



Textos

Licenciatura em Engenharia de Redes e Multimédia

Matemática Discreta e Programação

CARLOS LEANDRO

Lisboa
(2013)

Contents

1	Python	1
1.1	Características de Python:	1
1.1.1	Simples	1
1.1.2	Fácil de Aprender	1
1.1.3	Livre e de Código Aberto	1
1.1.4	Portável	2
1.1.5	Linguagem Interpretada	2
1.1.6	Ferramenta Pedagógica	3
1.1.7	Bibliotecas Extensivas	3
1.1.8	Diferentes sabores	4
1.2	Instalação	4
1.2.1	Para usuários Linux e BSD	4
1.2.2	Para utilizadores Windows	4
1.2.3	Para utilizadores MAC OS X	5
1.3	Fundamentos	5
1.3.1	Constantes Literais	5
1.3.2	Variáveis	7
1.3.3	Linhas Lógicas e Físicas	9
1.3.4	Identação	9
1.4	Operadores e Expressões	10
1.4.1	Introdução	10
1.4.2	Operandos	10
1.5	Controlo do fluxo de programas	11
1.6	Blocos controlados por um <i>if</i>	11
1.7	Ciclos <i>while</i>	13
1.8	Ciclo <i>for</i>	14
1.9	A instrução <i>break</i>	15
2	Lógica proposicional	17
2.1	Proposição	17
2.2	Proposição simples e proposição composta	18
2.3	Conectivas lógicas	18

2.4 Negação	19
2.5 Conjunção	20
2.6 Disjunção	20
2.7 Disjunção exclusiva	20
2.8 Implicação	21
2.9 Bi-implicação	21
2.10 Ordem de precedência das conectivas lógicas	23
2.11 Tautologia	23
2.12 Equivalências proposicionais	24
2.13 Considerações sobre a implicação	27
2.14 Fórmulas bem formadas	28
2.15 Semântica	29
2.16 Argumento	30
2.17 Argumento válido	31
2.18 Demonstração válida	34
2.19 Propriedades do operador	37
2.20 EXERCÍCIOS DE REVISÃO	37
3 Lógica de predicados	43
3.1 Variáveis	43
3.2 Predicado	43
3.3 Conjuntos de verdade	44
3.4 Paradoxo de Russel	44
3.5 Universo de discurso	45
3.6 Predicados impossíveis, possíveis e universais	45
3.7 Conjuntos de verdade e conectivas lógicas	46
3.8 O quantificador universal	46
3.9 O quantificador existencial	47
3.10 O universo vazio	48
3.11 Existe um e um só	49
3.12 Variáveis ligadas	50
3.13 Quantificadores múltiplos	50
3.14 Negação de proposições Quantificadas (Segundas Leis de De Morgan)	51
3.15 Relação de pertença	52
3.16 Descrição	53
3.17 Axioma da extensão	53
3.18 Axioma dos conjuntos elementares	55
3.19 Esquema da separação	55
3.20 Axioma da união	56
3.21 Axioma da potência	58
3.22 Cardinal dum conjunto	58
3.23 Par ordenado	60
3.24 Relação binária	62
3.25 Relação complementar	64

3.26 Reflexividade	65
3.27 Simetria	66
3.28 Transitividade	66
3.29 Fecho	67
3.30 Python: Cláusulas if	68
3.31 Python: Comando for	68
3.32 Python: A função range()	69
3.33 Python: Comando break e continue, e cláusulas else nos ciclos	70
3.34 Python: Mais de funções	71
3.34.1 Argumentos com valores por defeito	73
3.35 Python: Listas	74
3.35.1 Uso da lista como uma pilha	75
3.35.2 Python: Usando listas como filas	76
3.35.3 Python: Listas em Compreensão	76
3.35.4 Python: Matrizes como listas	78
3.36 Python: Tuples	79
3.37 Python: Conjuntos	80
3.38 EXERCÍCIOS DE REVISÃO	84
4 Teoria de Conjuntos	91
4.1 Relação inversa	91
4.2 Composição de relações	92
4.3 Potência dum relação	94
4.4 Partição dum conjunto	95
4.5 Relação de equivalência	95
4.6 Funções	97
4.7 Função sobrejectiva, injectiva e bijectiva	98
4.8 Igualdade	98
4.9 Aplicação composta	98
4.10 Associatividade	98
4.11 Aplicação identidade	99
4.12 Imagem	99
4.13 Imagem recíproca	99
4.14 Correspondência biunívoca	100
4.15 EXERCÍCIOS DE REVISÃO	101
5 Python	105
5.1 Técnicas para ciclos	105
5.2 Python: Mais relativamente a condições	107
5.3 Módulos	107
5.3.1 Mais sobre módulos	109
5.3.2 Executar um módulo como um script	109
5.3.3 O caminho de procura de módulos	110
5.3.4 Ficheiros Python compilados	110

5.3.5	A função dir()	110
5.3.6	Geração de números pseudo-aleatórios	112
6	Teoria de Conjuntos	113
6.1	Relação de Ordem Parcial	113
6.2	Diagrama de Hasse	114
6.3	O conjunto dos números naturais	118
6.4	Indução	118
6.5	Sucessões	122
6.6	Recorrência	124
6.7	EXERCÍCIOS DE REVISÃO	126
7	Estruturas Algébricas	131
7.1	Semi-grupos	131
7.2	Alfabetos e Linguagens	135
7.3	Expressões regulares	137
7.4	Gramáticas	140
7.5	Autómatos	144
7.6	EXERCÍCIOS DE REVISÃO	151
7.7	Python: Dicionários	154
8	PyGame	157
8.1	GUI vs. CLI	157
8.2	O primeiro script em Python usando Pygame	157
8.3	Import - Carregar o módulo	158
8.4	Init - Iniciar o motor de jogo	159
8.5	Quit - Terminar o motor de jogo	159
8.6	Módulos	159
8.6.1	pygame.draw	160
8.6.2	pygame.display	163
8.7	Módulo do pygame para reproduzir e carregar sons	164
8.8	pygame.image	167
8.9	pygame.mouse	170
8.10	Uso do teclado	172
9	Teoria de grafos	177
9.1	Introdução:Leonhard Euler e as sete pontes de Königsburg	177
9.2	Arcos múltiplos	185
9.3	Caminhos de Euler	186
9.4	Árvores de suporte	187
9.5	Árvores de suporte mínimo: Algoritmo de Prim	191
9.6	EXERCÍCIOS DE REVISÃO	194

A Teoria de grafos	201
A.1 O problema do caminho mais curto: Algoritmo de Dijkstra	201
A.2 EXERCÍCIOS DE REVISÃO	212

1

Python

Usualmente um computador é entendido como um dispositivo electrónico constituído por um conjunto de partes físicas, designado de hardware, e controlados através da execução de um conjunto de comandos organizados sob a forma de programas a que usualmente se designa por software. A forma e complexidade com que o hardware está ligado tem evoluído com o desenvolvimento das tecnologias de telecomunicação. O que originalmente eram sistemas definidos por componentes interconectadas por curtas ligações físicas, formadas por fios de cobre, são hoje ligadas por fibras ópticas ou sinais de rádio. O tipo de ligação está dependente das necessidades de desenpanho, mobilidade ou cobertura a que cada sistema tem de assumir. A cobertura e dispêndio do poder computacional por vastas áreas, muitas vezes em diferentes continentes, tem obridado a redefinir o que se entende por computador. Aqui para facilitar a exposição assume-se o computador como uma unidade capaz de executar um programa e cujos dados usados na operação podem ser adquiridos através de uma rede de sistemas de informação. Estes programas são descritos numa linguagem de programação.

Durante as últimos duas décadas tem sido criadas uma grande variedade de linguagens para programar computadores, muitas vezes vocacionadas para um determinado tipo de tarefas, determinado tipo de computador ou determinado metodologia de trabalho.

tipo de tarefa:

Entendendo o computador como uma ferramenta de trabalho, esta tem sido introduzida na maior parte da actividade humana. Desde o controlo do sistema de segurança e climatização das nossas casas, nos mecanismos de comunicação, controlo dos transportes e na produção, distribuição e comercialização do nosso vestuário, comida, e formas de entretenimento. Rápida e progressivamente registando e controlando a actividade humana.

Dada a diversidade de tarefas e a quantidade e heterogeneidade de pessoas envolvidas, um grande esforço tem sido posto no desenvolvimento de metodologias eficientes para a programação dos computadores. Este esforço tem sido dividido. Por um lado é apadrinhada por poderosas multi-nacionais que as usam como base do seu aparelho produtivo e em alguns fonte de parte dos seus lucros. Aqui incluímos empresas engajadas no desenvolvimento de bibliotecas básicas e controlo de hardware.

têm proliferado uma grande diversidade de linguagens de programação. Oriendas para a simplificação do ciclo de desenvolvimento e difusão de aplicações

Tipo de computador

Metodologia de trabalho:

[rever: paradigma] O Python é uma linguagem de programação de alto nível criada por Guido Van Rossum, cuja primeira versão foi publicada em 1991. O Python suporta vários paradigmas de programação, como a programação estruturada, a programação orientada a objectos e também a programação funcional. Nenhum deles é imposto ao programador, sendo possível utilizar o paradigma que se achar mais adequado. Nestes textos usa-se o Python como linguagem de programação estruturada.

1.1 Características de Python:

1.1.1 Simples

Python é uma linguagem simples e minimalista. Ler uma implementação bem feita em Python é quase como ler a descrição dum algoritmo em Inglês. Este carácter de pseudo-código do Python é um de seus maiores pontos fortes.

1.1.2 Fácil de Aprender

É extremamente fácil aprender Python. A linguagem possui uma sintaxe extraordinariamente simples.

1.1.3 Livre e de Código Aberto

Python é um exemplo de FLOSS (Free/Libre and Open Source Software), o que permite a distribuição livremente do interpretador e das suas bibliotecas. Permitindo a alteração do código para outros fins. As licenças FLOSS têm por base comunidades que partilham conhecimento sem fins lucrativos, característica que motiva a sua grande expansão para os mais diferentes fins.

1.1.4 Portável

O código aberto do Python estimulou o seu uso em muitas plataformas. Qualquer programa que se implemente em Python pode, em princípio, ser usado noutras plataformas sem alterações. Bastando que sejam evitadas funções, classes, ou módulos que dependam de características próprias dos sistemas.

Pode-se programar em Python em Linux, Android, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE, PocketPC entre outras.

1.1.5 Linguagem Interpretada

[rever arquitectura linguagem compilador vs interpretador] Os computadores a que temos acesso seguem uma arquitectura de Von Neumann



em que o processador executa um programa armazenado na memória, podendo ler ou escrever dados nessa memória e comunicar com dispositivos de entrada e saída. As instruções que o processador entende, designam-se de código máquina, permitindo realizar operações simples, sendo um programa descrito em memória por uma sequência de códigos numéricos.

Machine code	Instruction description
01	Move data from A to B
02	Move data from A to memory
03	Move data from memory to A
04	Move data from B to A
05	Move data from B to memory
06	Move data from memory to B
07	Add A and B and put result in A
08	Add A and B and put result in B
09	Take the 2's complement of A
0A	Take the 2's complement of B
0B	Clear A
0C	Clear B

program

```

03 SE00 : Move data from address SE0016 into A
06 SE01 : Move data from address SE0116 into A
07      : Add A and B and put result in A
02 SE02 : Move data from A to address SE0216
  
```

it is stored in memory as

```

03 SE 00 06 SE 01 07 02 SE 02
  
```

Para facilitar o seu uso pelos programadores, foi criada uma representação intermédia chamada *assembly*, onde cada uma das instruções passa a ser representada por uma mnemónica. A conversão da linguagem assembly para linguagem máquina, é feita por um programa chamado *assembler*. Apesar de o assembly ser mais fácil de utilizar é necessário compreender os detalhes da máquina, sendo muito exigente a sua utilização na escrita de programas extensos.

Assembly language	Machine code	Instruction description
MOVAB	01	Move data from A to B
MOVAM	02	Move data from A to memory
MOVMA	03	Move data from memory to A
MOVBA	04	Move data from B to A
MOVBM	05	Move data from B to memory
MOVMB	06	Move data from memory to B
ADDAB	07	Add A and B and put result in A
ADDBA	08	Add A and B and put result in B
CMPLA	09	Take the 2's complement of A
CMPLB	0A	Take the 2's complement of B
CLRA	0B	Clear A
CLRB	0C	Clear B

assembly language program

```

MOVHA SE00 : Move data from address SE0016 into A
MOVVA SE01 : Move data from address SE0116 into A
ADDAH SE02 : Add A and B and put result in A
MOVAM SE03 : Move data from A to address SE0216
  
```

Por forma a optimizar a produção de software, têm sido criadas uma grande variedade de linguagens de programação que, por comparação com o assembly, são linguagens de alto nível, como o Fortran, Pascal, C, C++ que permitem a utilização de abstracções

mais próximas, das usualmente usadas pelo homem. Nestes textos descreve-se a linguagem Python, que muitas vezes é tida como de muito alto nível, uma vez que permite a utilização de estruturas de ainda mais alto nível que as anteriores.

Em linguagens como o Fortran, Pascal,C, C++ os programas são traduzidos para linguagem máquina com recurso a um programa designado de compilador. Os compiladores permitem transformar uma implementação (código-fonte) em código executável directamente pela máquina.

No entanto o Python, por sua vez, não recorre a um compilador. Os programas podem ser executados sem que tenham de ser traduzidos integralmente para código máquina. Um programa pode ser executado directamente do código-fonte. Em vez dum compilador, o Python recorre a um interpretador. Que permite interpretar o código-fonte num formato intermediário chamado *bytecode*, o qual é traduzido para a linguagem nativa do computador. Esta estratégia aliada à simplicidade da linguagem acelera o processo de desenvolvimento. Reduzindo o tempo entre a detecção de erros e a sua correcção. Melhorando também a portabilidade dos programas escritos em Python, uma vez que permite normalizar os módulos tornando-os independentes das arquitecturas dos sistemas que correm o interpretador.

1.1.6 Ferramenta Pedagógica

[paradigma de programação] O Python suporta diferentes paradigmas de programação, permitindo adoptar uma programação de estilo tanto estruturada como orientada para objectos. Na perspectiva procedural, o programa é construído com base em procedimentos e funções, que nada mais são que partes de código reutilizáveis. Numa linguagem orientada a objectos, o código é definido com base em objectos que combinam dados e métodos. No Python podemos encontrar um meio simples mas poderoso de implementar programação orientada a objectos, especialmente quando comparada a linguagens como C++ ou Java. Para além das perspectivas de programação anteriores, o Python pode ainda adoptar-se uma programação funcional, pondo-se neste caso foco na resolução de problemas usando composição de funções. Note-se que, neste texto usa-se o Python apenas na perspectiva da programação estruturada.

1.1.7 Bibliotecas Extensivas

[Módulos] Os Módulos Padrão do Python (Python Standard Library) são muito completos. Podemos encontrar métodos e estruturas para tratar expressões regulares, gerar documentos, threading, acesso a bancos de dados, CGI, FTP, e-mail, XML, XML-RPC, HTML, arquivos WAV, criptografia, GUI (Interfaces Gráficas com o utilizador), Tk e chamadas ao sistema. Esta biblioteca é comum a qualquer instalação do Python e independente do sistema.

[diferentes Módulos por área de aplicação] Além dos módulos padrão, existem várias outros módulos que se centram em tarefas especializadas em tecnologias da informática, ou áreas da engenharia ou ciência, tais como wxPython , Twisted, Python Imaging Library e muitas outras. De forma simples os módulos standard podem ser enriquecidos, adaptando o interpretador às características do algoritmo a implementar,

com o objectivo último de simplificar a programação e os tempos de desenvolvimento.

1.1.8 Diferentes sabores

[caracterizar as diferenças entre as duas variantes] A linguagem Python é hoje (2013) oferecida em dois sabores, o Python 2.x e o Python 3.x. Estes dois sabores diferem na natureza de alguns dos métodos e funções standard. Aqui é adoptado o sabor Python 3.x (com x maior que 2).

1.2 Instalação

As distribuições das várias versões do sistema operativo Linux já incluem o interpretador de linguagem Python. Para o sistema operativo Windows pode instalar-se o interpretador de <http://www.python.org/>.

1.2.1 Para usuários Linux e BSD

Para uma distribuição Linux como Ubuntu, Fedora, OpenSUSE, ou um sistema BSD tal como FreeBSD, o Python já está instalado.

Para testar o Python, inicie uma shell (como konsole ou gnome-terminal) e execute o comando `python -V` como abaixo:

```
$ python -V  
Python 3.3
```

Caso obtenha um mensagem do tipo abaixo:

```
$ python -V  
bash: Python: command not found
```

É porque o Python não está instalado. Isto é altamente improvável, mas é possível. Neste caso, pode instalar os pacotes binários, com a versão 3.3, usando um gestor de pacotes fornecido pelo seu sistema operacional, tal como apt-get no Ubuntu/Debian e outros Linux baseados no Debian, yum no Fedora, pkg_add no FreeBSD, etc. Note que para isso necessita de uma conexão à internet.

Uma nota adicional: Pode ter diferentes versões do Python instalado no seu sistema.

1.2.2 Para utilizadores Windows

Visite o site <http://www.python.org/download/releases> e para acompanhar este texto, faça o download da última versão 3.x (com x>2).

Se pretende usar o Python da linha de comando do Windows, tem de configurar a variável PATH. No Windows XP, Vista, 7 ou 8, clique em Painel de Controle -> Sistema -> Avançado -> Variáveis de ambiente. Clique na variável chamada PATH na secção “Variáveis de Sistema”, e seleccione Editar e adicione “;C:\Python3.x” no fim do que já lá estiver. Naturalmente que deve usar aqui o directório que escolheu para instalar o Python.

1.2.3 Para utilizadores MAC OS X

O Mac OS X já tem o Python instalado. Execute *Terminal.app* e `python -V`, para avaliar a versão instalada. Sempre que a versão não é a adequada use o gestor de pacotes para instalar uma alternativa.

1.3 Fundamentos

1.3.1 Constantes Literais

Um exemplo de uma constante literal é um número como 5, 1.23, $9.25e - 3$ ou uma string (sequência de caracteres) como '*Isto é uma string*' ou '*É uma string!*'. Estas constantes são designadas literais porque devem ser interpretadas à letra (literalmente). O número 2 é entendido como dois, é uma constante porque o seu valor (significado) não pode ser alterado.

Números

[sistema de numeração]

Os números em Python são de três tipos: **inteiros**, **ponto flutuante** e **complexos**:

- 2 é um exemplo de inteiro, os inteiros são elementos do conjunto dos números inteiros em matemática \mathbb{Z} . Existem no entanto limitações à magnitude dos inteiros que se podem usar.
- 3.23 e $52.3E - 4$ são exemplos de números no sistema de vírgula flutuante (ou floats, para abreviar). A notação E indica as potências de 10. Neste caso, $52.3E - 4$ significa $52.3 * 10^{-4}$.
- $(-5 + 4j)$ e $(2.3 - 4.6j)$ são exemplos de números complexos, que em Matemática é normal representar por $-5 + 4i$ e $2.3 - 4.6i$.

Strings

[caracteres]

Uma string é uma sequência de caracteres. As strings são basicamente uma sequência de símbolos. As strings são usualmente usadas para representar palavras que podem ser da língua Inglesa ou de qualquer língua que seja suportada pelo padrão Unicode (permitindo codificar quase todas as línguas do mundo).

Aspas Unitárias

Uma string pode ser definida por uma sequência de caracteres delimitada por aspas unitárias (ou apóstrofes) tais como '*O estudo da lógica remonta à civilização helénica*'. Todos os espaços em branco, isto é, espaços e tabulações são preservados no estado em que se encontram.

Aspas Duplas

As strings podem ser também definidas usando aspas duplas por exemplo : "A arte da argumentação levou à morte de Sócrates."

Aspas Tripas

Outra forma a definir strings que ocupam várias linhas é usar aspas triplas (""" ou '''). Um exemplo:

```
'''A palavra "trivial" tem uma etimologia interessante.  
É a conjugação de "tri" (significando '3') e "via" (significando caminho).  
Originalmente refere-se ao "trivium", as três áreas fundamentais do  
'curriculae': gramática, retórica e lógica.  
Assuntos que se tem de dominar para aceder ao "quadrivium", que  
consiste na aritmética, geometria, música e astronomia.  
'''
```

Sequências de Escape

Para definir uma string que contenha um apóstrofe ('), como em '*Why was logic considered to be fundamental to one's education?*', sem que a apóstrofe interna entre em conflito com os delimitadores, usa-se uma sequência de escape. Para evitar o conflito, o caractere apóstrofe é representado na string por \'. A string deve assim ser definida como '*Why was logic considered to be fundamental to one\'s education?*'.

Outra forma de definir a string anterior seria "*Why was logic considered to be fundamental to one\'s education?*", através das aspas duplas. De forma idêntica, é usada uma sequência de escape para inserir aspas duplas numa string limitada por aspas duplas. A própria barra invertida pode ser inserida na string pela sequência de escape \\.

Como apresentado para definir uma string com mais de duas linhas usa-se por limitador aspas triplas. Outro processo é usar uma sequência de escape \n, para indicar o fim de uma linha e o início de outra linha na string. Por exemplo,

'A lógica centra-se na razão e na noção de verdade.\n A retórica fundamenta-se em ideias feitas e populistas.'

Existem outras sequências de escape, apresenta-se aqui apenas as mais usadas. Para um descrição sistemática use a documentação do interpretador.

Num editor, durante a programação, é frequente ter a necessidade de continuar uma string na linha imediatamente abaixo. Para isso é usada uma única barra invertida no fim da linha. Por exemplo em *Looking Glass* de Lewis Carroll:

```
>>> H=  "\"Reciprocamente\", continuou Tweedledee, \  
\"Se é assim, ele pode ser, \n \  
e se não é, será; mas como não é não se preocupa. \  
Isto é lógica.\""  
>>> print(H)  
"Reciprocamente", continuou Tweedledee, "Se é assim, ele pode ser,  
e se não é, será; mas como não é não se preocupa. Isto é lógica."
```

Se por algum motivo tem necessidade que o interpretador não trate as sequências de escape, na definição da string deve usar como prefixo um r ou um R. Por exemplo, na string anterior:

```
>>> H= r'"Reciprocamente\\"", continuou Tweedledee, \
\"Se é assim, ele pode ser, \n \
e se não é, será; mas como não é não se preocupa. \
Isto é lógica.\"
>>> print(H)
\"Reciprocamente\\"", continuou Tweedledee, \
\"Se é assim, ele pode ser, \n \
e se não é, será; mas como não é não se preocupa. \
Isto é lógica.\"
```

[unicode: descrição e exemplos]

As strings são Imutáveis

[seria mais lógico iniciar o estudo com listas, tuplos diferença entre estruturas mutáveis e imutáveis] Isso significa que, como constantes literais, uma vez definida uma string, esta não pode sofrer alterações.

Concatenação de Literais do Tipo String

Quando numa linha de código duas strings são postas lado a lado, o interpretador faz a sua concatenação. Por exemplo

```
>>> H='Originalmente' ' a lógica lidava' ' com linguagem natural'
>>> H
'Originalmente a lógica lidava com linguagem natural'
```

De forma mais descriptiva pode-se recorrer ao operador +:

```
>>> H='Seria útil'+ ' demonstrar a correcção'+ ' dum argumento.'
>>> H
'Seria útil demonstrar a correcção dum argumento.'
```

1.3.2 Variáveis

As variáveis podem ser usadas para referenciar literais, por forma a permitir o seu tratamento e manipulação. De forma genérica, uma variável pode ser entendida como uma referência a uma parte da memória do computador onde está armazenada informação. Diferindo das constantes literais, uma vez que o seu significado pode variar no decurso do programa.

```
>>> H1 = 'Seria útil '
>>> H2 = 'demonstrar a correcção '
>>> H3 = ' dum argumento.'
```

```
>>> print(H1+H2+H3)
Seria útil demonstrar a correcção dum argumento.
```

No exemplo H1, H2 e H3 são usadas para identificar três strings, cuja concatenação é imprimida na *shell* através do comando *print*.

As variáveis são entendidas como identificadores. Entendendo-se por identificador um nome dado para identificar um objecto. Existem regras para descrever os identificadores:

1. O primeiro carácter do identificador tem de ser uma letra do alfabeto (maiúsculo ASCII ou minúsculo ASCII) ou um *underscore* ('_').
2. O resto do nome do identificador pode consistir de letras (maiúsculo ASCII ou minúsculo ASCII), *underscores* ('_') ou dígitos (0-9).
3. Nomes de identificadores são *case-sensitive*. Por exemplo, myname e myName são identificadores diferentes.

Exemplos de nomes de identificadores válidos são i, __my_name, name_23 e a1b2_c3.

Exemplos de nomes de identificadores inválidos são 2things e my-name.

Um exemplo:

```
# Filename: var.py

i = 5
print(i)

i = i + 1
print(i)

s = '''Esta é uma string de múltiplas linhas.
Esta é a segunda linha.'''
print(s)
```

Output:

```
5
6
Esta é uma string de múltiplas linhas.
```

No programa começamos por atribuir o valor constante literal 5 à variável *i* através do **operador de atribuição** (=). Uma linha deste tipo é designada de instrução, e indica neste caso, que passámos a referenciar através do nome da variável *i* o objecto 5. Em seguida, imprime-se o valor de *i* através do comando **print**, que imprime o valor da variável na *shell*.

Na instrução seguinte somamos 1 ao valor referenciado por *i*. A partir deste momento *i* passa a referenciar o objecto 6. Em seguida, imprime-se o valor de *i*, agora 6.

Como já se tinha feito na secção anterior, de forma análoga referencia-se um objecto string pela variável *s*, que depois se imprime.

1.3.3 Linhas Lógicas e Físicas

As linhas físicas são aquelas que escrevemos num editor a quando da definição dum programa. Uma linha lógica é a que o interpretador de Python entende por uma única instrução. O Python implicitamente assume que cada linha física corresponde a uma linha lógica.

Um exemplo de uma linha lógica é uma instrução como `print('Sócrates é mortal')` caso esteja escrita no editor numa única linha, deve também ser entendida como uma linha física.

Implicitamente, Python incentiva o uso de uma única instrução por linha, com o propósito de tornar o código mais legível.

Se pretende definir mais do que uma linha lógica numa única linha física, então deve separar as linhas lógicas através de um ponto-e-virgula (';') para indicar o fim de cada linha lógica ou instrução. Por exemplo,

```
i = 5
print(i)
```

é o mesmo que

```
i = 5; print(i)
```

Voltando a um exemplo anterior:

```
>>> H=  "\"Reciprocamente\", continuou Tweedledee, \
\"Se é assim, ele pode ser, \n \
e se não é, será; mas como não é não se preocupa. \
Isto é lógica.\"
>>> print(H)
"Reciprocamente", continuou Tweedledee, "Se é assim, ele pode ser,
e se não é, será; mas como não é não se preocupa. Isto é lógica."
```

O objecto string que passa a ser referenciado por H deve ser entendido como definido numa única linha lógica, apesar de ocupar diferentes linhas físicas.

Neste sentido usa-se ; (ponto-e-virgula) para separar linhas lógicas na mesma linha física, enquanto \ para separar uma linha lógica em diferentes linhas físicas.

1.3.4 Identação

Os espaços em branco são muito importantes no Python. Na verdade, os espaços brancos no início de uma linha definem a estrutura do programa. Nesta situação são designados de identação. Os espaços ou tabulações no início de uma linha determinam o nível de identação de uma linha lógica, que por sua vez agrupam instruções. Significando isto que linhas com o mesmo nível de identação são executadas em sequência. Estes conjuntos de instruções são designados blocos. Tenta-se ao longo do texto descrever a importância dos blocos.

Note que, má identação pode originar erros a quando da execução. Por exemplo:

```
i = 5
print('São ', i) # Erro! Existe um espaço no início da linha
print('São,',i,' os macacos.')
```

Quando da execução obtém-se um erro do tipo:

```
File "whitespace.py", line 4
    print('São ', i) # Erro! Existe um espaço no início da linha
    ^
IndentationError: unexpected indent
```

Note-se que, existe um espaço simples no início da segunda linha. O erro indica que a sintaxe do programa está incorrecta, ou seja, o programa não está escrito com a estrutura correcta. Não podemos iniciar novos blocos de instruções arbitrariamente (com excepção do inicial). Os casos onde se podem usar novos blocos são apresentados ao longo do capítulo.

1.3.4.1 Como identar

Não misturar tabulações com espaços a quando da identação, já que nem todas as plataformas a suportam. É recomendado o uso de uma tabulação, ou dois espaços ou quatro espaços para distinguir cada nível de identação. Escolha um destes estilos de identação. Mais importante que a escolha que faz, deve manter-se consistente a ela, ou seja, mantenha apenas um tipo de identação ao longo de todo o código.

1.4 Operadores e Expressões

1.4.1 Introdução

A maioria das instruções (linhas lógicas) que escreve contêm expressões. Um exemplo simples de uma expressão é $2+3$. Uma expressão pode ser partida em operador e operandos. Os operadores são funções que podem ser identificadas por símbolos, como $+$, ou palavras especiais. Os operadores requerem argumentos, que chamamos de operandos. No exemplo anterior, 2 e 3 são operandos.

1.4.2 Operandos

Uma expressão pode ser avaliada no operador interactivamente.

Por exemplo, para testar a expressão $2+3$, usamos a *prompt* do interpretador de Python:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Operador	Nome	Explicação
+	Adição	Soma dois objectos
-	Subtracção	Define um número negativo ou a subtracção de um número por outro
*	Multiplicação	Devolve o produto de dois números ou uma string repetida uma certa quantidade de vezes.
**	Potência	Retorna x elevado à potência de y
/	Divisão	Divide x por y
//	Divisão Inteira	Devolve a parte inteira do quociente
%	Modulo	Devolve o resto da divisão inteira
<	Menor que	Compara x a y. Devolvendo True(verdadeiro) se x é menor que y, e False(falso) caso contrário
>	Maior que	Devolvendo True se x é maior que y, e False caso contrário.
<=	Menor ou igual a	Devolvendo True se x é menor ou igual a y, e False caso contrário.
>=	Maior ou igual a	Devolvendo True se x é maior ou igual a y, e False caso contrário.
==	Igual a	Avalia se os objectos são iguais
!=	Diferente de	Avalia se os objectos são diferentes
not	Operador booleano NOT	Se x é True, devolve False. Se x é False, ele devolve True.
and	Operador booleano AND	x and y devolve False se x é False, senão devolve a avaliação de y.
or	Operador booleano OR	Se x é True, devolve True, senão devolve a avaliação de y.

1.5 Controlo do fluxo de programas

Os programas que vimos até aqui, são descritos por uma série de declarações e o interpretador de Python executa-as seguindo uma ordem estipulada. Como alterar o fluxo execução? Por exemplo, se pretende que o programa tome algumas decisões e faça diferentes coisas dependendo das diferentes situações, como imprimir 'Bom Dia' ou 'Boa Tarde' dependendo da hora do dia.

Isto é consumado usando as instruções de controle de fluxo no Python *if*, *for* e *while*, permitindo executar um ou mais blocos de instruções apenas ou enquanto uma condição for verdadeira.

1.6 Blocos controlados por um *if*

A instrução *if* é usada para avaliar uma condição e se a condição é verdadeira, é executado um bloco de instruções (a que chamamos de *bloco-if* (**if-block**)), senão é executado outro bloco de instruções (a que chamamos de *bloco-else* (**else-block**)). A cláusula *else* é opcional.

```
#!/usr/bin/python
# Nome do ficheiro: if.py

number = 23
guess = int(input('Qual é o número inteiro? '))

if guess == number:
    print('Parabéns, você acertou.') # Novo bloco começa aqui
```

```

    print('Tenha um bom dia...') # Novo bloco termina aqui
elif guess < number:
    print('Não, é maior que isso.') # Outro bloco
    # O bloco pode conter uma ou mais linhas ...
else:
    print('Não, é menor que isso.')

print('Adeus.')
# Esta última instrução é sempre executada, depois da instrução if
# ser executada

```

Para diferentes execuções da script acima obtém-se:

```

Qual é o número inteiro? 50
Não, é menor que isso.
Adeus.

```

Outra tentativa:

```

Qual é o número inteiro? 22
Não, é maior que isso.
Adeus.

```

Mais uma tentativa:

```

Qual é o número inteiro? 23
Parabéns, você acertou.
Tenha um bom dia...
Adeus.

```

Neste programa, é inquirido o utilizador por um número inteiro e é verificado se este é igual a um número escondido. Usa-se uma variável *number* para referenciar o inteiro a adivinhar, neste caso *number* = 23. O utilizador tem apenas uma tentativa para adivinhar o número. A hipótese do utilizador é feita através da função *input()*. Uma função é entendida aqui como um módulo ou bloco de código reutilizável.

A função *input()*, cria uma *prompt* com o argumento da função na shell, e espera que o utilizador escreva uma cadeia de caracteres (*string*). A *string* que o utilizador fornece é usada como valor de saída da função. A função *int()*, converte essa *string* para um inteiro, passado a ser referenciado pela variável *guess*. Genericamente a função *int()* permite, sempre que possível, converter objectos para *string*. Neste caso é usada para converter uma cadeia de caracteres para um inteiro.

Em seguida, é comparado o inteiro referenciado pela variável *guess* com o número referenciado por *number*. Se eles forem iguais, imprime-se uma mensagem a felicitar o utilizador. Note que são utilizados níveis de identação para informar o interpretador de Python que a sequência de instruções pertence a um bloco.

Note que a linha da instrução *if* termina com “dois pontos” indicando que a seguir há um bloco de instruções.

Caso a tentativa do utilizador seja menor que o número referenciado pela variável *number*, informa-se que tente na próxima execução um número maior que o número da presente tentativa. Usa-se aqui uma cláusula *elif* para reduz a quantidade de identações requeridas.

Note-se que as linhas das instruções *elif* e *else* também terminam com “dois pontos”, sendo seguidas pelo seu bloco de instruções que controlam.

Devemos realçar que as partes *elif* e *else* são opcionais. Uma instrução *if* mínima válida assume a forma:

```
if True:  
    print 'Sim, é verdade'
```

Após ser executada a instrução *if* por completo, e as cláusulas *elif* e *else* que lhe estão associadas, a execução passa para o próximo bloco de instruções. Neste caso, volta ao bloco principal onde encontra a instrução *print 'Adeus.'*. Após executar esta linha de código, o interpretador termina a execução do código.

Na verdade, não é muito prático ter de executar o programa sempre que se quer fazer uma nova tentativa. Tentemos resolver o problema.

1.7 Ciclos *while*

A instrução *while* permite executar repetidamente um bloco de instruções, enquanto uma condição for verdadeira. Uma instrução *while* pode ter uma cláusula *else* opcional.

Exemplo:

```
#!/usr/bin/python  
# Nome do ficheiro: while.py  
  
number = 23  
running = True  
  
while running:  
    guess = int(input('Qual é o número inteiro?'))  
  
    if guess == number:  
        print('Parabéns, você acertou.')  
        running = False # Isto faz o loop while parar  
    elif guess < number:  
        print('Não, é maior que isso.')  
    else:  
        print('Não, é menor que isso.')  
else:  
    print('O loop while terminou.')  
  
print('Adeus.')
```

A execução destas linhas de código têm, para as mesmas tentativas do exemplo anterior, por *output*:

```
Qual é o número inteiro? 50
Não, é menor que isso.
Qual é o número inteiro? 22
Não, é maior que isso.
Qual é o número inteiro? 23
Parabéns, você acertou.
Tenha um bom dia...
Adeus.
```

Neste programa, melhorámos o jogo anterior, tendo a vantagem de se poder fazer várias tentativas na mesma execução, terminando apenas quando se acerta no número. Tenta-se assim exemplificar o uso do ciclo *while*.

Neste código a função *input* e o bloco *if*, do programa anterior formam um bloco controlado pelo *while*. É adicionada, antes do bloco *while*, uma nova variável *running*, que é iniciada como referenciando o valor de verdade *True*. Como a condição que controla o *while* é verdadeira, o bloco controlado pelo *while* é executado. Após a execução deste bloco de instruções, a condição é novamente avaliada. Se *running* continua a referenciar verdade, o bloco *while* volta a ser executado, caso contrário, se *running* passa a referenciar falso, a execução passa para o bloco opcional *else*, findo o qual passa a executar instruções no bloco principal.

Neste sentido, o bloco *else* é executado quando a condição que controla o *while* se torna *False* que pode acontecer na primeira avaliação da condição. Se existir uma cláusula *else* no ciclo *while*, este é sempre executado a menos que o ciclo *while* nunca termine.

Os valores *True* (verdadeiro) e *False* (falso) são designados de objectos de tipo Booleano ou valores de verdade.

Note-se que, um bloco *else* num ciclo *while* é redundante, já que se este bloco for usado no bloco anterior ao do *while* o programa tem o mesmo comportamento.

1.8 Ciclo *for*

A instrução *for <var> in <objecto>* permite impor a execução cíclica de um conjunto de instruções, por exemplo, executa o bloco para cada item numa sequência.

Exemplo:

```
#!/usr/bin/python
# Nome do ficheiro: for.py

for i in range(1, 5):
    print i
else:
    print('O ciclo terminou.')
```

Escreve na shell:

```
1
2
3
4
O ciclo terminou.
```

Neste programa, imprime-se uma sequência de números. A sequência de objectos a imprimir é gerada através da função interna *range*.

A função *range*, possibilita a utilização de dois argumentos numéricos, devolvendo uma sequência de números tendo início no primeiro argumento, que incrementa sucessivamente uma unidade, terminando antes de alcançar o segundo. No exemplo, usa-se *range(1,5)* originando a sequência [1, 2, 3, 4]. O incremento desta função entre cada objecto na lista pode ser controlado usando um terceiro argumento. Por exemplo, *range(1,5,2)* devolve uma referência para o objecto [1,3]. Note-se que na função *range* o segundo argumento funciona como limite da sequência, nunca sendo alcançado.

No ciclo *for i in range(1,5)* é equivalente a *for i in [1, 2, 3, 4]* em cada execução do bloco de instruções que controla *i*, assume valores diferentes, um por cada valor na lista [1, 2, 3, 4]. Na primeira execução *i* referencia o inteiro 1, na segunda execução *i* referencia o inteiro 2, na terceira execução *i* referencia o inteiro 3 e na quarta execução *i* referencia o inteiro 4. Em cada uma destas execuções imprimimos o valor de *i*, sendo este bloco executado 4 e apenas quatro vezes, tantas vezes quanto o número de objectos que definem a lista. Fimdo o qual imprime “O ciclo terminou”.

Lembre-se que a parte *else* é opcional. Quando incluída, ela será sempre executada uma vez após o ciclo *for* ter terminado, a não ser que uma instrução *break* seja encontrada.

Devemos notar que o ciclo “*for <var> in <objecto>*”, pode iterar em qualquer objecto iterável. Aqui temos uma lista de números, em geral, podemos usar por exemplo objectos do tipo tuplo, conjunto, string ou lista genérica.

1.9 A instrução *break*

A instrução *break* é usada para impedir a continuação da execução dum ciclo. Permitindo por exemplo terminar um ciclo *while* sem que a condição que a controla se torne falsa.

Deve no entanto notar que quando usada em ciclos *for* ou *while*, os correspondentes bloco *else* não são executados.

Exemplo:

```
#!/usr/bin/python
# Nome do ficheiro: break.py

while True:
    s = input('Diga alguma coisa: ')
```

```
if s == 'vai dar uma volta':  
    break  
print('Hoje não está muito comunicativo.')  
    print('Escreveu', len(s), 'caracteres.')  
print('Então adeus ;-(')
```

Neste programa, a cada entrada de valores o programa imprime “Hoje não está muito comunicativo.”, indicando de seguida o número de caracteres usados. Sendo o comportamento alterado sempre que escrever “vai dar uma volta”. Neste caso o ciclo termina executando *break*, voltando ao bloco principal, imprime “Então adeus ;-(" e termina o programa.

O tamanho duma string é determinada através da função interna *len*.

Lembre-se que a instrução *break* pode também ser usada com os ciclos *for*. No entanto, apesar desta instrução facilitar a implementação de certos algoritmos, vamos tentar aqui evitar a sua utilização.

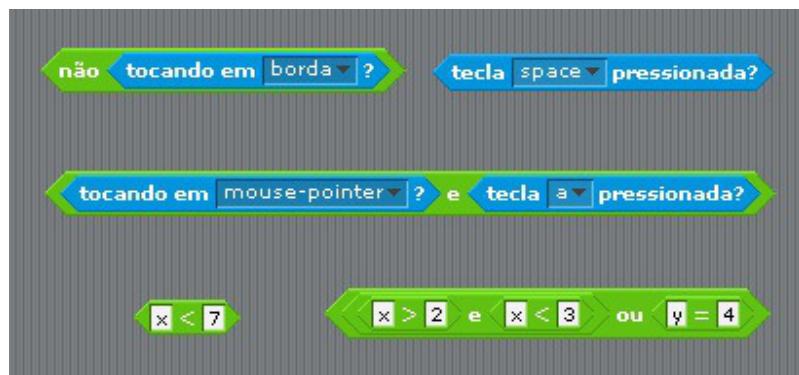
2

Lógica proposicional

“Poder-se-á definir a Lógica como a ciéncia das regras que legitimam a utilização da palavra portanto.” B. Ruyer in Logique.

2.1 Proposição

No caso das instruções **if** e **while**, a execução dum bloco de código está dependente da avaliação duma função proposicional (condição). Com o objectivo de estudar estas instruções e formalizar a noção de função proposicional começa-se por rever algumas noções de lógica proposicional e do cálculo de predicados.



Os elementos básicos da lógica são as **proposições** ou **sentenças** que se entendem como afirmações precisas. Na lógica clássica, que abordamos, a avaliação duma proposição é regida por dois princípios fundamentais:

- **Princípio da não contradição** - Uma proposição não pode ser simultaneamente verdadeira e falsa;
- **Princípio do terceiro excluído** - Uma proposição ou é verdadeira ou é falsa;

Por exemplo “1 é maior que 3” é uma proposição cujo valor lógico é o de “falsidade” enquanto que “todos os triângulos têm três lados e três ângulos” é uma proposição cujo valor lógico é o de “verdade”.

Por outro lado “ $x < 3$ ” não é uma proposição (depende do valor que venha a ser atribuído à variável x) sendo denominada **função proposicional**.

Representam-se por letras (geralmente minúsculas) as proposições genéricas (ou variáveis proposicionais) e por 1 (ou V) e 0 (ou F) os valores lógicos de “verdade” e “falsidade”, respectivamente.

A área da lógica que trata as proposições neste contexto é designada por **cálculo proposicional** ou lógica proposicional.

2.2 Proposição simples e proposição composta

Por vezes combinam-se várias proposições para obter proposições mais expressivas. Neste sentido, classificamos as proposições como **simples** (também denominada atómica) ou **composta** (também denominada molecular).

- As proposições simples apresentam apenas uma afirmação:
 1. $p : \sqrt{2}$ não é um número racional,
 2. $q :$ existem mais números reais que inteiros,
 3. $v : 1 = 2,$
 4. $r : 2 + 3 > 4$
- As proposições compostas são definidas por uma ou por mais do que uma proposição, usando na sua formação **operadores lógicos** (também designados de **conectivas lógicas** ou operadores para formação de proposições):
 1. $x = 2 \text{ e } y = 1;$
 2. **se** $x > y$ **então** $y < x;$
 3. não é verdade que $2 + 3 > 4.$

2.3 Conectivas lógicas

Em cálculo proposicional as proposições são geradas a partir de proposições simples, usando operadores para formação de proposições. Vamos tomar como sintaticamente válidas proposições compostas da forma:

- **não** $p,$
- $p \text{ e } q,$
- $p \text{ ou } q,$
- **ou** $p \text{ ou } (exclusivo) q,$
- **se** $p \text{ então } q,$

- p se e só se q

onde p e q são proposições (simples ou compostas). Neste casos, em geral, pretendemos obter os valores lógicos das proposições compostas em função dos valores lógicos conhecidos das proposições mais simples que as compõem. Por forma a podermos formalizar a lógica e a avaliação de proposições, convencionamos a seguinte representação para os operadores sintácticos usados na formação de proposições:

Operações Lógicas	Símbolos	Notação	Significado
Negação	\neg ou \sim	$\neg p$	não p
Conjunção	\wedge	$p \wedge q$	p e q
Disjunção	\vee	$p \vee q$	p ou q
Disjunção exclusiva	\oplus ou $\dot{\vee}$	$p \oplus q$	ou p ou (exclusivo) q
Implicação	\rightarrow	$p \rightarrow q$	se p então q
Bi-implicação	\leftrightarrow	$p \leftrightarrow q$	p se e só se q

2.4 Negação

Seja p uma proposição. A afirmação “não se verifica que p ” é uma nova proposição, designada de **negação** de p . A negação de p é denotada por $\neg p$ ou $\sim p$. A proposição $\neg p$ deve ler-se “não p ” e é verdadeira se p é falsa. A proposição $\neg p$ é falsa se p é verdadeira.

É usual definir a interpretação dum operador lógico através de tabelas do tipo:

p	$\neg p$	p	$\neg p$
T	F	ou	1
F	T		0

Estas tabelas são designadas por **tabelas de verdade**. Neste caso define completamente o operador negação, relacionando os valores lógicos de p e $\neg p$.

Note que, em linguagem corrente nem sempre se pode negar logicamente uma proposição, antepondo o advérbio “não” ao verbo da proposição, isto apenas se verifica nos casos mais simples.

Por exemplo: negar “Hoje é sábado.” é afirmar “Hoje não é sábado”.

Mas negar que “Todas as aves voam” é o mesmo que afirmar “não se verifica que todas as aves voam” o que é equivalente a afirmar que “Nem todas as aves voam” mas não é afirmar que “Todas as aves não voam”.

Em linguagem Matemática, dado o rigor da interpretação das designações usadas, o processo de negação fica simplificado. Por exemplo, negar “ $5 > 2$ ” é o mesmo que afirmar “ $\neg(5 > 2)$ ” que é equivalente, por definição da relação $>$, a escrever “ $5 \leq 2$ ”. Assim como “ $5 > 2$ ” é verdade, temos pela interpretação da negação que “ $\neg(5 > 2)$ ” é falso.

2.5 Conjunção

Sejam p e q proposições. A proposição “ p e q ”, denotada $p \wedge q$, é a proposição que é verdadeira apenas quando p e q são ambas verdadeiras, caso contrário é falsa. A proposição $p \wedge q$ diz-se a **conjunção** de p e q .

Assim, os valores lógicos das três proposições p , q , e $p \wedge q$ estão relacionados pela tabela de verdade:

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

Note que a tabela tem quatro linhas, uma por cada combinação possível de valores de verdade para as proposições p e q .

2.6 Disjunção

Sejam p e q proposições. A proposição “ p ou q ”, denotada $p \vee q$, é a proposição que é falsa apenas quando p e q são ambas falsas, caso contrário é verdade. A proposição $p \vee q$ diz-se a **disjunção** de p e q . A tabela de verdade de $p \vee q$ toma assim a forma:

p	q	$p \vee q$
V	V	V
V	F	V
F	V	V
F	F	F

A conectiva *ou* é interpretada na versão inclusiva da palavra *ou* em linguagem corrente. Note que, nas proposições seguintes *ou* tem ou **significado inclusivo** ou **significado exclusivo** consoante o contexto de interpretação:

1. O João pratica futebol ou natação.[ou ambas as coisas]
2. Ele é do Sporting ou do Porto.[mas não as duas coisas]

2.7 Disjunção exclusiva

Para tornar a interpretação da disjunção independente do contexto definimos: A **disjunção exclusiva** de p e q , denotada $p \oplus q$ ou $p \dot{\vee} q$, é a proposição que é verdade apenas quando, ou p é verdadeira ou q é verdadeira, caso contrário é falsa.

A tabela de verdade de $p \oplus q$ toma assim a forma:

p	q	$p \oplus q$
V	V	F
V	F	V
F	V	V
F	F	F

Exercício 2.7.1. Relacione o valor lógico das proposições p , q , r e $(p \wedge (\neg q)) \oplus (r \vee p)$.

Exercício 2.7.2. Indique os valores (de verdade ou falsidade) das seguintes afirmações:

1. $3 \leq 7$ e 4 é um número ímpar
2. $3 \leq 7$ ou 4 é um número ímpar
3. 5 é ímpar ou divisível por 4

2.8 Implicação

Sejam p e q proposições. A implicação $p \rightarrow q$ é a proposição que é falsa quando p é verdadeira e q é falsa, nos outros casos é verdadeira.

A tabela de verdade de $p \rightarrow q$ toma assim a forma:

p	q	$p \rightarrow q$
V	V	V
V	F	F
F	V	V
F	F	V

Numa proposição do tipo $p \rightarrow q$ a proposição p recebe o nome de **hipótese** (antecedente ou premissa) e a q chama-se **tese** (conclusão ou consequente). A proposição $p \rightarrow q$ também é muitas vezes designada por **declaração condicional**. Estas designações são compatíveis com o uso da implicação em linguagem corrente, devemos no entanto notar que a tabela entra em conflito com a interpretação que fazemos da implicação: neste caso não se dirá “ p implica q ” quando se sabe à priori que p é falsa. Na interpretação que apresentamos para a implicação ela é verdade sempre que “ p ” é falsa independentemente do valor lógico de “ q ”. Esta situação pode ilustrar-se com a implicação “se $1+1=1$ então $2=3$ ” que é verdadeira, uma vez que o antecedente é falso.

2.9 Bi-implicação

Sejam p e q proposições. A **bi-condicional** ou **bi-implicação** de p e q é a proposição $p \leftrightarrow q$ que é verdadeira quando p e q têm o mesmo valor lógico.

A tabela de verdade de $p \leftrightarrow q$ toma assim a forma:

p	q	$p \leftrightarrow q$
V	V	V
V	F	F
F	V	F
F	F	V

A proposição $p \leftrightarrow q$ deve ler-se “ p se e só se q ” (abreviado por “ p sse q ”) ou “ p é condição necessária e suficiente para q ”.

Facilmente podemos mostrar que as proposições $p \leftrightarrow q$ e $(p \rightarrow q) \wedge (q \rightarrow p)$ têm os mesmos valores lógicos, ou seja a proposição $(p \leftrightarrow q) \leftrightarrow ((p \rightarrow q) \wedge (q \rightarrow p))$ é sempre verdadeira.

$(p \leftrightarrow q)$	\leftrightarrow	$((p \rightarrow q) \wedge (q \rightarrow p))$	\wedge
V	V	V	V
V	F	F	F
F	F	V	F
F	V	F	F
1	2	1	4
		1	2
		3	1
		1	2
		1	1

Exercício 2.9.1. Suponhamos que p, q, r representam as seguintes sentenças:

- p : “7 é um número inteiro par”
- q : $3 + 1 = 4$
- r : “24 é divisível por 8”

1. Escreva em linguagem simbólica as proposições

- $3 + 1 \neq 4$ e 24 é divisível por 8
- não é verdade que 7 seja ímpar ou $3+1=4$
- se $3+1=4$ então 24 não é divisível por 8

2. Escreva por palavras as sentenças

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

Exercício 2.9.2. Construir as tabelas de verdade das seguintes proposições

1. $((p \rightarrow q) \wedge p) \rightarrow q$
2. $p \leftrightarrow (q \rightarrow r)$
3. $(p \wedge (\neg p)) \rightarrow q$
4. $((p \vee r) \wedge (q \vee r)) \wedge ((\neg p) \vee (\neg r))$

$$5. (p \wedge (q \vee r)) \wedge (q \wedge (p \vee r))$$

Exercício 2.9.3. Quantas linhas tem a tabela de verdade de uma proposição com n variáveis proposicionais?

2.10 Ordem de precedência das conectivas lógicas

Até aqui, temos usado parêntesis para definir a ordem de aplicação dos operadores lógicos numa proposição composta. Por forma a reduzir o número de parêntesis adotamos a seguinte convenção: Sempre que numa expressão estiverem presentes várias operações lógicas, **convenciona-se**, na ausência de parêntesis, que as operações se efectuem na ordem seguinte:

1. a negação;
2. a conjunção e a disjunção;
3. a implicação e a bi-implicação.

Exemplo 2.10.1. Assim,

1. $p \rightarrow ((\neg p) \vee r)$ pode escrever-se $p \rightarrow \neg p \vee r$;
2. $(p \wedge (\neg q)) \leftrightarrow c$ pode escrever-se $p \wedge \neg q \leftrightarrow c$;
3. $p \vee q \wedge \neg r \rightarrow p \rightarrow \neg q$ deve ser entendida como $((p \vee q) \wedge (\neg r)) \rightarrow p \rightarrow (\neg q)$.

2.11 Tautologia

Chama-se **tautologia** (ou fórmula logicamente verdadeira) a uma proposição que é verdadeira, para quaisquer que sejam os valores lógicos atribuídos às variáveis proposicionais que a compõem. Dito de outra forma, chama-se tautologia a uma proposição cuja coluna correspondente na tabela de verdade possui apenas Vs ou 1s. Exemplo duma tautologia é a proposição $p \vee (\neg p)$, designada de "Princípio do terceiro excluído",

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

A negação duma tautologia, ou seja uma proposição que é sempre falsa, diz-se uma **contra-tautologia** ou **contradição**. Se uma proposição não é nem uma tautologia nem uma contradição denomina-se por **contingência**.

Não deve confundir-se contradição com proposição falsa, assim como não deve confundir-se tautologia com proposição verdadeira. O facto de uma tautologia ser sempre verdadeira e uma contradição ser sempre falsa deve-se à sua forma lógica (sintaxe) e não ao significado que se lhes pode atribuir (semântica).

A tabela de verdade

p	q	$p \vee q$	$p \rightarrow (p \vee q)$	$p \rightarrow q$	$\neg q$	$p \wedge (\neg q)$	$(p \rightarrow q) \wedge [p \wedge (\neg q)]$
1	1	1	1	1	0	0	0
1	0	1	1	0	1	1	0
0	1	1	1	1	0	0	0
0	0	0	1	1	1	0	0

mostra que $p \rightarrow (p \vee q)$ é uma tautologia, enquanto que $(p \rightarrow q) \wedge (p \wedge (\neg q))$ é uma contradição.

Exercício 2.11.1. Mostre que são tautologias:

1. $(\neg q \rightarrow \neg p) \leftrightarrow (p \rightarrow q)$
2. $(p \leftrightarrow q) \leftrightarrow ((p \rightarrow q) \wedge (q \rightarrow p))$

Exemplos de outras tautologias são apresentadas abaixo:

1.	$p \vee \neg p$	
2.	$\neg[p \wedge (\neg p)]$	
3.	$p \rightarrow p$	
4.	a) $p \leftrightarrow (p \vee p)$	idempotência
	b) $p \leftrightarrow (p \wedge p)$	idempotência
5.	$\neg\neg p \leftrightarrow p$	dupla negação
6.	a) $(p \vee q) \leftrightarrow (q \vee p)$	comutatividade
	b) $(p \wedge q) \leftrightarrow (q \wedge p)$	comutatividade
	c) $(p \rightarrow q) \leftrightarrow (q \rightarrow p)$	comutatividade
7.	a) $(p \vee (q \vee r)) \leftrightarrow ((p \vee q) \vee r)$	associatividade
	b) $(p \wedge (q \wedge r)) \leftrightarrow ((p \wedge q) \wedge r)$	associatividade
8.	a) $(p \wedge (q \vee r)) \leftrightarrow ((p \wedge q) \vee (p \wedge r))$	distributividade
	b) $(p \vee (q \wedge r)) \leftrightarrow ((p \vee q) \wedge (p \vee r))$	distributividade
9.	a) $(p \vee 0) \leftrightarrow p$	identidade
	b) $(p \wedge 0) \leftrightarrow 0$	identidade
	c) $(p \vee 1) \leftrightarrow 1$	identidade
	d) $(p \wedge 1) \leftrightarrow p$	identidade
10.	a) $\neg(p \wedge q) \leftrightarrow (\neg p \vee \neg q)$	lei de Morgan
	b) $\neg(p \vee q) \leftrightarrow (\neg p \wedge \neg q)$	lei de Morgan
11.	a) $(p \rightarrow q) \leftrightarrow [(p \rightarrow q) \wedge (q \rightarrow p)]$	equivalência
	b) $(p \rightarrow q) \leftrightarrow [(p \wedge q) \vee (\neg p \wedge \neg q)]$	equivalência
	c) $(p \rightarrow q) \leftrightarrow (\neg p \leftrightarrow \neg q)$	equivalência
12.	a) $(p \rightarrow q) \leftrightarrow (\neg p \vee q)$	implicação
	b) $\neg(p \rightarrow q) \leftrightarrow (p \wedge \neg q)$	implicação
13.	$(p \rightarrow q) \leftrightarrow (\neg q \rightarrow \neg p)$	contrarrecíproca
14.	$(p \rightarrow q) \leftrightarrow [(p \wedge \neg q) \rightarrow 0]$	redução ao absurdo
15.	a) $[(p \rightarrow r) \wedge (q \rightarrow r)] \leftrightarrow [(p \vee q) \rightarrow r]$	
	b) $[(p \rightarrow q) \wedge (p \rightarrow r)] \leftrightarrow [p \rightarrow (q \wedge r)]$	
16.	$[(p \wedge q) \rightarrow r] \leftrightarrow [p \rightarrow (q \rightarrow r)]$	
17.	$p \rightarrow (p \vee q)$	adição
18.	$(p \wedge q) \rightarrow p$	simplificação
19.	$[p \wedge (p \rightarrow q)] \rightarrow q$	<i>modus ponens</i>
20.	$[(p \rightarrow q) \wedge \neg q] \rightarrow \neg p$	<i>modus tollens</i>
21.	$[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$	silogismo hipotético
22.	$[(p \vee q) \wedge \neg p] \rightarrow q$	silogismo disjuntivo
23.	$(p \rightarrow 0) \rightarrow \neg p$	absurdo
24.	$[(p \rightarrow q) \wedge (r \rightarrow s)] \rightarrow [(p \vee r) \rightarrow (q \vee s)]$	
25.	$(p \rightarrow q) \rightarrow [(p \vee r) \rightarrow (q \vee r)]$	

2.12 Equivalências proposicionais

As proposições p e q dizem-se **logicamente equivalentes** se $p \leftrightarrow q$ é uma tautologia. Por $p \equiv q$ ou $p \Leftrightarrow q$ denotamos que p e q são logicamente equivalentes.

Diz-se que a proposição p **implica logicamente** a proposição q se a veracidade da primeira arrastar necessariamente a veracidade da segunda, ou seja, se a proposição $p \rightarrow q$ for uma tautologia.

$$1. \neg q \rightarrow \neg p \Leftrightarrow p \rightarrow q$$

\neg	q	\rightarrow	\neg	p	p	\rightarrow	q
F	V	V	F	V	V	V	V
V	F	F	F	V	F	F	F
F	V	V	V	F	V	V	V
V	F	V	V	F	V	F	F
	2	1	3	2	1	1	2

$$2. p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$$

$(p \leftrightarrow q)$	\leftrightarrow	$((p \rightarrow q) \wedge (q \rightarrow p))$
V	V	V
V	F	F
F	F	V
F	V	F
1	2	1
4	1	2
1	3	1
1	2	1

Deste modo, a equivalência proposicional pode ser sempre verificada através duma tabela de verdade. Em particular, as proposições p e q são equivalentes se e só se as colunas, na tabela de verdade, que determinam os seus valores lógicos coincidirem.

Exercício 2.12.1. Mostre que são exemplos de equivalências proposicionais:

- i) $\neg(p \vee \neg p) \Leftrightarrow p \wedge \neg p$
- ii) $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$
- iii) $\neg p \vee q \Leftrightarrow p \rightarrow q$
- iv) $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$

Exercício 2.12.2. Indique quais das sentenças seguintes são equivalentes:

$$1. p \wedge (\neg q)$$

$$2. p \rightarrow q$$

$$3. \neg((\neg p) \vee q)$$

$$4. q \rightarrow (\neg q)$$

$$5. (\neg p) \vee q$$

$$6. \neg(p \rightarrow q)$$

$$7. p \rightarrow (\neg q)$$

$$8. (\neg p) \rightarrow (\neg q)$$

Exercício 2.12.3. Mostre que cada uma das proposições que se seguem:

$$1. (\neg p) \vee q$$

$$2. (\neg q) \rightarrow (\neg p)$$

$$3. \neg(p \wedge (\neg q))$$

é equivalente a $p \rightarrow q$.

Exercício 2.12.4. Mostre que

1. $p \vee (q \wedge r)$ não é logicamente equivalente a $(p \vee q) \wedge r$.
2. $p \vee (q \wedge r)$ é logicamente equivalente a $(p \vee q) \wedge (p \vee r)$.
3. $p \vee (\neg(q \vee r))$ é logicamente equivalente a $(p \vee (\neg q)) \vee (\neg r)$

De seguida apresentamos exemplos de equivalências úteis para o que se segue (que podem ser verificadas através de tabelas de verdade):

Comutatividade	$p \wedge q \Leftrightarrow q \wedge p$	$p \vee q \Leftrightarrow q \vee p$
Associativa	$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$	$(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$
Idempotência	$p \wedge p \Leftrightarrow p$	$p \vee p \Leftrightarrow p$
Identidade	$p \wedge V \Leftrightarrow p$	$p \vee F \Leftrightarrow p$
Dominância	$p \wedge F \Leftrightarrow F$	$p \vee V \Leftrightarrow V$
Absorção	$p \wedge (p \vee r) \Leftrightarrow p$	$p \vee (p \wedge r) \Leftrightarrow p$
Distributivas	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
Distributivas	$p \rightarrow (q \vee r) \Leftrightarrow (p \rightarrow q) \vee (p \rightarrow r)$	$p \rightarrow (q \wedge r) \Leftrightarrow (p \rightarrow q) \wedge (p \rightarrow r)$
Leis de De Morgan	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$
Def. Implicação	$p \rightarrow q \Leftrightarrow \neg p \vee q$	$p \rightarrow q \Leftrightarrow \neg(p \wedge \neg q)$
Def. Bi-condicional	$p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$	$p \leftrightarrow q \Leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p)$
Negação	$\neg(\neg p) \Leftrightarrow p$	
Contraposição	$p \rightarrow q \Leftrightarrow \neg q \rightarrow \neg p$	
Troca de premissas	$p \rightarrow (q \rightarrow r) \Leftrightarrow q \rightarrow (p \rightarrow r)$	

As equivalências lógicas apresentadas na tabela anterior, podem ser usadas na determinação de equivalências lógicas adicionais. Isso porque, podemos numa proposição composta, **substituir proposições por proposições que lhes sejam equivalentes** sem que isso altere os valores de verdade da proposição original.

Por exemplo:

$$\begin{aligned}
 \neg(p \vee (\neg p \wedge q)) &\Leftrightarrow \neg p \wedge \neg(\neg p \wedge q) && \text{da segunda lei de De Morgan} \\
 &\Leftrightarrow \neg p \wedge [\neg(\neg p) \vee \neg q] && \text{da primeira lei de De Morgan} \\
 &\Leftrightarrow \neg p \wedge (p \vee \neg q) && \text{da lei da dupla negação} \\
 &\Leftrightarrow (\neg p \wedge p) \vee (\neg p \wedge \neg q) && \text{da segunda distributividade} \\
 &\Leftrightarrow F \vee (\neg p \wedge \neg q) && \text{já que } \neg p \wedge p \Leftrightarrow F \\
 &\Leftrightarrow \neg p \wedge \neg q && \text{da lei identidade}
 \end{aligned}$$

Donde podemos concluir que $\neg(p \vee (\neg p \wedge q))$ e $\neg p \wedge \neg q$ são proposições logicamente equivalentes:

$$\neg(p \vee (\neg p \wedge q)) \Leftrightarrow \neg p \wedge \neg q$$

Exercício 2.12.5. Simplifique as seguintes proposições:

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

2. $\neg(p \vee (q \wedge (\neg r))) \wedge q$
3. $\neg((\neg p) \wedge (\neg q))$
4. $\neg((\neg p) \vee q) \vee (p \wedge (\neg r))$
5. $(p \wedge q) \vee (p \wedge (\neg q))$
6. $(p \wedge r) \vee ((\neg r) \wedge (p \vee q))$

Exercício 2.12.6. Por vezes usa-se o símbolo \downarrow para construir proposições compostas $p \downarrow q$ definidas por duas proposições p e q , que é verdadeira quando e só quando p e q são simultaneamente falsas, e é falsa em todos os outros casos. A proposição $p \downarrow q$ lê-se "nem p nem q ".

1. Apresente a tabela de verdade de $p \downarrow q$.
2. Expresse $p \downarrow q$ em termos das conectivas \wedge, \vee e \neg .
3. Determine proposições apenas definidas pela conectiva \downarrow que sejam equivalentes a $\neg p$, $p \wedge q$ e $p \vee q$.

Exercício 2.12.7. Expresse a proposição $p \leftrightarrow q$ usando apenas os símbolos \wedge, \vee e \neg .

2.13 Considerações sobre a implicação

As duas primeiras linhas da tabela da implicação

p	q	$p \rightarrow q$
V	V	V
V	F	F
F	V	V
F	F	V

não apresentam qualquer problema sob o ponto de vista intuitivo do senso comum. Quanto às duas últimas, qualquer outra escolha possível apresenta desvantagens sob o ponto de vista lógico, o que levou à escolha das soluções apresentadas, já que:

1. fazendo F na 3º linha e F na 4º linha, obtém-se a tabela da conjunção
2. fazendo F na 3º linha e V na 4º linha, obtém-se a tabela da bi-implicação
3. resta a possibilidade de fazer V na 3º linha e F na 4º linha que também não é, pois isso equivaleria a recusar a equivalência

$$(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$$

que é uma equivalência aconselhável, já que a proposição "se o Pedro fala, existe" é (intuitivamente) equivalente à proposição "se o Pedro não existe, não fala". A aceitação desta equivalência impõe a tabela considerada para a implicação.

\neg	q	\rightarrow	\neg	p	p	\rightarrow	q
F	V	V	F	V	V	V	V
V	F	F	F	V	V	F	F
F	V	V	V	F	F	V	V
V	F	V	V	F	F	V	F
2	1	3	2	1	1	2	1

e

A partir duma implicação r dada por $p \rightarrow q$ define-se as proposições:

- $q \rightarrow p$, designada de **recíproca** da implicação r ;
- $\neg q \rightarrow \neg p$, designada por **contra-recíproca** de r ;
- $\neg p \rightarrow \neg q$, designada por **inversa** de r .

Observe-se que, embora a contra-recíproca seja equivalente à proposição original, o mesmo não acontece com a recíproca (e a inversa, que lhe é equivalente) o que se pode verificar através das respectivas tabelas de verdade.

Exercício 2.13.1. Determine:

- a contra-recíproca de $(\neg p) \rightarrow q$
- a inversa de $(\neg q) \rightarrow p$
- a recíproca da inversa de $q \rightarrow (\neg p)$
- a negação de $p \rightarrow (\neg q)$

2.14 Fórmulas bem formadas

Estamos a descrever um sistema formal, denominado de cálculo proposicional clássico, que formaliza a chamada lógica clássica, ou lógica proposicional clássica. O vocabulário desse sistema tem os seguintes símbolos primitivos:

1. uma coleção de símbolos, A, B, C, \dots ou P_1, P_2, \dots ou p, q, r, \dots , chamados de símbolos proposicionais,
2. símbolos lógicos $\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \oplus$, e
3. símbolos de pontuação (e).

As expressões da linguagem do cálculo proposicional são sequências finitas destes símbolos (*strings*), como $\neg(\neg A) \vee B$. Convém distinguir entre as expressões ou fórmulas bem formadas (fbf) das demais. Isso é feito impondo um conjunto de regras gramaticais para a linguagem.

As regras gramaticais, para esta linguagem formal, definem por fbf (expressões grammaticalmente bem formadas na linguagem) unicamente aquelas expressões α obtidas por alguma das seguintes cláusulas:

1. α é igual a um símbolo proposicional A, B, C, \dots ou P_1, P_2, \dots ou p, q, r, \dots , ou

2. α é igual a $(\neg\beta)$ ou $(\beta \wedge \gamma)$ ou $(\beta \vee \gamma)$ ou $(\beta \rightarrow \gamma)$ ou $(\beta \leftrightarrow \gamma)$ ou $(\beta \oplus \gamma)$, onde onde β e γ ,
3. nada mais é uma fbf.

Note que, os parêntesis são usados para evitar ambiguidade. Por exemplo, $(\neg p \vee q)$ deve ser distinguida de $\neg(p \vee q)$. No segundo caso, o operador \neg aplica-se à fórmula $(p \vee q)$, ao passo que no primeiro ele é aplicado apenas a p .

Na escrita das fbf do cálculo proposicional adoptámos a convenção de eliminar parêntesis externos e impondo uma ordem de precedência aos operadores. Num sistema formal é frequente a definição de novos símbolos com base em símbolos lógicos primitivos. Na literatura é frequente usar como únicos símbolos primitivos, para o cálculo proposicional, \neg e \vee , a partir dos quais se pode definir os outros fazendo por exemplo:

- $p \wedge q =_{def} \neg(\neg p \vee \neg q)$
- $p \rightarrow q =_{def} \neg p \vee q$
- $p \leftrightarrow q =_{def} (p \rightarrow q) \wedge (q \rightarrow p)$

Note que neste contexto os novos símbolos $\wedge, \rightarrow, \leftrightarrow$ não fazem parte do vocabulário básico da linguagem.

Exercício 2.14.1. Quais das seguintes expressões são proposições bem formadas?

1. $\neg((A \rightarrow B) \rightarrow \neg(B \rightarrow A))$
2. $(S \wedge (((P \rightarrow Q) \wedge (\neg Q \rightarrow R)) \rightarrow (Q \vee \neg Q)))$
3. $((((A \wedge \neg B) \vee (\neg A \wedge B)) \leftrightarrow \neg \neg \neg C))$
4. $((((A \rightarrow B) \rightarrow C) \rightarrow \neg((A \vee B) \leftrightarrow \neg \neg(C \wedge A)))$

2.15 Semântica

Seja V um conjunto de variáveis proposicionais. A atribuição de valores às variáveis proposicionais é descrita por uma aplicação v de V no conjunto $\{0, 1\}$. Por $P(p_1, p_2, \dots, p_n)$ representamos uma fbf do cálculo proposicional que dependa dos símbolos proposicionais p_1, p_2, \dots, p_n . Entendendo os símbolos como variáveis proposicionais p_1, p_2, \dots, p_n no conjunto V , a atribuição v pode ser estendida a P , definindo a avaliação da fórmula $v(P)$. A avaliação de P reflecte a sua estrutura sintáctica, sendo definida em função da avaliação das suas partes (definição indutiva):

1. Se P é definida apenas pelo símbolo p_i , define-se $v(P) = v(p_i)$
2. Se $P = (\neg Q)$, define-se $v(P) = \neg v(Q)$
3. Se $P = (Q \wedge R)$, define-se $v(P) = v(Q) \wedge v(R)$
4. Se $P = (Q \vee R)$, define-se $v(P) = v(Q) \vee v(R)$

5. Se $P = (Q \rightarrow R)$, define-se $v(P) = v(Q) \rightarrow v(R)$
6. Se $P = (Q \leftrightarrow R)$, define-se $v(P) = v(Q) \leftrightarrow v(R)$
7. Se $P = (Q \oplus R)$, define-se $v(P) = v(Q) \oplus v(R)$

onde por Q e R se entende uma parte de P que é uma *fbf*, os operadores envolvidos devem ser interpretados pela correspondente tabela de verdade.

Exercício 2.15.1. Assumindo que as variáveis proposicionais p e q são verdadeiras e que r e s são falsas, avalie as fórmulas:

1. $(\neg(P \wedge Q) \wedge \neg R) \rightarrow ((Q \leftrightarrow \neg P) \rightarrow (R \rightarrow \neg S))$
2. $(P \vee Q) \rightarrow (\neg Q \rightarrow S)$
3. $(P \vee (Q \vee (R \rightarrow \neg P))) \leftrightarrow (Q \vee \neg S)$

Seja $P(p_1, p_2, \dots, p_n)$ uma fórmula proposicional nas variáveis proposicionais p_1, p_2, \dots, p_n . Se considerarmos todas as atribuições de valores de verdade possíveis a p_1, p_2, \dots, p_n , o valor de verdade de $P(p_1, p_2, \dots, p_n)$ define a tabela de verdade para P . Tal tabela com já vimos contém 2^n linhas.

2.16 Argumento

Sejam P, Q e R três proposições. Se da veracidade de P e Q se pode concluir que a veracidade da implicação

$$(P \wedge Q) \rightarrow R,$$

(ou seja, que da veracidade de P e Q resulta sempre a veracidade de R) então pode argumentar-se que da hipótese de P e Q resulta a veracidade de R . Assim, caso P e Q sejam aceites como verdadeiras bem como $(P \wedge Q) \rightarrow R$, então a veracidade de R resulta logicamente dos pressupostos. A uma tal justificação damos o nome de **argumento** e constitui o método usado para a demonstração matemática.

De um modo geral, chama-se argumento a uma sequência finita de proposições organizadas na forma seguinte

$$P_1 \wedge P_2 \wedge \dots \wedge P_n \Rightarrow Q$$

onde as proposições P_1, P_2, \dots, P_n são designadas de **premissas** (ou hipóteses) e Q a **conclusão** (ou tese). Ao fazer-se a leitura do argumento $(P_1 \wedge \dots \wedge P_n) \rightarrow Q$ é costume inserir uma das locuções "portanto", "por conseguinte", "logo", lendo-se por exemplo: " P_1, \dots, P_n , portanto, Q ". Para sugerir esta leitura usa-se, uma das seguintes notações

$$\frac{\begin{array}{c} P_1 \\ \vdots \\ P_n \end{array}}{Q} \quad \text{ou } P_1, \dots, P_n \therefore Q \quad \text{ou } \frac{P_1 \quad \dots \quad P_n}{Q}.$$

Interessa distinguir entre argumentos correctos e argumentos inválidos.

2.17 Argumento válido

Um argumento

$$P_1, \dots, P_n \therefore Q \text{ ou } \frac{P_1 \quad \dots \quad P_n}{Q}$$

diz-se **correcto** ou **válido** se a conclusão for verdadeira sempre que as premissas P_1, \dots, P_n forem simultaneamente verdadeiras e diz-se **incorrecto**, **inválido** ou **falacioso** no caso contrário, isto é, se alguma situação permitir que as premissas sejam todas verdadeiras e a conclusão falsa.

As regras que permitem passar de hipóteses feitas e resultados já demonstrados a novas proposições são conhecidas por regras de inferência. A regra de inferência mais frequentemente usada é o *modus ponens*:

$$\frac{\begin{array}{c} P \rightarrow Q \\ P \end{array}}{Q}$$

Se a implicação $P \rightarrow Q$ é verdadeira e também a proposição P , então Q tem de ser necessariamente verdadeira. A validade desta regra pode ser comprovada recorrendo a uma tabela de verdade.

p	q	$p \rightarrow q$	$p \wedge (p \rightarrow q)$	$[p \wedge (p \rightarrow q)] \rightarrow q$
1	1	1	1	1
1	0	0	0	1
0	1	1	0	1
0	0	1	0	1

neste caso dizemos que $P \rightarrow Q, P \therefore Q$ é um argumento válido.

De um modo geral,

$$P_1, \dots, P_n \therefore Q$$

é um argumento válido se e só se

$$P_1 \wedge P_2 \wedge \dots \wedge P_n \Rightarrow Q,$$

ou seja, se e só se

$$(P_1 \wedge P_2 \wedge \dots \wedge P_n) \rightarrow Q$$

for uma tautologia.

Exemplo 2.17.1. O argumento

$$p, q \rightarrow r, \neg r \therefore \neg q$$

é válido já que na tabela de verdade que relaciona os valores de verdade das proposições $p, q \rightarrow r, \neg r, \neg q$ nas linhas onde as premissas são verdadeiras a conclusão também o é.

p	q	r	p	$q \rightarrow r$	$\neg r$	$\neg q$
V	V	V	V	V	F	F
V	V	F	V	F	V	F
V	F	V	V	V	F	V
V	F	F	V	V	V	V
F	V	V	F	V	F	F
F	V	F	F	F	V	F
F	F	V	F	V	F	V
F	F	F	F	V	V	V

Exemplo 2.17.2. Mostremos que o argumento abaixo é válido:

- “Só se o Eclipse ganhar a corrida é que pagarei as minhas dívidas. Os meus credores não ficarão satisfeitos a não ser que eu pague as minhas dívidas. Logo, ou o Eclipse ganha a corrida ou os meus credores não ficarão satisfeitos.”

Interpretação:

- P : Eclipse ganha a corrida.
- Q : Pago as minhas dívidas.
- R : Os meus credores ficarão satisfeitos.

Formalização:

- $Q \rightarrow P, \neg Q \rightarrow \neg R \therefore P \vee \neg R$

Inspector de circunstâncias:

P	Q	R	$Q \rightarrow P$	$\sim Q \rightarrow \sim P$	\vdash	$P \vee \sim R$
1	1	1	1	1	1	1
1	1	0	1	1	1	1
1	0	1	1	0	1	1
1	0	0	1	1	1	1
0	1	1	0	1	0	0
0	1	0	0	1	1	1
0	0	1	1	0	0	0
0	0	0	1	1	1	1

O argumento é válido um vez que a formalização é um sequente tautológico, ou seja, a proposição

$$((Q \rightarrow P) \wedge (\neg Q \rightarrow \neg R)) \rightarrow (P \vee \neg R)$$

é uma tautologia.

Exemplo 2.17.3. Verifiquemos se o argumento abaixo é válido:

- “Ou o Eclipse ganha a corrida ou ganha o Estrela da Manhã. Se o Eclipse ganhar, Icabod ficará satisfeito. Logo, se o Estrela da Manhã ganhar, Icabod não ficará satisfeito.”

Interpretação:

- P : Eclipse ganha a corrida.
- Q : O Estrela da Manhã ganha a corrida.
- R : Icabod fica satisfeito.

Formalização:

- $P \vee Q, P \rightarrow R \therefore Q \rightarrow \neg R$

Inspector de circunstâncias:

P	Q	R	$P \vee Q$	$P \rightarrow R$	\vdash	$Q \rightarrow \neg R$
1	1	1	1	1	0	0
1	1	0	1	0	1	1
1	0	1	1	1	1	1
1	0	0	1	0	1	1
0	1	1	1	1	0	0
0	1	0	1	1	1	1
0	0	1	0	1	1	1
0	0	0	0	1	1	1

O argumento não é válido, pois há pelo menos uma circunstância que torna as premissas verdadeiras e a conclusão falsa o que faz com que a proposição

$$((P \vee Q) \wedge (P \rightarrow R)) \rightarrow (Q \rightarrow \neg R)$$

não seja uma tautologia.

Exercício 2.17.4. Formalize os seguintes argumentos e teste a sua validade usando inspectores de circunstância para determinar se a formalização é um sequente tautológico:

1. “Não existe tempo se não existe mudança. Não há mudança a não ser que existam objectos que possam mudar. Logo, ou existem alguns objectos que possam mudar ou o tempo não existe.”
2. “A vaca não existe a não ser que eu a veja. Se a vaca não existe, os campos e até a Terra não existem. Se os campos e a Terra não existem, eu não posso existir. Mas eu só posso ver a vaca se eu existir. É por isso óbvio que eu não existo.”

Podemos assim usar tautologias do tipo

$$(A_1 \wedge A_2 \wedge \dots \wedge A_n) \rightarrow B$$

para provar argumentos e definir regras de inferência.

Usando tautologias deste tipo ou inspectores de circunstância podemos mostrar serem válidas as relações de inferência da tabela que se segue.

Name	Tautologia	Regra de inferência	Notação de Gentzen
Modus Ponens	$(p \wedge (p \rightarrow q)) \rightarrow q$	$p, p \rightarrow q \therefore q$	$\frac{p \quad p \rightarrow q}{q}$
Modus Tollens	$(\neg q \wedge (p \rightarrow q)) \rightarrow \neg p$	$\neg q, p \rightarrow q \therefore \neg p$	$\frac{\neg q \quad p \rightarrow q}{\neg p}$
Silogismo Hipotético	$((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$	$p \rightarrow q, q \rightarrow r \therefore p \rightarrow r$	$\frac{p \rightarrow q \quad q \rightarrow r}{p \rightarrow r}$
Silogismo Disjuntivo	$((p \vee q) \wedge \neg p) \rightarrow q$	$p \vee q, \neg p \therefore q$	$\frac{p \vee q \quad \neg p}{q}$
Simplificação	$(p \wedge q) \rightarrow p$	$p \wedge q \therefore p$	$\frac{p}{p \wedge q}$
Adição	$p \rightarrow (p \vee q)$	$p \therefore p \vee q$	$\frac{p}{p \vee q}$
Lei da resolução	$((p \vee q) \wedge (\neg p \vee r)) \rightarrow (q \vee r)$	$p \vee q, \neg p \vee r \therefore q \vee r$	$\frac{p \vee q \quad \neg p \vee r}{q \vee r}$
Eliminação	$((p \rightarrow (q \vee r)) \wedge \neg q) \rightarrow (p \rightarrow r)$	$p \rightarrow (q \vee r), \neg q \therefore p \rightarrow r$	$\frac{p \rightarrow (q \vee r) \quad \neg q}{p \rightarrow r}$
Prova por Casos	$((p \rightarrow r) \wedge (q \rightarrow r)) \rightarrow ((p \vee q) \rightarrow r)$	$p \rightarrow r, q \rightarrow r \therefore (p \vee q) \rightarrow r$	$\frac{p \rightarrow r \quad q \rightarrow r}{(p \vee q) \rightarrow r}$
Lei da combinação	$(p \wedge q) \rightarrow (p \wedge q)$	$p, q \therefore p \wedge q$	$\frac{p \quad q}{p \wedge q}$
Redução ao absurdo	$(\neg p \rightarrow q \wedge \neg p \rightarrow \neg q) \rightarrow p$	$\neg p \rightarrow q, \neg p \rightarrow \neg q \therefore p$	$\frac{\neg p \rightarrow q \quad \neg p \rightarrow \neg q}{p}$

Exercício 2.17.5. Demonstre as seguintes regras de inferência:

1. Modus Tollens
2. Silogismo Hipotético
3. Redução ao absurdo

2.18 Demonstração válida

A sequência

$$A_1, A_2, \dots, A_n, B$$

é uma **demonstração válida** se e só se cada fórmula na sequência:

1. ou é uma hipótese,
2. ou é uma tautologia,
3. ou é derivada por uma regra de inferência tendo por premissas fórmulas que a antecedem na sequência,
4. ou é equivalente a uma fórmula que a antecede na sequência.

Neste caso a proposição $A_1 \wedge A_2 \wedge \dots \wedge A_n \rightarrow B$, e dizemos que a sequência

$$A_1, A_2, \dots, A_n, B$$

é uma demonstração válida do sequente ou argumento

$$C_1, C_2, \dots, C_m \therefore B$$

se as únicas hipóteses usadas na demonstração A_1, A_2, \dots, A_n, B , estão na sequência C_1, C_2, \dots, C_m .

A definição de demonstração válida assenta nos seguintes factos:

Teorema 2.18.1 (Substituição por equivalência). 1. Se $C_m \Leftrightarrow D$ e $C_1, C_2, \dots, C_m \therefore B$, então $C_1, C_2, \dots, C_{m-1}, D \therefore B$. O que pode ser representado por

$$\frac{C_1, C_2, \dots, C_m \therefore B \quad C_m \Leftrightarrow D}{C_1, C_2, \dots, C_{m-1}, D \therefore B}.$$

2. Se $B \Leftrightarrow D$ e $C_1, C_2, \dots, C_m \therefore B$, então $C_1, C_2, \dots, C_m \therefore D$. O que pode ser representado por

$$\frac{C_1, C_2, \dots, C_m \therefore B \quad B \Leftrightarrow D}{C_1, C_2, \dots, C_m \therefore D}.$$

Como exemplo de demonstração, seja $\Gamma = \{p, p \rightarrow q, q \rightarrow r\}$. Mostremos que $\Gamma \therefore r$. Com efeito, temos (1) p é hipótese (está em Γ); (2) $p \rightarrow q$ é hipótese (idem); (3) q (de 1 e 2, por Modus Ponens); (4) $q \rightarrow r$ é hipótese (está em Γ); (5) r (de 2 e 4, por Modus Ponens). Logo, como por hipótese $p, p \rightarrow q, q \rightarrow r$ são proposições verdadeiras temos de ter r verdadeira.

Admitindo agora que Γ tem duas fórmulas contraditórias, da forma p e $\neg p$. Prove-mos que se pode derivar qualquer fórmula q a partir de Γ (ou seja, mostramos que "duma falsidade tudo se segue"). Como p e $\neg p$ são hipóteses, proposições válidas, podemos inferir que $p \wedge \neg p$ é válida.

1	p	hipótese
2	$\neg p$	hipótese
3	$p \rightarrow (\neg q \rightarrow p)$	tautologia
4	$\neg p \rightarrow (\neg q \rightarrow \neg p)$	tautologia
5	$\neg q \rightarrow p$	Modus Ponens a 2 e 3
6	$\neg q \rightarrow \neg p$	Modus Ponens a 1 e 4
7	$(\neg q \rightarrow \neg p) \rightarrow ((\neg q \rightarrow p) \rightarrow q)$	tautologia
8	$(\neg q \rightarrow p) \rightarrow q$	Modus Ponens a 6 e 7
9	q	Modus Ponens a 5 e 8

Assim podemos concluir que se tivermos premissas contraditórias, podemos derivar qualquer proposição.

Exemplo 2.18.2. Aplicando regras de inferência, demonstremos que o argumento abaixo é correcto.

$$\neg p \rightarrow \neg q, (p \vee \neg q) \rightarrow r, \neg s \rightarrow \neg r, s \rightarrow (\neg w \vee \neg v) \therefore \neg w \vee \neg v$$

Temos:

1	$\neg p \rightarrow \neg q$	hipótese
2	$(p \vee \neg q) \rightarrow r$	hipótese
3	$\neg s \rightarrow \neg r$	hipótese
4	$s \rightarrow (\neg w \vee \neg v)$	hipótese
5	$p \vee \neg q$	Def. implicação em 1
6	r	Modus Ponens a 2 e 5
7	$r \rightarrow s$	Contraposição em 3
8	s	Modus Ponens a 6 e 7
9	$\neg w \vee \neg v$	Modus Ponens a 4 e 8

A lista de proposições que define a demonstração é conhecida por um **sequente**. A forma como foi escrita a demonstração é conhecida por **demonstração à Hilbert**. Existem outras formas de representar demonstrações. Uma das mais usadas em Teoria da Prova são as **árvores de derivação de Gentzen**, que apresentamos para este caso em abaixo.

$$\frac{s \rightarrow (\neg w \vee \neg v)}{\frac{\frac{(p \vee \neg q) \rightarrow r}{\frac{\frac{\neg p \rightarrow \neg q}{r} (5)}{s} (6)} \neg s \rightarrow \neg r (7)}{s} (8)} (9)$$

A árvore tem como folhas hipóteses e como raiz a tese, do argumento a demonstrar. Cada traço horizontal representa uma inferência ou dedução, que indexamos usando a numeração da primeira demonstração.

Exemplo 2.18.3. Aplicando regras de inferência, demonstremos que o argumento abaixo é correcto.

$$\neg(\neg p \vee q), \neg r \rightarrow \neg s, (p \wedge \neg q) \rightarrow s, \neg r \vee t \therefore t$$

Proof. Temos:

1	$\neg(\neg p \vee q)$	hipótese
2	$\neg r \rightarrow \neg s$	hipótese
3	$(p \wedge \neg q) \rightarrow s$	hipótese
4	$\neg r \vee t$	hipótese
5	$\neg\neg p \wedge \neg q$	Lei de De Morgan em 1
6	$p \wedge \neg q$	Dupla negação de 5
7	s	Modus Ponens a 3 e 6
8	$s \rightarrow r$	Modus Tollens a 2 e 7
9	r	Modus Ponens a 7 e 8
10	$r \rightarrow t$	Def. de implicação em 4
11	t	Modus Ponens a 9 e 10

□

Exercício 2.18.4. Sendo p, q, r e s quatro proposições, classifique os argumentos abaixo se são ou não válidos.

1. $(\neg p) \vee q, p \therefore q$

2. $p, p \rightarrow q, q \rightarrow r \therefore r$

3. $p \rightarrow q, r \rightarrow (\neg q) \therefore p \rightarrow (\neg r)$
4. $(\neg p) \vee q, (\neg r) \rightarrow (\neg q) \therefore p \rightarrow (\neg r)$

2.19 Propriedades do operador \therefore

De forma simples podemos mostrar que o operador \therefore satisfaz as seguintes propriedades:

- **(Inclusão)** Para todo $p \in \Gamma$, tem-se que $\Gamma \therefore p$.
- **(Monotonia)** Se $\Gamma \subseteq \Delta$ e se $\Gamma \therefore p$, então $\Delta \therefore p$. Informalmente, se algo é dedutível a partir dum dado conjunto de premissas Γ , continua a ser dedutível em qualquer conjunto de premissas contendo Γ .
- **(Corte)** Se $\Delta \therefore p$ e de $\Gamma \therefore q$ para cada $q \in \Delta$, então $\Gamma \therefore p$.
- **(Teorema da Dedução)** Se $\Gamma, p \therefore q$ então $\Gamma \therefore p \rightarrow q$.

Assim, por exemplo, como $p, p \rightarrow q \therefore q$, pelo Teorema da Dedução, temos que $p \therefore (p \rightarrow q) \rightarrow q$, ou seja $\therefore p \rightarrow ((p \rightarrow q) \rightarrow q)$. Donde podemos concluir que $p \rightarrow ((p \rightarrow q) \rightarrow q)$ é uma tautologia, já que é "sempre verdadeira" o seu valor de verdade não depende de nenhuma hipótese. Assim, sempre que a proposição P é uma tautologia escrevemos que $\therefore P$ já que o seu valor de verdade não depende de nenhuma hipótese.

Exercício 2.19.1. Mostre que:

1. $\therefore (\neg p \rightarrow p) \rightarrow p$ (outra forma de redução ao absurdo)
2. $\therefore (p \rightarrow q) \rightarrow ((p \rightarrow \neg q) \rightarrow \neg p)$ (redução ao absurdo intuicionista)
3. $\therefore p \rightarrow p \vee q$ (regra da adição)
4. $\therefore \neg p \rightarrow (p \rightarrow q)$
5. $\therefore (p \rightarrow \neg q) \rightarrow (q \rightarrow \neg p)$
6. $\therefore (p \wedge q) \leftrightarrow \neg(\neg p \wedge \neg q)$ (regra de De Morgan)
7. $\therefore \neg(p \wedge \neg p)$ (lei da contradição)
8. $\therefore (p \wedge \neg p) \rightarrow q$ (Lei de Duns Scotus)

2.20 EXERCÍCIOS DE REVISÃO

Exercício 2.20.1. Usando as afirmações

$$\begin{aligned} R: & O Nuno é rico \\ H: & O Nuno é feliz \end{aligned}$$

escreva as seguintes afirmações na forma simbólica

1. O Nuno é pobre mas feliz.

2. O Nuno é rico ou feliz.
3. O Nuno não é nem rico nem feliz.
4. O Nuno é pobre ou ele é simultaneamente rico e feliz.

Exercício 2.20.2. Quatro indivíduos são suspeitos de terem cometido um crime. É sabido que um e só um deles cometeu o crime. Quando interrogados pela polícia fizeram as seguintes afirmações:

Artur : Foi o José que cometeu o crime.

José : Foi o Tiago que cometeu o crime,

Gabriel : Eu não o fiz.

Tiago : José mente quando diz que fui eu.

Se exactamente uma destas afirmações é falsa, quem foi o criminoso?

Exercício 2.20.3. Quais das seguintes expressões são proposições bem formadas?

1. $\neg((A \rightarrow B) \rightarrow \neg(B \rightarrow A))$
2. $(S \wedge (((P \rightarrow Q) \wedge (\neg Q \rightarrow R)) \rightarrow (Q \vee \neg Q)))$
3. $((((A \wedge \neg B) \vee (\neg A \wedge B)) \leftrightarrow \neg \neg \neg C))$
4. $((((A \rightarrow B) \rightarrow C) \rightarrow \neg((A \vee B) \leftrightarrow \neg \neg(C \wedge A)))$

Exercício 2.20.4. Construa a tabela de verdade para as seguintes fórmulas:

1. $\neg(\neg P \vee \neg Q)$
2. $\neg(\neg P \wedge \neg Q)$
3. $P \wedge (P \vee Q)$
4. $P \wedge (Q \wedge P)$
5. $(\neg P \wedge (\neg Q \wedge R)) \vee (Q \wedge R) \vee (P \wedge R)$
6. $(P \wedge Q) \vee (\neg P \wedge Q) \vee (P \wedge \neg Q) \vee (\neg P \wedge \neg Q)$

Exercício 2.20.5. Assumindo que as variáveis proposicionais P e Q são verdadeiras e que R e S são falsas, determine o valor de verdade das afirmações:

1. $P \wedge (Q \vee R)$
2. $(P \wedge (Q \wedge R)) \vee \neg((P \vee Q) \wedge (R \vee S))$
3. $(\neg(P \wedge Q) \vee \neg R) \vee (((\neg P \wedge Q) \vee \neg R) \wedge S)$

Exercício 2.20.6. Mostre que o valor lógico das fórmulas apresentadas abaixo é independente das suas componentes:

1. $(P \wedge (P \rightarrow Q)) \rightarrow Q$
2. $(P \rightarrow Q) \leftrightarrow (\neg P \vee Q)$
3. $((P \rightarrow Q) \wedge (Q \rightarrow R)) \rightarrow (P \rightarrow R)$
4. $(P \leftrightarrow Q) \leftrightarrow ((P \wedge Q) \vee (\neg P \wedge \neg Q))$

Exercício 2.20.7. Construa a tabela de verdade das seguintes fórmulas:

1. $(Q \wedge (P \rightarrow Q)) \rightarrow P$
2. $\neg(P \vee (Q \wedge R)) \leftrightarrow ((P \vee Q) \wedge (P \vee R))$

Exercício 2.20.8. Assumindo que as variáveis proposicionais P e Q são verdadeiras e que R e S são falsas, determine os valores de verdade das seguintes fórmulas.

1. $(\neg(P \wedge Q) \vee \neg R) \vee ((Q \leftrightarrow \neg P) \rightarrow (R \vee \neg S))$
2. $(P \leftrightarrow Q) \wedge (\neg Q \rightarrow S)$
3. $(P \vee (Q \rightarrow (R \wedge \neg P))) \leftrightarrow (Q \vee \neg S)$

Exercício 2.20.9. Suponha-se que se define uma nova conectiva, denotada por $*$, tal que $p * q$ é verdadeira quando q é verdadeira e p falsa, e é falsa em todos os outros casos. Construa as tabelas de verdade para

1. $p * q$
2. $q * p$
3. $(p * q) * p$

Exercício 2.20.10. Elimine o maior número de parêntesis possível sem alterar o significado das expressões:

1. $((p \rightarrow (\neg q)) \wedge r)$
2. $(p \vee (q \vee r))$
3. $((((p \wedge (\neg q)) \wedge r) \vee s)$
4. $((p \vee (\neg q)) \vee (p \wedge q))$
5. $((p \leftrightarrow q) \leftrightarrow (\neg(r \vee s)))$

Exercício 2.20.11. Reponha os parêntesis:

1. $s \vee \neg q \wedge r$
2. $s \rightarrow \neg \neg \neg \neg q \wedge r$
3. $s \rightarrow \neg(q \wedge r \rightarrow s) \wedge \wedge q \leftrightarrow r$
4. $s \rightarrow r \rightarrow r \leftrightarrow \neg r \vee t$

Exercício 2.20.12. Mostre as seguintes equivalências

1. $P \rightarrow (Q \rightarrow P) \Leftrightarrow \neg P \rightarrow (P \rightarrow Q)$
2. $P \rightarrow (Q \vee R) \Leftrightarrow (P \rightarrow Q) \vee (P \rightarrow R)$
3. $(P \rightarrow Q) \wedge (R \rightarrow Q) \Leftrightarrow (P \rightarrow Q) \vee (P \rightarrow R)$
4. $\neg(P \leftrightarrow Q) \Leftrightarrow (P \vee Q) \wedge \neg(P \vee Q)$

Exercício 2.20.13. Mostre que P é equivalente às seguintes fórmulas $\neg\neg P$, $P \wedge P$, $P \vee P$, $P \vee (P \wedge Q)$, $P \wedge (P \vee Q)$, $(P \wedge Q) \vee (P \wedge \neg Q)$, e $(P \vee Q) \wedge (P \vee \neg Q)$.

Exercício 2.20.14. Mostre as seguintes equivalências

1. $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$
2. $\neg(P \wedge Q) \Leftrightarrow \neg P \wedge \neg Q$
3. $\neg(P \rightarrow Q) \Leftrightarrow P \wedge \neg Q$
4. $\neg(P \leftrightarrow Q) \Leftrightarrow (P \wedge \neg Q) \vee (\neg P \wedge Q)$

Exercício 2.20.15. Mostre as seguintes equivalências:

1. $A \rightarrow (P \vee C) \Leftrightarrow (A \wedge \neg P) \rightarrow C$
2. $(P \rightarrow C) \wedge (Q \rightarrow C) \Leftrightarrow (P \vee Q) \rightarrow C$
3. $((Q \wedge A) \rightarrow C) \wedge (A \rightarrow (P \vee C)) \Leftrightarrow (A \wedge (P \rightarrow Q)) \rightarrow C$
4. $((P \wedge Q \wedge A) \rightarrow C) \wedge (A \rightarrow (P \vee Q \vee C)) \Leftrightarrow (A \wedge (P \leftrightarrow Q)) \rightarrow C$

Exercício 2.20.16. Simplifique as fórmulas abaixo:

1. $((P \rightarrow Q) \leftrightarrow (\neg Q \rightarrow \neg P)) \wedge R$
2. $P \vee (\neg P \vee (Q \wedge \neg Q))$
3. $(P \wedge (Q \wedge S)) \vee (\neg P \wedge (Q \wedge S))$

Exercício 2.20.17. Mostre as seguintes implicações lógicas

1. $(P \wedge Q) \Rightarrow (P \rightarrow Q)$
2. $P \Rightarrow (Q \rightarrow P)$
3. $(P \rightarrow (Q \rightarrow R)) \Rightarrow ((P \rightarrow Q) \rightarrow (P \rightarrow R))$

Exercício 2.20.18. Escreva as fórmulas abaixo de forma equivalente, mas onde a negação seja aplicada apenas a variáveis.

1. $\neg(P \vee Q)$
2. $\neg(P \wedge Q)$
3. $\neg(P \rightarrow Q)$

$$4. \neg(P \leftrightarrow Q)$$

Exercício 2.20.19. Mostre que a conclusão C segue das premissas H_1, H_2, H_3 nos seguintes casos:

1. $H_1 : P \rightarrow Q, \therefore C : P \rightarrow (P \wedge Q)$
2. $H_1 : \neg P \vee Q, H_2 : \neg(Q \wedge \neg R), H_3 : \neg R, \therefore C : \neg P$
3. $H_1 : \neg P, H_2 : P \vee Q \therefore C : Q$
4. $H_1 : \neg Q, H_2 : P \rightarrow Q \therefore C : \neg P$
5. $H_1 : P \rightarrow Q, H_2 : Q \rightarrow R \therefore C : P \rightarrow R$
6. $H_1 : R, H_2 : P \vee \neg P \therefore C : R$

Exercício 2.20.20. Mostre a validade dos seguintes argumentos, onde as premissas aparecem à esquerda e as conclusões à direita:

- | | |
|--|---------------------------------|
| (a) $\neg(P \wedge \neg Q), \neg Q \vee R, \neg R$ | $\therefore \neg P$ |
| (b) $(A \rightarrow B) \wedge (A \rightarrow C), \neg(B \wedge C), D \vee A$ | $\therefore D$ |
| (c) $\neg J \rightarrow (M \vee N), (H \vee G) \rightarrow \neg J, H \vee G$ | $\therefore M \vee N$ |
| (d) $P \rightarrow Q, (\neg Q \vee R) \wedge \neg R, \neg(\neg P \wedge S)$ | $\therefore \neg S$ |
| (e) $(P \wedge Q) \rightarrow R, \neg R \vee S, \neg S$ | $\therefore \neg P \vee \neg Q$ |
| (f) $P \rightarrow Q, Q \rightarrow \neg R, R, P \vee (J \wedge S)$ | $\therefore J \wedge S$ |

Exercício 2.20.21. Sendo p, q, r e s quatro proposições, classifique os argumentos abaixo se são ou não válidos.

1. $p \vee (\neg q), \neg q \therefore p$
2. $\neg p \therefore p \rightarrow q$
3. $(p \wedge q) \rightarrow (r \wedge s), \neg r \therefore (\neg p) \vee (\neg q)$
4. $p \rightarrow q, (\neg q) \rightarrow (\neg r), s \rightarrow (p \vee r), s \therefore q$
5. $p \vee q, q \rightarrow (\neg r), (\neg r) \rightarrow (\neg p) \therefore \neg(p \wedge q)$
6. $p \rightarrow q, (\neg r) \rightarrow (\neg q), r \rightarrow (\neg p) \therefore \neg p$
7. $p \rightarrow (\neg p) \therefore \neg p$
8. $p \vee q, p \rightarrow r, \neg r \therefore q$
9. $p, q \rightarrow (\neg p), (\neg q) \rightarrow (r \vee (\neg s)), \neg r \therefore \neg s$
10. $p \rightarrow (q \vee s), q \rightarrow r \therefore p \rightarrow (r \vee s)$
11. $p \rightarrow (\neg q), q \rightarrow p, r \rightarrow p \therefore \neg q$
12. $p \rightarrow q, r \rightarrow s, \neg(p \rightarrow s) \therefore q \wedge (\neg r)$

Exercício 2.20.22. Quais dos seguintes argumentos são válidos?

1. $P \rightarrow Q, \neg Q \rightarrow R, \neg R, \therefore P$
2. $A \rightarrow (A \rightarrow (B \rightarrow C)), B, \therefore A \rightarrow C$
3. Se a Rute comprou um carro de luxo, foi porque ou assaltou um banco ou o seu tio rico morreu. Rute não assaltou um banco ou o seu tio rico não morreu. Consequentemente, o seu tio rico não morreu.
4. Hoje é domingo. Amanhã não é domingo. Consequentemente a Lua é feita de queijo verde.
5. O livro está na secretaria ou na estante. Não está na estante. Consequentemente, está na secretaria.
6. Se a função f não é contínua, então não é diferenciável. A função f é diferenciável. Consequentemente, a função f é contínua.
7. Se existe vida em Marte, então os especialistas estão enganados e o governo está a mentir. Se o governo está a mentir, então os especialistas estão certos ou não existe vida em Marte. O governo está a mentir. Consequentemente, existe vida em Marte.
8. (Lewis Carroll) Os bebés são ilógicos. Ninguém que consiga domar um crocodilo deve ser menosprezado. Pessoas ilógicas são menosprezadas. Consequentemente, os bebés não conseguem domar crocodilos.
9. (Lewis Carroll) Nenhum cão de caça vagueia pelo Zodíaco. Apenas os cometas vagueiam pelo Zodiaco. Só os cães de caça tem a cauda encaracolada. Consequentemente, nenhum cometa tem a cauda enrolada.
10. As frutas verdes não são saudáveis. Todas estas maçãs não são saudáveis. Nenhum fruto, que tenha crescido na escuridão é saudável. Estas maçãs não cresceram ao sol. Consequentemente, toda a fruta madura é saudável.

3

Lógica de predicados

Existem muitas afirmações que se fazem em Matemática que não podem ser simbolizadas e analisadas em termos de cálculo proposicional. Para além da complexidade externa introduzida pelas diferentes conectivas, uma afirmação pode conter complexidades internas que advém de palavras tais como "todo", "cada", "algum", etc. as quais requerem uma análise lógica que está para além do cálculo proposicional. Tal análise é o objecto da chamada **Lógica de Predicados**.

3.1 Variáveis

No desenvolvimento de qualquer teoria, aparecem muitas vezes afirmações sobre objectos genéricos da teoria, que são representados por letras designadas de **variáveis**.

3.2 Predicado

Representando por x um número inteiro genérico, pode ser necessário analisar (sob o ponto de vista lógico) afirmações do tipo

" x é maior que 3"

Como já foi referido, tal afirmação não é uma proposição: o seu valor lógico tanto pode ser verdade ou falso. Uma afirmação deste tipo denota-se genericamente por $P(x)$ para mostrar que P depende da variável x , obtendo-se assim, uma **fórmula** com uma **variável livre** x . Para evidenciar isso muitas vezes escrevemos:

$P(x) :$ " x é maior que 3"

Substituindo x em $P(x)$ por um dado valor, 6 por exemplo, obtém-se $P(6)$ que é uma proposição: $P(6)$ é uma proposição verdadeira; $P(2)$, no entanto, é uma proposição

falsa. Neste sentido, P é designada por **função proposicional ou predicado**, e $P(x)$ é designada de **expressão proposicional ou condição**

3.3 Conjuntos de verdade

Quando se estudam proposições - fórmulas sem variáveis livres - pode falar-se no seu valor lógico de verdade ou falsidade. Mas se uma fórmula contiver variáveis livres (uma ou mais) então não poderá falar-se no seu valor lógico e dizer simplesmente que tal fórmula é verdadeira ou falsa. O seu valor lógico depende do valor atribuído à variável (ou variáveis). A tais predicados associam-se então os chamados **conjuntos de verdade** que são os conjuntos de valores para os quais $p(x)$ é verdadeira. Escreve-se com este sentido

$$A = \{x : P(x)\}$$

o que se lê da seguinte forma: A é o conjunto cujos elementos satisfazem $P(x)$ ou para os quais $P(x)$ é verdadeira. Observe-se que, reciprocamente, dado um conjunto A qualquer, pode sempre definir-se uma fórmula com variáveis livres que tem A por conjunto de verdade. Isso pode ser feito, por exemplo, pelo predicado

$$P_A(x) :: x \in A''$$

e portanto

$$A = \{x : P_A(x)\}$$

3.4 Paradoxo de Russel

Neste contexto a existência dum **conjunto universal** $\{x : x = x\}$ e dum **conjunto vazio** $\{x : x \neq x\}$ intuitivamente podem parecer razoáveis. No entanto, podem conduzir a situações paradoxais, como deu conta Bertrand Russel, por volta de 1901.

Russel começou por notar que se se adoptar a concepção intuitiva de conjunto, pode dizer-se que alguns conjuntos são membros de si próprios enquanto outros não o são. Um conjunto de cadeiras, por exemplo não é uma cadeira, não sendo assim um elemento de si próprio; no entanto, o conjunto de todas as ideias abstractas é, ele próprio, uma ideia abstracta, pelo que pertence a si própria. As propriedades "ser elemento de si próprio" e "não ser membro de si próprio" parecem assim ser propriedades perfeitamente adequadas para definir conjuntos.

Seja R o conjunto de todos os conjuntos que não estão contidos em si próprio. Formalmente seja

$$R = \{x : x \notin x\}.$$

Os problemas surgem quando nos questionamos se R é elemento ou não de R . Porque:

- Supondo que $R \in R$ temos por definição de R que $R \notin R$,
- Supondo que $R \notin R$ temos por definição de R que $R \in R$.

Assim, a proposição $R \in R$ não tem valor lógico definido na Lógica Clássica contradizendo o **Princípio do terceiro excluído**.

Para evitar este tipo de problema vamos supor que todos os conjuntos considerados são constituídos por elementos de um conjunto \mathcal{U} suficientemente grande, chamado **conjunto universal** ou **universo de discurso**. Neste sentido, um conjunto é uma entidade do tipo

$$\{x \in \mathcal{U} : P(x)\}$$

onde supomos que o predicado P está definido para todo o valor de x em \mathcal{U} .

A ideia de um conjunto universal estará sempre presente, mesmo quando não seja explicitamente mencionado.

3.5 Universo de discurso

A atribuição de valores às variáveis duma condição $P(x_1, x_2, \dots, x_n)$ permite definir proposições. No entanto esta atribuição é limitada a um conjunto de valores do **universo de discurso** do predicado.

Exemplo 3.5.1. Temos por exemplo:

- $x = x$ tem por exemplos de universo de discurso: $\mathbb{R}, \mathbb{N}, \mathbb{Q}$ ou \mathbb{C} .
- $\sqrt{y} \geq 4$ tem por exemplos de universo de discurso: $\mathbb{Q}^+, \mathbb{R}^+$
- $\frac{1}{x} > 3$ tem como possível universo de discurso por exemplo o conjunto dos reais diferentes de zero.

3.6 Predicados impossíveis, possíveis e universais

Podemos assim classificar as expressões proposicionais em

- **Impossíveis**, se não existem valores da variável, ou variáveis, no universo, ou universos, que a transformam numa proposição verdadeira.
- **Possíveis**, se existem valores das variáveis no universo de discurso que a transformam numa proposição verdadeira. Podemos assim distinguir entre proposições:
 - **Universais**, se são predicados verificados para todos os valores que as variáveis podem assumir no universo,
 - **Contingentes (ou não universais)**, se não são predicados verificadas para todos os valores das variáveis.

Exemplo 3.6.1. Temos por exemplo:

- $x^2 + y^2 > -4$, com $x \in \mathbb{R}$ e $y \in \mathbb{R}$, é condição universal;
- $2n + 1 > 0$ é condição universal em \mathbb{N} ;
- $|x| + |y| > 0$, com x e y reais, é condição contingente, pois não é satisfeita com $x = 0$ e $y = 0$;

- $n^2 > 5$ é condição contingente em \mathbb{N} , pois é condição possível não universal.

3.7 Conjuntos de verdade e conectivas lógicas

Suponha-se que A é o conjunto de verdade da expressão proposicional $P(x)$ e B é o conjunto de verdade da expressão proposicional $Q(x)$ no universo de discurso \mathcal{U} . Então

$$A = \{x \in \mathcal{U} : P(x)\}$$

$$B = \{x \in \mathcal{U} : Q(x)\}$$

O conjunto de verdade da fórmula $P(x) \wedge Q(x)$ é tal que

$$\{x \in \mathcal{U} : P(x) \wedge Q(x)\} = A \cap B$$

De modo semelhante,

$$\{x \in \mathcal{U} : P(x) \vee Q(x)\} = A \cup B$$

Exercício 3.7.1. Sendo $A = \{x \in \mathcal{U} : P(x)\}$ e $B = \{x \in \mathcal{U} : Q(x)\}$. Determine os conjuntos de verdade das expressões proposicionais:

1. $\neg P(x)$
2. $\neg Q(x)$
3. $P(x) \wedge (\neg Q(x))$
4. $P(x) \rightarrow Q(x)$
5. $P(x) \leftrightarrow Q(x)$

3.8 O quantificador universal

Como se referiu acima, uma fórmula $P(x)$, contendo uma variável x , pode ser verdadeira para alguns valores de x pertencentes ao universo do discurso, e falsa para outros. Por vezes pretende-se dizer que uma dada fórmula $P(x)$ se verifica para todos os elementos x do universo, i.e. $\{x \in \mathcal{U} : P(x)\} = \mathcal{U}$. Escreve-se então

“para todo o x , $P(x)$ ”

o que se representa, simbolicamente, por

$$\forall x P(x).$$

O símbolo \forall é designado por quantificador universal. A fórmula anterior diz que $P(x)$ se verifica para todo o elemento x ou que $P(x)$ se verifica universalmente. Sendo \mathcal{U} o universo de discurso, escrever $\forall x P(x)$ é equivalente a

$$\forall x[x \in \mathcal{U} \rightarrow P(x)].$$

A quantificação universal pode ser feita apenas sobre uma parte de \mathcal{U} . Assim, se D designar um subconjunto próprio de \mathcal{U} e $P(x)$ for uma fórmula com uma variável cujo domínio é D , então

$$\forall x \in D : P(x) \text{ ou } \forall x [x \in D \rightarrow P(x)]$$

afirma que $P(x)$ se verifica para todo o $x \in D$.

Exemplo 3.8.1. Podemos assim avaliar as proposições abaixo

- $\forall x(x = x)$ é verdade em \mathbb{R} ;
- $\forall x \in \mathbb{R} \forall y \in \mathbb{R} : x^2 + y^2 > -4$ é verdadeira;
- $\forall n \in \mathbb{N} : 2n + 1 > 0$ é verdadeira;
- $\forall x \forall y(|x| + |y| > 0)$, é falsa no universo dos reais, pois não é satisfeita com $x = 0$ e $y = 0$;
- $\forall n \in \mathbb{N} : n^2 > 5$ é falsa.

Note que, caso D seja um conjunto finito, por exemplo

$$D = \{a_1, a_2, \dots, a_n\}$$

escrever

$$\forall x \in D : P(x)$$

é logicamente equivalente à conjunção

$$P(a_1) \wedge P(a_2) \wedge \dots \wedge P(a_n)$$

o que mostra que a proposição não tem variáveis livres, tratando-se, portanto duma proposição. O mesmo significado pode ser dado no caso em que D é um conjunto infinito correspondendo, neste caso, a um número infinito de conjunções.

Exemplo 3.8.2. Note-se que a proposição

$$\forall x \in \{2, 6, 12, 34\} : x \text{ é um número par,}$$

é uma proposição verdadeira e equivalente a escrever

$$(2 \text{ é um número par}) \wedge (6 \text{ é um número par}) \wedge (12 \text{ é um número par}) \wedge (34 \text{ é um número par}).$$

3.9 O quantificador existencial

Por outro lado, escreve-se

$$\exists x P(x)$$

para significar que existe (no universo do discurso) pelo menos um elemento x para o qual $P(x)$ se verifica, i.e. $\{x \in \mathcal{U} : P(x)\} \neq \emptyset$, o que se pode ler

"existe pelo menos um x tal que $P(x)$ ".

A fórmula $\exists x P(x)$ é uma abreviatura (usada normalmente) para a expressão

$$\exists x [x \in \mathcal{U} \wedge P(x)],$$

onde \mathcal{U} designa o universo do discurso. O símbolo \exists é chamado de **quantificador existencial**.

Se D for um subconjunto de \mathcal{U} e $P(x)$ for uma fórmula com uma variável cujo domínio é D , então escrevemos

$$\exists x \in D : P(x) \text{ ou } \exists x [x \in D \wedge P(x)].$$

Supondo que D é o conjunto finito,

$$D = \{a_1, a_2, \dots, a_n\},$$

escrever

$$\exists x \in D : P(x),$$

é logicamente equivalente à disjunção

$$P(a_1) \vee P(a_2) \vee \dots \vee P(a_n),$$

o que mostra que tal fórmula não tem variáveis livres, sendo, portanto, uma proposição. O mesmo significado pode ser dado no caso em que D é um conjunto infinito, envolvendo neste caso infinitas disjunções.

Exemplo 3.9.1. Assim, as fórmulas abaixo são proposições

- $\exists x(x^2 = 4)$ é verdade em \mathbb{R} ;
- $\exists x \in \mathbb{R} \exists y \in \mathbb{R} : x^2 + y^2 = 0$ é verdadeira;
- $\exists n \in \mathbb{N} : 2n + 1 > 0$ é verdadeira;
- $\exists y \forall x(|x| + |y| > 0)$, é verdadeira em \mathbb{R} ;
- $\exists x \in \mathbb{R} : x^2 = -4$ é falsa.

Exemplo 3.9.2. Note-se que a proposição

$$\exists x \in \{2, 6, 11, 34\} : x \text{ é ímpar},$$

é uma proposição verdadeira e equivalente a escrever

$$(2 \text{ é ímpar}) \vee (6 \text{ é ímpar}) \vee (11 \text{ é ímpar}) \vee (34 \text{ é ímpar}).$$

3.10 O universo vazio

Por uma questão de generalidade interessa considerar também o caso em que o domínio da variável da fórmula proposicional $P(x)$ é o conjunto vazio. Que valor lógico terão expressões da forma

$$\forall x [x \in \emptyset \rightarrow P(x)] \text{ e } \exists x [x \in \emptyset \wedge P(x)]?$$

Na primeira expressão a implicação é sempre verdadeira quando o antecedente é falso: é o que acontece aqui. Visto que $x \in \emptyset$ é sempre falso, então

$$\forall x [x \in \emptyset \rightarrow P(x)]$$

é uma proposição sempre verdadeira. Quando à segunda expressão ela tem a forma de uma conjunção de proposições, das quais uma é sempre falsa. Então,

$$\exists x [x \in \emptyset \wedge P(x)]$$

é uma proposição sempre falsa.

Exemplo 3.10.1. Note-se que assim a proposição

$$\exists x \in \{\} : x \text{ é ímpar},$$

é uma proposição falsa, e

$$\forall x \in \{\} : x \text{ é ímpar},$$

é uma proposição verdadeira.

3.11 Existe um e um só ...

Por vezes emprega-se o quantificador existencial numa situação simultânea de unicidade e existência, ou seja, quer afirmar se não só que

$$\exists x P(x)$$

mas ainda que a fórmula $P(x)$ se transforma numa proposição verdadeira para um e só para um elemento do domínio de quantificação, i.e. $\#\{x \in \mathcal{U} : P(x)\} = 1$. Neste caso empregam-se as abreviaturas

$$\exists!x P(x) \text{ ou } \exists^1 x P(x)$$

que significa

"**existe um e um só** x tal que $P(x)$ ".

Exemplo 3.11.1. Assim

- $\exists^1 x \in \mathbb{N} : x^2 = 4$ é uma proposição verdadeira;
- $\exists^1 x \in \mathbb{Z} : x^2 = 4$ é uma proposição falsa;

Exemplo 3.11.2. Assim a proposição

$$\exists^1 x \in \{1, 2, 3, 4, 5\} : x \text{ é ímpar},$$

é uma proposição falsa, e

$$\exists^1 x \in \{2, 3, 4\} : x \text{ é ímpar},$$

é uma proposição verdadeira.

3.12 Variáveis ligadas

Observe-se que enquanto a fórmula $P(x)$ tem uma variável livre, x , as fórmulas $\forall x P(x)$ e $\exists x P(x)$ não têm qualquer variável livre: nestas fórmulas x é sempre uma **variável ligada** (ou **muda**). Trata-se então de proposições, relativamente às quais se pode afirmar que são verdadeiras ou falsas (mas não ambas as coisas).

Exercício 3.12.1. Seja $\mathbb{N} = \{1, 2, 3, 4, \dots\}$, $P(x)$ a afirmação "x é par", $Q(x)$ a afirmação "x é divisível por 3" e $R(x)$ a afirmação "x é divisível por 4". Expressar em linguagem corrente cada uma das proposições que se seguem e determinar o seu valor lógico.

1. $\forall x \in \mathbb{N} : P(x)$
2. $\forall x \in \mathbb{N} : P(x) \vee Q(x)$
3. $\forall x \in \mathbb{N} : P(x) \rightarrow Q(x)$
4. $\forall x \in \mathbb{N} : P(x) \vee R(x)$

Exercício 3.12.2. Indicar se as proposições são sempre, às vezes ou nunca verdadeiras.

1. $(\forall x \in D : P(x)) \Rightarrow (\exists x \in D : P(x))$
2. $(\exists x \in D : P(x)) \Rightarrow (\forall x \in D : P(x))$
3. $(\forall x \in D : \neg P(x)) \Rightarrow \neg(\forall x \in D : P(x))$

3.13 Quantificadores múltiplos

Uma fórmula matemática pode ter mais de que uma variável. Considere-se, por exemplo, a afirmação:

"para cada número inteiro par n existe um número inteiro k para o qual se verifica a igualdade $n = 2k$ "

Denotando por $P(n, k)$ a fórmula $n = 2k$ e por \mathbb{P} o conjunto dos números inteiros pares, a afirmação pode ser assim apresentada simbolicamente por

$$\forall n \in \mathbb{P} \exists k \in \mathbb{Z} : P(n, k)$$

ou

$$\forall n [n \in \mathbb{P} \rightarrow \exists k [k \in \mathbb{Z} \wedge P(n, k)]]$$

que constitui uma proposição verdadeira. Outro exemplo de uma proposição com dois quantificadores é

$$\forall x \exists y x + y = 0$$

onde o domínio de quantificação é o conjunto dos números reais. Em linguagem corrente, escrever-se-ia

"para todo o número real x existe um número real y tal que $x + y = 0$ "

que constitui uma proposição verdadeira (sendo $y = -x$ para cada $x \in \mathbb{R}$). Se trocarmos os quantificadores obter-se-á

$$\exists y \forall x x + y = 0$$

que significa

"existe um número real y tal que para todo o número real x se tem $x + y = 0$ "

Esta proposição é falsa pois não existe número real y , sempre o mesmo, para o qual todo o número real x satisfaz a equação dada.

Estes exemplos ilustram a **não comutatividade** dos quantificadores universal, \forall , e existencial, \exists .

É importante notar que a **ordem** dos quantificadores é relevante:

Proposição	É verdadeira se:	É falsa se:
$\forall x \forall y P(x, y)$	$P(x, y)$ é verdadeira para todo o par x, y .	Existe um par x, y para o qual $P(x, y)$ é falso.
$\forall x \exists y P(x, y)$	Para todo x existe um y para o qual $P(x, y)$ é verdadeira.	Existe um x tal que $P(x, y)$ é falso para todo y .
$\exists x \forall y P(x, y)$	Existe um x para o qual $P(x, y)$ é verdadeira para todo y .	Para todo x existe um y tal que $P(x, y)$ é falso.
$\exists x \exists y P(x, y)$	Existe um par x, y para o qual $P(x, y)$ é verdadeira.	$P(x, y)$ é falso para todo o par x, y .

Mais geralmente, uma fórmula pode ter n variáveis

$$P(x_1, x_2, \dots, x_n).$$

Para transformar uma fórmula deste tipo numa proposição são necessários n quantificadores. Denotando um quantificador genérico (universal ou existencial) por Q , então

$$Qx_1 Qx_2 \dots Qx_n P(x_1, x_2, \dots, x_n),$$

é uma proposição. **Dois quantificadores da mesma espécie são sempre comutativos enquanto que dois quantificadores de espécies diferentes são geralmente não comutativos.**

Exercício 3.13.1. Seja $Q(x, y, z)$ a expressão " $x + y = z$ ". Determine o valor de verdade das proposições no conjunto dos números inteiros:

- $\forall x \forall y \exists z Q(x, y, z)$
- $\exists z \forall x \forall y Q(x, y, z)$

3.14 Negação de proposições Quantificadas (Segundas Leis de De Morgan)

Resumindo:

Proposição	Quando é verdadeira?	Quando é falsa?
$\forall x P(x)$	$P(x)$ é verdadeiro para todo o x	Existe um x para o qual $P(x)$ é falso
$\exists x P(x)$	Existe um x para o qual $P(x)$ é verdadeiro	$P(x)$ é falso para todo o x

Dadas as proposições com quantificadores

$$\forall x[x \in \mathcal{U} \rightarrow P(x)] \text{ e } \exists x[x \in \mathcal{U} \wedge P(x)]$$

pode ser necessário analisar (logicamente) as proposições

$$\neg(\forall x[x \in \mathcal{U} \rightarrow P(x)]) \text{ e } \neg(\exists x[x \in \mathcal{U} \wedge P(x)])$$

Tendo em conta que $p \rightarrow \neg q$ é equivalente a $\neg(p \wedge q)$, então

$$\neg(\exists x[x \in \mathcal{U} \wedge P(x)]) \Leftrightarrow \forall x \neg[x \in \mathcal{U} \wedge P(x)] \Leftrightarrow \forall x [x \in \mathcal{U} \Rightarrow \neg P(x)]$$

De modo semelhante, pode verificar-se que

$$\neg(\forall x[x \in \mathcal{U} \rightarrow P(x)])$$

equivale a

$$\exists x \neg[x \in \mathcal{U} \rightarrow P(x)] \Leftrightarrow \exists x[x \in \mathcal{U} \wedge \neg P(x)]$$

Em resumo, de um modo genérico, têm-se as equivalências

$$\neg(\forall x P(x)) \Leftrightarrow \exists x \neg P(x)$$

$$\neg(\exists x P(x)) \Leftrightarrow \forall x \neg P(x)$$

conhecidas por **Segundas Leis De Morgan**.

Exemplo 3.14.1. Usando estas propriedades temos por exemplo:

- $\neg\neg\forall x P(x) \Leftrightarrow \neg\exists x(\neg P(x)) \Leftrightarrow \forall x(\neg\neg P(x)) \Leftrightarrow \forall x P(x);$
- $\neg\neg\exists x P(x) \Leftrightarrow \neg\forall x(\neg P(x)) \Leftrightarrow \exists x(\neg\neg P(x)) \Leftrightarrow \exists x P(x);$
- $\neg\forall x\forall y P(x, y) \Leftrightarrow \exists x\neg\forall y P(x, y) \Leftrightarrow \exists x\exists y(\neg P(x, y));$
- $\neg\exists x\forall y P(x, y) \Leftrightarrow \forall x\neg\forall y P(x, y) \Leftrightarrow \forall x\exists y(\neg P(x, y)).$

3.15 Relação de pertença

A Teoria de Conjuntos é a teoria matemática que se ocupa dum universo \mathcal{U} de indivíduos a que chamamos conjuntos. Diz-se que os objecto a e A existem e são conjuntos, se pertencem ao domínio \mathcal{U} . Vamos supor que se verificam certas relações fundamentais, da forma $a \in A$, entre objectos a e A do domínio \mathcal{U} . Se para dois objectos a e A vale a relação $a \in A$, diremos que " a é um elemento de A " ou que " A contém a como

“elemento”. Um conjunto designa-se geralmente por uma letra maiúscula, reservando-se as letras minúsculas para os seus elementos. A negação de $x \in A$ representa-se simbolicamente por $x \notin A$.

3.16 Descrição

Um conjunto pode ser descrito em **extensão** (quando o número dos seus elementos for finito e suficientemente pequeno) enumerando explicitamente todos os seus elementos e colocados entre chavetas e separados por vírgulas ou em **compreensão**, enunciando uma propriedade caracterizadora dos seus elementos (isto é, uma propriedade que os seus e só os seus elementos possuem).

Exemplo 3.16.1. *Conjunto das vogais V é descrito em extensão $V = \{a, e, i, o, u\}$;*

Exemplo 3.16.2. *Conjunto L das palavras da língua portuguesa escritas com 5 letras vogais, está descrito em compreensão;*

Exemplo 3.16.3. *Conjunto dos números naturais pares pode ser descrito, em compreensão, como sendo definido por naturais n para os quais existe um natural q tal que $n = 2q$.*

Exemplo 3.16.4. $1 \in \{1, 2, 3\}$, $2 \in \{4, 2, 5\}$, $7 \notin \{1, 2, 3\}$, $\{1\} \notin \{1, 2, 3\}$

Exemplo 3.16.5. $1 \in \{n : n \text{ é um natural ímpar}\}$ $2 \notin \{n : n \text{ é um natural ímpar}\}$

3.17 Axioma da extensão

Dois conjuntos A e B são iguais se e só se têm os mesmos elementos. Neste caso escrevemos, $A = B$.

Simbolicamente temos

$$A = B \Leftrightarrow \forall x(x \in A \leftrightarrow x \in B)$$

Dada uma expressão proposicional $P(x)$ (um predicado colectivizante), há um só conjunto que tem por elementos os x que satisfazem $P(x)$, tal conjunto será denotado por

$$\{x : P(x)\}.$$

Exemplo 3.17.1. $\{1, 2, 3, 4, 5\} = \{5, 4, 1, 2, 3\}$ e $\{5, 5, 5, 5\} = \{5\}$ porque os conjuntos envolvidos têm os mesmos elementos.

$\{1, 2, 3, 4, 5\} \neq \{5, 4, 2, 3\}$ e $\{5, 5, 5, 5\} \neq \{5, 6\}$ porque os conjuntos envolvidos são definidos por elementos diferentes.

Proposição 3.17.2. No domínio \mathcal{U} dos conjuntos temos a relação de igualdade de conjuntos satisfaz:

1. $\forall x(x = x)$ (reflexiva)
2. $\forall x \forall y(x = y \rightarrow y = x)$ (simétrica)

$$3. \forall x \forall y \forall z (x = y \wedge y = z \rightarrow x = z) \text{ (transitiva)}$$

Quando não se quer ser tão formal, escrevemos de forma equivalente;

Proposição 3.17.3. A relação de igualdade de conjuntos satisfaz, para todos os conjuntos A , B e C :

1. $A = A$ (reflexiva)
2. $A = B \Rightarrow B = A$ (simétrica)
3. $A = B \wedge B = C \Rightarrow A = C$ (transitiva)

Enfraquecendo a relação de igualdade, diremos que A é subconjunto de B se todo o elemento de A é um elemento de B , que podemos traduzir simbolicamente escrevendo:

$$A \subseteq B \Leftrightarrow \forall x (x \in A \rightarrow x \in B)$$

Exemplo 3.17.4. Se $A = \{\{1\}, 2, 3\}$, então $\{1\} \in A$, $\{\{1\}, 2\} \subseteq A$, $2 \in A$, $\{2, 3\} \subseteq A$.

Com esta definição pode enunciar-se que para conjuntos A e B :

$$\text{"}A \subseteq B \text{ e } B \subseteq A \text{ se e só se } A = B.\text{"}$$

Propriedade que podemos demonstrar, recorrendo às propriedades da lógica proposicional.

$$\begin{aligned} A \subseteq B \wedge B \subseteq A &\Leftrightarrow \forall x (x \in A \rightarrow x \in B) \wedge \forall x (x \in B \rightarrow x \in A) \\ &\Leftrightarrow \forall x ((x \in A \rightarrow x \in B) \wedge (x \in B \rightarrow x \in A)) \\ &\Leftrightarrow \forall x (x \in A \leftrightarrow x \in B) \\ &\Leftrightarrow A = B \end{aligned}$$

De forma idêntica pode-se demonstrar a veracidade de cada uma das afirmações apresentadas na proposição abaixo:

Proposição 3.17.5. No domínio \mathcal{U} dos conjuntos valem para a relação de inclusão:

1. $\forall x (x \subseteq x)$ (reflexiva)
2. $\forall x \forall y (x \subseteq y \wedge y \subseteq x \rightarrow y = x)$ (anti-simétrica)
3. $\forall x \forall y \forall z (x \subseteq y \wedge y \subseteq z \rightarrow x \subseteq z)$ (transitiva)

Propriedades que podemos reescrever através de:

Proposição 3.17.6. Para conjuntos A , B e C valem:

1. $A \subseteq A$ (reflexiva)
2. $(A \subseteq B \wedge B \subseteq A) \Rightarrow A = B$ (anti-simétrica)
3. $(A \subseteq B \wedge B \subseteq C) \Rightarrow A \subseteq C$ (transitiva)

3.18 Axioma dos conjuntos elementares

Assumimos que no universo dos conjuntos existe um conjunto que não contém qualquer elemento, que denotamos por \emptyset , e que designamos de **conjunto vazio**. Se a é um conjunto, existe um conjunto que contém apenas o elemento a , que representamos em extensão por $\{a\}$, e que designamos de **conjunto singular**. Se a e b são conjuntos existe um conjunto que contém a e b , e apenas a e b como elementos, que representamos em extensão por $\{a, b\}$, que designamos de **par não ordenado**.

3.19 Esquema da separação

Um predicado $P(x)$ diz-se colectivizante se define um conjunto. Se $P(x)$ é colectivizante, para cada conjunto A existe um conjunto cujos elementos são precisamente os elementos a de A para os quais $P(a)$ é verdadeira.

Isto é, para cada expressão proposicional colectivizante $P(x)$ e cada conjunto A , a fórmula

$$x \in A \wedge P(x),$$

define um conjunto. Assegura-se assim que existe um conjunto definido pela expressão:

$$\{x : x \in A \wedge P(x)\} \text{ ou } \{x \in A : P(x)\}$$

Assegurando-se assim a existência da intersecção de dois conjuntos A e B ,

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

Proposição 3.19.1. *Para conjuntos A , B e C valem*

1. $A \cap A = A$ (*idempotência*)
2. $A \cap \emptyset = \emptyset$
3. $A \cap B = B \cap A$ (*comutatividade*)
4. $(A \cap B) \cap C = A \cap (B \cap C)$ (*transitividade*)
5. $(C \subseteq A \wedge C \subseteq B) \Rightarrow C \subseteq A \cap B$
6. $A \subseteq B \Leftrightarrow A \cap B = A$

Definição: 3.19.2 (Conjuntos Disjuntos). *Dois conjuntos A e B são disjuntos se e só se $A \cap B = \emptyset$, ou seja se não têm elementos em comum.*

Exemplo 3.19.3. *Se $A_1 = \{\{1, 2\}, \{3\}\}$, $A_2 = \{\{1\}, \{2, 3\}\}$ e $A_3 = \{\{1, 2, 3\}\}$, os conjuntos A_1 , A_2 , A_3 são disjuntos dois a dois, já que $A_1 \cap A_2 = \emptyset$, $A_1 \cap A_3 = \emptyset$ e $A_2 \cap A_3 = \emptyset$.*

3.20 Axioma da união

Dado um conjunto F , existe um conjunto cujos elementos são os elementos dos elementos de F . Este conjunto chama-se **união de F** e denota-se por

$$\bigcup_{A \in F} A.$$

Exemplo 3.20.1. *Dado o conjunto*

$$F = \{\{a, b\}, \{a, b, c\}, \{d\}\},$$

pelo axioma da união existe o conjunto

$$\bigcup_{A \in F} A = \{a, b, c, d\}.$$

Isto garante a existência do conjunto

$$A \cup B = \{x : x \in A \vee x \in B\}$$

a união de A e B .

Exemplo 3.20.2. *Dado o conjunto*

$$F = \{\{a, b\}, \{a, b, c\}, \{d\}\},$$

podemos assim escrever

$$\bigcup_{A \in F} A = \{a, b\} \bigcup \{a, b, c\} \bigcup \{d\} = \{a, b, c, d\}.$$

Exemplo 3.20.3. *Para $S = \{a, b, c, d\}$ e $Q = \{c, d, g, h\}$, temos $S \cup Q = \{a, b, c, d, g, h\}$ e $S \cap Q = \{c, d\}$.*

Proposição 3.20.4. *Para conjuntos A, B e C temos:*

1. $A \cup A = A$ (*idempotênci*a)
2. $A \cup \emptyset = A$
3. $A \cup B = B \cup A$ (*comutatividade*)
4. $(A \cup B) \cup C = A \cup (B \cup C))$ (*transitividade*)
5. $(A \subseteq C \wedge B \subseteq C) \Rightarrow A \cup B \subseteq C$
6. $A \subseteq B \Leftrightarrow A \cup B = B$

Podemos ainda acrescentar as propriedades:

Proposição 3.20.5. *Para a união e intersecção de conjuntos temos:*

$$1. A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

$$2. A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

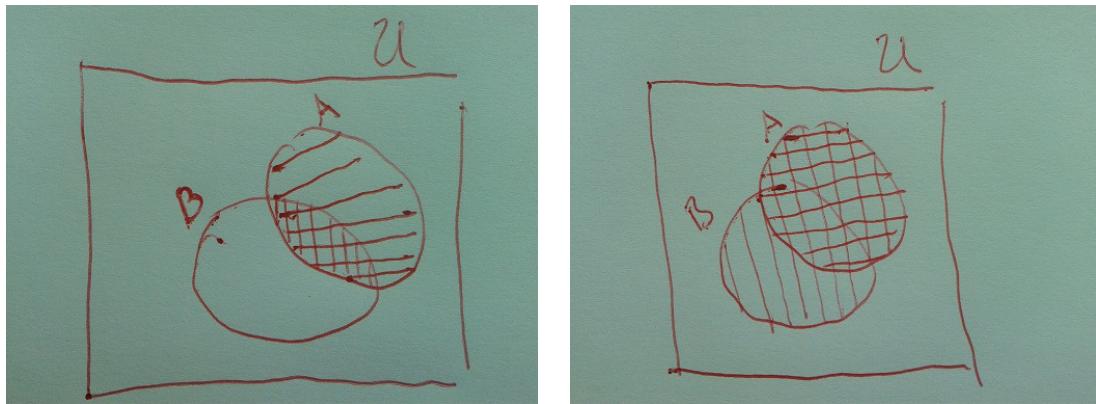
$$3. A \cup (A \cap B) = A$$

$$4. A \cap (A \cup B) = A$$

Proof. Apresentamos a demonstração de que $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

$$\begin{aligned} A \cup (B \cap C) &= \{x : x \in A \vee x \in B \cap C\} \\ &= \{x : x \in A \vee (x \in B \wedge x \in C)\} \\ &= \{x : (x \in A \vee x \in B) \wedge (x \in A \vee x \in C)\} \\ &= \{x : (x \in A \cup B) \wedge (x \in A \cup C)\} \\ &= (A \cup B) \cap (A \cup C) \end{aligned}$$

Para calcular $A \cup (A \cap B)$ e $A \cap (A \cup B)$ podemos usar diagramas de Venn:



Temos assim $A \cup (A \cap B) = A$ e $A \cap (A \cup B) = A$. □

A diferença entre dois conjuntos A e B ou o complementar de B em A , $A - B$ ou $A \setminus B$, está definida e é o conjunto

$$A \setminus B = \{x : x \in A \wedge x \notin B\}.$$

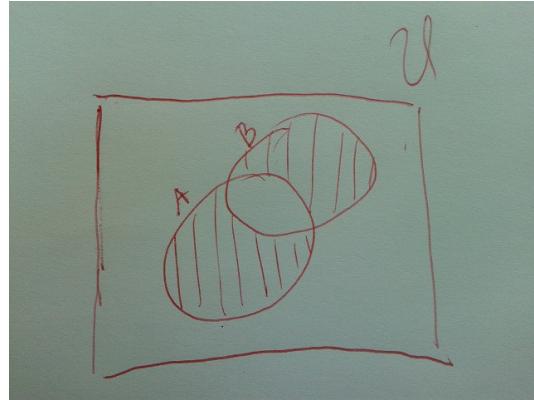
Exemplo 3.20.6. Dados $A = \{2, 5, 6\}$, $B = \{3, 4, 2\}$ e $C = \{1, 3, 4\}$, temos $A \setminus B = \{5, 6\}$, $B \setminus A = \{3, 4\}$, e $A \setminus C = \{2, 5, 6\}$.

Exercício 3.20.7. Mostre que para todo o par de conjuntos A e B ,

$$A \setminus (A \cap B) = A \setminus B$$

Entendemos por **diferença simétrica** ou **soma booleana** entre dois conjuntos A e B , ao conjunto

$$A \oplus B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B).$$



As seguintes igualdades são valias:

$$A \oplus B = B \oplus A, \quad A \oplus (B \oplus C) = (A \oplus B) \oplus C, \quad A \oplus \emptyset = A$$

Exercício 3.20.8. Mostre que para todo o conjunto A ,

$$A \oplus \emptyset = A$$

3.21 Axioma da potência

Dado um conjunto A , existe um conjunto cujos elementos são os subconjuntos de A . O conjunto cuja existência é decretada por este axioma, chama-se **conjunto potência** de A , ou **conjunto das partes** de A , que se denota por $\mathcal{P}(A)$, e representamos em compreensão por

$$\mathcal{P}(A) = \{x : x \subseteq A\}.$$

Exemplo 3.21.1. Temos assim:

1. $\mathcal{P}(\emptyset) = \{\emptyset\};$
2. $\mathcal{P}(\{\emptyset\}) = \{\emptyset, \{\emptyset\}\};$
3. $\mathcal{P}(\{\emptyset, \{\emptyset\}\}) = \{\emptyset, \{\emptyset\}, \{\{\emptyset\}\}, \{\emptyset, \{\emptyset\}\}\};$
4. $\mathcal{P}(\{0, 1\}) = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}.$
5. $\mathcal{P}(\{a, b, c\}) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{b, c\}, \{a, c\}, \{a, b, c\}\}$

3.22 Cardinal dum conjunto

Seja A um conjunto. Se o conjunto A tem n elementos, dizemos que A é um **conjunto finito** e que o **cardinal** de A é n . Neste caso escrevemos

$$\#A = n \text{ ou } |A| = n.$$

Um conjunto A diz-se **infinito** se não é finito.

Exemplo 3.22.1. Assim

1. $|\emptyset| = 0;$
2. $|\{\emptyset\}| = 1;$
3. $|\{\emptyset, \{\emptyset\}\}| = 2.$

Proposição 3.22.2 (Fórmula de Daniel da Silva). *Para conjuntos finitos A e B tem-se*

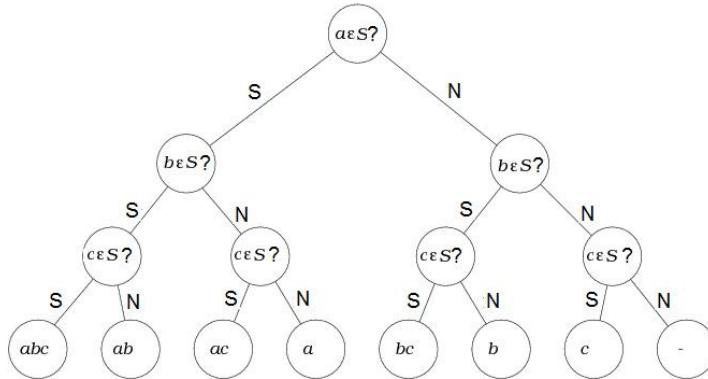
$$|A \cup B| = |A| + |B| - |A \cap B|.$$

Daniel Augusto da Silva (1814-1878)



Proposição 3.22.3. *Se A é um conjunto finito com n elementos, então $|\mathcal{P}(A)| = 2^n$.*

Procuremos ilustrar este resultado através da árvore de decisão representada pelo diagrama abaixo.



O diagrama tem a seguinte interpretação: Se seleccionar um subconjunto S de $\{a, b, c\}$. O círculo do topo, chamado nó, pode ser interpretado como sendo a pergunta: Está a em S ? Os dois arcos que saem dele têm como rótulos as duas respostas possíveis (Sim ou Não). Se seguirmos o arco com a resposta apropriada chegamos a um novo nó. Este nó codifica a próxima questão: O b é um elemento de S ? Seguindo o arco com a resposta certa chegamos a um novo nó. Que codifica a questão: O c é um elemento de S ? Escolhendo o arco com a resposta certa alcançamos a lista de elementos do conjunto S seleccionado.

Assim, a construção dum subconjunto de $\{a, b, c\}$ corresponde a um percurso no diagrama partido do topo até à base. Existem assim tantos subconjuntos quanto o número de folhas no último nível (a base). Como o número de nós duplica de nível para nível, existem $2^3 = 8$ nós no último nível (caso a selecção seja feita num conjunto com n elementos teríamos 2^n nós na árvore ou seja 2^n subconjuntos diferentes).

3.23 Par ordenado

Só com os axiomas enunciados pode arquitectar-se uma teoria de conjuntos satisfatória. Vamos exemplificar um pouco da teoria com mais algumas definições e teoremas. Dados dois conjuntos a e b , o axioma dos conjuntos elementares garante a existência do conjunto

$$\{\{a\}, \{a, b\}\},$$

a que se chama **par ordenado** de coordenadas a e b e que denotamos por

$$(a, b).$$

Com base nos axiomas e definições que apresentámos é fácil demonstrar que

$$(a, b) = (c, d) \Leftrightarrow a = c \wedge b = d.$$

Dados dois conjuntos A e B , existe um conjunto cujos elementos são precisamente todos os pares ordenados (a, b) , com $a \in A$ e $b \in B$, designado de *produto cartesiano* de A por B e que se denota por

$$A \times B,$$

i.e. define-se

$$A \times B = \{(a, b) : a \in A \wedge b \in B\}.$$

Adoptando-se as seguintes convenções $A^2 = A \times A$, $A^3 = A^2 \times A, \dots$, $A^n = A^{n-1} \times A$. Naturalmente, que se tem:

Proposição 3.23.1. Se A e B são conjuntos finitos com $|A| = n$ e $|B| = m$, então

$$|A \times B| = n \times m.$$

Exemplo 3.23.2. Se $A = \{\alpha, \beta\}$ e $B = \{1, 2, 3\}$ tem-se

1. $A \times B = \{(\alpha, 1), (\alpha, 2), (\alpha, 3), (\beta, 1), (\beta, 2), (\beta, 3)\}$
2. $B \times A = \{(1, \alpha), (2, \alpha), (3, \alpha), (1, \beta), (2, \beta), (3, \beta)\}$
3. $A \times A = \{(\alpha, \alpha), (\alpha, \beta), (\beta, \alpha), (\beta, \beta)\}$
4. $B \times B = \{(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)\}$

Para três conjuntos A, B, C temos

$$A \times (B \cup C) = (A \times B) \cup (A \times C)$$

$$A \times (B \cap C) = (A \times B) \cap (A \times C)$$

Exercício 3.23.3. Mostre que, para conjuntos A, B, C , $A \times (B \cup C) = (A \times B) \cup (A \times C)$

Definição: 3.23.4. Dois conjuntos A e B são disjuntos se e só se $A \cap B = \emptyset$. Um conjunto de conjuntos é definido por conjuntos disjuntos, se os conjuntos no conjunto são disjuntos dois a dois.

Exemplo 3.23.5. O conjunto de conjuntos $F = \{\{1, 3\}, \{2, 4, 5\}, \{0, 6, 7\}\}$, é definida por conjuntos disjuntos dois a dois, já que $\{1, 3\} \cap \{2, 4, 5\} = \emptyset$, $\{1, 3\} \cap \{0, 6, 7\} = \emptyset$ e $\{2, 4, 5\} \cap \{0, 6, 7\} = \emptyset$. Neste caso dizemos que F é uma partição $\{0, 1, 2, 3, 4, 5, 6, 7\}$

Definição: 3.23.6. A diferença entre dois conjuntos A e B (também designado o complementar de B em A), que se denota por $A - B$ ou $A \setminus B$, está definida e é o conjunto

$$A \setminus B = \{x : x \in A \wedge x \notin B\}.$$

Um conjunto E diz-se **universal** se contém todos os conjuntos em discussão. Neste sentido, para todo o conjunto A ,

$$A \subseteq E$$

ou dito de outra forma, para todo o predicado $P(x)$

$$E = \{x : P(x) \vee \sim P(x)\}.$$

Por outro lado o conjunto que não contém elementos é designado de conjunto vazio, ou seja, para todo o predicado $P(x)$ tem-se

$$\emptyset = \{x : P(x) \wedge \sim P(x)\}.$$

Neste sentido, para todo o conjunto A ,

$$\emptyset \subseteq A \text{ já que } \forall x(x \in \emptyset \rightarrow x \in A) \text{ é uma proposição verdadeira.}$$

Definição: 3.23.7. Seja E um conjunto universal. Para todo o conjunto A , o complemento de A em E , $E - A$ é designado de **complemento do conjunto** A e é denotado por \overline{A} . Neste sentido

$$\overline{A} = \{x \in E : \sim x \in A\} = \{x : x \notin A\}.$$

Proposição 3.23.8 (Leis de De Morgan). Para um universo valem as seguintes igualdades:

$$1. \overline{A \cup B} = \overline{A} \cap \overline{B}$$

$$2. \overline{A \cap B} = \overline{A} \cup \overline{B}$$

Proof. A igualdade $\overline{A \cup B} = \overline{A} \cap \overline{B}$ pode ser demonstrada através

$$\begin{aligned} \overline{A \cup B} &= \{x : \sim (x \in A \cup B)\} \\ &= \{x : \sim (x \in A \vee x \in B)\} \\ &= \{x : \sim (x \in A) \wedge \sim (x \in B)\} \\ &= \{x : x \notin A \wedge x \notin B\} \\ &= \{x : x \notin A\} \cap \{x : x \notin B\} \\ &= \overline{A} \cap \overline{B} \end{aligned}$$

□

Assim definido tem-se

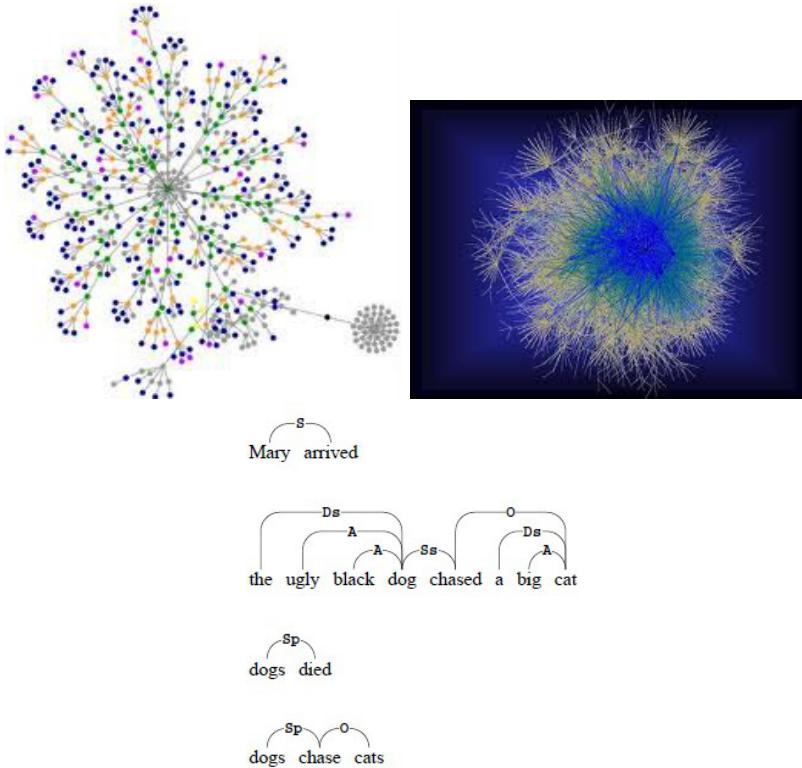
$$A \setminus B = A \cap \overline{B}.$$

Definição: 3.23.9. Entendemos por *diferença simétrica* ou *soma booleana* entre dois conjuntos A e B ao conjunto

$$A \oplus B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B).$$

3.24 Relação binária

Chama-se **relação binária** definida de A para B a qualquer subconjunto de $A \times B$. Diz-se **relação binária definida em A** a qualquer subconjunto de $A^2 = A \times A$.



Dada uma relação binária R

1. se $(a, b) \in R$ escrevemos aRb ;
2. se $(a, b) \notin R$ escrevemos $a\not Rb$;

Se R é uma relação, existe o conjunto, chamado **domínio** de R e denotado por $D(R)$, cujos elementos são os conjuntos x para os quais existe y tal que $(x, y) \in R$, i.e.

$$D(R) = \{x : \exists y(xRy)\}.$$

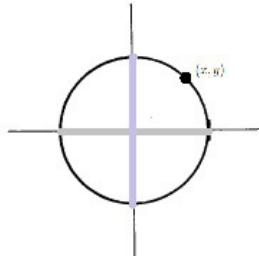
De modo análogo, existe o **conjunto imagem** de R dado por

$$Im(R) = \{y : \exists x(xRy)\}.$$

Exemplo 3.24.1. Qualquer gráfico no plano define uma relação em \mathbb{R} . Em particular o gráfico que descreve o conjunto dos pares de reais (x, y) solução da equação $x^2 + y^2 = 1$, define a relação

$$R = \{(x, y) : x^2 + y^2 = 1\}$$

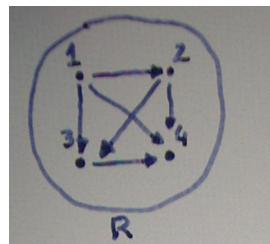
em \mathbb{R} . Neste caso $0R1$, $1R0$, $-1R0$, mas $1\not R1$. Temos ainda $D(R) = \{x : -1 \leq x \leq 1\}$ e $Im(R) = \{y : -1 \leq y \leq 1\}$.



Caso o conjunto A onde uma relação R está definida, não tenha muitos elementos, a relação binária R pode ser definida por:

1. Extensão enumerando os seus elementos;
2. Um diagrama sagital (grafo orientado ou digrafo) representado por um diagrama de Venn do conjunto e traçando setas de a para b para todos os pares aRb ;
3. Uma tabela de dupla entrada (ou matriz da relação) colocando V , ou um F , no cruzamento da linha a com a coluna b , respectivamente, se aRb ou $a\not Rb$.

Exemplo 3.24.2. A relação R definida no conjunto $A = \{1, 2, 3, 4\}$ em extensão por $R = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$ pode ser representada pelo diagrama sagital,



definida pela tabela de dupla entrada

R	1	2	3	4
1	F	V	V	V
2	F	F	V	V
3	F	F	F	V
4	F	F	F	F

ou pela matriz

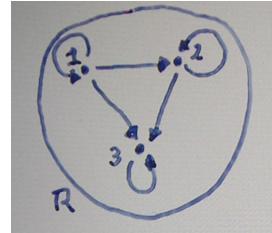
$$M_R = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Exemplo 3.24.3. Considere o conjunto $A = \{1, 2, 3\}$ a relação binária R , definida por

$$xRy \Leftrightarrow x \leq y, \forall a, b \in A$$

pode ser definida:

1. Por meio duma condição equivalente, por exemplo $\frac{y-x}{2} \geq 0$;
2. Por extensão, por $R = \{(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)\}$;
3. Por um diagrama sagital:



4. Por uma tabela de dupla entrada ou uma matriz de relação:

R	1	2	3
1	V	V	V
2	F	V	V
3	F	F	V

ou $M_R = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$

3.25 Relação complementar

A relação complementar da relação binária R em A , é a relação

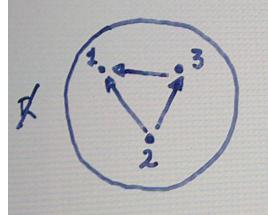
$$\mathcal{R} = A^2 \setminus R$$

que pode ser definida negando a condição que define R .

Exemplo 3.25.1. Seja R a relação binária em $A = \{1, 2, 3\}$ dada por

$$aRb \Leftrightarrow a \leq b, \forall a, b \in A.$$

Assim $a \mathcal{R} b \Leftrightarrow (a \leq b) \Leftrightarrow a > b, \forall a, b \in A$ ou seja $\mathcal{R} = \{(2,1), (3,1), (3,2)\}$ que tem por diagrama sagital

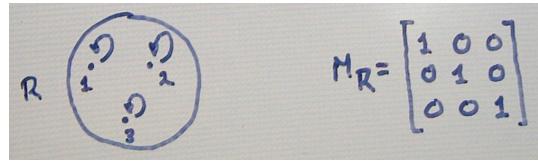


que podemos definir pela tabela

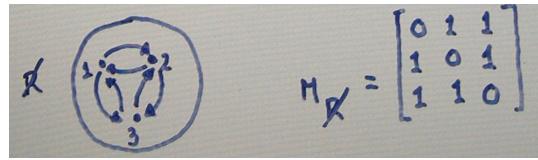
\mathcal{R}	1	2	3
1	V	V	V
2	F	V	V
3	F	F	V

ou $M_{\mathcal{R}} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$

Exemplo 3.25.2. No conjunto $A = \{1, 2, 3\}$ a relação dada por $aRb \Leftrightarrow a = b, \forall a, b \in A$ é representada por



a relação complementar é dada por $aRb \Leftrightarrow a \neq b, \forall a, b \in A$ e pode ser representada por



Vamos classificar uma relação binária R , definida num conjunto A , sob três pontos de vista:

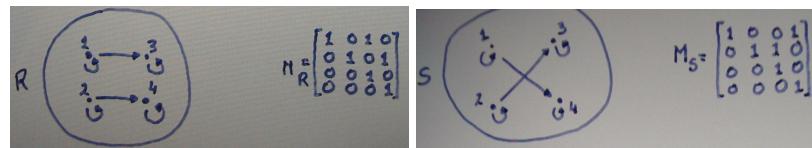
1. Reflexividade
2. Simetria
3. Transitividade

3.26 Reflexividade

Uma relação binária R definida no conjunto A diz-se:

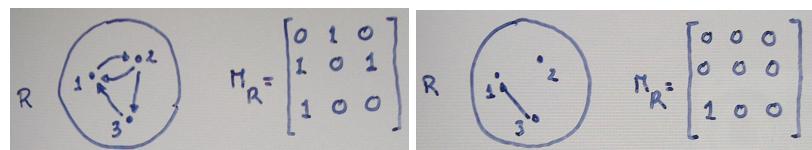
1. **Reflexiva** se: $\forall x \in A : xRx$.
2. **Não reflexiva** se: $\exists x \in A : xR \neq x$.
3. **Anti-reflexiva** se: $\forall x, y \in A : xRy \rightarrow x \neq y$.

Exemplo 3.26.1. Exemplos de relações reflexivas no conjunto $A = \{1, 2, 3, 4\}$



Note que, uma relação finita é reflexiva se no diagrama sagital cada ponto tem um laço, ou caso na matriz da relação todos os elementos da diagonal são 1.

Exemplo 3.26.2. Exemplos de relações anti-reflexivas no conjunto $A = \{1, 2, 3\}$



3.27 Simetria

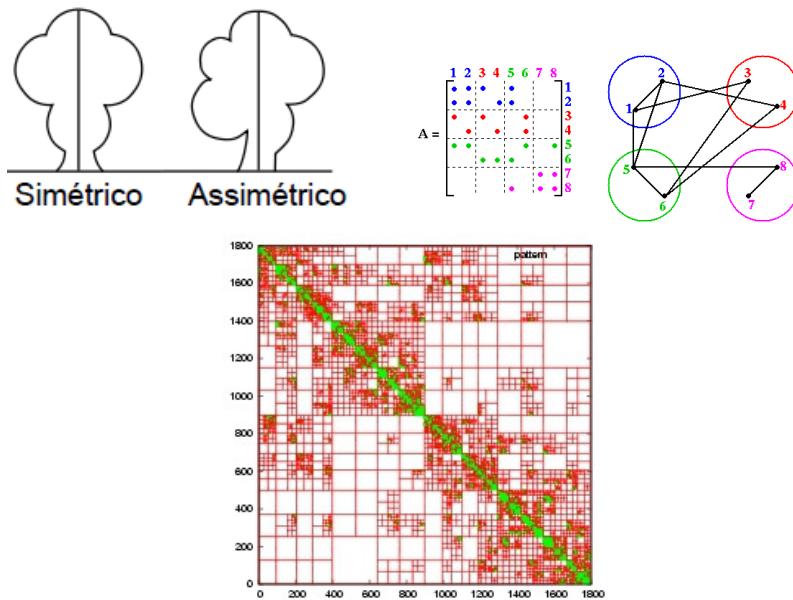
Uma relação binária R definida no conjunto A diz-se:

1. **Simétrica** se: $\forall x, y \in A : xRy \rightarrow yRx$.
2. **Não simétrica** se: $\exists x, y \in A : xRy \wedge y \not R x$.
3. **Anti-simétrica** se: $\forall x, y \in A : xRy \wedge yRx \rightarrow x = y$.

Exemplo 3.27.1. Exemplos de relações simétricas no conjunto $A = \{1, 2, 3, 4\}$

$$R \quad \text{Diagram showing a symmetric relation where every edge from } x \text{ to } y \text{ has a corresponding edge from } y \text{ to } x. \quad M_R = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Note que, no diagrama sagital duma relação simétrica, sempre que existe uma seta de x para y temos também uma seta de y para x . Uma matriz M_R representa uma relação simétrica se é simétrica.



Exemplo 3.27.2. Exemplos de relações anti-simétricas no conjunto $A = \{1, 2, 3\}$

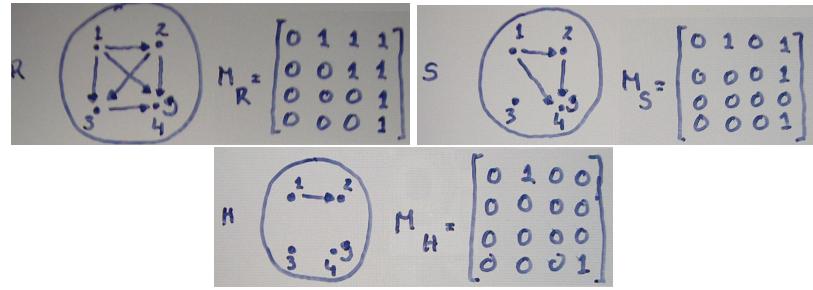
$$R \quad \text{Diagram showing an anti-symmetric relation where no edge from } x \text{ to } y \text{ exists if } y \neq x. \quad M_R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

3.28 Transitividade

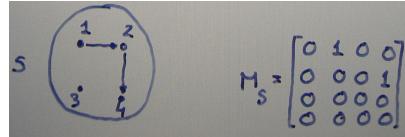
Uma relação binária R definida no conjunto A diz-se:

1. **Transitiva** se: $\forall x, y, z \in A : xRy \wedge yRz \rightarrow xRz$.
2. **Não transitiva** se: $\exists x, y, z \in A : xRy \wedge yRz \wedge x \not R z$.

Exemplo 3.28.1. Exemplos de relações transitivas no conjunto $A = \{1, 2, 3, 4\}$



Exemplo 3.28.2. Exemplo duma relação não transitiva no conjunto $A = \{1, 2, 3, 4\}$



3.29 Fecho

Dada uma relação binária R definida em A definimos:

1. O **fecho transitivo** da relação R é a relação R^+ de A dada por:

- (a) se aRb então aR^+b ,
- (b) se aR^+b e bR^+c então aR^+c .

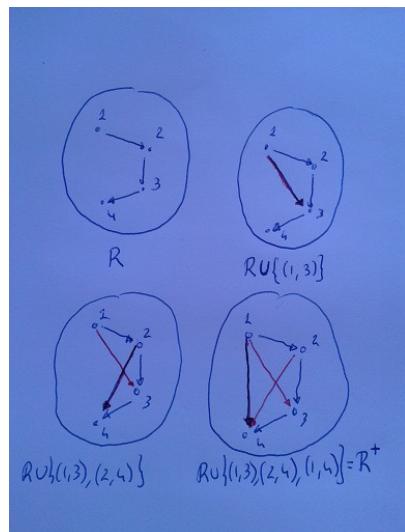
2. O **fecho reflexivo** da relação R é a relação R^- de A dada por:

$$R^- = R \cup \{(a, a) | a \in A\}.$$

3. O **fecho transitivo e reflexivo** da relação R é a relação R^* de A dada por:

$$R^* = R^+ \cup R^-.$$

Exemplo 3.29.1. Exemplificação do fecho transitivo duma relação $R = \{(1,2), (2,3), (3,4)\}$ em $A = \{1, 2, 3, 4\}$



Proposição 3.29.2. Uma relação binária R é transitiva se e só se $R = R^+$ i.e. R for igual ao seu fecho transitivo.

Exercício 3.29.3. Classifique em \mathbb{Z} as relações binárias definidas por:

1. $a \leq b$
2. $a < b$
3. $a = b$
4. $a + b = 1$
5. $ab > 0$
6. $ab \geq 0$

3.30 Python: Cláusulas if

O mecanismo que mais temos usado para controlo de fluxo da execução são cláusulas *if*. Por exemplo:

```
>>> x = int(input("Escreva um inteiro: "))
Escreva um inteiro: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
```

Entendemos neste Capítulo designmos a condição ou cláusula $x < 0$ por um predíaco. Já vimos que podem existir um ou mais blocos *elif*, e o bloco *else* é opcional. O comando *elif* é uma abreviação para “*else if*”, sendo útil para reduzir a quantidade de indentações. Uma sequência *if ... elif ... elif ...* é o substituto para os comandos *switch* ou *case* disponíveis noutras linguagens de programação.

3.31 Python: Comando for

Como vimos no Python o comando *for* permite iterar sobre objectos de qualquer sequência (uma lista ou uma string) ou um conjunto, nas sequências o ciclo *for* segue a ordem pela qual os objectos aparecem na sequência. Por exemplo:

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrarate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrarate 12
```

Caso tenha de modificar a sequência durante o ciclo *for* (por exemplo para duplicar elementos seleccionados), é conveniente fazer primeiro uma cópia. A noção de slice torna isso possível:

```
>>> for w in words[:]: # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrarate', 'cat', 'window', 'defenestrarate']
```

3.32 Python: A função range()

Quando temos de iterar numa sequência de números, a função built-in *range()* trata do assunto. Permitindo gerar progressões aritméticas:

```
>>> L=range(10)
>>> for i in L: print(i, ' ', end=' ')
0 1 2 3 4 5 6 7 8 9
```

O ponto final nunca é parte da lista gerada; *range(10)* gera uma sequência de 10 valores, os índices de uma lista com 10 objectos. É possível fazer o domínio ter inicio noutra número, ou indicar um incremento diferente (mesmo negativo; este incremento é usualmente designado de 'passo'):

```
>>> for i in range(5, 10): print(i, ' ', end=' ')
5 6 7 8 9
>>> for i in range(0, 10, 3): print(i, ' ', end=' ')
0 3 6 9
>>> for i in range(-10, -100, -30): print(i, ' ', end=' ')
-10 -40 -70
```

Para iterar nos índices de uma sequência, pode combinar *range()* com a função *len()* como por exemplo:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
```

```

...     print( i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb

```

Na maioria dos casos, é conveniente usar a função *enumerate()*.

3.33 Python:Comando break e continue, e cláusulas else nos ciclos

O comando *break*, permite encurtar os ciclos *for* ou *while*.

Os ciclos podem ter uma cláusula *else*; que é executado após ter percorrido todo o domínio do ciclo *for* ou quando a condição dum ciclo *while* se torna falsa, mas nunca quando o ciclo é interrompido com um comando *break*. Exemplificamos isto no ciclo seguinte, que tem por objectivo determinar números primos:

```

>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n/x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3

```

O comando *continue* pára a iteração corrente, saltando para a iteração seguinte do *loop*:

```

>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found a number", num)

```

```
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

3.34 Python: Mais de funções

Recorde que podemos criar uma função que descreva a série de Fibonacci com um limite variável:

```
>>> def fib(n):      # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, ' ', end=' ')
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

A palavra reservada *def* permite iniciar a descrição de uma função. Deve ser seguida do nome da função e da lista de parâmetros. O bloco que forma o corpo da função começa na linha seguinte, devendo estar identado.

A primeira instrução no corpo da função pode ser uma linha de documentação ou *docstring*. Existem ferramentas que usam as *docstrings* para produzir documentação; é uma boa prática introduzir *docstrings* no código que escreve, faça disso um hábito.

A execução de uma função cria uma nova tabela de símbolos usada para as variáveis locais da função. Mais precisamente, sempre que são atribuídos valores a variáveis na função, estes são armazenados na tabela de símbolos locais à função. Sempre que se usa uma variável o seu valor é primeiro procurado na tabelas de símbolos locais da função, depois nas tabelas de símbolos nas funções que a possam envolver, depois na tabela de símbolos globais, e por último na tabela de símbolos pré-definidos. Desta forma, as variáveis globais podem ser directamente acedidas de qualquer função mas o seu valor não pode ser alterado.

```
>>> def g():
def f():
n=11
return n+1
print(n, f(), ' ', end=' ')
```

```
>>> n=1
>>> g()
1 12
>>> n
1
```

Os parâmetros (argumentos) de chamada a uma função são introduzidas na tabela de símbolos locais da função quando é executada; sendo os argumentos passados como referência a um objecto. Quando uma função chama outra, uma nova tabela de símbolos é criada para essa chamada.

A definição duma função introduz o nome da função na tabela de símbolos corrente. O objecto referenciado, tem um tipo que é reconhecido pelo interpretador como função definida pelo utilizador. Sendo possível usar essa referência para outros nomes que chamam a mesma função. Servindo isto como um mecanismo genérico de renomeação:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Já que fib não devolve um valor, podemos argumentar que se deva chamar um procedimento. Na verdade, toda a função mesmo sem o comando return devolve um valor. As funções sem comando return devolvem um *None* (um nome pré-definido). Esta devolução é normalmente suprimida pelo interpretador, caso seja o único valor a escrever. Podendo no entanto ser visto usando um *print*:

```
>>> fib(0)
>>> print fib(0)
None
```

É tarefa simples escrever uma função que devolva uma lista de números da sucessão de Fibonacci, em vez de os imprimir:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)      # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)      # call it
```

```
>>> f100          # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Recorde que a instrução *return* devolve o valor de uma função. O comando *result.append(a)* chama um método associado a objectos de tipo lista. Um método é uma função que “pertencente” a um objecto, assumindo um nome de formato *obj.methodname*, onde *obj* é um objecto (podendo ser uma expressão), e *methodname* é o nome de um dos métodos associados aos objectos desse tipo. Diferentes objectos têm associado diferentes métodos. Métodos de diferentes tipos podem ter o mesmo nome sem provocar ambiguidades. O método *append()* do exemplo está definido para os objectos de tipo lista; permitindo adicionar um novo elemento no final da lista. Sendo equivalente a fazer *result = result + [a]*, mas mais eficiente.

É possível definir funções com um número variável de argumentos. Existindo três mecanismos que podem ser combinados.

3.34.1 Argumentos com valores por defeito

Muito útil é a definição de valores por defeito. Permitindo criar uma função que pode ser chamada com menos argumentos que os usados na definição da função. Por exemplo:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

Esta função pode ser chamada de diferentes formas:

1. dando apenas os argumentos obrigatórios: `ask_ok('Do you really want to quit?')`
2. dando um argumento opcional: `ask_ok('OK to overwrite the file?', 2)`
3. ou dando todos os argumentos: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Os valores por defeito são usados no ponto onde a função é definida, por exemplo

```
i = 5

def f(arg=i):
    print(arg)
```

```
i = 6
f()
```

imprime 5

Aviso importante: O valor por defeito é avaliado uma única vez. Com ressalva quando o valor por defeito é um objecto mutável, tal como uma lista, um conjunto ou um dicionário. Por exemplo, a função abaixo acumula os argumentos que lhe são passados, em chamadas seguintes. Por exemplo, a função seguinte acumula os argumentos passando-os às chamadas posteriores:

```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Imprimindo:

```
[1]
[1, 2]
[1, 2, 3]
```

Caso não pretenda que o valor por defeito seja partilhado entre chamadas seguintes, deve reescrever a chamada como se apresenta abaixo:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

3.35 Python: Listas

Os métodos mais usados do objecto list são:

1. *list.append(x)* - Acrescenta um elemento ao final duma lista: equivalente a $lista[len(lista):] = [x]$.
2. *list.extend(L)* - Estende a lista com todos os elementos da lista usada como argumento; equivalente a $lista[len(lista):] = L$.
3. *list.insert(i, x)* - Insere um elemento numa dada posição. O primeiro argumento é o índice anterior ao da inserção, assim *lista.insert(0, x)* insere no inicio da lista, e *lista.insert(len(lista), x)* é equivalente a *a.append(x)*.

4. `list.remove(x)` - remove o primeiro elemento da lista igual a `x`. Devolve um erro se tal elemento não existe.
5. `list.pop([i])` - remove o elemento numa dada posição da lista, e devolve-o. Caso nenhum índice for dado, remove e devolve o último elemento da lista. (Os parêntesis rectos que rodeiam o `i` no método indicam que o parâmetro é opcional, o índice que usa por argumento não pode usá-los)
6. `list.index(x)` - devolve o índice da primeira ocorrência de `x` na lista. Devolve um erro caso não exista.
7. `list.count(x)` - devolve o número de vezes em que `x` ocorre na lista.
8. `list.sort()` - ordena os elementos da lista.
9. `list.reverse()` - reverte a ordenação dos elementos.

Um exemplo:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

3.35.1 Uso da lista como uma pilha

Os métodos apresentados permitem usar uma lista como uma pilha, onde o último elemento a entrar na lista é o primeiro a sair (last-in, first-out). Para adicionar um elemento ao topo da pilha, usa-se `append()`. Para retirar um elemento do topo da pilha, usa-se o `pop()` sem índice explícito. Por exemplo:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
```

```
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

3.35.2 Python: Usando listas como filas

Podemos também usar as listas como filas, onde o primeiro elemento a entrar é o primeiro a sair (first-in, first-out); No entanto as listas não são muito eficientes com este propósito. Enquanto os *appends* e *pops* no fim da lista são rápidos, fazer *inserts* ou *pops* no inicio da lista é lento (já que todos os outros elementos têm de ser deslocados uma posição).

Para implementar uma fila, use *collections.deque*, desenvolvida para melhorar a eficiência neste tipo de estrutura. Por exemplo:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()              # The first to arrive now leaves
'Eric'
>>> queue.popleft()              # The second to arrive now leaves
'John'
>>> queue                      # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

3.35.3 Python: Listas em Compreensão

As listas definidas em compreensão oferecem uma forma concisa de definir uma lista. Uma aplicação é a construção de listas por recorrência, ou através da selecção de elementos numa sucessão definida por recorrência.

Por exemplo, assumindo que pretendemos criar uma lista de números quadrados, por exemplo:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
... 
```

```
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Podemos ter o mesmo resultado com:

```
squares = [x**2 for x in range(10)]
```

Uma lista definida em compreensão consiste numa expressão, seguida por zero ou mais cláusulas, definidas entre parêntesis. O resultado é uma nova lista resultando da avaliação da expressão no domínio do *for* satisfazendo as cláusulas. Por exemplo, a lista do exemplo abaixo combina os elementos de duas listas caso não sejam iguais:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

O mesmo resultado pode ser produzido por:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note que a ordem dos *for* e *if* na instrução é a mesma nas duas versões.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit  ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
```

```

File "<stdin>", line 1
    [x, x**2 for x in range(6)]
    ^
SyntaxError: invalid syntax

>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

```

As listas em compreensão podem conter expressões complexas e chamadas a funções:

```

>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']

```

3.35.4 Python: Matrizes como listas

A expressão inicial numa lista em compreensão pode ser qualquer expressão arbitrária, podendo ser mesmo outra lista em compreensão.

Recorde o exemplo de uma matriz de 3x4 descrita por 3 listas de comprimento 4:

```

>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]

```

Pode ser transposta através de:

```

>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

```

Sendo isto equivalente a fazer:

```

>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

```

Que por sua vez é equivalente a:

```

>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:

```

```

...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]

```

Exercício 3.35.1. Defina uma função $nul(int) \rightarrow list$ que devolva uma matriz quadrada $A = [a_{ij}]$ de ordem n de elementos todos nulos, isto é $\forall i, j \in \{0, \dots, n-1\} : a_{i,j} = 0$.

Exercício 3.35.2. Defina uma função $Id(list) \rightarrow list$ que devolva uma matriz identidade $I = [a_{ij}]$, isto é $\forall i, j \in \{0, \dots, n-1\} : (i \neq j \rightarrow a_{i,j} = 0)$ e $\forall i \in \{0, \dots, n-1\} : a_{i,i} = 1$.

Exercício 3.35.3. Defina uma função $Lagrange(int) \rightarrow list$, tal que $Lagrange(n)$ é uma matriz quadrada de ordem n tal que $\forall i, j \in \{0, \dots, n-1\} : a_{i,j} = i + j + 2$.

3.36 Python: Tuples

Vimos que as listas e as strings têm muitas propriedades em comum, tais como indexadores e operações de slicing. São exemplos de dados do tipo sequências (são deste tipo *str*, *unicode*, *list*, *tuple*, *bytearray*, *buffer*, *range*). Outra estrutura de dados que pode ser vista como uma sequência são os tuplos.

Um tuplo consiste numa sequência de valores separados por vírgulas, por exemplo:

```

>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])

```

Dos exemplos, o output dum tuplo está sempre envolto em parênteses; podendo ser definido com ou sem parênteses, sendo obrigatórios se o tuplo faz parte duma expressão complexa. Não é possível alterar um componente dum tuplo, no entanto é possível criar tuplos que contêm objectos mutáveis, tais como listas.

Apesar dos tuplos serem semelhantes a listas, eles são usados em situações diferentes e com propósitos diferentes. Os tuplos são imutáveis, e normalmente contêm uma sequência heterogênea de elementos.

Exemplos:

```
>>> empty = ()
>>> singleton = 'hello',      # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

O comando `t = 12345, 54321, 'hello'` é exemplo de um tuplo empacotando: os valores 12345, 54321 and 'hello!' no tuplo.

Exercício 3.36.1. Defina `Rec(int) -> tuple` uma função que devolve um tuplo dos primeiros m termos da sucessão dada por $a_0 = 1, a_1 = 2$, e $a_{n+1} = a_n - a_{n-1}$, para $n > 1$.

Exercício 3.36.2. Determine a lista de tuplos (x, y, z) , definidos por números naturais menores que 100, tais que $x^2 + y^2 = z^2$.

Exercício 3.36.3. Determine a lista de tuplos (x, y, z) , definidos por números inteiros x, y, z , maiores que -100 e menores que 100, tais que

$$x^2 - y^2 \leq 1 \wedge y^2 - z^2 \leq 1 \wedge x^2 - z^2 > 1.$$

3.37 Python: Conjuntos

Um conjunto A de elementos inteiros 2, 3, 4, 5 pode ser definido por:

```
>>> A={2, 3, 4, 5}
```

Ao conjunto A podem ser adicionados mais elementos através do método `add`. Por exemplo

```
>>> A.add(7)
```

adiciona a A o inteiro 7. Inversamente podem ser removidos elementos a um conjunto através do método `pop`.

```
>>> A.pop()
```

2

Neste caso, o método seleccionou um elemento de A , o inteiro 2, que removeu a A . Após esta sequência de comandos tem-se

```
>>> A
{3, 4, 5, 7}
>>> len(A)
4
```

Neste sentido A tem 4 elementos, $7 \in A$, é uma proposição verdadeira e $2 \in A$, é uma proposição falsa. Em Python:

```
>>> 7 in A
True
>>> 2 in A
False
```

Note que $R = \{(3, 3), (4, 5), (7, 7)\}$ é uma relação binária em A , que não é reflexiva porque $(4, 4) \in R$ é falso. Em Python:

```
>>> R={(3, 3), (4, 5), (7, 7)}
>>> (4, 4) in R
False
```

O domínio da relação R , definido como o conjunto

$$\text{dom}(R) = \{x \in A : \exists y \in A, (x, y) \in R\},$$

pode ser calculado em Python por:

```
>>> def dom(R):
    D=set()           # D é o conjunto vazio
    for (x,y) in R:
        D.add(x)      # junta a D um elemento x, a primeira
                        # componente do par (x,y) em R
    return D
>>> dom(R)
{3, 4, 7}
```

Recorde que $\text{set}([])$ representa \emptyset o conjunto vazio.

As expressões $\text{for } <\text{var}> \text{ in } <\text{obj}>$ podem ser usadas para construir conjuntos ou listas. Por exemplo:

```
>>> R={(x,x+1) for x in range(3)}
>>> R
{(0, 1), (1, 2), (2, 3)}
>>> M=[[0 for i in range(3)] for j in range(4)]
>>> M
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Podemos entender no último caso M como uma matriz booleana, onde por $M[i][j]$ identificamos o elemento da linha i e coluna j , assumindo para isso que a primeira linha e a primeira coluna têm índice 0. Podemos alterar as entradas da matriz usando estes índices.

```
>>> M[1][2]= 1 # altera para 1 a entrada da linha 1 e coluna 2 da matriz M
>>> M[0][1]= 1 # altera para 1 a entrada da linha 0 e coluna 1 da matriz M
>>> M
[[0,1,0],[0,0,1],[0,0,0],[0,0,0]]
```

Outro exemplo:

```
>>> T={x for x in range(100) if x%2==0 and x%3!=0}
>>> T
{2, 4, 8, 10, 14, 16, 20, 22, 26, 28, 32, 34, 38, 40, 44, 46, 50, 52, 56,
58, 62, 64, 68, 70, 74, 76, 80, 82, 86, 88, 92, 94, 98}
```

Aqui podemos entender T descrito, em “linguagem matemática”, em compreensão por $\{x \in \text{range}(100) : x\%2 = 0 \wedge x\%3 \neq 0\}$.

A função `set()` pode ser usada para construir um conjunto. Nota: o conjunto vazio é criado usando `set()` ou `set([])`, e não `{}`; o último cria o dicionário vazio (estrutura de dados que se apresenta na sequência).

Um caso de uso:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)                                     # create a set without duplicates
>>> fruit
{'orange', 'pear', 'apple', 'banana'}
>>> 'orange' in fruit                                    # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                         # Letras usadas na string a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                                     # Letras que estão
                                                # em a e não em b (diferença)
{'r', 'd', 'b'}
>>> a | b                                     # Letras que estão
                                                # em a ou em b (união)
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                     # Letras que estão
                                                # em a e b (intersecção)
{'a', 'c'}
>>> a ^ b                                     # Letras que estão em a ou b
                                                # mas não nas duas (XOR)
{'r', 'd', 'b', 'm', 'z', 'l'}
```

De forma semelhante à definição de listas em compreensão, podemos definir conjuntos em compreensão:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

Dadas relações R , de A em B , e S uma relação de B em C , define-se a SoR (composição de S e R) como a relação de A para C dada por:

```
def SoR(S, R):
    H=set()
    for (a,b) in R:
        for (d,c) in S:
            if d==b:
                H.add((a,c))
    return H
```

ou de forma equivalente:

```
{(a,c) for (a,b) in R for (d,c) in S if b==d}
```

O fecho transitivo assume a forma da função:

```
def fecho(R):
    PR=set(R)          # cópia de R
    H=set(R)          # cópia de R
    Hold=set()         # conjunto vazio
    while H!=Hold:
        Hold=set(H)
        PR=SoR(PR,R) # chama a função anterior
        H=Hold | PR   # união de dois conjuntos
    return H
```

Exercício 3.37.1. Crie uma função booleana $sim(set, set) \rightarrow \text{bool}$, usando um ciclo for, que devolve True se o primeiro conjunto está contido no segundo.

Exercício 3.37.2. Defina $Prim(int) \rightarrow set$ uma função que devolve o conjunto dos primeiros m números primos.

Exercício 3.37.3. Determine o conjunto dos tuplos (x, y) , definidos por números naturais menores que 100, tais que $x^2 + y^2 < 5^2$.

Exercício 3.37.4. Determine o conjunto dos tuplos (x, y, z) , definidos por números inteiros x, y, z , maiores que -100 e menores que 100, tais que

$$x^2 + y^2 \leq 1 \wedge y^2 + z^2 \leq 1 \wedge x^2 + z^2 > 1.$$

Exercício 3.37.5. Crie uma função $Val(set, int) \rightarrow set$ que quando aplicada a uma relação R definida em $\text{range}(n)$, e a um inteiro x devolve o conjunto $\{a : xRa\}$.

Exercício 3.37.6. Crie uma função $InVal(set, int) -> set$ que quando aplicada a uma relação R definida em $range(n)$, e a um inteiro x , devolve o conjunto $\{a : aRx\}$.

Exercício 3.37.7. Crie uma função $InVal(set, set) -> set$ que quando aplicada a uma relação R definida em $range(n)$ e para $A \subseteq range(n)$ devolve o conjunto $\{a : aRx \wedge x \in A\}$.

Exercício 3.37.8. Implemente funções, que tendo por argumento uma relação binária R num conjunto A , devolvem o conjunto indicado:

1. $Im(set) -> set$, que para a relação R , $Im(R) = \{y : (x, y) \in R\}$
2. $fim(set) -> set$, que para a relação R , $fim(R) = \{z : \exists x, y, (x, y) \in R \wedge (y, z) \in R\}$
3. $meio(set) -> set$, que para a relação R , $meio(R) = \{y : \exists x, z, (x, y) \in R \wedge (y, z) \in R\}$
4. $inicio(set) -> set$, que para a relação R , $inicio(R) = \{x : \exists y, z, (x, y) \in R \wedge (y, z) \in R\}$
5. $unico(set) -> set$, que para a relação R , $unico(R) = \{x : \exists! y, (x, y) \in R\}$ ou seja elementos x a partir dos quais sai um único arco no diagrama sagital de R .

3.38 EXERCÍCIOS DE REVISÃO

Exercício 3.38.1. Escrever as frases que se seguem usando notação lógica na qual x designa um gato e $P(x)$ significa “ x gosta de bolo”

1. Todos os gatos gostam de bolo.
2. Nenhum gato gosta de bolo.
3. Um gato gosta de bolo.
4. Alguns gatos não gostam de bolo.

Exercício 3.38.2. Sendo A, B, C três conjuntos, analise em termos lógicos, usando quantificadores, a proposição “se $A \subseteq B$ então A e $C \setminus B$ são disjuntos”.

Exercício 3.38.3. Traduzir em linguagem simbólica as proposições que se seguem, indicando as escolhas que são apropriadas para os domínios correspondentes.

1. Existe um inteiro x tal que $2 = x + 1$.
2. Para todos os inteiros x , $2 = x + 1$.
3. Todos os que entendem Lógica gostam dela.
4. $x^2 - 4 = 0$ tem uma raiz positiva.
5. Toda a solução da equação $x^2 - 4 = 0$ é positiva.
6. Nenhuma solução da equação $x^2 - 4 = 0$ é positiva.

Exercício 3.38.4. Indique o significado das proposições que se seguem, sendo a quantificação feita sobre \mathbb{N}

- $\forall x \exists y (x < y)$
- $\exists y \forall x (x < y)$
- $\exists x \forall y (x < y)$
- $\forall y \exists x (x < y)$
- $\exists x \exists y (x < y)$
- $\forall x \forall y (x < y)$

Indique qual o valor lógico de cada uma delas.

Exercício 3.38.5. Sendo \mathbb{R} o universo do discurso escreva em linguagem simbólica as seguintes afirmações:

- A identidade da adição é o 0.
- Todo o número real tem simétrico.
- Os números negativos não têm raízes quadradas.
- Todo o número positivo possui exactamente duas raízes quadradas.

Exercício 3.38.6. Seja $\mathbb{N} = \{1, 2, 3, 4, \dots\}$, $P(x)$ a afirmação "x é par", $Q(x)$ a afirmação "x é divisível por 3" e $R(x)$ a afirmação "x é divisível por 4". Expressar em linguagem corrente cada uma das proposições que se seguem e determinar o seu valor lógico.

1. $\forall x \in \mathbb{N} : P(x) \wedge Q(x)$
2. $\exists x \in \mathbb{N} : R(x)$
3. $\exists x \in \mathbb{N} : P(x) \wedge Q(x)$
4. $\exists x \in \mathbb{N} : P(x) \rightarrow Q(x)$
5. $\exists x \in \mathbb{N} : Q(x) \wedge Q(x + 1)$

Exercício 3.38.7. Indicar se as proposições são sempre, às vezes ou nunca verdadeiras.

1. $(\exists x \in D : \neg P(x)) \Rightarrow \neg(\exists x \in D : P(x))$
2. $\neg(\forall x \in D : \neg P(x)) \Rightarrow (\forall x \in D : \neg P(x))$
3. $\neg(\exists x \in D : \neg P(x)) \Rightarrow (\exists x \in D : \neg P(x))$

Exercício 3.38.8. Dados $S = \{2, a, 3, 4\}$ e $R = \{a, 3, 4, 1\}$, indique se são verdadeiras ou falsas as proposições:

1. $\{a\} \in S$
2. $\{a\} \in R$

3. $\{a, 4, \{3\}\} \subseteq S$

4. $\{\{a\}, 1, 3, 4\} \subset R$

5. $R = S$

6. $\{a\} \subseteq S$

7. $\{a\} \subseteq R$

8. $\emptyset \subset R$

9. $\emptyset \subset \{\{a\}\} \subseteq R$

10. $\{\emptyset\} \subseteq S$

11. $\emptyset \in R$

12. $\emptyset \subseteq \{\{3\}, 4\}$

Exercício 3.38.9. Mostre que

$$(R \subseteq S) \wedge (S \subseteq Q) \Rightarrow R \subseteq Q$$

Exercício 3.38.10. Determine o conjunto potência de:

1. $\{a, \{b\}\}$

2. $\{1, \emptyset\}$

3. $\{X, Y, Z\}$

Exercício 3.38.11. Dados conjuntos $A = \{x | x \text{ é um inteiro e } 1 \leq x \leq 5\}$, $B = \{3, 4, 5, 17\}$, e $C = \{1, 2, 3, \dots\}$, determine $A \cap B$, $A \cap C$, $A \cup B$, e $A \cup C$.

Exercício 3.38.12. Mostre que $A \subseteq A \cup B$ e $A \cap B \subseteq A$.

Exercício 3.38.13. Mostre que

$$A \subseteq B \Leftrightarrow A \cup B = B.$$

Exercício 3.38.14. Dado $A = \{2, 3, 4\}$, $B = \{1, 2\}$, e $C = \{4, 5, 6\}$, determine $A \oplus B$, $B \oplus C$, $A \oplus B \oplus C$, e $(A \oplus B) \oplus (B \oplus C)$.

Exercício 3.38.15. Se $S = \{a, b, c\}$, determine conjuntos disjuntos A_1 e A_2 , tais que $A_1 \cup A_2 = S$. Apresente outra solução, diferente da anterior.

Exercício 3.38.16. Determine:

1. $\emptyset \cap \{\emptyset\}$

2. $\{\emptyset\} \cap \{\emptyset\}$

3. $\{\emptyset, \{\emptyset\}\} - \emptyset$

Exercício 3.38.17. Liste os elementos de $\{a, b\} \times \{1, 2, 3\}$.

Exercício 3.38.18. Liste os elementos de $A \times B \times C$, B^2 , A^3 , $B^2 \times A$, e $A \times B$, onde $A = \{1\}$, $B = \{a, b\}$, e $C = \{2, 3\}$.

Exercício 3.38.19. Use exemplos para mostrar que

$$A \times B \neq B \times A \text{ e } (A \times B) \times C \neq B \times A \times (B \times C).$$

Exercício 3.38.20. Mostre que para conjuntos A e B , se tem

$$\mathcal{P}(A) \cup \mathcal{P}(B) \subseteq \mathcal{P}(A \cup B)$$

$$\mathcal{P}(A) \cap \mathcal{P}(B) \subseteq \mathcal{P}(A \cap B)$$

Usando um exemplo, mostre que

$$\mathcal{P}(A) \cup \mathcal{P}(B) \neq \mathcal{P}(A \cup B).$$

Exercício 3.38.21. Mostre que $A \times (B \cap C) = (A \times B) \cap (A \times C)$.

Exercício 3.38.22. Mostre que

$$A \times B = B \times A \Leftrightarrow (A = \emptyset) \vee (B = \emptyset) \vee (A = B).$$

Exercício 3.38.23. Mostre que $(A \cap B) \cup C = A \cap (B \cup C)$ se e só se $C \subseteq A$.

Exercício 3.38.24. Seja $P = \{(1, 2), (2, 4), (3, 3)\}$ e $Q = \{(1, 3), (2, 4), (4, 2)\}$.

1. Determine $P \cup Q$, $P \cap Q$, $D(P)$, $Im(Q)$, $D(P \cup Q)$, $Im(P)$, e $Im(P \cap Q)$.
2. Mostre que $D(P \cup Q) = D(P) \cup D(Q)$ e $Im(P \cap Q) \subseteq Im(P) \cap Im(Q)$.

Exercício 3.38.25. Seja L a relação “menor ou igual que” e D a relação “divide”, onde $x \text{ Dy}$ significa “ x divide y ”. Assumindo que L e D estão definidos no conjunto $\{1, 2, 3, 6\}$, escreva L e D na forma dum conjunto, e determine $L \cap D$.

Exercício 3.38.26. Apresente um exemplo de uma relação que não seja nem reflexiva nem anti-reflexiva.

Exercício 3.38.27. Apresente um exemplo de uma relação que seja simétrica e anti-simétrica.

Exercício 3.38.28. Se as relações R e S são ambas reflexivas, simétricas e transitivas, mostre que $R \cap S$ e $R \cup S$ também são reflexivas.

Exercício 3.38.29. Se a relação R e S são reflexivas, simétricas e transitivas, mostre que $R \cap S$ também é reflexiva, simétrica e transitiva.

Exercício 3.38.30. Mostre que as relações abaixo são transitivas:

1. $R_1 = \{(1, 1)\}$
2. $R_2 = \{(1, 2), (2, 2)\}$

$$3. R_3 = \{(1, 2), (2, 3), (1, 3), (2, 1)\}$$

Exercício 3.38.31. Dado $S = \{1, 2, 3, 4\}$ e uma relação R e S definida por

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

mostre que R é não transitiva. Determine uma relação R_1 , tal que $R \subseteq R_1$ e seja transitiva. Existe mais alguma relação R_2 que contenha R_1 e também seja transitiva?

Exercício 3.38.32. Classifique as relações binárias, em \mathbb{Z} , definidas abaixo quanto à reflexividade, simetria, anti-simetria, transitividade:

$$1. R_1 = \{(a, b) : a^2 - b^2 \leq 1\}$$

$$2. R_2 = \{(a, b) : a - b = 1\}$$

$$3. R_3 = \{(a, b) : ab \text{ é par}\}$$

$$4. R_4 = \{(a, b) : a + b \text{ é par}\}$$

$$5. R_5 = \{(a, b) : a \% 3 = b \% 3\}, \text{ onde } x \% y \text{ é o resto da divisão inteira de } x \text{ por } y$$

$$6. R_6 = \{(a, b) : a \% 2 = b \% 2\}$$

$$7. R_7 = \{(a, b) : ab \% 4 \text{ é par}\}$$

$$8. R_8 = \{(a, b) : b \% a = 0\}$$

Exercício 3.38.33. Classifique as relações binárias, no conjunto das strings, definidas abaixo quanto à reflexividade, simetria, anti-simetria, transitividade:

$$1. R_1 = \{(a, b) : a \text{ e } b \text{ têm o mesmo número de símbolos}\}$$

$$2. R_2 = \{(a, b) : a \text{ é simétrica a } b\}$$

$$3. R_3 = \{(a, b) : ab \text{ têm um número par de símbolos}\}$$

Exercício 3.38.34. Calcule o fecho transitivo das seguintes relações:

$$1. R = \{(1, 2), (2, 4), (4, 3)\} \text{ no conjunto } \{1, 2, 3, 4\}$$

$$2. Q = \{(a, a), (b, d), (d, a)\} \text{ no conjunto } \{a, b, c, d\}$$

$$3. V = \{(b, a), (b, d), (d, a)\} \text{ no conjunto } \{a, b, c, d\}$$

Exercício 3.38.35. Implemente funções, que tendo por argumentos um conjunto A e uma relação binária R , no conjunto A , devolvem o valor lógico da proposição:

$$1. f1(set, set) -> \text{bool} \text{ tal que } f1(A, R) \text{ é True se e só se } \forall x \in A \exists y \in A : (x, y) \in R$$

$$2. f2(set, set) -> \text{bool} \text{ tal que } f2(A, R) \text{ é True se e só se } \exists x \in A \forall y \in A : (x, y) \in R$$

$$3. f3(set, set) -> \text{bool} \text{ tal que } f3(A, R) \text{ é True se e só se } \forall x \in A \exists !y \in A : (x, y) \in R$$

$$4. f4(set, set) -> \text{bool} \text{ tal que } f4(A, R) \text{ é True se e só se } \forall x \in A \forall y \in A : (x, y) \in R \wedge (y, x) \in R$$

Exercício 3.38.36. Implemente funções, que tendo por argumento uma relação binária R num conjunto A , devolvem o valor lógica da proposicional:

1. $g1(set) -> \text{bool}$ tal que $g1(R)$ é True se e só se $\forall x \in A \forall y \in A \exists z \in A : (x, y) \in R \rightarrow (y, z) \in R$
2. $g2(set) -> \text{bool}$ tal que $g2(R)$ é True se e só se $\forall x \in A \exists y \in A \exists z \in A : (x, y) \in R \rightarrow (y, z) \in R$
3. $g3(set) -> \text{bool}$ tal que $g3(R)$ é True se e só se $\forall x \in A \exists !y \in A \exists z \in A : (x, y) \in R \rightarrow (y, z) \in R$
4. $g4(set) -> \text{bool}$ tal que $g4(R)$ é True se e só se $\forall x \in A \exists y \in A \exists !z \in A : (x, y) \in R \rightarrow (y, z) \in R$

Exercício 3.38.37. Implemente funções que tendo por argumentos relações binárias R e S num conjunto A , devolvem o valor lógica da proposicional:

1. $h1(set, set) -> \text{bool}$ tal que $h1(R, S)$ é True se e só se $\forall x \in A \forall y \in A \forall z \in A : (x, y) \in R \wedge (y, z) \in S \rightarrow (y, z) \in R$
2. $h2(set, set) -> \text{bool}$ tal que $h2(R, S)$ é True se e só se $\forall x \in A \forall y \in A \forall z \in A : ((x, y) \in R \wedge (x, z) \in S) \rightarrow (y, z) \in R$
3. $h3(set, set) -> \text{bool}$ tal que $h3(R, S)$ é True se e só se $\forall x \in A \forall y \in A \forall z \in A : ((x, y) \in R \vee (y, z) \in R) \rightarrow ((x, z) \in S \wedge (z, x) \in R)$
4. $h4(set, set) -> \text{bool}$ tal que $h4(R, S)$ é True se e só se $\forall x \in A \forall y \in A \forall z \in A : ((x, y) \in S \wedge (y, z) \in R) \rightarrow ((x, z) \in S \vee (z, x) \in R)$

Exercício 3.38.38. Implemente funções, que tendo por argumentos duas relações binárias R e S num conjunto A , devolvem o conjunto indicado:

1. $s1(set, set) -> set$, que para relações R e S , $s1(R, S) = \{y : \exists x, (x, y) \in R \wedge (y, x) \notin S\}$
2. $s2(set, set) -> set$, que para relações R e S , $s2(R, S) = \{z : \exists x, y, ((x, y) \in R \wedge (y, z) \in R) \wedge (x, z) \notin S\}$
3. $s3(set, set) -> set$, que para relações R e S , $s3(R, S) = \{y : \exists x, z, (x, y) \in S \wedge (y, z) \in R \wedge (x, z) \notin S\}$
4. $s4(set, set) -> set$, que para relações R e S , $s4(R, S) = \{x : \exists y, z, (x, y) \in R \wedge (y, z) \in S \wedge (x, z) \notin S\}$

4

Teoria de Conjuntos

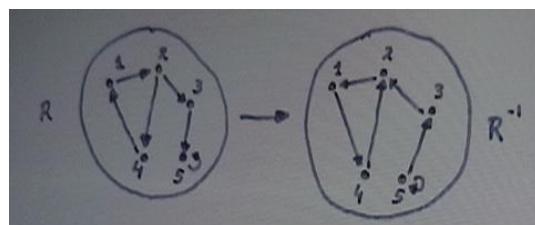
4.1 Relação inversa

Seja R uma relação binária em A . Chama-se **relação inversa** de R à relação

$$R^{-1} = \{(x, y) \in A^2 : yRx\}$$

isto é

$$xR^{-1}y \Leftrightarrow yRx.$$



$$\mathcal{M}_S \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} = \mathcal{M}_{S^{-1}}$$

Exemplo 4.1.1. Seja $T = \{(1, 5), (2, 6), (3, 7), (3, 8)\}$. Então $T^{-1} = \{(5, 1), (6, 2), (7, 3), (8, 3)\}$.

Se R é uma relação em A , então R^{-1} também é uma relação em A . Se R é uma relação de A para B , então R^{-1} é uma relação em B para A .

Podemos notar ainda que, se M_R é a matriz da relação R , a matriz $M_{R^{-1}}$ da relação inversa R^{-1} se obtém por transposição da matriz de R . Assim escrevemos

$$M_{R^{-1}} = (M_R)^T.$$

Para relações R e S definidas no mesmo conjunto A , com representações matriciais M_R e M_S de elementos genéricos a_{ij} e b_{ij} , é usual ainda definir:

1. $R \cup S$, a relação definida em A , como a união de R e S . Neste caso a matriz $M_{R \cup S}$ tem por elemento genérico $c_{ij} = a_{ij} \vee b_{ij}$.
2. $R \cap S$, a relação definida em A , como a intersecção de R e S . Neste caso a matriz $M_{R \cap S}$ tem por elemento genérico $c_{ij} = a_{ij} \wedge b_{ij}$.
3. $R \oplus S$, a relação definida em A , como a soma disjunta de R e S . Neste caso a matriz $M_{R \oplus S}$ tem por elemento genérico $c_{ij} = a_{ij} \oplus b_{ij}$.

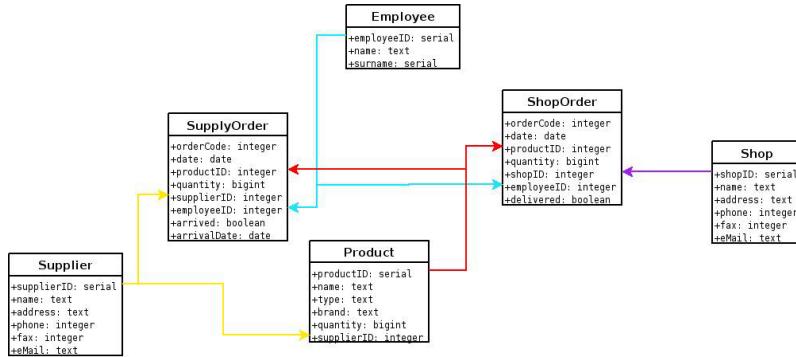
Exemplo 4.1.2. Para relações R e S , definidas no mesmo conjunto A , representadas pelas matrizes

$$M_R = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \quad M_S = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}.$$

tem-se pelas definições anteriores que:

$$M_{R \cup S} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \quad M_{R \cap S} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M_{R \oplus S} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

4.2 Composição de relações



Sejam A , B e C conjuntos. Seja R uma relação de A em B , i.e. $R \subseteq A \times B$, e S uma relação de B em C , i.e. $S \subseteq B \times C$. A composição de R e S é a relação dada por:

$$S \circ R = \{(a, c) \in A \times C : \exists b \in B, aRb \wedge bSc\}.$$

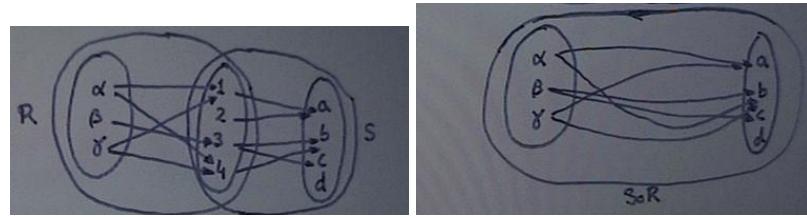
Exemplo 4.2.1. Seja R uma relação de $\{\alpha, \beta, \gamma\}$ em $\{1, 2, 3, 4\}$ dada por:

$$R = \{(\alpha, 1), (\alpha, 4), (\beta, 3), (\gamma, 1), (\gamma, 4)\}$$

e S a relação de $\{1, 2, 3, 4\}$ em $\{a, b, c, d\}$ dada por

$$S = \{(1, a), (2, a), (3, b), (3, c), (4, b)\}$$

temos



ou seja

$$S \circ R = \{(\alpha, a), (\alpha, b), (\beta, b), (\beta, c), (\gamma, a), (\gamma, b)\}$$

Exemplo 4.2.2. Para $S = \{(1, 2), (3, 4), (2, 2)\}$ e $R = \{(4, 2), (2, 5), (3, 1), (1, 3)\}$ tem-se

1. $R \circ S = \{(1, 5), (3, 2), (2, 5)\}$
2. $S \circ R = \{(4, 2), (3, 2), (1, 4)\} \neq R \circ S$
3. $(S \circ R) \circ S = \{(3, 2)\}$
4. $S \circ (S \circ R) = \{(3, 2)\} = (S \circ R) \circ S$
5. $S \circ S = \{(1, 2), (2, 2)\}$
6. $R \circ R = \{(4, 5), (3, 3), (1, 1)\}$
7. $S \circ S \circ S = \{(1, 2), (2, 2)\}$

Recorde que se a matriz M_R que representa a relação R , tem elemento genérico a_{ij} , então em M_R temos $a_{i,j} = 1$ se e só se iRj . Podemos concluir que:

Proposição 4.2.3. Dadas relações finitas R de A para B e S de B para C . Se

$$M_{S \circ R} = [c_{ij}], M_R = [a_{ik}] \text{ e } M_S = [b_{kj}]$$

temos que, para todo $i \in A$ e $j \in C$ temos para $B = \{1, 2, \dots, n\}$

$$c_{ij} = (a_{i1} \wedge b_{1j}) \vee (a_{i2} \wedge b_{2j}) \vee \dots \vee (a_{in} \wedge b_{nj})$$

ou seja

$$c_{ij} = \bigvee_{k \in B} (a_{ik} \wedge b_{kj}) \text{ para todo } i \in A, j \in C.$$

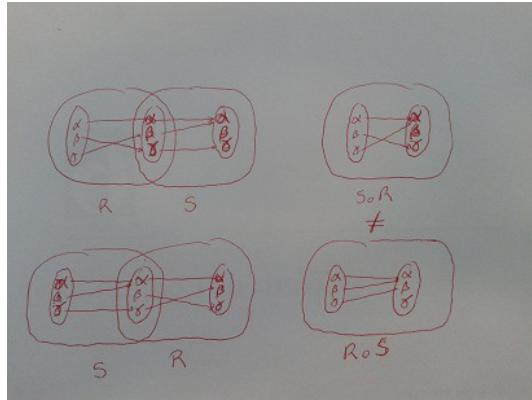
Podendo esta relação ser apresentada como o produto booleano das matrizes M_R e M_S :

$$M_{S \circ R} = M_R \times M_S$$

onde o produto das matrizes é definido na álgebra de Boole {0,1}

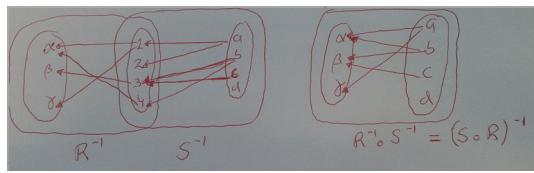
Proposição 4.2.4. A composição de relações:

1. não goza da propriedade comutativa;



$$2. \text{ é associativa } S \circ (R \circ T) = (S \circ R) \circ T;$$

$$3. (S \circ R)^{-1} = R^{-1} \circ S^{-1}$$



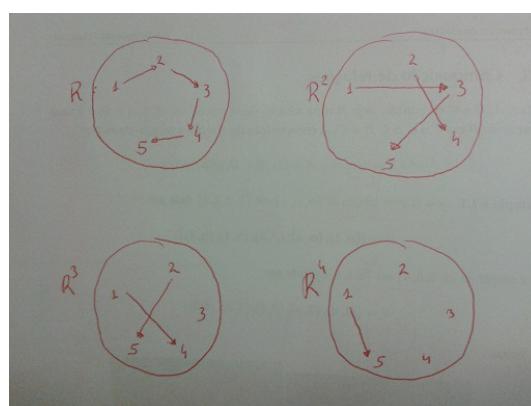
4.3 Potência dum relação

Seja R uma relação binária no conjunto A . A potência R^n é uma relação definida recursivamente por:

1. $R^1 = R;$
2. $R^{n+1} = R^n \circ R.$

Assim

- $(a, c) \in R^2 \Leftrightarrow \exists b(aRb \wedge bRc)$
- $(a, d) \in R^3 \Leftrightarrow \exists b, c(aRb \wedge bRc \wedge cRd)$
- $(a, e) \in R^4 \Leftrightarrow \exists b, c, d(aRb \wedge bRc \wedge cRd \wedge dRe)$
- \vdots



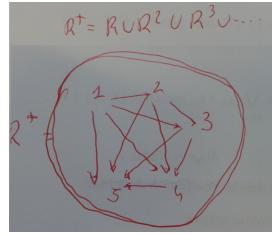
Podemos mostrar que:

Proposição 4.3.1. *Para um conjunto finito e uma relação R . A relação*

$$R \cup R^2 \cup R^3 \cup \dots$$

é igual ao fecho transitivo de R , i.e.

$$R^+ = R \cup R^2 \cup R^3 \cup \dots$$



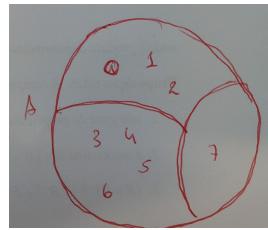
4.4 Partição dum conjunto

Seja A um conjunto e B_1, B_2, \dots, B_n uma família de subconjuntos de A . Diz-se que a família B_1, B_2, \dots, B_n é uma partição do conjunto A se:

1. nenhum dos subconjuntos é vazio, i.e. $B_i \neq \emptyset$ para $i = 1, 2, \dots, n$;
2. os subconjuntos são disjuntos dois a dois, i.e. $B_i \cap B_j = \emptyset$ para $i \neq j$;
3. a união de todos os subconjuntos é o conjunto A , i.e. $\bigcup_i B_i = A$.

Exemplo 4.4.1. São assim exemplos de partições do conjunto $A = \{0, 1, 2, 3, 4, 5, 6, 7\}$ as famílias:

1. $B_1 = \{0, 1, 2\}, B_2 = \{3, 4, 5, 6\}$ e $B_3 = \{7\}$



2. $B_1 = \{3, 1, 2\}, B_2 = \{0, 6\}$ e $B_3 = \{4, 5, 7\}$
3. $B_1 = \{0, 1, 2, 7\}$ e $B_2 = \{3, 4, 5, 6\}$

4.5 Relação de equivalência

Certas relações apresentam forte semelhança com a relação de igualdade. Um exemplo (da geometria) é a relação de congruência no conjunto dos triângulos. Falando de forma genérica, triângulos são congruentes se têm exactamente a mesma forma. Triângulos congruentes não são iguais, podem estar localizados em partes diferentes, mas

geometricamente são indistinguíveis. Mas a relação de congruência tem características idênticas à relação de equivalência, é reflexiva, simétrica e transitiva. As relações com estas características são designadas de relações de equivalência.

Seja A um conjunto e R uma relação binária definida em A . Diz-se que R é uma **relação de equivalência** em A se é:

1. **reflexiva**: $\forall x(xRx)$;
2. **simétrica**: $\forall x, y(xRy \rightarrow yRx)$;
3. **transitiva**: $\forall x, y, z((xRy \wedge yRz) \rightarrow xRz)$.

Exercício 4.5.1. Determinemos todas as relações de equivalência definidas num conjunto com três elementos.

Exemplo 4.5.2. Seja R a relação no conjunto dos números inteiros tal que aRb se e só se $a + b$ é um inteiro par

1. Para todo o inteiro a , aRa já que $a + a = 2a$ é par;
2. Para todos os inteiros a e b , se aRb então $a + b$ é par, logo pela comutatividade da adição $b + a$ é par. Assim bRa .
3. Para todos os inteiros a, b e c , se aRb e bRc então $a + b$ e $b + c$ são pares. Logo $a + 2b + c$ é par, ou seja $a + c$ é par. Donde aRc .

Por 1, 2 e 3 podemos concluir que a relação R é uma relação de equivalência.

Seja R uma relação de equivalência em A , para todo $a \in A$,

$$[a]_R = \{s : aRs\},$$

e.i. o conjunto de todos os elementos que estão relacionados com a por R . Este conjunto é designado de **classe de equivalência** de a na relação R

Exemplo 4.5.3. Dada a relação de equivalência R , do exemplo anterior, temos:

$$[1]_R = \{1, 2\}, [3]_R = \{3, 4, 5\} = [4]_R = [5]_R, [6]_R = \{6\}$$

Note que $[1]_R, [3]_R$ e $[6]_R$ definem uma partição de $\{1, 2, 3, 4, 5, 6\}$.

Dado $b \in [a]_R$, dizemos que b é um **representante** dessa **classe de equivalência**.

Notemos que, se aRb , temos naturalmente que $[a]_R = [b]_R$ e $[a]_R \cap [b]_R \neq \emptyset$. Mais, sempre que $[a]_R \cap [b]_R \neq \emptyset$ temos $[a]_R = [b]_R$, ou seja:

Proposição 4.5.4. Seja R uma relação de equivalência num conjunto A . As seguintes condições são equivalentes:

1. aRb
2. $[a]_R = [b]_R$
3. $[a]_R \cap [b]_R \neq \emptyset$

Por negação,

$$a \not R b \Leftrightarrow [a]_R \neq [b]_R \Leftrightarrow [a]_R \cap [b]_R = \emptyset,$$

garantindo que:

Proposição 4.5.5. Seja R uma relação de equivalência num conjunto A . Então, as classes de equivalência de R formam uma partição de A .

Esta partição denomina-se **conjunto quociente** de A por R , que denotamos por A/R .

4.6 Funções

Seja $f \subseteq A \times B$ uma relação de A para B . Se, para todo o $x \in A$ existir um e um só $y \in B$ tal que $(x, y) \in f$ dir-se-á que f é uma **aplicação** (ou **função**) de A em B ; para significar que f é uma aplicação de A em B costuma escrever-se

$$f : A \rightarrow B$$

e, neste caso, escreve-se $y = f(x)$, dizendo-se que $y \in B$ é a imagem por f de $x \in A$.

Dada uma função $f : A \rightarrow B$, ao conjunto A também se dá o nome de **domínio** de f e com este significado representa-se por $D(f)$.

Exemplo 4.6.1. Como exemplo considere-se

- $A = \{1, 2, 3, 4\}$
- $B = \{1, 2, 3, 4, 5\}$
- $f = \{(1, 2), (2, 3), (3, 4), (4, 5)\}$
- $g = \{(1, 2), (1, 3), (2, 4), (3, 5), (4, 5)\}$
- $h = \{(1, 1), (2, 2), (3, 3)\}$

assim definidas f, g e h são relações de A para B mas apenas f é uma função definida em A ; g e h não são funções definidas em A , a primeira porque $(1, 2)$ e $(1, 3)$ estão em g , a segunda porque $D(h) = \{1, 2, 3\} \neq A$. A função f é particularmente simples, podendo ser descrita pela fórmula $f(x) = x + 1$ qualquer que seja $x \in A$. Embora a maior parte das funções, normalmente consideradas em disciplinas de Análise Matemática, sejam dadas de forma semelhante, em geral, não se podem especificar as funções deste modo; de facto, a maioria das funções que se podem definir não podem ser descritas de forma simples usando uma fórmula algébrica.

O conjunto

$$Im(f) = f(A) = \{y \in B : \exists x, x \in A \wedge y = f(x)\}$$

designa-se por **contradomínio** da função f .

O gráfico duma função $f : A \rightarrow B$ é o subconjunto $G(f)$ do produto cartesiano $A \times B$ formado pelos pares ordenados $(x, f(x))$, onde $x \in A$ é arbitrário. Ou seja

$$G(f) = \{(x, y) \in A \times B : y = f(x)\}.$$

Da definição de igualdade de funções obtém-se que duas funções são iguais se, e somente se, possuem o mesmo gráfico.

4.7 Função sobrejectiva, injectiva e bijectiva

Se $f(A) = B$ dir-se-á que f é uma **função sobrejectiva**; a função $f : A \rightarrow B$ diz-se **injectiva** (ou unívoca) se cada elemento de $f(A)$ for imagem de um só elemento de A , isto é, f é injectiva se e só se

$$\forall x, y \in A, x \neq y \Rightarrow f(x) \neq f(y)$$

o que significa que elementos distintos de A têm necessariamente imagens por f diferentes em $f(A) \subset B$. Se a aplicação $f : A \rightarrow B$ for simultaneamente injectiva e sobrejectiva dizemos que f é uma **aplicação bijectiva**.

4.8 Igualdade

Duas aplicações f e g são iguais, escrevendo-se então $f = g$, se e só se forem satisfeitas as duas condições seguintes

1. $D(f) = D(g)$ e
2. $\forall x \in D(f), f(x) = g(x)$.

4.9 Aplicação composta

Sejam A, B, C três conjuntos não vazios e $f : A \rightarrow B$ e $g : B \rightarrow C$ duas aplicações de A em B e B em C , respectivamente. Chama-se **aplicação composta** de g com f à aplicação

$$g \circ f : A \rightarrow B,$$

definida por $g \circ f(x) = g(f(x))$, para todo $x \in A$.

4.10 Associatividade

A composição goza de algumas propriedades importantes das quais se destacam a associatividade. Dadas aplicações $f : A \rightarrow B$, $g : B \rightarrow C$ e $h : C \rightarrow D$ temos que

$$(h \circ g) \circ f = h \circ (g \circ f).$$

uma vez que, para todo $x \in A$:

$$\begin{aligned} ((h \circ g) \circ f)(x) &= (h \circ g)(f(x)) = h(g(f(x))) = \\ &= h((g \circ f)(x)) = (h \circ (g \circ f))(x). \end{aligned}$$

Podemos observar que, mais geralmente, basta que a imagem $f(A)$ da função f esteja contida no domínio de g para que a definição $(g \circ f)(x) = g(f(x))$ faça sentido.

Se $f : A \rightarrow B$ e $g : B \rightarrow C$ são injectivas então $g \circ f : B \rightarrow C$ é injectiva. Também a composta de funções sobrejectivas é sobrejectiva.

Por outro lado, qualquer função $f : A \rightarrow B$ pode ser escrita como a composta $f = h \circ f_1$ duma função injectiva h com uma função sobrejectiva. Basta para isso considerar $f_1 : A \rightarrow f(A)$, definida por $f_1(x) = f(x)$, e a inclusão $h : f(A) \rightarrow B$.

4.11 Aplicação identidade

Dado um conjunto A chama-se **aplicação identidade** em A à aplicação $id_A : A \rightarrow A$, definida por

$$id_A(x) = x,$$

qualquer que seja $x \in A$. Sendo $f : A \rightarrow B$ uma aplicação arbitrária então $id_B \circ f = f$ e $f \circ id_A = f$.

4.12 Imagem

Seja $f : A \rightarrow B$ uma função e E uma parte de A . Chama-se **imagem** de E por f e representa-se por $f(E)$ ao conjunto

$$f(E) = \{y \in B : \exists x \in E, y = f(x)\},$$

é usual também escrever-se

$$f(E) = \{f(x) \in B : x \in E\}.$$

4.13 Imagem recíproca

Se F for uma parte de B , chama-se **imagem recíproca** ou **inversa** de F e representa-se por $f^{-1}(F)$ ao conjunto assim definido

$$f^{-1}(F) = \{x \in A : \exists y \in F, y = f(x)\},$$

ou

$$f^{-1}(F) = \{x \in A : f(x) \in F\}.$$

Sejam $f : A \rightarrow B$ e $g : B \rightarrow C$ funções. Dado $X \subseteq A$, tem-se $(g \circ f)(X) = g(f(X))$. Se $Z \subseteq C$, temos $(g \circ f)^{-1}(Z) = f^{-1}(g^{-1}(Z))$.

Note que, se $f : A \rightarrow B$ é uma aplicação bijectiva a correspondência recíproca, que a cada $y \in B$ associa $f^{-1}(y)$, o único elemento do conjunto $f^{-1}(\{y\})$, é uma aplicação bijectiva e $f \circ f^{-1} = Id_B$ e $f^{-1} \circ f = Id_A$. A aplicação $f^{-1} : B \rightarrow A$ definida é chamada **aplicação inversa** ou **recíproca** de $f : A \rightarrow B$.

Neste sentido, dizemos que $g : B \rightarrow A$ é **inverso à esquerda** de $f : A \rightarrow B$ quando $g \circ f = Id_A : A \rightarrow A$, ou seja $g(f(x)) = x$ para todo $x \in A$.

Por exemplo, sejam A o conjunto dos reais maiores que zero e \mathbb{R} o conjunto de todos os números reais. Consideremos $f : A \rightarrow \mathbb{R}$, definida por $f(x) = x^2$, e $g : \mathbb{R} \rightarrow A$, definida por $g(y) = \sqrt{y}$ se $y \geq 0$ e $g(y) = 0$ se $y < 0$. Para todo $x \in A$, temos $g(f(x)) = g(x^2) = \sqrt{x^2} = x$. Logo $g \circ f = id_A$ e portanto, g é uma inversa à esquerda de f .

Proposição 4.13.1. *Uma função $f : A \rightarrow B$ possui inversa à esquerda se, e somente se, é injectiva.*

Uma função $g : B \rightarrow A$ é **inverso à direita** de $f : A \rightarrow B$ quando $f \circ g = Id_B : B \rightarrow A$, ou seja $f(g(y)) = y$ para todo $y \in B$. De forma dual à proposição anterior temos que: Uma função $f : A \rightarrow B$ possui inversa à direita se, e somente se, é sobrejectiva.

Podemos assim dizer que uma função f tem inversa caso tenha inversa à direita e à esquerda. Para que tal se verifique é condição necessária e suficiente que f seja bijectiva. Neste caso a inversa de f é única.

Escreve-se f^{-1} para identificar a inversa dumha função bijectiva f . Supondo que g também é bijectiva temos

$$(g \circ f)^{-1} = f^{-1} \circ g^{-1}.$$

4.14 Correspondência biunívoca

Se f é uma aplicação bijectiva de A para B , então f^{-1} também é uma aplicação bijectiva mas de B para A , e f diz-se uma correspondência biunívoca entre A e B .

A noção de correspondência biunívoca é da maior importância em Matemática. Dizer que existe uma tal correspondência entre dois conjuntos A e B é como dizer que "os dois conjuntos têm o mesmo número de elementos" já que os elementos se podem associar "um a um, sem exceção".

Se os dois conjuntos são finitos, dizer que entre eles existe uma correspondência biunívoca é afirmar, os dois conjuntos têm o mesmo cardinal (mesmo número de elementos). Em particular, se A é um conjunto finito e C é uma parte própria de A , não existe correspondência biunívoca entre A e B . Notemos no entanto que caso A seja infinito, por exemplo $\mathbb{N} = \{1, 2, 3, \dots\}$, é possível definir um correspondência deste conjunto uma das suas partes próprias (por exemplo, a correspondência $n \mapsto n + 1$ entre \mathbb{N} e $\{2, 3, 4, \dots\}$). Neste sentido é usual definir:

Definição: 4.14.1. *Um conjunto A diz-se infinito quando tem uma parte própria B que esteja em correspondência biunívoca com A . Caso contrário diz-se finito.*

Podemos aproveitar as considerações anteriores para motivar a definição:

Definição: 4.14.2. *Diz-se que dois conjuntos A e B têm o mesmo cardinal, o mesmo número cardinal ou a mesma potência, quando existe uma correspondência biunívoca entre A e B*

Noção que permite mostrar que os conjuntos \mathbb{N} , \mathbb{Z} e \mathbb{Q} , têm a mesma potência, mas que têm potência inferior a \mathbb{R} . Permitindo descrever duas ordens de infinito, o infinito do numerável de \mathbb{N} , \mathbb{Z} e \mathbb{Q} e o infinito do contínuo de \mathbb{R} e \mathbb{C} .

4.15 EXERCÍCIOS DE REVISÃO

Exercício 4.15.1. Para cada uma das seguintes relações definidas no conjunto $\{1, 2, 3, 4, 5\}$, determine se a relação é reflexiva, anti-simétrica e/ou transitiva.

1. $\{(1, 1), (2, 2), (3, 3), (4, 4), (5, 5)\}$
2. $\{(1, 2), (2, 3), (3, 4), (4, 5)\}$
3. $\{(1, 1), (1, 2), (1, 3), (1, 4), (1, 5)\}$
4. $\{(1, 1), (1, 2), (2, 1), (3, 4), (4, 3)\}$
5. $\{1, 2, 3, 4, 5\} \times \{1, 2, 3, 4, 5\}$

Exercício 4.15.2. Dado $S = \{1, 2, \dots, 10\}$ e a relação R em S ,

$$R = \{(x, y) : x + y = 10\}.$$

Classifique a relação R .

Exercício 4.15.3. Determine R^{-1} para cada uma das seguintes relações:

1. $R = \{(1, 2), (2, 3), (3, 4)\}$
2. $R = \{(1, 1), (2, 2), (3, 3)\}$
3. $R = \{(x, y) : x, y \in \mathbb{Z}, x - y = 1\}$
4. $R = \{(x, y) : x, y \in \mathbb{N}, y \text{ divide } x\}$
5. $R = \{(x, y) : x, y \in \mathbb{Z}, xy > 0\}$

Exercício 4.15.4. Quais dos seguintes conjuntos são relações de equivalência?

1. $R = \{(1, 1), (1, 2), (2, 1), (2, 2), (3, 3)\}$ no conjunto $A = \{1, 2, 3\}$
2. $R = \{(1, 2), (2, 3), (3, 1)\}$ no conjunto $A = \{1, 2, 3\}$
3. \leq em \mathbb{Z}
4. $\{1, 2, 3\} \times \{1, 2, 3\}$ no conjunto $\{1, 2, 3\}$
5. $\{1, 2, 3\} \times \{1, 2, 3\}$ no conjunto $\{1, 2, 3, 4\}$

Exercício 4.15.5. Para cada relação de equivalência, encontre a classe de equivalência pedida.

1. $R = \{(1, 1), (1, 2), (2, 1), (2, 2), (3, 3), (4, 4)\}$ em $\{1, 2, 3, 4\}$. Ache [1].
2. $R = \{(1, 1), (1, 2), (2, 1), (2, 2), (3, 3), (4, 4)\}$ em $\{1, 2, 3, 4\}$. Ache [4].
3. Em $\{x \in \mathbb{Z} : 100 < x < 200\}$, xRy se x e y têm o mesmo algarismo das dezenas. Determine [123].

Exercício 4.15.6. Seja R a relação no conjunto ordenado de pares de inteiros positivos tal que $(x, y)R(u, v)$ se e só se $xv = yu$. Mostre que R é uma relação de equivalência.

Exercício 4.15.7. Mostre que uma relação R é simétrica se e só se $R = R^{-1}$.

Exercício 4.15.8. Dê um exemplo de uma relação que seja simétrica e transitiva, mas não reflexiva. Explique o que está errado no argumento seguinte:

“Seja R simétrica e transitiva. Pela simetria se xRy , yRx . Aplicando a transitividade a xRy e yRx , obtemos xRx . Portanto, R é reflexiva.”

Exercício 4.15.9. Dado um conjunto $S = \{1, 2, 3, 4, 5\}$, apresente uma relação de equivalência S que determine a partição $\{\{1, 2\}, \{3\}, \{4, 5\}\}$. Apresente a relação graficamente.

Exercício 4.15.10. Mostre que a relação “congruente modulo m ” dada por

$$\equiv = \{(x, y) : x - y \text{ é divisível por } m\},$$

definida no conjunto dos inteiros, é uma relação de equivalência. Demonstre que, se $x_1 \equiv y_1$ e $x_2 \equiv y_2$, então $(x_1 + x_2) \equiv (y_1 + y_2)$.

Exercício 4.15.11. Para duas relações de equivalência R e S definidas pelas matrizes

$$M_R = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ e } M_S = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

Obtenha relações de equivalência R_1 e R_2 em $\{1, 2, 3\}$ tais que $R_1 \circ R_2$ seja também uma relação de equivalência.

Exercício 4.15.12. Seja \mathbb{Z} o conjunto dos inteiros, \mathbb{Z}^+ o conjunto dos inteiros positivos, e $\mathbb{Z}_n = \{0, 1, \dots, n-1\}$. Classifique as seguintes aplicações quanto à injectividade, sobrejectividade e bijectividade.

1. $f : \mathbb{Z} \rightarrow \mathbb{Z}$, $f(j) = \begin{cases} j/2 & \text{se } j \text{ é par} \\ (j-1)/2 & \text{se } j \text{ é ímpar} \end{cases}$
2. $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}^+$, $f(x) = \text{maior inteiro} \leq \sqrt{x}$
3. $f : \mathbb{Z}_7 \rightarrow \mathbb{Z}_7$, $f(x) = 3x \pmod{7}$
4. $f : \mathbb{Z}_4 \rightarrow \mathbb{Z}_4$, $f(x) = 3x \pmod{4}$

Exercício 4.15.13. Se A e B são conjuntos finitos, determine uma condição necessária e suficiente para que exista uma aplicação injectiva de A para B .

Exercício 4.15.14. Os conjuntos abaixo definem funções? Nesse caso, determine o seu domínio e codomínio.

1. $\{(1, (2, 3)), (2, (3, 4)), (3, (1, 4)), (4, (1, 4))\}$
2. $\{(1, (2, 3)), (2, (3, 4)), (3, (3, 2))\}$
3. $\{(1, (2, 3)), (2, (3, 4)), (1, (2, 4))\}$
4. $\{(1, (2, 3)), (2, (2, 3)), (3, (2, 3))\}$

Exercício 4.15.15. Liste todas as funções do conjunto $A = \{a, b, c\}$ para $B = \{0, 1\}$ e indique em cada caso quando é injectiva, sobrejectiva e bijectiva.

Exercício 4.15.16. Se $A = \{1, 2, \dots, n\}$, mostre que qualquer função de A para A que seja injectiva é sobrejectiva.

Exercício 4.15.17. Seja $f : \mathbb{R} \rightarrow \mathbb{R}$ e $g : \mathbb{R} \rightarrow \mathbb{R}$, onde \mathbb{R} é o conjunto dos números reais. Determine $f \circ g$ e $g \circ f$, onde $f(x) = x^2 - 2$ e $g(x) = x + 4$.

Exercício 4.15.18. Mostre que sempre que $f : A \rightarrow B$ e $g : B \rightarrow C$ são sobrejectivas, $g \circ f$ é sobrejectiva. A aplicação $g \circ f$ é bijectiva se g e f são bijectivas?

Exercício 4.15.19. Seja $f : \mathbb{R} \rightarrow \mathbb{R}$ dada por $f(x) = x^3 + 2$. Determine f^{-1} .

Exercício 4.15.20. Quantas funções diferentes pode definir entre os conjuntos dados abaixo? Determine o número de funções injectivas, sobrejectivas e bijectivas.

1. $A = \{1, 2, 3\}$ $B = \{1, 2, 3\}$
2. $A = \{1, 2, 3, 4\}$ $B = \{1, 2, 3\}$
3. $A = \{1, 2, 3\}$ $B = \{1, 2, 3, 4\}$

Exercício 4.15.21. Mostre que existe uma aplicação bijectiva de $A \times B$ para $B \times A$.

5

Python

Existe um mecanismo para remover um elemento de uma lista com base no seu índice: o comando *del*. Este comando difere do método *pop()* que devolve o valor removido. O comando *del* pode ser usado para remover *slices* de uma lista ou eliminar um objecto.

For example:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

del pode ser usado para eliminar variáveis:

```
>>> del a
```

5.1 Técnicas para ciclos

Quando se faz um ciclo através de uma sequência, o índice posição e o correspondente valor podem ser restritos ao mesmo tempo usando a função *enumerate()*.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print i, v
...
```

```
0 tic  
1 tac  
2 toe
```

O ciclo ao longo da sequência reversa, pode ser definido como tendo por domínio a sequência na ordem natural e revertida através de função *reversed()*.

```
>>> for i in reversed(xrange(1,10,2)):  
...     print i  
...  
9  
7  
5  
3  
1
```

Para fazer o ciclo numa sequência ordenada, pode usar a função *sorted()* que devolve uma lista ordenada sem que altere a fonte.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']  
>>> for f in sorted(set(basket)):  
...     print f  
...  
apple  
banana  
orange  
pear
```

Num ciclo ao longo de um dicionário, a chave e o correspondente valor podem ser obtidos ao mesmo tempo, usando o método *iteritems()*.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}  
>>> for k, v in knights.iteritems():  
...     print k, v  
...  
gallahad the pure  
robin the brave
```

Para iterar numa sequência à qual pretende fazer alterações no interior do ciclo é conveniente fazer uma cópia da estrutura. Não é garantido que o ciclo faça uma cópia. É uma técnica útil fazer aqui uma *slice*:

```
>>> words = ['cat', 'window', 'defenestrate']  
>>> for w in words[:]: # Loop over a slice copy of the entire list.  
...     if len(w) > 6:  
...         words.insert(0, w)  
...  
>>> words  
['defenestrate', 'cat', 'window', 'defenestrate']
```

5.2 Python: Mais relativamente a condições

As condições usadas com as instruções *while* e *if* podem conter qualquer operador para além de comparações.

Os operadores de comparação *in* e *not in* permitem avaliar se um elemento pertence a uma sequência. Os operadores *is* and *is not* comparam se dois objectos são realmente o mesmo; importantes apenas no caso dos objectos mutáveis como as listas. Todos os operadores de comparação têm a mesma prioridade, que é inferior que qualquer outro operador numérico.

As comparações podem aparecer em cadeia. Por exemplo, $a < b == c$ testa se a é menor que b e que b é igual a c .

As comparações podem ser combinadas usando operadores booleanos *and* e *or*, podendo o resultado da comparação ser negado usando o operador *not*. Estes operadores têm menor prioridade que os operadores de comparação; no entanto entre eles, o *not* tem maior prioridade e *or* a menor, assim $A \text{ and } \text{not } B \text{ or } C$ é equivalente a $(A \text{ and } (\text{not } B)) \text{ or } C$.

Qualquer sequência de objectos pode ser comparada a outro objecto com o mesmo tipo de sequências. A comparação usa ordem lexicográfica: os primeiros dois elementos são comparados, e se eles diferem isto determina o resultado da comparação; se são iguais, os próximos dois elementos são comparados, e assim sucessivamente. Se dois elementos a comparar são sequências do mesmo tipo, a comparação lexicográfica é feita recursivamente. Quando todos os elementos de uma sequência são iguais, as sequências são assumidos iguais.

Alguns exemplos:

```
(1, 2, 3)           < (1, 2, 4)
[1, 2, 3]          < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)       < (1, 2, 4)
(1, 2)             < (1, 2, -1)
(1, 2, 3)          == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a')), 4)
```

Note que, a comparação de objectos de tipo diferente é legal. O resultado é determinado mas arbitrário: uma lista é sempre mais pequena que um string, uma string é sempre menor que um tuple, etc. A comparação de tipos numéricos misturados é feita de acordo com o seu valor. Assim 0 é igual a 0.0, 1 é igual 1.0, etc.

5.3 Módulos

Notou de certeza que sempre que desliga o interpretador, todas as suas definições de funções e variáveis são perdidas. Para escrever programas longos é conveniente escreve-los num editor de texto, por forma a definir um *script*, muitas vezes por uma questão de organização, porque o código começa a ficar demasiado longo, ou porque

quer usar pares comuns de código em diferentes projectos, o código pode ser distribuído por diferentes ficheiros.

O Python permite incorporar partes de código, por diferentes ficheiros. Estas partes de código são designadas de módulos.

Um módulo é um ficheiro contendo definições e comandos em Python. O nome de um módulo é o nome de um ficheiro com extensão .py. Num módulo o seu nome pode ser recuperado através da variável global `__name__`. Por exemplo no seu editor de texto preferido ou no IDLE crie um ficheiro chamado `fibo.py` com o seguinte conteúdo:

```
# Fibonacci numbers module

def fib(n):      # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Agora no interpretador de Python importe este módulo com o seguinte comando:

```
>>> import fibo
```

Isto não instância as funções com os nomes definidos no ficheiro `fibo.py`. Para aceder às funções tem de explicitar o nome do módulo:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Caso queira usar uma das funções muitas vezes pode sempre associá-la a uma variável local:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

5.3.1 Mais sobre módulos

Um módulo pode conter tanto comandos executáveis como definição de funções. Os comandos, como se assume que servem para iniciar o módulo são executados apenas quando o módulo é importado.

Cada módulo tem a sua tabela privada de símbolos, usada como a tabela de símbolos globais para todas as funções definidas no módulo. Podendo assim o autor do módulo usar variáveis globais sem a preocupação de esmagar as variáveis globais dos utilizadores do módulo. Por outro lado o utilizador pode sempre alterar as variáveis globais do módulo de forma idêntica ao usado para funções, `modname.itemname`.

Um módulo pode ser sempre importado por outros módulos. É usual mas não obrigatório por as instruções `import` no inicio do módulo. Sendo os nomes do módulo colocados na tabela de símbolos globais no módulo que os importa.

Uma variante da instrução `import`, permite importar directamente nomes específicos da tabela de símbolos no módulo. Por exemplo:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Existe ainda outra variante que permite importar todos os nomes de um módulo:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Isto permite importar todos os nomes excepto os que têm início com um underscore (`_`).

Nota: Por razões de eficiência, cada módulo é apenas importado uma vez. Caso faça alteração num módulo, para usar na nova versão, tem necessariamente de reiniciar o interpretador ou, usar `reload()`, i.e. `reload(modulename)`.

5.3.2 Executar um módulo como um script

Pode sempre correr um módulo em Python com

```
python fibo.py <arguments>
```

o código no módulo será executado, da mesma forma que com a instrução `import`, mas com a variável `__name__` definida como "`__main__`". Caso junte ao final do código:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

pode fazer o código ter o comportamento de um script:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Neste caso o importe como um módulo, a chama á função não é despoletada:

```
>>> import fibo
>>>
```

Esta estratégia é muitas vezes usada para fornecer uma interface adequada ao módulo, ou por razões de teste.

5.3.3 O caminho de procura de módulos

Quando um módulo é importado, o interpretador procura primeiro os módulos pré-definidos com o nome. Caso não seja encontrado este é procurado na lista de diretórios dadas nas variáveis *sys.path*. O valor de *sys.path* é iniciado com os seguintes caminhos:

- o directório que contém o *script* (o directório corrente)
- *PYTHONPATH* (uma lista de directórios, com a mesma sintaxe que a variável da shell PATH)
- a parte de instalação por defeito do Python.

Depois de iniciado, um programa em Python pode modificar *sys.path*. Note que, o directório que contém o script que está a executar é o primeiro a ser procurado. POR FAVOR, não designe o seu ficheiro com o nome *pygame.py*

5.3.4 Ficheiros Python compilados

A existência de um ficheiro *project.pyc*, na pasta do seu projecto *project.py*, é um passo importante para acelerar o tempo para despoletar o programa, particularmente quando usa muitos módulos. Este ficheiro contém uma versão already-byte-compiled do seu programa. Após uma alteração em *project.py* a sua primeira execução é mais lenta, levando à criação de uma nova versão de *project.pyc*, que é usada nas subsequentes execuções.

Normalmente este tipo de ficheiro é criado, sempre que o código é compilado com sucesso.

5.3.5 A função *dir()*

A função pré-definida *dir()* é usada para determinar os nomes definidos num módulo. Devolvendo uma lista de strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
```

```
[ '__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
  '__stderr__', '__stdin__', '__stdout__', '_clear_type_cache',
  '_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
  'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
  'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
  'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
  'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
  'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
  'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
  'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
  'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
  'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
  'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
  'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
  'version', 'version_info', 'warnoptions' ]
```

Sem argumentos, *dir()* lista os nomes que tem definidos até esse momento:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', '__package__', 'a', 'fib', 'fibo', 'sys']
```

Note que lista todo o tipo de nomes: variáveis, módulos, funções, etc.

A função *dir()* não lista nomes de variáveis ou nomes pré-definidos. Caso pretenda listar também essas, são definidas no módulo standard *__builtin__*:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
 'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
 'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
 'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
 'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
 '__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
```

```
'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

5.3.6 Geração de números pseudo-aleatórios

O módulo *random* permite gerar números aleatórios seguindo diferentes distribuições.

1. *random.seed()* - inicia o gerador de números aleatórios
2. *random.randint(a, b)* - devolve um inteiro aleatório N tal que $a \leq N \leq b$.
3. *random.uniform(a, b)* - devolve um float aleatório N tal que $a \leq N < b$ quando $a \leq b$.

Exemplos simples:

```
>>> import random
>>> random.random()          # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)    # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)   # Integer from 1 to 10, endpoints included
7
```

6

Teoria de Conjuntos

6.1 Relação de Ordem Parcial

É frequente recorrer a ordens para ordenar os elementos de um conjunto. Por exemplo, ordenam-se palavras w_1 e w_2 , dizendo que $w_1 \leq w_2$ se w_1 vem no dicionário antes que w_2 . No cronograma dum projecto, descrito por uma relação entre tarefas, escrevemos $x \leq y$, se a tarefa x antecede a tarefa y . No conjunto dos números inteiros, é usual assumir que os números estão ordenados pela relação menor ou igual.

Uma **relação de ordem parcial** definida num conjunto A é uma relação:

1. reflexiva,
2. anti-simétrica, e
3. transitiva.

Exemplo 6.1.1. A relação menor ou igual, \leq , definida nos conjuntos de números inteiros, é neste sentido uma ordem parcial em \mathbb{Z} , já que:

1. \leq é reflexiva, para todo $n \in \mathbb{Z}$, $n \leq n$.
2. \leq é anti-simétrica, para todo $n, m \in \mathbb{Z}$, se $n \leq m$ e $m \leq n$ então temos $n = m$.
3. \leq é transitiva, para todo $n, m, p \in \mathbb{Z}$, se $n \leq m$ e $m \leq p$ então temos $n \leq p$.

Note que, a sua relação inversa \leq^{-1} , é uma relação que convencionamos denotar por \geq , é também uma ordem parcial em \mathbb{Z} . Enfatizamos a ordenação escrevendo (\mathbb{Z}, \leq) e (\mathbb{Z}, \geq) , e dizemos que são conjuntos parcialmente ordenados.

Exemplo 6.1.2. A relação R definida em $A = \{1, 2, 3, 4, 6\}$, por aRb se a divide b , é uma ordem parcial em A . Nesta caso a relação pode ser definida em extensão, através de

$$R = \{(1, 1)(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (2, 2), (2, 4), (2, 6), (3, 3), (3, 6), (4, 4), (6, 6)\}$$

É usual escrever (A, R) para deixar explícita que a relação R é usada para ordenar todos os elementos dum conjunto A .

Definição: 6.1.3 (Conjunto parcialmente ordenado). *O par $P = (A, R)$ é um conjunto parcialmente ordenado (CPO) se A é um conjunto e R é uma ordem parcial em A . É usual representar as relações de ordem parcial por \leq , neste caso denotamos genericamente um CPO por um par $P = (A, \leq)$, quando a relação \leq é uma ordem parcial definida em A .*

Genericamente para uma relação R , definida num conjunto A , e um par de elementos $(a, b) \in A \times A$, dizemos que a e b **estão em relação** se aRb ou bRa , caso contrário a e b **não estão em relação**. Neste sentido e no contexto dum CPO definimos:

Definição: 6.1.4 (Comparáveis). *Num conjunto parcialmente ordenado $P = (A, \leq)$, dois elementos a e b de A , dizem-se comparáveis se $a \leq b$ ou $b \leq a$.*

No entanto, sempre que num CPO $P = (A, \leq)$, não é válido tanto $a \leq b$ como $b \leq a$, a e b dizem-se **não comparáveis**.

Exemplo 6.1.5. Seja \leq a relação parcialmente ordenada definida em $A = \{1, 2, 3, 4, 6\}$, por $a \leq b$ se a divide b . No CPO (A, \leq) os elementos 2 e 4 são comparáveis já que 2 divide 4. Por outro lado os elementos 3 e 4 não são comparáveis, uma vez que 3 não divide 4, nem 4 divide 3. Neste sentido, o conjunto $\{1, 2, 3, 6\}$ é definido por elementos que são comparáveis dois a dois. Enquanto que o conjunto $\{3, 4\}$ é definido por elementos não comparáveis.

Um conjunto parcialmente ordenado (A, \leq) diz-se um **conjunto totalmente ordenado** se para qualquer par de elementos a e b , em A , a e b são comparáveis, ou seja, se se tem

$$a \leq b \text{ ou } b \leq a, \text{ para todo } a \text{ e } b \text{ em } A.$$

O CPO (\mathbb{Z}, \leq) é exemplo de um conjunto totalmente ordenado, para a ordem usual de menor ou igual. Para todo o par de inteiros x e y , ou x é menor ou igual a y , ou y é menor ou igual a x .

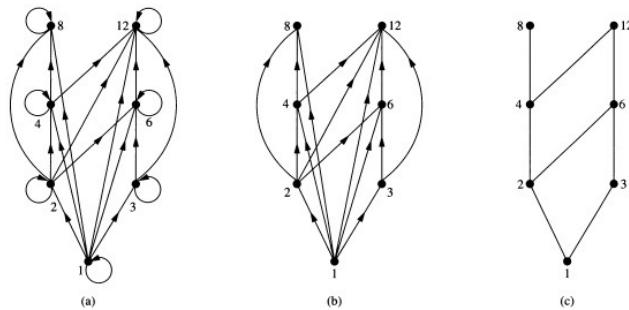
6.2 Diagrama de Hasse

Qualquer relação de ordem parcial, num conjunto finito, pode representar-se graficamente por um diagrama de Hasse. Ao contrário dos diagramas sagitais, neste diagrama os elementos são ligados por arcos não dirigidos. O diagrama de Hasse é obtido por simplificação do diagrama sagital da relação removendo todos os lacetes e todos os arcos que podem ser deduzidos pela transitividade da relação. Convenciona-se neste tipo de representação que, se um elemento precede (na relação \leq) um outro elemento, então o primeiro ficará situado mais abaixo na figura, permitindo a supressão da orientação usada no diagrama sagital, sem gerar ambiguidade.

Exemplo 6.2.1. Para desenhar o diagrama de Hasse que representa a ordem parcial

$$\leq = \{(a, b) : a \text{ divide } b\}$$

no conjunto $A = \{1, 2, 3, 4, 6, 8, 12\}$.



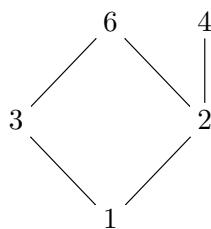
Começamos por desenhar em (a) o diagrama sagital da relação. Removendo em (b) todos os lacetes. Apagando depois todos os arcos que podem ser descritos pela transitividade. Os arcos removidos no terceiro passo são

$(1, 4), (1, 6), (1, 8), (1, 12), (2, 8), (2, 12)$ e $(3, 12)$.

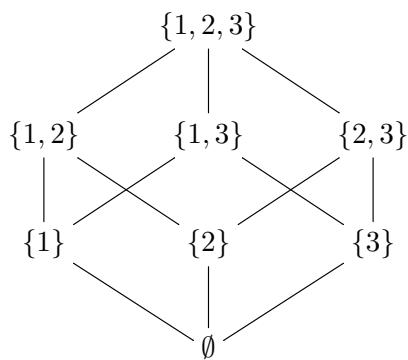
Ordenando os vértices por forma a que as setas apontem todas para cima e removendo a orientação, obtemos o diagrama de Hasse. O diagrama que resulta desta transformação é apresentado na figura acima, em (c).

De seguida apresentamos mais alguns exemplos de relações representadas através de diagramas de Hasse.

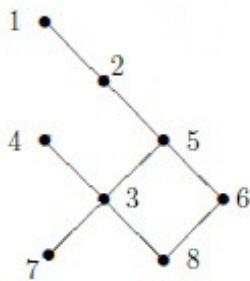
Exemplo 6.2.2. Em baixo apresentamos o diagrama de Hasse da relação parcialmente ordenada \leq definida em $A = \{1, 2, 3, 4, 6\}$, por $a \leq b$ se a divide b .



Exemplo 6.2.3. Em baixo apresentamos o diagrama de Hasse do conjunto das partes de $\{1, 2, 3\}$ ordenadas pela relação de inclusão.



Exemplo 6.2.4. Pelo diagrama de Hasse abaixo, que tem por suporte $A = \{1, 2, 3, 4, 5, 6, 7, 8\}$,

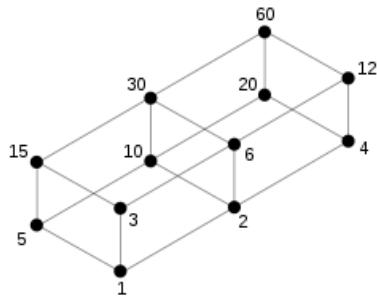


podemos concluir, por exemplo, que $7 \leq 3 \leq 4$, $7 \leq 5 \leq 2 \leq 1$, $8 \leq 3$, $8 \leq 6$, $8 \leq 4$ ou $8 \leq 1$. São proposições falsas no CPO $3 \leq 7$, $7 \leq 8$, $8 \leq 7$, $4 \leq 1$ e $3 \leq 5$.

Uma forma de caracterizar a complexidade dum CPO recorre à noção de altura e largura. Noções associadas à noção de cadeias e anti-cadeias.

Definição: 6.2.5 (Cadeia e anti-cadeia). *Seja $P = (A, \leq)$ um CPO e $C \subseteq A$. Diz-se que C é uma **cadeia** de P se todos os pares de elementos em C são comparáveis. Diz-se que C é uma **anti-cadeia** de P se todos os pares de elementos em C são não comparáveis.*

No CPO definido pelo diagrama de Hasse abaixo:

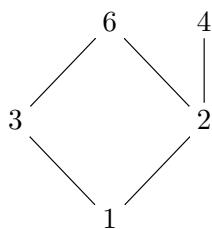


Os conjuntos $\emptyset, \{1\}, \{1, 2, 6\}, \{1, 3, 6\}, \{1, 3, 15, 30, 60\}$ definem cadeias, e são anti-cadeias os conjuntos $\emptyset, \{3, 4\}, \{5, 3, 4\}, \{6, 10, 4\}$. Neste exemplo, como 60 é maior que todos os outros elementos diz-se elemento máximo. O elemento 1 é elemento mínimo, já que é inferior a todos os outros elementos.

Definição: 6.2.6 (Altura e Largura). *Num CPO P , designa-se de altura de P ao tamanho máximo de uma cadeia. A largura de P é o tamanho máximo de uma anti-cadeia.*

No CPO definido pelo diagrama de Hasse anterior, são exemplos de cadeias de comprimento máximo $\{1, 2, 6, 30, 60\}$ e $\{1, 5, 10, 30, 60\}$, neste sentido o CPO tem altura 5. São exemplos de anti-cadeias de comprimento máximo $\{2, 3, 5\}$ e $\{2, 3, 5\}$, tendo o CPO tem largura 3.

O diagrama de Hasse



define um CPO com altura 2 e largura 2. Neste caso o CPO não têm máximo. Cada elemento do conjunto $\{4, 6\}$ diz-se maximal, porque cada um deles não tem elementos que lhe são maiores. No entanto, o CPO tem por mínimo 1.

Definição: 6.2.7 (Máximo e mínimo). *Seja $P = (A, \leq)$ um CPO. Dizemos que $n \in A$ é **máximo** se $\forall m \in A, m \leq n$. Dizemos que $n \in A$ é **mínimo** se $\forall m \in A, n \leq m$.*

Definição: 6.2.8 (Elemento maximal e Elemento minimal). *Seja $P = (A, \leq)$ um CPO. Dizemos que $n \in A$ é **maximal** se não existe $m \in A$ tal que $m \neq n$ e $n \leq m$. Dizemos que $n \in A$ é **minimal** se não existe $m \in A$ tal que $m \neq n$ e $m \leq n$.*

Relativamente à relação de ordem dum CPO, os seus subconjuntos são usualmente caracterizados recorrendo às noções apresentadas abaixo.

Definição: 6.2.9 (Máximos, mínimos, supremo, ínfimo, majorantes e minorantes). *Seja $P = (A, \leq)$ um CPO e C um subconjunto de A :*

1. *$m \in A$ é **majorante** de C se e só se $\forall a \in C, a \leq m$;*
2. *$m \in A$ é **minorante** de C se e só se $\forall a \in C, m \leq a$;*
3. *$s \in A$ é **supremo** de S se e só se s é majorante de S e para qualquer outro majorante s' de S temos $s \leq s'$. Se $s \in S$, então s designa-se por **elemento máximo de S** .*
4. *$i \in A$ é **ínfimo** de S se e só se i é minorante de S e para qualquer outro minorante i' de S temos $i' \leq i$. Se $i \in S$, então i designa-se por **elemento mínimo de S** .*
5. *$m \in S$ é elemento **maximal** de S se não existe $s' \in S$ tal que $s' \neq s$ e $s \leq s'$.*
6. *$m \in S$ é elemento **minimal** de S se não existe $s' \in S$ tal que $s' \neq s$ e $s' \leq s$.*

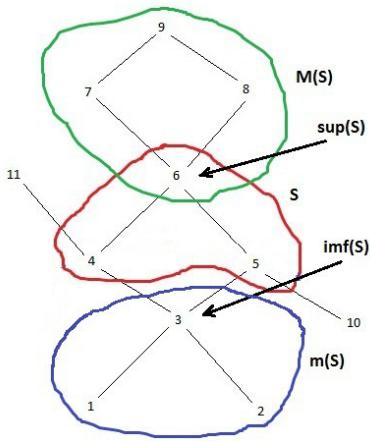
Aplicemos estas noções.

Exemplo 6.2.10. O diagrama abaixo define um CPO tendo por suporte

$$A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}.$$

Consideremos o subconjunto de A , $S = \{4, 5, 6\}$. Relativamente à ordem parcial S tem por:

1. *majorantes $M(S) = \{6, 7, 8, 9\}$, já que 6, 7, 8 e 9 são comparáveis e maiores ou iguais aos elementos de S ;*
2. *minorantes $m(S) = \{1, 2, 3\}$, já que 1, 2 e 3 são comparáveis e menores ou iguais aos elementos de S ;*
3. *supremo $\text{sup}(S) = 6$, já que 6 é o menor dos majorantes;*
4. *ínfimo $\text{inf}(S) = 3$, já que 3 é o maior dos minorantes;*
5. *elemento máximo de S é 6, já que 6 é supremo e $S \subseteq \text{sup}(S)$;*
6. *não existe elemento mínimo, já que $\text{inf}(S) \notin S$.*



6.3 O conjunto dos números naturais

Aqui tratamos o conjunto dos números naturais e revemos algumas das suas propriedades. Examinamos a construção dos naturais usando o método de indução e como este leva ao princípio da indução matemática.

6.4 Indução

O conjunto dos números naturais $\mathbb{N} = \{1, 2, 3, \dots\}$ pode ser gerado a partir do conjunto unitário $\{\emptyset\}$ com recurso à noção de conjunto sucessor. O sucessor dum conjunto A é denotado por $s(A)$ e definido como o conjunto $s(A) = A \cup \{A\}$.

Sendo $\{\emptyset\}$ o conjunto singular tendo por único elemento o conjunto vazio, podemos obter conjuntos sucessores $s(\{\emptyset\})$, $s(s(\{\emptyset\}))$, $s(s(s(\{\emptyset\})))$, Definindo conjuntos:

$$\{\emptyset\}, \{\emptyset\} \cup \{\{\emptyset\}\}, \{\emptyset\} \cup \{\{\emptyset\}\} \cup \{\{\emptyset\} \cup \{\{\emptyset\}\}\}, \{\emptyset\} \cup \{\{\emptyset\}\} \cup \{\{\emptyset\} \cup \{\{\emptyset\}\}\} \cup \{\{\emptyset\} \cup \{\{\emptyset\}\} \cup \{\{\emptyset\} \cup \{\{\emptyset\}\}\}\} \dots$$

Que podemos simplificar:

$$\{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\}, \dots$$

Se renomearmos o conjunto $\{\emptyset\}$ como 1, temos por sucessor de 1, $s(1) = \{\emptyset, \{\emptyset\}\} = 2$, por sucessor 2, $s(2) = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\} = 3$, $s(3) = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}, \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}\} = 4, \dots$, obtemos assim o conjunto $\{1, 2, 3, 4, \dots\}$, onde cada número é o sucessor do anterior elemento, excepto 1 que assumimos estar presente. Naturalmente, para esta notação representamos a função sucessor por $s(n) = 1 + n$. Podendo esta descrição ser resumida dizendo que o conjunto dos números naturais pode ser obtido dos seguintes axiomas, conhecidos por axiomas de Peano.

1. $1 \in \mathbb{N}$,
2. Se $n \in \mathbb{N}$, então $s(n) = 1 + n \in \mathbb{N}$
3. Para todo o conjunto $S \subseteq \mathbb{N}$ que satisfaça as propriedades
 - (a) $1 \in S$, e

(b) se $n \in S$, então $s(n) \in S$

tem-se $S = \mathbb{N}$.

A propriedade 3 é conhecida por propriedade minimal, e afirma que o conjunto mais pequeno que satisfaz (a) e (b) é o conjunto dos números naturais.

A propriedade 3 é também a base do *Princípio de Indução Matemática*, frequentemente aplicado como método de demonstração. Para facilitar a sua aplicação é usual reescrever-lo, assumindo a forma:

Proposição 6.4.1 (Princípio de Indução Matemática I). *Para $P(n)$ um predicado definido no conjunto dos números naturais. Se*

1. $P(1)$ é verdadeira, e
2. $P(n) \Rightarrow P(n + 1)$, para todo o $n \in \mathbb{N}$,

então a proposição $\forall n \in \mathbb{N}, P(n)$ é verdadeira

Este resultado garante que para $S = \{n \in \mathbb{N} : P(n)\}$, se o primeiro natural está em S , e caso o predicado $P(n)$ seja hereditário (se $n \in S$ então $n + 1 \in S$), podemos concluir que $S = \mathbb{N}$.

Em Matemática muitos resultados são definidos tendo por base predicados $P(n)$ verdadeiros para todo o número natural n . Por exemplo, $1+2+\dots+n = n(n+1)/2$, para todo o natural n ou $\forall n \in \mathbb{N}, n < 2^n$. Este tipo de resultados podem ser demonstrados com o recurso à indução matemática. Apresentamos de seguida alguns exemplos onde se aplica esta técnica de demonstração.

Exemplo 6.4.2. Mostremos que $\forall n \in \mathbb{N}, 1 + 2 + \dots + n = \frac{n(n+1)}{2}$.

A proposição é definida pelo predicado $P(n) : 1 + 2 + \dots + n = \frac{n(n+1)}{2}$

Primeiro Caso Para $n = 1$, a proposição $P(1) : 1 = \frac{1(1+1)}{2}$ é verdadeira.

Hereditariedade Para uma escolha arbitrária de $n \in \mathbb{N}$, assumindo por

Hipótese $P(n) : 1 + 2 + \dots + n = \frac{n(n+1)}{2}$, tentamos demonstrar a veracidade da

Tese $P(n + 1) : 1 + 2 + \dots + n + (n + 1) = \frac{(n+1)(n+2)}{2}$.

Como por hipótese $1 + 2 + \dots + n = \frac{n(n+1)}{2}$, tem-se

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

$$\text{como } \frac{n(n+1)}{2} + (n + 1) = \frac{n(n+1)+2(n+1)}{2} = \frac{(n+1)(n+2)}{2} \text{ vem}$$

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

Assim, $P(n) \Rightarrow P(n + 1)$.

Como $P(n)$ é válida no primeiro caso e hereditária, o princípio de indução matemática, garante que $P(n)$ é verdadeira para todo $n \in \mathbb{N}$.

Exemplo 6.4.3. Mostremos agora que $\forall n \in \mathbb{N}, 1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1$.

A proposição é definida pelo predicado $P(n) : 1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1$

Primeiro Caso Para $n = 1$, $P(1) : 1 = 2^1 - 1$ é verdadeira.

Hereditariedade Para uma escolha arbitrária de $n \in \mathbb{N}$, assumindo por

Hipótese $P(n) : 1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1$

Tese $P(n+1) : 1 + 2 + 4 + 8 + \cdots + 2^{n-1} + 2^n = 2^{n+1} - 1$.

Como por hipótese $1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1$, tem-se

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} + 2^n = 2^n - 1 + 2^n,$$

como $2^n - 1 + 2^n = 2 \times 2^n - 1 = 2^{n+1} - 1$, vem

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} + 2^n = 2^{n+1} - 1,$$

Assim, $P(n) \Rightarrow P(n+1)$.

Como $P(n)$ é válida no primeiro caso e hereditária, o princípio de indução matemática, garante que $P(n)$ é verdadeira para todo $n \in \mathbb{N}$.

Exemplo 6.4.4. $\forall n \in \mathbb{N}, n < 2^n$.

A proposição é definida pelo predicado $P(n) : n < 2^n$

Primeiro Caso Para $n = 1$, $P(1) : 1 < 2^1$ é verdadeira.

Hereditariedade Para uma escolha arbitrária de $n \in \mathbb{N}$, assumindo por

Hipótese $P(n) : n < 2^n$, demonstremos a veracidade da

Tese $P(n+1) : n+1 < 2^{n+1}$.

Note que, como por hipótese $n < 2^n$, temos $n+1 < 2^n + 1$, e uma vez que $1 < 2^n$ já que n é natural, resulta

$$n+1 < 2^n + 1 < 2^n + 2^n = 2 \times 2^n = 2^{n+1},$$

onde $n+1 < 2^{n+1}$, que descreve $P(n+1)$. Assim, $P(n) \Rightarrow P(n+1)$.

O princípio de indução matemática isto garante que $P(n)$ é verdadeira para todo $n \in \mathbb{N}$.

Em geral a definição indutiva de uma propriedade, ou mais precisamente de um conjunto P , assenta em três passos. O primeiro passo consiste em nomear os elementos base de P . O passo seguinte consiste na descrição dum conjunto de regras que permitem gerar os outros elementos a partir dos elementos base. Este passo é usualmente referido por passo indutivo. Por último, e muitas vezes omitido, deve garantir-se que P consista unicamente por elementos definidos ou gerados nos passos 1 e 2 anteriores.

A definição apresentada do conjunto dos números naturais \mathbb{N} segue esta metodologia. A definição que foi apresentada do conjunto das fórmulas bem formadas, no cálculo proposicional, também já seguia esta metodologia.

Exemplo 6.4.5. Seja P o conjunto definido induutivamente por:

1. $3 \in P$
2. Para $x, y \in P$, $x + y \in P$.
3. Só os elementos descritos e definidos nos passos 1 e 2 estão em P .

Assim definido $P = \{3, 6, 9, 12, 15, 17, \dots\}$ é formado por inteiros positivos múltiplos de 3.

Exemplo 6.4.6. O conjunto $P = \{2, 3, 4, \dots\}$ pode ser definido induutivamente por:

1. $2 \in P$, $3 \in P$,
2. Para $x, y \in P$, $x + y \in P$.
3. Só os elementos descritos e definidos nos passos 1 e 2 estão em P .

Exemplo 6.4.7. O conjunto das fórmulas bem formadas (fbf), da teoria dos conjuntos, usando \cap , \cup , e o complementar, é definido induutivamente por:

1. Os símbolos A, B, C, \dots são fbf's.
2. (a) Se X é uma fbf, também o é \bar{X} .
(b) Se X e Y são fbf, também o são $X \cup Y$ e $X \cap Y$.
3. Só os elementos descritos e definidos nos passos 1 e 2, são fbf.

De forma simples, o princípio de indução matemática pode ser adaptado a estruturas definidas induutivamente. Aqui apresentamos a sua adaptação para um conjunto de inteiros $A = \{x \in \mathbb{Z} : x \geq a\}$, para qualquer $a \in \mathbb{Z}$, que pode ser definido por recorrência através de:

1. $a \in A$,
2. Para $x \in A$, $x + 1 \in A$.
3. Só os elementos descritos e definidos em 1 e 2 estão em A .

Neste caso o princípio de indução matemática assume a forma:

Proposição 6.4.8 (Princípio de Indução Matemática II). *Para $P(n)$ um predicado definido no subconjunto dos números inteiros*

$$A = \{x \in \mathbb{Z} : x \geq a\},$$

para qualquer $a \in \mathbb{Z}$. Se

1. $P(a)$ é verdadeira, e
2. $P(x) \Rightarrow P(x + 1)$, para todo o $x \in A$,

então a proposição $\forall x \in A, P(x)$ é verdadeira.

Neste caso por exemplo:

Exemplo 6.4.9. Para mostrar que $n! > 2^n, \forall n \in \mathbb{N} \setminus \{1, 2, 3\}$.

A proposição é definida pelo predicado $P(n) : n! > 2^n$, de domínio $\mathbb{N} \setminus \{1, 2, 3\}$.

Primeiro Caso Para $n = 4$, a proposição $P(4) : 4! = 32 > 16 = 2^4$ é verdadeira.

Hereditariedade Para uma escolha arbitrária de $n \in \mathbb{N} \setminus \{1, 2, 3\}$, assumindo por

Hipótese $P(n) : n! > 2^n$, demonstremos a veracidade da

Tese $P(n+1) : (n+1)! > 2^{n+1}$.

Note que $(n+1)! = (n+1) \times n!$, e como por hipótese $n! > 2^n$, multiplicando ambos os membros por $n+1$, temos $(n+1) \times n! > (n+1) \times 2^n$, ou seja $(n+1)! > (n+1) \times 2^n$.

Já que neste exemplo $n+1 > 2$, uma vez que $n \geq 4$, podemos escrever

$$(n+1)! > (n+1) \times 2^n > 2 \times 2^n = 2^{n+1}.$$

Assim, $P(n) \Rightarrow P(n+1)$.

Isto garante que $P(n)$ é verdadeira, para todo $n \in \mathbb{N} \setminus \{1, 2, 3\}$.

6.5 Sucessões

Uma sucessão num conjunto A é uma aplicação $u : \mathbb{N} \rightarrow A$. Dada uma sucessão $u : \mathbb{N} \rightarrow A$, denotamos $u(n)$ por u_n a que chamamos o termo geral da sucessão, $u = (u_n)_{\{n \in \mathbb{N}\}}$. A u_1 chamamos o primeiro termo da sucessão, a u_2 o segundo, u_3 o terceiro, etc. Ao termo genérico da sucessão u_n também designamos de termo de ordem n .

Exemplo 6.5.1. A sucessão dos números pares $0, 2, 4, 6, 8, \dots$ é definida pela sucessão de termo geral $u_n = 2(n - 1)$.

Exemplo 6.5.2. A sucessão dos números pares $1, 2, 4, 8, 16, \dots$ é definida pela sucessão de termo geral $u_n = 2^{n-1}$.

Definição: 6.5.3 (Progressão Aritmética). Uma progressão aritmética é uma sucessão de números reais $u : \mathbb{N} \rightarrow \mathbb{R}$ em que qualquer termo, depois do primeiro, se obtém do termo anterior somando-lhe uma constante.

A sucessão dos números pares $0, 2, 4, 6, 8, \dots$ é uma progressão aritmética de termo geral $u_n = 2(n - 2)$, já que $u_{n+1} - u_n = 2$ (constante).

Uma progressão aritmética, $(u_n)_{\mathbb{N}}$, fica definida se conhecermos dois números, o primeiro termo, u_1 , e a razão r , já que assim fica definido

$$u_1 = a, \quad u_2 = u_1 + r = a + r, \quad u_3 = u_2 + r = a + 2r, \quad u_4 = u_3 + r = a + 3r, \dots, \quad u_n = a + (n-1)r, \dots,$$

Outro exemplo de progressão aritmética é a sucessão $1, 2, 3, 4, 5, \dots$, cuja soma dos primeiros n termos $s_n = 1 + 2 + 3 + 4 + 5 + \dots = \sum_{i=1}^n i$ é descrita pela expressão

$$1 + 2 + 3 + 4 + 5 + \dots = n \frac{1+n}{2}$$

De forma geral, é possível obter uma expressão simples para a soma dos n termos de uma progressão aritmética (u_n):

$$s_n = u_1 + u_2 + u_3 + \cdots + u_n = \sum_{i=1}^n u_i = n(u_1 + u_n)/2$$

Este resultado pode ser demonstrado por indução. Para isso seja (u_n) uma progressão aritmética de razão r , e considere-se o predicado $P(n) : u_1 + u_2 + u_3 + \cdots + u_n = n \frac{u_1 + u_n}{2}$.

Primeiro Caso Para $n = 1$, $P(1) : u_1 = 1 \times \frac{u_1 + u_1}{2}$ é verdadeira.

Hereditariedade Para uma escolha arbitrária de $n \in \mathbb{N}$,

Hipótese $P(n) : u_1 + u_2 + u_3 + \cdots + u_n = n \frac{u_1 + u_n}{2}$,

Tese $P(n+1) : u_1 + u_2 + u_3 + \cdots + u_n + u_{n+1} = (n+1) \frac{u_1 + u_{n+1}}{2}$.

Note que, como por hipótese $P(n) : u_1 + u_2 + u_3 + \cdots + u_n = n \frac{u_1 + u_n}{2}$, tem-se,

$$u_1 + u_2 + u_3 + \cdots + u_n + u_{n+1} = n \frac{u_1 + u_n}{2} + u_{n+1},$$

como a progressão é aritmética $u_{n+1} = u_1 + nr, u_n = u_1 + (n-1)r$, e assim

$$\begin{aligned} n \frac{u_1 + u_n}{2} + u_{n+1} &= n \frac{u_1 + u_1 + (n-1)r}{2} + u_1 + nr = \frac{2nu_1 + n(n-1)r + 2u_1 + 2nr}{2} = \\ &= \frac{u_1(2n+2) + n(n+1)r}{2} = (n+1) \frac{2u_1 + nr}{2} = (n+1) \frac{u_1 + u_1 + nr}{2} = (n+1) \frac{u_1 + u_{n+1}}{2} \end{aligned}$$

Garantindo isto que:

$$u_1 + u_2 + u_3 + \cdots + u_n + u_{n+1} = (n+1) \frac{u_1 + u_{n+1}}{2}.$$

Exercício 6.5.4. 1. Considere as sucessões $(a_n), (b_n)$, e (c_n) definidas por:

$$(a) \quad a_n = \frac{1+2n}{2}$$

$$(b) \quad b_n = n^2$$

$$(c) \quad c_1 = 2 \text{ e } c_{n-1} = 2n + 3 + c_n, \quad (n \geq 1).$$

Mostre que só uma das sucessões é uma progressão aritmética. Determine a soma dos 23 primeiros termos da progressão aritmética.

2. Determine a expressão do termo geral da progressão aritmética em que o quinto termo é 1 e a soma dos dez primeiros termos é 100.

Definição: 6.5.5 (Progressão Geométrica). Uma progressão geométrica é uma sucessão de números reais em que qualquer termo, depois do primeiro, se obtém do termo anterior multiplicando-o por uma constante, a razão.

A sucessão dos números pares $1, 2, 4, 8, 16, \dots$, é uma progressão geométrica já que $u_{n+1} = 2u_n$. Neste caso $u_{n+1}/u_n = 2$ é constante definindo a razão da progressão.

Uma progressão geométrica, $(u_n)_{\mathbb{N}}$, fica definida pelo primeiro termo, u_1 , e pela razão r . Já que

$$u_1 = a, \quad u_2 = u_1 \times r = ar, \quad u_3 = u_2 \times r = ar^2, \quad u_4 = u_3 \times r = ar^3, \dots, \quad u_n = ar^{n-1}, \dots,$$

A soma dos n primeiros termos da progressão geométrica $1, 2, 4, 8, 16, \dots$ é dada por:

$$1 + 2 + 4 + 8 + 16 + \dots + 2^{n-1} = 2^n - 1$$

De forma geral, por indução matemática, podemos mostrar que a expressão da soma dos n primeiros termos de uma progressão geométrica (u_n) , com primeiro termo $u_1 = a$ e razão r , é dada por:

$$s_n = \sum_{i=1}^n u_i = u_1 \frac{1 - r^n}{1 - r}.$$

Exercício 6.5.6. 1. Considere as sucessões (a_n) , (b_n) , e (c_n) definidas por:

- (a) $a_n = 2^{n^2}$
- (b) $b_n = \frac{3^{n-1}}{2^{2n-1}}$
- (c) $c_1 = 2$ e $c_{n-1} = 2nc_n$, ($n \geq 1$).

Mostre que só uma das sucessões é uma progressão geométrica. Determine a soma dos 13 primeiros termos da progressão aritmética.

2. Determine a expressão do termo geral da progressão geométrica em que o terceiro termo é 1 e o décimo primeiro termo é 16.

6.6 Recorrência

Uma sucessão de números reais está definida por recorrência quando é conhecido o primeiro termo, ou alguns dos primeiros termos, e é dada uma relação que permite obter um termo a partir dos anteriores.

Uma sucessão só está bem definida por recorrência quando é possível determinar, de forma única, qualquer termo da sucessão.

Exemplo 6.6.1. Considere-se a relação de recorrência $x_{n+2} = nx_{n+1} + (-1)^n x_n$.

A determinação, por exemplo, de x_5 conduz sucessivamente a:

1. para $n = 3$, $x_5 = 3x_4 + (-1)^3 x_3$,
2. para $n = 2$, $x_4 = 2x_3 + (-1)^2 x_2$, e
3. para $n = 1$, $x_3 = x_2 + (-1)^1 x_1$.

Conclui-se que para determinar x_5 é necessário conhecer x_2 e x_1 , o que significa que a relação de recorrência só por si não define uma sucessão. Para cada valor que se fixar a x_2 e x_1 , a relação define uma sucessão.

Tomando, por exemplo, $x_1 = 3$ e $x_2 = 4$, podemos definir uma sucessão (x_n) por:

$$\begin{cases} x_1 = 3 \\ x_2 = 4 \\ x_{n+2} = nx_{n+1} + (-1)^n x_n, \quad n \in \mathbb{N} \end{cases}$$

O cálculo de x_5 conduz, agora, a um valor único: $x_3 = x_2 - x_1 = 4 - 3 = 1$; $x_4 = 2x_3 + 2x_2 = 2 \times 1 + 4 = 6$; $x_5 = 3x_4 - x_3 = 3 \times 6 - 1 = 17$.

Interessa referir que, sendo uma sucessão definida por recorrência, não existe um processo geral para obter o seu termo geral. Na generalidade das situações não é mesmo conhecido o termo geral de sucessões definidas por recorrência.

Exercício 6.6.2. Determine a_4 e a_5 sendo (a_n) a sucessão tal que:

1. $a_1 = 1 \wedge a_2 = 0 \wedge a_{n+2} = na_{n+1} - n^2 a_n$
2. $a_1 = 2 \wedge a_{n+1} = \frac{(-1)^n n}{a_n}$

Exemplo 6.6.3. Toda a progressão aritmética (u_n) , cujo primeiro termo seja a e tenha razão r , pode ser definida por recorrência através de:

$$(u_n) : u_1 = a \wedge u_{n+1} = u_n + r, (n \in \mathbb{N})$$

A $u_1 = a$ chamamos de **condição inicial** ou **caso base** e a $u_{n+1} = u_n + r$ **relação de recorrência**.

Exemplo 6.6.4. Uma progressão geométrica (v_n) , cujo primeiro termo seja a e que tenha razão r , pode ser definida por recorrência através de:

$$(v_n) : v_1 = a \wedge v_{n+1} = v_n \times r, (n \in \mathbb{N})$$

Neste caso a **condição inicial** da definição é $v_1 = a$. A relação de recorrência que define a sucessão é $v_{n+1} = v_n \times r$.

Exercício 6.6.5. Considere a relação de recorrência $u_{n+2} = 3u_{n+1} - 2u_n$.

1. Averigue se as sucessões seguintes satisfazem a relação:

- (a) $u_n = 0$
- (b) $u_n = 3$
- (c) $u_n = 2^n$
- (d) $u_n = n2^n$

2. Mostre que da relação se conclui que

- (a) $u_{n+1} = 3u_n - 4u_{n-2}$
- (b) $u_{n+2} = 4u_{n+1} - 16u_{n-1} + 16u_{n-2}$

Exercício 6.6.6. Considere a sucessão

$$(a_n) : a_1 = 1/2 \wedge a_2 = -1/4 \wedge a_{n+2} = -\frac{1}{4}a_n, (n \in \mathbb{N}).$$

1. Mostre que a sucessão (u_n) de termo geral $u_n = a_{2n}$ é uma progressão geométrica.
2. Mostre que a sucessão (v_n) de termo geral $v_n = a_{2n-1}$ é uma progressão geométrica.
3. Obtenha uma expressão para $s_{2n} = a_1 + a_2 + a_3 + \dots + a_{2n-1} + a_{2n}$.

6.7 EXERCÍCIOS DE REVISÃO

Exercício 6.7.1. Desenhe os diagramas de Hasse dos conjuntos abaixo relativamente à relação de ordem parcial “divide” e indique se é uma ordem total.

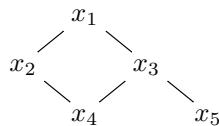
$$\{2, 6, 24\} \quad \{3, 5, 15\} \quad \{1, 2, 3, 6, 12\} \quad \{2, 4, 8, 16\} \quad \{3, 9, 27, 54\}$$

Exercício 6.7.2. Se R é uma ordem parcial em X e $A \subseteq X$, mostre que $R \cap (A \times A)$ é uma ordem parcial em A .

Exercício 6.7.3. Apresente um exemplo de um conjunto X , tal que $(\mathcal{P}(X), \subseteq)$ é um conjunto totalmente ordenado.

Exercício 6.7.4. Apresente um exemplo de uma relação que seja simultaneamente uma ordem parcial e uma relação de equivalência num conjunto X .

Exercício 6.7.5. A figura abaixo representa o diagrama de Hasse para uma ordem parcial (P, R) , onde $P = \{x_1, x_2, \dots, x_5\}$.

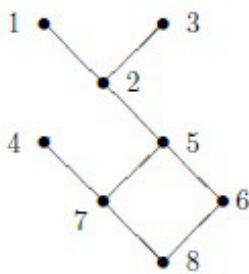


Determine o valor de verdade das proposições: x_1Rx_2 , x_4Rx_1 , x_3Rx_5 , x_2Rx_5 , x_1Rx_1 , x_2Rx_3 , e x_4Rx_5 . Caso exista, determine o maior elemento e o menor elemento em P . Determine os majorantes e minorantes de $\{x_2, x_3, x_4\}$, $\{x_3, x_4, x_5\}$ e $\{x_1, x_2, x_3\}$. Caso existam determine também, para estes conjuntos, o menor dos majorantes e o maior dos minorantes.

Exercício 6.7.6. Considere os seguintes conjuntos e as relações neles definidas. Determine quais as relações que são de ordem parcial.

1. $A = \{3, 5, 6, 10, 15, 18, 20\}$, $R \subset A \times A$, tal que $(x, y) \in R$ se e só se x é múltiplo de y .
2. $R \subset \mathbb{N} \times \mathbb{N}$, tal que $xy \% 3 = 0$
3. $A = \{1, 2, 3, 4, 5\}$, $R \subset A \times A$, tal que $(x, y) \in R$ se e só se x divide y .

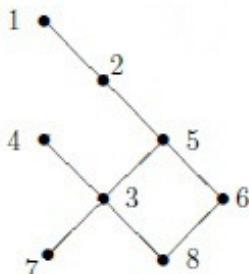
Exercício 6.7.7. Considere o conjunto $X = \{1, 2, 3, 4, 5, 6, 7, 8\}$ parcialmente ordenado de acordo com o seguinte diagrama de Hasse.



Considere ainda os subconjuntos $A = \{4, 5, 7\}$ e $B = \{1, 2, 4, 7\}$. Determine caso existam:

1. Os elementos maximais, minimais, máximo e mínimo de X .
2. Os majorantes, minorantes, supremo e ínfimo de A e B .

Exercício 6.7.8. Considere o conjunto $X = \{1, 2, 3, 4, 5, 6, 7, 8\}$ parcialmente ordenado de acordo com o seguinte diagrama de Hasse.



Considere ainda os subconjuntos $A = \{4, 3, 7\}$ e $B = \{2, 3, 4, 5\}$. Determine caso existam:

1. Os elementos maximais, minimais, máximo e mínimo de X .
2. Os majorantes, minorantes, supremo e ínfimo de A e B .

Exercício 6.7.9. Considere em \mathbb{N} a relação $|$ definida por

$$x|y \text{ se e só se } \exists k \in \mathbb{N} : y = kx$$

1. Mostre que $(\mathbb{N}, |)$ é um CPO mas não um conjunto totalmente ordenado.
2. Dados $a, b \in \mathbb{N}$, determine, caso existam, o supremo e o ínfimo de $\{a, b\}$.
3. Diga, justificando, se $(\mathbb{N}, |)$ tem elemento máximo ou elemento mínimo.
4. Seja $X = \{1, 2, 3, 4, 6, 8, 9, 12\}$.

- (a) Construa o diagrama de Hasse de $(X, |)$.
- (b) Indique, caso existam, os elementos maximais e os elementos minimais de X .

Exercício 6.7.10. Considere os conjuntos

$$A = \{\emptyset, \{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}, \{1, 2, 4\}, \{1, 2, 3, 4\}, \{4\}\}$$

$$\text{e } X = \{\{1\}, \{4\}, \{1, 2\}, \{1, 3\}\}.$$

1. Construa o diagrama de Hasse de (A, \subseteq) .
2. Indique, caso existam, os elementos maximais, minimais, supremo, ínfimo, máximo e mínimo de X .

Exercício 6.7.11. Considere no conjunto $A = \{1, 2, 3, 4\}$ as relações

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

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

1. Indique quais das relações são relações de ordem parcial. Justifique.
2. Construa os respectivos diagramas de Hasse.

Exercício 6.7.12. Considere a relação de recorrência $x_{n+2} = 3x_{n+1} - 2x_n$.

1. Averigue se as sucessões seguintes satisfazem a relação:

- (a) $x_n = 0$
- (b) $x_n = 3$
- (c) $x_n = 3^n$
- (d) $x_n = 1 + 2^n$

2. Mostre que da relação se conclui que $x_n = 7x_{n-2} + 3x_{n-1} - 6x_{n-2}$

Exercício 6.7.13. Determine a expressão simplificada da soma dos n primeiros termos das sucessões (a_n) , (b_n) e (c_n) :

1. $a_n = 1 + 3n$
2. $b_n = 2(1/2)^{2-n}$
3. $c_n = 2n + (-2)^n$

Exercício 6.7.14. Usando o princípio de indução matemática, mostre que:

1. $\sum_{p=1}^n (4 - 5p) = \frac{n(3-5n)}{2}, \forall n \in \mathbb{N};$
2. $\sum_{p=0}^n 2^p = 2^{n+1} - 1, \forall n \in \mathbb{N}$
3. $\sum_{p=1}^{2n} a(-r)^{p-1} = a \frac{1-r^{2n}}{1+r}, \forall n \in \mathbb{N}, (a, r \in \mathbb{N})$
4. $\frac{1}{1 \times 2} + \frac{1}{2 \times 3} + \cdots + \frac{1}{n(n+1)} = \frac{n}{n+1}, \forall n \in \mathbb{N}$
5. $1^2 + 3^2 + 5^2 + \cdots + (2n+1)^2 = \frac{(n+1)(2n+1)(2n+3)}{3}, n \geq 0$

6. $3^n < n!, n > 6$
7. $4^n > 5n^2, n \in \mathbb{N} \setminus \{1, 2\}$
8. $n! < n^n, n \in \mathbb{N} \setminus \{1\}$
9. $n^3 + 2n$ é divisível por 3, para n inteiro não negativo
10. $n^3 - n$ é divisível por 6, para n inteiro não negativo

Exercício 6.7.15. Considere as sucessões:

1. $(a_n) : a_1 = 1 \wedge a_{n+1} = \frac{n}{2n+1}a_n, n \in \mathbb{N}$
2. $(b_n) : b_1 = 2 \wedge b_2 = 1 \wedge b_{n+2} = 3b_{n+1} + 2b_n, n \in \mathbb{N}$
3. $(c_n) : c_1 = 2 \wedge c_{n+1} = \sqrt{4c_n - 3}, n \in \mathbb{N}$

Mostre por indução matemática:

1. $a_n \in]0, 1], n \in \mathbb{N}$
2. $b_{n+1} > b_n, n \in \mathbb{N} \setminus \{1\}$
3. $1 < c_n < 3, n \in \mathbb{N}$

Exercício 6.7.16. Considere a sucessão (u_n) :

$$\begin{cases} u_1 = 2 \\ u_{n+1} = \frac{u_n}{2} + \frac{2}{u_n}, \quad n \in \mathbb{N} \end{cases}$$

Mostre, por indução matemática, que $u_n \geq 2$ se $n \geq 2$.

Exercício 6.7.17. Determine $f(1), f(2), f(3)$, e $f(4)$ se $f(n)$ é definido recursivamente por $f(0) = 3$ e para $n \geq 1$:

1. $f(n+1) = f(n) + 2,$
2. $f(n+1) = 3f(n),$
3. $f(n+1) = 2^{f(n)}, e$
4. $f(n+1) = f(n)^2 + f(n) + 1.$

Exercício 6.7.18. Apresente funções recursivas para:

1. o conjunto dos inteiros pares,
2. o conjunto dos inteiros que têm resto 2 quando divididos por 3, e
3. o conjunto dos inteiros positivos divisíveis por 5.



Estruturas Algébricas

7.1 Semi-grupos

Quer a adição quer a multiplicação em \mathbb{N} , são operações que a cada par de números fazem corresponder um número natural. Diz-se, por tal facto, que a adição e a multiplicação são operações internas ou leis de composição interna em \mathbb{N} .

Definição: 7.1.1 (Lei de composição interna). *A operação \oplus é uma lei de composição interna ou operador binário no conjunto A , se e só se faz corresponder a cada par ordenado (a, b) em $A \times A$, um único elemento pertencente a A ,*

$$\oplus : A \times A \rightarrow A$$

Assim, não estamos em presença de uma operação interna sempre que para dois elementos dados, o resultado da operação não pertença ao mesmo conjunto.

Se \oplus é uma operação interna em A , diz-se que A é fechado relativamente a \oplus .

São as leis de composição interna que conferem aos conjuntos o que habitualmente chamamos de estruturas. Assim podemos desde já definir a mais simples das estruturas, o grupóide.

Um conjunto A diz-se um **grupóide** relativamente à operação \oplus , ou que a estrutura (A, \oplus) define um grupóide, se e só se \oplus é uma lei de composição interna em A .

Exemplo 7.1.2.

$$(\mathbb{Z}, +), (\mathbb{Z}, -), \text{ e } (\mathbb{R}, \times)$$

são exemplos de grupóides. $(\mathbb{Z}, +)$ é grupóide porque a adição está definida para números inteiros e a soma de dois inteiros é um inteiro. De forma idêntica, $(\mathbb{Z}, -)$ é um grupóide já que a diferença está definida para todo o par de inteiros, e o resultado é um único inteiro. (\mathbb{R}, \times) é um grupóide porque o produto está definido para todo o par de reais e o resultado é um real.

Exercício 7.1.3. Justifique que são grupóides:

1. $(\mathbb{N}, +)$
2. (\mathbb{R}, \times)
3. $(P(A), \cap)$, em que $P(A)$ é o conjunto das partes de A .

Exercício 7.1.4. Justifique que não são grupóides:

1. $(\mathbb{N}, -)$
2. $(\mathbb{R}, :)$, onde $:$ é a operação de divisão de reais
3. (B, \oplus) , em que $B = \{0, 1\}$ e a operação \oplus está definida pela tabela

\oplus	0	1
0	0	1
1	1	2

Numa estrutura (A, \oplus) diz-se que a operação \oplus é **associativa** se e só se

$$\forall a, b, c \in A : (a \oplus b) \oplus c = a \oplus (b \oplus c).$$

E diz-se que é **comutativo** se e só se

$$\forall a, b \in A : a \oplus b = b \oplus a.$$

Recorde que a adição quer a multiplicação são operações associativas e comutativas em \mathbb{N} .

Exemplo 7.1.5. Em \mathbb{Z} defina-se a operação \oplus por

$$a \oplus b = 3a - ab.$$

Calculemos por exemplo,

$$(1 \oplus 2) \oplus 3 = (3 - 2) \oplus 3 = 1 \oplus 3 = 3 - 3 = 0$$

e, em seguida

$$1 \oplus (2 \oplus 3) = 1 \oplus (6 - 6) = 1 \oplus 0 = 3 - 0 = 3.$$

Assim, $1, 2, 3 \in \mathbb{Z}$, e $(1 \oplus 2) \oplus 3 \neq 1 \oplus (2 \oplus 3)$.

Então, a operação \oplus não é associativa, pois,

$$\exists a, b, c \in \mathbb{Z} : (a \oplus b) \oplus c \neq a \oplus (b \oplus c).$$

Calculando agora

$$1 \oplus 2 = 3 - 2 = 1$$

e

$$2 \oplus 1 = 6 - 2 = 4.$$

Então

$$\exists a, b \in \mathbb{Z} : a \oplus b \neq b \oplus a$$

e a operação \oplus também não é comutativa.

Exemplo 7.1.6. Em $A = \{1, 2, 3\}$, defina-se agora o operador interno \oplus pela tabela

\oplus	1	2	3
1	2	1	3
2	1	3	1
3	3	1	1

Para averiguar se a operação \oplus é associativa temos de verificar se, para todos os elementos de A , se verifica

$$(a \oplus b) \oplus c = a \oplus (b \oplus c).$$

Temos de examinar 27 possibilidades. No entanto, se se encontrar um termo em que não se verifique, podemos logo concluir que \oplus não é associativa. Por exemplo:

$$(1 \oplus 2) \oplus 3 = 1 \oplus 3 = 3$$

e

$$1 \oplus (2 \oplus 3) = 1 \oplus 1 = 2.$$

Logo, \oplus não é associativa.

Para investigar se a operação é comutativa basta ver se existe simetria em relação à diagonal principal da tabela. Neste caso existe simetria, logo a operação \oplus é comutativa.

Se num conjunto A , está definida uma operação binária associativa, diz-se que (E, \oplus) é um semi-grupo.

Definição: 7.1.7 (Semi-grupo). Seja A um conjunto e \oplus uma operação. (A, \oplus) é um semi-grupo se \oplus é uma lei de composição interna em A e \oplus é associativa.

Exemplo 7.1.8. As estruturas algébricas

$$(\mathbb{N}, +), (\mathbb{N}, \times), \text{ e } (P(A), \cap)$$

são semi-grupos.

Caso num semi-grupo o operador seja comutativo, é usual designá-lo de **semi-grupo comutativo** ou **semi-grupo abeliano**.

No semi-grupo comutativo $(\mathbb{Z}, +)$, o número 0 tem a seguinte propriedade:

$$\forall a \in \mathbb{Z} : a + 0 = 0 + a = a.$$

Por esta razão se diz que zero é o elemento neutro para a adição em \mathbb{Z} .

Definição: 7.1.9. No semi-grupo (A, \oplus) diz-se que e é elemento neutro para a operação \oplus , se e só se

$$\forall a \in A : a + e = e + a = a.$$

Com base nesta definição podemos mostrar que:

Teorema 7.1.10. Se o semi-grupo (A, \oplus) tem elemento neutro, este é único.

Exemplo 7.1.11. No semi-grupo $(\{0, 1, 2\}, \oplus)$, a operação \oplus está definida pela tabela:

\oplus	0	1	2
0	2	0	1
1	0	1	2
2	1	2	0

O elemento neutro será o elemento a que corresponde uma linha e uma coluna iguais e pela mesma ordem dos elementos de suporte A . Logo o elemento neutro é 1.

Exercício 7.1.12. Verifique que \emptyset (conjunto vazio) é elemento neutro do semi-grupo $(P(A), \cup)$, que tem por suporte o conjunto das partes de A .

Exercício 7.1.13. Qual é o elemento neutro do semi-grupo $(P(A), \cap)$? Justifique.

Exercício 7.1.14. O semi-grupo $(\mathbb{Z}, -)$ tem elemento neutro? Justifique.

Exercício 7.1.15. Indique os elementos neutros, se existem, das operações definidas pelas tabelas:

1.

\oplus	1	2	3
1	2	3	1
2	3	1	2
3	1	2	3

2.

\oplus	1	2	3	4
1	2	3	1	4
2	4	1	2	3
3	1	2	3	4
4	1	4	4	2

Definição: 7.1.16 (Monóide). A estrutura (A, \oplus, e) diz-se um monóide se

1. (A, \oplus) é um semi-grupo e
2. e é elemento neutro da operação \oplus .

Exemplo 7.1.17. 1. $(\mathbb{N}, +)$ não tem elemento neutro.

2. $(\mathbb{Z}, +, 0)$ é um monóide porque $(\mathbb{Z}, +)$ é um semi-grupo e a adição tem por elemento neutro 0.
3. $(\mathbb{Z}, \times, 1)$ é um monóide porque (\mathbb{Z}, \times) é um semi-grupo e a multiplicação tem por elemento neutro 1.

7.2 Alfabetos e Linguagens

Definição: 7.2.1 (Alfabeto). *Um alfabeto é um conjunto finito e não-vazio de símbolos.*

Em geral usamos a letra grega Σ (Sigma) para designar um alfabeto. Por exemplo consideramos $\Sigma = \{0, 1\}$ como sendo o alfabeto binário.

Definição: 7.2.2 (Palavra). *Uma palavra sobre um alfabeto Σ é uma sequência finita de símbolos de Σ . O comprimento de uma palavra w em Σ é o número de ocorrências de símbolos em w , denotado por $|w|$. Em particular a palavra vazia, denotada de λ (lambda), é a palavra com zero ocorrências de símbolos.*

Por exemplo, 11011011 é uma palavra, de comprimento 8, sobre o alfabeto binário $\Sigma = \{0, 1\}$ e da mesma forma *python* é uma palavra, de comprimento 6, no alfabeto $\Sigma = \{h, n, o, p, t, y\}$.

O conjunto das palavras de comprimento k sobre Σ é dado por

$$\Sigma^k = \{a_1 \dots a_k : a_i \in \Sigma, \text{ para } i = 1, \dots, n\}$$

Em particular $\Sigma^0 = \{\lambda\}$. Por exemplo, se tomarmos $\Sigma = \{0, 1\}$, tem-se $\Sigma^0 = \{\lambda\}$, $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 10, 01, 11\}$, $\Sigma^3 = \{000, 010, 001, 011, 100, 110, 101, 111\}, \dots$

Como uma palavra não tem comprimento fixo, o conjunto das palavras sobre Σ , denotado por Σ^* , é o conjunto definido por:

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

Em geral recorremos à notação w^k , para $k \in \mathbb{N}$ e $w \in \Sigma^*$, para designar a palavra

$$w^k = \underbrace{ww\dots w}_{k \text{ vezes}}$$

Por exemplo, $0^3 = 000$ e $(10)^3 = 101010$. Onde se assume que $w^0 = \lambda$, por exemplo, $(101)^0 = \lambda$.

Definição: 7.2.3 (Concatenação). *Dadas duas palavras $w_1 = a_1 a_2 \dots a_n$ e $w_2 = b_1 b_2 \dots b_m$ sobre Σ , definimos a sua concatenação $w_1 \cdot w_2$ (também representada simplesmente por $w_1 w_2$) como sendo a palavra*

$$a_1 a_2 \dots a_n b_1 b_2 \dots b_m$$

onde os símbolos de w_2 aparecem após os símbolos de w_1 e pela mesma ordem.

Para as palavras sobre o alfabeto binário $\Sigma = \{0, 1\}$, temos por exemplo, para $w_1 = 100101$ e $w_2 = 0001001$, que $w_1 w_2 = 1001010001001$ e $w_2 w_1 = 0001001100101$ (note que a operação não é comutativa). Neste caso $w_1 \lambda = 100101$ e $\lambda w_1 = 100101$. Genericamente para toda a palavra $w_1, w_2 \in \Sigma^*$, temos

1. $|w_1 w_2| = |w_1| + |w_2|$, e
2. $\lambda w = w \lambda = \lambda$.

Neste sentido a palavra vazia é o elemento neutro para a concatenação de palavras. Garantido isto que, quando algebrizamos o conjunto das palavras com a concatenação a estrutura algébrica

$$(\Sigma^*, \cdot, \lambda),$$

define um monóide, já que para todo o alfabeto Σ e qualquer uma das suas palavras $w_1, w_2, w_3 \in \Sigma^*$, é válida a associatividade:

$$w_1(w_2w_3) = (w_1w_2)w_3.$$

Definição: 7.2.4 (Linguagem). *Uma linguagem sobre um alfabeto Σ é um subconjunto L de Σ^* .*

Por exemplo, $L = \{1, 11, 111\}$ é uma linguagem sobre $\{0, 1\}^*$, são exemplos de linguagens no mesmo alfabeto

$$\emptyset \text{ ou } \{w \in \{0, 1\}^* : \text{ que tem o mesmo número de 1's e 0's}\}.$$

Exemplo 7.2.5. *São exemplos de linguagens no alfabeto $\Sigma = \{a, b, c\}$:*

1. \emptyset o conjunto vazio
2. $L_1 = \{a^n : n \geq 0\}$
3. $L_2 = \{a^n : n > 0\}$
4. $L_3 = \{a, b, c\}$
5. $L_4 = \{aab, cab\}$

Definição: 7.2.6. *Sejam L_1 e L_2 linguagens sobre o alfabeto Σ . Definimos as seguintes operações cujo resultado são também linguagens sobre Σ .*

1. **União:** $L_1 \cup L_2 = \{w \in \Sigma^* : w \in L_1 \vee w \in L_2\}$
2. **Concatenação:** $L_1 \cdot L_2 = \{w_1w_2 \in \Sigma^* : w_1 \in L_1 \wedge w_2 \in L_2\}$
3. **Operador potência:** $L_1^k = \{w_1w_2 \dots w_k \in \Sigma^* : w_i \in L_1\}$
4. **Operador de fecho:** $L_1^* = \{w_1w_2 \dots w_k \in \Sigma^* : k \geq 0 \wedge w_i \in L_1\}$

Note que, L_1^* é o conjunto de todas as palavras que podem ser obtidas concatenando palavras de L_1 , tantas vezes quanto se queira. Por L_1^+ denotamos o conjunto das palavras que podem ser obtidas concatenando palavras de L_1 , excepto a palavra vazia.

$$L_1^+ = \{w_1w_2 \dots w_k \in \Sigma^* : k > 0 \wedge w_i \in L_1\}$$

Por exemplo, para $L_1 = \{10, 11\}$ e $L_2 = \{0, 111\}$, tem-se:

1. $L_1 \cup L_2 = \{0, 10, 11, 111\}$
2. $L_1 \cdot L_2 = \{100, 10111, 110, 11111\}$

3. $L_1^0 = \{\lambda\}$
4. $L_1^1 = \{10, 11\}$
5. $L_1^2 = \{1010, 1011, 1110, 1111\}$
6. $L_1^* = \{\lambda, 10, 11, 1011, 1110, 1111, 101010, 101011, \dots\}$
7. $L_1^+ = \{10, 11, 1011, 1110, 1111, 101010, 101011, \dots\}$

(concatenando zero palavras de L_1 , obtém-se a palavra vazia. Concatenando uma palavra de L_1 , obtém-se as palavras 10 e 11. Concatenando-se duas palavras de L_1 , obtém-se 1010, 1011, 1110 e 1111. Concatenando três palavras de L_1 , obtém-se ...).

Se o conjunto Σ estiver parcialmente ordenado, pode definir-se a ordem lexicográfica em Σ^* como sendo a mesma que é utilizada num dicionário, excepto que palavras mais curtas precedem palavras mais longas. Por exemplo, para a ordem usual $0 \leq 1$ em $\{0, 1\}$, é dada por:

$$\lambda \leq 0 \leq 1 \leq 00 \leq 01 \leq 10 \leq 11 \leq 000 \leq \dots$$

Formalmente diz-se que $u \leq v$, para $u, v \in \Sigma^*$, se:

1. $|u| < |v|$, ou
2. se $|u| = |v|$, então existe k tal que $u_i = v_i$, para $i = 1, \dots, k - 1$ e $u_k < v_k$.

7.3 Expressões regulares

As expressões regulares são uma forma de caracterizar a sintaxe de linguagens. Com base num alfabeto, definimos indutivamente o que se entende por uma expressão regular.

Definição: 7.3.1 (Expressão regular). *Uma expressão regular sobre um alfabeto Σ é uma expressão à qual associamos uma linguagem sobre Σ . Uma expressão regular pode somente ser obtida pelas seguintes regras:*

1. Se $a \in \Sigma$, então a é uma expressão regular associada à linguagem $\{a\}$,
2. λ e \emptyset são expressões regulares associadas às linguagens $\{\lambda\}$ e \emptyset , respectivamente,
3. Se r_1 e r_2 são expressões regulares associadas a linguagens L_1 e L_2 , respectivamente, então também são expressões regulares,
 - (a) $(r_1|r_2)$ associada à linguagem $L_1 \cup L_2$,
 - (b) $(r_1 \cdot r_2)$ associada à linguagem $L_1 \cdot L_2$,
 - (c) (r_1^*) associada à linguagem L_1^* , e
 - (d) (r_1^+) associada à linguagem L_1^+ .
4. Nada mais é uma expressão regular.

Para simplificar a notação, vamos utilizar as seguintes convenções quando escrevemos uma expressão regular:

1. sempre que estamos a concatenar duas expressões regulares, omitimos o símbolo \cdot ;
2. os operadores de fecho ($*$ e $^+$) tem precedência sobre os restantes operadores;
3. o operador de concatenação tem precedência sobre o operador de união.

Utilizando estas precedências, podemos eliminar muitos dos parêntesis. Por exemplo, para $\Sigma = \{0, 1\}$, a expressão:

$$001^*0|01$$

corresponde à expressão regular

$$((0 \cdot 0 \cdot (1^*) \cdot 0) | (0 \cdot 1))$$

são exemplo de palavras associadas a esta expressão regular por exemplo,

$$01, 000, 0010 \text{ e } 00110.$$

A linguagem associada a $001^*0|01$ é dada por

$$\{w \in \{0, 1\}^* : w = 01 \vee w = 001^k0 \text{ para algum } k \in \mathbb{N}\}.$$

Definição: 7.3.2 (Linguagens Regulares). As linguagens associadas a expressões regulares dizem-se *linguagens regulares*.

Exemplo 7.3.3. São exemplos de linguagens regulares no alfabeto $\Sigma = \{a, b, c\}$:

1. $L_1 = \{a^n : n \geq 0\}$ associada à expressão regular a^* ;
2. $L_2 = \{a^n : n > 0\}$ associada à expressão regular a^+ ;
3. $L_3 = \{a, b, c\}$ associada à expressão regular $a|b|c$;
4. $L_4 = \{aab, cab\}$ associada à expressão regular $aab|cab$;
5. $L_5 = \{a^n b^m c^p : n > 0, m \geq 0, p > 0, \}$ associada à expressão regular $a^+ b^* c^+$;
6. $L_6 = \{ab^m ac^p a : n > 0, m \geq 0, p > 0, \}$ associada à expressão regular $ab^* ac^+ a$;

Exercício 7.3.4. Determine exemplos de palavras que satisfazem a expressão regular:

$$(a|b|c)^+ d^* (a|c)(a|d)(a|b)$$

Notemos que nem todas as linguagens podem ser descritas por expressões regulares. Exemplo disso são as linguagens

1. $L = \{0^k 1^k : k \geq 0\}$, ou
2. $L = \{0^k 1^k 0^k : k \geq 0\}$.

Nestes casos, a igualdade entre sequências de 0's e 1's, inviabiliza a sua representação através de uma expressão regular.

Exercício 7.3.5. Para cada uma das seguintes expressões regulares, indique duas palavras que pertençam à linguagem que lhe está associada e duas palavras que não pertençam a esta linguagem. O alfabeto é $\Sigma = \{a, b\}$.

1. a^*b^*
2. $a^+b^*a^*$
3. $a(ba)^*b$
4. $a(ba)^+b^+$
5. $a^*|b^*$
6. $(aaa)^*$
7. $(a|b)^*a(a|b)^*b$
8. $aba|bab$
9. $(\lambda|a)b$
10. $(\lambda|a)b^+$
11. $(a|ba|bb)(a|b)^*$

Exercício 7.3.6. Apresente um exemplo de uma palavra. de comprimento mínimo, pertencente a cada uma das linguagens associadas às expressões regulares do exercício anterior.

Sejam r_1 , r_2 e r_3 expressões regulares. Então é válido afirmar:

1. $(r_1|r_2)|r_3 = r_1|(r_2|r_3)$, a operação $|$ é associativa,
2. $r_1|r_2 = r_2|r_1$, a operação $|$ é comutativa,
3. $r_1|r_1 = r_1$, a operação $|$ é idempotente,
4. $(r_1 \cdot r_2) \cdot r_3 = r_1 \cdot (r_2 \cdot r_3)$, a concatenação é associativa,
5. $r_1 \cdot \lambda = \lambda \cdot r_1 = r_1$, a palavra vazia é elemento neutro para a concatenação,
6. $r_1 \cdot (r_2|r_3) = r_1 \cdot r_2|r_1 \cdot r_3$, a concatenação é distributiva à esquerda relativamente à operação $|$,
7. $(r_1|r_2) \cdot r_3 = r_1 \cdot r_3|r_2 \cdot r_3$, a concatenação é distributiva à direita relativamente à operação $|$,
8. $r_1^+ = r_1 \cdot r_1^* = r_1^* \cdot r_1$
9. $r_1^* = \lambda|a^+$
10. $(r_1|\lambda)^+ = (r_1|\lambda)^* = r_1^*$

Exemplo 7.3.7. Considere-se que:

- A execução da tarefa A seguida da execução da tarefa B representa-se pela expressão regular AB .
- A execução da tarefa “if c então A senão B ” representa-se pela expressão regular $cA|\neg cB$.
- A execução da tarefa *skip* representa-se pela expressão regular λ (palavra vazia).

1. Determinemos expressões regulares que representem a execução das seguintes tarefas:

(a) enquanto c fazer A

A primeira acção a executar é o cálculo da condição c ; se esta for falsa não se executa mais nada; senão executa-se a acção A e depois voltamos ao mesmo ponto. Isto é,

$$\begin{aligned} \text{enquanto } c \text{ fazer } A &= \neg c \lambda | cA(\neg c \lambda | cA(\dots)) \\ &= \neg c | cA \neg c | cAcA \neg c | \dots \\ &= (cA)^* \neg c \end{aligned}$$

(b) repetir A até c

A primeira acção a executar é A seguida do cálculo da condição c ; se esta for verdadeira não se executa mais nada; senão voltamos ao mesmo ponto. Isto é,

$$\begin{aligned} \text{repetir } A \text{ até } c &= A(c \lambda | \neg c(A(c \lambda | \neg cA(\dots))) \\ &= Ac | A \neg cAc | A \neg cA \neg cAc | \dots \\ &= (A \neg c)^* Ac \end{aligned}$$

2. Usando as propriedades de expressões regulares (e o facto de $\neg \neg c = c$) mostremos que:

$$(\text{enquanto } c \text{ fazer } A) = (\text{Se } c \text{ então (repetir } A \text{ até } \neg c) \text{ senão skip})$$

Usando os resultados da alínea anterior temos:

$$\begin{aligned} \text{Se } c \text{ então (repetir } A \text{ até } \neg c) \text{ senão skip} &= c((A \neg c)^* | \neg c \lambda) \\ &= c(Ac)^* A(cA)^* cA \neg c | \neg c \\ &= ((cA)^* cA | \lambda) \neg c \\ &= ((cA)^+ | \lambda) \neg c \\ &= (cA)^* \neg c \\ &= \text{enquanto } c \text{ fazer } A \end{aligned}$$

7.4 Gramáticas

Por gramáticas entende-se um mecanismo que permite definir indutivamente as palavras dumha linguagem.

Definição: 7.4.1 (Gramática). Uma gramática é um tuplo

$$G = (\Sigma, V, P, S)$$

onde:

1. Σ é um alfabeto, cujos elementos designamos de conjunto de símbolos terminais;
2. V é um conjunto finito de símbolos não-terminais ou variáveis, diferentes de Σ , tal que $\Sigma \cap V = \emptyset$;
3. P é um conjunto finito de regras do tipo

$$w_1 \rightarrow w_2,$$

designadas de regras de produção, onde a primeira componente w_1 é uma palavra de $(\Sigma \cup V)^*$, e a segunda componente w_2 é uma palavra de $(\Sigma \cup V)^*$;

4. S é um símbolo não terminal, elemento de V , denominado símbolo inicial.

Os símbolos não terminais em Σ são aqueles que aparecem nas palavras geradas pela gramática. Os símbolos de V são usados como variáveis auxiliares na geração de palavras. As produções definem as condições de geração de palavras. A aplicação de uma produção é denominada derivação. Uma regra $w_1 \rightarrow w_2$ indica que w_1 pode ser substituída por w_2 sempre que ocorre w_1 . Enquanto houver símbolos não-terminais numa palavra, o processo de derivação deve continuar. O símbolo inicial é o símbolo usado para iniciar os processos de derivação. O processo termina quando deixar de existir símbolos não-terminais.

Vamos usar as produções para a derivação de novas palavras à custa de palavras dadas. Assim substitui-se a parte igual ao lado esquerdo da regra pela do lado direito da mesma.

Exemplo 7.4.2. Seja $G = (\{a, b\}, \{S, A, B\}, P, S)$ uma gramática onde o conjunto de regras de produção P é dado por:

1. $S \rightarrow AB$
2. $A \rightarrow a$
3. $B \rightarrow b$

Esta gramática permite gerar a linguagem com uma única palavra:

$$L = \{ab\}$$

que aparece associada às únicas duas cadeias de derivações:

$$S \Rightarrow_1 AB \Rightarrow_2 aB \Rightarrow_3 ab$$

$$S \Rightarrow_1 AB \Rightarrow_3 Ab \Rightarrow_2 ab$$

Onde por \Rightarrow_1 denotamos a aplicação da primeira produção, por \Rightarrow_2 e \Rightarrow_3 referenciamos a aplicação das produções 2 e 3. Neste caso chamamos a S , AB , Ab e aB palavras não terminais e a ab de palavra terminal.

Exemplo 7.4.3. Seja $G = (\{a, b, c\}, \{A, B\}, P, A)$ uma gramática onde o conjunto de regras de produção P é dado por:

1. $A \rightarrow aB$
2. $B \rightarrow bB$

3. $B \rightarrow c$

Neste caso a gramática tem por símbolo inicial A , e é uma sequência de derivações válidas:

$$A \Rightarrow_1 aB \Rightarrow_2 abB \Rightarrow_2 abbB \Rightarrow_3 abbc$$

Se aplicarmos a primeira produção n vezes, a segunda e uma vez a terceira, tem-se

$$A \Rightarrow_1 aB \Rightarrow_2 abB \Rightarrow_2 abbB \Rightarrow_2 \cdots \Rightarrow_2 ab^nB \Rightarrow_3 ab^nc$$

Podemos assim dizer que a linguagem gerada contém todas as palavras do tipo ab^nc , onde $n \geq 0$. A linguagem gerada pela gramática está assim associada à expressão regular ab^*c .

Exemplo 7.4.4. Seja $G = (\{a, b\}, \{S\}, P, S)$ uma gramática onde o conjunto de regras de produção P é dado por:

1. $S \rightarrow aSb$
2. $S \rightarrow ab$

Assim S é símbolo inicial e a única variável, a e b são símbolos terminais.

É nosso objectivo determinar a linguagem gerada pela gramática G , $L(G)$.

1. Ao aplicarmos a primeira produção $(n - 1)$ -vezes, seguida por uma aplicação da segunda produção tem-se

$$S \Rightarrow_1 aSb \Rightarrow_1 aaSbb \Rightarrow_1 \cdots \Rightarrow_1 a^{n-1}Sb^{n-1} \Rightarrow_1 a^n b^n.$$

2. Reparemos agora que sempre que se faz uso da primeira produção, o número de S 's se mantém igual a 1. Na utilização da segunda produção o número de S 's decresce de uma unidade. Logo a utilização da segunda produção elimina os S 's da sequência. Como ambas as produções têm um único S à esquerda, a ordem pela qual as produções podem ser aplicadas é $S \rightarrow aSb$ um certo número de vezes, seguido de uma aplicação de $S \rightarrow ab$.

Logo $L(G) = \{a^n b^n : n \geq 1\}$.

Neste exemplo, foi simples determinar as palavras que se podem derivar de S . Em geral pode ser difícil determinar a linguagem gerada por uma gramática dada.

Exemplo 7.4.5. Consideremos a gramática $G = (\{a, b, c\}, \{S, A, B\}, P, S)$ onde o conjunto de regras de produção P é dado por:

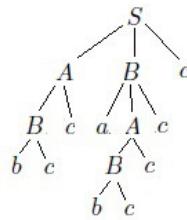
1. $S \rightarrow ABC$
2. $A \rightarrow aB$
3. $A \rightarrow Bc$
4. $B \rightarrow aAc$

$$5. B \rightarrow bc$$

A seguinte derivação em G

$$S \Rightarrow_1 ABc \Rightarrow_4 AaAcc \Rightarrow_3 BcaAcc \Rightarrow_5 bccaAcc \Rightarrow_3 bccaBccc \Rightarrow_5 bccabcccc$$

pode ser representada pela seguinte **árvore de derivação**.



Exercício 7.4.6. Considere uma gramática, de símbolo inicial S , com as seguintes produções:

1. $S \rightarrow aAb$
2. $A \rightarrow aA$
3. $A \rightarrow bA$
4. $A \rightarrow \lambda$

Construa árvores de derivação das palavras terminais abab e aabb.

Definição: 7.4.7. Dada uma gramática $G = (\Sigma, V, P, S)$ e duas palavras $w_0, w_1 \in (\Sigma \cup V)^*$, diz-se que w_1 deriva de w_0 por uma derivação, se existe uma produção $p \in P$ tal que, quando aplicada a w_0 , a transforma em w_1 , neste caso escrevemos

$$w_0 \Rightarrow_p w_1.$$

Caso exista uma sequência de produções p_1, p_2, \dots, p_n tal que

$$w_0 \Rightarrow_{p_1} w_1 \Rightarrow_{p_2} w_2 \Rightarrow_{p_3} \dots \Rightarrow_{p_n} w_n$$

escrevemos

$$w_0 \Rightarrow^* w_n$$

Assim para o exemplo 7.4.5 escrevemos

$$S \Rightarrow^* bccabcccc,$$

para indicar que do símbolo S se pode derivar $bccabcccc$, mediante a aplicação de uma sequência de produções.

Definição: 7.4.8. A linguagem $L(G)$ gerada pela gramática $G = (\Sigma, V, P, S)$ contém todas as palavras terminais geradas a partir do símbolo inicial S ,

$$L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}.$$

Definimos uma linguagem regular, como uma linguagem que pode ser descrita por uma expressão regular. Outra forma de caracterizar a sintaxe de linguagens regulares é através de gramáticas regulares.

Definição: 7.4.9 (Gramáticas regulares). *Uma gramática $G = (\Sigma, V, P, S)$ diz-se regular (mais precisamente regular à direita) quanto todas as produções são da forma:*

1. $A \rightarrow w$ ou
2. $A \rightarrow wB,$

em que $A, B \in V$ (símbolos não terminais) e $w \in \Sigma^$ (uma palavra terminal). Isto é, todas as produções têm do lado direito, no máximo um símbolo não terminal (e este é o último símbolo do lado direito).*

Exercício 7.4.10. Determine as produções em P de uma gramática regular (à direita) $G = (\Sigma, V, P, S)$, onde $\Sigma = \{a, b, c\}$ e $V = \{S, A, B, C, D\}$ tais que a linguagem gerada $L(G)$ seja descrita pelas expressões regulares abaixo:

1. a^*b^*
2. $(a|b)^*(a^*|c)$
3. aba^+c
4. ab^*
5. ab^*aba^+c
6. $(ab^*)^*a$

Genericamente, pode mostrar-se que:

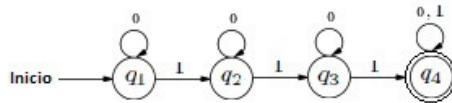
Teorema 7.4.11. Qualquer linguagem regular é gerada por uma gramática regular (no nosso caso, uma gramática regular à direita).

7.5 Autómatos

Para além das gramáticas regulares e expressões regulares, a sintaxe de uma linguagem regular, pode ser caracterizada por autómatos finitos.

Os autómatos finitos podem ser entendidos como a idealização de um computador que tem acesso apenas a uma quantidade limitada de memória. Computadores deste tipo podem ser utilizados para controlar um elevador, as portas automáticas de um banco, o dispositivo de injecção de um motor de combustão interna, etc.

Nesta secção, damos preferência a representações gráficas de autómatos finitos, por meio de diagramas. Limitamos no entanto a exposição a autómatos finitos deterministas. Abaixo apresentamos o exemplo dum diagrama que representa um autómato deste tipo. Estas representações gráficas são usualmente designadas de **diagramas de estados**.



Um autómato é definido por vários estados, representado por círculos, no exemplo os estados são q_1, q_2, q_3, q_4 . Um dos estados é identificado como estado inicial, no exemplo q_1 . Associado a um autómato está sempre um alfabeto Σ , no exemplo $\Sigma = \{0, 1\}$. Os símbolos do alfabeto são usados para rotular os arcos (setas) do autómato. Dado um estado tem de haver sempre uma transição para outro estado (possivelmente ele próprio) associada a cada elemento um dos símbolos em Σ . No nosso exemplo, para cada estado, existe sempre duas transições, uma associada ao símbolo 0 e a outra ao símbolo 1. Todas as possíveis transições devem estar definidas, podendo um arco ter associado mais do que um símbolo.

O input para um autómato de alfabeto Σ é uma palavra em Σ^* , que é aceite ou rejeitada pelo autómato. Supondo que o input, no exemplo é, $w = 0110$. Inicialmente a computação tem início no estado q_1 . O autómato lê o primeiro símbolo, que é 0. A regra de transição determina que, caso esteja em q_1 e se esteja a ler um 0, o estado do autómato deve continuar a ser q_1 . Lendo depois o próximo símbolo da palavra, que é um 1. A regra de transição determina que, caso esteja em q_1 e se esteja a ler um 1, o estado do autómato deve passar a ser q_2 . Lendo depois o próximo símbolo da palavra, que é outro 1. A regra de transição determina que, caso esteja em q_2 e se esteja a ler um 1, o estado do autómato passa a ser q_3 . O próximo símbolo é um 0, a regra de transição determina que, o estado do autómato deve continuar a ser q_3 . Como não existem mais símbolos na palavra para serem lidos a computação termina, no estado q_3 .

Uma palavra é **aceite** pelo autómato caso a computação termine num estado terminal. No diagrama de estado identificamos estados terminais por um duplo ciclo. No exemplo, o autómato só tem um estado terminal q_4 , como a computação da palavra $w = 0110$ não termina num estado terminal, diz-se que w não é **aceite** ou **reconhecida** pelo autómato.

Descrevemos abaixo a computação, agora da palavra $w = 011011$, através duma tabela. A primeira coluna representa o estado do autómato, num dado momento, e o símbolo da coluna da direita é o símbolo que está a ser lido:

Estado	Símbolo
q_1	0
q_1	1
q_2	1
q_3	0
q_3	1
q_4	1
q_5	Termina

A computação tem assim inicio no estado q_1 , a leitura progressiva de símbolos 0, 1, 1, 0, 1 e 1, faz com que o autómato assuma progressivamente os estados q_1, q_2, q_3, q_3, q_4 , e q_5 ,

terminando a computação no estado q_5 , por falta de mais símbolos para ler. Como q_5 é estado terminal a palavra 011011 diz-se reconhecida pelo autómato.

Podemos assim dizer que, as palavras reconhecidas por um autómato definem caminhos no diagrama de estados, que partindo do estado inicial, permitem chegar a um dos estados terminais.

Devemos notar ainda que a relação de transição num autómato finito pode ser sempre descrita por uma aplicação t . Identificando por E o conjunto de estados e por Σ o alfabeto, a relação de transição é uma aplicação:

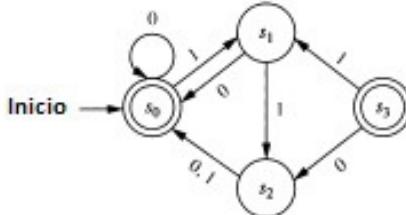
$$t : E \times \Sigma \rightarrow E.$$

No exemplo apresentado acima, esta aplicação pode ser descrita pela tabela abaixo:

t	0	1
q_1	q_1	q_2
q_2	q_2	q_3
q_3	q_3	q_4
q_4	q_4	q_4

Que indica que, quando no estado q_2 , por exemplo, caso esteja a ler um 0 passa para o estado q_2 , caso esteja a ler um 1 passa para o estado q_3 .

Exercício 7.5.1. Dado o diagrama de estados que descreve um autómato finito



1. Quais são os estados do autómato?
2. Qual é o alfabeto do autómato?
3. Qual é o estado inicial do autómato?
4. Quais são os estados terminais do autómato?
5. Qual é a sequência de estados para o input 0001011?
6. Será que o autómato aceita a palavra 01011?
7. Será que o autómato aceita a palavra λ (vazia)? E a palavra 010110?
8. Codifique a relação de transição do autómato através de uma tabela de dupla entrada.
9. O autómato pode terminar a sua computação no estado s_3 ? Justifique.

Quando se pretende ser mais formal representamos os autómatos através duma estrutura algébrica:

Definição: 7.5.2 (Autómato). *Um autómato A é definido por um tuplo*

$$A = (\Sigma, E, t, q, F)$$

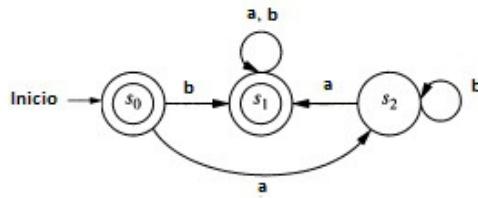
onde

1. Σ é um alfabeto,
2. E é um conjunto, designado de conjunto de estados,
3. t é uma função, designada de função de transição,
4. $q \in E$ é um estado, designado de estado inicial, e
5. $F \subseteq E$ é um subconjunto de estados, designado de estados terminais.

Exemplo 7.5.3. Seja $A = (\Sigma, E, t, s_0, F)$ o autómato finito de alfabeto $\Sigma = \{a, b\}$, estados $E = \{s_0, s_1, s_2\}$, de estado inicial s_0 , estados terminais $F = \{s_0, s_1\}$, tendo a função de transição $t : E \times \Sigma \rightarrow E$ descrita pela tabela de dupla entrada:

t	a	b
s_0	s_2	s_1
s_1	s_1	s_1
s_2	s_1	s_2

O autómato assim algebricamente definido tem por diagrama de estados:



Usando esta notação, podemos definir formalmente:

Definição: 7.5.4. Sejam $A = (\Sigma, E, t, q, F)$ um autómato finito e $w = w_1w_2 \dots w_k$ uma palavra sobre Σ . Então A aceita w se existe uma sequência de estados r_1, r_2, \dots, r_{k+1} tal que:

1. $r_1 = q$,
2. $t(r_i, w_i) = r_{i+1}$, para todo $i = 1, \dots, k$, e
3. $r_{k+1} \in F$

Formalizando o facto de que a iteração da função de transição sobre estados e símbolos de uma palavra, deve levar o autómato do estado inicial a um estado terminal, por forma a que a palavra seja reconhecida.

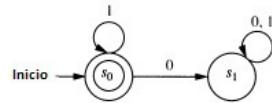
Definição: 7.5.5. *Dizemos que o autómato finito*

$$A = (\Sigma, E, t, q, F)$$

reconhece a linguagem $L(A)$ se

$$L(A) = \{w \in \Sigma^* : \text{o autómato } A \text{ aceita a palavra } w\}.$$

Exemplo 7.5.6. 1. No autómato

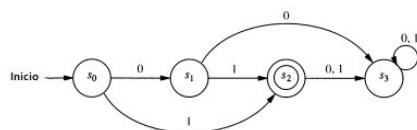


o único estado terminal é s_0 . As palavras que levam s_0 a s_0 , têm de ser formadas por zero ou mais 1's consecutivos. Assim,

$$L(A) = \{1^n : n \geq 0\}$$

a linguagem associada à expressão regular 1^* .

2. No autómato

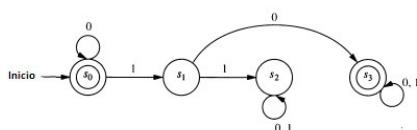


o único estado terminal é s_2 . As palavras que levam s_0 a s_2 , são 1 e 01. Assim,

$$L(A) = \{1, 01\}$$

a linguagem associada à expressão regular $1|01$.

3. O autómato



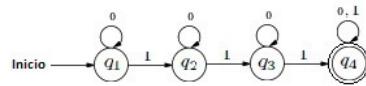
tem por estados terminais s_0 e s_3 . As palavras que levam s_0 a s_0 , são $\lambda, 0, 00, \dots$, ou seja qualquer palavra formada por zero ou mais 0's consecutivos. As palavras que levam s_0 a

a_3 , são palavras formadas por zero ou mais 0's consecutivos seguidos de 10, terminando em qualquer palavra

$$L(A) = \{0^n, 0^n 10(1|0)^* : n \geq 0\}$$

a linguagem associada à expressão regular $0^* | 0^* 01(0|1)^*$.

4. O autómato



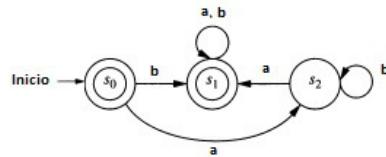
aceita a linguagem

$$L = \{w \in \{0,1\}^* : w \text{ tem pelo menos três } 1's\}$$

Note que, esta linguagem é regular e aparece associada à expressão regular

$$0^* 10^* 10^* 1(0|1)^*$$

5. O autómato



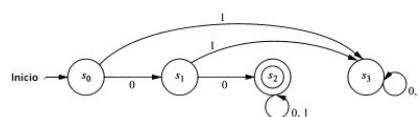
reconhece a linguagem regular associada à expressão

$$\lambda \mid b(a|b)^* \mid ab^*a(a|b)^*$$

Exemplo 7.5.7. Pretendemos agora construir autómatos finitos que reconheçam uma linguagem dada.

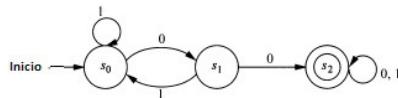
1. O conjunto de palavras binárias que têm início com dois 0's.

A palavra mais pequena, nesta linguagem é 00. Fixando s_0 como estado inicial, usamos um estado auxiliar s_1 , para o qual o autómato deve transitar, sempre que o primeiro símbolo é 0. Juntamos um estado terminal s_2 , atingido com a leitura de 00. Após a leitura de 00, caso exista símbolos que ocorram após 00, devem manter o autómato no estado terminal s_2 . Qualquer outra palavra não considerada deve fazer com que o autómato fique num estado não terminal s_3 .



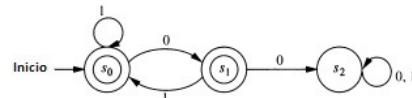
2. O conjunto das palavras contendo dois 0's seguidos.

A menor palavra nesta linguagem é 00. Escolhendo três estados s_0 , s_1 e s_2 , tomando s_0 por estado inicial e s_2 por estado final. O autómato deve transitar de s_0 para s_1 , com a leitura de 0, e deve transitar de s_1 para s_2 com a leitura do segundo 0. Caso a palavra tenha início com uma cadeia de 1's, mantemos o autómato no estado s_0 , na expectativa de que apareça 00. Caso após a leitura do primeiro 0 apareça um 1, devemos voltar ao estado s_0 , na expectativa de que apareça 00. Após a leitura de 00 a palavra é reconhecida, para qualquer outro símbolo, o autómato deve permanecer no estado terminal s_2 .



3. O conjunto das palavras que não contêm dois 0's seguidos.

Pela descrição do autómato que reconhece a linguagem anterior, sempre que uma palavra leva que o autómato termine nos estados s_0 e s_1 , não contém a subpalavra 00. Caso o autómato termine no estado s_2 , é porque a palavra contém dois 0's. Assim, a linguagem pretendida é reconhecida por um autómato com a mesma estrutura que o anterior, continuando a ter por estado inicial s_0 , mas tendo agora por estados terminais s_0 e s_1 .



Exercício 7.5.8. Cada uma das expressões regulares abaixo define uma linguagem de alfabeto $\{a, b, c\}$. Para cada uma delas apresente um exemplo de um autómato que reconheça a linguagem que lhe está associada.

1. $a|b$
2. $a^*|ab|ac$
3. $a^+|ab^*$
4. a^*bc^*b
5. $(a|b)^*c(a|b)$

Podemos mostrar que:

Teorema 7.5.9. Uma linguagem é regular se e só se pode ser reconhecida por um autómato finito.

Neste sentido, toda a linguagem associada a uma expressão regular ou gerada por uma gramática, é reconhecida por um autómato finito. O processo para a determinação

da estrutura que reconheça uma linguagem, pode não ser tarefa simples. Particularmente se o objectivo é a determinação directa de um autómato finito determinista. No exercício abaixo pode praticar, com linguagens regulares de “estrutura simples”.

Exercício 7.5.10. *Dê exemplos de autómatos que reconheçam as seguintes linguagens regulares (em todos os casos o alfabeto é $\{0, 1\}$)*

1. $A = \{w : w \text{ começa com } 1 \text{ e acaba com } 0\}$
2. $B = \{w : w \text{ começa por três } 1's\}$
3. $C = \{w : w \text{ contém a subpalavra } 0101\}$
4. $D = \{w : \text{ou } w \text{ começa com } 0 \text{ e } |w| \text{ é ímpar, ou } w \text{ começa com } 1 \text{ e } |w| \text{ é par}\}$
5. $E = \{w : w \text{ não contém a subpalavra } 0101\}$
6. $F = \{w : w \text{ não começa por três } 1's\}$
7. $G = \{w : |w| \leq 5\}$
8. $H = \{w : |w| > 5\}$

7.6 EXERCÍCIOS DE REVISÃO

Exercício 7.6.1. *Considere uma gramática, de símbolo inicial S , com as seguintes produções:*

- i. $S \rightarrow DSD$
- ii. $S \rightarrow B$
- iii. $B \rightarrow aCb$
- iv. $B \rightarrow bCa$
- v. $C \rightarrow DCD$
- vi. $C \rightarrow D$
- vii. $C \rightarrow \lambda$
- viii. $D \rightarrow a$
- ix. $D \rightarrow b$

1. *Dê exemplos de 3 palavras em $L(G)$.*
2. *Dê exemplos de 3 palavras que não pertencem a $L(G)$.*
3. *Quais das seguintes afirmações são verdadeiras ou falsas:*
 - (a) $C \Rightarrow aba$
 - (b) $C \Rightarrow^* aba$
 - (c) $C \Rightarrow C$
 - (d) $C \Rightarrow^* C$
 - (e) $DDD \Rightarrow^* aba$

$$(f) D \Rightarrow^* aba$$

$$(g) C \Rightarrow^* DD$$

Exercício 7.6.2. Para cada expressão regular abaixo, determine as produções em P , de uma gramática regular (à direita) $G = (\Sigma, V, P, S)$, tal que a expressão descreva a linguagem $L(G)$ gerada pela gramática.

$$1. a^*b^*cc$$

$$2. c^*(a^+|c)$$

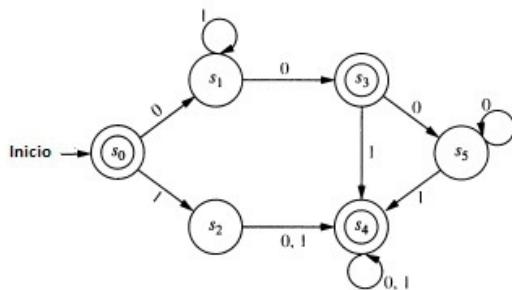
$$3. a^+cccc$$

$$4. cb^*c$$

$$5. acb^*a^*b$$

$$6. ab^*a$$

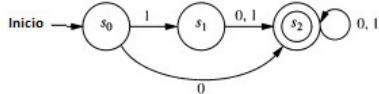
Exercício 7.6.3. Dado o diagrama de estados que descreve um autómato finito



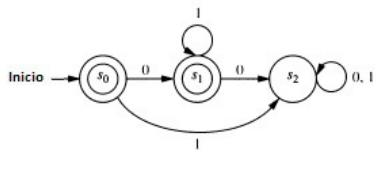
1. Quais são os estados do autómato?
2. Qual é o alfabeto do autómato?
3. Qual é o estado inicial do autómato?
4. Quais são os estados terminais do autómato?
5. Qual é a sequência de estados para o input 0001011?
6. Será que o autómato aceita a palavra 01011?
7. Será que o autómato aceita a palavra λ (vazia)? E a palavra 010110?
8. Codifique a relação de transição do autómato através de uma tabela de dupla entrada.
9. O autómato pode terminar a sua computação no estado s_1 ? Justifique.

Exercício 7.6.4. Determine as linguagens regulares reconhecidas por cada um dos autómatos abaixo:

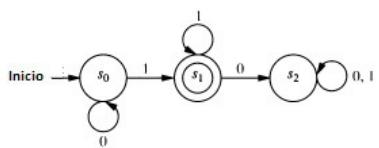
1.



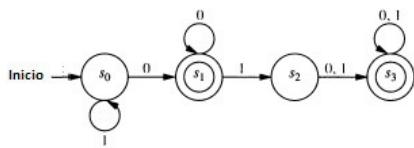
2.



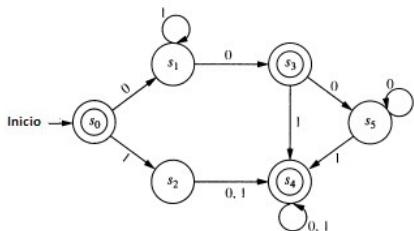
3.



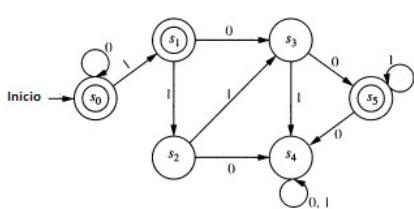
4.



5.



6.



Exercício 7.6.5. Dê exemplos de autômatos que reconheçam cada uma das seguintes linguagens regulares (em todos os casos o alfabeto é $\{0, 1\}$)

1. $A = \{w : w \text{ o segundo símbolo é } 1 \text{ e acaba com um } 1\}$
2. $B = \{w : w \text{ termina com três } 1's\}$
3. $C = \{w : w \text{ contém a subpalavra } 01 \text{ ou a subpalavra } 11\}$
4. $D = \{w : \text{ ou } w \text{ termina em } 0 \text{ se começa com } 0, \text{ ou } w \text{ termina com } 1 \text{ se começa com } 1\}$
5. $E = \{w : w \text{ não contém nem sequências } 01, \text{ nem } 11\}$
6. $H = \{w : \text{ se } |w| > 3 \text{ termina num } 1\}$

7.7 Python: Dicionários

Outra estrutura de dados útil em Python são os dicionários. Ao contrário das sequências, que têm os índices a variar em números, os dicionários são indexados por chaves. Um tuplo pode ser usado como chave se contém apenas strings, números, ou tuplos; os tuplos que contenham directa ou indirectamente objectos mutáveis, não pode ser usados como chave. Não se podem usar listas como chave.

A melhor forma de entender um dicionário é como um conjunto não ordenado de pares *chave:valor*, com o requisito de as chaves serem únicas. Um par de parêntesis {} define um dicionário vazio. Um dicionário pode ser definido em extensão através de uma lista de entidades *chave:valor* separadas por vírgulas, envoltas entre em parêntesis.

Exemplo de um dicionário:

```
idade = {'Carlos':20,'Manel':18,'Ana':22}
```

Aqui as chaves são as strings 'Carlos', 'Manel' e 'Ana', servindo de índices para os valores 20, 18 e 22, respectivamente.

A operação fundamental dum dicionário é o armazenamento de objectos com uma dada chave e a sua extracção dada a chave. É também possível apagar um par *chave:valor* com o comando *del*. Se armazenar um objecto com uma chave já existente, o valor antigo da chave é esquecido. São inválidas as tentativas de recuperar um objecto usando uma chave que não existe.

O método *key()* aplicado a um objecto dicionário devolve a lista de todas as chaves usadas num dicionário, ordenadas de forma arbitrária.

Um pequeno exemplo:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

O construtor *dict()* permite criar um dicionário a partir de uma sequência de pares chave-valor:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

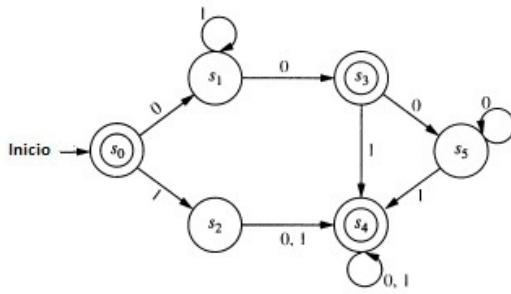
Adicionalmente pode-se definir dicionários em compreensão a partir de uma chave arbitrário sendo o valor definido por uma expressão:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Muitas vezes quando as chaves são *strings* simples, é mais fácil especificar pares usando chaves como argumentos:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

Exercício 7.7.1. *Dado o diagrama de estados que descreve um autómato finito*



1. Quais são os estados do autómato?
2. Qual é o alfabeto do autómato?
3. Qual é o estado inicial do autómato?
4. Quais são os estados terminais do autómato?
5. Qual é a sequência de estados para o input 0001011?
6. Será que o autómato aceita a palavra 01011?
7. Será que o autómato aceita a palavra λ (vazia)? E a palavra 010110?
8. Codifique a relação de transição do autómato através de uma tabela de dupla entrada.
9. O autómato pode terminar a sua computação no estado s_1 ? Justifique.
10. Codifique o autómato através de um dicionário.
11. Implemente uma função `auto(dict,str)->str`, tal que para um autómato descrito pelo dicionário G e para uma palavra w , no alfabeto do autómato, $auto(G, w)$ determina o estado final do autómato.
12. Implemente uma função `aceita(dict,set,str)->bool`, tal que para um autómato descrito pelo dicionário G , de estados terminais no conjunto T e para uma palavra w , $aceita(G, T, w)$ devolve True se e só se a palavra w é aceite pelo autómato.

8

PyGame

8.1 GUI vs. CLI

Os programas em Python escritos tendo apenas por base funções pré-definidas ("built-in functions") só "comunica" com recurso a instruções `print` e `raw_input`. Estes programas têm apenas a possibilidade de imprimir texto no ecrã e a introdução de dados a partir do teclado. Neste caso, dizemos que o programa tem uma interface em linha de comando, CLI, não possibilitando a visualização de elementos gráficos, e a sua manipulação com o rato. O Pygame fornece funcionalidade para ultrapassar estas restrições, permitindo a criação de interfaces gráficas com o utilizador, GUI, através de janelas cuja superfície pode ser usada para visualização de imagens a cores, "sensível" a eventos gerados pelo rato ou teclado.

8.2 O primeiro script em Python usando Pygame

O programa abaixo usa o Pygame e descreve como se pode criar uma janela de título "O PyGame é fixe!". Para isso crie um novo ficheiro no menu File do IDLE, seleccionando **new window**. Escreva o código abaixo no editor do IDLE e grave com o nome `fixepygame.py`. Depois corra o programa premindo F5 ou seleccionando **Run > Run Module** do menu no topo do editor de ficheiros.

Recorde que não deve escrever os números ou os pontos no início de cada linha.

```
1. # Exemplo dum programa em Python/Pygame
2. # Nome: fixepygame.py
3.
4. # Importa o modulo com o pygame
5. import pygame
6.
```

```
7.  
8. # Inicia o motor de jogo  
9. pygame.init()  
10.  
11. # Indica as dimensões da janela  
12. size=[700,500]  
13. screen=pygame.display.set_mode(size)  
14.  
15. # Escreve titulo na Janela  
16. pygame.display.set_caption('O PyGame é fixe!')  
17.  
18. #Para entrar em loop, até que a janela seja fechada.  
19. done=False  
20.  
21. # Usado para controlar a velocidade com que a janela é actualizada  
22. clock=pygame.time.Clock()  
23.  
24. while done==False:  
25.     for event in pygame.event.get(): # O utilizador actua na janela  
26.         if event.type == pygame.QUIT: # Se o utilizador escolheu fechar janela  
27.             done=True # o jogo deve terminar.  
28.  
29.  
30.     # Actualiza a janela para esta nova composição da cena.  
31.     pygame.display.flip()  
32.  
33.     # Limita a 20 o número de frames por segundo  
34.     clock.tick(20)  
35.  
36. # Para terminar o motor de jogo  
37. pygame.quit ()
```

O processo de carregamento e inicialização do pygame é muito simples. O pygame é uma coleção de módulos numa única biblioteca. Alguns destes módulos estão escritos em C, outros em Python. Alguns são opcionais, podendo não estar sempre presentes.

Esta é apenas uma introdução rápida ao pygame. Para uma descrição mais clara deve recorrer aos exemplos que se encontram no moodle.

8.3 Import - Carregar o módulo

Primeiro temos de importar a biblioteca, na maioria dos casos o mais simples é importar todos os módulos.

```
import pygame
```

```
from pygame.locals import *
```

A primeira linha importa todos os módulos de uma vez. A segunda linha é opcional define um conjunto de constantes e funções na tabela de símbolos globais do seu script.

Um ponto importante a não esquecer, é que alguns dos módulos são opcionais. Por exemplo, um destes módulos é *font*. Quando é executado import pygame, o pygame verifica se o módulo *font* está disponível. Caso este módulo esteja disponível, na plataforma, é importado como pygame.font. No caso de o módulo não estar disponível pygame.font referencia o object None. Tornando simples a verificação se o módulo existe na plataforma usada para a execução do jogo (por exemplo PSP, Android, Linux,...).

8.4 Init - Iniciar o motor de jogo

Antes de fazer seja o que for deve iniciar o motor de jogo. A forma mais usual de o fazer é através de:

```
pygame.init()
```

Iniciando todos os módulos por si. Nem todos os módulos necessitam ser iniciados, mas assim todos os que necessitam serão. Pode no entanto, de forma simples, iniciar individualmente cada módulo. Por exemplo:

```
pygame.font.init()
```

8.5 Quit - Terminar o motor de jogo

Os módulos inicializados são terminados usando o método quit(). Para a execução de um script não é obrigatório que este termine com pygame.quit(). Mas caso seja executado através do IDLE, se o script não termina os módulos do pygame, a janela gráfica não é fechada e o editor pode bloquear.

8.6 Módulos

O Pygame tem os seguintes módulos dos quais vamos usar um número muito restrito das suas funções:

1. **cdrom** - Controla o acesso à unidade de cdrom devices, permitindo a reprodução de audio.
2. **cursors** - Carrega imagens para o cursor do rato.
3. **display** - Controlo da janela gráfica.
4. **draw** - Permite desenhar elementos gráficos (linhas, círculos,...).
5. **event** - Gestão de eventos e controlo da lista de eventos.
6. **font** - Cria e faz o render de fontes para texto.

7. **image** - Carrega e grava imagens.
8. **joystick** - Controlo pelo joystick.
9. **key** - Controlo pelo teclado.
10. **mouse** - Controlo pelo rato.
11. **movie** - Reproduz ficheiro multimédia mpeg.
12. **sndarray** - Manipula sons com o módulo Numeric.
13. **surfarray** - Manipula imagens com o módulo Numeric.
14. **time** - Controlo do tempo.
15. **transform** - Transformações gráficas de mudança de escala, rotação, e flip.

8.6.1 pygame.draw

pygame.draw é o módulo do pygame para desenhar formas. Destacamos aqui alguns dos seus métodos

1. **pygame.draw.rect** - desenha rectângulos

```
pygame.draw.rect(Surface, color, Rect, width=0): return Rect
```

onde `Rect` descreve uma área rectangular. O argumento `width` é a largura do traço. Caso `width` seja zero o rectangular é preenchido.

```
import pygame
from random import *

pygame.init()
screen = pygame.display.set_mode((640, 480))
done = False

while done == False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    for count in range(10):
        random_color = (randint(0,255), randint(0,255), randint(0,255))
        random_pos = (randint(0,639), randint(0,479))
        random_size = (639-randint(random_pos[0],639), 479, randint (random_pos[1],479))
        pygame.draw.rect(screen, random_color, Rect(random_pos,random_size))
    pygame.display.flip()
pygame.quit()
```

2. **pygame.draw.polygon** - desenha uma figura com um número arbitrário de lados.

```
pygame.draw.polygon(Surface, color, pointlist, width=0): return Rect
```

Desenha um polígono numa superfície. O argumento `pointlist` descreve os vértices do polígono. O argumento `width` define a espessura do polígono. Caso `width` seja zero o polígono é preenchido.

3. pygame.draw.circle - desenha um círculo centrada num ponto

```
pygame.draw.circle(Surface, color, pos, radius, width=0): return Rect
```

O argumento `pos` define o centro do círculo, e `radius` o seu raio. O argumento `width` define a espessura do polígono. Caso `width` seja zero o polígono é preenchido.

```
import pygame
```

```
pygame.init()
screen = pygame.display.set_mode((640, 480))
points = []
done = False
while done == False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        if event.type == MOUSEBUTTONDOWN:
            points.append(event.pos)
            screen.fill((255,255,255))
        if len(points) >= 3:
            pygame.draw.polygon(screen, (0,255,0), points)
    for point in points:
        pygame.draw.circle(screen, (0,0,255), point, 5)
    pygame.display.flip()
pygame.quit()
```

4. pygame.draw.ellipse - desenha uma elipse dentro de um rectângulo

```
pygame.draw.ellipse(Surface, color, Rect, width=0): return Rect
```

O argumento `Rect` descreve o rectângulo na qual a elipse é desenhado. O argumento `width` define a espessura do polígono. Caso `width` seja zero o polígono é preenchido.

5. pygame.draw.arc - desenha a secção duma elipse

```
pygame.draw.arc(Surface, color, Rect, start_angle, stop_angle, width=1): r
```

O argumento `rect` descreve a área que a elipse preenche. O argumento `width` define a espessura do polígono. Caso `width` seja zero o polígono é preenchido.

```

import pygame
from random import *
from math import pi

pygame.init()
screen = pygame.display.set_mode((640, 480))
done = False
while done == False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    x, y = pygame.mouse.get_pos()
    angle = (x/639.)*pi*2.
    screen.fill((255,255,255))
    pygame.draw.arc(screen, (0,0,0), (0,0,639,479), 0, angle)
    pygame.display.flip()
pygame.quit()

```

6. pygame.draw.line - desenha um segmento

`pygame.draw.line(Surface, color, start_pos, end_pos, width=1): return Rect`

Os argumentos `start_pos` e `end_pos` definem o ponto inicial e final da linha.

7. pygame.draw.lines - desenha linhas múltiplas

`pygame.draw.lines(Surface, color, closed, pointlist, width=1): return Rect`

A lista `pointlist` define os pontos de referência. Se o argumento `closed` é `True` a linha é fechada, sendo desenhada um segmento entre o primeiro e último ponto de `pointlist`.

```

import pygame

pygame.init()
screen = pygame.display.set_mode((640, 480))
points = []

done = False
while done== False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        if event.type == pygame.MOUSEMOTION:
            points.append(event.pos)
    if len(points)>100:

```

```

    del points[0]
screen.fill((255, 255, 255))
if len(points)>1:
    pygame.draw.lines(screen, (0,255,0), False, points, 2)
pygame.display.flip()
pygame.quit()

```

8.6.2 pygame.display

Este módulo controla o ecrã gráfico. O Pygame tem uma única superfície de desenho que pode estar numa janela ou correr em full screen. Sempre que cria um ecrã gráfico este é tratado como uma superfície. As alterações não são imediatamente visíveis, para isso deve usar um dos métodos que permite actualizar o ecrã.

O ecrã original tem o topo esquerdo com coordenadas (0,0). Ambos os eixos crescem da esquerda para a direita e de cima para baixo.

O ecrã do pygame pode ser inicializado com diferentes características. Por defeito o ecrã é basicamente um buffer de frames. Podendo no entanto requerer módulos especiais, como aceleração por hardware ou suporte de OpenGL. Estes pedidos são controlados por flags na chamada a `pygame.display.set_mode`.

O pygame só pode ter um ecrã activo. A criação de outros com `pygame.display.set_mode` fecha o ecrã anterior. Uma vez criada uma superfície, as funções deste módulo actuam no ecrã existente. Se criar um novo ecrã, a superfície existente é transferida para o novo ecrã.

Após inicializar um ecrã, vários eventos são colocados na lista de eventos do pygame. `pygame.QUIT` é enviado para a lista quando o utilizador fecha o programa. A janela recebe um evento `pygame.ACTIVEEVENT` sempre que ganha ou perde focus de input. Caso se active a flag `pygame.RESIZABLE` flag, o evento `pygame.VIDEORESIZE` é enviado sempre que o utilizador redimensiona a janela.

Apresentam-se de seguida alguns dos seus métodos:

1. `pygame.display.set_mode` - inicia a janela ou ecrã

```
pygame.display.set_mode(resolution=(0,0), flags=0, depth=0): return Surface
```

O argumento `resolution` descreve a largura e altura da janela. O argumento (opcional) `flags` define um conjunto de opções adicionais. O argumento (opcional) `depth` descreve o número de bits usados para as cores.

As flags de controlo descrevem as características do ecrã pretendido. Existem várias escolhas, podendo combinar diferentes parametrizações, com o operador “|”. Caso use 0 ou não faça atribuições a `flags` cria um ecrã por defeito.

- `pygame.FULLSCREEN` cria fullscreen display
- `pygame.DOUBLEBUF` recomendado para HWSURFACE ou OPENGL
- `pygame.HWSURFACE` aceleração por hardware, apenas em FULLSCREEN
- `pygame.OPENGL` usa o render do opengl

- pygame.RESIZABLE janela de dimensão variável
- pygame.NOFRAME janela sem bordas ou controlos

2. pygame.display.flip - Faz o update de toda a superfície do ecrã.

```
pygame.display.flip() : return None
```

Se o ecrã está a usar as flags pygame.HWSURFACE ou pygame.DOUBLEBUF, espera por retrace vertical e faz o swap da superfície. Caso esteja a usar um tipo diferente de ecrã, faz simplesmente a actualização da superfície. Caso esteja a usar o ecrã em modo pygame.OPENGL esta função faz o gl buffer swap.

3. pygame.display.toggle_fullscreen - permuta entre fullscreen e janela.

```
pygame.display.toggleFullscreen() : return bool
```

Altera entre o modo janela e fullscreen. Esta função só funciona em unix usando um drive de video x11. Na maioria dos casos é exigido que se inicie um modo janela ou fullscreen usando as flags em pygame.display.set_mode.

4. pygame.display.set_icon - altera a imagem que o sistema usa para identificar a janela

```
pygame.display.setIcon(Surface) : return None
```

Permite identificar o icon a usar pelo sistema para representar a janela do jogo. Pode usar qualquer tipo de superfície, mas a maioria dos sistemas requer uma imagem pequena com 32x32. A maioria dos sistemas não permite que a imagem do icon seja alterada após a inicialização da janela.

5. pygame.display.set_caption - altera o titulo da janela

```
pygame.display.setCaption(title, iconTitle=None) : return None
```

Caso a janela tenha associado um título, esta função permite que este seja alterado.

8.7 Módulo do pygame para reproduzir e carregar sons

Este módulo contém uma classe para carregar objectos de tipo som e controlar a sua reprodução. O módulo mixer não existe para todas as plataformas que correm o pygame, sendo entendido como opcional. Caso pretenda correr o seu programa em diferentes arquitecturas deve testar se pygame.mixer está disponível, caso assim seja deve inicializá-lo antes de o usar.

O módulo mixer tem um número limite de canais para a reprodução de músicas. Na maioria dos caso quando o programa pede ao pygame que uma música deve ser reproduzida, e o pygame selecciona automaticamente um dos canais disponíveis. Por defeito existem 8 canais em simultâneo.

Todos os sons são reproduzidos em background threads. Quando inicia a reprodução dum som, o controlo é devolvido de imediato ao programa principal. Um som simples pode ser reproduzido diferentes vezes.

O mixer tem também um canal especial de streaming. Que é dedicado para music playback e é acessível através do módulo `pygame.mixer.music`.

O módulo mixer pode ser inicializado como qualquer outro módulo do pygame, mas tem de ter algumas cautelas. A função `pygame.mixer.init` que inicializa o mixer requer diferentes valores de controlo para controlar a playback rate e sample size. O pygame incializa estes valores, mas não consegue tratar o Sound resampling, neste sentido o mixer deve ser inicializado por forma a que a sua especificação se ajuste às cracterísticas do seu sistema.

Nota: Para obter menor latência, deve usar um buffer pequeno. Os valores por defeito são definidos por forma a eliminar os sons estridentes em alguns computadores. Este podem ser alterados executando a função `pygame.mixer.pre_init` antes de se chamar `pygame.mixer.init` ou `pygame.init`. Por exemplo:

```
pygame.mixer.pre_init(44100, -16, 2, 1024)
```

garante FREQ = 44100, o mesmo que um CD audio, BITSIZE = -16, unsigned 16 bit, CHANNELS = 2 (1 == mono, 2 == stereo), BUFFER = 1024, tamanho do audio buffer.

1. `pygame.mixer.init` - inicializa o módulo mixer:

```
pygame.mixer.init(frequency=22050, size=-16, channels=2, buffer=4096): ret
```

2. `pygame.mixer.pre_init` - pré-inicializa os parâmetros do mixer

```
pygame.mixer.pre_init(frequency=22050, size=-16, channels=2, buffersize=1024): ret
```

O argumento canal é usado para especificar se se usa som mono ou stereo. 1 indica som mono e 2 que o som é para ser reproduzido em stereo.

3. `pygame.mixer.stop` - Pára a reprodução de som em todos os canais.
4. `pygame.mixer.pause` - Faz um pausa na reprodução de sons.
5. `pygame.mixer.unpause` - Volta a reproduzir um som que esteja em pausa.
6. `pygame.mixer.get_busy` - Testa se algum som está a usar o mixer.
7. `pygame.mixer.Sound` - Cria o objecto som com base num ficheiro.
8. `pygame.mixer.Channel` - Cria um canal para a reprodução de sons.

Abaixo apresento exemplos de código:

```
import sys
import pygame
```

```
# global constants
FREQ = 44100    # O mesmo que o CD
BITSIZE = -16    # 16 bit sem sinal
CHANNELS = 2     # 1 == mono, 2 == stereo
BUFFER = 1024    # tamanho do audio buffer
FRAMERATE = 30   # frequência que deve testar se o playback terminou

def playsound(soundfile):
    """Play sound through default mixer channel in blocking manner.

    This will load the whole sound into memory before playback
    """
    sound = pygame.mixer.Sound(soundfile)
    clock = pygame.time.Clock()
    sound.play()
    while pygame.mixer.get_busy():
        clock.tick(FRAMERATE)

def playmusic(soundfile):
    """Stream music with mixer.music module in blocking manner.

    This will stream the sound from disk while playing.
    """
    clock = pygame.time.Clock()
    pygame.mixer.music.load(soundfile)
    pygame.mixer.music.play()
    while pygame.mixer.music.get_busy():
        clock.tick(FRAMERATE)

def playmusic2(soundfile):
    """Stream music with mixer.music module using the event module to wait
    until the playback has finished.

    This method doesn't use a busy/poll loop, but has the disadvantage that
    you neet to initialize the video module to use the event module.

    Also, interrupting the playback with Ctrl-C does not work :-(

    Change the call to 'playmusic' in the 'main' function to 'playmusic2'
    to use this method.
    """

```

```
pygame.init()

pygame.mixer.music.load(soundfile)
pygame.mixer.music.set_endevent(pygame.constants.USEREVENT)
pygame.event.set_allowed(pygame.constants.USEREVENT)
pygame.mixer.music.play()
pygame.event.wait()

pygame.mixer.init(FREQ, BITSIZE, CHANNELS, BUFFER)
```

8.8 pygame.image

Módulo para transferir imagens.

O módulo contém funções para carregar e gravar imagens, bem como para transformar superfícies para formatos usados noutras módulos.

Note que, não existe um classe imagem, uma imagem é carregada como uma superfície. A classe superfície pode ser manipulada (desenhar linhas, definir pixeis, capturar regiões, etc.).

Por defeito este módulo apenas carrega imagens BMP não comprimidas. Quando definido com suporte total de imagens, a função pygame.image.load permite carregar imagens em formato:

JPG, PNG, GIF (não animado), BMP, PCX, TGA (não comprimido), TIF, PBM

Permitindo gravar imagens nos seguintes formatos BMP,TGA,PNGe JPEG.

Neste módulo só vamos recorrer à função pygame.image.load

1. pygame.image.load - Para carregar imagens a partir de um ficheiro.

```
pygame.image.load(filename) : return Surface
pygame.image.load(fileobj, namehint="") : return Surface
```

Carrega uma imagem de um ficheiro. O argumento pode ser o nome de um ficheiro ou um objecto de tipo ficheiro.

O Pygame determina automaticamente o tipo da imagem (i.e. GIF ou bitmap) e cria uma nova superfície a partir dos dados. Em alguns casos é necessário saber a extensão do ficheiro (i.e., as imagens GIF têm extensões ".gif"). Se usar como referência à imagem um objecto de tipo ficheiro, pode ter a necessidade de definir o nome do ficheiro original como segundo argumento.

A superfície que é devolvida contém o mesmo formato de cores, e transparência alfa como no ficheiro que lhe deu origem. Em condições normais vai pretender chamar Surface.convert - para normalizar a estrutura que representa os pixeis.

Para transparências alfa, como nas imagens .png use o método convert_alpha() após serem carregadas por forma a definir pixeis transparentes.

Outra forma de definir a transparência numa imagem é através do método `Surface.set_colorkey`. Neste caso definimos a cor na superfície que deve ser assumida como transparente. Por exemplo

```
jogador = pygame.image.load("jogador1.gif").convert()
jogador.set_colorkey((255,255,255))
```

na superfície jogador é assumido que todos os pontos de cor (255,255,255) devem ser assumidos como transparentes.

Para facilitar a compatibilidade entre plataformas (Linux, Windows,...) deve recorrer a `os.path.join()`. Por exemplo

```
superficie = pygame.image.load(os.path.join('data', 'blabla.png'))
```

Embora o carregamento de uma imagem permita a definição de uma superfície, por vezes tem-se a necessidade de definir superfícies genéricas. Geralmente, com o propósito de processar imagens ou para a criação de formas no programa. Por exemplo

```
superficie = pygame.Surface((256, 256))
```

define uma superfície genérica de 256x256 de cor preta. Nestas superfícies podemos compor imagens.

```
import pygame
from random import randint

pygame.init()
screen = pygame.display.set_mode((640, 480), 0, 32)
done = False

while done == False :
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    superficie = pygame.Surface((256, 256))
    for n in range(100):
        rand_col = (randint(0, 255), randint(0, 255), randint(0, 255))
        rand_pos = (randint(0, 639), randint(0, 479))
        superficie.set_at(rand_pos, rand_col)
    screen.blit(superficie, (0,0))
    pygame.display.flip()
pygame.quit()
```

No script anterior é criada para cada frame uma superfície preenchida com pontos de cor aleatória. Note que, aqui usamos para localizar os pontos o método Surface.set_at, que tem por argumentos as coordenadas (x,y) de cada ponto e a correspondente cor. Cada superfície gerada é colado na superfície através do método usual Surface.blit.

Da execução do script anterior pode notar a pouca eficiência do processo. Esse facto tem a ver com a necessidade de proteger a superfície a quando do desenho de cada ponto. Para acelerar o processo é conveniente que esta protecção seja feita pelo próprio script. Garantindo isso que, o motor de jogo não tenha de proteger e desproteger a superfície por si, a quando da alteração da superfície.

```
import pygame
from random import randint

pygame.init()
screen = pygame.display.set_mode((640, 480))

done = False

while done == False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    superficie = pygame.Surface((256, 256))
    superficie.lock()
    for n in range(10000):
        rand_col = (randint(0, 255), randint(0, 255), randint(0, 255))
        rand_pos = (randint(0, 639), randint(0, 479))
        superficie.set_at(rand_pos, rand_col)
    superficie.unlock()
    screen.blit(superficie, (0,0))
    pygame.display.flip()
pygame.quit()
```

O recíproco do método Surface.set_at é o método Surface.get_at, que permite determinar a cor dum pixel na superfície, que usamos para testar algumas colisões. Por exemplo:

```
superficie = pygame.Surface((256, 256))
superficie.fill((255,255,255))
cor_ponto=superficie.get_at((100,100))
```

Determina a cor do ponto de coordenadas (100,100), atribuindo neste caso a cor_ponto é (255,255,255,255), onde a última componente indica o nível do canal

alfa da cor (nível de transparência). Neste caso, sendo o alfa da cor 255, indica que ela é totalmente opaca.

Notemos que, as funções do módulo pygame.draw que usamos anteriormente podem ser aplicadas a qualquer superfície. Caso use Surface.lock, não se esqueça de, após alterar a superfície, fazer Surface.unlock.

8.9 pygame.mouse

Módulo do pygame que permite trabalhar com o rato. As funções descritas permitem obter o estado actual do rato, podendo alterar o cursor do sistema para o rato.

Quando um ecrã é criado, a fila de eventos começa a receber os eventos gerados pelo rato. Os botões do rato geram eventos pygame.MOUSEBUTTONDOWN e pygame.MOUSEBUTTONUP sempre que são premidos ou soltos. Estes eventos contêm um atributo que permite distinguir qual o botão que foi premido. A roda do rato gera um evento pygame.MOUSEBUTTONDOWN sempre que for rodada. Sempre que o rato é movido este gera um evento pygame.MOUSEMOTION.

1. pygame.mouse.get_pressed - Devolve o estado dos botões do rato.

```
pygame.mouse.get_pressed(): return (button1, button2, button3)
```

Devolve uma sequência de booleanos representando o estado de todos os botões do rato. Um valor True indica que o rato estava premido no momento em que a função foi chamada.

2. pygame.mouse.get_pos - Devolve a actual posição do cursor do rato.

```
pygame.mouse.get_pos(): return (x, y)
```

Devolve as coordenadas do cursor do rato. Estas coordenadas são relativas ao canto superior esquerdo do ecrã. Apesar de o cursor poder estar fora da janela, as suas coordenadas devolvidas por esta função são reescritas a coordenadas da janela.

3. pygame.mouse.get_rel - Quantidade de movimento

```
pygame.mouse.get_rel(): return (x, y)
```

Devolve a deslocação em *x* e em *y* desde a última chamada à função. Sendo a deslocação restrita aos vértices da janela.

4. pygame.mouse.set_pos - Localiza o cursor do rato

```
pygame.mouse.set_pos([x, y]): return None
```

Indica a posição do rato na janela. Se o cursor do rato está visível este salta para a nova posição.

5. `pygame.mouse.set_visible` - Esconde ou mostra o cursor

```
pygame.mouse.set_visible(bool): return bool
```

Se o argumento é True, o cursor do rato passa a estar visível. Devolvendo o estado anterior.

6. `pygame.mouse.get_focused` - Verifica se o cursor do rato está na janela:

```
pygame.mouse.get_focused(): return bool
```

Devolve True quando o pygame recebe eventos do rato (ou quando a janela está activa). Este método é útil para detectar se o rato está dentro dos limites da janela. Em contraste, quando o motor de jogo está em modo full-screen, este método devolve sempre True.

7. `pygame.mouse.set_cursor` - define a imagem para o cursor do rato

8. `pygame.mouse.get_cursor` - Recupera a imagem usada para o cursor do rato

Abaixo exemplificamos como pode alterar a imagem que representa o cursor do rato.

```
background_image_filename = 'sky.jpg'
mouse_image_filename = 'fno.png'
import pygame

pygame.init()
screen = pygame.display.set_mode((640, 480))
pygame.display.set_caption("Hello, World!")
background = pygame.image.load(background_image_filename).convert()
mouse_cursor = pygame.image.load(mouse_image_filename)
mouse_cursor.set_colorkey((255,255,255))
done = False
while done == False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
    screen.blit(background, (0,0))
    x, y = pygame.mouse.get_pos()
    x = x - mouse_cursor.get_width() / 2
    y = y - mouse_cursor.get_height() / 2
    screen.blit(mouse_cursor, (x, y))
    pygame.display.flip()
pygame.quit()
```

8.10 Uso do teclado

Os scripts apresentados tratam eventos QUIT, que é fundamental, a menos que queira janelas imortais! O pygame permite tratar outros eventos como o movimento do rato e teclas primidas.

Nos exemplos anteriores chamamos `pygame.event.get()` para aceder à lista dos eventos. Dando possibilidade a tratar através do ciclo `for` todos os eventos à sua medida.

Os objectos na lista `pygame.event.get()` contêm atributos próprios que permitem descrever-los. A única coisa comum aos eventos é o seu tipo. Na lista abaixo apresentamos os diferentes tipos de eventos.

Evento	Objectivos	Parâmetros
QUIT	Fecho de janela	
ACTIVEEVENT	Janela está activa ou escondida	gain, state
KEYDOWN	A tecla foi primida	unicode, key, mod
KEYUP	A tecla foi largada	pos, button
MOUSEMOTION	O rato foi movido	pos, rel, buttons
MOUSEBUTTONDOWN	Um botão do rato foi primido	pos, button
MOUSEBUTTONUP	Um botão foi largado	pos, button
VIDORERESIZE	A janela foi redimensionada	size, w, h
USEREVENT	Um evento do utilizador	code

O teclado gera eventos KEYDOWN quando uma tecla é primida e KEYUP quando uma tecla é largada. Abaixo tratamos estes eventos:

```
fno_image_filename = 'fno.gif'

import pygame

pygame.init()
screen = pygame.display.set_mode((640, 480), 0, 32)
fno = pygame.image.load(fno_image_filename).convert()
fno.set_colorkey((255,255,255))
x, y = 0, 0
move_x, move_y = 0, 0
done = False

while done==False:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_LEFT:
                move_x = -1
            elif event.key == pygame.K_RIGHT:
                move_x = 1
            elif event.key == pygame.K_UP:
                move_y = -1
            elif event.key == pygame.K_DOWN:
                move_y = 1
    screen.blit(fno, (x, y))
    x += move_x
    y += move_y
    pygame.display.update()
```

```

        move_x = +1
    elif event.key == pygame.K_UP:
        move_y = -1
    elif event.key == pygame.K_DOWN:
        move_y = +1
    elif event.type == pygame.KEYUP:
        if event.key == pygame.K_LEFT:
            move_x = 0
        elif event.key == pygame.K_RIGHT:
            move_x = 0
        elif event.key == pygame.K_UP:
            move_y = 0
        elif event.key == pygame.K_DOWN:
            move_y = 0
    x= x + move_x
    y= y + move_y
    screen.fill((0, 0, 0))
    screen.blit(fno, (x, y))
    pygame.display.flip()
pygame.quit()

```

No script carregamos uma imagem e iniciamos o pygame. O loop de tratamento de eventos processa os eventos do tipo pygame.QUIT, pygame.KEYDOWN e pygame.KEYUP. Os últimos dois são gerados pelo teclado e têm associados três atributos:

- key - é um número que representa a tecla que foi primida ou libertada. Cada tecla tem associada uma constante, cujo nome tem inicio em K_. Por exemplo a tecla A é identificada pela constante K_a, teclas como a de return e espaço são identificadas por K_RETURN e F_SPACE. Pode encontrar a lista completa das constantes de descrevem as teclas do seu teclado em www.pygame.org/docs/ref/key.html.
- mod - Representa teclas que são usadas em conjunção com outras, como Shift, Alt e Ctrl. Este tipo de teclas são identificadas por constantes cujo nome têm inicio em KMOD_, tais como KMOD_SHIFT, KMOD_ALT e KMOD_CRTL.
- unicode - Este descreve o valor unicode da tecla. Sendo produzido combinando uma tecla e um modificador, usado por exemplo para descrever caracteres acentuados na língua portuguesa.

No interior do tratamento do evento KEYDOWN verificamos quatro teclas. Se K_LEFT é primida, o valor de move_x passa a -1; Se K_RIGHT é primida, o valor de move_x passa a +1; tendo por objectivo, respectivamente, de deslocar um objecto para a esquerda ou para a direita. Se K_UP é primida, o valor de move_y passa a -1; Se K_DOWN é primida, o valor de move_y passa a +1; tendo por objectivo, respectivamente, de deslocar um objecto para cima ou para baixo.

Também é tratado evento KEYUP, já que pretendemos que o objecto páre quando deixamos de premir as teclas de direcção. O código é semelhante ao tratamento anterior. Aqui sempre que a tecla é largada os valores de move_x e move_y são postos a 0.

Exercício 8.10.1. Na janela gráfica, deve ficar um círculo com centro nos pontos que selecciona com o rato.

Exercício 8.10.2. Na janela gráfica, sempre que selecciona um ponto as suas coordenadas devem ser armazenadas. Use estes pontos para traçar uma linha poligonal. Note que, ao segundo ponto seleccionado, uma linha deve ser traçada do primeiro para o segundo. Ao terceiro ponto uma linha deve ser acrescentada do segundo ponto para o terceiro ponto seleccionado, e assim sucessivamente. Use o botão direito do rato para seleccionar.

Adicionalmente faça último arco traçado seja removido através do botão direito do rato.

Exercício 8.10.3. Inicie a janela gráfica com quatro quadrados desenhados. Sempre que selecione um ponto dentro dos quadrados este deve mudar de cor.

Exercício 8.10.4.

Inicie a janela gráfica com dez quadrados, distribuídos aleatoriamente, que numeramos de 0 a 9. Fixe uma relação binária R no conjunto

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}.$$

O programa deve ter o seguinte comportamento, sempre que se selecciona um ponto dentro de um destes quadrados: Se selecciona o quadrado i , todos os quadrados j , tais que iRj , devem mudar de cor.

Exercício 8.10.5. Inicie a janela gráfica com dez quadrados, distribuídos aleatoriamente, numerados de 0 a 9. Cada um deles é pintado com cores de uma conjunto de quatro cores $B = \{a, b, c, d\}$. Assumindo que em B se define a função $f = \{(a, b), (b, c), (c, d), (d, a)\}$, e fixando uma relação binária R no conjunto $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. O programa deve ter o seguinte comportamento, sempre que selecciona um ponto dentro destes quadrados: Se selecciona o quadrado i , todos os quadrados j , tais que iRj , devem mudar de cor, mas de tal forma que, se j tiver cor $k \in B$ deve passar a ter cor $f(k)$.

Exercício 8.10.6. Tendo por base o código *Aula1_pygame.py* e os elementos disponíveis em *aula3pygame.zip* resolva os seguintes problemas:

1. O invasor:

- (a) Inicie o pygame com uma janela gráfica 400x500.
- (b) Junte imagem de fundo (*sky.jpg*).
- (c) Desloque no topo da janela, da esquerda para a direita, a imagem de um OVNI (*fno.png*). Impõe o branco como a cor transparente.
- (d) Use a metade inferior da janela para deslocar a imagem do jogador, controlada pelo rato. Estas imagens estão em *play1.gif*, *play2.gif* e *play3.gif*, devendo a imagem

escolhida depender do movimento do rato. Devendo usar play1.gif quando o rato não está em movimento. Reservado play2.gif e play3.gif, respectivamente, para identificar as situações onde o rato se desloca da direita para a esquerda e da esquerda para a direita.

- (e) *O jogador dispara um círculo, no sentido ascendente, sempre que se usa o botão esquerdo do rato. Atenção: os projécteis que não estejam visíveis devem ser removidas da estrutura auxiliar.*
- (f) *Meta no ecrã um contador de pontos. Sempre que um projéctil intersecte o OVNI, deve incrementá-lo.*
- (g) *Sempre que se dispara ou sempre que um projéctil intersecta o OVNI devem ser emitidos sons diferentes.*

2. *O Xe-ataque:*

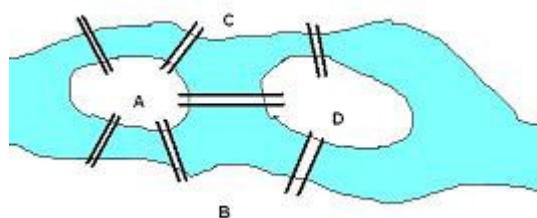
- (a) *Crie um tabuleiro de 17x17 com quadrados de 30x30 pontos.*
- (b) *Controle uma peça usando as teclas do cursor. Sempre que é pressionada uma tecla o jogador deve ser movido no tabuleiro uma posição na horizontal ou vertical. Sempre que usa uma tecla o jogador fica virado na direcção seleccionada, representada pela imagem jogador1.gif, jogador2.gif, jogador3.gif ou jogador4.gif.*
- (c) *No movimento, entre duas posições, o jogador deve alternar na posição média entre jogador0.gif e as imagens que definem a direcção.*

9

Teoria de grafos

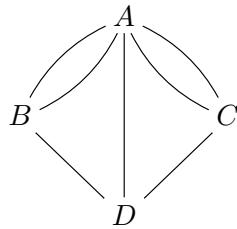
9.1 Introdução: Leonhard Euler e as sete pontes de Königsburg

Na antiga cidade Prussiana de Königsberg (actual Kaliningrad na Lituânia) sete pontes passavam o rio Pregel. Existia um jogo tradicional que envolvia estas pontes: É possível percorrer a cidade, passando por todas as pontes uma única vez, voltando ao ponto inicial da caminhada? Nunca ninguém em Königsburg conseguiu descobrir tal percurso, e não tinham desistido até 1736, altura em que o matemático suíço Leonhard Euler provou de uma vez por todas que não existe tal percurso. Euler argumentou da seguinte forma: Suponhamos que tal percurso existe, suponhamos que um transeunte inicia esse percurso. Se um observador estiver localizado na ilha A, vê o transeunte visitar a ilha um certo número de vezes. Quantas vezes o transeunte visita a ilha? Uma? Não, porque se o transeunte visitar a ilha uma vez, usa apenas duas das cinco pontes (uma para entrar e a outra para sair da ilha). Duas? Também não, porque duas visitas necessitam apenas de quatro pontes. Três vezes? Também não, porque neste caso necessitaríamos de seis pontes, e só temos disponíveis cinco. Podemos concluir que é impossível um percurso com as características dadas.



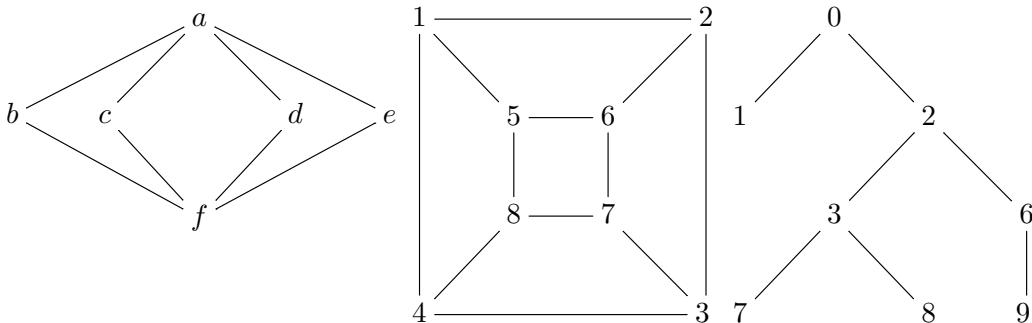
O problema das pontes de Königsburg por si só não é especialmente importante. No entanto os conceitos que Euler introduziu para resolver o problema, deram origem a um importante ramo da matemática chamado de Teoria de Grafos, que é o assunto tratado por neste capítulo.

Euler notou que o problema das pontes de Königsburg não têm nada a ver com as pontes, mas sim com as características do diagrama apresentado abaixo.

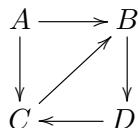


O diagrama obriga a que centremos a nossa atenção na essência: existem quatro localizações genéricas ligadas por sete pontes. Euler demonstrou que qualquer percurso que passe por todas as quatro zonas da cidade, voltando à inicial, tem de usar um número par de pontes da ilha A. Mas como a ilha A tem um número ímpar de pontes, não existe nenhum percurso que use todas as pontes da ilha uma única vez.

Após Euler ter introduzido o diagrama anterior este tipo de abstracções tornou-se extremamente útil nas matemáticas discretas; são designados de **grafos**. Geralmente um grafo é um conjunto de pontos designados **vértices** ligados por linhas designadas de **arestas**. Abaixo apresentamos mais exemplos de grafos.

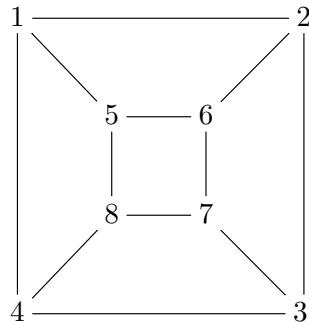


Algumas vezes as arestas dum grafo são **orientadas**, definido uma orientação ou sentido para a sua travessia. Um grafo cujas arestas estão todas orientadas diz-se um **grafo orientado**. Apresentamos de seguida um grafo orientado. Os grafos orientados têm muitas aplicações podem, por exemplo, ser usados para representar posições ou movimentos legais em determinados tipos de jogos, ou por exemplo podem ser usados para descrever máquinas de estados.

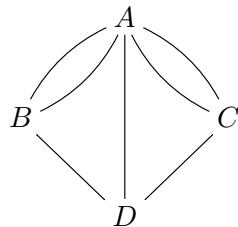


No decurso deste capítulo apresentaremos mais exemplos de grafos, orientados e não orientados, e veremos que eles são úteis para formalizar uma grande variedade de problemas. Para facilitar a sua apresentação vamos definir um conjunto de conceitos que fazem parte do 'jargon' da teoria de grafos. Por simplicidade, apenas nos centramos nos grafos não orientados, facilmente os conceitos apresentados se podem estender aos grafos orientados.

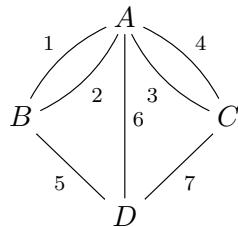
Um **caminho** num grafo é uma sequência de vértices v_1, v_2, \dots, v_n , tais que $\overline{v_1v_2}, \overline{v_2v_3}, \dots, \overline{v_{n-1}v_n}$ são arestas do grafo. Por exemplo, no grafo



um vértice está rotulado com os inteiros de 1 a 8, a sequência de vértices 567341 representa um caminho definido por seis arcos $\overline{56}, \overline{67}, \overline{73}, \overline{34}$, e $\overline{41}$. Note que esta notação não é adequada para um grafo como

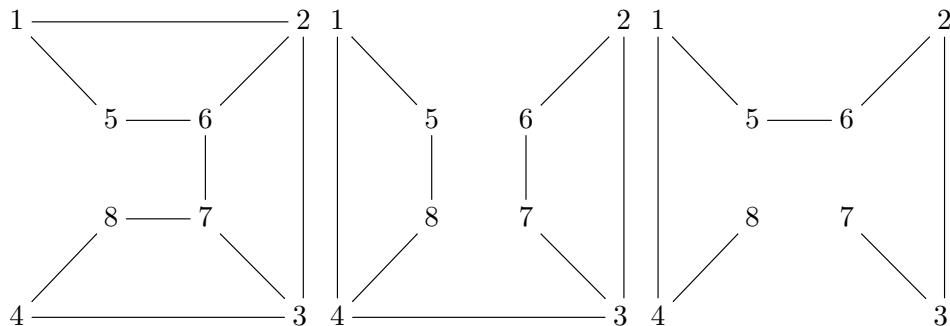


uma vez que têm **arcos múltiplos**, i.e. dois ou mais arcos unindo o mesmo par de vértices. Para este grafo é conveniente rotular as arestas para as distinguir. Caso tomemos o **multi-grafo**, i.e. grafo com arcos múltiplos:

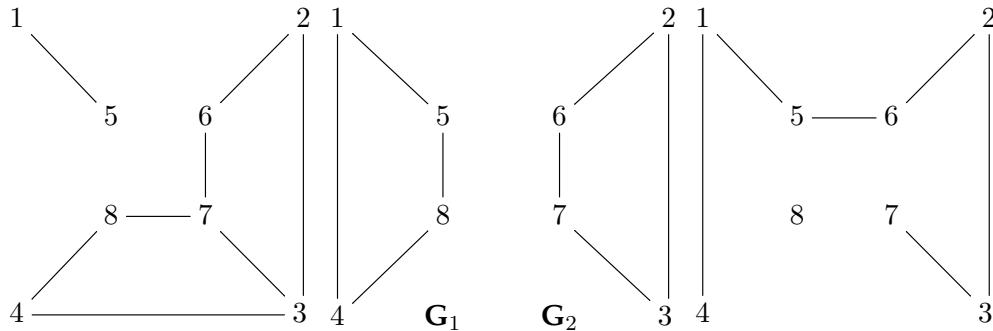


a sequência de arestas $\overline{147625}$ define, sem ambiguidades, o caminho BACDABD.

Um grafo diz-se **conexo** se para todo o par de vértices existe um caminho entre eles. São exemplos de grafos conexos:

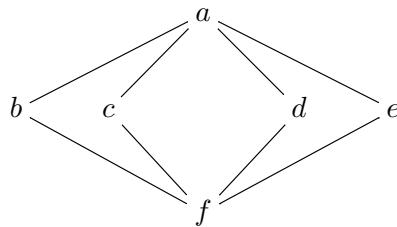


Temos por exemplo como grafos não conexos ou **desconexos**:



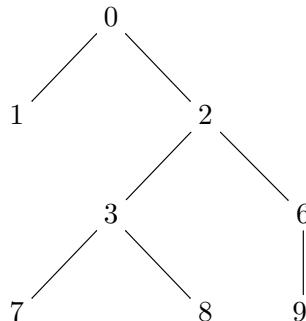
O primeiro grafo é desconexo porque não existe, por exemplo, caminho de 5 para 6. O segundo grafo é desconexo e definido por duas componentes conexas G_1 e G_2 .

Um **ciclo** é um caminho que tem inicio e fim no mesmo vértice, sem repetição de arestas. No grafo



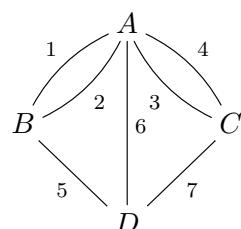
o caminho $abfea$ define um ciclo com quatro arestas, ou um **ciclo de comprimento 4**.

Um grafo conexo sem ciclos chama-se uma **árvore**. Apresentamos como exemplo de árvore:

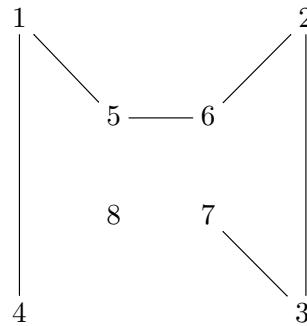


As árvores são extremamente importantes para a teoria de grafos e suas aplicações. Nas próximas secções vamos abordá-las em mais detalhe.

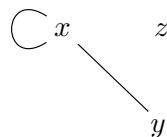
O número de arcos que saem dum vértice é designado de **grau** do vértice. Por exemplo no multi-grafo



o vértice A tem grau 5 e o vértice D tem grau 3. No grafo

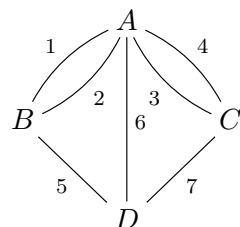


os vértices 4 e 7 têm grau 1, os vértices 1,5,6,2 e 9 têm grau 2, enquanto o vértice 8 tem grau 0. Os vértices com grau zero num grafo dizem-se **vértices isolados**. No grafo abaixo o vértice z é isolado uma vez que não está ligado a nenhum outro.

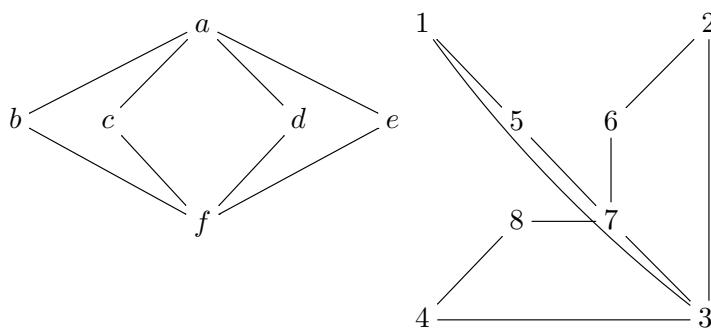


O arco peculiar que parte de x para x é designado por **lacete**. Assim, neste grafo o grau do vértice x é três já que podemos ver o lacete como definindo duas formas distintas para sair de x .

Definimos um **ciclo de Euler** como sendo um ciclo que passa uma única vez por cada aresta do grafo. Assim o problema das pontes de Königsburg pode ser enunciado como: O grafo



tem um ciclo de Euler? Resposta que já sabemos se for negativa. No entanto existem grafos com ciclos de Euler, por exemplo os dois grafos



têm ciclos de Euler. No primeiro caso $adfcadfea$ é um ciclo de Euler. Euler demonstrou uma condição necessária e suficiente para que um grafo tenha ciclos de Euler. Este é considerado o resultado que deu origem à teoria de grafos.

Teorema 9.1.1 (Teorema de Euler para os ciclos de Euler). *Se o grafo G não tem vértices isolados, então ele tem ciclos de Euler se e só se*

1. *G é conexo, e*
2. *o grau de cada vértice de G é par.*

Proof. Vamos dividir a demonstração deste resultado em duas partes. Vamos mostrar que:

- Se G tem um ciclo de Euler, então G é conexo e todos os vértices têm grau par.
- Se G é conexo e todos os vértices têm grau par então G tem um ciclo de Euler.

Comecemos por provar a primeira parte. Assim, suponhamos que o grafo tem um ciclo de Euler, então dados dois vértices x e y eles estão necessariamente no ciclo uma vez que, por hipótese, o grafo não tem pontos isolados. Uma vez que um ciclo de Euler passa por todas as arestas de G , o ciclo de Euler liga x e y . Assim G é conexo.

Continuando a assumir que G tem um ciclo de Euler. Seja x um qualquer vértice de G e imagine uma formiga, que partindo de x , percorre os vértices de G seguindo a ordem estabelecida pelo ciclo de Euler. Uma vez que após completar o ciclo de Euler a formiga volta a x , então no decurso da jornada, a formiga deve ter partido de x tantas vezes quantas as que chegou a x . Assim podemos classificar as arestas que no ciclo têm x como vértice de partida como "arestas de partida" e as que têm x como vértice de chegada como "arestas de chegada", existindo assim arestas com estas classificações em igual número. Assim o número de arestas definidas em x , i.e. o grau de x , é um número par.

A segunda parte do teorema é demonstrada construtivamente, vamos mostrar como se pode determinar um ciclo de Euler num grafo conexo onde todos os vértices tem grau par. Neste sentido vamos apresentar um **algoritmo** para encontrar um ciclo de Euler. Fixemos um vértice inicial a , e começamos uma travessia aleatória ao longo das arestas do grafo, sem a sua repetição até se ficar bloqueados, digamos no vértice b . Que razões podem levar a que não se possa sair dum vértice? A única razão é todos os arcos que partem de b já foram usados na travessia. Se $a \neq b$ isto significa que chegamos a b mais vezes que as que saímos. Isto quer dizer que o grau de b tem de ser ímpar, o que contradiz o facto de todos os vértices de G terem grau par. Isto implica que não podemos ficar bloqueados num vértice b e $b \neq a$. Assim, caso fiquemos bloqueados só pode ser em a onde iniciamos o processo, e os vértices escolhidos formam um ciclo.

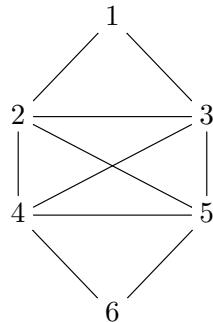
Se algum vértice, digamos c , no ciclo tem um arco que não foi usado, então podemos sempre obter um ciclo em G com inicio em c , percorrendo aleatoriamente o grafo sem recorrer a arestas já usadas. Isto porque se removermos as arestas já usadas de G temos um grafo conexo cujos vértices, com arestas por usar, têm grau par. Este segundo ciclo pode ser usado para substituir c no primeiro ciclo, permitindo criar um ciclo maior que o inicial. Sempre que podemos fazer isso dizemos que o ciclo é quebrado em c .

Podemos executar o procedimento anterior até que nenhum vértice tenha arestas por visitar, i.e. até que todas as arestas do grafo sejam usadas. Isto significa que encontrámos um ciclo que usa todas as arestas exactamente uma vez. Para justificar que

assim é note que: seja \bar{u} uma aresta no grafo tendo por vértices x e y . Assim como G é conexo deve existir pelo menos um caminho de a para x , digamos $aa_1a_2 \dots x$. A aresta \bar{aa}_1 deve estar algures no ciclo, porque caso contrário poderíamos ter quebrado o ciclo em a . Assim o vértice a_1 está no ciclo, que por sua vez implica que o arco $\bar{a_1a_2}$ está no ciclo, uma vez que uma quebra em a_1 não é possível. De forma semelhante para a_2, a_3, \dots, x têm de estar no ciclo. Finalmente como uma quebra em x não é possível, a aresta \bar{u} que liga x a y tem de estar no ciclo. Assim sempre que não é possível fazer quebras ao ciclo o ciclo é de Euler.

Isto completa a nossa demonstração da segunda parte do teorema de Euler. \square

Pelo resultado anterior podemos dizer que o grafo seguinte tem um ciclo de Euler visto ser conexo e todos os vértices terem grau par.



Usando o procedimento apresentado na demonstração anterior determinemos um circuito de Euler para este grafo. Para isso começamos por representar o grafo através dumha **matriz de adjacência**. A matriz de adjacência, dum grafo sem arcos duplos ou lacetes, é uma matriz binária i.e. preenchida com 0s e 1s, neste caso fica:

$$A = [a_{ij}] = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Usamos 1s para indicar a presença de arestas entre vértices e 0s para indicar a inexistência de arestas. Assim $a_{34} = 1$ porque existe um arco do vértice 3 para o vértice 4. De forma semelhante, $a_{15} = 0$ porque não existe um arco ligando 1 e 5.

Para encontrar um caminho de Euler começamos por escolher um vértice arbitrário, digamos 1, e uma aresta que parta de 1 por exemplo $\overline{12}$.

Tendo escolhido esta aresta mudamos na matriz de adjacência a_{12} e a_{21} de 1 para 0. Isto serve para indicar que esta aresta já foi usada no trajecto. Resulta assim o caminho

12, e a nova matriz de adjacência:

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

Do vértice 2 escolhemos a aresta $\overline{23}$ e "apagamos" os 1's de a_{23} e a_{32} . De 3 seguimos para 1 por $\overline{31}$ e alteramos a matriz. Neste ponto temos por caminho 1231 e a matriz de adjacência toma a forma:

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

Voltamos a 1 e estamos bloqueados. Não temos arestas disponíveis para deixar 1 e percorrer as arestas que ainda não foram usadas (a primeira linha da matriz de adjacência é nula). Assim somos obrigados a quebrar o ciclo $c(1) = 1231$. Para isso devemos notar que 2 está no ciclo e o arco $\overline{24}$ ainda não foi usado. Assim usando a última matriz de adjacência procuro um ciclo iniciado em 2. Usando o procedimento anterior $c(2) = 2452$ é um ciclo sem arestas repetidas, eliminando as arestas associadas da matriz de adjacência obtemos

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

associada ao caminho $c(1) = 1245231$. O vértice 4 define um corte no ciclo $c(1)$. A partir de 4 podemos definir o ciclo $c(4) = 46534$ dando origem por substituição ao ciclo $c(1) = 12465345231$ e por eliminação à matriz de adjacência

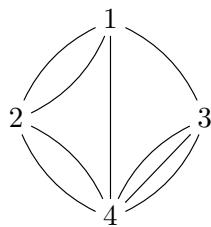
$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Como não temos arestas por usar (a matriz de adjacência é nula) podemos concluir que o ciclo $c(1)$ obtido é um ciclo de Euler.

9.2 Arcos múltiplos

Muitas vezes os grafos considerados têm arcos múltiplos, i.e. existem mais do que um arco ligando o mesmo par de vértices. Neste caso têm o nome de **multi-grafos**. Um exemplo disso é o grafo definido pelo problema das pontes de Königsberg. O Teorema de Euler cobre esta situação, mas o procedimento usado anteriormente para determinar ciclos de Euler com base na matriz de adjacência tem de ser modificado. Se existem arcos múltiplos as posições na matriz devem reflectir o grau dos vértices.

Por exemplo para o grafo



definimos por matriz de adjacência a matriz A tal que

$$A = [a_{ij}] = \begin{bmatrix} 0 & 2 & 1 & 1 \\ 2 & 0 & 0 & 2 \\ 1 & 0 & 0 & 3 \\ 1 & 2 & 3 & 0 \end{bmatrix}$$

onde $a_{12} = 2$ porque existem duas arestas com vértices 1 e 2, $a_{34} = a_{43} = 3$ porque existem três arestas com vértices 3 e 4, $a_{23} = 0$ porque não existem arestas com vértices 2 e 3.

Pelo teorema de Euler este multi-grafo tem um ciclo de Euler já que é conexo e todos os vértices têm grau par. Com base no vértice 1 podemos construir aleatoriamente o ciclo $c(1) = 121341$. Eliminando na matriz de adjacência os arcos não usados obtemos

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

Podemos quebrar o ciclo $c(1)$ no vértice 2 usando o ciclo $c(2) = 242$, obtendo $c(1) = 12421341$ e a matriz de adjacência definida por arcos não usados

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

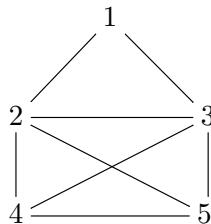
Como não foram usadas todas as arestas o ciclo $c(1)$ pode ser ainda quebrado.

Por exemplo no vértice 4, já que existem arcos que partem de 4 que ainda não foram usados. O ciclo $c(4) = 434$ permite definir o ciclo de Euler $c(1) = 1243421341$ já que tem associada a matriz de adjacência nula:

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

9.3 Caminhos de Euler

Um *caminho de Euler*, aparece como um relaxamento da noção de ciclo de Euler. É definido como sendo um caminho que liga dois vértices passando uma única vez por todas as arestas. Por exemplo o grafo



tem o caminho de Euler 435231245 mas, pelo teorema de Euler, não tem ciclos de Euler já que pelo menos um vértice tem grau ímpar (os vértices 4 e 5 têm grau 3).

As condições para a existência de caminhos de Euler são apresentadas no seguinte teorema.

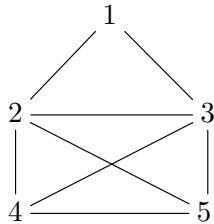
Teorema 9.3.1 (Teorema de Euler para caminhos de Euler). *Se G não tem vértices isolados, então tem um caminho de Euler se e só se*

1. G é conexo, e
2. tem exatamente dois vértices de grau ímpar.

Proof. A demonstração de que um grafo com um caminho de Euler deve satisfazer 1 e 2 é idêntica à prova da primeira parte do teorema dos ciclos de Euler. A única diferença é que devemos notar que o vértice "inicial" e "terminal" no caminho são diferentes: A formiga que percorre o caminho deixa o vértice de partida mais uma vez que o número de vezes que lá chega; e ela chega ao vértice final uma vez mais que o número de vezes que de lá parte. Assim ambos estes vértices têm grau ímpar.

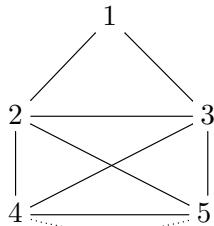
Por outro lado para provar que um grafo com estas propriedades tem um caminho de Euler usamos um truque: adicionamos um **arco imaginário** ligando os dois vértices de grau ímpar. O grafo resultante tem todos os vértices de grau par e continua a ser conexo. Podemos assim obter um ciclo de Euler neste grafo, usando o algoritmo anterior, então apagando no ciclo obtido o arco imaginário obtemos um caminho de Euler no grafo original. \square

Procuremos então um caminho de Euler para o grafo G



que existe pelo teorema anterior, já que é conexo e existem apenas dois vértices de grau ímpar, os vértices 4 e 5.

Com base no grafo G devemos criar um novo grafo G' acrescentando para isso uma **aresta imaginária** $\overline{45}$. Assim todos os vértices de G' têm grau par:



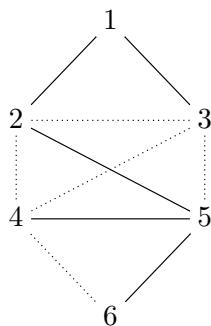
Cujo ciclo de Euler $c(4) = 4234521354$ pode ser obtido pelo algoritmo apresentado anteriormente. Deste ciclo podemos obter um caminho de Euler para G removendo a aresta imaginária $\overline{45}$:

423452135

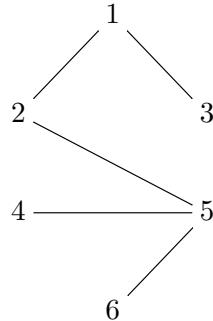
9.4 Árvores de suporte

Na secção anterior introduzimos a noção de grafo, e apresentamos um algoritmo simples para encontrar caminhos de Euler num grafo com determinadas características. Na verdade, muitas das soluções a problemas de teoria de grafos são melhor formalizadas através da apresentação de algoritmos. Nesta secção e nas seguintes abordaremos vários problemas deste tipo, com importantes aplicações práticas. Em cada caso apresentaremos um algoritmo eficiente para a resolução e procuraremos apresentar exemplos de aplicação do algoritmo.

Já vimos que uma árvore é um grafo conexo sem ciclos. Neste sentido nem todo o grafo é uma árvore, mas removendo arestas podemos sempre determinar uma árvore que está contida no grafo original. Por exemplo no grafo



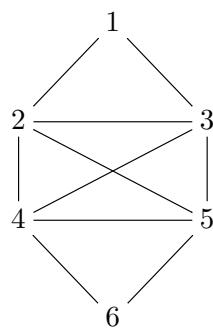
se removermos as arestas a tracejado obtemos uma árvore que é definida pelos vértices do grafo original.



Este tipo de árvore é denominada **árvore de suporte** do grafo.

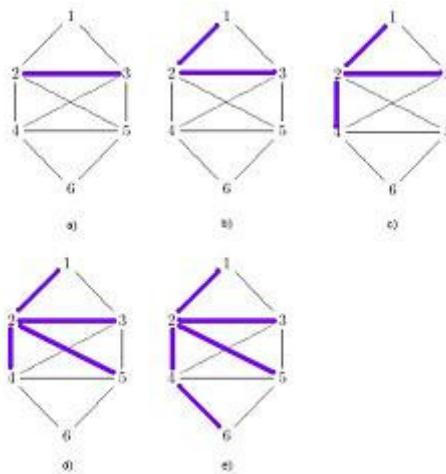
Definição: 9.4.1. A árvore de suporte dum grafo conexo G é uma árvore cujos conjuntos de vértices é o conjunto de vértices de G e as suas arestas definem um subconjunto das arestas de G .

Todo o grafo tem, naturalmente, uma árvore de suporte. Para determinar outra árvore de suporte para o grafo:



Determinemos uma nova árvore de suporte para o grafo anterior. Como a árvore apresentada não tem a aresta $\overline{23}$ tentemos agora determinar uma árvore de suporte contendo esta aresta.

A nossa estratégia passa por adicionar vértices a uma árvore, um de cada vez, até obtermos uma árvore de suporte. Para implementar esta estratégia, em cada iteração procuramos um vértice do grafo que não pertença à árvore mas seja **adjacente**, i.e. esteja ligada, a um vértice da árvore. Fazemos assim crescer a árvore adicionando-lhe essa aresta.



Na figura podemos acompanhar a evolução da árvore de suporte em cada iteração. Começando em a) com uma árvore definida por uma única aresta procuramos novos vértices que não estejam na árvore mas sejam adjacentes a vértices da árvore. Os vértices 1, 4 ou 5 estão nestas condições. Escolhemos o 1. Como o vértice 1 está ligado à árvore pela aresta $\overline{12}$ adicionamos esta aresta à árvore em b). São adjacentes aos vértices desta nova árvore 4 e 5. Escolhemos 4, estando ligada à árvore pelo arco $\overline{24}$ este passa a fazer parte da árvore em c). Assim, esta nova árvore tem vértices adjacentes a 5 e 6. Como $\overline{25}$ estabelece a ligação pode ser usada para construir a nova árvore d). O único vértice que ainda não está na árvore é o 6 que pode ser ligado à árvore através de $\overline{46}$. Dando assim origem à árvore de suporte d).

Podemos assim formalizar o algoritmo que acabamos de usar:

Algoritmo 1 Árvore de suporte dum grafo conexo

Input: Grafo conexo G

Output: Uma árvore T de suporte de G

Escolher um vértice de G para iniciar a árvore T ;

while Existam vértices em G que não estão na árvore T **do**

 Encontrar um vértice de G que ainda não esteja em T mas que seja adjacente a um dos seus vértices.

 Adicionar este vértice e a aresta que define a adjacência em T .

end while

Para termos a garantia de que o algoritmo devolve realmente uma árvore de suporte, temos de garantir que: Primeiro, se a árvore não contém todos os vértices é sempre possível encontrar uma aresta definida por um vértice na árvore e outro fora da árvore. Segundo, que o algoritmo termina devolvendo uma árvore de suporte.

No primeiro caso, seja v um vértice em T , e w um vértice que não está em T . Poderá não existir um arco a ligar v e w , mas como G é conexo existe um caminho de v para w . Como o caminho parte da árvore para fora da árvore existe necessariamente um arco $\overline{v'w'}$ no caminho satisfazendo as condições pedidas. Neste sentido o vértice w' é um bom candidato para ser adicionado à árvore.

Este argumento permite concluir que o processo de crescimento da árvore pode ser realizado até todos os vértices estarem na árvore. Temos no entanto de demonstrar que realmente obtemos uma árvore. Este resultado pode ser demonstrado por **indução**

matemática: Para isso denotemos por T_n o grafo produzido pelo algoritmo na iteração n do ciclo "while" do algoritmo e tomemos a proposição:

$$S_n : T_n \text{ é uma árvore.}$$

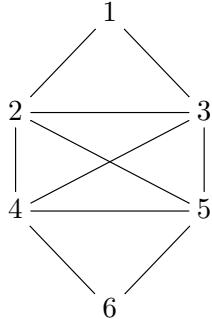
Primeiro, para $n = 1$, T_1 é uma árvore definida apenas por um vértice. Agora vamos mostrar que se S_n é verdade então S_{n+1} também é verdade. Para provar isso note que T_n dá origem a T_{n+1} adicionando um vértice v_{n+1} . Para que T_{n+1} seja uma árvore tem de ser conexo e não pode conter ciclos.

Para provar que T_{n+1} é conexo, note que T_k é uma árvore logo conexa e quando se junta v_{n+1} a nova aresta que define adjacência garante que T_{n+1} é conexo.

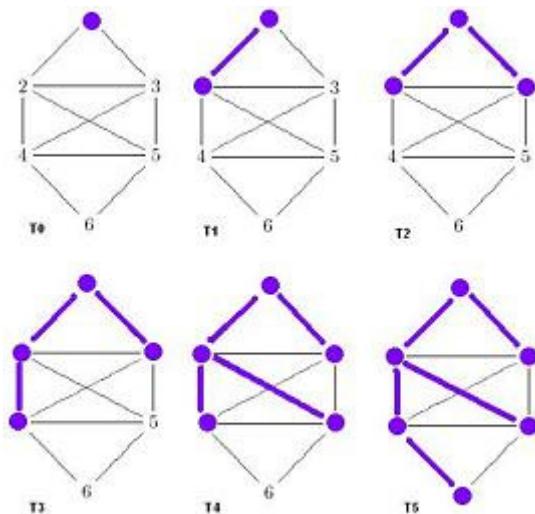
Para mostrar que T_{n+1} não contém ciclos, suponhamos por absurdo que T_{n+1} tem um ciclo. Como T_n é por hipótese uma árvore que não tem ciclos, o ciclo existente em T_{n+1} foi criado pela introdução da nova aresta u_{n+1} . Isto quer dizer que u_{n+1} liga dois vértices da árvore. O que é absurdo porque as novas arestas são definidas por um vértice dentro da árvore e o outro fora. Logo T_{n+1} não contém ciclos.

Isto termina a nossa demonstração por indução: os grafos T_1, T_2, T_3, \dots , são todos árvores. Se G tem m vértices o algoritmo pára após $n - 1$ passos, já que a árvore T_1 tem dois vértices, T_2 tem 3 vértices, T_3 tem 4, ..., T_{m-1} tem os m vértices de G .

Aplicando o algoritmo ao grafo



temos por exemplo:



Na próxima secção, vamos considerar aplicações para as árvores de suporte. Terminamos esta secção com um resultado que apresenta seis definições possíveis para árvore.

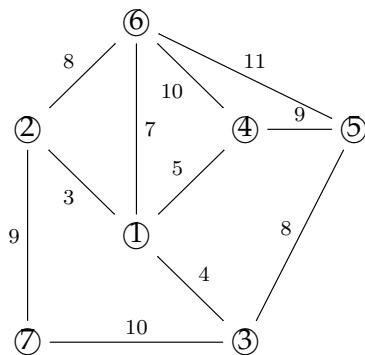
Teorema 9.4.2 (Teorema dos caminhos de Daisy). *Seja T um grafo com m vértices. As seguintes afirmações são equivalentes:*

1. T é uma árvore;
2. T não têm ciclos e $m - 1$ arestas;
3. T é conexo e $m - 1$ arestas;
4. T é conexo, mas apagando qualquer arco fica desconexo;
5. Existe exactamente um caminho entre quaisquer dois vértices;
6. T não tem ciclos, qualquer ligação estabelecida entre dois vértices por um arco cria um ciclo.

9.5 Árvores de suporte mínimo: Algoritmo de Prim

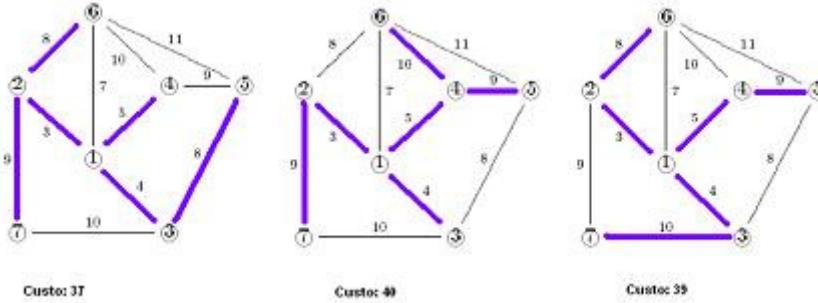
Na secção anterior introduzimos a noção de árvore de suporte. Nesta secção apresentamos uma das suas aplicações.

Considere o grafo apresentado abaixo: é um grafo conexo com 7 vértices e 11 arcos. Note que cada arco tem associado um número. Podemos interpretar o grafo como uma abstracção duma rede ligando 7 cidades, representados pelos vértices, e o número nas arestas representa o custo (por exemplo, em milhões de euros) necessário para a construção dum troço de linha férrea de alta velocidade entre elas.



Podemos encarar a falta de ligação entre 1 e 5 como indicando que o custo da sua ligação é proibitivo, i.e. efectivamente infinito. É uma aplicação interessante procurar a rede de alta velocidade de custo mínimo que une todas as cidades.

Para dar resposta ao problema basta determinar a árvore de suporte de custo mínimo. Existem, como vimos, várias árvore de suporte dum grafo e o custo de cada uma é definida como sendo a soma dos custos de cada arco. São exemplos, neste caso de árvores de suporte:



A questão que se pode levantar é: Existe uma árvore de suporte de custo mínimo?

Esta questão é um caso particular do chamado problema da **árvore de suporte de custo mínimo** (ASCM). Dado um grafo G no qual cada arco está rotulado com um custo, uma ASCM é uma árvore de suporte cuja soma dos rótulos de cada aresta é a menor possível. (Nas aplicações este rótulos determinam geralmente um custo, um comprimento ou um peso.)

Surpreendentemente existe um algoritmo muito simples para encontrar ASCM, basta para o definir fazer uma pequena alteração ao algoritmo usado na secção anterior para gerar árvores de suporte. Este algoritmo é conhecido por **algoritmo de Prim**.

Algoritmo 2 Algoritmo de Prim

Input: Grafo conexo G com as arestas rotuladas. G

Output: Uma ASCM T de G

Escolher um vértice de G para iniciar a árvore T ;

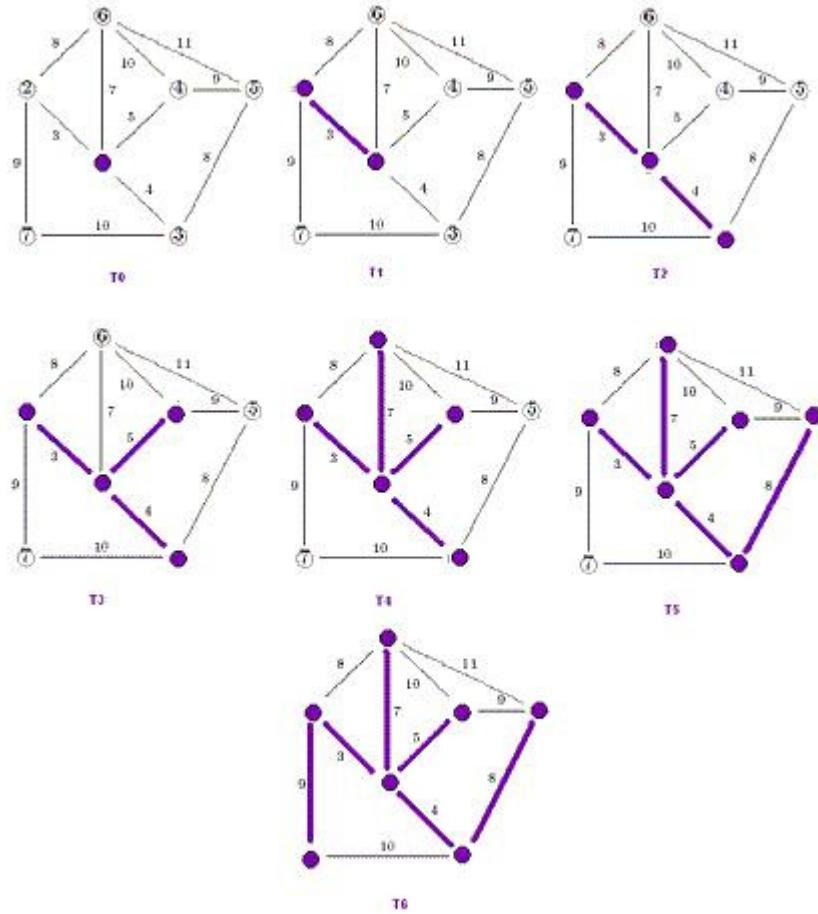
while Existam vértices em G que não estão na árvore T **do**

Determinar o vértice de G que não esteja em T e que esteja ligado a T pela ligação de custo mínimo.

Adicionar este vértice e a aresta que define a adjacência a T .

end while

Não é de todo óbvio como a escolha da aresta disponível de custo mínimo permite definir a ASCM. Na verdade o algoritmo de Prim devolve sempre a ASCM, antes de demonstrar por que assim é, determinemos a ASCM do exemplo inicial aplicando o algoritmo.



Escolhemos como vértice inicial o 1, sendo a árvore inicial T_0 definida por este vértice, são candidatos a ser usados para estender a árvore 2,6,4,3, escolhemos 2 para definir T_1 porque a adjacência é definida pelo arco de menor custo. Os vértices de T_1 são adjacentes a 6, 4, 3 e 7, o arco de custo mínimo é $\overline{13}$. Assim usamos 3 para definir T_2 . Os vértices 6, 4, 5 e 7 não estão em T_2 mas são adjacentes a vértices de T_2 . O arco $\overline{14}$ tem custo mínimo. Usamos assim 4 para definir T_3 . O arco $\overline{16}$ é o mais barato que saí de T_3 , logo usamos 6 para definir T_4 . E como $\overline{35}$ é o arco de custo mínimo que saí de T_4 usamos 5 para definir T_5 . Só nos falta adicionar o vértice 7, cuja adjacência de custo mínimo é definida pelo arco $\overline{72}$.

Terminamos esta secção com a demonstração de que o algoritmo de Prim determina a árvore de suporte de custo mínimo. Para isso, seja G um grafo com n vértices, com custos associados às suas arestas, e seja e_1, e_2, \dots, e_{n-1} a sequência de $n - 1$ arestas seleccionadas pelo algoritmo de Prim a quando da expansão das árvores. Vamos mostrar que o grafo definido pelas arestas e_1, e_2, \dots, e_{n-1} definem a ASCM de G . Para fazer isso, para cada $k = 0, 1, \dots, n - 1$ denotamos por T_k o grafo definido pelas primeiras k arestas seleccionadas pelo algoritmo. (O grafo T_0 consiste apenas no vértice inicial.) Definimos a proposição:

$$P_k : \text{As arestas de } T_k \text{ definem um subconjunto de arestas da ASCM de } G.$$

Queremos provar P_{n-1} . A ideia é demonstrar que as proposições $P_0, P_1, P_2, \dots, P_{n-1}$ são verdadeiras, usando indução matemática. Para isso vamos mostrar que se P_k é

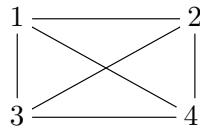
verdadeira, então P_k também é verdadeira.

A proposição P_0 é verdadeira, porque T_0 não contém nenhuma aresta.

Suponhamos agora que P_k é verdadeira, i.e., que se T é a ASCM de G então T_k é um subgrafo de T . Pela forma como o algoritmo trabalha as extremidades de e_{k+1} , digamos $e_{k+1} = \overline{vw}$, então v está em T_k e w está fora de T_k . Se o ASCM T contém e_{k+1} então T_k está contido em T o que prova o resultado. Se e_{k+1} não está em T_k então T contém um caminho ligando v e w . Este caminho começa dentro de T_k e termina fora, assim existe necessariamente uma aresta e' que tem um vértice dentro de T_k e o outro fora. Como e_{k+1} foi escolhida por forma a ter o menor custo mínimo em relação às arestas que partem de T_k , o custo de e' é maior ou igual ao de e_{k+1} . Assim se removermos e' de T e substituirmos por e_{k+1} obtemos um grafo conexo com $n - 1$ arcos, ou seja uma árvore. Mas assim o custo total de T' é menor que o custo de T . O que contradiz o facto de T ser a ASCM G . Isto implica que e_{k+1} tem de estar em T , ou seja T_k é um subgrafo de T . O que conclui a demonstração por indução de que P_k é verdadeiro para todo $k = 0, 1, \dots, n - 1$.

9.6 EXERCÍCIOS DE REVISÃO

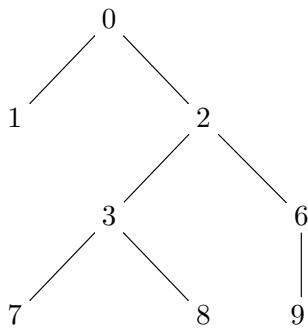
Exercício 9.6.1. Determine todos os ciclos do grafo



Exercício 9.6.2. Desenhe um grafo com quatro vértices $\{v_1, v_2, v_3, v_4\}$, tais que se $d(v)$ é o grau do vértice v , então $d(v_1) = 3$, $d(v_2) = d(v_3) = 2$, e $d(v_4) = 1$.

Exercício 9.6.3. Construa um grafo desconexo com cinco vértices e três componentes conexas.

Exercício 9.6.4. Considere a árvore:



Se uma nova aresta de vértices 1 e 7 é introduzida na árvore, verifique que existe um único ciclo no grafo resultante.

Exercício 9.6.5. Justifique porque é que, se um novo arco é acrescentado a uma árvore, o grafo resultante tem apenas um ciclo.

Exercício 9.6.6. Apresentamos abaixo matrizes de adjacência de grafos. Represente graficamente esses grafos e identifique os que têm ciclos de Euler. Se tais ciclos existirem determinemos caso tal não aconteça determine caso existam caminhos de Euler:

$$1. \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$2. \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$3. \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$4. \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 2 & 0 & 1 \\ 1 & 0 & 0 & 2 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

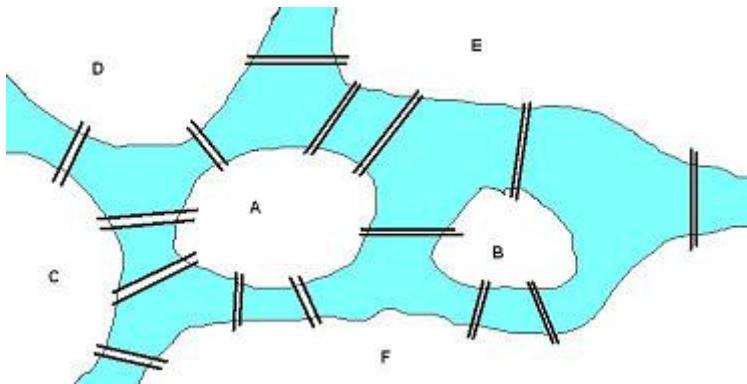
$$5. \begin{bmatrix} 2 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 2 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 2 \end{bmatrix}$$

Exercício 9.6.7. Mostramos no decurso desta apresentação que o grafo das pontes de Königsburg não tem ciclos de Euler. Ele tem caminhos de Euler? Se tem, determine-o.

Exercício 9.6.8. Qual o número mínimo de pontes extra que teria de se construir por forma a que o grafo das pontes de Königsburg tenha ciclos de Euler.

Exercício 9.6.9. Mostre que uma árvore com mais de dois vértices tem pelo menos um vértice de grau ímpar.

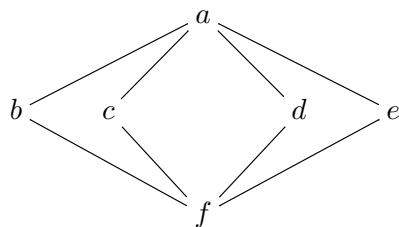
Exercício 9.6.10. No artigo original de Euler é descrito uma cidade imaginária com ilhas e pontes com a disposição apresentada abaixo. Existe algum ciclo ou caminho de Euler nesta cidade? Apresente a matriz de adjacência do grafo associado.



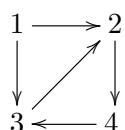
Exercício 9.6.11. Seja G um grafo e S denota a soma dos graus de todos os vértices do grafo G , e seja N o número de vértices de grau ímpar.

1. Mostre que tanto S como N são sempre números pares.
2. Mostre que S é o dobro do número de arestas.

Exercício 9.6.12. Quantos ciclos de Euler diferentes existem no grafo abaixo?



Exercício 9.6.13. Neste exercício esperamos que consiga demonstrar o teorema de Euler para grafos dirigidos. Num grafo dirigido um caminho deve respeitar o sentido dos arcos. Assim no grafo dirigido abaixo, a sequência 13243 define um caminho válido mas a sequência 13423 não é um caminho válido. Se denotarmos por $d(v)$ o grau dum vértice v é usual definir: num grafo orientado $d^+(v)$ é o grau de entrada no vértice v , i.e. o número de arestas definidas para v , $d^-(v)$ é o grau de saída no vértice v , i.e. o número de arestas partem de v . Naturalmente temos $d(v) = d^+(v) + d^-(v)$. No grafo



temos $d^+(1) = 2$, $d^-(1) = 0$, $d^+(3) = 1$, $d^-(3) = 2$ e $d(2) = d^+(2) + d^-(2) = 1 + 2 = 3$.

Assim o teorema de Euler toma a seguinte forma para grafos orientados.

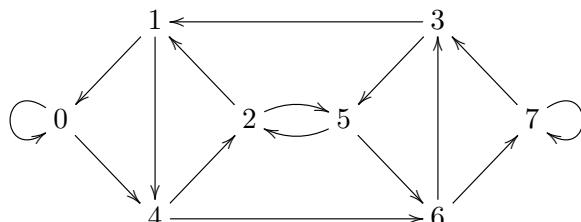
Teorema 9.6.14 (Teorema de Euler para grafos dirigidos). *Se o grafo G não tem vértices isolados, então ele tem ciclos de Euler se e só se*

1. G é conexo, e
2. o grau de cada vértice v de G é balanceado, i.e. $d^+(v) = d^-(v)$.

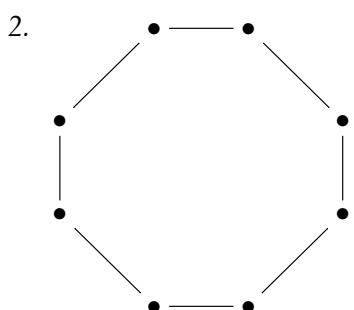
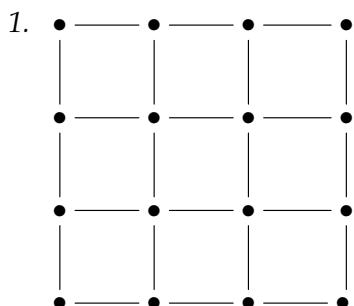
Demonstre o teorema de Euler para grafos dirigidos. Para o fazer note que o algoritmo para a determinação de ciclos de Euler é praticamente o mesmo apenas tem de se ter em conta o sentido do caminho quando se faz a quebra de caminhos. Note ainda que a matriz de adjacência pode reflectir a orientação das arestas: Seja $A = [a_{ij}]$ a matriz de adjacência do grafo G , se existe um arco v de i para j e não existe arco de j para i fazemos $a_{ij} = 1$ e $a_{ji} = 0$. Por exemplo a matriz de adjacência do grafo dirigido anterior é:

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

Exercício 9.6.15. Na continuação do exercício anterior. Use o algoritmo sugerido para determinar um circuito de Euler do grafo que se segue.



Exercício 9.6.16. Encontre árvores de suporte para os grafos:



Exercício 9.6.17. A matriz abaixo é a matriz de adjacência dum grafo. Desenhe esse grafo e determine uma árvore de suporte.

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

Exercício 9.6.18. A matriz abaixo é a matriz de adjacência dum grafo. Desenhe esse grafo e determine uma árvore de suporte.

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

Exercício 9.6.19. Poderá uma árvore com oito vértices satisfazer?

1. Cada vértice tem grau 1.
2. Os vértices têm graus 2, 2, 2, 2, 2, 2, 1, 1.
3. Todos os vértices têm grau 2.
4. Os vértices têm graus 7, 1, 1, 1, 1, 1, 1, 1.

Exercício 9.6.20. Classifique se as proposições abaixo são válidas. No caso das proposições serem válidas demonstre-as, caso contrário apresente contra-exemplos.

1. Se G é conexo, então é uma árvore.
2. Se G não tem ciclos, tem 25 arestas e 26 vértices, então G é conexo.
3. Se G é conexo e tem 10 arcos e 10 vértices, então G tem pelo menos um ciclo.

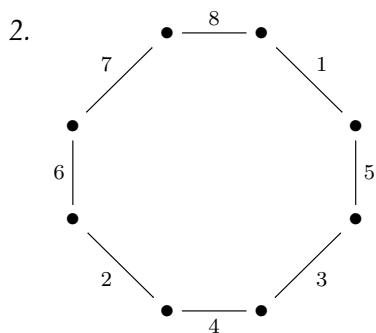
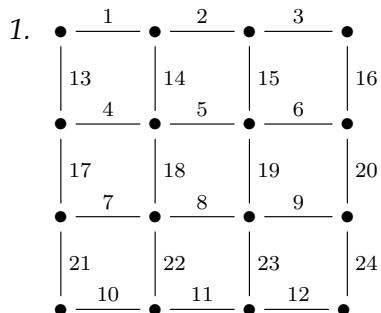
Exercício 9.6.21. Suponhamos que G é conexo, com 13 vértices e 17 arestas. Quantas arestas tem qualquer uma das suas árvores de suporte?

Exercício 9.6.22. Sejam T_1 e T_2 duas árvores de suporte de G . T_1 e T_2 têm o mesmo número de arestas? E o mesmo número de vértices?

Exercício 9.6.23. Abaixo apresentamos algoritmos com o propósito de determinarem árvores de suporte. Em cada caso verifique se realmente faz o pretendido:

1. Dado um grafo. Se tem um ciclo, removemos uma das suas arestas do grafo. Se o grafo resultante ainda tiver ciclos voltamos a remover um dos seus arcos. Continuamos assim enquanto existirem ciclos.
2. Iniciando o processo com um arco, adicionamos continuamente novos arcos, um por cada iteração até obtermos um grafo com $m - 1$ arestas.
3. Iniciando o processo com um arco. Adicionamos continuamente novos arcos, um por cada iteração, sujeita à restrição de não poderem definir ciclos. Continuamos o processo enquanto existam arestas que podem ser adicionadas, não definindo ciclos.

Exercício 9.6.24. Aplique o algoritmo de Prim para determinar a ASCM dos seguintes grafos:



Exercício 9.6.25. Represente graficamente e aplique o algoritmo de Prim para determinar a ASCM dos grafos definidos pelas **matrizes de custos** abaixo.

1.

$$\begin{bmatrix} 0 & 2 & \infty & \infty & 3 \\ 2 & 0 & 1 & \infty & \infty \\ \infty & 1 & 0 & 4 & \infty \\ \infty & \infty & 4 & 0 & 5 \\ 3 & \infty & \infty & 5 & 0 \end{bmatrix}$$

Com $a_{12} = 2$ queremos dizer que existe um arco de 1 para 2 e que tem custo 2. O facto de $a_{22} = 0$ indica que não existem lacetes e por $a_{24} = \infty$ não existe arco de 2 para 4.

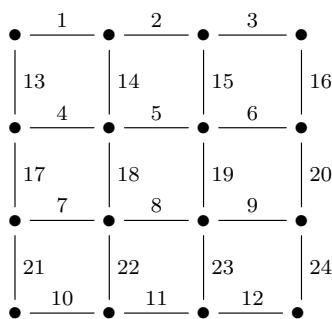
2.

0	2	∞	∞	3	1	∞	∞	∞	∞
2	0	1	∞	∞	∞	2	∞	∞	∞
∞	1	0	4	∞	∞	∞	3	∞	∞
∞	∞	4	0	5	∞	∞	∞	4	∞
3	∞	∞	5	0	∞	∞	∞	∞	5
1	∞	∞	∞	∞	0	∞	2	3	∞
∞	2	∞	∞	∞	∞	0	∞	4	5
∞	∞	3	∞	∞	2	∞	0	∞	1
∞	∞	∞	4	∞	3	4	∞	0	∞
∞	∞	∞	∞	5	∞	5	1	∞	0

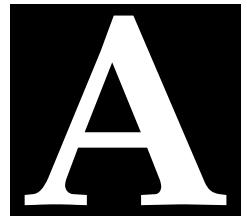
Exercício 9.6.26. Determine um grafo com mais do que uma árvore de suporte mínima.

Exercício 9.6.27. A árvore de suporte com custo máximo de um grafo cujas arestas têm custos associados é a árvore de suporte com custo total máximo. Apresente um algoritmo, baseado no algoritmo de Prime que permita calcular árvores de suporte com custo máximo.

Exercício 9.6.28. Use o algoritmo que apresentou na alínea anterior para determinar a árvore de suporte de custo máximo do grafo



Exercício 9.6.29. Considere o grafo da alínea anterior. Como procederia se tivesse de determinar uma árvore de suporte que contivesse a aresta de custo 23 e cujo custo total fosse o menor possível?



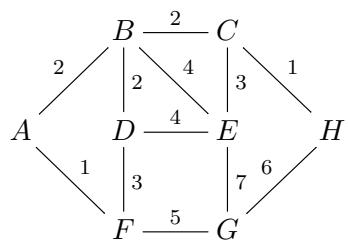
Teoria de grafos

O assunto abordado neste capítulo é de leitura facultativa.

A.1 O problema do caminho mais curto: Algoritmo de Dijkstra

Na secção anterior apresentámos o algoritmo de Prim, para o qual considerámos grafos com arestas rotuladas. Nesta secção também abordamos grafos rotulados, mas em vez de se interpretar os rótulos das arestas como um custo, elas são interpretadas como distâncias.

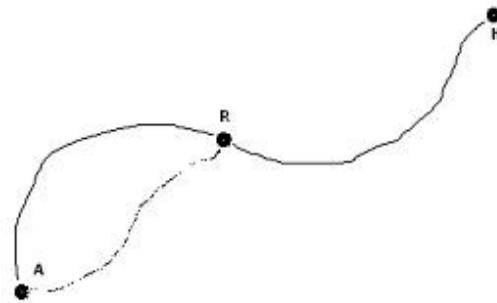
Suponhamos que estamos a observar um mapa de estradas ligando cidades e pretendemos determinar o caminho mais curto entre duas delas. Podemos representar a informação relevante num grafo, identificando os vértices como cidades e os arcos como percursos, rotulados pelo seu comprimento. Alternativamente o rótulo pode identificar o tempo necessário para percorrer o percurso ou custo associado à sua utilização. Para simplificar assume-se os rótulos com comprimentos (definidos por números positivos). Como exemplo vamos usar o grafo abaixo.



Num grafo rotulado definimos comprimento dum caminho como sendo a soma do comprimento dos arcos que definem esse caminho. No grafo anterior o caminho AFDEGH de A para H tem comprimento $1+3+4+7+6=21$. Um problema que surge naturalmente,

e que tem muitas aplicações, é o de saber se dados A e H, o caminho apresentado é o de comprimento mínimo. O algoritmo que vamos apresentar para dar resposta a este problema é similar ao algoritmo de Prim e denominado de algoritmo de Dijkstra.

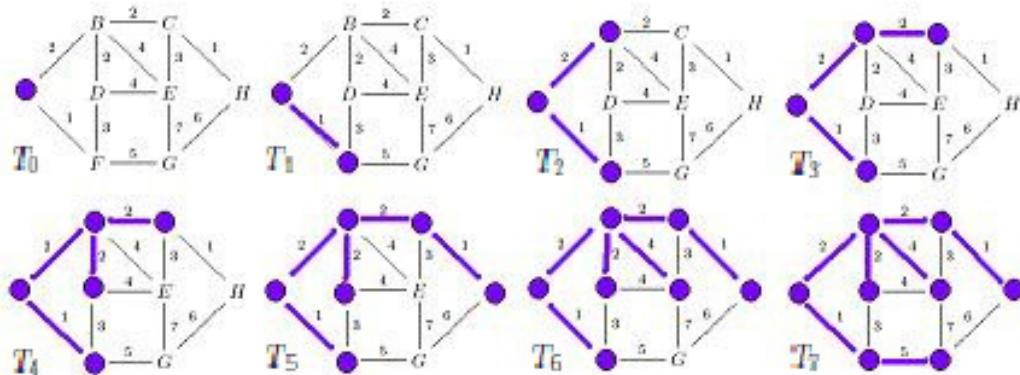
O algoritmo de Dijkstra tem por objectivo a determinação do caminho mais curto entre dois vértices dum grafo rotulado (ou melhor, um dos caminhos de comprimento mínimo se existem mais do que um de igual comprimento). Dados dois vértices A e H, a estratégia a adoptar consiste em começar no vértice A e progressivamente construir uma lista com os caminhos mais custos para todos os vértices que estão entre H e A, ordenados segundo a distância que os separa de A, até H ser alcançado.



Supondo que a linha continua na figura representa o caminho mais curto de A para H, e seja R um vértice no caminho algures entre estes vértices. Assim o caminho definido pela linha continua define o caminho mais curto de A para R. Isto porque, se existe um caminho mais curto de A para R, poderíamos encontrar um caminho mais curto que o caminho dado. O que contradiz a hipótese de na figura a linha continua representar o caminho mais curto. De forma semelhante a linha continua define o menor caminho de R para H. Este princípio é conhecido por **princípio da optimalidade** e não é só a base de funcionamento do algoritmo de Dijkstra mas de muitos outros algoritmos em Matemática discreta.

Princípio de optimalidade: o caminho mais curto de A para H contém o caminho mais curto para qualquer dos vértices intermédios.

Usamos o exemplo seguinte para se ficar com uma ideia informal de como o algoritmo de Dijkstra funciona.



Iniciamos o processo com o vértice A, que define a árvore T_0 , e localizamos o vértice que lhe seja adjacente e mais próximo possível. Que é o vértice F e tornamo visível

a aresta AF, definindo T_1 . Note que o caminho AF é o mais curto de A para F. Vamos pensar em AF como o primeiro arco da árvore de suporte final.

No passo seguinte vamos procurar vértices adjacentes à árvore T_1 . Estão nesta situação os vértices B (via AB), D (via FD) e G (via FG). Vamos adicionar a T_1 o vértice que está mais próximo de A, para isso usamos a tabela:

Vértice	Caminho	comprimento
B	AB	2
D	AFD	4
G	AFG	6

Como B está mais próximo adicionamos B à árvore tornando visível a aresta AB, definindo T_2 .

Seguimos esta estratégia, em cada iteração escolhemos para adicionar à árvore o vértice adjacente à árvore que está o mais próximo possível A. Existe exactamente um único caminho de A para cada vértice da árvore gerada, e impondo que este caminho seja sempre o caminho mais curto de A.

À árvore T_2 da figura é possível adicionar um dos vértices C, D, E ou G. Para determinar qual deve ser adicionado devemos notar que:

Vértice	Caminho	comprimento
C	ABC	4
D	ABD ou AFD	4
E	ABE	6
G	AFG	6

Temos dois caminhos de comprimento 4, temos duas escolhas possíveis, adicionamos C (via BG) ou então D (via BD ou FD). Escolhemos para adicionar C a T_2 e evidenciamos a aresta BC, definindo T_3 .

Na iteração seguinte, os candidatos são D, E, G e H:

Vértice	Caminho	comprimento
D	ABD ou AFD	4
E	ABE	6
G	AFG	6
H	ABCH	4

(Note que E pode também ser ligado à árvore via o caminho ABCE, mas uma vez que o caminho é maior que ABE não o metemos na tabela) O caminho mais curto tem comprimento 4, e assim devemos adicionar à árvore o vértice D e evidenciámos a aresta FD, definindo T_4 .

Na iteração seguinte são candidatos os vértices E, G ou H.

Vértice	Caminho	comprimento
E	ABE	6
G	AFG	6
H	ABCH	5

Como o caminho mais curto tem comprimento 5, adicionamos H à árvore via a aresta CH, definindo T_5 . Como queremos encontrar o caminho mais curto de A para H podemos parar. O caminho ABCH é o caminho mais curto que liga A a H.

Caso se queira continuar a construir a árvore (existem ainda dois vértices que não estão na árvore E e G). São candidatos a inserir na árvore E e G como:

Vértice	Caminho	comprimento
E	ABE	6
G	AFG	6

Como os vértices estão a igual distância de A é irrelevante. Escolhendo E via BE, definimos T_5 . Finalmente introduzimos o vértice G via FG, definindo T_6 . Obtemos assim uma árvore de suporte para o grafo inicial, chamaremos a esta árvore "**árvore de suporte de Dijkstra**".

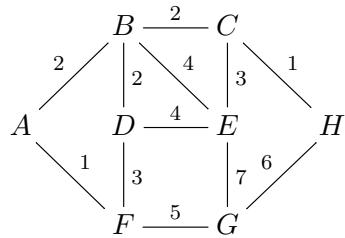
Podemos ver o algoritmo de Dijkstra como um método para determinar um tipo particular de árvore de suporte. A ideia é iniciar o processo com um vértice v_0 , e em cada iteração adicionar o vértice adjacente à árvore que esteja mais próximo de v_0 . O algoritmo de Prim com este processo de seleção determina uma árvore de suporte de Dijkstra. A nossa próxima tarefa consiste em formalizar o algoritmo.

Algoritmo 3 Algoritmo de Dijkstra

```
1: Input: Grafo conexo  $G$  rotulado com distâncias e um vértice  $v_0$ .
2: Output: Uma árvore de suporte de Dijkstra  $T$  de  $G$ 
3:  $\text{status}[v_0]=1;$ 
4:  $\text{dist}[v_0]=0;$ 
5:  $\text{prox}[v_0]=\text{nil};$ 
6: for all  $v \neq v_0$  do
7:    $\text{status}[v]=0;$ 
8:    $\text{dist}[v]=\text{dist}(v,v_0);$ 
9:    $\text{prox}[v]=v_0;$ 
10: end for
11: while Existam vértices em  $G$  que não estão na árvore  $T$  do
12:    $v^* \leftarrow$  o vértice de  $G$  mais perto de  $v_0$  que não esteja em  $T$  mas que lhe seja
    adjacente;
13:    $\text{status}[v^*]=1;$ 
14:   for all vértice  $v$  adjacente a  $v^*$  tal que  $\text{status}[v]==0$  do
15:      $\text{dist}=\text{dist}[v^*]+\text{dist}(v^*,v);$ 
16:     if  $\text{dist}<\text{dist}[v]$  then
17:        $\text{dist}[v]=\text{dist};$ 
18:        $\text{prox}[v]=v^*;$ 
19:     end if
20:   end for
21: end while
```

O grafo é representado através da **matriz das distâncias do grafo**, que contem informação relativa ao comprimento de cada aresta. Sempre que não existe arco entre dois vértices indicamos isso definindo o comprimento da ligação como sendo infinita.

O grafo rotulado abaixo:



tem por matriz de distâncias:

$$dist = \begin{bmatrix} 0 & 2 & \infty & \infty & \infty & \infty & \infty & 1 & \infty \\ 2 & 0 & 2 & 2 & 4 & \infty & \infty & \infty & \infty \\ \infty & 2 & 0 & \infty & 3 & \infty & \infty & \infty & 1 \\ \infty & 2 & \infty & 0 & 4 & 3 & \infty & \infty & \infty \\ \infty & 4 & 3 & 4 & 0 & \infty & 7 & \infty & \infty \\ 1 & \infty & \infty & 3 & \infty & 0 & 5 & \infty & \infty \\ \infty & \infty & \infty & \infty & 7 & 5 & 0 & 6 & \infty \\ \infty & \infty & 1 & \infty & \infty & \infty & 6 & 0 & \infty \end{bmatrix}$$

Assim o comprimento do arco \overline{BC} é dado por $dist_{23} = 2$ que simplificamos escrevendo $dist(B, C) = 2$.

O algoritmo é iniciando sob um **vértice inicial**, representado por v_0 , o algoritmo tem por função encontrar o caminho mais curto de cada vértice no grafo para v_0 , sendo encontrado um novo caminho em cada iteração. O vértice terminal do último caminho encontrado é representado por v^* . No algoritmo cada vértice v tem associado três tipos de informação: o seu **status**, representado por $status[v]$; a **menor distância** conhecida a v_0 , representada por $dist[v]$, e o **próximo vértice** no caminho, representada por $prox[v]$. Mais precisamente:

1. $status[v]$ assume o valor 1 para indicar que foi encontrado um caminho mais curto iniciado em v , i.e. para indicar que o vértice v foi incluído na árvore suporte. Este valor é iniciado igual a 0 para indicar que os vértices inicialmente não estão na árvore, exceptuando v_0 que sendo a raiz da árvore é de inicio introduzido na árvore, $status[v_0]=1$.
2. $dist[v]$ representa o comprimento do caminho mais curto de v para v_0 encontrado até ao momento. (Se $dist[v]=\infty$ nenhum caminho foi encontrado até ao momento.)
3. $prox[v]$ indica qual é o próximo vértice no caminho para v_0 . (Se $dist[v]=\infty$, então é porque nenhum caminho foi encontrado, até ao momento, assim $prox[v]$ deve continuar igual a v_0 o valor que foi inicializado. $prox[v]$ assume o valor *nil* para indicar que v é v_0 , o vértice terminal.)

O algoritmo termina quando todos os vértices têm "status" igual a 1, i.e. se os vértices estão todos na árvore. Quando o algoritmo termina para cada vértice v , $dist[v]$ será o comprimento do caminho mais curto de v para v_0 , e os arcos do caminho mais

curto podem ser encontrados seguindo a direcção dada por $\text{prox}[v]$. O próximo exemplo tenta tornar isto mais claro.

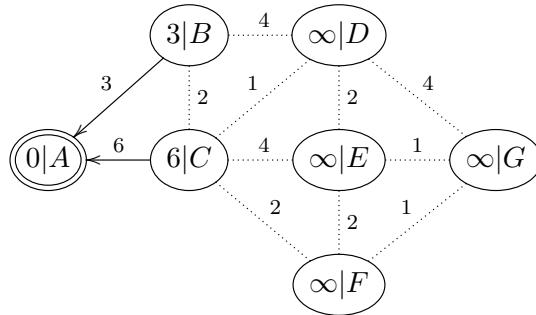
Quando aplicamos o algoritmo ao grafo representado pela matriz de comprimentos:

$$dist = \begin{bmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{bmatrix}$$

Comecemos por executar os passos de inicialização 3-10, onde fazemos v_0 como sendo o vértice A. Após esta inicialização, podemos apresentar a situação através da seguinte tabela:

Vértice	A	B	C	D	E	F	G
status	1	0	0	0	0	0	0
dist	0	3	6	∞	∞	∞	∞
next	nil	A	A	A	A	A	A

Graficamente podemos codificar esta informação através do diagrama:



Onde a distância ao vértice inicial, neste caso A, é indicado dentro do círculo. Os vértices com status 1 são indicados por círculos duplos, caso tenham status 0 são definidos por círculos simples. O vértice que lhe é anterior é definido pela direcção da aresta.

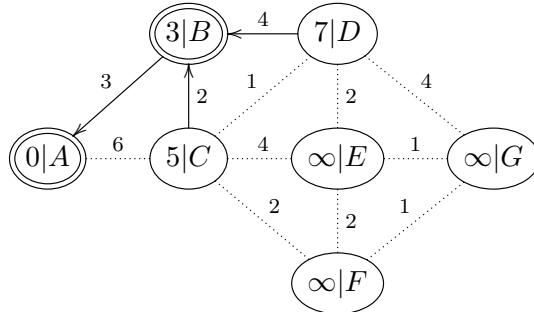
Quando é avaliada a condição da linha 11, do algoritmo, é verificado se existem vértices com status diferente de 1. Neste caso, como existem continuamos. Assim na linha 12 temos $v^* = B$, já que é o vértice que está mais próximo de A. Na linha 13, fazemos $\text{status}[B]=1$, i.e. B passa a ser um vértice que conhecemos o caminho mais curto para A. O ciclo definido pelas linhas 14-20, permite actualizar a informação tendo em conta a ampliação da árvore de suporte: Percorre os vértices que não estão na árvore, i.e. que têm status 0, neste caso C,D,E,F e G, e compara para cada um destes vértices v , o valor de $\text{dist}[v]$ com $\text{dist}[B]+\text{dist}(B,v)$. Podemos esquematizar os resultados através da tabela seguinte:

v	$\text{dist}[v]$	$\text{dist}[B] + \text{dist}(B, v)$	Muda?
C	6	$3+2=5$	Sim
D	∞	$3+4=7$	Sim
E	∞	$3+\infty=\infty$	Não
F	∞	$3+\infty=\infty$	Não
G	∞	$3+\infty=\infty$	Não

Assim após a execução do ciclo *for* em 14, temos:

Vértice	A	B	C	D	E	F	G
status	1	1	0	0	0	0	0
dist	0	3	5	7	∞	∞	∞
next	nil	A	B	B	A	A	A

Graficamente, passamos a ter:

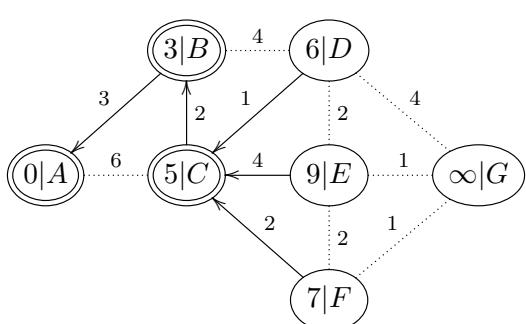


Como existem vértices cujo status é diferente de 0 executamos mais uma vez o ciclo 12-20. Sendo no passo 12 v^* identificado como C, já que é o vértice mais próximo de A. Mudamos o status de C para 1 e o ciclo 14-20 actualiza a informação referentes a vértices adjacentes à árvore com status 0. É assim actualizada a informação de D,E,F e G atendendo a que:

v	$\text{dist}[v]$	$\text{dist}[B] + \text{dist}(B, v)$	Muda?
D	7	$5+1=6$	Sim
E	∞	$5+4=9$	Sim
F	∞	$5+2=7$	Sim
G	∞	$5+\infty=\infty$	Não

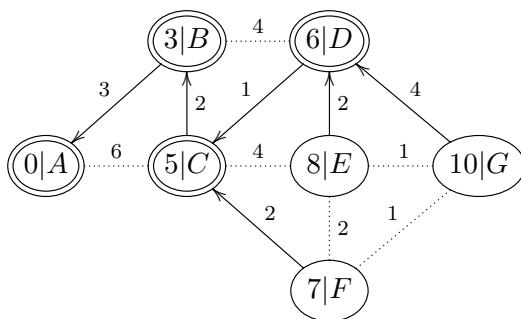
Assim, quando voltamos a avaliar a condição em 11 temos

Vértice	A	B	C	D	E	F	G
status	1	1	1	0	0	0	0
dist	0	3	5	6	9	7	∞
next	nil	A	B	C	C	C	A

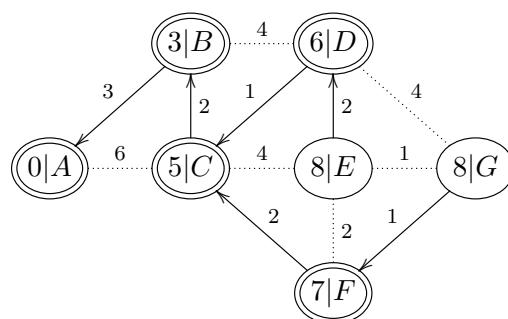


Continuam a haver vértices com status diferente de 1. Após executar o ciclo *while* e voltamos a avaliar 11 temos:

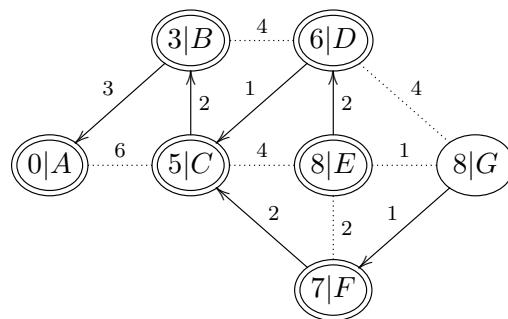
Vértice	A	B	C	D	E	F	G
status	1	1	1	1	0	0	0
dist	0	3	5	6	8	7	10
next	nil	A	B	C	D	C	D



Vértice	A	B	C	D	E	F	G
status	1	1	1	1	0	1	0
dist	0	3	5	6	8	7	8
next	nil	A	B	C	D	C	F

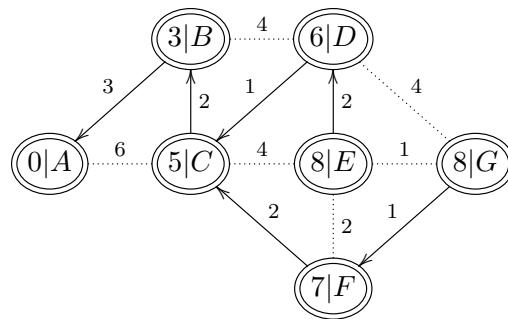


Vértice	A	B	C	D	E	F	G
status	1	1	1	1	1	1	0
dist	0	3	5	6	8	7	8
next	nil	A	B	C	D	C	F



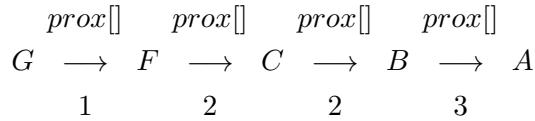
Ainda temos vértices com status diferente de 1. Após executar mais uma vez o ciclo *while* temos:

Vértice	A	B	C	D	E	F	G
status	1	1	1	1	1	1	1
dist	0	3	5	6	8	7	8
next	nil	A	B	C	D	C	F



Finalmente todos os vértices têm status 1, o algoritmo termina. Foram encontrados todos os caminhos mais curtos para A. Por exemplo o caminho mais curto de G para A

tem comprimento 8, uma vez que $\text{dist}[G]=8$, e o caminho pode ser obtido substituindo sucessivamente o valor de $\text{prox}[]$:



Note que, as arestas definidas pela estrutura $\text{prox}[]$ definem um árvore de suporte do grafo original.

Isto completa a nossa descrição do algoritmo de Dijkstra. A razão pela qual o algoritmo trabalha está subjacente ao facto de que a cada execução do ciclo *while* as seguintes proposições são válidas:

1. Para cada vértice v com status igual a 1, $\text{dist}[v]$ é o comprimento do caminho mais curto de v para v_0 , e $\text{next}[v]$ é o primeiro vértice ao longo deste caminho,
2. Para cada vértice v com status igual a 0, $\text{dist}[v]$ é o comprimento do caminho mais curto de v para v_0 definido usando arcos que têm status igual a 0, e $\text{next}[v]$ é o primeiro vértice ao longo desse caminho.

É possível usar indução matemática para mostrar que as proposições são verdadeiras em cada execução do ciclo, possibilitando demonstrar que o algoritmo faz o que é pretendido. No entanto a demonstração é trabalhosa, o que nos leva a omitida-la.

Podemos consolidar o que sabemos sobre o algoritmo de Dijkstra observando que é muito parecido com o algoritmo de Prim. Abaixo descrevemos o algoritmo de Prim em detalhe:

Algoritmo 4 Algoritmo de Prim em detalhe

```
1: Input: Grafo conexo  $G$  rotulado com distância e um vértice  $v_o$ .  $G$ 
2: Output: Uma árvore de suporte de Dijkstra  $T$  de  $G$ 
3:  $\text{status}[v_0]=1;$ 
4:  $\text{custo}[v_0]=0;$ 
5:  $\text{prox}[v_0]=\text{nil};$ 
6: for all  $v \neq v_0$  do
7:    $\text{status}[v]=0;$ 
8:    $\text{dist}[v]=C(v, v_0);$ 
9:    $\text{prox}[v]=v_0;$ 
10: end for
11: while Existam vértices em  $G$  que não estão na árvore  $T$  do
12:    $v^* \leftarrow$  o vértice de status 0 com menor custo e adjacente à árvore;
13:    $\text{status}[v^*]=1;$ 
14:   for all vértice  $v$  adjacente a  $v^*$  tal que  $\text{status}[v]==0$  do
15:      $\text{custo}=C(v^*, v);$ 
16:     if  $\text{custo}<\text{custo}[v]$  then
17:        $\text{custo}[v]=\text{custo};$ 
18:        $\text{prox}[v]=v^*;$ 
19:     end if
20:   end for
21: end while
```

No algoritmo $C(u, v)$ denota o custo do arco que une u a v , e o rótulo dos arcos são interpretados como um custo, em vez dum comprimento. A grande diferença entre os dois algoritmos (para além do nome das variáveis) é o ciclo de actualização 14. São válidos agora a cada execução do ciclo while:

1. Se $\text{status}[v]=1$, v está na árvore. $\text{custo}[v]$ é o custo do arco que foi usado para ligar v à árvore, e $\text{next}[v]$ define o vértice usado na árvore.
2. Se $\text{status}[v]=0$, v ainda não está na árvore. $\text{cost}[v]$ é o custo do arco mais barato que pode ser usado para ligar o vértice à árvore, sendo $\text{next}[v]$ o vértice usado na árvore.

Para clarificar esta formalização do algoritmo de Prim vamos aplicá-lo no grafo definido pela matriz de custos:

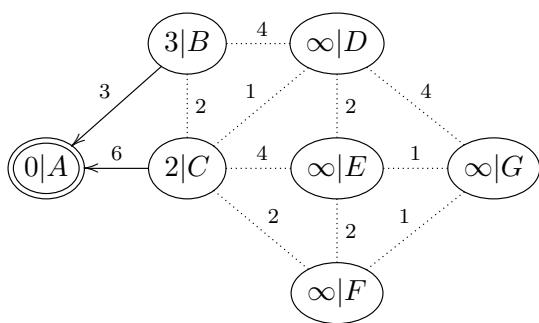
$$C = \begin{bmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{bmatrix}$$

```

graph LR
    A --- B[4]
    A --- C[6]
    B --- C[2]
    B --- D[4]
    C --- D[2]
    C --- E[4]
    C --- F[2]
    D --- E[2]
    D --- G[4]
    E --- G[1]
    E --- F[2]
    F --- G[1]
    F --- E[2]
  
```

Assim $C(u, v)$ denota o custo da aresta que une os vértices u e v . Iniciando o algoritmo em A, após os passos de inicialização 3-10 temos:

Vértice	A	B	C	D	E	F	G
status	1	0	0	0	0	0	0
dist	0	3	6	∞	∞	∞	∞
next	nil	A	A	A	A	A	A

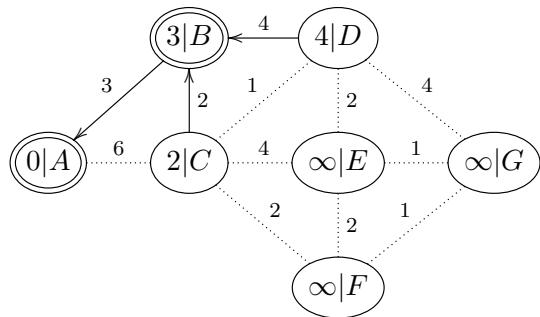


Assim, o vértice B tem associado o menor custo entre os vértices com status 0, mudamos o seu status para 1 em 13 e a actualização os dados dos vértices com status 0 realizada em 14-20 pode ser sumariada na tabela:

v	$\text{custo}[v]$	$C(v, B)$	Muda?
C	6	2	Sim
D	∞	4	Sim
E	∞	$\infty=\infty$	Não
F	∞	$\infty=\infty$	Não
G	∞	$\infty=\infty$	Não

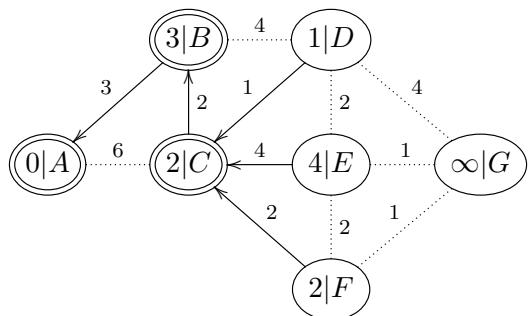
Assim, os vértices C e D são candidatos a expandir a árvore (via B), mudamos assim custo[C] e custo[D] para 2 e 4, respectivamente, e prox[C] e prox[D] para B. Donde obtemos:

Vértice	A	B	C	D	E	F	G
status	1	1	0	0	0	0	0
dist	0	3	2	4	∞	∞	∞
next	nil	A	B	B	A	A	A

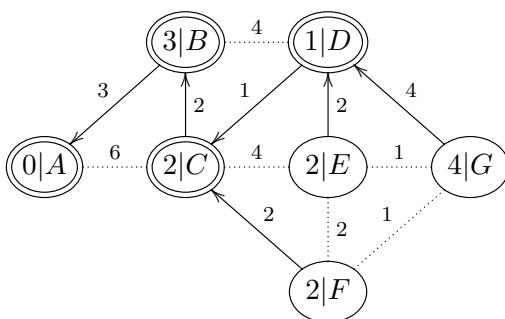


Cada uma das iterações do algoritmo é apresentada abaixo:

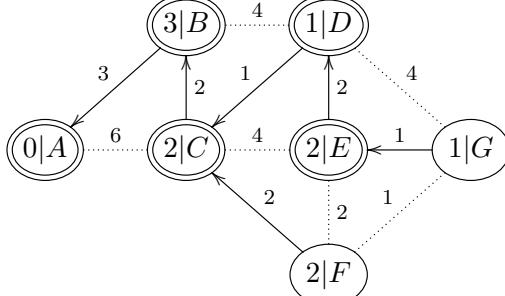
Vértice	A	B	C	D	E	F	G
status	1	1	1	0	0	0	0
dist	0	3	2	1	4	2	∞
next	nil	A	B	C	C	C	A



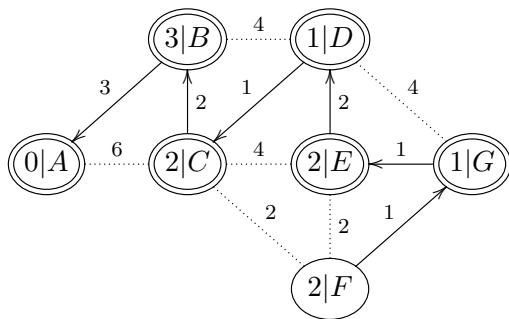
Vértice	A	B	C	D	E	F	G
status	1	1	1	1	0	0	0
dist	0	3	2	1	2	2	4
next	nil	A	B	C	D	C	D



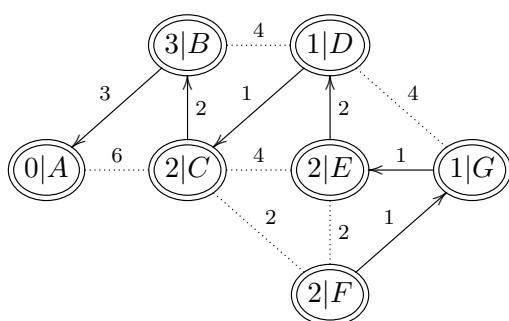
Vértice	A	B	C	D	E	F	G
status	1	1	1	1	1	0	0
dist	0	3	2	1	2	1	1
next	nil	A	B	C	D	G	E



Vértice	A	B	C	D	E	F	G
status	1	1	1	1	1	0	1
dist	0	3	2	1	2	1	1
next	nil	A	B	C	D	G	E



Vértice	A	B	C	D	E	F	G
status	1	1	1	1	1	1	1
dist	0	3	2	1	2	1	1
next	nil	A	B	C	D	G	E

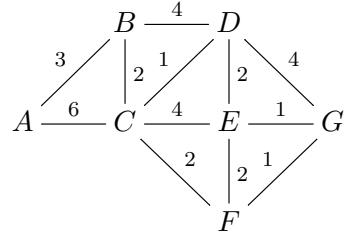


Obtemos assim a árvore de suporte de custo do grafo inicial.

A.2 EXERCÍCIOS DE REVISÃO

1. Aplique o algoritmo de Dijkstra ao grafo cuja matriz de distâncias é:

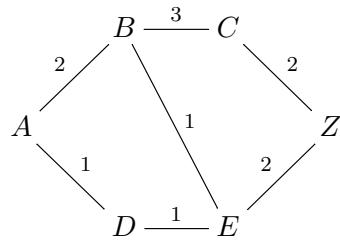
$$C = \begin{bmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 1 \\ \infty & \infty & \infty & 4 & 1 & 1 & 0 \end{bmatrix}$$



Tomando como vértice inicial:

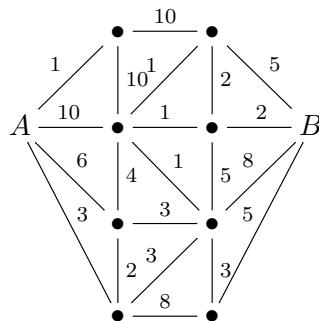
- (a) o vértice B;
- (b) o vértice C;
- (c) o vértice D;
- (d) o vértice E;
- (e) o vértice F;
- (f) o vértice G;
- (g) o vértice H.

2. Considere o grafo abaixo:

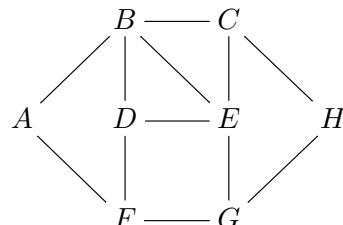


- (a) Usando o algoritmo de Dijkstra (sem necessidade de apresentar o estado de `status[]`, `dist[]` ou `prox[]`) encontre o caminho mais curto de B para cada um dos outros vértices. Liste todos os caminhos encontrados bem como o respectivo comprimento.
- (b) Para comparação, aplique o algoritmo de Prim partindo do vértice B para encontrar a árvore de suporte mínima.
- (c) Repita a alínea a) mas iniciando agora o processo no vértice A.

3. Use o algoritmo de Dijkstra para encontrar o caminho mais curto de A para B no grafo abaixo:



4. Defina rótulos distintos para o grafo abaixo por forma a que existam dois caminhos distintos mais curtos de A para H.



5. Para o grafo definido pela matriz de distâncias abaixo determine, apresentando os valores assumidos por `status[]`, `dist[]` e `prox[]` em cada iteração:

- (a) O caminho mais curto de C para E;

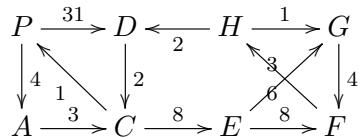
(b) Os caminhos mais curtos para B de vértices do grafo.

$$\begin{bmatrix} 0 & 3 & 6 & \infty & \infty & \infty & \infty \\ 3 & 0 & 2 & 4 & \infty & \infty & \infty \\ 6 & 2 & 0 & 1 & 4 & 2 & \infty \\ \infty & 4 & 1 & 0 & 2 & \infty & 4 \\ \infty & \infty & 4 & 2 & 0 & 2 & 1 \\ \infty & \infty & 2 & \infty & 2 & 0 & 4 \\ \infty & \infty & \infty & 4 & 1 & 4 & 0 \end{bmatrix}$$

6. Para o grafo definido pela matriz de distâncias abaixo determine o caminho mais curto de A para D e o respectivo comprimento, apresentando os valores assumidos por status[], dist[] e prox[] em cada iteração.

$$\begin{bmatrix} 0 & 2 & \infty & \infty & \infty & 1 & \infty & \infty \\ 2 & 0 & 2 & 2 & 4 & \infty & \infty & \infty \\ \infty & 2 & 0 & \infty & 3 & \infty & \infty & 1 \\ \infty & 2 & \infty & 0 & 4 & 3 & \infty & \infty \\ \infty & 4 & 3 & 4 & 0 & \infty & 7 & \infty \\ 1 & \infty & \infty & 3 & \infty & 0 & 5 & \infty \\ \infty & \infty & \infty & \infty & 7 & 5 & 0 & 6 \\ \infty & \infty & 1 & \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

7. Explique como podemos modificar o algoritmo de Dijkstra por forma a encontrar o caminho mais curto em grafos dirigidos. Use esse algoritmo para determinar o menor caminho de P para qualquer outro vértice no grafo dirigido abaixo:



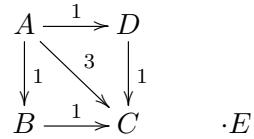
8. Dada a matriz de comprimentos abaixo (definindo um grafo dirigido), encontre o caminho mais curto de P para D e o seu comprimento. Indique os valores de status[], dist[] e prox[] em cada iteração.

$$\begin{bmatrix} 0 & \infty \\ 2 & 0 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 1 & 0 & \infty & \infty & \infty & 4 & \infty \\ \infty & \infty & 3 & 0 & \infty & \infty & 8 & \infty \\ \infty & \infty & \infty & \infty & 0 & 3 & \infty & \infty \\ 2 & \infty & \infty & 8 & \infty & 0 & \infty & \infty \\ \infty & 6 & \infty & \infty & \infty & \infty & 0 & \infty \\ 31 & \infty & \infty & \infty & 4 & \infty & \infty & 0 \end{bmatrix}$$

9. Se, durante a execução do algoritmo de Dijkstra, acabamos por ter nest[C]=F, que

podemos dizer da relação entre $\text{dist}[C]$ e $\text{dist}[F]$?

10. Que acontece ao algoritmo de Dijkstra se não existir caminho entre v_0 e um vértice v ? Por exemplo, que acontece ao algoritmo quando o aplicamos ao grafo abaixo?



11. Explique como podemos modificar o algoritmo de Dijkstra por forma a verificar se um grafo é ou não conexo. Use esta modificação para testar a conectividade do grafo definido pela matriz das distâncias:

$$\begin{bmatrix} 0 & \infty & 4 & 2 & \infty & \infty & 1 & \infty \\ \infty & 0 & \infty & \infty & 3 & 3 & \infty & \infty \\ 4 & \infty & 0 & 2 & \infty & \infty & 3 & 4 \\ 2 & \infty & 2 & 0 & \infty & \infty & \infty & 1 \\ \infty & 3 & \infty & \infty & 0 & 5 & \infty & \infty \\ \infty & 3 & \infty & \infty & 5 & 0 & \infty & \infty \\ 1 & \infty & 3 & \infty & \infty & \infty & 0 & 2 \\ \infty & \infty & 4 & 1 & \infty & \infty & 2 & 0 \end{bmatrix}$$