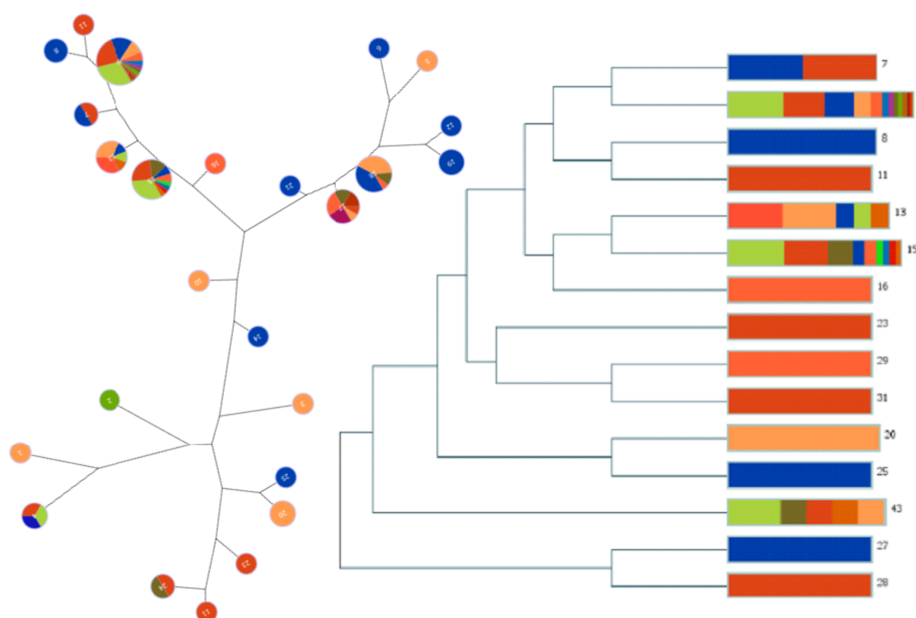
ISEL
INSTITUTO SUPERIOR DE
ENGENHARIA DE LISBOA

Visualização e Algoritmos de Clustering para Análise Filogenética

Adriano Sousa, 38210
Marta Nascimento, 38222



Orientadores: Cátia Vaz (ISEL)
João Carriço (IMM-FMUL)

Relatório final realizado no âmbito de Projeto e Seminário,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2014/2015

17 de Julho de 2015

Instituto Superior de Engenharia de Lisboa
Licenciatura em Engenharia Informática e de Computadores

Visualização e Algoritmos de Clustering para Análise Filogenética

Adriano Sousa, 38210
Marta Nascimento, 38222

Orientadores: Cátia Vaz (ISEL)
João Carriço (IMM-FMUL)

Relatório final realizado no âmbito de Projeto e Seminário,
do curso de licenciatura em Engenharia Informática e de Computadores
Semestre de Verão 2014/2015

17 de Julho de 2015

Resumo

Atualmente, com o avanço das tecnologias, têm sido desenvolvidos novos métodos de identificação de diferentes estirpes microbianas, chamados **métodos de tipagem**, que permitem avaliar as relações de descendência existentes entre as estirpes em estudo. Estes são fundamentais para os estudos epidemiológicos e para o estudo genético de populações microbianas, pois fornecem informação importante no controlo de doenças infecciosas e nos estudos sobre a patogénese, nomeadamente na evolução de infeções.

O **PHYLOViZ** é uma ferramenta que permite a análise, manipulação e visualização de diversos conjuntos de dados baseados em diferentes métodos de tipagem, com o objetivo de aprofundar os estudos epidemiológicos e de populações de bactérias. Permite ainda inferir padrões prováveis de descendência evolutiva entre perfis alélicos através do algoritmo **goeBURST** ou da sua expansão ***Full Minimum Spanning Tree*** (Full-MST).

Por vezes a realização apenas de uma inferência pode não ser suficiente. Logo, a existência de mais algoritmos que realizem esta inferência torna-se fundamental para que esta possa ser a mais aproximada da realidade. Isto faz também com que haja necessidade de guardar as diferentes inferências realizadas para evitar a repetição de todo o estudo. Ou seja, guardar o resultado gerado pelas inferências realizadas e das respetivas visualizações.

Deste modo, o objetivo deste projeto é desenvolver novos módulos para a ferramenta PHYLOViZ, permitindo a análise de dados de tipagem por outros algoritmos de inferência, assim como a visualização dos resultados dos mesmos. E ainda, incluir a possibilidade de persistir um conjuntos de dados e os seus resultados, introduzindo para isso a noção de projeto.

Palavras-chave: árvores filogenéticas; métodos de tipagem; persistência de dados;

Índice

1 Introdução **3**

2 Algoritmos para Análise Filogenética **7**

2.1 Descrição . . . 7

2.2 Algoritmo Hierárquico . . . 8

2.2.1 UPGMA . . . 8

2.2.2 Single-Linkage . . . 10

2.2.3 Complete-Linkage . . . 10

2.2.4 Observações . . . 11

2.3 Algoritmo Aglomerativo - Neighbor-Joining . . . 11

3 Implementação dos Algoritmos na Plataforma PHYLOViZ **15**

3.1 Plataforma NetBeans . . . 15

3.2 Plataforma PHYLOViZ . . . 16

3.3 Implementação . . . 18

3.3.1 Algoritmos UPGMA, Single-Linkage e Complete-Linkage . . . 18

3.3.2 Algoritmo Neighbor-Joining . . . 22

3.3.3 Estruturas de Dados . . . 24

4 Projeto PHYLOViZ e Serialização **27**

4.1 Estado do Projeto . . . 27

4.2 Serialização . . . 29

5 Visualização **33**

5.1 UPGMA . . . 33

5.1.1 Funcionalidades . . . 35

5.2 Neighbor-Joining . . . 36

5.2.1 Funcionalidades . . . 38

5.3 Comparação com softwares já existentes . . . 39

6 Avaliação Experimental **41**

6.1 Testes Unitários . . . 41

7 Conclusão **45**

A Schema para algoritmos hierárquicos **53**

Lista de Figuras

2.1 Exemplos de representações de árvores filogenéticas 8
2.2 Par 1 e 2 com metade da distância entre eles. 9
2.3 Representação final da árvore. 10
2.4 Exemplos de diferentes representações de árvores consoante o método escolhido. 11
2.5 Exemplos de representações de possíveis estados da árvore. 12
2.6 Representação da árvore gerada pelo algoritmo Neighbor-Joining para a tabela 2.1. 13

3.1 Representação UML do módulo Core. 16
3.2 Inserir novos módulos na plataforma PHYLOViZ 17
3.3 Representação da relação de dependência na integração do módulo `PHYLOViZ UPGMA` com os existentes. 18
3.4 Representação UML parcial do módulo `PHYLOViZ UPGMA`. 19
3.5 Representação interna do módulo `PHYLOViZ UPGMA`. 19
3.6 Representação UML do módulo `PHYLOViZ Neighbor-Joining`. 22
3.7 Representação da ligação entre nós de distância e a lista de indexação. 25
3.8 Representação da ligação entre nós de distância e a lista de indexação após remoção. 26

4.1 Representação de um DataSet. 28
4.2 Interfaces `ProjectItem` e `ProjectFactory`. 28

5.1 Framework `prefuse` 33
5.2 Visualizações da classe TreeView. 35
5.3 Visualização de UPGMA para ficheiro com 20 Táxon. 35
5.4 Visualização de Neighbor-Joining para ficheiro com 5 entradas. 37
5.5 Visualização de Neighbor-Joining para ficheiro com 500 entradas. 37
5.6 Visualização parcial de Neighbor-Joining para ficheiro com 500 entradas. . . . 38
5.7 Visualização parcial de UPGMA com integração de dados, nomeadamente por país de ocorrência. 40

6.1 Representação gráfica da tabela 6.1 42

6.2 Representação gráfica da tabela 6.2 43

Lista de Tabelas

2.1 Dados Multi-Locus Sequence Typing 9
2.2 Matriz de distâncias - Triangular Inferior. 9
2.3 Atualização da matriz de distâncias 10

3.1 Custos dos algoritmos hierárquicos. 21
3.2 Custo do algoritmo aglomerativo Neighbor-Joining. 24

5.1 Comparação entre as ferramentas PHYLOViZ e SplitsTree. 39

6.1 Média de tempos obtidos durante a execução dos testes do UPGMA, versão 1 e 2, sendo K o número de ciclos (50). 41
6.2 Tempos obtidos durante a execução os testes de Neighbor-Joining, versão 1 e final. 42

Capítulo 1

Introdução

As sequências biológicas, nomeadamente o **DNA** (ácido desoxirribonucleico), **RNA** (ácido ribonucleico) e **proteínas**, têm um papel fundamental na biologia molecular porque definem quase todas as atividades celulares que ocorrem em cada organismo. A chave para decifrar estes processos recai em compreender como estas sequências interagem umas com as outras e com o seu ambiente envolvente.

O **DNA** é uma macromolécula fundamental que contém o código genético da célula. É composto por duas cadeias de nucleótidos, entrelaçadas entre si, formando assim uma dupla hélice. Esta por sua vez é formada por diferentes bases unidas por pontes de hidrogénio. As quatro bases encontradas no DNA são a **adenina** (A), **citosina** (C), **guanina** (G) e **timina** (T). Encontram-se ligadas a um açúcar que por sua vez se liga a um grupo fosfato, formando assim um nucleótido completo. Os segmentos de DNA que contêm a informação genética são denominados de **genes**. Estes são a unidade molecular fundamental da hereditariedade de um organismo e apresentam variantes com sequências específicas de DNA denominadas **alelos**. A restante sequência de DNA tem importância estrutural ou está envolvida na regulação do uso da informação genética.

A sequenciação de DNA é um processo que permite determinar a ordem precisa dos nucleótidos numa molécula DNA.

Quando se obtém uma amostra microbiana, seja de um indivíduo infetado ou do meio ambiente, esta pode ser analisada criando culturas puras (*i.e.* de uma única espécie). As culturas puras são depois manipuladas em placas de petri de modo a obter colónias distintas na placa, que corresponderão cada uma a uma **estirpe isolada**.

As tecnologias mais recentes permitem a sequenciação do DNA e RNA muito mais rapidamente e de forma mais barata. Esta tecnologia é denominada de **"High Throughput Sequencing"** ou sequenciação de alto débito, produzem grandes quantidades de informação em forma de milhões de "short reads", isto é, pequenos pedaços de sequências da amostra original de DNA. Estes não possuem qualquer ordem entre si nem uma localização conhecida. Também podem ter comprimentos arbitrários, de cadeia indeterminada e com um número arbitrário de cópias sobrepostas, e por vezes com erros de sequenciação.

Após a reconstrução de um genoma parcial através da montagem das "reads"(*i.e.* avaliação da sobreposição de nucleótidos entre elas em sequências de grande dimensão denominadas "contigs"), é necessário identificar os genes que compõem o genoma. Para tal, utiliza-se uma base de dados de genes conhecidos para a mesma espécie do organismo que se sequenciou e tenta-se extrair os genes que ocorrem nesse genoma. A correspondência é feita com base num alelo específico que já foi previamente identificado.

Assim, através da comparação entre vários genes, é possível identificar a estirpe do organismo, isto é, a variante genética ou subtipo do organismo dentro da sua espécie. A identificação da estirpe do organismo é, em termos microbiológicos, designada por **tipagem**. O objetivo dos estudos de tipagem é permitir a avaliação de relações de descendência entre as estirpes em estudo. Os métodos de tipagem baseados em sequências representam as estirpes por sequências de caracteres.

A escolha do método de tipagem a utilizar depende do problema a resolver e do contexto epidemiológico no qual se vai utilizar o método. Estes métodos baseados em sequências permitem a análise ao nível da estirpe, fornecendo conhecimento importante para a vigilância das doenças infecciosas, investigação de surtos e a história natural de uma infeção. Além disso, com a recente introdução de tecnologias **"High Throughput Sequencing"** (HTS) e a possibilidade de ter acesso ao sequenciamento do genoma de uma estirpe microbiana em poucos dias, têm sido desenvolvidos novos métodos de tipagem, como por exemplo o **Multilocus Sequence Typing ribossomal** (MLST-ribossomal)[1] ou análise de **Single Nucleotide Polymorphism** (SNPs) [2] através da comparação com um genoma de referência. Estes avanços criaram a necessidade de algoritmos e ferramentas de processamento, análise e visualização desses dados, no contexto da epidemiologia, genética populacional e evolução.

Ao analisar um conjunto de isolados através de um método de tipagem, as relações inferidas entre os vários isolados é realizada recorrendo à utilização de algoritmos de inferência de árvores de filogenia.

O PHYLOViZ [3] é um *software* desenvolvido em Java e está acessível para todas as plataformas. Permite a análise e manipulação de diferentes conjuntos de dados baseados em diferentes métodos de tipagem, com o objetivo de aprofundar os estudos epidemiológicos e de populações de bactérias. Permite ainda inferir padrões prováveis de descendência evolutiva entre perfis alélicos através do algoritmo **goeBURST** ou da sua expansão **Full Minimum Spanning Tree** (Full-MST) baseados em diferentes matrizes distância. Posteriormente, é feita a visualização da árvore de filogenia correspondente.

Dependendo dos dados, por vezes é útil aplicar mais do que uma metodologia de análise. Isto também pode acontecer caso a visualização não seja a mais adequada a esse tipo de dados. Assim, através da implementação no PHYLOViZ de outros métodos de *clustering*, como **UPGMA**[4] e **Neighbor-Joining**[5], pode-se realizar uma inferência mais aproximada da realidade, e conseguir com a sua visualização, oferecer mais informação com vista a melhorar

esta análise.

Assim, este projeto tem como objetivos:

- implementação de dois tipos de *clustering*, um hierárquico e outro aglomerativo. Nomeadamente os algoritmos **UPGMA**, **Single-Linkage** e **Complete-Linkage** como algoritmos hierárquicos e **Neighbor-Joining** como algoritmo algomerativo.
- visualização dos *outputs* gerados para cada um dos algoritmos mencionados anteriormente. Nomeadamente para o hierárquico uma representação no formato de um dendrograma e para o aglomerativo uma visualização em árvore;
- associação de dados complementares às visualizações;
- possibilidade de efetuar cortes nas árvores resultantes dos algoritmos de *clustering*. Isto é, dividir a representação dos dados em diferentes grupos, consoante a sua distância.
- guardar e carregar o estado de processamento de um DataSet (persistência) através da serialização dos *outputs* dos algoritmos e das respetivas visualizações;

Isto irá tornar possível a realização de inferência filogenética com mais do que uma abordagem utilizando a mesma ferramenta, podendo-se associar à visualização os dados complementares e permitindo assim a análise de semelhança entre os diferentes modelos de inferência. Permitirá também evitar a repetição de todo o estudo uma vez que este poderá ser guardado e novamente carregado sem qualquer custo computacional adicional. Isto é bastante importante uma vez que, como após a geração das árvores podem ser feitas alterações à mesma (*e.g.* ajustar as posições dos elementos visuais), torna-se imprescindível que, quando se quiser repetir o estudo, se obtenha exatamente a mesma árvore.

No próximo capítulo, 2, irá ser abordada a importância destes algoritmos para a análise filogenética. O capítulo 3 irá conter uma breve descrição da plataforma PHYLOViZ, a implementação dos algoritmos mencionados anteriormente e a sua integração nesta plataforma. O capítulo 4 irá abordar o conceito de Projeto PHYLOViZ. A visualização do *output* dos algoritmos será apresentada no capítulo 5. No capítulo 6, será apresentada uma avaliação experimental, tanto aos algoritmos como na comparação com outros softwares já existentes. E por fim, no capítulo 7, serão abordadas as conclusões do projeto.

Capítulo 2

Algoritmos para Análise Filogenética

2.1 Descrição

A filogenia [6] é o estudo da relação evolutiva existente entre diferentes grupos de organismos, tais como espécies, populações, etc, que provêm de um antecessor comum.

A análise filogenética permite, através da comparação de genes equivalentes provenientes de diversas espécies, inferir as suas relações. Isto é conseguido agrupando os organismos de acordo com o seu nível de similaridade, ou seja, quanto mais semelhantes são as espécies mais perto se encontram do antecessor comum. A representação destas relações de evolução entre grupos de organismos pode ser feita através de dendrogramas ou árvores, também conhecidas como árvores filogenéticas. Esta árvore é composta por nós e ramos. Um nó representa uma unidade taxonómica (sequência ou *táxon*[1]) e cada ramo liga quaisquer dois nós adjacentes.

As árvores filogenéticas dividem-se em dois grandes grupos, com raiz e sem raiz. As árvores filogenéticas com raiz dividem-se em folhas, nós, ramos e raiz. As folhas representam as sequências (*taxa*[2]). Os nós representam a relação existente entre duas sequências, isto é, o antecessor comum a ambas. Por fim os ramos representam a ligação entre estas sequências e o seu tamanho o número de alterações genéticas ocorridas até à próxima separação. A raiz representa o antecessor comum a todas as *taxa* - Figura 2.1a. As árvores filogenéticas sem raiz diferem das anteriores uma vez que estas últimas não apresentam raiz e o tamanho dos seus ramos não representa qualquer tipo de informação - Figura 2.1b.

Existem vários métodos para a construção de árvores filogenéticas, mas apenas nos iremos focar no método baseado em matrizes de distância. Este baseia-se no número de diferenças genéticas entre pares de sequências (normalmente distância de Hamming[7]) e utiliza uma matriz durante a criação da árvore. Existem diversos algoritmos, hierárquicos e aglomerativos que seguem este princípio nomeadamente o UPGMA, Single-Linkage, Complete-Linkage como hierárquicos e o Neighbor-Joining como algomerativo.

[1] Táxon - unidade taxonómica de classificação de seres vivos.
[2] Taxa - plural de Táxon.

(a) Com raiz

(b) Sem raiz

Figura 2.1: Exemplos de representações de árvores filogenéticas

2.2 Algoritmo Hierárquico

Os métodos hierárquicos são dos métodos mais utilizados para realizar inferência filogenética, através de árvores filogenéticas com raiz. Procuram construir uma hierarquia de grupos (*clusters*) utilizando para isso uma estratégia aglomerativa. Isto é, começa por apresentar apenas nós simples, agrupando-os posteriormente e subindo um nível na hierarquia até atingir o ancestral comum (raiz).

A cada passo, os dois *clusters* mais próximos (distância mínima) são agrupados formando um novo *cluster* (Táxon $\mathcal{T}$), subindo assim o nível hierárquico. A distância entre estes dois *clusters* depende do método hierárquico escolhido, isto é, do critério de ligação entre os *clusters*.

2.2.1 UPGMA

O UPGMA (*Unweighted Pair Group Method with Arithmetic Mean*) é um dos métodos hierárquicos mais utilizados.

O critério de ligação (cálculo da distância) entre dois *clusters* $\mathcal{T}_1$ e $\mathcal{T}_2$ é o resultado da média das distâncias d entre o par de elementos i em $\mathcal{T}_1$ e j em $\mathcal{T}_2$, ou seja, a média das distâncias entre os elementos de cada *cluster*, como demonstrado pela fórmula seguinte:

$$\frac{1}{|\mathcal{T}_1|.|\mathcal{T}_2|}\sum_{i\in\mathcal{T}_1}\sum_{j\in\mathcal{T}_2} d_{ij} \tag{2.1}$$

Este algoritmo tem um custo de execução de $O(n^2)$, sendo n o número inicial de *clusters*.

Exemplo

Considere-se o conjunto de dados MLST da Tabela 2.1 e a respetiva matriz de distâncias representada na Tabela 2.2.

Sequência	gki	gtr	murI	mutS	recP	xpt	yqiZ
ST-11	2	3	4	3	8	4	5
ST-1149	2	3	4	53	8	4	5
ST-1149-2	2	3	4	53	9	4	5
ST-658	2	3	4	3	8	110	5
ST-50	2	3	19	3	8	4	5

Tabela 2.1: Dados Multi-Locus Sequence Typing

Sequência	ST-11	ST-1149	ST-1149-2	ST-658	ST-50
ST-11	0	1	2	1	1
ST-1149	1	0	1	3	2
ST-1149-2	2	1	0	4	3
ST-658	2	3	4	0	0
ST-50	1	2	3	3	0

Tabela 2.2: Matriz de distâncias - Triangular Inferior.

Primeiramente, e para facilitar a identificação, as sequências passarão a ser identificadas numericamente (*e.g.* ST-11 corresponderá a 1, ST-1149 a 2, etc.). O próximo passo será identificar o par (*cluster*) com distância mínima na matriz e criar um novo nó que contenha este par. Como existem diferentes possibilidades, isto é, com a distância mínima de 1, então será escolhido um dos pares aleatoriamente, nomeadamente os pares 1 e 2. Seguidamente, irá ser feita a representação do novo nó, U_1, e dos seus respetivos ramos e definido o tamanho destes como metade da distância entre o par anterior - Figura 2.2.

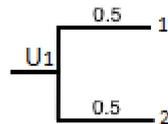


Figura 2.2: Par 1 e 2 com metade da distância entre eles.

Uma vez criada a união U_1, será necessário voltar a calcular a nova matriz de distâncias. Este passo é conseguido removendo as entradas dos pares que foram anteriormente unidos, acrescentando o novo nó criado no passo anterior e recalculando as novas distâncias ao novo nó através da média das distâncias entre os elementos de cada *cluster*- Tabela 2.3. Por exemplo, para a nova distância entre U_1 e 3, irá resultar:

$$d_{U_1 3} = \frac{d_{13} + d_{23}}{2} = \frac{2+1}{2} = 1.5 \tag{2.2}$$

Sequência	U_1	3	4	5
U_1	0	1.5	2.5	1.5
3	1.5	0	4	3
4	2.5	4	0	3
5	1.5	3	3	0

Tabela 2.3: Atualização da matriz de distâncias

Finalmente, se a matriz ficar apenas com uma entrada, cria-se um último nó unindo esses dois elementos da matriz. Caso contrário, repete-se todo o ciclo.

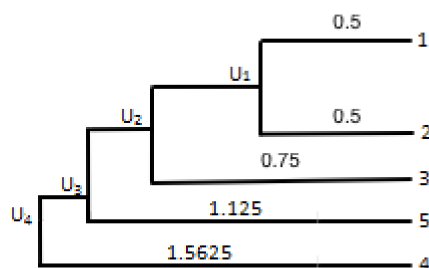


Figura 2.3: Representação final da árvore.

2.2.2 Single-Linkage

O método SL (*Single-Linkage*) permite calcular a distância entre dois *clusters* $\mathcal{T}_1$ e $\mathcal{T}_2$ como a mínima distância d entre par de elementos i em $\mathcal{T}_1$ e j em $\mathcal{T}_2$, como demonstrado pela fórmula seguinte:

$$\min_{i \in \mathcal{T}_1, j \in \mathcal{T}_2} d_{ij} \tag{2.3}$$

Em relação ao Exemplo da Secção 2.2.1 e uma vez que o comportamento deste método é idêntico ao UPGMA, apenas será necessário substituir a Fórmula 2.2 pela Fórmula 2.3, logo, substituir o critério de ligação.

Este algoritmo tem um custo de execução de $O(n^2)$, sendo n o número inicial de *clusters*.

2.2.3 Complete-Linkage

O método CL (*Complete-Linkage*) permite calcular a distância entre dois *clusters* $\mathcal{T}_1$ e $\mathcal{T}_2$ como a máxima distância d entre par de elementos i em $\mathcal{T}_1$ e j em $\mathcal{T}_2$, como demonstrado pela fórmula seguinte:

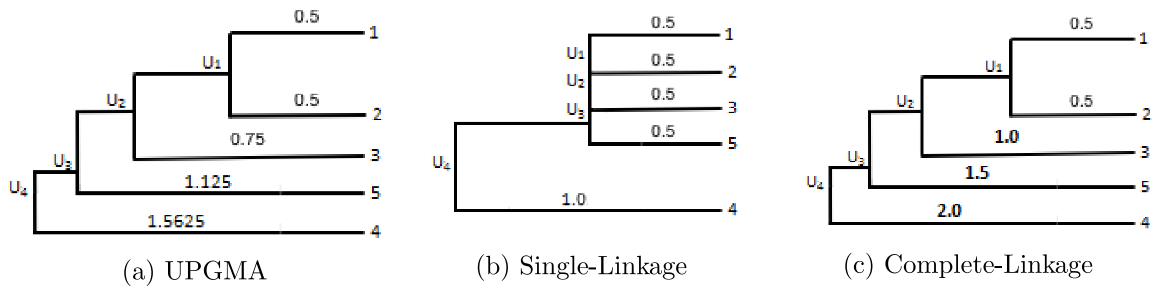
$$\max_{i \in \mathcal{T}_1, j \in \mathcal{T}_2} d_{ij} \tag{2.4}$$

Em relação ao Exemplo da Secção 2.2.1 e uma vez que o comportamento deste método é idêntico ao UPGMA, apenas será necessário substituir a Fórmula 2.2 pela Fórmula 2.4, logo, substituir o critério de ligação.

Este algoritmo tem um custo de execução de $O(n^2)$, sendo n o número inicial de *clusters*.

2.2.4 Observações

Como é possível verificar na Figura 2.4, dependendo do método para cálculo das distâncias, são geradas árvores diferentes.



(a) UPGMA (b) Single-Linkage (c) Complete-Linkage

Figura 2.4: Exemplos de diferentes representações de árvores consoante o método escolhido.

Na presença de um conjunto de dados maior, estas diferenças irão-se acentuar mais e poderão gerar árvores muito diferentes entre si.

Como o método hierárquico Single-Linkage agrupa os pares de *clusters* com base na distância mínima, a sua visualização irá representar todos os grupos mais próximos. Já com o Complete-Linkage, a sua visualização irá representar os grupos mais afastados entre si, uma vez que se baseia na distância máxima entre dois *clusters*.

Caso existam empates no valor da menor distância é escolhido o primeiro valor encontrado, logo, o primeiro par de *clusters*. Se fosse escolhido outro critério de desempate, poderia dar origem a uma árvore de filogenia diferente.

2.3 Algoritmo Aglomerativo - Neighbor-Joining

Neighbor Joining é um método para reconstrução de árvores filogenéticas a partir de uma matriz de distâncias utilizando o princípio da evolução mínima e calculando os comprimentos de cada ramo dessa árvore.

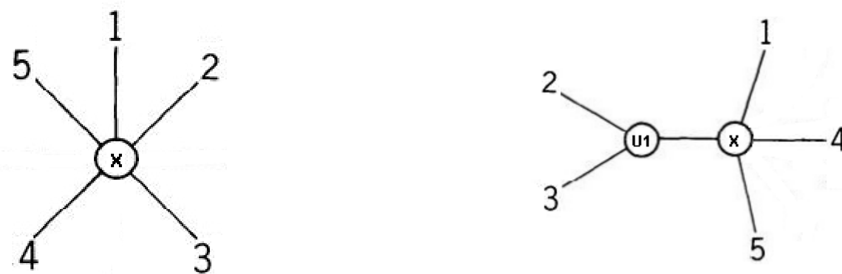
Este algoritmo foi originalmente escrito em 1987 por *Saitou and Nei*, sendo corrigido em 1988. Este método procura construir uma árvore que minimize a soma das distâncias de todos os ramos, adotando os critérios da evolução mínima [8].

Neighbor-Joining pode ser visto como um algoritmo ganancioso por querer optimizar uma árvore seguindo o critério da "Evolução Mínima Equilibrada"(BME - Balanced Minimum

Evolution). Por cada topologia, BME define o comprimento dos ramos para que seja uma soma das distâncias presentes na matriz de distâncias, sendo estas distâncias dependentes da topologia utilizada.

A topologia de BME perfeita será aquela que minimize a distância de todos os ramos. O algoritmo de *Neighbor-Joining* tenta então ligar os pares de *taxa* cuja distância seja a menor possível para o cálculo estimado do comprimento da árvore. Este procedimento não garante que no final a árvore obtida terá a melhor topologia BME possível.

Nós vizinhos são definidos como um par de *Táxon*, que têm uma ligação entre eles. No caso da figura 2.5, é possível ver como a árvore é inicialmente e como esta fica após agrupado o primeiro par.



(a) Estado da árvore inicial. (b) árvore após *Taxon* 2 e 3 terem sido agrupados.

Figura 2.5: Exemplos de representações de possíveis estados da árvore.

Nem sempre a árvore de evolução gerada é a árvore de evolução mínima, pois minimizar o comprimento da árvore a cada passo do algoritmo não implica minimizar o comprimento da árvore obtida no final do processo. Isto acontece porque a soma das distâncias mínimas calculadas a cada iteração pode diferir conforme a escolha feita nos casos de empate. Um empate acontece quando uma distância mínima é encontrada em vários pares de nós, podendo a escolha recair sobre qualquer um.

Começando com uma árvore em forma de estrela, recursivamente é identificado o par de nós vizinhos com menor distância (soma mínima dos comprimentos destes ramos). Para isto é utilizada uma matriz de distâncias, onde cada elemento deverá representar a relação de distância entre todos os nós.

Este algoritmo é então dividido em várias passos que se irão repetir N - 2 vezes (onde N é o número de elementos a calcular no momento).

Inicialmente é necessário realizar o cálculo da soma, S, de um dado nó i para todos os restantes nós - Passo 1.

Em seguida é então procurado qual o par de nós (i, j) que terá a menor distância, M_{ij}, - Passo 2.

Tendo sido encontrado o par de nós, é então possível proceder para a união dos mesmos. Com a criação desta união x, é necessário calcular qual a distância dos seus ramos (R_x) -

Passo 3. São então necessárias calcular duas distâncias, uma para cada elemento do par de nós de menor distância, R_{ix} e R_{jx}.

Após a criação da união, este é então colocado na árvore, substituindo o par de elementos que este representa, como demonstrado na Figura 2.5b.

Com um novo nó na árvore, a matriz de distâncias deve ser recalculada, removendo assim as distâncias para os nós antigos e inserindo a nova distância para o nó de união. Esta poderá ser calculada através da Fórmula 2.5.

$$d_{xu} = \frac{d_{ix} + d_{jx} - d_{ij}}{2} \tag{2.5}$$

Na Fórmula 2.5, i e j são os nós vizinhos selecionados anteriormente e x é o novo nó criado.

Calculada essa distância para o resto dos nós, esta é então colocada na matriz de distâncias.

Todo este procedimento é então repetido até que apenas exista uma matriz de distâncias com dois elementos, completando a árvore com o último valor existente na matriz.

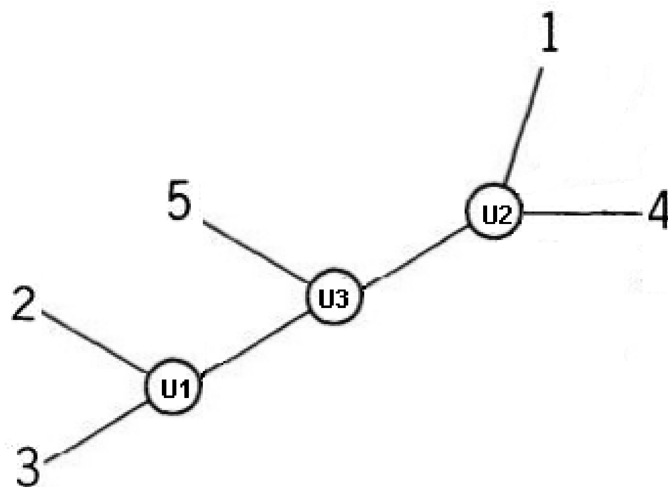


Figura 2.6: Representação da árvore gerada pelo algoritmo Neighbor-Joining para a tabela 2.1.

Desta forma são então disponibilizados dois critério de otimização, o critério de *Studier and Keppler* e de *Saitou N. and Nei M.*, sendo estas semelhantes mas com fórmulas de cálculo diferentes.

Estas diferenças são apresentadas abaixo:

Critério Studier and Keppler

1. **Passo 1:**

$$S_x = \sum_{ij \in \mathcal{T}} d_{ij}$$

2. **Passo 2:**

$$M_{ij} = (N-2)D_{ij} - S_i - S_j$$

3. **Passo 3:**

$$R_{iu} = \frac{D_{ij}}{2} + \frac{1}{2(n-2)}(S_i - S_j)$$
$$R_{ju} = D_{ij} - R_{iu}$$

Critério Saitou N. and Nei M.

1. **Passo 1:**

$$S_x = \frac{\sum_{ij \in \mathcal{T}} d_{ij}}{N-2}$$

2. **Passo 2:**

$$M_{ij} = D_{ij} - S_i - S_j$$

3. **Passo 3:**

$$R_{iu} = \frac{D_{ij}}{2} + \frac{S_i - S_j}{2}$$

Capítulo 3

Implementação dos Algoritmos na Plataforma PHYLOViZ

A plataforma Netbeans [9], inclui um sistema de *plugins*, o que permite desenvolver software de uma forma modular e extensível, sendo mais simples a integração de novos métodos de inferência filogenética.

Os utilizadores da ferramenta poderão também desenvolver e utilizar novos *plugins* de acordo com os seus tipos de dados e torná-los disponíveis para a comunidade, através do seu próprio *website* ou de um repositório.

3.1 Plataforma NetBeans

O NetBeans IDE (Ambiente de Desenvolvimento Integrado) é um ambiente de desenvolvimento de software integrado *open source*, utilizado para o desenvolvimento de aplicações em sistemas operativos Windows, Linux, Mac e Solaris.

Existem muitas razões pelo qual se deve utilizar esta plataforma, algumas delas são:

- Fornece as funcionalidades necessárias para o desenvolvimento do software e uma GUI (Interface Gráfica para o Utilizador) bastante intuitiva.

- Interface de desenvolvimento gráfico rápido e fácil. Fornece várias formas de construção e editores de "arrastar e largar"de forma fácil e eficiente, que em outros IDEs se torna confuso e cansativo de perceber.

- Boa estrutura, construída a volta de módulos. Nas outras plataformas como Eclipse não existe esta funcionalidade, trabalhando apenas no projeto. No NetBeans um projeto é constituído por vários módulos, permitindo uma melhor organização.

  A tecnologia de criação de *plugins* para a plataforma PHYLOViZ é a mesma utilizada na criação de *plugins* para a plataforma NetBeans. Isto permite a utilização de todas as caraterísticas fornecidas pela plataforma.

3.2 Plataforma PHYLOViZ

A plataforma PHYLOViZ[3] foi construída através de módulos. Isto proporciona uma maior facilidade na adição de funcionalidades extra.

A implementação de novas funcionalidades implica a criação de novos módulos (plugins). Assim, cada algoritmo de *clustering* implementado corresponde a um módulo diferente, assim como as suas respetivas visualizações. Isto permite uma rápida integração com a plataforma já existente.

Para que esta integração seja possível, é necessário que estes novos módulos interatuem com os já existentes, possibilitando assim disponibilizar as novas funcionalidades. Isto implica a análise de diversos módulos e das suas funcionalidades.

Existem vários módulos, sendo o mais importante o módulo **Core** que contém a lógica necessária aos restantes - Figura 3.1.

Outros módulos relevantes:

- **Algorithms** - fornece código reutilizável e um nível de abstração em relação aos restantes módulos responsáveis pelas implementações concretas de algoritmos.
- **MLST [1, 10]** - responsável pela implementação de um método de tipagem, nomeadamente, dados relativos a Multi-Locus Sequence Typing.
- **GTViewer** - responsável pela visualização de árvores, grafos de filogenias, incluindo integração de dados, baseado na biblioteca Prefuse [11].
- **goeBurst** - fornece uma implementação do algoritmo goeBurst.
- **Category** - responsável por incluir na visualização gráficos circulares, através da aplicação de filtros.

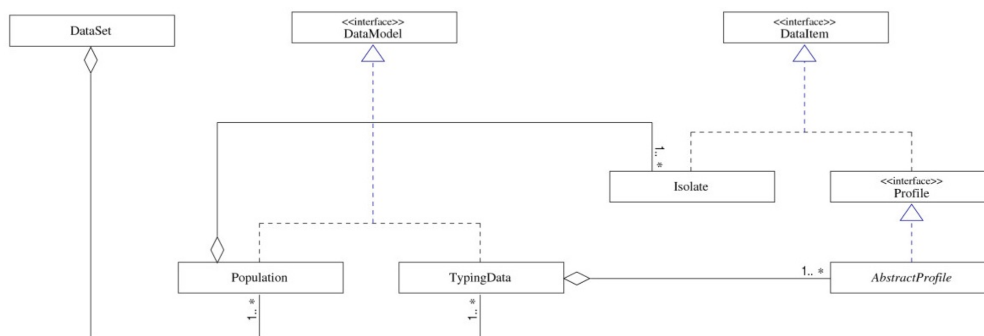


Figura 3.1: Representação UML do módulo Core.

Descrição da funcionalidade das classes mais relevantes existentes no módulo Core:

- **Isolate** - Representa as amostras dos microrganismos das populações bacterianas.

- **Population** - Representa o conjunto de *isolates* de uma população bacteriana.
- **TypingData** - Representa o conjunto de perfis isolados criada por um método de tipagem microbiana - Tabela 2.1.
- **DataSet** - Representa um conjunto que contém a informação da *Population* e o *TypingData* associado.

A utilização destes novos módulos é simples utilizando a adição de novas utilidades por *plugins*. Tal como referido anteriormente foram criados módulos com as novas funcionalidades, onde através de um simples carregamento a aplicação fica pronta a usar.

Selecionando no menu superior o botão de *Tools*, seguido de *Plugins*, irá aparecer uma nova janela onde é possível observar todas as opções sobre os *plugins*.

Em *Updates* poderemos saber se existem novas atualizações, através de uma ligação à internet e fazendo o *download* das mesmas caso desejado. Em *Downloaded* é possivel fazer o carregamento de todos os *plugins*, desde que o *download* destes já se encontre realizado, tal como demonstrado na Figura 3.2.

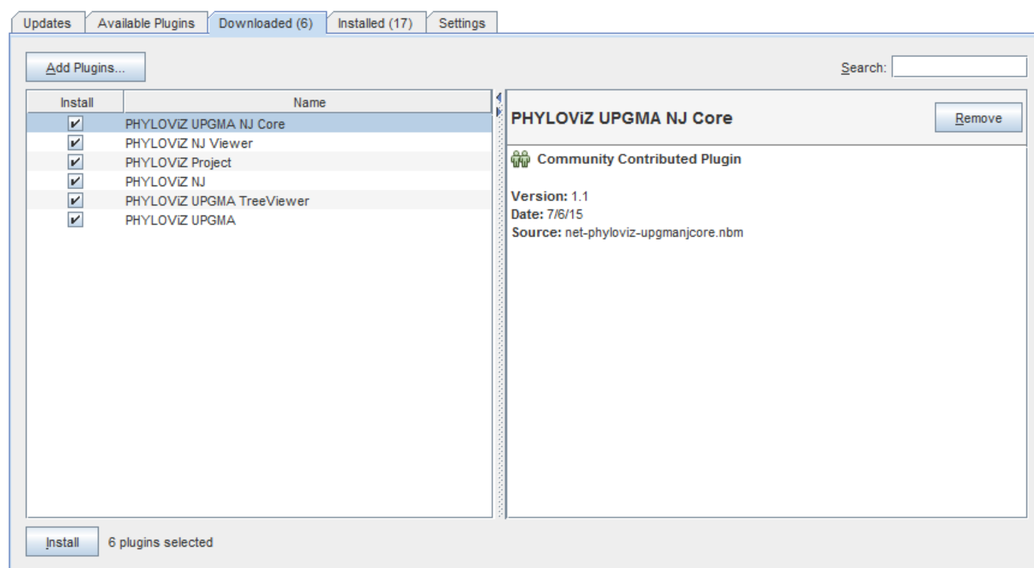


Figura 3.2: Inserir novos módulos na plataforma PHYLOViZ

Na Figura 3.2 podemos ver os *plugins* implementados para os novos algoritmos. O carregamento destes deve ser feito por ordem de dependências, caso contrário dará um aviso a indicar que falta instalar os plugins dos quais depende. Isto acontece devido às suas dependências. O *plugin* `PHYLOViZ NJ Viewer` depende do *plugin* `PHYLOViZ NJ`, ou seja, primeiro terá de ser instalado `PHYLOViZ NJ`. O mesmo acontece com os *plugins* do UPGMA, devendo ser instalado em primeiro lugar o `PHYLOViZ UPGMA`. Tanto este último como o `PHYLOViZ NJ` dependem por sua vez de `PHYLOViZ UPGMA NJ Core`, que contém toda a lógica comum a ambos os algoritmos e respetivas visualizações.

3.3 Implementação

A implementação dos algoritmos implicou um conhecimento aprofundado dos módulos `PHYLOViZ Core` e `PHYLOViZ Algorithms`. Isto foi necessário uma vez que era pretendido integrar os novos algoritmos com os já existentes na plataforma.

Após o estudo dos algoritmos, e uma vez que estes são baseados em matrizes de distância, iniciou-se a implementação criando uma matriz e guardando-a em memória. Rapidamente se percebeu que essa não seria a melhor solução uma vez que a principal característica desta matriz é ser triangular inferior - Tabela 2.2. Isto irá fazer com que apenas seja necessário guardar metade dessa matriz e, consequentemente, optimizar a quantidade de memória utilizada. A criação desta matriz tem o custo de $O(N)$, com N sendo o número de perfis a analisar.

3.3.1 Algoritmos UPGMA, Single-Linkage e Complete-Linkage

Para implementar estes algoritmos foi necessário adicionar um novo módulo `PHYLOViZ UPGMA`. Este módulo é responsável por toda a lógica necessária tanto à implementação como à integração destes algoritmos na plataforma existente - Figura 3.3 e 3.4.

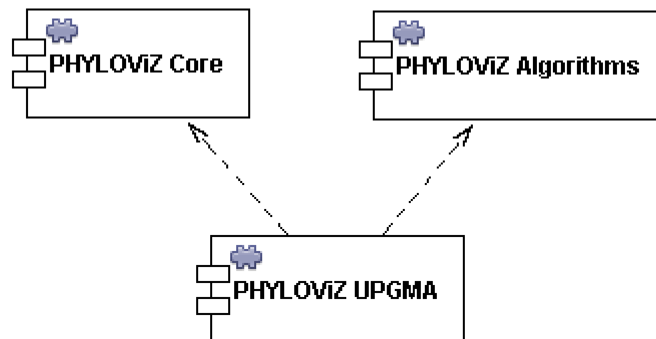


Figura 3.3: Representação da relação de dependência na integração do módulo `PHYLOViZ UPGMA` com os existentes.

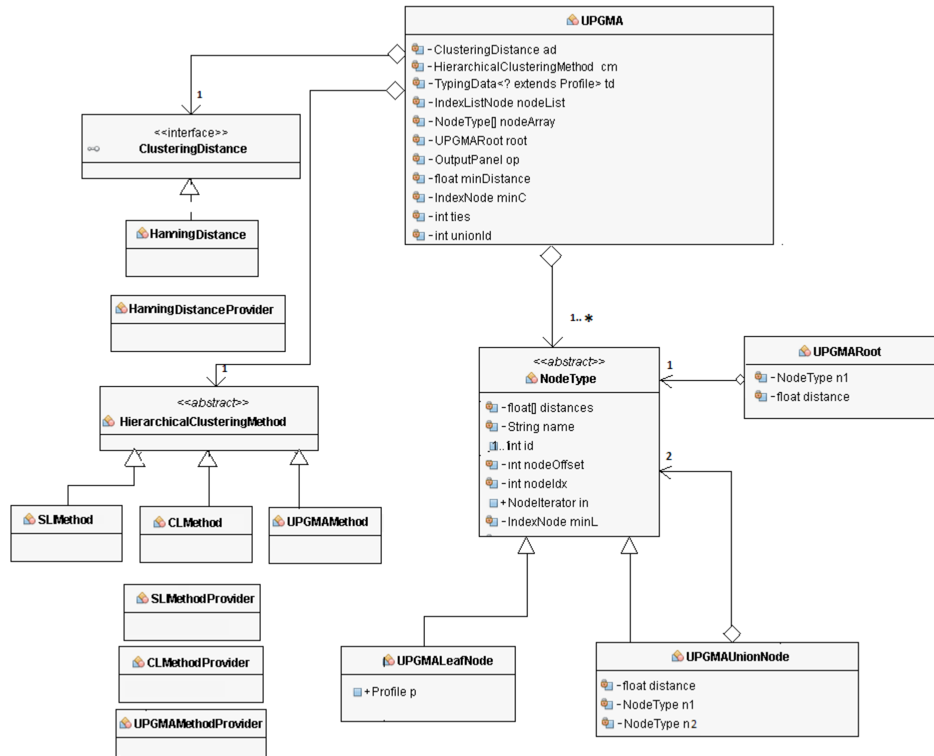


Figura 3.4: Representação UML parcial do módulo `PHYLOViZ UPGMA`.

Este módulo foi dividido em diferentes *packages* - Figura 3.5. Isto permitiu uma melhor organização e estruturação do código.

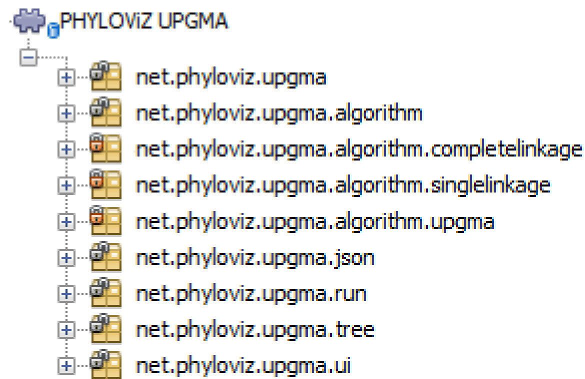


Figura 3.5: Representação interna do módulo `PHYLOViZ UPGMA`.

As principais responsabilidades destes *packages* são:

- `net.phyloviz.upgma` e `net.phyloviz.upgma.run` - contêm a lógica necessária para integrar estes novos métodos de *clustering* na plataforma.

- `net.phyloviz.upgma.algorithm` - responsável pela implementação dos algoritmos.

- `net.phyloviz.upgma.algorithm.completelinkage`, `net.phyloviz.upgma.algorithm.singlelinkage` e `net.phyloviz.upgma.algorithm.upgma` - definem o critério de ligação respetivo de cada um.

- `net.phyloviz.upgma.json` - responsável pela serialização do *output*.

- `net.phyloviz.upgma.ui` - responsável por perguntar ao utilizador sobre o critério de ligação pretendido.

- `net.phyloviz.upgma.tree` - contém a estrutura interna necessária à implementação do algoritmo.

Inicialmente, a implementação deste algoritmo não satisfazia os objetivos esperados, nomeadamente no tempo de execução e na memória utilizada. Como este algoritmo é baseado em matrizes de distância, assumiu-se que estas teriam que se encontrar sempre em memória e que para descobrir a distância mínima teria que se percorrer sempre toda a matriz, aumentando significativamente o tempo de execução. Esta implementação mostrou-se bastante ineficiente para conjuntos de dados na ordem dos milhares e a sua fraca performance fez com que se alterasse a abordagem inicial do problema.

Após novo estudo do algoritmo, a solução encontrada mostrou-se bastante eficiente tanto a nível de memória como de tempo de execução. O principal detalhe desta solução está na definição da estrutura interna de cada elemento da matriz (Nó):

- Cada Nó é diferenciado em Nó-Folha (nó inicial) e Nó-União (nó que contém outros dois nós).

- Um Nó-Folha representa uma sequência.

- Um Nó-União representa a união de dois outros nós e possui a distância entre estes.

- Um Nó possui um ID único que o diferencia dos restantes, um *array* de distâncias e uma referência para a lista de posições, referida posteriormente na Subsecção 3.3.3. Esta lista diminui consoante o ID do Nó uma vez que cada um apenas necessita saber a distância para um outro Nó com ID superior. Isto é, como a matriz é triangular inferior, apenas será necessário guardar no seu *array* de distâncias a distância aos Nós com índice superior e nunca inferior, pois estes últimos já a contêm.

A solução encontra-se apresentada no algoritmo 1.

Algoritmo 1 Algoritmo de clustering - UPGMA.

Dados: Nós/Profiles Nodes, Número de Nodes N
Resultado: Nó raiz Root

1. Seja $k = N$
2. Enquanto $k > 2$
   - 2.1 Pesquisar e obter o par P_1P_2 de nós com distância mínima
   - 2.2 Criar novo nó $Union$
   - 2.3 Associar P_1P_2 a $Union$
   - 2.4 Seja $l = 0$
   - 2.5 Enquanto $l < k$ //Para cada nó em $Nodes$
     - 2.5.1 Recalcular a distância de $Nodes[l]$ a $Union$
     - 2.5.2 Incrementar l
   - 2.6 Adicionar $Union$ em $Nodes$ na posição de P_1
   - 2.7 Remover P_1 e P_2 em $Nodes$
   - 2.8 Decrementar k

Algoritmo Hierárquico - UPGMA	**Custo** (melhor caso)	**Custo** (pior caso)
1 Seja $k = N$	$O(1)$	$O(1)$
2 Enquanto $k > 2$	$O(k)$	$O(k)$
2.1 Pesquisar e obter o par P_1P_2 de nós com distância mínima	$\sum_{k=3}^{N} O(k)$	$\sum_{k=3}^{N} O(k)$
2.2 Criar novo nó $Union$	$\sum_{k=3}^{N} O(1)$	$\sum_{k=3}^{N} O(1)$
2.3 Associar P_1P_2 a $Union$	$\sum_{k=3}^{N} O(1)$	$\sum_{k=3}^{N} O(1)$
2.4 Seja $l = 0$	$\sum_{k=3}^{N} O(1)$	$\sum_{k=3}^{N} O(1)$
2.5 Enquanto $l < k$ //Para cada nó em $Nodes$	$\sum_{k=3}^{N} O(k)$	$\sum_{k=3}^{N} O(k)$
2.5.1 Recalcular a distância de $Nodes[l]$ a $Union$	$\sum_{l=0}^{k-1}(\sum_{k=3}^{N} O(1))$	$\sum_{l=0}^{k-1}(\sum_{k=3}^{N} O(k))$
2.5.2 Incrementar l	$\sum_{l=0}^{k-1}(\sum_{k=3}^{N} O(1))$	$\sum_{l=0}^{k-1}(\sum_{k=3}^{N} O(1))$
2.6 Adicionar $Union$ em $Nodes$ na posição de P_1	$\sum_{k=3}^{N} O(1)$	$\sum_{k=3}^{N} O(1)$
2.7 Remover P_1 e P_2 em $Nodes$	$\sum_{k=3}^{N} O(1)$	$\sum_{k=3}^{N} O(1)$
2.8 Decrementar k	$\sum_{k=3}^{N} O(1)$	$\sum_{k=3}^{N} O(1)$

Tabela 3.1: Custos dos algoritmos hierárquicos.

Com base na Tabela 3.1, obtém-se para o melhor caso a complexidade temporal $O(N^2)$ e para o pior $O(N^3)$.

O melhor caso é obtido quando o passo 2.5.1 da Tabela 3.1 é reduzido para tempo constante. Isto é conseguido pois cada nó sabe o nó par com que têm a distância mínima. Assim, esta procura torna-se mais rápida pois não é necessário percorrer, para todos os nós, as suas respetivas distâncias. Apenas quando é removido um nó, e apenas se esse for o nó que representa a distância mínima, será necessário descobrir a nova distância mínima, percorrendo toda a lista de nós.

Esta solução procura reduzir o número de iterações à medida que se vão criando novos nós e ainda facilitar a procura da distância mínima.

O algoritmo 1 e a Tabela 3.1 também se aplicam aos restantes algoritmos hierárquicos implementados, Single-Linkage e Complete-Linkage.

3.3.2 Algoritmo Neighbor-Joining

Para a implementação deste algoritmo também foi necessário adicionar um novo módulo à plataforma. Esse módulo encontra-se representado na Figura 3.6.

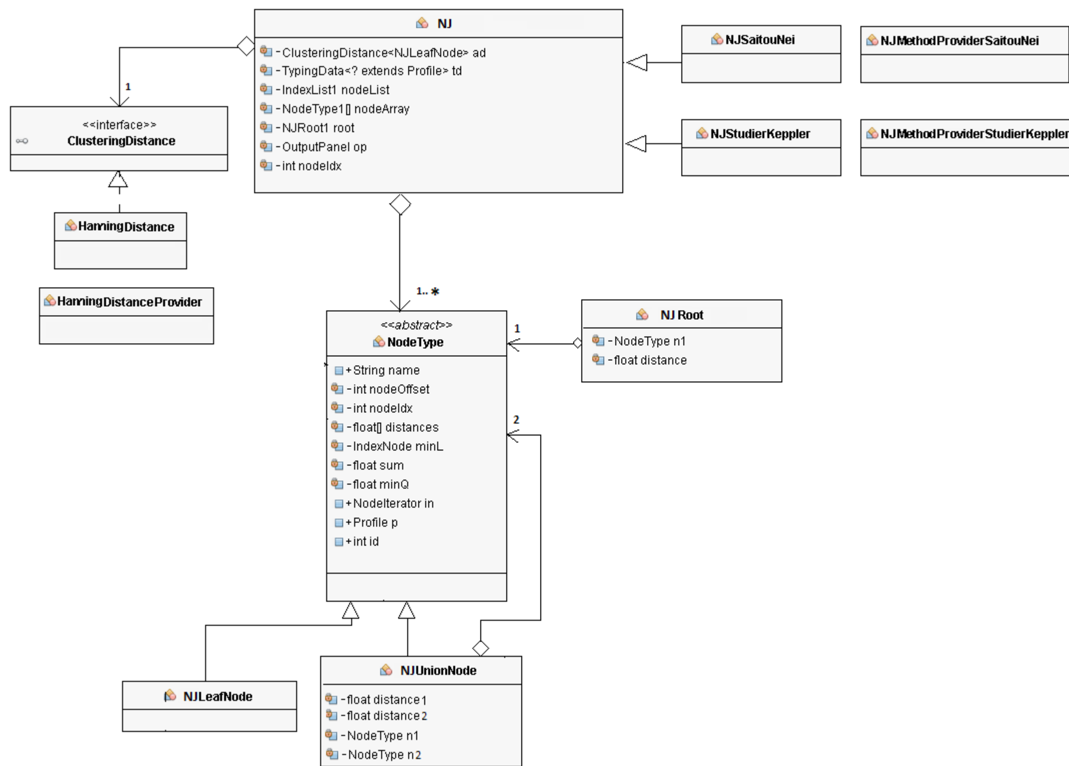


Figura 3.6: Representação UML do módulo `PHYLOViZ Neighbor-Joining`.

Um dos problemas com o algoritmo de Neighbor Joining [12] é o seu custo $O(N^3)$ de algoritmo e $O(N^2)$ de espaço em memória, que pode tornar-se problemático quando N seja

um número de elementos elevado.

Com visto a minimizar estes problemas, foram tomadas várias medidas, para que existisse uma melhoria significativa de performance do algoritmo.

Memória

Para a memória, uma matriz de N x N é desnecessário, visto esta ser espelhada, ou seja, as distâncias dos nós AB e BA são iguais, optando assim por uma matriz em forma triangular, tendo A distância para B, e B não tendo para A. Este procedimento é igual para todos os nós, até que chegue ao último, que não terá quaisquer distâncias.

Quando encontrado um par de nós e colocado numa união, podemos eliminar esse par, ajudando assim o *garbage collector*[13] do Java, libertando a memória a eles associada.

Tempo do algoritmo

Com um custo de $O(N^3)$, o tempo de processamento pode tornar-se demoroso para grandes conjuntos de dados, desta forma optou-se por realizar uma otimização no cálculo das somas. Para que a cada iteração não exista o custo $O(N)$ no cálculo das somas de cada nó, cada nó tem a si associado o valor da sua soma, reduzindo a complexidade temporal da atualização da soma para $O(1)$.

A solução implementada encontra-se no algoritmo 2.

Algoritmo 2 Algoritmo de Neighbor Joining

Dados: Nodes/Profiles Nodes, Número de Nodes N
Resultado: Nó raiz Root

1 Gerar a árvore
- 1.1 Seja $k = N$
- 1.2 Enquanto $k > 2$
  - 1.2.1 Por cada nó em $Nodes$
    - 1.2.1.1 Processa as distâncias deste nó a todos os outros e armazena o menor.
  - 1.2.2 Pesquisar e obter o par P_1P_2 de nós com distância mínima
  - 1.2.3 Cria União com os dados de MenorPar
  - 1.2.4 Seja $l = 0$
  - 1.2.5 Enquanto $l < k$ //Para cada nó em $Nodes$
    - 1.2.5.1 Remove as distâncias e insere distância de União
    - 1.2.5.2 Incrementa l
  - 1.2.5 Remove em $Nodes$ os nós utilizados na União
  - 1.2.6 Coloca União em $Nodes$
  - 1.2.7 Decrementa k

Algoritmo Aglomerativo - Neighbor-Joining	Custo
1 Gerar a árvore	
1.1 Seja $k = N$	$O(1)$
1.2 Enquanto $k > 2$	$O(k)$
1.2.1 Por cada nó n em $Nodes$	$\sum_{k=3}^{N} O(k)$
1.2.1.1 Procura a menor distância deste nó a todos os outros	$\sum_{n=0}^{k-1} (\sum_{k=3}^{N} O(k))$
1.2.2 Pesquisar e obter o par P_1P_2 de nós com distância mínima	$\sum_{k=3}^{N} O(k)$
1.2.3 Cria União com os dados de MenorPar	$\sum_{k=3}^{N} O(1)$
1.2.4 Seja $l = 0$	$\sum_{k=3}^{N} O(1)$
1.2.5 Enquanto $l < k$	$\sum_{k=3}^{N} O(k)$
1.2.5.1 Remove as distâncias e insere distância de União	$\sum_{l=0}^{k-1} (\sum_{k=3}^{N} O(1))$
1.2.5.2 Incrementa l	$\sum_{l=0}^{k-1} (\sum_{k=3}^{N} O(1))$
1.2.5 Remove em $Nodes$ os nós utilizados na União	$\sum_{k=3}^{N} O(1)$
1.2.6 Coloca União em $Nodes$	$\sum_{k=3}^{N} O(1)$
1.2.7 Decrementa k	$\sum_{k=3}^{N} O(1)$

Tabela 3.2: Custo do algoritmo aglomerativo Neighbor-Joining.

O custo do algoritmo de Neighbor-Joining, ao contrário do algoritmo de UPGMA, não tem melhores ou piores casos[3], ficando sempre com o custo de O(N^3).

3.3.3 Estruturas de Dados

As dificuldades principais destas implementações são a escalabilidade em termos de memória e conseguir manter sempre atualizadas as posições e as distâncias dos nós existentes, uma vez que estas se encontram sempre a mudar ao longo do tempo.

Indexação

Para solucionar o problema da atualização das posições dos nós, foi criada uma estrutura de indexação (*IndexList*). Esta funciona como uma lista, tendo cada elemento (*IndexNode*) um inteiro que representa uma posição válida no *array* de nós de distâncias (*NodeArray*) e quais os elementos anteriores e posteriores. Quando criado um nó de distâncias é-lhe então atribuído um *IndexNode*, que permitirá saber quais os nós de distâncias que este tem acesso,

[3]Optou-se por continuar a utilizar a notação O, embora neste caso possa ser utilizada a notação θ

como exemplificado na Figura 3.7.

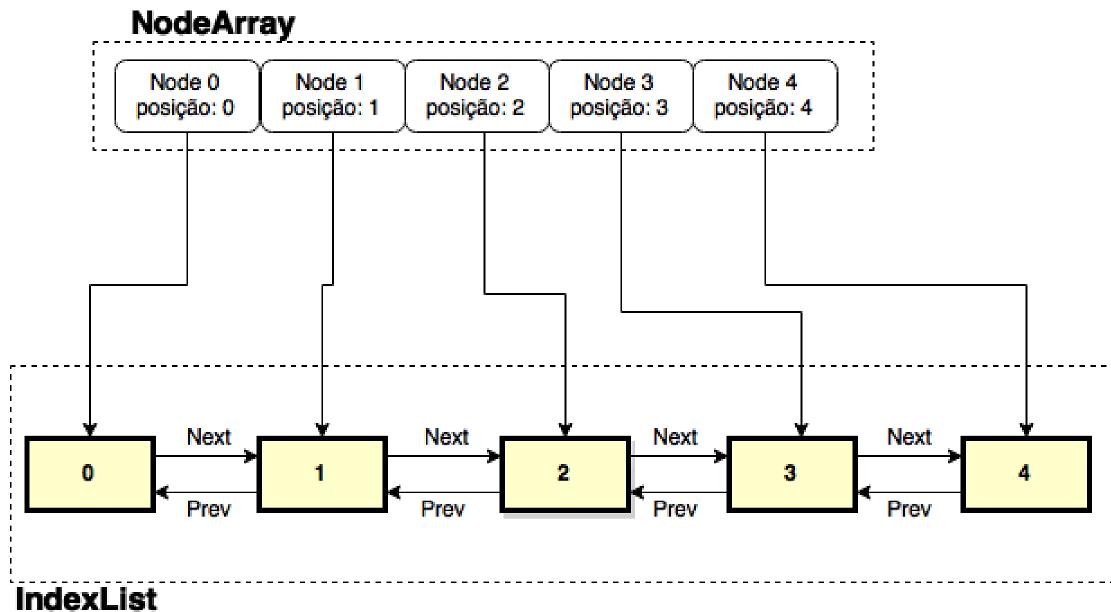


Figura 3.7: Representação da ligação entre nós de distância e a lista de indexação.

Como apresentado na Figura 3.7 o nó 0 (*Node 0*) irá então conter as distâncias para os nós com índice 0, 1, 2, 3 e 4, o nó 1 (*Node 1*) terá acesso às distâncias dos nós 1, 2, 3 e 4 e assim sucessivamente para os restantes nós, até ao *Node 4* que não terá distâncias para qualquer nó.

Esta estrutura tem ainda outra grande utilidade, a remoção de um determinado *IndexNode*. Quando se trata de uma remoção, apenas é necessário remover um determinado *IndexNode* da *IndexList*, tendo um custo de O(1), pois eliminado um nó, as ligações serão alteradas, como demostrado na Figura 3.8.

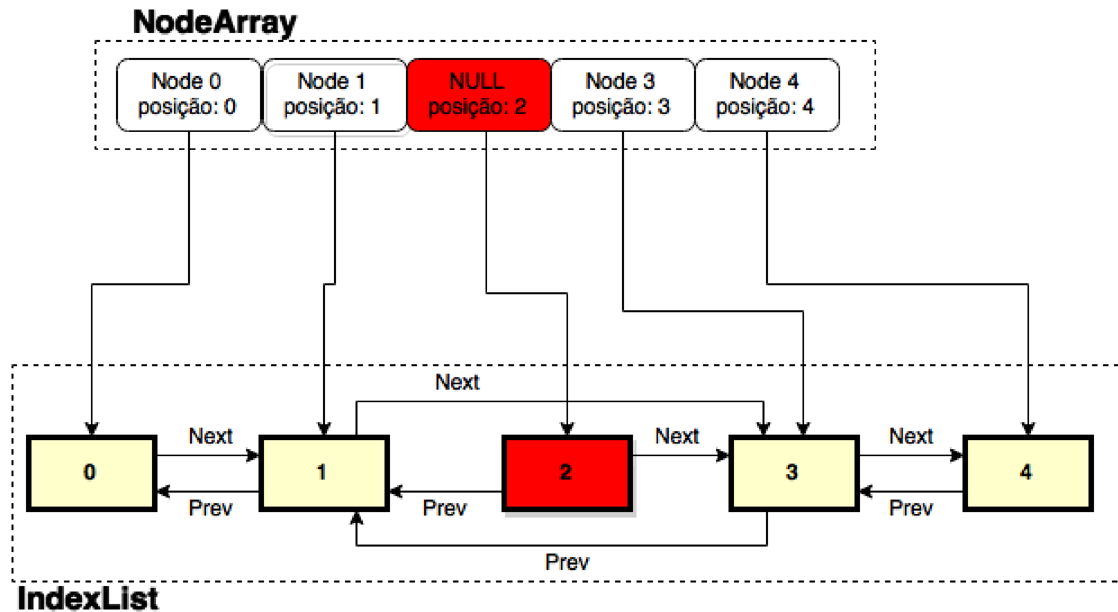


Figura 3.8: Representação da ligação entre nós de distância e a lista de indexação após remoção.

Como se pode observar, após a remoção do *IndexNode 2* o *Node 0* passou assim a ter acesso aos índices 0, 1, 3 e 4. O mesmo acontece para os restantes nós.

Memória

Outro dos problemas encontrados foi a escalabilidade em termos de memória. Isto pode acontecer quando milhares de nós de distância são criados inicialmente, provocando um aumento da memória utilizada. Ao longo da computação dos algoritmos existe a necessidade de criar novos nós para a união, aumentando assim o número de objetos em memória.

A solução encontrada foi a eliminação dos nós por cada iteração, isto é apagar toda a informação desnecessária presente nos nós a remover, principalmente os *arrays* de distância presentes em cada *Node*, que nunca mais serão utilizados.

Esta solução não resolve todos os casos, pois existe a possibilidade de escalabilidade em termos de memória durante a criação da matriz de distâncias inicial.

Capítulo 4

Projeto PHYLOViZ e Serialização

Por vezes pode existir a necessidade de consultar dados do mesmo ficheiro múltiplas vezes, o que pode ser demorado para o utilizador quando esse ficheiro tem um grande volume de dados (*i.e.* milhares de dados). Também toda a análise já realizada poderá ser novamente útil, assim como a visualização associada a cada estudo, uma vez que esta pode ter sofrido alterações, como a utilização da funcionalidade de corte. Por este motivo foi criada uma solução que permita ao utilizador guardar e abrir um projeto à sua escolha, podendo assim repor todo o estudo evitando novos processamentos, quer em relação à computação dos algoritmos quer à visualização.

4.1 Estado do Projeto

A necessidade de guardar o estado atual de processamento de um dataset torna-se cada vez maior à medida que se vão adicionando novas funcionalidades/ferramentas de análise à plataforma (*e.g.* novos métodos de *clustering*, métodos de tipagem, etc.). Ou seja, uma vez calculados e para evitar a repetição do estudo, permitir que o resultado gerado possa ser novamente visualizado.

Isto fez com que fosse adicionado um novo módulo à plataforma PHYLOViZ responsável por guardar o estado atual e poder repo-lo sempre que o utilizador o desejar. Assim, um projeto PHYLOViZ é essencialmente constituído por um DataSet - Figura 4.1.

Para guardar um projeto apenas será necessário efetuar uma cópia do *Typing Data* e, caso exista, do *Isolate Data*, utilizados e criar um ficheiro de configuração com a especificação necessária à recriação do mesmo. Caso já tenha ocorrido a análise desse *Typing Data*, também é possível guardar esse processamento, nomeadamente o *output*, em ficheiros *JSON*.

Todos os elementos que necessitem de ser gravados (associados a um projeto) devem implementar a interface pública `ProjectItem` e `ProjectFactory` - Figura 4.2.

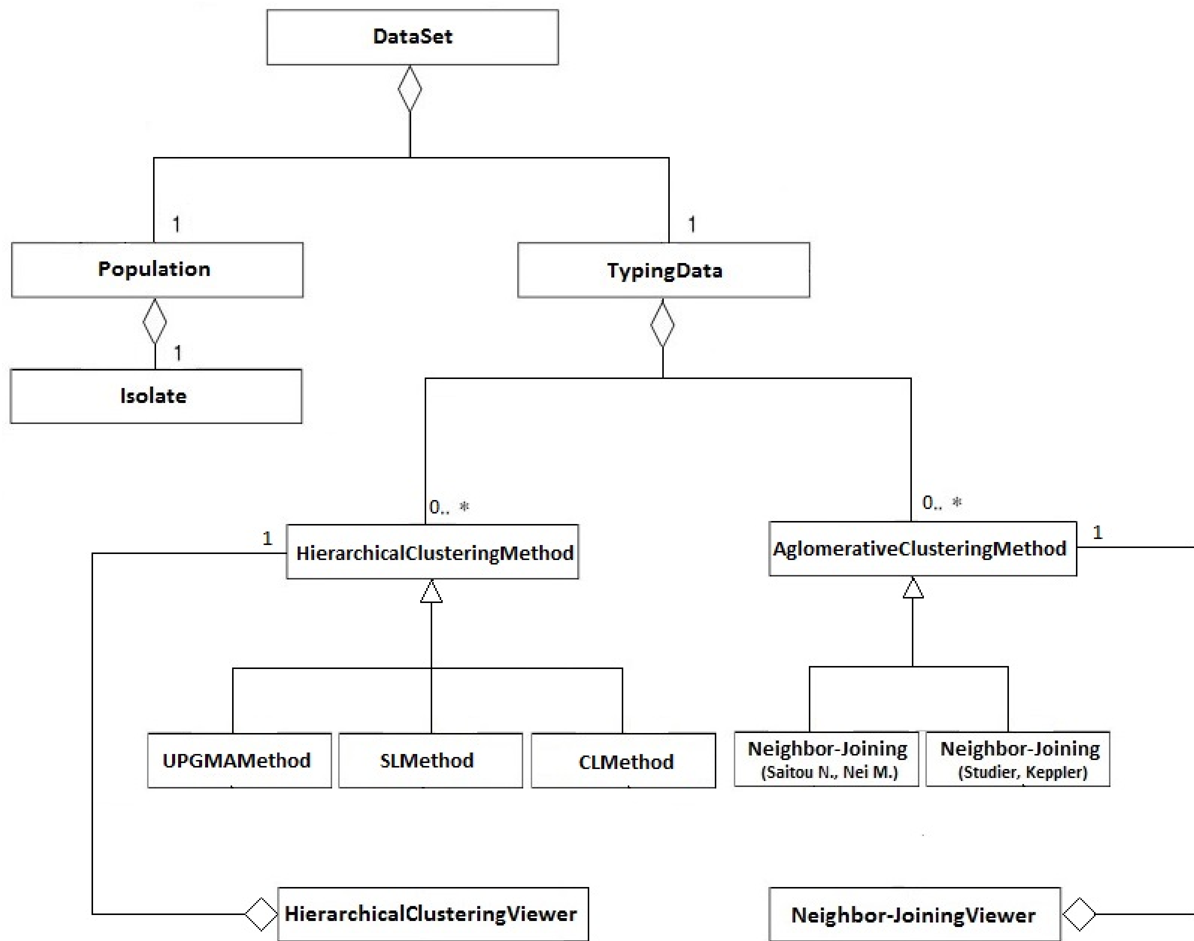


Figura 4.1: Representação de um DataSet.

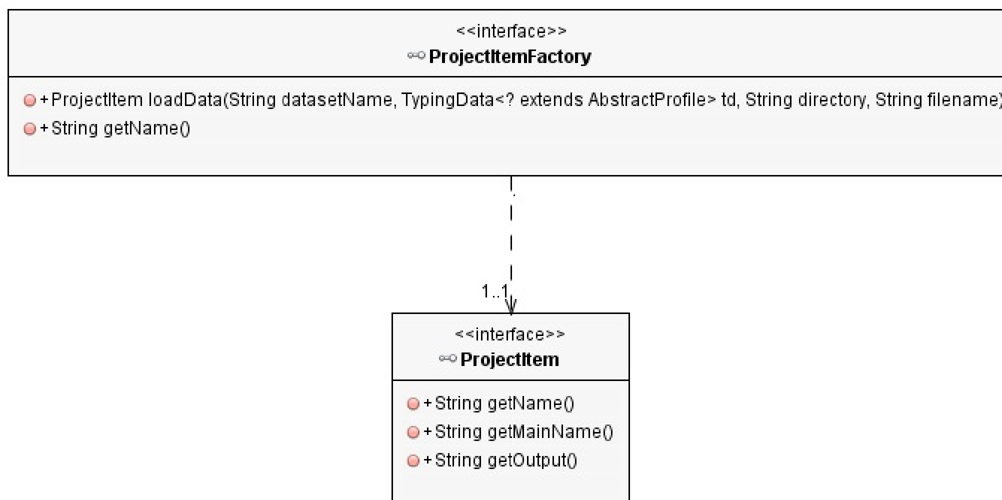


Figura 4.2: Interfaces `ProjectItem` e `ProjectFactory`.

A interface `ProjectItemFactory` é responsável por saber criar um `ProjectItem` através

da leitura de um ficheiro.

Um `ProjectItem` é responsável por saber gerar o *output* necessário a ser guardado. Isto é, cada implementação concreta de `ProjectItem` terá que saber gerar o que pretende que seja guardado (*e.g.*: Todos os algoritmos implementados sabem gerar o seu *output* através de objetos no formato JSON e, consequentemente, terão também que os saber interpretar aquando do carregamento de um projeto).

Para abrir um projeto apenas será necessário ler o ficheiro de configuração e, por reflexão [14], criar as entidades necessárias à reposição total do estado anterior.

Seguidamente, é apresentado um exemplo de um ficheiro de configuração.

```
1  dataset-name="S. pneumoniae"
2  typing-factory=net.phyloviz.mlst.MLSTypingFactory
3  typing-file=spneumoniae.typing.csv
4  population-factory=net.phyloviz.core.util.PopulationFactory
5  population-file=spneumoniae.isolate.csv
6  population-foreign-key=3
7  algorithm-output=spneumoniae.output.upgma.json,spneumoniae.output.nj.json
8  algorithm-output-factory=net.phyloviz.upgma.UPGMAItemFactory,net.phyloviz
       .nj.NJItemFactory
9  algorithm-output-distance=hamming,hamming
10 visualization=upgma.pviz,nj.pviz
```

A chave `population-foreign-key` permite correlacionar o *TypingData* com os dados complementares (*IsolateData*). Isto é, relaciona os perfis alélicos criados por um método de tipagem com os dados complementares (informação demográfica, informação epidemiológica, resistência a antibióticos, etc.) através de um identificador da sequência.

A funcionalidade de guardar a visualização está associada à chave `visualization` que tem como valor o nome do ficheiro responsável pela serialização específica de cada algoritmo.

4.2 Serialização

A serialização do *output* dos algoritmos permite à aplicação o armazenamento de uma árvore gerada para que futuramente possa ser lida, tendo apenas o custo da leitura, em vez do custo repetido de uma nova geração da mesma árvore.

A escrita é feita em formato JSON[4], de forma semelhante para ambos os algoritmos, mas cada um com as suas especificações.

Cada algoritmo tem a sua estrutura, tendo sido assim necessária a criação de dois esquemas que serão utilizados para a validação durante a leitura, isto é, carregamento de um projeto. Este procedimento apenas é necessário pois o ficheiro pode ter sido alterado, deixando-o com falhas.

[4]JSON - JavaScript Object Notification

Um esquema contém a estrutura de como deverá ficar o armazenamento, quais os nomes dos campos, a sua obrigatoriedade e valores mínimos.

```json
{
  "required": ["leaf", "union", "root"],
  "type":"object",
  "properties":{
    "leaf":{
      "type":"array",
      "items":{
        "properties":{
          "id":{"type":"integer", "minimum": 0},
          "profile":{"type":"integer", "minimum": 0}
        },
        "required":[ "id", "profile"],
        "additionalProperties": false
      },
      "uniqueItems": true
    },
    "union":{
      "type":"array",
       "items":{
        "properties":{
          "id":{"type":"integer", "minimum": 0},
          "left":{"type":"integer", "minimum": 0},
          "distanceLeft":{"type":"number"},
          "right":{"type":"integer", "minimum": 0},
          "distanceRight":{"type":"number"}
        },
        "required":["id", "left", "distanceLeft", "right", "distanceRight"],
        "additionalProperties": false
      },
      "uniqueItems": true
    },
    "root":{
      "type":"object",
      "items":{
        "properties":{
          "distance":{"type":"number"},
          "left":{"type":"integer", "minimum": 0},
          "right":{"type":"integer", "minimum": 0}
        },
        "required":["distance", "left", "right"],
        "additionalProperties": false
      },
      "uniqueItems": true
    }
  }
}
```

Esquema 4.1: Esquema JSON para validação de ficheiro para carregamento do algoritmo Neighbor-Joining.

O exemplo do esquema A pertence ao esquema do Neighbor-Joining, composto por dois

elementos, os nós folha e os nós de união. O esquema dos algoritmos hierárquicos encontra-se no Anexo A.

Um nó folha é construído através de um identificador, que será utilizado para a construção de uma união e o identificador para uma sequência. Ambos estes campos são obrigatórios e com um valor mínimo 0.

Uma união pode ser constituída por nós folhas ou outros nós de união, o que requer ordem na escrita/leitura, ou seja, caso uma união utilize outra união, esta segunda necessita que a sua criação esteja anteriormente feita.

A união requer assim de um identificador, uma referência para um elemento à esquerda e sua distância, e o mesmo para a sua direita.

O elemento raiz (*root*) é composto por duas referências, tal como a união, com a diferença que este apenas necessita uma distância entre ambos os nós.

É ainda especificado que todos os elementos presentes neste esquema são únicos.

Capítulo 5

Visualização

Associado a cada método de *clustering*, hierárquico e aglomerativo, existem diferentes modos de visualização. Nomeadamente, os hierárquicos são representados no formato de um dendrograma e os algomerativos em árvore.
Para a sua concretização foi necessário adicionar dois novos módulos, `UPGMA Viewer` e `NJ Viewer`. O primeiro irá possibilitar a visualização dos algoritmos hierárquicos (UPGMA, Single-Linkage e Complete-Linkage) e o segundo o algoritmo aglomerativo (Neighbor-Joining).
A representação é composta por nós folha e os ramos associados, cada um com o seu significado. A complexidade de geração é diferente pois cada um contém a sua própria representação.

5.1 UPGMA

A implementação da visualização dos métodos hierárquicos foi baseada na classe `TreeView` presente na biblioteca `prefuse`. Esta disponibiliza diferentes implementações para geração de diversos tipos de visualização dinâmica de dados. A sua *framework* disponibiliza ações de filtragem, renderização, etc. - Figura 5.1.

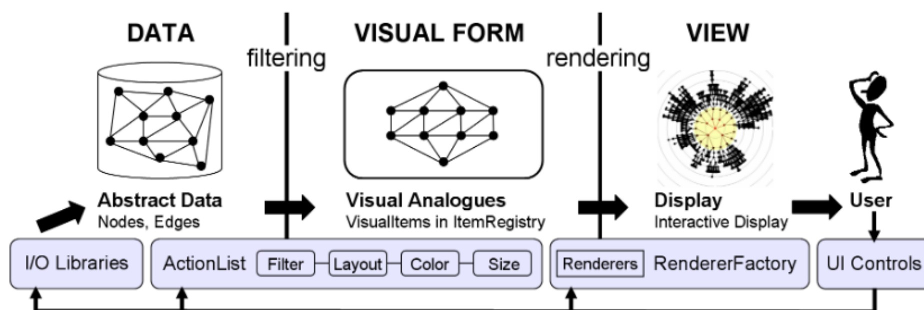


Figura 5.1: Framework `prefuse`

A Figura 5.1 representa a *framework* do `prefuse` e os seus componentes, tais como:

- *Abstract Data* - O processo de visualização é iniciado com dados abstratos não estru-

turados e estes são representados através de estruturas de dados, como grafos, árvores, etc. Existe um elemento de dados base que suporta atributos do tipo chave-valor e que serve de base aos elementos estruturados, como *Node*, *TreeNode* e *Edge*.

- *Filtering* - Filtragem é o processo de mapear os dados abstratos em representações passíveis de serem visualizadas. Primeiramente, um conjuntos de dados abstratos são selecionados para visualização. Seguidamente, são gerados *VisualItems* que permitem guardar atributos visuais como a sua localização, cor, tamanho, etc. São também fornecidos filtros individuais que são representados como *Actions*, definidos mais à frente. A aplicação destes filtros permite que cada *VisualItem* tenha os seus próprios controladores de visualização.

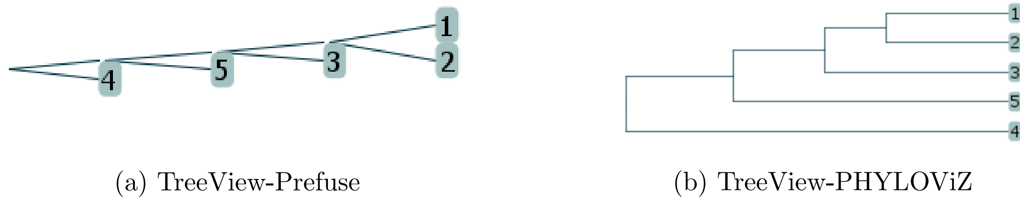
- Gestão de *Visual Items*: *ItemRegistry* - Existem três tipos de *VisualItem*: os *NodeItems* que permitem visualizar entidades individuais; os *EdgeItems* que correspondem a relações entre entidades (*NodeItems*); e *AggregateItems* que são representados por grupos de de entidades. O *ItemRegistry* é uma estrutura de dados centralizada onde são criados e guardados os *VisualItems*.

- *Actions* - Permitem atualização do estado dos *VisualItems* presentes no *ItemRegistry*. Possuem um mecanismo de seleção visual de dados e de atribuição de um conjuntos de propriedades, tais como filtragem, cor, *layout*, etc. Por exemplo, a ação de filtragem permite controlar quais as entidades que serão representadas por *VisualItems*.

- *Actions Lists* - São conjuntos de ações que permitem que estas sejam executadas sequencialmente. Estas listas podem também ser configuradas para serem executadas uma vez, ou periodicamente.

- *Rendering* e *Display* - Os *VisualItems* são desenhados no ecrã através de *Renderers*. Estes componentes utilizam os atributos visuais de um *VisualItem* (*e.g.* cor, localização) para determinar exatamente o aspeto desse elemento no ecrã. Existem disponíveis vários tipos de *Renderers*, utilizados para desenhar formas simples, tais como linhas (direitas ou curvas), texto, imagens, etc. O mapeamento entre cada elemento e a sua respetiva visualização é gerido por um *RendererFactory*: dado um *VisualItem*, o *RendererFactory* irá retornar o *renderer* mais apropriado. Isto permite alterar facilmente a visualização associada a cada *VisualItem*. A apresentação destes é realizada por um outro componente, *Display*, que, dada uma lista de *VisualItems* presentes no *ItemRegistry*, aplica as transformações necessárias para que estes sejam apresentados pelos *Renderers* apropriados. Suporta ainda interação com os elementos visíveis através do registo de listeners nos eventos de teclado e rato sobre os elementos.

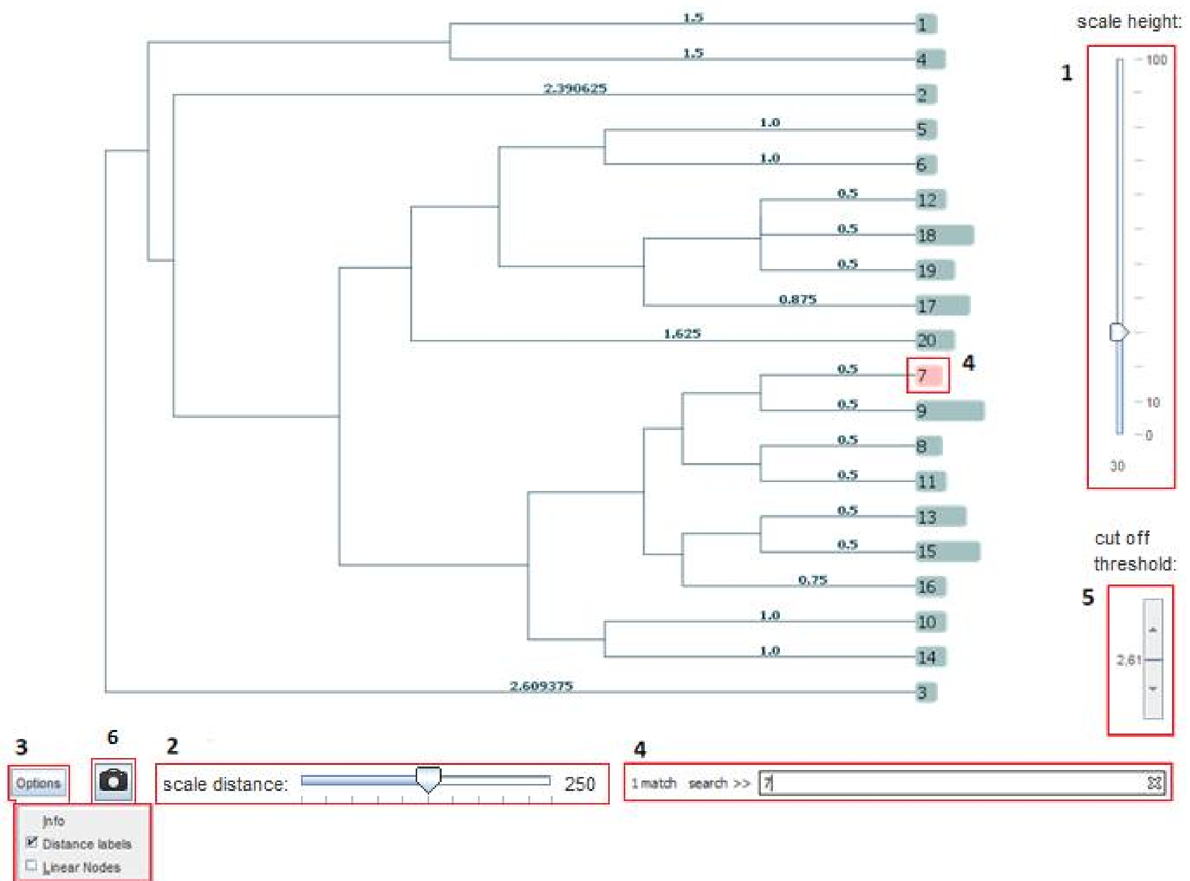
Assim, tendo como base a classe `TreeView` do `prefuse`, Figura 5.2a, foi então possível modificá-la (*e.g.* criação de novos *Renderers*), para corresponder ao pretendido, Figura 5.2b.



(a) TreeView-Prefuse (b) TreeView-PHYLOViZ

Figura 5.2: Visualizações da classe TreeView.

5.1.1 Funcionalidades



Legenda:

1. Escalar em altura
2. Escalar em largura
3. Painel de opções
4. Procura um *Táxon* e destaca-o
5. Filtra por distância
6. Exportar imagem

Figura 5.3: Visualização de UPGMA para ficheiro com 20 Táxon.

A Figura 5.3 representa a visualização de um DataSet com 20 Táxon. A distância encontra-se escalada em 250 unidades para que seja possível visualizar mais facilmente as diferenças existentes entre as distâncias de cada Táxon.
O painel de opções permite adicionar/remover o valor das distâncias e permite ainda visualizar cada perfil de acordo com a sua frequência entre todos os isolados. Caso este se encontre selecionado, então a visualização irá representar exatamente o valor dessa frequência, caso contrário será utilizada escala logarítmica.
A caixa de pesquisa foi adicionada com vista a encontrar um certo Táxon em DataSets de grande dimensão. Caso esse Táxon não se encontre presente no ecrã, este atualiza-se focando o Táxon desejado.
A funcionalidade de corte (separação de grupos de Táxon com base na distância entre estes) permite destacar apenas os grupos com distâncias iguais ou inferiores à definida.
A possibilidade de exportar permite que seja possível guardar a imagem que está a ser mostrada nesse momento em diversos formatos como *PNG*, *PDF*, *SVG*, *EPS*, etc.
Ao selecionar um certo Táxon, caso este não integre os dados complementares, é apresentada a informação que este representa, ou seja, o *Profile*. Caso contrário, é apresentada toda a informação complementar deste isolado.

5.2 Neighbor-Joining

A visualização do algoritmo Neighbor-Joining foi baseada na do algoritmo goeBURST que também utiliza a biblioteca `prefuse`. A sua visualização ficará representada em árvore, tendo cada folha a representação de um *Profile* e cada ramo da árvore representa a distância entre os dois nós.
As principais diferenças entre estas duas visualizações, Neighbor-Joining e goeBURST, são o desenho dos nós de união, que no caso do primeiro não são desenhados e os ramos, que não têm tamanho fixo, ao contrário do goeBURST. Isto é, no Neighbor-Joining representa um ramo de acordo com a distância entre dois nós, e o goeBURST apenas representa uma ligação entre dois nós.

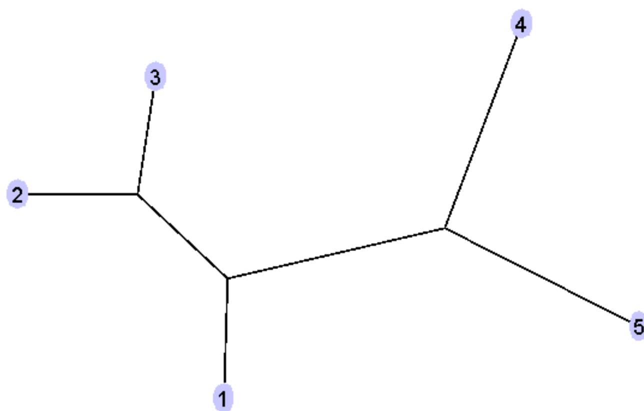


Figura 5.4: Visualização de Neighbor-Joining para ficheiro com 5 entradas.

Na Figura 5.4 podemos então visualizar a representação da árvore gerada contendo cinco folhas e sete ramos.

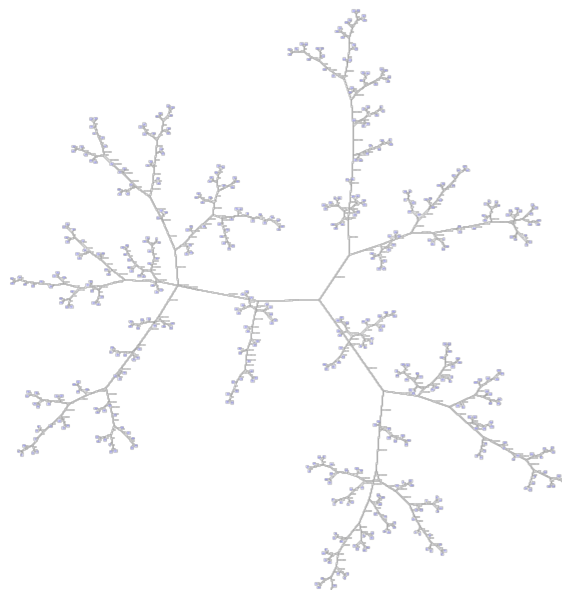
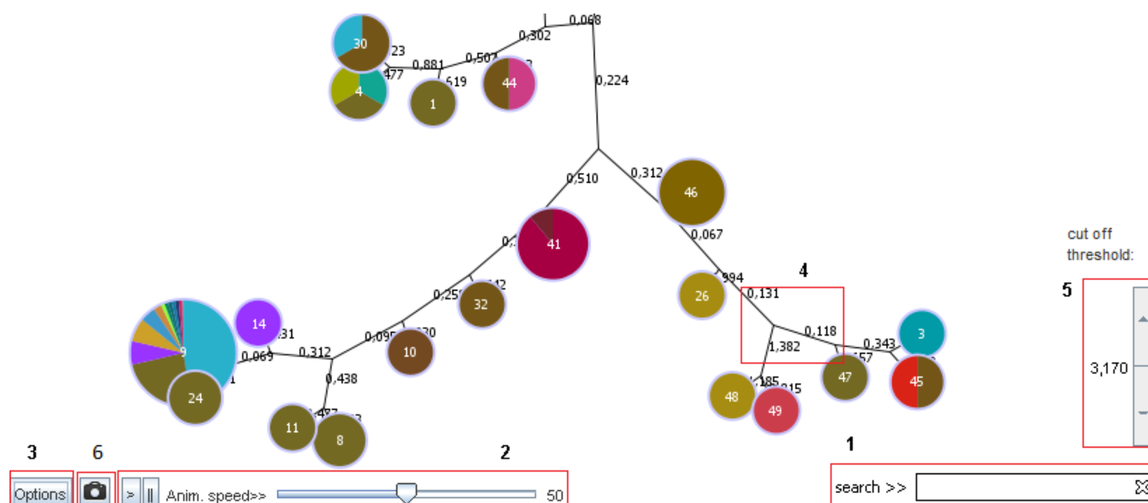


Figura 5.5: Visualização de Neighbor-Joining para ficheiro com 500 entradas.

Como se pode verificar na Figura 5.5, quando se começa a tratar de ficheiros cada vez maiores, maior será a árvore gerada, que faz com que se possa tornar ilegível para o utilizador.
Por este motivo foram criadas várias funcionalidades com visto a facilitar a sua interação, tais como, a pesquisa por determinada folha, opções de visualização, escala, etc.
Na Figura 5.6 é possível ver qual a visualização que será gerada e respetivas funcionalidades.

5.2.1 Funcionalidades



Legenda:

1. Procura um *Táxon* e destaca-o
2. Velocidade de movimento da árvore
3. Painel de Opções
4. Representação de distâncias
5. Filtra por distância
6. Exporta imagem

Figura 5.6: Visualização parcial de Neighbor-Joining para ficheiro com 500 entradas.

Para facilitar a procura de uma determinada folha, foi criada a funcionalidade de procura, onde o utilizador terá uma área de pesquisa e poderá inserir o identificador a pesquisar, presente na Figura 5.6 - elemento 1. As folhas encontradas pela pesquisa irão então ficar destacadas com uma cor diferente.

A geração da árvore pode ser demorada. Por este motivo foi criada uma funcionalidade de velocidade (Figura 5.6 - elemento 2) onde o utilizador poderá aumentar ou diminuir a velocidade de processamento da imagem.

Como se pode ver, na Figura 5.6 - elemento 3, existe um botão de opções onde irão aparecer duas escolhas, *Info*, *Distance Labels*, *Linear Nodes*, *Round Distances*, *High Quality* e *Control*. *Info* irá abrir um painel onde terá a informação em texto da geração da árvore, nomeadamente, informação sobre os *Profiles*. *Distance Labels* fará com que em todos os ramos apareça a informação com a distância desse mesmo ramo, como se pode ver na Figura 5.6 - elemento 4.

A funcionalidade *Linear Nodes* permitirá destacar os nós que representem *Profiles* com maior frequência. *Distance Labels* e *Round Distânces* permitirão ao utilizador controlar qual o número de casas decimais que irão aparecer nas distâncias de cada ramo. A opção *High*

Quality fará com que a qualidade de *display* apresentada seja melhor, isto é uma melhor qualidade de imagem. *Control* apresentará um painel ao utilizador para que este possa então controlar diversas ações sobre a árvore, força gravítica utilizada nos nós, força de arrasto, etc.
A funcionalidade de corte (separação de grupos de Táxon com base na distância entre estes), elemento 5, permite destacar apenas os grupos com distâncias iguais ou inferiores à definida, libertando os grupos com distância superior à definida.
O elemento 6 representa a possibilidade de exportar a imagem visível no momento em diversos formatos como *PNG*, *PDF*, *SVG*, *EPS*, entre outros.

5.3 Comparação com softwares já existentes

Atualmente, as outras ferramentas que permitem a análise e visualização de perfis alélicos falham na integração dos dados epidemiológicos, o que é crucial para a correta inferência filogenética e análise da relações entre estirpes. Outras tornam possível essa inferência mas apenas através de árvores bem definidas e não através dos dados resultantes da aplicação dos métodos de tipagem.
Duas das ferramentas mais usadas que tentam solucionar alguns destes problemas são o SplitsTree [15] e Mega[16]. As grandes diferenças encontram-se demonstradas pela Tabela 5.1.

Funcionalidades	**PHYLOViZ**	**SplitsTree**	**Mega**
Corte	Sim	Não	Não
Visualização da Distribuição Estatística	Sim	Não	Não
Integração de Dados Complementares	Sim	Não	Não
Escalabilidade	Sim	Sim	Sim
Armazenamento de Dados	Sim	Não	Não

Tabela 5.1: Comparação entre as ferramentas PHYLOViZ e SplitsTree.

Com base na Tabela 5.1 podem-se verificar as vantagens das visualizações utilizadas pelo PHYLOViZ em relação a ambas as outras aplicações. Estas diferenças são fundamentais quando se pretende fazer uma correta análise e visualização de perfis alélicos.
Abaixo, na Figura 5.7 é possível observar diferenças significativas quando se integram os dados complementares.

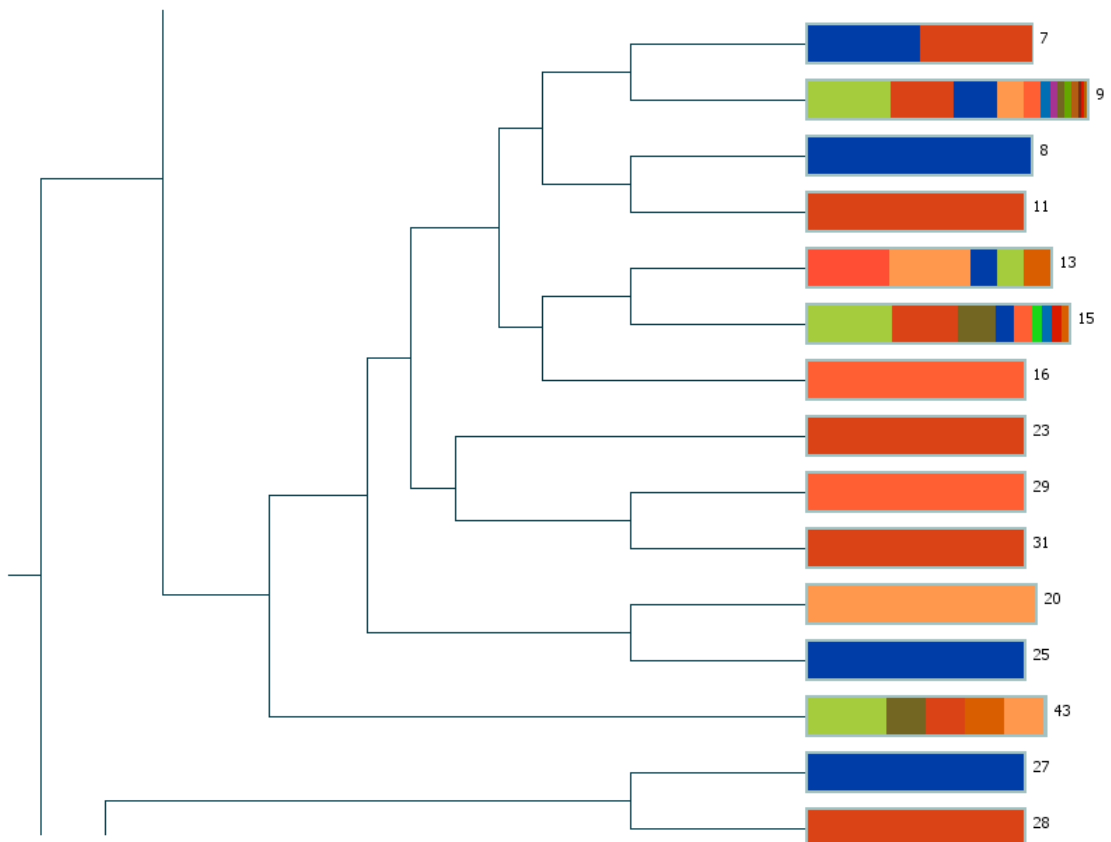


Figura 5.7: Visualização parcial de UPGMA com integração de dados, nomeadamente por país de ocorrência.

Outra caraterística relevante que foi adicionada ao **PHYLOViZ** foi o armazenamento de dados que possibilita a reposição de todo o projeto (dados e visualizações).

Capítulo 6

Avaliação Experimental

Para testar a eficiência das diferentes implementações dos algoritmos, foram realizados testes que visam demonstrar os tempos de execução de cada um. Uma vez que os algoritmos hierárquicos apenas diferem na Fórmula 2.2 do Capítulo 2 na Secção 2.2.1, o seu custo é semelhante e por isso apenas foram realizados testes para o algoritmo UPGMA. Em relação ao algoritmo aglomerativo, o critério de otimização utilizado foi o de *Saitou N. and Nei M.*. Para isto, recorreu-se à base de dados pública PubMLST [17], nomeadamente aos dados de MLST para a bactéria *Streptococcus pneumoniae*.
Todos os testes foram executados numa máquina com as seguintes características:

- Sistema Operativo: Windows 7
- CPU: Intel Core i3-2310M CPU @ 2.10GHz
- Memória: 4.00 GB

6.1 Testes Unitários

A Tabela 6.1 apresenta a média dos tempos de execução do algoritmo UPGMA nas versões implementadas.

	Tempo de execução (milissegundos) / K	
N - nº de elementos	Versão 1	Versão 2
50	31	14
100	65	20
500	1065	205
1000	7781	136
5000	527470	3155

Tabela 6.1: Média de tempos obtidos durante a execução dos testes do UPGMA, versão 1 e 2, sendo K o número de ciclos (50).

Como é possível verificar, existe um aumento super-linear do tempo com o aumento do volume de dados. Este facto é também observável na Figura 6.1.

A grande diferença apresentada entre as duas versões deve-se ao melhoramento das estruturas de dados utilizadas e ao aumento de eficiência na procura da distância mínima, eliminando a necessidade de percorrer toda a matriz de distâncias, diminuindo a complexidade temporal de $O(N^3)$ para $O(N^2)$. Esta diferença encontra-se explicada em detalhe no Capítulo 3 - Secção 3.3.3.

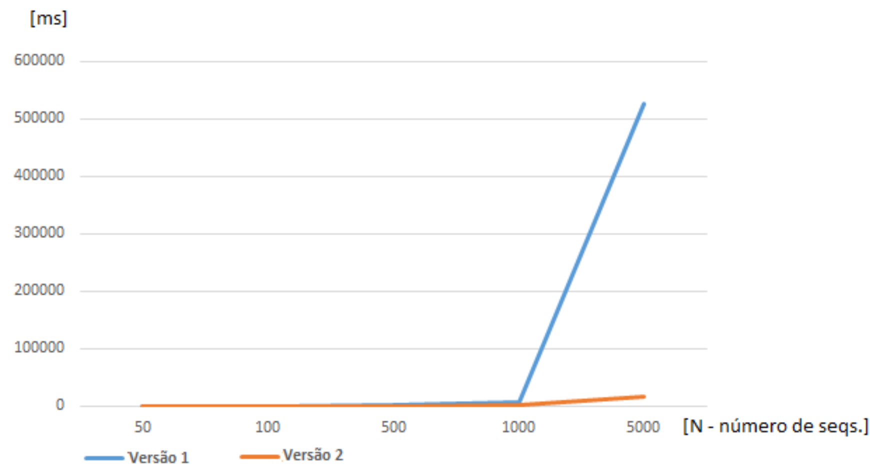


Figura 6.1: Representação gráfica da tabela 6.1

A Tabela 6.2 apresenta a média dos tempos de execução do algoritmo Neighbor-Joining nas versões implementadas.

	Tempo execução (milisegundos) / K	
N - º de elementos	Versão 1	Versão Final
50	44	32
100	756	34
500	984	359
1000	4020	1124
5000	265841	187079

Tabela 6.2: Tempos obtidos durante a execução os testes de Neighbor-Joining, versão 1 e final.

O mesmo acontece com os testes de Neighbor-Joining. Na Figura 6.2 estão representados os tempos para as duas implementações. Estes resultados poderão variar bastante conforme a máquina e o respetivo número de processadores.

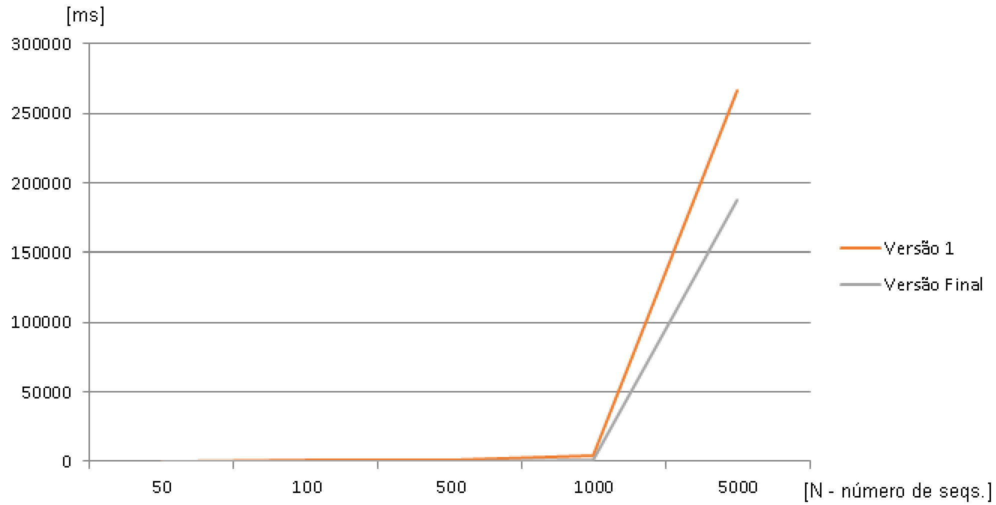


Figura 6.2: Representação gráfica da tabela 6.2

Em ambos os casos existe uma complexidade de N^3, mas é possível ver no gráfico 6.2 a melhoria do algoritmo.
A versão 1 é claramente a mais lenta, pois continha uma matriz de $N \times N$ elementos e a pesquisa era realizada por todos eles. Existia ainda a possibilidade de nessa pesquisa, os nós já terem sido apagados anteriormente, sendo sempre preciso verificar o estado dos mesmos para os cálculos necessários.
A melhoria do mesmo não foi maior devido ao elevado custo do algoritmo, que se torna bastante visível quando se analisa um maior volume de dados. Por outro lado a versão final realiza pesquisas por uma matriz triangular inferior, tendo apenas de fazer a pesquisa por metade dos elementos. Outro fator que ajudou na melhoria de tempos é a estrutura de indexação, onde através desta não é necessários verificar se um dado nó já tinha sido eliminado anteriormente, pois esta estrutura apenas contém índices válidos. Neste caso, não se conseguiu maior escalabilidade devido a complexidade computacional do algoritmo.

Capítulo 7

Conclusão

Este projeto permitiu perceber a importância dos estudos epidemiológicos e genético de populações microbianas, pois fornecem informação importante no controlo de doenças infecciosas e nos estudos sobre a patogénese, nomeadamente na evolução de infeções. Esta importância torna necessária a existência de ferramentas, como o PHYLOViZ, que permitam a análise, manipulação e visualização de diversos conjuntos de dados baseados em diferentes métodos de tipagem.

A adição de novas funcionalidades ao PHYLOViZ tornou esta ferramenta muito mais rica e útil, uma vez que:

- Poderão ser aplicados diferentes algoritmos de *clustering* para a realização de inferências filogenéticas.
- Foram criadas novas visualizações específicas para cada algoritmo.
- Foram integrados os dados complementares às visualizações.
- Poderão ser criados projetos que irão permitir guardar todo o estudo efetuado, nomeadamente os resultados dos algoritmos e as respetivas visualizações.

A implementação destas funcionalidades permite ainda, que no futuro, possa ser realizado *Bootstrapping.* Ou seja, como os algoritmos tentam obter uma árvore filogenética ótima, de acordo com determinados critérios de evolução, é necessário analisar os dados ao qual os algoritmos foram aplicados para perceber se representam árvores válidas ou não. Por exemplo, o Neighbor-Joining é um algoritmo ganancioso que tenta otimizar a árvore de acordo com critério de evolução mínima (Balanced Minimum Evolution - BME). Para cada topologia, o BME define que o comprimento da árvore deverá ser uma soma ponderada de distâncias presentes na respetiva matriz de distâncias, com as ponderações a depender da topologia. A topologia ótima de acordo com o BME é a que minimiza o comprimento da árvore.

Esta técnica de *Bootstrap* permite realizar uma amostragem aos dados, normalmente 1000 a 10000 vezes, executando cada algoritmo para cada uma dessas amostras. Com isto, obtém-se

a frequência de cada um dos arcos nas amostragens, tornando possível a sua visualização na respetiva árvore.

Todo o desenvolvimento do projeto e as soluções adotadas tiveram sempre em conta a possibilidade de adicionar novas funcionalidades a partir destas. Isto é, conseguiu-se tornar as classes implementadas o mais genéricas e abstratas possível de forma, a que no futuro, possam ser facilmente adicionados, por exemplo, outros métodos de *clustering* com diferentes algoritmos de cálculo de distância.

Referências

[1] M.C. Maiden et al. Multilocus sequence typing: a portable approach to the identification of clones within populations of pathogenic microorganisms. *Proceedings of the National Academy of Sciences of the United States of America*, (95(6)):pp.3140–3145, 1998. Consultado em 2015-04-27.

[2] Nature Education. A collaborative learning space for science - snp. http://www.nature.com/scitable/definition/single-nucleotide-polymorphism-snp-295 , Consultado em 2015-04-30.

[3] Alexandre Francisco, Cátia Vaz, Pedro Monteiro, José Melo-Cristino, Mário Ramirez, and João André Carriço. PHYLOViZ. 2014. Consultado em 2015-04-27.

[4] Robert R. Sokal and Charles D. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, (38):pp.1409–1438, 1958. Consultado em 2015-04-26.

[5] Naruya Saitou and Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Molecular Biology and Evolution*, (4(4)):pp.406–425, 1987. Consultado em 2015-04-26.

[6] Andreas D. Baxevanis and B.F. Francis Ouellette. Phylogenetic analysis. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*, (14), 2001. Consultado em 2015-04-25.

[7] Richard W. Hamming. Error detecting and error correting codes. *Bell System Technical Journal*, (29(2)):pp.147–160, 1950. Consultado em 2015-06-13.

[8] Andre Rzhetsky and Masatoshi Nei. *Theoretical Foundation of the Minimum-Evolution Method of Phylogenetic Inference*. Consultado em 2015-04-29.

[9] Netbeans ide features. https://netbeans.org/features/index.html; Consultado em 2015-04-26.

[10] M.C.J. Maiden and R. Urwin. Multi-locus sequence typing: a tool for global epidemiology. *Trends in Microbiology*, (11(10)):pp.479–487, 2003. Consultado em 2015-04-27.

[11] Stuart K. Card Jeffrey Heer and James Landay. prefuse: a toolkit for interactive information visualization. URL: http://vis.stanford.edu/files/2005-prefuse-CHI.pdf Consultado em 2015-06-03.

[12] Gabriel Robins William R. Pearson and Tongtong Zhang. More reliable phylogenetic tree reconstruction. *Generalized Neighbor-Joining*. Consultado em 2015-04-23.

[13] Oracle. *Java Garbage Collection Basics*. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html, Consultado em 2015-04-29.

[14] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications Co., 2004.

[15] D. H. Huson and D. Bryant. Application of phylogenetic networks in evolutionary studies, mol. biol. evol. (23(2)):pp.254–267, 2006. Consultado em 2015-06-03.

[16] Daniel Peterson Alan Filipski Koichiro Tamura, Glen Stecher and Sudhir Kumar. Mega6: Molecular evolutionary genetics analysis version 6.0. 2013. Consultado em 2015-06-03.

[17] Keith Jolley. Public databases for molecular typing and microbial genome diversity. http://pubmlst.org/databases/ , Consultado em 2015-04-29.

Índice Remissivo

alelo, 3
análise filogenética, 7

CL - Complete-Linkage, 10

DataSet, 17
DNA - Deoxyribonucleic Acid, 3

estirpe, 3

filogenia, 7
folhas, 7

Garbage Collector, 23
gene, 3

Isolate, 16

JSON - JavaScript Object Notification, 29

Multilocus Sequence Typing - MLST, 4

Neighbor-Joining, 11

Population, 17

Single Nucleotide Polymorphism-SNP, 4
SL - Single-Linkage, 10

táxon, 7
taxa, 7
tipagem, 4
Typing Data, 17

UPGMA - Unweighted Pair Group Method with Arithmetic Mean, 8

Simbologia

d	distância entre dois *clusters*.
i, j	elementos presentes em *clusters*.
M	par de nós com menor distância.
R	distância de um ramo
S	soma total das distâncias de um nó para todos os outros.
T	Táxon.
U	nó que representa a união de dois *clusters*.

Anexo A

Schema para algoritmos hierárquicos

```
{ "order": ["leaf", "union", "root"],
  "type":"object",
  "properties":{
    "leaf":{
      "type":"array",
      "items":{
        "properties":{
          "uid":{"type":"integer"},
          "profile":{"type":"integer"}
        },
        "required":["uid","profile"]
      }},
    "union":{
      "type":"array",
      "items":{
        "properties":{
          "uid":{"type":"integer"},
          "distance":{"type":"number"},
          "leftID":{"type":"integer"},
          "rightID":{"type":"integer"}
        },
        "required":["uid","distance","leftID","rightID"]
      }},
    "root":{
      "type":"object",
      "items":{
        "properties":{
          "distance":{"type":"number"},
          "left":{"type":"integer", "minimum": 0},
          "right":{"type":"integer", "minimum": 0}
        },
        "required":["distance", "left", "right"]
      },
      "uniqueItems": true
} } }
```

Esquema A.1: Esquema JSON para validação no carregamento dos algoritmos hierárquicos.