Tutorial 2

Manipulação de texto

BIZ0433 - INFERÊNCIA FILOGENÉTICA: FILOSOFIA, MÉTODO E APLICAÇÕES.

Conteúdo

Objetivo

O objetivo deste tutorial é apresentar algumas ferramentas disponíveis em LINUX para visualizar e manipular arquivos textos. Aprender o básico sobre essas ferramentas é fundamental, pois todos os programas ou aplicativos que iremos utilizar durante o curso requerem dados de entrada e imprimem informações em arquivos textos. Este tutorial não explora de forma exaustiva todos os conceitos apresentados relacionados às ferramentas de manipulação de texto disponíveis. No entanto, introduz termos e conceitos, bem como comandos e aplicativos, que certamente lhes darão bases para executar muitas tarefas apresentadas ao longo do curso. Considerem, no entanto, que estas ferramentas poderão ser melhor estudadas fora da sala de aula à medida em que encontre necessidades especiais para executar determinadas operações. Os arquivos associados a este tutorial estão disponíveis no GitHub. Você baixar todos os tutoriais com o seguinte comando:

```
svn checkout https://github.com/fplmarques/cladistica/trunk/tutorials/
```

2.1 Arquivos textos

Uma das grandes vantagens dos sistemas Unix/Linux é que a grande maioria dos arquivos que gere e sustentam o sistema, bem como aqueles manipulados por aplicativos, são arquivos texto. Durante o curso, você verá que todos os arquivos de entrada (*i.e.*, *input files*) e saída (*i.e.*, *output files*), são arquivos aos quais você terá acesso direto via editores de texto ou que vocês serão capazes de imprimí-los diretamente no terminal para visualização. Adicionalmente, você notará que a manipulação (edição) de textos é muito comum. Isso porque arquivos de saída de determinados programas, ou parte deles, podem ser utilizados como arquivos de entrada para outros programas em análises subsequentes.

Há uma lista de editores de texto disponíveis para todos os sistemas, mas o que queremos usar são aqueles que não inserem caracteres ocultos em seu arquivo (*e.g.*, *Word* da MicroSoft Windows). Dentre eles podemos citar *Notepad* para sistemas Windows, *TextWrangler* para MAC, e `vim`, `elvis`, `emacs`, `nedit`, `nano`, `kedit`, `gedit`, entre muitos outros para sistemas LINUX/UNIX. Para Linux, se o sistema no qual você está operando disponibiliza interfaces gráficas e seu ambiente de *Desktop* é GNOME - como é o caso da imagem disponibilizada no curso - o `gedit` é um editor de texto muito versátil. Dentre aqueles que não requerem interface gráfica e, desta forma, são apropriados para uso em terminais, nós iremos adotar `nano`. Para aqueles que estiverem muito interessados em usar editores dessa natureza, considerem estudar o `emacs`, é inacreditável o que você pode fazer com ele! Ao explorar esses editores, considerem que o conforto e poder desses aplicativos são inversamente proporcionais [1]!

2.1.1 Editores de textos

2.1.1.1 GEdit

Para abrir o `gedit` basta procurar o aplicativo no Painel Inicial do Ubuntu (veja Figura 2.1) e digitar a palavra `gedit` ou verifique se ele já se encontra na barra lateral de aplicativos do seu sistema. Abra o `gedit` e com ele abra o arquivo `10_tax_all_1000_trees.tre` que está no diretório `tutorial_02`. Veja as opções do programa na barra de menu e explore um pouco o que ele pode fazer. Note que ele não difere muito do que você está acostumado a fazer com o seu editor de texto (*e.g.*, *Word* da MicroSoft Windows). Este editor é intuitivo e eu acredito que não seja necessário explicar como ele funciona exceto pelo fato de que você pode abrí-lo diretamente via terminal com o comando:

```
$ gedit &
```

Observação: O símbolo "`&`" apenas faz com que o terminal rode o `gedit` no *backgroud*. Se o "`&`" é omitido, seu terminal fica vinculado à execução do `gedit` até que você feche o programa.

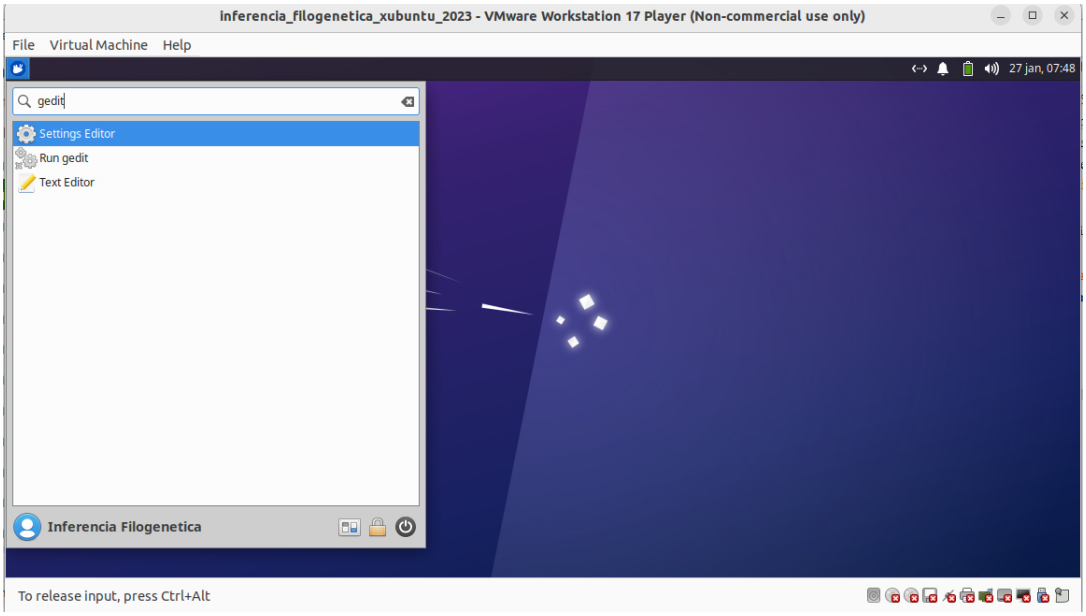

Figura 2.1: Abrindo `gedit` na imagem de Xubuntu via Painel inicial. Observe que a barra lateral de seu sistema lhe permite acesso direto ao gedit.

2.1.1.2 nano

O `nano` é um editor que deve ser executado a partir de um terminal, e se concentra em simplicidade. O `nano` é um clone do antigo editor de texto `pico`, o editor para o cliente de e-mail `pine`, que foi muito popular lá pelos anos 90, em UNIX e sistemas do tipo LINUX. O `nano` foi criado em 1999 com o nome de "`TIP`" (uma sigla, um acrônimo recursivo que significa "`TIP Isn't Pine`", por Chris Allagretta. Allagretta decidiu criar este clone do `pico` porque o programa não foi liberado sob a GPL. O nome foi mudado oficialmente em 10 de janeiro de 2000 para diminuir a confusão entre o novo editor e o comando "`tip`" (o comando "`tip`" é comum em Sun Solaris, uma distribuição de UNIX).
O `nano` usa combinações muito simples de teclas para trabalhar com arquivos. Um arquivo é aberto ou iniciado com o comando:

```
$ nano <arquivo>
```

Onde `<arquivo>` é o nome do arquivo que você deseja abrir.
Se você executa em um terminal o comando:

```
$ nano teste.txt
```

O `nano` irá abrir um arquivo vazio se `teste.txt` não existir no seu diretório de trabalho. No entanto, se você abrir o mesmo arquivo que abriu anteriormente utilizando `nano`, o arquivo `10_tax_all_1000_trees.tre` que está no diretório `tutorial_02`, você deverá observar o terminal ilustrado na Figura 2.2.

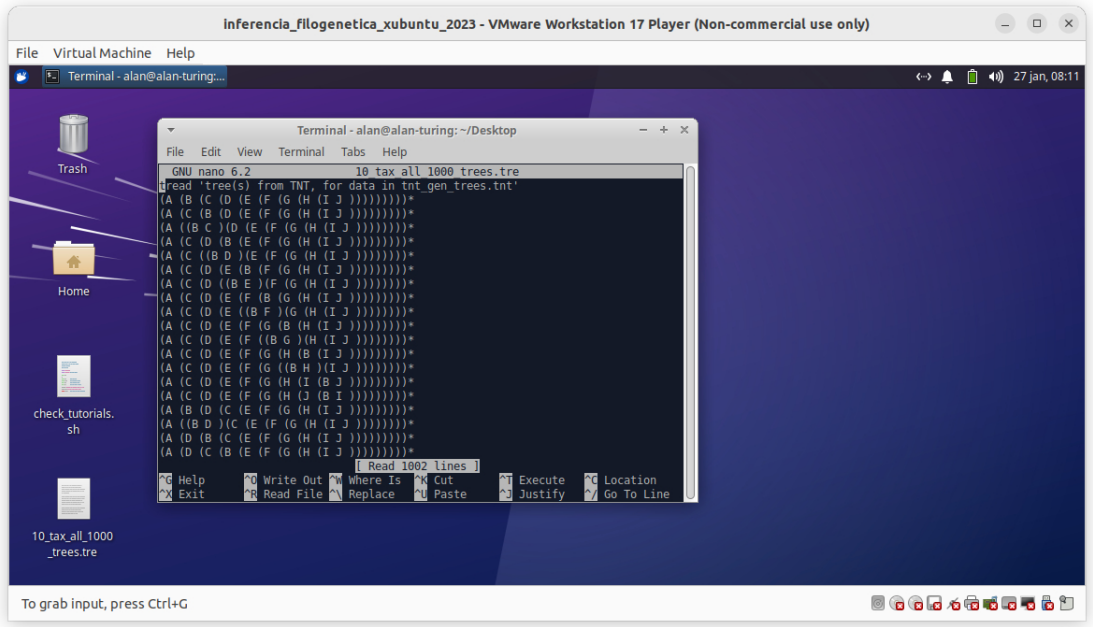

Figura 2.2: Abrindo `nano` na imagem do Xubuntu via terminal.

Quando o arquivo estiver aberto no `nano`, você verá uma pequena lista de sintaxe de comando na parte inferior da janela do terminal (marcado em vermelho na Figura 2.2). Todas as combinações de teclas para o `nano` começam com a tecla `Ctrl`, o `Ctrl` convencionalmente é representado pelo símbolo "^". Para executar um comando você deve manter a tecla `Ctrl` pressionada e clicar na segunda tecla para executar a ação. As combinações mais comuns para o `nano` são:

`Ctrl-x` - Sai do editor. Se você estiver no meio da edição de um arquivo, o processo de saída irá perguntar se você quer salvar seu trabalho.
`Ctrl-R` - Ler um arquivo em seu arquivo de trabalho atual. Isso permite que você adicione o texto de outro arquivo enquanto trabalha dentro de um novo arquivo.
`Ctrl-c` - Mostra a posição atual do cursor.
`Ctrl-k` - "recorta" o texto.
`Ctrl-U` - "cola" o texto.
`Ctrl S` - Salva o arquivo e continua trabalhando.
`Ctrl-T` - verifica a ortografia do seu texto.
`Ctrl-w` - faz uma busca no texto.
`Ctrl-a` - leva o cursor para o início da linha.
`Ctrl-e` - leva o cursor para o fim da linha.
`Ctrl-g` - mostra a ajuda do `nano`.

Existem muitos outros comandos disponíveis no `nano`. Para ver a lista de comandos, use o comando **`Ctrl-g`**.

Exercício 2.1

Você deverá criar um arquivo chamado `matrix.txt` utilizando `Nano` com o seguinte conteúdo:

```
p04 0000000
p02 1110000
p05 1110000
p06 1101000
p03 1101000
p01 1000100
p08 1000110
p10 1000111
p09 1000111
```

2.1.1.3 Outras formas de visualização de texto

Há outras formas de visualizar e/ou obter informações rápidas de arquivos texto. Abaixo iremos explorar algumas delas:

2.1.1.3.1 Comando `cat` : O comando `cat` foi introduzido a vocês no Tutorial 1. Na ocasião, aprenderam que este comando permite a impressão do conteúdo de arquivos texto no terminal, bem como seu uso para concatenar arquivos textos utilizando os caracteres de direcionamento (*i.e.*, “>” e “»”; veja o Tutorial 1). No entanto, o comando `cat` pode também ser utilizado para criar arquivos de forma bem mais rudimentar do que o `nano`. Um exemplo é dado no próximo exercício.

Exercício 2.2

Abra um terminal e digite:

```
$ cat > tnt_head.txt
```

Você deverá obter uma linha vazia no terminal. Digite:
“`xread`” (sem as aspas)
Pressione ENTER.
Agora digite:
“`7 10`” (novamente sem as aspas)
Pressione ENTER.
Finalmente pressione `Ctrl-D`

Examine o conteúdo de `tnt_head.txt`:

```
$ cat tnt_head.txt
```

Você deverá ter obtido um arquivo com o seguinte conteúdo:

```
xread
7 10
```

Muito bem! Você irá usar esse arquivo em breve, portanto deixe-o no diretório `tutorial_02`.

2.1.1.3.2 Comando `sort`: O `sort` é um programa interno de Linux que organiza as linhas de um arquivo texto ou da entrada padrão. A organização é feita por linhas e as linhas são divididas em campos que é a ordem em que as palavras aparecem na linha separadas por um delimitador (normalmente um espaço). Organizar aquivos pode facilitar muito a leitura, principalmente de tabelas e, em alguns casos ou seu arquivo é muito grande para ser organizado por programas como Excel ou você não terá acesso à interfaces gráficas. Portanto, o `sort` pode ser uma ferramenta muito poderosa para organizar informação.

Sintaxe:

```
$ sort -opção1 -opção2 -opção3 argumento
```

O `argumento` é, via de regra, o arquivo que contém as informações que você quer organizar. Há diversas opções para esse programa (veja `$ man sort`, para maiores detalhes); no entanto, iremos apresentar algumas poucas que podem ser úteis utilizando alguns exemplos.

Veja uma utilidade simples deste programa. O arquivo `sort_example_1.tre` contém uma série de topologias compiladas pelo POY [2, 3] – um programa desenvolvido para análises de homologia dinâmica, que veremos no Tutorial 9. Para saber o número de topologias contidas neste arquivo basta usar o comando `wc` com a opção `-l` (veja Tutorial 1, seção **??**). O programa `sort` tem o termo "`-u`" como uma de suas opções, nela são consideradas apenas as linhas que são únicas, ou seja, diferentes. Desta forma, a linha de comando:

```
$ sort -u sort_example_1.tre
```

retorna apenas as topologias únicas contidas neste arquivo.

Exercício 2.3

Quantas topologias únicas existem neste arquivo?.

Veremos outras utilidades do `sort`. Verifique o conteúdo do arquivo `random_taxon_table.tsv` no diretório `tutorial_02` utilizando o comando `cat`. A extenção "`.tsv`" (***T**ab **S**eparated **V**alues*) é utilizada para arquivos em que os valores ou conjunto de textos estão separados por `TAB`. Você consideraria que ele está organizado? Se você consultar o manual do `sort` utilizando o comando `man sort` constatará que há uma opção que permite verificar se seu arquivo possui ou não alguma ordem. A opção é a seguinte:

```
...
-c, -check, -check=diagnose-first
check for sorted input; do not sort
...
```

Se você executar:

```
$ sort -c random_taxon_table.tsv
```

Você deverá obter:

```
sort: random_taxon_table.tsv:3: desordenado: ...
```

A forma mais básica de organizar esse arquivo é executando `sort` sem nenhuma opção. Tente você mesmo e verifique como `sort` organizou seu arquivo. Note que o `sort` organizou sequencialmente as colunas de sua tabela a partir do ordenamento dos gêneros. Talvez isso fique mais óbvio se ordenarmos esses dados por estados. Para isso vejamos a opção **`-k`** de `sort`. Veja o que diz a documentação desse programa para esta opção (`man sort`):

```
...
-k, -key=POS1[,POS2]
start a key at POS1 (origin 1), end it at POS2 (default end of
line).  See POS syntax below
...
POS is F[.C][OPTS], where F is the field number and C the
character position in the field; both are origin 1.  If neither
-t nor -b is in effect, characters in a field are counted from
the beginning of the preceding whitespace.  OPTS is one or more
single-letter ordering options, which override global ordering
options for that key.  If no key is given, use the entire line
as the key.
```

```
...
```

As posições às quais a documentação se refere são os conjuntos de texto separados pelo delimitador, nesse caso TAB (tabulador). Por exemplo, note que a terceira coluna do arquivo `random_taxon_table.tsv` inclui os estados de origem do material listado. Vamos organizar a lista de exemplares deste documento por estado.

Se você executar:

```
$ sort -k 3 random_taxon_table.tsv
```

Você deverá obter:

```
Genus_3 sp_1 AC imaturo MZUSP 0.46 0.6 0.93 0.66 0.33
Genus_3 sp_1 AC imaturo MZUSP 0.87 0.82 0 0.1 0.8
Genus_4 sp_1 AC imaturo MZUSP 0.98 0.98 0.36 0.43 0.98
Genus_4 sp_0 AC indet.  BMNH 0.69 0.3 0.25 0.12 0.63
Genus_2 sp_1 AC indet.  MZUSP 0.16 0.54 0.89 0.08 0.04
Genus_1 sp_0 AC indet.  MZUSP 0.41 0.99 0.66 0.4 0.33
Genus_2 sp_2 AC indet.  MZUSP 0.52 0.96 0.38 0.76 0.05
Genus_2 sp_5 AC indet.  MZUSP 0.75 0.02 0.1 0.07 0.39
...
```

Observe o que acontece quando você delimitar o ordenamento à terceira coluna:

Se você executar:

```
$ sort -k 3,3 random_taxon_table.tsv
```

Você deverá obter:

```
Genus_1 sp_0 AC indet.  MZUSP 0.41 0.99 0.66 0.4 0.33
Genus_2 sp_1 AC indet.  MZUSP 0.16 0.54 0.89 0.08 0.04
Genus_2 sp_1 AC maturo BMNH 0.3 0.53 0.38 0.43 0.34
Genus_2 sp_2 AC indet.  MZUSP 0.52 0.96 0.38 0.76 0.05
Genus_2 sp_3 AC maturo AMNH 0.68 0.93 0.2 0.7 0.93
Genus_2 sp_4 AC maturo BMNH 0.52 0.06 1 0.31 0.84
Genus_2 sp_5 AC indet.  MZUSP 0.75 0.02 0.1 0.07 0.39
Genus_3 sp_1 AC imaturo MZUSP 0.46 0.6 0.93 0.66 0.33
...
```

Exercício 2.4

i. O que acontece com o ordenamento de duas colunas de seu arquivo quando você executa a linha de comando abaixo?

```
$ sort -k 5,5 -k 3,3 random_taxon_table.tsv
```

ii. Você deverá criar um arquivo chamado **`matrix.tnt`** em que as duas primeira linhas contenham o texto que você inseriu no arquivo `tnt_head.txt`, criado anteriormente, seguido do texto **ordenado** por táxon da matrix contida no arquivo `matrix.txt` e que a última linha contenha um ponto e vírgula (*i.e.*, "`;`"). Esta última etapa pode ser realizada com o comando `echo ';'` direcionado de forma apendiciada para o arquivo final. Em nenhum momento você deverá usar qualquer editor de texto! O comando `cat`, `echo` e as formas de redicionamento são suficienetes para criar esse arquivo.

Se você conseguiu **parabéns!**, você acabou de criar um arquivo de entrada para o programa de análise filogenética chamado **TNT** [4].

Finalmente, para terminar a discussão sobre o `sort` vejamos o conteúdo do arquivo `random_taxon_table.csv`. Ele contém as mesmas informações que o arquivo `random_taxon_table.tsv`. A extensão "`.csv`" refere-se à documentos cujo delimitador é uma vírgula (***C**omma **S**eparated **V**alues*), muito comum em bases de dados e uma forma de exportar informações de planilhas eletrônicas (*i.e.*, Excel e/ou OppenOffice Calculator) para arquivos que possam ser processados como textos.

Se você tentar executar as mesmas linhas de comando anteriores para o ordenamento do conteúdo deste arquivo, principalmente no que concerne ao ordenamento por valores internos, você não irá conseguir. Isso porque o `sort` usa, por *default*, espaços como delimitadores. No entanto, em `sort` há como você especificar o delimitador com a opção **`-t`**, vejamos o que diz a documentação:

```
...
-t, -field-separator=SEP
use SEP instead of non-blank to blank transition
...
```

Confuso não!? Enfim, o que esta opção define é qual delimitador você quer especificar. Por exemplo, se seu delimitador é "`,`" sua opção dever ser `-t ','`.

Se você executar:

```
$ sort -t ',' -k 3,3 random_taxon_table.csv
```

Você obterá o mesmo ordenamento feito anteriormente pelos estados de procedência.

Exercício 2.5

i. Considere o conteúdo do arquivo `indexed_trees.txt` no diretório `tutorial_02` que possui a sequinte estrutura:

```
(A,(B,(C,(E,(D,F)))))[4.1.4.7];
...
(A,((D,E),(C,(B,F))))[4.3.3.5];
(A,((D,E),(B,(C,F))))[4.3.3.4];
(A,(D,((B,C),(E,F))))[4.3.2.5];
(A,((E,F),(D,(B,C))))[4.3.1.2];
(A,((E,F),(C,(B,D))))[4.2.1.2];
...
```

Esse arquivo possui um conjunto de topologias com um indexador de 4 dígitos entre colchetes (*i.e.*, `[4.1.4.7]`).
Neste exercício você deverá ordenar as topologias pelos seus indexadores.

2.1.1.3.3 ***Começo e final de arquivos:*** Geralmente, arquivos que registram uma análise filogenética são longos e a impressão de seu conteúdo em um terminal excede o limite de linhas que ele retém como registro. Veja por exemplo o arquivo `garli_screen.log`. Este arquivo contém o *log* de uma análise em Garli, um programa de análises filogenéticas que utiliza máxima verossimilhança como critério de otimalidade [5].

Se você executar:

```
$ cat garli_screen.log | wc -l
```

Você observará que este arquivo possui uma quantidade enorme de texto e que ao ser impressa no terminal você perdeu o acesso ao início do documento (veja isso executando o comando).
Há uma série de informações importantes nesse arquivo que devem ser apresentadas quando você descreve sua análise e documenta seus resultados. Por exemplo, o início deste arquivo registra informações quantitativas e qualitativos de seus dados, modelos utilizados, entre outras coisas; ao passo que no final do mesmo arquivo há informações importantes sobre o resultado final de sua análise. Embora os detalhes destas informações sejam irrelevantes no momento, vejamos como podemos acessar o início e o final deste arquivo utilizando os comandos `head` e `tail`.

Se você executar:

```
$ head -n 70 garli_screen.log
```

Você obterá as 70 linhas iniciais deste arquivo.

Por outro lado, se você executar:

```
$ tail -n 35 garli_screen.log
```

Você obterá as 35 linhas finais deste arquivo.

Exercício 2.6

Execute os comandos abaixo e para cada um deles descreva quais tarefas ele efetuam:

```
$ head -n 1 random_taxon_table.tsv >
random_taxon_table_sorted_2.txt
```

Descrição:

_____

_____

```
$ tail -n +2 random_taxon_table.tsv | sort -k 5,5 -k 3,3 -k 4,4 »
random_taxon_table_sorted_2.txt
```

Descrição:

_____

_____

```
$ head -n -1 matrix.tnt | tail -n +3
```

Descrição:

_____

_____

Finalmente, vamos juntar alguns conceitos. Neste componente no exercício, você deverá usar o comando `head/tail` e `sort` em conjunção com os conceitos de redirecionamento do Tutorial 1 – especificamente a seções **??** e **??**, para executá-lo. Você deverá organizar o conteúdo do arquivo `random_taxon_table.tsv` sequencialmente por museu depositado, estado de origem e maturidade. Este arquivo deverá ser salvo com o nome `random_taxon_table_sorted.txt` e o cabeçário (*i.e.*, `#Genero Especie Estado matur. museu Var_1 ..`) deverá estar na primeira linha.

2.1.1.3.4 Comando `less`: Algumas vezes é desejável verificar o conteúdo de um documento total ou parcialmente, principalmente quando você está examinando a estrutura do documento e o arquivo ocupa mais do que seu terminal é capaz de reter. O comando `less` permite que isso seja feito com um arquivo texto e/ou a saída de algum comando.

A síntax do comando é simples:

```
$ less <arquivo>
```

ou ainda:

```
$ comando argumento | less
```

Após evocar o comando `less`, a tecla ENTER faz com que a rolagem do documento ocorra linha por linha ao passo que a tecla PAGE DOWN apresenta a página seguinte. Para sair do `less` basta pressionar a tecla `q`. Há dois comandos que você deve saber para o comando `less`. Se você digitar "`/`" seguido por qualquer padrão de busca (seja uma palavra ou uma expressão regular [Seção 2.2, abaixo]), `less` executa a busca a partir da segunda linha exibida na tela. Se você digitar "`?`" seguido por qualquer padrão, `less` executará a busca no sentido reverso.

Exercício 2.7

Examine o arquivo `garli_screen.log` usando o comando `less` e busque pela palavra `best`. Em qual réplica desta análise encontra-se a melhor solução?

2.1.1.3.5 Comando `grep`: O comando `grep` (de *Global regular expression parser*) pode ser visto como uma forma simplificada de consulta a um arquivo texto, em que cada linha representa um registro. Ele pode ser usado para retirar um conjunto de *strings* (cadeias de caracteres) do resultado de um comando dado ou de um arquivo texto. O `grep` é um dos comandos mais poderosos dentro de seu sistema, basta consultar sua documentação para ver a quantidade de opções disponíveis. Abaixo vamos explorar algumas propriedades desse comando com exemplos relacionados com nossa prática.
Sintaxe:

```
$ grep -opção1 -opção2 ...  argumento
```

Na sua forma mais simples, `grep` pode ser utilizado para imprimir as linhas que contém uma determinada palavra, ou caracteres literais. Considere um arquivo de *log* de uma análise realizada em POY, um programa para analisar caracteres filogenéticos utilizando homologia dinâmica [2]. O arquivo `onychophora_poy_std.err` possui 102418 linhas e registrou 10 horas de análise! Estas análises envolveram iterações de vários algorítimos. O que iremos fazer com `grep` é tentar extrair as informações relacionadas à estas iterações. Os dados que nos interessam estão em linhas com o seguinte padrão:

```
Information :  The search evaluated ...
```

Vamos tentar alguns comandos e ver o resultado que eles produzem:

i. `$ grep Information onychophora_poy_std.err`
Execute o comando acima. Ele imprime a informação que você gostaria de obter?

Certamente não, apenas uma das linhas impressas no final está de acordo com o padrão que estamos buscando. Tente o comando acima substituindo "`Information`" por "`search`". Melhorou? Sim, mas note que o resultado inclui linhas desnecessárias que fogem do padrão desejado!

ii. `$ grep Information onychophora_poy_std.err | grep search`
Vejamos o que acontece com os comandos concatenados acima. Você tem dificuldade em entender o que eles estão fazendo? Simples, o resultado de "`grep Information onychophora_poy_std.err`" serviu de entrada para "`grep search`", ou seja, você filtrou duas vezes.

Veja o que acontece com esses comandos:

```
$ grep '^Information .* search' onychophora_poy_std.err

$ grep '^Information .*\d*\stimes' onychophora_poy_std.err
```

Ambos os comandos imprimem o mesmo resultado do item iiiii, mas se valem de expressões regulares (ou operadores lógicos), tais como "`.*`" e "`.*\d*\s`", que iremos explorar a seguir.

2.2 Expressões regulares (REGEX)

Uma expressão regular, em ciências da computação, define um padrão a ser usado para procurar ou substituir palavras ou grupos de palavras. É um meio preciso de se fazer buscas de determinadas porções de texto. Como todas as buscas, o problema é encontrar [1]. Quanto melhor você definir o que está à procura, melhor será o resultado de sua busca. O uso de Expressões Regulares (ER, Ereg ou RegEx para *Regular Expression*) é um método rápido e simples de manipulação e combinação avançada de *strings*. Você verá que essas expressões podem ser úteis no seu dia a dia, principalmente se suas atividades de pesquisa e/ou estudo incluem manipulação de arquivos texto e busca rápida de informação. Se você utilizar REGEX com habilidade, elas podem simplificar muitas tarefas de programação e processamento de texto, além de permitir outras que não seriam possíveis sem elas [6].

2.2.1 TERMINOLOGIA E ESTRUTURA:

Antes de entendermos como as REGEX são expressas e usadas, é necessário introduzir alguns termos. **Caracteres literais**, *literals*, é um conjunto de caracteres que usamos em uma busca, por exemplo, para encontrar *inu* em *Linux*, *inu* é uma sequência de caracteres (em computação definida como *string*) literais. **Meta-caráter**, *meta character*, é um ou mais caracteres especiais que possuem um significado único e não são usados como **caracteres literais** na busca, ou define um **meta-caráter**. Por exemplo, no item iiiii de 2.1.1.3.5 os termos "`.`" e "`*`" são exemplos

de **meta-caracteres**. **Sequência de escape**, *scape sequence*, é uma forma de indicar que nós queremos usar **meta-caracteres** como **caracteres literais**. Em REGEX uma sequência de escape envolve a inserção de "\" anterior a um **meta-caráter**. Por exemplo, para encontrar um "`.`" ou um "`*`" em um texto é necessário o uso das sequências de escape "`\.`" e "`\*`", respectivamente. A **sequência de caracteres alvo**, *target string*, define a sequência de caracteres que estamos buscando e o **padrão de busca**, *search pattern* ou *construct*, descreve a expressão que estamos usando para buscar a **sequência de caracteres alvo**.

Há três componentes importantes em REGEX: âncoras, conjunto de caracteres e modificadores. **Âncoras** são usadas para especificar a posição do padrão em relação a uma linha de texto e, por definição, são meta-caracteres. **Conjunto de caracteres** correspondem a um ou mais caracteres em uma única posição, e aqui podem ser expressos caracteres literais e/ou meta-caracteres. **Modificadores** especificam quantas vezes o conjunto de caracteres anterior é repetido e, também são considerados meta-caracteres. Um exemplo simples que demonstra todos os três componentes é a expressão regular "`^#*`". O acento circunflexo é uma âncora que indica o início da linha. O caráter "`#`" é um conjunto de caracteres simples que corresponde ao de um único caráter "`#`". O asterisco é um modificador de "`#`" – aceitando zero ou mais ocorrência deste caráter. Esta é uma expressão regular inútil, mas exemplifica os componentes frequentemente encontrados em REGEX.

Vamos entender as duas instâncias em que foram utilizadas REGEX no item iiiii acima e entender os componentes das REGEX utilizadas. Ao executarmos o comando `grep '^Information .* search' onychophora_poy_std.err` a REGEX está contida entre aspas, "`^`", "`.`" e "`*`" são **meta-caracteres** - dentre os quais "`^`" é uma âncora e "`*`" é um **modificador**, ao passo que "`Information`" e "`search`" são **caracteres literais**. No comando `grep '^Information .*\d*\stimes' onychophora_poy_std.err`, "`times`" é um novo **caráter literal** e "`\d`" "`\s`" são novos **meta-caracteres** expressos como **sequência de escape**.

2.2.2 ÂNCORAS:

Não é fácil de fazer uma busca de padrão de caracteres simples que corresponda ao de um único caráter "`*`". O asterisco é um meta-caráter modificador. Em uma REGEX, "`*`" especifica que o caráter, literal ou não, se repete de zero a N vezes pela linha. O fim da linha de caracteres não está incluído no bloco de texto que é pesquisado. O fim da linha é um separador. Expressões regulares examinam o texto entre os separadores. Se você quiser procurar por um padrão que está em uma extremidade ou em outra, você usa âncoras. O caráter "`^`" é a âncora inicial, e o caráter "`$`" é a âncora final. A expressão regular "`^A`" irá corresponder a todas as linhas que começam com um A maiúsculo. A expressão "`A $`" irá corresponder a todas as linhas que terminam com A maiúsculo. Se os caracteres de ancoragem não são usados no final do próprio padrão, então eles

não funcionaram como âncoras. Ou seja, o "^" é apenas uma âncora se for o primeiro caráter em uma expressão regular. O "$" é apenas uma âncora se for o último caráter. A expressão "$1" não tem uma âncora. O mesmo ocorre para a regex "1^". Se você precisa encontrar um "^" no início da linha, ou um "$" no final de uma linha, você deve escapar os caracteres especiais com uma barra invertida (e.x., "^" ou "$", veja Tabela 2.1). Finalmente, o uso de "^" e "$" como indicadores do início ou final de uma linha é uma convenção em várias ferramentas de Linux/Unix. O editor **vi**, por exemplo, usa esses dois caracteres como comandos para ir para o início ou fim de uma linha.

Para ilustrar o conceito de âncoras, vamos executar uma busca no arquivo `random_trees.tre` para identificar em quantas topologias os táxons `A`, `B`, `C` e `D` estão na raiz.

Exercício 2.8

Execute o seguinte comando:

```
$ egrep A random_trees.tre
```

Observe que este comando imprime as linhas (topologia) onde o táxon `A` é encontrado, independente de sua posição. O comando `egrep` é a versão REGEX do `grep`. Para obter as topologias nas quais `A` está na raiz devemos inserir a seguinte REGEX:

```
$ egrep '^\(A' random_trees.tre
```

Nesta REGEX, expressa entre aspas, o **padrão de busca** `'^\(A'` descreve a expressão que estamos usando para buscar uma **sequência de caracteres alvo** em que a topologia (linha) esteja escrita de forma que os caracteres `"(A"` estejam no início da linha.
Finalmente, execute o comando e responda:

```
$ egrep '^\(A|^\(B' random_trees.tre
```

Qual e a fução do pipe (*i.e.*, "`|`") nesta REGEX?

_____

_____

_____

Exercício 2.9

A opção "`-c`" do `egrep`, faz com que esse comando retorne o número de linhas que contém o padrão de busca que você usou. Isto posto, você deverá anotar abaixo o número de topologias que os táxons `A`, `B`, `C` e `D` estão na raiz.

`A:` ___ `B:` ___ `C:` ___ `D:` ___

Exemplos: A Tabela 2.1 ilustra alguns exemplos do padrão e busca com âncoras.

Tabela 2.1: Exemplos de âncoras.

Padrão	Busca
`^A`	"A" no começo da linha
`A$`	"A" no final da linha
`A^`	"A^" qualquer lugar da linha
`$A`	"$A" qualquer lugar da linha
`^^`	"^" no começo da linha
`$$`	"$" no final da linha

2.2.3 Modificadores e caracteres de escape:

As Tabelas 2.2, 2.3 e 2.4 contém padrões de RegEx relacionados com classes de caracteres pré-definidos, caracteres de escape e modificadores. Esta é uma pequena amostra de meta-caracteres disponíveis para expressar padrões de busca em RegEx. O uso de RegEx é pessoal, pois basta examinar alguns destes meta-caracteres para perceber que há diversas maneiras de expressar um mesmo padrão de busca. Para o propósito deste curso, é importante que vocês saibam o que é uma RegEx e como elas funcionam e podem ser usadas. Para atingir este objetivo, alguns poucos exemplos serão apresentados a seguir. Você verá que alguns desses meta-caracteres são usados frequentemente e que o domínio de alguns poucos elementos de RegEx pode ser uma ferramenta muito vantajosa NO SEU DIA A DIA. Como só se aprende na prática, vamos fazer algumas tentativas de utilizar RegEx para extrair algumas informações de alguns arquivos.

Exercício 2.10

i. Considere a seguinte topologia gerada por TNT:

```
(D (H ((E (A F ))(B (J (G (C I )))))))
```

Esta topologia está entre as 2924 árvores contidas no arquivo `random_trees.tre`. Suponha que você queira saber quais são e quantas são as topologias que contém um grupo monofilético formado por `E, A` e `F`. Qual seria seu padrão de busca e quantas topologias contém este clado?
Dica: Consulte a Tabela 2.3 para ver como expressar parênteses literais, a Tabela 2.2 para ver como você busca um padrão ou outro e examine todos os clados possíveis antes de conjugá-los em uma única RegEx. Adicionalmente considere que esses três táxons podem estar relacionados de outra forma (*e.g.*. `(F (A E ))`) o que requer que você considere uma coisa **ou** outra – veja Tabela 2.2.

ii. Considere o seguinte o comando: `egrep '.*T\w.*' random_taxon_table.tsv`
O padrão de busca dessa REGEX está recuperando qual sequência de caracteres alvo?

iii. Em uma única REGEX extraia do arquivo `random_taxon_table.tsv` todos os exemplares coletados nos estados do AM e RO.
Escreva sua REGEX abaixo:

```
$ egrep '                                    '
```

Tabela 2.2: Classes de caracteres pré-definidas utilizadas em REGEX.

Padrão	Busca
`.`	Qualquer caráter
`[r|s]`	Ou "r" ou "s", pode ser escrito "(r|s)"
`[a-z]`	Qualquer letra minúscula
`[A-Z]`	Qualquer letra maiúscula
`[a-zA-Z]`	Qualquer letra maiúscula ou minúscula
`[0-9]`	Qualquer número
`[0-9.-]`	Qualquer número, ponto ou sinal de subtração
`[^-0-9]`	Qualquer caráter exceto um número ou "-"
`[[:alpha:]]`	Qualquer letra (alfabética)
`[[:digit:]]`	Qualquer número (dígito)
`[[:alnum:]]`	Qualquer letra ou número (alfanumérico)
`[[:space:]]`	Qualquer caráter de espaço
`[[:upper:]]`	Qualquer letra maiúscula
`[[:lower:]]`	Qualquer letra minúscula
`[[:punct:]]`	Qualquer caráter de pontuação

Tabela 2.3: Caracteres de Escape usados em REGEX.

Padrão	Busca
`\t`	Caráter de tabulação (TAB)
`\n`	Linha nova
`\)`	Um parêntese literal
`\\`	Uma barra invertida literal

Tabela 2.3 – Continuação.

Padrão	Busca
`\-`	Um hífen literal
`\w`	Qualquer caráter alfanumérico incluindo "_"
`\W`	Qualquer caráter que não seja alfanumérico
`\s`	Qualquer espaço em branco
`\S`	Qualquer espaço que não seja em branco
`\d`	Qualquer caracter que seja um dígito
`\D`	Qualquer caracter que não seja um dígito

Tabela 2.4: Multiplicadores de caracteres em REGEX.

Padrão	Busca
`?`	Uma ocorrência ou nenhuma (Equivale a {0,1})
`*`	Nenhuma ocorrência ou qualquer número de ocorrência (Equivale a {0,})
`+`	Uma ou mais ocorrências (Equivale a {1,})
`r|s`	equivalente a [rs], "r" ou "s"
`^[[:alpha:]]{3}$`	Qualquer palavra de três letras
`^[[:digit:]]{3}$`	Qualquer número com três dígitos
`^a{4}$`	A expressão recupera o padrão "aaaa"
`^a{2,4}$`	A expressão recupera os padrões "aa", "aaa" e "aaaa"
`^a{2,}$`	A expressão recupera os padrões "aa", "aaa", "aaaa", ...

2.3 Sed:

2.3.1 INTRODUÇÃO E SINTAXE BÁSICA:

Sed, de ***s**tream **ed**itor*, é uma das ferramentas mais antigas e usadas em Linux. Ele é um editor não interativo e orientado por linhas. Isso significa que os comandos de edição são inseridos por comandos de linhas ou estão contidos em um arquivo. **Sed** manipula os arquivos sem modificar o original e caso você queira manter a cópia modificada basta direcionar o resultado do comando para um outro arquivo (veja Tutorial 1, seção **??**). **Sed** é fundamentalmente uma ferramenta para substituir textos e sua vantagem está no fato de que ele lê linha por linha, diferentemente editores de texto convencionais que carrega todo o arquivo inicialmente. Por esta razão, **Sed** é capaz lidar com arquivos muito grandes, o que pode ser impraticável utilizando os editores de texto que você tem mais familiaridade. Considere por exemplo o arquivo `sed_Exercício_1.txt` no diretório `tutorial_02`. Este arquivo contém 5000 dados para 20 gêneros que variam de 1 a 20 espécies. No total, este documento possui 180 linhas, 900.360 palavras e 5.403.645 de caracteres! Para cada linha você encontra dados para um gênero, uma espécie e 5000 medidas obedecendo

o seguinte formato: `Genus_01 species_0 0.869 0.610 0.904 ....` Suponha que você desejasse, substituir todos os pontos (“`.`”) por vírgulas (“`,`”). Como você faria isso? A edição convencional em um computador com 8 processadores e 24 G de RAM tomou mais de 9 minutos entre abrir, substituir e salvar a cópia modificada! Outro benefício do **Sed** que você pode criar arquivos contendo comandos de substituição para tarefas repetitivas, algumas das quais você se deparará durante este curso.

Sintaxe:

```
$ sed -opção 'COMANDO_DE_EDIÇÃO' arquivo_de_entrada
```

Por exemplo, execute o comando:

```
$ sed -n 'p' sed_example_1.txt
```

Você obterá:

```
>seq_1
acgcaggaatggcaga
>seq_2
acgcagcaagggacgttt
>seq_3
acgcagctaccgacgttt
>seq_4
acgttctacaccgacgttt
>seq_5
attcaataccgacgttt
```

A opção **`-n`** em conjunção com o modificador **`p`** faz com que **Sed** imprima o texto do arquivo `sed_example_1.txt` no terminal. As substituições em **Sed** são relativamente simples.

Por exemplo, execute o comando:

```
$ sed 's/seq/taxon/' sed_example_1.txt
```

Você obterá:

```
>taxon_1
acgcaggaatggcaga
>taxon_2
acgcagcaagggacgttt
>taxon_3
acgcagctaccgacgttt
>taxon_4
acgttctacaccgacgttt
```

```
>taxon_5
attcaataccgacgttt
```

A sintaxe de substituição é super simples:
`'s/`*`conjunto de caracteres alvo/conjunto de caracteres de substituição/`*`'`

Há alguns detalhes que merecem atenção. Veja por exemplo o comando abaixo.

Execute o comando:

```
$ sed 's/a/A/' sed_example_1.txt
```

Você obterá:

```
>seq_1
Acgcaggaatggcaga
>seq_2
Acgcagcaagggacgttt
>seq_3
Acgcagctaccgacgttt
>seq_4
Acgttctacaccgacgttt
>seq_5
Attcaataccgacgttt
```

Observe que somente a substituição do primeiro par de base da sequência de cada linha foi modificado. Isso porque, por *default*, **Sed** faz a primeira modificação ao encontrar o padrão de busca e parte para a outra linha. No entanto, da mesma forma que o modificador **`p`** gerencia o padrão de impressão de **Sed** o modificador **`g`**, de *global*, faz com que o **Sed** verifique múltiplas ocorrências do padrão de busca na mesma linha. Veja como esse modificador funciona executando o comando abaixo:

```
$ sed 's/a/A/g' sed_example_1.txt
```

Você obterá:

```
>seq_1
AcgcAggAAtggcAgA
>seq_2
AcgcAgcAAgggAcgttt
>seq_3
AcgcAgctAccgAcgttt
```

```
>seq_4
AcgttctAcAccgAcgttt
>seq_5
AttcAAtAccgAcgttt
```

Há uma forma de implementar substituições sequenciais em **Sed** que é fundamental conhecer. Considere o exemplo abaixo.

Se você executar:

```
$ sed -e 's/a/A/g' -e 's/c/C/g' sed_example_1.txt
```

Você obterá:

```
>seq_1
ACgCAggAAtggCAgA
>seq_2
ACgCAgCAAgggACgttt
>seq_3
ACgCAgCtACCgACgttt
>seq_4
ACgttCtACACCgACgttt
>seq_5
AttCAAtACCgACgttt
```

Neste caso em particular, você poderia criar REGEX sequenciais e transformar todas as sequências em letras minúsculas. No entanto, considerem o exemplo abaixo:

Se você executar:

```
$  sed 's/^\w*/\U&\E/' sed_example_1.txt
```
[1]

Você obterá:

```
>seq_1
ACGCAGGAATGGCAGA
>seq_2
ACGCAGCAAGGGACGTTT
>seq_3
ACGCAGCTACCGACGTTT
>seq_4
ACGTTCTACACCGACGTTT
>seq_5
ATTCAATACCGACGTTT
```

[1] Se você estiver utilizando Mac OS X esse comando não funciona. Há variações nas versões de **sed** entre alguns sistemas operacionais.

Uau! Vamos entender as REGEXS utilizadas acima. Meu padrão de busca é "`^\w*`", ou seja, toda linha que inicie ("`^`") com um caráter alfanumérico ("`\w`") que seja seguido por 0 a infinito dos mesmos caracteres ("`*`"), veja Tabelas 2.3 e 2.4. Por outro lado, meu padrão de busca é "`\U&\E`" definem dois meta-caracteres desconhecidos até o momento para você. O meta-caráter "`\U`" transforma todas as letras em maiúsculas, "`&`", concatena o próximo meta-caráter "`\E`" que garante que as modificações efetuadas por "`\U`" sejam mantidas.

Exercício 2.11

Quanto tempo você leva para substituir os pontos (*i.e.*,"`.`") por vírgulas (*i.e.*,"`,`") do arquivo sed_Exercício_1.txt?

2.3.2 ENDEREÇAMENTO:

Como dito anteriormente, **Sed** aplica comandos de edição linha por linha, de seu início (*i.e.*, "`^`") ao fim (*i.e.*,"`$`"), principalmente a opção de edição global (*i.e.*,"`g`") é evocada. O endereçamento em **Sed** limita a ação das edições.

Sintaxe:

```
$ sed -opção 'ENDEREÇAMENTO+COMANDO_DE_EDIÇÃO'
arquivo_de_entrada
```

Vejamos na prática como isso funciona:

O arquivo `sed_example_2.txt` contém 17 linhas das quais 15 delas possuem todas as topologias possíveis para 5 terminais geradas em **TNT** [4]. Portanto, o arquivo possui a seguinte estrutura:

```
tread 'tree(s) from TNT, for data in tnt_gen_trees_T5.tnt'
(A (B (C (D E ))))*
(A (C (B (D E ))))*
...
(A (E (B (C D ))))*
(A (E (C (B D ))))*
(A (E (D (B C ))));
proc-;
```

Neste arquivo, a primeira e a última linha são parte do *tree block* do **TNT** e segue a sintaxe do programa para o comando `tread` cuja função é ler topologias.

Se você executar:

```
$ sed -n 'p' sed_example_2.txt
```

Sed executa o comando *print* (*i.e.*,"`p`") de todas as linhas do arquivo.

No entanto, se você executar:

```
$ sed -n '1p' sed_example_2.txt
```

Sed executa o comando *print* (*i.e.*,"`p`") apenas na primeira linha do arquivo (*i.e.*,"`1`") e você terá:

```
tread 'tree(s) from TNT, for data in tnt_gen_trees_T5.tnt'
```

Os limites de endereçamento são feitos da seguinte maneira.

Se você executar:

```
$ sed -n '1,3p' sed_example_2.txt
```

Sed imprime as 3 primeiras linhas do arquivo.

Observe como você combina o endereçamento com edição.

Se você executar:

```
$ sed -n '2,16 s/\s//gp' sed_example_2.txt
```

Sed executa o comando *print* (*i.e.*,"`p`") e substitui globalmente (*i.e.*,"`s`" com "`g`") as linhas 2 a 16 (*i.e.*,"`2,16`") e você terá:

```
(A(B(C(DE))))*
(A(C(B(DE))))*
...
(A(E(B(CD))))*
(A(E(C(BD))))*
(A(E(D(BC))));
```

Note que um caráter de escape (e meta-caráter) de REGEX "`\s`", que representa espaço em branco, foi utilizado. Portanto, embora nos exemplos acima tenhamos utilizados caracteres literais em substituições, `sed` permite o uso de REGEX.

Exercício 2.12

i. Neste exercício, você deverá usar os conceitos acima para fazer algumas edições no arquivo `sed_example_2.txt`. Uma dica importante: o argumento "`'1d;$d'`" de `sed` imprime todas as linhas do arquivo exceto a primeira e a última linha.

Observe atentamente a estrutura desse arquivo e transforme-o em um arquivo que deverá ser salvo com o nome de `sed_example_2.tre` que contenha apenas as topologias no seguinte formato:

```
(A,(B,(C,(D,E))));
(A,(C,(B,(D,E))));
...
(A,(E,(B,(C,D))));
(A,(E,(C,(B,D))));
(A,(E,(D,(B,C))));
```

Se você conseguiu fazer esse exercício parabéns! Você acaba de transformar um arquivo de topologias de TNT em um arquivo que pode ser lido por outros programas, tais como PAUP [7] e Figtree [8]!

ii. Neste exercício, você deverá criar uma arquivo texto no qual cada linha contenha o padrão de substituição que você utilizou no exercício acima. Por exemplo:

```
1d;$d s/A/B/g
s/X/Y/g
...
```

Você deverá salvar esse arquivo com o nome `tnt2figtree.sed`.

Qual o resultado da execução do comando abaixo?

```
$ sed -f tnt2figtree.sed sed_example_2.txt
```

Esse arquivo que contém as regras de substituições para trasnformar topologias vindas de TNT para que possam ser lidas em outros programas é últil. Guarde-o, pois você deverá usá-lo no futuro.

2.4 Trabalho para entregar

No diretório deste tutorial há dois arquivos destinados a este exercício:

1. `opilio_tree.tre`: Contém uma topologia no formato parentético.

2. `rename_opilio.csv`: Contém uma tabela no formato CSV com os nomes dos terminais que se encontram na topologia. Esse arquivo poder ser aberto em qualquer editor de texto ou no `LibreOffice Calc`. No entanto, voce poderá modificá-lo simplesmente utilizando o terminal e aplicando os conceitos que foram explorados nesse tutorial.

Com esses dois arquivos você deverá gerar uma figura editada da topologia usando Figtree e Inkscape em formato PDF. Para cumprir este objetivo, siga os seguintes passos:

a. Gerar um arquivo com as regras de substituição (veja Excercício 2.3.2) que lhe permita usar o **sed** para fazer as substituições dos terminais no arquivo `opilio_tree.tre` (veja seção 2.3.1).
Uma dica importante: O Figtree requer que os terminais que possuam nomes compostos (*e.g.*, *Hypophyllonomus longipes*), e que portanto, possuam espaços em branco entre palavras, estejam contidos entre aspas (*e.g.*, "`Hypophyllonomus longipes`"). **Sua topologia deverá conter todo o conteúdo da segunda coluna do arquivo CSV**.

b. Execute o **sed** redirecionando o resultado para um arquivo com outro nome.

c. Assista ao vídeo `figtree_1.ogv` para uma explicação breve de como o programa funciona e como você deverá proceder para gerar uma figura em SVG com ele.

d. Assista ao vídeo `inkscape_1.ogv` para uma explicação breve de como o programa funciona e como você deverá proceder para editar a figura e gerar um arquivo em formato PDF com resolução de 150 dpi chamado "`seu_nome_topologia_figura.pdf`".

e. Submeter a figura no formulário de *upload* ao final desta aula.

2.5 Referências

1. Wünschiers, R. 2004. Computational Biology: Unix/Linux, data processing and programming. Berlin, Germany: Springer, 2004. 284.

2. Varón, A.; Vinh, L. S. & Wheeler, W. C. 2010. POY version 4: phylogenetic analysis using dynamic homologies. *Cladistics* **26**: 72–85.

3. Varon, A.; Lucaroni, N.; Hong, L. & Wheeler, W. C. 2011–2014. POY version 4: phylogenetic analysis using dynamic homologies, version 5.0. New York, NY: American Museum of Natural History, 2011–2014.

4. Goloboff, P.; Farris, J. S. & Nixon, K. 2008. TNT a free program for phylogenetic analysis. *Cladistics* **24**: 1–14.

5. Zwickl, D. J. Genetic algorithm approaches for the phylogenetic analysis of large biological sequence datasets under the maximum likelihood criterion. Tese de doutorado (The University of Texas, Austin, 2006).

6. Goyvaerts, J. & Levitahn, S. 2009. Expressões Regulares Cookbook. São Paulo: Novatec Editora Ltda, 2009. 156.

7. Swofford, D. 2003–2016. PAUP*. Phylogenetic Analysis Using Parsimony (*and Other Methods, Version 4.0a131). Sunderland, Massachusetts: Sinauer Associates, 2003–2016.

8. Rambaut, A. Figtree: Tree Figure Drawing Tool. http://tree.bio.ed.ac.uk/software/figtree/. Version 1.3.1.