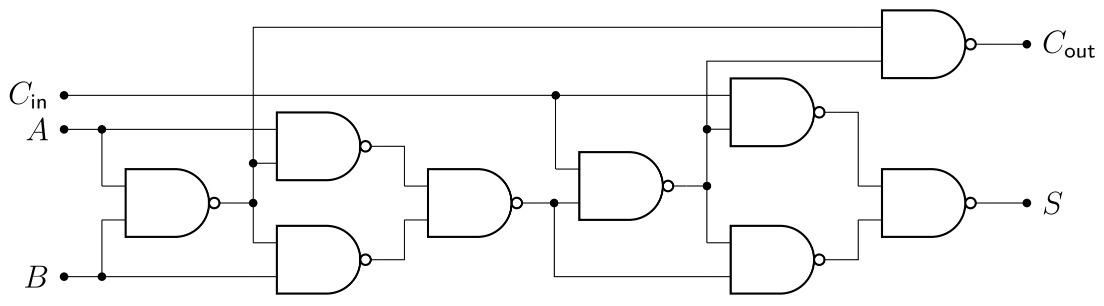
Manchester Metropolitan University

Computational Mathematics

Computing Mathematics Lecture Notes

Dr Jon Shiach

Spring 2022



© Department of Computing and Mathematics

Contents

This version compiled on March 9, 2022.

Preliminaries

0.1 Learning and Teaching

0.1.1 Lecture notes

These are the lecture notes that accompany the computing mathematics part of the unit Computational Mathematics with the other part covering computer graphics. Students are provided with a printed copy of the lecture notes so that they can focus on what is being covered in the lecture without having to worrying about making their own notes. There is also an electronic copy in PDF format which is available on the moodle area for this unit. These notes are quite comprehensive and are written specifically for this unit so should serve as your main point of reference. However, it is always advisable to seek out other sources of information either on the internet or better still textbooks from the library. Mathematical notation can differ between authors and these notes have been written to use notation that is most commonly found online.

These notes use a tried and tested format of definition (what is it that we are studying?) $\to$ explanation (why do we use it?) $\to$ examples (how is it used?) $\to$ exercises (now you try it). The examples in the printed version of the notes are left empty for students to complete in class. This is done because by writing out the steps used in a method it helps students to better understand that method. The PDF version of the notes contains the full solutions to the examples so if you do happen to miss an example you can complete the example by looking it up in the PDF version.

You should read through the lecture notes prior to attending the lectures that focus on that particular chapter. Don't worry about trying to understand everything when you first read through them. Reading mathematics is not like reading your favourite novel, it often requires repeated reading of a passage before you fully grasp the concepts that are being explained. In the lectures we will explain the various topics and provide more insight than what is written in the notes.

The PDF version of the notes contains lots of hyperlinks for easy navigation around the document. Hyperlinks show up as blue text and can save lots of time scrolling up and down the pages.

0.1.2 Unit timetable

The timetable for the computing mathematics part of the unit is shown below (this can also be accessed via MyMMU). The material will be covered in one 2-hour lecture and one 2-hour lab/tutorial.

- Lecture: Thursdays 12:00 - 14:00 in JD E239
- Lab: Thursdays 15:00 - 17:00 in JD C2.04

Students are expected to devote at least 30 hours per week to their studies and should complete all reading and tutorial exercises during this time.

0.1.3 Teaching Schedule

The computing mathematics material will be covered over the 6 weeks of the teaching block as outlined in table 1 so you should be aware of what is being covered and when. There may be times when we have to deviate from this slightly and you will be informed of this in the lecture.

Table 1: Computing mathematics teaching schedule

Week	Date (w/c)	Material
1	14/03/2022	
2	21/03/2022	**Coursework assignment handed out**
3	28/03/2022	
4	04/04/2022	
		Easter Break
5	25/04/2021	
6	02/05/2021	Revision. **Coursework assignment deadline – 6th May 2022**
7	02/05/2022	Assessment week
		Examination – 9th May to 10th May 2022

0.2 Assessment

The assessment for this unit takes the form of two coursework assignments and a take home examination.

- **Coursework (20%) – Computing Mathematics**
  - Released to students on **Monday 21st March 2022**;
  - Deadline is **9pm Friday 6th April 2022**;
- **Coursework (20%) – Computer Graphics**
  - Released to students on **Monday 21st March 2022**;
  - Deadline is **9pm Friday 6th May 2022**;
- **Examination (60%)**
  - Released to students at **09:00 on Tuesday 10th May 2022**;
  - Deadline is **12:30 on 9pm Friday 6th May 2022**;
  - Students will have access to the lecture notes, unit materials on moodle and any other sources of information available;
  - Total of 4 questions, 2 questions on computer graphics and 2 questions on computing mathematics.

You will receive a percentage mark for each assessment component and your overall mark for the unit will be calculated using the simple equation

$$\text{unit mark} = 0.2 \times \text{coursework 1 mark} + 0.2 \times \text{coursework 2 mark} + 0.6 \times \text{examination mark}.$$

To successfully pass the unit you will need a unit mark of at least 40%.

0.3 Advice to students

Some general advice to students:

- **Attend all of the classes** – the key to successfully passing the unit is to attend all of your classes. Mathematics is not a subject that can be learned easily in isolation just by reading the lecture notes. You will get much more out of the unit by attending, and more importantly, actively engaging in the classes.
- **Complete all of the exercises** – you would not expect an athlete to get faster or stronger without exercising and the same applies to studying mathematics. The tutorial exercises are designed to give you practice at using the various techniques and to help you to fully grasp the content. Try to make sure that you complete all of the exercises before the following week's lectures. Full worked solutions are provided in Appendix A at the end of these notes but do try to be disciplined and avoid looking up the answers before you have attempted the questions.
- **Catch up on missed work** – for whatever reason there may be a day when you cannot attend your classes, if this happens make sure you make the effort to catch up on missed work. Read through the appropriate chapter in the notes (as specified in the teaching schedule), complete the examples and attempt the tutorial exercises. It is very easy to start falling behind and the longer you leave it the more difficult it will be to catch up.
- **Start the coursework as soon as you are able** – the coursework is released to students early in the teaching block so you are not expected to be able to answer all of the questions right away. As we progress through the unit you will be told which questions you should now be able to answer. Try to start these questions as soon as you can and not leave it to the last minute.
- **Ask questions!** – perhaps the most important piece of advice here, there will be times when you are not quite sure about a concept, application or question. You can ask for help from the teaching staff and your fellow students. Mathematics is a hierarchical subject which means it requires full understanding at a fundamental level before moving onto more advanced topics, so if there are any gaps in your knowledge don't be afraid to ask questions.

Chapter 1

Mathematical Fundamentals

This chapter will provide a brief explanation of the fundamental mathematical concepts required for students to understand the material in the subsequent chapters.

1.1 Binary numbers

A **binary number** is a number that is represented using **base-2** number system where two symbols (usually 0 and 1) are used in combination with their position in a number to represent a value. Consider numbers expressed in **decimal** where the nine symbols 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 are used to represent a value. The digit on the far right-hand side of a number has the value of 1 and each digit to the left has the value 10 times that of the digit to the right. For example, the number 12345_{10}[1] in decimal has the value

$$12345_{10} = (1 \times 10000) + (2 \times 1000) + (3 \times 100) + (4 \times 10) + (5 \times 1),$$

or alternatively

$$12345_{10} = (1 \times 10^4) + (2 \times 10^3) + (3 \times 10^2) + (4 \times 10^1) + (5 \times 10^0).$$

Since the value of each digit is the base 10 raised to the power of n where n is the number of digits from the digit on the far right-hand side then decimal numbers are said to be expressed using **base-10**.

Since binary numbers are expressed using base-2 then the binary number 10110_2 has the value

$$\begin{aligned}
10110_2 &= (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \\
&= (1 \times 16) + (0 \times 8) + (1 \times 4) + (1 \times 2) + (0 \times 1) \\
&= 16 + 0 + 4 + 2 + 0 \\
&= 22.
\end{aligned}$$

Example 1.1

Convert following binary numbers to decimal:

(a) 11_2; (b) 1011_2; (c) 11000_2; (d) 101011101_2.

Solution:

(a) $11_2 = 2 + 1 = 3$;
(b) $1011_2 = 4 + 2 + 1 = 7$;
(c) $11000_2 = 8 + 4 = 12$;
(d) $101011101_2 = 256 + 64 + 16 + 8 + 4 + 1 = 349$.

[1]To avoid ambiguity the base of the number is expressed as a subscript.

Back to Table of Contents

1.1.1 Convert a decimal number to binary

To convert a decimal number to a binary number we use the following steps:

- divide the number by 2 and find the remainder;
- divide the result of the division by 2 and find the remainder;
- repeat the previous step until the result of the division is 0, the digits of the binary number are the remainders in reverse order.

For example, consider the binary representation of 22_{10}

$$\begin{aligned}
22 \div 2 &= 11 \text{ remainder } 0, \\
11 \div 2 &= 5 \text{ remainder } 1, \\
5 \div 2 &= 2 \text{ remainder } 1, \\
2 \div 2 &= 1 \text{ remainder } 0, \\
1 \div 2 &= 0 \text{ remainder } 1,
\end{aligned}$$

therefore $22_{10} = 10110_2$.

Example 1.2

Convert the following decimal numbers to binary:

(a) 10_{10}; (b) 100_{10}; (c) 256_{10};

Solution:

(a)

$$\begin{aligned}
10 \div 2 &= 5 \text{ remainder } 0, \\
5 \div 2 &= 2 \text{ remainder } 1, \\
2 \div 2 &= 1 \text{ remainder } 0, \\
1 \div 2 &= 0 \text{ remainder } 1, \\
\therefore 10_{10} &= 1010_2.
\end{aligned}$$

(b)

$$\begin{aligned}
100 \div 2 &= 50 \text{ remainder } 0, \\
50 \div 2 &= 25 \text{ remainder } 0, \\
25 \div 2 &= 12 \text{ remainder } 1, \\
12 \div 2 &= 6 \text{ remainder } 0, \\
6 \div 2 &= 3 \text{ remainder } 0, \\
3 \div 2 &= 1 \text{ remainder } 1, \\
1 \div 2 &= 0 \text{ remainder } 1, \\
\therefore 100_{10} &= 1100100_2.
\end{aligned}$$

(c)

$$\begin{aligned}
256 \div 2 &= 128 \text{ remainder } 0,\\
128 \div 2 &= 64 \text{ remainder } 0,\\
64 \div 2 &= 32 \text{ remainder } 0,\\
32 \div 2 &= 16 \text{ remainder } 0,\\
16 \div 2 &= 8 \text{ remainder } 0,\\
8 \div 2 &= 4 \text{ remainder } 0,\\
4 \div 2 &= 2 \text{ remainder } 0,\\
2 \div 2 &= 1 \text{ remainder } 0,\\
1 \div 2 &= 0 \text{ remainder } 1,\\
\therefore 256_{10} &= 100000000_2.
\end{aligned}$$

1.1.2 Binary addition

To add two binary numbers we add the last digits together. If the value of the sum is greater than 1 then we use the last digit of the sum and **carry** a 1 and include it in the sum of the digits to the left. This continues until all digits and carries have been summed. For example, consider the sum of the binary numbers $10110 + 1101$

$$\begin{array}{cccccc c}
 & & 1 & 0 & 1 & 1 & 0 \\
 & + & & 1 & 1 & 0 & 1 \\
\hline
\text{carry} & 1 & 1 & 1 & & & \\
\hline
\text{sum} & 1 & 0 & 0 & 0 & 1 & 1
\end{array}$$

Note that this sum is equivalent to $22_{10} + 13_{10} = 35_{10} = 100011_2$.

Example 1.3

Use binary addition to calculate the following:

(a) $101 + 11$; (b) $101010 + 11011$.

Solution:

(a)

$$\begin{array}{ccccc}
 & & 1 & 0 & 1 \\
 & + & & 1 & 1 \\
\hline
\text{carry} & 1 & 1 & 1 & \\
\hline
\text{sum} & 1 & 0 & 0 & 0
\end{array}$$

(b)

$$\begin{array}{cccccccc}
 & & 1 & 0 & 1 & 0 & 1 & 0 \\
 & + & & 1 & 1 & 0 & 1 & 1 \\
\hline
\text{carry} & 1 & 1 & 1 & & 1 & & \\
\hline
\text{sum} & 1 & 0 & 0 & 0 & 1 & 0 & 1
\end{array}$$

1.2 Hexadecimal numbers

An obvious drawback of using binary numbers is that the number of digits required to represent a value is more than when representing the value using decimal numbers. In computing, it is common to use **hexadecimal** numbers which are expressed using **base-16** . Since we require an additional six symbols to represent a single digit of a hexadecimal numbers we use the first six uppercase letters in the alphabet as shown in table 1.1.

Table 1.1: Letters used to represent hexadecimal digits.

Decimal	Hexadecimal
10	A
11	B
12	C
13	D
14	E
15	F

The techniques for converting between decimal and hexadecimal and addition of hexadecimal is the same as those for binary numbers. For example, the hexadecimal number $12AB_{16}$ has the decimal value

$$\begin{aligned}
12AB_{16} &= 1 \times 16^3 + 2 \times 16^2 + 10 \times 16^1 + 11 \times 16^0 \\
&= 1 \times 4096 + 2 \times 256 + 10 \times 16 + 11 \times 1 \\
&= 4096 + 512 + 160 + 11 \\
&= 4779.
\end{aligned}$$

Example 1.4

1. Convert 64087 to hexadecimal;
2. Convert FFF to decimal;
3. Evaluate $\mathrm{43ED} + \mathrm{F12B}$.

Solution:

1.

$$\begin{aligned}
64087 \div 16 &= 4005 \text{ remainder } 7, \\
4005 \div 16 &= 250 \text{ remainder } 5, \\
250 \div 16 &= 15 \text{ remainder } \mathrm{A}, \\
15 \div 16 &= 0 \text{ remainder } \mathrm{F}, \\
\therefore 64087_{10} &= \mathrm{FA57}_{16}.
\end{aligned}$$

2.

$$\begin{aligned}\mathrm{FFF} &= \mathrm{F}\times 16^2 + \mathrm{F}\times 16^1 + \mathrm{F}\times 16^0 \\ &= 15\times 256 + 15\times 16 + 15\times 1 \\ &= 3840 + 240 + 15 \\ &= 4095.\end{aligned}$$

3.

		4	3	E	D
	+	F	1	2	B
carry	1		1	1	
sum	1	3	5	1	8

1.3 Logic

Mathematical logic deals with statements that can be assigned a value of true or false but not both. Such statements are known as **propositions** propositions and the logic using propositions is often known as **propositional logic**. For example the statement "it is raining" is a proposition as it is either true or false whereas the statement "is it raining" is not a proposition as the answer may not be true or false.

In mathematical notation a true value is denoted using the symbol '1' and a false value is denoted using the symbol '0'[2]

1.3.1 Logical connectives

Individual propositions can be combined using what are known as **logical connectives** to form a **compound proposition**. For example the compound proposition "it is raining and I have an umbrella" combines the two propositions "it is raining" and "I have and umbrella" using the logical connective 'and'. A compound proposition, by definition, must have a value which is either true or false, in our example the compound proposition is only true if each of the propositions are both true.

In propositional logic we have three logical connectives: conjunction, disjunction and negation.

Definition 1.1: Logical conjunction

A **logical conjunction** (often referred to as logical AND) between two propositions p and q is true only if both p AND q are true. In symbolic notation the conjunction of p and q is written as $p \wedge q$.

Definition 1.2: Logical disjunction

A **logical disjunction** (often referred to as logical OR) between two propositions p and q is true if p is true OR q is true. In symbolic notation the disjunction of p and q is written as $p \vee q$.

[2]Some authors use T and F to denote true and false respectively.

Definition 1.3: Logical negation

A **logical negation** (often referred to as logical NOT) of the proposition p is true if p is false and false if p is true. In symbolic notation negation of p is written as $\neg p$.

Note that we assume an order of precedence negation $\longrightarrow$ conjunction $\longrightarrow$ disjunction, e.g., in the compound proposition $\neg p \wedge q \vee r$ we evaluate $\neg p$ first then $\neg p \wedge q$ before evaluating the disjunction with r.

1.3.2 Truth tables

A **truth table** is a table which expresses the values of a logical expression for all possible combination of values for the individual propositions. The truth table for the three logical connectives AND, OR and NOT is shown in table 1.2.

p	q	$p \wedge q$	$p \vee q$	$\neg p$	$\neg q$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

Table 1.2: A truth table for the logical connectives AND, OR and NOT.

Here the columns for p and q give all of the possible combinations of the values of the propositions. Since we have 2 propositions there will be a total of $2^2 = 4$ combinations. The order of which the combinations are written does not matter although it is recommended practice to write them in ascending order of their binary value for consistency.

1.3.3 Logical equivalence

Two logical statements are said to be **equivalent** if they have the same truth values for all possible combinations of the individual propositions. This is represented symbolically using $p \equiv q$. To check whether two compound propositions are equivalent we can write down the truth table for each one and check whether they have the same values for all of the combinations of their individual propositions. For example, consider the two compound propositions $\neg p \wedge \neg q$ and $\neg(p \vee q)$ which are represent in the truth table shown in table 1.3.

p	q	$\neg p$	$\neg q$	$\neg p \wedge \neg q$	$p \vee q$	$\neg(p \vee q)$
0	0	1	1	1	0	1
0	1	1	0	0	1	0
1	0	0	1	0	1	0
1	1	0	0	0	1	0

Table 1.3: Truth table for the compound propositions $\neg p \wedge \neg q$ and $\neg(p \vee q)$

Note that the columns for $\neg p \wedge \neg q$ and $\neg(p \vee q)$ are the same so we can conclude that $\neg p \wedge \neg q \equiv \neg(p \vee q)$.

Example 1.5

Use truth tables to confirm the following:

(a) $p \vee (p \wedge q) \equiv p$ (absorption law); (b) $p \wedge (\neg p \vee q) \equiv p \wedge q$;

Solution:

(a)

p	q	$p \wedge q$	$p \vee (p \wedge q)$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

(b)

p	q	$\neg p$	$\neg p \vee q$	$p \wedge (\neg p \vee q)$	$p \wedge q$
0	0	1	1	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	1	0	1	1	1

1.3.4 Laws of logic

The laws of logic are given in theorem 1.1 and can be used to find expressions that are equivalent to a compound proposition.

Theorem 1.1: Laws of logic

- Commutative law: $p \vee q \equiv p \vee q$ and $p \wedge q \equiv q \wedge p$;
- Associate law: $p \vee (q \vee r) \equiv (p \vee q) \vee r$ and $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$;
- Distributive law: $p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$ and $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$, $(p \vee q) \wedge (r \vee s) \equiv (p \wedge r) \vee (p \wedge s) \vee (q \wedge r) \vee (q \wedge s)$ and $(p \wedge q) \vee (r \wedge s) \equiv (p \vee r) \wedge (p \vee s) \wedge (q \vee r) \wedge (q \vee s)$;
- Identity law: $0 \vee p \equiv p$ and $1 \wedge p = p$;
- Idempotence law: $p \vee p \equiv p$; and $p \wedge p = p$;
- Absorption law: $p \wedge (p \vee q) \equiv p$ and $p \vee (p \wedge q) \equiv p$;
- De Morgan's law: $\neg(p \wedge q) \equiv \neg p \vee \neg q$ and $\neg(p \vee q) \equiv \neg p \wedge \neg q$;
- Complement law: $p \vee \neg p \equiv 1$ and $p \wedge \neg p \equiv 0$;
- Double negation: $\neg\neg p \equiv p$;

The definition of conjunction and disjunction also give the following results

$$0 \wedge p \equiv 0,$$
$$1 \vee p \equiv p.$$

For example, consider the compound proposition $(p \wedge q) \vee (\neg p \wedge q) \vee (p \wedge \neg q)$

$$\begin{aligned}
& (p \wedge q) \vee (\neg p \wedge q) \vee (p \wedge \neg q) \\
\equiv\ & (p \wedge q) \vee (p \wedge \neg q) \vee (\neg p \wedge q) && \text{(commutative law)} \\
\equiv\ & (p \wedge (q \vee \neg q)) \vee (\neg p \wedge q) && \text{(distributive law)} \\
\equiv\ & (p \wedge 1) \vee (\neg p \wedge q) && \text{(complement law)} \\
\equiv\ & p \vee (\neg p \vee q) && \text{(identity law)} \\
\equiv\ & (p \vee \neg p) \wedge (p \vee q) && \text{(distributive law)} \\
\equiv\ & 1 \wedge (p \vee q) && \text{(complement law)}
\end{aligned}$$

$$\equiv p \vee q \qquad \text{(identity law)}$$

We can confirm that $(p \wedge q) \vee (\neg p \wedge q) \vee (p \wedge \neg q) \equiv p \vee q$ using a truth table.

p	q	$\neg p$	$\neg q$	$p \wedge q$	$\neg p \wedge q$	$p \wedge \neg q$	$(p \wedge q) \vee (\neg p \wedge q) \vee (p \wedge \neg q)$	$p \vee q$
0	0	1	1	0	0	0	0	0
0	1	1	0	0	1	0	1	1
1	0	0	1	0	0	1	1	1
1	1	0	0	1	0	0	1	1

Example 1.6

Use the laws of logic to simplify the following compound propositions starting clearly which law you are using at each step:

(a) $\neg(\neg p \wedge \neg q)$; (b) $p \vee (q \wedge \neg p)$; (c) $\neg((p \vee \neg q) \vee (r \wedge (p \vee \neg q)))$.

Solution:

(a)

$$\begin{aligned}
\neg(\neg p \wedge \neg q) &\equiv \neg\neg p \vee \neg\neg q && \text{(De Morgan's law)} \\
&\equiv p \vee q && \text{(double negation law)}
\end{aligned}$$

(b)

$$\begin{aligned}
p \vee (q \wedge \neg p) &\equiv p \vee (\neg p \wedge q) && \text{(commutative law)} \\
&\equiv (p \vee \neg p) \wedge (p \vee q) && \text{(distributive law)} \\
&\equiv 1 \wedge (p \vee q) && \text{(complement law)} \\
&\equiv p \vee q && \text{(identity law)}
\end{aligned}$$

(c)

$$\begin{aligned}
\neg((p \vee \neg q) \vee (r \wedge (p \vee \neg q))) &\equiv \neg(p \vee \neg q) \wedge \neg(r \wedge (p \vee \neg q)) && \text{(De Morgan's law)} \\
&\equiv \neg(p \vee \neg q) \wedge (\neg r \vee \neg(p \vee \neg q)) && \text{(De Morgan's law)} \\
&\equiv \neg(p \vee \neg q) \wedge (\neg(p \vee \neg q) \vee \neg r) && \text{(commutative law)} \\
&\equiv \neg(p \vee \neg q) && \text{(absorption law)} \\
&\equiv \neg p \wedge \neg\neg q && \text{(De Morgan's law)} \\
&\equiv \neg p \wedge q && \text{(double negation law)}
\end{aligned}$$

1.4 Tutorial exercises

Exercise 1.1. Convert the following binary numbers to decimal:

(a) 1011; (b) 11010; (c) 1101010; (d) 10001001.

Exercise 1.2. Convert the following decimal numbers into binary:

(a) 9; (b) 23; (c) 80; (d) 163;

Exercise 1.3. Convert the following hexadecimal numbers to decimal:

(a) 10; (b) 16; (c) B14; (d) ABCD.

Exercise 1.4. Convert the following decimal numbers to hexadecimal

(a) 24; (b) 128; (c) 485; (d) 1435.

Exercise 1.5. Convert the number 1851 to base-3.

Exercise 1.6. Use binary addition to evaluate 101101 + 100111.

Exercise 1.7. Use hexadecimal addition to evaluate 1CE4F + 10A54.

Exercise 1.8. Write down the truth table for the following compound propositions:

(a) $\neg p \vee (p \wedge q)$; (b) $\neg(p \vee q) \wedge (p \vee \neg q)$; (c) $\neg(p \wedge \neg(q \vee \neg r))$.

Exercise 1.9. Use the laws of logic to simplify the compound propositions from exercise 1.8. State which law you are using at each step.

The solutions to these exercises are given in appendix A.1.

Chapter 2

Logic Circuits

Learning outcomes

On successful completion of this chapters students will be able to:

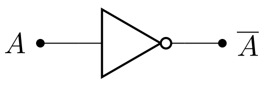
- identify the different logic gates used to construct logic circuits and draw their truth tables;
- draw circuit diagrams of logic circuits and write down the equivalent Boolean expression;
- simplify Boolean expressions using the laws of Boolean algebra;
- draw Karnaugh maps for a Boolean expression and use them to produce a Boolean expression in canonical disjunctive normal form;
- Construct the half adder and full adder circuits and use them to sum two binary numbers.

2.1 Logic gates

A computer processor is constructed using **logic gates** which perform an individual logical operation on one or more binary inputs. By connecting different logic gates in certain configurations we can build **logic circuits** that can perform arithmetic operations

2.1.1 NOT gate

A **NOT gate**, also known as an **inverter**, is a logic gate which negates the the input. It is represent mathematically using $\overline{A}$ and the symbol representing a NOT gate is shown in figure 2.1. The small circle in the output generally represents negation.

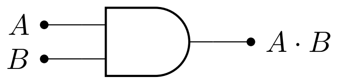


A	$\overline{A}$
0	1
1	0

Figure 2.1: Symbolic representation of a NOT gate and its truth table.

2.1.2 AND gate

An **AND gate** implements logical conjunction on two or more inputs. It is represent mathematically using $A \cdot B$ (often abbreviated to AB) and called the **product** of A and B. The symbol representing a AND gate is shown in figure 2.2.

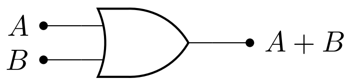


A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2.2: Symbolic representation of an AND gate and its truth table.

2.1.3 OR gate

An **OR gate** implements logical disjunction on two or more inputs. It is represent mathematically using $A + B$ and is called the **sum** of A and B. The symbol representing a OR gate is shown in figure 2.3.

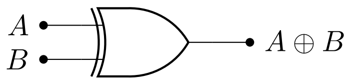


A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.3: Symbolic representation of an OR gate and its truth table.

2.1.4 XOR gate

An **XOR gate** (also known as an **exclusive OR gate**) implements a logical operation on two inputs that outputs a true value when there are an odd numbers of true inputs. It is represent mathematically using $A \oplus B$ which is equivalent to $A \cdot \overline{B} + \overline{A} \cdot B$ and its symbol and truth table are shown in figure 2.4

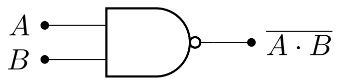


A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Figure 2.4: Symbolic representation of an XOR gate and its truth table.

2.1.5 NAND gate

A **NAND gate** implements a logical operation that is equivalent to the negation of a logical conjunction between two or more inputs (i.e., the output is false only if all of the inputs are true). It is represent mathematically using $\overline{A \cdot B}$ and its symbol and truth table are shown in figure 2.5.

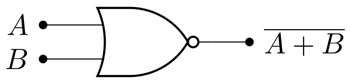


A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

Figure 2.5: Symbolic representation of an OR gate and its truth table.

2.1.6 NOR gate

A **NOR gate** implements a logical operation that is equivalent to the negation of a logical disjunction between two or more inputs (i.e., the output is true only if all of the inputs are false). It is represent mathematically using $\overline{A+B}$ and its symbol and truth table are shown in figure 2.6.



A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

Figure 2.6: Symbolic representation of an OR gate and its truth table.

2.2 Circuit diagrams

Logic gates are arranged to form **logic circuits** which are designed to perform logic operations on one or more **inputs** and produce one or more **outputs**. The configuration of logic gates, inputs and outputs are represented by a **circuit diagram**. In real circuits, the logic gates are connected using wires which are represented in circuit diagrams as lines. If a wire carries a charge above a certain current then it is deemed to have a value of 1, if it below this current then it is deemed to have a value of 0.

There are no rules to how circuit diagrams are drawn although is preferable that they are as simple and easy to read as possible. For this reason it is standard practice that the inputs are on the left and the outputs are on the right with the direction of flow going left-to-right. The connecting wires are usually draw using horizontal or vertical lines. Dots are used to represent the start or end point of a wire or where there is a split in the wires. Where two wires cross and there is no dot we assume that the two wires do not intersect.

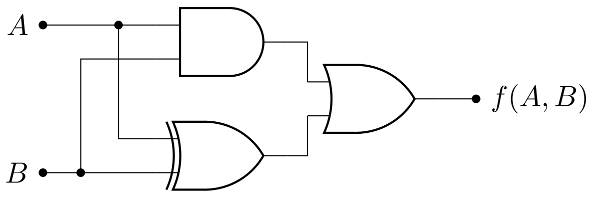


Figure 2.7: A simple circuit diagram.

Consider the circuit diagram shown in figure 2.7. Here we have two inputs, A and B, and the circuit produces a single output $f(A,B)$. The circuit consists of three logic gates: an AND gate, an XOR gate and an OR gate. The AND and XOR gates both has two inputs A and B, and the outputs from these gates are then inputted into the OR gate.

Note that the wire from input A is split before the AND gate so A is inputted into both the AND and the XOR gate (and similar for the input B). In the circuit diagram the A and B wires cross but since there is no dot we assume that they do not intersect.

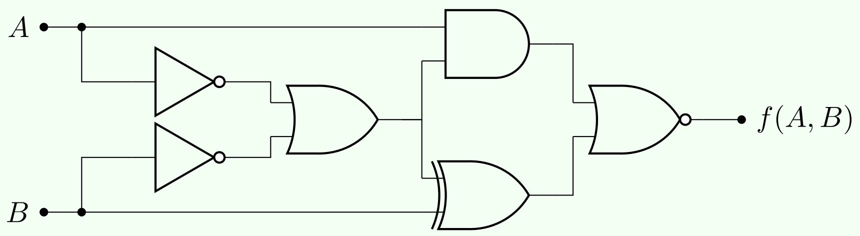
The mathematical expression that is equivalent to the circuit in figure 2.7 is

$$f(A,B) = A\cdot B + A\oplus B.$$

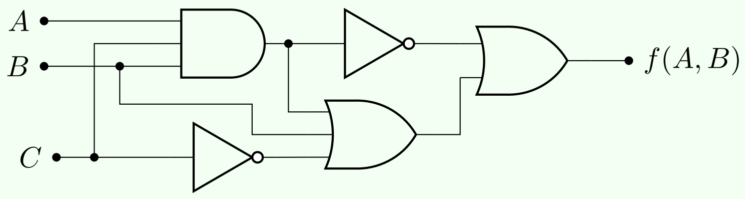
Example 2.1

For each circuit diagram below write down a mathematical expression that is equivalent to $f(A,B)$:

1.



2.



Solution:

1.

$$\begin{aligned} f(A,B) &= \overline{\text{AND gate} + \text{XOR gate}} \\ &= \overline{(A \cdot \text{OR gate}) + \text{OR gate} \oplus B} \\ &= \overline{A \cdot (\overline{A} + \overline{B}) + (\overline{A} + \overline{B}) \oplus B} \end{aligned}$$

2.

$$\begin{aligned} f(A,B) &= \overline{\text{AND gate}} + \text{OR gate} \\ &= \overline{A \cdot B \cdot C} + (B + \overline{C} + \text{AND gate}) \\ &= \overline{A \cdot B \cdot C} + (B + \overline{C} + A \cdot B \cdot C) \end{aligned}$$

2.3 Boolean algebra

When working with logic gates we can simplify circuits by utilising **Boolean algebra**. Boolean algebra involves variables that can only take the values of 1 and 0 (or true and false) and thus shares similarities with propositional logic. The laws of Boolean algebra are given in theorem 2.1 (which are essentially the same as those for propositional logic) and are used to find expressions that have a value that are equivalent with the purpose of either simplifying the original expression or using particular logic gates.

Not that we assume and order of precedence where conjunction takes precedence over disjunction, e.g., in $A + B \cdot C$ the conjunction $B \cdot C$ is evaluated before the disjunction of the result with A is evaluated.

Theorem 2.1: Laws of Boolean algebra

- Commutative law: $A + B \equiv B + A$ and $A \cdot B \equiv B \cdot A$;
- Associative law: $A + (B + C) \equiv (A + B) + C$ and $A \cdot (B \cdot C) \equiv (A \cdot B) \cdot C$;
- Distributive law: $A \cdot (B + C) = A \cdot B + A \cdot C$ and $A + B \cdot C \equiv (A + B) \cdot (A + C)$ and
- $(A + B) \cdot (C + D) \equiv A \cdot C + B \cdot C + A \cdot D + B \cdot D$;
- Identity law: $0 + A \equiv A$ and $1 \cdot A = A$;
- Idempotence law: $A + A \equiv A$; and $A \cdot A = A$;
- Absorption law: $A \cdot (A + B) \equiv A$ and $A + (A \cdot B) \equiv A$;
- De Morgan's law: $\overline{A \cdot B} \equiv \overline{A} + \overline{B}$ and $\overline{A + B} \equiv \overline{A} \cdot \overline{B}$;
- Complement law: $A + \overline{A} \equiv 1$ and $A \cdot \overline{A} \equiv 0$;
- Double negation: $\overline{\overline{A}} \equiv A$;

We can also make use of the following which come from the definition of AND and OR

$$
\begin{aligned}
0 \cdot A &\equiv 0, \\
1 + A &\equiv 1.
\end{aligned}
$$

2.4 Simplifying circuits

Boolean algebra can be used to simplify circuits so that fewer logic gates, or logic gates of a preferred type and to used to produce the same logical output. To do this we express the circuit as a Boolean equation using the logic gates that are in the circuit and apply the laws of Boolean algebra from theorem 2.1 to reduce or rewrite the Boolean equation in a form that is more preferable.

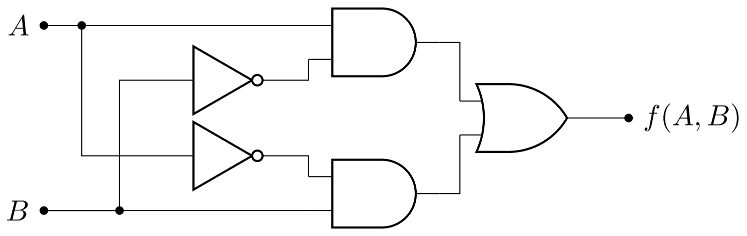


Figure 2.8: A logic circuit for an XOR gate.

For example, consider the logic circuit shown in figure 2.8. This circuit consists of 5 logic gates: 2 NOT gates, 2 AND gates and an OR gate. Writing this in Boolean algebra we have

$$f(A, B) = A \cdot \overline{B} + \overline{A} \cdot B.$$

The truth table for this circuit is shown in table 2.1 which shows that this circuit produces the same output as an XOR gate, i.e.,

$$A \cdot \overline{B} + \overline{A} \cdot B \equiv A \oplus B.$$

If we want to build an XOR gate using the fewest logic gates then we can apply the laws of Boolean algebra to simplify $A \cdot \overline{B} + \overline{A} \cdot B$. At first glance it isn't immediately obvious how we can simplify this expression, however if we let $C \equiv A \cdot \overline{B}$, $D \equiv \overline{A}$ and $E \equiv B$ then the Boolean expression is

$$A \cdot \overline{B} + \overline{A} \cdot B \equiv C + D \cdot E$$

and using the distributivity law on the right-hand side then

$$(C + D) \cdot (C + E) \equiv (A \cdot \overline{B} + \overline{A}) \cdot (A \cdot \overline{B} + B)$$

Table 2.1: Truth table for the circuit $A \cdot \overline{B} + \overline{A} \cdot B$.

A	B	$\overline{A}$	$\overline{B}$	$A \cdot \overline{B}$	$\overline{A} \cdot B$	$A \cdot \overline{B} + \overline{A} \cdot B$
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	0

so

$$A \cdot \overline{B} + \overline{A} \cdot B \equiv (\overline{A} \cdot B + A) \cdot (\overline{A} \cdot B + \overline{B}).$$

Using the distributivity law again for each of the two bracketed terms

$$(\overline{A} \cdot B + A) \cdot (\overline{A} \cdot B + \overline{B}) \equiv (A + \overline{A}) \cdot (A + B) \cdot (\overline{A} + \overline{B}) \cdot (B + \overline{B})$$

Here we can use the complement law $A + \overline{A} \equiv 1$

$$(A + \overline{A}) \cdot (A + B) \cdot (\overline{A} + \overline{B}) \cdot (B + \overline{B}) \equiv 1 \cdot (A + B) \cdot (\overline{A} + \overline{B}) \cdot 1$$

Using the identity law $1 \cdot A \equiv A$ then

$$1 \cdot (A + B) \cdot (\overline{A} + \overline{B}) \cdot 1 \equiv (A + B) \cdot (\overline{A} + \overline{B}).$$

Finally using De Morgan's law $\overline{A} + \overline{B} \equiv \overline{A \cdot B}$

$$(A + B) \cdot (\overline{A} + \overline{B}) \equiv (A + B) \cdot \overline{A \cdot B}.$$

The term $\overline{A \cdot B}$ is equivalent to a NAND gate so we now have a circuit with 3 logic gates: an OR gate, a NAND gate and an AND gate. The circuit diagram is shown in figure 2.9 and the truth table for the circuit is shown in table 2.2 showing that the output is equivalent to that of an XOR gate.

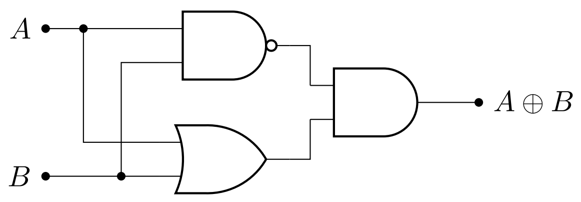


Figure 2.9: A simplified logic circuit for and XOR gate using 3 logic gates.

Table 2.2: Truth table for the circuit $(A + B) \cdot \overline{A \cdot B}$.

A	B	$A + B$	$A \cdot B$	$\overline{A \cdot B}$	$(A + B) \cdot \overline{A \cdot B}$
0	0	0	0	1	0
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	1	0	0

2.4.1 Propagation delay

In a physical circuit the speed current that flows along the wires and through the logic gates is affected by the length of the wire and the number of logic gates it passes through. The **propagation delay** is the delay in the speed of the current through a circuit. When considering the length of the connections in a computer chip the speed loss along a wire is negligible (approximately 1ns per 15cm of wire) so we use the number of logic gates as a measure of the propagation delay. Since there are multiple routes from the inputs to the outputs of a circuit we use the route with the most logic gates for the propagation delay.

Consider the circuit diagram shown in figure 2.7 with the routes from the inputs to the outputs traced in figure 2.10. Two of the routes pass through 2 logic gates and two of the routes pass through 3 logic gates so this circuit has a propagation delay of 3.

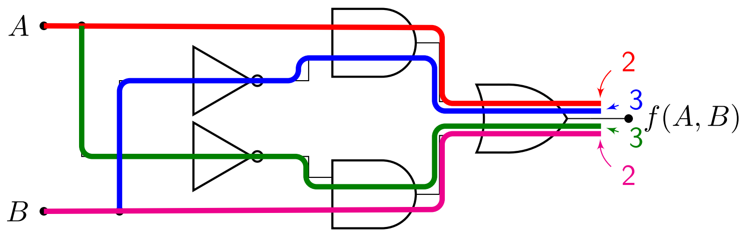
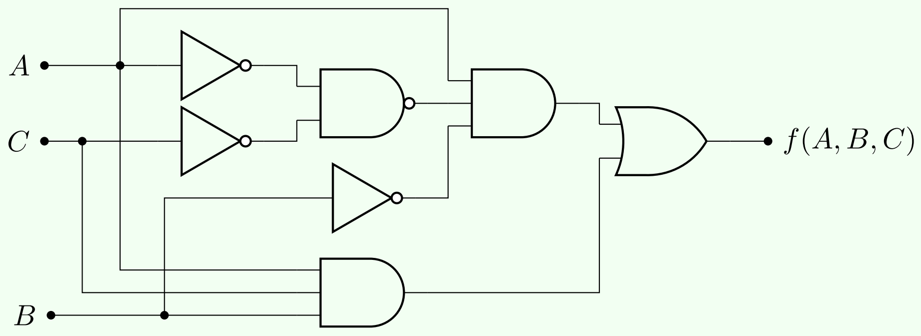


Figure 2.10: Tracing the propagation delay for the inputs to the outputs.

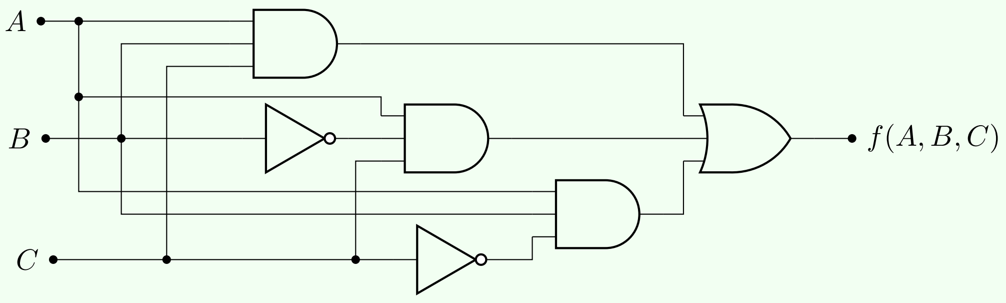
Example 2.2

For each of the circuits below (Tocci et al. 2007):

1.



2.



(a) write down the Boolean expression for the circuit;

(b) use the laws of Boolean algebra to simplify the Boolean expression;

(c) draw a circuit diagram of the simplified Boolean expression;

(d) construct a truth table for the simplified circuit.

Solution:

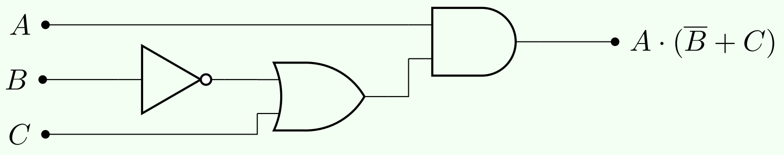
1. (a) $f(A,B,C) = A \cdot \overline{B} \cdot \overline{\overline{A} \cdot \overline{C}} + A \cdot B \cdot C$

   (b)

$$\begin{aligned} f(A,B,C) &= A \cdot \overline{B} \cdot \overline{\overline{A} \cdot \overline{C}} + A \cdot B \cdot C \\ &\equiv A \cdot \overline{B} \cdot (\overline{\overline{A}} + \overline{\overline{C}}) + A \cdot B \cdot C && \text{(De Morgan's law)} \end{aligned}$$

$$
\begin{aligned}
&\equiv A \cdot \overline{B} \cdot (A + C) + A \cdot B \cdot C && \text{(double negation)}\\
&\equiv A \cdot A \cdot \overline{B} + A \cdot \overline{B} \cdot C + A \cdot B \cdot C && \text{(distributivity law)}\\
&\equiv A \cdot \overline{B} + A \cdot \overline{B} \cdot C + A \cdot B \cdot C && \text{(idempotence law)}\\
&\equiv A \cdot \overline{B} + A \cdot C \cdot (\overline{B} + B) && \text{(distributivity law)}\\
&\equiv A \cdot \overline{B} + A \cdot C \cdot 1 && \text{(complement law)}\\
&\equiv A \cdot \overline{B} + A \cdot C && \text{(identity law)}\\
&\equiv A \cdot (\overline{B} + C) && \text{(distributivity law)}
\end{aligned}
$$

(c)



(d)

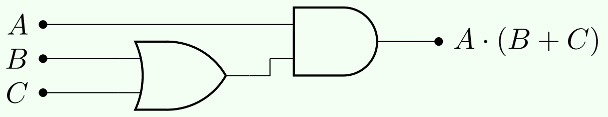
A	B	C	$\overline{B}$	$\overline{B} + C$	$A \cdot (\overline{B} + C)$
0	0	0	1	1	0
0	0	1	1	1	0
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	0	0	0
1	1	1	0	1	1

2. (a) $f(A, B, C) = A \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C}$

(b)

$$
\begin{aligned}
f(A, B, C) &= A \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C}\\
&\equiv A \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C && \text{(idempotence law)}\\
&\equiv A \cdot C \cdot (B + \overline{B}) + A \cdot B \cdot (C + \overline{C}) && \text{(distributivity law)}\\
&\equiv A \cdot C \cdot 1 + A \cdot B \cdot 1 && \text{(complement law)}\\
&\equiv A \cdot C + A \cdot B && \text{(identity law)}\\
&\equiv A \cdot (B + C) && \text{(distributivity law)}
\end{aligned}
$$

(c)



(d)

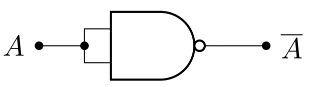
A	B	C	$B+C$	$A\cdot(B+C)$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

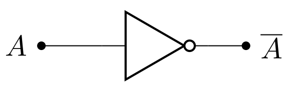
2.5 Universal gates

NAND gates and NOR gates are known as **universal gates** because they can be implemented to represent any other Boolean operation using the same logic gate. They are also more straight forward and cheaper to build so if we can construct a logic circuit using just NAND gates then we can reduce the costs and time taken to manufacture a computing chip.

To prove that the NAND gate is a universal gate we need to show that it can be used to represent the other types of logic gates.

- NOT gate: $\overline{A} \equiv \overline{A\cdot A}$



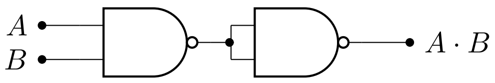
equivalent to


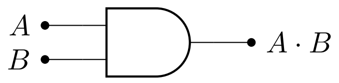
Proof.

$$\overline{A\cdot A} \equiv \overline{A} \qquad \text{(idempotence law)}$$

□

- AND gate: $A\cdot B \equiv \overline{\overline{A\cdot B}\cdot\overline{A\cdot B}}$



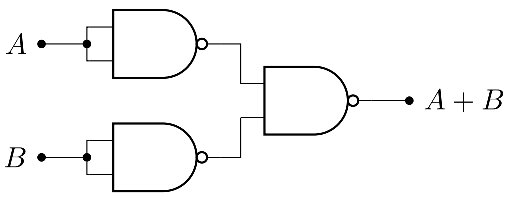
equivalent to


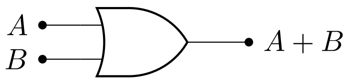
Proof.

$$\begin{aligned}\overline{\overline{A\cdot B}\cdot\overline{A\cdot B}} &\equiv \overline{\overline{A\cdot B}} && \text{(idempotence law)}\\ &\equiv A\cdot B && \text{(double negation)}\end{aligned}$$

□

- OR gate: $A+B \equiv \overline{\overline{A\cdot A}\cdot\overline{B\cdot B}}$



equivalent to


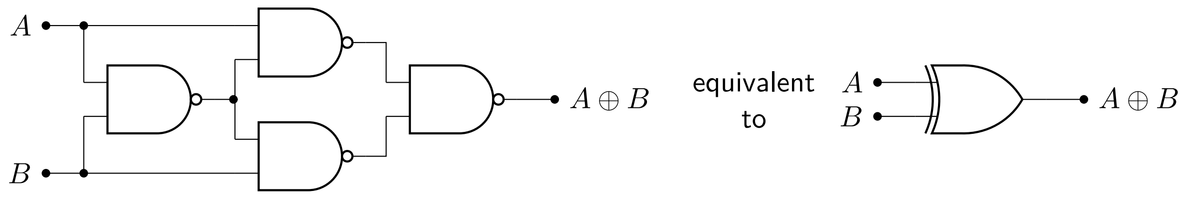
Proof.

$$\begin{aligned}\overline{\overline{A\cdot A}\cdot\overline{B\cdot B}} &\equiv \overline{\overline{A}\cdot\overline{B}} && \text{(idempotence law)}\\ &\equiv \overline{\overline{A}}+\overline{\overline{B}} && \text{(De Morgan's law)}\end{aligned}$$

$$\equiv A + B \qquad \text{(double negation)}$$

□

- XOR gate: $A \oplus B \equiv \overline{\overline{A\cdot\overline{A\cdot B}}\cdot\overline{B\cdot\overline{A\cdot B}}}$

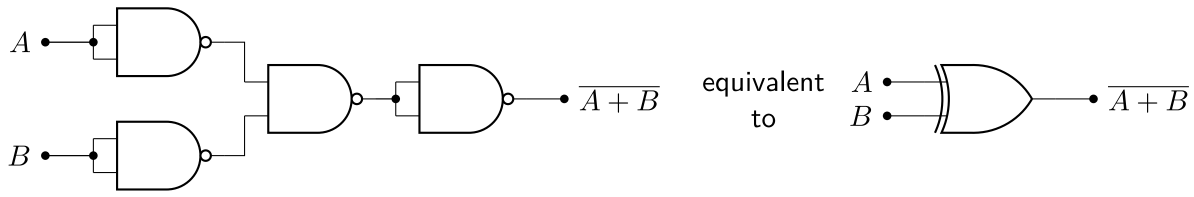


Proof.

$$\begin{aligned}
\overline{\overline{A\cdot\overline{A\cdot B}}\cdot\overline{B\cdot\overline{A\cdot B}}} &\equiv \overline{\overline{A\cdot\overline{A\cdot B}}} + \overline{\overline{B\cdot\overline{A\cdot B}}} && \text{(De Morgan's law)}\\
&\equiv A\cdot\overline{A\cdot B} + B\cdot\overline{A\cdot B} && \text{(double negation)}\\
&\equiv A\cdot(\overline{A}+\overline{B}) + B(\overline{A}+\overline{B}) && \text{(De Morgan's law)}\\
&\equiv A\cdot\overline{A} + A\cdot\overline{B} + B\cdot\overline{A} + B\cdot\overline{B} && \text{(distributivity law)}\\
&\equiv 0 + A\cdot\overline{B} + \overline{A}\cdot B + 0 && \text{(complement law)}\\
&\equiv A\cdot\overline{B} + \overline{A}\cdot B && \text{(identity law)}\\
&\equiv A\oplus B. && \text{(definition of } A\oplus B\text{)}
\end{aligned}$$

□

- NOR gate: $\overline{A+B} \equiv \overline{\overline{\overline{A\cdot A}\cdot\overline{B\cdot B}}\cdot\overline{\overline{A\cdot A}\cdot\overline{B\cdot B}}}$



Proof.

$$\begin{aligned}
\overline{\overline{\overline{A\cdot A}\cdot\overline{B\cdot B}}\cdot\overline{\overline{A\cdot A}\cdot\overline{B\cdot B}}} &\equiv \overline{\overline{\overline{A}\cdot\overline{B}}\cdot\overline{\overline{A}\cdot\overline{B}}} && \text{(idempotence law)}\\
&\equiv \overline{\overline{\overline{A}\cdot\overline{B}}} && \text{(idempotence law)}\\
&\equiv \overline{A}\cdot\overline{B} && \text{(double negation)}\\
&\equiv \overline{A+B} && \text{De Morgan's law}
\end{aligned}$$

□

A similar approach can be used to prove the NOR gate is also a universal gate.

2.6 Canonical normal form

A Boolean expression with inputs $A_1, A_2, \ldots, A_n$ can be represented in **Canonical Disjunctive Normal Form (CDNF)** where a sum of terms where each term is a product of the inputs. So CDNF is also known as **Sum Of Products (SOP)** form. For example, the expression

$$f(A,B,C) = A\cdot B\cdot C + \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C}, \tag{2.1}$$

is in SOP form.

Alternatively we can represent an expression in **Canonical Conjunctive Normal Form (CCNF)** where a product of terms where each term is a sum of the inputs. So CCNF is also known as **Product Of Sums (POS)** form. For example, the expression

$$f(A,B,C) = (A+\overline{B}+\overline{C})\cdot(\overline{A}+\overline{B}+C)\cdot(A+\overline{B}+C)\cdot(A+B+\overline{C}), \tag{2.2}$$

is in POS form.

2.6.1 Minterms

A **minterm** is a product of the inputs with the condition that each input appears once, either in its uncomplemented or complemented form. A Boolean expression in SOP form is a sum of minterms, i.e.,

$$f(A_1, A_2, \ldots, A_n) = m_1 + m_2 + \cdots + m_p,$$

where m_i are the minterms and $A_1, A_2, \ldots, A_n$ are inputs. For example, the Boolean expression in equation (2.1) has four minterms $A\cdot B\cdot C$, $\overline{A}\cdot B\cdot C$, $A\cdot\overline{B}\cdot C$ and $\overline{A}\cdot B\cdot\overline{C}$. Minterms can also be thought of as the product of the inputs where the truth table for $f(A_1, A_2, \ldots, A_n)$ has a value of 1.

We can use minterms to represent a Boolean expression in a compact form. We determine a minterm index, i, for the minterm $m_i = A_1\cdot A_2\cdot\ldots\cdot A_n$ which is a decimal number by assigning a value of 1 for the uncomplemented input A or 0 for the complemented input 1. The values in a minterm are concatenated to give a binary number which is converted to a decimal which is the minterm index. For example, for the minterm $\overline{A}\cdot B\cdot C$ we have

$$\overline{A}\cdot B\cdot C \equiv m_{011} \equiv m_3.$$

A Boolean expression can then be represented by

$$f(A_1, A_2, \ldots, A_n) = \sum_{i=1}^{p} m_i,$$

where p is the number of minterms and the summation operator $\sum$ represents a sequence of disjunction operations.

For example, for the Boolean expression in equation (2.1)

$$\begin{aligned} f(A,B,C) &= A\cdot B\cdot C + \overline{A}\cdot B\cdot C + A\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C} \\ &\equiv m_{111} + m_{011} + m_{101} + m_{010} \\ &\equiv m_7 + m_3 + m_5 + m_2, \end{aligned}$$

so $f(A,B,C) = \displaystyle\sum_{i=2,3,5,7} m_i$.

2.6.2 Maxterms

A **maxterm** is similar to a minterm but for expressions in canonical conjunctive normal form, i.e., in the expression

$$f(A_1, A_2, \ldots, A_n) = M_1\cdot M_2\cdot\ldots\cdot M_p,$$

M_i are the maxterms. For example, the expression in equation (2.2) has the four maxterms $A+\overline{B}+\overline{C}$, $\overline{A}+\overline{B}+C$, $A+\overline{B}+C$ and $A+B+\overline{C}$. Maxterms can also be thought of as the sum of the inputs where the truth table for $f(A_1, A_2, \ldots, A_n)$ has a value of 0.

Similar to minterms, we can assign an index, i, for the maxterm $M_i = A_1 + A_2 + \cdots + A_n$. For example, for the maxterm $A+\overline{B}+C$ we have

$$A+\overline{B}+C \equiv M_{101} \equiv M_5.$$

A Boolean expression can then be represented using

$$f(A_1, A_2, \dots, A_n) = \prod_{i=1}^{p} M_i,$$

where p is the number of maxterms and the product operator $\prod$ represents a sequence of conjunction operations.

For example, for the Boolean expression in equation (2.2)

$$\begin{aligned} f(A,B,C) &= (A+\overline{B}+\overline{C})\cdot(\overline{A}+\overline{B}+C)\cdot(A+\overline{B}+C)\cdot(A+B+\overline{C}) \\ &\equiv M_{100}\cdot M_{001}\cdot M_{101}\cdot M_{110} \\ &\equiv M_4\cdot M_1\cdot M_5\cdot M_6, \end{aligned}$$

so $f(A,B,C) \equiv \displaystyle\prod_{i=1,4,5,6} M_i$.

2.6.3 Writing Boolean expressions using minterms

Any Boolean expression can be expressed using minterms by:

1. use the laws of Boolean algebra to remove brackets and negation of disjunctive and conjunctive (OR and AND) operations;
2. for each term not containing all inputs, append a conjunction (AND) with the disjunction with the missing input and its complement (e.g., $A \equiv A\cdot(B+\overline{B})$);
3. use the distributivity law to expand out the brackets;
4. remove any duplicate minterms.

For example, consider the expression $A\cdot B\cdot\overline{C}+A\cdot B+\overline{A}\cdot C$. Here the input C is missing from the second term and B is missing from the third term, so

$$\begin{aligned} A\cdot B\cdot\overline{C}+A\cdot B+\overline{A}+C &\equiv A\cdot B\cdot\overline{C}+A\cdot B\cdot(C+\overline{C})+\overline{A}\cdot C\cdot(B+\overline{B}) \\ &\equiv A\cdot B\cdot\overline{C}+A\cdot B\cdot C+A\cdot B\cdot\overline{C}+\overline{A}\cdot B\cdot C+\overline{A}\cdot\overline{B}\cdot C \\ &\equiv A\cdot B\cdot\overline{C}+A\cdot B\cdot C+\overline{A}\cdot B\cdot C+\overline{A}\cdot\overline{B}\cdot C \\ &\equiv m_{110}+m_{111}+m_{011}+m_{001}, \\ &\equiv \sum_{i=1,3,6,7} m_i. \end{aligned}$$

2.6.4 Converting between SOP and POS forms

Consider the truth table for the expression in SOP form: $f(A,B,C)=A\cdot B\cdot C+\overline{A}\cdot B\cdot C+A\cdot\overline{B}\cdot C+\overline{A}\cdot B\cdot\overline{C}$,

A	B	C	$f(A,B,C)$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Table 2.3: Truth table for the function $f(A,B,C)=A\cdot B\cdot C+\overline{A}\cdot B\cdot C+A\cdot\overline{B}\cdot C+\overline{A}\cdot B\cdot\overline{C}$.

The minterms are the product of the inputs where $f(A,B,C)=1$ and the maxterms are the sum of the inputs where $f(A,B,C)=0$. So to find an expression in POS form that is equivalent to $f(A,B,C)$ we find the maxterms and invert each input.

To convert from SOP to POS:

1. find the minterms, m_i;
2. identify the maxterm indices by removing the minterm indices from $\{0,1,\dots,2^n-1\}$;
3. convert maxterm indices to binary, find the complement and convert to decimal j;
4. change the $\sum$ to $\prod$ and m_i to M_j;

For example, consider the SOP expression

$$f(A,B,C)=A\cdot B\cdot C+\overline{A}\cdot B\cdot C+A\cdot\overline{B}\cdot C+\overline{A}\cdot B\cdot\overline{C}.$$

Expressing $f(A,B,C)$ using the minterms gives

$$\begin{aligned}f(A,B,C)&\equiv m_{111}+m_{011}+m_{101}+m_{110}\\&\equiv m_7+m_3+m_5+m_2\end{aligned}$$

Since there are 3 inputs so the set of all possible minterms is $\{0,1,2,3,4,5,6,7\}$ so the maxterms are M_0, M_1, M_4 and M_6.

$$\begin{aligned}f(A,B,C)&\equiv M_{\overline{000}}\cdot M_{\overline{001}}\cdot M_{\overline{010}}\cdot M_{\overline{110}}\\&\equiv M_{111}\cdot M_{110}\cdot M_{101}\cdot M_{001}\\&\equiv(A+B+C)\cdot(A+B+\overline{C})\cdot(A+\overline{B}+C)\cdot(\overline{A}+\overline{B}+C).\end{aligned}$$

To convert from POS to SOP we use the same steps for conversion between SOP to POS with the exception that we change $\prod$ to $\sum$. For example, consider following expression in POS form

$$f(A,B,C)=(A+\overline{B}+\overline{C})\cdot(\overline{A}+\overline{B}+C)\cdot(A+\overline{B}+C)\cdot(A+B+\overline{C})$$

Expressing $f(A,B,C)$ using the maxterms gives

$$\begin{aligned}f(A,B,C)&\equiv M_{100}\cdot M_{001}\cdot M_{101}\cdot M_{110}\\&\equiv M_4\cdot M_1\cdot M_5\cdot M_6.\end{aligned}$$

Here the minterms are m_0, m_2, m_3 and m_7 so

$$\begin{aligned}f(A,B,C)&\equiv m_{\overline{000}}+m_{\overline{010}}+m_{\overline{011}}+m_{\overline{111}}\\&\equiv m_{111}+m_{101}+m_{100}+m_{000}\\&\equiv A\cdot B\cdot C+A\cdot\overline{B}\cdot C+A\cdot\overline{B}\cdot\overline{C}+\overline{A}\cdot\overline{B}\cdot\overline{C}.\end{aligned}$$

Example 2.3

For each Boolean expression below write it in SOP and POS forms.

1. $f(A,B)=\overline{A}\cdot(A+\overline{B})$;
2. $f(A,B,C)=(A+C)\cdot(\overline{A}+\overline{B})$.

Solution:

1.

$$\begin{aligned} f(A,B) &\equiv \overline{A}\cdot(A+\overline{B}) \\ &\equiv \overline{A}\cdot A + \overline{A}\cdot B \\ &\equiv \overline{A}\cdot B \end{aligned}$$

Expressing $f(A,B)$ using minterms gives

$$f(A,B) \equiv m_{00} = m_0$$

so the maxterms are M_1, M_2 and M_3 therefore

$$\begin{aligned} f(A,B) &\equiv M_{\overline{01}}\cdot M_{\overline{10}}\cdot M_{\overline{11}} \\ &\equiv M_{10}\cdot M_{01}\cdot M_{00} \\ &\equiv (A+\overline{B})\cdot(\overline{A}+B)\cdot(\overline{A}+\overline{B}). \end{aligned}$$

2.

$$\begin{aligned} f(A,B,C) &= (A+C)\cdot(\overline{A}+\overline{B}) \\ &\equiv A\cdot\overline{A}+A\cdot\overline{B}+\overline{A}\cdot C+\overline{B}\cdot C \\ &\equiv A\cdot\overline{B}+\overline{A}\cdot C+\overline{B}\cdot C \\ &\equiv A\cdot B\cdot(C+\overline{C})+\overline{A}\cdot C\cdot(B+\overline{B})+\overline{B}\cdot C\cdot(A+\overline{A}) \\ &\equiv A\cdot B\cdot C+A\cdot B\cdot\overline{C}+\overline{A}\cdot B\cdot C+\overline{A}\cdot\overline{B}\cdot C+A\cdot\overline{B}\cdot C+\overline{A}\cdot\overline{B}\cdot C \\ &\equiv A\cdot B\cdot C+A\cdot B\cdot\overline{C}+\overline{A}\cdot B\cdot C+\overline{A}\cdot\overline{B}\cdot C+A\cdot\overline{B}\cdot C \end{aligned}$$

Expressing $f(A,B,C)$ using minterms we have

$$\begin{aligned} f(A,B,C) &\equiv m_{111}+m_{110}+m_{011}+m_{001}+m_{101} \\ &\equiv m_7+m_6+m_3+m_1+m_5 \end{aligned}$$

so the maxterms are M_0, M_2 and M_4 therefore

$$\begin{aligned} f(A,B,C) &\equiv M_{\overline{000}}\cdot M_{\overline{010}}\cdot M_{\overline{100}} \\ &\equiv M_{111}\cdot M_{101}\cdot M_{011} \\ &\equiv (A+B+C)\cdot(A+\overline{B}+C)\cdot(\overline{A}+B+C). \end{aligned}$$

2.7 Karnaugh maps

Simplifying Boolean expressions can often be a long winded procedure requiring knowing all of the laws of Boolean algebra. **Karnaugh maps** (also known as **K-maps**) were introduced by American mathematician Maurice Karnaugh in 1953 and are a useful tool for simplifying Boolean expressions since it uses our inherent pattern recognition ability.

Consider the truth table of the Boolean expression $A+B$ shown in table 2.4. We can represent this as a Karnaugh map by forming a grid with the inputs represented by the rows and columns of the grid

Table 2.4: Truth table for $A + B$.

A	B	$A+B$
0	0	0
0	1	1
1	0	1
1	1	1

(figure 2.11). The outputs of the Boolean expression are represented by 0s and 1s in the Karnaugh map corresponding to their inputs.

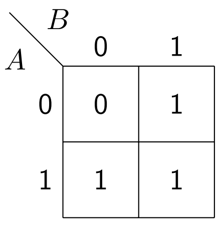


Figure 2.11: Karnaugh map for $A + B$.

2.7.1 Finding the Boolean expression from a Karnaugh map

When presented with a Karnaugh map we can establish the Boolean expression that it represents by grouping grid cells that contain adjacent 1s that form a rectangle. Each group of 1s corresponds to a minterm of the Boolean expression. For example, consider the Karnaugh map in figure 2.12 where the 1s in the second vertical column and the 1s in the second horizontal row of the Karnaugh map for $A + B$ (figure 2.11) have been grouped. The rules for groupings in Karnaugh maps are given in section 2.7.2.

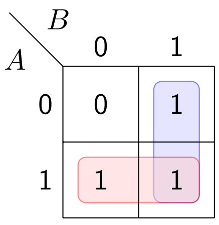


Figure 2.12: The Karnaugh map for $A + B$ with the 1s grouped.

Taking the vertical column (blue) in figure 2.12 to start with here we see that the the input value A is 1 for both the elements in the group and the input values B is both 0 and 1. What this means is that this group represents the minterm A since it is alway true.

Now looking at the horizontal row (red) we see that the input value A is both 0 and 1 whilst the input value B is 1 for both of the 1s in the group. Therefore this group represents the minterm B.

Since each grouping of a Karnaugh map represents a minterm for the Boolean expression it represents then the expression for this Karnaugh maps is $A + B$ as expected..

Alternatively, groups of 0s in a Karnaugh map corresponds to the maxterms in a Boolean expression which is the complement of the expression that the Karnaugh map represents $\overline{f(A_1, A_2, \ldots, A_n)}$. The Karnaugh map in figure 2.13 is the Karnaugh map for $A+B$ (figure 2.11) with the 0s grouped. This group corresponds to the maxterm $\overline{A} \cdot \overline{B}$ which is equivalent $\overline{A + B}$, i.e., the complement of $A + B$.

2.7.2 Rules of Karnaugh maps

When implementing Karnaugh maps we have to abide by the following rules (Belton 1998):

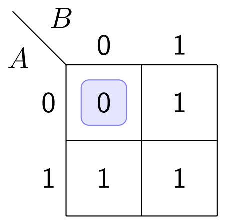
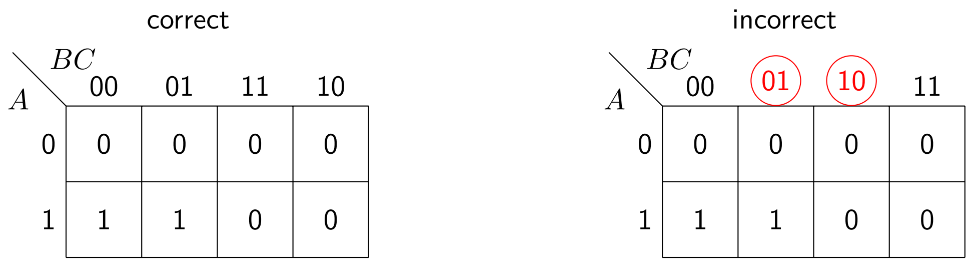
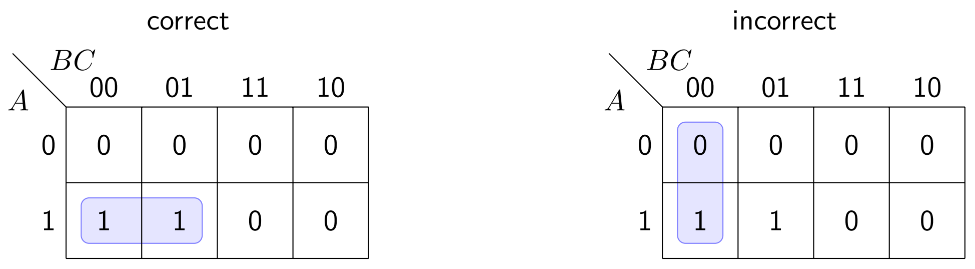


Figure 2.13: The Karnaugh map for $A + B$ with the 0s grouped.

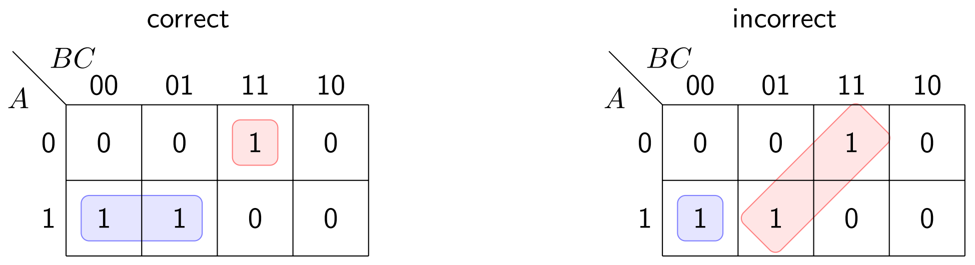
- Input numbering can only change by a single bit from one row or column to the next.



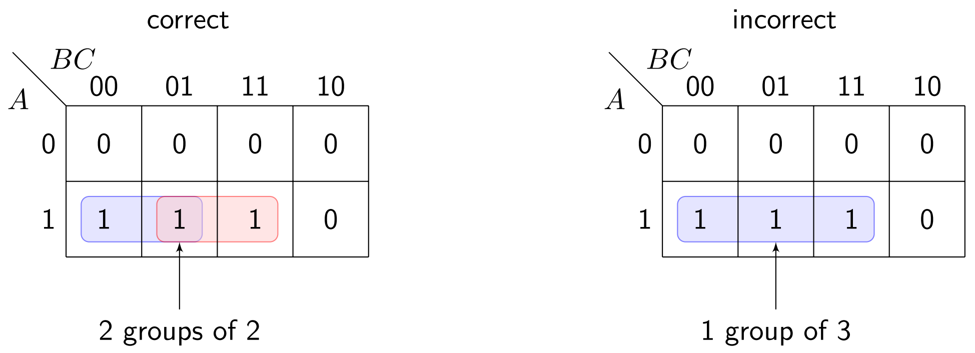
- Groups may not include any cell containing a zero



- Groups can be horizontal or vertical but not diagonal.

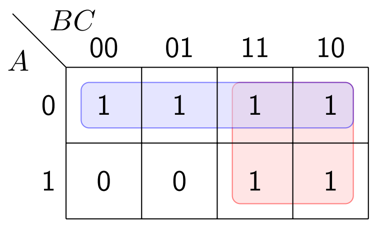


- The number of cells in a group must be 2^n, i.e., 1, 2, 4, 8, . . .

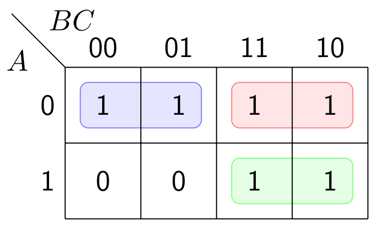


- Each group should be as large as possible (not doing this will not break the laws of Boolean algebra but it wouldn't result in the simplest form).

correct

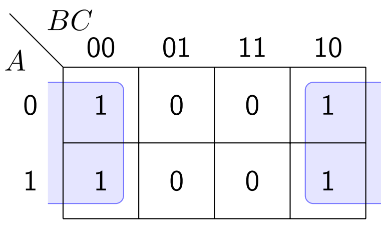


incorrect

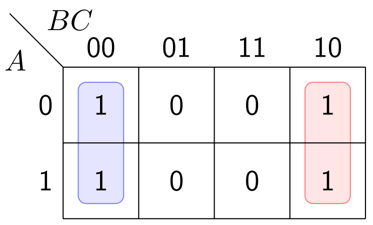


- Groups may wrap around the edges of the Karnaugh map. The leftmost column may be grouped with the rightmost column and the top row may be grouped with the bottom row.

correct



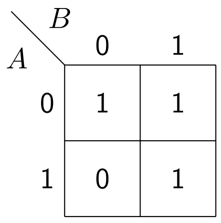
incorrect



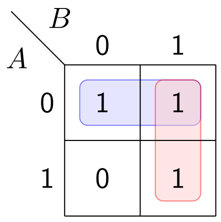
- Each cell containing a 1 must be in at least one group.
- Groups may overlap.
- There should be as few groups as possible as long as the previous rules are not contradicted.

2.7.3 Simplify Boolean expressions using Karnaugh maps

Consider the Boolean expression $\overline{A} \cdot \overline{B} + \overline{A} \cdot B + A \cdot B$. The minterms are $\overline{A} \cdot \overline{B}$, $\overline{A} \cdot B$ and $A \cdot B$ so the Karnaugh maps has 1s where $A = B = 0$, $A = 0$ and $B = 1$ and $A = B = 1$.



We can form two groups:



The horizontal (blue) group corresponds to the minterm $\overline{A}$ and the vertical (red) group corresponds to the minterm B so this Karnaugh map represents $\overline{A} + B$. If we simplify the original expression using the laws of Boolean algebra

$$\begin{aligned}
\overline{A} \cdot \overline{B} + \overline{A} \cdot B + A \cdot B &\equiv \overline{A} \cdot (\overline{B} + B) + A \cdot B && \text{(distributivity law)} \\
&\equiv \overline{A} + A \cdot B && \text{(complement law)} \\
&\equiv (\overline{A} + A) \cdot (\overline{A} + B) && \text{(distributivity law)} \\
&\equiv \overline{A} + B. && \text{(complement law)}
\end{aligned}$$

Which shows that the simplification using the Karnaugh map is correct. Note that Karnaugh maps give the Boolean expression in SOP form which may be simplified further by use of the laws of Boolean algebra.

Example 2.4

For each of the Boolean expressions below:

1. $f(A,B) = A \cdot \overline{B} + \overline{A} \cdot B + \overline{A} \cdot \overline{B}$;
2. $f(A,B,C) = \overline{A} \cdot B + B \cdot \overline{C} + B \cdot C + A \cdot \overline{B} \cdot \overline{C}$;

do the following:

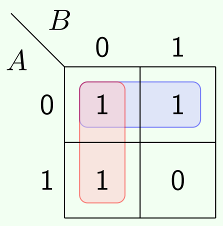
(a) construct a truth table;
(b) use a Karnaugh map to find the SOP form;
(c) use a Karnaugh map to find the POS form;
(d) construct a truth table for the SOP and POS forms.

Solution:

1. (a)

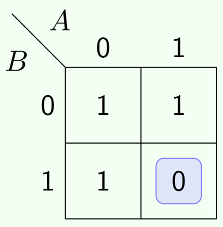
A	B	$A \cdot \overline{B} + \overline{A} \cdot B + \overline{A} \cdot \overline{B}$
0	0	1
0	1	1
1	0	1
1	1	0

(b)



$f(A,B) = \overline{A} + \overline{B}$

(c)



$$\begin{aligned}\overline{f(A,B)} &\equiv A \cdot B\\ f(A,B) &\equiv \overline{A \cdot B}\\ &\equiv \overline{A} + \overline{B}\end{aligned}$$

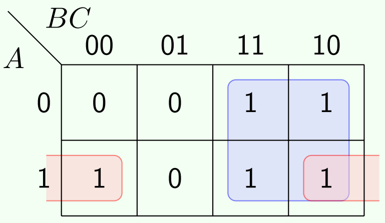
(d)

A	B	$\overline{A} + \overline{B}$
0	0	1
0	1	1
1	0	1
1	1	0

2. (a)

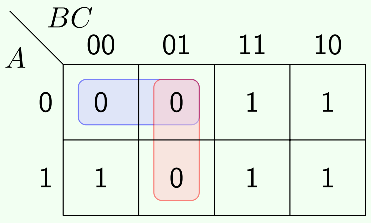
A	B	C	$\overline{A}\cdot B+B\cdot\overline{C}+B\cdot C+A\cdot\overline{B}\cdot\overline{C}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

(b)



$f(A,B,C)\equiv B+A\cdot\overline{C}$

(c)



$$\begin{aligned}
\overline{f(A,B,C)} &\equiv (\overline{A}\cdot\overline{B})+(\overline{B}\cdot C)\\
f(A,B,C) &\equiv \overline{(\overline{A}\cdot\overline{B})\cdot(\overline{B}\cdot C)}\\
&\equiv \overline{(\overline{A}\cdot\overline{B})}+\overline{(\overline{B}\cdot C)}\\
&\equiv (\overline{\overline{A}}+\overline{\overline{B}})\cdot(\overline{\overline{B}}+\overline{C})\\
&\equiv (A+B)\cdot(B+\overline{C})
\end{aligned}$$

(d)

A	B	C	$B+A\cdot\overline{C}$	$(A+B)\cdot(B+\overline{C})$
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	0	0
1	1	0	1	1
1	1	1	1	1

2.8 Half and full adders

The purpose of logic gates is to build circuits that perform arithmetic operations. The two basic arithmetic operations are addition and subtraction, from these we can derive multiplication, division, exponents and roots. The half adder of full adder are logic circuits that are used to sum two or more binary inputs.

2.8.1 Half adder

Consider the binary addition of two inputs $A + B$ (here the $+$ symbol represents addition and not the OR gate), the four possible combinations are

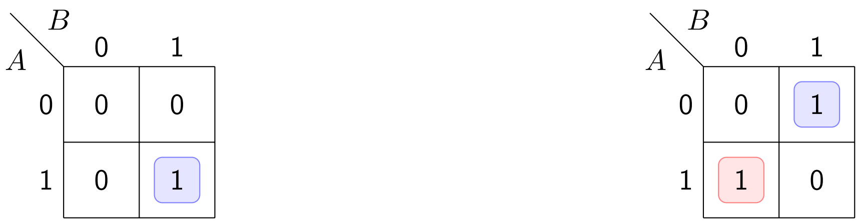
$$\begin{aligned}0 + 0 &= 00,\\ 0 + 1 &= 01,\\ 1 + 0 &= 01,\\ 1 + 1 &= 10.\end{aligned}$$

We refer to the digit on the right of the output as the sum or S and the digit on the left as the carry or C. The first three cases we only need 1 bit to store the output since the value of the carry is 0 but in the fourth case where the value of the carry is 1 we will require 2 bits to store the output. So we need to create a logic circuit that can perform this addition and output the sum and the carry. The truth table for the addition of two 1-bit numbers is shown in table 2.5.

Table 2.5: Truth table for the addition of two 1 bit numbers.

inputs		outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The corresponding Karnaugh map for the S and C columns from table 2.5 are shown in figure 2.14.



(a) carry C

(b) sum S

Figure 2.14: Karnaugh maps for the sum and carry column for the addition of two 1 bit numbers.

Grouping the 1s in figure 2.14(a) gives

$$S = A \cdot B, \tag{2.3}$$

and grouping the 1s in figure 2.14(b) gives

$$\begin{aligned}C &= A \cdot \overline{B} + \overline{A} \cdot B\\ &\equiv A \oplus B,\end{aligned} \tag{2.4}$$

which is an XOR gate. Therefore the logic circuit for adding two 1 bit numbers is shown in figure 2.15.

This circuit is known as the **half adder**.

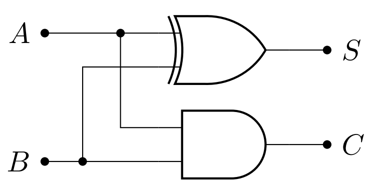


Figure 2.15: The half adder logic circuit.

2.8.2 Full adder

The problem with the half adder is that whilst it is able to sum two 1 bit inputs it cannot be combined so that more than two bit inputs are summed. Consider the binary addition $01 + 11$.

$$\begin{array}{lccc} & 0 & 1 & \\ & 1 & 1 & + \\ \hline \text{carry} & & & \\ \hline \text{sum} & & & \end{array}$$

Starting at with the right column, we have the sum $1 + 1 = 10$ so we put 0 in the sum and we carry 1 over to the next column to the left.

$$\begin{array}{lccc} & 0 & 1 & \\ & 1 & 1 & + \\ \hline \text{carry} & 1 & & \\ \hline \text{sum} & & 0 & \end{array}$$

Moving to the next column to the left we have the sum $0 + 1 + 1 = 10$ so again we put 0 in the sum and carry the 1 to the next column on the left.

$$\begin{array}{lcccc} & & 0 & 1 & \\ & & 1 & 1 & + \\ \hline \text{carry} & 1 & 1 & & \\ \hline \text{sum} & & 0 & 0 & \end{array}$$

We now have no more columns and the answer can be read as the last carry value along with the sum which is 100 as required (i.e., $01 + 11$ in binary is equivalent to $1 + 3$ in decimal which is 4 and the binary equivalent is 100).

To build circuits that can sum 2 or more bit numbers we need to be above to sum to 1 bit numbers and a carry from previous sums. This circuit is known as the **full adder** which takes inputs of two 1 bit values A and B as well as a carry value C_{in} which may have come from a previous calculation. The truth table for a full adder is shown in table 2.6 and the Karnaugh maps for the C_{out} and S columns are shown in figure 2.16.

Table 2.6: The truth table for a full adder.

inputs			outputs	
A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

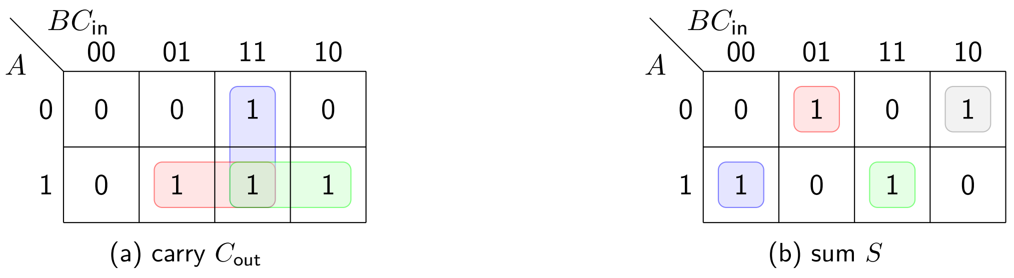


Figure 2.16: Karnaugh maps for the sum and carry column for a full adder.

Grouping the 1s in figure 2.16(a) gives the Boolean expression $A\cdot B + B\cdot C_{\text{in}} + A\cdot C$ which can be simplified using the distributivity law to

$$C_{\text{out}} = A\cdot B + C_{\text{in}}\cdot(A+B). \tag{2.5}$$

Grouping the 1s in figure 2.16(b) (which is trivial as we only have groups of size 1) gives the Boolean expression $A\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C} + A\cdot B\cdot C + A\cdot\overline{B}\cdot\overline{C}$. Simplifying using the laws of Boolean algebra

$$\begin{aligned}
S &= A\cdot\overline{B}\cdot C + \overline{A}\cdot B\cdot\overline{C} + A\cdot B\cdot C + A\cdot\overline{B}\cdot\overline{C} \\
&\equiv C\cdot(A\cdot B + \overline{A}\cdot\overline{B}) + \overline{C}\cdot(\overline{A}\cdot B + A\cdot\overline{B}) && \text{(distributivity law)} \\
&\equiv C\cdot(\overline{\overline{A}\cdot B + A\cdot\overline{B}}) + \overline{C}\cdot(\overline{A}\cdot B + A\cdot\overline{B}) && \text{(see below)} \\
&\equiv C\cdot(\overline{A\oplus B}) + \overline{C}\cdot(A\oplus B) && \text{(definition of } A\oplus B\text{)} \\
&\equiv A\oplus B\oplus C. && \text{(definition of } A\oplus B\text{)}
\end{aligned} \tag{2.6}$$

It might not immediately be obvious to you that $A\cdot B + \overline{A}\cdot\overline{B} \equiv \overline{\overline{A}\cdot B + A\cdot\overline{B}}$ in the third line so let's take moment here to check this

$$\begin{aligned}
\overline{\overline{A}\cdot B + A\cdot\overline{B}} &\equiv \overline{A+\overline{B}}\cdot\overline{\overline{A}+B} && \text{(De Morgan's law)} \\
&\equiv (\overline{A}+B)\cdot(A+\overline{B}) && \text{(De Morgan's law)} \\
&\equiv \overline{A}\cdot A + \overline{A}\cdot\overline{B} + A\cdot B + B\cdot\overline{B} && \text{(distributive law)} \\
&\equiv 0 + \overline{A}\cdot\overline{B} + A\cdot B + 0 && \text{(complement law)} \\
&\equiv \overline{A}\cdot\overline{B} + A\cdot B. && \text{(identity law)}
\end{aligned}$$

So we now have two Boolean expressions for C_{out} and S in equations (2.5) and (2.6) for a full adder. One thing to note that the $A\cdot B$ term in equation (2.5) is the carry output from the half adder equation (2.3) and the $A\oplus B$ in S is the sum output from the half adder equation (2.4). Therefore the circuit for the full adder can be built by combining two half adder circuits where the sum output from the first half adder is combined with C_{in} in the second half adder where the sum output from the second half adder is S and the two carry outputs are combined using an XOR gate to give C_{out}. The circuit for the full adder is shown in figure 2.17.

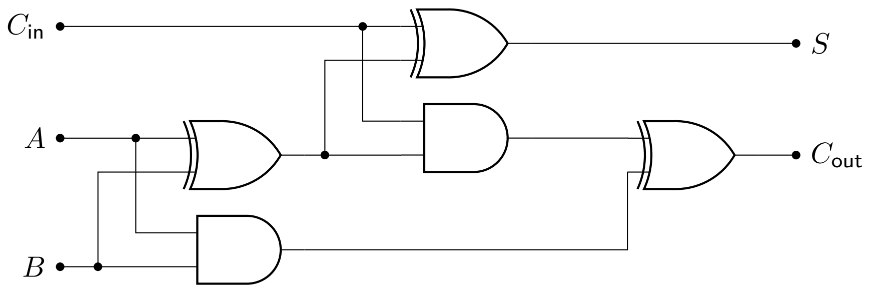


Figure 2.17: Two half adders combined to make a full adder.

Note that we could replace the XOR gate connecting the two carry outputs with an OR gate and not affect the circuit, however this would mean using three different types of gates (XOR, AND and OR) when for manufacturing reasons fewer types is often preferable.

2.8.3 The ripple carry adder

We can string together multiple full adders to create a circuit capable of computing the addition of n bit numbers. For each bit we require a full adder where the inputs are the two bits that are being added together along with the carry output from the previous digit.

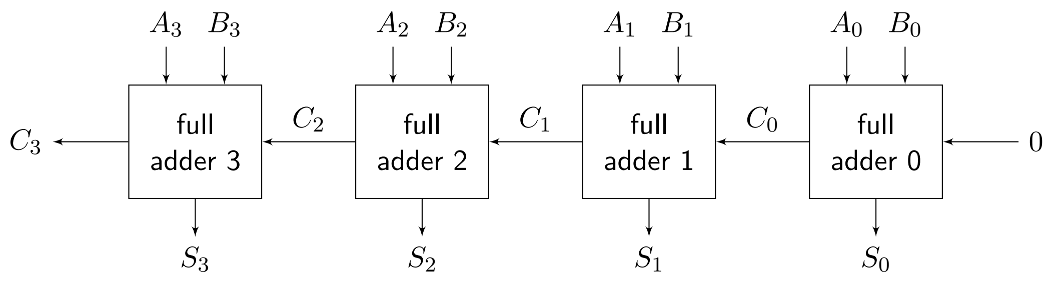


Figure 2.18: Four single bit full adders combined to make a ripple adder to compute the sum of two 4 bit numbers.

Consider the ripple carry adder shown in figure 2.18. This consists of four single bit full adders so it can perform the addition of two 4 bit binary numbers $A_3A_2A_1A_0$ and $B_3B_2B_1B_0$. The inputs to full adder 0 are the rightmost digits of the two numbers, A_0 and B_0, and 0 (note that full adder 0 could also be a half adder since the C_{in} input is 0 but it is simpler just to use all full adders here). The outputs for adder 0 are the rightmost digit of the sum S_0 and the carry C_0. The inputs to adder 1 are the next two digits to the left, A_1 and B_1, and the carry from adder 0. The outputs for adder 1 are the next digit to the left of the sum S_1 and the carry C_1. This pattern continues along the chain until all bits have been added together, the digits of the final sum are $C_3S_3S_2S_1S_0$.

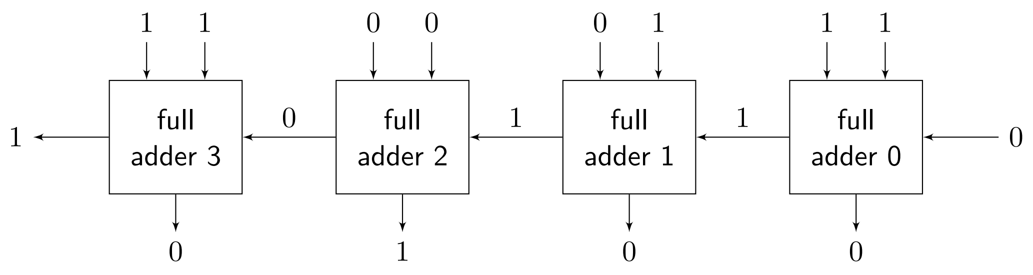


Figure 2.19: The addition of $A = 1001$ and $B = 1011$ using a ripple carry adder.

For example, consider the sum $1001 + 1011$.

- adder 0: inputs are $A_0 = 1$, $B_0 = 1$ and 0 so $S_0 = 0$ and $C_0 = 1$;
- adder 1: inputs are $A_1 = 0$, $B_1 = 1$ and $C_0 = 1$ so $S_1 = 0$ and $C_1 = 1$;
- adder 2: inputs are $A_2 = 0$, $B_2 = 0$ and $C_1 = 1$ so $S_2 = 1$ and $C_2 = 0$;
- adder 3: inputs are $A_3 = 1$, $B_3 = 1$ and $C_2 = 0$ so $S_3 = 0$ and $C_3 = 1$.

Therefore the answer to the sum is $C_3S_3S_2S_1S_0 = 10100$. The decimal equivalent to this sum is $9 + 11 = 20 = 1 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 0 \times 1$ given the binary equivalent 10100.

2.8.4 Representing the half adder and adder using NAND gates

As mentioned in the previous section, it is preferable to use fewer types of gates in a logic circuit where possible. Since NAND gates are universal gates we can replace the XOR and AND gates in figures 2.15 and 2.17 with the NAND gates equivalents from section 2.5 to give the circuit diagrams shown in figures 2.20 and 2.21.

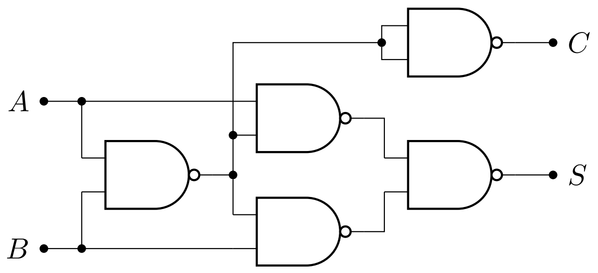


Figure 2.20: The half adder constructed using only NAND gates.

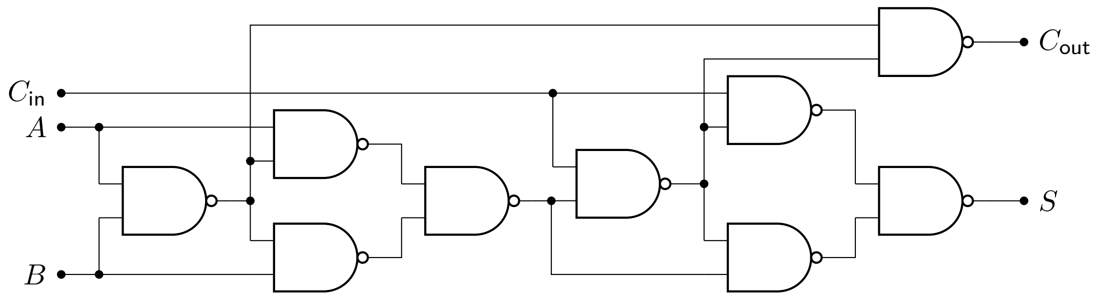
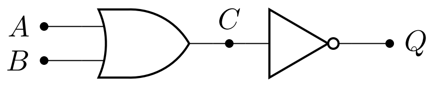


Figure 2.21: The full adder constructed using only NAND gates.

2.9 Tutorial exercises

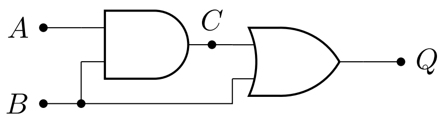
Exercise 2.1. Complete the truth tables for the following circuits:

(a)



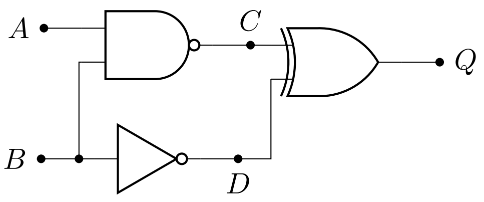
A	B	C	Q
0	0		
0	1		
1	0		
1	1		

(b)



A	B	C	Q
0	0		
0	1		
1	0		
1	1		

(c)



A	B	C	D	Q
0	0			
0	1			
1	0			
1	1			

Exercise 2.2. Draw circuit diagrams for the following Boolean expressions:

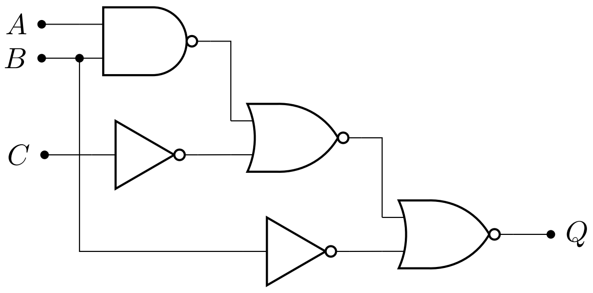
(a) $\overline{A+B\cdot\overline{B}}$; (b) $\overline{(A\cdot B)\cdot(A+C)}$; (c) $\overline{\overline{A}\cdot B+C}+A\cdot\overline{B}$.

Exercise 2.3. Simplify the following Boolean expressions. For each step of the simplification, state which laws you have used.

(a) $\overline{A}\cdot(A+B)$; (b) $(A+A\overline{B})\cdot(C+B\cdot C)$;

(c) $\overline{(\overline{A}+B)\cdot(A+\overline{B})}$; (d) $(A+B\cdot C)\cdot(A+\overline{B})$.

Exercise 2.4. Simplify the Boolean expression for the circuit below so that it uses the fewest possible number of logic gates and draw the simplified circuit diagram.



Exercise 2.5. Prove that the NOR gate is a universal gate.

Exercise 2.6. For each of the Boolean expressions below find the SOP and POS forms.

(a) $f(A,B)=A\cdot(\overline{A}+B)$;

(b) $f(A,B,C)=A\cdot(B+\overline{C})$;

(c) $f(A,B,C) = (A+B)\cdot(\overline{B}+\overline{C})$.

Exercise 2.7. For each of the truth tables below, draw the corresponding Karnaugh map and use them to find the SOP and POS forms.

(a)

A	B	Q
0	0	1
0	1	0
1	0	1
1	1	1

(b)

A	B	C	Q
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(c)

A	B	C	Q
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Exercise 2.8. Given the following truth table:

A	B	C	$f(A,B,C)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

(a) design a circuit that uses the fewest possible logic gates to produce the output $f(A,B,C)$;

(b) draw the circuit using only NAND gates.

The solutions to these exercises are given on page 91.

Chapter 3

Algorithms

Learning outcomes

On successful completion of this chapters students will be able to:

- read an algorithm from pseudocode and implement it;
- apply the bubble sort, quicksort and merge sort algorithm to sort a list of numbers into ascending order;
- understand and apply recursive operations;
- compare the time cost of algorithms using big-O notation.

3.1 Greatest common divisor algorithm

An **algorithm** is a series of steps that is used to perform computation, usually with the intention to solve a problem. The word algorithm is derived from the work of the 9th Century Persian mathematician Muḥammad ibn Mūsā al-Khwārizmī which when translated to Latin the word *algoritmi* (Wikipedia contributors 2001).

A well known algorithm is Euclid's algorithm that is used to calculate the Greatest Common Divisor (GCD) of two numbers a and b. At the first step (known as step $k = 0$) the smaller number, b say, is subtracted from the larger number a. At the start of the second step ($k = 1$) one of the values of a or b has changed during the first step and we repeat the subtraction of the small number from the larger number. The steps are repeated until $a = b$ which is the GCD.

For example, using Euclid's algorithm to calculate the GCD of 315 and 588

$$\begin{array}{lllcl}
k = 0, & a = 588, & b = 315, & \implies & a \leftarrow 588 - 315 = 273, \\
k = 1, & a = 273, & b = 315, & \implies & b \leftarrow 315 - 273 = 42, \\
k = 2, & a = 273, & b = 42, & \implies & a \leftarrow 273 - 42 = 231, \\
k = 3, & a = 231, & b = 42, & \implies & a \leftarrow 231 - 42 = 189, \\
k = 4, & a = 189, & b = 42, & \implies & a \leftarrow 189 - 42 = 147, \\
k = 5, & a = 147, & b = 42, & \implies & a \leftarrow 147 - 42 = 105, \\
k = 6, & a = 105, & b = 42, & \implies & a \leftarrow 105 - 42 = 63, \\
k = 7, & a = 63, & b = 42, & \implies & a \leftarrow 63 - 42 = 21, \\
k = 8, & a = 21, & b = 42, & \implies & b \leftarrow 42 - 21 = 21, \\
k = 9, & a = 21, & b = 21. & &
\end{array}$$

So $\mathrm{GCD}(315, 588) = 21$ (note that the $a \leftarrow x$ notation means a is defined to be equal to x). Here steps $k = 2$ to $k = 7$ involved repeated subtraction of 42 which eventually resulted in a assuming the value of

21. This is the same as dividing 273 by 42 and finding the remainder which is 21, i.e.,

$$273 = 6 \times 42 + 21.$$

Since a takes on the value of the remainder and b is now larger than a then the next calculation is to set b to $b - a$. To ensure we are always doing the same calculation we can set b to the remainder and a to b. Therefore Euclid's algorithm can be rewritten so that the values at the k^{th} step are

$$\begin{aligned} a_k &\leftarrow b_{k-1}, \\ b_k &\leftarrow \operatorname{mod}(a_{k-1}, b_{k-1}), \end{aligned}$$

where the subscript $k-1$ denotes the values from the previous step and $\operatorname{mod}(a,b)$ is the **modulo** operator which returns the remainder of $a \div b$. The algorithm terminates when $a = b$ which will be when $\operatorname{mod} = 0$. Applying the rewritten Euclid's algorithm to $a = 588$ and $b = 315$

$$\begin{array}{llllll} k=0, & a=588, & b=315, & \implies & a \leftarrow 315, & b \leftarrow \operatorname{mod}(588,315) = 273, \\ k=1, & a=315, & b=273, & \implies & a \leftarrow 273, & b \leftarrow \operatorname{mod}(315,273) = 42, \\ k=2, & a=273, & b=42, & \implies & a \leftarrow 42, & b \leftarrow \operatorname{mod}(273,42) = 21, \\ k=3, & a=42, & b=21, & \implies & a \leftarrow 21, & b \leftarrow \operatorname{mod}(41,21) = 0, \end{array}$$

so $\text{GCD}(588, 315) = 21$ as before.

The Euclid's algorithm is presented in **pseudocode** in algorithm 1. Pseudocode is a method of presenting an algorithm in a way that resembles the commands used by a programming language but without using the particular syntax of any programming language. This way the algorithm is written so that it can be applied to a problem or translated to a specific programming language.

Algorithm 1 Pseudocode for Euclid's algorithm

```
function GCD($a$, $b$)
    if $a < b$ then
        $t \leftarrow a$                    ▷ swap $a$ and $b$
        $a \leftarrow b$
        $b \leftarrow t$
    end if
    while $b \neq 0$ do
        $t \leftarrow b$                    ▷ $temp$ is a temporary variable used to store the value of $b$
        $b \leftarrow \operatorname{mod}(a, b)$
        $a \leftarrow t$
    end while
    return $a$
end function
```

3.1.1 Python and MATLAB code

Python and MATLAB code for Euclid's algorithm are shown in listings 3.1 and 3.2. Note that since Python can perform multiple operations on a single line we do not need to use a temporary variable for swapping a and b.

Listing 3.1: Python code for Euclid's algortihm.

```python
def gcd(a, b):
    if a < b:
        a, b = b, a
    while b != 0:
        a, b = b, a % b
    return a
```

Listing 3.2: MATLAB code for Euclid's algortihm.

```
function a = gcd(a, b)

if a < b
    t = a;
    a = b;
    b = t;
end

while b ~= 0
    t = b;
    b = mod(a, b);
    a = t;
end

end
```

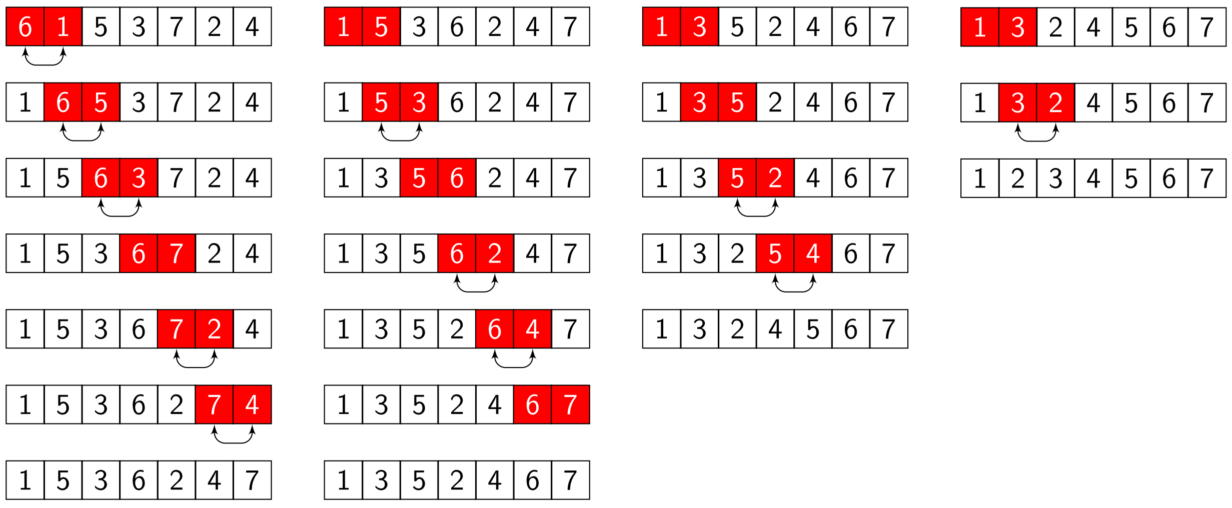
3.2 Sorting algorithms

One of the most common problems encountered in computing is the sorting of items into ascending or descending order.

3.2.1 Bubble sort

Bubble sort is the simplest sorting algorithm. Given a list of numbers, the algorithm loops through pairs of numbers and swaps them if the first number has a higher value than the second number. This will require several passes through the list and the algorithm terminates when no swaps have occurred in the last pass through. This has the effect that the higher value numbers will gradually migrate to the end of the list similar to bubbles rising to the top of a liquid, which is how the algorithm gets it's name.

For example, lets apply the bubble sort algorithm to the list of numbers [6, 1, 5, 3, 7, 2, 4] as shown in figure 3.1.



(a) Pass 1 (b) Pass 2 (c) Pass 3 (d) Pass 4

Figure 3.1: Implementation of the bubble sort algorithm.

In the first pass, figure 3.1(a), five swaps were required to move 6 and 7 up the list. In the second pass, figure 3.1(b), since we know that the largest number in the list must now be at the top we have one fewer pairs to check. After the second pass we have the highest two numbers in the correct place. The algorithm requires a further two passes through the list to sort it into the correct order and another pass (not shown) that would not have any swaps so the algorithm terminates.

Example 3.1

Use the bubble sort algorithm to sort the numbers [3, 1, 6, 5, 4, 2].

Solution:

Pass 1: there are 6 numbers in the list so we need to check 5 pairs in pass 1

$[\underline{3}, \underline{1}, 6, 5, 4, 2]$	swap 3 and 1
$[1, \underline{3}, \underline{6}, 5, 4, 2]$	do not swap 3 and 6
$[1, 3, \underline{6}, \underline{5}, 4, 2]$	swap 6 and 5
$[1, 3, 5, \underline{6}, \underline{4}, 2]$	swap 6 and 4
$[1, 3, 5, 4, \underline{6}, \underline{2}]$	swap 6 and 2
$[1, 3, 5, 4, 2, 6]$	

Pass 2: need to check 4 pairs

$[\underline{1}, \underline{3}, 5, 4, 2, 6]$	do not swap 1 and 3
$[1, \underline{3}, \underline{5}, 4, 2, 6]$	do not swap 3 and 5
$[1, 3, \underline{5}, \underline{4}, 2, 6]$	swap 5 and 4
$[1, 3, 4, \underline{5}, \underline{2}, 6]$	swap 5 and 2
$[1, 3, 4, 2, 5, 6]$	

Pass 3: need to check 3 pairs

$[\underline{1}, \underline{3}, 4, 2, 5, 6]$	do not swap 1 and 3
$[1, \underline{3}, \underline{4}, 2, 5, 6]$	do not swap 3 and 4
$[1, 3, \underline{4}, \underline{2}, 5, 6]$	swap 4 and 2
$[1, 3, 2, 4, 5, 6]$	

Pass 4: need to check 2 pairs

$[\underline{1}, \underline{3}, 2, 4, 5, 6]$	do not swap 1 and 3
$[1, \underline{3}, \underline{2}, 4, 5, 6]$	swap 3 and 2
$[1, 2, 3, 4, 5, 6]$	

Pass 5: need to check 1 pair

$[\underline{1}, \underline{2}, 2, 4, 5, 6]$	do not swap 1 and 3
$[1, 2, 3, 4, 5, 6]$	

The bubble sort algorithm is presented in **pseudocode** in algorithm 2.

Algorithm 2 Pseudocode for the bubble sort algorithm

```
function BUBBLESORT($X$)
    $n \leftarrow$ length of $X$
    $k \leftarrow 0$
    $swap \leftarrow \text{True}$                    ▷ $swap$ is a flag that shows if a swap has occurred
    while $swap = \text{True}$ do
        $swap \leftarrow \text{False}$
        for $i = 1 \ldots n - k - 1$ do
            if $X(i) > X(i + 1)$ then
                $t \leftarrow X(i)$          ▷ $temp$ is a temporary variable used to store the value of $X(i)$
                $X(i) \leftarrow X(i + 1)$
                $X(i + 1) \leftarrow t$
                $swap \leftarrow \text{True}$
            end if
        end for
        $k \leftarrow k + 1$
    end while
    return $X$
end function
```

3.2.2 Quicksort

The major disadvantage with the bubble sort algorithm is that it is slow to implement. A quicker algorithm is **quicksort** which is a popular sorting algorithm developed by British computer scientist Tony Hoare (1961). The algorithm works by choosing a number from the list to be the **pivot**, the list is then partitioned about the pivot so that those numbers that are less than the pivot number are moved to the left of the pivot and those numbers that are greater than the pivot are moved to the right of the pivot. This process is repeated for the two sub-lists which are less than or greater than the pivot. Eventually the sub-lists will contain just element and the list will be in sorted order.

The choice of the pivot element is arbitrary, in the examples presented here the last element in the list is chosen for simplicity. The partitioning of the list is done setting an index i to 1 (the index of the first element in the loop so for a Python implementation this would be 0) and look for an element in the list that is less than the pivot element. We swap this element with the one with index i and increment i by 1. This continues until all non-pivot elements have been checked and then we swap the pivot element with the one with index i which places the pivot between the left and right sub-lists.

The pseudocode for the quicksort algorithm is shown in algorithm 3. The algorithm uses two functions, QUICKSORT is a function that calls itself recursively starting with $l = 1$ and $r = n$ (the index of the last element in the list). The function PARTITION performs that partitioning of the lists about the pivot.

For example, consider the implementation of the quicksort algorithm used to sort the list [6, 1, 5, 3, 7, 2, 4] as shown in figure 3.2. The steps of the algorithm are:

(1) $X = [6, 1, 5, 3, 7, 2, 4]$: $l = 1$ and $r = 2$ then $l < r$ so swap 6 and 2;

(2) $X = [2, 1, 5, 3, 7, 6, 4]$: $l = 3$ and $r = 4$ then $l < r$ so swap 5 and 3;

(3) $X = [2, 1, 3, 5, 7, 6, 4]$: $l = 4$ and $r = 3$ then $l > r$ so swap 5 with the pivot 4;

(4) $X = [2, 1, 3, 4, 7, 6, 5]$: partitioning is complete for pivot 4, apply quicksort to sub-lists $[2, 1, 3]$ and $[7, 6, 5]$

(5) $X = [2, 1, 3]$: $l = 3$ and $r = 1$ then $l > r$ so swap 3 with the pivot (which is itself);

$X = [5, 6, 7]$: $l = r = 7$ so swap 7 with the pivot 5;

(6) $X = [2, 1, 3]$: partitioning is complete for pivot 3, apply quicksort to the sub-list $[2, 1]$;

$X = [5, 6, 7]$: partitioning is complete for pivot 5, apply quicksort to the sub-list $[6, 7]$;

(7) $X = [1, 2]$: $l = r = 1$ so swap 2 with the pivot 1;

$X = [6, 7]$: $l = 7$ and $r = 6$ then $l > r$ so swap 7 with the pivot (which is itself);

(8) $X = [1, 2]$: partitioning is complete for pivot 2, apply quicksort to the sub-list $[2]$;

$X = [6, 7]$: partitioning is complete for pivot 7, apply quicksort to the sub-list $[6]$;

(9) $X = [2]$: list is of length 1 so exit;

$X = [6]$: list of of length 1 so exit and terminate algorithm. The final sorted list is $X = [1, 2, 3, 4, 5, 6, 7]$.

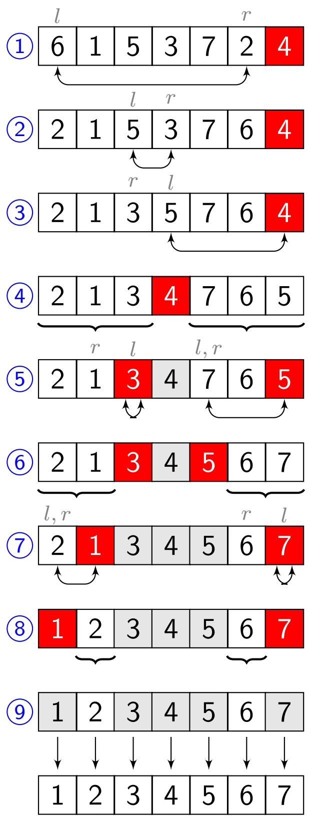


Figure 3.2: Implementation of the quicksort algorithm.

Example 3.2

Use the quicksort algorithm to sort the numbers [3, 1, 6, 5, 4, 2].

Solution:

Choosing the last number in the list to be the pivot.

[3, 1, 6, 5, 4, $\underline{2}$]	swap 3 and 1
[1, 3, 6, 5, 4, $\underline{2}$]	swap 3 and 2]
[1, $\underline{2}$, 3, 6, 5, 4]	partition sub-lists [1] and [3, 6, 5, 4]
[[$\underline{1}$], 2, [3, 6, 5, $\underline{4}$]]	sub-list [1]: do nothing, sub-list [3, 6, 5, 4]: swap 6 and 4
[[$\underline{1}$], 2, [3, $\underline{4}$, 6, 5]]	partition sub-lists [3] and [6, 5]
[[1], 2, [[$\underline{3}$], 4, [6, $\underline{5}$]]]	sub-list [3]: do nothing, sub-list [6, 5]: swap 6 and 5
[[1], 2, [[3], 4, [$\underline{5}$, 6]]]	partition sub-list [6]
[[1], 2, [[3], 4, [5, [$\underline{6}$]]]]	sub-list [6]: do nothing

All sub-lists only have one elements so the sorted list is [1, 2, 3, 4, 5, 6].

Algorithm 3 Pseudocode for the quicksort algorithm

```
function PARTITION($X$, $l$, $r$)
    $pivot \leftarrow X(r)$                          ▷ store the partition index in $p$
    while $l < r$ do
        while $X(l) < X(pivot)$ do                   ▷ move $l$ to the right
            $l = l + 1$
        end while
        while $X(r) \geq X(pivot)$ and $r \geq 1$ do ▷ move $r$ to the left
            $r = r - 1$
        end while
        if $l < r$ then                              ▷ swap $X(l)$ and $X(r)$
            $temp = X(l)$
            $X(l) = X(r)$
            $X(r) = temp$
        end if
    end while
    $temp \leftarrow X(l)$                           ▷ move the pivot between the two sub-lists
    $X(l) \leftarrow X(pivot)$
    $X(pivot) \leftarrow temp$
    return $X$ and $l$
end function

function QUICKSORT($X$, $l$, $r$)
    if $l < r$ then
        $X, pivot \leftarrow$ QUICKSORT$(X, l, r)$   ▷ move list elements either side of the pivot
        $X \leftarrow$ QUICKSORT$(X, l, p - 1)$      ▷ repeat algorithm for left sub-list
        $X \leftarrow$ QUICKSORT$(X, p + 1, r)$      ▷ repeat algorithm for right sub-list
    end if
    return $X$
end function
```

3.2.3 Merge sort

Quicksort is what is known as a "divide and conquer" method since the list is recursively partitioned until we have lists containing two elements which are sorted in ascending order. Another divide and conquer sorting method is **merge sort**. Merge sort is similar to quicksort in that we divide the list into two sub-lists A and B but instead of populating the sub-lists depending on the value of the elements compared to a pivot, the list is simply split at the element that is midway along the list so that A contains the numbers in the lower half (including the midpoint) and B contains the numbers in the upper half. This splitting continues until A and B contain just one element. The sub-lists are then merged by populating a sub-list by taking the next smallest element from A or B. The final merged list will contain the sorted list.

For example, consider the implementation of the merge sort algorithm used to sort the list [6, 1, 5, 3, 7, 2, 4] as shown in figure 3.3. The steps of the algorithm are:

(1) List [6, 1, 5, 3, 7, 2, 4] – the midpoint of the list is 3 the index of which is found using $i = \text{int}((\text{listlength} + 1)/2)$. The list is split into two sub-lists [6, 1, 5, 3] (containing the midpoint since there are an odd number of elements) and [7, 2, 4].

(2) Sub-list [6, 1, 5, 3] – the midpoint is 1 so the list is split into the sub-lists [6, 1] and [5, 3].

(3) Sub-list [6, 1] – the midpoint is 6 so the list is split into the sub-lists [6] and [1].

(4) The sub-lists [6] and [1] are merged into the sub-list [1, 6].

(5) Sub-list [5, 3] – the midpoint is 5 so the list is split into the sub-lists [5] and [3].

(6) The sub-lists [5] and [3] are merged into the sub-list [3, 5].

(7) The sub-lists [1, 6] and [3, 5] are merged into the sub-list [1, 3, 5, 6].

(8) Sub-list [7, 2, 4] – the midpoint is 2 so the list is split into the sub-lists [7, 2] and [4].

(9) Sub-list [7, 2] – the midpoint is 7 so the list is split into the sub-lists [7] and [2].

(10) The sub-lists [7] and [2] are merged into the sub-list [2, 7].

(11) The sub-lists [2, 7] and [4] are merged into the sub-list [2, 4, 7].

(12) the sub-lists [1, 3, 5, 6] and [2, 4, 7] are merged into the sorted list [1, 2, 3, 4, 5, 6, 7].

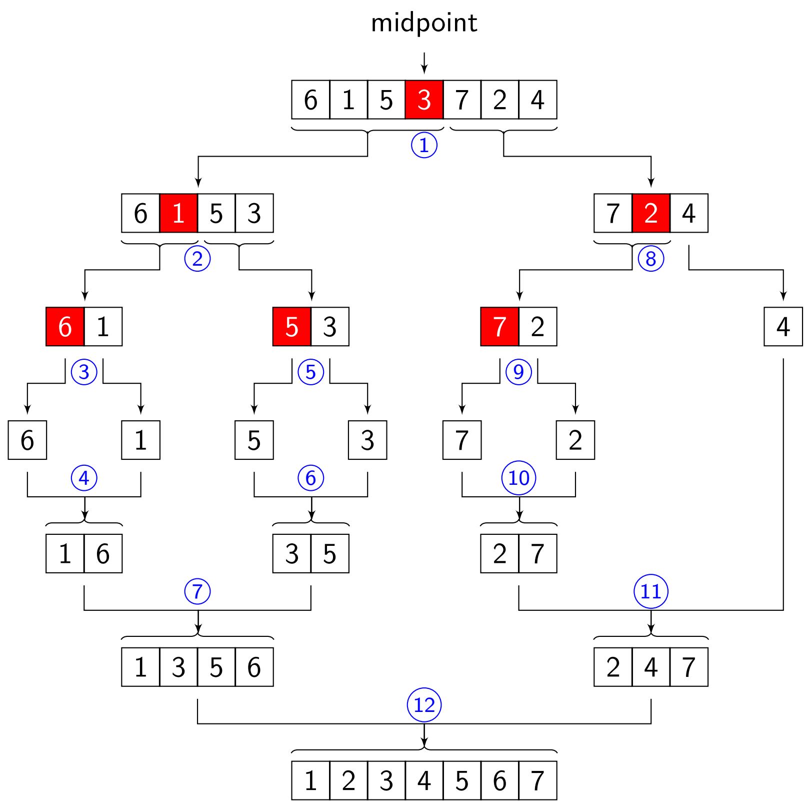


Figure 3.3: Implementation of merge sort.

Example 3.3

Use the merge sort algorithm to sort the numbers [3, 1, 6, 5, 4, 2].

Solution:

Underlined numbers denote the midpoint.

[3, 1, $\underline{6}$, 5, 4, 2]	partition list
[[3, $\underline{1}$, 6], [5, $\underline{4}$, 2]]	partition sub-lists [3, 1, 6], [5, 4, 2]
[[[$\underline{3}$, 1], [$\underline{6}$]], [[$\underline{5}$, 4], [$\underline{2}$]]]	partition sub-lists [3, 1] and [5, 4]
[[[[$\underline{3}$], [1]], [6]], [[[$\underline{5}$], [4]], [2]]]	merge [3] with [1], and [5] with [4]
[[[1, 3], [6]] , [[4, 5], [2]]]	merge [1, 3] with [6] and [4, 5] with [2]
[[1, 3, 6], [2, 4, 5]]	merge [1, 3, 6] with [2, 4, 5]
[1, 2, 3, 4, 5, 6]	

The pseudocode for the merge sort algorithm is shown in algorithm 4

Algorithm 4 Pseudocode for the merge sort algorithm

```
function MERGE($L$, $R$)
    $i \leftarrow 1$
    $j \leftarrow 1$
    $X \leftarrow$ empty list
    while $i < \text{length}(L)$ and $j < \text{length}(R)$ do
        if $L(i) < R(i)$ then
            append $L(i)$ to $X$
            $i \leftarrow i + 1$
        else
            append $R(j)$ to $X$
            $j \leftarrow j + 1$
        end if
    end while
    if $i < \text{length}(L)$ then                          ▷ $L$ still has elements not in $X$
        append the rest of $L$ to the end of $X$
    else                                                    ▷ $R$ still has elements not in $X$
        append the rest of $R$ to the end of $X$
    end if
    return $X$
end function

function MERGESORT($X$)
    if $\text{length}(X) > 1$ then
        set $m \leftarrow \text{int}((\text{length}(X) + 1)/2)$        ▷ calculate index of midpoint
        call $L \leftarrow$ MERGESORT($X(1 : m)$)               ▷ apply merge sort to left-hand list
        call $R \leftarrow$ MERGESORT($X(m + 1 : end)$)         ▷ apply merge sort to right-hand list
        call $X \leftarrow$ MERGE($L, R$)                       ▷ merge left and right-hand lists
    end if
    return $X$
end function
```

3.3 Recursion

The pseudocode for the quicksort and merge sort algorithms shown in algorithms 3 and 4 both involve a function making a call to itself. This is known as **recursion** (see section 3.3[1]) and is often used in divide and conquer algorithms.

When using recursion in an algorithm the following structure is used:

- **base case** – this is a command used when a recursive step is not required and the current scenario should terminate. For example, in algorithm 4, the MERGESORT function will return the list X if its length is equal to 1, i.e., the list cannot be partitioned further.
- **recursive step** – this is a set of commands that are applied which will eventually result in the base case being met. For example, in algorithm 4 if the length of the list X is greater than 1 then X is split into two and the MERGESORT function is called recursively to each sub-list.

Consider the Fibonacci series which is defined by

$$F_n = \begin{cases} 0, & \text{if } n = 0, \\ 1, & \text{if } n = 1, \\ F_{n-2} + F_{n-1}, & \text{if} i > 1. \end{cases}$$

This gives the series

$$F_0 = 0, \qquad F_1 = 1, \qquad F_2 = 1, \qquad F_3 = 2, \qquad F_4 = 3, \qquad F_5 = 5, \qquad \ldots$$

The pseudocode for computing the Fibonacci number F_n is shown in algorithm 5

Algorithm 5 Pseudocode for computing Fibonacci numbers

```
function FIBONACCI($n$)
    if $n < 2$ then
        return $n$
    else
        $F_0 \leftarrow 0$
        $F_1 \leftarrow 1$
        for $i = 1 \ldots n$ do
            $F_2 \leftarrow F_0 + F_1$
            $F_0 \leftarrow F_1$
            $F_1 \leftarrow F_2$
        end for
        return $F_1$
    end if
end function
```

We can write an algorithm for computing the Fibonacci number F_n using recursion is shown in algorithm 6. The base case is when $n < 2$ and the algorithm returns $F_n = n$ (i.e.. $F_0 = 0$ or $F_1 = 1$). The recursive step is implemented when $n \geq 2$ and

We can visualise recursion using a tree structure. Consider the Fibonacci number F_5 calculated using recursion as shown in algorithm 6 which is represented using a tree in figure 3.4. Since $n \geq 2$ then $F_5 = F_3 + F_4$ which also require a recursive step to calculate $F_3 = F_1 + F_2$ and $F_4 = F_2 + F_3$ and so on. Where a base case is encountered we have a leaf node (see chapter 4 for details on trees) where $F_0 = 0$ and $F_1 = 1$. The values of the parent nodes are the sum of the values of the child nodes, e.g., $F_2 = F_0 + F_1 = 0 + 1 = 1$. Computing the values for all nodes in the tree gives $F_5 = 5$.

[1]This is a joke stolen from Google where if you search for 'recursion' it returns a message above the search results that states "Did you mean: recursion" which links to the same page.

Algorithm 6 Pseudocode for computing Fibonacci numbers using recursion

```
function FIBONACCI($n$)
    if $n < 2$ then
        return $n$
    end if
    return FIBONACCI($n-1$) + FIBONACCI($n-2$)
end function
```

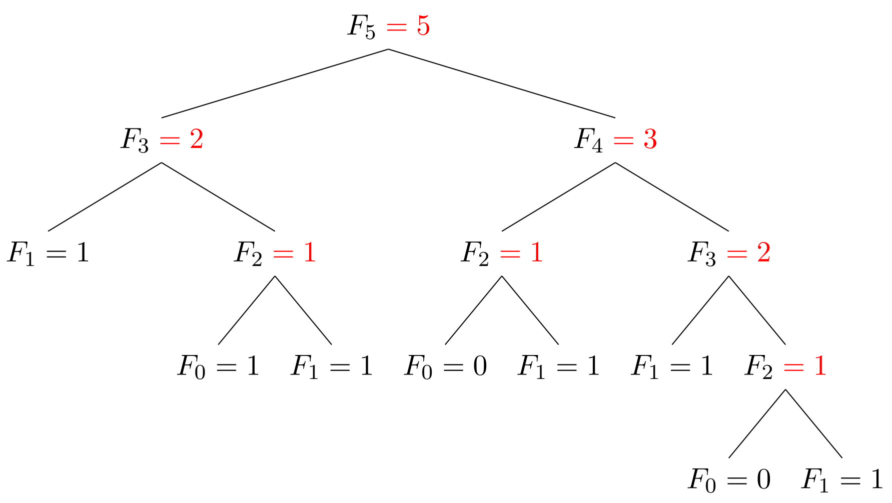


Figure 3.4: Tree representing the calculation of the Fibonacci number F_5.

Example 3.4

Write pseudocode that uses recursion to compute the sum of the numbers between 1 and n.

Solution:

We need to write a function $\textsc{sum}(n)$ which will return the value $1+2+\cdots+n$. So if n is the input we need to go through the numbers in descending order, i.e.,

$$\begin{aligned}\textsc{sum}(n) &= n + \textsc{sum}(n-1)\\ &= n + (n-1) + \textsc{sum}(n-2)\\ &\ \ \vdots\\ &= n + (n-1) + \cdots + 2 + \textsc{sum}(1)\end{aligned}$$

So the base case is when $n = 1$ and we return the value of n and in the recursive step we need to add the current number n to the sum of all of the numbers less than n, i.e., $n + \textsc{sum}(n-1)$. Therefore the pseudocode is

```
function SUM($n$)
    if $n = 1$ then
        return $n$
    end if
    return $n$ + SUM($n-1$)
end function
```

3.4 Complexity

Complexity is the cost of implementing an algorithm and is based on the size of the input n. Even though different algorithms may achieve the same results, they may have required more steps or computations to get there. For example, in the previous section we looked three different algorithms for performing the same action, sorting a list of numbers into ascending order. These algorithms were coded in Python and used to sort an lists containing 1000 random integers. The times taken to sort the list are shown below.

Listing 3.3: Times taken for sorting algorithms to sort 1000 random numbers.

```
Bubble sort : 0.3325s
Quicksort   : 0.0552s
Merge sort  : 0.0054s
```

So the bubble sort algorithm took the longest, followed by the quicksort algorithm and merge sort algorithm was the quickest. So different algorithms take different amounts of time to run on a computer. The reason for this is that they use different number of arithmetic operations and different memory requirements. It is useful to know which algorithm will perform the quickest when presented with a task.

The factors affecting the speed which an algorithm runs on a computer are:

(i) the speed of the computer hardware;

(ii) the efficiency programming language used;

(iii) the efficiency of the program (i.e., the programming skills of the programmer);

(iv) the hardware requirements of the tasks running in the background (i.e., other programs running at the same time);

(v) the complexity of the algorithm;

(vi) the size of the input.

Here (i) to (iv) will vary so can be ignored when analysing the speed of an algorithm where we focus on (v) and (vi). The speed of computer hardware tends to increase over time so we need a way of measuring the cost of an algorithm that is independent of time. For this reason we count the number of times a "principal activity" is performed in an algorithm. The principal activity may depend on the type of algorithm, for example in the bubble sort algorithm we were comparing pairs of numbers whereas in Euclid's algorithm we we calculating the modulo between two numbers.

Another thing we need to consider is that an algorithm may require different number of operations for different inputs, for example, using the quicksort algorithm to sort an list that has already been sorted would require no swaps so would be performed quicker than when used to sort and unsorted list. So we think of the worst case, best case and average case

- **worst case** – this is the maximum number of times the principal activity has been performed for all inputs of size n;
- **best case** – the minimum number of times the principal activity is performed for specific inputs of size n;
- **average case** – the number of times the principal activity is performed on average.

The average case is the most useful measure as in practice we will rarely have an input that results in the worst or best cases. However, this is difficult to measure since we are unlikely to know how the input set will be distributed.

3.4.1 Big-O notation

Let's determine the number of principal activities required to perform the bubblesort algorithm on a list of n numbers. If we consider the principal activity to be checking whether a pair of numbers requires swapping then if we have n numbers in the list on the first pass we have $n-1$ pairs to check. At the end of the first pass we know that the largest number in the list is in the correct position at the end so on the second pass we only need to check $n-2$ pairs. Assuming that each pass requires at least one swap then

$$\text{number of checks} = 1 + 2 + \cdots + (n-2) + (n-1).$$

This is the sum of an arithmetic series with the first term $a = 1$ and common difference $d = 1$ so

$$\begin{aligned}\text{number of checks} &= \frac{n}{2}(2a + (n-1)d) \\ &= \frac{(n-1)}{2}(2 + ((n-1)-1)) \\ &= \frac{1}{2}n^2 - \frac{1}{2}n.\end{aligned}$$

So the cost of the worst case for the bubble sort algorithm is $\frac{1}{2}n^2 - \frac{1}{2}n$. Let $t(n)$ be the time taken for a computer to apply the bubble sort algorithm, if we have two computers, A and B, where computer B is 10 times faster than computer A then

$$\begin{aligned}\text{computer } A: &\qquad t(n) = 10n^2 - 10n, \\ \text{computer } B: &\qquad t(n) = n^2 - n.\end{aligned}$$

Given an input of size $n = 1000$ we have

$$\begin{aligned}\text{computer } A: &\qquad t(n) = 10(1000^2) - 10(1000) = 9990000, \\ \text{computer } B: &\qquad t(n) = 1000^2 - 1000 = 999000.\end{aligned}$$

Note that for both computers, the value of the first term, 10^7 and 10^6, is much larger than the value of the second term, 10^4 and 10^3 so we say that the first term dominates. When talking about the complexity of an algorithm, we ignore all but the first term, n^2, and make the statement

"$t(n)$ grows like n^2 as n increases"

What this means is that if we apply the bubblesort algorithm to sort two lists one twice as long as the other, then we would expect the larger list to take $2^2 = 4$ times as long as the smaller list. Also, note that this statement is independent of the speed of different computers.

This concept represented mathematically using **big-O notation** where if the value of the function $t(n)$ grows like n^2 as n increases then we say

$$t(n) = O(n^2).$$

Some examples of different complexities are:

- $O(1)$ **constant time**: the time taken to perform an algorithm does not change as the size of the input changes
- $O(n)$ **linear time**: the time taken to perform and algorithm is directly proportional to the size of the input n. Consider an algorithm that calculates the numbers of 0s in a list of numbers by checking each one. This would have complexity $O(n)$.
- $O(n^2)$ **quadratic time**: the time taken to perform an algorithm is proportional to the square of the size of the input n. We have seen the the bubble sort has quadratic time complexity;.
- $O(\log n)$ **logarithmic time**: the time taken for each step of an algorithm decreases. For example, imagine you are looking up a word in a dictionary (not on a computer). Checking each page for the word you are looking for starting at the beginning would be $O(n)$ is if we increased the number of pages in our dictionary the complexity would increase proportionally. However, if we used the more sensible method of opening the dictionary halfway and then splitting the half which will contain our word halfway again we will soon find our word since the number of pages in our splits will halve each time.

- $O(n!)$ **factorial time**: the time taken for each step of an algorithm is proportional to the factorial of the size of the input n. Take the classic travelling salesman problem where a sales man wants to find a path that visits a number of cities with the shortest travelling distance (we look at shortest path problems in chapter 4). If we were to exhaustively check each possible path then for n cities we have $n!$ permutations, i.e., for 10 cities we have $10! = 3628800$ permutations to consider but for 20 cities we have $20! = 2.4 \times 10^{18}$ permutations (to put this number into perspective if it took 1 millisecond to check each permutation then it would take 77 million years to check them all).

The sorting algorithms studied in this chapter have the following complexities:

- Bubblesort
  - worst case $O(n^2)$
  - average case $O(n^2)$
  - best case $O(n)$
- Quicksort
  - worst case $O(n^2)$
  - average case $O(n \log n)$
  - best case $O(n \log n)$
- Merge sort
  - worst case $O(n \log n)$
  - average case $O(n \log n)$
  - best case $O(n \log n)$

3.5 Tutorial Exercises

Exercise 3.1. Use Euclid's algorithm to find the greatest common divisor between:

(a) 48 and 102; (b) 585 and 1027.

Exercise 3.2. Showing each step of the algorithm, use bubble sort to sort the numbers $[5, 1, 3, 6, 2, 4]$ into ascending order.

Exercise 3.3. Repeat exercise exercise 3.2 using quicksort.

Exercise 3.4. Repeat exercise exercise 3.2 using merge sort.

Exercise 3.5. Write a Python or MATLAB function that uses the bubble sort algorithm to sort a list of numbers. Use your function to sort the numbers in exercise 3.2.

Exercise 3.6.

(a) Write down the pseudocode for a function MINRECURSION(X, n) that uses recursion to find the smallest number in the list X which has n elements.

(b) Write down the steps used by your algorithm to find the minimum value for the list [5, 6, 3, 7, 4].

(c) Code your function in Python or MATLAB and text your code on the list in part (b).

Exercise 3.7. The pseudocode for Gaussian elimination, which solves a system linear equations of the form $A\mathbf{x} = \mathbf{b}$, is given in algorithm 7.

Algorithm 7 Gaussian elimination

```
function GaussianElimination($A$, $\mathbf{b}$)
    for $j = 1 \ldots n-1$ do                          ▷ row operations
        for $i = j+1 \ldots n$ do
            $r \leftarrow a_{i,j}/a_{j,j}$
            for $k = j \ldots n$ do
                $a_{i,k} \leftarrow a_{i,k} - ra_{j,k}$
            end for
            $b_i \leftarrow b_i - rb_j$
        end for
    end for
    $\mathbf{x} \leftarrow \mathbf{b}$
    for $i = n \ldots 1$ do                            ▷ back substitution
        for $j = i+1 \ldots n$ do
            $x_i \leftarrow x_i - a_{i,j}x_{j+1}$
        end for
        $x_i \leftarrow x_i/a_{i,i}$
    end for
    return $\mathbf{x}$
end function
```

When using Gaussian elimination to solve a linear system of n equations in n unknowns:

(a) how many division operations are required?

(b) how many subtraction operations are required?

(c) how many multiplication operations are required?

(d) what is the complexity of Gaussian elimination?

The solutions to these exercises is given in appendix A.3.

Chapter 4

Graph Theory

Learning outcomes

On successful completion of this chapters students will be able to:

- identify graphs, trees, walks trails and paths;
- represent a graph using an adjacency matrix;
- perform depth-first and breadth-first search to obtain node ordering and a spanning tree;
- solve shortest path problems using Dijkstra's, Bellman-Ford and A* algorithms.

4.1 Graphs

In mathematics a **graph** is an ordered pair $G = (V, E)$ where

- $V = \{v_1, v_2, \dots, v_n\}$ are a set of **nodes** (also called **vertices** ;
- $E = \{(u, v)$: edge joining node u to node $v\}$ is a set of **edges**.

A graph is represented diagrammatically using circles to represent the nodes and lines to represent the edges (figure 4.1). The configuration of nodes does not matter as long as the lines that represent the edges connect the correct nodes.

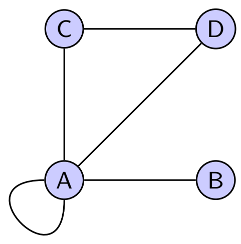


Figure 4.1: An example of a graph.

Definition 4.1: Adjacent nodes

Two nodes are said to be **adjacent** (or **incident**) if they are connected by an edge. For example, nodes A and D are adjacent to node A in figure 4.1.

Definition 4.2: Degree

The **degree** of a node v_i, denoted by $\deg(v_i)$ is the number of nodes that are adjacent to node v_i. For example, the node D in figure 4.1 has degree $\deg(D) = 2$. In the case where an edge is a loop that connects a node with itself we add 2 to the degree of the node. The node A in figure 4.1 has degree $\deg(A) = 5$.

Definition 4.3: Leaf nodes

A node with a degree of 1 is called a **leaf node**.

4.1.1 Walks, trails and paths

We use the terms walk, trails and paths to describe the traversal of a graph:

- A **walk** is a sequence of edges that joins nodes of a graph. Let $G = (V, E)$ be a graph then a walk is denoted using the ordered set $W = (v_1, v_2, \ldots, v_n)$. The walk is **closed** if $v_1 = v_n$ and is **open** otherwise.
- A **trail** is a walk where no two edges are repeated.
- A **path** is a trail where no two nodes are repeated.
- A **cycle** is a trail where the start and end node are the same.

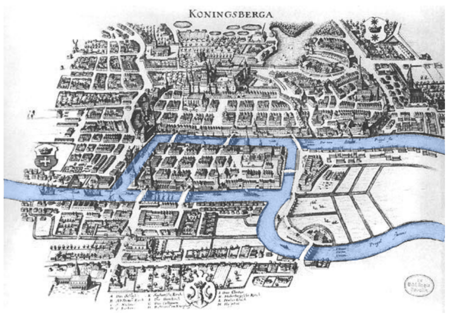
For example, in the graph figure 4.1

- (B, A, C, D, A, B) is a walk but not a cycle since the edge joining nodes A and B is traversed twice;
- (A, D, C, A, B) is a trail but not a walk since node A is visited twice;
- (C, A, A, D, C) is a cycle.

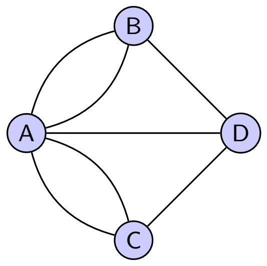
Definition 4.4: Parent node

A **parent node** is a node that precedes the current node in a walk.

The origins of graph theory can be traced to swiss mathematician Leonard Euler (1707 – 1783) where in 1736 he used it to solve the problem of whether it is possible to walk over the seven bridges of Königsberg crossing each one only once (now known as an ‘Eulerian path’). To solve this problem Euler created the first mathematical graph where the map of Königsberg (figure 4.2(b)) is represented as a graph (figure 4.2). The seven bridges are the edges of the graph which join four nodes which are the land masses that are separated by the river Pregel where node A is the island in the centre, node B is the north of the city, node C is the south of the city and node D is the west of the city.



(a) Map of Königsberg



(b) Graph for the bridge of Königsberg problem

Figure 4.2: The bridges of Königsberg problem.

In his solution, Euler noticed that in order to traverse an edge just once, zero or two nodes of the graph must have an odd degree. Since all four nodes in figure 4.2 have an odd degree then it is not possible to create a path that crosses each of the seven bridges just once.

4.1.2 The adjacency matrix

A convenient way to mathematically describe a graph is using a **adjacency matrix**. The adjacency matrix is a square matrix A where the element a_{ij} denotes the number of edges connecting the nodes V_i to V_j. Where an edge exits a node, loops round and re-enters the same node the convention is to assign a value of 2 for that edge. For example the adjacency matrix for the graph shown in figure 4.2 is

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{bmatrix} 0 & 2 & 2 & 1 \\ 2 & 0 & 0 & 1 \\ 2 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}.$$

Theorem 4.1: Properties of the adjacency matrix

The adjacency matrix A has the following properties:

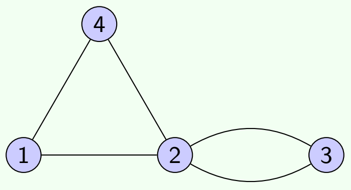
- The adjacency matrix for a non-directed graph is symmetric;
- If there are no loops in the graph then the main diagonal elements are all zero;
- The number of non-zero elements on the main diagonal is the number of loops in the graph;
- The value of $[A^n]_{ij}$ is equal to the number of walks of length n from v_i to v_j.
- The degree of node v_i is equal to the sum of row i or column i.

Definition 4.5: Connected graphs

A graph G is said to be **connected** if a path exists between any two nodes in G. If a graph is not connected then it is **disconnected**.

Example 4.1

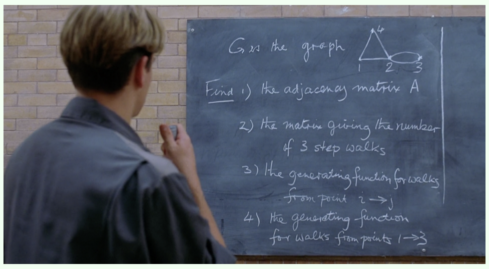
Given the graph below:



find:

1. the adjacency matrix A;
2. the matrix giving the number of 3 step walks.[a]

[a]This problem is from the film *Good Will Hunting* where M.I.T. professor Gerald Lambeau poses this problem to his class of graduate students claiming that it is "an advanced Fourier system" (there is no such thing as a 'Fourier system') and that whomever solves it will have their name mentioned in M.I.T. Tech.


G is the graph

Find 1) the adjacency matrix A

2) the matrix giving the number of 3 step walks

3) the generating function for walks from point $i \to j$

4) the generating function for walks from points $1 \to 3$

Solution:

1.

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

2. The number of walks of length 3 is given by A^3

$$A^2 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 2 & 1 \\ 1 & 6 & 0 & 1 \\ 2 & 0 & 4 & 2 \\ 1 & 1 & 2 & 2 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 1 \\ 0 & 2 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 2 & 1 & 2 & 1 \\ 1 & 6 & 0 & 1 \\ 2 & 0 & 4 & 2 \\ 1 & 1 & 2 & 2 \end{bmatrix} = \begin{bmatrix} 2 & 7 & 2 & 3 \\ 7 & 2 & 12 & 7 \\ 2 & 12 & 0 & 2 \\ 3 & 7 & 2 & 2 \end{bmatrix}$$

I.e., there are 12 walks of length 3 between nodes 2 and 3.

4.1.3 Weighted graphs

A **weighted graph** is a graph $G = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of nodes and $E = \{w(u, v) :$ a weighted edge between nodes u and $v\}$. $w(u, v)$ is a numerical **weight** assigned to an edge joining nodes u and v (figure 4.3).

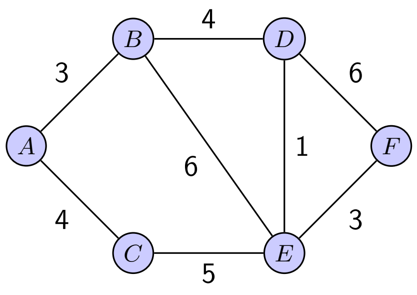


Figure 4.3: A weighted graph.

The elements in the adjacency matrix for a weighted graph are the weights assigned to the edges, i.e., $[A]_{ij} = w(i, j)$. For example, the adjacency matrix for the weighted graph in figure 4.3 is

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 3 & 4 & 0 & 0 & 0 \\ 3 & 0 & 0 & 4 & 6 & 0 \\ 4 & 0 & 0 & 0 & 5 & 0 \\ 0 & 4 & 0 & 0 & 1 & 6 \\ 0 & 6 & 5 & 1 & 0 & 3 \\ 0 & 0 & 0 & 6 & 5 & 0 \end{bmatrix} \end{array}$$

4.1.4 Directed graphs

A **directed graph** (also known as a **digraph**) is a graph where the edges have a direction. The edges of a directed graphs are represented diagrammatically by arrows indicating the direction of the arrow (figure 4.4).

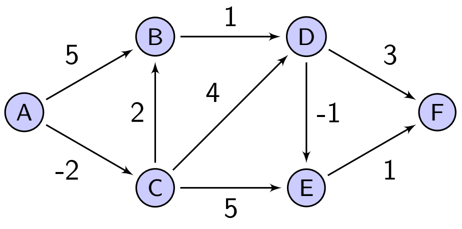


Figure 4.4: A directed graph.

The adjacency matrix for a directed graph may not be symmetric. For example, the edge joining nodes A to B has weight $w(A,B) = 5$ but there is no edge joining B to A so $w(B,A) = 0$. The adjacency matrix for the weighted graph shown in figure 4.4 is

$$A = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 5 & -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 4 & 5 & 0 \\ 0 & 0 & 0 & 0 & -1 & 3 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

4.1.5 Trees

A **tree** is a connected graph, G, that satisfies the following:

- Node G contains no cycles;
- adding an edge between any two nodes in G a cycle is created;
- removing any edge in G creates a disconnected graph;
- any two nodes can be connected by a single path;
- if G has n nodes then it has $n-1$ edges.

For example, the graph in figure 4.5 is a tree.

Definition 4.6: Spanning tree

A **spanning tree** is a tree that contains all of the nodes in a graph.

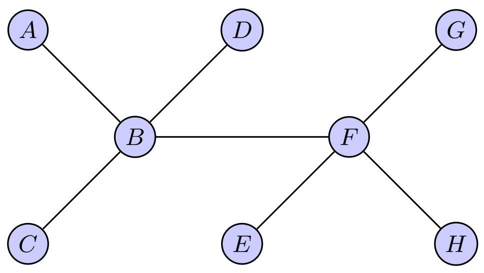


Figure 4.5: A tree.

Definition 4.7: Minimum spanning tree

A **minimum spanning tree** is a spanning tree for a weighted graph where the sum of the weights of the edges are the minimum possible for the tree.

4.2 Depth-first and breadth-first search

Depth-first and search are algorithms for traversing a graph and producing spanning trees.

4.2.1 Depth-first search (DFS)

The concept behind depth-first search is that a path is traversed as far as possible until either a leaf node is reached or all nodes adjacent to the current nodes have been visited. Then we return to the last node that has an unvisited adjacent node and traverse as far as possible again. We continue to do this until all nodes in the graph have been visited. The spanning tree is defined by the edges that are traversed to visited each node.

Given an n-node connected graph $G = (V, E)$ the depth-first search proceeds as follows:

1. Assume all nodes are classified as 'unvisited' and create an empty list called $\mathrm{closedList}$ which will contain the visited nodes.
2. Create an empty list called $\mathrm{openList}$ will contain the nodes that are to be checked. Put the start node into $\mathrm{openList}$.
3. Create an n-element list $\mathrm{parent} = [\varnothing, \ldots \varnothing]$ which contains null entries. This will contain the parent nodes for the nodes added in $\mathrm{openList}$.
4. If $\mathrm{openList} \neq \emptyset$ remove the **last** node u and check whether it is in $\mathrm{closedList}$ (i.e., has it already been visited). If u is not in $\mathrm{closedList}$ add it to $\mathrm{closedList}$ and add u to the spanning tree.
5. Append all of the nodes adjacent to u that are not in $\mathrm{closedList}$ to $\mathrm{openList}$ and set their parent node to u.
6. Repeat steps 3 and 4 until the $\mathrm{openList}$ is empty.

For example, consider the graph in figure 4.6 with node A as the start node. Initialise the $\mathrm{closedList}$ and $\mathrm{openList}$ lists

$$\mathrm{closedList} = \emptyset, \qquad \mathrm{openList} = (A).$$

1. Remove node A from $\mathrm{openList}$. Node A is not in $\mathrm{closedList}$ so we add it. Node B is adjacent to node A and not in $\mathrm{closedList}$ so we add it to $\mathrm{closedList}$ and to the spanning tree.

$$\mathrm{closedList} = \{A\}, \qquad \mathrm{openList} = (B).$$

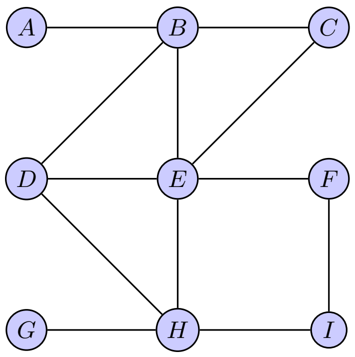
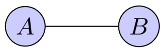


Figure 4.6: Graph used in the search algorithms.

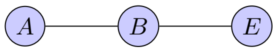
2. Remove node B from openList. Node B is not in closedList so we add it to closedList and to the spanning tree (node A is the parent node). Nodes A, C, D and E are adjacent to node B but node A is in closedList so we add nodes C, D, E to openList

$$\text{closedList} = \{A, B\}, \qquad \text{openList} = (C, D, E).$$



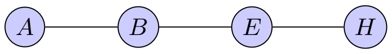
3. Remove node E from openList. Node E is not in closedList so we add it to closedList and the spanning tree (node B is the parent node). Nodes B, C, D, F and H are adjacent to node E but node B is in closedList so we add nodes C, D, F, H to openList

$$\text{closedList} = \{A, B, E\}, \qquad \text{openList} = (C, D, C, D, F, H).$$



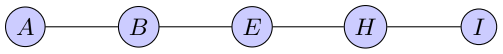
4. Remove node H from openList. Node H is not in closedList so add it to closedList and to the spanning tree (node E is the parent node). Nodes D, E, G and I are adjacent to node H but node E is in closedList so add nodes D, G, I to openList

$$\text{closedList} = \{A, B, E, H\}, \qquad \text{openList} = (C, D, C, D, F, D, G, I).$$



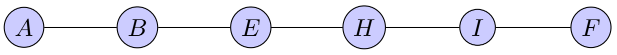
5. Remove node I from openList. I is not in closedList so add it to closedList and to the spanning tree (node H is the parent node). Nodes F and H are adjacent to I but H is in closedList so add F to openList

$$\text{closedList} = \{A, B, E, H, I\}, \qquad \text{openList} = (C, D, C, D, F, D, G, F).$$



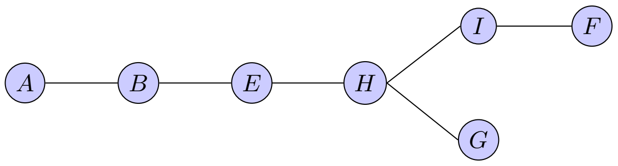
6. Remove F from openList. F is not in closedList so add it to closedList and to the spanning tree (node I is the parent node). Nodes E and I are adjacent to F but both are in closedList so we cannot add them

$$\text{closedList} = \{A, B, E, H, I, F\}, \qquad \text{openList} = (C, D, C, D, F, D, G).$$



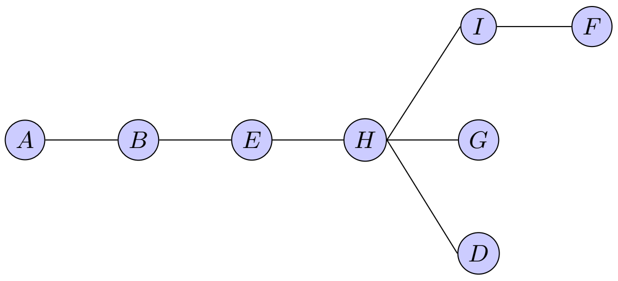
7. Remove G from openList. G is not in closedList so add it to closedList and to the spanning tree (node H is the parent node). Node H is adjacent to G but is in closedList so we cannot add it to openList

$$\text{closedList} = \{A, B, E, H, I, F, G\}, \qquad \text{openList} = (C, D, C, D, F, D).$$



8. Remove D from openList. D is not in closedList so add it to closedList and to the spanning tree (node H is the parent node). Nodes B, E and H are adjacent to D but are in closedList so we cannot add them

$$\text{closedList} = \{A, B, E, H, I, F, G, D\}, \qquad \text{openList} = (C, D, C, D, F).$$



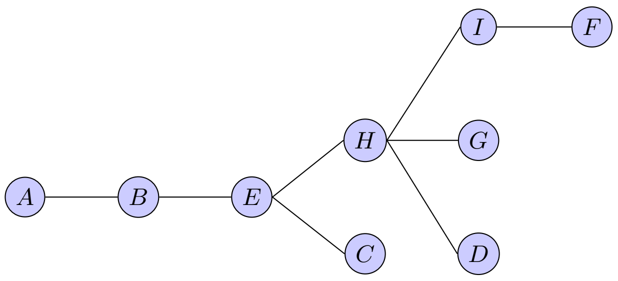
9. Remove F from openList. F is in closedList so we do not add it to closedList or the spanning tree.

$$\text{closedList} = \{A, B, E, H, I, F, G, D\}, \qquad \text{openList} = (C, D, C, D).$$

10. Remove D from openList. D is in closedList so we do not add it to closedList or the spanning tree.

$$\text{closedList} = \{A, B, E, H, I, F, G, D\}, \qquad \text{openList} = (C, D, C).$$

11. Remove C from openList. C is not in closedList so we add it to closedList and to the spanning tree (node E is the parent node). All nodes are now in closedList so the algorithm terminates.

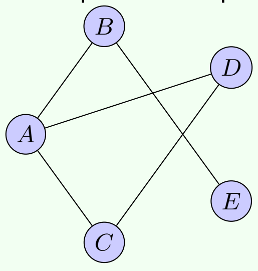


The ordering of the nodes by the depth-first search algorithm starting at node A is

$$\text{closedList} = \{A, B, E, H, I, F, G, D, C\}.$$

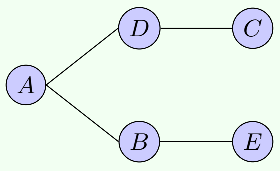
Example 4.2

Starting at node A, use depth-first search to produce a spanning tree for the graph below.



Solution:

$$
\begin{aligned}
\text{closedList} &= \emptyset, & \text{openList} &= [A],\\
\text{closedList} &= \{A\}, & \text{openList} &= [B, C, D],\\
\text{closedList} &= \{A, D\}, & \text{openList} &= [B, C, C],\\
\text{closedList} &= \{A, D, C\}, & \text{openList} &= [B, C],\\
\text{closedList} &= \{A, D, C, B\}, & \text{openList} &= [E],\\
\text{closedList} &= \{A, D, C, B, E\}, & \text{openList} &= \emptyset.
\end{aligned}
$$



The pseudocode for the depth-first search algorithm is shown in algorithm 8.

Algorithm 8 Pseudocode for the depth-first search algorithm

```
function DFS($A$, $startnode$)
    $n \leftarrow$ number of rows in $A$
    closedList $\leftarrow$ empty list
    openList $\leftarrow [startnode]$
    parent $\leftarrow [-1, \dots, -1]$                ▷ parent is an $n$-element array
    while openList $\neq \emptyset$ do
        $u \leftarrow$ last element in openList
        remove $u$ from openList
        if $u \notin$ closedList then
            append $u$ to closedList
            for $v = 1, \dots, n$ do
                if $G(u, v) \neq 0$ and $G(u, v) \notin$ closedList then
                    append node $v$ to openList
                    parent($v$) $\leftarrow u$
                end if
            end for
        end if
    end while
    return closedList and parent
end function
```

4.2.2 Python and MATLAB code for depth-first search

Python and MATLAB code for depth-first search are shown in listings 4.1 and 4.2. Both codes define a function called `DFS` which takes inputs of the adjacency matrix `A` and the index of the starting node `startnode` and returns the node order in `closedList` and the parent nodes for each node in `parent`.

Listing 4.1: Python code for depth-first search.

```
def DFS(A, startnode):

    # Initialise lists
    closedList, openList, parent = [], [startnode], [-1] * len(A)
    while openList:

        # Get last node from the open list
        u = openList.pop()

        # Check if current node has not already been visited
        if u not in closedList:

            # Add current node to closed list
            closedList.append(u)

            # Look for unvisited nodes adjacent to the current node
            for v in range(len(A)):
                if A[u,v] != 0 and v not in closedList:
                    openList.append(v)
                    parent[v] = u

    return closedList, parent
```

Listing 4.2: MATLAB code for depth-first search.

```
function [closedList, parent] = DFS(A, startnode)

% Initialise lists
closedList = []; openList = [startnode]; parent = -ones(1, size(A, 1));

while isempty(openList) == false

    % Get last node from open list
    u = openList(end);
    openList(end) = [];

    % Check if current node has not already been visited
    if ismember(u, closedList) == false

        % Append current node node to closed list
        closedList = [closedList, u];

        % Look for unvisited nodes adjacent to the current node
        for v = 1 : size(A, 1)
            if A(u, v) ~= 0 && ismember(v, closedList) == false
                openList = [openList, v];
            end
        end
    end
end

end
```

4.2.3 Breadth-first search

Breath-first search**breadth-first** is another algorithm for traversing a graph and producing a spanning tree. Starting at a chosen node in the graph, we visit all nodes adjacent to the current node that has not yet been visited. For each new visited node we repeat this process until all nodes have been visited.

Given an n-node connected graph $G = (V, E)$ and a start node, depth-first search proceeds as follows:

1. Assume all nodes are classified as 'unvisited' and create an empty list $\mathrm{closedList}$ which will contain the visited nodes.
2. Create an empty list $\mathrm{openList}$ will contain a queue of nodes to be checked. Put the start node into $\mathrm{openList}$.

3. Create an n-element list $\text{parent} = [\varnothing, \ldots \varnothing]$ which contains null entries. This will contain the parent nodes for the nodes added to openList.
4. If $\text{openList} \neq \emptyset$ remove the **first** node u from openList and check whether it is in closedList. If u is not in closedList add it to closedList and add u to the spanning tree.
5. Append all of the nodes v adjacent to u that are not in closedList to openList and, if this is the first time a node has been added to openList, set their parent node to u.
6. Repeat steps 3 and 4 until the openList is empty.

Note that the breadth-first search algorithm is very similar to the depth-first search algorithm, the difference being is that we remove the first node from the list. For example, consider applying breath-first search to the graph in figure 4.6 with A as the start node.

1. Node A is the start node so we initialise the closedList and openList lists

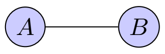
$$\text{closedList} = \emptyset, \qquad \text{openList} = (A).$$

2. Remove node A from openList. Node A is not in closedList so add it to closedList and to the spanning tree. Node B is adjacent to node A and not in closedList so it is added to openList

$$\text{closedList} = \{A\}, \qquad \text{openList} = (B).$$

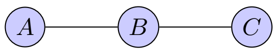
3. Remove node B from openList. Node B is not in closedList so add it to closedList and to the spanning tree (node A is the parent node). Nodes A, C, D and E are adjacent to node B but node A is in closedList so add nodes C, D, E to openList

$$\text{closedList} = \{A, B\}, \qquad \text{openList} = (C, D, E).$$



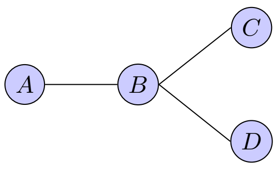
4. Remove node C from openList. Node C is not in closedList so add it to closedList and to the spanning tree (node B is the parent node). Nodes B and E are adjacent to node C but node B is in closedList so add node E to openList

$$\text{closedList} = \{A, B, C\}, \qquad \text{openList} = (D, E, E).$$



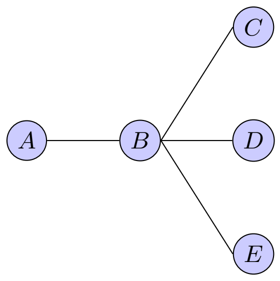
5. Remove node D from openList. Node D is not in closedList so add it to closedList and to the spanning tree (node B is the parent node). Nodes B, E and H are adjacent to node D but node B is in closedList so add nodes E, H to openList

$$\text{closedList} = \{A, B, C, D\}, \qquad \text{openList} = (E, E, E, H).$$



6. Remove E from openList. Node E is not in closedList so add it to closedList and to the spanning tree (node B is the parent node). Nodes B, C, D, F and H are adjacent to node E but nodes B, D and C are in closedList so add nodes F, H to openList

$$\text{closedList} = \{A, B, C, D, E\}, \qquad \text{openList} = (E, E, H, F, H).$$



7. Remove node E from openList. Node E is in closedList so do not add to closedList.

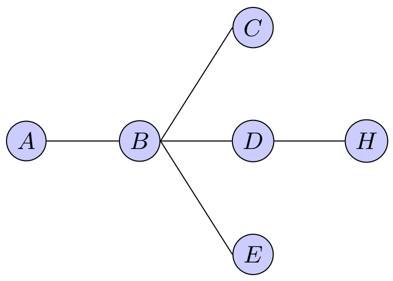
$$\text{closedList} = \{A, B, C, D, E\}, \qquad \text{openList} = (E, H, F, H).$$

8. Remove node E from openList. Node E is in closedList so do not add to closedList.

$$\text{closedList} = \{A, B, C, D, E\}, \qquad \text{openList} = (H, F, H).$$

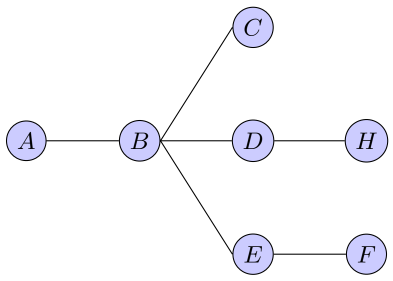
9. Remove node H from openList. Node H is not in closedList so add it to closedList and to the spanning tree (node D is the parent node). Nodes D, E, G and I are adjacent to node H but nodes D and E are in closedList so add nodes G, I to openList

$$\text{closedList} = \{A, B, C, D, E, H\}, \qquad \text{openList} = (F, H, G, I).$$



10. Remove node F from openList. Node F is not in closedList so add it to closedList and to the spanning tree. Nodes E and I are adjacent to node F but node E is in closedList so add node I to openList

$$\text{closedList} = \{A, B, C, D, E, H, F\}, \qquad \text{openList} = (H, G, I, I).$$

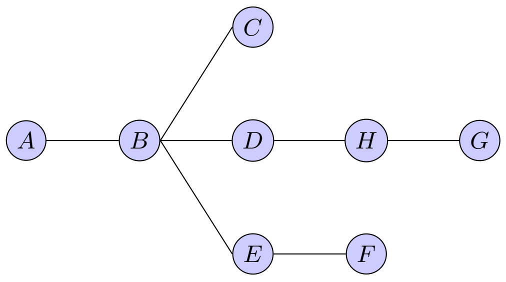


11. Remove node H from openList. H is in closedList so do not add to closedList.

$$\text{closedList} = \{A, B, C, D, E, H, F\}, \qquad \text{openList} = (G, I, I).$$

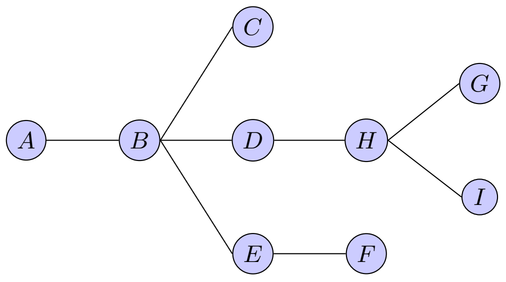
12. Remove node G from openList. Node G is not in closedList so add it to closedList and to the spanning tree (node H is the parent node). Node H is adjacent to node G but is already in closedList so do not add to openList

$$\text{closedList} = \{A, B, C, D, E, H, F, G\}, \qquad \text{openList} = (I, I).$$



13. Remove Node I from openList. Node I is not in closedList so add it to closedList and to the spanning tree (node H is the parent node). Nodes F and H are adjacent to node I but are already in closedList so do not add to openList. All nodes are now in closedList so the algorithm terminates.

$$\text{closedList} = \{A, B, C, D, E, H, F, G, I\}.$$

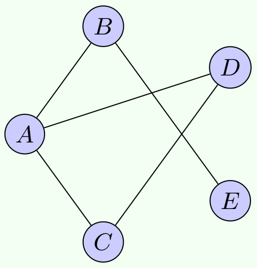


Algorithm 9 Pseudocode for the breadth-first search algorithm

```
function BFS($A$, $startnode$)
    $n \leftarrow$ number of rows in $A$
    $\text{closedList} \leftarrow \emptyset$
    $\text{openList} \leftarrow [startnode]$
    $\text{parent} \leftarrow [-1, \ldots, -1]$                    ▷ parent is an $n$-element array
    while $\text{openList} \neq \emptyset$ do
        $u \leftarrow$ last node in openList
        remove $u$ from openList
        if $v \notin \text{closedList}$ then
            append $u$ to closedList
            for $v = 1 \ldots n$ do
                if $A(u, v) \neq 0$ and $v \notin \text{closedList}$ then
                    append node $v$ to openList
                    if $\text{parent}(v) = -1$ then
                        $\text{parent}(v) \leftarrow u$    ▷ update parent($v$) the first time $v$ is added to openList
                    end if
                end if
            end for
        end if
    end while
    return closedList
end function
```

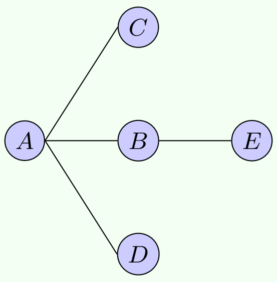
Example 4.3

Starting at node A, use breadth-first search to produce a spanning tree for the graph below.



Solution:

$$
\begin{aligned}
&\text{closedList} = \emptyset, && \text{openList} = [A], \\
&\text{closedList} = \{A\}, && \text{openList} = [B, C, D], \\
&\text{closedList} = \{A, B\}, && \text{openList} = [C, D, E], \\
&\text{closedList} = \{A, B, C\}, && \text{openList} = [D, E, D], \\
&\text{closedList} = \{A, B, C, D\}, && \text{openList} = [E, D], \\
&\text{closedList} = \{A, B, C, D, E\}, && \text{openList} = [D].
\end{aligned}
$$



4.3 Shortest path problems

The **shortest path problem** is the problem of finding the shortest path between two nodes on a graph. Let graph $G = (V, E)$ be defined by a set of nodes, $V = \{v_1, v_2, \dots, v_n\}$ and a set of weighted edges, $E = \{w(u, v)$: weighted edge joining u to $v\}$. The shortest path between the two nodes v_0 and v_m is defined as the path $P = (v_0, v_1, v_2, \dots, v_m)$ that minimises the sum of edges that form P, i.e.,

$$f(P) = \sum_{i=1}^{m-1} w(v_i, v_{i+1}).$$

When each edge has a weight of $w(u, v) = 1$ then the shortest path is the one with the fewest edges.

For example consider the weighted graph in figure 4.3. There are numerous possible paths from node A to node F. The path $P_1 = (A, B, D, F)$ gives $f(P_1) = 3 + 4 + 6 = 13$ but the path $P_2 = (A, C, E, F)$ gives $f(P_2) = 4 + 5 + 3 = 12$ so its easy to see that path P_2 is the shorter path. The question is whether P_2 is the shortest possible path from A to F. Since this graph contains a small number of nodes it is possible for us to see that the that $P = (A, B, D, E, F)$ that gives $f(P) = 3 + 4 + 1 + 3 = 11$ is the shortest path. The question is, if we were presented with a weighted graph that has thousands of nodes how would we be able to determine which is the shortest path?

4.4 Dijkstra's algorithm

Dijkstra's algorithm was developed by Dutch computer scientist Edsger Dijkstra (1930 – 2002) (Dijkstra 1959). Given a graph $G = (V, E)$ with n nodes and the start and end nodes $\textit{startnode}$ and $\textit{endnode}$ then the steps of the algorithm are as follows:

1. Create empty $\mathrm{openList}$ and $\mathrm{closedList}$ lists and put the start node into $\mathrm{openList}$.
2. Create an n-element array $d = [\infty, \ldots, \infty]$ and set $d(startnode) = 0$. The d array is used to contain the shortest distance for the path from the start node to each node, hence why the start node has a distance of zero.
3. Create another n-element array $\mathrm{parent} = [\varnothing, \ldots, \varnothing]$ with null entries for each element. The parent array is used to contain the parent node, the node on the shortest path that precedes each node, for the nodes in $\mathrm{openList}$.
4. Choose the node u from $\mathrm{openList}$ which has the smallest distance value. Move u from $\mathrm{openList}$ to $\mathrm{closedList}$.
5. If $u = endnode$ exit the algorithm.
6. For each node v that is adjacent to u and not in $\mathrm{closedList}$. If v is not in $\mathrm{openList}$ and the distance through from the start through u is less than $d(v)$ then update $d(v) \leftarrow d(u) + w(u, v)$ and $\mathrm{parent}(v) = u$. If v is not in $\mathrm{openList}$ add it.
7. Repeat steps 4 to 6 until the end node is reached.

The distance values now define the **minimum spanning tree** (a tree which contains all of the nodes of the graph where the sum of the weights is minimum) for G which includes the start and end nodes. The shortest path is found by initialising $P = (endnode)$ and backtracking to the start node and pre-pending the adjacent node which gives the correct shortest distance through the node.

For example, consider the implementation of Dijkstra's algorithm to the shortest path problem between nodes A and F in figure 4.7(a). Initialising $\mathrm{openList} \leftarrow \{A\}$ and using the notation $(d(node), \mathrm{parent}(node))$ to record the distance and parent node we have

iteration	A	B	C	D	E	F
0	$(0, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$

Iteration 1 (figure 4.7(b)):

- The node in $\mathrm{openList}$ with the smallest distance is node A so we move this to $\mathrm{closedList}$;
- Node B: $d(A) + w(A, B) = 0 + 5 = 5 < \infty$ so we update $d(B) = 5$ and $\mathrm{parent}(B) = A$;
- Node C: $d(A) + w(A, C) = 0 + 2 = 2 < \infty$ so we update $d(C) = 2$ and $\mathrm{parent}(C) = A$;

iteration	A	B	C	D	E	F
0	$(0, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
1	$\underline{(0, \varnothing)}$	$(5, A)$	$(2, A)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$

Here the underline $\underline{(0, \varnothing)}$ denotes that the node has been added to $\mathrm{closedList}$.

Iteration 2 (figure 4.7(c)):

- The node in $\mathrm{openList}$ with the smallest distance is node C so we move this to $\mathrm{closedList}$;
- Node B: $d(C) + w(C, B) = 2 + 2 = 4 < 5$ so we update $d(B) = 4$ and $\mathrm{parent}(B) = C$;
- Node D: $d(C) + w(C, D) = 2 + 4 = 6 < \infty$ so we update $d(D) = 6$ and $\mathrm{parent}(D) = C$;
- Node E: $d(C) + w(C, E) = 2 + 5 = 7 < \infty$ so we update $d(E) = 7$ and $\mathrm{parent}(E) = C$.

iteration	A	B	C	D	E	F
0	$(0, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
1	$\underline{(0, \varnothing)}$	$(5, A)$	$(2, A)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
2	$\underline{(0, \varnothing)}$	$(4, C)$	$\underline{(2, A)}$	$(6, C)$	$(7, C)$	$(\infty, \varnothing)$

Iteration 3 (figure 4.7(d)):

- The node in openList with the smallest distance is node B so we move this to closedList;
- Node D: $d(B) + w(B, C) = 4 + 1 = 5 < 6$ so we update $d(D) = 5$ and $\text{parent}(D) = B$.

iteration	A	B	C	D	E	F
0	$(0, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
1	$\underline{(0, \varnothing)}$	$(5, A)$	$(2, A)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
2	$\underline{(0, \varnothing)}$	$(4, C)$	$\underline{(2, A)}$	$(6, C)$	$(7, C)$	$(\infty, \varnothing)$
3	$\underline{(0, \varnothing)}$	$\underline{(4, C)}$	$\underline{(2, A)}$	$(5, B)$	$(7, C)$	$(\infty, \varnothing)$

Iteration 4 (figure 4.7(e))

- The node in openList with the smallest distance is node D so we move this to closedList;
- Node E: $d(D) + w(D, E) = 5 + 1 = 6 < 7$ so we update $d(E) = 6$ and $\text{parent}(E) = D$;
- Node F: $d(D) + w(D, F) = 5 + 3 = 8 < \infty$ so we update $d(F) = 8$ and $\text{parent}(E) = D$.

iteration	A	B	C	D	E	F
0	$(0, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
1	$\underline{(0, \varnothing)}$	$(5, A)$	$(2, A)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
2	$\underline{(0, \varnothing)}$	$(4, C)$	$\underline{(2, A)}$	$(6, C)$	$(7, C)$	$(\infty, \varnothing)$
3	$\underline{(0, \varnothing)}$	$\underline{(4, C)}$	$\underline{(2, A)}$	$(5, B)$	$(7, C)$	$(\infty, \varnothing)$
4	$\underline{(0, \varnothing)}$	$\underline{(4, C)}$	$\underline{(2, A)}$	$\underline{(5, B)}$	$(6, D)$	$(8, D)$

Iteration 5 (figure 4.7(f)):

- The node in openList with the smallest distance is node E so we move this to closedList;
- Node F: $d(E) + w(E, F) = 6 + 1 = 7 < 8$ so we update $d(F) = 7$ and $\text{parent}(F) = E$.

iteration	A	B	C	D	E	F
0	$(0, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
1	$\underline{(0, \varnothing)}$	$(5, A)$	$(2, A)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$	$(\infty, \varnothing)$
2	$\underline{(0, \varnothing)}$	$(4, C)$	$\underline{(2, A)}$	$(6, C)$	$(7, C)$	$(\infty, \varnothing)$
3	$\underline{(0, \varnothing)}$	$\underline{(4, C)}$	$\underline{(2, A)}$	$(5, B)$	$(7, C)$	$(\infty, \varnothing)$
4	$\underline{(0, \varnothing)}$	$\underline{(4, C)}$	$\underline{(2, A)}$	$\underline{(5, B)}$	$(6, D)$	$(8, D)$
5	$\underline{(0, \varnothing)}$	$\underline{(4, C)}$	$\underline{(2, A)}$	$\underline{(5, B)}$	$\underline{(6, D)}$	$(7, E)$

Iteration 6:

- The node in openList with the smallest distance is node F so we move this to closedList;
- Node F is the end node so the algorithm terminates.

The shortest path is found by backtracking through the minimal spanning tree starting at the end node which is node F so $P = (F)$. The parent node for node F is node E which is prepended to the path $P = (E, F)$. The parent node for node E is node D so $P = (D, E, F)$. Continuing to prepend the parent nodes until the start node A is reached gives

$$P = (A, C, B, D, E, F).$$

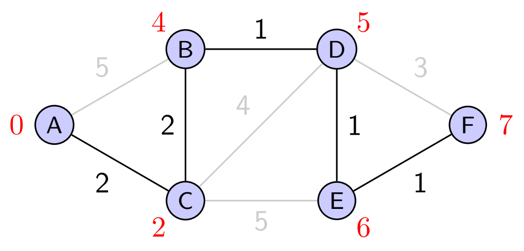
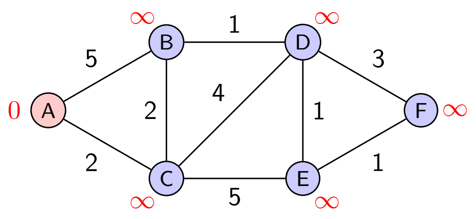
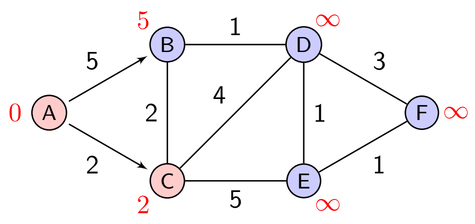


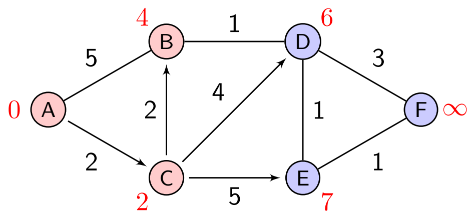
Figure 4.8: The shortest path between nodes A and F.



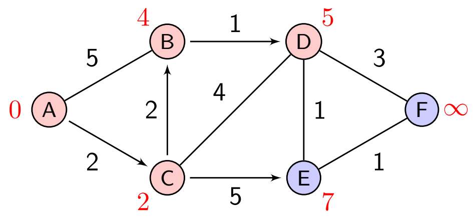
(a) Initial graph



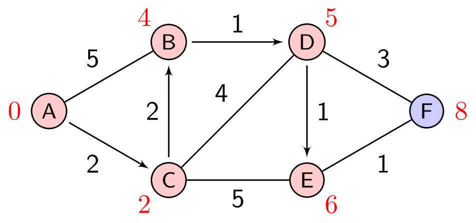
(b) iteration 1



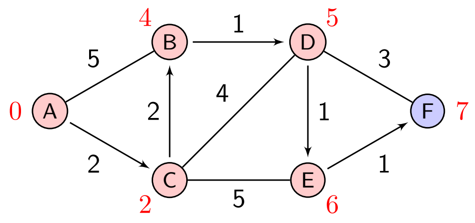
(c) iteration 2



(d) iteration 3



(e) iteration 4



(f) iteration 5

Figure 4.7: Implementation of Dijkstra's algorithm to find the shortest path from A to F.

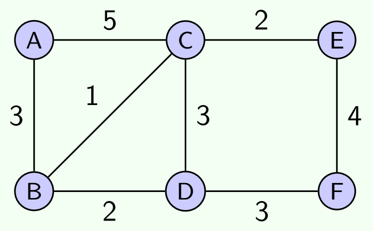
The pseudocode for Dijkstra's algorithms is shown in algorithm 10.

Algorithm 10 Pseudocode for Dijkstra's algorithm

```
function DIJKSTRA($A$, $startnode$, $endnode$)
    $n \leftarrow$ number of rows in $A$
    $\mathrm{closedList} \leftarrow$ empty list
    $\mathrm{openList} \leftarrow [startnode]$
    $d \leftarrow [\infty, \ldots, \infty]$                    ▷ $d$ and $\mathrm{parent}$ are $n$-element arrays
    $\mathrm{parent} \leftarrow [-1, \ldots, -1]$
    $d(startnode) \leftarrow 0$
    while $\mathrm{openList}$ is not empty do
        $u \leftarrow$ node in $\mathrm{openList}$ with the smallest $d$ value
        append $u$ to $\mathrm{closedList}$
        remove $u$ from $\mathrm{openList}$
        if $u = endnode$ then
            exit while loop
        end if
        for $v = 1 \ldots n$ do
            if $A(u,v) \neq 0$ and $v \notin \mathrm{closedList}$ and $d(v) > d(u) + A(u,v)$ then
                $d(v) \leftarrow d(u) + A(u,v)$
                $\mathrm{parent}(v) \leftarrow u$
                if $v \notin \mathrm{openList}$ then
                    append $v$ to $\mathrm{openList}$
                end if
            end if
        end for
    end while
    $\mathrm{path} \leftarrow [endnode]$
    $i \leftarrow endnode$
    while $\mathrm{parent}(i) \neq -1$ do                    ▷ backtrack through spanning tree
        $\mathrm{path} \leftarrow [\mathrm{parent}(i), \mathrm{path}]$          ▷ prepend $\mathrm{parent}(i)$ to path
        $i \leftarrow \mathrm{parent}(i)$
    end while
    return $\mathrm{path}$
end function
```

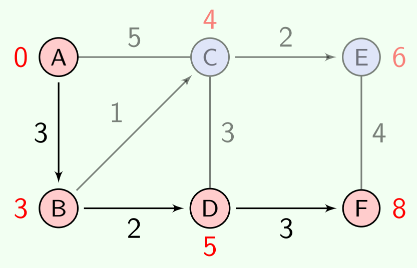
Example 4.4

Use Dijkstra's algorithm to find the shortest path between nodes A and F in the graph below.



Solution:

iteration	A	B	C	D	E	F
0	(0, $\varnothing$)	(∞, $\varnothing$)	(∞, $\varnothing$)	(∞, $\varnothing$)	(∞, $\varnothing$)	(∞, $\varnothing$)
1	$\underline{(0, \varnothing)}$	(3, A),	(5, A)	(∞, $\varnothing$)	(∞, $\varnothing$)	(∞, $\varnothing$)
2	$\underline{(0, \varnothing)}$	$\underline{(3, A)}$	(4, B)	(5, B)	(∞, $\varnothing$)	(∞, $\varnothing$)
3	$\underline{(0, \varnothing)}$	$\underline{(3, A)}$	$\underline{(4, B)}$	(5, B)	(6, C)	(∞, $\varnothing$)
4	$\underline{(0, \varnothing)}$	$\underline{(3, A)}$	$\underline{(4, B)}$	$\underline{(5, B)}$	(6, C)	(3, D)
5	$\underline{(0, \varnothing)}$	$\underline{(3, A)}$	$\underline{(4, B)}$	$\underline{(5, B)}$	$\underline{(6, C)}$	(3, D)
6	$\underline{(0, \varnothing)}$	$\underline{(3, A)}$	$\underline{(4, B)}$	$\underline{(5, B)}$	$\underline{(6, C)}$	$\underline{(3, D)}$



4.5 The Bellman-Ford algorithm

The **Bellman-Ford algorithm** is an algorithm for finding the shortest path on a weighted graph. Where the Bellman-Ford algorithm differs from Dijkstra's algorithm is that the Bellman-Ford algorithm can handle negative weights. Named after American mathematicians Richard Bellman (1920 – 1984) and Lester Randolph Ford Jr. (1927 - 2017) who discovered it separately in 1958 and 1956 respectively (Bellman 1958; Ford 1956)

Given an n-node graph G and the start and end nodes, $startnode$ and $endnode$, the steps of the algorithm are as follows:

1. Create an n-element array $d \leftarrow [\infty, \dots, \infty]$ and set $d(startnode) \leftarrow 0$. The d array is used to contain the shortest distance for the path from the start node to each node.
2. Create another n-element array $\mathrm{parent} \leftarrow [\varnothing, \dots, \varnothing]$ with null entries in each element. The parent array is used to contain the parent nodes for each node.
3. For each edge $w(u, v)$ in the graph, if the distance to v through u is less than the current distance $d(v)$ then we update $d(v) = d(u) + w(u, v)$ and $\mathrm{parent}(v) = u$. This is called **relaxing** the distances.
4. Perform step 3 for a total of $n - 1$ times.

After $n - 1$ iterations of the relaxation process all distances will be the minimum from the start node and define the minimum spanning tree. The shortest path is found using backtracking in the same way as for Dijkstra's algorithm.

For example, consider the implementation of the Bellman-Ford algorithm to the shortest path problem

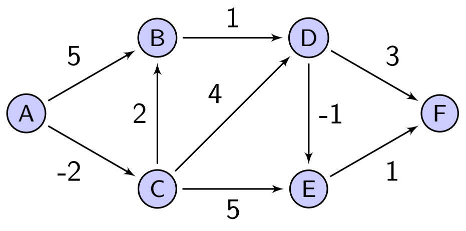


Figure 4.9: A directed graph.

between nodes A and F in figure 4.9. We initialise the distances to ∞ except for the start node A (figure 4.10(a))

step	A	B	C	D	E	F
0	(0,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)

Iteration 1:

- Node A is the start node so we don't update $d(A)$;
- Node B: $d(A) + w(A, B) = 0 + 5 = 5 < \infty$ and $d(C) + w(C, B) = \infty + 2 = \infty$ so we update $d(B) = 5$ and $\text{parent}(B) = A$;
- Node C: $d(A) + w(A, C) = 0 - 2 = -2 < \infty$ so we update $d(C) = -2$ and $\text{parent}(C) = A$;
- Node D: $d(B) + w(B, D) = \infty + 1 = \infty$ and $d(C) + w(C, D) = \infty + 4 = \infty$ so we do not update $d(D)$ or $\text{parent}(E)$;
- Node E: $d(C) + w(C, E) = \infty + 5 = \infty$ and $d(D) + w(D, E) = \infty - 1 = \infty$ so we do not update $d(E)$ or $\text{parent}(E)$;
- Node F: $d(D) + w(D, F) = \infty + 3 = \infty$ and $d(E) + w(E, F) = \infty + 1 = \infty$ so we do not update $d(F)$ or $\text{parent}(F)$.

iteration	A	B	C	D	E	F
0	(0,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
1	(0,$\varnothing$)	(5,A)	(-2,A)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)

After the first step the distances for all paths at most 1 edge from the start node are found. This means we don't need to check nodes A or C again.

Iteration 2:

- Node B: $d(A) + w(A, B) = 0 + 5 = 5 \not< 5$ and $d(C) + w(C, B) = -2 + 2 = 0 < 5$ so we update $d(B) = 0$ and $\text{parent}(B) = C$;
- Node D: $d(B) + w(B, D) = 5 + 1 = 6 < \infty$ and $d(C) + w(C, D) = -2 + 4 = 2 < \infty$ so we update $d(D) = 2$ and $\text{parent}(D) = C$;
- Node E: $d(C) + w(C, E) = -2 + 5 = 3 < \infty$ and $d(D) + w(D, E) = \infty - 1 = \infty$ so we update $d(E) = 3$ and $\text{parent}(E) = C$;
- Node F: $d(D) + w(D, F) = \infty + 3 = \infty$ and $d(E) + w(E, F) = \infty + 1 = \infty$ so we do not update $d(F)$ or $\text{parent}(F)$.

iteration	A	B	C	D	E	F
0	(0,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
1	(0,$\varnothing$)	(5,A)	(-2,A)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
2	(0,$\varnothing$)	(0,C)	(-2,A)	(2,C)	(3,C)	(∞,$\varnothing$)

After the second iteration the distances for all paths at most 2 edges from the start node are found so we no longer need to check node B again.

Iteration 3:

- Node D: $d(B) + w(B, D) = 0 + 1 = 1 < 2$ and $d(C) + w(C, D) = -2 + 4 = 2 \not< 2$ therefore we update $d(D) = 1$ and $\text{parent}(D) = B$;
- Node E: $d(C) + w(C, E) = -2 + 5 = 3 \not< 3$ and $d(D) = 2 - 1 = 1 < 3$ therefore we update $d(E) = 1$ and $\text{parent}(E) = C$;
- Node F: $d(D) + w(D, F) = 2 + 3 = 5 < \infty$ and $d(E) + w(E, F) = 3 + 1 = 4 < \infty$ therefore we update $d(F) = 4$ and $\text{parent}(F) = E$.

iteration	A	B	C	D	E	F
0	(0,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
1	(0,$\varnothing$)	(5,A)	(-2,A)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
2	(0,$\varnothing$)	(0,C)	(-2,A)	(2,C)	(3,C)	(∞,$\varnothing$)
3	(0,$\varnothing$)	(0,C)	(-2,A)	(1,B)	(1,C)	(4, E)

Now we know the distances to all paths at most 3 edges from node A so we no longer need to check node D again.

Iteration 4:

- Node E: $d(C) + w(C, E) = -2 + 5 = 3 > 1$ and $d(D) = 1 - 1 = 0 < 1$ therefore update $d(E) = 0$ and $\text{parent}(E) = D$;
- Node F: $d(D) + w(D, F) = 1 + 3 = 4 \not< 4$ and $d(E) + w(E, F) = 1 + 1 = 2 < 4$ therefore we update $d(F) = 2$ and the parent node is unchanged.

iteration	A	B	C	D	E	F
0	(0,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
1	(0,$\varnothing$)	(5,A)	(-2,A)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
2	(0,$\varnothing$)	(0,C)	(-2,A)	(2,C)	(3,C)	(∞,$\varnothing$)
3	(0,$\varnothing$)	(0,C)	(-2,A)	(1,B)	(1,C)	(4, E)
4	(0,$\varnothing$)	(0,C)	(-2,A)	(1,B)	(0,D)	(2, E)

Now we know the distances to all paths at most 4 edges from node A so we no longer need to check node E.

Iteration 5:

- Node F: $d(D) + w(DF) = 1 + 3 = 4$ and $d(E) + w(EF) = 0 + 1 = 1$ therefore we update $d(F) = 1$ and the parent node is unchanged.

step	A	B	C	D	E	F
0	(0,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
1	(0,$\varnothing$)	(5,A)	(-2,A)	(∞,$\varnothing$)	(∞,$\varnothing$)	(∞,$\varnothing$)
2	(0,$\varnothing$)	(0,C)	(-2,A)	(2,C)	(3,C)	(∞,$\varnothing$)
3	(0,$\varnothing$)	(0,C)	(-2,A)	(1,B)	(1,C)	(4, E)
4	(0,$\varnothing$)	(0,C)	(-2,A)	(1,B)	(0,D)	(2, E)
5	(0,$\varnothing$)	(0,C)	(-2,A)	(1,B)	(0,D)	(1, E)

Since we have done $n - 1$ steps of the algorithm none of the distances can be updated so we stop here. The shortest path is found by back tracking from the end node and prepending the parent nodes in the same way as used in Dijkstra's algorithm which gives $P = (A, C, B, D, E, F)$ (figure 4.11).

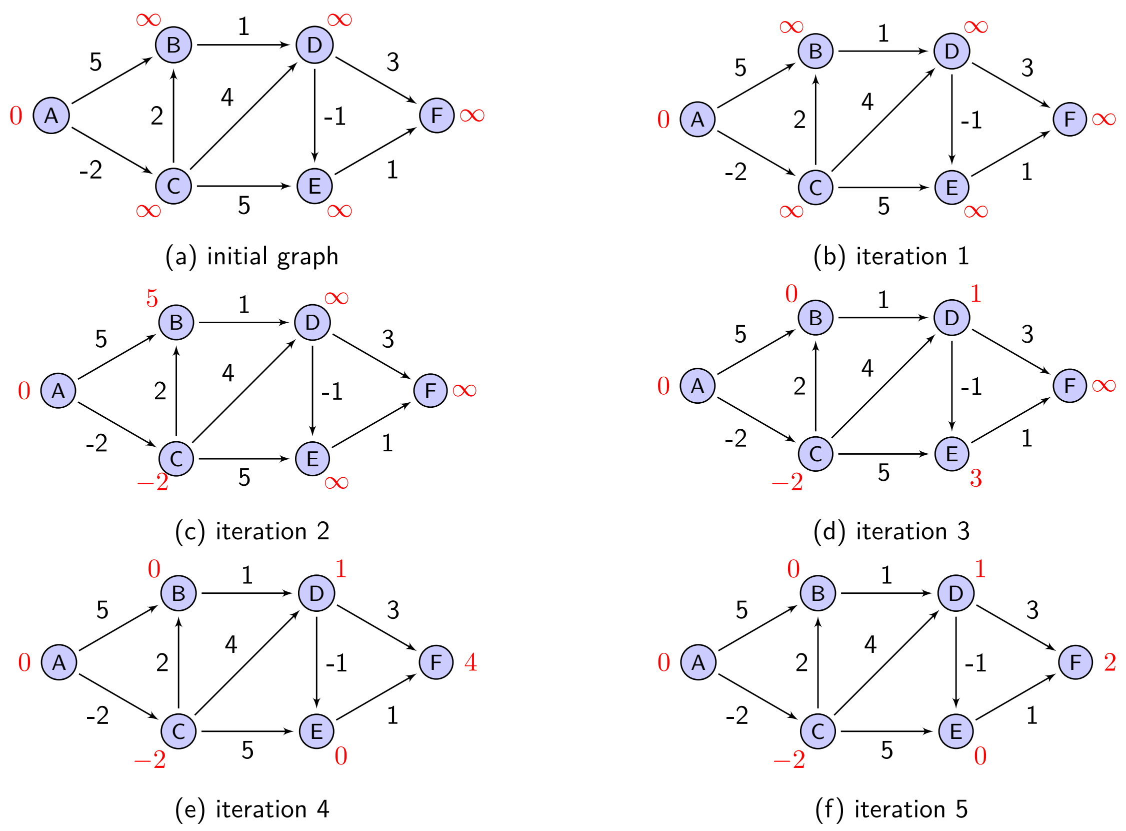


Figure 4.10: Implementation of the Bellman-Ford algorithm to find the shortest path from A to F on a directed graph with negative weights.

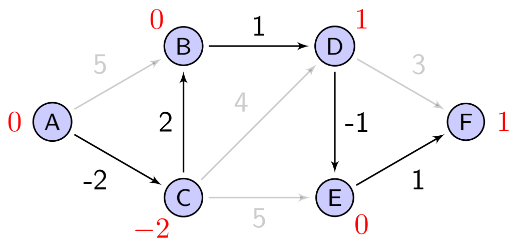


Figure 4.11: The shortest path from node A to node F found using the Bellman-Ford algorithm.

4.5.1 Negative cycles

A **negative cycle** is a cycle where the weights sum to a negative number. A directed graph which contains a negative cycle does not have a shortest path because the negative cycle will cause a shortest path algorithm to enter a cycle. The directed graph shown in figure 4.12 contains the negative cycle (B, C, D).

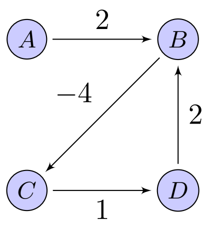


Figure 4.12: A directed graph with a negative cycle.

To find out whether a directed graph has a negative cycle we simply attempt to relax the distance values for the node once again. If one of the distances can be reduced further then we have a negative cycle. This is because after $n-1$ iterations all distances should be minimised, so if we can relax a distance value then we must have a negative cycle.

The pseudocode for the Bellman-Ford algorithm is shown in algorithm 11.

Algorithm 11 Pseudocode for the Bellman-Ford algorithm

```
function BELLMANFORD($A$, $startnode$, $endnode$)
    $n \leftarrow$ number of rows in $A$
    $d \leftarrow [\infty, \dots, \infty]$
    $\text{parent} \leftarrow [-1, \dots, -1]$                ▷ $d$ and parent are $n$ element arrays
    $d(startnode) \leftarrow 0$
    for $k = 1, \dots, n-1$ do                              ▷ iterate $n-1$ times
        for $u = 1 \dots n$ do
            for $v = 1 \dots n$ do
                if $A(u,v) \neq 0$ and $d(v) > d(u) + A(u,v)$ then
                    $d(v) \leftarrow d(u) + A(u,v)$          ▷ relax distances
                    $\text{parent}(v) \leftarrow u$          ▷ update parent node
                end if
            end for
        end for
    end for
    for $u = 1 \dots n$ do
        for $v = 1 \dots n$ do                              ▷ check for negative cycles
            if $d(v) > d(u) + A(u,v)$ then
                return error                                ▷ negative cycle found
            end if
        end for
    end for
    $\text{path} \leftarrow [endnode]$     ▷ backtrack from $endnode$ through the parent nodes
    $i \leftarrow endnode$
    while $\text{parent}(i) \neq -1$ do                     ▷ stop when no parent exists
        $\text{path} \leftarrow [\text{parent}(i), \text{path}]$   ▷ prepend $\text{parent}(i)$ to path
        $i \leftarrow \text{parent}(i)$
    end while
    return path
end function
```

4.6 The A* algorithm

The **A* algorithm** (Hart et al. 1968), pronounced as "A star", is an algorithm for finding the shortest path on a weighted graph. Unlike Dijkstra's algorithm and the Bellman-Ford algorithm where all nodes in a graph are considered before finding the shortest path, the A* algorithm attempts to find the shortest path by only considering nodes which are close to the shortest path. To do this, the algorithm makes a choice as to which adjacent node to move to using an estimate of the cost required to extend the path to the end node and then chooses the adjacent node with the smallest cost.

The estimated cost for node n, $f(n)$, is calculated using

$$f(n) = g(n) + h(n)$$

where $g(n)$ is the distance from the start node[1] and $h(n)$ is a **heuristic**[2] that estimates the cost of the cheapest path from node n to the end node.

Given a weighted graph G and the start and end nodes, $startnode$ and $endnode$, the steps of the A* algorithm are:

1. Create two lists, openList and closedList, where openList will contain the vertices that have been evaluated by the heuristic function and closedList contains those nodes that have been visited. Put the start node $startnode$ into openList.
2. Create n-element lists $\text{parent} = [\varnothing, \ldots \varnothing]$, $g = [\infty, \ldots, \infty]$ and $f = [\infty, \ldots, \infty]$. parent contains the parent nodes, g contains the shortest distance from the start node and f contains the estimated cost.
3. Pick node u from the openList list with the smallest value of $f(u)$ and move this to the closedList list.
4. If $u = endnode$ exit the algorithm.
5. For each node v that is adjacent to u and not in closedList. If the distance from the start node through u is less than $g(v)$ then update $\text{parent}(v) = u$, $g(v) \leftarrow g(u) + w(u, v)$ and $f(v) \leftarrow g(v) + h(v)$.

   If v is not in openList then add it.
6. Repeat steps 3 and 4 until the end node has been moved to closedList.

The shortest path is found using back tracking through the parent list in the same way as in Dijkstra's and the Bellman-Ford algorithm.

For example, consider the implementation of the A* algorithm to the shortest between nodes A and G in the graph shown in figure 4.13. Here the heuristic values have been given for each node.

We start by putting the start node A into openList and initialising $g(A) = 0$ and $f(A) = h(A) = 7$.

Iteration 1 (figure 4.14(a)):

- The node in openList with the minimum $f(n)$ value is node A so we move this to closedList;
- Node B is not in openList so we add it and set $\text{parent}(B) = A$, $g(B) = g(A) + w(A, B) = 0 + 4 = 4$ and $f(B) = g(B) + h(B) = 4 + 5 = 9$;
- Node C is not in openList so we add it and set $\text{parent}(C) = A$, $g(C) = g(B) + w(A, C) = 0 + 6 = 6$ and $f(C) = g(C) + h(C) = 6 + 6 = 12$;
- Node D is not in openList so we add it and set $\text{parent}(D) = A$, $g(A) = g(A) + w(A, D) = 0 + 4 = 4$ and $f(D) = g(D) + h(D) = 4 + 4 = 8$:

[1]So far in this chapter I have used $d(n)$ to denote the distance from the start node which is the game as $g(n)$. Here I have chosen to use $g(n)$ as this is what is commonly used when presenting the A* algorithm.

[2]A heuristic is a method for giving a solution to a problem that does not necessarily give the optimum solution but one that is good enough for our needs.

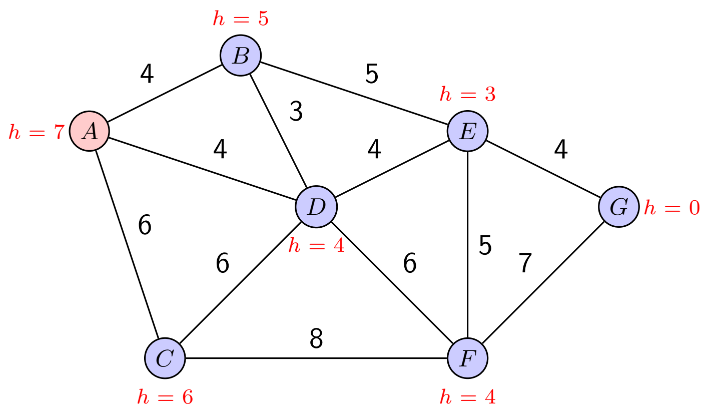


Figure 4.13: A weighted graph with heuristic values for each node.

Iteration 2 (figure 4.14(a)):

- The node in $\mathrm{openList}$ with the minimum $f(n)$ value is node D so we move this to $\mathrm{closedList}$;
- Node A is in $\mathrm{closedList}$ so we do nothing;
- Node B is already in open. $g(D)+w(D,B)=4+3=7>4$ so we do not update $\mathrm{parent}(B)$, $g(B)$ or $f(B)$;
- Node C is already in $\mathrm{openList}$. $g(D)+w(D,C)=4+6=10>6$ so we do not update $\mathrm{parent}(C)$, $g(C)$ or $f(C)$;
- Node E is not in $\mathrm{openList}$ so we add it and set $\mathrm{parent}(E)=D$, $g(E)=g(D)+w(D,E)=4+4=8$ and $f(E)=g(E)+h(E)=8+3=11$;
- Node F is not in $\mathrm{openList}$ so we add it and set $\mathrm{parent}(F)=D$, $g(F)=g(D)+w(D,F)=4+6=10$ and $f(F)=g(F)+h(F)=10+4=14$;

Iteration 3 (figure 4.14(a)):

- The node in $\mathrm{openList}$ with the minimum $f(n)$ value is node B so we move this to $\mathrm{closedList}$;
- Node A and D are in closed so we do nothing;
- Node E is already in $\mathrm{openList}$. $g(B)+w(B,E)=4+5=9>8$ so we do not update $\mathrm{parent}(E)$, $g(E)$ or $f(E)$;

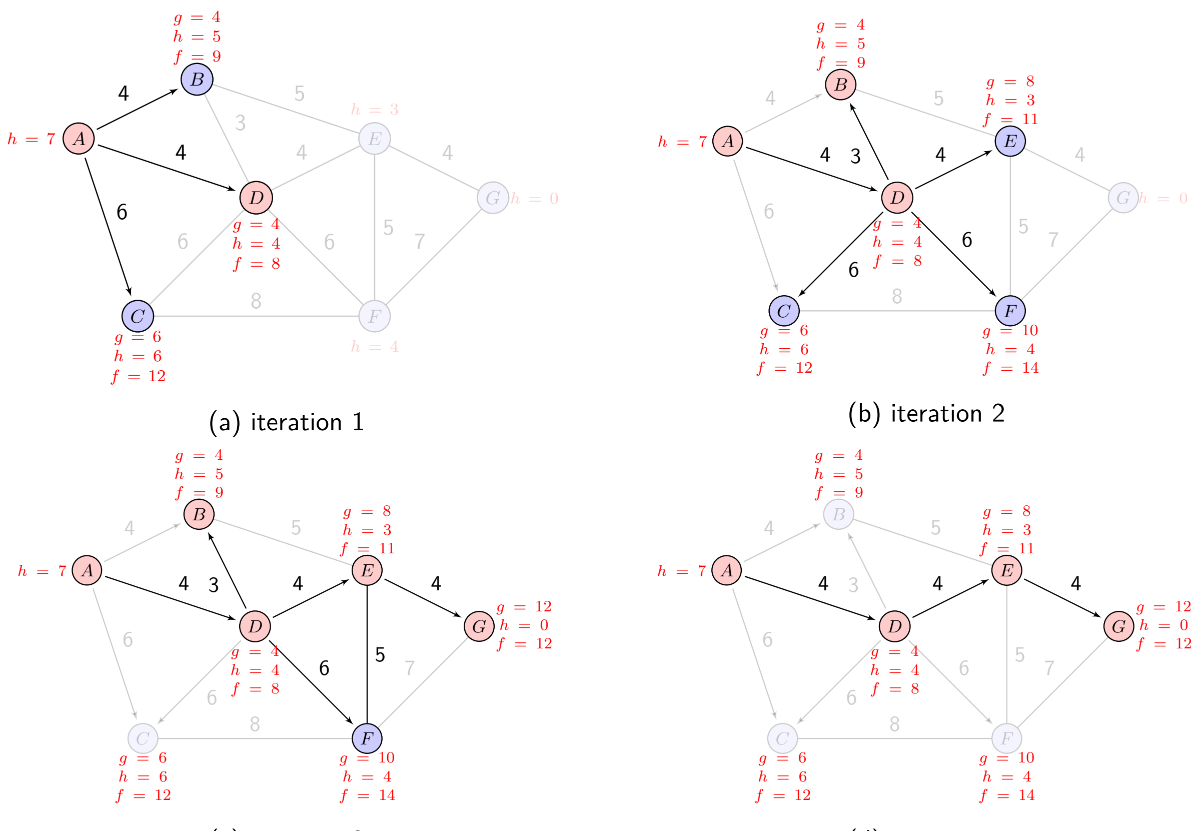
Iteration 4 (figure 4.14(a)):

- The node in $\mathrm{openList}$ with the minimum $f(n)$ value is node E so we move this to $\mathrm{closedList}$;
- Nodes B and D are already in $\mathrm{closedList}$ so we ignore these;
- Node F is already in $\mathrm{openList}$. $g(E)+w(E,F)=8+5=13>10$ so we do not update $\mathrm{parent}(F)$, $g(F)$ or $f(F)$;
- Node G is not in $\mathrm{openList}$ so we add it and set $\mathrm{parent}(G)=E$, $g(G)=g(E)+w(E,G)=8+4=12$ and $f(G)=g(G)+h(G)=12+0=12$.

Iteration 5:

- The node in $\mathrm{openList}$ with the minimum $f(n)$ value is node G so we move this to $\mathrm{closedList}$;
- Node G is the end node so the algorithm terminates.

Back tracking from G using the parent nodes gives the shortest path $P=(A,D,E,G)$.



(a) iteration 1 (b) iteration 2

(c) iteration 3 (d) iteration 4

Figure 4.14: Implementation of the A* algorithm to find the shortest path between nodes A and G.

The pseudocode for the A* algorithm is shown in algorithm 12.

4.6.1 Heuristics

One of the problems with implementing the A* algorithm is how do we calculate the value of the heuristic $h(n)$? The heuristic is the cost of extending the path to the end node but since the algorithm adds one node at a time we do not know what nodes will be selecting in future steps so we do not know what this cost will be.

Some of the possible solutions for calculating the heuristic are:

- **Exact heuristic** – $h(n)$ is calculated for all possible paths from each node to the end node is calculated. Whilst this is possible for graphs will a small number of nodes the computational cost soon becomes prohibitive for graphs with relatively modest number of nodes.
- **Manhattan distance** – where nodes are positioned due to some location metric, e.g., the location of a road junction on a map or a position in a virtual world, we calculates $h(n)$ are the sum of the distance in the x and y directions. If the start and end nodes have co-ordinates (x_s, y_s) and (x_e, y_e) respectively then the heuristic is calculated using

$$h(n) = |x_e - x_s| + |y_e - y_s|.$$

  This method is named after the block like structure of roads in Manhattan in New York.
- **Euclidean distance** – the straight line distance between the location of two nodes. If the start and end nodes have co-ordinates (x_s, y_s) and (x_e, y_e) respectively then the heuristic is calculated using

$$h(n) = \sqrt{(x_e - x_s)^2 + (y_e - y_s)^2}.$$

Algorithm 12 Pseudocode for the A* algorithm

```
function ASTAR($A$, $startnode$, $endnode$)
    $n \leftarrow$ number of rows in $A$
    $\mathrm{openList} \leftarrow \emptyset$
    $\mathrm{closedList} \leftarrow \emptyset$
    $g \leftarrow [\infty, \dots, \infty]$                    ▷ $g$, $f$ and $\mathrm{parent}$ are $n$-element arrays
    $f \leftarrow [\infty, \dots, \infty]$
    $\mathrm{parent} \leftarrow [-1, \dots, -1]$
    $g(start) \leftarrow 0$
    $f(start) \leftarrow 0$
    while $\mathrm{openList} \neq \emptyset$ do
        $u \leftarrow$ node in $\mathrm{openList}$ with the smallest $f$ value
        append $u$ to $\mathrm{closedList}$ and remove $u$ from $\mathrm{openList}$
        if $u = endnode$ then
            exit while loop
        end if
        for $v = 1 \dots n$ do
            if $A(u,v) \neq 0$ and $v \notin \mathrm{closedList}$ and $g(v) > g(u) + A(u,v)$ then
                $\mathrm{parent}(v) \leftarrow u$
                calculate heuristic $h(v)$
                $g(v) \leftarrow g(u) + A(u,v)$
                $f(v) \leftarrow g(v) + h(v)$
                if $v \notin \mathrm{openList}$ then
                    append $v$ to $\mathrm{openList}$
                end if
            end if
        end for
    end while
end function
```

4.7 Applications of shortest path problems

4.7.1 Map directions

The most common application of the A* algorithm is finding directions from one point on a map to another. Consider a map of a city, we can represent the map as a graph where each road junction is a node and the roads between junctions the edges between nodes. The weight associated with each edge could be the distance between the junctions or the time taken to travel the distance depending on whether we require the shortest route in space or time. Given a starting location and a destination, we can apply the A* algorithm to the graph to give the best route.

Consider figure 4.15 which shows a map of the area of Manchester between the John Dalton Building and Piccadilly train station. Overlaid onto the map is a graph with the node A located at the John Dalton building and node P located at Piccadilly train station. The remaining nodes are located at road junctions and the co-ordinates of each node relative to node A are shown next to each node.

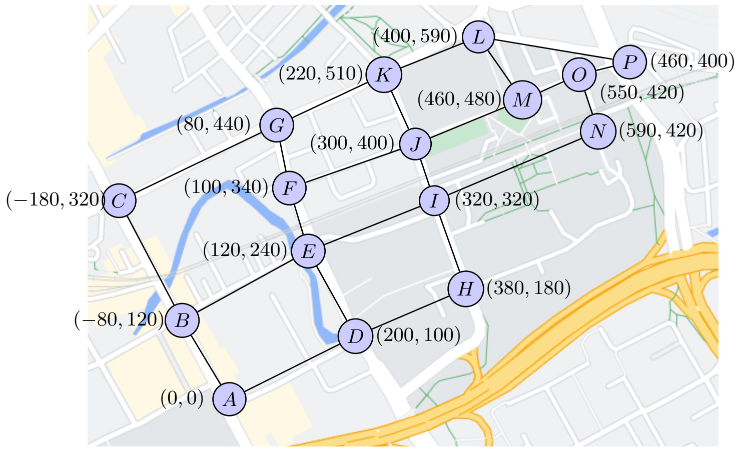
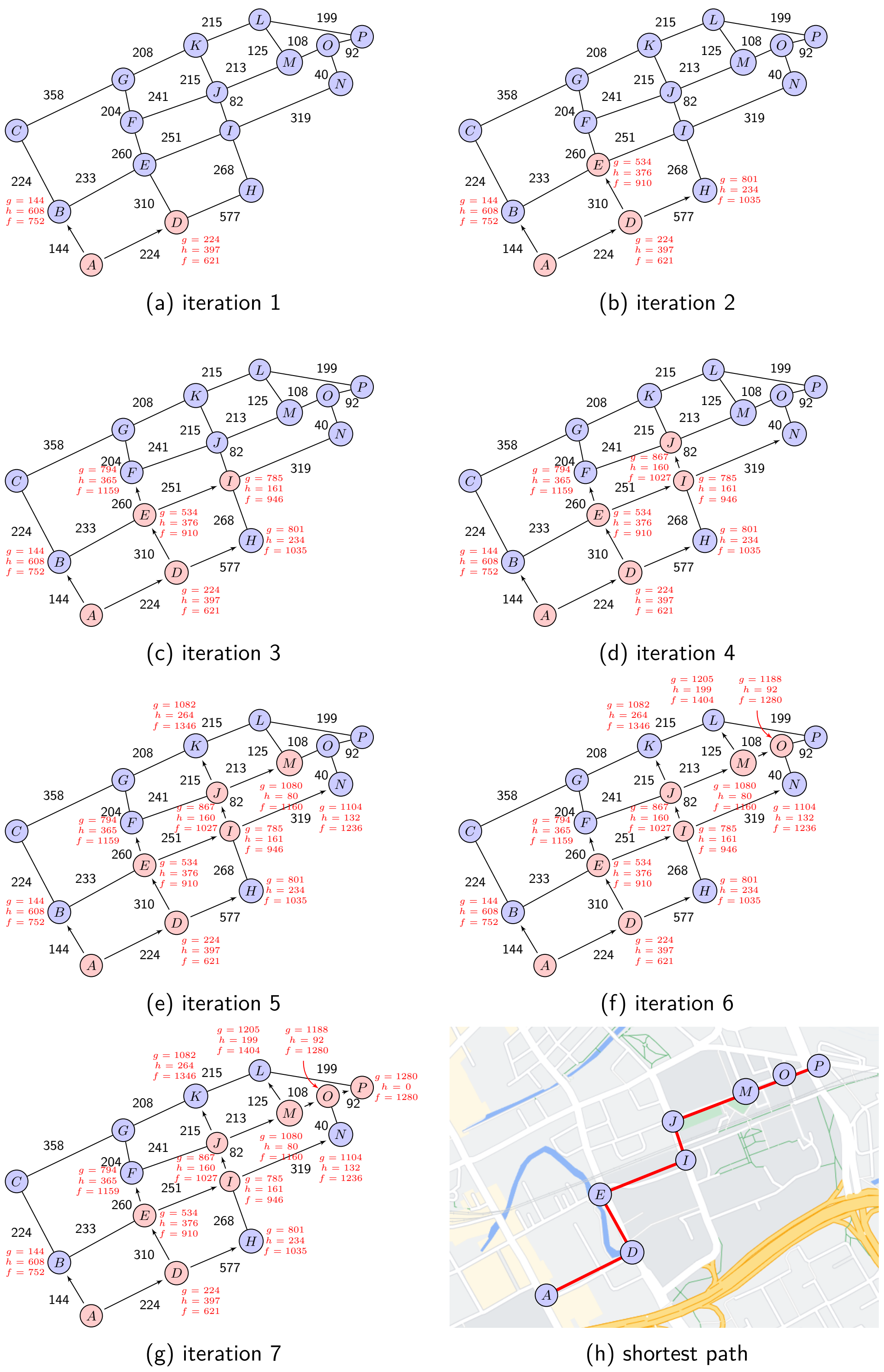


Figure 4.15: <caption>

Applying the A* algorithm to this graph using the Euclidean distance to node P as the heuristic results in the graphs shown in figure 4.16.



(a) iteration 1 (b) iteration 2

(c) iteration 3 (d) iteration 4

(e) iteration 5 (f) iteration 6

(g) iteration 7 (h) shortest path

Figure 4.16: Implementing the A* algorithm to find the shortest path between nodes A to P.

4.7.2 Path finding

The A* algorithm is commonly used in computer games in path finding for Non-Player Characters (NPCs). Consider a situation where a NPC wants to move to a position in a virtual world where there are obstacles in the way (figure 4.17). By dividing the world into discrete areas we can form a graph where each area is represented by a node in the graph. The edges join the nodes in the eight directions left, right, top, bottom, top-left, top-right, bottom-left, bottom-right, with the exception of nodes adjacent to an obstacle or boundary of the world. The weights for the edges are either 1 for an edge parallel to the horizontal or vertical axis and $\sqrt{2} \approx 1.4$ for diagonal edges.

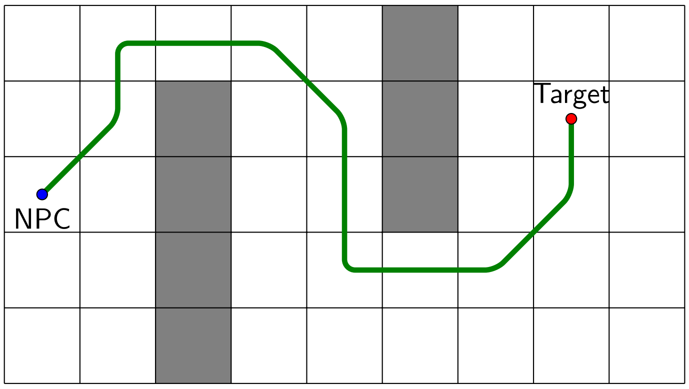


Figure 4.17: The path a non-play character will take to reach a target.

Consider the implementation of the A* algorithm for the path finding problem shown in figure 4.17 with $h(n)$ calculated using the Manhattan distance. The start node has co-ordinates $(1, 3)$ and is moved to $\text{closedList} = \{(1, 3)\}$. The start node has 5 adjacent nodes with co-ordinates $(1, 4)$, $(2, 4)$, $(2, 3)$, $(2, 2)$ and $(1, 2)$.

- $(1, 4)$: $g(1, 4) = 0 + 1 = 1$, $h(1, 4) = |8 - 1| + |4 - 4| = 7$, $f(1, 4) = 1 + 7 = 8$;
- $(2, 4)$: $g(2, 4) = 0 + 1.4 = 1.4$, $h(2, 4) = |8 - 2| + |4 - 4| = 6$, $f(2, 4) = 1.4 + 6 = 7.4$;
- $(2, 3)$: $g(2, 3) = 0 + 1 = 1$, $h(2, 3) = |8 - 2| + |4 - 3| = 7$, $f(2, 3) = 1 + 7 = 8$;
- $(2, 2)$: $g(2, 2) = 0 + 1.4 = 1.4$, $h(2, 2) = |8 - 2| + |4 - 2| = 8$, $f(2, 2) = 1.4 + 8 = 9.4$;
- $(1, 2)$: $g(1, 2) = 0 + 1 = 1$, $h(1, 2) = |8 - 1| + |4 - 2| = 9$, $f(1, 2) = 1 + 9 = 10$.

All of the $g(n)$ values were updated for these nodes so we record their parent node as $(1, 3)$ and add them to openList. The node in openList with the smallest $f(n)$ value is $(2, 4)$ so we move this to closedList (figure 4.18(a))

$$\text{closedList} = \{(1, 3), (2, 4)\}, \qquad \text{openList} = \{(1, 4), (2, 3), (2, 2), (1, 2)\}.$$

The node at $(2, 4)$ has 4 adjacent nodes not in closedList with co-ordinates $(1, 5)$, $(2, 5)$, $(2, 3)$ and $(1, 4)$.

- $(1, 5)$: $g(1, 5) = 1.4 + 1.4 = 2.8$, $h(1, 5) = |8 - 1| + |4 - 5| = 8$, $f(1, 5) = 2.8 + 8 = 10.8$;
- $(2, 5)$: $g(2, 5) = 1.4 + 1 = 2.4$, $h(2, 5) = |8 - 2| + |4 - 5| = 7$, $f(2, 5) = 2.4 + 7 = 9.4$;
- $(2, 3)$: $g(2, 3) = 1.4 + 1 = 2.4$ which is not less than the current value of $g(2, 3) = 1$ so $f(2, 3) = 8$ is unchanged;
- $(1, 4)$: $g(1, 4) = 1.4 + 1 = 2.4$ which is not less than the current value of $g(1, 4) = 1$ so $f(1, 4) = 8$ is unchanged.

The nodes at $(1, 5)$ and $(2, 5)$ had their $g(n)$ values updated so we update their parent node to the node at $(2, 4)$ and add them to openList. The node in openList with the smallest $f(n)$ value are nodes $(2, 3)$ and $(1, 4)$. We could select any of these to going into closedList, we will choose $(1, 4)$ (choosing $(2, 3)$

will not affect the final path) so we move this to closedList (figure 4.18(b))

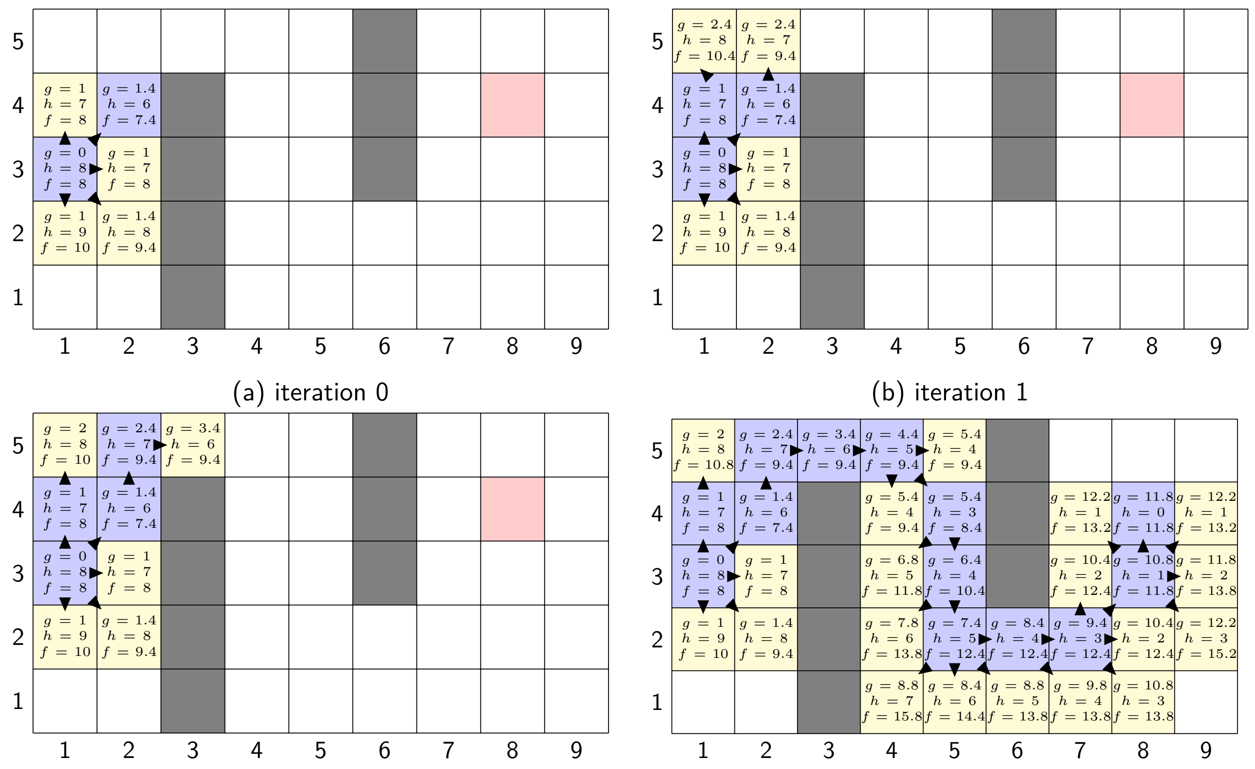
$$\text{closedList} = \{(1,3),(2,4),(1,4)\}, \qquad \text{openList} = \{(2,3),(2,2),(1,2)\}.$$

The node at $(1,4)$ has 3 adjacent nodes not in closedList with co-ordinates $(1,5)$, $(2,5)$, $(2,3)$.

- $(1,5)$: $g(1,5) = 1 + 1 = 2$ which is less than the current value $g(1,5) = 7$ so we update $f(1,5) = 2 + 7 = 9$;
- $(2,5)$: $g(2,5) = 1 + 1.4 = 2.4$, $h(2,5) = |8-2| + |4-5| = 7$, $f(2,5) = 2.4 + 7 = 9.4$;
- $(2,3)$: $g(2,3) = 1 + 1.4 = 2.4$ which is not less than the current value of $g(2,3) = 1$ so $f(2,3) = 8$ is unchanged.

The nodes at $(1,5)$ and $(2,5)$ had their $g(n)$ values updated so we update their parent node to the node at $(1,4)$ and add them to openList. The node in openList with the smallest $f(n)$ value is the node at $(2,5)$ so we move this to closedList (figure 4.18(c))

$$\text{closedList} = \{(1,3),(2,4),(1,4),(2,5)\}, \qquad \text{openList} = \{(2,3),(2,2),(1,2),(1,5),(2,5)\}.$$


(a) iteration 0 (b) iteration 1

(c) iteration 2 (d) iteration 10

Figure 4.18: Using the A* algorithm for path finding.

Continuing to apply the algorithm until the end node is reached gives

$$\text{closedList} = \{(1,3),(2,4),(1,4),(2,5),(3,5),(4,5),(5,4),(5,3),(5,2),(6,2),(7,2),(8,3),(8,4)\}$$

The path between the NPC and the target is found by back tracking through the parent nodes to give the path shown in figure 4.19. Note that the node at $(1,4)$ was in closedList but not in the path.

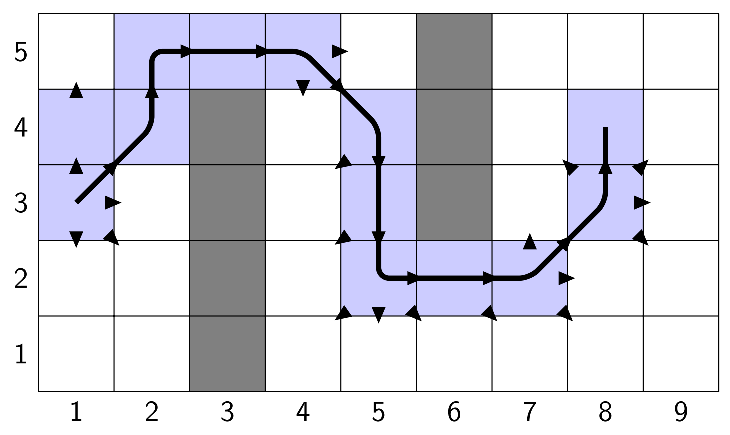


Figure 4.19: The path between the NPC and the target is found by backtracking through the parent nodes.

4.8 Tutorial exercises

Exercise 4.1. Draw the graph $G = (V, E)$ where V and E are given by

$$\begin{aligned}V &= \{A, B, C, D, E, F\},\\ E &= \{(A, C), (A, D), (A, F), (B, C), (B, D), (C, E), (D, E), (E, F)\}.\end{aligned}$$

Exercise 4.2. For the graph in exercise 4.1:

(a) Find a walk of length 7 between nodes B and C;

(b) Find a path of length 5 between nodes E and F.

(c) Find a cycle of length 4 starting and ending at node C.

Exercise 4.3. Write down the degree of each node for the graph in exercise 4.1. Does this graph have an Eulerian cycle, if so what is it?

Exercise 4.4. Write down the adjacency matrix for the graph in exercise 4.1. How many walks of length 4 are there between C and F?

Exercise 4.5. Draw all non-isomorphic (not having the same form) trees which have 5 nodes. (Hint: there are only three in total).

Exercise 4.6. A graph is defined by the following adjacency matrix

$$\begin{array}{c|ccccccccc} & A & B & C & D & E & F & G & H & I \\ \hline A & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ B & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ C & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ D & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ E & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ F & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ G & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ H & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ I & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}.$$

(a) Draw the graph;

(b) Starting at node A and adding nodes to the queue in alphabetical order, write down the order of the nodes visited using depth-first search;

(c) Draw the minimal spanning tree produced by the depth-first search.

Exercise 4.7. Repeat parts (b) and (c) from exercise 4.6 using breadth-first search.

Exercise 4.8. A weighted graph is defined by the following adjacency matrix

$$\begin{array}{c|ccccccc} & A & B & C & D & E & F & G \\ \hline A & 0 & 4 & 3 & 7 & 4 & 0 & 0 \\ B & 4 & 0 & 0 & 2 & 5 & 0 & 0 \\ C & 3 & 0 & 0 & 2 & 0 & 0 & 0 \\ D & 7 & 2 & 2 & 0 & 5 & 3 & 0 \\ E & 4 & 5 & 0 & 5 & 0 & 0 & 6 \\ F & 0 & 0 & 0 & 3 & 0 & 0 & 5 \\ G & 0 & 0 & 0 & 0 & 6 & 5 & 0 \end{array}.$$

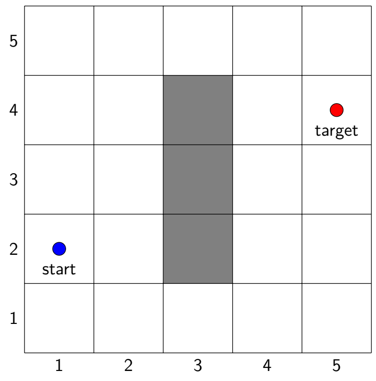
Use Dijkstra's algorithm to determine the shortest path between A and G.

Exercise 4.9. A directed graph is defined by the following adjacency matrix

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array}\begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 \\ 0 & 0 & -2 & 0 & 4 \\ 0 & 2 & 0 & 3 & 0 \\ 0 & -1 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}\end{array}$$

Use the Bellman-Ford algorithm to determine the shortest path between nodes A and E.

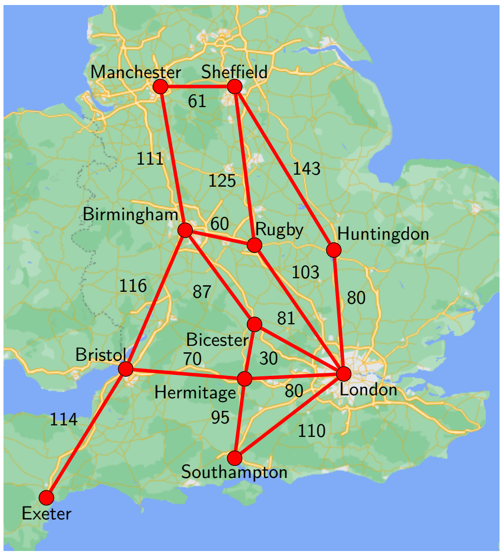
Exercise 4.10. Use the A* algorithm to determine the shortest path between the start cell and the target cell.



Exercise 4.11. Below is a map of a part of England with the towns and cities on the main road network marked using nodes of a graph and the distances, in kilometres, are given as weights on the graph. Using the Manhattan distance to calculate the heuristic, use the A* algorithm to find the shortest path between:

(a) London to Manchester;

(b) Exeter to Rugby.



The solutions are given in the appendices on page 97.

Bibliography

Bellman, R. (1958). "On a routing problem". In: *Quarterly of Applied Mathematics* 16.

Belton, D. (1998). *Karnaugh Maps - Rules of Simplification*. URL: http://www.ee.surrey.ac.uk/Projects/Labview/minimisation/karrules.html (visited on 02/03/2022).

Dijkstra, E.W. (1959). "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1, pp. 269–271.

Ford, L.R. (1956). *Network Flow Theory*. Tech. rep. RAND Corporation.

Hart, P.E., Nilsson, N.J., and Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 2.4, pp. 100–107.

Hoare, C.A.R. (1961). "Algorithm 64: Quicksort". In: *Comm. ACM* 7.4, p. 321.

Karnaugh, M. (1953). "The map method for synthsis of combinational logic circuits". In: *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics* 5.72, pp. 593–599.

Tocci, R.J., Widmer, N.S., and Moss, G.L. (2007). *Digital systems*.

Wikipedia contributors (2001). *Algorithm — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/wiki/Algorithm (visited on 02/07/2022).

Appendix A

Exercise solutions

A.1 Mathematics Fundamentals

These are the solutions to the exercises on mathematics fundamentals on page 9.

Solution 1.1.

(a) 11; (b) 26; (c) 106; (d) 137.

Solution 1.2.

(a) 1001; (b) 10111; (c) 1010000; (d) 10100011.

Solution 1.3.

(a) 16; (b) 22; (c) 2836; (d) 43981.

Solution 1.4.

(a) 18; (b) 80; (c) 1E5; (d) 59B;

Solution 1.5. $1851_{10} = 2112120_3$

Solution 1.6.

$$\begin{array}{r|ccccccc}
 & & 1 & 0 & 1 & 1 & 0 & 1 \\
 & + & 1 & 0 & 0 & 1 & 1 & 1 \\
\hline
\text{carry} & 1 & & 1 & 1 & 1 & 1 & \\
\hline
\text{sum} & 1 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}$$

Solution 1.7.

$$\begin{array}{r|ccccc}
 & 1 & \text{C} & \text{E} & 4 & \text{F} \\
 & 1 & 0 & \text{A} & 5 & 4 \\
\hline
\text{carry} & & 1 & & 1 & \\
\hline
\text{sum} & 2 & D & 8 & A & 3
\end{array}$$

Solution 1.8.

(a)

p	q	$\neg p \vee (p \wedge q)$
0	0	1
0	1	0
1	0	1
1	1	1

(b)

p	q	$\neg(p \vee q) \wedge (p \vee \neg q)$
0	0	0
0	1	0
1	0	0
1	1	1

(c)

p	q	q	$\neg(p \wedge \neg(q \vee \neg r))$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Solution 1.9.

(a)

$$\begin{aligned} & \neg p \vee (p \wedge q) & \\ \equiv\ & (\neg p \vee p) \wedge (\neg p \vee q) & \text{(distributive law)} \\ \equiv\ & 1 \wedge (\neg p \vee q) & \text{(complement law)} \\ \equiv\ & \neg p \vee q & \text{(identity law)} \end{aligned}$$

(b)

$$\begin{aligned} & \neg(p \vee q) \wedge (p \vee \neg q) & \\ \equiv\ & \neg p \wedge \neg q \wedge (p \vee \neg q) & \text{(De Morgan's law)} \\ \equiv\ & \neg p \wedge \neg q & \text{(absorption law)} \end{aligned}$$

(c)

$$\begin{aligned} & \neg(p \wedge \neg(q \vee \neg r)) & \\ \equiv\ & \neg p \vee \neg\neg(q \vee \neg r) & \text{(De Morgan's law)} \\ \equiv\ & \neg p \vee q \vee \neg r & \text{(double negation)} \end{aligned}$$

A.2 Logic Circuits

The are the solutions to the exercises on logic circuits on page 35.

Solution 2.1.

(a)

A	B	C	Q
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

(b)

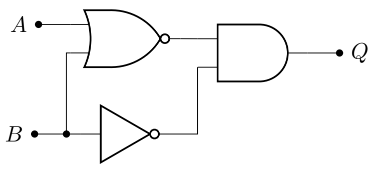
A	B	C	Q
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

(c)

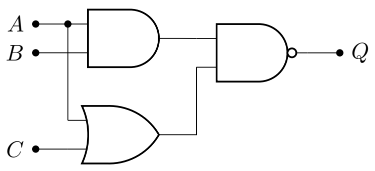
A	B	C	D	Q
0	0	1	1	0
0	1	1	0	1
1	0	1	1	0
1	1	0	0	1

Solution 2.2.

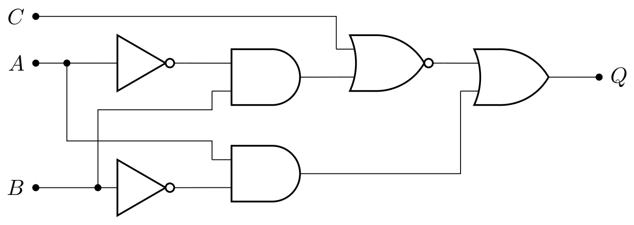
(a)



(b)



(c)



Solution 2.3.

(a)

$$
\begin{aligned}
\overline{A} \cdot (A + B) &\equiv \overline{A} \cdot A + \overline{A} \cdot B && \text{(distributivity law)} \\
&\equiv 0 + \overline{A} \cdot B && \text{(complement law)} \\
&\equiv \overline{A} \cdot B && \text{(definition of AND)}
\end{aligned}
$$

(b)

$$
\begin{aligned}
(A + A \cdot \overline{B}) \cdot (C + B \cdot C) &\equiv A \cdot C + A \cdot B \cdot C + A \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C && \text{(distributivity law)} \\
&= A \cdot C + A \cdot B \cdot C + A \cdot \overline{B} \cdot C && \text{(idempotence law).}
\end{aligned}
$$

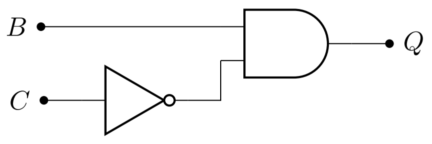
(c)

$$
\begin{aligned}
\overline{(\overline{A} + B) \cdot (A + \overline{B})} &\equiv \overline{\overline{A} + B} + \overline{A + \overline{B}} && \text{(de Morgan's law)} \\
&\equiv \overline{\overline{A}} \cdot \overline{B} + \overline{A} \cdot \overline{\overline{B}} && \text{(de Morgan's law)} \\
&\equiv A \cdot \overline{B} + \overline{A} \cdot B && \text{(double negation)} \\
&\equiv A \oplus B && \text{(definition of XOR).}
\end{aligned}
$$

(d)

$$\begin{aligned}
(A+B\cdot C)\cdot(A+\overline{B}) &\equiv A\cdot A + A\cdot\overline{B} + A\cdot B\cdot C + B\cdot\overline{B}\cdot C && \text{(distributivity law)}\\
&\equiv A + A\cdot\overline{B} + A\cdot B\cdot C + B\cdot\overline{B}\cdot C && \text{(idempotence law)}\\
&\equiv A + A\cdot\overline{B} + A\cdot B\cdot C + 0\cdot C && \text{(complement law)}\\
&\equiv A + A\cdot\overline{B} + A\cdot B\cdot C && \text{(definition of AND).}
\end{aligned}$$

Solution 2.4. $Q \equiv B\cdot\overline{C}$



Solution 2.5. We need to show that the other logic gates can be constructed from NOR gates.

- NOT gate: $\overline{A+A}\equiv\overline{A}$;
- AND gate: $\overline{\overline{A+A}+\overline{B+B}}\equiv\overline{\overline{A}+\overline{B}}\equiv\overline{\overline{A}}\cdot\overline{\overline{B}}\equiv A\cdot B$;
- OR gate: $\overline{\overline{A+B}+\overline{A+B}}\equiv\overline{\overline{A+B}}=A+B$;
- XOR gate:

$$\begin{aligned}
&\overline{\overline{\overline{A+\overline{A+B}}+\overline{\overline{A+B}+B}}+\overline{\overline{A+\overline{A+B}}+\overline{\overline{A+B}+B}}}\\
&\equiv\overline{\overline{\overline{A+\overline{A+B}}+\overline{\overline{A+B}+B}}}\\
&\equiv\overline{A+\overline{A+B}}+\overline{\overline{A+B}+B}\\
&\equiv\overline{A}\cdot\overline{\overline{A+B}}+\overline{\overline{A+B}}\cdot B\\
&\equiv\overline{A}\cdot(A+B)+\overline{B}\cdot(A+B)\\
&\equiv\overline{A}\cdot A+\overline{A}\cdot B+A\cdot\overline{B}+\overline{B}\cdot B\\
&\equiv A\cdot\overline{B}+\overline{A}\cdot B\\
&\equiv A\oplus B.
\end{aligned}$$

- NAND gate: $\overline{\overline{\overline{A+A}+\overline{B+B}}+\overline{\overline{A+A}+\overline{B+B}}}\equiv\overline{\overline{\overline{A}+\overline{B}+\overline{A}+\overline{B}}}\equiv\overline{\overline{\overline{A}+\overline{B}}}\equiv\overline{A}+\overline{B}\equiv\overline{A\cdot B}$

Therefore the NOR gate is a universal gate. □

Solution 2.6.

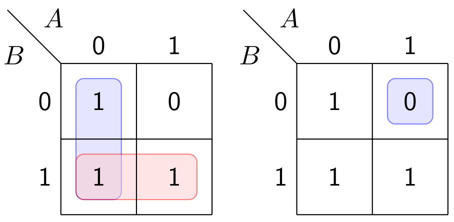
(a) SOP: $f(A,B,C)\equiv A\cdot B$;
POS: $f(A,B,C)\equiv(A+B)\cdot(A+\overline{B})\cdot(\overline{A}+B)$.

(b) SOP: $f(A,B,C)\equiv A\cdot B+A\cdot\overline{C}$;
POS: $f(A,B,C)\equiv(A+B+C)\cdot(A+B+\overline{C})\cdot(A+\overline{B}+C)\cdot(A+\overline{B}+\overline{C})+(\overline{A}+B+C)$.

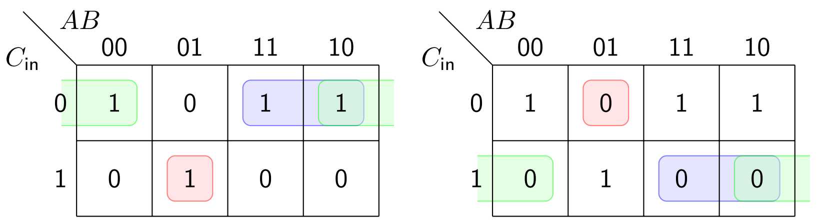
(c) SOP: $f(A,B,C)\equiv A\cdot\overline{B}\cdot C+A\cdot\overline{B}\cdot\overline{C}+A\cdot B\cdot\overline{C}+\overline{A}\cdot B\cdot\overline{C}$;
POS: POS: $f(A,B,C)\equiv(A+B+C)\cdot(A+B+\overline{C})\cdot(A+\overline{B}+\overline{C})\cdot(\overline{A}+\overline{B}+\overline{C})$.

Solution 2.7.

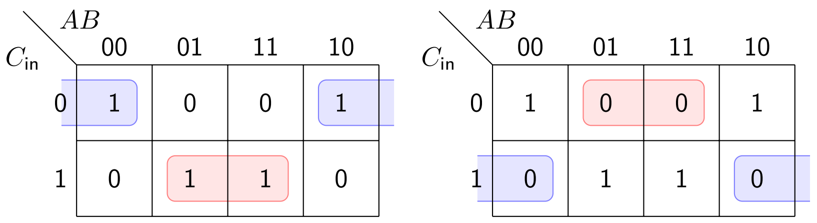
(a) SOP and POS: $f(A, B) \equiv A + \overline{B}$.



(b) SOP: $f(A, B, C) \equiv \overline{A} \cdot \overline{C} + \overline{A} \cdot C + \overline{B} \cdot C$;
POS: $f(A, B, C) \equiv (A + B + \overline{C}) \cdot (\overline{A} + \overline{B}) \cdot (\overline{A} + C)$.

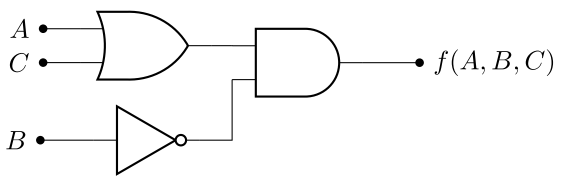


(c) SOP: $f(A, B, C) \equiv A \cdot C + \overline{A} \cdot \overline{C}$;
POS: $f(A, B, C) \equiv (A + \overline{C}) \cdot (\overline{A} + C)$.

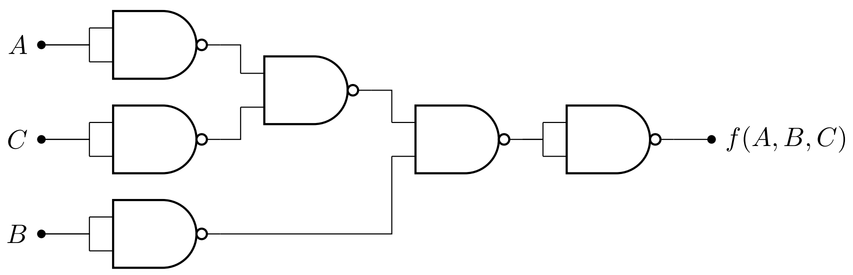


Solution 2.8.

(a)



(b)



A.3 Algorithms

These are the solutions to the exercises on graph theory on page 51.

Solution 3.1.

1.

$$\begin{array}{llllll} k=0, & a=102, & b=48, & \implies & a \leftarrow 48, & b \leftarrow \operatorname{mod}(102,48)=6, \\ k=1, & a=48, & b=6, & \implies & a \leftarrow 6, & b \leftarrow \operatorname{mod}(48,6)=0 \end{array}$$

Therefore $\gcd(48,102)=6$

2.

$$\begin{array}{llllll} k=0, & a=1027, & b=585, & \implies & a \leftarrow 585, & b \leftarrow \operatorname{mod}(1027,585)=442, \\ k=1, & a=585, & b=442, & \implies & a \leftarrow 442, & b \leftarrow \operatorname{mod}(585,442)=143, \\ k=2, & a=442, & b=143, & \implies & a \leftarrow 143, & b \leftarrow \operatorname{mod}(442,143)=13, \\ k=3, & a=143, & b=13, & \implies & a \leftarrow 13, & b \leftarrow \operatorname{mod}(143,13)=0. \end{array}$$

Therefore $\gcd(585,1027)=13$

Solution 3.2.

Pass 1:

[$\underline{5}$, $\underline{1}$, 3, 6, 2, 4]	swap 5 and 1
[1, $\underline{5}$, $\underline{3}$, 6, 2, 4]	swap 5 and 3
[1, 3, $\underline{5}$, $\underline{6}$, 2, 4]	do not swap 5 and 6
[1, 3, 5, $\underline{6}$, $\underline{2}$, 4]	swap 6 and 2
[1, 3, 5, 2, $\underline{6}$, $\underline{4}$]	swap 6 and 4
[1, 3, 5, 2, 4, 6]	

Pass 2:

[$\underline{1}$, $\underline{3}$, 5, 2, 4, 6]	do not swap 1 and 3
[1, $\underline{3}$, $\underline{5}$, 2, 4, 6]	do not swap 3 and 5
[1, 3, $\underline{5}$, $\underline{2}$, 4, 6]	swap 5 and 2
[1, 3, 2, $\underline{5}$, $\underline{4}$, 6]	swap 5 and 4
[1, 3, 2, 4, 5, 6]	

Pass 3:

[$\underline{1}$, $\underline{3}$, 2, 4, 5, 6]	do not swap 1 and 3
[1, $\underline{3}$, $\underline{2}$, 4, 5, 6]	swap 3 and 2
[1, 2, $\underline{3}$, $\underline{4}$, 5, 6]	do not swap 3 and 4
[1, 2, 3, 4, 5, 6]	

Pass 4:

[$\underline{1}$, $\underline{2}$, 3, 4, 5, 6]	do not swap 1 and 2
[1, $\underline{2}$, $\underline{3}$, 4, 5, 6]	do not swap 2 and 3

No swaps needed in last pass so sorted list is [1, 2, 3, 4, 5, 6]

Solution 3.3.

Underlined numbers denote the pivot.

[5, 1, 3, 6, 2, $\underline{4}$]	swap 5 and 2
[2, 1, 3, 6, 5, $\underline{4}$]	swap 6 and 4
[2, 1, 3, $\underline{4}$, 5, 6]	partition sub-lists [2, 1, 3] and [5, 6]
[[2, 1, $\underline{3}$], 4, [5, $\underline{6}$]]	sub-list [2, 1, 3]: swap 2 and 1, sub-list [5, 6]: do nothing
[[1, 2, $\underline{3}$], 4, [5, $\underline{6}$]]	partition sub-lists [1, 2] and [5]
[[[$\underline{1}$], [$\underline{2}$], 3], 4, [[5], 6]]	

All sub-lists only have one element so sorted list is [1, 2, 3, 4, 5, 6].

Solution 3.4.

Underlined numbers denote the midpoint.

[5, 1, $\underline{3}$, 6, 2, 4]	partition list
[[5, $\underline{1}$, 3], [6, $\underline{2}$, 4]]	partition sub-lists [5, 1, 3] and [6, 2, 4]
[[[$\underline{5}$, 1], [3]], [[$\underline{6}$, 2], [4]]]	partition sub-lists [5, 1] and [6, 2]
[[[[5], [1]], [3]], [[[6], [2]], [4]]]	merge [5] with [1] and [6] with [2]
[[[1, 5], [3]], [[2, 6], [4]]]	merge [1, 5] with [3] and [2, 6] with [4]
[[1, 3, 5], [2, 4, 6]]	merge [1, 3, 5] with [2, 4, 6]
[1, 2, 3, 4, 5, 6]	

Solution 3.5. Python:

```
def bubblesort(X):

k, swap = 0, True
while swap:
    swap = False
    for i in range(len(X) - k - 1):
        if X[i] > X[i+1]:
            X[i], X[i+1] = X[i+1], X[i]
            swap = True

    k += 1

return X


# Define list
X = [5, 1, 3, 6, 2, 4]

# Sort list
print(f'X = {X}\nsorted X = {bubblesort(X)}')
```

MATLAB:

```
% Define list
X = [5, 1, 3, 6, 2, 4]

% Sort list
bubblesort(X)


function X = bubblesort(X)

k = 0;
swap = true;
while swap
    swap = false;
    for i = 1 : length(X) - k - 1
        if X(i) > X(i + 1)
            t = X(i);
            X(i) = X(i + 1);
            X(i + 1) = t;
            swap = true;
        end
    end
    k = k + 1;
end

end
```

Solution 3.6.

(a)

```
function MINRECURSION($X$, $n$)
    if $n = 1$ then
        return $X(0)$                          ▷ assuming zero indexing ($X(0)$ is the first element)
    else
        $Xmin \leftarrow$ MINRECURSION($X, n-1$)
        if $Xmin < X(n-1)$ then
            return $Xmin$
        else
            return $X(n-1)$
        end if
    end if
```

```
        end function
```

(b) Call the function MINRECURSION$([5, 6, 4, 7, 3], 5)$

- MINRECURSION$([5, 6, 3, 7, 4], 5)$: $n > 1$ so call MINRECURSION$([5, 6, 3, 7, 4], 4)$
- MINRECURSION$([5, 6, 3, 7, 4], 4)$: $n > 1$ so call MINRECURSION$([5, 6, 3, 7, 4], 3)$
- etc.
- MINRECURSION$([5, 6, 3, 7, 4], 1)$: $n = 1$ so return $Xmin = 5$
- MINRECURSION$([5, 6, 3, 7, 4], 2)$: $Xmin = 5 < X(1) = 6$ so return $Xmin = 5$
- MINRECURSION$([5, 6, 3, 7, 4], 3)$: $Xmin = 5 > X(2) = 4$ so return $Xmin = 4$
- MINRECURSION$([5, 6, 3, 7, 4], 4)$: $Xmin = 4 < X(3) = 7$ so return $Xmin = 4$
- MINRECURSION$([5, 6, 3, 7, 4], 5)$: $Xmin = 4 > X(4) = 3$ so return $Xmin = 3$

(c) Python:

```python
def minrecursion(X, n):

if n == 0:
    return X[0]
else:
    Xmin = minrecursion(X, n - 1)
    if Xmin < X[n-1]:
        return Xmin
    else:
        return X[n-1]


print(f'Xmin = {minrecursion([5, 6, 7, 4, 3], 5)}')
```

MATLAB:

```matlab
fprintf('Xmin = %i', minrecursion([5, 6, 4, 7, 3], 5))


function Xmin = minrecursion(X, n)

if n == 1
    Xmin = X(1);
else
    Xmin = minrecursion(X, n - 1);
    if Xmin > X(n)
        Xmin = X(n);
    end
end

end
```

Solution 3.7.

(a) $\frac{1}{2}n^2 + \frac{1}{2}n$;

(b) $\frac{1}{3}n^3 + \frac{1}{3}n^2 - \frac{5}{6}n$;

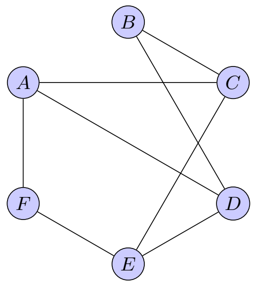
(c) $\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n$;

(d) $O(n^3)$.

A.4 Graph theory

These are the solutions to the exercises on graph theory on page 85.

Solution 4.1.



Solution 4.2. There are multiple correct answers to this question.

(a) $W = (B, D, A, F, E, D, B, C)$;

(b) $P = (E, C, B, D, A, F)$;

(c) $C = (C, E, D, B)$.

Solution 4.3. $\deg(A) = \deg(E) = \deg(D) = \deg(C) = 3$, $\deg(B) = \deg(F) = 2$.

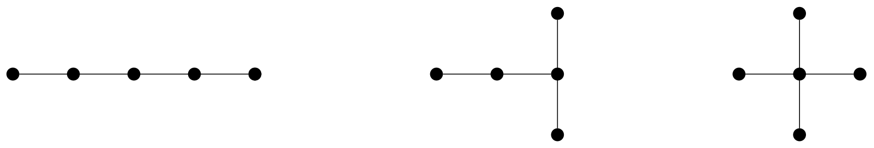
This graph does not have an Eulerian cycle since the number of nodes with an odd degree is not zero or two.

Solution 4.4.

$$A = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}, \qquad A^4 = \begin{bmatrix} 22 & 16 & 0 & 0 & 22 & 0 \\ 16 & 12 & 0 & 0 & 16 & 0 \\ 0 & 0 & 22 & 22 & 0 & 16 \\ 0 & 0 & 22 & 22 & 0 & 16 \\ 22 & 16 & 0 & 0 & 22 & 0 \\ 0 & 0 & 16 & 16 & 0 & 12 \end{bmatrix}.$$

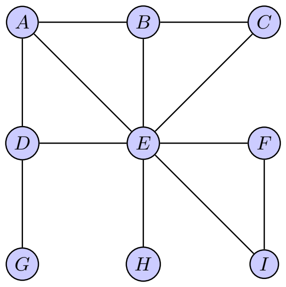
There are 16 walks of length 4 between C and F.

Solution 4.5.



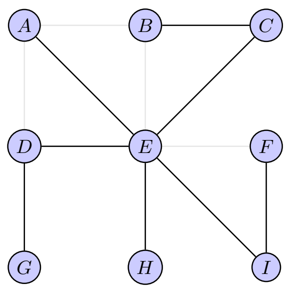
Solution 4.6.

(a)



(b) $\text{closedList} = (A, E, I, F, H, D, G, C, B)$

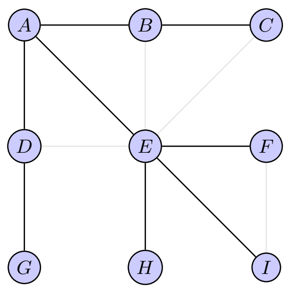
(c)



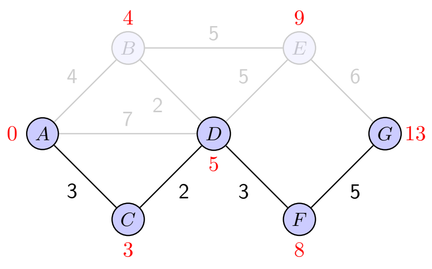
Solution 4.7.

(b) closedList $= (A, B, D, E, C, G, F, H, I)$

(c)

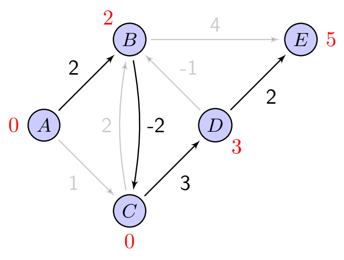


Solution 4.8.



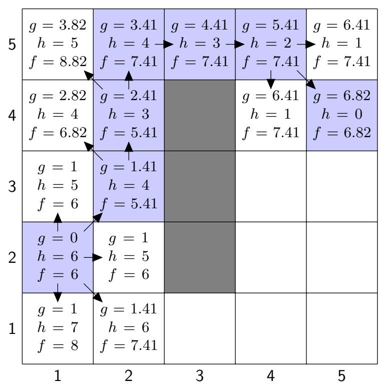
Shortest path: $P = (A, C, D, F, G)$

Solution 4.9.



Shortest path: $P = (A, B, C, D, E)$

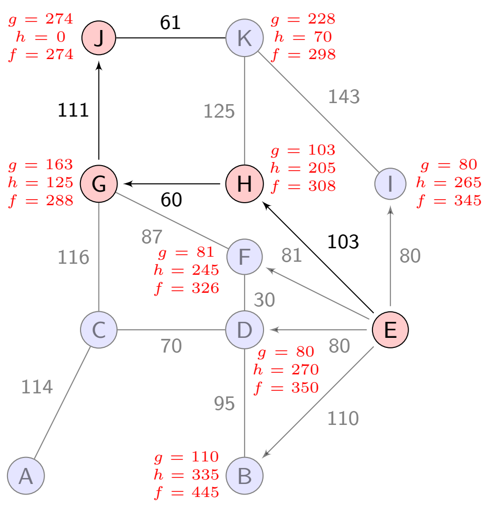
Solution 4.10. Shortest path: $P = ((1, 2), (2, 3), (2, 4), (2, 5), (3, 5), (4, 5), (5, 4))$



$g = 3.82$ $h = 5$ $f = 8.82$
$g = 3.41$ $h = 4$ $f = 7.41$
$g = 4.41$ $h = 3$ $f = 7.41$
$g = 5.41$ $h = 2$ $f = 7.41$
$g = 6.41$ $h = 1$ $f = 7.41$
$g = 2.82$ $h = 4$ $f = 6.82$
$g = 2.41$ $h = 3$ $f = 5.41$
$g = 6.41$ $h = 1$ $f = 7.41$
$g = 6.82$ $h = 0$ $f = 6.82$
$g = 1$ $h = 5$ $f = 6$
$g = 1.41$ $h = 4$ $f = 5.41$
$g = 0$ $h = 6$ $f = 6$
$g = 1$ $h = 5$ $f = 6$
$g = 1$ $h = 7$ $f = 8$
$g = 1.41$ $h = 6$ $f = 7.41$
5
4
3
2
1
1
2
3
4
5

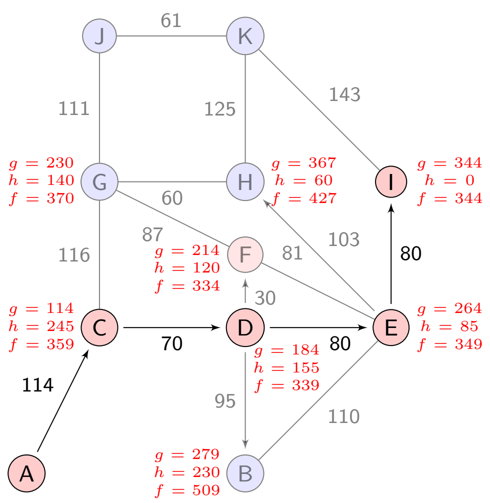
Solution 4.11.

(a) Shortest path: $P = (E, H, G, J)$;



$g = 274$ $h = 0$ $f = 274$
J
61
K
$g = 228$ $h = 70$ $f = 298$
111
125
143
$g = 163$ $h = 125$ $f = 288$
G
60
H
$g = 103$ $h = 205$ $f = 308$
I
$g = 80$ $h = 265$ $f = 345$
87
$g = 81$ $h = 245$ $f = 326$
F
81
103
80
116
30
C
70
D
80
E
$g = 80$ $h = 270$ $f = 350$
114
95
110
A
$g = 110$ $h = 335$ $f = 445$
B

(b) Shortest path: $P = (A, C, D, E, I)$.



J
61
K
111
125
143
$g = 230$ $h = 140$ $f = 370$
G
60
H
$g = 367$ $h = 60$ $f = 427$
I
$g = 344$ $h = 0$ $f = 344$
87
$g = 214$ $h = 120$ $f = 334$
F
81
103
80
116
30
$g = 114$ $h = 245$ $f = 359$
C
70
D
80
E
$g = 264$ $h = 85$ $f = 349$
$g = 184$ $h = 155$ $f = 339$
114
95
110
A
$g = 279$ $h = 230$ $f = 509$
B

Index