Capítulo 8

Dicionários

They were standing under a tree, each with an arm round the other's neck, and Alice knew which was which in a moment, because one of them had 'DUM' embroidered on his collar, and the other 'DEE.' 'I suppose they've each got "TWEEDLE" round at the back of the collar,' she said to herself.

Lewis Carroll, *Through the Looking Glass*

8.1 O tipo dicionário

Um *dicionário*, em Python designado por `dict`[1], é um tipo mutável constituído por um conjunto de pares. O primeiro elemento de cada par tem o nome de *chave* e o segundo elemento do par corresponde ao *valor* associado com a chave. Num dicionário não podem existir dois pares distintos com a mesma chave. Esta definição de dicionário é semelhante à definição de função apresentada na página 74, contudo, a utilização que faremos dos dicionários corresponde à manipulação direta do conjunto de pares que carateriza um dado dicionário (recorrendo à *definição por extensão*), em lugar da manipulação da expressão designatória que define esse dicionário. Em programação, o tipo correspondente a um dicionário é normalmente conhecido como *lista associativa* ou *tabela de dispersão*[2].

[1] Do inglês, "dictionary".
[2] Do inglês, "hash table".

Notemos que ao passo que as estruturas de dados que considerámos até agora, os tuplos e as listas, correspondem a sequências de elementos, significando isto que a ordem pela qual os elementos surgem na sequência é importante, os dicionários correspondem a conjuntos de elementos, o que significa que a ordem dos elementos num dicionário é irrelevante.

Utilizando a notação BNF, a representação externa de dicionários em Python é definida do seguinte modo:

⟨dicionário⟩ ::= {} |
{⟨pares⟩}

⟨pares⟩ ::= ⟨par⟩ |
⟨par⟩ , ⟨pares⟩

⟨par⟩ ::= ⟨chave⟩ : ⟨valor⟩

⟨chave⟩ ::= ⟨expressão⟩

⟨valor⟩ ::= ⟨expressão⟩ |
⟨tuplo⟩ |
⟨lista⟩ |
⟨dicionário⟩

Os elementos de um dicionário são representados dentro de chavetas, podendo corresponder a qualquer número de elementos, incluindo zero. O dicionário {} tem o nome de *dicionário vazio*.

O Python exige que a chave seja um elemento de um tipo imutável, tal como um número, uma cadeia de carateres ou um tuplo. Esta restrição não pode ser representada em notação BNF. O Python não impõe restrições ao valor associado com uma chave.

São exemplos de dicionários, {`'a':3, 'b':2, 'c':4`}, {`chr(56+1):12`} (este dicionário é o mesmo que {`'9':12`}), {`(1, 2):'amarelo', (2, 1):'azul'`} e {`1:'c', 'b':4`}, este último exemplo mostra que as chaves de um dicionário podem ser de tipos diferentes. A entidade {`'a':3, 'b':2, 'a':4`} não é um dicionário pois tem dois pares com a mesma chave `'a'`[3].

Como os dicionários são *conjuntos*, quando o Python mostra um dicionário ao utilizador, os seus elementos podem não aparecer pela mesma ordem pela qual

[3]Na realidade, se fornecermos ao Python a entidade {`'a':3, 'b':2, 'a':4`}, o Python considera-a como o dicionário {`'b':2, 'a':4`}. Este aspeto está relacionado com a representação interna de dicionários, a qual pode ser consultada em [Cormen et al., 2009].

foram escritos, como mostra a seguinte interação:

```
>>> {'a':3, 'b':2, 'c':4}
{'a': 3, 'c': 4, 'b': 2}
```

Os elementos de um dicionário são referenciados através do conceito de nome indexado apresentado na página 105. Em oposição aos tuplos e às listas que, sendo sequências, os seus elementos são acedidos por um índice que correspondem à sua posição, os elementos de um dicionário são acedidos utilizando a sua chave como índice. Assim, sendo `d` um dicionário e `c` uma chave do dicionário `d`, o nome indexado `d[c]` corresponde ao valor associado à chave `c`. O acesso e a modificação dos elementos de um dicionário são ilustrados na seguinte interação:

```
>>> ex_dic = {'a':3, 'b':2, 'c':4}
>>> ex_dic['a']
3
>>> ex_dic['d'] = 1
>>> ex_dic
{'a': 3, 'c': 4, 'b': 2, 'd': 1}
>>> ex_dic['a'] = ex_dic['a'] + 1
>>> ex_dic['a']
4
>>> ex_dic['j']
KeyError: 'j'
```

Na segunda linha, `ex_dic['a']` corresponde ao valor do dicionário associado à chave `'a'`. Na quarta linha, é definido um novo par no dicionário `ex_dic`, cuja chave é `'d'` e cujo valor associado é `1`. Na sétima linha, o valor associado à chave `'a'` é aumentado em uma unidade. As últimas duas linhas mostram que se uma chave não existir num dicionário, uma referência a essa chave dá origem a um erro.

Sobre os dicionários podemos utilizar as funções embutidas apresentadas na Tabela 8.1 como se ilustra na interação seguinte.

```
>>> ex_dic
{'a': 4, 'c': 4, 'b': 2, 'd': 1}
>>> 'f' in ex_dic
```

Operação	*Tipo dos argumentos*	*Valor*
`del(`d`[`e`])`	Elemento de dicionário	Remove do dicionário d o elemento com índice e.
c `in` d	Chave e dicionário	`True` se a chave c pertence ao dicionário d; `False` em caso contrário.
c `not in` d	Chave e dicionário	A negação do resultado da operação c `in` d.
`len(`d`)`	Dicionário	O número de elementos do dicionário d.

Tabela 8.1: Operações sobre dicionários em Python.

```
False
>>> 'a' in ex_dic
True
>>> len(ex_dic)
4
>>> del(ex_dic['c'])
>>> ex_dic
{'b': 2, 'a': 4, 'd': 1}
```

Analogamente ao que acontece com tuplos e listas, podemos utilizar um ciclo `for` para percorrer todos os elementos de um dicionário, como o mostra a seguinte interação:

```
>>> inventores = {'Dean Kamen':'Segway',
... 'Tim Berners-Lee': 'World Wide Web',
... 'Dan Bricklin':'Spreadsheet'}
...
>>> for i in inventores:
...     print(i, 'inventou:', inventores[i])
...
Dean Kamen inventou: Segway
Tim Berners-Lee inventou: World Wide Web
Dan Bricklin inventou: Spreadsheet
```

8.2 Contagem de letras

Suponhamos que queremos escrever uma função que recebe uma cadeia de carateres e que determina o número de vezes que cada caráter aparece na cadeia. Esta tarefa é realizada pela função `conta_carateres` que recebe uma cadeia de carateres e devolve um dicionário cujas chaves são carateres, estando cada caráter associado ao número de vezes que este aparece na cadeia.

```
def conta_carateres(cc):
    cont = {}
    for c in cc: # o dicionário é atualizado
        if c not in cont:
            cont[c] = 1
        else:
            cont[c] = cont[c] + 1
    return cont
```

Suponhamos que `Lusiadas` é uma cadeia de carateres contendo o texto completo de *Os Lusíadas*, incluindo os saltos de linha. Por exemplo, para os primeiros 60 carateres desta cadeia, obtemos:

```
>>> Lusiadas[0:60]
'As armas e os barões assinalados,\nQue da ocidental praia Lus'
```

Fornecendo à função `conta_carateres` a cadeia `Lusiadas`, verificamos que o dicionário criado inclui chaves associadas ao caráter de salto de linha (`\n`), ao espaço em branco e a todos os símbolos de pontuação:

```
>>> nc = conta_carateres(Lusiadas)
>>> nc['a']
29833
>>> nc['\n']
9942
>>> nc[' ']
50449
>>> nc['.']
1618
```

Se desejarmos ignorar os saltos de linha, os espaços em branco e os símbolos de pontuação, podemos criar uma lista chamada `pontuacao` que contém os carateres a ignorar[4]:

```
pontuacao = ['!', '"', "'", '(', ')', '-', ',', '.', \
             ';', ':', '?', '[', ']', '{', '}', ' ', '\n']
```

Usando esta lista, podemos alterar o ciclo `for` da nossa função do seguinte modo:

```
def conta_letras(cc):
    pontuacao = ['!', '"', "'", '(', ')', '-', ',', '.', \
                 ';', ':', '?', '[', ']', '{', '}', ' ', '\n']
    cont = {}
    for c in cc:
        if c not in pontuacao:
            # o dicionário é atualizado
            if c not in cont:
                cont[c] = 1
            else:
                cont[c] = cont[c] + 1
    return cont
```

Podendo obter a interação:

```
>>> nc = conta_letras(Lusiadas)
>>> nc['a']
29833
>>> '.' in nc
False
>>> nc['A']
1222
```

A nossa função conta separadamente as letras minúsculas e as letra maiúsculas que aparecem na cadeia, como o mostra a interação anterior. Se desejarmos não fazer a distinção entre letras maiúsculas e minísculas, podemos pensar em

[4]Note-se a a utilização de dois delimitadores para cadeias de carateres, uma contendo as aspas e a outra a plica.

escrever uma função que transforma letras maiúsculas em minúsculas. Esta função recorre ao *Unicode*, calculando qual a letra minúscula correspondente a uma dada maiúscula. Para as letras não acentuadas, esta função é simples de escrever; para símbolos acentuados, esta função obriga-nos a determinar as representações de cada letra maiúscula acentuada, por exemplo "Á", e da sua minúscula correspondente, "á". De modo a que o nosso programa seja completo, teremos também que fazer o mesmo para letras acentuadas noutras línguas que não o português, por exemplo "Ü" e "ü". Uma outra alternativa, à qual vamos recorrer, corresponde a utilizar uma função embutida do Python, a qual para uma dada cadeia de letras, representada pela variável `s`, transforma essa cadeia de carateres numa cadeia equivalente em que todos os carateres correspondem a letras minúsculas. Esta função, sem argumentos tem o nome `s.lower()`[5]. Por exemplo:

```
>>> ex_cadeia = 'Exemplo DE CONVERSÃO'
>>> ex_cadeia.lower()
'exemplo de conversão'
```

Podemos agora escrever a seguinte função que conta o número de letras existentes na cadeia de carateres que lhe é fornecida como argumento, não distinguindo as maiúsculas das minúsculas:

```
def conta_letras(cc):
    pontuacao = ['!', '"', "'", '(', ')', '-', ',', '.', \
                 ';', ':', '?', '[', ']', '{', '}', ' ', '\n']
    cont = {}
    for c in cc.lower(): # transformação em minúsculas
        if c not in pontuacao:
            # o dicionário é atualizado
            if c not in cont:
                cont[c] = 1
            else:
                cont[c] = cont[c] + 1
    return cont
```

[5]O nome desta função segue a sintaxe de ⟨nome composto⟩ apresentada na página 97, mas, na realidade, esta corresponde a um tipo de entidade diferente. Este aspecto é abordado na Secção 11.5. Para os efeitos deste capítulo, iremos considerá-la como uma simples função.

Obtendo a interação:

```
>>> nc = conta_letras(Lusiadas)
>>> nc
{'è': 2, 'ã': 1619, 'é': 823, 'ü': 14, 'ê': 378, 'â': 79, 'ç': 863,
'õ': 115, 'í': 520, 'h': 2583, 'i': 12542, 'j': 1023, 'ó': 660,
'l': 6098, 'm': 10912, 'n': 13449, 'o': 27240, 'à': 158,
'a': 31055, 'b': 2392, 'c': 7070, 'd': 12304, 'e': 31581,
'f': 3055, 'g': 3598, 'x': 370, 'y': 7, 'z': 919, 'ô': 52,
'ú': 163, 'á': 1377, 'p': 5554, 'q': 4110, 'r': 16827, 's': 20644,
't': 11929, 'u': 10617, 'v': 4259, 'ò': 1}
```

Esta interação mostra o que já foi referido na página 237, a ordem por que são mostrados os elementos de um dicionário é escolhida pelo Python, pois um dicionário corresponde a um conjunto. No caso da contagem de carateres, estamos à espera que os resultados sejam ordenados por algum critério, por exemplo, pela sua ordem alfabética. Para obter esse efeito, escrevemos a função `mostra_ordenados`, a qual utiliza a última versão da função `conta_letras` e uma das funções de ordenação, `ordena`, apresentadas na Secção 5.5:

```
def mostra_ordenados(cc):
    res_conta = conta_letras(cc)

    # constroi uma lista de índices e ordena-a
    lista_ind = []
    for c in res_conta:
        lista_ind = lista_ind + [c]
    ordena(lista_ind)

    for c in lista_ind:
        print(c, res_conta[c])
```

Depois da contagem de letras (cujos valores estão guardados no dicionário associado à variável `res_conta`), esta função cria uma lista de todas as letras que existem como chave no resultado da contagem, ordena essa lista e usa essa ordenação para mostrar o número de ocorrências de uma dada letra. A interação seguinte mostra parcialmente o resultado produzido pela função `mostra_ordenados`:

```
>>> mostra_ordenados(Lusiadas)
a 31055
b 2392
c 7070
d 12304
e 31581
...
```

Em muitas aplicações que efetuam contagem de letras, é mais importante a frequência de ocorrência de cada letra do que o valor absoluto da sua ocorrência. Para abordar este tipo de aplicações, vamos escrever a função, `mostra_freq_ordenadas`, que após a contagem das letras na cadeia de carateres que é seu argumento, cria um dicionário, `freqs`, no qual cada índice (letra existente na cadeia de carateres) é associado à sua frequência relativa. Após o cálculo das frequências, a função escreve as letras existentes na cadeia de carateres, ordenadas por ordem decrescente da sua frequência.

Esta última tarefa necessita de alguma explicação adicional. Começamos por criar duas listas paralelas (ver a definição de listas paralelas na página 152) contendo as letras, `lista_ind`, e as respetivas frequências, `lista_freqs`. Recorrendo à ordenação por borbulhamento, a função `ordena_paralelas` ordena estas listas por ordem decrescente das frequências. Finalmente, as listas de índices e frequências, agora ordenadas, são escritas. Como exercício, o leitor deverá compreender o funcionamento da função `ordena_paralelas`.

```
def mostra_freq_ordenadas(cc):
    res_conta = conta_letras(cc)

    # calcula as frequencias
    total_c = 0 # número total de carateres
    for c in res_conta:
        total_c = total_c + res_conta[c]
    freqs = {}  # dicionário com frequências
    for c in res_conta:
        freqs[c] = res_conta[c] / total_c

    # constroi lista de índices e freqs e ordena-a
    lista_ind = []
```

```
    lista_freqs = []
    for c in freqs:
        lista_ind = lista_ind + [c]
        lista_freqs = lista_freqs + [freqs[c]]
    ordena_paralelas(lista_freqs, lista_ind)

    for i in range(len(lista_ind)):
        print(lista_ind[i], lista_freqs[i])

def ordena_paralelas(lst1, lst2):

    maior_indice = len(lst1) - 1
    nenhuma_troca = False # garante que o ciclo while é executado

    while not nenhuma_troca:
        nenhuma_troca = True
        for i in range(maior_indice):
            if lst1[i] < lst1[i+1]:
                lst1[i], lst1[i+1] = lst1[i+1], lst1[i]
                lst2[i], lst2[i+1] = lst2[i+1], lst2[i]
                nenhuma_troca = False
        maior_indice = maior_indice - 1
```

Com as funções anteriores, podemos gerar a interação:

```
>>> mostra_freq_ordenadas(Lusiadas)
e 0.1278779731294693
a 0.12574809079939425
o 0.1103003700974239
s 0.08359180764652052
r 0.06813598853264875
n 0.054457770831140014
...
```

8.3 Dicionários de dicionários

Suponhamos que desejávamos representar informação relativa à ficha académica de um aluno numa universidade, informação essa que deverá conter o número do aluno, o seu nome e o registo das disciplinas que frequentou, contendo para cada disciplina o ano letivo em que o aluno esteve inscrito e a classificação obtida.

Comecemos por pensar em como representar a informação sobre as disciplinas frequentadas. Esta informação corresponde a uma coleção de entidades que associam o nome (ou a abreviatura) da disciplina às notas obtidas e o ano letivo em que a nota foi obtida. Para isso, iremos utilizar um dicionário em que as chaves correspondem às abreviaturas das disciplinas e o seu valor ao registo das notas realizadas. Para um aluno que tenha frequentado as disciplinas de FP, AL, TC e SD, este dicionário terá a forma (para facilidade de leitura, escrevemos cada par numa linha separada):

```
{'FP': ⟨registo das classificações obtidas em FP⟩,
 'AL': ⟨registo das classificações obtidas em AL⟩,
 'TC': ⟨registo das classificações obtidas em TC⟩,
 'SD': ⟨registo das classificações obtidas em SD⟩}
```

Consideremos agora a representação do registo das classificações obtidas numa dada disciplina. Um aluno pode frequentar uma disciplina mais do que uma vez. No caso de reprovação na disciplina, frequentará a disciplina até obter aprovação (ou prescrever); no caso de obter aprovação, poderá frequentar a disciplina para melhoria de nota no ano letivo seguinte. Para representar o registo das classificações obtidas numa dada disciplina, iremos utilizar um dicionário em que cada par contém o ano letivo em que a disciplina foi frequentada (uma cadeia de carateres) e o valor corresponde à classificação obtida, um inteiro entre 10 e 20 ou REP. Ou seja, o valor associado a cada disciplina é por sua vez um dicionário! O seguinte dicionário pode corresponder às notas do aluno em consideração:

```
{'FP': {'2010/11': 12, '2011/12': 15},
 'AL': {'2010/11': 10},
 'TC': {'2010/11': 12},
 'SD': {'2010/11': 'REP', '2011/12': 13}}
```

Se o dicionário anterior tiver o nome `disc`, então podemos gerar a seguinte interação:

```
>>> disc['FP']
{'2010/11': 12, '2011/12': 15}
>>> disc['FP']['2010/11']
12
```

O nome de um aluno é constituído por dois componentes, o nome próprio e o apelido. Assim, o nome de um aluno será também representado por um dicionário com duas chaves, `'nome_p'` e `' apelido'`, correspondentes, respetivamente ao nome próprio e ao apelido do aluno. Assim, o nome do aluno António Mega Bites é representado por:

```
{'nome_p': 'António', 'apelido':'Mega Bites'}
```

Representaremos a informação sobre um aluno como um par pertencente a um dicionário cujas chaves correspondem aos números dos alunos e cujo valor associado é um outro dicionário com uma chave `'nome'` à qual está associada o nome do aluno e com uma chave `'disc'`, à qual está associada um dicionário com as classificações obtidas pelo aluno nos diferentes anos letivos.

Por exemplo, se o aluno Nº. 12345, com o nome António Mega Bites, obteve as classificações indicadas acima, a sua entrada no dicionário será representada por (para facilidade de leitura, escrevemos cada par numa linha separada):

```
{12345: {'nome': {'nome_p': 'António', 'apelido':'Mega Bites'},
         'disc': {'FP': {'2010/11': 12, '2011/12': 15},
                  'AL': {'2010/11': 10},
                  'TC': {'2010/11': 12},
                  'SD': {'2010/11': 'REP', '2011/12': 13}}}}
```

Sendo `alunos` o dicionário com informação sobre todos os alunos da universidade, podemos obter a seguinte interação:

```
>>> alunos[12345]['nome']
{'nome_p': 'António', 'apelido': 'Mega Bites'}
>>> alunos[12345]['nome']['apelido']
'Mega Bites'
>>> alunos[12345]['disc']['FP']['2010/11']
12
```

Vamos agora desenvolver uma função utilizada pela secretaria dos registos académicos ao lançar as notas de uma dada disciplina. O professor responsável da disciplina fornece à secretaria a pauta da disciplina, uma lista contendo como primeiro elemento a identificação da disciplina, o segundo elemento desta lista contém a indicação do ano letivo ao qual as notas correspondem, seguido por uma lista de pares, em que cada par contém o número de um aluno e a nota respetiva. Por exemplo, a seguinte lista poderá corresponder às notas obtidas em FP no ano letivo de 2014/15:

```
['FP', '2014/15', [(12345, 12), (12346, 'REP'), (12347, 10),
 (12348, 14), (12349, 'REP'), (12350, 16), (12351, 14)]]
```

A função `introduz_notas` recebe a pauta de uma disciplina e o dicionário contendo a informação sobre os alunos e efetua o lançamento das notas de uma determinada disciplina, num dado ano letivo. O dicionário contendo a informação sobre os alunos é alterado por esta função.

```
def introduz_notas (pauta, registos):

    disc = pauta[0]
    ano_letivo = pauta[1]

    for num_nota in pauta[2]:
        num, nota = num_nota[0], num_nota[1]
        if num not in registos:
            raise ValueError ('O aluno ' + str(num) + ' não existe')
        else:
            registos[num]['disc'][disc][ano_letivo] = nota
    return registos
```

8.4 Caminhos mais curtos em grafos

Como último exemplo da utilização de dicionários, apresentamos um algoritmo para calcular o caminho mais curto entre os nós de um grafo. Um *grafo* corresponde a uma descrição formal de um grande número de situações do mundo real. Um dos exemplos mais comuns da utilização de grafos corresponde à repre-

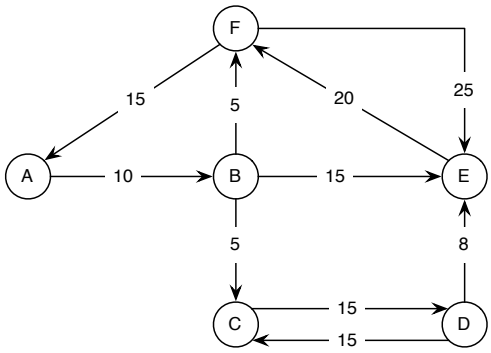

Figura 8.1: Exemplo de um grafo.

sentação de mapas com ligações entre cidades. Podemos considerar um mapa como sendo constituído por um conjunto de *nós*, os quais correspondem às cidades do mapa, e por um conjunto de *arcos*, os quais correspondem às ligações entre as cidades. Formalmente, um grafo corresponde a uma estrutura (N, A), na qual N é um conjunto de entidades a que se dá o nome de *nós* e A é um conjunto de ligações, os *arcos*, entre os nós do grafo. Os arcos de um grafo podem ser *dirigidos*, correspondentes, na analogia com os mapas, a estradas apenas com um sentido, e podem também ser *rotulados*, contendo, por exemplo, a distância entre duas cidades seguindo uma dada estrada ou o valor da portagem a pagar se a estrada correspondente for seguida.

Na Figura 8.1, apresentamos um grafo com seis nós, com os identificadores A, B, C, D, E e F. Neste grafo existe, por exemplo, um arco de A para B, com a distância 10, um arco de C para D, com a distância 15 e um arco de D para C, também com a distância 15. Neste grafo, não existe um arco do nó A para o nó F. Dado um grafo e dois nós desse grafo, n_1 e n_2, define-se um *caminho* de n_1 e n_2 como a sequência de arcos que é necessário atravessar para ir do nó n_1 para o nó n_2. Por exemplo, no grafo da Figura 8.1, um caminho de A para D corresponde ao arcos que ligam os nós A a B, B a C e C a D.

Uma aplicação comum em grafos corresponde a determinar a distância mais curta entre dois nós, ou seja, qual o caminho entre os dois nós em que o somatório das distâncias percorridas é menor. Um algoritmo clássico para determinar a distância mais curta entre dois nós de um grafo foi desenvolvido por Edsger

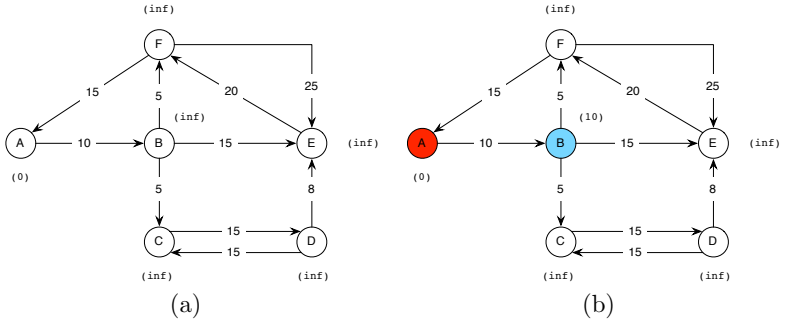

Figura 8.2: Dois primeiros passos no algoritmo de Dijkstra.

Dijkstra[6] (1930–2002).

O algoritmo de Dijkstra recebe como argumentos um grafo e um nó desse grafo, a que se chama o *nó inicial*, e calcula a distância mínima do nó inicial a cada nó do grafo. O algoritmo começa por associar cada nó com a sua distância conhecida ao nó inicial. A esta associação chamamos a *distância calculada*. No início do algoritmo, apenas sabemos que a distância do nó inicial a si próprio é zero, podendo a distância do nó inicial a qualquer outro nó ser infinita (se não existirem ligações entre eles). Por exemplo, em relação ao grafo apresentado na Figura 8.1, sendo A o nó inicial, representaremos as distâncias calculadas iniciais por:

```
{'A':0, 'B':'inf', 'C':'inf', 'D':'inf',  'E':'inf',  'F':'inf'}
```

em que `'inf'` representa infinito. Na Figura 8.2 (a), apresentamos dentro de parênteses, junto a cada nó, a distância calculada pelo algoritmo. No início do algoritmo cria-se também uma lista contendo todos os nós do grafo.

O algoritmo percorre repetitivamente a lista dos nós, enquanto esta não for vazia, executando as seguintes ações:

1. Escolhe da lista de nós, o nó com menor valor da distância calculada. Se existir mais do que um nó nesta situação, escolhe arbitrariamente um deles;

2. Remove o nó escolhido da lista de nós;

[6] [Dijkstra, 1959].

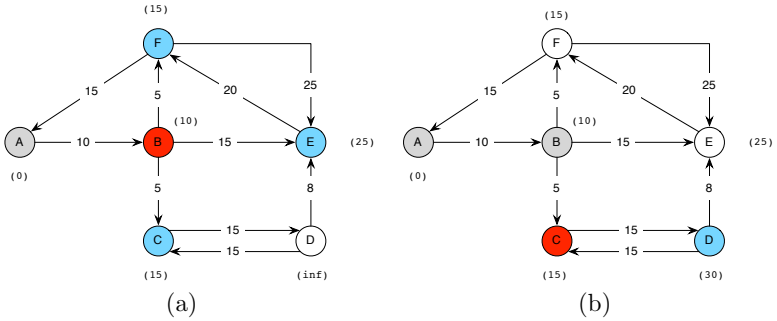

Figura 8.3: Terceiro e quarto passos no algoritmo de Dijkstra.

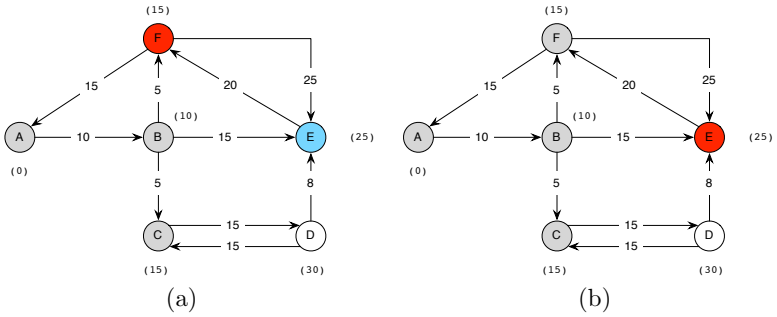

Figura 8.4: Quinto e sexto passos no algoritmo de Dijkstra.

3. Atualiza a distância calculada dos nós adjacentes ao nó escolhido (os nós a que o nó escolhido está diretamente ligado). Esta distância só é atualizada se a nova distância calculada for inferior à distância anteriormente calculada para o nó em causa.

Quando o algoritmo termina, todos os nós estão associados com a distância mínima ao nó original.

Na Figura 8.2 (b), mostramos o passo correspondente a selecionar o primeiro nó, o nó A, atualizando-se a distância calculada para o nó B. Note-se que A não está ligado a F, mas F está ligado a A. Seleciona-se de seguida o nó B, atualizando-se as distâncias calculadas dos nós C, E e F (Figura 8.3 (a)). Existem agora dois nós com a distância calculada de 15, seleciona-se arbitrariamente o nó C e atualiza-se a distância calculada associada ao nó D (Figura 8.3 (b)). Escolhe-se

de seguida o nó F, o qual não leva à atualização de nenhuma distância calculada (Figura 8.4 (a)). O nó E é então escolhido, não levando também à atualização das distâncias calculadas (Figura 8.4 (b)). Finalmente, o nó D é escolhido, não originando a atualização de nenhuma distância calculada.

De modo a poder escrever este algoritmo em Python, teremos que escolher uma representação para grafos. Iremos escolher uma representação que é conhecida por *lista de adjacências*[7], na qual um grafo é representado por um dicionário em que as chaves correspondem aos nós e cada nó está associado a uma lista de tuplos, contendo os nós ligados ao nó em questão e as respetivas distâncias. No nosso exemplo, o nó B está ligado aos nós C, E e F, respetivamente, com as distâncias 5, 15 e 5, pelo que a sua representação será dada pelo par `'B':[('C', 5), ('E', 15), ('F', 5)]`. A representação do grafo da Figura 8.1, que designaremos por `g1`, é:

```
g1 = {'A':[('B', 10)], 'B':[('C', 5), ('E', 15), ('F', 5)],
      'C':[('D', 15)], 'D':[('C', 15), ('E', 8)], 'E':[('F', 20)],
      'F':[('A', 15), ('E', 25)]}
```

A função em Python que corresponde ao algoritmo da distância mínima é:

```
def cdm(g, origem):
    nos = nos_do_grafo(g)
    dc = dist_inicial(nos, origem)
    while nos != []:
        n = dist_min(nos, dc) # seleciona o nó com menor distância
        remove(nos, n)
        for nn in g[n]: # para os nós a que nn está ligado
            atualiza_dist(n, nn[0], nn[1], dc)
    return dc
```

A função `nos_do_grafo` devolve a lista dos nós do grafo que é seu argumento. Note-se a utilização de um ciclo `for` que, quando aplicado a um dicionário itera sobre as chaves do dicionário, tal como descrito na página 238.

```
def nos_do_grafo(g):
    res = []
```

[7] Do inglês, "adjacency list".

```
    for el in g:
        res = res + [el]
    return res
```

A função `dist_inicial` recebe como argumentos a lista dos nós do grafo e o nó inicial e devolve o dicionário que corresponde à distância inicial dos nós do grafo à origem, ou seja, zero para a origem e `'inf'` para os restantes nós. Para o grafo do nosso exemplo, esta função devolve o dicionário apresentado na página 249.

```
def dist_inicial (nos, origem):
    res = {}
    for n in nos:
        res[n] = 'inf'
    res[origem] = 0
    return res
```

A função `dist_min` recebe a lista dos nós em consideração (`nos`) e o dicionário com as distâncias calculadas (`dc`) e devolve o nó com menor distância e a função `remove` remove o nó que é seu argumento da lista dos nós. Note-se que esta função altera a lista `nos`.

```
def dist_min(nos, dc):
    n_min = nos[0]    # o primeiro nó
    min = dc[n_min]   # a distância associada ao primeiro nó

    for n in nos:
        if menor(dc[n], min):
            min = dc[n]  # a distância associada ao nó n
            n_min = n
    return n_min

def remove(nos, n):
    for nn in range(len(nos)):
        if nos[nn] == n:
            del nos[nn]
            return
```

A função `atualiza_dist` recebe um nó de partida (`no_partida`), um nó a

que o `no_partida` está diretamente ligado (`no_chegada`), a distância entre o `no_partida` e o `no_chegada` e as distâncias calculadas (`dc`) e atualiza, nas distâncias calculadas, a distância associada ao `no_chegada`.

```
def atualiza_dist(no_partida, no_chegada, dist, dc):
    dist_no_partida = dc[no_partida]
    if menor(soma(dist_no_partida, dist), dc[no_chegada]):
        dc[no_chegada] = dist_no_partida + dist
```

Finalmente, as funções `menor` e `soma`, respetivamente comparam e somam dois valores, tendo em atenção que um deles pode ser `'inf'`.

```
def menor(v1, v2):
    if v1 == 'inf':
        return False
    elif v2 == 'inf':
        return True
    else:
        return v1 < v2

def soma(v1, v2):
    if v1 == 'inf' or v2 == 'inf':
        return 'inf'
    else:
        return v1 + v2
```

Usando a função `cdm` e o grafo apresentado na Figura 8.1, podemos originar a seguinte interação:

```
>>> cdm(g1, 'A')
{'A': 0, 'C': 15, 'B': 10, 'E': 25, 'D': 30, 'F': 15}
>>> cdm(g1, 'B')
{'A': 20, 'C': 5, 'B': 0, 'E': 15, 'D': 20, 'F': 5}
```

8.5 Notas finais

Neste capítulo apresentámos o tipo dicionário, o qual corresponde ao conceito de lista associativa ou tabela de dispersão. Um dicionário associa chaves a valores, sendo os elementos de um dicionário indexados pela respetiva chave. Um dos aspetos importantes na utilização de dicionários corresponde a garantir que o acesso aos elementos de um dicionário é feito a tempo constante. Para estudar o modo como os dicionários são representados internamente no computador aconselhamos a consulta de [Cormen et al., 2009].

Apresentámos, como aplicação de utilização de dicionários, o algoritmo do caminho mais curto em grafos. O tipo grafo é muito importante em informática, estando para além da matéria deste livro. A especificação do tipo grafo e algumas das suas aplicações podem ser consultadas em [Dale and Walker, 1996]. Algoritmos que manipulam grafos podem ser consultados em [McConnell, 2008] e em [Cormen et al., 2009].

8.6 Exercícios

1. Considere a seguinte atribuição:

```
teste = {'Portugal':{'Lisboa':[('Leonor', '1700-097'),
                               ('João', '1050-100')],
                     'Porto':[('Ana', '4150-036')]},
         'Estados Unidos':{'Miami':[('Nancy', '33136'),
                                    ('Fred', '33136')],
                           'Chicago':[('Cesar', '60661')]},
         'Reino Unido':{'London':[('Stuart', 'SW1H 0BD')]}}
```

   Qual o valor de cada um dos seguintes nomes? Se algum dos nomes originar um erro, explique a razão do erro.

   (a) `teste['Portugal']['Porto']`

   (b) `teste['Portugal']['Porto'][0][0]`

   (c) `teste['Estados Unidos']['Miami'][1]`

   (d) `teste['Estados Unidos']['Miami'][1][0][0]`

   (e) `teste['Estados Unidos']['Miami'][1][1][1]`

2. Considere a seguinte lista de dicionários na qual os significados dos campos são óbvios:

```
l_nomes = [{'nome':{'nomep':'Jose', 'apelido':'Silva'},
'morada':{'rua':'R. dos douradores', 'num': 34, 'andar':'6 Esq',
'localidade':'Lisboa', 'estado':'', 'cp':'1100-032',
'pais':'Portugal'}}, {'nome':{'nomep':'John', 'apelido':'Doe'},
'morada':{'rua':'West Hazeltine Ave.', 'num': 57, 'andar':'',
'localidade':'Kenmore', 'estado':'NY', 'cp':'14317', 'pais':'USA'}}]
```

   Diga quais são os valores dos seguintes nomes:

   (a) `l_nomes[1]`

   (b) `l_nomes[1]['nome']`

   (c) `l_nomes[1]['nome'][apelido]`

   (d) `l_nomes[1]['nome'][apelido][0]`

3. Uma carta de jogar é caracterizada por um naipe (espadas, copas, ouros e paus) e por um valor (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, K). Uma carta pode ser representada por um dicionário com duas chaves, `'np'` e `'vlr'`, sendo um conjunto de cartas representado por uma lista de cartas.

   (a) Escreva uma função em Python que devolve uma lista contendo todas as cartas de um baralho. Por exemplo,

```
>>> baralho()
[{'np': 'esp', 'vlr': 'A'}, {'np': 'esp', 'vlr': '2'},
{'np': 'esp', 'vlr': '3'}, {'np': 'esp', 'vlr': '4'},
{'np': 'esp', 'vlr': '5'}, {'np': 'esp', 'vlr': '6'},
{'np': 'esp', 'vlr': '7'}, {'np': 'esp', 'vlr': '8'},
{'np': 'esp', 'vlr': '9'}, {'np': 'esp', 'vlr': '10'},
{'np': 'esp', 'vlr': 'J'}, {'np': 'esp', 'vlr': 'Q'},
{'np': 'esp', 'vlr': 'K'}, {'np': 'copas', 'vlr': 'A'},
{'np': 'copas', 'vlr': '2'}, {'np': 'copas', 'vlr': '3'},
... }
```

   (b) Recorrendo à função `random()`, a qual produz um número aleatório no intervalo $[0, 1[$, escreva a função `baralha`, que recebe uma lista correspondente a um baralho de cartas e baralha aleatoriamente essas cartas, devolvendo a lista que corresponde às cartas baralhadas.

SUGESTÃO: percorra sucessivamente as cartas do baralho trocando cada uma delas por uma outra carta seleccionada aleatoriamente. Por exemplo,

```
>>> baralha(baralho())
[{'np': 'esp', 'vlr': '3'}, {'np': 'esp', 'vlr': '9'},
{'np': 'copas', 'vlr': '6'}, {'np': 'esp', 'vlr': 'Q'},
{'np': 'esp', 'vlr': '7'}, {'np': 'copas', 'vlr': '8'},
{'np': 'copas', 'vlr': 'J'}, {'np': 'esp', 'vlr': 'K'},
... ]
```

(c) Escreva uma função em Python que recebe um baralho de cartas e as distribui por quatro jogadores, devolvendo uma lista que contém as cartas de cada jogador. Por exemplo,

```
distribui(baralha(baralho()))
[[{'np': 'ouros', 'vlr': 'A'}, {'np': 'copas', 'vlr': '7'},
{'np': 'paus', 'vlr': 'A'}, {'np': 'esp', 'vlr': 'J'},
{'np': 'paus', 'vlr': '6'}, {'np': 'esp', 'vlr': '10'},
{'np': 'copas', 'vlr': '5'}, {'np': 'esp', 'vlr': '6'},
{'np': 'copas', 'vlr': '8'}, {'np': 'esp', 'vlr': '3'},
 {'np': 'ouros', 'vlr': '5'}, {'np': 'ouros', 'vlr': '8'},
{'np': 'copas', 'vlr': 'K'}],
[{'np': 'paus', 'vlr': '2'}, {'np': 'esp', 'vlr': '2'},
...]]
```

4. Escreva uma função que recebe uma cadeia de caracteres correspondente a um texto e que produz uma lista de todas as palavras que este contém, juntamente com o número de vezes que essa palavra aparece no texto. SUGESTÃO: guarde cada palavra como uma entrada num dicionário contendo o número de vezes que esta apareceu no texto. por exemplo,

```
>>> cc = 'a aranha arranha a ra a ra arranha a aranha ' \
     + 'nem a aranha arranha a ra nem a ra arranha a aranha'
>>>conta_palavras(cc)
{'aranha': 4, 'arranha': 4, 'ra': 4, 'a': 8, 'nem': 2}
```

5. Usando a ordenação por borbulhamento, escreva a função `mostra_ordenado` que apresenta por ordem alfabética os resultados produzidos pelo exercício anterior. Por exemplo,

```
>>> mostra_ordenado(conta_palavras(cc))
a 8
aranha 4
arranha 4
nem 2
ra 4
```

6. Suponha que o índice de um livro é representado por um dicionário em que os títulos dos capítulos são as chaves e os seus valores correspondem a uma lista contendo a primeira e a última página do capítulo. Suponha também que o índice remissivo é representado por um dicionário em que cada palavra está associada às páginas em que aparece (representadas por uma lista). Escreva um programa em Python que recebe um índice e um índice remissivo e que devolve um dicionário em que cada palavra no índice remissivo é associada aos títulos dos capítulos em que aparece. Nota: uma palavra não deve ser associada duas vezes ao mesmo capítulo. Por exemplo, sendo

```
ind = {'c1': [1, 34], 'c2': [35, 42], 'c3': [43, 52],
       'c4': [53, 60]}
i_r = {'p1': [2, 54], 'p2': [36, 37, 50], 'p3': [40]}
```

então

```
>>> caps(ind, i_r)
{'p3': ['c2'], 'p2': ['c2', 'c3'], 'p1': ['c1', 'c4']}
```

7. Suponha que `bib` é uma lista cujos elementos são dicionários e que contém a informação sobre os livros existentes numa biblioteca. Cada livro é caracterizado pelo seus autores, título, casa editora, cidade de publicação, ano de publicação, número de páginas e ISBN. Por exemplo, a seguinte lista corresponde a uma biblioteca com dois livros:

```
[{'autores': ['G. Arroz', 'J. Monteiro', 'A. Oliveira'],
'titulo': 'Arquitectura de computadores', 'editor': 'IST Press',
'cidade': 'Lisboa', 'ano': 2007, 'numpags': 799,
'isbn': '978-972-8469-54-2'}, {'autores': ['J.P. Martins'],
'titulo': 'Logica e Raciocinio', 'editor': 'College Publications',
'cidade': 'Londres', 'ano': 2014, 'numpags': 438,
'isbn': '978-1-84890-125-4'}]
```

(a) Escreva um programa em Python que recebe a informação de uma biblioteca e devolve o número médio de páginas dos livros da biblioteca. Por exemplo:

```
>>> media_pags(bib)
618.5
```

(b) Escreva um programa em Python que recebe a informação de uma biblioteca e devolve o túlulo do livro mais antigo. Por exemplo:

```
>>> mais_antigo(bib)
'Arquitectura de computadores'
```

8. Uma matriz é dita *esparsa* (ou rarefeita) quando a maior parte dos seus elementos é zero. As matrizes esparsas aparecem em grande número de aplicações em engenharia. Uma matriz esparsa pode ser representada por um dicionário cujas chaves correspondem a tuplos que indicam a posição de um elemento na matriz (linha e coluna) e cujo valor é o elemento nessa posição da matriz. Por exemplo, {`(3, 2): 20, (150, 2): 6, (300, 10): 20`} corresponde a uma matriz esparsa com apenas três elementos diferentes de zero.

(a) Escreva uma função em Python que recebe uma matriz esparsa e a escreve sob a forma, na qual os elementos cujo valor é zero são explicitados.

$$\begin{matrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{matrix}$$

```
>>> escreve_esparsa({(1,5): 4, (2, 3): 9, (4, 1): 1})
0  0  0  0  0  0
0  0  0  0  0  4
0  0  0  9  0  0
0  0  0  0  0  0
0  1  0  0  0  0
```

(b) Escreva uma função em Python que recebe duas matrizes esparsa e devolve a sua soma. Por exemplo,

```
e1 = {(1,5): 4, (2, 3): 9, (4, 1): 1}
```

```
e2 = {(1, 6): 2, (4, 1): 2, (5,4): 2}
>>> escreve_esparsa(soma_esparsa(e1, e2))
0  0  0  0  0  0  0
0  0  0  0  0  4  2
0  0  0  9  0  0  0
0  0  0  0  0  0  0
0  3  0  0  0  0  0
0  0  0  0  2  0  0
```

9. Escreva uma função em Python que recebe um dicionário cujos valores associados às chaves correspondem a listas de inteiros e que devolve o dicionário que se obtém "invertendo" o dicionário recebido, no qual as chaves são os inteiros que correspondem aos valores do dicionário original e os valores são as chaves do dicionário original às quais os valores estão associados. Por exemplo,

```
>>> inverte_dic({'a': [1, 2], 'b': [1, 5], 'c': [9], 'd': [4]})
{1: ['a', 'b'], 2: ['a'], 4: ['d'], 5: ['b'], 9: ['c']}
```

10. Um tabuleiro de xadrez tem 64 posições organizadas em 8 linhas e 8 colunas. As linhas são numeradas de `1` a `8` e as colunas de `A` a `H`, sendo a posição inferior esquerda a `1`, `A`. Neste tabuleiro são colocadas peças de duas cores (brancas e pretas) e de diferentes tipos (rei, rainha, bispo, torre, cavalo e peão). Um tabuleiro de um jogo de xadrez pode ser representado por um dicionário cujos elementos são do tipo `(`l`, `c`):  (`cor`, `t`)`. Por exemplo o elemento do dicionário `(2, C): (branca, rainha)` indica que a rainha branca está na segunda linha, terceira coluna.

    A situação do jogo de xadrez apresentado na Figura 8.5 é representada pelo seguinte dicionário:

```
j = {(1, 'H'): ('branca', 'torre'), (2, 'F'): ('branca', 'peao'), \
     (2, 'G'): ('branca', 'rei'), (6, 'F'): ('branca', 'bispo'), \
     (5, 'C'): ('branca', 'rainha'), (6, 'G'): ('preta', 'peao'), \
     (7, 'F'): ('preta', 'peao'), (8, 'F'): ('preta', 'torre'), \
     (8, 'G'): ('preta', 'rei'), (2, 'C'): ('preta', 'peao')}
```

    Num jogo de xadrez, a rainha movimenta-se na vertical, na horizontal ou nas diagonais e pode atacar qualquer peça da cor contrária que possa ser atingida num dos seus movimentos, desde que não existam outras peças

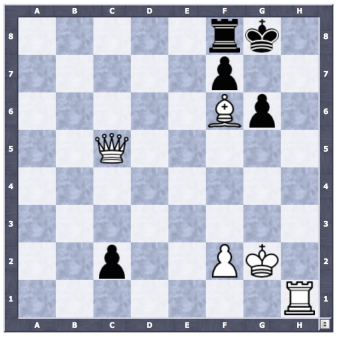

Figura 8.5: Tabuleiro de xadrez.

no caminho. Escreva uma função em Python que recebe um tabuleiro de xadrez e determina quais as peças que podem ser atacadas pelas rainhas. Por exemplo,

```
>>> ataques_rainhas(j)
[['peao', (2, 'C')], ['torre', (8, 'F')]]
```

11. Tendo em conta a representação de um tabuleiro de xadrez apresentada no exercício anterior, escreva um programa que recebe a posição de um cavalo e determina os seus movimentos possíveis. Um cavalo move-se andando duas posições para a frente e uma para o lado. Note-se que um cavalo pode mover-se para uma casa onde exista uma peça do adversário mas não se pode mover para uma casa em que exista uma peça com a sua cor.