Warwick Mathematics Exchange

MA252

Combinatorial Optimisation

2023, February 3th

Desync, aka The Big Ree

Contents

Introduction

Combinatorial Optimisation is the study of finding optimal objects from finite sets, where the search space is discrete or discretisable. For instance, given a weighted finite graph, what is the optimal route to get from point A to point B? Combinatorial optimisation is closely related to complexity theory and theoretical computer science, as well as more practical applications in logistics and distribution optimisation problems.

This document is intended to broadly cover all the topics within the Combinatorial Optimisation module. Some basic graph theory and common data structures (arrays, queues, stacks) will be assumed; some basic definitions will be reiterated here, but for a more general overview, I recommend viewing my book, where you can also find connections between different areas conveniently hyperlinked together in one file.

Disclaimer: I make *absolutely no guarantee* that this document is complete nor without error. This document was written during the 2022 academic year, so any changes in the course since then may not be accurately reflected. Also, this module doesn't have any notes, so the structure of this document is based on (non-note) material from previous years.

Notes on formatting

Due to the nature of this module, I will be mixing mathematical and programming notation, a *lot*. obj.flag represents an instance attribute, flag, attached to an object, obj. In algorithm blocks, single equality (=) or left arrow (←) represents variable assignment while double equality (==) represents an equality check. Setting a variable to "[ ]" indicates a *list* or an *array* being instantiated.

New terminology will be introduced in *italics* when used for the first time. Named theorems will also be introduced in *italics*. Important points will be **bold**. Common mistakes will be underlined. The latter two classifications are under my interpretation. YMMV.

Content not taught in the course will be outlined in the margins like this. Anything outlined like this is not examinable, but has been included as it may be helpful to know alternative methods to solve problems.

The table of contents above, and any inline references are all hyperlinked for your convenience.

History

First Edition: 2023-01-18*
Current Edition: 2023-02-03

Authors

This document was written by R.J. Kit L., a maths student. I am not otherwise affiliated with the university, and cannot help you with related matters.

Please send me a PM on Discord @Desync#6290, a message in the WMX server, or an email to Warwick.Mathematics.Exchange@gmail.com for any corrections. (If this document somehow manages to persist for more than a few years, these contact details might be out of date, depending on the maintainers. Please check the most recently updated version you can find.)

If you found this guide helpful and want to support me, you can buy me a coffee!

(Direct link for if hyperlinks are not supported on your device/reader: ko-fi.com/desync.)

*Storing dates in big-endian format is clearly the superior option, as sorting dates lexicographically will also sort dates chronologically, which is a property that little and middle-endian date formats do not share. See ISO-8601 for more details. This footnote was made by the computer science gang.

1 Complexity Analysis

1.1 Asymptotic Notation

Big O notation describes the limiting or *asymptotic* behaviour of a function as its argument tends towards some value, often infinity. Asymptotic descriptions like this allow us to quantify how good or bad an algorithm is mathematically, instead of running and testing several implementations of that algorithm on different machines, etc.

Let f be a real (or complex) valued function, and let g be a real valued function. Furthermore, let f and g be defined on some unbounded above subset of the positive real numbers, and let $g(x) > 0$ for all sufficiently large x.

Then, if there exists $M > 0$ and x_0 such that $|f(x)| \leq Mg(x)$ for all $x \geq x_0$, we write $f(x) \in O(g(x))$ as $x \to \infty$. The assumption that x is tending to infinity is often implicit, and we just write $f(x) \in O(g(x))$ alone. Essentially, $f(x) \in O(g(x))$ if $|f|$ is bounded above by g asymptotically, up to a constant factor.

$$f(x) \in O(g(x)) \coloneqq \exists M > 0 \exists x_0 \forall x > x_0 : |f(x)| \leq Mg(x)$$

There is also an analogous definition for x tending to a finite value a involving deltas, but it will not be discussed here. Instead, we can unify the two cases with the following alternative characterisation: if

$$f(x) \in O(g(x)) \coloneqq \limsup_{x \to a} \frac{|f(x)|}{g(x)} < \infty$$

then $f(x) \in O(g(x))$ as $x \to a$.

We sometimes use $=$ instead of $\in$, but this equality is <u>not</u> symmetric. $O(f(n)) = O(g(n))$ is not the same as $O(g(n)) = O(f(n))$. For example, $O(n) = O(n^2)$, but $O(n^2) \neq O(n)$. For this reason, $\in$ will be preferred in this document. You can also interpret $O(g(n))$ as a class of functions that don't grow faster than g, so the notation $\in$ also makes sense there (though, in this interpretation, you may be tempted to write $O(n) \subset O(n^2)$ when comparing these classes, which is not common notation).

Many of these classes have names, particularly in the context of analysing algorithm efficiency. Here are a few classes and algorithms, ordered by growth rate.

Class	Name	Example
$O(1)$	Constant	Returning the first element of a list, calculating $(-1)^n$
$O(\log\log n)$	Double Logarithmic	Implementing a van Emde Boas priority queue
$O(\log n)$	Logarithmic	Binary Search
$O(n)$	Linear	Searching through an unsorted list
$O(n \log n)$	Log-Linear or Linearithmic	Fast Fourier transform, merge sort, heapsort
$O(n^2)$	Quadratic	Naive multiplication of n-digit numbers, bubble sort
$O(n^3)$	Cubic	Naive matrix multiplication
$O(n^k)$, $k \in \mathbb{N}$	Polynomial	Determinant with LU decomposition, finding maximum matching in bipartite graph
$O(k^n)$	Exponential (linear exp)	Travelling salesman with dynamic programming, solving 3-SAT
$O(k^{n^m})$	Exponential	Decide a winning strategy for a game with polynomial turns and exponential moves, such as chess or go on arbitrary sized boards.
$O(n!)$	Factorial	Travelling salesman with brute force, determinant with Laplacian expansion
$O(k^{m^n})$	Double Exp	Deciding a FOL sentence over the naturals with the addition operation and equality predicate

Another notation is Ω or *big-Omega notation.* There are two incompatible definitions for this notation, but we will follow Knuth's convention: $f(x) \in \Omega(g(x))$ if and only if $g(x) \in O(f(x))$: f is bounded below by g asymptotically, up to a constant factor.

$$f(x) \in \Omega(g(x)) \coloneqq \exists M > 0 \exists x_0 \forall x > x_0 : |f(x)| \geq Mg(x)$$

or equivalently,

$$f(x) \in \Omega(g(x)) \coloneqq \liminf_{x\to\infty} \frac{f(x)}{g(x)} > 0$$

This is just the dual of big-O notation.

The last important notation we will cover is Θ, or *big-Theta notation.* $f(x) \in \Theta(g(x))$ if both $f(x) \in O(g(x))$ and $f(x) \in \Omega(g(x))$: f is bounded both above and below by g, up to a constant factor.

$$\exists M_1 \exists M_2 \exists x_0 \forall x > x_0 : M_1 g(x) \leq f(x) \leq M_2 g(x)$$

As we only care about the shape of growth as n becomes very large, when analysing runtime complexity of algorithms, we discard all coefficients, and keep only the term with the highest growth rate, as it will eventually dominate everything else. For instance, $2x^5 + 93x^2 + 50x + 12 \in O(x^5)$.

We also don't care about the base of logs in asymptotic notations:

$$\begin{aligned}\log_a(n) &= \log_a(b)\log_b(n)\\ &= \frac{1}{\log_b(a)} \cdot \log_b(n)\\ &= k\log_b(n)\end{aligned}$$

so the base only affects the constant in the front which is discard by the asymptotic notation. Next, we give a sufficient (but not necessary) condition to test the asymptotic behaviour of a function:

Consider two functions, $f(n)$ and $g(n)$.

Suppose

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

If,

- $a = 0$, then $f(n) \in O(g(n))$
- $a = \infty$, then $f(n) \in \Omega(g(n))$
- $a \in (0,\infty)$, then $f(n) \in \Theta(g(n))$

Applying a concave function, such as log, to both $f(n)$ and $g(n)$ does not change the asymptotic relationship between them.

Example. Is $2^{\log(\log(n))^3} \in \Omega\left(\sqrt{n}\right)$?

$$\begin{aligned}f(n) &= 2^{\log(\log(n))^3}\\ g(n) &= \sqrt{n}\\ f^*(n) &= \log(\log(f(n)))\\ &= \log\left(\log\left(2^{\log(\log(n))^3}\right)\right)\\ &= \log\left(\log(\log(n))^3\log(2)\right)\\ &= \log\left(\log(\log(n))^3\right) + \log(\log(2))\end{aligned}$$

$$
\begin{aligned}
&= 3\log(\log(\log(n))) + \log(\log(2)) \\
g^*(n) &= \log\log(g(n)) \\
&= \log\left(\log\left(\sqrt{n}\right)\right) \\
&= \log\left(\frac{1}{2}\log(n)\right) \\
&= \log(\log(n)) + \log\left(\frac{1}{2}\right) \\
\lim_{n\to\infty}\frac{f^*(n)}{g^*(n)} &= \lim_{n\to\infty}\frac{3\log(\log(\log(n))) + \log(\log(2))}{\log(\log(n)) + \log\left(\frac{1}{2}\right)}
\end{aligned}
$$

Let $N = \log(\log(n))$. As $n \to \infty$, $N \to \infty$.

$$
\begin{aligned}
&= \lim_{N\to\infty}\frac{3\log(N) + \log(\log(2))}{N + \log\left(\frac{1}{2}\right)} \\
&= \lim_{N\to\infty}\frac{3\log(N)}{N + \log\left(\frac{1}{2}\right)} + \lim_{N\to\infty}\frac{\log(\log(2))}{N + \log\left(\frac{1}{2}\right)} \\
&= \lim_{N\to\infty}\frac{3\log(N)}{N + \log\left(\frac{1}{2}\right)}
\end{aligned}
$$

As $N \to \infty$, $3\log(N) \to \infty$ and $N + \log\left(\frac{1}{2}\right) \to \infty$, so apply l'hopital's rule.

$$
\begin{aligned}
&= \lim_{N\to\infty}\frac{3}{N} \\
&= 0
\end{aligned}
$$

So $2^{\log(\log(n))^3} \in O\left(\sqrt{n}\right)$, and $2^{\log(\log(n))^3} \notin \Omega\left(\sqrt{n}\right)$.

Example. Bubble sort.

To sort a list using *bubble sort*, we check the first two elements of the list, and swap them, if they are out of order. Then, we move along one, and check two elements again, then repeat until we reach the end of the list, where we start another pass. Once we pass through the list without performing any swaps, we know that the list is sorted.

Algorithm 1 Bubble Sort

```
 1: procedure BUBBLESORT($A$)                          ▷ Input array
 2:     $n \leftarrow$ LEN($A$)
 3:     repeat
 4:         swapped = false
 5:         for $i = 1$ to $n - 1$ do
 6:             if $A[i-1] > A[i]$ then                ▷ Check if the elements are out of order
 7:                 ($A[i-1]$,$A[i]$) ← ($A[i]$,$A[i-1]$)    ▷ Swap elements
 8:                 swapped = True
 9:             end if
10:         end for
11:     until swapped = False
12:     return $A$                                     ▷ Return the sorted list
13: end procedure
```

The comparison and swapping takes $\Theta(1)$ time, but runs $(n-1)$ times due to the for loop. The repeat statement will also run the for loop $(n-1)$ times, so overall, the algorithm takes $\Theta((n-1)^2) = \Theta(n^2)$.

One way to remember this result is to think about what happens if the smallest element is in the last place. Every time it is swapped, it only moves one place back, so $(n-1)$ passes are required, each one taking $\Theta(n)$ time, giving $\Theta(n^2)$.

1.2 Master Theorem

Master Theorem: For an algorithm that has complexity that obeys the equation,

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d), T(c) = \Theta(1)$$

we have,

$$T(n) \in \begin{cases} \Theta(n^d) & a < b^d \\ \Theta(n^d \cdot \log n) & a = b^d \\ \Theta(n^{\log_b a}) & a > b^d \end{cases}$$

Example. Merge sort.

In *merge sort*, we divide the list into halves, then run the algorithm again on each half, returning the list once the list is length 1. Then, we merge the sorted sublists together until only one list remains.

Algorithm 2 Merge Sort

```
procedure MERGESORT($A$)                     ▷ Input array
    $n \leftarrow$ LEN($A$)
    if $n \leq 1$ then
        return $A$                           ▷ If the list only contains one element, it is already sorted
    end if
    left ← [ ]
    right ← [ ]
    for $i = 1$ to $n$ do
        if $i < \frac{n}{2}$ then
            APPEND(left)                     ▷ Split the list into two sublists, left and right
        else
            APPEND(right)
        end if
    end for
    left←MERGESORT(left)                     ▷ Sort the two sublists
    right←MERGESORT(right)
    return MERGE(left,right)                 ▷ Merge the two sorted sublists together
end procedure
```

where the merge subroutine combines two sorted lists into one sorted list in linear time.

The algorithm takes

$$T(n) = \underbrace{T\left(\left\lfloor\frac{n}{2}\right\rfloor\right)}_{\text{Sort left}} + \underbrace{T\left(n - \left\lfloor\frac{n}{2}\right\rfloor\right)}_{\text{Sort right}} + \underbrace{\Theta(n)}_{\text{Merge}}, n > 1$$

and we know $T(1) = \Theta(1)$, as the algorithm just returns the list for an array of length 1, taking constant time.

$$\sim 2T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + \Theta(n)$$

So, using the Master theorem, we have $a = 2$, $b = 2$, $d = 1$, so,

$$= \Theta(n \log n)$$

2 Graph Theory

2.1 Minimal & Maximal Elements

If for some x, $y \leq x$ only if $y = x$, then x is *minimal.* Or equivalently, x is minimal if there does not exist any y such that $y < x$. A partial order may have any number of minimal elements, including none. For example, the integers have no minimal element, the naturals have one minimal element, 0, and a set with k mutually incomparable elements has k minimal elements.

If an element x satisfies $x \leq y$ for all y, then x is a *minimum.* A partial order may have at most one minimum, such as 0 in the naturals, but can also have none at all, either because it contains an infinite descending chain like with the integers, or because it has more than one minimal element. Any minimum element is also minimal.

We define maximal and maximum elements similarly, as elements that are not less than any other element and elements that are greater than all other elements, respectively. Again, maximum elements are also maximal.

While these definitions seem similar, they are distinct, elements can be maximal, but not maximum. For example, consider the family of all subsets of $\mathbb{N}$ with at most three elements, ordered by $\subseteq$. Then, the set $\{0,1,2\}$ is a maximal element of this family, because it's not a subset of any larger set, but it's not a maximum, because it's not a superset of $\{3\}$ (and similarly for any other three-element set).

2.2 Basic Definitions & Theorems

A *graph*, G, is represented by V, a set of *vertices* or *nodes*, and E, a set of pairs of vertices, called *edges* or *arcs*, and we write $G = (V,E)$. If we are using multiple graphs at once, we can refer to the vertex (edge) set of a graph G by writing $V(G)$ ($E(G)$), but when the context is clear, we will often write things like $G \cup v$ to mean the graph formed by adding the vertex v to the graph G.

If the edge pairs are ordered, the graph is *directed* or *oriented*, and can also be referred to as a *digraph.*

A vertex and an edge are *incident* if the vertex is at either end of the edge. The *degree*, *valency* or *order* of a vertex is the number of edges incident to it. The *indegree* and *outdegree* of a vertex of a digraph is the number of edges pointing into and out from the vertex. A vertex of degree 1 is called a *leaf.* If every vertex of a graph have the same degree k, then the graph is said to be *k-regular.*

The *degree sequence* of a graph is a sorted list of its vertex degrees. If a sequence of number is the degree sequence of some graph, it is *graphical.* For example, 3, is not a graphical sequence, as there is no graph with a single node of degree 3, while 2,2,2 *is* a graphical sequence, because the triangle graph has 2,2,2 as a degree sequence.

For a vertex $x \in V$, we define $N(x) = \{y \in V : (x,y) \in E\}$ to be the *neighbourhood* of x. The degree of x can then also be written as $\deg(x) = |N(x)|$.

An edge that starts and ends at the same vertex is called a *loop.* If multiple copies of the same edge pair exists in the edge set, then the edges are called *parallel edges.*

A graph that does not contain loops or parallel edges is called a *simple* graph. A graph that can contain parallel edges and loops is a *multigraph.*

If each edge also has a number associated with it (the *weight* of the edge), the graph is a *weighted graph.* We write (G,w) for a weighted graph, where G is the underlying unweighted graph, and w is a function that maps edges to weights. When we write $w(S)$, where $S \subseteq G$ (or $S \subseteq E(G)$), we mean the sum of the weights of the edges of S, $\sum_{e \in V(S)} w(e)$ (or $\sum_{e \in S} w(e)$, respectively).

A *walk* is a route through a graph. A walk is *closed* if the first and last vertices are the same, and *open* otherwise. A *path* is a walk in which no vertex is visited more than once. A *trail* is a walk in which no edge is visited more than once. A *cycle* is a path in which the ending and starting vertex are the same. A *ray* is an infinite path that starts at a vertex, then travels through infinitely many other vertices.

A *Hamiltonian cycle* is a cycle that visits every vertex. An *Eulerian walk* is a trail which traverses every edge. An *Eulerian circuit* is both a trail and cycle which traverses every edge.

Theorem (Euler)**.** *An Eulerian circuit exists if and only if every vertex is of even degree.*

Corollary 2.0.1. *An Eulerian walk exists if and only if there are at most two vertices of odd degree.*

A graph that admits an Eulerian walk is *traversable* or *semi-Eulerian.* A graph that admits an Eulerian circuit is *Eulerian.*

Two vertices are *connected* if there is a path between them. Two vertices, u and v, are *adjacent* if they are connected by an edge, so $(u,v) \in E$. u and v are also called *neighbours.* In a directed graph, the *in-neighbours* of a vertex v, are all vertices u such that $(u,v) \in E$, and the *out-neighbours* are all vertices u such that $(v,u) \in E$.

$N(v)$ represents the set of neighbours of v, but does not include v itself. This notation can also be used on sets of vertices to represent the set of neighbours of that set of vertices.

A *path* graph, P_n, is a graph consisting of a sole path, without cycles. That is, a line of nodes, with a single path/trail running through it. Symbolically, the path graph is a graph on n nodes, $V = \{v_1, \cdots, v_n\}$ with $E = \{(v_i,v_{i+1}) : i \in [1,n-1]\}$.

A *cycle* graph, C_n, is a graph consisting of a sole cycle.

A *complete* graph, K_n, is a graph on n nodes with every possible edge included once.

A *tournament* is a directed complete graph. If an edge points from a vertex a to a vertex b, then *a dominates b.* If $D = (V,E)$ is a tournament, and $S \subset V$ with $|S| = k$, then S is a *k-strong set* if for every $v \in V \setminus S$, there exists a $u \in S$ such that $(u,v) \in E$. In other words, every vertex not in S is dominated by at least one vertex in S.

A *bipartite* graph is a graph that has a vertex set that can be partitioned into two subsets, commonly denoted L and R, such that for, every edge, $(u,v) \in E$ either $u \in L$ and $v \in R$ *or* $u \in R$ and $v \in L$. If a graph $G = (V,E)$ has partites L and R, we also write $G = (L \cup R,E)$ to represent this data.

The *complete bipartite graph $K_{n,k}$* is the graph with two vertex partites of cardinality n and k with all possible edges between them. $K_{2,2} = C_4$. We also call a graph $K_{1,n}$ a *star graph,* and in particular, $K_{1,3}$ is the *claw graph.*

For all $k \in \mathbb{N}$, there exists a tournament on $n \in \mathbb{N}$ vertices without a k-strong set.

A graph is *connected* if every pair of vertices is connected.

A graph is a *tree* if it does not contain a cycle. An disconnected tree graph may also be called a *forest.* A directed forest is an *arborescence.* Every tree is bipartite.

A tree is *rooted* by distinguishing a vertex to be the *root.* From the root, a natural orientation of the edges can be assigned (i.e. pointing away or towards the root), forming a directed rooted tree. The maximum distance from the root to any leaves in the tree is called the *height* of the tree. If two nodes u and v are adjacent in a rooted tree, with u closer to the root, then we say that u is the *parent* of v, or that v is the *child* of u. If two vertices have the same parent node, then they are *sibling* nodes.

Given a graph G, we can *delete* a vertex by removing a vertex and removing all edges incident to it. We can similarly *delete* an edge by removing it. More interestingly, we can *contract* an edge by removing that edge, then combining the two incident vertices, such that every edge connecting to one of the original vertices connects to the new joined vertex (note that if one of several parallel edges are contracted, all the remaining parallel edges become loops on the joined vertex).

A *subgraph* of a graph G is a graph whose vertices and edges all belong to G. A subgraph is *induced* if every edge that can be included is included. In other words, an induced subgraph can be obtained by deleting vertices in G, but not edges. Given a graph G, and a subset $U \subseteq V(G)$, the subgraph of G induced by U is denoted $G[U]$.

A *spanning tree* is a subgraph that contains every vertex of the original graph, and is also a tree. A *connected component* of G is a maximal (with respect to inclusion) connected subgraph of G.

A subset of vertices $S \subseteq V$ is an *independent set* of the graph if there are no edges between any pair of vertices in S (this allows us to alternatively characterise trees as graphs whose vertex sets can be partitioned into two independent sets). Conversely, a *clique* is a subset of pairwise adjacent vertices. More generally, a *l-clique* is a subgraph that is a *complete* graph on l vertices. The *independence number* is the size of a maximum independent set, while the *clique number* is the size of a maximum clique.

Two graphs, $G = (V,E)$ and $H = (W,F)$ are *isomorphic* if there exists a bijective function, $\phi : V \to W$ such that if $(v_1,v_2) \in E$, then $\phi(v_1),\phi(v_2) \in F$, and vice versa. If such a function exists, we write $G \cong H$. The best known algorithm to determine whether two given graphs are isomorphic is $O(n^{\log n})$.

A graph G is called *H-free* if no induced subgraph of G is isomorphic to H.

The *complement* of a graph, $G = (V,E)$, denoted $\bar{G}$ or G^c is the graph (V,E'), where E' is the set of edges over V that are not in E. A graph is *self-complementary* if it is isomorphic to its complement.

A *matching* over a graph $G = (V,E)$ is a set of edges $M \subseteq E$ such that no vertex is incident to more than one edge. The *matching number* is the size of a maximum matching. A matching in which every vertex is incident to an edge is a *perfect matching.* A perfect matching is only possible on graphs with an even number of vertices.

An *alternating chain* with respect to a matching, M, is a path whose edges alternate between matched and unmatched edges. M admits an alternating chain if and only if M is not maximal.

A *vertex cover* is a subset, $S \subseteq V$ such that every edge in E is incident to at least one vertex in S.

For a graph, $G = (V,E)$, if $M \subseteq E$ is a matching and $S \subseteq V$ is a vertex cover, then $|M| \leq |S|$. This also implies that the size of a maximal matching is at most the size of a minimal vertex cover.

Consider $G = (V,E)$ and let $S \subseteq V$. S is a vertex cover of G if and only if $V \setminus S$ is an independent set.

The distance between two vertices, u and v, written as $d(u,v)$, is the length of the shortest path from u to v. On an undirected graph,

- $d(u,v) = 0 \leftrightarrow u = v$ (Point separating);
- $d(u,v) = d(v,u)$ (Symmetry);
- $d(u,v) + d(v,w) \geq d(u,w)$ (Triangle inequality).

thus satisfying the requirements for a metric. A graph, along with this definition of a distance function, is a metric space.

Theorem 2.1. *A tree on n nodes has $n - 1$ edges.*

Proof. Let $P(n)$ be the statement that every tree on n nodes has $n - 1$ edges. $P(1)$ holds, as the trivial graph has $0 = 1 - 1$ edges. Assume that $P(n)$ holds for some fixed arbitrary value of $n \geq 1$.

Let T be a tree with $n+1$ nodes. As T is a tree, it cannot contain cycles, so at least one leaf node exists. Remove the leaf, and the edge incident to it. The new graph is a tree with n nodes. By the induction hypothesis, this new graph has $n-1$ edges, so it follows that T has n edges. Thus, $P(n) \implies P(n+1)$, completing the induction. ■

Corollary 2.1.1. *Every connected graph has a spanning tree. Every connected graph over n nodes has at least $(n-1)$ edges, with exactly $(n-1)$ edges if and only if the graph is a tree.*

A *cut* is a partition of the vertex set of a graph into two disjoint sets, L and R. An edge is *in* the cut (L,R) if it connects a vertex in L with a vertex in R. The set of edges in the cut C is denoted $\delta(C)$. The *value* of the cut C is the number of edges in the cut, $|\delta(C)|$.

If $G = (V,E)$ is a graph, then there exists a cut in G with value at least $\frac{|E|}{2}$.

The deletion of any edge from a tree partitions it into two connected components.

Lemma (Euler's Handshaking Lemma)**.** *In any undirected graph (V,E), the sum of the degrees of the vertices is equal to twice the number of edges.*

$$\sum_{v \in V} \deg(v) = 2|E|$$

Proof. Every edge connects two vertices, each contributing exactly 2 to the sum of the degrees. ■

Corollary 2.1.2. *The number of odd degree vertices is even.*

Corollary 2.1.3. *Every tree on $n \geq 2$ vertices has at least two leaves.*

Theorem 2.2. *The following statements are equivalent for any <u>connected</u> graph $G = (V,E)$:*

1. *G is a tree;*
2. *G has no cycles;*
3. *Any two vertices of G are connected by a unique path;*
4. *$G' = (V, E \setminus \{e\})$ is disconnected for any $e \in E$;*
5. *$|E| = |V| - 1$*

Proof. (1) $\leftrightarrow$ (2) by definition.

(2) $\leftrightarrow$ (3) because, if the path is not unique, then the two paths together form a cycle.

(3) $\leftrightarrow$ (4) because, if G', then the endpoints of e would be connected in G by two different paths.

(5) $\leftrightarrow$ (1) by Theorem 2.1. ■

2.3 Graph Traversal

Given a finite simple graph G and a vertex $v \in V$, also called the root, we wish to find a set $R \subseteq V$ of vertices reachable from v (i.e. for every $u \in R$, there exists a path from v to u), and a set $T \subseteq E$ of edges such that (S,T) is a tree.

The two classical algorithms for this are *depth first search* (DFS) and *breadth first search* (BFS).

DFS traverses the depth of any particular path as far as possible at each step. The algorithm starts from the root, moving from the current vertex to an adjacent unvisited vertex, continuing until there are no unvisited nodes left. Then, the algorithm backtracks along previously visited nodes in reverse order until one of these visited nodes has unvisited neighbours, at which point it proceeds down the new path as far as possible again. BFS starts at the root, and explores all nodes at a given depth, before moving on to nodes at the next depth level.

These algorithms perform the same task, but are suited to different applications. For instance, if you are building a chess AI, you might use a graph traversal algorithm to explore the possible graph of future game states. In this case, BFS would look at all possible first moves, before exploring all possible combinations of first and second moves. On the other hand, a naive application of DFS would almost immediately get stuck in an infinite branch and never backtrack.

DFS can be given recursively, but we give a stack based implementation here:

Algorithm 3 Depth First Search

```
vertices = [ ]
edges = [ ]
procedure DFS($G$,$v$)
    $v$.visited = true
    $S$ = stack()
    $S$.push($v$)
    while $S$ is not empty do
        parent = $v$
        $v$ = $S$.pop()
        if $v$.visited == false then
            $v$.visited = true
            for $u \in N(v)$ do
                $S$.push($u$)
            end for
            edges.append((parent,$v$))
        end if
    end while
    for $v \in V$ do
        if $v$.visited == true then
            vertices.append($v$)
        end if
    end for
    return (vertices,edges)
end procedure
```

Note that the checking of whether a vertex has been visited is deferred until after the vertex is popped from the stack.

When giving a proof for an algorithm, we need to show termination, and correctness; that the algorithm will terminate within a finite number of steps, and that the algorithm works as intended, respectively.

We give a proof of DFS.

Proof. There are finitely many vertices, so the algorithm will terminate. Next, we prove correctness. At each pass of the while loop, the visited vertices form a tree by induction. Suppose there exists a vertex $t \in V \setminus \{R\}$ that is reachable from the root v, and let P be the path between v and t. Since $v \in R$ and $t \notin R$, there must exist two vertices x and y such that $x \in R$, $y \notin R$, and $(x,y) \in E$. Since $x \in R$, it must have been visited by the algorithm and hence have been in the stack. But then, all the neighbours of x, including y, would have been pushed onto the stack and hence marked visited (if not already visited), contradicting that $y \notin R$, and hence P and t do not exist. ∎

BFS can similarly be given recursively, but we give a queue based implementation here:

Algorithm 4 Breadth First Search

```
vertices = [ ]
edges = [ ]
procedure BFS($G$,$v$)
    $v$.visited = true
    $S$ = queue()
    $S$.enqueue($v$)
    while $S$ is not empty do
        parent = $v$
        $v$ = $S$.dequeue()
        $v$.visited = true
        for $u \in N(v)$ do
            if $u$.visited == false then
                $S$.enqueue($u$)
            end if
        end for
        edges.append((parent,$v$))
    end while
    for $v \in V$ do
        if $v$.visited == true then
            vertices.append($v$)
        end if
    end for
    return (vertices,edges)
end procedure
```

The proof of correctness is similar to the proof for DFS.

Theorem 2.3. *A BFS-tree contains a path from the root v to every vertex reachable from v, which is shortest in G.*

Proof. Let $d_G(u,v)$ denote the distance between u and v in a graph G. Suppose (S,T) is the tree returned by the BFS algorithm.

Suppose that, when the algorithm ends, there are vertices $x \in S$ such that $d_G(v,x) < d_{(S,T)}(v,w)$. Without loss of generality, let w denote the vertex closest to v with this property.

Let P be a shortest path from v to w in G, and let (u,w) be the last edge in P. Then, by assumption, $d_G(v,u) = d_{(S,T)}(v,u)$, and hence $(u,v) \notin T$.

$$\begin{aligned} d_{(S,T)}(v,w) &> d_G(v,w) \\ &= d_G(v,u) + 1 \\ &= d_{(S,T)}(v,u) + 1 \end{aligned}$$

This implies that u was enqueued earlier than w, since vertices are enqueued according to their distance from the root in (S,T). In particular, w was not enqueued until after u was dequeued, since vertices are also dequeued in order with nondecreasing distance. But, w must have been enqueued via the edge (u,w) when u was enqueued, contradicting that $(u,w) \in T$. It follows that the assumption that there exists vertices $x \in S$ such that $d_G(v,x) < d_{(S,T)}(v,w)$ is false. ■

Theorem 2.4. *Graph traversal algorithms can be implemented in a graph G with $|V| = n$ vertices and $|E| = m$ edges to run in $O(n + m)$ time. Furthermore, the connected components of G can be detected in linear time.*

2.4 Minimum Cost Spanning Tree

Given a finite connected weighted simple graph G, we wish to find a spanning tree T of G such that the total weights of the edges in T is minimal. This is the minimum cost spanning tree (MST) problem.

Let (G,w) be a weighted graph.

Theorem 2.5. *The following statements are equivalent for any spanning tree T in G:*

1. *T is an optimum solution.*
2. *For every edge $f = (x,y) \notin E(T)$, no edge on the path from x to y in T has higher cost than f.*
3. *For every edge $e \in E(T)$, e is a minimum cost edge of $\delta(V(C))$, where C is a connected component of $T \setminus \{e\}$*

Proof. (1) $\rightarrow$ (2): Suppose T is optimum, but there is an edge $f = (x,y) \notin E(T)$, and an edge e on the path connecting x to y in T such that $w(f) < w(e)$. Then $(T \setminus \{e\}) \cup \{f\}$ is a spanning tree with lower cost.

(2) $\rightarrow$ (3): Suppose (3) does not hold, so there exists an edge $f = (x,y) \in \delta(V(C))$ such that $w(f) < w(e)$ Observe that e is the only edge in $\delta(V(C))$ in T, so $f \notin E(T)$, contradicting (2).

(3) $\rightarrow$ (1): Suppose T satisfies (3), but is not optimum. Let T' be an optimum tree maximising $|E(T) \cap E(T')|$, and suppose there exists $e \in E(T) \setminus E(T')$. Let C be a connected component of $T \setminus \{e\}$, so $e \in \delta(C)$. Clearly, $T' \cup \{e\}$ contains a cycle. Let $f \in \delta(C)$ be any other edge of the cycle. $(T' \setminus \{f\}) \cup \{e\}$ is a spanning tree, and since T' is optimum, $w(f) \leq w(e)$. However, we have $w(e) \leq w(f)$ from (3), so $w(f) = w(e)$, and hence $(T' \setminus \{f\}) \cup \{e\}$ is an optimum spanning tree. But this tree has more edges in common with T than T'. This contradiction shows that $E(T) \subseteq E(T')$, and hence $E(T) = E(T')$, so T is optimum. ■

The two classical algorithms here are *Kruskal's algorithm* and *Prim's algorithm.*

Algorithm 5 Kruskal's Algorithm

```
1: Sort the edges into ascending order of weight.
2: Select an edge of least weight to start the tree.
3: Consider the next edge of least weight. If it would form a cycle with the edges already included,
   move to the next edge. Otherwise, include the edge.
4: Repeat the previous step until all vertices are connected (or equivalently, if all edges remaining would
   form a cycle).
```

Proof. The algorithm terminates as G is finite. Correctness is proven in two parts: that the graph T produced is indeed a spanning tree, and that T is minimal.

T cannot have a cycle, as edges that would form a cycle are excluded by definition. T also cannot be disconnected, since the first encountered edge that joins disconnected components of T would have been added by the algorithm. Thus, T is a spanning tree of G.

Let P be the statement that if F is the set of edges chosen at any step of the algorithm, then there is some minimal spanning tree T that contains F and none of the edges rejected by the algorithm.

Clearly, P holds at the beginning when $F = \emptyset$ as any minimal spanning tree will suffice. Assume that P holds for some arbitrary non-final step of the algorithm.

If the next chosen edge e is in T, then P also holds for $F \cup \{e\}$. Otherwise, if $e \notin E(T)$, then $T \cup \{e\}$ has a cycle by construction, C. This cycle contains edges which are not in F, since e does not form a cycle when added to F, but does in T. Let $f \in C \setminus F$ be such an edge. Note that $f \in T$, and by P, has not been considered by the algorithm. f must therefore have a weight at least as large as e. Then, $(T \setminus \{f\}) \cup \{e\}$ is a tree with the same (or less) weight as T that contains $F \cup \{e\}$, so P also holds in this case.

By induction, P holds when F is itself a spanning tree, which is possible only if F also minimum. ■

Theorem 2.6. *For a graph $G = (V,E)$, a standard implementation of Kruskal's algorithm runs in $O(|E| \log |E|)$, or equivalently, $O(|E| \log |V|)$ time.*

Remark. These time classes are equivalent as $|E|$ is at most $|V|^2$, and $\log |V|^2 = 2 \log |V| \in O(\log |V|)$.

Proof. Sorting the edges takes $O(|E| \log |E|)$ time.

Selecting an edge of least weight is just returning the first element of a sorted list, which takes constant $O(1)$ time.

Checking if this edge creates a cycle is equivalent to checking if the edge connects two vertices that lie in different trees. We track which tree each vertex lies in using a union-find structure (similar to a disjoint union in set theory), which takes $O(|V|)$ operations to initialise. Then, during runtime, in the worst case, every edge needs to be iterated over, and for each edge, we perform two tree lookups, and possibly a union, which takes at most $O(|E| \log |V|)$ time.

Thus, the total time complexity is $O(|E| \log |E|) = O(|E| \log |V|)$. ■

Algorithm 6 Prim's Algorithm

```
1: Select any vertex to start the tree.
2: Select an edge of least weight that joins a vertex already in the tree to a vertex not yet in the tree.
3: Repeat the previous step until all vertices are connected (or equivalently, if all edges remaining would
   form a cycle).
```

Proof. The proof that the produced tree T is spanning is almost identical to that of Kruskal's algorithm.

Condition (3) of Theorem 2.5 guarantees that T is optimum. ■

Theorem 2.7. *Given the adjacency matrix of a graph $G = (V,E)$, a simple implementation of Prim's algorithm runs in $O\left(|V|^2\right)$ time.*

Proof. We can find the minimum weight edge to add by linearly searching the adjacency matrix, which has $|V|^2$ entries, giving $O\left(|V|^2\right)$ time complexity. ■

Remark. Using binary or Fibonacci heaps and adjacency lists, Prim's algorithm can be improved to run in $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$ and $O(|E| + |V| \log |V|)$ time, respectively.

Kruskal's algorithm will find a spanning forest if G is disconnected, but Prim's algorithm will only find the tree spanning the connected component containing the starting vertex. Prim's algorithm can be extended to find spanning forests by iterating over the vertices.

2.4.1 Number of Spanning Trees

How many trees are there with n labeled vertices, up to isomorphism? Or equivalently, how many spanning trees does the complete graph K_n have?

On 3 vertices, the spanning tree is unique. On 3 vertices, the spanning tree is a path P_3 on the three nodes, and there are 3 ways to permute the order in which the path passes through the nodes, giving 3 spanning trees. On 4 vertices, there are $4!/2$ trees that are paths, for similar reasons, and 4 trees that are stars, giving 16 total. On 5 vertices, there are $5!/2$ copies of P_4, 5 stars with 4 leaves, and $5 \cdot 4 \cdot 3$ "stars" with 3 leaves, where one leaf is a chain of two vertices, giving 125 total.

Is there a pattern, or a general formula?

Theorem (Cayley)**.** *There are n^{n-2} trees on n labelled vertices.*

Proof. (Prüfer, 1918). For a tree T, consider its vertex set $V = \{1,2,\dots,n\}$. Note that the number of sequences of length $n-2$ from N is n^{n-2}. We will construct a bijection from the set of trees on n labelled vertices and the set of these sequences.

We convert a labelled tree into a sequence of length $n-2$ by removing the lowest labelled leaf until two vertices remained; each time a leaf is removed, its neighbour is added to the sequence.

To convert a sequence $S = (t_1,t_2,\dots,t_{n-2})$ into a labelled tree T, let s_1 be the smaller vertex in $V \setminus S$, and we join s_1 to t_1 with an edge. Then, let s_2 be the smaller vertex in $V \setminus \{s_1\} \setminus S$, and join s_2 to t_2. Repeat this process until the elements of S have been exhausted, at which point n_2 edges have been added. Join the two vertices in $V \setminus \{s_1,s_2,\dots,s_{n-2}\}$ to complete the tree. ■

2.5 Shortest Path Algorithm

Given a weighted digraph (G,w) and two vertices $s,t \in V(G)$, how can we find the shortest path from s to t (or decide that no such path exists).

If G is simple, unweighted and undirected, this can be solved using BFS by picking s to be the root node, as shown in Theorem 2.3.

Note that if a negative cycle exists, then there is no solution to this problem, as the path can be made arbitrarily negative by traversing the cycle arbitrarily many times.

If we have an instance of this problem where the weights are non-negative, then we can solve this problem with *Dijkstra's algorithm.*

1. Mark all nodes as *unvisited.*
2. Assign to every node a *tentative distance*; set this value to 0 for the starting node, and infinity to all other nodes. As the algorithm progresses, this value represents the length of the shortest path from the starting node to the given node discovered. Since no path is known initially, (apart from the starting node, with path length 0), all other tentative distances are set to infinity.
3. Also assign each node a *previous node*, representing the vertex preceding it in the path. At the beginning, this value is undefined for each vertex.
4. Set the starting node as the *current node.*
5. For the current node, consider all of its unvisited neighbours, and calculate their tentative distances through the current node. That is, the add the tentative distance of the current node to the weight of the edge connecting the current node to that neighbour. If this tentative distance is lower than the one currently assigned to that neighbour, overwrite it, and also set the previous node of the neighbour to be the current node.
6. Once all neighbours have been visited, mark the current node as visited.

7. If the destination node has been marked visited or the smallest tentative distance among the unvisited nodes is infinity (this happens if the graph is disconnected and no path exists from the starting node to infinite tentative distance node), then terminate the algorithm.
8. Otherwise, select the unvisited node with the minimum tentative distance as the new current node, and return to step (5).

This method of approximating and updating tentative distances is known as a *relaxation* method.

Note that Dijkstra's algorithm does *not* work on digraphs with negative weights, as, once a vertex is marked as visited, it is never searched again, as it assumes that the path developed to this vertex is most efficient. However, this is not necessarily true with negative weights, as an overall more efficient longer path with negative weights may exist that is locally less efficient.

$$\begin{tikzcd}
 & a \arrow[dl, "2"'] \arrow[dr, "1"] & \\
b \arrow[rr, "-3"'] & & c
\end{tikzcd}$$

For instance, in this graph, starting at a, the algorithm would search first search c, then declare it visited. Then, it would search b, and discard the path to c, as it is already marked visited.

Algorithm 7 Dijkstra's Algorithm

```
procedure BFS(($G$,$w$), start, end)
    for $v \in V(G)$ do
        $v$.visited = false
        $v$.distance = $\infty$
        $v$.previous = null
    end for
    start.distance = 0
    current = start
    while end.distance = $\infty$ do
        for $v \in N$(current) do
            if $v$.unvisited = false then
                newDist = current.distance + $w$(current,$v$)
                if newDist < $v$.distance then
                    $v$.distance = newDist
                    $v$.previous = current
                else
                    continue
                end if
            end if
        end for
        current.visited = true
        if $\min_{\{v:v.\text{visited}=\text{false}\}}(v.\text{distance}) = \infty$ then
            break
        end if
        current = $\min_{\{v:v.\text{visited}=\text{false}\}}(v.\text{distance})$
    end while
end procedure
```

We can also run Dijkstra's algorithm without giving a destination node, in which case, we change the termination condition to when all nodes have been visited, or if the smallest tentative distance among the unvisited nodes is infinity. This alternative version would find the shortest path from a source vertex to all other vertices.

Proof. G is finite, so the algorithm always terminates. Now, we show correctness.

To reduce the mixing of notation, for the purposes of this proof only, let $D(v) = v.\text{distance}$ and $p(v) = v.\text{previous}$. Also let R denote the set of *visited* vertices, and let $P_{[a,b]}$ denote the restriction of a path P to between vertices a and b in that path.

We will show that at any step of the algorithm, if a node v is the current node and s is the starting node, then,

1. $D(v) = d_{(G,w)}(s,v)$
2. If $D(v) < \infty$, then the path $v,p(v),p(p(v)),\ldots$ contains s and is furthermore the shortest path from s to v.

(1): We induct on the number of while loop iterations. In the first iteration, the current node is s, and $D(d) = 0 = d(s,s)$.

Suppose a vertex v is selected, but the shortest path P from s to v has length $w(P) < D(v)$. If all vertices of P (except for v) have been visited, then $d(v) = w(P)$ by induction. Otherwise, let y be the first unvisited vertex of P, and let $x = p(y)$ be its predecessor.

$$\begin{tikzcd} s \arrow[r] & {} \arrow[r, dotted] & x \arrow[r] & y \arrow[r, dotted] & {} \arrow[r] & v \end{tikzcd}$$

$$\begin{aligned} D(y) &\leq D(x) + w\big((x,y)\big) \\ &= d_{(G,w)}(s,x) + w\big((x,y)\big) \\ &= w(P_{[s,x]}) + w\big((x,y)\big) \\ &\leq w(P) \\ &< D(v) \end{aligned}$$

contradicting the choice of v to be the current vertex.

(2): If $D(v) < \infty$, then $D(v)$ was decreased at some point, where $p(v)$ was also created.

The values of $D(v)$ and $p(v)$ can change several times before v is visited, but after the last change, $D(v) = d_{(G,w)}(s,v)$ by (1). Also, the sequence $p(v),p(p(v)),\ldots$ contains s and defines a shortest path from s to $p(v)$ by induction, since $p(x)$ is visited for all visited x (with $p(v)$ being the base case). Thus, the sequence $v,p(v),p(p(v)),\ldots$ contains s and defines a shortest path from s to v. ■

This implementation of Dijkstra's algorithm runs in $\Theta(|V|^2)$ time, where $|V|$ is the number of vertices in the graph. However, this can be optimised with the use of Fibonacci heap min-priority queues, running in $\Theta(|E| + |V| \log |V|)$ time. This variant is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights.

Another algorithm, is the *Bellman-Ford algorithm.* Bellman-Ford is slower than Dijkstra, but it works on a larger class of problems. Notably, it can handle graphs that contain negative weights, and can also detect negative cycles.

Like Dijkstra's algorithm, Bellman-Ford proceeds by relaxation. In Dijkstra's algorithm, this is done greedily by selecting the closest vertex that hasn't been searched yet in a priority queue. Bellman-Ford just relaxes all edges, and does so $|V| - 1$ times.

Bellman-Ford also requires a cycle detection subroutine, of which $O(|V|)$ solutions are known.

Algorithm 8 Bellman-Ford

```
procedure bellmanFord(($G$,$w$), start)
    for $v \in V(G)$ do
        $v$.visited = false
        $v$.distance = $\infty$
        $v$.previous = null
    end for
    start.distance = 0
    current = start
    $i = 0$
    while $i \leq |V| - 1$ : do
        for $(u,v) \in E(V)$ do
            if $u$.distance + $w((u,v))$ < $v$.distance then
                $v$.distance = $u$.distance + $w((u,v))$
                $v$.previous = $u$
            end if
        end for
        $i = i + 1$
    end while
    for $(u,v) \in E(G)$ do
        if $u$.distance + $w((u,v))$ < $v$.distance then
            negativeLoop = [$v$,$u$]
            $i = 0$
            while $i \leq |V| - 1$ : do
                $u$ = negativeLoop[0]
                for $(u,v) \in E(G)$ do
                    if $u$.distance + $w((u,v))$ < $v$.distance then
                        negativeLoop = [$v$].append(negativeLoop)
                    end if
                end for
                $i = i + 1$
            end while
            return cycleDetect(negativeLoop)
        end if
    end for
end procedure
```

Proof. G is finite, so the algorithm always terminates. Now, we show correctness.

To reduce the mixing of notation, for the purposes of this proof only, let $D(v) = v$.distance.

We induct on the number of iterations n of the for loop in line 11. In the zeroth iteration, the starting vertex has distance 0, which is correct. For other other vertices u, $D(u) = \infty$, which is also correct because there is no path from source to u with 0 edges.

For the base case, consider when a vertex's distance is updated by $D(v) = D(u) + w((u,v))$. By the induction hypothesis, $D(u)$ is the length of a path from the starting vertex to u. Then, $D(u) + w((u,v))$ is the length of the path from the starting vertex to v that follows the path from the starting vertex to u then to v.

Now, consider a shortest path P from the starting vertex to v with at most n edges. Let u be the last vertex before v on this path. Then, the section of the path from the start to u is a shortest path from the start to u with at most $n-1$ edges, since if it were not, then there would be a path from the start to u to which we could append the edge (u,v) to construct a path from the start to v strictly shorter than

P, contradicting the choice of P. By the induction hypothesis, $D(u)$ after $n-1$ iterations is at most the length of this path from the start to u. It follows that $D(u) + w((u,v))$ is at most the length of P. In the nth iteration, $D(v)$ is compared to $D(u) + w((u,v))$, and is set to that, if it is shorter. So, after n iterations, $D(v)$ is at most the length of P, which is the length of a shortest path from the start to v with at most n edges, as required.

If there are no negative cycles, then every shortest path visits each vertex at most once, so in the for loop on line 19, no further improvements can be made. Now, suppose no improvements can be made. Then, for any cycle $v_1,v_2,\dots,v_{k-1}$, we have,

$$D(v_i) \leq D(v_{i-1 \pmod k}) + w\big((v_{i-1 \pmod k},v_i)\big)$$

Summing over the cycle, the $D(v_i)$ and $D(v_{i-1 \pmod k})$ terms cancel, leaving,

$$0 \leq \sum_{i=1}^{k} w\big((v_{i-1 \pmod k},v_i)\big)$$

so the cycle is non-negative. It follows that the algorithm returns a cycle if and only if it is negative. ■

Bellman-Ford runs in $O(|V| \cdot |E|)$ time.

A graph is *locally finite* if every vertex in the graph has finite degree.

Lemma (Kőnig). *Suppose G is connected, infinite, and locally finite. Then, G contains a ray.*

Proof. We give an inductive algorithm to generate such a ray.

Pick any vertex, $v_0 \in V(G)$. This vertex can be thought of a path of zero length, consisting of one vertex and no edges. By the assumptions of the lemma, each of the infinitely many vertices of G can be reached by a simple path that starts from v_0.

Next, as long as the current path ends at some vertex v_i, consider the infinitely many vertices that can be reached by paths that extend the current path, and for each of these vertices, construct a path to it that extends the current path. There are infinitely many of these extended paths, each of which connects from v_i to one of its neighbours, but v_i only has finitely many neighbours. It follows from the set theoretic variant of the pigeonhole principle that at least one of these neighbours is used as the next step of infinitely many of these extended paths. Let v_{i+1} be such a neighbour, and extend the current path along the edge from v_i to v_{i+1}. By construction, this extension preserves the property that infinitely many vertices can be reached by paths that extend the current path.

Repeating this process for extending the path produces an infinite sequence of finite paths, each extending the previous path in the sequence by one edge. The union of these paths gives the required ray. ■

Corollary 2.7.1. *Every infinite tree contains either a vertex of infinite degree, or an infinite path.*

Proof. If the tree is locally finite, the lemma above applies, and thus contains a ray. Otherwise, it is not locally finite and contains a vertex of infinite degree. ■

2.6 Network Flow

A *network* (G,u,s,t) is a directed graph G with two distinguished nodes called the *source* node, s, and the *sink* node, t, along with a function $u : E(G) \to \mathbb{R}_{\geq 0}$ called the *edge capacity* function.

A *flow* in a network (G,u,s,t) is a function $f : E(G) \to \mathbb{R}_{\geq 0}$ such that $f(e) \leq u(e)$ for all $e \in E(G)$: that is, the flow over an edge cannot be higher than its capacity. This is called the *capacity constraint*. A flow must also satisfy the *skew symmetry constraint*: $f\big((u,v)\big) = -f\big((v,u)\big)$. That is, the flow on an edge from a vertex u to a vertex v is equivalent to the negation of the flow from v to u.

A flow is *integral* if every edge is assigned an integer - that is, an integral flow is instead a function $f : E(G) \to \mathbb{Z}$.

A network equipped with a flow function is called a *flow network*.

Recall that a cut is a partition of a vertex set of a graph into two partites. If $X \subseteq V(G)$, then X and $V(G) \setminus X$ partition $V(G)$, thus defining a cut. Because this cut is determined entirely by X, we also denote it by X. Recall further that an edge is in the cut if it connects a vertex in one partite to a vertex in the other, and the set of edges in the cut X is denoted $\delta(X)$.

However, because G is directed, we can divide this set further. We let $\delta^+(X) \subseteq \delta(X)$ denote the set of edges leaving X, and $\delta^-(X) \subseteq \delta(X)$ denote the set of edges entering X.

If X is a singleton set containing the sole vertex v, we abbreviate $\delta(\{v\})$ as $\delta(v)$.

We define the *excess* function $x_f : V \to \mathbb{R}$ by,

$$x_f(v) \coloneqq \underbrace{\sum_{e\in\delta^-(v)} f(e)}_{\text{flow received by } v} - \underbrace{\sum_{e\in\delta^+(v)} f(e)}_{\text{flow sent by } v}$$

A node v is said to be *active* if $x_f(v) > 0$ (the node consumes flow), *deficient* if $x_f(v) < 0$ (the node produces flow), or *conserving* if $x_f(v) = 0$. A conserving node is said to satisfy the *flow conservation rule*. In flow networks, the source s is deficient and the sink t is active.

If every node apart from the source and sink is conserving, the flow is said to be a *feasible flow*. We only consider feasible flows, and shorten the name to just *flow*.

An *s-t-flow* in (G,u,s,t) is a flow f such that $x_f(s) < 0$, and $x_f(v) = 0$ for all $v \neq s,t$. The *value* of such a flow is the excess at the sink t:

$$\text{value}(f) \coloneqq x_f(t)$$

or equivalently, the negative of the excess at the source s:

$$\text{value}(f) \coloneqq -x_f(s)$$

Given a network (G,u,s,t), the *maximum flow problem* is to find an s-t-flow of maximum value.

An *s-t-cut* is a $\delta^+(A)$ for some $A \subseteq V(G)$ such that $s \in A$, $t \notin A$. A *minimum* s-t-cut is an s-t-cut of minimum total capacity.

2.6.1 Residual Networks

Let (G,u,s,t) be a network. For an edge $e = (x,y) \in E(G)$, let $\overleftarrow{e} = (y,x)$ denote the *reverse edge*.

Let $\overleftrightarrow{G}$ be the graph contained from G by adding the reverse edge for every edge of G. Note that $\overleftrightarrow{G}$ may be a multigraph, as there may now be parallel edges.

Given a network (G,u,s,t), and a flow f in it, the *residual network* (G_f,u_f,s,t) is defined by,

- $V(G_f) = V(G)$;
- $E(G_f) = \left\{e \in E\big(\overleftrightarrow{G}\big) : u_f(e) > 0\right\}$;
- $u_f(e) \coloneqq u(e) - f(e)$ for $e \in E(G)$;
- $u_f(\overleftarrow{e}) \coloneqq f(e)$, where $\overleftarrow{e}$ is a reverse edge.

The residual network indicates the additional possible flow in the original network. If there is a path from source to sink in the residual network, then it is possible to add flow. The value of an edge in the residual graph is called the *residual capacity*, which is equal to the original capacity of the edge, minus the current flow given by f. An *f-augmenting path* is a path from s to t in the residual network.

Let f be a flow, P be an f-augmenting path, and let $0 < \gamma \leq \min\limits_{e \in E(P)} \big(u_f(e)\big)$.

We *augment f along P by γ*, by,

- Increasing $f(e)$ by γ for each $e \in E(P) \cap E(G)$;
- Decreasing $f(e)$ by γ for each $\overleftarrow{e} \in E(P)$.

We now have enough machinery to tackle the maximum flow problem.

Algorithm 9 Ford-Fulkerson Algorithm

```
1: Set $f(e) = 0$ for all $e \in E(G)$.
2: Find an $f$-augmenting path $P$. If none exist, then terminate the algorithm.
3: Compute $\gamma := \min_{e \in E(P)} \big(u_f(e)\big)$.
4: Augment $f$ along $P$ by $\gamma$, and return to line (2).
```

Lemma 2.8. *For any $A \subset V(G)$ such that $s \in A$ and $t \notin A$, and any s-t-flow f, we have,*

1.
$$\text{value}(f) = \sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e)$$

2.
$$\text{value}(f) \leq \sum_{e \in \delta^+(A)} u(e)$$

Note that (2) in the lemma above states that the value of a maximum s-t-flow cannot exceed the capacity of a minimum s-t-cut.

Proof. (1):

$$\begin{aligned} \text{value}(f) &= -x_f(s) \\ &= \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e) \end{aligned}$$

Because $x_f(v) = 0$ for all $v \neq s$,

$$= \sum_{v \in A} \left(\sum_{e \in \delta^+(v)} f(e) - \sum_{e \in \delta^-(v)} f(e) \right)$$

For each $e = (x,y)$ with $x,y \in A$, $f(e)$ appears once positively and once negatively, so,

$$= \sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e)$$

(2) follows from (1) by using $0 \leq f(e) \leq u(e)$ for all $e \in E(G)$. ∎

Theorem 2.9. *An s-t-flow is maximum if and only if there is no f-augmenting path.*

Proof. If there is no f-augmenting path, then t is not reachable from s in G_f. Let R be the set of vertices reachable from s in G_f. By the definition of G_f,

$$\forall e \in \delta^+_G(R), f(e) = u(e)$$

else $e \in G_f$, in which case, there is a vertex not in R reachable from s. We also have,

$$\forall e \in \delta^-_G(R), f(e) = 0$$

else $\overleftarrow{e} \in G_f$, in which case, there is a vertex not in R reachable from s. Then, by (1) of the above lemma, we have,

$$\text{value}(f) = \sum_{e \in \delta^+(A)} u(e)$$

and hence by (2) of the above lemma, f is maximum. ■

Remark.

1. If we allow irrational capacities, the Ford-Fulkerson algorithm may not terminate at all.
2. Even in the case of integer capacities, the number of augmentations can be exponential.
3. The maximum flow problem admits a polynomial-time implementation.

Theorem (Max-Flow Min-Cut)**.** *In a network, the maximum value of an s-t-flow equals the minimum capacity of an s-t-cut.*

Theorem (Integral Flow)**.** *If each edge in a flow network has integer capacity, then there exists an integral maximum flow.*

Note that this theorem does <u>not</u> say that the *value* of the flow is an integer (which follows directly from the max-flow min-cut theorem), but that the flow on *every edge* is an integer.

Theorem (Flow Decomposition)**.** *Let (G,u,s,t) be a network, and let f be an s-t-flow in G. Then, there exists a family P^* of s-t-paths and a family C^* of cycles in G, along with a function $\omega : P^* \cup C^* \to \mathbb{R}_{\geq 0}$, such that,*

1.
$$f(e) = \sum_{\substack{K \in P^* \cup C^* \\ e \in K}} \omega(K)$$

2.
$$\text{value}(f) = \sum_{k \in P^*} \omega(K)$$

Moreover, if f is integral, then ω can be chosen to be integral.

Proof. We construct P^*, C^*, and ω by induction on the number of edges with non-zero flow. Let $e_0 = (v_0,w_0)$ be an edge with $f(e_0) > 0$. If $w_0 = t$, then we stop. Otherwise, there exists an edge $e_1 = (w_0,w_1)$ with $f(e_1) > 0$. If $w_1 = t$ r $w_1 = v_0$, then we stop. Otherwise, there exists an edge $e_2 = (w_1,w_2)$ with $f(e_2) > 0$. If $w_2 = t$ or $w_2 \in \{v_0,w_0\}$, then we stop. Continuing this process, in at most n steps, we either find a cycle, or reach vertex t. In the latter case, we repeat the procedure in the other direction and either find a cycle, or reach vertex s. In either case, we find either a cycle L, or a path L from s to t.

Set $\omega(L) = \min_{e \in L}\big(f(e)\big)$. For every $e \in L$, define $f'(e) := f(e) - \omega(L)$, and for all $e \notin L$, define $f'(e) := f(e)$.

There are strictly fewer edges of G with non-zero flow f', so, by the induction hypothesis, (1) and (2) holds for the flow f'.

We show that (1) also holds for f. If $e \notin L$, then (1) is valid for f, because in this case, $f(e) = f'(e)$. Let $e \in L$, and denote the members of $P^* \cup C^*$ containing e by $K_1, K_2, \dots, K_t$, where $K_t = L$. By induction,

$$f'(e) = \sum_{i=1}^{t-1} \omega(K_i)$$

and by definition, $f'(e) = f(e) - \omega(L)$. Therefore,

$$\begin{aligned} f(e) &= \omega(L) + f'(e) \\ &= \omega(K_t) + \sum_{i=1}^{t-1} \omega(K_i) \\ &= \sum_{i=1}^{t} \omega(K_i) \end{aligned}$$

so (1) holds for f.

Now, we show that (2) also holds for f. Suppose that L is a cycle. Then, G contains a cut $\delta(A)$ separating s from t which does not cross L, and hence value(f) = value(f') by claim (1) of the above lemma. Since (2) holds for f', we conclude it also holds for f in this case.

Now, suppose L is instead a path. Then,

$$\begin{aligned} \text{value}(f') &= \sum_{K \in P^* \setminus \{L\}} \omega(K) \\ &= \sum_{e \in \delta^+(A)} f'(e) - \sum_{e \in \delta^-(A)} f'(e) \\ &= \sum_{e \in \delta^+(A)} f(e) - \sum_{e \in \delta^-(A)} f(e) - \omega(L) \\ &= \text{value}(f) - \omega(L) \end{aligned}$$

and hence (2) holds for f. ■

Theorem (Menger (Edge Connectivity))**.** *Let G be a graph (directed or undirected), and let s,t be two distinct vertices of G. Let $k \in \mathbb{N}$. Then, these two statements are equivalent:*

1. *There are k edge-disjoint s-t-paths in G.*
2. *After deleting any $k-1$ edges from G, t is still reachable from s (e.g. G is connected).*

If the latter property holds in a graph for all s,t, then the graph is said to be *k-edge connected.*

Proof. (1) $\rightarrow$ (2) is trivial, because to destroy k edge-disjoint paths, at least k edges must be deleted (one per path).

(2) $\rightarrow$ (1): First, let G be directed. By assigning capacity $u(e) = 1$ to every edge $e \in E(G)$, we produce the network $G^* = (G,u,s,t)$. The capcity of a cut in this network is just the number of edges in the cut.

Assuming (2), the minimum capacity of a directed s-t-cut is at least k. By the max-flow min-cut theorem, G^* has an (integral) flow f of value at least k. Then, by the flow decomposition theorem,

$$\text{value}(f) = \sum_{L \in P^*} \omega(L)$$

where P^* is a family of s-t-paths, and $\omega(L) = 1$ for all $L \in P^*$. The members of P^* are edge disjoint, because by the same theorem,

$$f(e) = \sum_{L \in P^* \cup C^*} \omega(L)$$

It follows that G contains k edge-disjoint paths.

Now, let G be undirected. Transform G into a directed graph $\overrightarrow{G}$ by replacing every edge as follows:

$$\begin{tikzcd} & & & & \bullet \arrow[dd] & \\ u \arrow[r] & v & \Longrightarrow & u \arrow[ur] & & v \arrow[ul] \\ & & & & \bullet \arrow[ul] \arrow[ur] & \end{tikzcd}$$

If (2) holds on G, then (2) also holds for $\overrightarrow{G}$. So, $\overrightarrow{G}$ has k edge-disjoint s-t-paths, and hence G has k edge-disjoint s-t-paths. ■

Corollary 2.9.1. *An undirected graph G on at least two vertices is k-edge connected if and only if for each pair of distinct vertices s and t, there are k edge-disjoint s-t-paths.*

Proof. Follows directly from the theorem. ■

Theorem (Menger (Vertex Connectivity))**.** *Let G be a graph (directed or undirected), and let s,t be two non-adjacent vertices of G. Let $k \in \mathbb{N}$. Then, these two statements are equivalent:*

1. *There are k vertex-disjoint s-t-paths in G.*
2. *After deleting any $k-1$ vertices (distinct from s or t) from G, t is still reachable from s (e.g. G is connected).*

If the latter property holds in a graph for all non-adjacent s,t, then the graph is said to be *k-vertex connected.*

Proof. (1) $\to$ (2) is trivial, because to destroy k vertex-disjoint paths, at least k vertices must be deleted (one per path).

(2) $\to$ (1): First, let G be directed. Transform G into a new graph G' by replacing each vertex of G as follows:

$$\begin{tikzcd} {} \arrow[dr] & & {} & & {} \arrow[dr] & & & {} \\ & v \arrow[ur] \arrow[dr] & & \Longrightarrow & & v' \arrow[r] & v'' \arrow[ur] \arrow[dr] & \\ {} \arrow[ur] & & {} & & {} \arrow[ur] & & & {} \end{tikzcd}$$

Supose G' contains $k-1$ edges whose deletion makes t' unreachable from s''. Then, G has at most $k-1$ vertices whose deletion makes t unreachable from s. Because this contradicts (2), we conclude that after deleting any $k-1$ edges from G', t' is still reachable from s''. From the edge connectivity statement of Menger's theorem, G' has k edge-disjoint s''-t'-paths. It should be clear that these paths must also be vertex disjoint. It follows that G contains k vertex-disjoint s-t-paths.

The undirected version follows from the directed one in by the same construction as in the proof of the edge connectivity statement. ■

Corollary 2.9.2. *An undirected graph G on at least k vertices is k-vertex connected if and only if for each pair of distinct vertices s and t, there are k vertex-disjoint s-t-paths.*

Proof. Suppose G is k-vertex connected, but there exists vertices s and t in G such that there are not k vertex-disjoint s-t-paths. If s is not adjacent to t, then we apply the vertex connectivity statement of Menger's theorem to conclude that there is a set $U \subset V(G)$ of at most $k-1$ vertices such that $G \setminus U$ is disconnected, giving a contradiction.

Instead suppose that s and t are adjacent, and denote this edge e. By deleting e from G, we obtain a graph G' such that there are not $k-1$ vertex-disjoint s-t-paths. We again apply Menger's theorem to G', concluding there exists a set $X \subset V(G')$ of at most $k-2$ vertices such that $G' \setminus X$ is disconnected. Denote by S the connected component of $G' \setminus X$ containing s, and by T the connected component of $G' \setminus X$ containing T. At least one of them contains a vertex v different from s and t because $|V(G')| > k$. Without loss of generality, suppose that v is unreachable from s in $G' \setminus X$. Then, s and v are in different components of $G \setminus (X \cup \{t\})$, contradicting the assumption that G is k-vertex connected, as $|X \cup \{t\}| \leq k-1$. This completes the forward implication.

We prove the reverse implication by contraposition. Suppose G is not k-vertex connected, so there exists a set U of at most $k-1$ vertices such that $G \setminus U$ is disconnected. Take s from one connected component of $G \setminus U$, and t from another. Then, G has no k-vertex-disjoint s-t-paths. ■

The *vertex-connectivity* of a graph G is the maximum k such that G is k-(vertex)-connected. The *edge-connectivity* is similarly the maximum k such that G is k-edge connected.

2.7 Matchings

Let G be a simple graph. A *matching* in G is a subset $M \subseteq V(E)$ such that no two edges in M are incident to the same vertex. We say that a vertex v is *covered* by a matching M if v is incident to an edge $e \in M$. The *matching number* of a graph is the size of a maximum matching.

A matching is *perfect* if it covers all vertices of the graph. A perfect matching is only possible on graphs with an even number of vertices.

Given a simple graph G, the *maximum matching problem* is to find a matching of maximum cardinality in G.

Recall that a subset of vertices $S \subseteq V$ is an *independent set* of the graph if there are no edges between any pair of vertices in S, and that a graph is *bipartite* if its vertex set can be partitioned into two independent sets.

- The path graph P_n is bipartite for any value of n.
- The cycle graph C_n is bipartite for even values of n

Theorem (Characterisation of Bipartite Graphs)**.** *A graph is bipartite if and only if every closed walk in the graph is of even length.*

Proof. Suppose G is a bipartite graph with partites L and R, and let $C = (c_1, c_2, \ldots, c_k)$ be a closed walk in G. Without loss of generality, suppose we have $c_1 \in L$. Then, because G is bipartite, we have $c_2 \in R$, $c_3 \in L$, $c_{2n} \in R$, $C_{2n+1} \in L$. Because the walk is closed, it must be the case that $c_k \in R$, so k must be even. This completes the forward implication.

Now, suppose G is a simple graph with no closed walks of odd length. Without loss of generality, G is connected. Let $v, x, y \in G$.

Let P_x be a shortest path connecting v to x, and let P_y be a shortest path connecting v to y.

Let z be a vertex in both P_x and P_y closest to x and y. Then, $d(z,x)$ and $d(z,y)$ have the same parity. It follows that x and y are not adjacent, or else an odd cycle is created. This suggests the set $V_1 = \{u \in V(G) : d(u,v) \equiv 1 \pmod 2\}$ is an independent set. Through a similar argument, $V_2 = \{u \in V(G) : d(u,v) \equiv 0 \pmod 2\}$ is also an independent set. These two sets are clearly disjoint and together cover the vertex set of G, so they partition the vertex set of G and hence G is bipartite. ■

Corollary 2.9.3. *A graph is bipartite if and only if every cycle in the graph is of even length.*

Proof. The cycles in a graph are a strict subset of the closed walks. ■

In a graph G, a *vertex cover* is a subset $S \subseteq V(G)$ such that every edge in $E(G)$ is incident to at least one vertex in S. The *vertex cover number* is the size of a minimum vertex cover.

Lemma 2.10. *For all simple graphs G, the following statements hold:*

1. *A set $S \subseteq V(G)$ is a vertex cover if and only if $V(G) \setminus S$ is an independent set.*
2. *The sum of the independence number and the vertex cover number is equal to the number of vertices in G.*
3. *The vertex cover number is at most twice the matching number.*

Proof. (1) and (2) follow from the definitions of a vertex cover and an independent set.

(3): Let M be a maximal matching. Let V_m be the set of vertices incident with edges of M. Since M is maximal, every edge of the graph is incident to a vertex of V_m, and hence V_m is a vertex cover with $|V_m| = 2|M|$. Some of these vertices may not be required to form a vertex cover, so this gives an upper bound on the minimum vertex cover V, namely $|V| \leq 2|M|$. ■

If G is bipartite we can tighten this bound to an equality.

Theorem (Kőnig)**.** *In any bipartite graph, $|M| = |S|$ for a maximum matching M and minimum vertex cover S.*

Proof. Let $G = (L \cup R, E)$ be a bipartite graph. Denote by G' the graph obtained from G by adding two vertices s and t, connecting s to every vertex of L, and connecting t to every vertex of R.

Then, the maximum number of vertex-disjoint s-t-paths in G' is equal to the matching number of G. The minimum number of vertices whose deletion makes t unreachable from s is also equal to the vertex cover number of G.

It follows from the vertex connectivity statement of Menger's theorem that these two values are equal. ■

2.7.1 Hall's Condition

Theorem (Hall)**.** *Let $G = (L \cup R, E)$ be a bipartite graph. Then, G admits a matching covering L (an L-perfect matching) if and only if for all $X \subseteq L$, we have,*

$$|N(X)| \geq |X|$$

Proof. If G has an L-perfect matching, then $|N(X)| \geq |X|$ holds for all $X \subseteq L$ trivially.

Now, suppose $|N(X)| \geq |X|$ holds for all $X \subseteq L$, but there does not exist an L-perfect matching. Then by Kőnig's theorem, the vertex cover number is less than $|L|$.

Let $A \subseteq L$ and $B \subseteq R$ such that $A \cup B$ is a vertex cover of size $|A \cup B| \leq |L|$. Because G is bipartite, $N(L \setminus A) \subseteq B$, so it follows that,

$$\begin{aligned} |N(L \setminus A)| &\leq |B| \\ &< |L| - |A| \\ &= |L \setminus A| \end{aligned}$$

■

We can restate Hall's theorem in set theoretic terms.

Consider a family of sets, S, with $A_1, A_2, \cdots A_n \subseteq S$. A *system of distinct representatives* (an *SDR*) is a set of distinct elements, $\{x_1, x_2, \cdots, x_n\} \subseteq S$, such that for all $i \in [1,n]$, $x_i \in A_i$.

A family of sets, S, satisfies *Hall's condition* if, for each subfamily $W \subset S$, we have,

$$|W| \leq \left| \bigcup_{A \in W} A \right|$$

A family of sets admits an SDR if and only if Hall's condition is satisfied. That is, there exists an SDR for a family of sets $A_1, A_2, \cdots A_n$ if the union of any k of these sets contains at least k elements for all $k \in [1,n]$.

We now give necessary and sufficient conditions for the existence of a perfect matching.

Theorem 2.11. *A bipartite graph $G = (L \cup R,E)$ admits a perfect matching if and only if $|A| = |B|$ and $|N(X)| \geq |X|$ for all $X \subseteq L$.*

Remark. If $G = (L \cup R,E)$ is k-regular, then $|E| = k|L| = k|R|$, and hence $|L| = |R|$. This allows us to rephrase the previous theorem.

Theorem 2.12. *Every regular bipartite graph has a perfect matching.*

Proof. Let $G = (L \cup R,E)$ be a k-regular bipartite graph. Let $X \subseteq L$. Because G is k-regular, there are $k|X|$ edges connected to $X \subseteq L$, and $k|N(X)|$ edges connected to $N(X) \subseteq R$. The former set of edges is contained within the latter, so $k|X| \leq k|N(X)|$, and hence $|X| \leq |N(X)|$, satisfying Hall's condition. By Hall's theorem, there exists a matching on L, so every vertex in L is paired with a vertex in R. But, $|L| = |R|$, so the matching is perfect. ∎

Theorem 2.13. *The maximum matching problem can be solved for bipartite graphs with n vertices and m edges in $O(nm)$ time.*

Proof. Let $G = (L \cup R,E)$ be a bipartite graph. Construct a network G^* by:

- Adding a source s and connecting it to every vertex of L;
- Adding a sink t and connecting it to every vertex of R;
- Orienting all edges to point from s to A, from A to B, and from B to t;
- Defining the capacity function $u : E \to \mathbb{R}_{\geq 0}$ by $u(e) = 1$ for all edges $e \in E$.

Since all capacities are integers, there exists an integral maximum flow f. Because of flow conservation, the edges of G with a non-zero flow form a matching. Since the flow is maximum, the matching is maximum.

The maximum is attained after at most n augmentations in the Ford-Fulkerson algorithm. Since each augmentation takes $O(m)$ time, the total time complexity of finding a maximum matching in G is $O(nm)$. ∎

2.7.2 Maximum Independent Set

Given a simple graph G, the *maximum independent set problem* is to find an independent set in G of maximum cardinality.

Recall:

- For any bipartite graph, the matching number is equal to the vertex cover number (Kőnig's theorem)
- For any graph, the sum of the independence number and the vertex cover number is equal to the number of vertices in the graph.

Theorem 2.14. *The maximum independent set problem can be solved for bipartite graphs with n vertices and m edges in $O(nm)$ time.*

2.7.3 Augmenting Paths

Let G be a graph, and M a matching in G. A path P is an *M-alternating chain* if $E(P) \setminus M$ is a matching. An M-alternating chain is additionally *M-augmenting* if its endpoints are not covered by M. That is, P is M-alternating if its edges alternate between being in and not in M. If *both* endpoints are not in M, then P is additionally M-augmenting.

Theorem (Berge)**.** *A matching M is maximum if and only if there are no M-augmenting chains.*

Proof. If an M-augmenting chain, P, exists, then $(M \setminus P) \cup (P \setminus M)$ is a matching of cardinality strictly greater than M, so M is not maximum. Intuitively, we simply flip the edges in the augmenting chain, and take the result to be the new matching.

Conversely, if M is not maximum, and M' is a matching such that $|M'| > |M|$, then $(M \setminus M') \cup (M' \setminus M)$ consists of vertex-disjoint alternating cycles and alternating paths, where at least one path has more edges in M' than in M. This path is M-augmenting. ■

2.7.4 Maximum Weight Matching

Given a simple weighted graph (G,w), the *maximum weight matching problem* is to find a matching in G of maximum total weight. Conversely, the *minimum weight perfect matching problem* is to find a matching in G of minimum total weight, or decide that G has no perfect matching.

Theorem 2.15. *The maximum weight matching problem is equivalent to the minimum weight perfect matching problem.*

Proof. Let (G,w) be an instance of minimum weight perfect matching, and let $K = 1 + \sum_{e \in E(G)} |w(e)|$. If $w'(e) = K - w(e)$ for each edge $e \in E(G)$, then any maximum weight matching in (G,w') gives a solution to the minimum weight perfect matching in (G,w).

Let (G,w) be an instance of maximum weight matching. Then, add $|V(G)|$ new vertices to G and all possible edges to create a complete graph G' on $2|V(G)|$ vertices. Define $w'(e) = -w(e)$ for the original edges of G and $w'(e) = 0$ for new edges. Then, a minimum weight perfect matching in (G',w') yields a maximum weight matching in G by deleting the edges not in G. ■

2.7.5 Maximum Independent Set

Given a simple graph G, the *conjugate*, *adjoint*, or *line graph* of G is a graph $\mathrm{L}(G)$ that represents the adjacencies between edges of G. We construct $\mathrm{L}(G)$ as follows:

- For each edge in G, we have a vertex in $\mathrm{L}(G)$;
- For every pair of edges in G that are incident to the same vertex, we include an edge between their corresponding vertices in $\mathrm{L}(G)$.

Remark. The claw graph $K_{1,3}$ is not a line graph, so any graph containing the claw as an induced subgraph is not a line graph.

A graph that does not contain the claw as an induced subgraph is called a *claw-free graph.*

Given a simple graph G, the *maximum independent set problem* is to find an independent set in G of maximum cardinality.

Theorem 2.16. *The maximum independent set problem restricted to the class of line graphs is equivalent to the maximum matching problem.*

Proof. $M \subseteq E(G)$ is a matching G if and only if M is an independent set in $\mathrm{L}(M)$. That is, finding a maximum matching in G is equivalent to finding a maximum independent set in $\mathrm{L}(G)$. ■

Let $G = (V,E)$ be a graph, and let $S \subseteq V$ be an independent set. Let H be a bipartite subgraph of G with partites A and B such that,

- $A \subseteq S$;
- $B \subseteq V \setminus S$;
- $\forall e \in B : N(e) \cap (S \setminus A) = \emptyset$ - the vertices of B do not have neighbours in $S \setminus A$;
- $|A| < |B|$.

Then, H is an *augmenting graph* for S.

Corollary (Characterisation of Maximum Independent Sets)**.** *An independent set S is maximum if and only if there is no augmenting graphs for S.*

Proof. If there is an augmenting graph for an independent set S, then S is not maximum, because $(S \setminus A) \cup B$ is a larger independent set. This proves the forward implication by contraposition.

If S is not maximum, and R is a larger independent set, then the bipartite graph with partites $S \setminus R$ and $R \setminus S$ is augmenting for S. This proves the reverse implication by contraposition. ■

The class of line graphs is a subclass of claw-free graphs.

Remark. Every bipartite claw-free graph has vertex degree at most 2. Every connected bipartite claw-free graph is either a path or a cycle. Every connected augmenting graph in the class of claw-free graphs is a path with odd number of vertices.

Theorem 2.17. *An independent set S in a claw-free graph is maximum if and only if there is no augmenting path for S.*

Theorem 2.18. *The problem of finding augmenting paths in claw-free graphs (and hence line graphs) is solvable in polynomial-time.*

2.8 Graph Transformations for Maximum Independent Sets

Lemma 2.19. *Let G be a graph and x,y be two adjacent vertices of G. If every vertex z adjacent to x is also adjacent to y, then the independence number of G is equal to the independence number of $G \setminus \{y\}$.*

Proof. Clearly, the independence number of G is at least the independence number of $G \setminus \{y\}$.

To prove the reverse inequality, let $S \subset V(G)$ be an independent set in G. If it does not contain y, then it is also an independent set in $G \setminus \{y\}$. Otherwise, if S contains y, then it contains neither x, nor any neighbour of x. But then, $(S \setminus \{y\}) \cup \{x\}$ is an independent set on $G \setminus \{y\}$ of size $|S|$. Then, the independence number of G is at least the independence number of $G \setminus \{y\}$. ■

Given a vertex x, suppose $N(x) = Y \cup Z$. We *vertex split* x by replacing it by three vertices, x', y, and z, such that $N(x') = \{y,z\}$, $N(y) = Y$, and $N(z) = Z$.

Lemma 2.20. *Let G' be the graph obtained by vertex splitting a vertex x in G. Then, the independence number of G' is one greater than the independence number of G.*

Proof. Let S be an independent set in G containing x. Then, $(S \setminus \{x\}) \cup \{y,z\}$ is an independent set in G' of size $|S| + 1$. If S does not contain x, then $S \cup \{x'\}$ is an independent set in G' of size $|S| + 1$. So, the independence number of G' is at least one greater than the independence number of G.

Conversely, let S be an independent set in G'. If it contains at most one vertex in $\{x',y,z\}$, then by deleting this vertex, we obtain an independent set in G of size $|S| - 1$. If S contains two vertices in $\{x',y,z\}$, then these vertices must be y and z, and hence $(S \setminus \{y,z\} \cup \{x\})$ is an independent set on G of size $|S| - 1$. So, the independence number of G' is at most one greater than the independence number of G. ■

2.9 Stable Matching

Given two sets A and B of equal cardinality n, a *matching* is a bijection from the elements of one set to the other. Suppose further that each element $x \in A$ has an ordered list of preferences of elements in B, and similarly, each element in $y \in B$ has an ordered list of preferences of elements in A. If an element a prefers b to c, we write $a : b > c$.

A matching is *stable* if there does not exist elements $x \in A$ and $y \in B$ such that x prefers y over its assigned element and y also prefers x over its assigned element.

The *stable marriage problem* or *stable matching problem* (SMP) is to find a stable matching arrangement for two such sets A and B.

Example. $A = \{x,y\}$, $B = \{u,v\}$,

$x : u > v$

$y : v > u$

$u : x > y$

$v : x > y$

The matching $\{x,v\}$,$\{y,u\}$ is unstable, because x prefers u over v, and u also prefers x over y.

The matching $\{x,u\}$,$\{y,v\}$ is stable, because no pair prefers each other over their assigned elements.

One algorithm to solve this problem is the *Gale-Shapley algorithm.*

1. At each point of the algorithm, each element is either *fixed* or *free*, with every element initially being free. Elements of A may alternate between being fixed and being free, but elements of B cannot be free after being fixed.
2. In each round of the algorithm, each element $x \in A$ interacts with its preferences in order, provided the preferences haven't been interacted with in previous rounds.
3. If the preference element y is free, the two are matched and both become fixed. Otherwise, y is fixed and already has a match, z. y then compares x to z. Whichever is preferred by y becomes the new match, becoming fixed, and the rejected element becomes free.
4. Repeat until every element is fixed.

Algorithm 10 Gale-Shapley Algorithm

```
procedure SMP($A$, $B$)
    matches = [ ]
    for $n \in A \cup B$ do
        $n$.free = true
    end for
    while $\exists x \in A : x$.free = true do
        $y = x$.preferences.pop()
        if $y$.free = true then
            matches.append(($x$,$y$))
            $x$.free = false
            $y$.free = false
        else if $\exists z : (z,y) \in$ matches then
            if $y : x > z$ then
                matches.remove(($z$,$y$))
                matches.append(($x$,$y$))
                $z$.free = true
            end if
        end if
    end while
end procedure
```

Proof. Each element in A has an interact at most n times, so the algorithm terminates after at most n^2 operations.

The algorithm stops when all elements are matched, and the two input sets are of equal cardinality, so the produced matching is perfect.

Now, suppose an element $x \in A$ prefers an element $y \in B$ to its assigned element. Then, x interacted with y and, either x was not preferred over the element assigned to y at that time, or, y preferred x over its assigned element, but later changed for a more preferable element. In both cases, y prefers its current assigned element over x, and hence the matching is stable. ■

A stable matching is *optimal* for an element x if there is no stable matching with an assignment x would prefer. Conversely, a stable matching is *pessimal* for x if there is no stable matching with a worse assignment for x. We say that x and y are a stable pair if there exists a stable matching where x and y are matched.

Theorem 2.21. *The stable matching produced by the Gale-Shapley algorithm is:*

- *Independent of the order of elements selected to interact;*
- *Optimal for elements of A;*
- *Pessimal for elements of B.*

Proof. Order the elements of A arbitrarily, and let x and y be matched by the algorithm in a stable matching M_1.

Suppose there exists y' such that $x : y > y'$, and suppose that (x,y') is a stable pair, so there exists a stable matching M_2 where x is matched with y'.

Then, x was rejected by y', and without loss of generality, suppose this was the first time a stable pair was rejected by the algorithm.

Now, suppose y' rejected x in favour of x', and let y'' be the match of x' in M_2. Then (x',y'') is also a stable pair. If $x' : y'' > y'$, then x' interacted with y'' before y', which means the stable pair (x',y'') was rejected before (x,y'), contradicting the assumption that (x,y') was the first stable pair rejected by the algorithm.

So, $x' : y' > y''$. But then, the matching M_2 is not stable, because x' and y' are not matched, and they both prefer each other over their assigned elements.

This contradiction shows that every element $x \in A$ is matched with its favourable stable partner; the matching is optimal for elements of A. Because the ordering was arbitrary, any ordering produces the same result.

Now, suppose $y : x > x'$, and suppose the algorithm matches y with x, but there is a stable matching M_3 where y is matched with x'. Let y' be the match of x in M_3. Since the algorithm produces an matching optimal for elements of A, it must be the case that $x : y > y'$. But then, x and y are not matched, and they both prefer each other over their assigned elements, contradicting the stability of M_3. ■

2.10 Eulerian Graphs

Recall that an *Eulerian walk* is a trail which traverses every edge. An *Eulerian circuit* is both a trail and cycle which traverses every edge.

A graph that admits an Eulerian walk is *traversable* or *semi-Eulerian.* A graph that admits an Eulerian circuit is *Eulerian.*

Theorem (Euler)**.** *A connected undirected graph admits an Eulerian circuit if and only if the degree of each vertex is even.*

A connected directed graph admits an Eulerian circuit if and only if the in-degree $|\delta^-(v)|$ is equal to the out-degree $|\delta^+(v)|$ for each vertex v.

Proof. The necessity of the degree conditions is obvious. Sufficiency is proved by the following algorithm. ■

Given a connected undirected graph G with even degree vertices, or a digraph with in-degree equal to out-degree for all vertices, *Fleury's algorithm* returns an Eulerian circuit.

1. Start at an arbitrary vertex, v_0.
2. At each step, choose an edge whose deletion would not disconnect the graph, unless no such edge exist, in which case, pick the remaining edge left at the current vertex.
3. Move to the other endpoint of the edge and delete the edge.
4. Now repeat until no edges remain.
5. The sequence from which the edges were chosen forms an Eulerian cycle.

Algorithm 11 Fleury's Algorithm (Undirected)

```
1: Let $v_0 \in V(G)$ be arbitrary.
2: procedure FLEURY($G$, $v_0$)
3:     $W = [v_0]$
4:     $x = v_0$
5:     while $E(G) \neq \emptyset$ do
6:         if $\delta(x) = \emptyset$ then                    ▷ For digraph $G$, check $\delta^+(x) = \emptyset$
7:             $W = [v_0,e_0,v_1,e_1,\dots,v_k,e_k,v_{k+1}]$
8:             for $i = 0$ to $k$ do $W_i$ = FLEURY(()$G$, $v_i$)
9:             end for
10:            $W = W_0,e_0,W_1,e_1,\dots,W_k,e_k,v_{k+1}$
11:            return $W$
12:        else
13:            $e = (x,y),y \in \delta(x)$                    ▷ For digraph $G$, $y \in \delta^+(x)$
14:            $W = W,e,y$
15:            $x = y$
16:            $E(G) = E(G) \setminus \{e\}$
17:        end if
18:    end while
19: end procedure
```

Theorem 2.22. *Fleury's algorithm runs in $O(n+m)$ time for a graph with n vertices and m edges.*

Proof. We prove correctness by induction on m. The case $E(G) = \emptyset$ is trivial.

When line 7 is run, $v_{k+1} = v_1$ because of the degree conditions, so W is a closed walk at this stage. Let G' be the subgraph of G at this stage. Then, G' also satisfies the degree conditions.

Since G is connected, every connected component of G' contains at least one of v_i. Then, by the induction hypothesis, every edge of G' belongs to one of W_i, and hence the closed walk W composed in the last step is indeed Eulerian.

The runtime is linear because each edge is deleted immediately after being examined. ■

Corollary 2.22.1. *An Eulerian walk exists if and only if there are at most two vertices of odd degree.*

2.11 Chinese Postman

A postman must deliver mail along all streets of a town. How can he leave the post office, finish his job and return to the post office having traversed a minimum distance?

That is, given an weighted connected graph, the *Chinese postman problem* is to find a closed walk of minimum total weight visiting each edge at least once. More symbolically, the problem is, given a weighted connected graph (G,w), the task is to find a function $n : E(G) \to \mathbb{R}_{\geq 0}$ such that the graph G' constructed from G by taking $n(e)$ copies of each edge $e \in E(G)$ is Eulerian, and

$$\sum_{e \in E(G)} n(e)w(e)$$

is minimum.

If the graph is Eulerian, then the Eulerian walk provides an optimal solution. Otherwise, some edges must be visited more than once. It makes no sense to walk through an edge more than twice, so we we can restrict $n : E(G) \to \{1,2\}$. Therefore, the task simplifies to finding a subset $S \subseteq E(G)$ of minimum weight such that the graph obtained from G by doubling the edges in S is Eulerian.

As an aside, let us look at another problem.

Let G be an undirected graph, and let $T \subseteq V(G)$ be a subset of even cardinality. A subgraph J is a *T-join* if $|J \cap \delta(x)|$ is odd if and only if $x \in T$. In other words, a T-join is a spanning subgraph of G with the same vertex set as G, but only the edges that ensure that all the vertices in T have odd degree, and all the vertices not in T have even degree.

The fact that there are no T-joins for $|T|$ odd directly follows from the handshaking lemma.

Given an undirected weighted graph (G,w) and a set $T \subseteq V(G)$ of even cardinality, the *minimum weight T-join problem* is to find a minimum weight T-join in G, or decide that none exists.

Lemma 2.23. *Let G be a graph and let $T \subseteq V(G)$ be a subset of even cardinality. There exists a T-join in G if and only if $|V(C) \cap T|$ is even for each connected component C in G.*

Proof. If J is a T-join, then for each connected component C in G, we have,

$$\sum_{v \in V(C)} |J \cap \delta(v)| = 2|J \cap E(C)|$$

So, $|J \cap \delta(v)|$ is odd for an even number of vertices in $V(C)$. Since J is a T-join, this means that $|V(C) \cap T|$ is even. This completes the forward implication.

Conversely, let $|V(C) \cap T|$ be even for each connected component C of G. Then T can be partitioned into pairs $\{v_1,w_1\},\ldots,\{v_k,w_k\}$ with $k = |T|/2$ such that for each i, the pair $\{v_i,w_i\}$ belongs to the same connected component. Let P_i be an arbitrary v_i-w_i-path, and let,

$$J := \mathop{\triangle}_{i=1}^{k} E(P_i)$$

where $\triangle$ is the symmetric difference operation $(A \triangle B := (A \setminus B) \cup (B \setminus A) = \{x : (x \in A) \oplus (x \in B)\})$.

The symmetric difference of more than two sets consists of the elements that belong to an odd number of the sets. Observe that if the paths $P_1,\ldots,P_k$ are disjoint, then J is a T-join by definition, as it respects the degrees of vertices in T and not in T. If the paths are not disjoint, then the degree of each vertex has the same parity with respect to J as with respect to the disjoint union of the paths. In either case, J is a T-join, completing the reverse implication. ■

Lemma 2.24. *A T-join J in a weighted graph (G,w) has minimum weight if and only if for each cycle C in G, we have,*

$$w(J \cap E(C)) \leq w(E(C) \setminus J)$$

Proof. If $w(J \cap E(C)) > w(E(C) \setminus J)$, then $J \triangle E(C)$ is a T-join of lower weight than J.

Conversely, if J' is a T-join with $w(J) < w(J)$, then the subgraph of G formed by the edges of $J \triangle J'$ is Eulerian, as the degree of each vertex in this subgraph is even, in which case, it is the union of cycles. For at least one cycle C, we must have $w(J \cap E(C)) > w(J' \cap E(C)) = w(E(C) \setminus J)$. ■

Lemma 2.25. *Let (G,w) be a weighted graph, and let $T \subseteq V(G)$ be a subset of even cardinality. Every optimum T-join in G is the symmetric difference of $|T|/2$ paths whose endpoints are distinct and belong to T, and possibly some zero-weight cycles.*

Proof. We induct on T. The case $T = \emptyset$ holds trivially.

Let J be any optimum T-join in G. Without loss of generality, J contains no zero-weight cycle. By Theorem 2.24, J contains no cycle of positive weight. Since w is non-negative, J is a forest. Let x and y be two leaf nodes in the same connected component of this forest, and let P be the unique x-y-path in J. Then, by the definition of a T-join, $x,y \in T$. So, $J \setminus E(P)$ is an optimum T'-join, where $T' = T \setminus \{x,y\}$, so a cheaper T'-join would imply a cheaper T-join. The lemma follows by induction. ■

Theorem 2.26. *The minimum weight T-join problem with non-negative weights can be solved in polynomial time.*

Proof. For each pair $x,y \in T$, we find a shortest x-y-path $P_{x,y}$ and construct an auxiliary complete edge-weighted graph G^* with vertex set T, in which the weight of the edge (x,y) equals the length of the path $P_{x,y}$. Finding these paths for all possible pairs of vertices x and y can be done in $O(|V|^3)$ time using the Floyd-Warshall algorithm.

Then, we find in G^* a perfect matching M of minimum weight, which takes polynomial time.

Let J be the symmetric difference of the paths $P_{x,y}$ taken over all edges (x,y). Then J is a T-join, and is minimum because M has minimal weight. ■

Theorem 2.27. *If the weights are non-negative, then the minimum weight T-join problem coincides with the undirected Chinese postman problem.*

Proof. Otherwise, let T be the set of vertices of odd degree, noting that $|T|$ is even by the handshaking lemma, and set $w(e) = 1$ for all edges $e \in E(G)$. Now compute a minimum-cost T-join J with respect to w, and form the multigraph G^* by duplicating the edges in J. A Euerian cycle in G^* is now the desired Chinese postman tour in G. ■

2.12 Independence System

For a finite set S, we denote by $\mathcal{P}(S)$ or 2^S the power set of S (the set of all subsets of S).

A *set system* $(V,\mathcal{F})$ consists of a finite set V and some set of subsets $\mathcal{F} \subseteq \mathcal{P}(V)$.

A set system $S = (V,\mathcal{I})$ is furthermore an *independence system* if,

(M1) $\emptyset \in \mathcal{I}$

(M2) For each $Y \subseteq X$, $Y \in F \to X \in \mathcal{I}$.

This latter property is also called the *hereditary property* or *downward-closedness.*

Elements of $\mathcal{I}$ are called *independent* or *feasible*, while elements of $\mathcal{I} \setminus V$ are *dependent* or *infeasible.*

Minimal dependent sets are called *circuits*, and maximal independent sets are called *bases.* For $X \subseteq V$, the maximal independent subsets of X are called *bases of X.*

Let $(V,\mathcal{I})$ be an independence system. For $X \subseteq V$, we define the rank $\text{rank}(X)$ of X as the size of a maximum subset of X that belongs to $\mathcal{I}$.

Example. The following are all independence systems:

1. $V = V(G)$ and $\mathcal{I}$ is the set of independent sets in a graph G.
2. $V = E(G)$ and $\mathcal{I}$ is the set of forests in G.
3. V is the set of columns of a matrix over some field and $\mathcal{I}$ is the power set of linearly independent columns.

4. V is any finite set, k is an integer, and $\mathcal{I}$ the subsets of V of cardinality at most k.

Given an independence system $(V,\mathcal{I})$ and a weight function $w : V \to \mathbb{R}$, a *minimisation problem* is to find a basis of minimum total weight, while a *maximisation problem* is to find an independent set of maximum total weight.

Many combinatorial optimisation problems can be formulated as minimisation and maximisation problems. For instance,

- MAXIMUM-WEIGHT-STABLE-SET
- TSP
- SHORTEST-PATH
- KNAPSACK
- MINIMUM-WEIGHT-SPANNING-TREE
- MAXIMUM-WEIGHT-FOREST
- STEINER-TREE
- MAXIMUM-WEIGHT-BRANCHING
- MINIMUM-WEIGHT-BRANCHING
- JSSP (JOB-SHOP-SCHEDULING-PROBLEM)

An independence system $(V,\mathcal{I})$ is a *matroid* if,

M3 $\forall X,Y \in \mathcal{I} : |X| > |Y| \to \Big(\exists x \in X \setminus Y : \big(Y \cup \{x\}\big) \in \mathcal{I}\Big)$ - if $X,Y \in \mathcal{I}$ and $|X| > |Y|$, then there exists an $x \in X \setminus Y$ such that $Y \cup \{x\} \in \mathcal{I}$.

Example.

1. (Independent sets in a graph) is not a matroid.
2. (Forests in a graph) is a matroid known as the *cycle (graphic) matroid.*
3. (Linearly independent columns) is a matroid known as the *vector matroid.*
4. (Subsets of size at most k) is a matroid known as the *uniform matroid.*

Theorem 2.28. *Let $(V,\mathcal{I})$ be an independence system. Then the following statements are equivalent:*

(M3) $\forall X,Y \in \mathcal{I} : |X| > |Y| \to \Big(\exists x \in X \setminus Y : \big(Y \cup \{x\}\big) \in \mathcal{I}\Big)$

(M3)' For all $Z \subseteq V$, all bases of Z have the same cardinality.

Proof. Suppose (M3) is valid, but (M3)' is not, and let X and Y be two bases if Z such that $|X| > |Y|$. Then by (M3), there is an $x \in X \setminus Y$ such that $Y \cup \{x\} \in \mathcal{I}$. Since $x \in X \setminus Y \subseteq X \subseteq Z$, $Y \cup \{x\} \subseteq Z$, contradicting that Y is a basis of Z.

Conversely, suppose (M3)' is valid. If $|X| > |Y|$, the set Y cannot be a basis of $X \cup Y$ as Y is not maximal. Therefore, there exists at least one element $x \in (X \cup Y) \setminus Y = X \setminus Y$ such that $Y \cup \{x\} \in \mathcal{I}$. ∎

Corollary 2.28.1. *Let $(V,\mathcal{I})$ be a matroid and let $X,Y \in \mathcal{I}$. If $|X| > |Y|$, then there exists a subset of $X \setminus Y$ of cardinality $|X| - |Y|$ such that $Y \cup Z \in \mathcal{I}$.*

Proof. By induction on $k = |X| - |Y|$. ∎

(M3) and this corollary are known as the *exchange*, *augmentation*, or *growth* property of matroids.

Algorithm 12 Greedy Algorithm for Matroid Minimisation

```
1: procedure MINIMISE(($V$,$\mathcal{I}$),$w$)
2:     SORT($V$, key = $\lambda t.w(t)$)      ▷ Sort elements by weight, so $w(e_1) \le w(e_2) \le \ldots \le w(e_{|V|})$
3:     $B = \emptyset$
4:     for $i = 1$ to LEN(V) do
5:         if $B \cup \{e_i\}$ then      ▷ Check the next cheapest edge is independent
6:             $B = B \cup \{e_i\}$
7:         end if
8:     end for
9:     return $B$
10: end procedure
```

Theorem 2.29. *The greedy algorithm solves the matroid minimisation problem optimally.*

Proof. Let $B = \{e_{j,1},e_{j,2},\ldots,e_{j,n}\}$ be the solution found by the algorithm. Suppose there is an element $e \in V \setminus B$. If $B \cup \{e\}$ were independent, then this element would have been added to to B in line 6. Since this element was rejected, $B \cup \{e\}$ is not independent, and hence B is a basis.

To prove optimality of B, let $B^* = \{e^*_{j,1},e^*_{j,2},\ldots,e^*_{j,n}\}$ be any optimal solution whose elements are sorted according by weight, as in the algorithm. Without loss of generality, B^* has the longest "prefix" coinciding with B.

Let j_k be the smallest index such that $e_{j,k} \neq e^*_{j,k}$. Since the set $\{e_{j,1},e_{j,2},\ldots,e^*_{j,k}\}$ is independent, $e_{j,k}$ appears before $e^*_{j,k}$ in the order, and hence $w(e_{j,k}) \le w(e^*_{j,k})$.

If $e_{j,k}$ is the last element of B, then $wB \le wB^*$, so B is optimal. Otherwise, $e_{j,k}$ is not the last element of B. Consider the set $B' = \{e_{j,1},e_{j,2},\ldots,e_{j,k}\}$. Since $|B'| < |B^*|$, there exists a set $Z \subseteq B^* \setminus B'$ of cardinality $|B^*| - |B'|$ such that the set $B'' = B' \cup Z$ is independent, and hence a basis. Then, $w(B'') \le w(B^*)$, so B'' is an optimal basis, and this has a longer prefix coinciding with B, contradicting the choice of B^*. ■

Corollary 2.29.1. *An almost identical algorithm solves the matroid maximisation problem optimially.*

An *independence oracle* for an independence system $(V,\mathcal{I})$ is a function $\mathfrak{D} : \mathcal{P}(E) \to \{0,1\}$ defined by

$$\forall F \subseteq V, \mathfrak{D}(F) = \begin{cases} 1 & F \subseteq \mathcal{I} \\ 0 & \text{otherwise} \end{cases}$$

Remark. Because an independence system is determined entirely by V and $\mathcal{I}$, the independence oracle provides enough information to recover the independence system it describes, so we can flip the definition, and say that every independence oracle defines an independence system.

A *basis-superset oracle* is a function $\mathfrak{D} : \mathcal{P}(E) \to \{0,1\}$ defined by

$$\forall B \subseteq V, \mathfrak{B}(B) = \begin{cases} 1 & B \in \mathcal{I} \wedge \neg\exists x \in E : B \cup \{x\} \in \mathcal{I} \\ 0 & \text{otherwise} \end{cases}$$

The greedy algorithm requires sorting the elements of V, which takes $O(|V| \log |V|)$ time. However, more significantly, we need to consult with the basis-superset oracle at every step, so the complexity of the algorithm depends on the complexity of the oracle, given by $O(\mathfrak{D})$.

Theorem 2.30. *Let $(V,\mathcal{I})$ be an independence system. The greedy algorithm solves the maximisation problem optimally for any $w : E \to \mathbb{R}$ if and only if $(V,\mathcal{I})$ is a matroid.*

This theorem allows us to bound how well greedy algorithms can solve certain problems. For instance, the travelling salesman problem is an independence system, but *not* a matroid, so this theorem tells us that a greedy algorithm cannot optimally solve the travelling salesman problem.

Proof. Suppose $V,\mathcal{I}$ is not a matroid. That is, there exists $X,Y \in \mathcal{I}$ with $|X| < |Y|$ such that for all $e \in Y \setminus X$, $X \cup \{e\} \notin \mathcal{I}$.

Let $\epsilon > 0$. Define the weight function by,

$$w(e) := \begin{cases} 1+\epsilon & e \in X & \text{Choose first } |X| \text{ steps} \\ 1 & e \in Y \setminus X & \text{Can't choose} \\ 0 & e \in E \setminus \{X \cup Y\} & \text{Don't change weight} \end{cases}$$

So greedy outputs F with $w(F) = |x|(1+\epsilon) + 0$. So, $w(F) = |X|(1+\epsilon) < w(Y) = |Y|$ for $\epsilon < |Y|/|X| - 1$, a contradiction to $w(F)$ being maximum for all weight functions w. This completees the forward implication.

Now, suppose $(V,\mathcal{I})$ is a matroid. This portion of the proof is similar to the proof of correctness for the greedy algorithm, so we give it more tersely. Let w be an arbitrary weight function, and let $F = \{f_1,f_2,\dots,f_r\}$ be the output of the greedy algorithm. Without loss of generality, suppose $w(f_1) \geq w(f_2) \dots \geq w(f_r)$. Suppose there exists $G \in \mathcal{I}$ such that $|F| < |G|$. By the augmentation property, there exists $g \in G \setminus F$ such that $F \cup \{g\}$ implies there exists t such that $\{f_1,\dots,f_t,g,f_{t+1},\dots,f_s\} = F \cup \{g\} \in \mathcal{I}$ with $w(f_t) \geq w(g) \geq w(f_{t+1})$. We also have $\{f_1,\dots,f_t\} \subseteq \{f_1,\dots,f_t,g\} \in \mathcal{I}$, so g should have been chosen in step $t+1$ of the greedy algorithm. So, F has maximum cardinality by contradiction.

Suppose there exists $G = \{g_1,g_2,\dots,g_r\} \in \mathcal{I}$ such that $w(G) > w(F)$, and $w(g_i) \geq w(g_{i+1})$. So,

$$\sum_{g_i \in G} w(g_i) > \sum_{f_i \in F} w(f_i)$$

so, there exists k such that $w(g_k) > w(f_k)$ since $|G| \leq |F|$. Take $X = \{f_1,f_2,\dots,f_{k-1}\}$ ($= \emptyset$if $k = 1$), and $Y = \{g_1,g_2,\dots,g_k\}$. Clearly, $|X| < |Y|$, so by the augmentation property, there exists $g_t \in Y \setminus X$ with $t \leq k$ such that $\{f_1,f_2,\dots,g_t\} = X \cup \{g_t\} \in \mathcal{I}$. Because $w(g_t) \geq w(g_k) > w(f_k)$, g_t should have been chosen before step k of the greedy algorithm, contradicting correctness, and so G does not exist and hence $w(f)$ is maximum.

This completes the reverse implication. ∎

Given two matroids, $(V,\mathcal{I}_1)$ and $(V,\mathcal{I}_2)$, the *matroid intersection problem* is to find a set $X \in \mathcal{I}_1 \cap \mathcal{I}_2$ such that $|X|$ is maximum.

Theorem 2.31. *Edmonds' algorithm solves the matroid intersection problem. If the matroids are given by independence oracles with maximum complexity T, then the algorithm solves the problem in $O(|V|^3T)$ time.*

The *partition matroid* is defined as follows. Let B_i be a collection of disjoint subsets of V, and let d_i be integers with $0 \leq d_i \leq |B_i|$. Define $I \subseteq V$ to be independent if $|I \cap B_i| \leq d_i$ for each i.

In particular, if $i = 1$ and $B_1 = V$, the partition matroid is the uniform matroid.

Given a bipartite graph $G = (A, \cup B, E)$, define two partition matroids M_A and M_B on E as follows:

M_A: for each vertex $i \in A$, let A_i be the set of edges incident to i and $d_i = 1$.

M_B: for each vertex $i \in B$, let B_i be the set of edges incident to i and $d_i = 1$.

Theorem 2.32. *The maximum matching problem for G coincides with the matroid intersection problem for M_A and M_B.*

Theorem 2.33. *The family of independent sets in a graph G forms a matroid if and only if every connected component of G is a clique.*

Proof. Let G be a graph, of which every connected component is a clique. For a subset $U \subseteq V(G)$, every basis in U contains exactly one vertex in each connected component in $G[U]$ (the subgraph induced by U). Therefore, all bases in U have the same size, and hence the family of independent sets in G forms a matroid, completing the forward implication.

If G contains a connected component with is not a clique, then it contains a subset $U \subseteq V(G)$ inducing a path on 3 nodes. But then U has two bases of size 1 and 2, so the family of independent sets in G do not form a matroid, completing the reverse implication. ■

Theorem 2.34. *Every independence system is the intersection of finitely many matroids.*

Proof. Let C be a circuit of $(V,\mathcal{I})$, and $\mathcal{I}_C$ the family of subsets $A \subseteq E$ such that C is not a subset of A. Then, (V,F_C) is a matroid because,

(M1) $\emptyset \in \mathcal{I}_C$.

(M2) For each $A \subseteq B$, $B \in F \to A \in \mathcal{I}$.

(M3)' All bases of (V,F_C) have size $|V| - 1$.

and $(V,\mathscr{I})$ is the intersection $\bigcap(V,\mathcal{I}_C)$ taken over all circuits C of $(V,\mathscr{I})$. ■

Theorem 2.35. *The problem of finding a maximum independent set in the intersection of 3 matroids is NP-hard.*

3 Polynomial Time Solvability

For many combinatorial optimization problems, polynomial-time algorithms are known. However, there are also many important problems for which no polynomial-time algorithms are known to exist. Although we cannot prove that none exists, we can show that a polynomial-time algorithm for one "hard" problem would imply a polynomial-time algorithm for other "hard" problems.

3.1 Decision Problems

An *alphabet* in the context of formal languages is any set of symbols, often denoted by Σ. A *word* over an alphabet is any finite sequence of letters.

The *Kleene star*, also known as the *free monoid constructor*, is a unary operation, either on sets of strings, or sets of symbols or characters. The application of the Kleene star to a set V is written as V^*.

1. If V is a set of strings, then V^* is defined as the smallest superset of V that contains the empty string, ε, and is closed under string concatenation.
2. If V is a set of symbols or characters, then V^* is the set of all strings over symbols in V, including the empty string ε.

More formally, given a set V, we define the sets

$$V^0 = \{\varepsilon\}$$
$$V^1 = V$$

and recursively define the set,

$$V^{i+1} = \{wv : w \in V^i, v \in V\} \text{ for each } i > 0$$

If V is a formal language, then V^i is a shorthand for the concatenation of V with itself i times. That is, V^i represents the set of all strings that can be represeted as the concatenation of i strings in V. The Kleene star on V is then defined as:

$$V^* = \bigcup_{i \geq 0} V^i$$

The Kleene star is highly important in theoretical computer science, particularly in complexity and computability theory.

Remark. If V is countable, then V^* is the countable union of countable sets, and is hence countable.

The set of all words over an alphabet Σ is then Σ^*. A *formal language* L over an alphabet Σ is a subset of Σ^*.

Let $\{0,1\}^*$ be the set of all binary words, and let $L \subseteq \{0,1\}^*$ be a language. L can be interpreted as a decision problem as follows: given any binary string, decide whether it belongs to L.

Conversely, assuming a fixed efficient encoding, we can encode the input to any problem that can be answered positively or negatively as a binary string, in which case the set of all instances of the problem defines a language X, and the set of "yes" instances defines a subset $Y \subseteq X$.

A *decision problem* is a pair $P = (X,Y)$ where X is a language decidable in polynomial time, and $Y \subseteq X$. The elements of X are called *instances*, the elements of Y are *yes-instances*, and the elements of $X \setminus Y$ are *no-instances*. Decision problems in theoretical computer science are often written in (abbreviated) full capital letters.

An algorithm for a decision problem $P = (X,Y)$ is an algorithm computing the function $f : X \to \{0,1\}$ such that $f(x) = 1$ for $x \in Y$ and $f(x) = 0$ for $x \in X \setminus Y$. For instance, given an undirected graph, encoded as a binary string, we might ask, "Is there a Hamiltonian cycle in G?"

Theorem (Cantor's Diagonal Argument)**.** *There are functions $f : \mathbb{N} \to \{0,1\}$ that cannot be computed by any algorithm.*

Proof. Algorithms are finite sequences of a finite alphabet of possible instructions, so there are countably many possible algorithms, while the set of functions f has size $2^{\aleph_0} = \mathcal{P}(\mathbb{N}) = \mathfrak{c}$ which is uncountable, so no bijection can exist between the sets.

More specifically, by Cantor's Diagonal Argument, $\mathfrak{c}$ is strictly larger than $\aleph_0$, so there are more functions than there are algorithms, as required. ■

An *oracle* is an abstract machine (a generalisation of a function) that is assumed to be able to solve a specific problem (even non-decision problems) in a single operation. The problem is not assumed to even be computable - an oracle is simply a black box that is able to produce a solution for any instance of a given computation program.

A *certificate* or a *witness* is a string that certifies the membership of some string in a language. So, for the Hamiltonian cycle question, a certificate for a graph G would simply be a Hamiltonian cycle: clearly, if you have one, the graph G should be in X.

The class of all decision problems which admit a polynomial time algorithm is called *P* or *PTIME* (for *Polynomial Time*).

In contrast, *NP* (*Non-Deterministic Polynomial time*) is the class of decision problems that admit a polynomial-time certificate-checking algorithm. P is a subclass of NP, as every problem that is solvable in polynomial time can also be checked in polynomial time by just solving the problem. As shown above, Hamiltonian Cycle is NP, and, currently, there does not exist a polynomial time algorithm for Hamiltonial Cycle, so it is not P.

Many decision problems encountered in combinatorial optimisation belong to NP. For many of them, such as Hamiltonian Cycle, it is not known whether they admit polynomial time algorithms. However, we can say that certain problems are not easier than others. This can be formalised through the concept of *polynomial reduction.*

Let $P_1 = (X_1,Y_1)$ and $P_2 = (X_2,Y_2)$ be decision problems. Let $f : X_2 \to \{0,1\}$ with $f(x) = 1$ for $x \in Y_2$ and $f(x) = 0$ for $x \in X_2 \setminus Y_2$. We say that P_1 *polynomially reduces* to P_2 if there exists a polynomial-time algorithm for P_1 using f as an oracle.

Theorem 3.1. *If P_1 polynomially reduces to P_2, and there is a polynomial-time algorithm for P_2, then there is a polynomial-time algorithm for P_1.*

Proof. The oracle for P_2 is queried at most polynomially many times in the polynomial-time algorithm for P_1. If there is a polynomial-time algorithm for P_2, then it can be used as the oracle, so P_1 is the composition of two polynomial-time algorithms, and is hence polynomial-time. ■

Let $P_1 = (X_1,Y_1)$ and $P_2 = (X_2,Y_2)$ be decision problems. We say that P_1 *polynomially transforms* to P_2 if there exists a function $f : X_1 \to X_2$ computable in polynomial time such that $f(x_1) \in Y_2$ for all $x_1 \in Y_1$ and $f(x_1) \in X_2 \setminus Y_2$ for all $x_1 \in X_1 \setminus Y_1$.

A decision problem $\Pi \in$ NP is called *NP-complete* if all other problems in NP polynomially transform to Π. So, to prove a problem is NP-complete, we need to show it is in NP, and to polynomially transform a known NP-complete problem into it.

Conversely, a problem $\Pi \in$ NP is *NP-hard* if all problems in NP polynomially-*reduce* to Π.

3.2 Boolean Satisfiability

In *propositional logic* or *zeroth-order logic*, we deal with statements called *propositions* and *logical connectives* between them. Propositions cannot contain variables, and are therefore either always true, or always false. We also use the symbols, $\top$ and $\bot$, or 1 and 0, for true and false, respectively.

Propositions:

- $2 + 2 = 4$ (always true).
- $2 + 2 = 5$ (always false).
- "Socrates is a man" (always true).
- "Socrates is a dog" (always false).

Non-propositions:

- $x + 2 = 4$ (either true or false, depending on the value of x).
- $0x = 0$ (always true, but not a proposition because it contains a variable).
- $0x = 1$ (always false, but still not a proposition).
- "Socrates" (this is an object and doesn't have a truth value by itself).

Notably, in propositional logic, the proposition "Socrates is a man" is an indivisible atom of truth or falsity that says nothing about "Socrates" or "[being] a man" individually. Because it is an indivisible statement, we can represent the whole proposition with a single letter, for example, p. We cannot, however, represent either individual part alone.

Such an indivisible proposition is called an *atom*, an *atomic formula* or a *literal*. Literals can also be divided into positive and negative *polarities*, where a negative literal is the negation of a positive literal; i.e., "p" is a positive literal, and "$\neg p$" is a negative literal. Positive and negative literals are also called each other's *complementary* literals.

Propositions in isolation are not very interesting. So much so that we often don't even consider specific propositions, and just refer to general ones with letters, often p and q. We can make these propositions slightly more interesting by combining them with *logical connectives* into *compound propositions.*

- *Negation* or *NOT* - the negation of p is written as $\neg p$ or sometimes $\bar{p}$. It is false when p is true, and true when p is false. This is pretty much the same as in normal conversation.
- *(Inclusive) Disjunction*, *Join* or *OR* - the disjunction of p and q is written $p \vee q$, and is true if at least one of p and q is true.

  Note that this is different than how we often use "or" in normal conversation: if I were to, completely truthfully, say "You will give me your wallet, or I will stab you with this rusty kitchen knife", you would be understandably quite upset if you handed me your wallet and still get stabbed. However, to a logician mugger, this would be entirely justified, as the first part of a true inclusive disjunction being true doesn't preclude the second from also being true.
- *Exclusive Disjunction* or *XOR* - the exclusive disjunction of p and q is written as $p \oplus q$ or $p \veebar q$, and is true if exactly one of p or q is true. Exclusive disjunction is not often used in classical logic, but has many important applications, particularly in computing and finite field algebra.

  To indicate exclusive disjunction, we sometimes use the wording, "either p or q", to distinguish it from inclusive disjunction. Now, if you are ever being mugged by a logician, you know what to ask to clarify your chances of being stabbed.
- *Conjunction*, *Meet* or *AND* - the and of p and q is written as $p \wedge q$, and is true when both p is true and q is true. This one is generally the same as in common speech.
- *Material Implication* or *Material Conditional* - This is perhaps the most important connective for proofs, corresponding to the "If... then..." pattern of speech. The implication of p and q is written $p \to q$ or $p \Rightarrow q$. p is called the *antecedent* or *premise*, and q the *consequent* of the implication. The implication is true when (p is true and q is true), or when p is false. In fact, the only way for $p \to q$ to be false, is if p is true, but q is false, so another way to write this is $\neg p \vee q$.

  $p \to q$ being true when p is false but q is true often causes some surprise; after all, if p is false, then how can it claim any credit for q being true? Both statements being false also leading to the compound being true also seems somewhat suspect.

  This surprise might be because in ordinary language, we usually aren't interested in implications where the first proposition is known to be false, so we don't usually think to assign them any truth values. However, one reason why it's nice to define the truth values in this way, is that we often use the implication symbol in this way. For example, we should all agree that the proposition,

  $$\forall x \in \mathbb{Z} : (x > 1) \implies (x^2 > 2)$$

  is true.

  The statement contains infinitely many implications - one for each integer - so included within it is the statement, $0 > 1 \implies 0^2 > 2$, where the antecedent and consequent are both clearly false, but we still say that the proposition is true overall. Because of this, we define $p \to q$ to be true whenever p is false, regardless of the value of q.

  In ordinary language, we often interpret "if... then..." to be the much stronger *biconditional* where it otherwise carries connotations of causality. This is another reason why we define our terms so stringently in mathematics and logic, due to natural language being rife with hidden rules and assumptions*.

I could once again, entirely truthfully, say, “If the moon is made of green cheese, then the world will end at midnight”. It may sound like I know of some mechanism by which a green-cheese moon will cause the end of the world, but I am simply making a trivially true statement by starting with a false premise and violating the implicit assumption that a statement in a conversation should mean something and not just be an exercise in logic.

Conditional propositions like this where the antecedent is false, are called *vacuous truths*, because the proposition is true while not really saying anything meaningful - in particular, we can’t infer anything about the truth value of the consequent from a vacuous truth. These can sometimes cause seemingly incoherent statements to be true. For example, the proposition “All the lights in the room are turned on *and* turned off” is true if there are no lights in the room to begin with. In the equation above, the proposition as a whole is considered to be true non-vacuously, since some integers are indeed greater than 1 and the proposition still holds for them, but we would say that the *cases* where $x < 1$ are vacuously true.

Alternative wordings to “if p then q” include; “p is *sufficient* for q”, because knowing p is true is sufficient information to tell us that q is true; or “q is *necessary* for p”, because q being true is guaranteed by p being true (or equivalently, it is impossible to have p be true without q also being true). We will generally use the “if... then...” pattern in this document, but sufficient and necessary are commonly used in other fields.

- *Material Equivalence*, *Material Biconditional* or *XNOR* - If both $p \to q$ and $q \to p$, such that p and q always share the same truth values, then we write $p \leftrightarrow q$ or $p \Leftrightarrow q$, and say p holds *if and only if* q holds.

  Again, however, this is purely a logical proposition, and no causality between p and q has to be enforced. For example, the compound proposition, “The moon is made of green cheese if and only if $2 + 2 = 5$” is true, despite the lack of connection between green-cheesiness and faulty arithmetic, purely because both sub-propositions are false.

  Alternatives to “p if and only if q” include; “q is necessary and sufficient for p”, which is a combination of the two alternative wordings for material implication; “p precisely/exactly when q”; or the abbreviation, “p iff q”. This last alternative, “iff”, is sometimes regarded as unsuitable for formal writing, so a style guide should be consulted before it is used in such a setting. We will continue to use “if and only if” in this document.

A *valuation* on a Boolean expression is an assignment of truth values to the literals in the expression.

A compound proposition is in *conjunctive normal form* or *CNF* if it is a conjunction of one or more *clauses*, where a clause is a disjunction of atoms; it is an AND of OR statements. A compound proposition is similarly in *disjunctive normal form* or *DNF* if it is the disjunction of one or more clauses, where a clause is a conjunction of atoms; it is an OR of AND statements.

Propositions in CNF:

- p
- $(p \lor \neg q) \land r$
- $(p \lor q) \land (\neg p \lor r) \land q \land (\neg q \lor \neg r)$
- $p \land \neg q \land r \land t \land \neg u \land v$

Propositions not in CNF:

- $(p \land q) \land (q \lor r)$

*Search up “Grice’s maxims” or “the cooperative principle” for an interesting discussion on this topic.

- $(p \vee q) \wedge (q \rightarrow \neg r) \wedge (\neg p \vee r)$
- $(p \vee (q \wedge r)) \wedge (p \vee \neg r)$

Interchanging $\wedge$ and $\vee$ above gives examples of clauses in and not in DNF.

Using the equivalence of material implication and disjunction, along with De Morgan's laws and the distributive laws, it is possible to rewrite any compound proposition in a normal form. However, applying these laws blindly does not necessarily produce the simplest normal form for a compound proposition.

For example,

$$\begin{aligned}(P \to Q) \wedge (\neg P \to Q) &\equiv (\neg P \vee Q) \wedge (P \vee Q)\\ &\equiv (\neg P \wedge P) \vee (\neg P \wedge Q) \vee (Q \wedge P) \vee (Q \wedge Q)\\ &\equiv 0 \vee (\neg P \wedge Q) \vee (Q \wedge P) \vee Q\\ &\equiv (\neg P \wedge Q) \vee (Q \wedge P) \vee Q\end{aligned}$$

Inspecting the clauses closer, we see that Q controls the value of the entire expression, so a simpler CNF for the proposition is just Q.

$$\equiv Q$$

We should really draw out a truth table to prove this formally, but it should be clear enough that this is true.

Theorem 3.2. *There is a polynomial time algorithm to reduce any Boolean expression to a DNF and CNF representation.*

Given a CNF C, the *satisfiability problem* or *SAT* is to determine if there is a valuation such that C evaluates to true.

Theorem (Cook)**.** *SAT is NP-complete.*

The satisfiability problem restricted to instances where each clause contains at most three literals is called 3-*SAT*.

Theorem 3.3. *3-SAT is NP-complete.*

Proof. Clearly, 3-SAT belongs to NP. To prove completeness, we show that SAT polynomially transforms to 3-SAT.

Let $Z = (x_1 \vee x_2 \vee \ldots \vee x_k)$ be a clause containing $k > 3$ literals. Transform Z as follows:

$$(x_1 \vee x_2 \vee \ldots \vee x_k) \mapsto (x_1 \vee x_2 \vee \ldots \vee x_{k-1} \vee u) \wedge (\neg u \vee x_{k-1} \vee x_k)$$

where u is a new variable.

Suppose there is an assignment φ satisfying the original CNF. If Z is satisfied by one of the first $k-2$ literals, then by defining, $\varphi(u) = 0$, we extend φ to an assignment satisfying the transformed CNF. If Z is satisfied by x_{k-1} or x_k, we define $\varphi(u) = 1$ and obtain an assignment satisfying the transformed CNF.

Conversely, suppose there is an assignment φ satisfying the transformed CNF. If $\varphi(u) = 0$, then Z is satisfied by one of the first $k-2$ literals. If $\varphi(u) = 1$, then Z is satisfied by x_{k-1} or x_k.

So, an assignment φ satisfies Z if and only if it satisfies the transformed CNF.

Because a Boolean formula contains finitely many terms, this algorithm always terminates. Applying this transformation repeatedly, the original CNF can be transformed into an instance of 3-SAT which is satisfiable if and only if the original one is. ■

Theorem 3.4. *2-SAT is P.*

Proof. The implication $a \to b$ is logically equivalent to $\neg a \vee b$, so, in 2-SAT, the clause $x_1 \vee x_2$ is equivalent to the pair of implications $\neg x_1 \to x_2$ and $\neg x_2 \to x_1$. If x_1 is true, then x_2 must be true, and if x_2 is false, then x_1 must be false.

These implications are straightforward, so we just follow every possible implication chain and see if we ever derive both $\neg x$ from x or x from $\neg x$. If we do for some x, then the 2-SAT formula is unsatisfiable. Otherwise, it is satisfiable. The number of possible implication chains is polynomially bounded in the size of the input formula, so they are checkable in polynomial time. ■

Remark. With 3-SAT, we can express implications of the form $a \to b \vee c$, where a,b,c are literals. Now, if a is true, then one or both of b and c are true, but we don't know which. In this case, we have to do case analysis, and combinatorial explosion occurs.

Theorem 3.5. *HAMILTONIAN-CYCLE is NP-complete.*

Proof. Membership in NP is obvious. To prove completeness, we polynomially transform 3-SAT to HAMILTONIAN-CYCLE.

Let C be a CNF with clauses $Z_1,Z_2,\dots,Z_m$ over the set of variables $X = \{x_1,x_2,\dots,x_n\}$, with each clause containing three variables. We note that there are 2^n possible valuations on C. We model these 2^n possible valuations using a digraph with 2^n different Hamiltonian cycles.

Construct n paths $P_1,P_2,\dots,P_n$ corresponding to the n variables, each consisting of $2k$ nodes, $P_i = (v_{i,1},v_{i_2}\dots,v_{i,2k})$. We add edges from $v_{i,j-1}$ to $v_{i,j}$ on P_i corresponding to the assignment $x_i =$ true (picturing the paths as lying left to right, these edges point left to right). We add edges from $v_{i,j}$ to $v_{i,j-1}$ on P_i corresponding to the assignment $x_i =$ false (right to left). Next, add the edges $(v_{i,1},v_{i,k}) \times (v_{i+1,1},v_{i+1,k})$.

Next, add a source node s and target node node t, and connect s to $v_{1,1}$ and $v_{i,k}$, and connect t to $v_{n,1}$, $v_{n,k}$, and s.

Next, add a node $C_1,C_2,\dots,C_m$ for each clause. If a clause C_j contains the variable x_i, connect C_j to $x_{i,2j-1}$ and $x_{i,2j}$, left to right (add edges $(x_{i,2j-1},C_j)$, and $(C_j,x_{i,2j})$) if C_j contains the positive literal x_i, and right to left (add edges $(x_{i,2j},C_j)$, and $(C_j,x_{i,2j-1})$) if C_j contains the negative literal $\neg x_i$.

Any Hamiltonian cycle in the graph traverses P_i either from right to left, or left to right, because any path entering a node $v_{i,j}$ has to exit from $v_{i,j+1}$ either immediately, or via one clause-node in between, in order to maintain the Hamiltonian property. Similarly, all paths entering at $v_{i,j-1}$ must exit from $v_{i,j}$.

Note that this graph can be constructed in polynomial time.

Since each path P_1 can be traversed in two possible ways, and we have n paths mapping to n variables, there can be 2^n Hamiltonian cycles in the graph $G \setminus \{C_1,C_2,\dots,C_k\}$, each corresponding to a different valuation of $x_1,x_2,\dots,x_n$.

If there exists a Hamiltonian cycle H in G

- If H traverses P_i from left to right, assign $x_i =$ true;
- If H traverses P_i from right to left, assign $x_i =$ false.

Since H visits each clause node C_j, at least one of the P_i was traversed in the correct direction relative to the node C_j, so the assignment obtained here satisfies the given 3-CNF.

Conversely, if there exists a satisfying assignment for the 3-CNF, select the path that traverses P_i from left to right if $x_i =$ true, or right to left if $x_i =$ false, including the clause nodes whenever possible. Connect the source to P_1, P_2 to t, and P_i to P_{i+1} appropriately so as to maintain the continuity of the path, then connect t to s to complete the cycle. Since the assignment is such that every clause is satisfied, the clause-nodes are included in the path. The P_i nodes, s and t are all included, and all the paths are traversed in one direction only so no node is repeated twice, so the path obtained is a Hamiltonian Cycle. ■

Theorem 3.6. *MAXIMUM-INDEPENDENT-SET is NP-complete.*

Proof. Membership in NP is obvious. To prove completeness, we polynomially transform SAT to MAXIMUM-INDEPENDENT-SET.

Let Z be a collection of clauses $Z_1,Z_2,\dots,Z_m$ over the set of variables $X = \{x_1,x_2,\dots,x_n\}$, with each Z_i containing k_i literals $(\beta_{i,1} \vee \beta_{i,2} \vee \dots \vee \beta_{i,k})$. We construct a graph G such that G has an independent set of size m if and only if there is a truth assignment satisfying all m clauses.

For each clause Z_i, we introduce a clique of k_i vertices, one vertex per literal. Two vertices in different cliques (clauses) are connected by an edge if and only if they represent the same variable, but of different polarity.

Suppose G has an independent set S of size m. Then each of these cliques contains exactly one vertex. Setting each of these literals to be true, we obtain an assignment satisfying all m clauses since no two literals in S are in conflict.

Conversely, if there is a truth assignment satisfying all m clauses, then we choose a true literal out of each clause. The set of corresponding vertices defines an independent set in G of size m. ■

Theorem 3.7. *MAXIMUM-INDEPENDENT-SET is NP-complete for graphs of degree at most* 3.

Proof. If G has a vertex x of degree at least 4, we apply vertex splitting with $|Y| = 2$. In the transformed graph, x' has degree 2, y has degree 3, and z has degree $\deg(x) - 1$. Repeated applications of vertex splitting transforms G into a graph of vertex degree at most 3, and clearly this transformation is polynomial. ■

Theorem 3.8. *MINIMUM-VERTEX-COVER and MAXIMUM-CLIQUE are NP-complete.*

The *travelling salesman problem* (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

That is, given a weighted complete graph (K_n,w), find a Hamiltonian cycle of minimum weight.

Theorem 3.9. *TSP is NP-hard.*

Proof. We give a reduction from HAMILTONIAN-CYCLE.

Let G be an instance of HAMILTONIAN-CYCLE. Construct an instance G' of TSP as follows: $V(G') = V(G)$ with every two vertices of G' being adjacent. Define $w((u,v)) = 1$ if $(u,v) \in E(G)$ and $w((u,v)) = 2$ otherwise. Then G has a Hamiltonian cycle if and only if the optimum tour in G' has length n. ■

3.3 Approximation Algorithms

An *absolute approximation algorithm* for an optimization problem P is a polynomial-time algorithm A for P for which there exists a constant k such that $|A(I) - \mathrm{Opt}(I)| \leq k$ for all instances I of P, where $A(I)$ is the size of the solution found by the algorithm A and $\mathrm{Opt}(I)$ is the size of an optimal solution.

Let P be an optimization problem with non-negative weights and $k \geq 1$. A *k-factor approximation algorithm* for P is a polynomial-time algorithm A for P such that $1/k\,\mathrm{Opt}(I) \leq A(I) \leq k\,\mathrm{Opt}(I)$ for all instances I of P. We also say that A has *performance ratio k*.

The first inequality applies to maximization problems and the second one to minimization problems.

A 1-factor algorithm is an exact polynomial-time algorithm.

Theorem 3.10. *There is no k such that the greedy algorithm for VERTEX-COVER is a k-factor approximation algorithm.*

Proof. Let $p \in \mathbb{N}$ and G_p be a graph with vertex set $V(G) = A \cup B \cup C$, where $|A| = |B| = p$. For each $i \in [2,3,\ldots,p-1]$, split the vertices of B into $\lfloor p/i \rfloor$ groups and for each group introduce the vertex of A adjacent to the vertices of that group. The algorithm may first delete the vertices of A, in which case the size of the solution is $|A| + p$. On the other hand B is an optimal solution of size p, and the ratio $|A|/p + 1$. ∎

Theorem 3.11. *There is a 2-factor approximation algorithm for VERTEX-COVER.*

Proof. Let M be a maximum matching in G. Then the set of vertices covered by M is a vertex cover containing $2|M|$ vertices. On the other hand, any vertex cover must contain at least $|M|$ vertices, so $|M| \leq \tau(G) \leq 2|M|$, where $\tau(G)$ is the size of a minimum vertex cover in G. Therefore, $2|M|/\tau(G) \leq 2$, so this algorithm is a 2-factor approximation. ∎

3.4 Chromatic Numbers

A *vertex colouring* of G is a mapping $f : V(G) \to \mathbb{N}$ with $f(u) \neq f(v)$ for all $(u,v) \in E(G)$.

In other words, in a vertex colouring, every colour class is a independent set, so vertex colouring is a partition of $V(G)$ into independent sets.

A *edge colouring* of G is a mapping $f : E(G) \to \mathbb{N}$ with $f(e) \neq f(e')$ for all edges e and e' incident to the same vertex.

Remark. An edge colouring of G is equlvalent to a vertex colouring of the line graph of G.

Given an undirected graph G, the *vertex colouring problem* is to find a vertex colouring of G with minimum colours. The optimum value of the vertex colouring problem for G is called the *chromatic number* of G, denoted $\chi(G)$.

Given an undirected graph G, the *edge colouring problem* is to find an edge colouring with minimum colours. The optimum value of the edge colouring problem for G is called the *edge-chromatic number* or *chromatic index* of G, denoted $\chi'(G)$.

Theorem 3.12. *The following decision problems are NP-complete for any fixed value $k \geq 3$:*

1. *Decide whether a given graph has a chromatic number at most k.*
2. *Decide whether a given graph has a chromatic index at most k.*

Moreover, (1) is NP-complete even for planar graphs of vertex degree at most 4, and (2) is NP-complete for graphs of vertex degree at most 3.

Theorem 3.13. *Both problems can be solved in polynomial time for $k = 1,2$.*

Proof. $\chi(G) = 1$ if and only G has no edges. $\chi(G) = 2$ if and only if G is bipartite. In both caes, the problem can be solved in polynomial time. The chromatic index of G is at most 2 if and only if the chromatic number of $L(G)$ is at most 2. ∎

Theorem 3.14. *For any graph G,*

$$\chi'(G) \geq \max_{v \in V(G)} \deg(v)$$

Proof. To reduce clutter, define $\triangle(G) := \max_{v\in V(G)} \deg(v)$. We induct $|E(G)|$.

Let $\triangle(G) = k$, and let $e = (u,v) \in E(G)$. By the induction hypothesis, $G \setminus \{e\}$ has an edge colouring f with k colours. Since the degree of u and v is strictly less than k in $G \setminus \{e\}$, there is a colour $i \in \{1, \ldots, k\}$ which is missing at u, and a colour $j \in \{1, \ldots, k\}$ which is missing at v. If $i = j$, we can assign this colour to the edge $e = (u,v)$ in G.

Otherwise, we consider the subgraph H of $G \setminus \{e\}$ formed by the edges of colour i and j. Every vertex of H has degree at most 2, and hence every connected component of H is either a path or a cycle. Each of u and v has degree 1 in H (degree 2 is not possible because each of them misses one of the two colours; degree 0 would allow to use the same argument as when both of them miss the same colour). Therefore, the connected component of H containing u is a path, and the connected component of H containing v is a path, and these two paths are different, otherwise we would have $i = j$. But now we can exchange the colours on the path containing u, and assign colour j to the edge e in G. ■

Theorem (Vizing). *For any graph G,*

$$\triangle(G) \leq \chi'(G) \leq \triangle(G) + 1$$

Corollary 3.14.1. *The edge colouring problem admits an absolute approximation algorithm on simple graphs.*

Let $\omega(G)$ denote the size of a maximum clique in G.

Theorem 3.15. *For any graph G,*

$$\omega(G) \leq \chi(G) \leq \triangle(G) + 1$$

Proof. Since the vertices of any clique must have pairwise different colours in any proper colouring of G, we must have $\omega(G) \leq \chi(G)$.

For the second inequality, let $V(G) = \{v_1, \ldots, v_n\}$, and let S be a set of $\triangle(G) + 1$ colours. Assign any colour from S to v_1, and then proceed by induction as follows: for each i, vertex v_i has at most $\triangle(G)$ neighbours among $v_1, \ldots, v_{i-1}$, and hence at least one colour from S is missing among the neighbours of v_i. Assign this colour to v_i, and proceed to v_{i+1}. ■

Corollary 3.15.1. *A vertex colouring of G with $\triangle(G) + 1$ colours can be found in linear time.*

Theorem (Brooks). *If G is a connected graph which is neither complete nor an odd cycle, then $\chi(G) \leq \triangle(G)$.*

Corollary 3.15.2. *Every connected graph of vertex degree at most 3 is 3-colourable, except for K_4.*

A graph is *planar* if it can be drawn on the plane in such a way that no edges cross each other. A *face* of a planar graph is a maximal section of the plane in which any two points can be joined by a curve that does not intersect any part of G. The *degree* of a face is the number of edges in the boundary surrounding the face.

Theorem (Euler). *Let G be a connected planar graph with n vertices, m edges, and f faces. Then,*

$$n - m + f = 2$$

Proof. By induction on m. For $m = 0$, $G = K_1$, a graph with 1 vertex and 1 face. Suppose the formula is true for any connected planar graph with fewer than m edges, and let G have m edges. If G is a tree, then $m = n - 1$ and $f = 1$ and the formula holds. Otherwise, if G is not a tree, consider a cycle C, and an edge $e \in C$. The graph $G \setminus \{e\}$ is connected, has the same number of vertices, one edge fewer, and one face fewer. By the induction hypothesis, in $G \setminus \{e\}$, we have $n - (m - 1) + (f - 1) = 2$. Thereefore in G, we have $n - m + f = 2$. ■

Corollary 3.15.3. *If G is a connected planar graph with $n \geq 3$ vertices and m edges, then $m \leq 3n - 6$. If G is additionally triangle-free, then $m \leq 2n - 4$.*

Proof. If we trace the boundary of all faces, we encounter each edge exactly twice. Denoting the number of faces of degree k by f_k, we conclude that,

$$\sum_k k f_k = 2m$$

Since the degree of any face in a simple planar graph is at least 3, we have,

$$\begin{aligned} 3f &= 3\sum_{k\geq 3} f_k \\ &= \sum_{k\geq 3} k f_k \\ &= 2m \end{aligned}$$

Together with Euler's formula, this proves $m \leq 3n - 6$.

If G is additionally triangle-free, then,

$$\begin{aligned} 4f &= 4\sum_{k\geq 4} f_k \\ &= \sum_{k\geq 4} k f_k \\ &= 2m \end{aligned}$$

and therefore $m \leq 2n - 4$. ■

Corollary 3.15.4. *K_5 and $K_{3,3}$ are not planar.*

Proof. For K_5, we have $m > 3n - 6$, since $n = 5$ and $m = 10$, so K_5 is not planar. $K_{3,3}$ is triangle-free, and we have $m > 2n - 4$, since $n = 6$ and $m = 9$, so $K_{3,3}$ is not planar. ■

Corollary 3.15.5. *Every planar graph has a vertex of degree at most 5.*

Proof. Let G be a connected planar graph with n vertices and m edges. Then,

$$\begin{aligned} \sum_{v\in V(G)} \deg(v) &= 2m \\ &\leq 2(3n-6) \\ &= 6n - 12 \end{aligned}$$

and hence G has a vertex of degree at most 5. ■

A graph H is a *minor* of G if H can be obtained from G by vertex deletions, edge deletions, and edge contractions.

Theorem (Kuratowski)**.** *A graph G is planar if and only if G does not contain K_5 nor $K_{3,3}$ as minors.*

Theorem (Six Colour)**.** *Every planar graph G can be vertex coloured with at most 6 colours.*

Proof. We induct on the number of vertices. Obviously, every graph with at most 6 vertices is 6-colourable.

If a planar graph has more than 6 vertices, delete from G any vertex v. By Corollary 3.15.5, this vertex has degree at most 5. By the induction hypothesis, $G \setminus \{v\}$ is 6-colourable. Then, then neighbours of v use at most 5 different colours, so the unused colour can be used to colour v, and G is 6-colourable. ■

Theorem (Five Colour). *Every planar graph G can be vertex coloured with at most 5 colours.*

Proof. We induct on the number of vertices. Obviously, every graph with at most 5 vertices is 5-colourable.

If a planar graph has more than 5 vertices, then by Corollary 3.15.5, there exists a vertex v of degree at most 5. Delete from G this vertex to form $G' = G \setminus \{v\}$. By the induction hypothesis, G' is 5-colourable. If the neighbours of v do not use all 5 different colours, the unused colour can be used to colour v, and G is 5-colourable. Otherwise, consider the vertices v_1,v_2,v_3,v_4,v_5 adjacent to v in cyclic order (which will depend on how we draw G), coloured with colours 1,2,3,4, and 5, respectively.

Consider the subgraph $G_{1,3}$ of G' consisting of the vertices coloured with colours 1 and 3 only, and the edges connecting them (this is called a *Kempe chain*). If v_1 and v_3 lie in different connected components of $G_{1,3}$, we can swap the 1 and 3 colours on the connected component containing v_1 without affecting the colouring of the rest of G'. This frees colour 1 for v, completing the task. If v_1 and v_3 lie in the same connected component of $G_{1,3}$, then we can find a path in $G_{1,3}$ consisting of only colour 1 and 3 vertices.

Now consider the subgraph $G_{2,4}$ of G' consisting of the vertices coloured with colours 2 and 4 only, and the edges connecting them, and appply the same arguments as before. We are then either able to reverse the 2-4 colouration on the subgraph of $G_{2,4}$ containing v_2 and colour v colour 2, or we can connect v_2 and v_4 with a path that consists of only colour 2 and 4 vertices. Such a path would necessarily intersect the 1-3 coloured path constructed before, since the vertices were given in cyclic order, contradicting the planarity of G. ■

Theorem (Four Colour). *Every planar graph G can be vertex coloured with at most 4 colours.*

Remark. The Four Colour Theorem was one of the first major theorems to be proved with significant computer assistance.

If the Four Colour Theorem were false, then there would exist a minimal counterexample. After reducing the possibilities with various mathematical techniques, the remaining configurations were checked using a computer, taking over a thousand computer core hours to finish.

3.5 Bin Packing

The *knapsack problem* (KNAPSACK) is as follows: given a set of items, each with a weight and a value, determine which items to include in the collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Theorem 3.16. *Deciding the knapsack problem ("Can a value of at least V be achieved without exceeding weight W?") is NP-complete.*

The *subset sum problem* (SSP) is as follows: given integers $C = (c_1,c_2,\dots,c_n)$ and a target number T, decide if there is a subset $S \subseteq \{1,\dots,n\}$ such that

$$\sum_{i \in S} c_j = T$$

SSP is a special case of KNAPSACK.

Theorem 3.17. *SSP is NP-hard. If $T = 0$, the problem is NP-complete. If the integers in C are all positive, then the problem is NP-complete.*

The *partition problem* is a variant of SSP where all inputs are positive, and the target sum is exactly half the inputs. Or equivalently,

$$\sum_{i \in S} c_j = \sum_{i \in \{1,\dots,n\} \setminus S} c_j$$

Theorem 3.18. *PARTITION is NP-complete.*

Suppose we have n objects, each of a given size, and some bins of equal capacity. We want to assign the objects to the bins, using as few bins as possible. Of course, the total size of the objects assigned to one bin should not exceed its capacity. Without loss of generality, we assume that the capacity of each bin is 1.

Given a list of non-negative numbers $a_1, a_2, \dots, a_n \leq 1$, the *bin packing problem* is to find a natural k and an assignment $f : \{1,\dots,n\} \to \{1,\dots,k\}$ with,

$$\sum_{\{i : f(i) = j\}} a_i \leq 1$$

for all $j \in \{1,\dots,k\}$, such that k is minimum.

Theorem 3.19. *The following problem is NP-complete: given an instance I of Bin Packing, decide whether I has a solution with two bins.*

Proof. Membership in NP is obvious. To prove completeness, we transform PARTITION by choosing

$$a_i = \frac{2c_i}{\sum_{i=1}^{n} c_i}$$

■

Corollary 3.19.1. *It is NP-complete to distinguish whether the optimal solution is 2 or 3, and hence for any $\epsilon > 0$, there is no $(3/2 - \epsilon)$-factor approximation algorithm for Bin Packing.*

In the *online* variant of the problem, items arrive one after another, and the irreversible decision of where to place an item has to be made before knowing the next item, or even if there will be another one.

Most algorithms follow the same general pattern: if the next item fits in one of the currently open bins, put it in one of the bins. Otherwise, open a new bin and put the new item in it. These algorithms differ in the criterion by which they choose the open bin for the new item in the first step.

One algorithm for the online variant of Bin Packing is the *next fit algorithm.* In next fit, we always keep a single open bin. When the new item doesn't fit into it, it closes the current bin, and opens a new bin. Its advantage is that it is a bounded-space algorithm, since it only needs to keep a single open bin in memory.

Algorithm 13 Next Fit

```
1:  procedure NF($C$)
2:      $k = 1$
3:      $S = 0$
4:      for $i = 1$ to $n$ do
5:          if $S + c_i > 1$ then
6:              $k = k + 1$
7:              $S = 0$
8:          end if
9:          $f(i) = k$
10:         $S = S + c_i$
11:     end for
12:     return $k$,$f$
13: end procedure
```

Let NF(I) denote the number of bins found by the next fit algorithm, OPT(I) be the minimum number of bins, and $\Sigma(I) = \sum_i c_i$

Theorem 3.20.

$$\mathrm{NF}(I) \le 2\lceil \Sigma(I) \rceil - 1 \le 2\,\mathrm{OPT}(I) - 1$$

Proof. Let $k = \mathrm{NF}(I)$, and let f be the assignment found by the next fit algorithm. For $j \in \{1, \ldots, \lfloor k/2 \rfloor\}$, we have,

$$\sum_{\left\{i:f(i)\in\{2j-1,2j\}\right\}} c_i > 1$$

Adding these inequalities, we obtain $\lfloor k/2 \rfloor < \Sigma(I)$. Since the left side is an integer, we conclude that $(k-1)/2 \le \lfloor k/2 \rfloor \le \lceil \Sigma(I) \rceil - 1$, proving $k \le 2\lceil \Sigma(I) \rceil - 1$. The second inequality follows from the obvious fact that $\lceil \Sigma(I) \rceil \le \mathrm{OPT}(I)$. ■

Corollary 3.20.1. *Next fit is a* 2*-factor approximation algorithm for BIN-PACKING.*

The *next-k-fit algorithm* (NkF) keeps the last k bins open, and chooses the first bin in which the item fits. It is therefore a k-bounded-space algorithm. For $k \ge 2$, NkF gives better results than NF, but increasing k to constant values large than 2 improves the alorithm no further in its worst-case behaviour.

The *first fit algorithm* keeps all bins open, in the order they were opened, attempting to place new items into the first bin in which it fits.

Algorithm 14 First Fit

```
1: procedure FF($C$)
2:     for $i = 1$ to $n$ do
3:         $f(i) = \min\left\{ j \in \mathbb{N} : \sum_{\{k<i:f(k)=j\}} a_k + a_i \le 1 \right\}$
4:         $k = \max_{i\in\{1,\ldots,n\}} f(i)$
5:     end for
6:     return $k$,$f$
7: end procedure
```

Theorem 3.21. $\mathrm{FF}(I) \le 7/4\,\mathrm{OPT}(I)$.

The *first fit decreasing algorithm* (FFD) sorts the items by descending size, then calls first fit. FFD requires being able to see the entire list first and thus solves the *offline* variant of Bin Packing.

Algorithm 15 First Fit Decreasing

```
1: procedure FFD($C$)
2:     SORT(C)
3:     return NF(C)
4: end procedure
```

Theorem 3.22. *FFD is a 3/2-factor approximation algorithm for Bin Packing.*

Proof. Let I be an instance of the problem and let $k = \mathrm{FFD}(I)$. Consider the jth bin for $j = \lceil 2k/3 \rceil$. If it contains an item of size greater than 1/2, then each bin with smaller index did not have space for this item. Therefore, each of these bins has been assigned an item before. As the items are considered in non-increasing order, there are at least j items of size greater than 1/2. Thus, $\mathrm{OPT}(I) \geq j \geq 2k/3$.

Otherwise, the jth bin and hence each bin with greater index contains no item of size greater than 1/2. Therefore, the bins $j,j+1,\ldots,k$ contain at least $2(k-j)+1$ items, none of which fit into bins $1,\ldots,j-1$. Thus,

$$\begin{aligned}\Sigma(I) &> \min\{j-1, 2(k-j)+1\} \\ &\geq \min\left\{\left\lceil\frac{2k}{3}\right\rceil - 1, 2\left(k - \left(\frac{2k}{3}+\frac{2}{3}\right)\right)+1\right\} \\ &= \left\lceil\frac{2k}{3}\right\rceil - 1\end{aligned}$$

since $\mathrm{OPT}(I) \geq \Sigma(I) > \lceil 2k/3 \rceil - 1$, $\mathrm{OPT}(I) \geq \lceil 2k/3 \rceil \geq 2k/3$. ■

3.6 Steiner Trees

Let G be an undirected graph, and let $T \subseteq V(G)$. A *Steiner tree* for T is a set S such that $T \subseteq V(S) \subseteq V(G)$ and $E(S) \subseteq E(G)$. The elements of T are called *terminals*, and the elements of $V(G) \setminus T$ are the *Steiner points* of S.

Given an undirected weighted graph (G,w) and a set $T \subseteq V(G)$, the *Steiner tree problem* is to find a Steiner tree S for T of minimum weight.

MST ($T = V(G)$) and SHORTEST-PATH ($|T| = 2$) are special cases of STEINER-TREE solvable in polynomial-time.

Theorem 3.23. *STEINER-TREE is NP-hard, even for unit weights.*

Proof. We give a transformation from MINIMUM-VERTEX-COVER, which is known to be NP-complete.

Given a graph G, we transform it to a graph H by adding, for each edge $(u,v) \in E(G)$ a new vertex $x_{u,v}$ which is adjacent to both u and v, and by adding edges which are missing in G.

We set $w(e) = 1$ for all edges $e \in E(H)$, and set $T = \{x_{u,v} : u,v \in E(G)\}$. We will show that G has a vertex cover of size k if and only if H has a Steiner tree for T with $k + |E(G)| - 1$ edges.

Let $T \cup X$ be the set of vertices of a Steiner tree S in H, where $X \subseteq V(G) \subseteq V(H)$. The set T is independent in H as these vertices have neighbours only among the vertices of G. We also have that S is a connected graph, so every vertex of T has a neighbour in X. This means that X is a vertex cover in G, and $E(S) = |T| + |X| - 1 = |E(G)| + |X| - 1$, completing the forward implication.

Conversely, let X be a vertex cover in G. We can connect the vertices of X in the graph H by $|X|-1$ edges. Since every edge of G is covered by a vertex of X, every vertex of T is connected by an edge to a vertex of X in H. The $|X|-1$ edges connecting X and the $|T|$ edges incident to the vertices of T create a Steiner tree with $|T|+|X|-1 = |E(G)|+|X|-1$ edges, as required, thus completing the reverse implication. ■

Let (G,w) be a weighted graph with all weights positive. The *metric closure* of (G,w) is the pair (G^*,c^*), where G^* is the graph with $V(G^*) = V(G)$ in which two vertices x and y are adjacent if and only if they are connected in G by a path and $w^*(xy)$ equals the length of a shortest path between x and y in G.

Remark. w^* is symmetric, point separating, and satisfies the triangle inequality, thus defining a metric on G^*.

Theorem 3.24. *Let (G,w) be a weighted graph with all weights positive, let (G^*,w^*) be its metric closure, and let $T \subseteq V(G)$. If S is an optimum Steiner tree for T in G, and M is a minimum spanning tree in $G^*[T]$, then $w^*(E(M)) \leq 2w(E(S))$.*

Proof. Consider the graph H containing two copies of each edge of S. Then, H is Eulerian and hence contains an Eulerian walk W in H. This walk defines a Hamiltonian cycle W' in $G^*[T]$. Since w^* satisfies the triangle inequality,

$$\begin{aligned} w^*(W') &\leq w(W) \\ &= w(E(H)) \\ &= 2w(E(S)) \end{aligned}$$

However, we also have $w^*(E(M)) \leq w^*(W')$ since by deleting one edge of W' we obtain a spanning tree in $G^*[T]$. ■

This suggests the following 2-factor approximation algorithm for STEINER-TREE.

Algorithm 16 Steiner Tree

```
1: Compute the metric closure $(G^*,w^*)$
2: Compute the shortest path $P_{s,t}$ for all $s,t \in T$
3: Find a minimum spanning tree $M$ in $G^*[T]$
4: $E(S) = \bigcup_{(u,v)\in E(M)} P_{u,v}$
5: $V(S) = \{R \subseteq V(G) : (\forall v \in R : (\exists(u,v) \in E(S)) \vee (\exists(v,u) \in E(S)))\}$
6: return A minimal connected subgraph of $S$
```

Theorem 3.25. *This algorithm is a 2-factor approximation for Steiner Tree and can be implemented in $O(|V(G)|^3)$ time.*

Recall that TSP is NP-hard.

Theorem 3.26. *Unless $P = NP$, there is no k-factor approximation algorithm for TSP for any $k \geq 1$.*

Proof. We will show that a k-factor approximation algorithm A for TSP implies that HAMILTONIAN-CYCLE (which is NP-complete) can be solved in polynomial time.

Given an instance G of HAMILTONIAN-CYCLE, we construct an instance G^* of TSP with $n = |V(G)|$ nodes and distances $w((i,j))$ as follows: if i is adjacent to j, then $w((i,j)) = 1$; otherwise, $w((i,j)) = 2 + (k-1)n$.

Now, we apply A to the constructed instance of TSP. If the returned tour has length n, then this tour is a Hamiltonian cycle in G. Otherwise the returned tour has length at least $(n-1)+2+(k-1)n = kn+1$. Assuming that A is a k-factor approximation algorithm, we conclude that $(kn+1)/\operatorname{OPT}(G^*) \geq A(G^*)/\operatorname{OPT}(G^*) \geq k$, where $\operatorname{OPT}(G^*)$ is the length of the optimum tour. Hence, $\operatorname{OPT}(G^*) \geq n+1/k > n$, showing that G has no Hamiltonian cycle. ■

Metric TSP, also known as Δ-TSP, is TSP such that the underlying graph is its own metric closure. That is, given a positive weighted complete graph (K_n,w) with $w : (E(K_n)) \to \mathbb{R}_{\geq 0}$ satisfying $w(x,z) \leq w((x,y)) + w(y,z)$ for all $x,y,z \in V(K_n)$, find a Hamiltonian cycle of minimum weight.

Theorem 3.27. *METRIC-TSP is NP-hard.*

The greedy algorithm works badly in this problem and is not a k-factor approximation algorithm. However, the *double tree algorithm* has better performance on this problem.

Algorithm 17 Double Tree

```
1: Find a minimum weight spanning tree $T$ in $K_n$ with respect to $w$.
2: Walk around the tree, doubling each edge to create a Eulerian walk (circuit).
3: In the Eulerian walk, ignore all but the first occurence of each vertex.
4: return the tour constructed in line 3.
```

Theorem 3.28. *The double tree algorithm is a 2-factor approximation algorithm for METRIC-TSP.*

Proof. Clearly the algorithm is polynomial. Also, we have $w(E(T)) \leq \operatorname{OPT}(K_n,c)$, since by deleting an edge from any tour we obtain a spanning tree. Finally, the solution found by the algorithm is of weight at most $2w(E(T)) \leq 2\operatorname{OPT}(K_n,c)$. ■