Projeto SO

Gestão de armazenamento: Monitorização do espaço ocupado

Pedro Pinto nº115304

Jorge Domingues nº113278

Resumo

O objetivo deste trabalho foi desenvolver componentes capazes de monitorizar memória em disco, componentes essenciais em todos os sistemas de ficheiros. Estes algoritmos permitem a monitorização do espaço ocupado em disco, bem como a sua variação ao longo do tempo, nos diretórios fornecidos como argumento, incluindo os seus subdiretórios. A implementação foi realizada em Bash, sendo a saída apresentada no terminal de execução. O projeto foi estruturado em módulos e foram feitas várias simulações e testes para garantir o correto funcionamento do sistema.

Keywords: *Sistema Operativo; Sistema de Ficheiros; Bash; Shell; Monitorização do espaço ocupado; Variação do espaço em disco no tempo*

Conteúdo

1 Introdução **3**
1.1 Spacecheck . . . 3
1.2 Spacerate . . . 3

2 A solução spacecheck.sh **4**
2.1 Ideias . . . 4
2.2 Implementação . . . 4
2.2.1 Seleção de diretórios . . . 4
2.2.2 Processamento de diretórios . . . 4
2.3 Opções de saída . . . 5
2.3.1 Opções de ordenação . . . 5
2.3.2 Opções de limite . . . 6
2.4 Validação e tratamento de argumentos . . . 6
2.5 Funções auxiliares . . . 6
2.6 Testes . . . 7
2.6.1 Testes de saída . . . 7
2.6.2 Testes de erro . . . 8

3 A solução spacerate.sh **9**
3.1 Ideias . . . 9
3.2 Implementação . . . 9
3.2.1 Guardar conteúdo . . . 9
3.2.2 Conversão em somas não acumulativas . . . 9
3.2.3 Processamento de Variações . . . 10
3.3 Opções de ordenação . . . 10
3.4 Validação e tratamento de argumentos . . . 11
3.5 Funções auxiliares . . . 11
3.6 Testes . . . 11
3.6.1 Testes de saída . . . 11
3.6.2 Testes de erro . . . 12

4 Conclusão **13**

5 Bibliografia **13**

1. Introdução

No âmbito da disciplina Sistemas Operativos, este projeto tem como objetivo principal a implementação de dois "módulos" complementares. Estas ferramentas permitem a monitorização do espaço ocupado em disco e a sua variação ao longo do tempo, por ficheiros com determinadas propriedades, simplificando assim a gestão de armazenamento. Como ponto de partida, utilizámos os comandos Bash que foram ensinadas nas aulas teóricas e práticas, bem como os comandos recomendados no enunciado do trabalho.

1.1. Spacecheck

Este componente é responsável pela visualização do espaço ocupado pelos ficheiros selecionados nos diretórios que lhe são passados como argumento e em todos os subdiretórios destes. A visualização é afetada pelas opções solicitadas na execução do código:

Opção -n <regex>: Seleção de ficheiros através de expressão regular e nome

Opção -d <date>: Data máxima de modificação dos ficheiros

Opção -s <size>: Indicação de tamanho mínimo de ficheiro

Opção -l <limit>: Limitar o número de linhas da tabela

Opção -r: Ordenação inversa

Opção -a: Ordenação alfabética

$./spacecheck\ [-d\ <date>]\ [-s\ <size>]\ [-l\ <limit>]\ [-r]\ [-a]\ [-n\ <regex>]\ [<directories>]$

1.2. Spacerate

Este componente é responsável pela comparação de dois ficheiros resultantes da execução da função spacecheck.sh, com o intuito de possibilitar a visualização da evolução do espaço ocupado entre os diversos diretórios nele contidos. O processo inclui a identificação dos diretórios comuns em ambos os ficheiros, apresentando a diferença entre os espaços ocupados, bem como a deteção de acréscimos ou eliminações de diretórios. A visualização é afetada pelas opções solicitadas na execução do código:

Opção -r: Ordenação inversa

Opção -a: Ordenação alfabética

$./spacerate\ [-r]\ [-a]\ [new\_file]\ [old\_file]$

2. A solução spacecheck.sh

2.1. Ideias

Primeiramente, concentrámo-nos na busca de soluções para identificar diretórios e subdiretórios, bem como na extração das operações requeridas a partir da linha de comando. Posteriormente, elaboramos o procedimento para o cálculo dos bytes, levando em consideração as solicitações apresentadas. Posto tudo isto, também devemos destacar que ao longo do desenvolvimento do projeto foram utilizados valores padrão para as opções do utilizador. Fizemos uso de uma função do sistema Unix, $2 > /dev/null$, que é utilizada para redirecionar a saída de erro para o dispositivo /dev/null, permitindo descartar efetivamente quaisquer mensagens de erro usadas na própria Bash, facilitando assim a implementação das nossas próprias funções de erro. Além disso, para padronizar o formato de data utilizado no programa, foi configurado o *LC_ TIME=en_ US.utf8*, que faz com que o sistema adote o formato de data e hora dos Estados Unidos.

2.2. Implementação

2.2.1. Seleção de diretórios

A seleção de diretórios é o primeiro passo na realização deste módulo. Consiste em validar os diretórios passados em argumento e guardá-los em uma estrutura de dados adequada. Neste subcapítulo iremos explorar um ponto de vista mais prático, para isso, deve entender que o valor padrão já foi definido como *directories = ".".*

Validação de diretórios: Caso exista diretórios passados nos argumentos, o valor padrão é ignorado e passamos à leitura dos argumentos. Em cada argumento verificamos a validade do diretório e terminamos a execução do programa no caso do diretório ser inválido com a função auxiliar *invalidDirectory()*, Figura 6. Realizamos também uma verificação para retirar as "/" (à direita) dos diretórios, para padronizar os diretórios inseridas pelo utilizador, Figura 1. Após o diretório ser validado adicionamos-lo a uma estrutura de dados que irá guardar todos os diretórios. Como esta validação é bastante complexa procedemos a exaustivos testes para confirmar a sua correta execução (ver testes executados no Capítulo 2.6).

```
87 if [[ "$#" -ge 1 ]]
88 then
89   directories=()
90   for dir in "$@"; do
91     [[ -d "$dir" ]] || invalidDirectory "$dir" # check: valid directory
92     [[ "$dir" = "/" ]] || [[ "$dir" = "//" ]] || dir=$(echo "$dir" | sed 's:/*$::')
93     directories+=("$dir")
94   done
95 fi
```

Figura 1. Validação de diretórios.

Mapeamento de subdiretórios: Neste ponto, procedemos a um mapeamento dos diretórios validados anteriormente. Para isso, utilizamos os comandos Bash *find* e *sort*, que ajudam a armazenar numa estrutura de dados sem repetição todos os diretórios e subdiretórios, Figura 2. Esta estrutura garante a não repetição de diretórios (opção $-u$ do comando *sort*) pois, no caso do utilizador colocar diretórios com alguma relação de herança, não deve ser analisado nenhum diretório mais do que uma vez, (ver testes executados no Capítulo 2.6).

```
97 # all folders in directory
98 folders=$(find "${directories[@]}" -type d 2>/dev/null | sort -u)
```

Figura 2. Mapeamento de subdiretórios.

2.2.2. Processamento de diretórios

A análise de diretórios é a parte que envolve mais esforços computacionais e uma maior complexidade. Para além disso, é uma função que obriga o entendimento geral do funcionamento do programa para ser bem compreendida por parte do leitor. Para facilitar a sua compreensão iremos explicar de uma forma faseada.

Ponto de partida: Neste momento, dispomos de uma estrutura de dados contendo todos os diretórios a serem analisados, *folders*. Aplicaremos modelos a cada diretório para calcular a quantidade de bytes que o diretório e seus subdiretórios possuem em ficheiros que atendam às condições especificadas pelo utilizador. Será necessário percorrer todos os arquivos correspondentes, realizar as validações pertinentes e efetuar os cálculos necessários para obter os valores desejados. Neste subcapítulo, considere que o valor padrão de cada opção é utilizado no caso do utilizador não selecionar a respetiva opção, reduzindo assim a complexidade do código (ver Capítulo 2.1).

Permissões de diretórios: Para garantir o funcionamento adequado do sistema, é crucial considerar casos adversos, como a possibilidade de não conseguir visualizar totalmente o espaço ocupado por um diretório devido à falta de permissões. Assim, para abranger todos os diretórios, verificamos se é possível executar $[-x]$ e ler $[-r]$ esse diretório. No caso de não conseguirmos, o número de bytes definidos para o diretório é marcado como NA[1]. Compreendemos que, tendo este diretório tamanho indefinido, os diretórios pai também não têm o seu valor definido. Porém, para este projeto, achámos mais adequado apenas marcar como NA o diretório sem permissão. Sendo assim, o diretório de maior hierarquia irá somar apenas os diretórios que têm permissão e vai ignorar todos aqueles que não a têm.

Soma de ficheiros: Com base nas opções e argumentos do utilizador é realizada a soma de todos os ficheiros contidos no diretório e nos seus subdiretórios. Para isso, é utilizado o comando $find$ com opções específicas para selecionar os ficheiros que atendem a critérios como data máxima de modificação, regex/nome e tamanho mínimo especificados pelo utilizador, cobrindo assim todas as opções de filtro disponíveis. Por outro lado, a soma dos ficheiros é feita usando a combinação do comando $find\ -exec$ com o comando du, que fornece a quantidade de bytes ocupados. Por fim, o resultado é processado com o $tail$ e awk para extrair o valor desejado que veio dos comandos anteriores (zero no caso de nulidade).

```
107 # Analise of folders
108 while IFS= read -r f; do
109
110   if [[ -x "$f" ]] && [[ -r "$f" ]]; then
111     bytes=$(find "$f" -not -newermt "$date_ref" -type f \( -regex "$name_exp" -o -name "
      $name_exp" \) -not -size -"$size"c  -exec du -bc {} + 2>/dev/null | tail -n 1 | awk '{
      print $1}')
112     [[ -z "$bytes" ]] && bytes=0
113     echo "$bytes" "$f"
114   else
115     echo "NA" "$f"
116   fi
117
118 done <<< "$folders" | sort $sort_option | head -n "$limit_lines"
```

Figura 3. Processamento de diretórios.

2.3. Opções de saída

Após o processamento de diretórios, é necessário apresentar a informação de forma clara e conforme solicitado pelo utilizador. A solução que desenvolvemos envolve a utilização de um *pipe* | após o uso do comando *echo* para o número de bytes correspondente a cada diretório, Figura 3. Esta abordagem simplifica significativamente a ordenação $-a$ e $-r$ e o limite $-l$ definidos pelo utilizador, já que agora só precisamos de ajustar corretamente as variáveis *sort_ option* e *limit_ lines*.

2.3.1. Opções de ordenação

$$sort\_option = \begin{cases} -k1,1nr, & [\varnothing] \\ -k1,1n\ -k1,1r\ -k2\ -r, & [-r] \\ -k1,1n\ -k1,1r\ -k2, & [-r\ ,\ -a] \\ -k2, & [-a] \\ -k2\ -r, & [-a\ ,\ -r] \end{cases} \tag{1}$$

[1] NA: Not Available

A nossa abordagem segue o conceito de prioridade de chegada, onde o primeiro argumento é considerado principal e o segundo, secundário. Assim, aplicando o comando *sort* e considerando a fórmula (1), obtemos, quando o utilizador não especifica opções de ordenação [∅], uma ordenação por ordem decrescente de bytes (a ordenação padrão). Quando o utilizador escolhe a ordem inversa [$-r$] temos uma ordenação por ordem crescente de bytes, e para garantir que a saída seja exatamente o inverso da ordenação padrão, consideramos, em caso de empate, uma ordenação alfabética inversa (sem esquecer os possíveis valores *NA* na primeira coluna, ver Capítulo 2.2.2). Por outro lado, ao combinar duas opções [$-a$, $-r$], obtemos uma ordenação alfabética em ordem inversa, e [$-r$, $-a$] resulta numa ordenação inversa (por tamanho), que em caso de empate faz uma ordenação alfabética. Neste caso, consideramos que a opção $-r$ se refere ao inverso da ordenação padrão (ver testes executados no Capítulo 2.6).

2.3.2. Opções de limite

A opção de limite foi implementada com um valor padrão, definido pelo número de diretórios analisados, imprimindo assim todos os diretórios em estudo. Caso o utilizador inclua a opção $-l$ e um limite válido, o argumento será diretamente atribuído à variável *limit_ lines* para ser utilizado com o comando *head*, Figura 3.

```
100 if [[ -z "$limit_lines" ]]; then
101   limit_lines=$(echo "$folders" | wc -l)
102 fi
```

Figura 4. Valor padrão $-l$.

2.4. Validação e tratamento de argumentos

Nesta fase de desenvolvimento, implementámos um ciclo *while getopts* para recolher as opções e os respetivos argumentos. Após a recolha das opções utilizámos um deslocamento[2] para eliminar os argumentos que correspondiam às opções, ficando apenas com os diretórios especificados. Para explicar a validação, destacamos a condição de nulidade, Figura 5, usada na validação da data [$-d$], tamanho mínimo em bytes [$-s$] e número máximo de linhas [$-l$]. Essa condição verifica se o valor é nulo e, em caso afirmativo, chama a função *argError()*, Figura 6 . Importante notar que na validação da data, se não for nula, é convertida para o formato *AAAA-MM-DD HH:MM:SS* e que esta deve estar sempre em inglês, caso contrário, a função *argError()* é chamada novamente (ver padronização da data no Capítulo 2.1). Na validação do tamanho mínimo em bytes [$-s$] e número máximo de linhas [$-l$], verificamos também se o valor é positivo (no caso do tamanho mínimo pode ser zero). Quanto às opções de ordenação, [$-r$] e [$-a$], estabelecemos condições específicas para permitir a utilização conjunta (ver Capítulo 2.3.1). A opção [$-n$] está associada exclusivamente ao tipo de ficheiro desejado, não sendo necessária nenhuma validação.

```
[[ -z "$OPTARG" ]] && argError "-d/-s/-l"
```

Figura 5. Condição de nulidade.

2.5. Funções auxiliares

Vamos abordar mais detalhadamente cada função utilizada na implementação.

usage(): Esta função tem como objetivo informar ao utilizador como devia ter utilizado o comando *./spacecheck*.

argError(): Esta função tem como objetivo informar ao utilizador quando um argumento específico não está em conformidade com o definido no *usage()*.

invalidDirectory(): Esta função tem como objetivo informar ao utilizador que inseriu um diretório inválido como argumento.

É relevante destacar algumas funcionalidades comuns a estas funções, como o comando específico $1>\&2$ que redireciona a saída padrão para o mesmo destino que a saída de erro. Neste comando, *1*, representa

[2] Deslocamento: *shift $((OPTIND-1))*

a saída padrão *stdout*, > &, indica que a saída padrão será redirecionada, e *2* representa a saída de erro *stderr*. Isto é feito para encaminhar a mensagem de uso para a saída de erro, ressaltando que se trata de uma mensagem de erro e não de um resultado comum. Assim, como a instrução *exit*, que é utilizada para encerrar imediatamente o programa caso estas funções sejam chamadas.

```
usage() { echo "Usage: $0 [-d <date>] [-s <size>] [-l <limit>] [-r] [-a] [-n <regex>] [<
    directories>]" 1>&2; exit 1; }

argError() { echo "ERROR: \"$1\" arg is invalid." 1>&2; exit 1; }

invalidDirectory() { echo "ERROR: \"$1\" directory is invalid." 1>&2; exit 1; }
```

Figura 6. Funções auxiliares - *spacecheck.sh*

2.6. Testes

2.6.1. Testes de saída

Apresentamos a seguir uma pequena amostra dos testes realizados no script *spacecheck.sh*. Dada a multiplicidade de combinações de opções disponíveis não foram possíveis legendas, mas todos as saídas têm um cabeçalho com informação do comando. Como forma de exemplo, seguem os três comandos utilizados para a realização dos três primeiros testes (na horizontal):

Teste 1: Comando utilizado: `./spacecheck.sh sop`

Teste 2: Comando utilizado: `./spacecheck.sh -r sop`

Teste 3: Comando utilizado: `./spacecheck.sh -a sop`

```
SIZE NAME 20231111 sop
177 sop
81 sop/space Dir
74 sop/praticas2
51 sop/praticas2/aula2
12 sop/praticas2/aula1
11 sop/teoricas1
0 sop/teste
0 sop/teste2
NA sop/perm
```

```
SIZE NAME 20231111 -r sop
NA sop/perm
0 sop/teste2
0 sop/teste
11 sop/teoricas1
12 sop/praticas2/aula1
51 sop/praticas2/aula2
74 sop/praticas2
81 sop/space Dir
177 sop
```

```
SIZE NAME 20231111 -a sop
177 sop
NA sop/perm
74 sop/praticas2
12 sop/praticas2/aula1
51 sop/praticas2/aula2
81 sop/space Dir
11 sop/teoricas1
0 sop/teste
0 sop/teste2
```

```
SIZE NAME 20231111 -r -a sop
NA sop/perm
0 sop/teste
0 sop/teste2
11 sop/teoricas1
12 sop/praticas2/aula1
51 sop/praticas2/aula2
74 sop/praticas2
81 sop/space Dir
177 sop
```

```
SIZE NAME 20231111 -a -r sop
0 sop/teste2
0 sop/teste
11 sop/teoricas1
81 sop/space Dir
51 sop/praticas2/aula2
12 sop/praticas2/aula1
74 sop/praticas2
NA sop/perm
177 sop
```

```
SIZE NAME 20231112 -s 0 -l 4
    sop
177 sop
81 sop/space Dir
74 sop/praticas2
51 sop/praticas2/aula2
```

```
SIZE NAME 20231112 -s 10 -a -l
    4 sop
177 sop
NA sop/perm
74 sop/praticas2
12 sop/praticas2/aula1
```

```
SIZE NAME 20231112 -s 30 -n .*
    txt -l 4 sop
56 sop
56 sop/space Dir
0 sop/praticas2
0 sop/praticas2/aula1
```

```
SIZE NAME 20231111 -l 4 sop
177 sop
81 sop/space Dir
74 sop/praticas2
51 sop/praticas2/aula2
```

```
SIZE NAME 20231111 -l 4 -a -r
    sop
0 sop/teste2
0 sop/teste
11 sop/teoricas1
81 sop/space Dir
```

```
SIZE NAME 20231111 -l 4 -r -a
    sop
NA sop/perm
0 sop/teste
0 sop/teste2
11 sop/teoricas1
```

```
SIZE NAME 20231112 -l 4 -n
    space*
27264 .
20717 ./data
0 ./.git
0 ./.git/branches
```

```
SIZE NAME 20231112 -l 5 -n .*sh
24733 .
17838 ./test_a1
9534 ./test_a1/rrr
121 ./sop
74 ./sop/praticas2
```

```
SIZE NAME 20231112 -l 4 -n .*
    pdf .
219231 .
219231 ./data
0 ./.git
0 ./.git/branches
```

```
SIZE NAME 20231112 -d Nov 11
     sop
70 sop
25 sop/space Dir
23 sop/praticas2
12 sop/praticas2/aula2
11 sop/teoricas1
0 sop/praticas2/aula1
0 sop/teste
0 sop/teste2
NA sop/perm
```

```
SIZE NAME 20231112 -d Oct 7 sop
0 sop
0 sop/praticas2
0 sop/praticas2/aula1
0 sop/praticas2/aula2
0 sop/space Dir
0 sop/teoricas1
0 sop/teste
0 sop/teste2
NA sop/perm
```

```
SIZE NAME 20231112 -d   sop
177 sop
81 sop/space Dir
74 sop/praticas2
51 sop/praticas2/aula2
12 sop/praticas2/aula1
11 sop/teoricas1
0 sop/teste
0 sop/teste2
NA sop/perm
```

```
SIZE NAME 20231112 -l 4 sop
     data/
219531 data
177 sop
81 sop/space Dir
74 sop/praticas2
```

```
SIZE NAME 20231112 -l 4 sop/
     space Dir/
81 sop/space Dir
```

```
SIZE NAME 20231112 -l 4 sop/
     sop/praticas2
177 sop
81 sop/space Dir
74 sop/praticas2
51 sop/praticas2/aula2
```

```
SIZE NAME 20231112 -s 0 -l 2 sop
177 sop
81 sop/space Dir
```

Nos últimos três testes (horizontais) foi verificada a possibilidade de colocar mais de um diretório por comando, colocar um diretório com espaço, colocar diretórios com '/' na frente e colocar diretórios com relações hierárquicas, e em ambas as situações o programa comportou-se como esperado, verificando-se a padronização da saída e analisando todos os diretórios sem haver diretórios duplicados.

2.6.2. Testes de erro

```
Usage: ./spacecheck.sh [-d <date>] [-s <size>] [-l <limit>] [-r] [-a] [-n <regex>] [<directories>]
```

Figura 7. ./spacecheck.sh $-d$

```
ERROR: "dirNaoExiste" directory is invalid.
```

Figura 8. ./spacecheck.sh dirNaoExiste

```
ERROR: "-d" arg is invalid.
```

Figura 9. ./spacecheck.sh -d "NaoExiste"

```
ERROR: "-l" arg is invalid.
```

Figura 10. ./spacecheck.sh -l 0

```
ERROR: "-s" arg is invalid.
```

Figura 11. ./spacecheck.sh -s -1

Estes comandos não vão de encontro com o padrão das opções, logo são interrompidos com uma mensagem de erro, como era de se esperar. Compreendemos também, que existe uma infinidade de opções de testes diferentes, por isso, recomendamos ao leitor executar mais testes se achar conveniente.

3. A solução spacerate.sh

3.1. Ideias

Após concluir a implementação do primeiro módulo, *spacecheck.sh*, dedicamo-nos a uma análise minuciosa do propósito subjacente ao script *spacerate.sh*. Este módulo tem como principal objetivo a observação da evolução do espaço ocupado por um diretório, assim como pelos seus subdiretórios, em diferentes períodos temporais. Inicialmente, considerámos mais conveniente criar duas estruturas de dados, que armazenam os valores vindos dos ficheiros, para nos ajudar no cálculo das diferenças. Posteriormente, convertemos essa estrutura para um formato que não realiza uma soma acumulativa[3]. Isso permitiu-nos determinar precisamente a quantidade de bytes presentes em cada diretório (com profundidade 1). Por fim, desenvolvemos uma comparação entre as estruturas de dados com o objetivo de analisar as diferenças entre cada diretório. Destaca-se ainda a utilização de funcionalidades, com o mesmo propósito do script anterior (ver Capítulo 2.1).

3.2. Implementação

3.2.1. Guardar conteúdo

Neste subcapítulo, o objetivo é armazenar os conteúdos dos dois ficheiros numa estrutura de dados. Esta estrutura utiliza os caminhos para os diretórios como chaves e armazena os respetivos bytes. Inicialmente, ignoramos o cabeçalho utilizando a função *tail*, indo diretamente para a segunda linha. Em seguida, para extrair os bytes e o caminho correspondente, utilizámos o comando *awk* para obter os valores das colunas relevantes. Devido à presença de caminhos com espaços, implementamos um ciclo *for* para recolher corretamente o caminho completo. Por fim, para armazenar na estrutura desejada, iniciamos um ciclo *while* onde realizamos o processo de armazenamento. Os índices correspondem aos caminhos dos diretórios, e os valores representam o número de bytes ocupados em cada um. Este procedimento é repetido para ambos os ficheiros.

```
63 content_new_file="$(tail -n +2 "$new_file" |
64 awk '{ path=$2; for(i=3; i<=NF; i++) path=path" "$i; print $1, path }')"
65 ...
66
67 # Save content of new file
68 declare -A new_array
69 while read -r size path; do
70   new_array["$path"]=$size
71 done <<< "$content_new_file"
72
73 # Save content of old file
74 ...
```

Figura 12. Estruturas de dados spacerate.sh

3.2.2. Conversão em somas não acumulativas

Esta fase é crucial para o funcionamento eficaz deste módulo, pois prepara os dados de forma a facilitar a análise das discrepâncias entre os ficheiros. O objetivo é converter os valores de cada diretório numa soma que considera apenas os ficheiros diretos, excluindo os subdiretórios. Para realizar essa transformação, implementámos uma ordenação alfabética inversa, percorrendo primeiro os subdiretórios (diretórios com caminhos mais longos) e, em seguida os diretórios mais curtos. Essa prática garante que cada iteração do ciclo avance progressivamente para cima na hierarquia dos caminhos dos diretórios, dado que o conteúdo foi previamente ordenado. Após essa etapa, procedemos à atualização na estrutura de dados. Calculamos e armazenamos a diferença entre o valor do diretório pai e o valor do seu subdiretório. Essa técnica, ao subtrair os tamanhos dos subdiretórios dos diretórios pais, converte as somas para somas não acumulativas. A variável *father_path* é verificada iterativamente, permitindo que, ao identificar um diretório que não é pai do anterior, avance para o próximo diretório sem acabar o ciclo. Este processo é repetido para ambas as estruturas de dados. Para uma completa compreensão, recomendamos ao leitor a análise detalhada do código desenvolvido para esta finalidade, Figura 13.

[3]Soma não acumulativa: não soma os bytes dos ficheiros presentes em subdiretórios, apenas os ficheiros filho

```
# Convert new array to not cumulative sum
while read -r size path; do
  father_path="$path"
  while true; do
    [[ ${new_array["${father_path##/*}"]} ]] && break
    [[ "${father_path%/*}" = "${father_path##/*}" ]] && break
    father_path="${father_path%/*}"
    new_array["$father_path"]=$((${new_array["$father_path"]} - ${new_array["$path"]}))
  done
done <<< "$(sort -k2 -r <<< "$content_new_file")"

# Convert old array to not cumulative sum
...
```

Figura 13. Conversão dos arrays numa soma não cumulativa

3.2.3. Processamento de Variações

Na fase subsequente, a estrutura do código é dedicada à comparação dos diretórios presentes nas estruturas de dados *new_ array* e *old_ array*. Inicialmente, estabelecemos um ciclo *for* que percorre cada diretório na estrutura de dados *new_ array* e verifica a sua presença em *old_ array*. Se o diretório estiver presente, calcula-se a diferença entre os tamanhos. Por outro lado, caso o diretório não pertença ao arquivo antigo, é marcado como *NEW*. Posteriormente, é criado um novo ciclo *for* para percorrer todos os diretórios em *old_ array*. Dentro deste ciclo, estabelece-se uma condição que verifica se o valor existe exclusivamente em *old_ array*. Se essa condição for atendida, é apresentada a mensagem *"REMOVED"* ao lado do valor. Assim, para além de calcularmos as diferenças, os diretórios que se encontram apenas num dos ficheiros são também apresentados e assinalados de forma especial.

```
# Process the data
{

  # Check differences and NEW directories
  for path in "${!new_array[@]}"; do

    if [[ ${old_array["$path"]} ]]; then
      if [[ ${new_array["$path"]} == "NA" ]]; then
        echo "NA" $path
      else
        echo $((${new_array["$path"]} - ${old_array["$path"]})) $path
      fi
    else
      echo ${new_array["$path"]} $path "NEW"
    fi
  done

  # Check for REMOVED directories
  for path in "${!old_array[@]}"; do
    if [[ ! "${new_array["$path"]}" ]]; then
      if [[ ${old_array["$path"]} == "NA" ]]; then
        echo "NA" $path "REMOVED"
      else
        echo "-${old_array["$path"]}" $path "REMOVED"
      fi
    fi
  done

} | sort $sort_option
```

Figura 14. Processamento de variações

3.3. Opções de ordenação

Após o processamento de diretórios, apresentamos a informação conforme solicitado pelo utilizador. Ao contrário do *spacecheck.sh*, neste script, temos apenas opções de ordenação. A ordenação é realizada utilizando um *pipe* |, igual à solução apresentada anteriormente, ver Capítulo 2.3.1.

3.4. Validação e tratamento de argumentos

À semelhança do script spacecheck.sh, implementámos um *while getops*. As opções de ordenação foram tratadas da mesma forma que no script anterior. Em seguida, verificamos a presença exata de dois ficheiros passados como argumentos. Se não estiverem presentes, a função *usage()* encerra o programa. Posteriormente, verificamos a existência e legibilidade de cada ficheiro. Se tal não se verificar, a função *invalidFile()* é chamada, resultando no término do programa. Caso contrário, o programa continua a sua normal execução.

```
52 [[ "$#" -eq 2 ]] || usage
53
54 new_file="$1"
55 [[ -r "$new_file" ]] || invalidFile "$new_file"
56
57 old_file="$2"
58 [[ -r "$old_file" ]] || invalidFile "$old_file"
```

Figura 15. Validação

3.5. Funções auxiliares

Vamos abordar mais detalhadamente cada função utilizada na implementação.

usage(): Esta função tem como objetivo informar ao utilizador como devia ter utilizado o comando *./spacerate*.

invalidFile(): Esta função tem como objetivo informar ao utilizador que inseriu um ficheiro inválido como argumento.

```
usage() { echo "Usage: $0 [-r] [-a] [new_file] [old_file]" 1>&2; exit 1; }

invalidFile() { echo "ERROR: \"$1\" file is invalid." 1>&2; exit 1; }
```

Figura 16. Funções auxiliares - *spacerate.sh*

3.6. Testes

3.6.1. Testes de saída

Apresentamos a seguir uma pequena amostra dos testes realizados no script *spacerate.sh*. Os testes foram feitos com diretórios que contêm espaços (sop 1/ praticas 1/ aula 2) porém também foram feitos testes para diretórios sem espaços. Para mostrar que as ordenações funcionam para todos os casos, achamos mais indicado apresentar diretórios com espaços.

```
SIZE NAME 20220923 sop
10100 sop 1
5000 sop 1/teoricas
5000 sop 1/praticas 1
5000 sop 1/praticas 1/aula 1
100 sop 1/dados
NA sop 1/etc
```

Figura 17. Ficheiro antigo

```
SIZE NAME 20230923 sop
15635 sop 1
5090 sop 1/teoricas
10545 sop 1/praticas 1
5000 sop 1/praticas 1/aula 1
2668 sop 1/praticas 1/aula 3
2668 sop 1/praticas 1/aula 2
```

Figura 18. Ficheiro novo

De seguida, de forma a simplificar os testes, as legendas representarão as opções de ordenação utilizadas no comando *./spacerate.sh*.

```
SIZE NAME
2668 sop 1/praticas 1/aula 2 NEW
2668 sop 1/praticas 1/aula 3 NEW
209 sop 1/praticas 1
90 sop 1/teoricas
0 sop 1
0 sop 1/praticas 1/aula 1
NA sop 1/etc REMOVED
-100 sop 1/dados REMOVED
```

Figura 19. sem opções

```
SIZE NAME
-100 sop 1/dados REMOVED
NA sop 1/etc REMOVED
0 sop 1/praticas 1/aula 1
0 sop 1
90 sop 1/teoricas
209 sop 1/praticas 1
2668 sop 1/praticas 1/aula 3 NEW
2668 sop 1/praticas 1/aula 2 NEW
```

Figura 20. opção $[-r]$

```
SIZE NAME
0 sop 1
-100 sop 1/dados REMOVED
NA sop 1/etc REMOVED
209 sop 1/praticas 1
0 sop 1/praticas 1/aula 1
2668 sop 1/praticas 1/aula 2 NEW
2668 sop 1/praticas 1/aula 3 NEW
90 sop 1/teoricas
```

Figura 21. opção $[-a]$

```
SIZE NAME
-100 sop 1/dados REMOVED
NA sop 1/etc REMOVED
0 sop 1
0 sop 1/praticas 1/aula 1
90 sop 1/teoricas
209 sop 1/praticas 1
2668 sop 1/praticas 1/aula 2 NEW
2668 sop 1/praticas 1/aula 3 NEW
```

Figura 22. opções $[-r, -a]$

```
SIZE NAME
90 sop 1/teoricas
2668 sop 1/praticas 1/aula 3 NEW
2668 sop 1/praticas 1/aula 2 NEW
0 sop 1/praticas 1/aula 1
209 sop 1/praticas 1
NA sop 1/etc REMOVED
-100 sop 1/dados REMOVED
0 sop 1
```

Figura 23. opções $[-a, -r]$

3.6.2. Testes de erro

```
Usage: ./spacerate.sh [-r] [-a] [new_file] [old_file]
```

Figura 24. ./spacerate.sh data/spacecheck_20230923

```
ERROR: "dirNaoExiste" file is invalid.
```

Figura 25. ./spacerate.sh data/spacecheck_20230923 dirNaoExiste

```
ERROR: "dirNaoExiste" file is invalid.
```

Figura 26. ./spacerate.sh dirNaoExiste data/spacecheck_20220923

Estes comandos não vão de encontro com o padrão das opções, logo são interrompidos com uma mensagem de erro, como era de se esperar. Compreendemos também, que existe uma infinidade de opções de testes diferentes, por isso, recomendamos ao leitor executar mais testes se achar conveniente.

4. Conclusão

Em suma, o trabalho realizado está de acordo com os requisitos indicados no enunciado. A criação dos scripts *spacecheck.sh* e *spacerate.sh* segue os exemplos fornecidos no enunciado, mostrando um desenvolvimento consistente. Durante o processo, aplicámos os conhecimentos adquiridos nas aulas teóricas e práticas, complementados por alguma pesquisa na Internet. Isso não só consolidou os conceitos da linguagem Bash, mas também melhorou a nossa habilidade em usar comandos no terminal. Em geral, o maior desafio foi compreender todos os comandos necessários para implementar o projeto, já que algumas funcionalidades utilizadas eram novas para nós.

5. Bibliografia

[1] Shell Style Guide, https://google.github.io/styleguide/shellguide.html

[2] Stack Overflow, https://pt.stackoverflow.com/

[3] Bash scripting cheatsheet, https://devhints.io/bash