University of Portsmouth
BSc (Hons) Computer Science
Second Year

Discrete Mathematics and Functional Programming (DMAFP)
M21274
January 2024 - June 2024
20 Credits

Thomas Boxall
up2108121@myport.ac.uk

Contents

Part I

Discrete Maths

Page 1

Lecture - Sets

📅 2024-01-23   🕐 17:00   🎓 Janka

1.1 Introduction

Sets underpin maths and Computer Science. A set is a collection of objects, which are called the elements (also known as members of the set). For example, a set of the numbers 1, 3, 8; or the collection of students in a class born in March. There are two characteristics of sets:

1. There are no repeated occurrences of elements

2. There is no particular order of the elements

1.2 Set Notation

The elements of a set are enclosed in braces with their names being denoted by a *letter*, for example:

$$A = \{1, 2, 3\}, \quad C = \{Portsmouth, Brighton, London\}$$

There are two ways that we can describe the members of a set. We can *list the elements* which is mainly used for finite sets, for example:

$$A = \{3, 6, 9, 12\}$$

Alternatively, we can *specify a property* that all the elements in the set have in common. The '|' character is read 'such that', sometimes ':' is used in it's place. For example:

$$B = \{x | x \text{ is a multiple of 3 and } 0 < x < 15\}$$

We can also use *three dots* to informally denote a sequence of elements that we don't wish to write down, for example:

$$C = \{1, \ldots, 10\}$$

1.2.1 Sets of Numbers

There are some reserved letters to denote specific sets of numbers in maths. These are shown below:

- $\mathbb{N}$ (or N) is used for the set of natural numbers (integers $>= 0$). $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$

- $\mathbb{Z}$ (or Z) is used for the set of integers. $\mathbb{N} = \{\ldots, -1, 0, 1, \ldots\}$

- $\mathbb{Q}$ (or Q) is used for the set of rational numbers (number which can be expressed as a quotient or fraction). $\mathbb{Q} = \{0, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots\}$

- $\mathbb{R}$ (or R) is used for the set of real numbers. $\mathbb{R} = \{\ldots, -1, 0, \frac{1}{2}, \ldots\}$

1.2.2 Elements of a Set

We can use the $\in$ symbol to denote if an element is a member of a given set. For example, if x is a member of S - then we can say:
$$x \in S$$

The symbol $\notin$ denotes an element is not a member of a given set. For example, if y is **not** a member of S - then we can say:
$$y \notin S$$

1.2.3 Many Ways to Say The Same Thing

There are several ways of describing the same set, for example for the set S of *odd integers*:

$$\begin{aligned}
S &= \{\ldots, -5, -3, -1, 1, 3, 5, \ldots\} \\
&= \{x | x \text{ is an odd integer }\} \\
&= \{x | x = 2k + 1 \text{ for some integer } k\} \\
&= \{x | x = 2k + 1 \text{ for some } k \in \mathbb{Z}\} \\
&= \{2k + 1 | k \in \mathbb{Z}\}
\end{aligned}$$

The phrase "for some [integers K]", means "for all [integers k]"

1.2.4 Empty Sets

Where a set has *no elements*, it is called an empty set or null set. It's denoted with the $\emptyset$ symbol, for example:
$$\emptyset = \{\}$$

1.2.5 Finite & Infinite Sets

If the number of elements in the set is fixed (for example when counting the elements at a fixed rate for a set amount of time), then the set is *finite*. If the set X is finite, then we call $|X|$ the *cardinality* of X therefore:
$$|X| = \text{number of elements in } X$$

If the counting never stops then X is an infinite set.

1.2.6 Subsets

A subset is where one set's elements are entirely present in another set. There are three conditions we need to know about:

- $A \subseteq B$: A is a subset of B therefore every element in A is also in B.

- $A \nsubseteq B$: A is not a subset of B.

- $A \subset B$: A is a proper subset of B, therefore B has at least one additional element which is not in A.

1.2.7 Equality of Sets

Two sets are *equal* if they have exactly the same elements. This is denoted by writing $A = B$. Where $A = B$, the following conditions are also true:

- $A \subseteq B$ for every a if $a \in A$, then $a \in B$

- $B \subseteq A$ for every b if $b \in B$, then $b \in A$

1.3   Operations on Sets

Sets can have *operations* performed on them - this will change something about them.

1.3.1   Intersection

The intersection of two sets A and B is defined as:

$$A \cap B = \{x | x \in A \text{ and } x \in B\}$$

This is the set of elements which appear in both sets only. If we take a Venn Diagram with a set on either side - its the overlapped elements which would be returned from an intersection operation. For example if $A = \{a, b, c\}$ and $B = \{c, d\}$ then $A \cap B = \{c\}$.
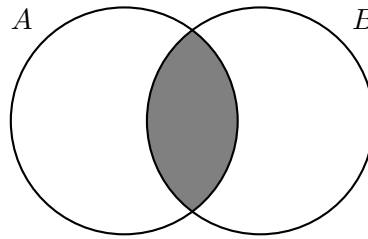

Figure 1.1: $A \cap B$

1.3.2   Disjoint

If an intersection returns no elements, then the two sets are *disjoint*. This is shown by:

$$A \cap B = \emptyset$$

1.3.3   Union

The *union* of the two sets A and B is defined as:

$$A \cup B = \{x | x \in A \text{ or } x \in B\}$$

This is the set of elements which are in either A or B, this means elements which appear in both are returned. For example, if $A = \{a, b, c\}$ and $B = \{c, d\}$ then $A \cup B = \{a, b, c, d\}$.
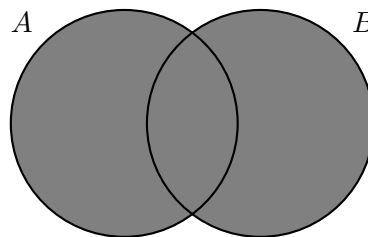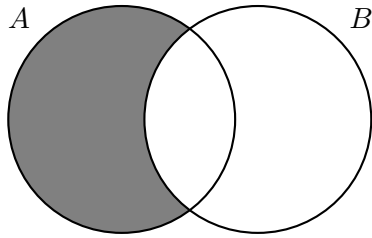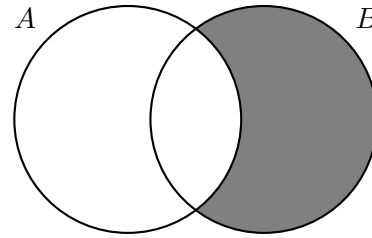

Figure 1.2: $A \cup B$

1.3.4   Difference

The *difference* between two sets, A and B is defined as:

$$A \setminus B = \{x | x \in A \text{ or } x \in B\}$$

This is the set of elements which are in A but not in B, so could be represented as $A - B$. Note that $A \setminus B \neq B \setminus A$.

Figure 1.3: $A \setminus B$

Figure 1.4: $B \setminus A$

1.3.5   Counting Elements In a Set

If we take A and B to be finite sets, we can calculate the number of elements in the union of A and B. The correct way to count this is as follows:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

We have to minus $|A \cap B|$ from the sum because otherwise it is as though we are counting it twice due to the fact that we are summing the total number of elements in A and B.

1.4   Complement

If we consider that all subsets are the subset of a particular set, U for example (the universe of discourse), then the difference $U \setminus A$ is called the *complement* of A is shown as either $\overline{A}$ or A'. For example:

$$A' = \{X | x \in U \text{ and } x \notin A\}$$

1.5   Basic Set Properties

Sets have a number of basic properties - many of these are the same as that for Boolean Expressions

- $A \cup \emptyset = A$

- $A \cap \emptyset = \emptyset$

- $A \cup A = A$

- $A \cap A = A$

- Commutative

    - $A \cup B = B \cup A$
    - $A \cap B = B \cap A$

- Associative

    - $(A \cup B) \cup C = A \cup (B \cup C)$
    - $(A \cap B) \cap C = A \cap (B \cap C)$

- Distributive

    - $A \cap (B \cap C) = (A \cap B) \cup (A \cap C)$
    - $A \cup (B \cup C) = (A \cup B) \cap (A \cup C)$

- de Morgan's

    - $(A \cap B)' = A' \cup B'$
    - $(A \cup B)' = A' \cap B'$

1.6 Power Set

A *power set* is the collection of all subsets of a set, S which is denoted by $P(S)$. For example, if $S = \{a, b, c\}$ then:

$$P(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$$

1.7 Partition

A *partition* of the set S is a collection of non-empty subsets of set S where every element form S belongs to exactly one member of S. This means that the sets are mutually disjoint and that the union of all the sets in the collection results in the original set, S. For example, if $S = \{a, b, c, d, e, f\}$ then $\{\{a, e\}, \{c\}, \{f, d\}, \{b\}\}$ is a partition of S.

Page 2

Lecture - Relations

📅 2024-01-30　　　　　🕐 17:00　　　　　🎓 Janka

2.1 Ordered Pairs

An ordered pair of elements is a group of two elements which are in a specific order. They are written as (a, b) and the order matters - this means (a, b) is distinct from the pair (b, a). Note that ordered pairs use the brackets () while sets use curly braces {}

2.2 Cartesian Product

The Cartesian Product of two sets is the set of **all** ordered pairs where the first element is taken from set 1 and the second element from set 2. The formal definition is as follows:

$$A \times B = \{(a, b) | a \in A \text{ and } b \in B\}$$

For example - if $X = \{1, 2, 3\}$ and $Y = \{a, b\}$, then

$$X \times Y = \{(1, a), (1, b), (2, a), (2, b), (3, a), (3, b)\}$$

2.3 Relations

A relation is the set of subsets from a cartesian product. For example, if we take $A = \{a, b, c, d, e\}$ and $B = 1, 2, 3$ then:

$$R_1 = \{(a, 1), (b, 1), (c, 2), (c, 3)\}$$
$$R_2 = \{(a, 3), (a, 1), (c, 2), (c, 1), (b, 2)\}$$

R_1 and R_2 are both examples of Binary relations from A to B.

2.3.1 Describing Relations

To describe a relation, we could list all of its elements, however this can be very long and obtuse so it's better & more common practice to use "the characteristics of their elements".

2.3.2 Relations On A Set

A relation *on a set* is where both sets are equal. For example $A = B$ then a relation on A is a relation from A to A, hence a subset of $A \times A$.

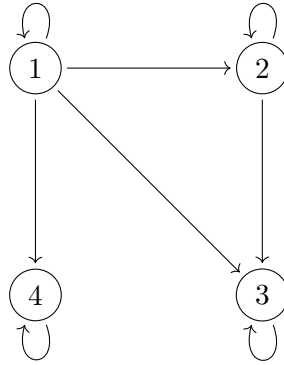For example, let R be the relation on $A = \{1, 2, 3, 4\}$ as defined by:

$$(x, y) \in R \text{ if and only if } x \text{ divides } y, \text{ for all } x, y \in A$$

Then we can conclude that R is:

$$R = \{(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 3), (4, 4)\}$$

2.4   A Digraph

We can use a *digraph* to picture a relation on a set. An example is shown below:

The dots (vertices) represents the elements of $A = \{1, 2, 3, 4\}$. If the element (x, y) is in the relation, an arrow (directed edge) is drawn from x to y.

$$R = \{(1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 3), (4, 4)\}$$

2.4.1   Reflexivity

A relation is reflexive where for all elements in the set - there is an ordered pair in which both elements are the same. For example the set $A = \{1, 2, 3, 4\}$ has the following relation:
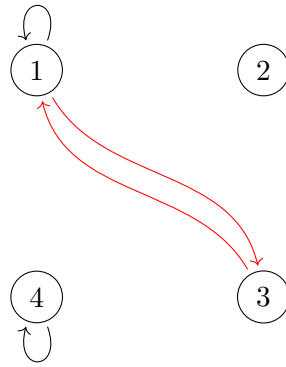
$$R = (1, 1), (1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 3), (4, 4)$$

Of which $(1, 1)$ and $(2, 2)$ are examples of reflexivity. This can be seen on the following digraph, the red arrows are the ones which display the reflexivity.
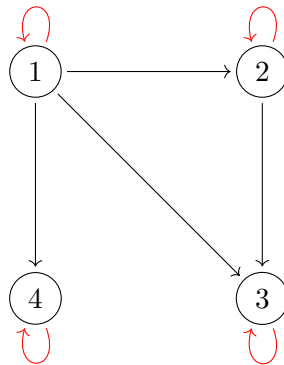
2.4.2   Symmetry

A relation is symmetrical where $(x, y) \in R$ and $(y, x) \in R$. If the condition is not true, then we do not have symmetry.

2.4.3   Transitivity

For a binary relation, R on set A; R is transitive if and only if for all $x, y, z \in A$ if $(x, y) \in R$ and $(y, z) \in R$ and $(x, z) \in R$. We initially assume that a relation is transitive and try to disprove it; if we are unable to disprove it then the relation is transitive. In the event that there is only one element in the relation - the relation will *always* be transitive.
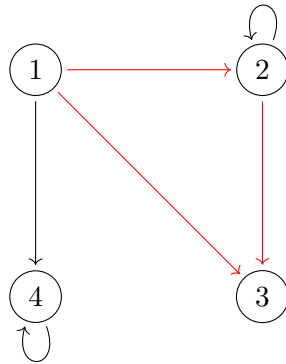
2.4.4   Equivalence

Where a relation is reflexive, symmetric and transitive - it is classed as an equivalence.

2.4.4.1   Equivalence Class

To be continued.

Page 3

Lecture - Functions

📅 2024-02-06   🕐 17:00   🎓 Janka

A *function* can be described in two ways. The mathematical definition is that "a function is a special type of relation in which a single input will have at most one output". The alternative definition of a function is that it is a mysterious black box which takes an input and returns and output. The same function, with the same input will always return the same output.

If we take A and B to be nonempty sets, then: A is a (total) function f from A to B, $f : A \to B$, is a relation from A to B such that

$$\text{for all } x \in A \text{ there is exactly on element in } B, f(x)$$

associated with x by relation f. Note that the word 'total' is used to describe the above function which means that every input has a defined output. The function $f : \mathbb{Z} \to \mathbb{Z}$ which is defined by $f(x) = 2x$ is also an example of a total function.

It is also possible to have a 'partial' function, this is where some of the inputs do not have defined outputs. For example the function $f(\frac{1}{x})$ where $x = 0$, would be undefined therefore the function is classed as 'partial'.

3.1 Describing A Function

There are a few different ways in which a function can be described.

3.1.1 By A Formula

This is the most common method used. The function f from $\mathbb{N} \to \mathbb{N}$ that maps every natural number x to its cube x^3 can be described as:
$$f(x) = x^3$$

3.1.2 By All Possible Associations

Whilst this is a valid method, it will generally not be used for efficiency reasons. The function g from $A = \{a, b, c\}$ to $B = \{1, 2, 3\}$ would be shown as:

$$g(a) = 1, g(b) = 1, g(c) = 2$$

3.2 Domain, Co-Domain & Range

The domain of a function is the set of all input values for which there is a defined output. For example if we let $f : A \to B$ then the subset $D \subset A$ of all elements for which f is defined is the domain. In the case of a total function, $D = A$ and in the case of a partial function, $D \subseteq A$

The co-domain of a function is the set of all possible output values; not just the ones which map to an input. For example, if we let $f : A \to B$ then the set B is the co-domain.

The range (also sometimes known as the image) of a function is the set of elements in the co-domain which map to an input. For example, if we let $f : A \to B$ then the range is denoted by $range(f)$. The range can also be expressed as:
$$range(f) = \{f(x) | x \in A\}$$

3.3 Properties of Functions

Functions have a number of properties.

3.3.1 Injective

The function $f : A \to B$ is injective (or one-to-one) if there is only one input that maps to each output. It can mathematically be defined as:
$$\text{for all } x, y \in A \text{ if } x \neq y \Rightarrow f(x) \neq f(y)$$

3.3.2 Surjective

A function $f : A \to B$ is surjective (or onto) if the $range(f)$ is the co-domain B. It can mathematically be defined as
$$\text{for all } y \in B \text{ there exists } x \in A \text{ such that } f(x) = y$$

A function which is not *onto* is *into*.

3.3.3 Bijective

A function $f : A \to B$ is bijective (or one-to-one correspondence) if it is both injective and surjective.

3.4 Composite Functions

A new function can be constructed by combining other simpler functions in some way. If we let $f : A \to B$ and $g : B \to C$ be functions. The composition of g with f is the function denoted by $g \circ f : A \to C$ and defined by:
$$(g \circ f)(x) = g(f(x)) \text{ for all } x \in A$$

$(g \circ f)(x) = g(f(x))$ is read as g of f, which means do f first then do g.

3.5 Inverse Functions

An inverse function is where the output of function f can be fed into the input of function f^{-1} to get the original input of f. This is mathematically defined as: $f : X \to Y$ is a bijective function, then there is an inverse function $f^{-1} : X \to Y$ that is defined as:
$$f^{-1}(y) = x \text{ if and only if } f(x) = y$$

Page 4

Lecture - Logic I: Introduction to Propositions & Logic

📅 2024-02-13 🕐 1700 🎓 Janka

4.1 Reasoning

Reasoning is something that we are introduced to doing from a young age. Younger children will continually ask "why?" as they attempt to make sense of the world while growing up; as they grow older they will generally only want the facts however. *Logic* is the discipline which deals with the method of reasoning:

- in mathematics to prove theorems

- in computer science to verify the correctness of programs and to prove some theorems

- in the natural and physical sciences to draw conclusions from experiments

- and in our everyday lives to solve a multitude of problems!

Over time, people come to understand that the following statement is how logic & reasoning works:

"If X then Y" is true and "X" is true $\Rightarrow$ so "Y" must be true

4.2 Propositions

A *proposition* it a *statement* (which is a declarative sentence) that can either be true or false; however not both. A proposition will be exact, not wishy-washy. For example - $3 - x = 5$ is not a proposition as it has an unknown value of x; however "The earth is flat" is a proposition as it can, and only can, categorically be True or False.

4.2.1 Propositional Variables

Propositions can be quite long. Mathematicians like efficiency, therefore they apply a single letter variable to a propositional statement to make their lives easier. The letters $p, q, r, \ldots$ are used to denote propositional variables.

Statement can be combined with logical connectives to obtain compound statements. For example *p and q*

4.3 Logical Connectives

Logical Connectives are used to combine propositional statements together; they are very similar to Boolean Algebra[1]. The truth value of a compound statement depends only on:

- the truth values of the statements being combined

- the types of connectives being used

4.3.1 Negation (not)

If p is a statement, the negation of p is the statement *not p*, denoted by $\neg p$. The truth table of negation is shown below:

p	$\neg p$
T	F
F	T

4.3.2 Conjunction (and)

If p and q are statements, the conjunction of p and q is the compound statement *p and q*, as denoted by $p \wedge q$

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

4.3.3 Disjunction (or)

If p and q are statements, the (inclusive) disjunction of p and q is the compound statement *p or q* as $p \vee q$

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

4.3.4 Conditional Proposition (implication)

If p and q are statements then the compound statement "*if p then q*" dented $p \rightarrow q$ (or $p \Rightarrow q$) is called implication. The hypotheses in the statement is p and the conclusion is denoted by q.

---

[1]Covered in A-Level Electronics, A-Level Computer Science, 1st Year ArchOS Module

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

4.3.5 Conditional Proposition (bidirectional)

If p and q are statement, the compound statement "if and only if" (abbreviated to *iff*), denoted by $p \Leftrightarrow q$, is called the biconditional of p and q.

p	q	$p \Leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

4.4 Truth of Compound Properties

It is possible to combine the truth tables for single logical connectives to make *more complicated truth tables*. In similar fashion to Algebraic & standard numeric expression evaluation - there is a hierarchy of evaluation, as seen below (listed highest to lowest):

1. brackets

2. negation ($\neg$)

3. conjunction ($\wedge$)

4. disjunction ($\vee$)

5. implication ($\rightarrow$)

6. bidirectional ($\leftrightarrow$)

For connectives of equal priority - work from left-to-right through them.

4.5 Special Conditions

4.5.1 Tautology

A statement that is true for all possible values of its propositional variables is called a tautology. For example $p : p \vee \neg p$ is a tautology. In a truth table, the final column (where the output is) will always be false.

4.5.2 Contradiction

A statement that is false for all possible values of its propositional variables is called a contradiction. For example, any proposition $p : p \wedge \neg p$ is a contradiction. In a truth table, the final column (where the output is) will always be false.

4.5.3 Contingency

A statement that can be either true or false depending on the truth values of its propositional variables is called a contingency. For example the proposition "Murray will win the Wimbledon next year" us a contingency because the truth of the statement is dependent on the propositional variable.

4.5.4 Contrapositive

The contrapositive of a conditional statement $p \to q$ is $\neg q \to \neg p$. The conditional statement is logically equivalent to its contrapositive.

4.6 Logical Equivalence

Two statements are said to be *logically equivalent*, $\equiv$, if and only if they have identical truth values for each possible value of their statement variables. Logical equivalence corresponds to $=$ with numbers.

Shown below are the rules of Logical Equivalence. There is not a requirement to memorise these as it should be possible to derive them in the exam when required. Note that they are the same as Boolean algebra's laws, except with funky symbols.

- $p \wedge p \equiv p$

- $p \vee p \equiv p$

- $p \wedge T \equiv p$

- $p \wedge F \equiv F$

- $p \vee T \equiv T$

- $p \vee F \equiv p$

- $\neg(\neg p) \equiv p$

- $p \vee (\neg p) \equiv T$

- $p \wedge (\neg p) \equiv F$

- $p \wedge q \equiv q \wedge p$ (commutativity)

- $p \vee q \equiv q \vee p$ (commutativity)

- $(p \vee (q \wedge r)) \equiv ((p \vee q) \wedge (p \vee r))$ (distributivity)

- $(p \wedge (q \vee r)) \equiv ((p \wedge q) \vee (p \wedge r))$ (distributivity)

- $p \to q \equiv (\neg p \vee q)$

- $\neg(p \vee q) \equiv ((\neg p) \wedge (\neg q))$ (De Morgan's Law)

- $\neg(p \wedge q) \equiv ((\neg p) \vee (\neg q))$ (De Morgan's Law)

4.6.1 Example of logical equivalence proof

Question. Prove that $(p \to q) \vee (p \to r) \equiv p \to (p \vee r)$

$$(p \to q) \vee (p \to r) \equiv$$

$\equiv (\neg p \vee q) \vee (\neg p \vee r)$	logical equivalence law
$\equiv \neg p \vee q \vee \neg p \vee r$	all are `or' so remove brackets
$\equiv \neg p \vee q \vee r$	get rid of second$\neg p$
$\equiv \neg p \vee (q \vee r)$	add brackets in
$\equiv p \to (q \vee r)$	logical equivalence law again

4.7 Necessary and Sufficient Condition

A necessary condition is a condition such that statement B cannot be true without statement A being true. However it is possible for statement A to be true to even if statement B is not true.

A sufficient condition is a condition such that knowing statement A is true guarantees that statement B is true.

A statement (A) is said to be "necessary and sufficient" for the statement B when B is true if and only if A is also true.

Page 5

Lecture - Logic II: Quantified Statements

📅 2024-02-20   🕐 1700   🎓 Janka

5.1   Propositional Logic: A Recap

As we saw last week, *Propositional Logic* applies to a declarative statement where the basic propositions are either *True* or *False*. "Mr Bean is a Mathematical Major" is an example of a proposition, however "*He* is a Mathematical Major is not" because it depends on the value of "he".

The sentence "All students sitting in this class in the first three rows are mathematical majors" ia a more complex example however. To fully discover whether this is a proposition or not, we would need to analyse each individual atomic propositions; which would be to analyse all individuals sitting in the first three rows. For example:

- "Bella is a mathematical major"

- "Joe is a mathematical major"

- "Fred is a mathematical major"

This is obviously a lot of work and mathematicians don't like to over-exert themselves, so there's a method which can be use to simplify this predicament, called *Predicate Logic*.

5.2   Predicate Logic

A *predicate* (or propositional function) is a statement containing one or more variables. If values from a given set (domain) are assigned to all the variables, the resulting statement is a proposition. For example:

- $P(n) : n^2 + 2n$ is an odd integer (domain $\mathbb{Z}$) is a predicate because when n is substituted in for an integer, we have a proposition.

- $S(n) :$ The student passed the exam (domain of all students sitting in this class) is a predicate because when n is substituted for a student - we have a proposition.

- $Test(x, y, z) : x < y + z$ (domain of all integers) is a predicate because once x, y and z have been substituted for an integer - we have a proposition as the mathematical expression can either be True or False.

- $Distance(x, y) :$ whether the distance between the towns x and y is less than 300km (domain of all towns in the UK) is a predicate because once we have substituted x and y for two different Towns then we will have a proposition (as this can either be True or False).

From programming, we are familiar with If and While statements. The condition which powers these are in fact predicates (if $p(y)$ then; or while $p(y)$ do).

5.2.1 Changing a Predicate to a Proposition

1. Assign values (from the domain) to all their variables
   For example: "x is divisible by 5" (the domain $\mathbb{N}$) would be converted through substituting x for a number (i.e 35) then this would result in either True or False (i.e True).

2. to add *quantifiers*

5.3 Quantifiers

Quantifiers are words that refer to quantities such as "some" or "all". Most of the statements in Maths and Computer Science use terms such as "for every" or "for some".

For example: "For all $x \in \mathbb{N}, x$ is divisible by 5" can use a different quantifier and be be written as "There exists $x \in \mathbb{N}$ such that x is divisible by 5". From this - we know that there are two quantifiers:

5.3.1 Universal Quantifiers

The symbol $\forall$ (an upside down A) is called the universal quantifier; which has the meaning "for all" or "for each".

The formal definition of the universal quantifier is as follows: For a predicate $p(x)$ with domain D, the statement:

$$\text{"for every } x \text{ from domain } D, p(x)\text{"}$$

may be written as $\forall x \in D \; p(x)$.

For example:

- Example 1: "All DMAFP students are happy" can be re-written as:
  Let D be the set of all DMAFP students, then

  $$\forall x \in D, x \text{ is happy}$$

- Example 2: Let $S = \{1, 2, 3, 4, 5, 6\}$ and consider the statement

  $$\forall x \in S, x^2 \geq x$$

- Example 3: $\forall x \in \mathbb{R}, x^2 \geq x$

5.3.1.1 True Statements with $\forall$

We know the fact: "The statement $\forall x \in D \; p(x)$ is true if $p(x)$ is true for every $x \in D$". To prove a quantified statement including $\forall$ to be true, we have to show the truth of the each individual element of the domain to be true.

- Example 1: "Let D be the set of all DMAFP students, then $\forall x$ in D, x is happy", is a statement. It is true if and only if every student answers 'Yes'.

- Example 2: "Let $S = \{1, 2, 3, 4, 5, 6\}$". The statement $\forall x \in S, x^2 \geq x$ is true because $1^2 \geq 1$, $2^2 \geq 2$, ..., $6^2 \geq 6$

5.3.1.2 False Statements with $\forall$

We know the fact: "The statement $\forall x \in D \ p(x)$ is false if $p(x)$ is false for at least one $x \in D$". To prove that a quantified statement including $\forall$ to be false, we have to show at least one counterexample.

- Example 1: "Let D be the set of all DMAFP students, then $\forall x$ in D, x is happy", is a statement. It is false if and only if there is at least one unhappy student.

- Example 3: The statement "$\forall x \in \mathbb{R}$, $x^2 \geq x$" is not true. This is because there exists $x \in \mathbb{R}$ for which $x^2 < x$ for example where $x = \frac{1}{2}$. Therefore the statement is false.

5.3.2 Existential Quantifier

The symbol $\exists$ (a backwards E) is called the existential quantifier; which has the meaning "there exists".

The formal definition of an existential quantifier is as follows: For a predicate $p(x)$ with the domain D:

$$\text{"there exists an } x \text{ from the domain } D \text{ such that } p(x)\text{"}$$

may be written as $\exists x \in D$ such that $p(x)$.

For example:

- Example 1: "There is a happy DMAFP student". can be rewritten:
  Let D be the set of all DMAFP students then:

$$\exists x \in D \text{ such that } x \text{ is happy}$$

- Example 2: Let $S = \{1, 2, 3, 4, 5, 6\}$ and consider the statement

$$\exists x \in S \text{ such that } x^2 \geq x$$

- Example 3: $\exists x \in \mathbb{N}$ such that $x^2 > x$

5.3.2.1 True Statements with $\exists$

We know the fact: "The statement $\exists x \in D$ such that $p(x)$ is true if $p(x)$ is true for at least one $x \in D$" To prove a quantified statement including $\exists$ to be true, we have to find one proposition for which the predicate is true.

- Example 1: "Let D be the set of all DMAFP students, the statement $\exists x \in D$ such that x is happy." To prove this is true - we only need to find one student who is happy.

- Example 3: The statement $\exists x \in \mathbb{R}$ such that $x^2 \geq x$ can be proved as true because it is true where $x = 2$.

5.3.2.2 False Statements with $\exists$

We know the fact that "The statement $\exists x \in D$" such that $p(x)$ is false if $p(x)$ is false for all $x \in D$. To prove a quantified statement including $\exists$ to be false, we have to prove that it can never be true. For a small, finite domain - this can be brute forced; or for a large (or infinite) domain - this has to be proved logically.

- Example n: The statement $\exists x \in \mathbb{Z}$ such that $x^2 < -2017$ is false because there are members of $\mathbb{Z}$ which when squared, are bigger than -2017

5.4   Negation of Quantified Statements

When negating a quantified statement, it is not as simple as one might imagine. You do not simply add (or remove) the word 'not' and hope for the best; rather you negate components and from here a new statement will be birthed.

For example, taking the statement "There exists a fluffy cat", to negate this - we would need to write "all cats are not fluffy". Note how this has gone from an 'exists' quantifier to a 'for all' quantifier. Alternatively, we can take the statement "All cats are fluffy", which negated would be "There exists a cat that is not fluffy". Again, note that the quantifier has been changed.

Now for some mathematical examples rather than (un)fluffy cats:

The negation of a statement of the form $\forall x \in D$, $Q(x)$ is logically equivalent to a statement of the form $\exists x \in D$ such that $\neg Q(x)$. Symbolically:

$$\neg(\forall x \in D, Q(x)) \equiv \exists x \in D \text{ such that } \neg Q(x)$$

The negation of a statement of the form $\exists x \in D$ such that $Q(x)$ is logically equivalent to a statement of the form $\forall x \in D, \neg Q(x)$ Symbolically:

$$\neg(\exists x \in D \text{ such that } Q(x)) \equiv \forall x \in D, \neg Q(x)$$

5.5   Nested Quantifiers

Multiple quantifiers such as $\forall x \exists y$, $\exists x \forall y$, ... are said to be nested quantifiers.

In actual English - you would see this such as something along the lines of "There is a student solving every exercise of the tutorials". However! This sentence is ambiguous as it would have at least two different meanings:

- There is one student who solves all the exercises of the tutorials

- For any particular exercise, there is a student who solves that exercise

5.5.1   Example

Let $P(x, y)$ be the property "the student x solves the exercise y, S - the set of students, E - the set of all exercises of the tutorials."

- $\exists x \in S \ \forall y \in E$ such that x and y satisfy property $P(x, y)$ is a statement.
  It is true if there is (at least) one student who solves all the exercises of the tutorials.

- $\forall y \in E \ \exists x \in S$ such that x and y satisfy property $P(x, y)$ is a statement.
  It is true if each exercise is solved by at least one student

Page 6

Lecture - Methods of Proof

📅 2024-02-27   🕐 1700   🎓 Janka

6.1   What is a Proof?

A Mathematical Proof is a deductive argument for a mathematical statement, showing that the stated assumptions logically guarantee the conclusion. Mathematical Proofs are carefully reasoned and there are a number of different ways we can deduce one.

6.1.1   Formal Example

Theorem 6.1.1. *Prove that for all integers m and n, if m is odd and n is even, then $m + n$ is odd.*

An *argument* (theorem) is a finite collection of statements $(p_1, p_2, \ldots, p_n)$ called *premises* (hypothesis) followed by a statement q called the *conclusion*.

$$(p_1 \wedge p_2 \wedge \ldots \wedge p_n) \Rightarrow q$$

In the above, the premises are "m, n, integers, m is odd, n is even" and the conclusion is "$m + n$ is odd". The argument is valid (or the theorem holds) if, whenever p_1, p_2, ..., p_n are all true, then q is also true.

6.1.2   Methods of Proof

To "prove a theorem" means to show that if all premises are true then the conclusion is also true.

m, n are integers

m is odd, n is even

. . .

Can we prove that $m + n$ is odd?

We have a number of different techniques which we can use. Which of the following to choose depends on the problem and experience:

1. A direct proof

2. A proof by contradiction

3. A proof by contrapositive

4. A proof by mathematical induction

6.2   A Direct Proof

In a direct proof, we start with the hypothesis of a statement (premises) and make one deduction after another until we reach the conclusion. Theorems which are able to be proved are often of the form:

$$p \ (hypothesis) \Rightarrow q \ (conclusion)$$

While proving the proof, we can use:

- Previously Proven Facts

- Definitions

- Known basic properties

Our task is to prove that then q is also true.

6.2.1   Our First Theorem

It's time to prove our first theorem, how exciting!

Theorem 6.2.1. *For all integers m and n, if m is odd and n is even, then $m + n$ is odd.*

As we are proving this using a direct proof, we assume that the hypothesis is true and derive the conclusion. We start doing this by realising some properties of odd and even numbers.

Proof. An integer r is even if and only if there exists an integer k such that $r = 2k$. (This is the basic definition of an even number, no room for disagreement here!)

Similarly, an integer r is odd if and only if there exists an integer k such that $r = 2k + 1$

To apply these derivations to our theorem:

m is odd $\Rightarrow$ there exists an integer k such that $m = 2k + 1$

n is even $\Rightarrow$ there exists an integer l such that $n = 2l$

We can use maths to show that the sum is:

$$m + n = (2k + l) + 2l$$
$$= 2(k + l) + 1$$

$\therefore m + n$ is odd.   ∎

Note the black square (∎) on the right above at the end of the proof. This is stands for QED which stands for "Quod Erat Demonstrandum", which stands for "what was to be demonstrated" which stands for "boom done, there I proved it, I'm leaving now".

6.3   Proof by Contradiction (indirect proof)

As we know there are only, and can only, be two options for the truth value of a conclusion: *True* or *False*. If supporting that the premises are true and the conclusion is false, we are able to arrive at a contradiction (a conclusion that is contradictory to our assumptions or something obviously untrue like $1 = 0$) $\Rightarrow$ our conclusions must be true!

While proving the proof, we can use:

- Previously known facts

- Definitions

- Known basic properties

Our task is to prove a contradiction, which is done through proving that q is true (assuming that p is true).

6.3.1 Our Second Theorem

Theorem 6.3.1. *For every $n \in \mathbb{N}$, if n^2 is even, then n is even*

We can rewrite this as:

$$n^2 \text{ is even}, n \in \mathbb{N} \Rightarrow n \text{ is even}$$

We can then take our hypothesis to be to be that n^2 is even while n is a natural number; and a false conclusion to be n is not even, hence n is odd. We want to prove any contradiction (which could be $r \wedge \neg r$ for a proposition r).

Proof. Since we take that n is odd, there exists $k \in \mathbb{N}$ such that $n = 2k + 1$. We can now derive the following:

$$\begin{aligned} n^2 &= (2k+1)^2 \\ &= 4k^2 + 4k + 1 \\ &= 2(2k^2 + 2k) + 1 \end{aligned}$$

This means that n^2 is odd.

Now n^2 is even (the hypotheses) and n^2 is odd (which we have just proved). This is a *crazy* situation, an impossible contradiction! As we have found a contradiction $\Rightarrow$ the conclusion must be true!

$\therefore$ for every $n \in \mathbb{N}$ if n^2 is even, then n is even. ∎

6.4 Proof by Contrapositive (indirect proof, again)

The contrapositive of the condition proposition $p \rightarrow q$ is the proposition $\neg q \rightarrow \neg p$. Note that the conditional proposition and its contrapositive are logically equivalent

$$p \rightarrow q \equiv \neg q \rightarrow \neg p$$

To prove a statement by contrapositive, we prove that the contrapositive statement is a direct proof and conclude that the original statement is true. This means that instead of the original theorem, $p \rightarrow q$, we prove by a direct proof the contrapositive theorem $\neg q \rightarrow \neg p$.

While proving the proof, we can use

- previously proven facts

- definitions

- known basic properties

Our task is to prove that $\neg p$ is true. In this way we prove that $\neg q \rightarrow \neg p$ and because of $p \rightarrow q \equiv \neg q \rightarrow \neg p$ necessarily $p \rightarrow q$ must be true as well (meaning that the theorem $p \rightarrow q$ is valid).

6.4.1 Our Second Theorem (again)

Theorem 6.4.1. *For every $n \in \mathbb{N}$, if n^2 is even then n is even.*

Our contrapositive statement is: "For every $n \in \mathbb{N}$ if n is not even, then n^2 is not even". We can use the fact "An integer is not even, if and only if, it is odd" to derive our contrapositive:

Contrapositive. For every $n \in \mathbb{N}$, if n is odd then n^2 is odd.

We are now able to prove the contrapositive statement using a direct proof by proving that n^2 is odd.

Proof. Since n is odd, there exists $k \in \mathbb{N}$ such that $n = 2k + 1$ We can now derive the following:

$$
\begin{aligned}
n^2 &= (2k + 1)^2 \\
&= 4k^2 + 4k + 1 \\
&= 2(2k^2 + 2k) + 1
\end{aligned}
$$

Therefore n^2 is odd.

This means that the contrapositive statement is true and by logical equivalence also the theorem:

"For every $n \in \mathbb{N}$, if n^2 is even, then n is even"

is valid ∎

6.5 Mathematical Induction

Mathematical Induction is one of the most basic methods of proof. It is a useful technique to use to establish the truth of a statement about natural natural numbers. It is a method for proving a statement (given in the form of a proposition) $P(n)$ is true for every natural number, n, and that the infinitely many cases $P(0), P(1), P(2), P(3), \ldots$ all hold. This is done in two stages.

1. Proving a simple case (the base case), which proves the statement for $n = x$ (where x is any number) without assuming any knowledges of other cases.

2. Having proven that when $n = x$, $P(n)$ is true; we assume that $P(n)$ is true for all $n \geq x$. (the inductive step) (this is *just* the case, which works through magic, don't look to deep into it.)

6.5.1 The Third Theorem

Theorem 6.5.1. *Prove that for any integer $n \geq 1$, the sum of the first n natural numbers is $S(n) = \dfrac{n(n + 1)}{2}$*

Proof. We will start by looking at the basic step, where we will take $n = 1$.

$$
\begin{aligned}
S(1) &= 1 \\
&= \frac{1(1 + 1)}{2} \\
&= 1
\end{aligned}
$$

We will now complete the inductive step.

(a) We can assume that $S(n)$ of the first n natural numbers is $\dfrac{n(n + 1)}{2}$

(b) We need to prove that the statement is true also for the first $n+1$ natural numbers, this means:

$$S(n+1) = \frac{(n+1)(n+2)}{2}$$

Which leaves us with the following assumption:

$$S(n) = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

Now we can derive the following:

$$
\begin{aligned}
S(n+1) &= 1 + 2 + \ldots + n + (n+1) \\
&= S(n) + (n+1) \\
&= \frac{n(n+1)}{2} + (n+1) \\
&= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} \\
&= \frac{(n+1)(n+2)}{2}
\end{aligned}
$$

Therefore, assuming that the formula is true for n, we have proved the formula is true for $n+1$. ∎

6.6 Disproving a Universally Quantified Statement

As we saw in the previous lecture, a universally quantified statement is one which applies "for all" or "for some".

To disprove
$$\forall x \in D \ P(x)$$
we have to find one $x \in D$ that makes $P(x)$ false. This value of x is called the 'counterexample'.

For example, the statement:
$$\forall n \in \mathbb{N}(2^n + 1 \text{ is prime})$$
is false because, where $n = 3$, $2^3 + 1 = 9$ which is not prime.

Page 7

Lecture - Graphs: An Introduction

📅 2024-03-12   ⏱ 1700   🎓 Janka

7.1 History of Graphs

Euler introduced the concept of *graph theory* in 1736 when he abstracted the Bridges of Königsberg problem into a series of *vertices* and *edges*. Graphs were further developed with the Travelling Salesman Problem whereby adding *weighting* values to an edge allows the most efficient route to be calculated.

Graphs can be a useful tool in solving a number of different mathematical problems, including those which are abstractions of a real world thing - for example the cities in which the travelling salesman needs to travel through in their problem. There are four main applications of graph theory today:

Existence Problems are problems in which we want to prove the existence (or lack thereof) of something. For example the Königsberg bridge problem.

Optimisatoin Problems are problems in which we are looking to find the best (most efficient) solution (this can either be a maximisation or minimisation). For example - the Travelling Salesman Problem.

Construction Problems are problems in which we are trying to prove if a solution exists and if it does - how it can be constructed. For example - finding a walk in the Köingsberg bridge problem

Enumeration Problems are problems in which we are looking for how many objects have a given property. For example - how many optimal routes can be found in the Travelling Salesman Problem.

7.2 Basic Terminology

A graph, *G* is a pair (V, E) of sets:

V is a non-empty set of vertices (nodes)

E is a set of edges. Each element of *E* is a set of two distinct elements of *V*
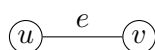
Figure 7.1: Example Graph

If $e \in E$, then $e = \{u, v\}$, u and v are different elements of *V* called the end vertices of *e* (where *e* joins vertices u and v).

The notation for v and u changes too; we can use uv instead of $e = \{u, v\}$ for the edge e, meaning uv and vu are the same edge. The vertices u and v are said to be *incident* with the edge uv or that they are *adjacent* because they are the end vertices of an edge.

We are only considering Finite Graphs in this module. These are graphs in which all their elements are defined and are capped in a certain place.
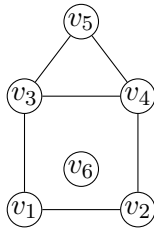
$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\}$$
$$E = \{\{v_1, v_2\}, \{v_2, v_4\}, \{v_1, v_3\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}\}$$
$$\text{or } E = \{v_1v_2, v_2v_4, v_1v_3, v_3v_4, v_3v_5, v_4v_5\}$$

Figure 7.2: Finite Graph

Note that although the diagrams of the graph can look nice, and help us to visualise them - they are not actually important for most of our problems.

7.2.1 Multigraph

A multigraph / pseudograph is like a graph, however it may contain loops and/or multiple edges. In this module, there is not a formal definition of one and we won't study them too much.
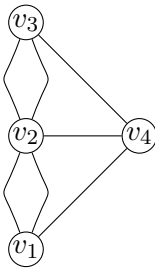
Figure 7.3: Graph with multiple edges
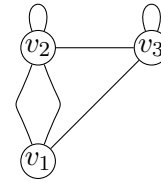
Figure 7.4: Graph with loops

7.2.2 Degree of a Vertex

The number of edges incident with a vertex, v, is called the degree of v and is denoted by $\deg v$. A vertex of degree 0 is said to be isolated.
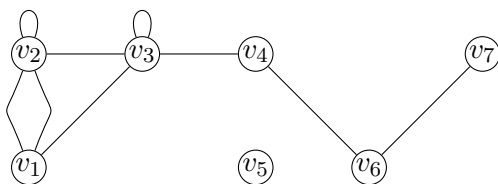
Figure 7.5: Degree of Vertex example graph

$$\deg V_1 = 2, \qquad \deg V_6 = 2,$$
$$\deg V_7 = 1, \qquad \deg V_5 = 0,$$
$$\deg V_1 = 3, \qquad \deg V_2 = 5,$$
$$\deg V_3 = 5$$

Note that V_5 is an isolated vertex in the above example.

7.2.3 Degree Sequence

When d_1, d_2, ..., d_n are the degrees of the vertices of a graph (or multigraph), G, ordered so that $d_1 \leq d_2 \leq \cdots d_n$ Then $(d_1, d_2, \ldots, d_n)$ is called the *degree sequence* of G.
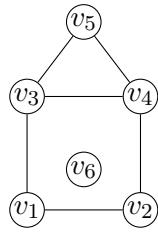
The degree sequence: $(0, 2, 2, 2, 3, 3)$

Figure 7.6: Example Graph For Degree Sequence

7.3 Euler Theorem

Euler's Theorem (or Handshaking Lemma) states that: In any graph $G = (V, E)$, the sum of all the vertex-degrees is equal to twice the number of edges,

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

This is a great revelation, however it lead to two more revelations:

1. In any graph, the sum of all the vertex-degrees is an even number

2. In any graph, the number of vertices of odd degree is even

7.4 Special Types of Graphs

We will now see that there are special types of graphs that have their own names.

7.4.1 Complete Graphs

For any positive integer n, the complete graph on n vertices, denoted K_n, is that graph with n vertices every two of which are adjacent.

Figure 7.7: K_1 Figure 7.8: K_2 Figure 7.9: K_3 Figure 7.10: K_4
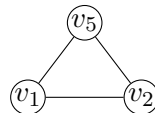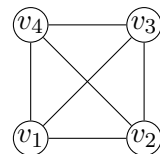
The complete graph on n has $\dfrac{n(n-1)}{2}$ edges because Eulers theorem.

7.4.2 Bipartite Graphs

A bipartite graph is one whose vertices can be partitioned into disjoint sets V_1 and V_2 in such a way that every edge joins in a vertex in V_1 and a vertex in V_2 (no edges within V_1 nor within V_2).
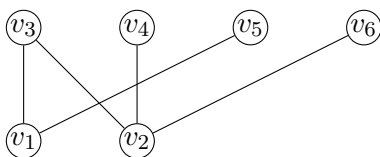
Figure 7.11: Incomplete bipartite graph   Figure 7.12: Complete bipartite graph
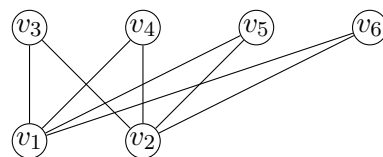
A complete bipartite graph is a bipartite graph in which every vertex in V_1 is joined to every vertex in V_2.
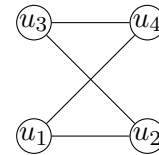
7.5   Subgraphs

A graph H is a subgraph of G if and only if the vertex and edge set of H are respectively subsets of the vertex and edge set of G.
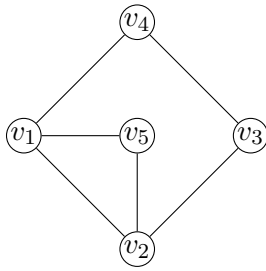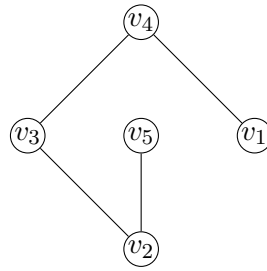
Figure 7.13: G Figure 7.14: H_1 Figure 7.15: H_2
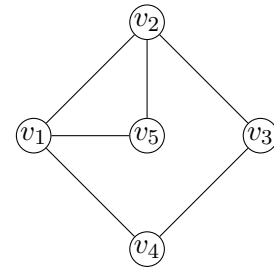
7.6   Isomorphic Graphs

Two graphs, G and H are said to be isomorphic if H can be obtained from G by re-labelling the vertices.
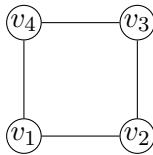
Figure 7.16: G Figure 7.17: H

In the above example, v_1 maps to u_1, v_2 to u_2 and so on. What this means is that if the u labelling was to be replaced with the v labelling - the same graph would be obtained.

G is isomorphic to H is there is a bijective function $f : V(G) \rightarrow V(H)$ such that

- if u and v are adjacent in G then $f(u)$ and $f(v)$ are adjacent in H

- if u and v are not adjacent in G then $f(u)$ and $f(v)$ are not adjacent in H.

It is difficult to prove that two graphs are isomorphic, as we have to try all the bijections between vertex sets and check them. It can be shown, that if G and H are isomorphic graphs then G and H:

- have the same number of vertices

- have the same number of edges

- have the same degree sequence

- either both are connected or both are not connected (to be covered next week)

So it can be easier to show that graphs are not isomorphic, which is shown by proving that one of the properties above is broken.

Page 8

Lecture - Walks, Trails, Paths

📅 2024-03-19   🕐 1700   🎓 Janka

8.1   Walks

A *walk* in a multigraph is an alternating sequence of vertices and edges (beginning and ending with a vertex), where each edge is incident with the vertex immediately preceding and following it. The length of a walk is the number of edges in it.

Many real problems, when translated to graph theory - enquire about the possibility of walking through a graph. Most of the definitions and results about walks are valid for graphs and multigraphs, even if we don't always specific this.
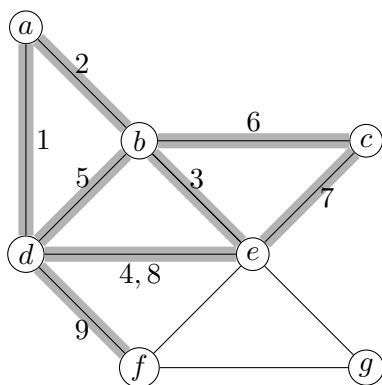
A walk of length 9: d - da - a - ab - b - be - e - ed - d - db - b - bc - c - ce - e - ed - d - df

Alternatively represented as:
(d, a, b, e, d, b, c, e, d, f)

Figure 8.1: Example of a Walk

A walk is *closed* if the first vertex is the same as the last, for example the walk (d, a, b, e, d, b, c, e, d), and is otherwise said to be an *open* walk.

8.1.1   Trails and Paths

A *trail* is a walk in which all edges are distinct. A *path* is a walk in which all vertices are distinct.

Figure 8.2: Example of a Trail
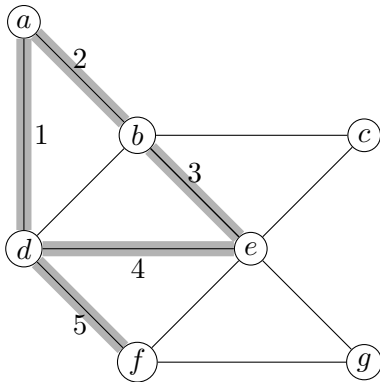
A trail (d, a, b, e, d, f) of length 5.

Not all the vertices of a trail are necessarily different.


Figure 8.3: Example of a Path
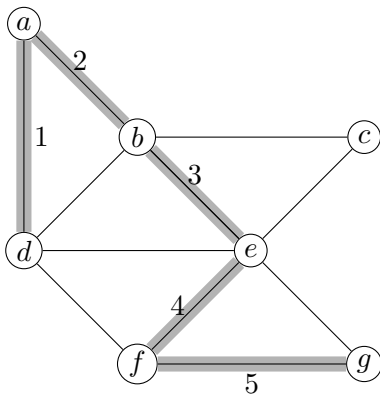
A path of length 5: (d, a, b, e, f, g).

All the vertices and edges of a path are different.

8.1.2   Circuits and Cycles

A closed walk in which all edges are different is called a *circuit* (this is a closed trail). A closed walk in which all vertices (except are the first and the last vertex) are different is called a *cycle* (this is a closed path).

Figure 8.4: Examples of Circuits and Cycles
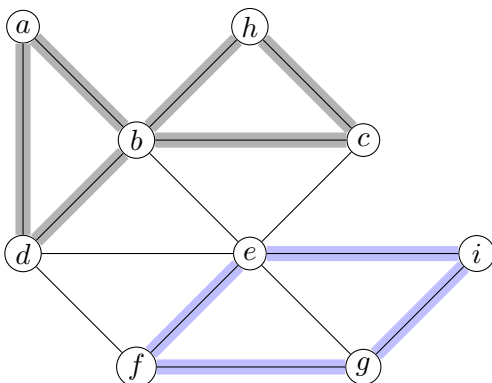
A circuit of length 6: (d, a, b, c, h, b, d)

A cycle of length 4: (f, e, i, g, f)

8.2   Connected Graphs

A graph, *G* is *connected* if there is a path in *G* between any pair of vertices; if this condition is not true then the graph is *disconnected*.
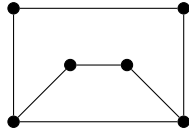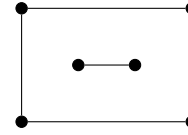
Figure 8.5: Connected Graph

Figure 8.6: Disconnected Graph

8.2.1 Bridges

An edge in a connected graph is a *bridge* if deleting it would create a disconnected graph.
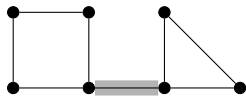

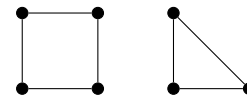Figure 8.7: Example of a Bridge

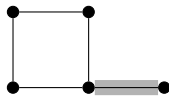Figure 8.8: Disconnected Graph
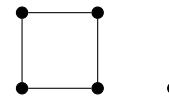

Figure 8.9: Example of a Bridge

Figure 8.10: Disconnected Graph

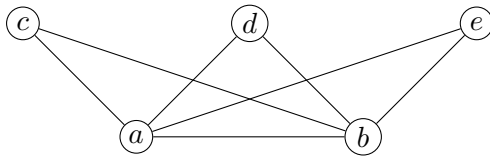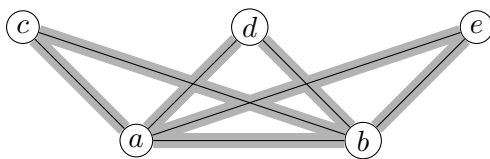## 8.3 Königsberg Bridge Problem & Eulerian Graphs

Figure 8.11: Königsberg bridges represented as a graph

Königsberg Bridge Problem: Is it possible to start on one of the land masses, walk over each of the seven bridges exactly once, and return to the starting point (without getting wet!)?

A graph is Eulerian if and only if it has a circuit that contains every edge - expressed a different way this is a closed walk using each edge exactly once (called an Eulerian circuit).

Figure 8.12: Königsberg bridges represented as a graph, showing circuit

Eulerian circuit: (a, c, b, d, a, e, b, a)

8.3.1 Eulerian Graphs

Not all graphs are Eulerian graphs, in fact there are many which are not.

We can characterise Eulerian graphs as graphs which can be drawn without removing the pen from the paper, and without covering any edges twice. This means that for each vertex there is one edge "in" and one edge "out" which means that the degree of each vertex must be even. This is a necessary and sufficient condition.

From the above information, we can deduce the theorem: *A multigraph is Eulerian if and only if it is connected and every vertex has an even degree.*

Applying the Theorem to our Königsberg bridge problem, we see that: the graph has vertices of odd degree ⇒ the graph is not Eulerian ⇒ a closed walk containing each edge exactly once does not exist. Therefore Euler has to destroy one bridge! (or just accept that the problem isn't solvable...)

8.3.2 Construction of Eulerian Circuit

We have a necessary and sufficient condition for a graph being Eulerian. To find an Eulerian circuit within Eulerian graphs, there exists an efficient algorithm. This algorithm is called *Fleury's Algorithm.*

8.3.3 Fleury's Algorithm

We start with an Eulerian Graph on the input

1. Choose any vertex to start

2. From that vertex, choose an edge to traverse. Choose a bridge only if there is no alternative.

3. After traversing that edge, erase it (and vertices of degree 0), coming to the next vertex.

4. Repeat steps 2 - 3 until all edges have been traversed, and you should be back at the starting Vertex.
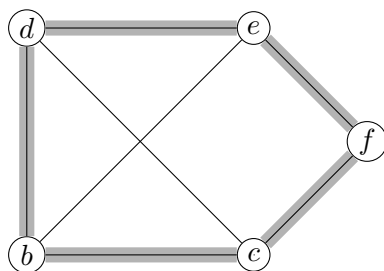
8.3.4 Semi-Eulerian Graphs

A connected graph with exactly two vertices of odd degree (called semi-Eulerian) contains an open (Eulerian trail) which includes every edge. This works because when we add an edge connecting the vertices of odd degree, we get a graph with all vertices of even degree. Therefore the graph is Eluerian, and therefore must contain an Eluerian circuit.

To find an Eluerian trail:

- Start at one of the odd degree vertices

- Construct an Eluerian circuit (and use the new edge at the end)

- The last vertex must be the second odd degree vertex

- You now have an open trail which includes every edge

8.4 Travelling Salesman Problem & Hamiltonian Graphs

A graph is *Hamiltonian* if and only if it has a cycle that contains every vertex - a closed path using each vertex exactly once (this is called a *Hamiltonian cycle*). For example:

Hamiltonian cycle: (d, e, f, c, b, d)

Figure 8.13: Hamiltonian cycle

Note that the definition might look similar to Eulerian graphs, however the results are very different.

There is no "if and only if" sufficient and necessary condition which can be used to categorise a Hamiltonian Graph.

8.4.1   Construction of a Hamiltonian Graph

There are known algorithms for finding a Hamiltonian cycle however none at present are known that would guarantee to find it in a reasonable amount of time. The known algorithms use an exhaustive search of all possibilities, requiring exponential or factorial time in the worst case.

8.5   Adjacency Matrix

So far, we have explored graphs using pictorial representations of them - this is all well and good when we are looking at them in notes, however when we are trying to get a computer to understand them, we have to store the graph in a slightly different format. There are several possibilities of how the information about a graph can be coded when working in a program, for example using sets, or more commonly the *adjacency matrix*.

We can take the formal definition of the adjacency matrix to be: Let G be a graph with n vertices labelled v_1, v_2, ..., v_n. The *adjacency matrix* of G is the $n \times n$ matrix $A = (a_{ij})$ whose (i, j) entry is a_{ij}, where for each i and j with $1 \leq i, j, \leq n$, define:

$$a_{ij} \begin{cases} 1 & \text{if } v_i v_j \text{ is an edge} \\ 0 & \text{if } v_i v_j \text{ is not an edge} \end{cases}$$

8.5.1   Example Adjacency Matrix

Shown below is an example graph, G, the corresponding adjacencies represented in a table and the corresponding adjacency matrix, A.
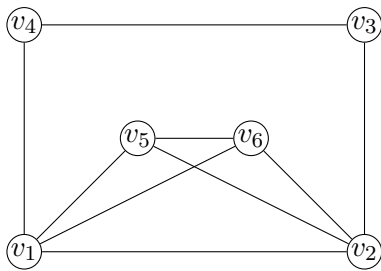

Figure 8.14: G

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	0	1	1	1
v_2	1	0	1	0	1	1
v_3	0	1	0	1	0	0
v_4	1	0	1	0	0	0
v_5	1	1	0	0	0	1
v_6	1	1	0	0	1	0

Table 8.1: Table of adjacencies

$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

8.5.2 Properties of an Adjacency Matrix

- The diagonal entries of A are always 0; that is $a_{ii} = 0$ for $i = 1, \ldots, n$.

- The adjacency matrix is symmetric, that is $a_{ij} = a_{ji}$ for all i, j.

- $\deg v_i$ is the number of 1's in row i; this is also the number of 1's in column i (row i and column i are the same).

- If we square the matrix A, we get A^2 and some interesting properties:

    - The (i, i) entry of A^2 is the degree of v_i
    - The (i, j) entry of A^2 is the number of different walks of length 2 from v_i to v_j.
    - In general, for any $k \geq 1$, the (i, j) entry of A^k is the number of walks of length k from v_i to v_j.
    - In general, for any $k \geq 1$, the (i, j) entry of A^k is the number of walks of length k from v_i to v_j.

8.6 Matrix Multiplication: a recap

NB: This was originally taught in M30943 (Architectures & Operating Systems, Maths Component) at level 4.

If we take two matricies:

$$A = \begin{bmatrix} 1, 7 \\ 2, 4 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 3, 3 \\ 5, 2 \end{bmatrix}$$

We can find the matrix $C = A \times B$.

However first, we will assign each element of the matrix an identifier and see how the rows of A are multiplied by the columns of B to produce C:

$$\begin{bmatrix} a_{1,1}, a_{1,2} \\ a_{2,1}, a_{2,2} \end{bmatrix} \times \begin{bmatrix} b_{1,1}, b_{1,2} \\ b_{2,1}, b_{2,2} \end{bmatrix} = \begin{bmatrix} (a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1}), (a_{1,1} \times b_{1,2} + a_{1,2} \times b_{2,2}) \\ (a_{2,1} \times b_{1,1} + a_{2,2} \times b_{2,1}), (a_{1,2} \times b_{2,1} + a_{2,2} \times b_{2,2}) \end{bmatrix}$$

Now we can substitute the references in for their values into the equations:

$$\begin{bmatrix} 1, 7 \\ 2, 4 \end{bmatrix} \times \begin{bmatrix} 3, 3 \\ 5, 2 \end{bmatrix} = \begin{bmatrix} (1 \times 3 + 7 \times 5), (1 \times 3 + 7 \times 2) \\ (2 \times 4 + 3 \times 5), (2 \times 3 + 4 \times 2) \end{bmatrix}$$

Now we can, finally, substitute the actual values into C:

$$\begin{bmatrix} 1,7 \\ 2,4 \end{bmatrix} \times \begin{bmatrix} 3,3 \\ 5,2 \end{bmatrix} = \begin{bmatrix} 38,17 \\ 26,14 \end{bmatrix}$$
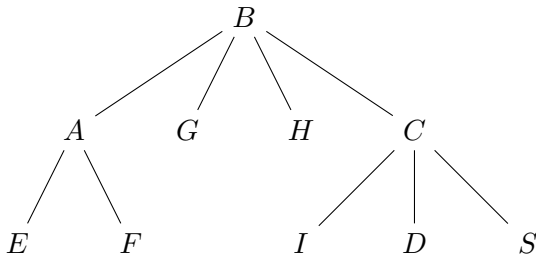
Page 9

Lecture - Trees

📅 2024-04-16 🕐 17:00 🎓 Janka

9.1 Trees: A Gentle Introduction

The word "tree" has many different uses in the English Language. We will not be exploring Christmas trees, outdoor trees or file trees in this module - rather we will be exploring the mathematical sub-class of graphs called a "tree". In maths, a "tree" is a connected graph that contains no cycles.

Alternatively, we can consider the following *mathematical* definition of a tree: G is a tree. G is connected and acyclic (without cycles). Between any two vertices of G there is precisely one path

Figure 9.1: G, a tree

As we can see above, trees have a simple structure. However, to enumerate all (non-isomorphic) trees with n vertices is very difficult unless n is small. An example of this can be seen below.

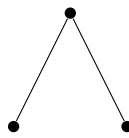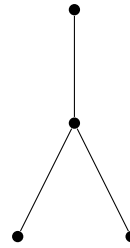Figure 9.2: Tree   Figure 9.3: Tree   Figure 9.4: Tree   Figure 9.5: Tree   Figure 9.6: Tree

The above trees all are $n \leq 4$

9.1.1 Basic Properties of Trees

A tree with more than one vertex must contain a vertex of degree 1 - this is considered to be a leaf (or terminal vertex). This is because if we take a vertex at random, v_1, we can then search outward along a path from v_1 looking for a vertex of degree 1; finding this vertex would indicate the end of the path and therefore indicate we have a tree. Should this vertex not be found - we would find a circuit (proving this is not a tree).

9.2 Is It A Tree?

To work out if a graph contains a tree, we can use the following theorem: *A connected graph with n vertices is a tree **if and only if** it has n − 1 edges.*

Within this theorem - we are most concerned with the conditional propositional ("*if and only if*") as this is the deciding factor as to if the graph contains a tree or not.

9.2.1 Example

If we take a connected graph's degree sequence as:

$$(1, 1, 2, 2, 2)$$

Is it a tree?

The solution to this starts with identifying the number of vertices and the number of edges. The number of edges can be calculated from the sum of the degree sequence, divided by two. Therefore we know that this graph has 4 edges. The number of vertices is the number of entries in the degree sequence, therefore we know that this graph has 5 vertices. As the number of edges is 1 less than the number of vertices, the condition $n − 1$ is true and therefore we have a tree!

9.3 Spanning Trees

A *Spanning Tree* of a connected graph, G, is a subgraph that is a tree and that includes every vertex of G. Spanning Trees are considered to be different if they make use of different edges on the graph.
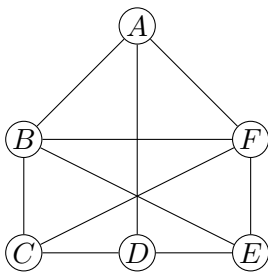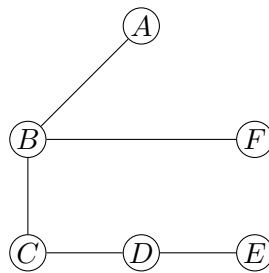

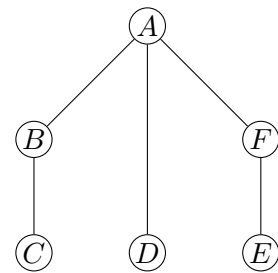Figure 9.7: Original Graph   Figure 9.8: Spanning Tree 1   Figure 9.9: Spanning Tree 2

Spanning Trees can be used in a variety of cases, one of these is in a problem of a cash-strapped Council attempting to repave some pavements...
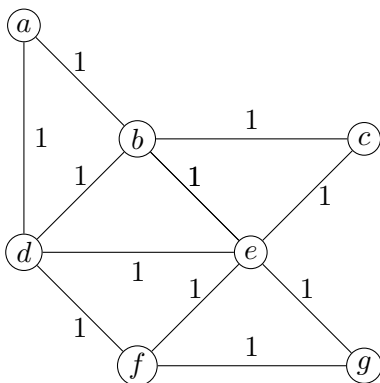

Figure 9.10: Untouched Pavement Layout

Problem: The Council plans to pave certain roads in a way such that anyone can get between any two towns on pavement. What roads should be paved so as to minimise the total length of pavement required? (Note that, at this stage, all pavements have length 1.)
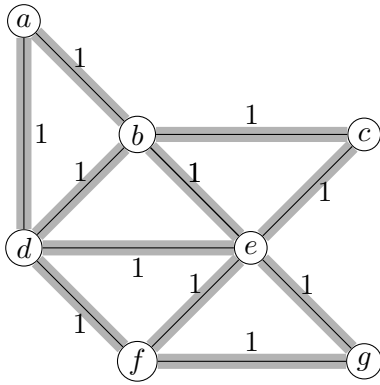
Figure 9.11: First Attempt at paving

In this first attempt, we pave all roads. This means that the Council would need to pay for 11 roads to be paved.

However this is not the ideal solution - we can use the idea of a Spanning Tree to improve this solution and therefore reduce the number of roads which need paving.
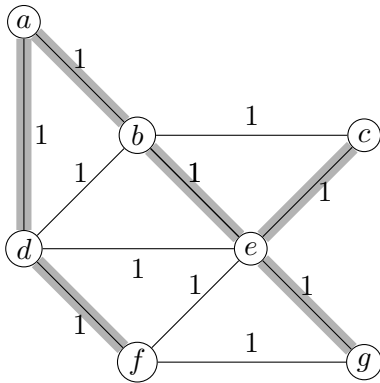

Figure 9.12: An optimum paved solution

This is now *an* optimum solution. We have removed all edges from the Tree other than those which are required to satisfy the property that there is one path between any two given nodes.

9.3.1 Finding a Spanning Tree

It is relatively easy to find a spanning tree in a connected graph, G. If G, with n vertices, has $n - 1$ edges - it is already a spanning tree; or if G has no cycles then it is already a tree, so G itself is a spanning tree for G. For trees where it's spanning tree has not already presented itself to us - we work through the edges in the graph, deleting them until the spanning tree property is achieved. This can be seen below.
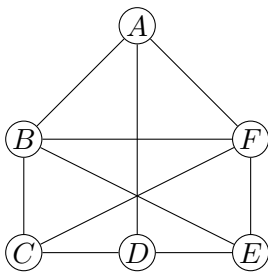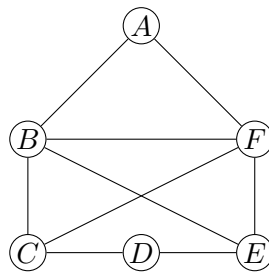

Figure 9.13: Original Graph
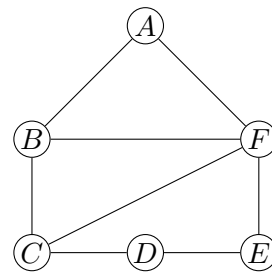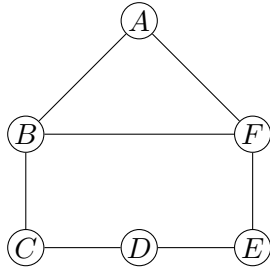

Figure 9.14: Removed *AD*
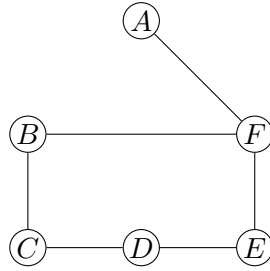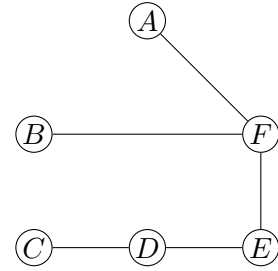

Figure 9.15: Removed *BE*

Figure 9.16: Removed CF Figure 9.17: Removed AB Figure 9.18: Removed BC

9.3.2   Using a Depth First Search To Find Spanning Trees

Rather than testing to see if each edge is in a cycle, and removing it if it is, to find spanning trees - it is possible to use a Depth First Search based algorithm. Depth First Searches are also useful in other Graph Applications, for example to test whether a graph is connected and to produce a spanning tree in the connected case. The method is based on exploring the vertices.

A Depth First Search to find a spanning tree works by:

1. Start at any vertex (label it)

2. Choose any adjacent unlabelled vertex to it (label it, and move to it).

3. Repeat step 2 until there is no unlabelled adjacent vertex to it

4. Find the last labelled vertex with an unlabelled adjacent vertex (backtrack to this) then go to step 2

5. Algorithm complete when you get back to the first labelled vertex

The algorithm for finding a Spanning Tree using DFS is shown below. The input is a connected graph G with vertices ordered $(v_1, v_2, ..., v_n)$ and the output is a spanning tree $T = (V', E')$.

```
V' = {v₁}, E' = ∅, w = v₁
while (true)
    while (∃wv ∈ W such that T and wv don't create a cycle in T)
        choose the vertex with min k, vₖ, that when added to T
        doesn't create a cycle in T
        E' = E' ∪ {wvₖ}, V' = V' ∪ {vₖ}, w = vₖ
    end while
    if w = v₁
        return T
    w = parent of w in T //backtrack
end while
```

9.4   Minimum Spanning Trees

It's all well and good fining a tree when all the edge 'weights' are equal to 1 - this works for *some* graphs. However, when we consider a properly weighted graph with it's edges having different weightings, we need to look for a slightly different tree type. A *minimum spanning tree* of a weighted graph is a spanning tree of least weight (that is, a spanning tree for which the sum of the least weights of all its edges is least among all spanning trees). In some good news, there's a few algorithms which can be used to find the minimum spanning tree...

9.4.1   Kruskal's Algorithm

Kruskal's algorithm works by starting with a weighted graph then adding edges in order from lowest weight to highest weight, as long as they don't create a circuit. The steps are shown below:

1. Start with the last edge

2. Add the next edge with the least weight, as long as this won't create a circuit

3. Repeat step 2

4. Stop when you have $n - 1$ edges (where n is the number of vertices)

Kruskal's algorithm is a *greedy algorithm*, this means it makes decisions without any regard for consequences which this decision might have in the future, however the current decisions are appearing to be good.

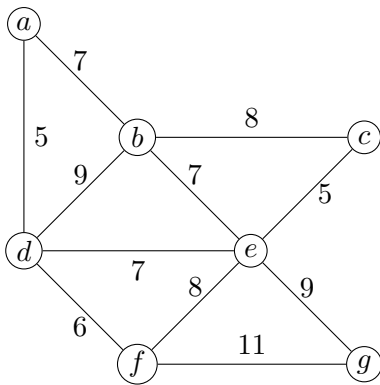The full process of this is displayed below.

Figure 9.19: Initial Graph

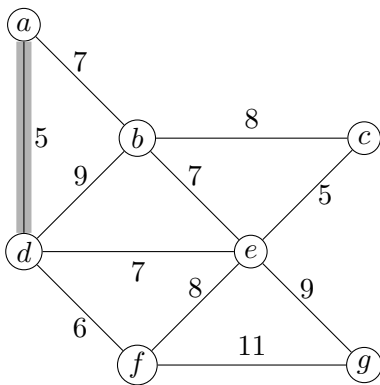This is the graph which we need to find the spanning tree of.

Figure 9.20: Kruskal's step 1

To start, we analyse what weights we have on the edges. The lowest weight is 5 and the highest is 11. We therefore pick one of the edges with weight 5 and mark that as part of our Minimum Spanning Tree.
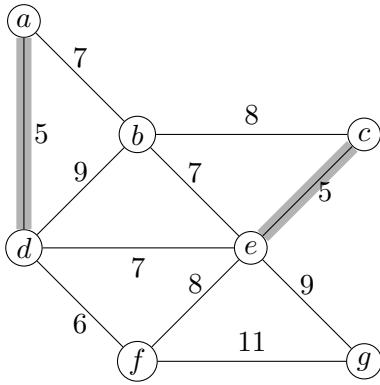
Figure 9.21: Kruskal's step 2

Next, we repeat the same process again. As there is still one edge with weight 5, we analyse that one; and as when connecting it to the Minimum Spanning Tree, we don't get a circuit - we can add it to the Minimum Spanning Tree.
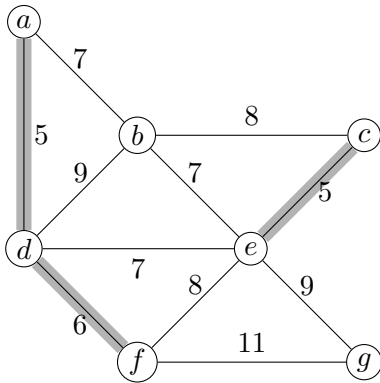
Figure 9.22: Kruskal's step 3

We've now exhausted all edges with weight 5, so we look for the next weight up. In this case it's 6. We check for circuits then add to the Minimum Spanning Tree.
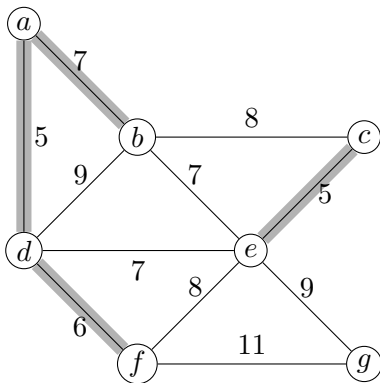
Figure 9.23: Kruskal's step 4

We've now exhausted all edges with weight 6, so we look for the next weight up. In this case it's 7. As there are a few options for which edge we could pick, we choose one at random. We check for circuits then add to the Minimum Spanning Tree.
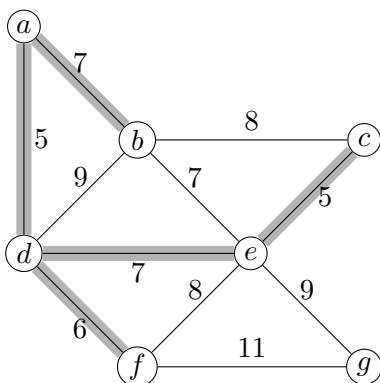
Figure 9.24: Kruskal's step 5

As there are still edges with weight 7 which we are yet to examine and add, we do so. We check for circuits then add to the Minimum Spanning Tree.
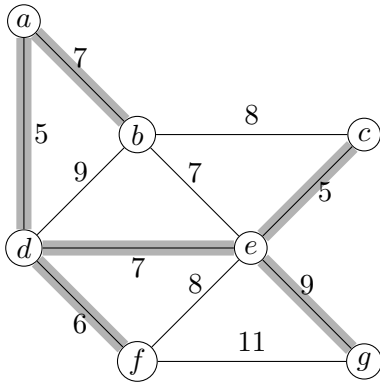
Figure 9.25: Kruskal's step 6

We now find ourselves in the most complicated step, typical for the last step! We know that we need to add 1 more edge to the Minimum Spanning Tree to make the $n - 1$ criteria come true.

We cannot add the *be* edge, as this would result in a circuit being formed, so we look to the next logical option. However, we cannot add either the *bc* or *ef* edges as either of these would also cause a circuit to be formed. Therefore, we are forced to go for the edge *eg*, as this is the lowest weighted edge which we can add without forming a circuit.

9.4.2 Prim's Algorithm

Prim's Algorithm works by starting with any random vertex, then adding all the adjacent edges to that to a list of possible edges. From these possible edges, an edge is selected where it doesn't already connect to another visited vertex and that has the least weighting value, and this is added to the Minimum Spanning Tree. This process is then repeated, with edges of the new vertex added to the possible list and the edge with the lowest weighting (that doesn't cause a circuit when added) is added to the Minimum Spanning Tree. The algorithm stops when either there are n vertices in the Minimum Spanning Tree or when there are $n - 1$ edges in the Minimum Spanning Tree. The stages of the algorithm are outlined below, with the edges added to the Minimum Spanning Tree in grey and the potential edges in blue.
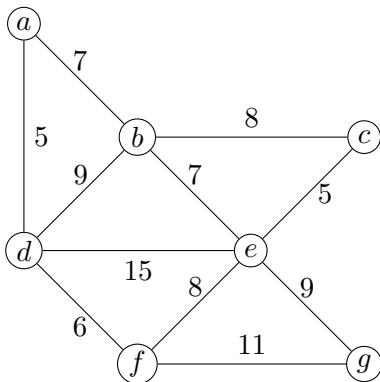
Figure 9.26: Initial Graph

This is the graph which we need to find the spanning tree of.
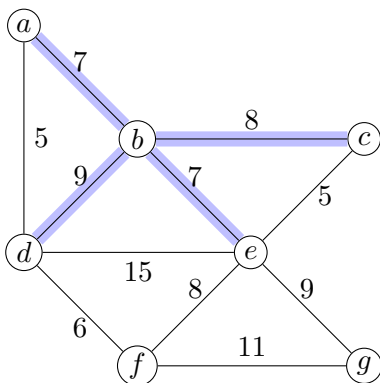
Figure 9.27: Prim's step 1

We choose node *b* to start with. From this, we mark all it's adjacent edges as potential to add.
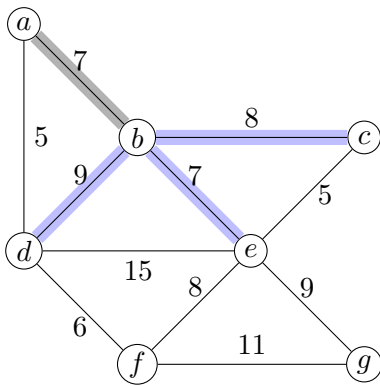
Figure 9.28: Prim's step 2

Now we have our potential vertices, we can choose one. After looking at the options, we see that there are two which are possibilities (*ab* and *be*) which both have weights of 7. As neither would cause a circuit to be created, we choose one at random. In this case, we've chosen *ab*.
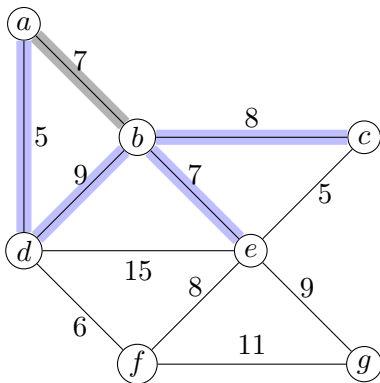
Figure 9.29: Prim's step 3

We now have a new vertex available to use. From this, we can add all it's adjacent nodes which are not part of the Minimum Spanning Tree to the list of potentials.
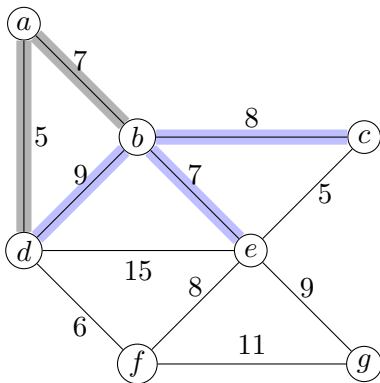
Figure 9.30: Prim's step 4

We now take a look at the options for edges to add to the Minimum Spanning Tree and see that the edge with the lowest value is our new one, *ad*. As from adding this to the Minimum Spanning Tree, we don't get a circuit - we can progress with adding *ad* to our Minimum Spanning Tree.
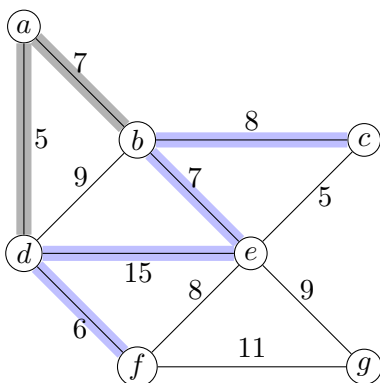
Figure 9.31: Prim's step 5

As vertex *d* is now in our Minimum Spanning Tree, we can add all it's adjacent edges to our potential list. We can also remove *db* as it is no longer a viable option for adding to the Minimum Spanning Tree as it would cause a circuit.
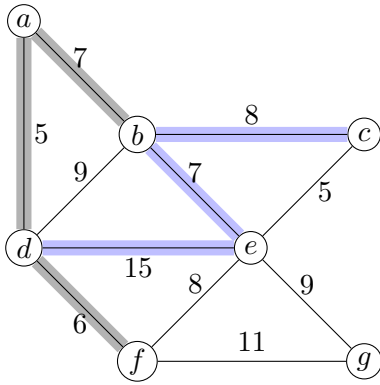
Figure 9.32: Prim's step 6

We now look at the options for what edge to add to the Minimum Spanning Tree. The next lowest edge value is df, at a weight of 6. As this doesn't create a circuit - we can add it to the Minimum Spanning Tree.
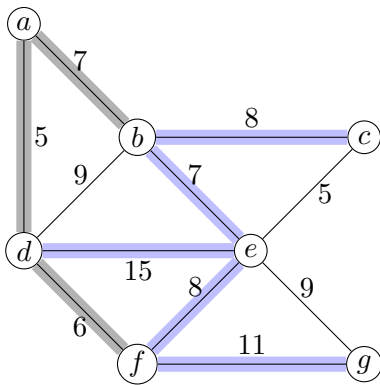
Figure 9.33: Prim's step 7

We can now add edges adjacent to f to the potential list.
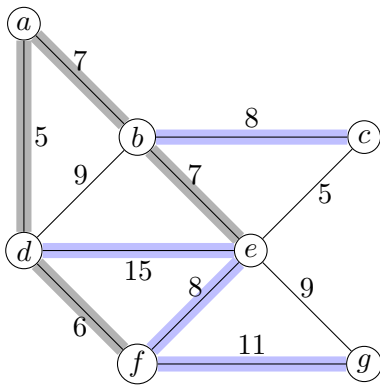
Figure 9.34: Prim's step 8

We now look again at options for what edges we can add to the Minimum Spanning Tree. The option with the lowest weight is be, and if adding it, it wouldn't create a circuit. Therefore we add it.
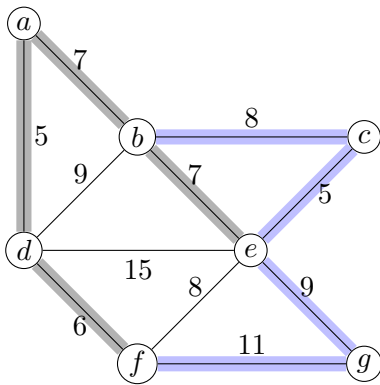
Figure 9.35: Prim's step 9

As we have added a new node to our Minimum Spanning Tree, we can add it's adjacent edges to the potential list. We also need to remove a number of edges from the potential list as these are no longer viable solutions, as they would cause circuits to be created.
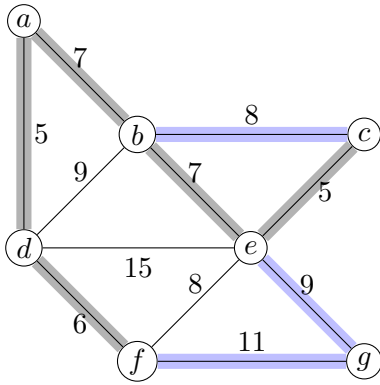
Figure 9.36: Prim's step 10

We can now repeat the step of looking for the edge with the least weighting and then checking to see if by adding that we would introduce a circuit or not. The edge *ec* has weight 5 which is the lowest of the potential list, and by adding it we would not be introducing a circuit.
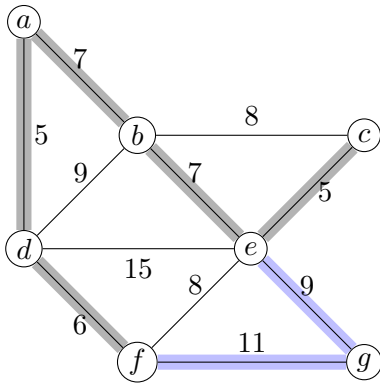
Figure 9.37: Prim's step 11

As we have now added *c* to the Minimum Spanning Tree, we can review the potential list and remove invalid options, such as *bc*. There are no new edges to add to the potential list as they all have existed from other vertices.
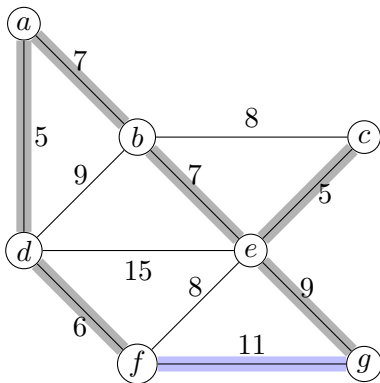
Figure 9.38: Prim's step 12

We can now look for the edge in the potential list with the lowest weight. This is *eg*. We then check to see if by adding it to the Minimum Spanning Tree, a circuit would be created and as the answer is no, we can add it to the Minimum Spanning Tree.
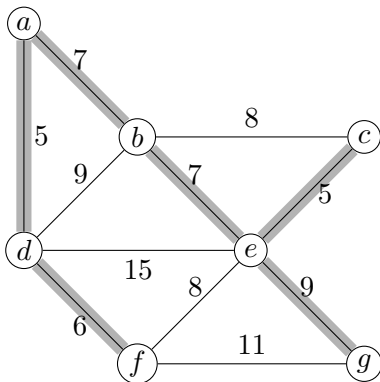
Figure 9.39: Prim's step 13

The final stage for Prim's algorithm is to remove any edges from the potential list.

Prim's Algorithm has produced a Minimum Spanning Tree with length 39.

9.5 Rooted Tree Terminology

A tree is rooted if it comes with a specified vertex, called the root. Each vertex in a tree has zero or more children - the vertices "below" it in the tree. A vertex that has a child is called the child's parent vertex. If two vertices have the same parent, they are called siblings.
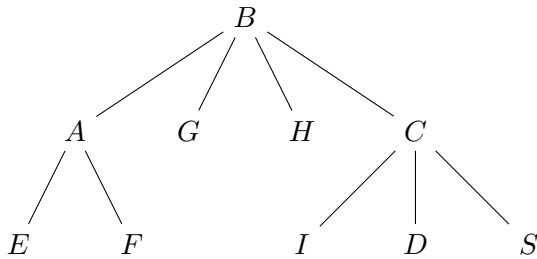
B is the root node
A is a child of B
E and F are siblings
D is the parent of C

Figure 9.40: Example of a Rooted Tree

Page 10

Lecture - Network Models and Digraphs

📅 2024-04-23   🕐 17:00   🎓 Janka

10.1   Directed Graphs

Another use of graphs, which we are yet to explore, is how they can be used to model real life situations. In situations where edges are representing roads or pipes then they need to be associated with directions / weights, from this we get a directed graph (a *digraph*).

A Digraph is a pair (V, E) of sets, V is a nonempty set of vertices, E is a set of ordered pairs of distinct elements of V, called arcs (edges).

A digraph can be pictured like a graph, with the orientation of an arc indicated by an arrow. A digraph is just a graph in which each edge has an orientation or direction assigned to it.
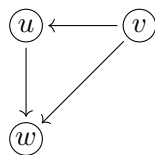
$V = \{u, v, w\}$
$E = \{vu, vw, uw\}$
Note that for vu, the direction is from v to u, therefore uv and vu are different.

Figure 10.1: Example of a Digraph

10.1.1   Some Properties

Similar to graphs, digraphs do not contain multiple arcs / loops.

Each vertex of a digraph has:

indegree which is the number of arcs directed into that vertex; and

outdegree which is the number of arcs directed out of that vertex
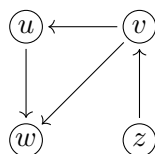
vertex u has indegree 1 and outdegree 1,
vertex v has indegree 1 and outdegree 2,
vertex w has indegree 2 and outdegree 0,
vertex z has indegree 0 and outdegree 1

Figure 10.2: Example of a Digraph

Similar, formal, definitions exist for digraphs as do that for walks, paths, etc. However it is necessary to follow the direction of the arcs.

10.1.2 Adjacency Matrix

The adjacency matrix A of G with vertices v_1, v_2, …, v_n is defined by setting $a_{ij} = 1$ if there is an arc from v_i to v_j (and set to 0 otherwise). A is generally not symmetric.
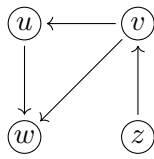
$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Figure 10.3: A digraph, G

Most assertions made about the adjacency matrix for graphs apply with appropriate changes for digraphs as well. Within an adjacency matrix, the outdegree of the vertex v_1 is the number of 1's in the row i; the indegree of the vertex v_i is the number of 1's in the column i. The (i, j) entry of A^k is the number of different walks of length k from v_i to v_j respecting the orientation of arcs, where $k \geq 1$.

10.2 Network Models

Directed graphs can be useful for modelling network problems such as a transportation network; a pipeline network; or a computer network. In each case, the problem is generally to find the maximum flow. Maximising the flow in a network is a problem that belongs to both graph theory and operations research.

If we take an example in which the arcs of a digraph (a network) can represent an oil pipeline network and show the direction the oil can flow.
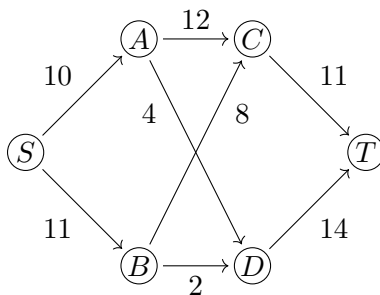
Oil is unloaded at the dock, S, (the source) and pumped through the network to the refinery, T, (the sink).

Figure 10.4: Example of a Digraph

The weight on the arcs shows the capacities of the pipelines. Our goal is to pump as much oil as possible from S to T. To formalise such a concept, we use the term *a flow*.

10.2.1 A Flow

A flow in a network is a description of the amount of commodity that can flow along the network (in a unit of time). No pipe must receive more than it can cope with (therefore 'flow ≤ capacity' for each arc); and no commodity must be lost along the way ('flow in = flow out' for all vertices except S and T).

A flow assigns to each arc e which is a non-negative number, $f(e)$, subject to the previous two constraints. Note the notation is of the form *flow, capacity* assigned to each arc. This can be seen in the following example:

Figure 10.5: Example of a Flow
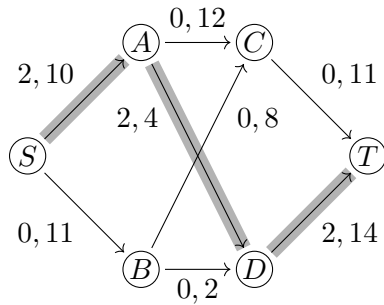
We camp pump 2 units from S to T through the A and D and 0 through the other arcs.

potential edges in blue. a 7 5 8 b 9 This assignment has the properties:

- Each arc "flow $(e) \leq$ capacity (e)"

- For each vertex A, B, C, D "the flow into each one is equal to the flow out of it"

Therefore this is a flow!

The value of a flow is the sum of all flow for all outgoing arcs form the source S. In our above example, this would be $2 + 0 = 2$.

The value of the flow must be the same as the sum of flows for all incoming arcs to T. So, the value of this flow is 2.

10.2.2 Formalisation

10.2.2.1 Formal Definition of a Network

Let $G = (V, E)$ be a directed weighted graph with the following properties:

- a vertex S (source) has no incoming arcs - and is the start of the flow

- a vertex T (sink) has no outgoing arcs - and is the end of the flow

The non-negative weight on each arc is the capacity of the edge, e, denoted by $c(e)$, and $c(e) \geq 0$, which is the maximum amount of some commodity which can flow through it in a unit of time (for example, litres of oil, kW of electricity, etc).

10.2.2.2 Formal Definition of a Flow

A flow is a mapping that assigns to each edge, e, a number, denoted by $f(e)$, which satisfies two conditions:

- Feasibility Condition where $0 \leq f(e) \leq c(e)$ for each edge $e \in E$, in words - the flow along each arc must be less than or equal to the capacity of that arc.

- Conservation of flow where for each internal vertex u (not source or sink), the sum o flows along the arcs into u is equal to the sum of the flows along the arcs out of u.

$$\sum_{v \in V} f(uv) = \sum_{v \in V} f(vu) \text{ for all } u \in V - \{S, T\}$$

10.2.2.3 Formal Definition of the Value of a Flow

The value of a flow: $\sum\limits_{v \in V} f(Sv)$ (which is the sum of S-outgoing arcs).

By the flow conservation law, none of the flow is lost at any vertex. This means, the value of the flow is also equal to the sum of flows of all arcs going into T:

$$\sum_{v \in V} f(vT) = \sum_{v \in V} f(Sv)$$

Which is equal to the sum of T-incoming arcs and the sum of S-outgoing arcs.

A maximum flow is a flow of maximum value.

10.2.3 Construction of Flows

The simple way of finding a maximum flow is to locate a path, P, from S to T (which follows the direction specified by the arrows on arcs) and define a flow by setting:

$$f(e) = \begin{cases} 1, & \text{if } e \in P \\ 0, & \text{if } e \notin P \end{cases}$$
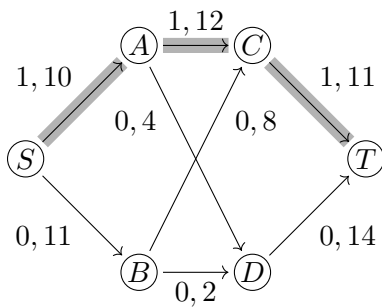
This digraph shows the flow $SACT$, which has a value of 1.

Figure 10.6: Flow Construction example pt. 1

The next stage is to increment the arcs flow rate on the path until you reach the maximum capacity of the smallest arc on that path.
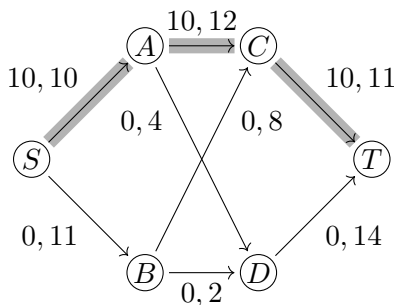
The path $SACT$ has a saturated arc SA, this is a flow (10), equal to its capacity (10). A flow along the patch $SACT$ cannot be improved.

The value of the flow is 10.

Figure 10.7: Flow Construction example pt. 2

We now look for another path from S to T with an unsaturated edge.
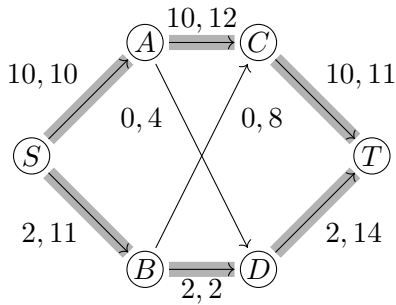
The path $SBDT$ has a saturated arc BD (where the flow along the path can't be improved)

The value of the flow is 12.

Figure 10.8: Flow Construction example pt. 3

We now look for another unsaturated edge which we can form into a path from S to T.
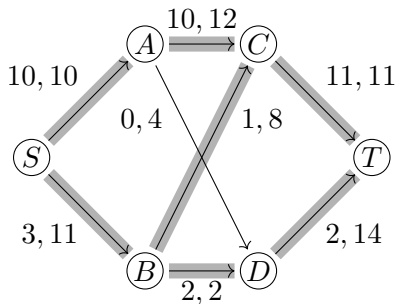
The path $SBCT$ has a saturated arc, CT.

The value of the flow is 13.

Figure 10.9: Flow Construction example pt. 4

10.2.4   Improving the Flow

However, as we could expect - we're not done yet! There are still further improvements to be made to our flow.

If we consider a path, P, from S to T in the underlying undirected graph (eg $P = (S, B, C, A, D, T)$). Then:

- an edge, e, is a forward arc (with respect to the path P) if the orientation of e follows the direction of p, eg SB or AD,

- otherwise (if arcs are directed against the direction of the path P), e is a backwards arc, eg CA.

We will now also consider a path from S to T (in an underlying undirected graph) with the properties:

- All forward arcs on the path are unsaturated

- All backwards arcs on the paths are carrying a non-zero flow
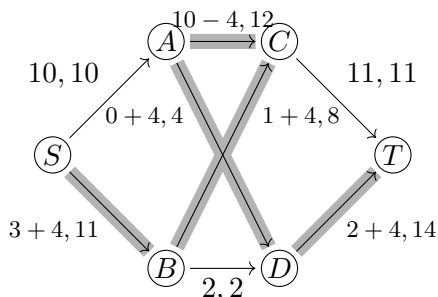
Using the grey path the flow can be increased by 4.

Figure 10.10: Increasing Flow

Note that no feasibility condition has been broken in the process. The conservation condition (where before "flow out = flow in" for all internal vertices) is now:

- For vertices B and D: "flow out + 4 = flow in + 4"

- For vertex C: "flow out = flow in + 4 - 4"

- For vertex A: "flow out + 4 - 4 = flow in"

Here we can see that changing the flow by 4 (+4 for forward arcs, -4 for backward arcs) along the "path" (S, B, C, A, D, T) increases the flow by 4.

We now need to look at why the value 4 is so special. For forward arcs, if we calculate e using $c(e) - f(e)$:

- $SB : 11 - 3 = 8$

- $BC : 8 - 1 = 7$

- $AD : 4 - 0 = 4$

- $DT : 14 - 2 = 12$

And for backwards arcs, we take e as $f(e)$ therefore:

- $CA : 10$

To calculate the value of improvement which can be made on a path, we first have to calculate the values as we did above, then find the minimum resulting value (in our case, that's 4). We then add 4 to our forward arcs, and minus 4 from our backwards arcs on this new path. From this we get the new arc values which we can substitute in on our digraph.

Figure 10.11: Increased Flow
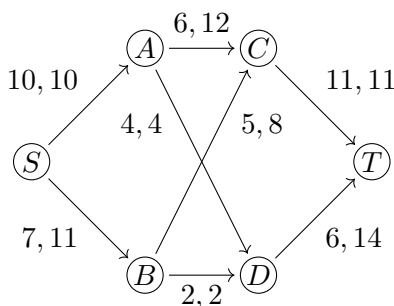
By checking all the possibilities, we can see that there are no further possibilities for improvement. This means tha maximum flow has now been achieved.

The value of the flow is 17.

10.2.5 Formal Definitions for Flow Improvement

Let P be a path from S to T in an underlying undirected graph to a network G satisfying the following conditions:

- for each forward arc e in $P : f(e) < c(e)$,

- for each backward arc e in $P : 0 < f(e)$

Such a path is called a flow-augmenting path; and if on$\{f(e)|e$ is a backward arc in $P\}e$ exists in the network - a flow can still be improved.

The outline of the algorithm for finding the maximum flow is shown below:

1. Start with a flow (e.g. a flow with 0 for each edge)

2. Search for a flow-augmenting path P. If no such path exists, stop - the flow is maximum

3. Increase the flow through the path by Δ, where Δ is the minimum over the values $\{f(e)|e$ is a backward arc in $P\}$ and $\{c(e) - f(e)|e$ is a forward arc in $P\}$. Change the flow along the arcs in P to be:

   - for each forward arc e in P: $f(e) := f(e) + \Delta$
   - for each backward arc e in P: $f(e) := f(e) - \Delta$

   Go back to step 2.

An exhaustive search of all paths from S to T can be used for finding a flow-augmenting path P. When there exists more than one flow-augmenting path, the order in which we use them is not important. There is double check, whether our value is correct, using the following theorem:
Theorem: In a directed graph, the value of a maximum flow is equal to the value of a minimum cut.

10.2.6 (S, T)-cut

An (S, T)-cut is a partition $\{\mathcal{S}, \mathcal{T}\}$ of V such that $S \in \mathcal{S}$ and $T \in \mathcal{T}$. This can be seen below:

$\mathcal{S} = \{S, A, B\}, \mathcal{T} = \{C, D, T\}$
or
$\mathcal{S} = \{S, A\}, \mathcal{T} = \{C, D, T, B\}$
or
$\mathcal{S} = \{S\}, \mathcal{T} = \{C, D, T, B, A\}$
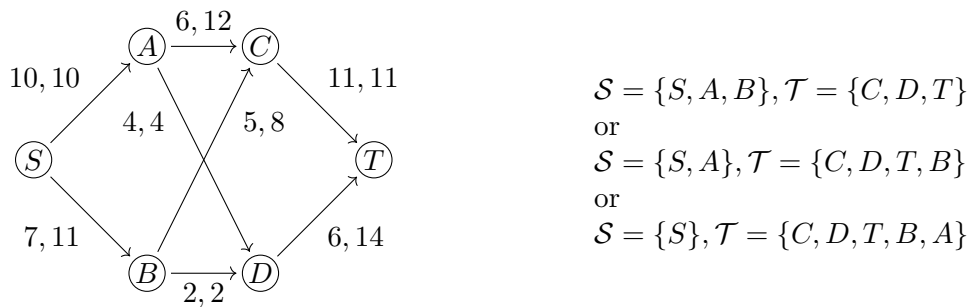
Figure 10.12: Example Directed Graph

The capacity of an (S, T)-cut, $\{\mathcal{S}, \mathcal{T}\}$, is the sum of capacities of all arcs from $\mathcal{S}$ to $\mathcal{T}$.

A minimum (S, T)-cut is an (S, T)-cut of the smallest possible capacity.

For example, if we take the above Digraph and the following (S, T)-cut:

$$\mathcal{S} = \{S\}$$
$$\mathcal{T} = \{A, B, C, D, T\}$$

Then we can calculate the capacity of the (S, T)-cut as:

$$cap(\mathcal{S}, \mathcal{T}) =$$
$$= c(SA) + c(SB)$$
$$= 10 + 11$$
$$= 21$$

We can use the property of a (S, T)-cut and max-flow to verify that we have found the maximum possible flow for a digraph - in that the value of a maximum flow is equal to the value of a minimum (S, T)-cut.

Page 11

Lecture - Connectivity and Cuts

📅 2024-04-30   ⏱ 1700   🎓 Janka

In Graph Theory, properties of graphs are intensively studied, These properties will be covered in this lecture:

- edge-connectivity;

- vertex-connectivity

11.1   Edge-Connectivity

The edge-connectivity, $\lambda(G)$ of a connected graph, G, is the smallest number of edges whose removal may disconnect G. This can be seen with a graph below:

If we remove any edge, the graph is still connected. However, there exists two edges such that if we remove them, the graph becomes disconnected. Therefore $\lambda(G) = 2$.
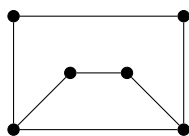
Figure 11.1: Connected Graph

As we have seen above, this means that we can take G to have an edge-connectivity $\lambda(G)$ if:

- there exists $\lambda(G)$ edges such that their removal disconnects G

- removal of any $\lambda(G) - 1$ edges does not disconnect G.

11.1.1   Edge-Cuts

The edge cut of a graph is a set of edges whose removal renders the graph disconnected. Two examples of this can be seen below, with the edges to remove in grey.
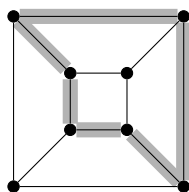
Figure 11.2: 6 edges, displayed
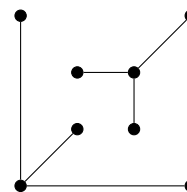
Figure 11.3: 6 edges, removed
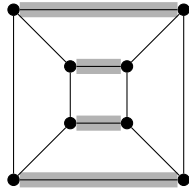
Figure 11.4: 4 edges, displayed
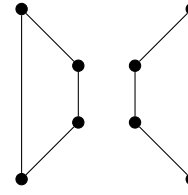
Figure 11.5: 4 edges, removed

The edge-connectivity of a graph is the size of the smallest edge cut of G. A graph, G, is k-edge connected when $\lambda(G) \geq k$. This means it remains connected whenever fewer than k edges are removed.

For example, a 2-edge-connected or 3-edge-connected graph would have an edge-connectivity $(\lambda(G))$ of 3, therefore:

$$\lambda(G) \geq k \text{ for } k = 1, 2, 3$$

11.2   Vertex Connectivity

The vertex-connectivity, $\kappa(G)$, of a connected graph, G, is the smallest number of vertices whose removal may disconnect G. Note that when a vertex is removed from a graph, the edges incident to it are also removed. This can be seen in an example below
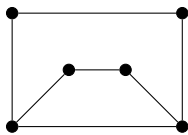
Figure 11.6: Example Graph
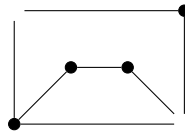
Figure 11.7: TL & BR vertices removed
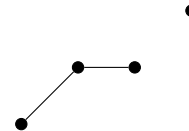
Figure 11.8: Incident edges removed

After removing any one vertex, the graph is still connected; however there exists two vertices such that removing both of them means the graph becomes disconnected. Therefore $\kappa(G) = 2$.

A vertex cut of a connected graph, G, is a set of vertices whose removal renders G disconnected. The vertex-connectivity, $\kappa(G)$ is the size of the smallest vertex cut.

In complete graphs, the vertex-connectivity definition breaks down because G cannot be disconnected by removing vertices. The vertex connectivity of a complete graph, K_n, is $n - 1$.

A graph, G, is said to be k-vertex connected (or k-connected) if the graph remains connected when you delete fewer than k vertices from the graph.

11.3   Upper Bounds for Connectives

To get the upper bounds on $\lambda(G)$ and $\kappa(G)$ we can use the following theorem: *For any connected graph $G = (V, E)$:*

$$\kappa(G) \leq \lambda(G) \leq \delta(G) \leq \frac{2|E|}{|V|}$$

$\delta(G)$ *is the smallest vertex-degree in G.*

Note that $\frac{2|E|}{|V|}$ is the average of the vertex-degrees; therefore the sum of the vertex-degrees is $2|V|$.

11.4   Optimal Connectivity

A graph, G, where:

$$\kappa(G) = \lambda(G) = \delta(G) = \frac{2|E|}{|V|}$$

is a graph which has the maximum vertex-connectivity and the maximum edge-connectivity possible for any graph with $|V|$ vertices and $|E|$ edges. A graph with these conditions is said to have *optimal connectivity*.

All graphs which have optimal connectivity are regular graphs where every vertex has the same degree, however not every graph has optimal connectivity.

Part II

Functional Programming

Page 12

Lecture - Introduction to Functional Programming

📅 2024-01-22   🕐 1200   🎓 Matthew

12.1   Introduction

Functional Programming is a different programming paradigm. There are all sorts of different ways we can classify a programming language, paradigm being one of them. More details on this in another module.

12.1.1   Imperative vs Functional Programming

Before we go too deep into Functional Programming, we will first look at the structure of Imperative Programming.

Imperative Programming is a paradigm where the execution of the program consists of executions of *statements*, which each impact the program's *state*. *Side Effects* can be caused by the statements; these are things that the program does where it cannot guarantee the outcome - for example get the current temperature, ask the user to enter a number or getting the system time.

Pure Functional Programming does not have a state, does not have statements and does not have side effects. However - side effects are a "necessary evil" so they get brought back in isolated from the main program. Once side effects are introduced, our functional programming becomes impure.

12.2   Functional Programming

In Functional Programming, there are three key terms - expression, evaluation and value. An expression is some text which has a value, for example `2 * 3 + 1`; a value is the thing which the expression has, for example `7`; and evaluation is the process used to obtain a value from an expression.

We will start our FunProg journey looking at Mathematical Functions, which we can think of as a box which maps argument values to a result value. A Haskell program is mainly made up of Function definitions, for example

```
square :: Int -> Int
square n = n * n
```

The first line above starts with the function name, then lists the parameters (a single `Int`), then finally the result type. The second line is the actual function, starting with the name, then the names assigned to the parameters, then the value which gets returned. The double colon (`::`) is read as 'is type of'.

12.3   Data Types: A Brief Introduction

Haskell includes a number of basic data types which we can make use of.

12.3.1   Boolean

The `Bool` data type has the values `True` and `False`. As well as having the boolean operators `&&` (and); `||` (or); and `not` (not). These can be seen implemented in the following function which implements the *exclusive or* operator (which gives True when exactly one of its arguments is True):

```
exOr :: Bool -> Bool -> Bool
exOr x y = (x || y) && not (x && y)
```

12.3.2   Int & Float

Haskell includes a number of different numerical data types - we'll start of using the `Int` and `Float` types. `Int` is a fixed-space integer data type; and `Float` is a floating point data type. Operators for these data types include:

- arithmetic operators: `+`, `-` and `*`

- The `Float` data type includes floating point division `/`

- The `Int` has integer division and remainder functions `div` and `mod`

- relation operators: `==`, `/=` (not equals), `<`, `>`, `<=` and `>=`

The operators use the standard precedence rules (as experienced in other languages).

12.4   Conditional Expressions

Haskell includes a conditional expression which takes the form:

```
if condition then m else n
```

where `condition` is a boolean expression and `m` & `n` are expressions of the same type Where `condition` is true - the expression evaluates to `m`; and where `condition` is false - the expression evaluates to `n`.

In the next lecture, we'll see an alternative to conditional expressions.

Page 13

Lecture - Introduction To Functional Programming II

📅 2024-01-29   🕐 1200   🎓 Matthew

13.1   Evaluation and Calculation

In a similar way to that of tracing an imperative program to understand the effect of executing their statements on the program state; we can evaluate expressions of a functional program step-by-step to understand the operation of it. This is also known as calculation.

The process of calculation will be explained with the example function:

```
twiceSum x y = 2 * (x + y)
```

which we can evaluate using the example inputs `4` and `(2 + 6)`. The complete calculation of the expression is as follows:

```
twiceSum 4 (2 + 6)
⤳ 2 * (4 + (2 + 6))        def of twiceSum
⤳ 2 * (4 + 8)              arithmetic
⤳ 2 * 12                   arithmetic
⤳ 24                       arithmetic
```

13.2   Guards

Guards are Boolean expressions used in function definitions to give alternative results dependent on the parameter values. The following function gives the largest of the two `Int` values:

```
maxVal :: Int -> Int -> Int
maxVal x y
| x >= y     = x
| otherwise  = y
```

We saw the `if ... then ... else ...` construct last week, which could be used in place of guards. However guards are the preferential thing to use where there are more than one case - as they allow for this easier. Shown below is a function, `maxThree`, which returns the largest of three `Int` values passed in:

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
| x >= y && x >= z   = x
| y >= z             = y
| otherwise          = z
```

Calculations involving guards are represented slightly differently, note the `??` denoting when the function is inside the guard. Shown below is the calculation of `maxThree`:

```
maxThree 3 2 5
?? 3 >= 2 && 3 >= 5      first guard
?? ⤳ True && 3 >= 5     def of >=
?? ⤳ True && False      def of >=
?? ⤳ False              def of &&
?? 2 >= 5               second guard
?? ⤳ False              def of >=
?? otherwise            third guard
?? ⤳ True               def of otherwise
⤳ 5
```

13.3   Local Definitions

Single line definitions in Haskell can be slightly unwieldy to read, write and understand. For this reason - we may want to breakdown the definition's expression to make it easier to read. For example the function:

```
distance :: Float -> Float -> Float -> Float -> Float
distance x1 y1 x2 y2 = sqrt ((x1-x2)^2 + (y1-y2)^2)
```

can be broken down to the following:

```
distance x1 y1 x2 y2 = sqrt (dxSq + dySq)
where
    dxSq = (x1 - x2) ^ 2
    dySq = (y1 - y2) ^ 2
```

From the above example, we see that the main expression uses the local definitions and the local definitions use the function's parameters. The local definitions can only be used within the functions that they are defined within; they are "hidden" from the rest of the program. The local definitions can appear in any order within the `where`.

Page 14

Lecture - Pattern Matching & Recursion

📅 2024-02-12   🕛 1200   🎓 Matthew

14.1 Modules

As with Python and other popular programming languages, Haskell supports modules (libraries) which provide pre-written and tested code to do certain things. As with other languages, when using a module in Haskell - we have to import it with the `import` command. The first line below shows importing the entire module and the second line shows importing only two functions:

```
import Data.Char
import Data.Char (toUpper, toLower)
```

Haskell's standard library which is auto-imported into every other module & the interpreter is called the *standard prelude*.

14.2 Functions & Operators

Haskell includes both functions and operators. Functions (`sqrt`, `mod`, etc) are used in prefix notation (i.e. `mod n 2`). Operators (`+`, `-`, `**`, etc) are used in infix notation (i.e. `1 + x`); apart from the unary minus operator which is a prefix operator.

It is possible to use any binary (two-argument) function as an operator by surrounding it with back-quotes (`` ` ``). For example ``n `mod` 2`` is equivalent to `mod n 2`. Similarly, we can use an operator as a function by encasing the operator in parentheses (`()`). For example `(+) 1 x` is equivalent to `1 + x`.

14.3 Pattern Matching

So far, we have seen two ways of defining functions:

- using single equations

- using guards

We will now add a third function definition mechanism to our options: pattern matching.

Pattern matching consists of a sequence of equations; each *pattern* (on the left hand side) is associated with a different result (on the right hand side). An example of this is seen below, defining the `not` function which is included in the Prelude.

```
not :: Bool -> Bool
not True  = False
not False = True
```

It is also possible to use a wildcard to simplify pattern matching. This can be seen below where the Boolean *or* operator is redefined.

```
(||) :: Bool -> Bool -> Bool
False || False = False
_ || _         = True
```

Alternatively, we can also use *named parameters* which can take a value from the pattern and use it in the output. An example of this is shown below

```
(||) :: Bool -> Bool -> Bool
True || _   = True
False || p  = p
```

14.4   Recursion

As with other programming languages, Haskell supports recursion. A recursive definition is one that is defined in terms of itself.

Recursion is a critical component in Haskell as pure functional programming does not support loops as these are imperative constructs (as they operate on a program's state). We shall recuse lots especially when using lists.

14.4.1   Recursive Definition of Factorial

To illustrate recursion in Haskell, we will examine an example of the Factorial function. $fact(n)$ is the product of the integer n and all the integers below it $n-1$, $n-2$, …. For example:

$$fact(3) = 3 \times 2 \times 1 = 6$$

Note that the factorial of a number $n > 0$ can be defined in terms of the factorial of $n-1$, for example

$$fact(4) = 4 \times fact(3)$$

This leads us to the following recursive function definition

```
fact :: Int -> Int
fact n
    | n > 0       = n * fact (n - 1)
    | n == 0      = 1
    | otherwise   = error "undefined for negative ints"
```

14.4.2   Primitive vs General Recursion

Primitive recursion is recursion where

- the base case considers the parameter value 0

- the recursive case considers how to get from value $n-1$ to n

General Recursion is a recursive function where there is not a precise halting case, but there is a halting condition. For example, in the `divide` function below - the halting case is where `n < m` - not when either of them are at exact values.

```
divide :: Int -> Int
divide n m
    | n < m       = 0
    | otherwise   = 1 + divide (n - m) m
```

Page 15

Lecture - Tuples, Strings & Lists

📅 2024-02-12   ⏰ 12:00   🎓 Matthew

15.1   Characters & Strings

Haskell comes with both the `String` and `Char` types. In Haskell, a single quote (`'`) is used to denote characters and double quotes (`"`) for strings. For example:

```
ghci> :type 'a'
'a' :: Char

ghci> :type "Sam"
"Sam" :: String
```

The `Char` module defines some useful functions on characters. Such as converting to uppercase or checking if a provided character is a digit.

```
ghci> import Data.Char

ghci> toUpper 'a'
'A'

ghci> isDigit 'a'
False
```

15.2   Tuples

As seen in other languages, we can combine multiple pieces of data into one data type. For example, we may want to combine a name (represented as a string) and a mark (represented as an integer) for example (`"Dave"`, 74) which would have the tuple type (`String, Int`). This example can be furthered to a small program which takes two Tuples representing a student and outputs the name of the student with the higher mark:

```
betterStu :: (String, Int) -> (String, Int) -> String
betterStu (s1, m1) (s2, m2)
    | m1 >= m2  = s1
    | otherwise = s2
```

We can define a *type synonym* which can be used in the place of the raw definition of the tuple. For example, if we define the type synonym on the first line below, we can then re-define `betterStu` to use that.

```
type StudentMark = (String, Int)
betterStu :: StudentMark -> StudentMark -> String
```

```
betterStu (s1, m1) (s2, m2)
    | m1 >= m2  = s1
    | otherwise = s2
```

Tuples can also be used to enable a function to return more than one value.

15.3 Lists

Lists are the main data structure in Haskell. They are used to store any number of data values *of the same type*. For example the first list below is a list of integers and the second example is a list of strings.

```
[12, 64, -92, 85, 12]
["This", "is", "a", "list"]
```

The type of a list (`l`) is denoted by `[l]`. For example:

```
ghci> :type [True, False, False]
[True, False, False] :: [Bool]
```

The empty list (`[]`) is an element of any list type.

15.3.1 Strings as a List of Chars

Strings in Haskell are simply lists of characters, therefore the type `String` is declared as:

```
type String = [Char]
```

This means that the two expressions below are the same

```
['h', 'e', 'l', 'l', 'o']
"hello"
```

This also means that the operations that we will cover later in this lecture (for example concatenation of lists) therefore also apply to strings.

15.3.2 Lists from Ranges

It is possible to use the operator `..` to generate lists. For example:

```
[3 .. 9]     = [3, 4, 5, 6, 7, 8, 9]
[3.1 .. 9]   = [3.1, 4.1, 5.1, 6.1, 7.1, 8.1, 9.1]
['a' .. 'z'] = "abcdefghijklmnopqrstuvwxyz"
```

We can also add an argument to give steps different from 1 as seen below:

```
[3, 5 .. 15]      = [3, 5, 7, 9, 11, 13, 15]
[0, 0.1 .. 0.5]   = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5]
```

15.3.3 List Comprehension

We can use list comprehension to build one list from another list. For example, if we define:

```
aList = [1, 2, 3, 4, 5]
```

THen the list comprehension shown on the first line below will have the value on the second line below:

```
[2*i | i <- aList]

[2, 4, 6, 8, 10]
```

We read this as "take all `2*i` where `i` comes from `aList`". The `<-` symbol represents the set member symbol ($\in$) from Maths.

List comprehension is extremely powerful as can be seen in the following example:

```
bList = [2, 3, 6, 9, 4, 7]

ghci> [mod i 2 == 0 | i <- bList]
[True, False, True, False, True, False]
```

We can take this another step further where we can add a test at the end of the generator. The following example has given all the values that are less than 5 from `cList`

```
cList = [2, 3, 6, 9, 4, 8]

ghci> [i * 2 | i <- cList, i < 5]
[4, 6, 8]
```

We can use list comprehension and a pattern on the left hand side of the `<-` as seen below:

```
addPairs :: [(Int, Int)] -> [Int]
addPairs pairList = [i + j | (i, j) <- pairList]

ghci> addPairs [(1, 2), (4, 8), (6, 3)]
[3, 12, 9]
```

15.4   Polymorphic Functions

The *Prelude* comes with a number of functions built in, we will now examine some of these. While doing so, we will examine *polymorphism*.

If we consider the `length` function, which gives the number of elements in a list (returning an `Int`), which works for any type of lists. For this reason, we may think of it to have the following types:

```
length :: [String] -> Int
length :: [Bool] -> Int
```

However! The actual type of `length` is given as:

```
length :: [a] -> Int
```

Here, we are using a *type variable* which stands for an *arbitrary type*. By convention `a`, `b`, `c`, … are used as type variables.

The expressions `[a] -> Int` is known as the *most general type* for `length`

15.4.1   List Functions

We can define polymorphic functions, by not giving a type declaration. Haskell will try and infer the mst general type by looking at the structure of the function.

15.4.1.1   Adding Elements To Front Of List

For lists in Haskell, one of the most used operations is `:`. This adds an element to the front of the list and is of type `a -> [a] -> [a]`. For example:

```
ghci> 3:[5, 7, 2]
[3, 5, 7, 2]
```

15.4.1.2 List Concatenation

The ++ operator can be used to join two lists together:

```
ghci> "hello" ++ "world"
"helloworld"
```

15.4.1.3 Return Element From Specific Position

The !! operator returns an element at a given position. For example:

```
ghci> ["fish", "ham", "cheese", "spam"] !! 2
"cheese"
```

Note that eventhough indexing a list like this is common in other languages, we will not tend to use this operator that often.

15.4.1.4 Checking If List Is Empty

The null function tests whether a list is empty. For example:

```
ghci> null [1, 2]
False
```

Page 16

Lecture - List Patterns and Recursion

📅 2024-02-19   🕛 1200   🎓 Matthew

16.1   Patterns, a Recap

Many of the functions we have been using so far have been defined using patterns. These can include literal values, variables, and wildcards. All of these can be seen below in the redefinition of `or` from lecture 3.

```
or :: Bool -> Bool -> Bool
or True _    = True
or False a   = a
```

Patterns can also include Tuples. For example the `fst` and `snd` functions from the Prelude:

```
fst (x,_)  = x
snd (_,y)  = y
```

Note that these projection functions are polymorphic - they work for tuples of data of any kind.

16.2   Lists and List Patterns

As we saw in the last lecture, the `:` operator will append whatever precedes it to the list which succeeds it. For example:

```
ghci> 4:[7,3]
[4,7,3]
```

This means that every lit can be built from `[]` and `:`, which are constructors for lists.

Note that the `:` operator is right-associative meaning that it works from right to left adding the right-most to the second right-most, and so on. For example:

```
ghci> 1:2:[]
[1,2]
```

We can use these constructors in patterns. Where we use `:` in list patterns when we want to deal with the first element and the rest of the list separately. For example, the following example (from the Prelude) returns the first element of a list:

```
head :: [a] -> a
head (x:xs) = x
```

Note that the naming convention `x:xs` is standard; where we say "x, exes".

It is also possible to use wildcard matching:

```
head :: [a] -> a
head (x:_) = x
```

16.3 Recursion over Lists

As we already know from recursion in the previous weeks, we require a base case and a recursive case. When recursing over lists - the base case considers the empty list `[]` and the recursive case gives the result for any non-empty list (one matched by `x:xs`) from the result of the tail of the list `xs`.

If we take the example of the implementation of a Prelude defined function `sum`.

```
sum :: [Int] -> Int
```

The function is defined to return a sum of a list of integers.

The base case of our function will be when the list is empty, as the sum of `[]` is obviously `0`. The recursive case will be where `xs` is greater than 0. Therefore, we can define the function as

```
sum []      = 0
sum (x:xs)  = x + sum xs
```

16.3.1 General Recursion over Lists

The above recursive function we have explored is of the type 'primitive recursion'. It is also possible to use general recursion over lists.

For example, we take the Prelude function `zip` which has the following definition:

```
zip :: [a] -> [b] -> [(a,b)]
```

This function joins two lists into a single list of tuples:

```
ghci> zip ['r', 'h', 'a'] [4, 7, 2]
[('r',4),('h',7),('a',2)]
ghci> zip [5, 7, 1, 5] ['a', 'b']
[(5,'a'),(7,'b')]
```

Note that the lists don't have to be of the same length, and that if the lists are different lengths then the last few elements of the longer elements are dropped.

The easiest way to define `zip` is by beginning with the recursive case of two non-empty lists `x:xs` and `y:ys`. In this case we can place `x` and `y` into tuple, and zip up the tails `xs` and `ys` which give us:

```
zip (x:xs) (y:ys) = (x.y) : zip xs ys
```

Now we have one part of the program left - handling when one argument is `[]`. There are three possibilities for this:

- `y` is empty

- `x` is empty

- `x` and `y` are empty

These can all be written together with a wildcard:

```
zip _ _ = []
```

which gives us the entire function:

```
zip :: [x] -> [y] -> [(x,y)]
zip (x:xs) (y:ys) = (x.y) : zip xs ys
zip _ _ = []
```

Page 17

Lecture - Functions as Values

📅 2024-02-26   🕐 12:00   🎓 Matthew

17.1   Functions as Arguments

In Functional Programming, functions can be treated as data. This means that they can be passed as arguments to other functions; and that they can be returned as results from functions. A function that either takes another function as an argument; or returns a function is known as a *Higher-Order Function*. Higher-Order programming can be very expressive.

The following function definition applies a function to a value and then applies the same function again to the result.

```
twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)
```

We would also need to define a function to be used in the above function:

```
succ :: Int -> Int
succ n = n + 1
```

Now that we have two functions, we can look at an example calculation:

```
twice succ 5
⤳  succ (succ 5)    def of twice
⤳  succ (5 + 1)     def of succ
⤳  (succ 6)         arithmetic
⤳  (6 + 1)          def of succ
⤳  7                arithmetic
```

17.1.1   Function Composition

Haskell includes a function composition operator ".". For two functions, `f` and `g`, the expression

```
(f . g) x
```

means apply `g` to `x` then apply `f` to the result. Note that this is the same as the mathematical operation $f \circ g$. The below code snippet shows the order in which the operator works:

```
(f . g) x = f (g x)
```

The output type of the first function must be of the same type as the input type of the second function.

17.1.2 Function-Level Definitions

The composition operator, `.`, allows us to easily define functions in terms of other functions. For example, we can replace the definition:

```
twice f x = f (f x)
```

with:

```
twice f = f . f
```

A function defined just in terms of other functions (without reference to other arguments) is known as a function-level definition. This notation can also be known as "point-free style"

17.2 Partial Application

A powerful feature shared by many functional languages is that functions can be partially applied. If we consider the following simple function definition which multiplies two numbers together:

```
multiply :: Int -> Int -> Int
multiply x y = x * y
```

We would typically consider the only thing that this function does is to multiply two parameters together and return the result. However - it would be more correct to consider that we can apply the function to one argument and are left with a function of one argument.

Returning to our example, we may use it as follows:

```
double = multiply 2
```

which would then mean that:

```
ghci> double 5
10
```

As we can see - `double` is taking a single argument which is being passed to `multiply` along with it's defined argument. This allows us to understand more about how Haskell works, in that every function in Haskell takes exactly one argument. This means that `multiply`, with two input parameters, is actually taking a single parameter (`Int`) then returning another function, `Int -> Int` which is used to produce the final result.

We can see that:

```
multiply :: Int -> Int -> Int
```

is shorthand for:

```
multiply :: Int -> (Int -> Int)
```

17.2.1 Operator Sections

It is not only the functions which can be partially applied - it is also possible to partially apply an operator. The following are examples of operator sections:

- `(2*)` - a function that multiplies its arguments by 2

- `(/2)` - a function that divides its argument by 2

- `(2/)` - a function which gives 2 divided by its argument

- `(>3)` - a function which tests if its argument is greater than 3

17.3   Patterns of Computation

When designing algorithms, there are a number of things that we often want to do to a list:

- transform every element of a list in some way

- remove those elements of a list that don't possess a given property

- combine all the elements with a particular operation

Haskell includes a number of higher-order functions that implement each of these patterns.

17.3.1   Mapping

If we take the example function, `doubleAll` which doubles every element in the list. A recursive example of this is below:

```
doubleAll :: [Int] -> [Int]
doubleAll [] = []
doubleAll (x:xs) = 2*x : doubleAll xs
```

This function has a pattern in it, where an operator (could also be a function) is applied to every element of the list.

The Prelude defines a function `map` which takes the operation to be applied as a parameter. Using `map`, the definition of the function `doubleAll`become (note the use of an operator section):

```
doubleAll xs = map (2*) xs
```

However, we can make this simpler. We do not need to specify `xs` because that is added complexity. The simplest form of this function can be seen below:

```
doubleAll = map (2*)
```

17.3.2   Filtering

Filtering doesn't change any of the elements of the list - rather it checks if the elements pass the test and keeps them if so. For example:

```
keepPositive [] = []
keepPositive (x:xs)
    | x > 0     = x : keepPositive xs
    | otherwise = keepPositive xs
```

Using the `filter` function, we are able to condense this down to the following:

```
keepPositive = filter (>0)
```

Note here that we have used the simplest form of the function as we do not need to specify the list we want to filter in the function definition.

17.3.3   Folding

The Prelude contains the function `foldr` which apples the function specified on the provided list. It is used to combine the elements of the list in some way. For example, the following function which adds every element in the list:

```
addUp :: [Int] -> [Int]
addUp [] = 0
addUp (x:xs) = x + addUp xs
```

can be rewritten using `foldr` as follows

```
addUp = foldr (+) 0
```

Note that you still have to specify the type for the function when using `foldr`, for example for `addUp` this would be:

```
addUp :: [Int] -> Int
```

Page 18

Lecture - Algebraic Types

"There's nothing in the blob, it's a blob"

18.1   Types in Haskell: A Recap

As we have already seen, there are a number of built-in types in Haskell:

- `Int`, `Float`, `Bool` and `Char` (the basic types)

- `(Int, Int, Char)` (tuple types)

- `[Int]` or `[(Int, Char)]` (the list type)

It is also possible to give convenient names to types using synonyms, for example:

- `type HouseNumber = Int`

- `type StreetName = String`

- `type Address = (HouseName, StreetName)`

These types are only so well and good, as they do not provide a convenient way to model more complex structures (such as Binary Trees). Within Haskell, we are able to use *Algebraic Types* to define arbitrarily complex types.

18.2   Algebraic Types

We declare the algebraic type using the keyword `data`, followed by:

- the name of the type being defined

- a list of constructors

Note that both have to begin with a capital letter. For example:

```
data Day = Mon | Tue | Wed | Thur | Fri | Sat | Sun
```

As we can see above, the simplest algebraic types are those where the constructors don't take any arguments. Here, the constructors are the data values (or members of the type). We call a type which has been defined in this way an *enumerated type*.

18.2.1 Enumerated Type

Another example of an enumerated type is Haskell's Boolean type:

```
data Bool = False | True
```

The simplest way to define a function on an Enumerated Type is by pattern matching. Using the `Day` data type an example is:

```
isWeekend :: Day -> Bool
isWeekend Sat = True
isWeekend Sun = True
isWeekend _   = False
```

This might look rather obtuse, and that would be a correct observation. This is because our data type (`Day`) doesn't include any operators such as `==`. It is possible to define `==` ourselves, however this would be tedious.

18.2.2 Algebraic Types & Type Classes

We can get Haskell to provide a `==` operator by declaring that we want our type (`Day`) to be a member of the `Eq` type class. (Note that `Eq` includes all types that include `==` and `/=`). To do this, we would need to define `Day` as follows:

```
data Day = Mon | Tue | Wed | Thur | Fri | Sat | Sun
            deriving (Eq)
```

Now that we have the `==` operator implemented for `Day`, we can redefine `isWeekend` as:

```
isWeekend :: Day -> Bool
isWeekend day = day == Sat || day == Sun
```

However, `Eq` only includes *Eq*uality operators - what happens if we want to compare two values and see if one is greater than the other.

Fortunately, Haskell includes more type classes which can provide us with the functionality that we require for this:

- `Ord` provides us with `<`, `<=`, `>` and `>=`

- `Show` to provide a function

    ```
    show :: Day -> String
    ```

    which converts `Day` values into strings

- `Read` which provides a function

    ```
    read :: String -> Day
    ```

    which converts strings (for example "Mon") into `Day` values.

When using all these additional type classes, we may end up with the following definition of `Day`:

```
data Day = Mon | Tue | Wed | Thur | Fri | Sat | Sun
            deriving (Eq,Ord,Show,Read)
```

This therefore means we can redefine `isWeekend` as follows:

```
isWeekend :: Day -> Bool
isWeekend day = day >= Sat
```

18.2.3 Product Types

As we saw in an earlier lecture, we can define student records (their name and a mark) as a type synonym giving a name to a tuple:

```
type StudentMark = (String, Int)
```

However, it is also possible for us to use an algebraic type with a constructor that has two arguments - one for the name and one for the mark:

```
data StudentMark = Student String int
```

Here, the constructor `Student` is followed by it's argument types. For example:

```
Student "Sam" 44
Student "Jill" 64
```

As could be expected, it is also possible to define functions which use the product data types:

```
betterStu :: StudentMark -> StudentMark -> String
betterStu (Student s1 m1) (Student s2 m2)
    | m1 >= m2  = s1
    | otherwise = s2
```

Which would be executed with:

```
ghci> betterStu (Student "Sam" 44) (Student "Jo" 73)
"Jo"
```

The main advantage of using an algebraic type is that every data value has an explicit label of its purpose (for example, `Student`)

18.2.4 Sum Types

It is possible for us to combine the ideas of enumerated types and product types, this gives us types whose elements can be built in different ways. For example, a shape might be specified by it's radius or a rectangle specified by its height and width. It is possible to represent this using a *sum* type:

```
data Shape = Circle Float |
             Rectangle Float Float
```

Which would be declared as:

```
Circle 9.0
Rectangle 4.5 6.0
```

This is a bit like an Abstract Class, where the parent class has no types of it's own and that the child classes are Concrete classes. `Shape` would be the parent, abstract class, and `Circle` & `Rectangle` would be the child, concrete classes.

As expected, it is possible to write a single function for a sum type (i.e `Shape`) which runs different code depending on the type of `Shape` used. This is done through Pattern Matching:

```
area :: Shape -> Float
area (Circle r)      = pi * r * r
area (Rectangle h w) = h * w
```

This can be extremely useful, for example in the problem of representing addresses. Some buildings have numbers, and some buildings have names. We can define a data type for representing the first line of an address as:

```
data Address = Address Building String
data Buildings = Number Int | Name String
```

Which we can use as:

```
Address (Number 42) "High Street"
Address (Name "Seaview") "Uplands Road"
```

18.2.5 Recursive Types
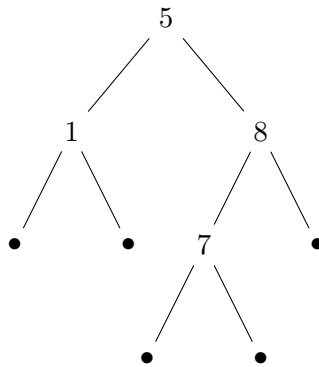
Figure 18.1: Binary Tree Example

Types can be described in terms of themselves. If we take the example of a binary tree, which we know is defined recursively as:

- a null node; or

- a node with a value, left sub-tree and a right-sub-tree

We can define a data type directly from this definition:

```
data Tree = Null |
            Node Int Tree Tree
```

Which we can see in-use below (note that the final value is represented by the diagram on the previous page):

```
Null
Node 7 Null Null
Node 5 (Node 1 Null Null)
       (Node 8 (Node 7 Null Null) Null)
```

It is possible to define functions which interact with recursive types. Generally, functions on trees will mirror the recursive structure of the type (meaning they use `Null` as the base case & `Node` for the recursion). For example, the following function returns the height of a binary tree:

```
height :: Tree -> Int
height Null = 0
height (Node _ st1 st2) = 1 + max (height st1) (height st2)
```

Page 19

Lecture - Input / Output

So far, our Haskell programs have been self-contained (meaning they have had no interaction with the user). This lecture will introduce the concept of handling input and output in a Haskell program.

19.1 Input & Output - Breaking Referential Transparency

As we have seen for many lectures leading to this point, the fundamental building block of functional programming is the function definitions. The most important property of a function, is that it will always give the same result when given the same arguments. If we take the following example:

```
square :: Int -> Int
square n = n * n
```

then if `x = y` then `square x = square y`. This property is known as *referential transparency*.

Referential Transparency allows us to mre easily reason about program code, both formally and informally. For example, if we take the following expression

```
e - e
```

then for any number, `e`, the expression will always evaluate to `0`. It is easy to prove a functional program as being correct, when compared to a imperative language using a loop.

This approach to I/O taken by some functional languages is to provide "functions" to read values from the keyboard and return the read value. For example the generic, non Haskell, function below:

```
inputInt :: Int
```

This approach breaks referential transparency, as we now do not know what the user has entered. The fact that `inputInt` will now return different values every time the program is run is due to the *side effects* of reading a new value from the keyboard.

19.2 Haskell's Approach to I/O

Since any function in a functional program might include an `inputInt`, the whole program becomes difficult to understand. For this reason - Haskell provides a different approach to input / output. Haskell's approach is known as the *monadic approach* since it is based on the mathematical concept of a monad. Input & Output is viewed as a sequence of actions (or programs) that happens in the specified sequence. Haskell provides the types

```
IO a
```

of I/O actions of type `a`.

A value of type `IO a` is an action which when executed:

- performs an input or output operation and then

- returns a value of type `a`

Haskell also provides a mechanism for sequencing actions, meaning actions can be sequenced to run one after the other. This can be seen to behave in a way similar to that of a simple imperative language. This means that typically programs in Haskell therefore comprise of some:

- function definitions (without any I/O)

- I/O programs

These imperative I/O programs are really an illusion, rather they are actually "syntactic sugar" for purely functional expressions.

19.2.1 Reading Input

The Prelude contains two functions for reading input, one for getting a string and one for getting a character.

```
getLine :: IO String
getChar :: IO Char
```

The first line above gets a string from the standard input and the second line above reads a single character form the standard input.

19.2.2 Writing Output

Unsurprisingly, performing output is different to retrieving input - as we do not expect output actions to return results. Despite this fact, Haskell I/O programs have to be of type `IO a` for some `a`. Haskell provides a one-element type called `()`, which contains the single value `()` - this is used to denote that a Haskell I/O program returns nothing of interest.

Haskell includes the function to print strings:

```
putStr :: String -> IO ()
```

Haskell also includes the function which appends a newline after the outputted string:

```
putStrLn :: String -> IO ()
```

For example - the Haskell "Hello, World!" program is:

```
main :: IO ()
main = purStrLn "Hello, World!"
```

This also illustrates the concept of a 'starting point' of a *complete* Haskell program: an action of type `IO ()` called `main`.

Haskell also provides a polymorphic function which displays data of any type that is an instance of the `Show` class (i.e. any value that can be converted to a string):

```
print :: Show a => a -> IO ()
print = putStrLn . show
```

19.3   The do notation

Now we have covered the basics of how input and output functions in Haskell, it's time to build a program with it! As part of this, we need to consider how to sequence I/O actions so that they complete in the correct order. Our example will consider:

1. Ask the user to enter any string

2. Reads in (but does not store) the string

3. Display a "done" message to the user

To ensure that these actions perform in the prescribed order, we must use the `do` notation:

```
readALine :: IO ()
readALine = do
    putStrLn "Enter a string"
    getLine
    putStrLn "Done"
```

Note that each line in the `do` consists of an action of type `IO a` for some `a`.

19.4   Capturing Inputted Values

As we are now well versed in the `getLine` expression - it's time to take it one step further, capturing them into a named value so we can use it in the future! This is done using the `<-` operator and would look something like this:

```
line <- getLine
```

This symbol might look, and operate, a bit like an assignment (commonly the `=` symbol) however it is not an assignment! This means what we can do with `line`, without doing something else to it first, is very limited.

19.4.1   Reading Integers

An excellent illustration of this behaviour is the case in which we are reading an integer from the Standard Input. In this example, we'll be writing our own action:

```
getInt :: IO Int
```

for reading an integer from the standard input.

To do this - we first need to use `getLine` to obtain a string from the standardInput:

```
do str <- getLine
```

We will then need to translate `str` into an integer using the `read` function, as declared in the `Read` type class:

```
read str :: Int
```

(This forces a conversion to a `Int` using `:: Int`)
Putting these stages together, we have the following function to obtain an Integer inputted from the Standard I/O

```
getInt :: IO Int
getInt = do
    str <- getLine
    return (read str :: Int)
```

19.4.2   Read

In the above example, we saw the `read` function be used to "read" the integer contents of the string returned from `getLine`. An example of this in action can be seen below:

```
read "  456      " :: Int
456
```

19.5   File I/O

The Prelude comes with built-in functions for reading from and writing to files:

```
readFile :: String -> IO String
writeFile :: String -> String -> IO ()
appendFile :: String -> String -> IO ()
```

The first argument of each function is the file's path; and the second is the contents (for `write` and `append`) only.

The example function below displays the files content on the screen:

```
displayFile :: IO ()
displayFile = do
    putStr "Enter the filename: "
    name <- getLine
    contents <- readFile name
    putStr contents
```

19.6   Conditionals in do

As we saw in week 1, Haskell includes a conditional operator. This can be used within a `do` construct. For example, in the following program - a string is read from the user and is then tested to see if it's a palindrome or not.

```
pal :: IO ()
pal = do
    str <- getLine
    if str == reverse str
        then putStr (str ++ " is a palindrome")
        else putStr (str ++ " is not a palindrome")
```

The test which is completed is of type `Bool`, and the branches are single actions. Note that if one of the branches had two or more actions then these would need to be combined into another `do` construct.

An alternative solution to the above program is to move more of the computation to a normal function:

```
isPalindrome :: String -> String
isPalindrome str
    | str == reverse str = str ++ " is a palindrome"
    | otherwise          = str ++ " is not a palindrome"
```

and then simplify the I/O program

```
pal :: IO ()
pal = do
    line <- getLine
    putStrLn (isPalindrome line)
```

This example illustrates a typical separation of the purely functional core of a program from its user interface code.

19.6.1   Local Definitions in I/O Programs

It would be possible to break the last line of `pal` into two parts (separating the computation from the output), as follows:

```
pal = do
    line <- getLine
    response <- return (isPalindrome line)
    putStrLn response
```

However the middle line is ugly, we have had to introduce some IO (through introducing `return`) just so we can use `<-`. There is a better way to do this with a local definition:

```
pal = do
    line <- getLine
    let response = isPalindrome line
    putSrtLn response
```

19.7   Recursion

Like functions, IO programs can be recursive. The following program allows the user to enter several lines, checking whether each one is a palindrome; it terminates when the user enters a blank line.

```
palLines :: IO ()
palLines = do
    putStr "Enter a line: "
    str <- getLine
    if str == "" then
        return ()
    else do
        putStrLn (isPalindrome str)
        palLines
```

Page 20

Lecture - Functional Programming in Python

📅 2024-04-22   🕐 12:00   🎓 Matthew

This lecture is for information only; the content will not be assessed.

In recent years, functional programming concepts have become increasingly common in mainstream imperative languages. These include languages such as: Python, JavaScript, C#, Java (from v8). This lecture will introduce the concept of Functional Programming in Python.

20.1 Functions are Data in Python

When defining a function in Python, we are in fact introducing a new variable with the function as it's value. This can be seen below:

```
def f(arg):
    return arg + 1
print(f) # prints <function f at 0xb56b11ec>
g = f
g(3) # returns 4
f = 7
f, g # returns (7, <function f at 0xb56b11ec>)
```

In the above example we can see the function, `f`, be defined then used. Then the data value of `f` (this is the function) gets assigned to `g` and we re-assign `f` to `7`. We finally return the values of `f` and `g` as a tuple.

20.2 List Comprehension

Python's list comprehension is very similar to that of Haskell's. They work by forming a new list from an old list.

```
>>> aList = [1, 2, 3, 4, 5]
>>> [2 * i for i in aList]
[2, 4, 6, 8, 10]
```

In the above example, we can see how we can use the keyword `for` and `in` to iterate through every item in the list and multiply it by 2. The below example takes this further, discarding all values where they are less than or equal to 3.

```
>>> [i * 2 for i in aList if i > 3]
[8, 10]
```

In Python, tuples are indexed like lists, however they are immutable. In the following example, note how you can iterate through a tuple containing list, using a `for` loop. This example is written in an imperative way.

```
testData = [("Sam",67), ("Kate",35), ("Jill",75),
            ("Fred", 45), ("Alice",50), ("Bob", 38)]
def passed(lst):
    names = []
    for (name, mark) in lst:
        if mark >= 40:
            names.append(name)
    return names
```

This is all well and good, however it's a bit too imperative - we want to make it more functional, as below:

```
def passed(lst):
    return [name for (name, mark) in lst if mark >= 40]
```

20.3   Lambda Expressions

Python includes, in a similar way to Haskell, anonymous functions - called lambda expressions; for example:

```
lambda x : x * 10
```

A lambda expression represents a function that multiplies its arguments by 10 (and is therefore similar to the `(*10)` in Haskell).

Python also supports higher-order functions, for example the following function returns a function which can then be called.

```
def multiplier(n):
    return lambda x: x * n
```

Which can therefore be used:

```
>>> m10 = multiplier(10)
>>> m10(6)
60
```

20.3.1   Higher Order Functions

We recall that Haskell has the higher order functions `map`, `filter` and `foldr`. Python includes built-in equivalents functions `map`, `filter` and `reduce` (which is almost equivalent to `foldr`).

Python's `map` function takes a function and a list and gives a new list:

```
>>> aList = [2, 3, 4, 9, 4, 7]
>>> map(lambda x : x * 10, aList)
[20, 30, 60, 90, 40, 70]
```

The `filter` function gives a new list of only those elements that have a given property:

```
>>> filter(lambda x : x % 2 == 0, aList)
[2, 6, 4]
```

The `reduce` function takes takes a binary function (which is a function of two arguments), and a list and then returns a single value. It applies the function to the first two elements of the list then on the result and the next item and so on.

```
>>> reduce(lambda x, y : x + y, aList)
31
>>> reduce(lambda x, y : x * y, aList)
9072
```

There is an optional third argument which can be passed as a 'starting' value:

```
>>> aList = []
>>> reduce(lambda x, y: x * y, aList, 1)
1
```

20.4   Benefits to Functions as Data

If we consider the following simple function which maps the number of days, based on the month number passed as parameter:

```
def daysInMonth(month):
    numDays =  [31,28,31,30,31,30,31,31,30,31,30,31]
    return numDays[month - 1]
```

We here have an *inefficiency* (no, not just the fact we're using Python), in that for every time the function is called - a new instance of `numDays` is stored in memory. So a possible solution to this problem could be to define `numDays` as a global variable:

```
numDays = [31,28,31,30,31,30,31,31,30,31,30,31]
def daysInMonth(month):
    return numDays[month - 1]
```

However, this is another bad idea - as it's using a global variable. Another potential solution could be to encapsulate it as a class and write the method as a static method, however this is also a bad use of an OO concept and is overly complicated.

So, instead we use a *closure*:

```
def makeDaysInMonth():
    numDays =  [31,28,31,30,31,30,31,31,30,31,30,31]
    def f(month):
        return numDays[month-1]
    return f
daysInMonth = makeDaysInMonth()
```

The returned function (which we've assigned to `daysInMonth`) has access to a local variable of another function that has exited.

20.5   Recursion

Recursion fanboys will argue that using recursion instead of iteration will lead to simpler, more readable functions. For example:

```
def fibonacci(n):
    if n < 2:
        return n
    else:
        return fibonacci(n-2) + fibonacci(n-1)
```

However, this function will suffer from a major issue of efficiency caused by the repeated computation of intermediate results. We can attempt to solve this using a higher-order function. Our solution will make use of a Python Dictionary.

20.5.1 Dictionaries

Dictionaries are unordered collections of data, whose values are indexed by key. Dictionary literals
are written as a sequence of `key:value` paris within braces (`{` and `}`). For example:

```
>>> shopping  = {"eggs" : 2, "ham" : 4}
```

Further notes on Dictionaries in Python are available in 1st Year Programming Module.

20.5.2 Memoization

Our solution to the recursive problem is to use a technique called memoization for remembering results
of calls to a function such as fibonacci. We can use a general purpose memoization function:

```
def memoize(f):
    cache = {}
    def g(arg):
        if arg in cache:
            return cache[arg]
        else:
            cache[arg] = f(arg)
            return cache[arg]
    return g
```

Which we can then give to the function we want to optimise:

```
fibonacci = memoize(fibonacci)
```