ESM III A - lecture

Patrick Bangert

Chapter 1

Notation

Below are the symbols and notations used in the lecture.

Symbol	Explanation
$\forall$	for all
$\exists$	exists
$\in$	in, belongs to
$\notin$	not in, doesn't belong to
$\equiv$	congruent, equivalent
$\Rightarrow$	implies
$=$	equals
$\subset$	subset, included in
&	and
T / F	true / false
$\downarrow$	neither... nor ...
$-$	minus, negation
$\cap$	intersection, and
$\cup$	union, or

Symbol	Explanation
iff	if and only if
$\emptyset$	empty set
$\mathbb{P}(X)$	power set of X
$\times$	cartesian product
dom	domain
ran	range
$\circ$	composition of maps
$\|A\|$	number of elements in A, cardinality
xRy	x is in relation with y
$\leq$	less or equal, partial order relation
!	factorial
$\rightarrow$	if... then ...
$\leftarrow$	assignment operation
$\square$	q.e.d.

axiom!of extension
Zermelo-Fraenkel
transitive
anty-symmetric
symmetric

Chapter 2

Set Theory

Axiom 1. (Axiom of extension)
Two sets are equal if and only if they have the same elements.

That is, $\forall x\ (x \in a \ \equiv \ x \in b) \Rightarrow a = b$. First axiom of Zermelo-Fraenkel.

$A \subset B$ if every element of A is an element of B, thus this includes $A = B$.

$$A \subset B \ \& \ B \subset C \ \Rightarrow \ A \subset C \text{ (transitive)} \tag{2.1}$$

$$\text{if } A \subset B \ \& \ B \subset A \ \Rightarrow \ A = B \text{ (anty-symmetric)} \tag{2.2}$$

$$A = B \ \equiv \ B = A \text{ (symmetric)} \tag{2.3}$$

Could formulate the Axiom of extension using anti-symmetry and os convenient to prove equality this way. Notation of membership on belonging $x \in A$ is basic.

Example 1. *x & A are humans, we write $x \in A$ if x is ancestor of A. extension $\rightarrow$ if two humans are equal, they have the same ancestors (only if). True. extension $\rightarrow$ if two humans have the same ancestors, they are equal (if). False: brother & sister.*
So extension makes important statement about the relation of belonging.

We may specify a set by giving a condition. Much of set theory makes new sets from old ones. Note that:

$$\{x \in A \ : \ x \text{ is married}\}$$

is not itself a married person !

A sentence is constructed by taking atomic statements and combining them by logical connectives and proceeding by quantifiers:

axiom!of specification
Russell's Paradox

Atomics	**Connectives**	**Quantifiers**
$x \in A$ membership		
$A = B$ equality	$A \downarrow B$ neither A nor B	$\forall$ for all
T true		

The other connectives can be defined.

F	as an abbreviation for	$T \downarrow T$	falsehood
$-A$	"	$A \downarrow A$	negation
$A \cap B$	"	$-A \downarrow -B$	and
$A \cup B$	"	$-(A \downarrow B)$	or
$A \to B$	"	$-(A \cap -B)$	if, then
$A \equiv B$	"	$(A \to B) \cap (B \to A)$	identity
$\exists\ a\ A$	"	$-((\forall a)\ -A)$	there exists
$x = y$	"	$\forall z\ [(z \in x)\ \equiv\ (z \in y)]$	equality

The concept of a set may now be defined:

$$y \text{ is a set}:\ -\forall x\ [(x \in y)\ \equiv\ (x = y)] \tag{2.4}$$

Axiom 2. (Axiom of specification or Aussonderungs axiom)
To every set A and to every condition $S(x)$ there corresponds a set B whose elements are exactly those elements x of A for which $S(x)$ holds.

Theorem 1. (Russell's Paradox)
Nothing contains everything (or there is no universe).

Proof. Consider $S(x) =\ not(x \in x)$. We will write $x \notin A$ for $not(x \in A)$. So $S(x) =\ x \notin x$. Let $b = \{x \in A\ :\ x \notin x\}$, then

$$\forall y\ [y \in B \text{ if and only if } (y \in A \text{ and } y \notin y)] \tag{2.5}$$

If $B \in A$, then $B \in B$ or $B \notin B$. If $B \in B$, (2.5) leads to contradiction, but $B \notin B$ leads to contradiction too.
Thus $B \notin A$.

Thus there exists something, B, that does not belong to A, but A was totally arbitrary!!! □

Remark. We need not only have a condition but also a set from which this condition will be evaluated, i.e. a universe. Then specification makes sense. This is Russell's Paradox in other words. Usually it is that the set of all sets that do not contain themselves does not belong to itself, but neither is it absent.

axiom!of existence
axiom!of pairing
empty set!existence
axiom!of unions

Thus we proceed to define furthermore requirements.

Axiom 3. *There exists a set.*

Axiom 4. Axiom of pairing
For any two sets, there exists a set that they both belong to.

Then we may deduce that:

Theorem 2. *There exists an empty set denoted $\emptyset$.*

PROOF. Let

$$\emptyset \;=\; \{x \in A \;:\; x \notin x\;\} \tag{2.6}$$

for any set A existing by assumption. Clearly the set $\emptyset$ contains no elements but exists. □

Theorem 3. *For any two sets, there exists a set which contains both and nothing else.*

PROOF. Let a and b be sets and let A be such that $a \in A$ and $b \in B$. Apply the Axiom of Specification to A with the condition

$$S(x) = \;[(x = a) \;\cup\; (x = b)] \tag{2.7}$$

where $\cup$ is the logical **OR** operation. The result is easily seen to be the set $\{a,\; b\}$. □

The empty set has several nice characteristics:

1. $\emptyset \subset A$ for all sets A, i.e. the empty set belongs to all sets, including itself.

2. To prove properties about the empty set we proceed by contradiction, i.e. we prove that the conditions can not be false.

Example 2. *We can prove the above statement **1**:*
The statement can be false only if $\emptyset$ had an element that did not belong to A but $\emptyset$ has no element at all, and thus the statement is true.

2.1 Operations on sets

To operate on sets, we need a further assumption.

Axiom 5. (Axiom of Unions)
For every collection of sets, there exists a set that contains all the elements that belong to at least one of the sets in the collection

union
intersection
disjoint
complement
Venn Diagrams
duality

Denote the collection by $\mathbb{C}$, then

$$U \ = \ \{x: \ x \in Y \text{ for some } Y \text{ in } C\} \tag{2.8}$$

This is called the **union** of the sets in the collection $\mathbb{C}$.
Suppose $\mathbb{C} = \{\text{A, B}\}$, then we denote $\mathbb{U}$= A ∪ B, as a special case for two sets. We could also have written $A \cup B \ = \ \{x: \ x \in A \text{ or } x \in B\}$.

Definition 1. *The* ***intersection*** $A \ \cap B$ *of two sets is*

$$A \ \cap B \ = \{x \in A \ : \ x \in B\}$$

and if $A \cap B \ = \emptyset$*, the sets are called* ***disjoint****.*

Definition 2. *The* ***complement*** *of a set* A *relative to set* B *is*

$$A \ - \ B \ = \ \{x \in A \ : \ x \notin B\}$$

When we know what the universe of discourse is, we denote complementation with respect to it by A'

Using **Venn Diagrams** we may display these operations like:

Figure 2.1: Universe U; complement A’, union A∪B, intersection A∩B

An important relationship between ∪ and ∩ is that known as the principle of duality.

Principle of Duality:
Any theorem that exists with ∪ and ∩ in it, also exists with ∩ and ∪ interchanged.

We have seen how we can combine sets, now we need an operation to get a large set from a single set.

axiom!of powers
powerset!existence
order
ordered pair
cartesian product

Axiom 6. (Axiom of Powers)
For each set, there exists a collection of sets that contains among its elements all the subsets of the given set.
A is a set, $\exists$ a set $\mathbb{P}$ such that if $X \subseteq A$, then $X \in \mathbb{P}$.

We may limit this set to the subsets and call it the **powerset** of A:

$$\mathbb{P}(A) \;=\; \{X \;:\; X \subset A\} \tag{2.9}$$

If A has n elements, then $\mathbb{P}(A)$ will have 2^n elements.

2.2 Order in Set Theory

Let $A \;=\; \{a,\; b,\; c,\; d\}$

We wish to define an **order** on this set. We do so by first singly taking out the smallest element, c say, and then the ext smallest and putting then each into a set like

$$C \;=\; \{\{c\},\; \{c,\; b\},\; \{b,\; c,\; d\},\; \{a,\; b,\; c,\; d\}\} \tag{2.10}$$

This specifies the order by giving a set of sets of elements. We denote $(a, b) \;=\; \{\;\{a\},\; \{a,\; b\}\}$ and this is our notation of an **ordered pair**, i.e. a precedes b in the pair (a, b).

Theorem 4. $(a, b) \;\in \mathbb{P}(\mathbb{P}(A \cup B))$

Proof. Let A and B be sets. Consider the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$. Does this set exist ?

We know that $\{a\} \subset A$ and $\{b\} \subset B$. Thus we conclude that $\{a,\; b\} \;\subset\; A \cup B$ and also $\{a\} \;\subset\; A \cup B$ and so both $\{a\}$ and $\{a,\; b\}$ and elements of $\mathbb{P}(A \cup B)$ because the powerset is the set of all subsets.

Thus $(a, b) \;= \{\;\{a\},\; \{a,\; b\}\}$ is an subset of $\mathbb{P}(A \cup B)$ and so an element of $\mathbb{P}(\mathbb{P}(A \cup B))$. □

We may also define the **Cartesian Product** of two sets:

$$A \;\times\; B \;=\; \{x \;:\; x = (a, b) \text{ for some } a \in A,\; b \in B\} \tag{2.11}$$

relation
domain
range
refelxive

2.3 Relations

Definition 3. *A set R is a* ***relation*** *iff each element is an ordered pair. If $(a, b) \in R$, we write aRb.*

Thus the empty set is a relation as it has no members that are not ordered pairs.

The cartesian product $A \times B$ is also a relation, by definition.

Let A be any set and R the set (x, y) of pairs in $A \times A$ for which $x = y$. The R is a relation of equality in A.

Let R be the set of pairs (x, A) in $A \times \mathbb{P}(A)$ for which $x \in A$. This is the relation of belonging between elements of A and subsets of A.

Definition 4. *Given a relation R we define its* ***domain*** *and* ***range****:*

$$\begin{aligned} dom\ R &= \{x \ : \ \text{for some } y \ (xRy)\} \\ ran\ R &= \{y \ : \ \text{for some } x \ (xRy)\} \end{aligned}$$

Example 3. *If $R \equiv$ marriage and the ordered pair is written (man,woman), then dom R = married man; ran R = married women.*

Some simple properties are:

$$dom\ \emptyset \ = \ ran\ \emptyset \ = \ \emptyset \tag{2.12}$$

$$R \ = \ X \times Y \ \Rightarrow \ dom\ R \ = \ X; \ ran\ R \ = \ Y \tag{2.13}$$

$$R \ = \ \text{equality in } X \ \Rightarrow \ dom\ R \ = \ ran\ R \ = \ X \tag{2.14}$$

$$R = \text{belonging between } X \text{ and } \mathbb{P}(X) \ \Rightarrow \ dom\ R = X; \ ran\ R = \mathbb{P}(X) - \{\emptyset\} \tag{2.15}$$

If R is contained in $X \times Y$, then

$$dom\ R \ \subset \ X; \ ran\ R \ \subset \ Y \tag{2.16}$$

and we say that it is a relation **from** X **to** Y. A relation from X to X is called a relation **in** X.

If R is in X and:

$$xRx \ \forall x \in X \ \Rightarrow \ R \text{ is } \textbf{refelexive} \tag{2.17}$$

$$xRy \ \Rightarrow\ yRx \ \Rightarrow\ R \text{ is } \textbf{symmetric} \tag{2.18}$$

$$xRy \ \&\ yRz \ \Rightarrow\ \ xRz \ \Rightarrow\ R \text{ is } \textbf{transitive} \tag{2.19}$$

symmetric
transitive
equivalence relation
partition
equivalence class
induced
function
surjective
onto
image
inclusion
embedding
injection
identity!map
inverse!map

Definition 5. *An **equivalence relation** is a relation in X that has all three properties. The smallest such relation (in X) is equality and the largest is $X \times X$.*

Definition 6. *A **partition** of X is a disjoint set $\mathcal{C}$ of non-empty subsets of X whose union is X.*

If R is an equivalence relation in X, and $x \in X$, then the set of all those elements $y \in X$ for which xRy is the **equivalence class** of X with respect to R.

If R is equality, then every element is an equivalence class. If R is $X \times X$, then X itself is the only equivalence class. We denote it x/R and X/R for the set of all equivalence classes.

Given a partition $\mathcal{C}$ of X, we write $(x\ X/\mathcal{C}\ y)$ as a new relation **induced** by $\mathcal{C}$. If R is an equivalence relation in X, then the set of equivalence classes is a partition of X that induces R.

If $\mathcal{C}$ is a partition of X, then the induced relation is an equivalence relation whose set of equivalence classes is exactly $\mathcal{C}$.

2.4 Functions

Definition 7. *Given two sets X and Y, a **function** f is a relation with dom $f = X$ from (or **on**) X to (or **into**) Y such that for each $x \in X$ there is a unique $y \in Y$ with $(x, y) \in f$ and this unique y is denoted by $f(x)$.*
We sometimes denote the function by $f : X \to Y$. The set of all functions from X to Y is denoted by x^Y.

Example 4. *$X \equiv$ people in a city; $Y \equiv$ addresses; $f \equiv$ address book.*

For a function f, if *ran* $f = Y$, the function is called **onto** or **surjective** . If $A \subset X$, then $f(A)$ is the **image** of A. If $x \subset Y$, the function $f(x) = x$ is called the **inclusion**, **embedding**, or **injection** map. The inclusion map of X into X is called the **identity** map (relation of equality).

The **inverse** of a function f is denoted by f^{-1} and is defined as follows:

$$\text{if } f : X \to Y \text{ then } f^{-1} : \mathbb{P}(y) \to \mathbb{P}(X) \text{ such that}$$

one-to-one
composite mapping
axiom!of infinity
successor set

$$\text{if } B \subset Y,\ f^{-1}(B) = \{x \in X \ :\ f(x) \in B\} \tag{2.20}$$

f is called **one-to-one** if it maps distinct elements to distinct elements. If f is one-to-one, $A = f^{-1}(f(A))$ but there are other cases. So what f does can not always be undone. If $f(x) = x^2$, then $f^{-1}(x) = \sqrt{x}$ does NOT work for all $x \in \mathbb{R}$, but would work for $x \in I^+$.

Definition 8. *If $f : X \to Y$ and $g : Y \to Z$, then ran f = dom g and thus $g(f(x))$ is defied for every $x \in X$. The function*

$$h : X \to Z \ h(x) = g(f(x))$$

*is called the **composite** of g and f. We denote it by $g \circ f$ or gf.*

If gf exists, fg does not necessarily defined as

$$dom\ f =^? \ ran\ g, \text{ i.e. } X =^? \ Z$$

and even if

$$dom\ f = ran\ g\ (= X).$$

It is not not necessarily true that $fg = gf$, i.e. even if it exists, composition is not always commutative.

It is, however, associative,

$$f : X \to Y,\ g : Y \to Z,\ h : Z \to U, \text{ then } h \circ (g \circ f) = (h \circ g) \circ f \tag{2.21}$$

2.5 Numbers

We may now define the integers ($^+$ is called successor).

$$0 = \emptyset,\ 1 = 0^+ = \{0\},\ 2 = 1^+ = \{0, 1\},\ 3 = 2^+ = \{0, 1, 2\}$$

Axiom 7. (Axiom of infinity)
*There exists a set containing 0 and the sucessor of each of its elements. Such a set is called a **successor set**.*

Theorem 5. *The intersection of every non-empty family of successor sets is also a successor set.*

Theorem 6. *The intersection of all successor sets included r a successor set A, is a successor set ω, which is a subset of every successor set.*

Proof.
If A and B are successor sets, then so is $A \cap B$ and since $(A \cap B) \subset A$, $A \cap B$ went into the definition of ω and hence $\omega \subset (A \cap B)$ and so $\omega \subset B$. □

natural number
Peano Axioms
induction principle
minimality of $“omega$
recursion theorem

REMARK. This is crucial. The set ω is a set that exists independently of A. This is a deep theorem.

The axiom of extension guaranties that there is a unique successor set that is included in every successor set. Thus ω is unique. We then define a **natural number** to be a member of the unique set ω.

We get at numbers via the **Peano Axioms** which give the basis of the definition of natural numbers through which we can get all other numbers and also analysis:

1. $0 \in \omega$
2. $(n \in \omega) \rightarrow (n^+ \in \omega)$ where $n^+ = n \cup \{n\}$
3. $[(S \subset \omega)$ and $(0 \in \omega)$ and $(n^+ \in \omega)] \rightarrow (S = \omega)$
4. $n^+ \neq 0 \;\forall\; n \in \omega$
5. $[(n, m \in \omega)$ and $(n^+ = m^+)] \rightarrow (n = m)$

Comments and translation into English:

1. Zero is a natural number. This is the base case for the induction to come later.
2. If n is a natural number so is the successor. The successor is the union of n and $\{n\}$.
3. If S is a subset of the natural numbers that contains zero and contains the successor of every natural number, then S is ω. This is known as the **induction principle** or **the minimality of** ω. It basically says that ω contains only natural numbers and that all natural numbers are reached by the successor operation, hence the names.
4. Zero is not the successor of any natural number. This is saying that zero really is the base case for the number system.
5. If the successors are equal, the numbers are equal. This asserts that successorship is a unique operation, a bijective function.

   This statement can be proven form set theory studied so far but the proof is not easy. It is thus not required as an axiom if one stipulates set theory in addition.

Theorem 7. (Recursion Theorem)
If $a \in X$ and $f : X \rightarrow X$, then there exists a function $u : \omega \rightarrow X$ such that $u(0) = a$ and

$$u(n^+) = f(u(n)) \;\forall n \in \omega$$

.

sum
product
associative
commutative
distributive
power

Remark. The proof is difficult but could be given on present knowledge.

The theorem asserts that for each function f, we can construct a function that is given by a series of f,

$$u(n^+) = \underbrace{f(f(...u(0)...))}_{n+1 \ times} \tag{2.22}$$

since $n^+ = 0^{\overbrace{++...+}^{}}_{n+1 \ times}$. This is true also for each base value a.

This is important for definitions. We define something for zero and give a formula for the successor and it can be always be calculated. This is definition by induction.

2.6 Arithmetic

Theorem 8. *There exists a function $s_m : \omega \to \omega$ such that*

$$s_m(0) = m, \ s(n^+) = (s_m(n))^+ \tag{2.23}$$

for each number n. We define $s_m(n)$ to be the ***sum*** *of n and m.*

Proof. By recursion theorem.

Theorem 9.

$$\forall \ k, m, n \in \omega : \begin{cases} (k+m)+n = k+(m+n) \\ k+m = m+k \end{cases}$$

In a similar way, we may define the **product** of two numbers:

$$p_m : \omega \to \omega \ : \ p_m(0) = 0, \ p_m(n^+) = p_m(n) + m \tag{2.24}$$

and we define $p_m(n)$to be the product nm. Now we may prove that multiplication is associative, commutative and distributive:

$$(km)n = k(mn) \tag{2.25}$$

$$km = mk \tag{2.26}$$

$$k(m+n) = km + kn \tag{2.27}$$

The **power** is defined just the same,

$$e_m : \omega \to \omega \ : \ e_m(0) = 1, \ e_m(n^+) = e_m(n) \cdot m \tag{2.28}$$

and e_m is the power m^n. In this, all the functions may be defined by inverse functions, the system of numbers extended to negatives, rationals, reals, complex numbers.

Definition 9. *two natural numbers are **comparable** if and only if $n \in m$, $n = m$, or $m \in n$.*

comparable
least element
equivalent
finite set
infinite
maximal element
anti-symmetric
order!partial

Theorem 10. *Any two natural numbers are comparable and exactly one of the above conditions can hold for any two natural numbers n and m.*

Theorem 11.

$$\text{if } m < n \;\Rightarrow\; m + k < n + k \text{ and}$$

$$\text{if } k \neq 0 \;\Rightarrow\; mk < nk$$

Let E be a non-empty set of natural number, then there exists a $k \in E$ such that $k \leq m$ for all $m \in E$, i.e. there is always a **least** element in any collection of natural numbers.

Definition 10. *Two sets are **equivalent** if and only if there exists a one-to-one correspondence between them, this is denoted by $\approx$.*

*A set is **finite** if it is equivalent to some natural number and **infinite** otherwise.*

We may prove that ω is infinite, i.e. the set of all natural numbers not a natural number itself.

If we denote the number of elements in E by $|E|$, then $|E|$ is the unique natural number equivalent to E. We may show that:

$$|E \cup F| = |E| + |F| \tag{2.29}$$

$$|E \times F| = |E||F| \tag{2.30}$$

$$|E^F| = |E|^{|F|} \tag{2.31}$$

Furthermore, the union of any two finite set is also finite. If E is finite then so is $\mathbb{P}(E)$, in fact

$$|\mathbb{P}(E)| = 2|E| \tag{2.32}$$

If E is a finite, non-empty set of natural numbers, there exists a $k \in E$ such that $m \leq k$ for all $m \in E$, i.e. there exists a **maximal** element in a finite collection of natural numbers.

2.7 Axiom of choice

Definition 11. *A relation in a set X is called **anti-symmetric** if $\forall x, y \in X$ the condition $(xRy$ and $yRx)$ implies $x = y$.*

Definition 12. *A reflexive, transitive, anti-symmetric relation in X is called a **partial order** in X.*

order!total
axiom!of choice
Zorn Lemma
order! well-ordered

Example 5. *Let R be denoted by $\leq$, then*

reflexive: $x \leq x$.
anti-symmetric: if $x \leq y$ and $y \leq x$, then $x = y$.
transitive: if $x \leq y$ and $y \leq z$, then $x \leq z$.

But $\leq$ does not need to refer to numbers ! If in addition we have that ***either*** *$x \leq y$* ***or*** *$y \leq x$ $\forall x, y \in X$ then the ordering is called* ***total.***

The axiom of choice is a set theoretical statement that have been found unprovable from the other axioms. Many mathematicians dispute it because it claims the existence of something one has no recipe for constructing. Some people like it and some do not, most are ambivalent. It comes in many statements, here are some:

1. The cartesian product of a non-empty family of non empty sets is non-empty.

   Explain:

   (a) If a set is non-empty, it has an element.

   (b) If X and Y are non-empty so is $X \times Y$.

   (c) If $\{X_i\}$ is a sequence of sets, then $X_1 \times X_2 \times \ldots \times X_n$ is empty if and only if one set in $\{X_i\}$ is empty.

2. If $S = \{X_i\}$ a collection of sets, then there exists a set $\mathcal{C}$ such that it contains exactly one element from each X_i.

   Explain:

   Let $\mathcal{U} = X_1 \times X_2 \times \ldots \times X_n$. An element $f \in \mathcal{U}$ is a function with *dom* $f = U$ such that if $A \in \mathcal{S}$, then $f(A) = A$.

   If $\mathcal{S}$ is the set of all non-empty subsets of a set X such that *dom* $f = \mathbb{P}(X) \setminus \{\emptyset\}$ such that if $A \in$ *dom* f, $f(A) = A$. Thus f is simultaneous **choice** of an element from each of many sets. Hence the name of the axiom.

3. **Zorn's Lemma:** If X is a partially ordered set such that every totally ordered subset of X has an upper bound, then X contains a maximal element.

   Explain:

   Let $X = \{1, 2, 3\}$. It is bounded by 4 and thus contains a maximal element, namely 3.

Definition 13. *A partially ordered set is* ***well-ordered*** *if every non-empty subset has a smallest element.*

induction
permutation
cycle
Heap, B.R.

Principle of transfinite induction:
If X is well-ordered and $S \subseteq X$ and whenever $x \in X$ is such that the initial segment $s(x) \in S$, then $x \in S$; then we must have $S = X$.
Explain:
$X = \omega$ the set of natural numbers. If $S \subseteq X$ and for each number for which every lesser number belongs to S, that number belongs also, we must have $S = \omega$. This is the same as the induction previously but it allows for an infinite number of steps so it must be explicitly accepted.

We may use this to show:

1. Every set can be well ordered.

2. Every well-ordered set is totally ordered.

2.8 Permutations

Definition 14. *A* ***permutation*** *is a bijection of the elements of a set A on A.*

The number of elements is nPn if $|A| = n \ \in \mathbb{N}$ and $nPn = n!$
A permutation can be represented

$$\begin{aligned} \psi &= \{(1, x_1),\ (2, x_2)\ \ldots\ (n, x_n)\} \\ &= \begin{pmatrix} 1 & 2 & \ldots & n \\ x_1 & x_2 & \ldots & x_n \end{pmatrix} \end{aligned}$$

so that $\psi(2) = x_2$. A permutation is a **cycle** over A if $A = \{a_1, a_2, \ldots, a_n\}$ and

$$\psi = \begin{pmatrix} a_1 & a_2 & \ldots & a_{n-1} & a_n \\ a_2 & a_3 & \ldots & a_n & a_1 \end{pmatrix}$$

This is a cycle of length n.

Example 6.

$$\psi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 1 \end{pmatrix}$$

has one cycle of length 2, [1,2], and one of length 3, [3,4,5].

Theorem 12. *Every permutation of a finite set can be decomposed into a composition of disjoin cycles. The order of the cycles does not matter and the decomposition is unique up to order.*

The best algorithm to generate permutations is Heap's algorithm. [B.R. Heap, Comp. I. **G** 1963, 293 - 294]

Heap:
Algorithm generates all permutations $a_0 a_1 \ldots a_{n-1}$ of a set $\{1, 2, \ldots, n-1\}$ using a central table $c_n \ldots c_2 c_1$.

lexicographical order

Theorem 13. *This algorithm generates all permutations exactly once. (It does* ***not*** *do so in lexical order).*

```
for $0 \leq j < n$ do
    $a_j \leftarrow$ j
    $c_{j+1} \leftarrow 0$
end for
k ← 1
do
    visit $a_0 a_1 \ldots a_{n-1}$
    k ← 1
    while $c_k$=$k$ do
        $c_k \leftarrow 0$
        k ← k+1
    end while
    if $k$=$n$ then
        quit loop and algorithm, we have all permutations
    end if
    $c_k \leftarrow c_{k+1}$
    if $k$=$n$ then
        quit
    end if
    t ← $a_k$
    if $k$ is odd then
        j ← $c_k - 1$
    else
        j ← 0
    end if
    $a_k \leftarrow a_j$
    $a_j \leftarrow$ t
end do
```

2.9 Subset generation

The set $[n] = \{1, 2, \ldots, n\}$ has $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$ subsets of k elements. We want to generate all subsets in **lexicographical order**, i.e. $1 < 2 < 3 < ... < n$.

$$\begin{aligned} (v_1, v_2, \ldots, v_n) \; &< \; (\omega_1, \omega_2, \ldots, \omega_n) \; iff \; \exists \, k \leq n \\ &\text{such that } v_i = \omega_i \; \forall 1 \leq i < k \text{ and } v_k < \omega_k \qquad (2.33) \\ &\Rightarrow \; (1, 2, 3) \; < \; (1, 3, 2) \end{aligned}$$

This algorithm generates all k-element subsets of $[n]$ in this order: rank

```
for $i \leftarrow 1$ to $k$ do
    $v_i \leftarrow$ i
end for
$v_{k+1} \leftarrow$ n+1
while $v_1 <$ n-k+1 do
    OutPut($v$)
    j $\leftarrow$ 0
    do
        $v_{j+1} \leq v_j + 1$
    while $j \leftarrow j+1$
end while
for $i \leftarrow 1$ to $j - 1$ do
    $v_i \leftarrow i$
end for
```

Since we are generating them in order, we can number the subsets. This number is the **rank** . To get subsets given their rank is the unrank procedure.

```
$Rank(v_1, v_2, \ldots, v_n)$
$R \leftarrow 0$
for $i \leftarrow 1$ to $k$ do do
    $R \leftarrow R + \binom{v_i - 1}{i}$
end for
output($R$)
```

```
$Unrank(m)$
for $i \leftarrow k$ down to 1 do
    $p \leftarrow i - 1$
    while $\binom{p}{i} \leq m$ do
        $p \leftarrow p + 1$
    end while
    $m \leftarrow m - \binom{p-1}{i}$
    $v_i \leftarrow p$
end for
output($v$)
```

We want the subsets of $[v_k - 1]$ and by some others before it in the lexicographical order, $Rank(v_1 \ldots v_k) = \binom{v_k - 1}{k} + Rank(v_1 \ldots v_{k-1})$.

This is simply turns ranking around.

integer partition
Ferrer diagram
Young diagram
row

2.10 Integer Partitioning

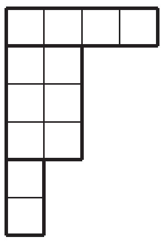
An integer n can be **partitioned** into $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ if

$$\lambda_1 + \lambda_2 + \cdots + \lambda_k = n \text{ and } \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_k \geq 1. \tag{2.34}$$

Example 7. $\lambda = (4, 2, 2, 2, 1, 1)$ *is a partition of 12 to 6 parts. We also denote this by* $4^1 2^3 1^2$.

We can denote this graphically by **Ferrer's diagram** or **Young's diagram** The largest part is the top **row** and so on...

$4^1 2^3 1^2 \equiv$



The conjugate partition is when the diagram is rotated by 90°.

$$conj\ (4^1 2^3 1^2) = 6^1 4^1 1^2$$

We want to generate all partitions of a given n.
Let's generate them in reverse lex ordering:

$$7,\ 61,\ 52,\ 51^2,\ 43,\ 421,\ 41^3,\ 3^21,$$

$$32^2,\ 321^2,\ 31^4,\ 2^31,\ 2^23,\ 2^21^3,\ 21^5,\ 1^7.$$

In the algorithm we will use $p = (p_1, p_2, \ldots, p_k)$ to denote the parts and $m = (m_1, m_2, \ldots, m_k)$ to denote their multiplicities.

Thus $p = (4, 2, 1)$ and $m = (1, 2, 3)$ means $4^1 2^3 1^2$.

Given a partition, we find the smallest part not equal to 1. then we use this parti p_i and all the 1's to make as many copies of p_{i-1} as possible.

Example 8. 54^21^4 *and so* $p_i = 4$ *and we change the 4 and the two 1's into* 3^2 *and the remaining two 1's into a 2 so that the next partition is* 543^22.

Here is the algorithm:

```
$p_1 \leftarrow n$
$m_1 \leftarrow 1$
$r \leftarrow 1$
done $\leftarrow$ false
while not done do
    output(p)
    if $p_r > 1$ or $r > 1$ then
        if $p_r = 1$ then
            $s \leftarrow p_{r-1} + m_r$
            $k \leftarrow r - 1$
        else
            $s \leftarrow p_r$
            $k \leftarrow r$
        end if
        $\omega \leftarrow p_k - 1$
        $u \leftarrow \lfloor s/\omega \rfloor$ // the floor function $\lfloor\ \rfloor$
        $v \leftarrow s \bmod \omega$
        $m_k \leftarrow m_{k-1}$
        if $m_k = 0$ then
            $q \leftarrow k$
        else
            $q \leftarrow k + 1$
        end if
        $m_q \leftarrow u$
        $p_q \leftarrow \omega$
        if $v=0$ then
            $r \leftarrow q$
        else
            $m_q \leftarrow 1$
            $p_{q+1} \leftarrow v$
            $r \leftarrow q + 1$
        end if
    else
        done $\leftarrow$ true
    end if
end while
```

2.11 Gray Codes

We have generated the subsets of size k so far. What if we want all subsets ? We can represent this by a binary string of length n. For example let n=3 and a subset is 1,2; we denote this by (1,1,0).

Gray Code
Gray Code!binary
Gray code!binary reflected

We can generate this in the order of binary numbers or by swapping one bit. The later way turns useful in many applications and is called the **Gray code**.

Example 9. *Let* $n = 3$

Bit Counting		***Gray Code***	
Subset	***Vector***	***Subset***	***Vector***
{}	*(0,0,0)*	{}	*(0,0,0)*
{*3*}	*(0,0,1)*	{*3*}	*(0,0,1)*
{*2*}	*(0,1,0)*	{*2,3*}	*(0,1,1)*
{*2,3*}	*(0,1,1)*	{*2*}	*(0,1,0)*
{*1*}	*(1,0,0)*	{*1,2*}	*(1,1,0)*
{*1,3*}	*(1,0,1)*	{*1,2,3*}	*(1,1,1)*
{*1,2*}	*(1,1,0)*	{*1,3*}	*(1,0,1)*
{*1,2,3*}	*(1,1,1)*	{*1*}	*(1,0,0)*

Gray Codes are used for example by genetic algorithms to generate mutations. This is preferred because each successive piece differs from its ancestor by only one move and thus prevents genetic explosion.

Frank Gray patented this idea in 1953 for a pulse code modulation (PCM) transmission of data signals. The **binary Gray Code** uses 2^n numbers for n digits. The two digit code has 00,01,10,11.

To be used practically, we want:
(1). Rules for formation of the code should apply to all natural numbers
(2). There should be simple translation rules from code to numbers and vice-versa.
The simplest such code is the **binary reflected Gray code**:

1. The one digit code is 0,1.
2. To generate the n-digit code from the n-1 digit code do:
    - Reflect the code and append it.
    - Prepend 0 to the first half and 1 to the second.

Example 10. *We start with 0, 1. We reflect (1,0) and append it to get 0, 1, 1, 0. Then we prepend 0 and 1 to the respective halves of the code resulting in 00, 01, 11, 10, the n=2 code.*

The components of a binary reflected Gray code may be visually arranged on the corners of a hypercube of dimension n. Gray codes on a different number basis (nor binary) can be viewed as Hamiltonian paths on a lattice - see Martin Gardner.
Any path that visits each corner exactly once (Hamiltonian path) is a valid binary Gray code. We have:

towers of Hanoi

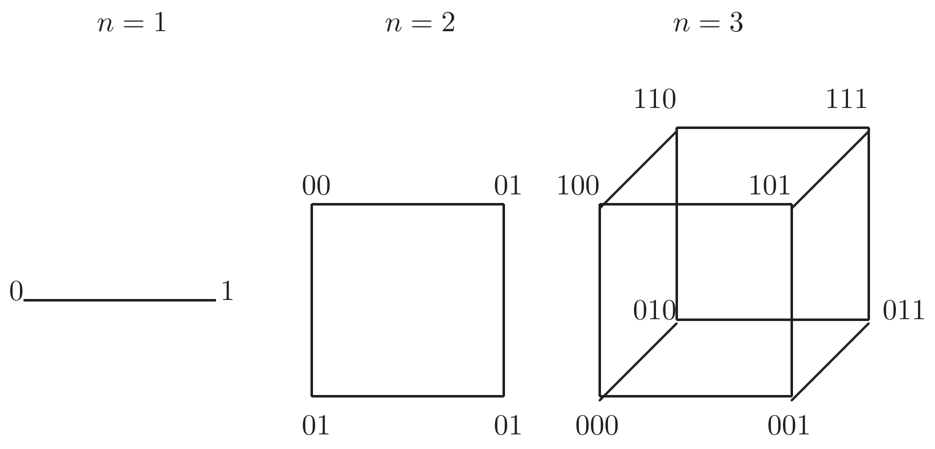


Figure 2.2: The unit cube is drawn and then the Gray code values are simply the coordinate vectors of the corner points

n	**# Hamiltonian Cycles**	**# Hamiltonian paths** (non-cyclic)
1	2	0
2	8	0
3	96	48
4	43008	48384
5	58018928640	187499658240

Cycles mean that the terminal node is adjacent to the starting one. Those numbers count a path and its reversal separately. The numbers for $n > 5$ are unknown because so far they can only be found by enumerating them all. This is a serious research problem - find a formula or even bounds for these numbers.

Let's solve two popular problems using Gray codes.

The Towers of Hanoi :
We have three pegs and n disks of pairwise distinct radius stacked on one peg in order from longest on the bottom to smallest at top. We are to move the whole stack to another peg accordingly to the rules:

1. We must only move one disk at a time.
2. At no time is a longer disk allowed to reside over a lower one.

Solution
Label the disks A, B, C ... from smallest to largest. Construct the binary reflected Gray code on n digits an write them in a column. For each digit in the stack of Gray code element name it from right to left by A, B, C ....

Chinese-Ring Puzzle

Figure 2.3: Starting position for Towers of Hanoi

Interpret this moving that disk for which the corresponding Gray code bit changed. The rules assume that you can only move the disk to one place (except the first). So for $n = 3$ we have:

C	**B**	**A**	**Disk Moved**
0	0	0	
0	0	1	A
0	1	1	B
0	1	0	A
1	1	0	C
1	1	1	A
1	0	1	B
1	0	0	A

This solution is exactly $2^n - 1$ steps long which can be shown to be minimum possible.

The Chinese-Ring Puzzle
The initial position of the puzzle with $n = 5$ rings is shown in the diagram. We are to remove all n rings from the stick subject to the rule:

1. The first ring may be put on or be removed at free will.
2. The $(n+1)^{th}$ ring may be put on or be removed if and only if the n^{th} ring is on, and the $1, 2, ..., n-1$ rings are all off.

Solution
We construct a n-digit binary reflected Gray code and interpret each bit as the ring being on or off depending on whether the bit is equal to 1 or 0 respectively.

if n is the number of rings, the number of moves necessarily made is:

$$\mathrm{N} \; = \; \begin{cases} \frac{1}{3}(2^{n+1} - 2) : n \; even \\ \frac{1}{3}(2^{n+1} - 2) : n \; odd \end{cases}$$

If however, we are allowed to solve the puzzle the "fast" way in which we are allowed to move the first two rings as a unit as well as individually, then:

$$\mathrm{N} = \begin{cases} 2^{n-1} - 1 : n \textit{ even} \\ 2^{n-1} : n \textit{ odd} \end{cases}$$

The ratio of slow to fast converges rapidly to $\frac{4}{3}$.

Jesse Watson constructed the "maximum effort" puzzle in which the initial position is not 11...11, but 00...001, i.e. all the rings off except the last one. This requires more moves than any starting position. If the slow method is used, we need $2^n - 1$ moves.

Gray codes can be used to solve (to mention a few):

- **more puzzles** :

  Martin Gardner "The binary Gray Code", in "Knotted Doughnuts and other mathematical entertainments" F.H. Freeman and Co., NY, 1986.

- **analog-digital connectors**

- **hypercubes** :

  F.Gilbert "Gray codes and paths on the n-cube", Ball System Tech. I. 37(1998) 815-826.

- **Cayly graphs of Coxeter groups**:

  I.Conway, N.Sloane, A.Wilks "Gray codes and reflection groups" Graphs and Combinatorics 5(1989) 315 -325.

- **Capanology, Bell-ringing**:

  A.White "Ringing the Cosets" Amer. MAth. Monthly 94(1987) 721-746.

- **cont. space, filling curves**:

  W.Gilbert "A cube filling Hilbert curve" Math. Intels. 6(1984) 78.

- **classification of Venn Diagrams**:

  F.Ruskey "A survey of Venn Diagrams" Elec.I. of Combinatorics (1997).

- **communication codes**

- **increase spectrometer resolution** (CODACON at LASP Univ. of Colorado for satellite application).

Here is how to generate the N-ary code (as opposed to the binary discussed here): M.C. Er "On generating N-ary reflected Gray codes", IEEE Transactions on Computers 33(1984) 739-741.

set cover
hitting set
set packing
dual

2.12 Set Covering and Packing

We are given a set S of subsets of a universal set U.

$$U = \{1, 2, ..., n\};\ S = \{S_1, S_2, ..., S_m\}$$

with $S_i \subseteq U\ \forall i : 1 \leq i \leq m$. We ask the following questions:

1. A **set cover** T is the smallest subset of S such that $\cup_{i=1}^{|T|} T_i = U$.
2. A **hitting set** H is the smallest subset of U such that each S_i contains at least one element of H.
3. A **set packing** P is the largest set of mutually disjoint subsets of S.

Example 11. *Suppose* $\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$,

$$\mathcal{S}_1 = \{1, 2\} \qquad \mathcal{S}_2 = \{4, 5, 7, 8\}$$

$$\mathcal{S}_3 = \{1, 2, 3, 4, 5, 6\} \qquad \mathcal{S}_4 = \{6, 7, 8, 9\}$$

$$\mathcal{S}_5 = \{3, 6, 9\} \qquad \mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4, \mathcal{S}_5\}$$

$$\Rightarrow \mathcal{T} = \{\mathcal{S}_3, \mathcal{S}_4\}\ \&\ \mathcal{H} = \{2, 6, 7\}\ \&\ \mathcal{P} = \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_5\}$$

This is much easier graphically:

Cover=$\{\mathcal{S}_3, \mathcal{S}_4\}$
Packing=$\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_5\}$
Hitting set=$\{2, 6, 7\}$

Definition 15. *The* ***dual*** *of a set problem is obtained by replacing each element of U by the set of all sets S_i that contain this element of U. We see that thus U and S have exchanged roles.*

Theorem 14. *The hitting set is dual to the set cover.*

set!partition

Remark.
If a problem Π_2 is dual to a problem Π_1, then any algorithm for Π_1 solves Π_2. Thus we will be discussing only set cover and packing.

Theorem 15. *Set cover and packing are NP - complete.*

Note the fundamental difference is that a packing need not all elements of U whereas a cover does, but may include any element more than once. A packing that omits no single element is called a **set partition** . Moreover, one is a minimum problem and the other is a maximization problem. Note also that packing and cover always have a solution but partition does not !

Let's reformulate the problems a bit. Let $A = [a_{ij}]$ be a matrix (n x m) such that rows correspond to elements of U and columns to elements of S and $a_{ij} = 1$ if $i \in S_j$ and $a_{ij} = 0$ otherwise. In addition we let $c = [c_j]$ be a cost vector, i.e. every set S_j is allowed to cost or make a profit. Furthermore let $e = [e_j]$ be the vector of 1's. Then,

- packing : maximize profit by $max(c^T x) \ : \ Ax \leq e$
- partition : minimize cost by $min(c^T x) \ : \ Ax = e$
- cover : minimize cost by $min(c^T x) \ : \ Ax \geq e$

$\forall x_j = 0$ or 1 and $1 \leq j \leq m$, where x_j decides whether to include set S_j or not.

Theorem 16.

$$(1)\ min(\ c^T : Ax = e,\ x_j = 0\ or\ 1) = -Nn + min(\ \overline{c}^T : Ax \geq e,\ x_j = 0\ or\ 1)$$

$$(2)\ min(\ c^T : Ax = e,\ x_j = 0\ or\ 1) = -Nn + min(\ \widetilde{c}^T : Ax \leq e,\ x_j = 0\ or\ 1)$$

where $1 \leq j \leq m$; $\overline{c}^T = c^T + Ne^T Aj$; $\widetilde{c}^T = Ne^T A - c^T$ and N is some sufficiently large integer.

Corollary 1. *Set partition can be brought into the form of set cover.*

Corollary 2. *Set partition can be brought into the form of set packing.*

Theorem 17. *Set packing can be brought into the form of set partition (i.e. it is a special case of a set partition).*

Why do we want to do this ? Here are a few applications.
Airline crew scheduling:
We need to schedule planes, fights and cabin crew such that constrains are met. We assume that the plane schedule is done and are concerned with assigning crew to planes. And since every plane has to have a crew and no

greedy
Chratal Algorithm

crew member can be on two planes at the same time, we have a set partition problem. Airlines also want costs to be low !

Delivery problem:
We have a warehouse of goods and customers to deliver them to. Find the minimum cost transportation schedule to do this. This is a set cover problem between the sets of orders and the stock in the warehouse.

Switching circuit:
We are given a set of k input bits and one output which depends only on a function of the 2^k input vectors (everything is in binary) and also a table giving the output for each input vector. Construct the cheapest circuit with only AND and OR gates, i.e. the one with fewest gates. This equivalently asks for the simplest disjunctive normal form equivalent to a given formula in Boolean algebra.

Fault - testing:
A person can detect a fault only if they meet it. Finding the faults in a network requires many walks and so find an optional family of paths between specified nodes that cover the network.
This is a graph problem and it is solved by first finding a maximum matching in the graph and then adding arbitrary edges to cover the remaining vertices.

the best way to solve the cover problem is via a greedy heuristic algorithm:

Chr*á*tal's Algorithm:
(V.Chr*á*tal "A Greedy Heuristic for the set covering problem" Math. oper. Res. 4(1979) 233-235).

```
cover ← $\emptyset$
do until all $S_j$ are empty
    find a set $P_k$ maximizing $h_k/c_k$ where $h_k = |S_k|$
    cover ← cover $\cup$ $\{k\}$
    for all nonempty $S_j$ do
        $S_j \leftarrow S_j - S_k$
    end for
end do
```

In other words, we choose the set of most elements per unit cost to include first. Chr*á*tal proved that this gives the cover exceeding the cost of an optimal cover by at most a factor

$$\sum_{j=1}^{d} \frac{1}{j} \text{ where d is the size of the largest } S_j.$$

This sweeps through the entire input instance at each step. Using linked lists and bounded-height priority guess, we can implement this algorithm is $\mathcal{O}(S)$ were

Bar-Yeuda and Even algorithm

$$S' = |S_1| + |S_2| + \cdots + |S_m|$$

This algorithm gives a cover that uses at most $ln(n)$ times as many sets as optimal. There exists another good algorithm.

Bar-Yehuda and Even:
(R.Bar-Yehuda and S.Even "A Linear-Time Approximation Algorithm for the Weighted Vertex Cover Problem" J. Algorithms 2(1981) 198-203)

```
$Q \leftarrow U$, the universal set
cover $\leftarrow \emptyset$
**for** $1 \leq j \leq m$ **do**
    $c_j^{'} \leftarrow c_j$
**end for**
**do until** $Q = \emptyset$
    let $i \in Q$  *
    $r \leftarrow min(c_l^{'} : l \in S_i)$
    let k be such that $c_k^{'} = r$
    **for** *all* $l \in S_i$ **do**
        $c_l^{'} \leftarrow c_l^{'} - r$
    **end for**
    cover $\leftarrow$ cover $\cup\{k\}$
    $Q \leftarrow Q - P_k$
**end do**
```

$*$ Augmenting heuristics with some exhaustive search or randomization is likely to give better results at the cost of higher run time. Choosing the i here is the right place. $*$

This algorithm selects the set of minimal cost. It can be shown that the resultant cover has at most the cost

$$f \; = \; max(\; \{|S_i| \; : \; i \in U\} \;)$$

times the optimal cover cost. This usually is worse than the previous algorithm but not bad for the vertex covering problem in graph theory (for which f=2, the subsets S_i being the edges).
For set packing we apply essentially the same heuristics only
1. If we want a packing with many sets , we repeatedly choose the smallest set and delete those that clash with it.
2. If we want a packing with few sets, we repeatedly choose the largest set and delete those that clash with it.

Euler's Konigsberg bridge problem

Chapter 3

A Graph-theoretical Beginning

3.1 History of Graph Theory

Leonard Euler lived in Kőnigsberg, Prussia (now Kalinigrad, Russia) and heard of the bridge problem:

The task is to start anywhere on one piece of land (A, B, C or D) and come back there crossing each bridge **exactly once**.

We may model this situation by representing each piece of land by a dot and each bridge by a line:

The answer is that it is **not** possible to start at any node and cross each edge only once and return to the starting node. The argument brought forward by Euler in 1736 is:

Every region (node) has to be entered **and** exited an integer number of times for the condition to hold. Thus the number of edges incident on a node has to be an **even** integer. This is not the case in this graph.

If we do not require the terminal and starting nodes to be identical, it is possible if exactly two nodes have an odd number of edges incident on them. Those two are then the ends of the path.

This is the first graph theoretical problem stated and solved. Many other many other questions can be asked and solved in a similar way. Part of the beauty is that the expected background material is virtually nothing and so the subject can be learnt very quickly.

We proceed to outline the connection between graphs and sets.

3.2 Graphs as Relations on Sets

A relation looks like:

In general a relation is between subsets of A and B.
Given a set, we define the identity and universal relations as:

identity!relation
universal!relation
graph

$$I_A \;=\; \{\ (a,a)\ :\ a \in A\ \} \tag{3.1}$$

$$U_A \;=\; \{\ (a,b)\ :\ a \in A\ \} \;=\; A \times A \tag{3.2}$$

Those relations can be displayed graphically. Take the set

$$X = \{a,b,c,d,e\} \quad R = \{(a,b),(a,c),(b,d),(c,e),(e,b)\}$$

Let's try to draw I_X, U_X and R in various ways.

The simplest diagram is to draw relations like a function on a cartesian plot.
The horizontal axis is the domain and the vertical axis is the range. We put a dot at a coordinate point if this ordered pair is in the relation.
This shows very little structure in the relation and we will not use this method.

Based on the same idea we can draw arrows from the elements of the domain to the element of the range for each ordered pair, both for perpendicular and parallel axes but both are useful for **very small** relations because we get too many arrows.

We get much more flexibility when we draw a node for each element and draw arrows between them for each ordered pair somewhere in the plane. This gives us the freedom to make use of some symmetry of the relation, if any. This will be called **graph** .

Thus a graph is a graphical representation of a relation of a set into itself.

Friendship Graph
directed
loop
weight
path
connected

3.3 Graph Questions

The **Friendship Graph** in the graph of the set of people to itself with arrows drawn between friends. We may ask some basic questions in this language which we can ask then about any graph.

1. Is the graph directed ? A graph is **directed** if some edges cannot be traversed both ways, i.e. some edges have arrows. Here this means: if I am your friend, are you my friend ?

2. Are there loops ? A **loop** is an edge that connects a node to itself. Here this means, am I my own friend ?

3. Are there multiedges ? A **multiedge** is an collection of more than one edge each of which have the same starting and ending vertices. Here this means, can I be your friend more than once ?

4. Are the edges weighted ? A **weight** is a real number associated to an edge. Here this means, on some scale, how close a friend to you am I ?

5. Is there a path from one node to the another ? A **path** is a sequence of edges such that the second node of the first edge is equal to the first node of the second edge. Here this means, is there a sequence of friendships that leads me to person X ?

6. Is the graph connected ? A graph is **connected** if any two nodes are linked by a path. Here this means that any two people have a sequence

of friends linking them.

degree
clique
cycle
simple cycle
girth
Hamiltonian cycle
tree
forrest

7. What is the degree of the node ? The **degree** of a node is the number of edges incident on it. This asks, how many friends do you have ?
8. Is there a clique ? A **clique** is a set of nodes such that an edge runs from every member to any other member. So this asks, is there a group of mutual friends ?
9. Is there a cycle ? A **cycle** is a path whose endpoints are identical. So this asks how loy gossip returns to the instigator.

A cycle with no repeated vertices is a **simple cycle** . The number of edges in the shortest cycle is the graph's **girth** . A simple cycle through all the vertices is a **Hamiltonian cycle** . An unidirected graph with no cycles is a **tree** if it is connected and a **forrest** otherwise.

3.4 Graph Representations

How do we represent a graph in such a way that we can work with it by computerized means ? There are two popular ways: adjacency matrices and adjacency lists.

Suppose there are n nodes, then the adjacency matrix M_{ij} is an n x n matrix such that:

$$M_{ij} = \begin{cases} 1 \text{ , } if\ (i,j)\ is\ an\ edge \\ 0 \text{ , } otherwise \end{cases}$$

An adjacency list contains a list of n items each of which list the nodes incident on that node.

Example 12. *A partially directed graph.*

$$M = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{array}{l} 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow ||| \\ \hookrightarrow 2 \rightarrow ||| \\ \hookrightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow ||| \\ \hookrightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow ||| \\ \hookrightarrow ||| \end{array}$$

That $M_{12} \neq M_{21}$ reflects the directed edge; the list includes this possibility too.

Lists are almost always the way to go in practice even though matrices are easier for theoretical work. Lists are faster and use less memory. They are a good motivation for plentiful pointer programming.

queue
breadth first search
stack
depth first search
breadth first search

3.5 Graph Traversal

We would like to have an efficient method to walk through the graph to visit every vertex in order to, for example:

1. print out or validate the information
2. copy the graph or convert to another representation
3. count edges/vertices
4. find paths/cycles if any
5. identify connected components

This is done as follows. Each vertex is marked as one of the three possibilities:

1. undiscovered
2. discovered (but not fully explored)
3. completely explored

The labels are given in that order. Initially we discover a single vertex and then proceed. We may proceed with the vertices in two ways:

Queue - Stack vertices in a FIFO (first in first out) so that we explore old vertices first. This is called **breadth first search** .

Stack - Store vertices in a LIFO (last in first out) stack so that we explore quickly away from the starting vertex. This is called **depth first search** .

The idea will become clear with the algorithms:

Breadth first search:

```
input a graph $G$ and a starting vertex $s$
for each vertex $U$ in $G$ except $s$ do
    state[n] = undiscovered
    p[n] = nil
end for
state[s]=discovered
p[s]=nil
$Q = \{s\}$
while $Q \neq \emptyset$ do
    u=dequeue[Q]
    process vertex u as desired
    for each vertex $v$ adjacent to $u$ do
        process edge (u,v) as desired
        if state[v]=undiscovered then
            state[v]=discovered
            p[v]=u
            enqueue [Q,v]
        end if
    end for
    state[n] = completely explored
end while
```

In this algorithm, Q is the queue (FIFO) and p holds the parent of a vertex. This meets each vertex one directed edge exactly once and each undirected edge exactly twice.

Depth first search

```
DFS - Graph(G)
input a graph G
for each vertex u in G do
    state[u] = undiscovered
end for
for each vertex u in G do
    if state[u]=undiscovered then
        initialize new component, if desired
        DFS[G,n]
    end if
end for
DFS[G,n]
state[u] =discovered
process vertex u as desired
for each vertex v adjacent to u do
    process (u,v) as desired
    if state[v]=undiscovered then
        p[v]=u
        DFS(G,v)
    end if
end for
state[u] =completely explored.
```

Both algorithms work only within only one component and hence the first algorithm for DSF - Graph was given to be complete for a graph of more than one connected component.

Example 13. *Each node has exactly one parent except the first which has no parents, the the result is a tree (that defines the shortest path from the root node to every node in BFS).*

graph
edge
end points
adjacent
loop
multiple edges
graph!simple
graph!directed
head
tail
subgraph
subgraph!induced

3.6 Basic Formal Definitions

A **graph** G with n vertices and m edges consists of an vertex set $v(G)$ and an edge set $E(G)$,

$$V(G) = \{v_1, v_2, \ldots, v_n\}; \quad E(G) = \{e_1, e_2, \ldots, e_m\} \tag{3.3}$$

where an **edge** consists of two (possibly equal) vertices called its **end points** . uv denotes $e = \{u, v\}$ and if $uv \in E(G)$ then u and v are **adjacent** , which we denote by $u \leftrightarrow v$.

A **loop** is an edge whose endpoints are equal. **Multiple edges** are edges that show both endpoints. A **simple graph** is a graph without loops or multiple edges.

A **directed graph** has ordered pairs as edges. If $(u, v) \in E(G)$, then u is the **head** and v is the **tail** of the edge.

The complement $\overline{G}$ of a simple graph G is the simple. graph with vertex set $V(G)$ and $uv \in E(\overline{G})$ if and only if $uv \notin E(G)$.

Example 14.

A **subgraph** of a graph G, is a graph H such that $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We denote this by $H \subseteq G$ and say that G contains H. An **induced subgraph** is such that $E(H)$ consists of all edges of G whose endpoints belong to $V(H)$, we denote this by $H = G[S]$.

graph!complete
clique
independent set
graph!bipartite
graph!k-partite
isomorphism
Petersen graph

Figure 3.1: G

Figure 3.2: $\overline{G}$

A **complete graph** K_n of n vertices is a simple graph in which every pair of vertices is an edge; this is called also a **clique**

An **independent set** in a graph G is a vertex subset $S \subseteq V(G)$ that contains no edge of G ($G[S]$ has no edges).

A graph G is **bipartite** if $V(G)$ is the union of two disjoint sets such that each edge consists of one vertex from each set. A graph is **k-partite** if $V(G)$ is the union of k (possibly empty) independent sets (every k-partite graph is also a k+1 - partite).

Example 15.

Figure 3.3: complete bipartite

Figure 3.4: 3-partite

3.7 Graph Equality

Definition 16. *An **isomorphism** from G to H is a bijection $f : V(G) \to V(H)$ such that if $uv \in E(G)$ then $f(u)f(v) \in E(H)$, we write $G \cong H$ and say that G is isomorphic to H.*

Example 16. *The **Petersen graph** is various drawings:*

graph!sum
graph!union

How might we test for isomorphism ? Suppose both graphs are represented by adjacent matrices $A(G)$ and $A(H)$. By definition, $G \cong H$ if and only if we can apply a permutation to the **rows** of $A(G)$ and the same permutation to the **columns** of $A(G)$ to obtain $A(H)$.

Permuting like this, corresponds to a vertex relabelling:

$$v_1, v_2, \ldots, v_n \to v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)}$$

for the permutation π.

3.8 Graph Operations

We put K_n compete graph on n vertices
P_n path on n vertices
C_n cycle on n vertices
and $K_{n,s}$ or a complete bipartite graph with partite sets of size n and s.

We may ask, how many graphs of n vertices are there ? If $|X| = n$, then X contains $\binom{n}{2} = \frac{n(n-1)}{2}$ unordered paths. We may include each edge so there are

$$2^{\binom{n}{2}} = 2^{\frac{n(n-1)}{2}} \tag{3.4}$$

simple graphs with n vertices. If $n = 4$, then there are 64 such graphs that fall into 11 isomorphism classes. Here are those 11 classes.
Here are some operations that we can perform on graphs to get many more graphs.

The **sum** of graphs G and H has two disconnected components, namely G and H, written $G + H$.

The **union** of graphs G and H has the vertex set and the edge set

graph! disjoint union
graph!join
Reconstruction Conjecture
articulation vertex
bridge
graph!non-trivial

It can not, at present, be calculated how many isomorphism classes there are for an n except by enumerating them. However their number grows exponentially.

$$V(G \cup H) = V(G) \cup V(H) \tag{3.5}$$
$$E(G \cup H) = E(G) \cup E(H) \tag{3.6}$$

The **disjoint union** ($V(G) \cap V(H) = \emptyset$) is the sum.
In general mG means $\underbrace{G + G + \cdots + G}_{m\ times}$.

The **join** of G and H is obtained from $G + H$ by adding the edges

$$\{xy \ : \ x \in V(G),\ y \in V(H)\}$$

and is denoted by $G \vee H$.

We may also subtract a vertex. If $v \in V(G)$ then $G - v$ is the graph with

$$V(G - v) = V(G) - \{v\};\ E(G - v) = E(G) - xy$$

for $x = v$ and $y \in V(G)$ or $x \in V(G)$ and $y = v$.

Theorem 18. *If G has $V(G)$ such that $|V(G)| \geq 3$ and at least two of $G - v_1, G - v_2 \ldots G - v_n$ are connected, then G is connected.*

Conjecture 1. (Reconstruction Conjecture)
*For every graph G with $|V(G)| \geq 3$ **all** properties of G may be computed from the list*
$\underline{G} = \{G - v_1, G - v_2, \ldots, G - v_n\} = \{G - v_i \ : \ v_i \in V(G)\}$

This conjecture is as yet unproved.

3.9 Articulation in Graphs

An **articulation vertex** or **bridge** is a vertex or edge respectively whose delation increases the connected components of the graph.

Theorem 19. *An edge is a bridge if and only if it belongs to no cycle.*

Theorem 20. *Every non-trivial graph has at least two vertices that are **not** articulation vertices.*

A graph is **non-trivial** if it contains an edge.

3.10 Handshaking

degree
regular
k-regular
isolated vertex
order
size

Definition 17. *The* ***degree*** *of a vertex v in graph G is equal to the number of edges containing v. It is denoted by $d_G(v)$ or $d(v)$ when it is clear what graph we are talking about.*

We denote the maximum degree in G by $\Delta(G)$ and the minimum degree by $\delta(G)$. A graph is ***regular*** *if $\Delta(G) = \delta(G)$ and* ***k-regular*** *if $\Delta(G) = \delta(G) = k$. An* ***isolated vertex*** *has degree 0.*

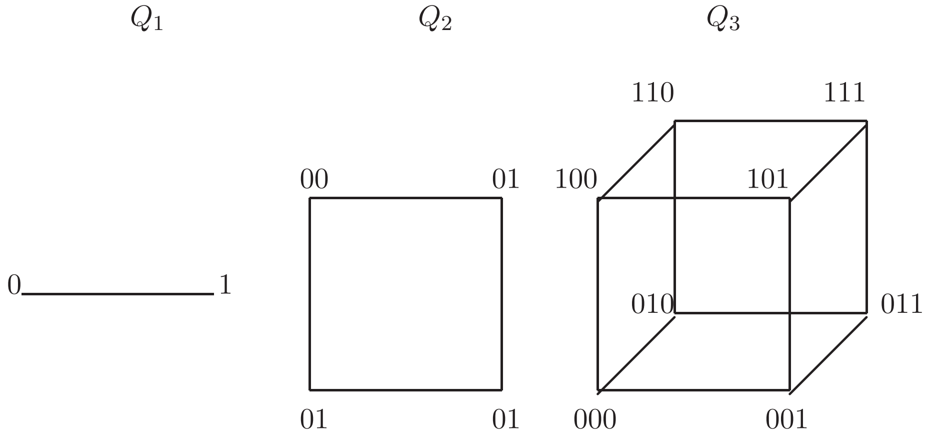
The ***order*** *of a graph G is $n(G)$ the number of vertices and the* ***size*** *is $e(G)$ the number of edges.*

Theorem 21. (Handshaking Lemma) *If G is a graph, then*

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

Proof. Every edge has two vertices.

Example 17. *The hypercube graphs Q_k.*



We construct Q_0 by a single isolated vertex with empty name. Given Q_{k-1} we construct Q_k by taking two disjoint copies of Q_{k-1}, call them A and B. For every vertex in Q_{k-1} append a 0 to the name of that vertex in A ant a 1 to the name of that vertex in B, and add the edge between these two vertices.

The degree of every vertex in Q_k is k and the number of vertices is 2^k so

$$\sum_{v\in V(Q_k)} d(v) = 2^k k = 2e(Q_k)$$

$$\Rightarrow e(Q_k) = k2^{k-1}$$

complete k-partite graph
graph!Turan

3.11 Tournaments

Seldom do two enemies have a mutual enemy but two will ally against the third usually. Given n factions, how many pairs of enemies can there be if no two enemies have a common enemy ?

Translated into graph theory, this asks for the maximum number of edges in a simple n-vertex graph with no triangles (K_3).

Definition 18. *The* ***complete k-partite graph*** *G is a graph whose vertexes can be partitioned into k sets such that $u \leftrightarrow v$ for any $u, v \in V(G)$ if and only if u and v belong to different sets in this partition.*

Thus every connected component of $\overline{G}$ is a complete graph.

When $k \geq 2$, we write $G = K_{n_1,n_2\ldots,n_k}$ for this graph with partite sets os size $n_1, n_2 \ldots, n_k$ and complement $K_{n_1} + K_{n_2} + \cdots + K_{n_k}$.

The ***Turán*** *$T_{n,r}$ is the complete r-partite graph with n vertices that has b parts of size $a+1$ and $r-b$ parts of size a, where*

$$a = \lfloor \frac{n}{r} \rfloor \text{ and } b = n - ra.$$

Theorem 22. *The graph $T_{n,r}$ is the unique graph in the set of all simple n-vertex graphs that have no $r+1$-clique that has the maximum number of edges.*

Example 18. *Let $n = 17$ and $v = 3$, then:*

$$a = \lfloor \frac{7}{3} \rfloor = 2 \text{ and } b = 7 - 3 * 2 = 1$$

and thus we have 1 part of size 3 and 2 parts of size 2.

Definition 19. *let v be a vertex in a directed graph. The* ***out-degree*** $d^+(v)$ *is the number of edges with tail v, and the* ***in-degree*** $d^-(v)$ *is the number of edges with head v.*

out-degree
in-degree
orientation
graph!oriented
tournament
king

An ***orientation*** *of a graph G is a di-graph D obtained from G by choosing an orientation for each edge $xy \in E(G)$, i.e. $x \to y$ or $y \to x$.*

A ***oriented graph*** *is an orientation of a simple graph, a di-graph without loops and no two edges having the same end point.*

A ***tournament*** *is an oriented complete graph.*

Consider a game with n teams such that each pair of teams plays exactly once. We record the outcome by drawing a directed edge from the winner to the looser. After it is done, the out-degree equals the number of wins for each team. Clearly

$$d^+(v) + d^-(v) = n - 1 \ \forall v \tag{3.7}$$

as each team played $n - 1$ times (no draws allowed).

A **king** is a vertex from which every other vertex is reachable by a path of length at most 2.

Theorem 23. *Every tournament has a king (every vertex is reachable by a a path of length at most 2).*

Thus there exists a team (king) such that it either beats team Z or beats some team that beats team Z for every other team Z.

It is possible to prove a formula for the number of distinct tournaments for a given n, but it is very complicated.

graph!planar

Chapter 4

Planarity

4.1 Example

In 1917, the following question appeared:
Three enemies live in three houses and each needs water, gas and electricity. It is possible to lay the lines from the three different companies such that they never cross to avoid confrontations ?

This asks whether $K_{3,3}$ can be drawn without intersection.
In the picture below we have one crossing between B(water) and A(electricity).

We will prove later that the answer is no.

Definition 20. *A* ***planar graph*** *is a graph which can be drawn on the plane without self-intersections.*

dual
length

4.2 Duals of Graphs

Definition 21. *The **dual** G^* of a graph G has one vertex for each region that G divides the plane into and one edge between two vertices if the corresponding regions are adjacent.*

Note that different planar embeddings of G may yield non-isomorphic duals !

There is a procedure (Whitney's Z-isomorphism theorem) to compute all duals of a planar graph.

Theorem 24. *$(G^*)^*$ is isomorphic to G if and only if G is connected.*

A statement about G turns into one about G^* if we interchange the roles of vertices and edges. The **length** of a face is the length of the walk in G that bounds that face.

Theorem 25. *If $l(F_i)$ denotes the length of face F_i in a planar graph G, then $2 * e(G) = \sum_l l(F_i)$.*

4.3 Euler's formula

Theorem 26. *If a planar graph G has e edges, n vertices and f faces and k connected components, then:*

$$n - e + f = k + 1$$

Proof. Let's do this by induction on n. If $n = 1$, then all edges are loops and each edge gives rise to one more face in addition that present originally. Thus $f = 1 + e$ and clearly $k = 1$ as $n = 1$ and so

$$1 - e + (1 + e) = 2 = k + 1 \checkmark$$

So base case of induction holds. Now add a vertex. If we don't add an edge we increase the vertices and components by one and the formula holds. If we add an edge, it may be a loop or not but it **will** split one region into two, and thus increase f by one and e by one and the formula holds. ■

Thus all planar embeddings of G have the same number of faces, thus the dual G^* has f vertices even though there might be several non-isomorphic duals.

Theorem 27. *Every simple n-vertex planar graph has at most* $3n-6$ *edges.*

Proof. This is true for $n = 3$. For $n \geq 3$ consider a graph with the maximum number of edges. G must be connected or else we could add an edge. Thus

$$n - e + f = 2$$

Every face has at least 3 edges on its boundary and every edge lies on the boundary of at most two faces. Thus $2e > 3f$.

$$n - e + f = 2 \;\Rightarrow\; 3n - 3e + 3f = 6$$

$$\Rightarrow\; 3n - 3e + 2e \geq 6 \;\Rightarrow\; e \leq 3n - 6$$

Example 19. *Consider* $K_5 \;\Rightarrow\; n = 5;\; e = 10;\; k = 1.$
A connected planar graph has at most $3n-6$ *edges if* $n \geq 3$*. Here* $e = 10 \;>\; 3*5-6$ *and so* K_5 *is not planar.*

Consider $K_{3,3} \;\Rightarrow\; n = 6;\; e = 9;\; k = 1$
and we have that $e = 9 \;<\; 3*6-6$*, but* $K_{3,3}$ *contains no triangle and here we have to have at most* $2n-4$ *edges.*
$e = 9 \;>\; 2*6-4$*, and so* $K_{3,3}$ *is not planar.*

Theorem 28. (Fary's theorem) *Every planar graph may be drawn using only straight lines.*

4.4 Planar Graph Coloring

vertex coloring
proper coloring

Definition 22. *A* ***vertex coloring*** *is a map* $\varphi : V(G) \to C$ *from the set of the vertices of a graph G to the set* $C = [k] = \{1, 2, \ldots, k\}$.

A ***proper coloring*** *is such that no two adjacent vertices receive the same color.*

This came up in history by coloring maps such that no two neighboring countries would get the same color.

Any boundary can be represented by the edges of a graph. Then we want to color the regions. If we construct then we are asking for a proper coloring of the dual.

chromatic number

The least number of colors necessary for a proper coloring of a graph is the **chromatic number** of G, $Chi(G)$.

The natural question is how many colors are at most required, i.e. over all planar graphs what is $Chi(G)$?

The answer is 4. This result is known as the four color theorem and is very important to the history of mathematics.

It was first posed to Augustus de Morgan in 1852 in the form of map coloring. Many people hence claimed proofs but they were all found to contain mistakes until 1976 when Appel and Haken solved the problem. The theorem itself is interesting because of its simple statement and difficulty in proving but also due to a number of practical applications we will get to later.

What made the problem even more famous was the manner in which Appel and Haken proved the result.

Let's first prove the five-coloring theorem.

Theorem 29. *Every planar graph G has a proper coloring using at most 5 colors.*

PROOF. First we argue that G contains a vertex v such that $d(v) \leq 5$:

If every node had degree at least 6, then the number of edges would be at least $3n$ which contradicts the fact that $3n - 6$ is the most number of edges that G can have.

Let H be a planar graph that requires at least six colors and has fewest vertices, i.e. a node-minimal counterexample.

H has a vertex v with $d(v) \leq 5$. If $d(v) < 5$, delete v to obtain H' which is five colorable as H' was node minimal. We can now 5-color H from H' by assigning v a color different from its neighbors and we need not use another color since $d(v) < 5$.

Consider the case when $d(v) = 5$. Let the nodes adjacent to v be called q_1, q_2, q_3, q_4, q_5. These five may not form K_5, i.e. there must be a pair (q_i, q_j) which is not an edge of H. This is true because K_5 is not planar and thus would cause H to be non-planar.

We contract the edges (v, q_i) and (v, q_j) to obtain a five-colorable graph since it has fewer vertices. We can use this five-coloring to 5-color H as follows: q_i and q_j get that same color as the combined vertex of $q_i\&q_j\&v$, and v gets the color not used in its five neighbors (there are at most 4 colors used as two of five neighbors have the same color). ■

four-color theorem

Remark. Thus we construct a five-coloring of a bigger graph from a coloring of a smaller graph. The two cases of local coloring we looked at are called "unavoidable configurations". We proved the theorem by assuming a counter-example, showing that it contained an unavoidable configuration and that any local five-coloring of a graph can be extended to a longer graph through an unavoidable configuration. As we had only 2 such configurations the proof was short.

If we want to prove that we can also produce a legal four-coloring of every planar graph, we must go through the same arguments. The search for an unavoidable set of configurations had to be done by computer and initially yielded 1936 cases. these were all checked for reducibility like above and thus constitute a proof of the **four-color theorem**.

This proof lead to uproar in the mathematical community as it was the first important proof delivered by computer. It is criticized for these reasons:

1. It offers no insight into why the result is true. It merely checks a large number of cases. Thus we learn nothing and can not extend the theory based on it.

2. It cannot realistically be checked. Thus there is doubt about its truth.

Criticism 2 was fixed by an algorithmic proof by Appel and Haken this proof was 741 pages long and so is not much simpler to check.

There is no proof known which does not check a large number(¿1000) of cases even though improvements were made. This is an outstanding challenge.

We demonstrate the simplest graph that requires 4 colors, were the numbers are to be interpreted as colors and this graph is K_4 which is clearly planar.

The graph K_5 is not planar and requires 5 colors. In general K_n requires n colors.
Note that $K_{3,3}$ is a 2-colorable graph as is any bipartite graph !

Next, we want to investigate when graphs are planar.

Kuratowski Theorem
Wagner Theorem

4.5 Planarity Results

Theorem 30. (Kuratowski's Theorem) *A graph G is planar if and only if it contains no subdivision of K_5 or $K_{3,3}$.*

By subdivision we mean replacing an edge by a path of length 2, i.e. introducing a new vertex.

Theorem 31. (Wagner's theorem) *Deletion and contraction of edges preserves planarity.*

We can seek the minimal non-planar graphs on this basis.

Unfortunately these results are not useful in actually testing (on a computer) whether G is planar. We have to find alternate means and we want these means to be fast.

There are many algorithms and it is typical to introduce an easy one and refer to the harder ones but we see no point in wasting time on obviously inferior algorithms. The algorithm by Hapcroft and Tarjan runs in linear time and is generally the best available.

```
Data   : An graph adjacency list $G$.
Result : A Boolean variable $P$ which is TRUE if the graph is planar
         and false otherwise.
integer $E$
$P \leftarrow TRUE$
$E \leftarrow 0$
for each edge of $G$ do
    $E \leftarrow E + 1$
    if $E > 3V - 3$ then
        $P \leftarrow$ FALSE
        return $P$
    end
end
Divide $G$ into biconnected components
for each biconnected component $C$ of $G$ do
    Transform $C$ into a palm tree $T$
    Find a cycle $c$ in $T$
    Construct a planar representation for $c$
    for each piece formed when $c$ is deleted do
        Apply this algorithm recursively to determine if the piece plus
        $c$ is planar
        if piece plus $c$ is planar and piece may be added to planar
        representation then
            add it
        else
            $P \leftarrow$ FALSE
            return $P$
        end
    end
end
```

Algorithm 1: Planarity Testing

For each point p added to the depth-first search tree, we compute the lowest vertex on this tree that can be reached from descendants of p. If this lowest vertex is *not* the root, then the second point (i.e. the neighbor of the root in the depth-first search tree) is an articulation vertex and must be deleted from the graph in order to obtain a biconnected component. We do this until no articulation vertices remain.

Data : An graph adjacency list G and two vertices u and v such that (u, v) is an edge. This procedure will be called DFS(G,u,v). $A(v)$ is the adjacency list of the vertex v. This will be a depth-first search augmented with additional bits. Each vertex p is assigned a distinct integer N_p which serves as its identifier and two integers $L_p^{(1)}$ and $L_p^{(2)}$ which measure the low points. $L_p^{(1)}$ is the lowest vertex below p reachable by a frond from a descendant of p and $L_p^{(2)}$ is the second lowest vertex below p reachable by a frond from a descendant of p.

Result : A list B of graph adjacency lists that are the biconnected components of G.

```
integer $n$
$n \leftarrow N_v \leftarrow n + 1$
$L_v^{(1)} \leftarrow L_v^{(2)} \leftarrow N_v$
for $w \in A(v)$ do
    if $N_w = 0$ then
        // $a$ is a new vertex
        mark $(v, w)$ as a tree arc
        DFS($G$,$w$,$v$)
        if $L_w^{(1)} < L_v^{(1)}$ then
            $L_v^{(2)} \leftarrow min\left\{L_v^{(1)}, L_w^{(2)}\right\}$
            $L_v^{(1)} \leftarrow L_w^{(1)}$
        else
            if $L_v^{(1)} = L_w^{(1)}$ then
                $L_v^{(2)} \leftarrow min\left\{L_v^{(2)}, L_w^{(2)}\right\}$
            end
        end
        else
            $L_v^{(2)} \leftarrow min\left\{L_v^{(2)}, L_w^{(1)}\right\}$
        end
    else
        if $N_w < N_v$ and $w \neq v$ then
            mark $(v, w)$ as a frond
            if $N_w < L_v^{(1)}$ then
                $L_v^{(2)} \leftarrow L_v^{(1)}$
                $L_v^{(1)} \leftarrow N_w$
            else
                if $N_w > L_v^{(1)}$ then
                    $L_v^{(2)} \leftarrow min\left\{L_v^{(2)}, N_w\right\}$
                end
            end
        end
    end
end
```

Algorithm 2: Division in biconnected components

Note that $v- \rightarrow w$ refers to v being connected to w via a frond in the depth-first search tree. Define $\phi((v,w))$ as a function of an edge (v,w) by

$$\phi((v,w)) = \begin{cases} 2w & \text{if } v- \rightarrow w. \\ 2L_w^{(1)} & \text{if } v \rightarrow w \text{ and } L_w^{(2)} \geq v. \\ 2L_w^{(1)} + 1 & \text{if } v \rightarrow w \text{ and } L_w^{(2)} < v. \end{cases} \tag{4.1}$$

We calculate $\phi((v,w))$ for every edge in the tree T and order the adjacency lists according to increasing value of ϕ using a radix sort to achieve an $O(V+E)$ time bound.

```
Data    : A list $B$ of graph adjacency lists that are the biconnected
          components of $G$. The $S_i$ are buckets that help with the
          sorting and $\emptyset$ is the empty list.
Result  : The list $B$ of sorted graph adjacency lists.
for $i \leftarrow 1$ to $2V+1$ do
    $S_i \leftarrow \emptyset$
end
for $(v,w)$ an edge of $G$ do
    Compute $\phi((v,w))$
    add $(v,w)$ to $S_{\phi((v,w))}$
end
for $v \leftarrow 1$ to $V$ do
    $A(v) \leftarrow \emptyset$
end
for $i \leftarrow 1$ to $2V+1$ do
    for $(v,w) \in S_i$ do
        Add $w$ to the end of $A(v)$
    end
end
```

Algorithm 3: Sorting of adjacency lists according to ϕ.

Next we have to find paths in the graph. To do this we search again by depth-first search using the newly sorted adjacency lists. Each time we traverse an edge we add it to the path currently being constructed and each time we traverse a frond, it becomes the last edge of the current path.

```
Data   : A depth-first search tree $T$. This procedure is called
         Pathfinder($i$) where $i$ is a vertex. Initially $s$ is set to zero
         and $i$ to one.
Result : A list of all paths.
for $w \in A(v)$ do
    if $v \to w$ then
        if $s = 0$ then
            $s \leftarrow v$
            start new path
        end
        add $(v, w)$ to current path
        Pathfinder($w$)
    else
        // this is reached if $v - \to w$ if $s = 0$ then
            $s \leftarrow v$
            start new path
        end
        add $(v, w)$ to current path
        output current path
        $s \leftarrow 0$
    end
end
```

Algorithm 4: Finds all paths in a depth-first search tree generated from a biconnected graph.

Now that we know all the paths in all the biconnected components of a graph G we must determine whether these paths can be embedded into the plane. The idea is as follows. We attempt to embed the paths one at a time. Let c be the first path. It consists of a set of tree edges $1 \to v_1 \to v_2 \to \cdots \to v_n$ followed by a frond $v_n - \to 1$. The vertices are numbered such that $1 < v_1 < v_2 < \cdots < v_n$. When c is removed from its biconnected component, this component falls into several pieces called *segments*. Each segment S consists of a single frond (v_i, w) or a tree edge (v_i, w) plus a subtree with root w plus all fronds leading from the subtree.

The order of the paths generated is such that all paths in one segment are generated before the paths in any other segment are generated. The segments are encountered in increasing order in v_i.

By the Jordan Curve Theorem, any segment must be embedded on a single side of c. We have the following result: A path from v_i to v_j may be added to the planar embedding by placing it on the left (right) of c if and only if no previously embedded frond (x, v_k) on the left (right) satisfies $v_j < v_k < v_i$.

We use this result to test for planarity as follows: First we embed c in the plane. Then we embed the segments one at a time in the order they are encountered during path finding. To embed a segment S, we find a path in it, say p. We choose a side of c on which to embed p. We compare p with previously embedded fronds to determine if p may be embedded. If not, we move segments which have fronds blocking p to the other side. If p can be embedded now, we do so. However, the movements of segments may require other segments to be moved also and so it may be impossible to embed p. If so, the graph is not planar. We embed the rest of S by recursion and then embed all other segments likewise.

We are going to use three stacks to store the information. Stack L is going to hold the vertices v_k in order such that $1 \rightarrow \cdots \rightarrow v_k \rightarrow \cdots \rightarrow v_i$ with $1 < v_k < v_i$ and such that some embedded frond enters v_k on the left. Stack R does the same except for frond entering on the right.

The stacks must be updated in four ways:

1. After all segments starting at v_{i+1} have been explored and embedded, all occurrences of v_i on L and R must be deleted and future segments start at vertices no greater than v_i.

2. If p is the first path on segment S and the terminal vertex f has $f > 1$, then f must be added to a stack when p is embedded. p can only be embedded on the left (right) when every vertex on L (R) is no greater than f. So f may be added on top of L (R).

3. The recursive application of the algorithm must add entries for other paths in the segment S.

4. Entries must be shifted from one stack to another as the corresponding segments are moved. Embedding a frond on the left (right) forces fronds in the same segment to be embedded on the left (right) and may force fronds in other segments to be embedded on the right (left). By a *block* B we mean a maximal set of entries on L and R which correspond to fronds such that the placement of any one of the fronds determines the placement of all the others. The blocks change as the content of the stacks change, but the blocks always partition the stack entries.

```
Data   : An graph adjacency list $G$. We use stacks $L$, $R$ and $B$.
Result : A Boolean variable $P$ which is TRUE if the graph is planar
         and false otherwise.
$L \leftarrow R \leftarrow B \leftarrow \emptyset$
Find first cycle $c$
while some segment is unexplored do
    Initiate search for path in next segment $S$
    When backing down tree edge $v \to w$ delete entries on $L$ and $R$
    and blocks on $B$ containing vertices no smaller than $v$.
    Let $p$ be the first path found in $S$.
    while position of top block determines position of $p$ do
        delete top block from $B$
        if block entries on left then
            switch block of entries from $L$ to $R$ and from $R$ to $L$ by
            switching list pointers.
        end
        if block still has an entry on left in conflict with $p$ then
            $P \leftarrow$ FALSE
            return $P$
        end
    end
    if $p$ is normal then
        add last vertex of $p$ to $L$
    end
    Add new block to $B$ corresponding to $p$ and blocks just removed
    from $B$
    Add end-of-stack marker to $R$
    Call embedding algorithm recursively
    for each new block $(x, y)$ on $B$ do
        if $(x \neq 0)$ and $(y \neq 0)$ then
            $P \leftarrow$ FALSE
            return $P$
        end
        if $y \neq 0$ then
            move entries in block to $L$.
        end
        Delete $(x, y)$ from $B$.
    end
    Delete end-of-stack marker on $R$. Add one block to $B$ to represent
    $S$ minus path $p$. Combine top two blocks on $B$
end
```

Algorithm 5: Checks if a biconnected graph can be embedded in the plane.

k-connected
cut-set
connectivity
connected
biconnected
disconnected

Chapter 5

Connectivity

5.1 Basic Definitions

Definition 23. *A graph G is **k-connected** if there does not exist a cut set of $k-1$ vertices, where a **cut-set** is a set $S \subseteq V(G)$ such that $G \setminus S$ is disconnected.*

*The **connectivity** $\kappa(G)$ is the minimum number of vertices, whose deletion disconnects G, i.e. it is the maximum k such that G is k-connected.*

*A **connected** graph is 1-connected; a **biconnected** graph graph is 2-connected and a **disconnected** graph is 0-connected.*

We may offer an alternative definition: A graph is k-connected if any two of its disjoint vertices can be joined by the k independent (i.e. disjoint) paths in G.

The fact that this is indeed an equivalent definition is a theorem.

Theorem 32. *For any graph G and $u, v \in V(G)$, the minimum number of vertices separating u from v along any path in G is equal to the maximum number of disjoint paths from u to v in G.*

The connectivity is interesting in practice because it represents the minimum number of damages that a network can sustain and still remain functional. This applies to electricity, water, telephone distributions and so on.

5.2 Computing $\kappa(G)$

Let us summarize the situation so far:
A k-connected graph is (k-1)-connected and **may be** (k+1)-connected.

k-connected	**meaning**	**test**
0	any graph	all graphs are 0-connected
1	connected	depth-first search starting at any vertex eventually reaches all vertexes. This test is linear, i.e. it runs in $O(V(G)+E(G))$
2	$\nexists$ articulation vertex	depth-first augmented with low-point calculation will find all articulation vertices. In this case, we must fail to find such a vertex, i.e. the low-point of vertices must be the DFS tree root vertex.

In general we may proceed as follows: If G is not connected, remove a vertex and see if it is now disconnected. Recurse this procedure and $\kappa(G)$ is the shortest branch on recursion time. This gives my algorithm (**not** found in books).

```
con($G, k$)
if $G$ is disconnected return $k$
$k \leftarrow k+1$
for $i \leftarrow 1$ to $|V(G)|$ do
    $p_i \leftarrow$ con($G - v_i, k$)
end for
return $min(p_i)$
```

The vertex v_i is removed and the function is called by **con**$(G, 0)$ i.e. the initial k must be 0. This function returns the minimum number of vertices whose removal disconnects G. If we add bookkeeping to a (non)-recursive or the recursive version of this, we can also find such a set by the same algorithm.

This requires (if we put $V = |V(G)|$)

$$\epsilon = V(V-1)(V-2)\dots(V-\kappa(G)+1) = \frac{V!}{(V-\kappa(G))!} \tag{5.1}$$

evaluations of the function **con**(G, k). So far for a given V, the most expensive calculation is $V!$ for the case that $\kappa(G) = V-1$. There is a unique graph with this property, i.e. it is K_V. Every other graph (as it is missing at least one possible edge) can be disconnected such that at least two vertices remain. Note that we have adapted the convention that K_V disconnects when a single vertex remains. This is not strictly true as K_V may not be disconnected at all. We **choose** this **convention** so that other results are consistent.

Harary graph
Harary Theorem

Example 20. *$K_{i,i}$ may be disconnected by deleting one of its partite sets X or Y but not in any other way. Thus $\kappa(K_{i,i}) = min\{i, j\}$.*

5.3 Some Bounds

Definition 24. *The **Harary graph** $H_{k,k}$ is constructed as follows:*
Place n vertices in circular order on the plane and suppose $k < n$. If k is even ($k = 2r$), we connect every r vertices and if k is odd ($k = 2r + 1$) we connect every r vertices and the diagonally opposite ones.

$H_{k,k}$ is k-regular (i.e. all vertices have degree k).

If $k = 2r + 1$ and n is odd, index the vertices by the integers mod n. We construct $H_{k,k}$ from $H_{2r,n}$ by adding $i \leftrightarrow i + n + 1$ for $0 \leq i \leq n - 1$.

We may prove:

Theorem 33. (Harary) *$\kappa(H_{k,k}) = k$ and so the minimum number of edges in a k-connected graph on n vertices is $\lceil \frac{kn}{2} \rceil$.*

Proving $\kappa(G) = k$ means showing that there exists **no** cut set S with $|S| < k$ and demonstrating one with $|S| = k$.
This can be done here.

In order to **allow** G to be k-connected, we have to have quite a few edges, otherwise a small set of vertices could already disconnect it. So the condition

$$|E(G)| \geq \lceil \frac{k|V(G)|}{2} \rceil \tag{5.2}$$

allows a graph G to be k-connected, i.e. if the condition is violated, G is **not** k-connected.

Bondy Theorem
weakly conditioned
unilaterally connected
strongly connected

Theorem 34. (Bondy) *For any simple graph G with vertex degrees $d_1 \leq d_2 \leq \cdots \leq d_n$ and $d_j \geq j + k$ whenever $j \leq n - 1 - d_n - k$ for $|V(G)| = n$, then G is (k+1)-connected.*

Note that connectivity of graphs is the same as simple graphs after loops and multi-edges have been deleted. Thus for vertex degree counting arguments, it is simpler to assume that the graph is simple.

5.4 Path Counting

Theorem 35. (Cooke & Bez) *Let A be the adjacency matrix of a graph G with $|V(G)| = n$, then:*

1. *$(A^k)_{ij}$ is the number of paths of length k from v_i to v_j.*
2. *$\exists$ a path $v_i \to v_j$ $(i \neq j)$ iff $(A + A^2 + \cdots + A^{n-1})_{ij} \neq 0$.*
3. *$\exists$ a path $v_i \to v_j$ iff $(A + A^2 + \cdots + A^n)_{ij} \neq 0$.*

We also want the connectivity matrix A_* such that

$$(A^*)_{ij} = \begin{cases} 1 \text{ if } \exists \text{ a path } i \to j \\ 0 \text{ otherwise} \end{cases}$$

We compute this

$$A^* = I \surd A \surd A^{(2)} \surd \ldots \surd A^{(n-1)} \tag{5.3}$$

where $\surd$ means infinimum, and any non-zero element in A^k is set to 1 to obtain $A^{(k)}$. This operation selects the lower of the two values and is applied element-wise for the matrices. This means that A^* will have elements 0 or 1. Clearly G is connected if and only if $(A^*)_{ij} = 1$ for all i, j.

A^* defines a partition of $V(G)$ which is equal to the connected components of G.

For a directed graph the situation is more complex. So suppose G is directed, we call $F(G)$ the graph associated to G by removing the directionality form all the edges. Then

Definition 25. *A di-graph G is:*

1. ***weakly-conditioned** if $F(G)$ is connected.*
2. ***unilaterally-connected** if for each pair of distinct vertices n G there is a path going between them in at least one direction.*
3. ***strongly-connected** is it is unilaterally connected un both directions.*

acyclic
DAG
disconnecting set
h-edge-connected
edge-connectivity
bridge

In terms of A^*, we have:

$$G \text{ is unilaterally-connected } iff\ (A^*)_{ij} \surd (A^*)_{ji} = 1\ \forall i \neq j \tag{5.4}$$

$$G \text{ is strongly-connected } iff\ (A^*)_{ij} = 1\ \forall i \neq j \tag{5.5}$$

The graph is also **acyclic** if there are no cycles, i.e. if

$$(A^*)_{ij} = 0\ \forall i.$$

We call a graph a **DAG** if it is directed and acyclic. These graphs turn out to be important in practice.

What remains is how to find A^* in a an efficient way. To compute from teh formula takes $4n^4 - 7n^3$ steps and Warshall's algorithm takes $4n^3$ steps. There are others but this is good enough.

```
Warshall (A)
$A^* \leftarrow A$
for $i \leftarrow 1$ to $n$ do
    for $j \leftarrow 1$ to $n$ do
        if $(A^*)_{ij} = 1$ then
            for $k \leftarrow 1$ to $n$ do
                if $(A^*)_{jk}$ then
                    $(A^*)_{ik} \leftarrow 1$
                end if
            end for
        end if
    end for
end for
return $A^*$.
```

Algorithm 6: Warshall's Algorithm

5.5 Edge-connectivity

Definition 26. *A* ***disconnecting set*** *is a set $E' \in E(G)$ such that $G \setminus E'$ has more than one component.*

A graph is ***h-edge-connected*** *if there does not exist a disconnecting set of size $k - 1$. The* ***edge-connectivity*** *$\kappa'(G)$ is the minimum size of a disconnecting set.*

This definition simply extends what we already know about vertices to edges. A disconnecting set of size 1 has been earlier called a **bridge** .

line graph

We proceed as before: All graphs are 0-edge-connected and a graph that is 1-edge-connected if it is connected. It is 2-edge-connected if there is no bridge, etc.

We may test easily:

Definition 27. *The* ***line graph*** $L(G)$ *is obtained from* G *by making each edge of* G *a vertex in* $L(G)$ *and connecting two vertices if they share an endpoint in* G.

Example 21.

Theorem 36. *A bridge in* G *is an articulation vertex in* F *unless the bridge's removal leaves a single isolated vertex.*

Proof.

Whitney's Theorem
x,y-cut
Menger's Theorem!Original

Thus bridges can be found in linear time.
The important result here is Whitney's theorem.

Theorem 37. (Whitney) $\kappa(G) \leq \kappa'(G) \leq \delta(G)$

Recall that $\delta(G)$ is the minimum vertex degree in G.

We note that $\kappa'(G) \leq \delta(G)$ because the edges incident on a vertex of minimum degree form a disconnecting set.
Also deleting an endpoint of each edge in a disconnecting set, deletes all edges in that set and thus $\kappa(G) \leq \kappa'(G)$. We have to be careful and this carefulness leads to a proof, but these are the ideas.

Now $\kappa(G) = \delta(G)$ for complete graphs, complete bipartite graphs, hypercubes and Harary graphs and so $\kappa'(G) = \delta(G)$ for these graphs also !

The case $\kappa'(G) < \delta(G)$ is the case when no minimum disconnecting set disconnects a single vertex from the graph.

Example 22.
$\kappa(G) = 1$ because 3 is an articulation vertex and G is connected.
$\kappa'(G) = 2$ because G has no bridges but $(2,3)$ and $(7,3)$ disconnect G if removed.
$\delta(G) = 3$ as the minimum degree is 3.

Thus $\kappa(G) < \kappa'(G) < \delta(G)$ and so it is possible that the strict inequalities hold as well as the equalities.

Definition 28. *A set $S \subseteq V(G) \setminus \{x,y\}$ is an **x,y-cut** if $G \setminus S$ has no $x - y$ path. Let $\kappa(x,y)$ be the minimum size of an x,y-cut.*

The maximum number of pairwise internally-disjoint (intersecting only at the endpoints) x,y-paths is denoted by $\lambda(x,y)$.

$\lambda(G)$ is the longest k such that $\lambda(x,y) \geq k$ for all $x,y \in V(G)$.

Theorem 38. (Menger's original theorem) $\kappa(x,y) = \lambda(x,y)$.

In words: If $x,y \in V(G)$ and $xy \notin E(G)$, then the minimum size of an x,y-cut equals the maximum number of pairwise internally-disjoint x,y-paths.

Menger's Theorem!Modern

This is a local theorem about two vertices x and y.
Naturally, we may extend it to a global version.

Theorem 39. (Modern Menger Theorem) $\kappa(G) = \lambda(G)$.

Theorem 40. *If $x, y \in V(G)$, $(x \neq y)$, where G is a graph or directed graph, then the minimum size of an x,y-disconnecting set of edges equals the maximum number of pairwise edge-disjoint x,y-paths.*

Theorem 41. *$\kappa(G)$ is the maximum k such that $\lambda(x, y) \geq k$ for all $x, y \in V(G)$, i.e. $\kappa(G) = \lambda(G)$.*

$\kappa'(G)$ is the maximum k such that $\lambda'(x, y) = k$ for all $x, y \in V(G)$, i.e. $\kappa'(G) = \lambda'(G)$.

where we define $\lambda'(x, y)$ to be the maximum number of pairwise edge-disjoint x,y-paths.

circuit
cycle
Euler circuit
Hamiltonian Cycle
Chinese Postman Problem
Travelling Salesman Problem

Chapter 6

Circuits and Cycles

6.1 Concepts

Definition 29. *A* ***circuit*** *is a path from a vertex v to v in which no edge appears twice.*

A ***cycle*** *is a path from a vertex v to v in which no edge appears twice (except v which appears exactly twice).*

A ***Euler Circuit*** *is a circuit containing every edge.*

A ***Hamiltonian Cycle*** *is a cycle containing vertex*

The ***Chinese Postman Problem*** *asks for the shortest Eulerian circuit.*

The ***Travelling Salesman Problem*** *asks for the shortest Hamiltonian cycle.*

If a graph is not Eulerian (Hamiltonian), then we extend the Chinese Postman Problem (Travelling Postman Problem) to finding the shortest path from a vertex v to v covering all edges (vertices) at least once.

Example 23.

The Graph	***Eulerian***	***Hamiltonian***	***The circuit or cycle***
G	*no*	*no*	*no circuit, no cycle*
H	*yes*	*yes*	*circuit* = *cycle* $= 1 \to 2 \to 3 \to 4 \to 1$
I	*no*	*yes*	*cycle* $= 1 \to 2 \to 4 \to 3 \to 1$
J	*yes*	*no*	*circuit* $= 1 \to 2 \to 3 \to 1$

So any of properties is possible. It is o great practical importance to be able to judge if a graph is Eulerian or Hamiltonian and to find the required circuit or cycle if it exists. Some such problems are listed below:

1. Scheduling for mailman, delivery trucks, garbage trucks, snow plows, security agents, etc. And finding the routes.
2. More examples to be added on a later date.

6.2 Shortest paths

Before we investigate these problems, we solve a related one: Given two vertices $u, v \in V(G)$, determine the shortest path between them, if any.

We start with assigning each edge a weight. We will assume that all weight are positive integers, zero or infinity. Note that the prohibition of negative numbers and real numbers does not matter as we can always translate the unit in which we measure distance and the computer representation of **any** number is essentially that of an integer.

The problem is easily done via the Warshall algorithm that computes the connectivity matrix. We will give another algorithm here which is most commonly used for this problem and is to be recommended.

Let G be a graph (possibly directed) and $\omega(e)$ the weight of edge e. The adjacency list of vertex v is A_v and the starting vertex is s. It is **not** necessary to specify the terminal vertex as this procedure gives all shortest paths between s and every other vertex. No procedure is known which gives the shortest path between two vertices in less complexity-time than between any one and the others.

We first give the basic algorithm and then improve it.

```
Dijhstra(G,s)
$d(s) \leftarrow 0$
$T \leftarrow V(G)$
for $v \in V(G) - s$ do
    $d(v) \leftarrow \infty$
end for
while $T \neq \emptyset$ do
    find a $u \in T$ with $d(u)$ minimal
    $T \leftarrow T - u$
    for $v \in T \cap A_u$ do
        $d(v) \leftarrow min(d(v), d(u) + \omega_{uv})$
    end for
end while
```

Algorithm 7: Dijhstra Algorithm

This algorithm computes $d(t)$ which we **claim** is the shortest distance between s and t (that is the length of the shortest path). Note that with some bookkeeping the actual path may be output as well. Let the actual shortest distance be $d(s,t)$. We want to prove that $d(t) = d(s,t)$ to prove the correctness of the algorithm and to try to understand it.

If we adopt the convention that $d(s,t) = \infty$ if there is **no** path between s and t, we already have $d(s,t) \leq d(t)$, as $d(s,t)$ is the minimum by definition.

Example 24. (**Dijhstra**)

We have a weighted digraph and want to find the shortest path between vertex 1 and the others.

*We use **Dijhstra(G,1)**:*

1. $d(1) = 0;\ d(i) = \infty\ i = 2, 3, \ldots, 8;\ T = V$;
2. *find the* u *in* T *with* $d(u)$ *minimal* $\longrightarrow 1$ *with* $d(u) = 0$;
   $T = \{2, 3, ..., 8\}$
   $A_n = \{2, 3, 5\}$
   and so put $d(2) = \omega_{uv} = 28;\ d(3) = \omega_{uv} = 2;\ d(5) = \omega_{uv} = 1$;
3. *minimal one is 5*
   $T = \{2, 3, 4, 6, 7, 8\}$
   $A_n = \{2, 6\} \Rightarrow d(2) = d(5) + \omega_{25} = 9;\ d(6) = d(5) + \omega_{56} = 27$;
4. $u = 3$
   $T = \{2, 4, 6, 8, 7\} \Rightarrow d(6) = 26;\ d(8) = 29$;
5. $u = 2$
   $T = \{4, 6, 7, 8\} \Rightarrow d(4) = 18;\ d(6) = 19$;
6. $u = 4$
   $T = \{6, 7, 8\} \Rightarrow d(7) = 26;\ d(8) = 25$;
   Note that T *does not contain 5 even though 5 neighbors 4.*
7. $u = 6$
   $T = \{7, 8\} \Rightarrow d(8) = 20$;

8. $u = 8$
   $T = \{7\}$;

9. $u = 7$
   $T = \emptyset$;

and thus $d = (0, 9, 2, 18, 1, 19, 26, 20)$.

We want to show that $d(t) \leq d(s,t)$. We do this by induction. Consider $t = s$ and so $d(s) = d(s,s) = 0$ and the inequality holds. The induction hypothesis will be that $d(t) \leq d(s,t)$ for all vertices t that were removed from the set T before some vertex u.

Let $s = v_0 \longrightarrow^{l_1} v_1 \longrightarrow^{l_2} v_2 \ldots \longrightarrow^{l_n} v_n = u$ be a shortest path between s and u. Then by definition

$$d(s,u) = \sum_{j=1}^{n} \omega(e_j) \tag{6.1}$$

Choose i to be the largest index of a v_i such that v_i has been removed from T already. Then

$$d(s,v_i) = \sum_{j=1}^{i} \omega(e_j) = d(v_i) \tag{6.2}$$

by the induction hypothesis. (Note that if the path above is the shortest path to u, it is a shortest path to any v_i on that path.)

$$d(v_{i+1}) \leq d(v_i) + \omega(e_{i+1}) = d(s,v_i) + \omega(e_{i+1}) = d(s,v_{i+1}) \leq d(s,u) \qquad (*)$$

If $d(s,u) < d(u)$, then $d(v_{i+1}) < d(u)$ and we must remove v_{i+1} from T before u but this contradicts the assumption that i was minimal and thus,

$$d(s,u) \geq d(u)$$

For the case that $u = v_{i+1}$, the result is already written in the inequality $(*)$ above.

Thus $d(t) \leq d(s,t)$ and $d(t) \geq d(s,t)$ so $d(t) = d(s,t)$ for all t and Dijhstra is correct. It is easy to see that it runs in $O(|v|^2)$.

This run-time is not great and so we would like to improve it by introducing some data-structures to make that process faster.

A **heap** or **priority-queue** is a list of elements together with a priority of that element and a function to delete the element of minimal priority. This can be done in $O(log\ n)$ time with n being the number of elements in the heap.

We use $d(t)$ as the priority function and T as the heap. We thus get a slightly modified version that runs in $O(E\ logV)$.

```
Dijhstra(G,s)
$T \leftarrow s$
$d(s) \leftarrow 0$
for $v \in V(G) - s$ do
    $d(v) \leftarrow \infty$
end for
while $T \neq \emptyset$ do
    $u \leftarrow minT$
    delete $min$ from $T$
    for $v \in A_u$ do
        if $d(v) = \infty$ then
            $d(v) \leftarrow d(u) + \omega_{uv}$
            insert $v$ with priority $d(v)$ into $T$
        else
            if $d(u) + \omega_{uv} < d(v)$ then
                change priority of $v$ to $d(v) \leftarrow d(u) + \omega_{uv}$
            end if
        end if
    end for
end while
```

Algorithm 8: Dijhstra Algorithm, improved version

6.3 Eulerian Circuits

We know that we must have one component of even degree vertices to have an Eulerian circuit. The circuit will be given by Fleury's algorithm :

```
**Fleury(G)**
check that $G$ is connected and has even degree vertices
start at any vertex, $s$
$u \leftarrow s$
$C \leftarrow \{\}$
**while** $C \neq E(G)$ **do**
    find an edge adjacent to $u$ such that it is not a bridge of the
    untravelled part of the graph, unless there is no alternative. Let
    this edge be $(u, u')$
    add$(u, u')$ to $C$ in order
    when there is no more edge, we stop
    $u \leftarrow u'$
**end while**
```

Algorithm 9: Fleury's Algorithm

We need to modify this to work for directed graphs:

```
**Fleury(G)**
check that $d^-(v) = d^+(v)$ for all $v \in V(G) \& G$ connected
start at any vertex, $s$
let $G'$ be the graph obtained from $G$ by reversing all edge directions
use BFS/DFS to construct a tree $T'$ in $G'$
let $T$ be the reversal of $T'$; thus $T$ contains a $u - v$ path for each
$u \in V(G)$
order all edges leaving every vertex $u$ such that the edge in $T$ is last
construct the circuit: whenever $u$ is the current vertex exit along the
next unused edge in this ordering.
```

Algorithm 10: Fleury's Algorithm, modified for directed graphs

6.4 Chinese Postman

If the graph is Eulerian then any Euler circuit is the answer as they all have the same length (they cover every edge once).

If the graph is **not** Eulerian, it can be made Eulerian by adding edges to it. So we do it like this:

```
Edwards and Johnson (G)
Suppose there are $2k$ vertices of odd degree
find the shortest path between all pairs of **these** vertices
use these lengths to weight $K_{2k}$
The problem is then to find the minimum total weight of $k$ edges
that pair up the $2k$ vertices. This is known as weighted maximal
matching
Add then these edges to the graph
It's now Eulerian and any Euler circuit is a solution to the Chinese
Postman Problem
```

Algorithm 11: Edwards and Johnson

The matching problem will be discussed in the next chapter. It can be done in $O(|2k|^3)$ and the whole algorithm here runs in $O(V^3)$.

So even the duplicated version of Chinese Postman is solvable in polynomial-time.

It is clear that all algorithm work.

6.5 Hamiltonian Cycles

If we delete a vertex from a Hamiltonian graph, we leave at least a Hamiltonian path (covering every vertex exactly once, but not necessarily a closed path). Thus every Hamiltonian graph is 2-connected.

This is trivial to extend to further deletions: if $S \subseteq V(G)$, then $G \setminus S$ has at most $|S|$ components.

This is necessary but not sufficient condition. The Petersen graph satisfies the condition but it is **not** Hamiltonian.

If G has $|V(G)| \geq 3$ and $\delta(G) \geq \frac{|V(G)|}{2}$ then G is simple.

Ore's Theorem
Hamiltonian closure
Bondy-Chratal's Theorem
Chratal's Theorem
Chratal - Erdos Theorem

This bound is best-possible. To show that there exists a graph with just one less edge that it is not Hamiltonian consider the graph made of cliques of order $\lfloor \frac{n+1}{2} \rfloor$ and $\lceil \frac{n+1}{2} \rceil$ showing a vertex of degree $\lfloor \frac{n-1}{2} \rfloor$. This graph is not 2-connected and has $\delta(G) = \lfloor \frac{n-1}{2} \rfloor$.

We rephrase this to:

Theorem 42. (Ore)
Let G be a simple graph and u, v be non-adjacent vertices with $d(u) + d(v) \geq |V(G)|$, then G is Hamiltonian if and only if $G + uv$ is Hamiltonian.

We make a definition and can then rephrase again.

Definition 30. *The **Hamiltonian closure** of a graph is obtained from the graph by adding edges between pairs of non-adjacent vertices whose degree sum is at last $|V(G)|$, until no such pair remains.*

Theorem 43. (Bondy-Chrátal)
A simple graph is Hamiltonian if and only if its closure is Hamiltonian.

It follows that:

Theorem 44. (Chrátal)
Suppose the vertex degrees satisfy $d_1 \leq d_2 \leq \cdots \leq d_n$ and $i < \frac{n}{2} \Rightarrow (d_i > i$ & $d_{n-i} \geq n - i)$, then the graph is Hamiltonian.

Finally,

Theorem 45. (Chrátal - Erdös)
If the connectivity is at least equal to the independence number, then G is Hamiltonian (unless $G \cong K_2$).

6.6 Is G Hamiltonian ?

This question is NP-complete. Thus we must effectively search all possibilities.

```
HamCycle(G,P,e)
// initially call with $(S, \{\ \}, s)$ with $s$ any vertex
while $e$ has unvisited neighbors do
    $X \leftarrow$ some unvisited neighbor
    pruning:
    if the graph with $P$ contracted to $X$ admits an Ham. Cycle then
        $P \leftarrow P + X$
    end if
    if $P$ includes all vertices then
        if we can form a cycle then
            return success and output $P$
        else
            $P \leftarrow P - X$
        end if
    end if
    HamCycle(G,P,X)
end while
return failure
```

Algorithm 12: Hamiltonian Cycle

Pruning can get very heavy (?) but this is the only way to find Hamiltonian cycles in a same way.

6.7 Planar and Hamiltonian Graphs

Suppose G is hamiltonian **and** planar, then we construct a proper 4-coloring by coloring the regions inside the cycle A and B alternately and regions outside the cycle C and D alternately.

It is clear that this procedure colors the faces of the graph properly. We expand on this idea to give a planarity detection routine for Hamiltonian graphs.

```
HamPlanarity(G)
Draw the Hamiltonian cycle in the plane, this is $H$
List all edges $E = E(G) - E(H)$
Form the graph $K$ in which each $e \in E$ is a vertex and $\{e_i, e_j\}$ an edge if and only if $e_i$ and $e_j$ cross in the drawing given
$G$ is planar if and only if $K$ is bipartite.
```

Algorithm 13: Planarity detection for Hamiltonian graphs

Example 25.

with K - bipartite and so G is planar which we knew.

6.8 Line Graphs and Hamiltonian Graphs

An Eulerian circuit in G gives a Hamiltonian cycle in $L(G)$. This is obvious by construction of $L(G)$.

However, a Hamiltonian cycle in $L(G)$ does **not** imply an Eulerian circuit in G:

Here G is not Eulerian, although $L(G)$ is Hamiltonian.

Nevertheless it is a useful condition. Note that $L(G)$ above has $L(G) \cong K_3$ and $L(K_3) \cong K_3$. Thus $L(L(G)) \ncong G$ in general. Therefore we have to find some clever scheme for finding G given $L(G)$ or even determining whether a given graph is a line graph.

Consider:

the 2-valent vertices correspond to the marked edges in G

now construct the graph H from $L(G)$ by adding a vertex and two edges from this vertex to α and β.

This implies that we must add an edge to G that is adjacent to the edges α and β and **not** the other edges. As this is clearly impossible, H is **not** a line graph. Thus there exist graphs that are not line graphs.

Theorem 46. *A simple graph G is the line graph of simple simple graph iff $E(G)$ has a positive partition into cliques using each vertex of G at most twice.*

Beineke's Theorem
P
NP
NP-hard
NP-complete

This is interesting but not efficient to test.

Theorem 47. (Beineke)
A simple graph G is the line graph of some simple graph iff G does not contain any of the nine graphs below as an induced subgraph.

6.9 The Travelling Salesman

TSP = Find the Hamiltonian cycle of shortest total edge weight or if G is not Hamiltonian, find a cycle covering each vertex of minimum total edge weight.

Definition 31. *An algorithmic problem is*

- ***P*** *if there exists a polynomial-time algorithm for it.*
- ***NP*** *if a solution can be verified in polynomial-time.*
- ***NP - hard*** *if a polynomial-time algorithm can be used to give a polynomial-time algorithm for every problem in NP.*
- ***NP - complete (NPC)*** *when it is NP-hard and in NP.*

After we have one problem in NPC we may demonstrate that another problem is also in NPC by constructing a polynomial-time algorithm that translates any instance of the known problem to the desired problem **and** proving that the desired problem is NP.

There are many problems in NPC. One of them in TSP. By definition, if we found a polynomial-time algorithm for **any one** of these (by translation) we have a polynomial-time algorithm for all NP problems, i.e. we would have NP=P.

While no such algorithm exists, it has also **not** been proven that NP $\neq$ P.

If you prove NP $\neq$ P, then you would have to generate significant insight into the structure of algorithmic problems which would be invaluable for research. If you demonstrate (by an algorithm) that NP=P, then you would have done something of extreme practical importance. Currently most people think NP $\neq$ P merely by the weight of past attempts to demonstrate good algorithms.

Thus, for the moment, we are stuck with using heuristics. These give the right answer only sometimes. We hope that we can at least prove that the solution cannot differ from the optimum by some amount. We have:

Theorem 48. *For TSP, we have a polynomial-time algorithm that produces a solution at most twice the length of the optimal one when the triangle inequality holds:*

$$\omega(i,j) + \omega(j,k) \geq \omega(i,k)$$

for all i,j,k. This must hold if the weights are really distances in the plane, so this solves the classical TSP.

Proof.
Find a tree that meets all vertices and is of minimum weight. This is called the minimum-spanning tree. Let its weight be M.

Suppose we have the optimal TSP solution. If we delete one edge, we have a spanning tree of weight at least M.

Replace each edge of the tree by two oppositely oriented edges. Since $d^-(v) = d^+(v)$ for all v, this graph is Eulerian. It has $2n$ edges and weight $2n$.

Now reduce the number of edges (without increasing the total weight) while maintaining the property that the cycle visits all vertices, we obtain a cycle of cost at most $2n$.

This reduction can only be made due to the fact that the triangle inequality holds. Otherwise creating the cycle **may** necessitate the increase in total length.

6.10 Insertion Methods

As TSP is NPC we need heuristics to get an answer in practice. One example is the furthest-point insertion method that gets to within 5% to 10 % of the optimal tour length on average.

We note that the TSP can be asked in a variety of ways. The classic case of cities gives rise to an undirected complete graph. In this case the problem is symmetric and a proper solution (i.e. no repetition of vertices) is possible. If the problem is asymmetric, the problem is much harder. It is also harder if the graph is not complete because then it is not certain that a Hamiltonian cycle exists. The condition that the TSP tour is a Hamiltonian cycle is the triangle inequality:

In any undirected Hamiltonian graph that satisfies the triangle inequality ($\omega_{ij} \leq \omega_{ik} + \omega_{kj}$) no closet tour that visits every node can have weight smaller than the smallest-weight Hamiltonian cycle.

In general the graph does not obey the triangle inequality and it is not symmetric or Hamiltonian.

First compute the all-pairs shortest paths and start at any vertex. We have a partial tour and now select a new vertex to add to the row. Suppose the partial tour is

$$T = \{v_1, v_2, \ldots, v_k\}$$

Then we select the next vertex by this formula

$$v_{k+1} = \max_{v \in A(v_k)} \min_{i=1}^{|T|-1} (d(v, v_i) + d(v, v_{i+1})) \tag{6.3}$$

That is, from the vertices that we can reach from v_k, we select the vertices that add the smallest distance to the tour and from these select the worst one. This gets us a rough tour first and then allows us to fill in the details near the end. One might use other criteria but this seems to work best in practice.

We have two steps: (1) select a new vertex; (2) insert it somewhere in the already built tour. We maintain an array $d(v)$ that measures the shortest distance from v to the already built tour. The node with furthest distance is to be inserted next. Now we need to agree on where to insert it. For every possibility compute the cost ($c_{ij} = \omega_{if} + \omega_{fj} - \omega ij$) over all edges (i, j) in the tour and the new vertex f. Then select the edge (t, h) with smallest cost and add f between them. Thus:

k-optimal

```
FITSP:
$v_T \leftarrow \{s\}$                        // we have an initial cycle of one node
$E_T \leftarrow \{(s,s)\}$              //and zero weight. This is fictitious of course
$\omega_{ss} \leftarrow 0$
$t_\omega \leftarrow 0$                                              // the total weight
for every node $u \in V \setminus V_T$ do
    $d(u) \leftarrow \omega_{sn}$
end for
while $|V_T| < |T|$ do
    $\leftarrow$ vertex in $V \setminus V_T$ with largest $d(u)$
    for every edge $(i,j) \in E_T$ do
        $c_{ij} \leftarrow \omega_{if} + \omega_{fj} - \omega ij$
    end for
    $(t,h) \leftarrow$ edge in $E_T$ with smallest $c_{th}$
    $E_T \leftarrow E_T \cup \{(t,f),(f,h)\} \setminus \{(t,h)\}$
    $V_T \leftarrow V_T \cup \{f\}$
    $t_\omega \leftarrow t_\omega + c_{th}$
    for all $x \in (V \setminus V_T)$ do
        $d(x) \leftarrow min(d(u), \omega_{fx})$
    end for
end while
```

Algorithm 14: FITSP

Observe that this is $O(n^2)$ and that it depends on the choice of initial vertex. If we are allowed $O(n^3)$ then run this for all starting vertices and select the minimum.

6.11 k-optimal Methods

Suppose that we have a tour and we would like to improve it. Suppose we delete $k \geq 2$ edges from the tour and add edges to make it a tour again with lower weight. We call a tour **k-optimal** when the re exists no set of k edges that can be improved like this. From experiments we conclude that 3-optimal are within a few percent of the optimal tour. For $k > 3$, the computation time needed increases sharply the relative increase in the solution quality. Note that the chosen edges do not have to be adjacent.

So, for each edge-triple, select three replacement edges that lower the total weight. Continue until no such replacements are possible.

Example 26.

Figure 6.1: solid lines - paths kept; dashed lines - edges deleted; pointed lines - new edges

acyclic
forrest
tree
leaf
pendant
spanning subgraph
spanning tree

Chapter 7

Trees

7.1 Basics

Definition 32. *A graph without cycles is called* ***acyclic*** *. A* ***forrest*** *is an acyclic graph and a* ***tree*** *is a connected forrest. A* ***leaf*** *or* ***pendant*** *is a vertex of degree 1. A* ***spanning subgraph*** *of a graph G is a subgraph with vertex set $V(G)$. A* ***spanning tree*** *is a spanning subgraph that is also a tree.*

Note that a spanning subgraph need not be connected.

First we note that a tree, upon deletion of a leaf, is still a tree. Thus trees can be grown by adding leaves. This is useful for roving results by induction.

Theorem 49. *Every tree with at least two vertices has at least two leaves. Deleting a leaf from a tree with n vertices leaves a tree with $n-1$ vertices.*

Theorem 50. *For a graph on n vertices ($n \geq 1$) te following are equivalent:*

1. *G is a tree*
2. *G is connected and acyclic*
3. *G is connected and has $n-1$ edges*
4. *G has $n-1$ edges and no cycles*
5. *For any $u, v \in V(G)$, G has exactly one u,v-path.*

The characteristics allow checking of the question: Is G a tree ? Clearly a single DFS or BFS is sufficient to test condition 3 and this is simples to check.

Theorem 51. *Every connected graph has a spanning tree (delete an edge for every cycle).*

Pigeonhole Principle
eccentricity
diameter
radius
center

Theorem 52. (Pigeonhole Principle) *Every n-vertex graph with $n-k$ edges has at least k components.*

Theorem 53. *If T is a tree with k edges and G is a simple graph with $\delta(G) \geq k$, then T is a subgraph of G. (use induction on k)*

Theorem 54. *In a tree, every edge is a bridge and adding any edge creates exactly one cycle.*

Theorem 55. *If T and T' are spanning trees of a connected graph G and $e \in E(T) \setminus E(T')$, then there is an edge $e' \in E(T') \setminus E(T)$ such that $T-e+e'$ and $T+e-e'$ are spanning trees of G.*

Proof. Every edge of T is a bridge. Let U and U' be the vertex sets of the two components of $T-e$. Since T' is connected it has an edge e' with exactly one endpoint in U and U'. Thus $T-e+e'$ is a spanning tree. Same thing in reverse for the other case. □

Definition 33. *The **eccentricity** $\epsilon(u)$ is the maximum distance of the vertex u from other vertices. The **diameter** $diam(G)$ and **radius** $rad(G)$ are the maximum and the minimum eccentricities in G respectively. The **center** of G is the subgraph induced by the vertices of minimum eccentricity.*

Recall that the graph G' induced by $A \subseteq V(G)$ from G is the graph $V(G') = A$ and $E(G')$ consisting of all edges with both endpoints in A.

Every path in a tree is the only path between those two vertices and thus the diameter of a tree is the length of the largest path.

Theorem 56. *The center of a tree is one vertex or one edge.*

Theorem 57. *If u is a vertex on a tree T of n vertices, then*

$$\sum_{v \in V(T)} d(u,v) \leq \binom{n}{2} = \frac{n!}{(n-2)!2!} = \frac{n(n-1)}{2}$$

[for P_n this sum is clearly $\sum\limits_{i=0}^{n-1} i = \binom{n}{2}$ and we extend this by induction on n for all trees.]

Theorem 58. *If H is a subgraph of G, $d_G(u,v) \leq d_h(u,v)$.*

Theorem 59. *If u is a vertex of a connected graph G*

$$\sum_{v \in V(G)} d(u,v) \leq \binom{|V(G)|}{2}$$

[we know it for trees and the above theorem is obvious].

Wiener index
Cayley's Formula
Prufer's Sequence

Definition 34. *The* ***Wiener index*** *is*

$$W(G) = \sum_{u,v \in V(G);\ u \neq v} d(u,v)$$

for any graph G.

If we draw a graph of a molecule by vertices for atoms and edges for bounds, we may study the chemical structure of elements using graph theory. This index was originally used to study the boiling point of paraffin.

7.2 Counting Trees

Theorem 60. *There are n^{n-2} trees with vertex set $[n] = \{1, 2, ..., n\}$.*

The idea is to establish a bijection between the trees and the sequences of length $n-2$ of $[n]$ of which there are n^{n-2}. The formula is known as **Cayley's formula** and the sequence as **Pr*ü*fers's sequence** .

To compute the Pr*ü*fer sequence for a tree, iteratively delete the last leaf from the tree (in terms of the vertex labels) and add the label of its neighbor to the sequence.

Example 27.

We delete in this order:
1,5,2,4,3
and so the sequence is
42436.
We have obtained a sequence of length $n-2$.

Suppose we have a sequence a of length n. We begin with $n+2$ isolated vertices. At the i^th step: let x be the label of a in position i and let y be the smallest label that does not appear after i in a and has not been marked "finished". Add the edge yx and mark y "finished". After exhausting a, join the two remaining unfinished vertices by an edge. The result is a tree. We will not prove that this is the inverse of the above, but it seems plausible.

incidence matrix
rank
totally unimodular

Example 28.

a=42436 and we need 7 vertices:
we add the edges xy in this order:
41, 25, 42, 34, 63, 67
α, β, γ, δ, ϵ, ξ
which is obviously isomorphic to the above example.

Theorem 61. *The number of trees with vertex set $[n]$ and the vertices $1, 2, ..., n$ with degrees $d_1, d_2, \ldots, d_n$ respectively is*

$$\frac{(n-2)!}{\prod_{i=1}^{n}(d_i-1)!}$$

7.3 Counting Spanning Trees

Suppose G is directed and e_k is an edge, then

$$\omega_i(e_k) = \begin{cases} 1 : \text{if } v_i \text{ is the head of } e_k \\ -1 : \text{if } v_i \text{ is the head of } e_k \\ 0 : \text{not in } e_k \end{cases}$$

The matrix $M = \omega_i(e_k)$ is called the **incidence matrix** of the graph G.

The number of linearly independent rows is called the **rank** of a matrix.

Theorem 62. *If G has n vertices and p connected components, then the rank of its incidence matrix is $n - p$.*

Moreover G is a tree if and only if all columns are linearly independent.

A matrix is called **totally unimodular** if the determinant of any square submatrix is either ± 1 or 0.

Theorem 63. *The incidence matrix of any graph is totally unimodular.*

Theorem 64. *Let B be the matrix obtained from the incidence matrix of G by deleting any row. Then G is a tree if $|det(B)| = 1$ and not a tree if $det(B) = 0$.*

Theorem 65. *The number of spanning trees of G is $det(BB^T)$.*

So far this was for the directed case. For undirected graphs we have:

decomposition
Ringel's Conjecture
Haggkvist's Conjecture
Graceful Tree Conjecture

Theorem 66. *If G is a loopless, undirected graph whose adjacency matrix is A then we define*

$$Q = -A + diag(d_1, d_2, \dots, d_n)$$

where d_i is the degree of vertex i and the matrix Q^* is obtained from Q by deleting any row a and column a from Q. Then the number of spanning trees is

$$det(Q^*).$$

7.4 Decomposition

Definition 35. *A **decomposition** of a graph G is a partition of $E(G)$ into pairwise edge-disjoint subgraphs.*

So any graph can be partitioned into parts consisting of only one edge. We ask if we can decompose G into a number of isomorphic copies of a tree T.

Conjecture 2. (Ringel's Conjecture) *If T is a fixed tree of m edges, then K_{2m+1} can be decomposed into $2m + 1$ copies of T.*

Conjecture 3. (Häggkvist's Conjecture) *If T is a fixed tree of m edges, then any $2m$-regular graph can be decomposed into $V(G)$ copies of T.*

Both are unknown. Proofs have focused on a more powerful result:

Conjecture 4. (Graceful tree Conjecture)
*It T is a tree of m edges, then the vertices if T can be labelled with distinct numbers $0, 1, 2, ..., m$ such that edge-differences are $\{1, 2, ..., m\}$. Such a labelling is called **graceful***

Example 29.

minimum spanning tree
greedy algorithm
Kruskal Algorithm

Example 30.

K_5 can be decomposed into 5 copies of T as numbered below.

7.5 Minimum Spanning Tree

Definition 36. *The* ***minimum spanning tree*** *T of a graph G is a spanning tree of minimum total edge-weight of G.*

The graph must be connected and we assume that it is weighted as otherwise any spanning tree would have the same weight (as they al have the same number of edges).

Definition 37. *A* ***greedy algorithm*** *is an algorithm that, at each step, chooses the best next step regardless of history of future steps, i.e. it is a local choice.*

Most of the time greedy algorithms are fast but deliver suboptimal solutions. Here a greedy idea gives the optimal solution:

Kruskal Algorithm:

1. Sort all edges in order of increasing weight.
2. Add the shortest edge left to the tree if and only if it creates no cycle, i.e. if its inclusion joins two as-yet unconnected components of the tree. We resolve conflicts arbitrarily.

After sorting the edges, we maintain the component in which each vertex is in memory. We accept the next edge if the component number of the endpoints differ. Then we update the vertex labels. The search is $O(n)$ and the update $O(log_2n)$ for n vertices and **pre-sorted** edges. If we also have to sort the edges we have to multiply this by the complexity of the sorting in the number of edges.

Theorem 67. *Kruskal's algorithm is correct.*

Proof.
Clearly it produces a spanning tree, T. Assume that the minimal spanning tree is T'. Suppose $T \neq T'$ and let e be the first edge in T that is not in T'.

Adding e to T' creates a cycle so there is an edge e' in T' that is not in T. But $T' + e - e'$ is a spanning tree. Since T' contains e' and all edges chosen before e, both edges e and e' are available when the construction algorithm has to choose, and hence chooses e. Thus $\omega(e) \leq \omega(e')$. Thus $T' + e - e'$ has weight at most that of T' and contains a longer piece of T. Iterating this shows that we can make T' to contain all of T and that thus T is of minimum weight.

The graph on the left has its edges sorted and we insert the edges into the tree on the right in the order of the Greek letters. This is the minimum spanning tree of that graph.

As the number of edges can scale as n^2, sorting the edges by weight makes this algorithm potentially expensive. We have another greedy idea to improve it.

Prim:

We grow the spanning tree from a single vertex and add an edge between the tree already built and the remaining vertices by choosing the cheapest such edge. Ties are broken arbitrarily.

Boruvka:

Start from a forrest of isolated vertices. Pick the cheapest edge having each component of the current forrest.

Huffman's Algorithm

These algorithms are all competitive if the edge-weights are pre-sorted. Like this we can implement Kruskal in $O(nlog_2n)$ and Prim runs in $O(n^2)$. Thus Prim is good for dense graphs and Kruskal for sparse ones.

We want the minimum spanning tree for example to be able to find the minimum amount of wire to connect n towns together.

7.6 Huffman encoding

Given some text, we want to store it efficiently. We represent each character by a bit sequence such that frequent characters are assigned short sequences. As these sequences differ in length,we must be able to decide when one is finished, i.e. no sequence may be a prefix of another sequence.

The prefix-tree condition is satisfied if we assign the codes from a tree in which each vertex has two children and left means 0 while right means 1. Shorter sequences terminate at a shorter path from the root vertex.

The length of a message then is $\sum p_i l_i$ where l_i is the length of the code assigned to p_i. Normally we need to use the same length codes for each and this gives the ratio.

How do we deal with frequencies then? We need to kill some branches corresponding to likely items:

Huffman's Algorithm:

- **input:** n probabilities $p_1, p_2, ..., p_n$
- **base case:** $n = 2$. The tree $0 - 1$ will do, i.e. we assign the symbol 0 to p_1 and 1 to p_2
- **recursion:** select the two least likely items p and p' and replace them by a single item of probability $p + p'$. Solve this new problem.

  After the solution, give two children to the vertex corresponding to the combined vertex and give them weights p and p'.

  In other words, replace the code word from the combined item with by the two words corresponding to p and p' by appending a 0 and a 1 respectively.

Example 31. *$n = 5$ and $p = (0.1,\ 0.2,\ 0.3,\ 0.15,\ 0.25) =: (A, B, C, D, E)$. and we combine in this order:*

$$\begin{gathered} F = A + D \ (0.25) \\ G = B + E \ (0.45) \\ H = C + F \ (0.55) \\ I = G + H \ (1.0) \end{gathered}$$

and this looks like:

and so we put $B = 00,\ E = 01,\ C = 10,\ A = 110,\ D = 111$. The code reduction is $\sum_{i=1}^{5} p_i l_i = 2.25$ as opposed to 3.

matching
marriage!unstable
marriage!stable

Chapter 8

Matching

8.1 Stable Marriage

Let there be n men and women. For each man/woman, we have a list of members of the other sex in the order of preference for marriage. A **matching** is an assignment of monogamous marriages, i.e. a bijection between men and women.

A marriage is **unstable** if two people of different sex prefer each other to their spouses. It is **stable** if it is not unstable.

Example 32. *Let men be A, B, C, D and the women be a, b, c, d. Then consider the following tables.*

Man:		***Women:***	
A	$cbda$	a	$ABDC$
B	$bacd$	b	$CADB$
C	$bdac$	c	$CBDA$
D	$cadb$	d	$BACD$

This gives rise to many matchings, for example (Aa, Bb, Cc, Dd) which is unstable because A and b prefer each other to their spouse.

One may think that arranging some divorces and some marriages will lead to a stable marriage. While this is true it depends heavily on the method chosen.

We generalize in two ways: get rid of monogamy and make some people infinitely undesirable. We could also force some given marriages.

Polygamous stable marriage occurs when m universities want to admit n students. More than one student goes to a university but never to more than one. Let the k^{th} university admit u_k students. By creating either fictitious students or fictitious universities both very undesirable, we create a situation in which every student is taken and all places are filled, i.e.

$$u_1 + u_2 + \cdots + u + m = n$$

This problem is translated into stable marriage by replacing each university by u_k places that each have the same wish list, that of the university.

If the list of preferences do not include all n members of the other sex, the missing people are effectively infinitely undesirable. So we introduce the obvious condition that the pairing must be with a person on the list.

What we do here is to introduce a widower V and a widow W to the set of men and women respectively. V will be $\omega's$ last choice, ω will be $V's$ last choice. Women will have their existing listing, append V to the end and list any missing men in arbitrary order. Men will do likewise with ω.

For these versions we find:

Theorem 68. *The stable marriage problem is solvable, i.e. there always exists at least one stable marriage.*

Theorem 69. *The incomplete marriage problem is solvable if and only if the augmented system has a stable marriage with V married to ω.*

Theorem 70. *If there exists a stable marriage of the augmented incomplete marriage problem with V married to ω, then V is married to ω for all stable marriages.*

Having the existence theorem (hunting licence) we must find it (do the hunting).For this, let n be the number of women and men, k is the number of couples already formed, X is a suitor, ω is the woman to whom X advances and Ω is the least desirable man - we introduce this as a further man, not already listed and append it to the end of women's lists, i.e. Ω is the termination character of a woman's preference list.

```
Stable Marriage:
$k \leftarrow 0$                [init: all women are temporarily engaged to $\Omega$]
while $k < n$ do
    $X \leftarrow (k+1) - th$ man
    while $X \neq \Omega$ do
        $\omega \leftarrow$ best remaining choice in the list of $X$
        if $\omega$ prefers $X$ to her fiancé then
            engage $\omega$ and $X$
            $X \leftarrow$ preceding fiancé of $\omega$
        end if
        if $X \neq \Omega$ then
            withdraw $\omega$ from $X$'s list
        end if
    end while
    $k \leftarrow k + 1$
end while
celebrate !!!
```

Algorithm 15: Stable Marriage

We note that the following points are valid at any time during the algorithm.

- If a is removed from A's list, no stable marriage can contain Aa.
- If A prefers a to his fiancé, it means that a has rejected him for another.
- Two women are never the fiancés of one man (except Ω).
- A woman's situation never gets worse.
- No man's preference list is emptied.
- The obtained matching is stable [holds at the end].

In fact: note that the roles of men and women can be switched.

Theorem 71. *This algorithm produces the optimal solution for all men, i.e. there exists no stable marriage in which any man could have a spouse he prefers to his current one.*

Theorem 72. *In any stable marriage every woman has a partner superior or equal to her current one, i.e. the algorithm produces the worst answer for women.*

Take-home lesson:
The more **you** take initiative in anything, the better off you will be.

regret
matching
saturated
matching!perfect
1-factor

[This algorithm applies very generally to life and should be carefully studied for strategy in general.]

The largest number of proposals in this algorithm is $n^2 - n + 1$. If we define:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + frac1n = ln(n) + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \epsilon \tag{8.1}$$

where $0 < \epsilon < \frac{1}{120n^4}$ and γ= Euler's number = 0.57721566...
then it can be shown that the mean number of proposals for random preferences is $nH_n + O(n\ log^4 n)$ and the average complexity is thus $O(n\ ln\ n)$. This is a rare case of an algorithm for which the complexity of the average case is known.

So far the solution is best for one group and worst for the other group. Can we do ti fairly? Define the **regret** of a person as the distance to the spouse of that person's list. Let U be the maximum regret of $2n$ people. We now try to minimize U.

Finding an optimal solution for all men and then also for all women, we get upper an lower bounds on the regret for every individual. We form marriages between those people with equal upper and lower bounds and so reduce the system.

Then we chose a woman [could be a man] at random from those of greatest upper bound. Let this woman be a. Let A be the worst remaining choice of a. We also note that a is the best choice of A among the remaining women. We remove a from A's list and obtain a new optimal solution for the men. This increases the lower bound of A and at least one more man. It also lowers the upper bound of a and at least one more woman. It is clear that we will find a stable marriage in $O(n^2)$ iterations, i.e. this is an $O(n^3\ ln\ n)$ algorithm.

Take-home lesson:
Selfishness leads to a good solution for you. A fair solution requires cooperation between all concerned parties.

8.2 Matching

We have investigated a case of a matching problem so far. Let us treat this in more generality then:

Definition 38. *A* ***matching*** *of size k in a graph G is a set of k pairwise disjoint edges. The vertices covered by these edges are called* ***saturated*** *. If a matching saturates every vertex of G, it is a* ***perfect matching*** *or a* ***1-factor*** *.*

Clearly the stable marriage assignment was a perfect matching. In the graph $K_{n,n}$ we have $n!$ perfect matchings; in K_{2n+1} the same and in k_{2n} there are $\frac{2n!}{2^n n!}$. Why is that so ?

maximal matching
M-alternating path
M-augmenting path

We seek a large matching in the sense of total edge weight. This would be the satisfaction of the people in the marriage case for example. We imagine that we have some matching. We want to enlarge it by iteratively selecting an edge disjoint from the previous ones. The result is a **maximal matching** i.e. one which can not be easily enlarged but this is not, in general, a maximum matching. Consider P_4:

The edge $2-3$ is a maximal matching but the maximum matching has two edges, $1-2$ and $3-4$.

Definition 39. *For any matching $\mathcal{M}$ an $\mathcal{M}$-**alternating path** is a path that alternates between edges in $\mathcal{M}$ and edges not in $\mathcal{M}$. An $\mathcal{M}$-**augmenting path** is an $\mathcal{M}$-alternating path starting and ending on unsaturated vertices. If P is an $\mathcal{M}$-augmenting path and $E(P)$ is the edge set, we observe that $\mathcal{M} \cap E(P) \longrightarrow E(P) \setminus \mathcal{M}$ produces a new matching $\mathcal{M}'$ from $\mathcal{M}$ with one more edge.*

Thus maximum matching have no augmented paths !

Theorem 73. *A matching is minimum if and only if the graph has no augmented paths.*

8.3 Bipartite Matching

The marriage problem was a matching problem in a bipartite graph as two men and two women are not allowed to marry, at least in the traditional case. We might suppose that there are more women than men (which is actually true in the world today) and so no perfect matching exists.

Thus we ask if there exists a matching saturating the men, such a matching is called a matching of {men} into {women}.

min-max relation
independence number
edge cover

We note that for any group of men, there must be at least as many women liking some of them. Thus if we want a matching of set X into Y, for any $S \subseteq X$, there must be at least $|S|$ vertices in Y having neighbors in S, i.e. if we represent the set of vertices of Y with neighbors in S by $N(S)$, then $|N(S)| \geq |S|$.

Theorem 74. *If G is bipartite with partite sets X and Y, then G has a matching from X into Y if and only if $|N(S)| \geq |S|$ for all $S \subseteq X$.*

Theorem 75. *For $k > 0$, every k-regular bipartite graph has a perfect matching.*

This last theorem is a generalization of the fact the one may always marry the men to women (not necessarily stably).

Given a matching, we would like to determine if it is maximum. We could look for an augmenting path but that would take exponentially long. The following helps:

Theorem 76. *If G is bipartite, then the maximum size of a matching equals them minimum size of a vertex cover of G, a set of vertices that contains at least one endpoint of every edge.*

This is known as the **min-max relation** as it relates the answers of a minimization to a maximization problem. So we can prove that the matching **AND** the cover are both optimal merely by exhibiting that they are of the same size.

This is nice but still not helpful. We thus define the **independence number** to be the maximum size of an independent set of vertices, i.e. vertices that are pairwise not connected. This is the analogue of finding a matching. We can also define a covering problem so that we can prove a min-max relation.

An **edge cover** is a set of edges that cover the vertices, i.e. a vertex belongs to exactly one edge in the cover. We denote:

max size	independent set	$\alpha(G)$
max size	matching	$\alpha'(G)$
max size	vertex cover	$\beta(G)$
max size	edge cover	$\beta'(G)$

We may prove a few nice theorems.

Theorem 77. *If G is bipartite, $\alpha'(G) = \beta(G)$. Note that this is the previous theorem in our new notation.*

Theorem 78. *In a graph G, $S \subseteq V(G)$ is an independent set if and only if $\overline{S}$ is a vertex cover and thus $\alpha(G) + \beta(G) = n(G)$. Recall that $\overline{S}$ is the complement of S in $V(G)$.*

Theorem 79. *If G has no isolated vertices, $\alpha'(G) + \beta'(G) = n(G)$.*

Theorem 80. *If G is bipartite and has no isolated vertices, then $\alpha(G) = \beta'(G)$.*

8.4 Finding a Bipartite Matching

Those conditions did not allow us to actually find a matching in a bipartite graph. So now we solve this remaining issue.

```
Augmenting Path Algorithm(G)
Data    : $G$ is bipartite, $X$ and $Y$ are partite sets
$\mathcal{M} \leftarrow$ some matching
$U \leftarrow$ all $\mathcal{M}$ unsaturated vertices in $X$
$S \leftarrow U$
$T \leftarrow \emptyset$
while $S$ has no unmarked vertex do
    $X \leftarrow$ an unmarked vertex in $S$
    for every $y \in N(X)$with $xy \notin \mathcal{M}$ do
        if $y$ unsaturated then
            report $\mathcal{M}$-augmenting path from $U$ to $y$
        else
                                        // match $y$ to some $w \in X$
            $T \leftarrow T + \{y\}$
            $S \leftarrow S + \{w\}$
        end if
    end for
    mark $X$
end while
```

Algorithm 16: Augmenting Path Algorithm

This produces a matching and vertex cover of the same size and thus optimal. This algorithm runs in $O(n^3)$ time. It has not been written in a full detail because it is not that good but presents the main idea. We go on to consider the weighted case which is a generalization of this case.

transversal
weighted cover
minimum weighted cover
equality subgraph

8.5 Weighted Bipartite Matching

Each man/woman assigns a desire to each member of the other sex. A desire of 0 is assigned for forbidden people. We thus seek the minimum weight matching of $K_{n,n}$.

Definition 40. *A* ***transversal*** *of an n by n matrix A consists of n elements, one from each tow and column.*

Compare with the rook problem in chess.

Finding a transversal of A with maximum sum is weighted matching if the entries of A are the edge weights.

Given the weight ω_{ij} a **weighted cover** is a choice of labels $\{u_i\}$ and $\{v_j\}$ such that $u_i+v_j \geq \omega_{ij}$ for all i, j. The cost $c(u, v)$ of the cover is $\sum u_i+\sum v_j$. The **minimum weighted cover** problem asks for a cover of minimum weight. That is, we are asked to place large enough labels on the vertices in order to "cover" the edges. We seek the least values such that this is satisfied.

Theorem 81. *If $\mathcal{M}$ is a perfect matching in a bipartite graph G and u, v is a cover, then $c(u, v) \geq \omega(\mathcal{M})$. And $c(u, v) = \omega\mathcal{M}$ if and only if $\mathcal{M}$ consists of edges x_iy_j such that $u_i + v_j = \omega_{ij}$. In this case $\mathcal{M}$ is a maximum weight matching and u, v is a minimum cost cover.*

This is the weighted analogue to the min-max relations above. We define:

Definition 41. *The* ***equality subgraph*** *$G_{u,v}$ w.r.t. a cover u, v is the spanning subgraph of $K_{n,n}$ containing the edges x_iy_j such that $u_i + v_j = \omega_{ij}$. If $G_{u,v}$ has a perfect matching, then it has weight $\sum u_i + \sum v_j$ and thus we have an optimal matching and cover. If not, we change the cover.*

This idea gives an algorithm.

We input a matrix A of weights on $K_{n,n}$ with a bipartition X and Y.

symmetric difference

```
Hungarian Algorithm
$u_i \leftarrow max_j\ \omega_{ij}$
$v_i \leftarrow 0$
$\mathcal{M} \leftarrow$ maximum matching in $G_{u,v}$; the equality subgraph for the cover $(u, v)$
while $\mathcal{M}$ is not perfect do
    $U \leftarrow$ set of $\mathcal{M}$-unsaturated vertices
    $S \leftarrow$ set of vertices in $X$ reachable by alternating paths from $U$
    $T \leftarrow$ set of vertices in $V$
    $\epsilon \leftarrow min\{u_i + v_j - \omega_{ij}\ :\ x_i \in S,\ y_j \in Y - T\}$
    for all $x_i \in S$ do
        $u_i \leftarrow u_i - \epsilon$
    end for
    for all $y_j \in T$ do
        $v_j \leftarrow v_j + \epsilon$
    end for
    $G' \leftarrow$ the new equality subgraph
    if $G'$ has an augmenting path then
        $\mathcal{M} \leftarrow$ maximum matching in $G'$
    end if
end while
```

Algorithm 17: Hungarian Algorithm

This algorithm terminates in at most $\frac{n^2}{2}$ iterations. This can be improved if we seek the augmenting paths in a special order. Using breath-first search simultaneously from all vertices of X that are unsaturated, we find many paths of the same length with this single examination of the edge set. We can show that subsequent paths must be longer than previous paths and thus these searches can be grouped in phases of finding paths of the same length.

Definition 42. *If G and H are graphs with vertex set V, then the **symmetric difference** $G \triangle H$ is the graph with vertex set V whose edges appear in exactly one of G or H. We also use this notation for sets of edges. So if $\mathcal{M}$ and $\mathcal{M}'$ are matchings, then*

$$\mathcal{M} \triangle \mathcal{M}' = (\mathcal{M} \cup \mathcal{M}') \setminus (\mathcal{M} \cap \mathcal{M}')$$

We not that if $\mathcal{M}$ is a matching of size r and $\mathcal{M}^*$ is a matching of size s, with $s > r$, then at least $s - r$ vertex disjoint $\mathcal{M}$-augmenting paths exist because at least that many paths may be found in $\mathcal{M} \triangle \mathcal{M}^*$.

From that it follows that:

Lemma 1. *If P is a shortest $\mathcal{M}$-augmenting path and P' is $\mathcal{M}\triangle P$-augmenting, then $|P'| \geq |P| + |P \cap P'|$ where P is an edge set.*

Lemma 2. *If $P_1, P_2, ...$ is a sequence of shortest successive augmentations, then the augmentations of the same length are vertex disjoint paths.*

So we search simultaneously from all unsaturated X vertices for shortest augmenting paths and this yields vertex disjoint paths after which all augmenting paths are longer. This all augmentations can be found by one examination of the edge set, i.e. in $O(m)$. We can now prove that there will be at least $2\lfloor\sqrt{n}\rfloor + 2$ of this so that the total weighted bipartite matching can be done in $O(m\sqrt{n})$, and if the graph scales like $O(n^2)$, the algorithm takes $O(n^{2.5})$ time.

The algorithm then looks like:

```
Algorithm A
Step 0.
$\mathcal{M} \leftarrow \emptyset$
Step 1.
Let $l(\mathcal{M})$ be the length of a shortest augmenting path relative to $\mathcal{M}$.
Find a minimal set of paths $\{Q_1, Q_2, ..., Q_i\}$ with the properties that
(a) for each $i$, $Q_i$ is an augmenting path relative to $\mathcal{M}$ and $|Q_i| = l(\mathcal{M})$;
(b) the $Q_i$ are vertex-disjoint
Halt if no such path exists.
Step 2.
$\mathcal{M} \leftarrow \mathcal{M} \oplus Q_1 \oplus Q_2 \oplus ... \oplus Q_i$; go to step 1 ;
```

Algorithm 18: Maximum Matching Algorithm **A**

Where the crucial step is more complicated and requires a few data structures:

We assume that the graph is represented as follows: for each vertex u, a read-only linear list LIST(u) is given containing, in arbitrary order, the vertices v such that (u, v) is an edge. The algorithm also uses an auxiliary last-in first-out list called STACK, which is initially empty, and a set B of vertices which is initially the empty set. The following primitives occur in the algorithm.

factor
k-factor

Variables	
TOP	top element of STACK
FIRST	
Operations	
PUSH x	push element x onto STACK
POP	pop an element from STACK
DELETE	delete the first element from LIST(TOP)
PRINT	POP until STACK is empty and print the successive elements
Predicates	
EMPTY	STACK is empty
NULL	LIST(TOP) is empty

Using these variables we may write down the algorithm:

```
Algorithm B
PUSH $s$
while $STACK \neg EMPTY$ do
    while $LIST(TOP) \neg NULL$ do
        FIRST = first element of LIST(TOP)
        if $FIRST \notin B$ then
            PUSH FIRST
            if $TOP \neg = t$ then
                $B \leftarrow B \cup \{TOP\}$
            else
                PRINT, PUSH $s$
            end if
        end if
    end while
    POP
end while
```

Algorithm 19: Maximal Set of $s - t$ paths

This finds the maximum weight bipartite matching, i.e. the marriage assignment of maximum desire or satisfaction for example. What if we want to do this for a general (non-bipartite) graph ?

8.6 Factors in Graphs

Definition 43. *A* ***factor*** *of a graph G is a spanning subgraph of G. A* ***k-factor*** *is a k-regular factor (a perfect matching is a 1-factor). An* ***odd***

odd component
f-factor

***component** is a component of odd order (number of vertices). The number of odd components of G is $o(G)$.*

Theorem 82. *A graph G has a 1-factor iff $o(G \setminus S) \leq |S|$ for every $S \subseteq V(G)$.*

We see this by analogy to previous theorems. Thus there may be a gap between the largest matching and the smallest vertex cover.

We note that $o(G \setminus S) - |S|$ has the same parity as $u(G)$ and thus $o(G \setminus S)$ exceeds $|S|$ by at least two for some S if $u(G)$ is even, and G has no 1-factor.

Theorem 83. *The largest number of vertices in a matching in G is*

$$\min_{S \subseteq V(G)} \{n - d(s)\}$$

where $d(s) = o(G \setminus S) - |S|$.

Thus to prove that we have a maximum matching, we must exhibit a vertex set S whose deletion leaves the appropriate number of odd components. This is short, but finding S is hard.

Theorem 84. *Every 3-regular graph with no bridges has a 1-factor.*

Theorem 85. *Every regular graph of even degree has a 2-factor.*

We recall that a k-factor is a k-regular factor. Given a function $f : V(G) \rightarrow \mathbb{N} \cup \{0\}$, we ask whether a graph G has a subgraph H such that $d_H(v) = f(v)$ for all $v \in V(G)$. Such a graph is called a **f-factor** .

Multiple edges do not affect the existence of 1-factors, but they do affect the existence of f-factors. We can assume that $f(\omega) \leq d(\omega)$ for all ω because if that is not so, then G trivially has no f-factor.

Let $e(\omega) = d(\omega) - f(\omega) \geq 0$ be the "excess" degree of vertex ω. To construct the graph H, replace each vertex v by a bipartite graph $K_{d(v),e(v)}$ with partite sets $A(v)$ of size $d(v)$ and $B(v)$ of size $e(v)$. For each edge $v\omega$ in G join one vertex of $A(v)$ to one vertex in $A(\omega)$ so that each A-vertex participates in one such edge.

Example 33.

In general:

Theorem 86. *A graph G has an f-factor if and only if the graph H has a 1-factor.*

Thus if we can test for 1-factors (perfect matchings) we can test for f-factors. We now look at finding maximum matchings.

8.7 Maximum Matching in General Graphs

We do this by looking for augmenting paths. In bipartite graphs, it is easy to look for them because starting with any unsaturated vertex v, we reach vertices in the same partite set as v via edges in the matching and vertices in the other partite set via edges not in the matching. So it is enough to consider each vertex once. In the general graph case, the loops of odd length will make it much harder:

flower
stem
blossom

The marked edges are a matching and we have two augmenting paths from a:

$$a, b, c, d$$

$$a, b, c, g, f, e, d, h.$$

We want the second one. Remember that we used to look for shortest augmenting paths. To solve this we define:

Definition 44. *Let $\mathcal{M}$ be a matching in a graph G and u be an unsaturated vertex. A **flower** is the union of two $\mathcal{M}$-alternating paths from u that reach a vertex X on paths of opposite parity in length (i.e. one odd, one even). The **stem** is the maximal length of the common initial sequence of non-negative even length. The **blossom** is the odd cycle obtained by deleting the stem from the flower.*

Hence, the flower is the whole graph except for the vertex h; the stem is $a - b - c$; and the blossom is $c - g - f - e - d - c$.

Since the blossom has odd length, we may not leave the blossom on an edge in the matching. The idea is to collapse the whole blossom onto the "super-vertex". Since we may also reach each vertex in a blossom by a saturated edge, when we collapse the blossom we are left with a saturated vertex.

So let's incorporate these ideas into our original algorithm. We have a graph G, a matching $\mathcal{M}$ and an unsaturated vertex u. We now explore $\mathcal{M}$-alternating paths leaving from u. e let S be the vertex set reached along edges in $\mathcal{M}$ and let T be the vertex set reached along edges not in $\mathcal{M}$. We mark the members of S that have been explored and for each vertex in $S \cup T$, we save the vertex from which it was reached. When we find a blossom, we contract the whole of it.

We call a vertex even/odd if there exists an even/odd length augmenting path form a given base vertex to this one. In a bipartite graph, no vertex is both even and odd but in a general graph, it may be (due to blossoms).

```
Data      : Initially we have a graph $G$ with vertex and edge sets
            $V(G), E(G)$. $P(v)$ is the parent of $v$.
$\overline{E}(G) \leftarrow \{(v,\omega),(\omega,v) \ : \ \{v,\omega\} \in E(G)\}$
for all unmatched vertices $v$ do
    $label(v) \leftarrow$ "even"
end for
for all unmatched vertices do
    $lavel(v) \leftarrow$ "unreached"
    $P(v) \leftarrow NULL$
    if $v$ is matched to edge $\{v,\omega\}$ then
        $mate(v) \leftarrow \omega$
    end if
end for
```

Algorithm 20: Blossom Algorithm

Now comes the main loop.

```
while there exists an unexplored edge $\{v, \omega\} \in \overline{E}$, where $v$ is even do
    if $\omega$ is odd then
        do nothing
    else
        if $\omega$ is "unreached" and matched then
            $label(\omega) \leftarrow$ odd
            $label(mate(\omega)) \leftarrow$ even
            $P(\omega) \leftarrow v$
            $P(mate(\omega)) \leftarrow \omega$
        else
            if $\omega$ is "even" and $v$ and $\omega$ are in distinct trees $T, T'$ then
                output augmenting path $P$ from root of $T$ to $v$,
                through $\{v, \omega\}$ in $T'$ to root
            else
                if $\omega$ is "even" and $v$ and $\omega$ are in the same tree then
                    //$\{v, \omega\}$ forms a blossom $B$ containing all vertices
                    that both (1) a descendant of the least common
                    ancestor of $v$ and $\omega$ in $T$, and (2) and ancestor of
                    $v$ or $\omega$.
                    Shrink all vertices of $B$ to a single vertex $b$
                    $P(b) \leftarrow P(LCA(v, \omega))$
                    for all $x \in B$ do
                        $P(x) = b$
                    end for
                end if
            end if
        end if
    end if
end while
```

Algorithm 21: Blossom Algorithm continued

Theorem 87. *This algorithm produces an augmenting path if there exists one.*

We then use the past method to get a longer matching from this path. The algorithm runs in $O(n^4)$, but it can be done in $O(n)$ stages and $O(m)$ for each run of finding the path, so $O(n\ m)$ in general.

k-coloring
color
color class
coloring!proper
k-colorable
chromatic number
k-chromatic
color-critical
k-critical

Chapter 9

Graph Coloring

9.1 Concepts

Definition 45. *A **k-coloring** of a graph G is a vertex labelling $f : V(G) \to \{1, 2, ..., k\}$. The labels are **colors** and the set of vertices with a certain color are a **color class** . A coloring is **proper** if $x \leftrightarrow y$ implies $f(x) \neq f(y)$.*

*A graph is **k-colorable** is a k-coloring exists. The **chromatic number** $\chi(G)$ is the minimum k, such that G is k-colorable. If $\chi(G) = k$, then G is **k-chromatic** . If $\chi(G) = k$ and $\chi(H) < k$ for every proper subgraph H of G, G is **color-critical** or **k-critical** .*

We observe a few things:

Loops and multi-edges make no difference to coloring and so we may ignore them henceforth.

If the coloring is proper, each color class is an independent set. Thus:

Theorem 88. *G is k-colorable iff G is k-partite.*

Theorem 89. $\chi(G) \geq \frac{n}{\alpha(G)}$ *where $\alpha(G)$ is the independence number.*

Theorem 90. $\chi(G) \geq \omega(G)$ *where $\omega(G)$ is the number of vertices of the largest complete subgraph of G.*

Theorem 91. $\chi(G) \leq n$.

All three previous theorems are best-possible bounds as they hold with equality if $G = K_n$.

The average values of $\omega(G), \alpha(G)$ and $\chi(G)$ over all labelled n-vertex graphs are very close to $2\ log\ n$, $2\ log\ n$ and $\frac{n}{2\ log\ n}$ respectively. Thus $\omega(G)$is generally a bad lower bound whereas $\frac{n}{\alpha(G)}$ is generally a good lower bound.

Can we get $\chi(G)$ by divide and conquer ?

Cartesian Product

Definition 46. *The **Cartesian Product** $G \Box H$ of two graphs G and H is the graph with vertex set $V(G) \times V(H)$ and, (u, v) adjacent to (u', v') if and only if (1) $u = u'$ and $vv' \in E(H)$ or (2) $v = v'$ and $uu' \in E(G)$.*

Theorem 92. $\chi(G \Box H) = max\{\chi(G), \chi(H)\}$.

Theorem 93. *G is m-colorable iff $G \Box K_m$ has an independent set of size $n(G)$.*

Thus chromatic number is equal to independence number as a problem and therefore an algorithm for one will give one for the other. We also note:

Theorem 94. $\chi(G + H) = max\{\chi(G), \chi(H)\}$. *(disjoin union)*

Theorem 95. $\chi(G \vee H) = \chi(G) + \chi(H)$. *(join)*

9.2 Greedy Coloring

We color a graph greedily with respect to a vertex ordering $v_1, v_2, \ldots, v_n$ by coloring v_i with the smallest index color not used in its lower-index (in the ordering) neighbors.

If $\Delta(G)$ is the maximum degree of a vertex, we see that any vertex has at most $\Delta(G)$ previous neighbors and that thus:

Theorem 96. $\chi(G) \leq \Delta(G) + 1$.

Furthermore, greedy coloring will achieve this inequality. In fact,

Theorem 97. *$\chi(G) \leq \Delta(G)$ if $G \neq K_n$ and $G \neq C_m$ with m odd.*

Surprisingly,

Theorem 98. *Every subgraph has a vertex ordering such that greedy coloring uses $\chi(G)$ colors. This ordering is usually hard to find.*

However, using a random ordering, usually gives a coloring using 2 $\chi(G)$ colors.

Vertices of high degree should be considered first, so

Theorem 99. *If a graph G has a vertex degree sequence $d_1 \leq d_2 \leq \cdots \leq d_n$, then $\chi(G) \leq 1 + max_i\ min\{d_i, i-1\}$ by applying greedy coloring to the vertices in non-decreasing order.*

Given an ordering σ of $V(G)$, we let G_i be the subgraph of G induced by $\{v_1, v_2, \ldots, v_i\}$ and $f(\sigma)$ be:

$$f(\sigma) = 1 + max_i\ d_{G_i}(x_i). \tag{9.1}$$

Theorem 100. *Greedy coloring applied to G using σ gives $\chi(G) \leq f(\sigma)$.*

We define σ^*: Let v_n be a vertex of minimum degree in G and for $i < n$ let v_i be a vertex of minimum degree in $G \setminus \{v_{i+1}, \ldots, v_n\}$.

Theorem 101. *σ^* minimizes $f(\sigma)$ and $f(\sigma^*) = 1 + \max_{H \subseteq G} \delta(H)$.*

We can also do it if we orient the graph:

Theorem 102. *If D is an orientation of G with largest path of length $l(D)$, then $\chi(G) \leq 1 + l(D)$, and equality for some orientation.*

9.3 Applications

Coloring a graph in which vertices represent activities and edges conflicts, are colored in order to produce good schedule for these activities. Edges may be labelled and/or weighted to represent conflicts of different kinds, and conflicts of different severity. We then ask for the minimal coloring or a coloring minimizing some function which contains the edge weights as well as the number of colors.

This is used by compiles to schedule variables to registers for example.

It is used in sports tournaments to schedule matches as well as universities to schedule classes and exams.

9.4 Counting Colorings

Definition 47. *The function $\chi(G;k)$ counts the mappings $f : V(G) \to [k]$ that properly color G. Not all the colors need to be used and permuting the colors should be counted twice.*

chromatic polynomial

For an independent set, we can color each vertex as we like. Thus any assignment is proper,

$$\chi(\overline{K_n}; k) = k^n. \tag{9.2}$$

For K_n, we color each vertex a different color but we make this choice from $k - i + 1$ available colors at stage i. Thus,

$$\chi(K_n; k) = k(k-1)(k-2)...(k-n+1). \tag{9.3}$$

If T is any tree, we color the root in k ways. At every stage only one color is forbidden so we color every other vertex by one of these. Hence,

$$\chi(T; k) = k(k-1)^{n-1}. \tag{9.4}$$

for n-vertices in T.

The value of $\chi(G; k)$ is a polynomial and is known as the **chromatic polynomial** .

Theorem 103. *If G is simple and $e \in E(G)$, then*

$$\chi(G; k) = \chi(G - e; k) - \chi(G \cdot e; k)$$

where $G \cdot e$ means, contract the edge e in G.

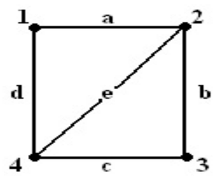
Proof. Every k-coloring of G is a k-coloring of $G - e$. Every proper k-coloring of $G - e$ is a proper k-coloring of G iff the endpoints of e get distinct colors. Thus $\chi(G; k) = \chi(G - e; k)$ - the number of proper colorings of G that give the same endpoints of e, the **same** color, i.e. $G \cdot e$.

Thus the formula follows. □

This is an example of a topological invariant. This may be used recursively to compute $\chi(G; k)$ as both $G - e$ and $G \cdot e$ have fewer edges than G. We already have the right initial condition or base case for the recursion, i.e.

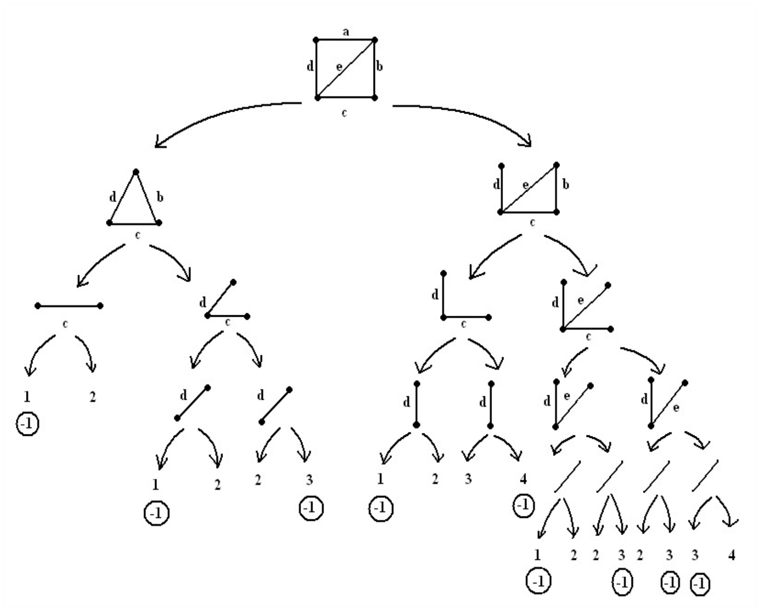
$$\chi(\overline{K_n}; k) = k^n. \tag{9.5}$$

Example 34.



G

We want to find $\chi(G; k)$. For this we draw the tree of operations and write the size of each independent set on the leaf nodes. Each left turn is a contraction and each right turn a substraction of an edge. If we go down and contract an odd number of times the factor k^α is subtracted, otherwise added.



$$\Rightarrow \chi(G;k) = k^4 - 5k^3 + 8k^2 - 4k$$

Theorem 104. *$\chi(G;k)$ is a polynomial in k of degree $n(G)$, with integer coefficients alternating in sign and beginning $1, -e(G), \ldots$.*

For graphs with many edges, we want to rewrite the formula like this:

$$\chi(G-e;k) = \chi(G;k) + \chi(G \cdot e;k) \tag{9.6}$$

Example 35. *Take the same as the previous example.*

$$\begin{aligned} \chi(G;k) &= \chi(K_4;k) + \chi(K_3;k) \\ &= k(k-1)(k-2)(k-3) + k(k-1)(k-2) \\ &= k(k-1)(k-2)^2 \\ &= k^4 - 5k^3 + 8k^2 - 4k \end{aligned}$$

Much faster !

Theorem 105. *If $c(G)$ is the number of components of G and $G(S)$ denotes the spanning subgraph of G with edge set S, then*

$$\chi(G;k) = \sum_{S \subseteq E(G)} (-1)^{|S|} k^{c(G(S))}.$$

While this contains too many terms to be useful in calculations, it is useful for providing theorems.

Theorem 106. *If $a(G)$ is the number of acyclic orientations of G, then*

$$\chi(G;-1) = (-1)^{n(G)} a(G).$$

9.5 Obtaining a Coloring

Testing for bipartite is easy and the graph then has $\chi = 2$. For more, we use greedy coloring on a non-increasing degree order of the vertices.

To improve a greedy coloring observe that exchanging two colors is also a proper coloring. So, having greedy colored G, delete all vertices except those of color i and j. If the remaining graph (i.e. the induced subgraph of G induced by the color classes i and j) has two or more components, we may repaint one or more components, while leaving a proper coloring. After such recoloring, a vertex v previously adjacent to both i **and** j could now be adjacent only to i and not to j leaving it to us to color j.

Simulated Annealing with this exchange process is likely to be the best solution:

```
Simulated Annealing:
Create initial solution $S$
Initialize temperature $t$
while no change in $C(S)$ do
    for $i = 1$ to iteration-length do
        Generate transition from $S$ to $S_i$
        if $C(S) \geq C(S_i)$ then
            $S \leftarrow S_i$
        else
            if $exp[(C(S) - C(S_i))/kt > random[0, 1]]$ then
                $S \leftarrow S_i$
            end if
        end if
    end for
    Reduce temperature $t$
end while
Return $S$
```

Chapter 10

Network Flow

network
capacity
source
sink
flow
feasible
capacity constraint
conservation constraint
value
maximum flow
minimum flow

10.1 Network Flow

Given a network of channels together with their capacities and special source and sink vertices we ask what is the maximum flow from the source to the sink ?

Definition 48. *A **network** is a directed edge weighted graph with two distinguished vertices s and t. The edge weights are called **capacities** and are non-negative. Vertices s and t are called **source** and **sink** .*

*A **flow** assigns a value to each edge, $f(e)$. We write $f^+(s)$ for the total flow on edge exiting the set of vertices s. The total flow exiting s is denoted by $f^-(s)$. A flow is **feasible** if $0 \leq f(e) \leq c(e)$ for all edges e and capacities $c(e)$, and if $f^+(v) = f^-(v)$ for each vertex $v \notin \{s, t\}$. These two conditions are called the **capacity constraint** and the **conservation constraint** .*

*The **value** of the flow $val(f)$ is*

$$val(f) = f^-(t) - f^+(t)$$

*,i.e. the net flow into the sink. A **maximum flow** is a flow minimizing the value. A **minimum flow** is a flow minimizing the value.*

The problem of finding the maximum flow is of great importance in practice as many problems may be reformulated in terms of it. The other reason is that efficient algorithms for finding maximum flows are known. First, we note:

Theorem 107. *A feasible flow always exists.*

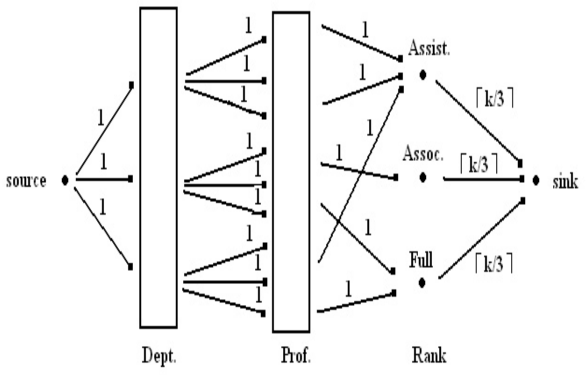
Proof. $f(e) = 0 \; \forall e$.
Thus a maximum flow and minimum flow also always exists.

Let's look at an example before we continue.

cut
capacity

Consider a university with k departments. The President wants to appoint a committee for some important task. Every department sends one Professor. Many Professors have joint appointments but each can only be the representative of one department. The President also wants equal representations among the three classes of Professor: Assistant, Associate, Full. How shall we find the committee ?

We construct a single node for each department, each Professor and each Professorial rank. The source has unit edges to every department which send unit edges to their ranks which send edges of weight $\lceil k/3 \rceil$ to the sink.



The edges from the source to the departments ensure that each department is represented by exactly one person. The edges leaving Professors (only one for each) ensure that no Professor represents more than one department. The three final edges of equal capacity ensure a balanced representation over seniority. Thus the committee exists if and only if we can find a flow of value k.

10.2 Cuts and Paths

Definition 49. *Given a network N, a* ***cut*** *is a partition of the vertex set V into two sets S and T such that $s \in S$ and $t \in T$. The* ***capacity*** *of the cut is*

$$c(S,T) = \sum_{e^- \in S;\ e^+ \in T} c(e)$$

where $e^- \equiv$ head of e; $e^+ \equiv$ tail of e.

Theorem 108. *If N is a network with (S,T) a cut of a feasible flow,*

$$\omega(f) = \sum_{e^- \in S;\ e^+ \in T} f(e) - \sum_{e^+ \in S;\ e^- \in T} f(e)$$

and thus, $\omega(f) \le c(S,T)$.

augmenting path

We use this to get us an algorithm eventually. But first we need to generalize the notion of augmented path.

Definition 50. *An undirected path from s to t in N is an* ***augmenting path*** *(with respect to the flow f) if $f(e) < c(e)$ and $f(e) > 0$ holds over any forward and backward edges respectively.*

Theorem 109. *A flow f on N is maximum if and only if there exists no augmenting path.*

Proof. Consider the case that f is maximum. Suppose an augmenting path exists., call it ω. Let d be the minimum of $c(e) - f(e)$ and $f(e)$ over all forward and backward edges of ω respectively, i.e.

$$d \leftarrow \min(\min_{forward\ edges\ e \in \omega}(c(e) - f(e)), \min_{backward\ edges\ e\in\omega}(f(e))) \tag{10.1}$$

by definition of an augmenting path, $d > 0$. We define

$$f'(e) = \begin{cases} f(e) + d : e \text{ forward edge in } \omega. \\ f(e) - d : e \text{ backward edge in } \omega. \\ f(e) : e \text{ not in } \omega. \end{cases}$$

f' is a feasible flow with $\omega(f') = \omega(f) + d > \omega(f)$ and thus f is not maximum. By this contradiction, we have proven the "only if" part.

Now consider the case when there is no augmenting path. Let S be the vertex set such that there exists an augmenting path from s to v (including s itself) and $T = V \setminus S$. Then (S,T) is a cut. Thus $f(e) = c(e)$ for all e with $e^- \in S,\ e^+ \in T$. Also $f(e) = 0$ for all e with $e^+ \in S,\ e^- \in T$. Thus by the previous theorem $\omega(f) = c(S,T)$ and thus f is maximum. □

Theorem 110. *A flow f is maximum if and only if the set $S \neq V$ and if this is so then, in addition, $\omega(f) = c(S,T)$ where $T = V \setminus S$.*

Theorem 111. *If all capacities are integers, so are all $f(e)$ for some maximum flow f.*

This is important in practice where we often want to deal only in integral wholes of things (i.e. the Professors from before).

Note that any computer representation necessarily uses at most rational numbers. Even these can be made integral by trivial "change of units". Furthermore multi-edges do not matter as we may simply replace them by a single edge of summed capacity.

The crucial min-max relation follows:

Theorem 112. *If f is maximum, its weight is equal to the minimum weight of a cut.*

Thus we do the following to find a maximal flow:

```
for all edges do
    $f(e) \leftarrow 0$
end for
while there exists an augmenting path $\omega = (e_1, e_2, ..., e_m)$ from $s$ to $t$ with respect to $f$ do
    $d \leftarrow \min(\min_{forward\ edges\ e \in \omega}(c(e) - f(e)), \min_{backward\ edges\ e \in \omega}(f(e)))$
    for all forward edges $e_i$ of $\omega$ do
        $f(e_i) \leftarrow f(e_i) + d$
    end for
    for all backward edges $e_i$ of $\omega$ do
        $f(e_i) \leftarrow f(e_i) - d$
    end for
end while
```

Algorithm 22: Maximal flow algorithm

This leaves the condition in the while loop to be assessed ...

We now give an algorithm due to Edmonds and Karp to find an augmenting path.

```
for $e \in E$ do
    $f(e) \leftarrow 0$
end for
label $s$ with $(-, \infty)$
for $v \in V$ do
    $u(v) \leftarrow$ false
    $d(v) \leftarrow \infty$
end for
while $u(v) =$ false for some labelled vertex $v$ do
    among all vertices with $u(v)$ =false, let $v$ be the one labelled first
    for $e \in \{e \in E \ : \ e^- = v\}$ do
        if ($\omega = e^+$ is not labelled) and ($f(e) < c(e)$) then
            $d(\omega) \leftarrow \min(c(e) - f(e), d(v))$
            label$(\omega) \leftarrow (v, +, d(\omega))$
        end if
    end for
    for $e \in \{e \in E \ : \ e^+ = v\}$ do
        if ($\omega = e^-$ is not labelled) and ($f(e) > 0$) then
            $d(\omega) \leftarrow \min(f(e), d(v))$
            label$(\omega) \leftarrow (v, -, d(\omega))$
        end if
    end for
    $u(v) \leftarrow$ true
    if $t$ is labelled then
        let $d$ be the last component of the label of $t$
        $\omega \leftarrow t$
        while $\omega \neq s$ do
            let $v$ be the first component of $\omega$'s label
            if second component of $\omega$'s label is $t$ then
                $f(e) \leftarrow f(e) + d$ for $e = v\omega$
            else
                $f(e) \leftarrow f(e) - d$ for $e = \omega v$
            end if
        end while
        delete all labels except the one of $s$
        for $v \in V$ do
            $d(v) \leftarrow \infty$
            $u(v) \leftarrow$ false
        end for
    end if
end while
Let $S$ be the set of all labelled vertices
$T \leftarrow V \setminus S$
```

Theorem 113. *This algorithm finds a maximum flow f and a minimum cut (S,T) such that*

$$\omega(f) = c(S,T).$$

Theorem 114. *This algorithm has complexity $O(|V|\ |E|^2)$.*

Example 36. *We use the algorithm of Edmonds and Karp to determine a maximal flow and a minimal cut in the network N given in Figure 10.1. The capacities are given in parenthesis; the numbers without parenthesis in the following figures always give the respective values of the flow. We also state the labels which are assigned at he respective stage of the algorithm; when examining the possible labelling coming from some vertex v on forward edges (steps (6) through (9)) and on backward edges (steps (10) through (13)), we consider the vertices ω in alphabetical order, so that the course of the algorithm is uniquely determined. The augmenting path used for the construction of the flow is drawn bold.*

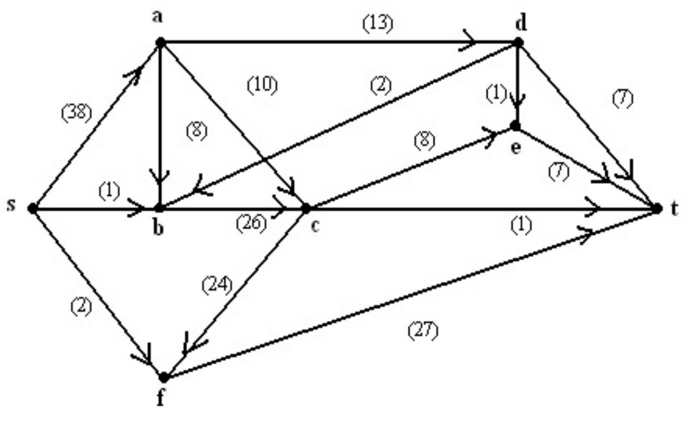


Figure 10.1: A flow network

We start with the zero flow f_0, that is, $\omega(f_0) = 0$. The vertices are labelled in the order a, b, f, c, d, t as shown in Figure 10.3; e is not labelled because t is reached before e is considered. Figures 10.4 to 10.12 show how the algorithm works. Note that the last augmenting path uses a backward edge, see Figure 10.11. In Figure 10.12, we have also indicated a minimal cut resulting from the algorithm.

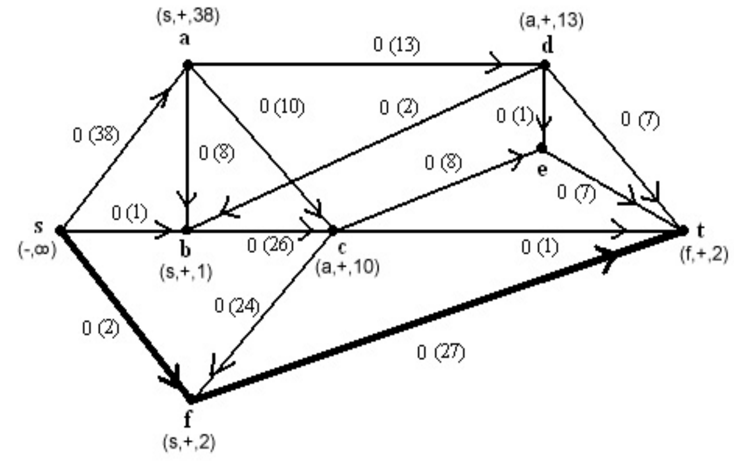


Figure 10.2: $\omega(f_0) = 0$

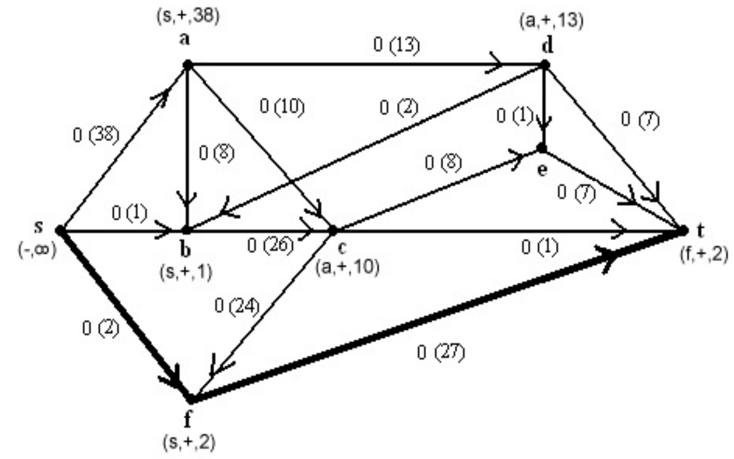


Figure 10.3: $\omega(f_0) = 0$

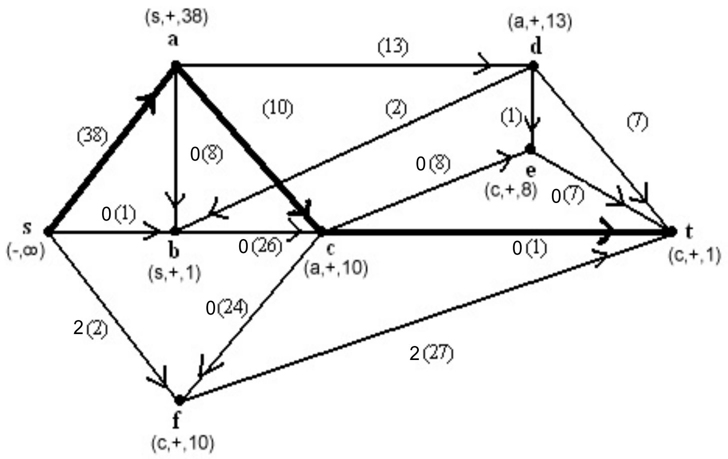


Figure 10.4: $\omega(f_1) = 2$

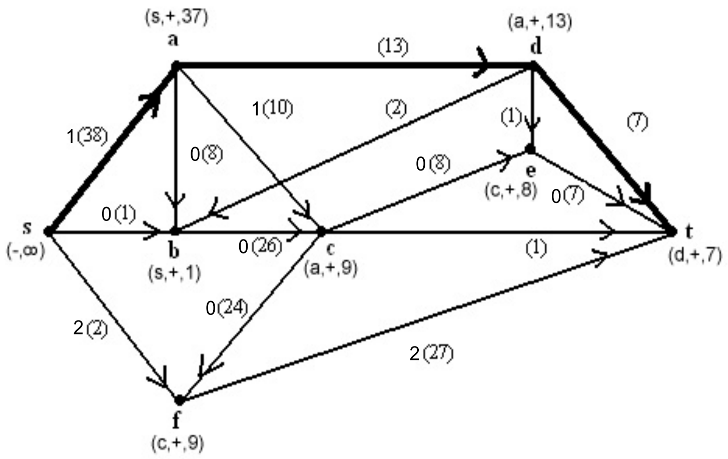


Figure 10.5: $\omega(f_2) = 3$

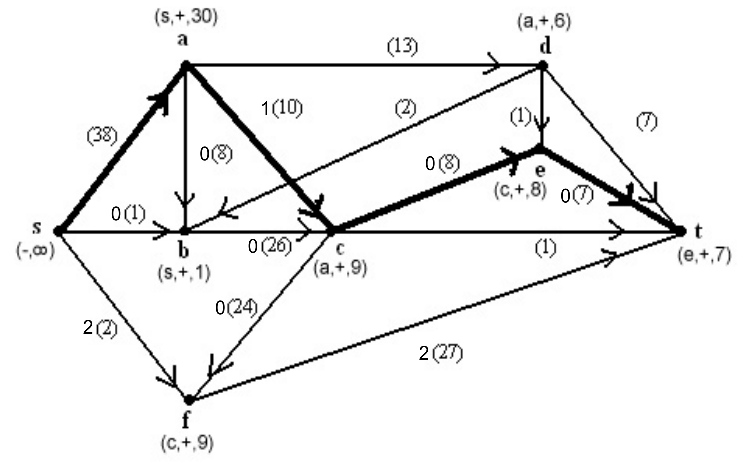


Figure 10.6: $\omega(f_3) = 10$

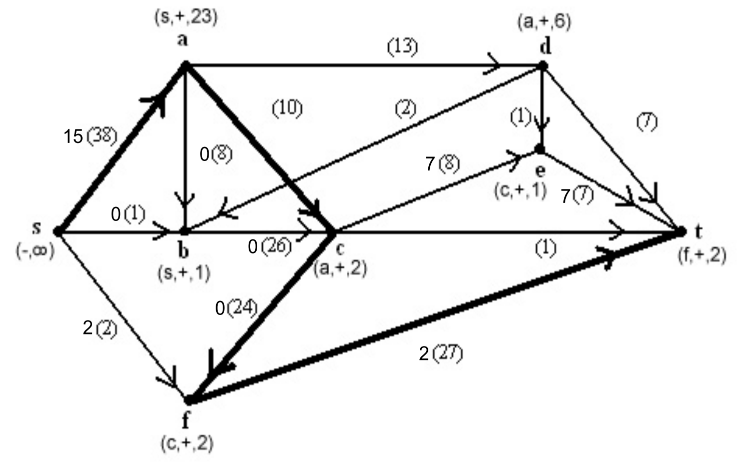


Figure 10.7: $\omega(f_4) = 17$

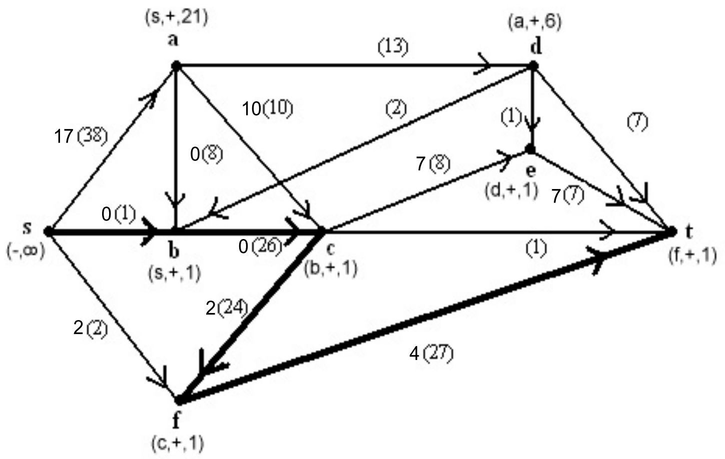


Figure 10.8: $\omega(f_5) = 19$

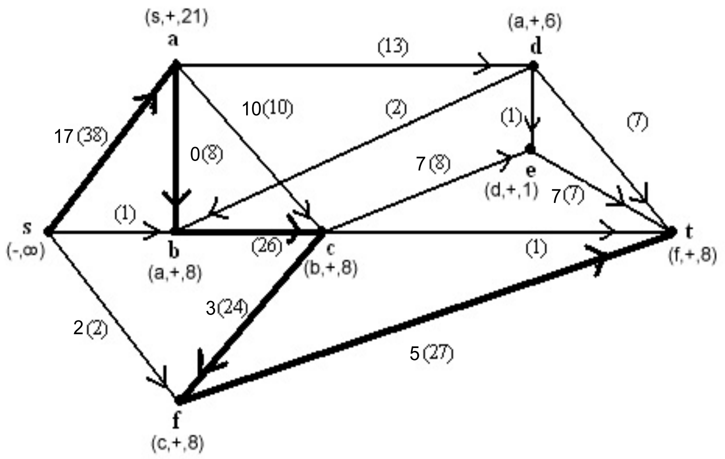


Figure 10.9: $\omega(f_6) = 20$

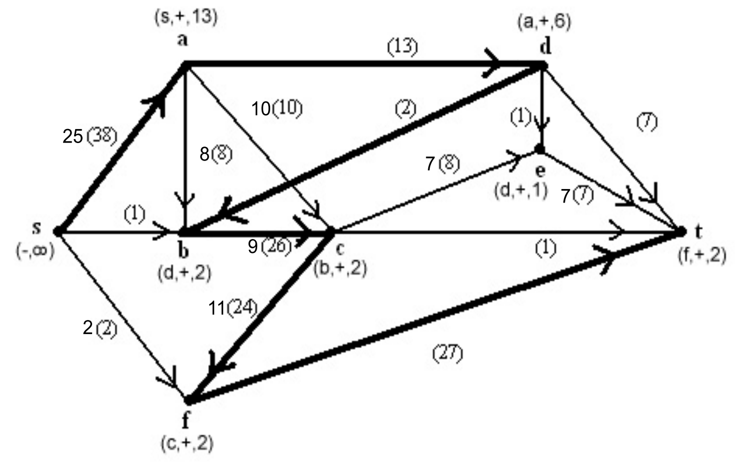


Figure 10.10: $\omega(f_7) = 28$

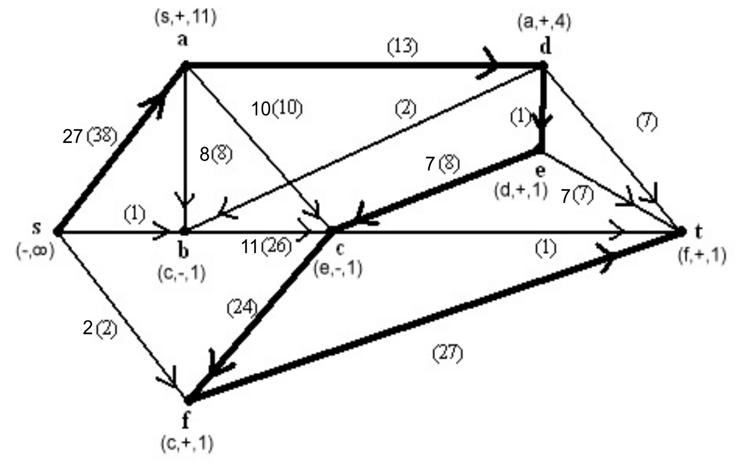


Figure 10.11: $\omega(f_8) = 30$

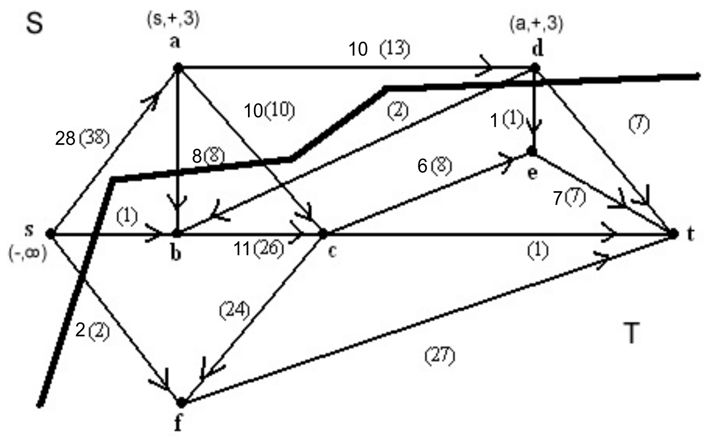


Figure 10.12: $\omega(f_9) = 31 = c(S, T)$

10.3 Applications

We have seen the matrix-rounding as an example of where we want to find a flow. We know one algorithm to get a flow in all possible cases. There are many other more efficient algorithms either for special cases or that just make use of more complex ideas. Let's just look at a few more uses of all this.

We have set a set of J jobs to be scheduled on M machines. Each job has a time length p_j for processing (in days), a release date r_j when this job can be started and a deadline $d_j \geq r_j + p_j$. Each machine can do only one job at a time and each job can be done on only one machine at a time. We do allow the jobs to be interrupted and transferred to another machine. Determine a feasible schedule.

Rank all release and due dates in ascending order and determine the $P \leq 2|J| - 1$ mutually disjoint intervals between these dates. Let $T_{k,l}$ represent the interval between date k and date $k + 1$.

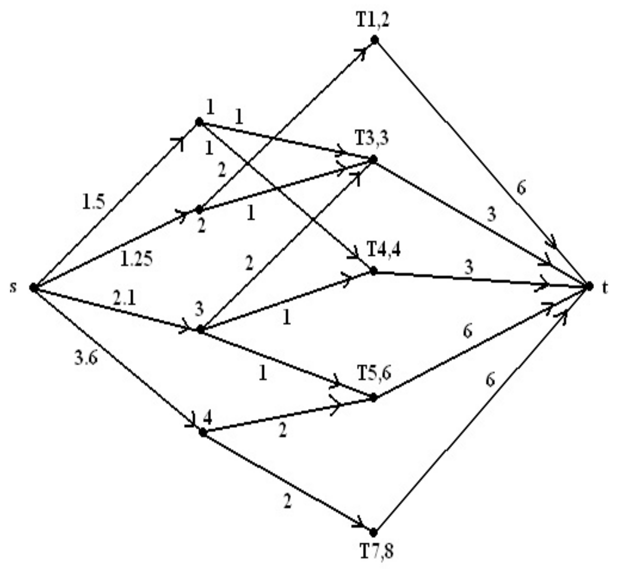
make a node for each job and interval. Connect the source to the jobs with capacity p_j. Connect the intervals to the sink with capacity $(l - k + 1)$ representing the total number of machine days available in between k and $l + 1$. We connect a job j to an interval $T_{k,l}$ if $r_j \leq k$ and $d_j \geq l + 1$ and the one has capacity $(l-k+1)$; i.e. if the job can be done in that interval, we weight it by the maximum number of machine days available in that interval.

The schedule exists if and only if the maximum flow value is $\sum_j p_j$. (total time length)

Example 37.

Job	1	2	3	4
Time	1.5	1.25	2.1	3.6
Release	3	1	3	5
Due	5	4	7	9

The list of dates is $1, 3, 4, 5, 7, 9$.
The intervals are $T_{1,2}, T_{3,3}, T_{4,4}, T_{5,6}, T_{7,8}$.
We have $\sum_j p_j = 8.45$, and the network is:



10.4 Multiple Sinks and Sources

Consider the case where we have a set $X = \{x_i\}$ of sources and $Y = \{y_i\}$ of sinks. We also associate a **supply** $\sigma(x_i)$ and **demand** $\delta(y_i)$ to each source and sink respectively. We also ass the **transportation constraints** :

supply
demand
transportation constraints

$$f^+(x_i) - f^-(x_i) \leq \sigma(x_i) \; \forall x_i \in X \tag{10.2}$$

net demand

$$f^-(y_i) - f^+(y_i) \geq \delta(y_i)\ \forall y_i \in Y \tag{10.3}$$

The zero flow is not necessarily feasible.

For a set A of sources ($A \subseteq X$) and B of sinks ($B \subseteq Y$)we write

$$\sigma(A) = \sum_{v \in A} \sigma(v)\ ;\ \delta(B) = \sum_{v \in B} \delta(v). \tag{10.4}$$

Fora set of edges F, we write:

$$c(F) = \sum_{e \in F} c(e). \tag{10.5}$$

Given any set of vertices T, the **net demand** $\delta(Y \cap T) - \sigma(X \cap T)$ must be satisfied by the flow from the other vertices. Thus $c([\overline{T}, T])$ has to be at least as large as this net demand. If this holds true for every T, then we have a feasible flow.

Theorem 115. *A feasible flow exists if and only if*

$$c([S,T]) \geq \delta(Y \cap T) - \sigma(X \cap T)$$

for every partition of the set of vertices of the network into sets S and T.

Proof. We know that this is necessary. For sufficiently, we construct a network N' from N by adding a super source s and a super sink t.

We add edges from s to x_i with capacity $\sigma(x_i)$ and from y_i to t with capacity $\delta(y_i)$. Then N has a feasible flow if and only if N' has a flow saturating each edge to t, i.e. if and only if N' has a flow of value $\delta(Y)$.

From before (min cut = max flow) we know that N' has this value if and only if

$$cap(S \cup s, T \cup t) \geq \delta(Y) \tag{10.6}$$

for each partition S, T of the vertices of N. The cut $[S \cup s, T \cup t]$ in N' consists of $[S,T]$ from N and edges from s to T and edges from S to t in N'.

$$cap(S \cup s, T \cup t) = c(S,T) + \sigma(T \cap X) + \delta(S \cap Y) \tag{10.7}$$

Thus

$$cap(S \cup s, T \cup t) \geq \delta(Y) \tag{10.8}$$

if and only if

$$c(S,T) + \sigma(X \cap T) \geq \delta(Y) - \delta(Y \cap S) = \delta(Y \cap T) \tag{10.9}$$

□

Thus we solve feasible flow in the general situation by the max flow from before. We can now solve a wide variety of flow problems.

Index