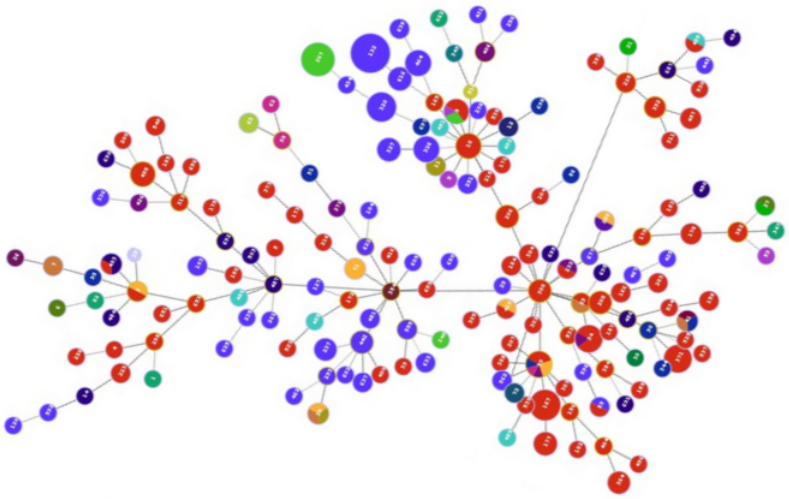
ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Visualização e Algoritmos de Clustering para Análise Filogenética

Adriano Sousa, 38210
Marta Nascimento, 38222



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

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

4 de Maio de 2015

Resumo

Atualmente, com o avanço das tecnologias, têm sido desenvolvidos novos métodos de identificação de diferentes estirpes, 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 de populações de bactérias, pois permitem a sua caracterização a nível da estirpe fornecendo assim informação importante no controlo de doenças infecciosas, estudos sobre a patogénese, na evolução de infeções e na genética de populações de bactérias.

Com a diminuição dos custos de sequenciação de **DNA**, esses métodos têm rapidamente vindo a tornar-se imprescindíveis na análise epidemiológica e têm vindo a substituir os métodos moleculares tradicionais, uma vez que fornecem os resultados necessários para a análise de populações de bactérias à escala global, mantendo também a sua utilidade na análise epidemiológica local. Isto é, tornam os seus resultados portáveis facilitando assim a comparação das estirpes microbianas entre laboratórios.

Apesar da existência de bases de dados on-line que disponibilizam perfis alélicos associados aos seus dados epidemiológicos, estas não são muito usadas e os seus dados nem sempre são os mais corretos uma vez que não existem ferramentas que permitam a análise dos mesmos. Estes avanços criaram a necessidade de algoritmos e ferramentas de processamento, análise e visualização desses dados, no contexto da epidemiologia, genética populacional e evolução. As ferramentas atuais que permitem a análise e visualização desses dados 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.

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 ***Minimum Spanning Tree*** (MST).

Palavras-chave: árvores filogenéticas;

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

As sequências biológicas, nomeadamente o **DNA**, **RNA** 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** (ácido desoxirribonucleico) 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 o processo (método, tecnologia, etc) que permite determinar a ordem precisa dos nucleótidos numa molécula DNA. Quando se obtém uma amostra microbial ou viral, seja de um indivíduo infetado ou do ambiente, uma cultura pura (*i.e.* de uma única espécie) é denominada de **estirpe isolada**. Desta é então extraído o DNA para a sua caracterização.

As tecnologias mais recentes permitem a sequenciação do DNA e RNA muito mais rapidamente e de forma mais barata. As máquinas que implementam esta tecnologia denominada de **"High Througput 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.* ava-

liação 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 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] que usa **Single Nucleotide Polymorphism** (SNP) [2]. 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 a 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 **Minimum Spanning Tree** (MST) baseados em diferentes matrizes distância. Posteriormente, é feita a visualização da árvore de filogenia correspondente.

Assim, este projeto tem como objetivos:

- implementação de dois algoritmos de *clustering*, um hierárquico (**UPGMA**) e outro aglomerativo (**Neighbor-Joining**);

- visualização dos outputs gerados para cada um dos algoritmos mencionados anteriormente. Nomeadamente para o algoritmo para o UPGMA uma representação no formato de um dendrograma e para o Neighbor-Joining uma visualização em árvore;

- salvar o estado de processamento de um DataSet através da serialização dos outputs dos algoritmos;

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.

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. E por fim, no capítulo 5, será apresentada uma avaliação experimental, tanto aos algoritmos como na comparação com outros softwares já existentes.

Capítulo 2

Algoritmos para Análise Filogenética

2.1 Descrição

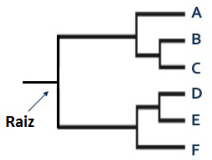
A filogenia [4] é o estudo da relação evolutiva existente entre diferentes grupos de organismos, tais como espécies, populações, etc, que provêm de um antecessor comum.

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.

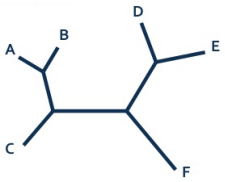
As árvores filogenéticas dividem-se em dois grandes grupos, com raiz e sem raiz. As árvores filogenéticas com raiz dividem-se em folhas, nós, ramos e raiz. As folhas representam as sequências (*taxa*[2]). Os nós representam a relação existente entre duas sequências, isto é, o antecessor comum a ambas. 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 e utiliza uma matriz durante a criação da árvore. Existem diversos algoritmos, hierárquicos e aglomerativos que seguem este princípio como o UPGMA e o Neighbor-Joining, respetivamente.

[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 - UPGMA

O UPGMA (*Unweighted Pair Group Method with Arithmetic Mean*) é um dos métodos mais utilizados para realizar inferência filogenética, através de árvores filogenéticas com raiz. É um algoritmo hierárquico pois procura 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* $\mathcal{T}_1$ e $\mathcal{T}_2$ é o resultado da média das distâncias 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.

2.2.1 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
ST-11	2	3	4	3	8	4
ST-1149	2	3	4	53	8	4
ST-1149-2	2	3	4	53	9	4
ST-658	2	3	4	3	8	110
ST-50	2	3	19	3	8	4

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

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 e dos seus respetivos ramos e definido o tamanho destes como metade da distância entre o par anterior - Figura 2.2

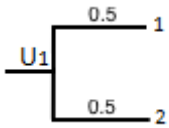


Figura 2.2: Par 1 e 2 com metade da distância entre eles.

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 recalcular 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.

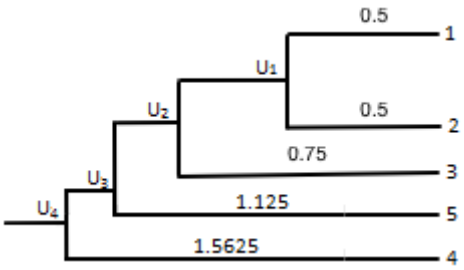


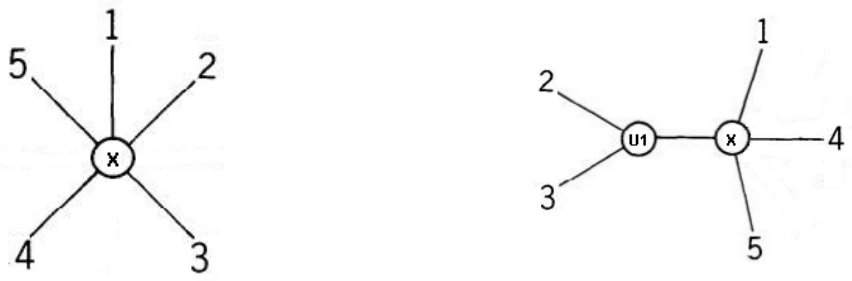
Figura 2.3: Representação final da árvore.

2.3 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, conseguindo baixar o custo do algoritmo para $O(N^2)$. Este método procura construir uma árvore que minimize a soma das distâncias de todos os ramos, adotando os critérios da evolução mínima [5].

Nós vizinhos são definidos como um par de *Táxon*, que têm uma ligação entre eles. No caso da figura 2.4, 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* 1 e 2 terem sido agrupados.

Figura 2.4: 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 identificando o par de nós vizinhos com menor distância (soma mínima dos comprimentos destes ramos).Para isto é utilizada uma matriz de distância, onde cada elemento deverá representar a relação de distância entre todos os nós.

Passo 1: Cálculo da soma de todas as distâncias para um dado nó:

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

Onde N é o numero de *Taxa* no conjunto

Passo 2: Calcular o par com menor (M) onde:

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

Em seguida, como exemplificado na figura 2.4b esse par é colocado num agrupamento X (novo nó com ligação da árvore aos nós vizinhos) e calculando as distâncias dos ramos desse novo nó para os nós vizinhos (1-U1 e 2-U1).

Passo 3: Criar o novo nó (U) que junta o par de nós com menor Mij e calcular qual a distância dos ramos:

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

Passo 4: Agregar o novo nó com a árvore (figura 2.4)

Em seguida é calculada a distância do nó X para o resto dos nós, e colocando essas distâncias dentro da 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 ultimo valor existente na matriz.

Passo 5: Calcular qual a distância deste novo nó (u) para os nós restantes:

$$D_{xu} = \frac{D_{ix} + D_{jx} - D_{ij}}{2}$$

Onde i e j são os nós vizinhos selecionados anteriormente.

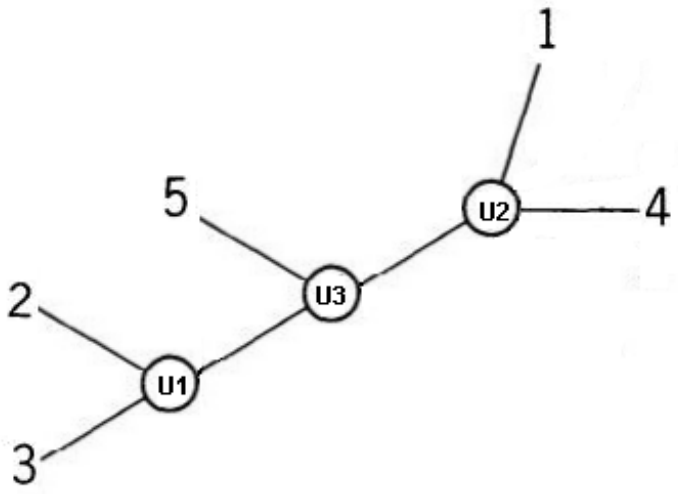


Figura 2.5: Representação da árvore gerada pelo algoritmo Neighbor-Joining para a tabela 2.1

Capítulo 3

Implementação dos Algoritmos na Plataforma PHYLOViZ

A plataforma NetBeans permite-nos um fácil trabalho de grupo graças à utilização de módulos, como tal existindo dois algoritmos a implementar, a divisão foi fácil , ficando cada elemento do grupo com um algoritmo.

3.1 Plataforma NetBeans

O NetBeans IDE [6] (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.

- Esta plataforma fornece um número variado de temas de visualização, permitindo ao programador alterar conforme o seu gosto, cativando mais a sua atenção.

- 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.

- Tem uma grande comunidade de utilizadores onde se pode facilmente obter apoio, tal como em blogues. Desta forma existe um grande número de *plug-ins* que podem ser instalados conforme a necessidade do utilizador.

- Permite uma melhor exploração do código e documentação, que por vezes é necessário quando é preciso saber como determinado código foi feito, cabendo ao programador rever e perceber.

- Por vezes é necessário ter mais de uma janela com código aberta, sendo assim possível fazer uma fácil e percetível divisão de ecrã. Desta forma pode-se ter projetos diferentes abertos em múltiplas janelas, pode existir a necessidade de ter classes diferentes do mesmo projeto, entre muitos outros motivos.

- Dispõe de um rápido e inteligente editor, indentação de linhas para uma melhor perceção do código, geração automática de conselhos e sugestão de auto-preenchimento. O editor não têm apenas suporte para Java, mas também para C/C++, XML e HTML, JavaScript, entre outros.

- 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.

Muitas destas características existem também em outros IDEs como o Eclipse e outros. Cada um dispõe das suas características, o que torna no final uma escolha do programador qual a utilizar.

3.2 Plataforma PHYLOViZ

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

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

Apresentação de 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, 7**] - 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.

- **goeBurst** - fornece uma implementação do algoritmo goeBurst.

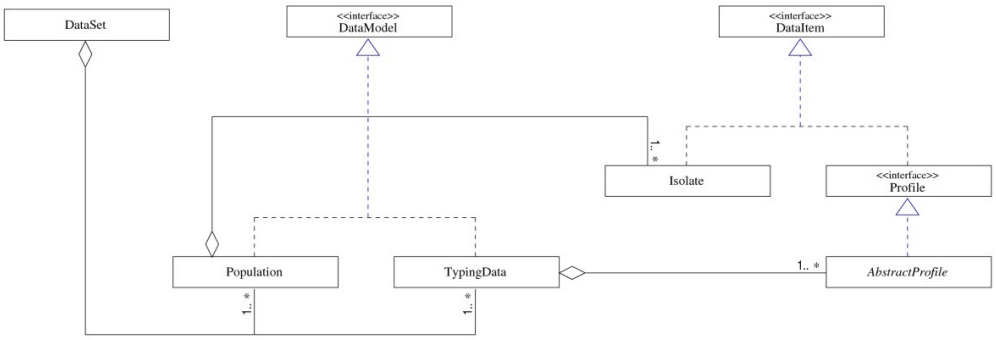


Figura 3.1: Representação UML do módulo Core.

Descrição da funcionalidade destas classes:

- **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 contem a informação da *Population* e o *TypingData* associado.

3.3 Algoritmo UPGMA

Para implementar este algoritmo foi necessário adicionar um novo módulo "PHYLOViZ UPGMA". Este módulo é responsável por toda a lógica necessária tanto à implementação como à integração deste na plataforma existente.

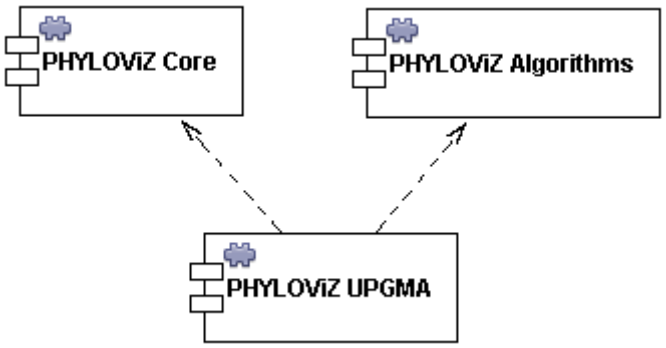


Figura 3.2: Representação da relação de dependência na integração do módulo PHYLOViZ UPGMA com os existentes

Core é o módulo onde existe grande parte da lógica da aplicação, existem todos os tipos principais da aplicação, tais como *Isolate, TypingData, Profiles*, etc. Desta forma, é possível

a utilização de todos estes tipo na implementação dos novos algoritmos a implementar.

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 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 e um posição.
- Como a matriz de distâncias é triangular inferior, cada nó tem a distância dele aos restantes, de acordo com o seu ID. (*e.g.* Sendo N o número inicial de nós, o nó com ID 1 terá até #N-1 distâncias, o nó com ID 2 terá #N-2, etc. Ou seja, uma vez que o nó 1 já tem a distância ao nó 2, este último não irá necessitar de guardar também a distância ao nó 1 pois é a mesma.)

A solução anterior encontra-se apresentada no algoritmo 1.

Esta solução procura reduzir o número de iterações à medida que se vão criando novos nós e ainda facilitar a procura da distância mínima. A dificuldade principal desta implementação é manter sempre atualizadas as posições e as distâncias dos nós existentes, uma vez que estas se encontram sempre a mudar ao longo do tempo. Isto é resolvido com a ajuda da biblioteca **Colt**[3] [8], nomeadamente através da classe `SparseDoubleMatrix1D`.

Esta classe é constituída por um array de elementos do tipo double que, neste caso, irão representar as distâncias a outros elementos, e por um outro array de indexação. Como cada nó tem uma posição específica, este array de indexação ajuda a identificar a que nó corresponde cada distância. Tudo isto, incluindo a adição de um novo né, é realizada com tempo constante, ou seja, complexidade $O(1)$.

A procura da distância mínima fica facilitada uma vez que cada nó sabe o nó par com que têm a distância mínima. Assim, esta procura torna-se mais rápida pois não é necessário

[3]Colt - biblioteca *Open Source* desenvolvida pelo CERN (European Organization for Nuclear Research) e procura o aumento da performance e a redução do gasto de memória.

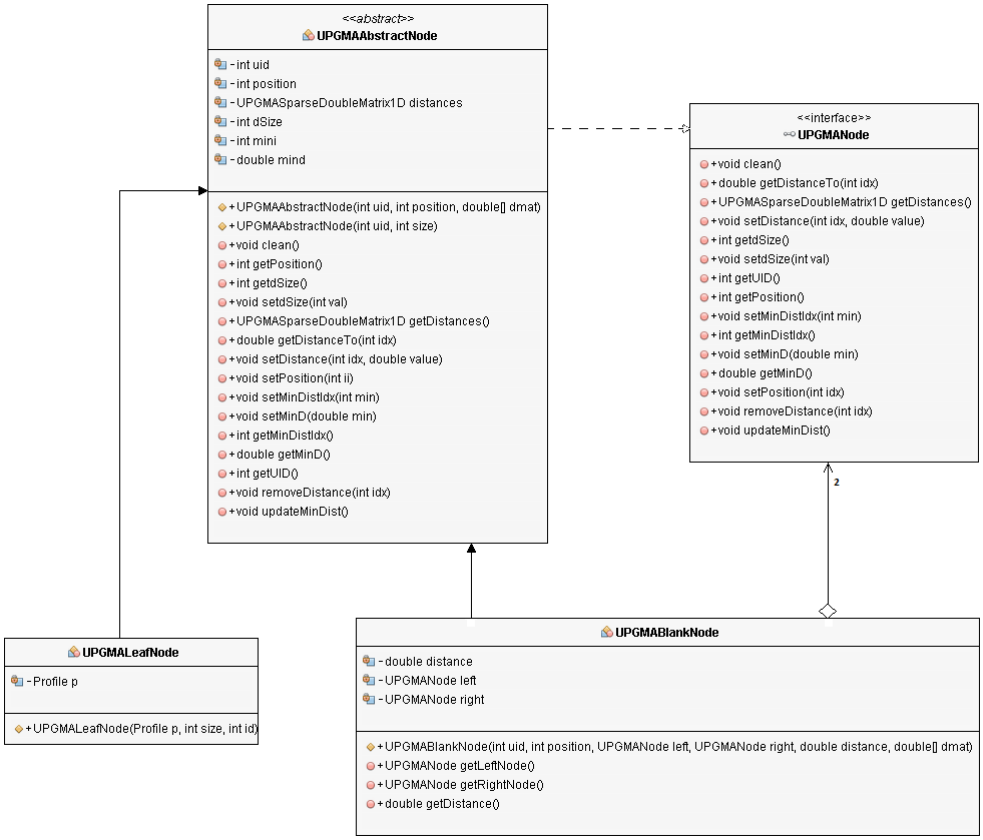


Figura 3.3: Representação UML dos elementos presentes na matriz de distância.

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 percorrer o array de distâncias para descobrir a nova distância mínima.

3.4 Algoritmo Neighbor-Joining

Um dos problemas com o algoritmo de Neighbor Joining [9] é o seu elevado 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. Durante o desenvolvimento da aplicação o custo algorítmico foi reduzido para $O(N^2)$.

Com visto a minimizar estes problemas, foram tomadas várias medidas, para que existisse uma grande melhoria 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 ultimo, que não terá quaisquer distâncias.

Algoritmo 1 Algoritmo de clustering - UPGMA.

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

1. Enquanto houver nós N em $Nodes$
   - 1.1 Pesquisar e obter o par P_1P_2 de nós com distância mínima
   - 1.2 Criar novo nó $Union$
   - 1.3 Associar P_1P_2 a $Union$
   - 1.4 Para cada nó N em $Nodes$
     - 1.1.1 Recalcular a distância de N a $Union$
   - 1.5 Adicionar $Union$ em $Nodes$ na posição de P_1
   - 1.6 Remover último elemento em $Nodes$ e adiciona-lo na posição de P_2
   - 1.7 Remover último nó em $Nodes$
   - 1.8 Remover P_1P_2 em $Nodes$

Quando encontrado um par de nós e colocado numa união, podemos eliminar esse par, ajudando assim o *garbage collector*[10] 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, tendo assim sido tomadas várias decisões:

- 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 o custo para $O(1)$;
- O cálculo das distâncias entre nós é realizado através de paralelismo [11], dividindo o trabalho por várias threads, tornando o processamento mais rápido;
- O pior caso encontrado era quando se procurava o par vizinhos, que é necessário fazer cálculos com as distâncias entre todos os nós, sendo neste reduzido de $O(N^3)$ para $O(N^2)$ (através criação de uma soma interna em ca nó). Mesmo com esta redução, também se optou pela utilização de paralelismo, dividindo o trabalho pelo número de processadores existentes na máquina, o que fará com que sejam possíveis ver tempos bastante diferentes de máquina para máquina.
- Sempre que utilizado o paralelismo, existe a possibilidade de para conjuntos pequenos, não existir qualquer vantagem a divisão de trabalho, como tal foi definido um limite mínimo de elementos.

A solução implementada encontra-se no algoritmo 2.

Algoritmo 2 Algoritmo de Neighbor Joining com paralelismo

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

1 Criação da matriz de distâncias

1.1 Por cada $u \in P$

1.1.1 Cria nó e guarda em Array A

1.2 Por cada processador

1.2.1 Seja c o índice a começar

1.2.2 Seja f o índice a acabar

1.2.3 Processa as distâncias dos elementos c a f

2 Gerar a árvore

2.1 Enquanto $N > 2$

2.1.1 MenorPar = ProcuraMenorPar()

2.1.2 Cria União com os dados de MenorPar

2.1.3 Remove as distâncias e insere distância de União

2.1.3 Remove de A os nós utilizados na União

2.1.3 Coloca União em A

2 ProcuraMenorPar

2.1 Procura novo par vizinho

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 conjunto de dados. Por este motivo foi criada uma solução que permite ao utilizador guardar e abrir um projeto à sua escolha.

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 sem qualquer custo computacional adicional.

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.

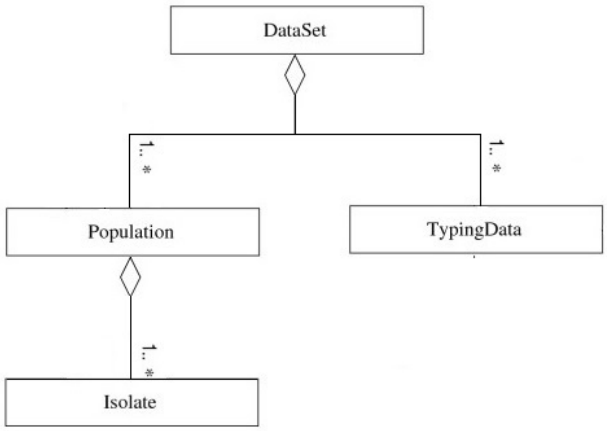


Figura 4.1: Representação de um DataSet.

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.

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

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

```
dataset-name="S. pneumoniae"
typing-factory=net.phyloviz.mlst.MLSTypingFactory
typing-file=spneumoniae.typing.csv
population-factory=net.phyloviz.core.util.PopulationFactory
population-file=spneumoniae.isolate.csv
population-foreign-key=3
```

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. Sendo a escrita feita pela aplicação, sabemos que esta fica corretamente feita, ao contrário da leitura

Cada algoritmo tem a sua estrutura, tendo sido assim necessária a criação de dois esquemas que serão utilizados para a validação durante a leitura. Este procedimento apenas é necessário pois o ficheiro pode ter sido alterado e deixado com falhas, devendo então avisar o utilizador do sucedido.

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

```
{
"required": ["leaf", "union"],
"type":"object",
   "properties":{
      "leaf":{
         "type":"array",
         "items":{
             "properties":{
                 "id":{"type":"integer", "minimum": 0},
                 "profile":{"type":"integer", "minimum": 0}
```

[4] JSON - JavaScript Object Notification

```
                },
                "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
        }
    }
}
```

O exemplo anterior pertence ao esquema do Neighbor-Joining, composto por dois elementos, os nós folha e os nós de união.

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.

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

Capítulo 5

Avaliação Experimental

Para testar a eficiência das diferentes implementações dos algoritmos UPGMA e Neighbor Joining foram realizados testes que visam demonstrar os tempos de execução de cada um dos algoritmos. Para isto, recorreu-se à base de dados pública PubMLST [12], nomeadamente 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 2.10 GHz
- Memória: 4.00 GB

5.1 Testes Unitários

A Tabela 5.1 apresenta a média dos tempos de execução do algoritmo UPGMA nas versões implementadas.

	Tempo de execução (milisegundos) / K	
N - nº de elementos	Versão 1	Versão 2
50	31	14
100	65	26
500	1065	305
1000	7781	845
5000	527470	16649

Tabela 5.1: Média de tempos obtidos durante a execução os testes de UPGMA, versão 1 e 2, sendo K o número de ciclos (50).

Como é possível verificar, existe um grande aumento de tempo quando se começa a analisar dados de grande dimensão. Este aumento torna-se mais visível na Figura 5.1.

A grande diferença apresentada entre as duas versões deve-se ao melhoramento das estruturas de dados utilizadas e a maior facilidade em encontrar a distância mínima. Esta diferença encontra-se explicada mais detalhadamente no Capítulo 3 - Secção 3.3.

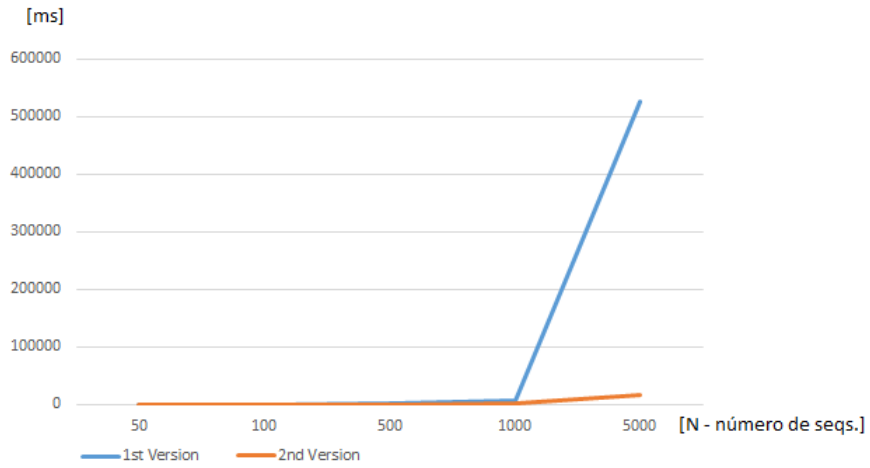


Figura 5.1: Representação gráfica da tabela 5.1

A Tabela 5.2 apresenta a média dos tempos de execução do algoritmo Neighbor-Joining nas versões implementadas.

	Tempo de execução (milisegundos)		
N - nº de elementos	Versão 1	Versão COLT	Versão Final
50	44	14	28
100	756	107	114
500	984	353	403
1000	4020	2890	1068
5000	265841	140552	25325

Tabela 5.2: Tempos obtidos durante a execução os testes de Neighbor-Joining, versão 1, COLT e final.

O mesmo acontece com os testes de Neighbor-Joining. Na Figura 5.2 estão representados os tempos para as três implementações. Estes resultados poderão variar bastante conforme a máquina e o respetivo número de processadores.

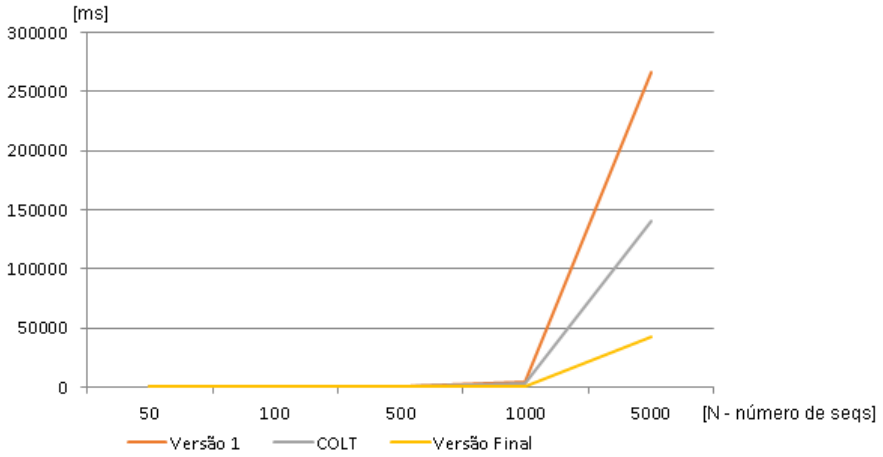


Figura 5.2: Representação gráfica da tabela 5.2

Este gráfico mostra de forma clara a evolução do algoritmo conforme se implementavam novas versões. A versão 1 é claramente a mais lenta, com pesquisas constante a N^2 elementos onde era ainda preciso ter cuidados com acessos a nós, que poderiam estar vazios.

A versão COLT mostrou grandes melhorias, grande parte devido á utilização de um array de indexação, acedendo unicamente a posições válidas.

Por fim, a versão final mostrou uma grande melhoria em relação a qualquer versão anterior. Utilizando também um array de indexação, colocando ao mesmo nível que a versão COLT, mais o paralelismo, onde se tentou usufruir de todas as potencialidades da máquina onde se encontra a correr. Fazendo divisão de tarefas por várias *threads*, o tempo decresceu acentuadamente.

Capítulo 6

Progresso do Projeto

O projeto tem decorrido normalmente, tentando ao máximo seguir a planificação definida inicialmente e presente na Tabela 6.1.

Início	Duração (semanas)	Descrição
02 de Março	1	Estudo sobre os algoritmos Neighbor-Joining e UPGMA.
02 de Março	1	Análise de alguns módulos mais relevantes da aplicação PHYLOViZ.
09 de Março	2	Escrita da Proposta.
23 de Março	3	Implementação dos algoritmos Neighbor-Joining e UPGMA.
06 de Abril	1	Criação de testes que validem os algoritmos implementados.
13 de Abril	2	Divisão dos diferentes tipos de abstração suportados para cada tipo de visualização.
27 de Abril	1	Relatório de progresso e apresentação individual.
04 de Maio	4	Implementação da visualização do output dos algoritmos.
01 de Junho	2	Criação e desenvolvimento do cartaz e entrega da versão beta.
15 de Junho	2	Serialização do output dos algoritmos.
29 de Junho	2	Otimização dos algoritmos e criação de testes de escalabilidade.
13 de Julho	2	Finalização do relatório e entrega da versão final.

Tabela 6.1: Calendarização Inicial do Projeto.

Com o decorrer do desenvolvimento das várias fases do projeto, começou-se a perceber que algumas destas fases poderiam ser agrupadas noutras. Isto é difícil de prever numa fase inicial uma vez que depende muito do tipo de implementações e das soluções adotadas para resolver cada problema. Tendo em conta estas alterações, foi redefinida a planificação - Tabela 6.2.

Atualmente, já se encontram implementados os dois algoritmos definidos, UPGMA e Neighbor-Joining, onde se conseguiu minimizar a memória ocupada e o tempo de processamento dos algoritmos para grandes datasets. Também já é possível guardar o estado atual de um dataset (excluindo para ja os outputs dos algoritmos) como um projeto PHYLOViZ, permitindo assim que este possa ser novamente carregado. Após carregar o projeto é então possível carregar o dataset associado.

O próximo passo será desenvolver a serialização dos outputs dos algoritmos para que também estes possam ser carregados e incluídos no projeto PHYLOViZ, evitando assim novo processamento.

Início	**Duração (semanas)**	**Descrição**
02 de Março	1	Estudo sobre os algoritmos Neighbor-Joining e UPGMA.
02 de Março	1	Análise de alguns módulos mais relevantes da aplicação PHYLOViZ.
09 de Março	2	Escrita da Proposta.
23 de Março	3	Implementação dos algoritmos Neighbor-Joining e UPGMA.
06 de Abril	1	Criação de testes que validem a implementação dos algoritmos.
13 de Abril	1	Serialização do output dos algoritmos.
20 de Abril	2	Divisão dos diferentes tipos de abstração suportados para cada tipo de visualização.
27 de Abril	1	Relatório de progresso e apresentação individual.
04 de Maio	4	Implementação da visualização do output dos algoritmos.
1 de Junho	3	Criação e desenvolvimento do cartaz e entrega da versão beta.
22 de Junho	2	Otimização dos algoritmos e criação de testes de escalabilidade.
06 de Julho	3	Finalização do relatório e entrega da versão final.

Tabela 6.2: Nova Calendarização do Projeto.

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 Jo ao André Carriço. PHYLOViZ. 2014. Consultado em 2015-04-27.

[4] 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.

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

[6] Netbeans ide features. Consultado em 2015-04-26.

[7] 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.

[8] CERN European Organization for Nuclear Research. Colt library. URL: http://dst.lbl.gov/ACSSoftware/colt/. Consultado em 2015-04-26.

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

[10] Oracle. *Java Garbage Collection Basics*. Consultado em 2015-04-29.

[11] Sheldon McKay Travis Wheeler Robert McLay, Dan Stanzione. *A Scalable Parallel Implementation of the Neighbor Joining Algorithm for Phylogenetic Trees*. Consultado em 2015-04-24.

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

Índice