compatible jobs.*

Proof. Let $G = \{g_1, \ldots, g_k\}$ be the "greedy" solution the algorithm returns, where $g_1, \ldots, g_k$ are added to G in this given order. We also consider independently an optimal solution $O = \{j_1, \ldots, j_{k'}\}$, where $f(j_1) < \cdots < f(j_{k'})$. We will prove that $k = k'$.

We will first prove by induction that $f(g_\ell) \leq f(j_\ell)$ for $\ell = 1, \ldots, k$. To begin, since g_1 is the job that finishes first in J, clearly, $f(g_1) \leq f(j_1)$. As an inductive hypothesis, suppose that $f(g_m) \leq f(j_m)$ for an integer m, where $1 \leq m < k$. Since j_m and j_{m+1} are compatible jobs, $f(j_m) \leq s(j_{m+1})$. Since $f(g_m) \leq f(j_m)$, then, $f(g_m) \leq s(j_{m+1})$. Since g_{m+1} is a job with the earliest finish time that is compatible with g_m, and j_{m+1} is compatible with g_m, it follows that $f(g_{m+1}) \leq f(j_{m+1})$.

To complete the proof, we suppose conversely that $k' > k$ and consider $j_{k+1} \in O$. We have just shown that, in particular, $f(g_k) \leq f(j_k)$. Therefore, $s(j_{k+1}) \geq f(j_k) \geq f(g_k)$. As j_{k+1} is compatible with g_k, the algorithm must have added at least one more job to G. This is a contradiction. □

As its runtime is dominated by the time for initialization involving sorting and labeling all jobs in J, we conclude with

Theorem 4.2. interval-scheduling(J) *runs in $O(n \log n)$ time for a set J of n jobs.* □

We next turn to the problem of scheduling *all* tasks, now using multiple resources. In particular, we consider a set of n *events*, each defined again by a pair (s_e, f_e) such that $s_e < f_e$, where s_e and f_e indicate, respectively, the start and finish time of an event e for $e = 1, \ldots, n$. The goal is to find the minimum number of rooms required to schedule all events in the rooms so that no two events overlap in the same room. That is, we consider

Interval partitioning.
Given: *a set of events E, each specifying the start and finish time.*
Find: *the minimum number of rooms required to schedule all events of E so that no two events overlap in the same room.*

An obvious simple strategy is to schedule given events in the order of the start time, one at a time, allocating an additional room as needed. Indeed, in this case, it turns out that this approach actually finds an optimal solution.

Formally, consider this

```
procedure interval-partitioning($E$):
begin
    sort and label all $e \in E$ so that $s_1 \leq \cdots \leq s_n$;
    $g \leftarrow 0$;
    for $e \leftarrow 1$ until $n$ do
        if $e$ is compatible with a room $r \leq g$ then
            schedule $e$ in $r$
        else begin
            schedule $e$ in an additional room $g + 1$;
            $g \leftarrow g + 1$
        end;
    return $g$
end
```

To prepare for proving optimality, we first define the notion of a depth. In general, the *depth* of a set of intervals I, denoted by $d(I)$, is the maximum number of intervals that overlap at any given time. Clearly, in this case, the number of rooms required is at least as many as the depth of the given set of events.

Evidently, the algorithm schedules all events of E, but it never schedules overlapping events in the same room. An immediate consequence of this is

Lemma 4.3. interval-partitioning(E) *allocates at least $d(E)$ rooms.* $\square$

We now prove

Theorem 4.4. interval-partitioning(E) *finds an optimal schedule in $d(E)$ rooms.*

Proof. Let g be the "greedy" number of rooms the algorithm allocates and returns. We will prove that $g = d(E)$.

The algorithm allocated the last room g to schedule an event e because there are $g - 1$ overlapping events already scheduled in the $g - 1$ rooms at that time. Since the algorithms considers all events in the order of the start time, all such overlapping events start no later than $s(e)$. With e, there are g overlapping events at $s(e)$. That is, $d(E) \geq g$. On the other hand, Lemma 4.3 asserts that $g \geq d(E)$. We conclude that $g = d(E)$. □

It remains to prove

Theorem 4.5. interval-partitioning(E) *runs in* $O(n \log n)$ *time for a set* E *of* n *events.*

Proof. As before, for initialization, we sort and label all events in E. This, of course, requires $O(n \log n)$ time. Then, for each event, we need to determine a compatible room if it exists. For this, we maintain a *priority queue* of all allocated rooms using the finish time of the last events scheduled as their "keys" for which, the earlier the finish time, the higher the priority. It then suffices to check if an event is compatible with the room with the highest priority. We recall that maintaining a priority queue of at most n elements this way requires $O(n \log n)$ time (see, e.g., [5, §2.5]). □

We return to single-resource scheduling. This time, we are given a set of n jobs, each specifying the deadline d_j and length of time t_j required to complete for $j = 1, \ldots, n$. While it requires a contiguous time interval to complete each job, both start and finish time are now flexible. As before, scheduled intervals cannot overlap, but, given the start time s_j and finish time f_j now being flexible, the goal is to schedule *all* jobs, while minimizing the worst of all delays or "lateness" defined by:

$$\ell = \max_{j=1,\ldots,n} \ell_j, \text{ where } \ell_j = \begin{cases} f_j - d_j & \text{if } f_j > d_j \\ 0 & \text{otherwise} \end{cases}$$

In summary, we consider

Minimizing lateness.

Given: *a set of jobs J, each specifying the deadline and length of time required.*

Find: *a schedule of all jobs of J minimizing the worst of all lateness.*

An obvious simple strategy is to schedule given jobs, one at a time, in the increasing order of their deadlines. Indeed, this approach actually finds an optimal solution.

Formally, consider this

```
procedure minimizing-lateness($J$):
begin
    sort and label all $j \in J$ so that $d_1 \leq \cdots \leq d_n$;
    $G \leftarrow \emptyset$;
    $t \leftarrow 0$;
    for $j \leftarrow 1$ until $n$ do begin
        schedule $j$ from $s_j = t$ until $f_j = t + t_j$ in $G$;
        $t \leftarrow t + t_j$
    end;
    return $G$
end
```

To prepare for proving optimality, we first define the notion of an inversion. In general, an *inversion* is a pair of jobs j and j' in a schedule S such that $d_j < d_{j'}$, but j' is scheduled before j in S. We call such a pair *inverted jobs*.

Evidently, since the algorithm schedules all jobs in the increasing order of their deadlines, the "greedy" schedule the algorithm returns, namely, G, has no inversions; in addition, G also has no idle time.

We recall in general that, if a sequence is not sorted, then there is at least one pair of adjacent elements out of order, or "inverted", in the sequence. In the present context, this simply means

Lemma 4.6. *If a schedule S with no idle time has an inversion, then S has one with a pair of adjacent inverted jobs.* □

This brings us to this important

Lemma 4.7. *If a pair of adjacent inverted jobs are swapped in a schedule S, then the number of inversions is reduced by one, while not increasing the worst of all lateness, in S.*

Proof. We consider adjacent jobs i and i' such that $d_i < d_{i'}$, but i' is scheduled before i in S. Let $\bar{\ell}_j$ be the lateness of a job j after swapping i and i'. Let $\bar{f}_j$ be the finish time of a job j after swapping i and i'.

Clearly, $\bar{\ell}_j = \ell_j$ for all $j \neq i$ nor i'. Since i is scheduled earlier after the swap, $\bar{\ell}_i \leq \ell_i$. As for i', if $\bar{\ell}_{i'} > 0$ after the swap, then:

$$\begin{aligned} \bar{\ell}_{i'} &= \bar{f}_{i'} - d_{i'} \\ &= f_i - d_{i'} && \text{since } \bar{f}_{i'} = f_i \\ &\leq f_i - d_i && \text{since } d_i < d_{i'} \\ &= \ell_i \end{aligned}$$

Thus, swapping i and i' does not increase the worst of all lateness in S. $\square$

We are now ready to prove

Theorem 4.8. minimizing-lateness(J) *finds a schedule of all jobs of J minimizing the worst of all lateness.*

Proof. Let G be the "greedy" schedule the algorithm returns. Let O be an optimal schedule with the fewest number of inversions. We may assume that there is no idle time in O.

If O has no inversion, then, since G schedules all jobs in the order of their deadlines, it follows that $O = G$. Suppose now that O has an inversion with a pair of adjacent inverted jobs i and i'. By Lemma 4.7, swapping i and i' reduces the number of inversions while not increasing the worst of all lateness. This is a contradiction. $\square$

As its runtime is dominated by the time for initialization involving sorting and labeling all jobs in J, we conclude with

Theorem 4.9. minimizing-lateness(J) *runs in $O(n \log n)$ time for a set J of n jobs.* $\square$

We now return to the fundamental problem of finding shortest paths in graphs. We focus on directed graphs whose edges are equipped with lengths this time. In particular, we consider

Single-source shortest path.
Given: *a directed graph $G = (V, E)$, where each edge $e \in E$ has a length $\ell_e \geq 0$, and a node $s \in V$.*
Find: *for each node $v \in V$, the length of a shortest path from s to v.*

In such a graph, the *length of a path p*, denoted by ℓ_p, is the sum of the lengths of all edges of p.

It turns out that there is a simple greedy approach to solve this problem. *Dijkstra's algorithm*, originally proposed in [3] by E. W. Dijkstra, "greedily" builds, starting from $\{s\}$, a set S of nodes whose shortest distances from s have been known, by adding one node $v \in V - S$ to S at a time, until $S = V$. Formally, consider this

```
procedure Dijkstra($G, s$):
begin
    $S \leftarrow \{s\}$;
    $d(s) \leftarrow 0$;
    while $S \neq V$ do begin
        select $v \in V - S$ that minimizes

            $d'(v) = \min_{\substack{(u,v) \in E \\ u \in S}} d(u) + \ell_{(u,v)}$;

        add $v$ to $S$;
        $d(v) \leftarrow d'(v)$
    end
end
```

Its correctness follows from

Theorem 4.10. Dijkstra(G, s) *maintains the following statement as a loop invariant: for each $v \in S$, $d(v)$ is the length of a shortest path from s to v.*

Proof. Initially, the assertion holds with $S = \{s\}$ and $d(s) = 0$. We suppose that the assertion holds for $S \neq V$ and that $v \in V$ is now being added to S via $(u, v) \in E$, where $u \in S$. It suffices to prove that $d'(v)$ is the length of a shortest path from s to v.

By definition, $d'(v)$ is the sum of $d(u)$, the length of a shortest path from s to u, and the length of (u, v). Consider any path p from s to v. If p's edge that leaves S is $(x, y) \in E$, where $x \in S$ but $y \notin S$, and the path from s to x in p is denoted by p', then:

$$\begin{array}{rcll} d'(v) & \leq & d'(y) & \text{since } v \text{ minimizes } d' \\ & \leq & d(x) + \ell_{(x,y)} & \text{by the definition of } d'(y) \\ & \leq & \ell_{p'} + \ell_{(x,y)} & \text{since } x \in S \\ & \leq & \ell_p & \text{since } p' \text{ and } (x,y) \text{ are parts of } p \end{array}$$

That is, $d'(v)$ is the length of a shortest path from s to v. □

The above version of Dijkstra's algorithm is quite succinct, and it is not entirely clear how to select efficiently an appropriate $v \in V - S$ to add to S. It appears that, whenever it considers $v \in V - S$ to add to S, the algorithm must examine all edges from the nodes of S to v, resulting in $O(mn)$ time in the worst case. It turns out though, using a priority queue, it is possible to achieve considerably better runtime. For this, we maintain the nodes of $V - S$ in a priority queue, using $d(v)$ as the key of $v \in V - S$ for which, the smaller $d(v)$, the higher the priority.

We refine the algorithm in the following way:

```
procedure Dijkstra($G, s$):
begin
    $S \leftarrow \{s\}$;
    $d(s) \leftarrow 0$;
    for each $v \in V - \{s\}$ do
        if $(s, v) \in E$ then $d(v) \leftarrow \ell_{(s,v)}$
        else $d(v) \leftarrow \infty$;
    while $S \neq V$ do begin
        select $v \in V - S$ that minimizes $d(v)$;
        add $v$ to $S$;
        for each $(v, w) \in E$ for $w \in V - S$ do
            $d(w) \leftarrow \min(d(w), d(v) + \ell_{(v,w)})$
    end
end
```

To select $v \in V - S$ that minimizes $d(v)$, it suffices to remove the node with the highest priority from the priority queue. Having added such a v to S, we update, if it improves, $d(w)$ in the priority queue for each $(v, w) \in E$ for $w \in V - S$ with $d(v) + \ell_{(v,w)}$.

In general, we recall that, in a priority queue of n elements, insertion, removal and updating a key each takes $O(\log n)$ time. After initialization, the algorithm performs n removals and m key updates. We conclude with

Theorem 4.11. Dijkstra(G, s) *runs in* $O(m \log n)$ *time for a directed graph* G *of* n *nodes and* m *edges.* $\square$

References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, Reading, Mass., 1974.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to algorithms*, 4th ed., MIT Press, Cambridge, Mass., 2022.

[3] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numer. Math. **1** (1959), 269–271.

[4] D. Gale and L. S. Shapley, *College admissions and the stability of marriage*, Amer. Math. Monthly **69** (1962), 9–15.

[5] J. Kleinberg and É. Tardos, *Algorithm design*, Addison-Wesley, Boston, 2006.

[6] D. E. Knuth, *Big omicron and big omega and big theta*, ACM SIGACT News **8**(2) (1976), 18–24.

[7] _____, *Fundamental algorithms*, 3rd ed., The Art of Computer Programming, vol. 1, Addison-Wesley, Reading, Mass., 1997.

[8] E. C. Titchmarsh, *The theory of functions*, 2nd ed., Oxford Univ. Press, Oxford, 1939.

Trinity College
Hartford, Connecticut