



Università
Ca' Foscari
Venezia

Master's Degree
in Computer Science
Software Dependability and Cyber
Security

Final Thesis

**Computational analysis
of Nav1.7 protein
variants and tool for 3D
visualization of protein
structures**

Supervisor

Ch. Prof. Marta Simeoni

Graduand

Nikita Baldan
Matriculation number
857172

Academic Year

2019 / 2020

Contents

1	Introduction	5
2	Biological background	8
2.1	Proteins	8
2.2	Protein synthesis	15
2.2.1	Transcription	15
2.2.2	Translation	15
2.2.3	Non-covalent bonds	17
2.2.4	Mutations	17
2.3	PDB database and format	19
2.4	Neuropathies	19
2.5	Protein NaV1.7	21
3	Computational methods and tools	22
3.1	Protein structure determination	22
3.1.1	Homology modelling	23
3.2	Residue Interaction Network	24
3.3	Graph comparison through Graph Kernels	28
3.3.1	Graphs	28
3.3.2	Kernel functions	30
3.3.3	Kernel trick	30
3.3.4	Support Vector Machines	31

3.4	Graph Kernels	33
3.5	Computing Graph Kernel	43
4	Case study: description and results	45
4.1	Description of the case study	45
4.2	Results	49
4.2.1	Vertex histogram	49
4.2.2	Edge histogram	51
4.2.3	Random walk	53
4.2.4	Shortest path	53
4.2.5	Graphlet-sampling	55
4.2.6	Weisfeiler-Lehman	57
4.2.7	SVM	60
4.3	Conclusion	61
5	SphereMole	63
5.1	Technological choices	63
5.1.1	Unity scene structure	65
5.2	Functional description of the application	66
5.3	Requirements	68
5.3.1	Functional requirements	69
5.3.2	Non-functional requirements	72
5.3.3	Device requirements	73
5.4	User interface	73
5.5	Tests	78
5.5.1	Comparison with Chimera	78
5.5.2	Rendering time tests	79
6	Conclusion and future work	83

Abstract

This thesis is composed of two parts. The first part explores the possibility to use Graph Kernels to discriminate pathogenic versus non-pathogenic variants of a specific protein. All variants are represented as Residue Interaction Networks (RIN), where nodes are amino acids and edges represent non-covalent bonds between atoms of the two involved amino acids. This part is guided by a previous Master degree thesis that considered protein NaV1.7, which is responsible for the transmission of the pain signal from the peripheral nervous system to the brain. The thesis considered 85 genetic variants of the human NaV1.7, among which 30 are known to cause neuropathic diseases and 55 are instead neutral. The results of the first part highlight that some Graph Kernels are actually able to discriminate between pathogenic and neutral variants. This prompted the idea of realizing a 3D viewer able to show the three-dimensional structure of a protein and also its non-covalent bonds. The second part of the thesis describes *SphereMole*, an application for the visualization of the three-dimensional structure of a protein. In particular, *SphereMole* allows the visualization of two proteins structures and their visual comparison, also based on their non-covalent bonds.

Chapter 1

Introduction

This thesis has been inspired by a previous Master thesis [5] whose aim was to set up a computational pipeline to verify whether there is a relationship between neuropathic painful diseases and genetic mutations in sodium channel proteins. In the human body, peripheral nerve cells connect the brain to the rest of the body, allowing it to decode signals coming from outside. The signal management is essential for health because physico-chemical signals are converted into a potential action that propagates along the axons to the brain. Neuropathies are a category of disease which afflicts millions of people. Their mechanism is still not clear; approximately the 40% of patients could not relive their pain with currently available drugs. All these disorders share the same propagation channel for the stimulus, NaV1.7, a sodium ion channel protein, that can be afflicted by gain-of-function mutations along its primary sequence, that is, point mutations causing the protein to modify its functionality with deleterious effects. Those mutations have been directly connected to the onset of pain disorders such as IEN (Inherited Erythromelalgia), PEPD (Paroxysmal Extreme Pain Disorder) and SFN (Small Fibre Neuropathy). All these neuropathies share an incorrect way to recognise and feel the pain signals coming from the outside.

The computational pipeline adopted in [5] included a graph-based representation of proteins and machine learning techniques to be able to discern between

pain mutations and neutral variants not related to neuropathies. In particular, Residue Interaction Networks (RINs) were used to represent proteins as graphs where nodes identify amino acids and edges represent non-covalent bonds between atoms of the two involved amino acids. Moreover, a pair of Graph Kernels methods were used to compare RINs in order to find a feature that could discern pathological mutations from neutral ones. The Weisfeiler-Lehman Graph Kernel turned out to be a promising means to discriminate between pain and non-pain mutations.

Starting from that achievement, the first part of this thesis is devoted to explore whether other Graph Kernel methods could be useful for the same case of study and same goal. The input data are 85 mutations represented as RINs whose nodes (i.e. amino acids) are labelled with their position in the amino acid sequence. Among the 85 mutations, 30 are known to be related with neuropathies and 55 are instead neutral variants. Beside Weisfeiler-Lehman, the following kernel methods have been employed on the same input data: Vertex histogram, Edge histogram, Random walk, Shortest path and Graphlet-sampling. Among these, we show that three methods produce good results, which are Vertex histogram, Shortest path and Weisfeiler-Lehman Graph Kernels. The ability of these kernels to discern between mutations pain related and neutral genetic variants and, more in general, the importance of the non-covalent bonds for the three-dimensional protein shape, prompted the idea to visualize the three-dimensional structure of a protein together with its non-covalent bonds.

The second part of the thesis describes SphereMole, a standalone application for the visualization of the three-dimensional structure of proteins that runs on the main operating systems such as MacOS, Windows and Linux. SphereMole gives the possibility to visualize the three-dimensional structure of a single protein or two proteins in a split-view, at the atomic level, allowing the user to visualize the single non-covalent bond by just selecting it in a set of toggles. The split-view visualization also allows for evidentiating their non-covalent bonds and to visually compare the two given proteins and highlights their differences.

The thesis is organized as follows: Chapter 2 introduces the biological background on proteins and protein representation. Chapter 3 illustrates all the computational methods and tools employed in the first part of the thesis, such as RINs and Graph Kernels. Chapter 4 describes the original case study in [5] and the results of the application of the Graph Kernel methods to the same input data. Chapter 5 describes SphereMole, the stand-alone application developed for the visualization of the three-dimensional structure of proteins. Chapter 6 draws some conclusions and illustrates some possible future developments.

Chapter 2

Biological background

This chapter presents a general overview regarding the biological part that this thesis includes. All the biological references are taken from [4] [8] [S1] [S3].

2.1 Proteins

Although different from each other, proteins are polymers built from the same group of 20 amino acids. Polymers of amino acids are called polypeptides. A protein is made up of one or more polypeptides, whose folds or wraps determine a specific three-dimensional structure. Amino acids are organic molecules containing both carboxylic and amino groups. Figure 1 shows the general formula of an amino acid; at the center of the molecule there is an asymmetrical carbon atom called alpha (α) carbon linked to four different groups: an amino group, a carboxylic group, a hydrogen atom and a variable group indicated by the letter R.

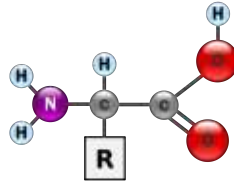


Figure 1: Amino acid structure

Group R, also called side chain, differs from one amino acid to another. Figure 2 shows the formulas of the 20 amino acids used by cells to build thousands of different protein molecules.

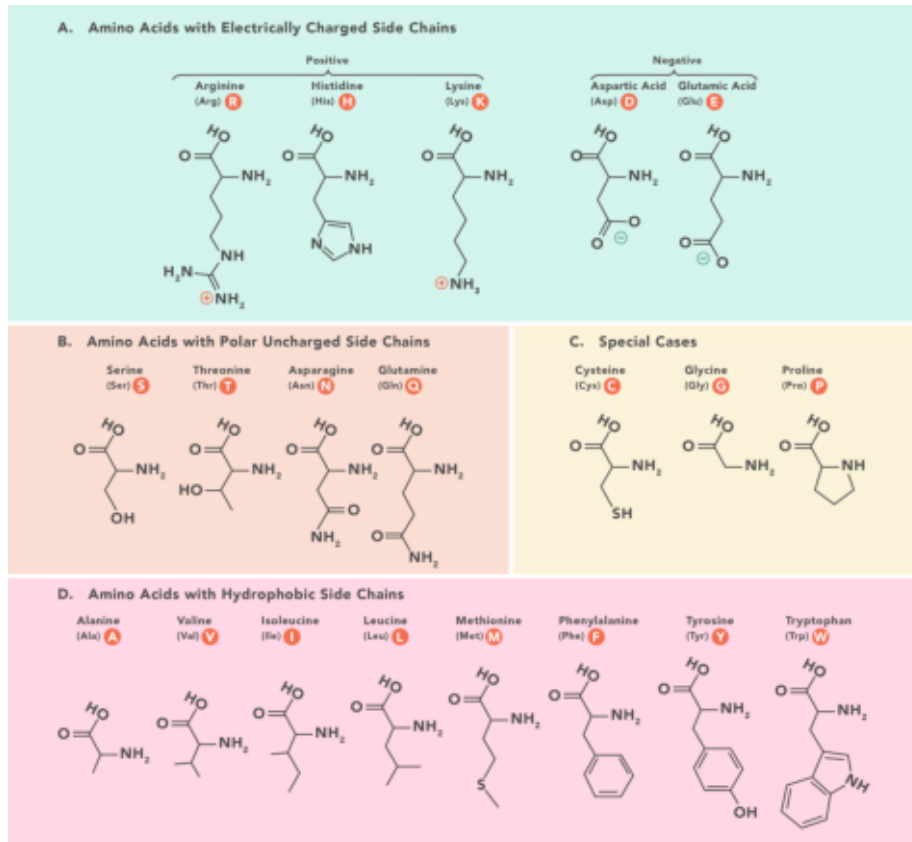


Figure 2: Amino acids that form a protein

The group R can be a simple hydrogen atom, as in the amino acid Glycine (the only one without asymmetric carbon atoms because two of the groups linked to the carbon atom are hydrogen atoms), or it can be constituted by a chain containing different functional groups, as in the case of Glutamine.

The different physical and chemical properties of the side chain determine the peculiar characteristics of a particular amino acid and influence its functional role in a polypeptide molecule.

The specific activities of proteins are the result of their complex three-dimensional architectures, the simplest level of which is given by the amino acid sequence.

Once determined, what information does the amino acid sequence of a protein provide regarding the three-dimensional structure (often referred to simply as “structure”) and the its function?

It is the particular amino acid sequence of each protein that determines its three-dimensional structure. When a cell synthesizes a polypeptide chain, this generally assumes the biologically active structure of the protein. This folding process is determined and made more efficient by the formation of a variety of bonds between different parts of the not yet folded polypeptide chain which, in turn, depend on the amino acid sequence.

The specific conformation of a protein determines its mode of action. Almost always the function of a protein depends on its ability to recognize other molecules to which it can bind. A particularly indicative example of association between form and function, which shows the exact complementarity of form between an antibody (a protein of the organism) and the particular foreign substance present on a particle of the flu virus to which the antibody binds, signaling the virus for destruction.

For a complete understanding of the function of a protein, information on its structure is often needed. Despite the remarkable diversity, all proteins share three levels of structure: primary, secondary and tertiary. A fourth structural level, the quaternary structure, appears when the protein is made up of several polypeptide chains.

- **Primary structure:** the primary structure of a protein is its specific amino acid sequence. The primary structure is analogous to the arrangement of the letters of a very long word. The number of random combinations of the 20 amino acids to form a polypeptide chain of 127 amino acids corresponds to 20^{127} . However, the precise primary structure of a protein is not determined by the random binding of amino acids but by hereditary genetic information, since the proteins are based on the RNA, consequently the RNA is based on the DNA, so if there is a mutation on the DNA, it could afflict the protein function.

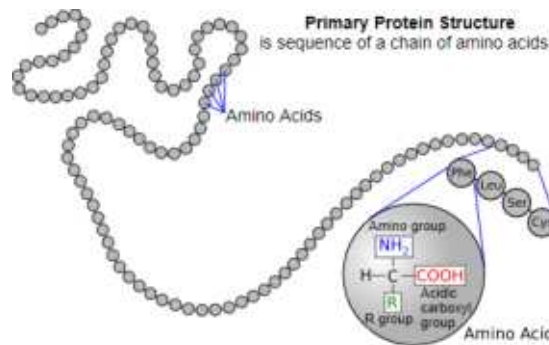


Figure 3: Proteins primary structure

- **Secondary structure:** many proteins have segments of polypeptide chain wound or folded in a repetitive way to form structures that contribute to the overall shape of the protein. These configurations, called as a whole secondary structure, are the result of the presence of hydrogen bonds at regular intervals along the skeleton of the polypeptide chain (not of the side chains of amino acids). Taken individually, these hydrogen bonds are weak; however, since they are repeated many times in a relatively large region of the polypeptide chain, they can stabilize the particular structure of that part of the protein.

One of these secondary structures is the α -helix, a delicate helical structure stabilized by hydrogen bonds present at intervals of three peptide bonds.

The other main type of secondary structure is the folded β -sheet.

- Tertiary structure: beyond the secondary structure of a protein is the tertiary structure. While the secondary structure implies interactions between the components of the peptide skeleton, the tertiary structure is the overall shape resulting from the interactions between the side chains (R groups) of the various amino acids, a type of interaction that contributes to the increase of the stability of the tertiary structure is represented by the so called hydrophobic interactions. When a polypeptide becomes functional natively, amino acids with hydrophobe side chains (apolar) usually associate in the inner nucleus of the protein, away from contact with water. The so-called idrophobic interaction is caused by the action of the water molecules which exclude the substances and the apolar groups when they fold into it, they bind to each other and with the hydrophilic portions of the protein, through hydrogen bonds. When the non-polar side chains of amino acids come into close mutual contact, van der Waals forces help to keep them close. The tertiary structure is also stabilized by the hydrogen bonds that form between polar side chains, and by the ionic bonds between side chains with positive and negative electric charge. In all these cases these are weak interactions, however the overall effect of the stabilizes the peculiar three-dimensional structure of a protein. The shape of a protein can be further stabilized by the presence of strong covalent bonds called disulfide bridges.

Other possible non-covalent bonds are the π -cation and π - π stacking interactions. Those bonds can be described as strong non-covalent interactions based on of aromatic rings. An aromatic ring is a property of cyclic (ring-shaped), planar (flat) structures of atoms with a ring of resonance bonds that gives increased stability.

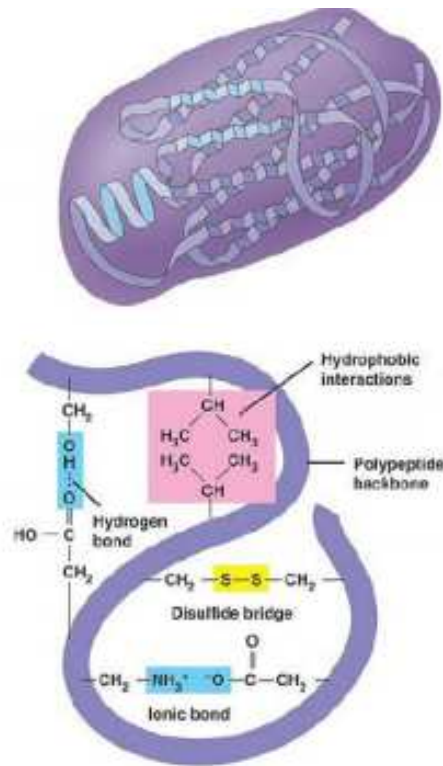


Figure 4: Non-covalent bond representation

- Quaternary structure: Certain proteins are made up of two or more polypeptide chains aggregated into a functional macromolecule. The quaternary structure is the protein as a whole resulting from the aggregation of these polypeptide subunits. Each subunit contains a non-peptide component, called heme, in which there is an iron atom which represents the binding site for oxygen.

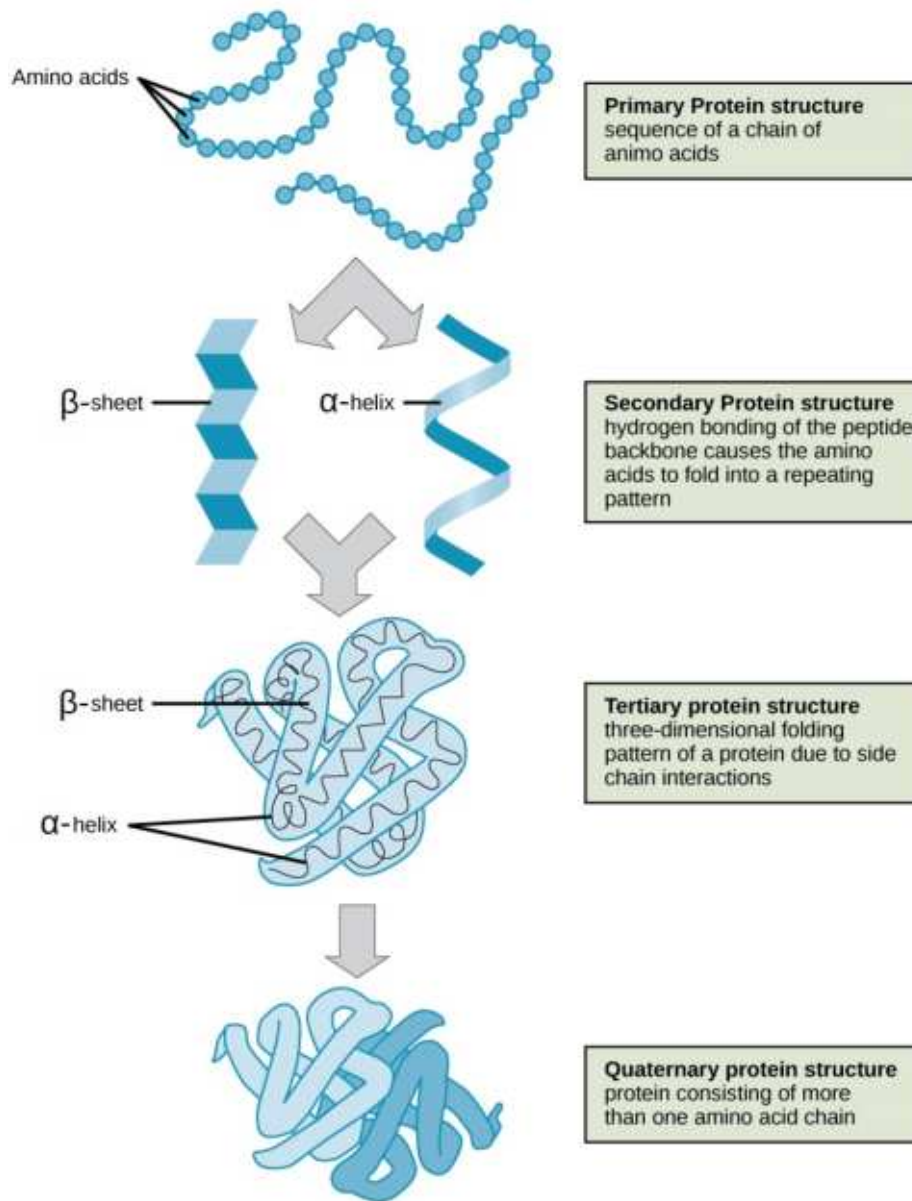


Figure 5: Protein structure

2.2 Protein synthesis

The protein synthesis is the process by which a nucleotide sequence is converted into the sequence of amino acids forming a protein. In the protein synthesis take active part the m-RNA, t-RNA and r-RNA.

The m-RNA copies the information contained in the DNA and transports it from the nucleus to the cytoplasm (this stage is called transcription); t-RNA and r-RNA translate the message written on the m-RNA into a sequence of amino acids (this stage is called translation). During protein synthesis therefore, genetic information passes from DNA to RNA and from RNA to proteins.

2.2.1 Transcription

Transcription is the stage of protein synthesis in which information is transferred from DNA to RNA, according to the rules of pairing complementary bases. Nitrogen bases need to protrude from the DNA double helix. Therefore the stretch of DNA to be transcribed is opened at a specific point, characterized by the “start reading” AUG triplet. An enzyme, RNA-polymerase, binds to one of the two DNA strands that serves as a “template”, and proceeds from end 3' to end 5' by binding the complementary ribonucleotides present in the nucleus. In this way, m-RNA is formed.

When the RNA polymerase reaches the “end of reading” triplet, the m-RNA separates from the DNA chain, passes through the pores of the nuclear membrane and enters the cytoplasm, where it binds to the ribosomes. The “model” DNA rewinds to form the double helix, or binds to a new RNA-polymerase molecule to synthesize a new strand of m-RNA.

2.2.2 Translation

Translation is the stage of protein synthesis in which the instructions carried by the m-RNA are translated into the correct sequence of amino acids to form a protein.

The translation (Figure 6) takes place in the ribosome (formed by r-RNA and proteins), composed of two subunits: the small one contains a binding site for the m-RNA; the large one has two binding sites for two t-RNA molecules and a site that catalyzes the formation of the peptide bond between two adjacent amino acids.

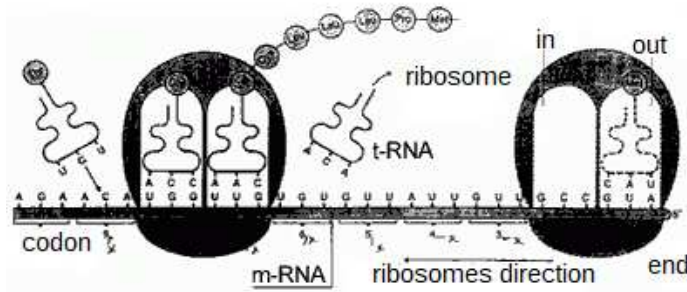


Figure 6: Translation phase

Each t-RNA molecule is specific for a single amino acid and is able to recognize both the amino acid that it has to carry, and the complementary codon of m-RNA associated with the ribosome.

Translation begins when two codons of the m-RNA strand bind to the small subunit of a ribosome. The first codon is the AUG “start reading” triplet, which corresponds to the amino acid methionine; the second encodes the first true amino acid of the protein. The two t-RNAs, which respectively have the starting anticodon and the complementary anticodon to the second codon, bind to the large subunit and a peptide bond is formed (i.e. the link between amino acids that forms proteins) between the two amino acids transported .

The initial t-RNA detaches from the ribosome while the dipeptide (the two amino acids joined by the peptide bond) remains bound to the second t-RNA. The ribosome moves over another m-RNA codon and a new t-RNA molecule with its amino acid is placed at the empty binding site of the ribosome. A new peptide bond is created and the tripeptide welds to the last t-RNA. The process

of lengthening the polypeptide chain continues in this way until all the triplets have been translated and the “end of reading” codon is reached. The complete protein detaches from the ribosome and specific enzymes cleave the bond with methionine.

2.2.3 Non-covalent bonds

A non-covalent bond is a type of chemical bond that typically bond between macromolecules. Non-covalent bonds are used to bond large molecules such as proteins and nucleic acids, those bonds are weaker than covalent bonds, but they are crucial for biochemical processes such as the formation of double helix and the folding that brings to the three-dimensional structure that assumes a protein [3]. The non-covalent bonds analyzed in this thesis are the following: Hydrogen bond, Van der Waals interaction, Ionic bond, $\pi - \pi$ stacking and π -cation. The first three bonds are the most numerous and significant in forming the three-dimensional structure of the protein.

2.2.4 Mutations

Human genome is identical between cells of the same organism but it is almost the same between people, it contains a number of nucleotide changes. The genetic variation represents the 0.6% of different genomes.

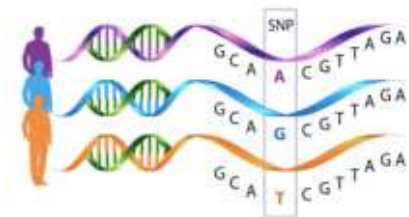


Figure 7: Mutations in genetic code

The DNA mutations can be classified by their effects on the DNA module.

They can be:

- Substitution: base is replaced by one of the other three bases
- Deletion: block of one or more DNA pairs is lost
- Insertion: block of one or more DNA pairs is added
- Inversion: 180 degrees rotation of piece of DNA
- Reciprocal translocation: parts of non homologous chromosomes change places
- Chromosomal rearrangements: affect many genes at one time.

The triplet nature of the genetic code means that base changes within coding sequence can have several different outcomes.

- Missense mutation: changes an amino acid to another amino acid. This may or may not affect protein function, depending on whether the change is “conservative” or “nonconservative”, and what the amino acid actually does
- Nonsense mutation: changes an amino acid to a STOP codon, resulting in premature termination of translation
- “Silent” mutation: does not change an amino acid, but in some cases can still have a phenotypic effect, e.g., by speeding up or slowing down protein synthesis, or by affecting splicing.
- Frameshift mutation: Deletion or insertion of a number of bases that is not a multiple of 3. Usually introduces premature STOP codons in addition to lots of amino acid changes.

The presence of genetic variations is responsible for differences between one person and another. Those variations can have different consequences.

They can have no effects on the person afflicted by that mutation; they can cause non-pathogenic effect as, for example, inter-individual differences (eyes

color, hair, etc) but they can cause also pathogenic effects, as disease-causing, disease predisposition and drug responses.

Mutations are also classified by their impact on proteins function which can be classified by loss-of-function and gain-of-function. The distinction between those two effect on the protein is not always clear.

Loss-of-function usually means that the protein function has been compromised, or in other words, the protein does not perform the function that used to perform. Gain-of-function can be linked to an erroneous increasing of the protein activity increase in protein activity.

2.3 PDB database and format

The term PDB refers to the Protein Data Bank (<http://www.rcsb.org/pdb/>). Since 1971, the Protein Data Bank archive (PDB) has served as the single repository of information about the 3D structures of proteins, nucleic acids, and complex assemblies. The Worldwide PDB (wwPDB) organization manages the PDB archive and ensures that the PDB is freely and publicly available to the global community [S9].

Proteins in the PDB database are represented using the PDB format, a text format where all relevant information of the three-dimensional structure of the protein are included. In particular, a PDB file include preliminary information about the protein, such as name, the species and tissue from which is was obtained, authorship, revision history, journal citation, references, amino acid sequence, stoichiometry, secondary structure locations, crystal lattice and symmetry group, and more importantly, the 3D coordinates of all atoms of the protein.

2.4 Neuropathies

The Peripheral neuropathies are a heterogeneous group of disorders that affect the peripheral nerve fibers. The basic unit of the peripheral nervous system

is the nerve cell or neuron. Each neuron consists of a cell body and a long extension, called axon, leading pulses between the cell body and the periphery, where it comes into contact with the receptors, specialized structures, present in the muscles, in the skin and internal organs. Many axons are surrounded by a membrane, the myelin sheath, which allows electrical impulses to be transmitted more quickly and efficiently.

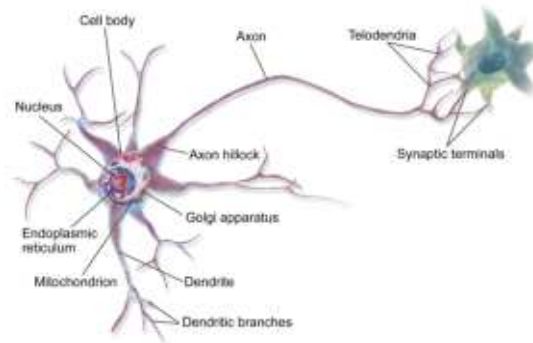


Figure 8: Interaction between axon and cell

The causes of neuropathy are varied. Depending on the origin, neuropathies can be classified as:

- Hereditary: they are caused by genetic abnormalities
- Acquired: they constitute the majority of neuropathies and are due to diseases acquired in the course of life, metabolic factors, oxidative stress of nervous tissue, trauma, infection and inflammation.
- Idiopathic: the cause of the neuropathy is unknown. Depending of its manifestations, also the idiopathic neuropathy can be sensory, motor, or mixed.

Neuropathic pain is pathological as it is characterized by an amplification process of nociceptive messages that can occur in both the peripheral and central nervous system.

The neuropathies that have been taken into consideration are: primary erythromelalgia (IEM), paroxysmic extreme pain disorder (PEPD) and small fiber neuropathy (SFN).

The primary erythromelalgia can be diagnosed to people that has as consequences vasodilatation (mainly on feet), burning pain and those are triggered by warm temperatures (from 27°C to 32°C degrees).

The paroxysmic extreme pain disorder can be diagnosed on people that have as consequences skin redness and warmth, attacks of severe pain in various parts of the body. Those attacks can start in the infancy.

The congenital insensitivity to pain is a rare condition in which a person cannot feel physical pain. It is an extremely dangerous condition (childhood death risk for unnoticed serious injuries or infections).

2.5 Protein NaV1.7

NaV1.7 is a voltage-gated sodium channel protein, present in both eukaryotes and prokaryotes, its role is the transmission of pain from the periphery to the brain.

NaV1.7 is present at the endings of pain-sensing nerves, the nociceptors, close to the region where the impulse is initiated. Stimulation of the nociceptor nerve endings produces “generator potentials”, which are small changes in the voltage across the neuronal membranes.

It is usually expressed at high levels in two types of neurons: the nociceptive (pain) neurons at dorsal root ganglion (DRG) and trigeminal ganglion and sympathetic ganglion neurons, which are part of the autonomic (involuntary) nervous system.

Chapter 3

Computational methods and tools

This chapter presents some essential methods and tools used in this thesis. In particular, it introduces Residue Interaction Networks (RINs) as a means to represent proteins and Graph Kernels as a method to compare them.

3.1 Protein structure determination

Computational methods for protein structure determination, such as homology modeling, are thought to determine the third structure of a protein. The main goal of those methods is, starting from a sequence, use existing structures to determine the candidate three-dimensional structure of the given sequence, more complex structure. Starting from a given protein sequence, whose three-dimensional structure is not available, the main goal of the computational methods is to use homology modelling if the given sequence is more than the 30% similar to a known structure, otherwise to use fold recognition and AB initio.

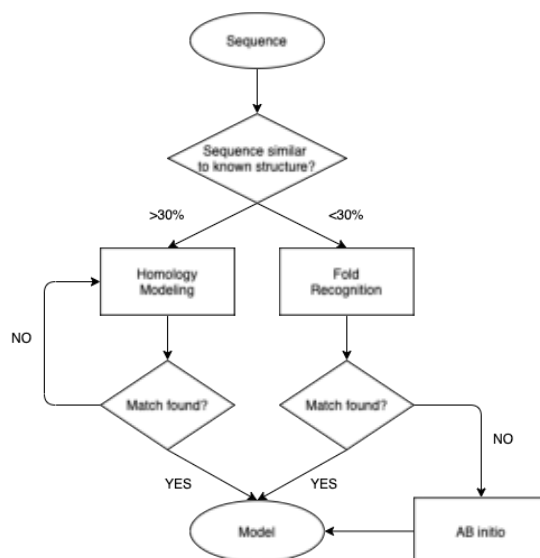


Figure 9: Flowchart for choosing the most suitable computational method

3.1.1 Homology modelling

The main idea behind homology modelling is that, if the sequence similarity is high, then the structural similarity is high too.

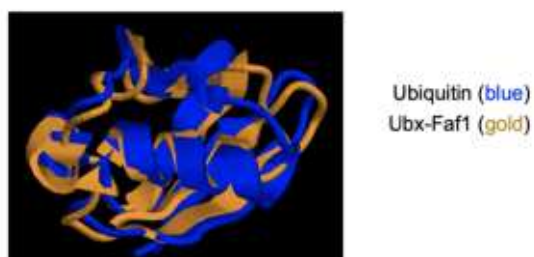


Figure 10: Sequence/structural similarity

The steps followed by this method are the following:

1. Template selection (tertiary structure known) and alignment with the target sequence (tertiary structure unknown)

2. Alignment correction
3. Backbone generation
4. Loop modelling
5. Side chain modelling
6. Model optimization
7. Model validation
8. Iteration

This method produces good models if the sequence similarity is high enough.

3.2 Residue Interaction Network

Proteins are the eventual result of the flow of genetic information within a biological system via transcription and translation. A protein folds in order to execute varied biological functions. Protein folding results from interactions among the constituents of a protein's amino acid residues. It provides information for proteins to adopt the correct 3D structure. As concluded by Anfinsen in [2], the primary structure of the proteins contains all the information for the folding process. Thus, understanding proteins by means of residue interactions serves as a rational and potent approach.

The challenge remains to map these structures onto simple yet effective representations, which are capable of characterizing the essential and functional properties of the analyzed structures [6].

The final solution was found by using a graph as representation of the residues interactions in a protein.

A Residue Interaction Network is a graph where nodes represent amino acids and edges represent non-covalent interactions. Figure 11 shows some RIN exam-

ples where, in the upper part, there are some three-dimensional representations of various proteins and below them there are their corresponding RINs.

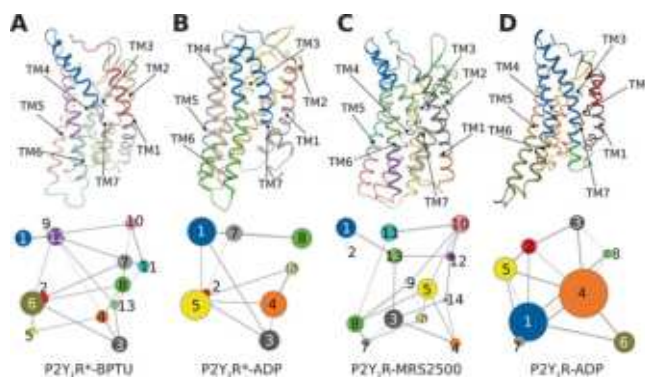


Figure 11: RIN examples

Starting from the three-dimensional protein structure, its RIN representation can be computed by finding all non-covalent bonds between its composing atoms. In order to perform this task it has been used RING 2.0. RING 2.0 is a webserver that takes in input the PDB file representing the three-dimensional structure of a protein, and, by setting some options, such as distance thresholds, chain, model, sequence separation, network policy, interaction type, gives in output the corresponding RIN of the given PDB file.

Figure 12: RING input option

The non-covalent bonds taken into consideration are: Hydrogen bond, Van der Waals interactions, Ionic bonds, $\pi - \pi$ stacking and π -cation. The output format provided by RING2.0 is a XML file, which describes all the nodes present in the RIN (amino acids) and all the edges involved as non-covalent bonds. An example of node description is the following one:

```

1 <node id="n1401">
    <data key="v_Residue">GLU</data>
3    <data key="v_Bfactor_CA">0</data>
    <data key="v_Tap">0</data>
5    <data key="v_Degree">3</data>
    <data key="v_Chain">A</data>
7    <data key="v_NodeId">A:1757: -:GLU</data>
    <data key="v_Rapdf">-68.807</data>
9    <data key="v_pdbFileName">A81S.pdb#1757.A</data>
    <data key="v_x">128.522</data>
11   <data key="v_y">158.343</data>
    <data key="v_Position">1757</data>
13   <data key="v_z">141.147</data>
    <data key="v_Dssp"> </data>
15   <data key="v_name">A:1757: -:GLU</data>
</node>

```

An example of edge description is the following one:

```

<edge source="n0" target="n3">
2   <data key="e_Distance">2.856</data>
    <data key="e_Interaction">HBOND:MC.MC</data>
4   <data key="e_Angle">28.865</data>
    <data key="e_Orientation">None</data>
6   <data key="e_Positive">None</data>
    <data key="e_Energy">17</data>
8   <data key="e_Atom1">O</data>
    <data key="e_Atom2">N</data>
10  <data key="e_Cation">None</data>
    <data key="e_NodeId2">A:117: -:ARG</data>
12  <data key="e_NodeId1">A:114: -:PRO</data>

```

```
14 <data key="e_Donor">A:117: _:ARG</data>  
</edge>
```

The XML file given in output by RING2.0, describes the protein as a graph, where all the nodes represent amino acids and the edges represent the non-covalent bonds present in that given protein. Figure 13 represents a visual representation of a RIN where in red is highlighted a $\pi - \pi$ stacking interaction and in blue an ionic bond.

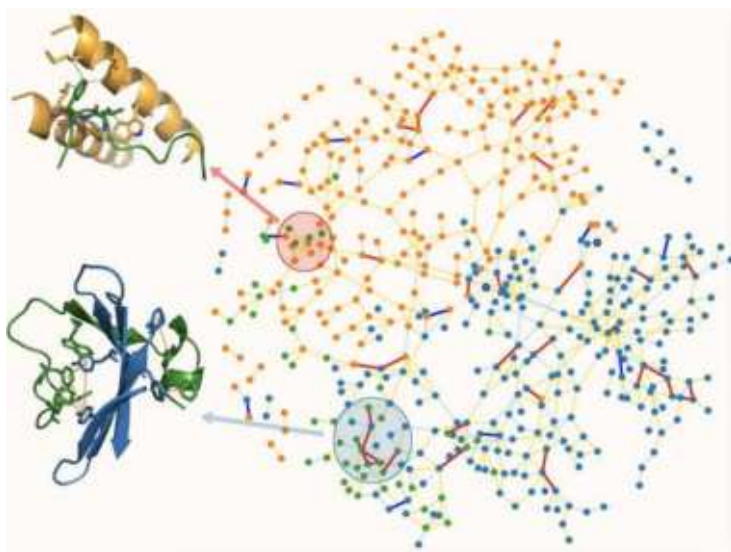


Figure 13: RIN visualization

3.3 Graph comparison through Graph Kernels

This section first presents some basic definitions on graphs and then the formal theory about graph kernel methods [10] [12].

3.3.1 Graphs

An undirected graph is a pair $G = (V, E)$ consisting of a set of vertices V and a set of edges $E \subseteq V \times V$ which connect pairs of vertices.

A graph may have labels on its nodes and edges (most graphs derived from chemistry are annotated by categorical labels from a finite set) so it should be given the definition of labeled graph.

A labeled graph is a graph $G = (V, E)$ provided with a function $l : V \cup E \rightarrow \Sigma$ that assigns, from a discrete set of labels Σ , labels to the vertices and edges of the graph. A node-labeled graph is a graph with labels on its vertices, similarly, a graph with labels on edges is called edge-labeled. A fully-labeled graph is a graph with labels on both the vertices and edges.

Two nodes are said to be adjacent if there is an edge that links directly those two nodes. Let A_{ij} be the element in the i -th column of matrix A . Then, the adjacency matrix A of a graph $G = (V, E)$, such that $|V| = n$, can be defined as follows

$$A_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

Matrix A is of dimensionality $n \times n$. The *neighborhood* $\mathcal{N}(v_i)$ of vertex v_i is the set of all vertices adjacent to v_i . Hence, $\mathcal{N}(v_i)$ of vertex v_i is the set of all vertices adjacent to v_i .

A concept closely related to the neighborhood of a vertex v_i is its *degree* $deg_G(v_i)$. Given an undirected graph $G = (V, E)$ and a vertex $v_i \in V$, the degree of v_i in G is the number of edges incident to v_i , and is defined as

$$deg_G(v_i) = |\{v_j : \{v_i, v_j\} \in E\}| = |\mathcal{N}(v_i)|$$

A *path* in a graph $G = (V, E)$ is a sequence of vertices v_1, v_2, \dots, v_k where $v_i \in V \forall 1 \leq i \leq k+1$ and $\{v_i, v_{i+1}\} \in E \forall 1 \leq i \leq k$. The length of the walk is equal to the number of edges in the sequence, i. e. k in the above case. A walk in which $v_i \neq v_j \Leftrightarrow i \neq j$ is called path.

A *shortest path* from vertex v_i to vertex v_j of a graph G is a path from v_i to v_j such that there exist no other path between these two vertices with smaller length, that could be the path that uses less edges, as shown in Figure 14.

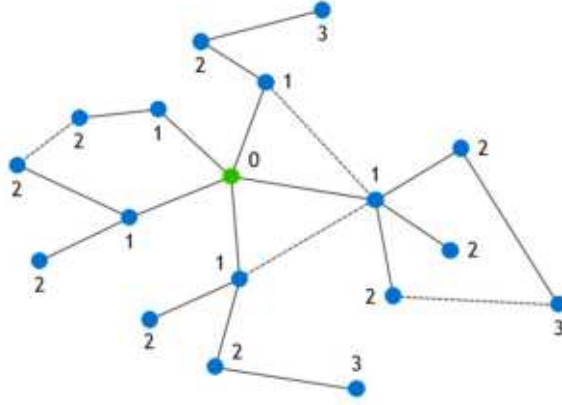


Figure 14: Example of shortest path based on path with fewer edges

Since the project is about similarity between graphs, the graph isomorphism must be introduced.

Let G_1, G_2 be two graphs and $f : V_{G_1} \rightarrow V_{G_2}$ a mapping such that (x, y) is an edge of G_1 iff $(f(x), f(y))$ is an edge of G_2 . Then f is an *isomorphism*, and G_1 and G_2 are said to be isomorphic.

At the moment we do not know a polynomial time algorithm for graph isomorphism, but we also do not know whether the problem is NP-complete or not [10].

On the other hand, we know that subgraph isomorphism is NP-complete [10]. Subgraph isomorphism checks whether there is a subset of edges and vertices of

G_1 that is isomorphic to a subgraph G_2 .

3.3.2 Kernel functions

Given a set of N inputs $x_1, \dots, x_N \in X$ and a function $k : X \times X \rightarrow \mathbb{R}$, the $N \times N$ matrix K defined as $K_{ij} = k(x_i, x_j)$ is called the *kernel matrix* with respect to the inputs x_1, \dots, x_N .

A real $N \times N$ matrix K satisfying

$$\sum_{i=1}^N \sum_{j=1}^N c_i c_j K_{ij} \geq 0$$

$\forall c_i \in \mathbb{R}$ is called positive semidefined.

Informally, a kernel function measures the similarity between two objects. Furthermore, kernel functions can be represented as inner products between the vector representations of these objects. Specifically, if we define a kernel k on $X \times X$, then there exists a mapping $\phi : X \rightarrow H$ into a Hilbert space with inner product such that:

$$\forall x_i, x_j \in X : k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$$

A Hilbert space is an inner product space (that is a vector space or function space with an operation for combining two vectors or functions) which also possesses the completeness property that every Cauchy sequence of points taken from the space converges to a point in the space itself [12].¹

3.3.3 Kernel trick

Kernel methods exploit kernel functions to work on high-dimensional spaces, implicit feature spaces without having to compute the coordinates of the data in that space. This is achieved by performing inner products between the images of all pairs of data in the feature space. This operation is called kernel trick. It

¹A sequence $\{p_n\}$ in a metric space X is called a Cauchy sequence if for every $\epsilon > 0$ there exists $N \in \mathbb{N}$ such that for all $m, n \geq N$ we have $d(p_m, p_n) < \epsilon$. Assuming that X is a compact metric space, $\{p_n\}$ is a Cauchy sequence in X that converges to some point.

is extremely useful in the case the dataset is not linearly separable, but can be easily separated by an hyperplane in a higher-dimensional space.

Formally, a kernel maps two objects x and x' via a mapping ϕ into the feature space \mathcal{H} , measuring their similarity in \mathcal{H} as $\langle \phi(x), \phi(x') \rangle$. The kernel trick is nothing but computing the inner product in \mathcal{H} as kernel in the input space: $k(x, x') = \langle \phi(x), \phi(x') \rangle$. These methods implicitly represent data in a feature space and compute inner products between them in that space using a kernel function. These inner products can be interpreted as the similarities between the corresponding objects. Machine learning tasks such as classification and clustering can be carried out by using only the inner products computed in that feature space. Kernel methods are very popular and have been successfully used in a wide variety of applications.

3.3.4 Support Vector Machines

Support-vector machines (SVMs, also support-vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier[1].

A SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on the side of the gap on which they fall.

The SVM's are less effective when the data have noise and overlapping points, there is a problem in drawing a clear hyperplane without misclassifying.

One problem of SVM could be overfitting. In statistics, overfitting is the production of an analysis that corresponds too closely or exactly to a particular

set of data, and may therefore fail to fit additional data or predict future observations reliably. Figure 15 gives an example of an overfitted model, since the computation of the green line is computationally very hard.

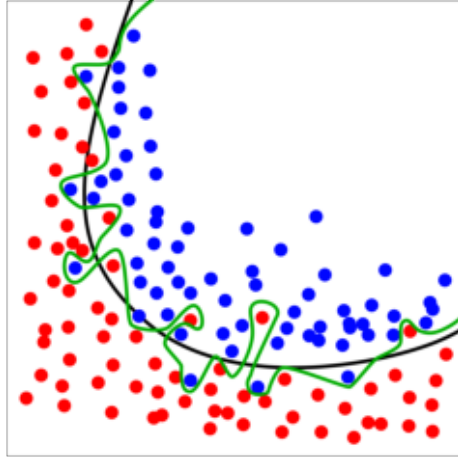


Figure 15: The green line represents an overfitted model and the black line represents a regularized model. While the green line best follows the training data, it is too dependent on that data and it is likely to have a higher error rate on new unseen data, compared to the black line.

Overfitting could happen when a model that would just repeat the labels of the samples that it has just seen would have a perfect score but would fail to predict anything useful on yet unseen data. To avoid it, it is a common practice when performing a (supervised) machine learning experiment to hold out part of the available data as a test set. Figure 13 shows the cross-validation flowchart, that represents how the method is implemented: part of the data are hold out as test set and a supervised machine learning algorithm is performed on the other data.

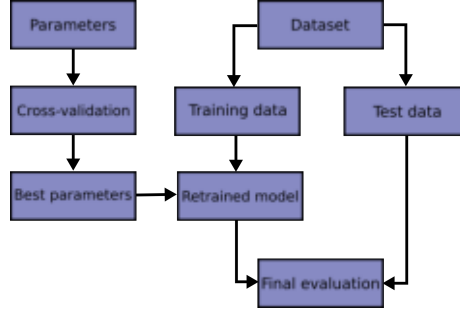


Figure 16: Cross-validation flowchart

The optimization problem of several kernel methods such as the Support Vector Machines is convex, that is possible to find a solution, only if the employed function is positive semidefinite [12].

3.4 Graph Kernels

Given two graphs G and G' from the space of graphs \mathcal{G} , the problem of graph comparison is to find a mapping

$$s : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}$$

such that $s(G, G')$ quantifies the similarity (or dissimilarity) of G and G' .

A graph kernel is a kernel function that computes an inner product on graphs. Graph kernels can be intuitively understood as functions measuring the similarity of pairs of graphs. They allow kernelized learning algorithms (that are algorithms based on kernel trick) such as SVMs to work directly on graphs, without having to do feature extraction to transform them to fixed-length, real-valued feature vectors.

To better explain graph kernels, let us introduce R-convolution kernels, a family graph kernels are instances of. These kernels compare decompositions of two discrete, structured, compound objects. Most R-convolution kernels simply count the number of isomorphic substructures in the two compared graphs and

differ mainly by the type of substructures used in the deconvolution and the algorithms used to count them efficiently.

$$k_{convolution}(x, x') = \sum_{(x_d, x) \in R} \sum_{(x'_d, x') \in R} k_{parts}(x_d, x'_d)$$

Graph kernels are nothing but convolution kernels on pairs of graphs. A new decomposition relation R results in a new graph kernel. A graph kernel makes the whole family of kernel methods applicable to graphs. Formally, once we define a positive semi-definite kernel $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ on a set X , there exists a map $\phi : X \rightarrow \mathcal{H}$ into a Hilbert space \mathcal{H} such that $k(x, y) = \phi(x)^T \phi(y) \quad \forall x, y \in X$. Also, the distance between $\phi(x)$ and $\phi(y)$ can be computed as

$$\|\phi(x), \phi(y)\|^2 = \phi(x)^T \phi(x) + \phi(y)^T \phi(y) - 2\phi(x)^T \phi(y)$$

Concluding, the main challenge in applying kernel methods to graphs is to define appropriate positive semidefinite kernel functions on the set of input graphs which are able to reliably assess the similarity among them [10].

Figure 14 represents an example of feature space and map defined by graph kernels. Any kernel on a space of graphs \mathcal{G} can be represented as an inner product after graphs are mapped into a Hilbert space \mathcal{H}

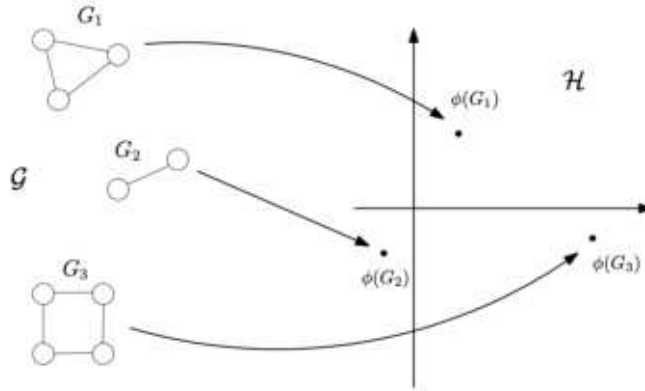


Figure 17: Example of graph kernels mapping

Some of the well-known graph kernel methods are presented below.

Vertex histogram kernel

The vertex histogram kernel is a basic linear kernel on vertex label histograms. The kernel assumes node-labeled graphs. Let G be a collection of graphs, and assume that each of their vertices comes from an abstract vertex space \mathcal{V} . Given a set of node labels Σ , $l : \mathcal{V} \rightarrow \Sigma$ is a function that assigns labels to the vertices of the graphs. Without loss of generality, assume that $\Sigma = \{1, \dots, d\}$, the vertex label histogram of a graph $G = (V, E)$ is a vector $f = (f_1, \dots, f_d)^T$, such that $f_i = |\{v \in V : l(v) = i\}|$ for each $i \in \Sigma$. Let f, f' be the vertex label histograms of two graphs G, G' , respectively. The vertex histogram kernel is then defined as the linear kernel between f and f' , that is

$$k(G, G') = \langle f, f' \rangle$$

The complexity of the vertex histogram kernel is linear in the number of vertices of the graphs [10].

Edge histogram kernel

The edge histogram kernel is a basic linear kernel on edge label histograms. The kernel assumes edge-labeled graphs. Let \mathcal{G} be a collection of graphs, and assume that each of their edges comes from an abstract edge space ε . Given a set of node labels Σ , $l : \varepsilon \rightarrow \Sigma$ is a function that assigns labels to the edges of the graphs. Without loss of generality, assume that $\Sigma = \{1, \dots, d\}$. The edge label histogram of a graph $G = (V, E)$ is a vector $f = (f_1, \dots, f_d)^T$, such that $f_i = |\{(v, u) \in E : l(v, u) = i\}| \forall i \in \Sigma$. Let f, f' be the edge label histograms of two graphs G, G' , respectively. The edge histogram kernel is then defined as the linear kernel between f and f' , that is

$$k(G, G') = \langle f, f' \rangle$$

The complexity of the edge histogram kernel is linear in the number of edges of the graphs [10].

The two kernels defined above are indeed positive semidefinite, but they both correspond to rather naive concepts.

Random walk

The k -step random walk kernel compares random walks up to length k in the two graphs. The most widely-used kernel from this family is the geometric random walk kernel which compares walks up to infinity assigning a weight λ^k ($\lambda < 1$) to walks of length k in order to ensure convergence of the corresponding geometric series. We next give the formal definition of the geometric random walk kernel. Given two node-labeled graphs $G_i = (V_i, E_i)$ and $G_j(V_j, E_j)$, their direct product $G_X = (V_X, E_X)$ is a graph with vertex set:

$$V_X = \{(v_i, v_j) : v_i \in V_i \wedge v_j \in V_j \wedge l(v_i) = l(v_j)\}$$

and edge set:

$$E_X = \{ \{(v_i, v_j), (u_i, u_j)\} : \{v_i, u_i\} \in E_i \wedge \{v_j, u_j\} \in E_j \}$$

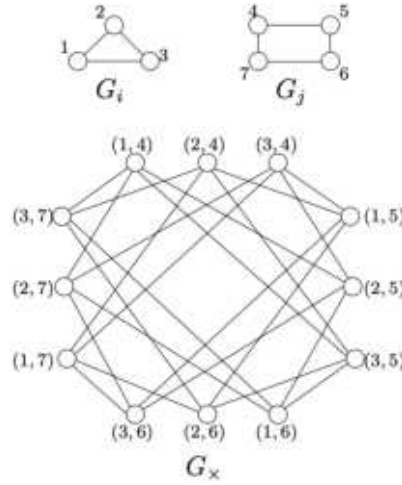


Figure 18: Example of direct product graph

Performing a random walk on G_X is equivalent to performing a simultaneous random walk on G_i and G_j . Now G_i and G_j be two graphs, let A_X denote the

adjacency matrix of their product graph G_X , and let V_X denote the vertex set of the product graph G_X .

Then, the geometric random walk kernel is defined as

$$K_X^\infty(G_i, G_j) = \sum_{p,q=1}^{|V_X|} \left[\sum_{l=0}^{\infty} \lambda^l A_X^l \right]_{pq} = e^T (\mathcal{I} - \lambda A_X)^{-1} e$$

where \mathcal{I} is the identity matrix, e is the all-ones vector, and λ is a positive, real-valued weight. The geometric random walk kernel converges only if $\lambda < 1/\lambda_X$ where λ_X is the largest eigenvalue of A_X . Direct computation of the geometric random walk kernel requires $\mathcal{O}(n^6)$ time. The computational complexity of the method severely limits its applicability to real-world applications. To account for this, Vishwanathan proposed in [12] four efficient methods to compute random walk graph kernels which generally reduce the computational complexity from $\mathcal{O}(n^6)$ to $\mathcal{O}(n^3)$ [10].

Shortest path

The high computational complexity of the graph kernels based on walks, subtrees and cycles renders them impractical for most real-world scenarios. Computing all the paths in a graph and computing the longest paths in a graph are both NP-hard problems. Instead, shortest paths can be computed in polynomial time.

The shortest-path kernel decomposes graphs into shortest paths and compares pairs of shortest paths according to their lengths and the labels of their endpoints. The first step of the shortest-path kernel is to transform the input graphs into shortest-paths graphs. Given an input graph $G = (V, E)$, the algorithm creates a new graph $S = (V, E_S)$. The shortest-path graph S contains the same set of vertices as the graph from which it originates. The edge set of the former is a superset of that of the latter, since in the shortest-path graph S , there exists an edge between all vertices which are connected by a walk in the original graph G . To complete the transformation, the algorithm assigns labels

to all the edges of the shortest-path graph S . The label of each edge is set equal to the shortest distance between its endpoints in the original graph G . Given the above procedure for transforming a graph into a shortest-path graph, the shortest-path kernel is defined as follows.

Let G, G' be two graphs, and $S = (V, E), S' = (V', E')$ their corresponding shortest-path graphs. The shortest-path kernel is then defined as

$$k(G, G') = \sum_{e \in E} \sum_{e' \in E'} k_{walk}^{(1)}(e, e')$$

where $k_{walk}^{(1)}(e, e')$ is a positive semidefinite kernel on edge walks of length 1.

In labeled graphs, the $k_{walk}^{(1)}(e, e')$ kernel is designed to compare both the lengths of the shortest paths corresponding to edges e and e' , and the labels of their endpoint vertices.

Let $e = \{v, u\}$ and $e' = \{v', u'\}$. Then $k_{walk}^{(1)}(e, e')$ is usually defined as

$$\begin{aligned} k_{walk}^{(1)}(e, e') &= k_v(l(v), l(v'))k_e(l(e), l(e'))k_v(l(u), l(u')) \\ &\quad + k_v(l(v), l(u'))k_e(l(e), l(e'))k_v(l(u), l(v')) \end{aligned}$$

where k_v is a kernel comparing vertex labels, and k_e a kernel comparing shortest path lengths.

In terms of runtime complexity, the shortest-path kernel can be computed in $\mathcal{O}(n^4)$ [10].

Figure 19 shows an example of explicit computation of the shortest path Graph Kernel. Note that each triple is a feature and corresponds to: (label of source vertex; label of sink vertex; shortest path length between the two vertices)

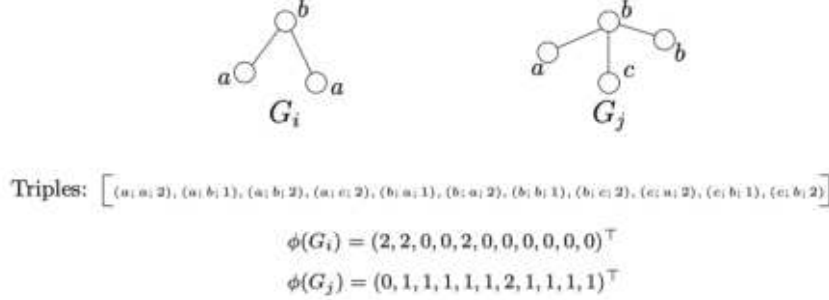


Figure 19: Example of explicit computation of the shortest path kernel.

Graphlet sampling

The graphlet kernel decomposes graphs into graphlets (i.e. small subgraphs with k nodes where $k \in \{3, 4, 5\}$) and counts matching graphlets in the input graphs. The kernel was originally designed to address scalability issues experienced by earlier approaches. In fact, the graphlet kernel was one of the first kernels that could cope with very large graphs using a simple sampling scheme. However, apart from the scalability issue, the graphlet kernel was also motivated by the graph reconstruction conjecture which states that any graph of size n can be reconstructed from the set of all its subgraphs of size $n - 1$.

Formally, let $\mathcal{G} = \{graphlet_1, \dots, graphlet_d\}$ be the set of $size - k$ graphlets. Let also $f_G \in \mathbb{N}^d$ be a vector such that its $i - th$ entry is equal to the frequency of occurrence of $graphlet_i$ in G , $f_{G,i} = \#(graphlet_i \subseteq G)$. Then, the graphlet kernel is defined as follows.

Let G, G' be two graphs of size $n \geq k$, and $f_G, f_{G'}$ vectors that count the occurrence of each graphlet of size k (not necessarily connected) in the two graphs. Then the graphlet kernel is defined as

$$k(G, G') = f_G^T f_{G'}$$

As is evident from the above definition, the graphlet kernel is computed by explicit feature maps. First, the representation of each graph in the feature space is computed. And then, the kernel value is computed as the dot product

of the two feature vectors. The main problem of graphlet kernel is that an exhaustive enumeration of graphlets is very expensive.

Since there are $\binom{n}{k}$ size k subgraphs in a graph, computing the feature vector k for a graph of size n requires $\mathcal{O}(n^k)$. To account for that, Shervashidze et al. (2009) resorted to sampling. Following Weissman et al. (2003), they showed that by sampling a fixed number of graphlets the empirical distribution of graphlets will be sufficiently close to their actual distribution in the graph. An alternative proposed strategy that reduces the expressivity of the kernel is to enumerate only the connected graphlets of k vertices, and not all the possible graphlets [10]. Figure 20 shows all the possible graphlets, with graphlet size equals to four, that are computed by the Graph Kernel.

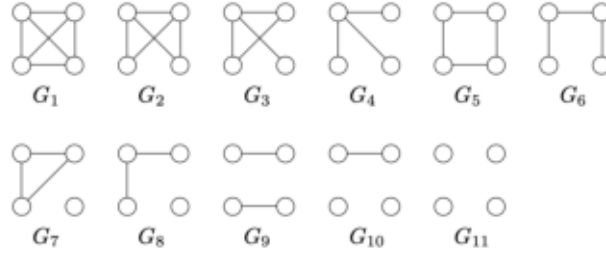


Figure 20: All graphlets of size 4

Weisfeiler-Lehman

The key idea of the Weisfeiler-Lehman algorithm is to replace the label of each vertex with a multiset label consisting of the original label of the vertex and the sorted set of labels of its neighbors. The resultant multiset is then compressed into a new, short label. This relabeling procedure is then repeated for h iterations. Note that this procedure is performed simultaneously on all input graphs. Therefore, two vertices from different graphs will get identical new labels if and only if they have identical multiset labels.

More formally, given a graph $G = (V, E)$ endowed with a labeling function $l = l_0$, the Weisfeiler-Lehman graph of G at height i is a graph $G_i = (V, E)$

endowed with a labeling function l_i which has emerged after i iterations of the relabeling procedure described above. The Weisfeiler-Lehman sequence up to height h of G consists of the Weisfeiler-Lehman graphs of G at heights from 0 to h , $\{G_0, \dots, G_h\}$.

Formally, let k be any kernel for graphs, that we will call the base kernel. Then the Weisfeiler-Lehman kernel with h iterations with the base kernel k between two graphs G and G' is defined as

$$k_{wl}(G, G') = k(G_0, G'_0) + k(G_h, G'_h)$$

where h is the number of Weisfeiler-Lehman iterations, and $\{G_0, \dots, G_h\}$ and $\{G'_0, \dots, G'_h\}$ are the WL sequences of G and G' respectively. From the above definition, it is clear that any graph kernel that takes into account discrete node labels can take advantage of the Weisfeiler-Lehman framework and compare graphs based on the whole Weisfeiler-Lehman sequence.

When the base kernel compares subtrees extracted from two graphs, the computation involves counting common original and compressed labels in the two graphs. The emerging Weisfeiler-Lehman subtree kernel is a byproduct of the Weisfeiler-Lehman test of isomorphism.

Let G, G' be two graphs. Define $\Sigma_i \subseteq \Sigma$ as the set of letters that occur as node labels at least once in G or G' at the end of the i -th iteration of the WL algorithm. Let Σ_0 be the set of original labels of G and G' . Assume all Σ_i are pairwise disjoint. Without loss of generality, assume that every $\Sigma_i = \{\sigma_{i1}, \dots, \sigma_{i|\Sigma_i|}\}$ is ordered. Define a map $c_i : \{G, G'\} \times \Sigma_i \rightarrow \mathbb{N}$ such that $c_i(G, \sigma_{ij})$ is the number of the occurrences of the letter σ_{ij} in the graph G . The Weisfeiler-Lehman subtree kernel on two graphs G and G' with h iterations is defined as

$$k(G, G') = \langle \phi(G), \phi(G') \rangle$$

where

$$\phi(G) = (c_0(G, \sigma_{(01)}), \dots, c_h(G, \sigma_{h|\Sigma_h|}))$$

and where

$$\phi(G') = (c_0(G', \sigma_{(01)}), \dots, c_h(G', \sigma_{h|\Sigma_h|}))$$

It can be shown that the above definition is equivalent to comparing the number of shared subtrees between the two input graphs.

The Weisfeiler-Lehman subtree kernel considers all subtrees up to height h , instead of subtrees of exactly height h . Furthermore, the Weisfeiler-Lehman subtree kernel checks whether the neighborhoods of two vertices match exactly, while the subtree kernel considers all pairs of matching subsets of the neighborhoods of two vertices.

In Figure 21, right after the algorithm, it is shown the computation of the Weisfeiler-Lehman subtree kernel with $h = 1$ for two graphs. Here $\{1, 2, \dots, 12\} \in \Sigma$ are considered as letters. Note that compressed labels denote subtree patterns: for instance, if a node has label 8, this means that there is a subtree pattern of height 1 rooted at this node, where the root has label 2 and its neighbours have labels 3 and 5 [10].

Algorithm:

1. Multiset-label determination
 - Assign a multiset-label $M_i(v)$ to each node v in G which consist of the multiset $\{l_{i-1}(u) | u \in \mathcal{N}(v)\}$
2. Sorting each multiset
 - Sort elements in $M_i(v)$ in ascending order and concatenate them into a string $s_i(v)$
 - Add $l_{i-1}(v)$ as a prefix to $s_i(v)$
3. Label compression
 - Map each string $s_i(v)$ to a compressed label using hash function $f : \Sigma^* \rightarrow \Sigma$ such that $f(s_i(v)) = f(s_i(w))$ iff $s_i(v) = s_i(w)$
4. Relabeling
 - Set $l_i(v) := f(s_i(v)) \forall$ nodes in G

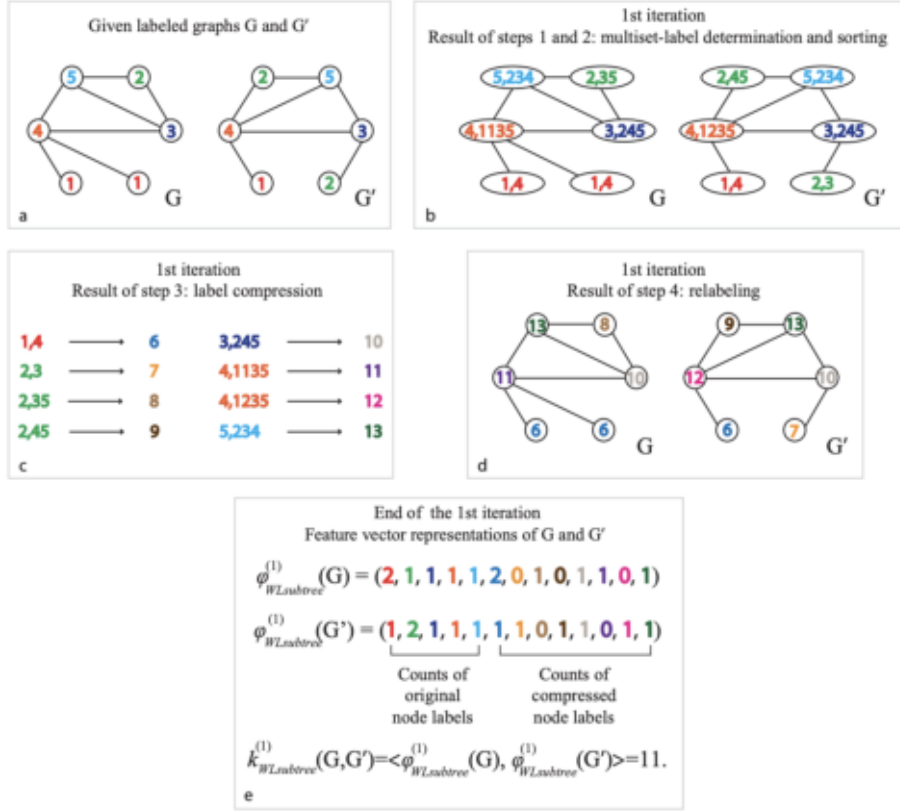


Figure 21: Computation example of Weisfeiler-Lehman Graph Kernel

Per pair of graphs the runtime takes $\mathcal{O}(mh)$.

For N graphs, the Weisfeiler-Lehman subtree kernel with h iterations on all pairs of these graphs can be computed in $\mathcal{O}(Nhm + N^2hn)$

3.5 Computing Graph Kernel

In order to compare RINs through Graph Kernels methods some scripts have been realized. In particular, Python3 [S10] has been used as programming language and the GraKel[S4], matplotlib[S5], scikit-learn[S11] libraries.

Python is a multi-paradigm programming language. Object-oriented programming and structured programming are fully supported, and many of its features

support functional programming and aspect-oriented programming. Many other paradigms are supported via extensions, including design by contract and logic programming. Python uses dynamic typing and a combination of reference counting and a cycle-detecting garbage collector for memory management. It also features dynamic name resolution (late binding), which binds method and variable names during program execution.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

Chapter 4

Case study: description and results

In this chapter we illustrate the case study that inspired the first part of the thesis and that was developed in a previous Master Thesis [5]. We moreover show the results of applying some graph kernel methods to the case study itself.

4.1 Description of the case study

This thesis was inspired by a previous Master thesis [5] whose aim was to build a computational pipeline for the NaV1.7 protein, a sodium channel protein that is responsible for the transmission of the pain signals from the peripheral nervous system to the brain.

In particular, the main goal of the thesis was to set up a computational pipeline able to discern between mutations of NaV1.7 that are known to be pain related and genetic variants that are known to be neutral. As described in Figure 22, the pipeline is divided in four parts:

- **Protein structure:** it starts with models generated from a given by Homology modelling which produces as final result the PDB file that rep-

resents the three-dimensional structure of the protein variant.

- **RIN generation:** the PDB file is given in input to RING2.0 [6] that computes the corresponding RIN.
- **Centrality metrics calculation:** some metrics on the generated RINs are computed to see if they can be useful to discriminate pathogenic vs non pathogenic mutations.
- **Graph Kernels and Dominant Set Clustering:** the last part of the pipeline is the application of Graph Kernels methods and Dominant Set Clustering in order to find a feature that can discriminate between pathogenic and neutral mutations.

Regarding Graph Kernels, the Weisfeiler-Lehman Graph Kernel showed to be able to produce good results.

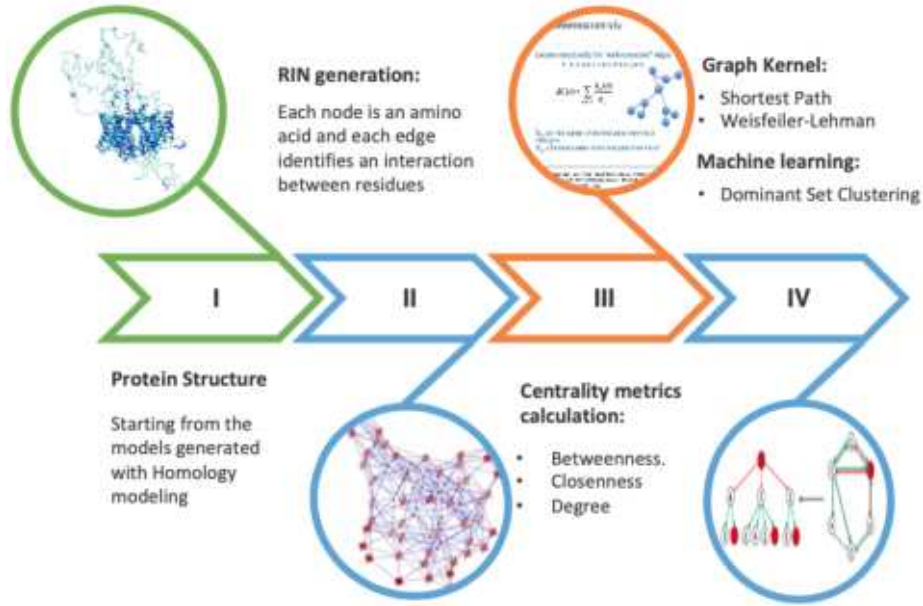


Figure 22: Computational pipeline proposed by [5]

The pipeline described in Figure 22 was applied to 85 mutations of the NaV1.7 protein: 30 mutations are known to be related with pain disorders and

55 are neutral variants, i.e, they do not affect the functionality of the protein. All the considered mutations are listed in Table 1 : each mutation has its own ID and name and is either neutral or related to one of the three considered diseases: IEM, PEPD and SFN.

A mutation name has to be read in the following way: the first letter is the amino acid that is mutated, the number in the middle is the position of that amino acid in the protein sequence and the last letter is the amino acid that has replaced the first one. For example, take into consideration the mutation **I136V**:

- **I** represents the amino acid that is mutated;
- **136** is its position in the protein sequence;
- **V** represents the substituting amino acid.

In Table 1, the ids 0-29 are occupied by the pathogenic mutations, while the other ids 30-84 are relative to the neutral ones.

ID	NAME	PAT	ID	NAME	PAT	ID	NAME	PAT
0	I136V	IEM	29	M1532I	SFN	58	K1412I	Neutral
1	S211P	IEM	30	S126A	Neutral	59	K1415I	Neutral
2	F216S	IEM	31	L127A	Neutral	60	S1419N	Neutral
3	I234T	IEM	32	M145L	Neutral	61	V1428I	Neutral
4	S241T	IEM	33	N146S	Neutral	62	A1505V	Neutral
5	N395K	IEM	34	V194I	Neutral	63	S1509A	Neutral
6	V400M	IEM	35	L201V	Neutral	64	S1509T	Neutral
7	L823R	IEM	36	N206D	Neutral	65	Q1530D	Neutral
8	I848T	IEM	37	T370M	Neutral	66	Q1530K	Neutral
9	L858H	IEM	38	E759D	Neutral	67	Q1530P	Neutral
10	L858F	IEM	39	A766T	Neutral	68	H1531Y	Neutral
11	A863P	IEM	40	A766V	Neutral	69	M1532V	Neutral
12	V872G	IEM	41	I767V	Neutral	70	E1534D	Neutral
13	P1308L	IEM	42	T773S	Neutral	71	Y1537N	Neutral
14	V1316A	IEM	43	V795I	Neutral	72	T1548S	Neutral
15	F1449V	IEM	44	A815S	Neutral	73	H1560C	Neutral
16	W1538R	IEM	45	D890E	Neutral	74	H1560Y	Neutral
17	A1746G	IEM	46	D890V	Neutral	75	V1565I	Neutral
18	V1298D	PEPD	47	T920N	Neutral	76	I1577L	Neutral
19	V1298F	PEPD	48	K1176R	Neutral	77	D1586E	Neutral
20	V1299F	PEPD	49	R1207K	Neutral	78	T1590K	Neutral
21	G1607R	PEPD	50	T1210N	Neutral	79	T1590R	Neutral
22	M1627K	PEPD	51	I1235V	Neutral	80	T1596I	Neutral
23	A1632E	PEPD	52	N1245S	Neutral	81	V1613I	Neutral
24	R185H	SFN	53	L1267V	Neutral	82	D1662A	Neutral
25	I228M	SFN	54	T1398M	Neutral	83	G1674A	Neutral
26	I739V	SFN	55	I1399D	Neutral	84	K1700A	Neutral
27	G856D	SFN	56	D1411N	Neutral			
28	M932L	SFN	57	K1412E	Neutral			

Table 1: List of all the mutations taken into account

The first part of this thesis is a methodological part whose goal is to explore other kernel methods, besides Weisfeiler-Lehmann, to see if they are able to discriminate between mutations related to pain disorders and mutations not involved in alteration of functionality. The results of the exploration are shown in the next section.

4.2 Results

This section shows the results of applying some kernel methods to the 85 RINs of the case study. The results are shown by visualizing the similarity matrices resulting from the graph kernels application to the input data.

Each depicted similarity matrix has rows and columns labeled according with the mutation ids 0 – 84 of Table 1. Accordingly, 0 – 29 correspond to mutations pain-related while 30 – 84 are neutral variants. Each cell (i, j) in a matrix shows the similarity value between the i -th and j -th RINs color-coded so that lighter colors correspond to RINs with high degree of similarity (from yellow to blue). Clearly the main diagonal shows always the lightest color, being the result of the comparison of a graph with itself.

4.2.1 Vertex histogram

The results of the Vertex histogram Graph Kernel are in Figure 23-28. The results denote that there is a clear pattern that distinguish the RINs of pathogenic mutations (ids from 0 to 29 in the similarity matrices), and of the neutral ones (ids from 30 to 84 in the similarity matrices). This pattern is evident for IONIC, HBOND and VDW bonds, while it is unclear in $\pi - \pi$ stacking and π -cation interactions, since those type of non-covalent bonds are less numerous and significant. In the ALL matrix (which represents all the non-covalent bonds), the pattern can be easily seen. In conclusion, it can be understood by the following matrices that the vertices, that represents the amino acids positions in the protein sequence, are fundamental to distinguish the two classes of mutations

taken into account.

Code:

```
def computeKernelVH(graphs):
    print("— computing kernel")
    vh_kernel = GraphKernel(kernel=[{"name": "vertex_
4 histogram"}], normalize=True)
    return vh_kernel.fit_transform(graphs)
```

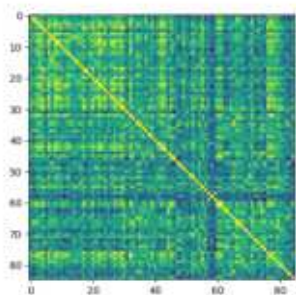


Figure 23: VH on $\pi - \pi$ stacking

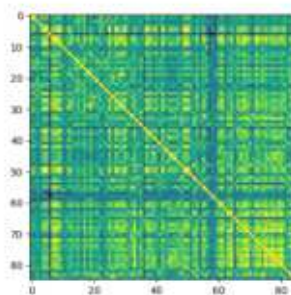


Figure 24: VH on π -cation

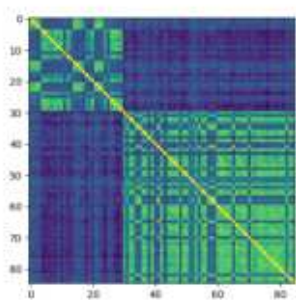


Figure 25: VH on IONIC

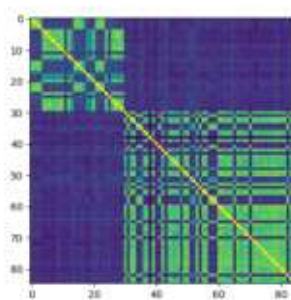


Figure 26: VH on HBOND

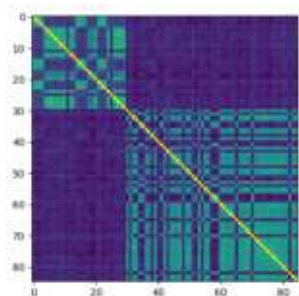


Figure 27: VH on VDW

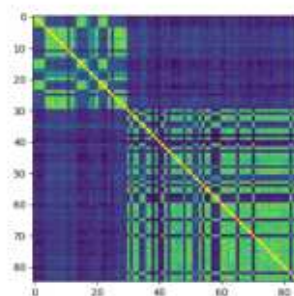


Figure 28: VH on ALL

4.2.2 Edge histogram

The results provided by the Edge histogram Graph Kernel, represented by Figure 29-34, show that edges are not a feature that allows to discern pathogenic mutations (ids from 0 to 29 in the similarity matrices) from the neutral ones (ids from 30 to 84 in the similarity matrices).

Only in the Hydrogen bond similarity matrix (Figure 32) we can notice that all the pathogenic mutations (ids 0-29) turn out to be very similar and to form a separate cluster. Moreover, since the Hydrogen bonds are the most numerous bonds, the cluster can be noticed also in the ALL matrix. **Code:**

```

1 def computeKernelVH(graphs):
    print("— computing kernel")
3     vh_kernel = GraphKernel(kernel=[{"name": "edge_
        histogram"}], normalize=True)
5     return vh_kernel.fit_transform(graphs)

```

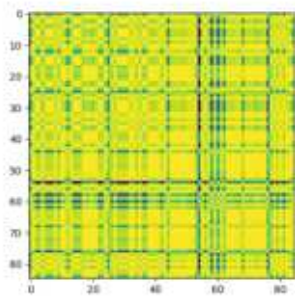
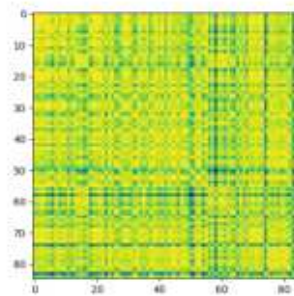
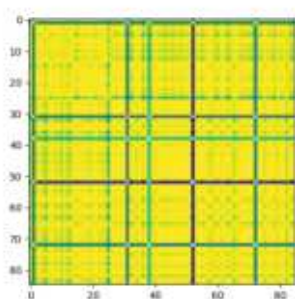
Figure 29: EH on $\pi - \pi$ stackingFigure 30: EH on π -cation

Figure 31: EH on IONIC

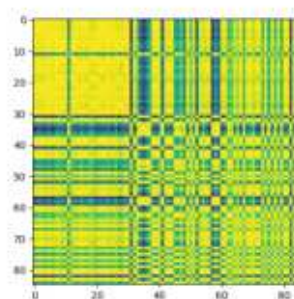


Figure 32: EH on HBOND

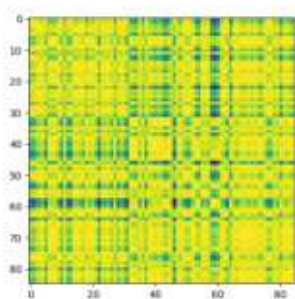


Figure 33: EH on VDW

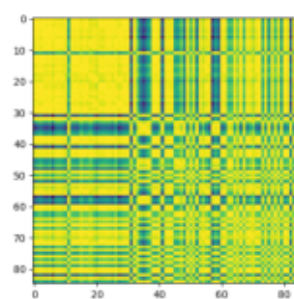


Figure 34: EH on ALL

4.2.3 Random walk

Due to the complexity of this graph kernel it has been possible to compute the algorithm only in the two smallest bonds, $\pi - \pi$ stacking and π -cation.

The results given by this graph kernel gives no interesting information.

Code:

```

1 def computeKernelRW( graphs ):
    print( "— computing kernel" )
3   rw_kernel=GraphKernel( kernel=[{ "name": "random_walk" ,
    "with_labels": True , "lamda" 0.1 }], normalize=True)
5   return  rw_kernel.fit_transform( graphs )

```

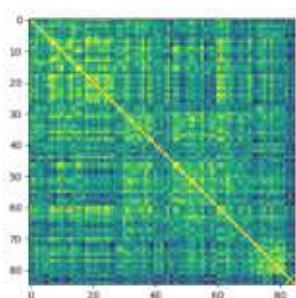


Figure 35: RW on $\pi - \pi$ stacking

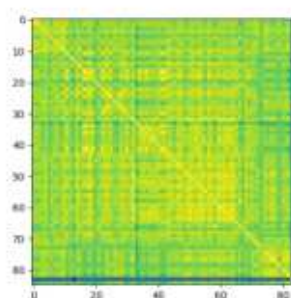


Figure 36: RW on π -cation

4.2.4 Shortest path

As reported for the vertex histogram kernel, the separate bonds giving good results are IONIC, HBOND, VDW while π -cation and $\pi - \pi$ stacking do not show any relevant pattern.

The result provided by ALL 42 is good because IONIC, HBOND and VDW bonds are the more numerous and significant for the mutation.

Code:

```

1 def computeKernelSP(graphs):
    print("— computing kernel")
3     sp_kernel = GraphKernel(kernel=[{"name":
        "shortest_path"}], normalize=True)
5     return sp_kernel.fit_transform(graphs)

```

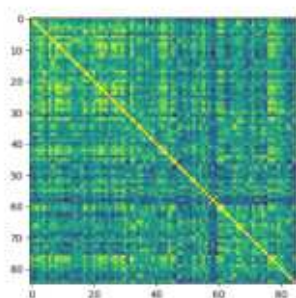
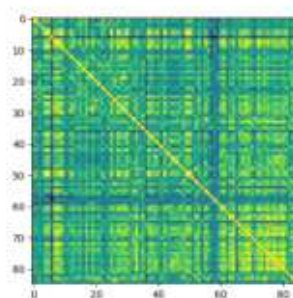
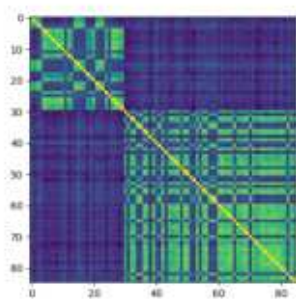
Figure 37: SP on $\pi - \pi$ stackingFigure 38: SP on π -cation

Figure 39: SP on IONIC

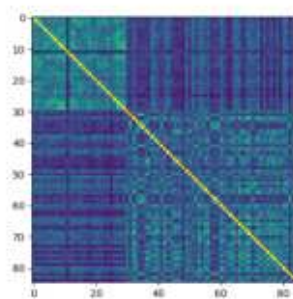


Figure 40: SP on HBOND

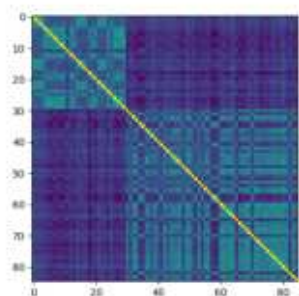


Figure 41: SP on VDW

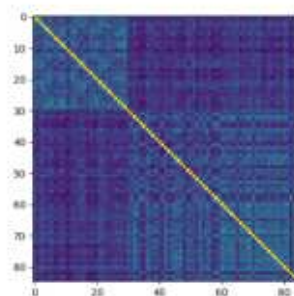


Figure 42: SP on ALL

4.2.5 Graphlet-sampling

The results obtained by this kernel method show that all the mutations taken into consideration share a similar structure. The main problem is that all the mutation are very similar, since they derive from the same protein, the feature taken into consideration by this kernel cannot be distinguished the pathogenic proteins from the not pathogenic ones.

Code:

```

1 def computeKernelGS(graphs):
    print("— computing kernel")
3   gs_kernel = GraphKernel(kernel=[{"name":
    "graphlet_sampling", "sampling":{"n_samples":400}}],
5   normalize=True)
    return gs_kernel.fit_transform(graphs)

```

The size of the graphlet is set by default at 5. The parameters that can be customized are:

- `random_state`: a random number generator instance or an int to initialize a `RandomState` as a seed.
- `k`: the size of the graphlets

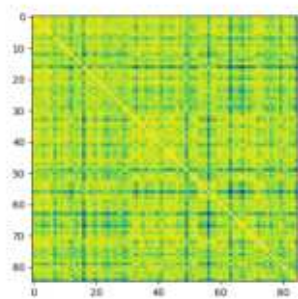
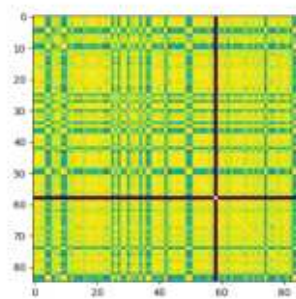
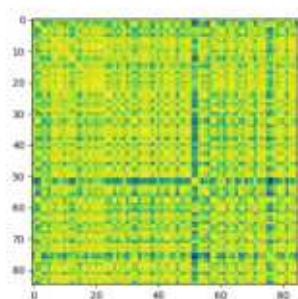
Figure 43: GS on $\pi - \pi$ stackingFigure 44: GS on π -cation

Figure 45: GS on IONIC

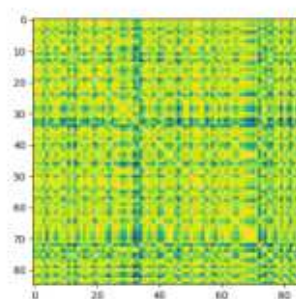


Figure 46: GS on HBOND

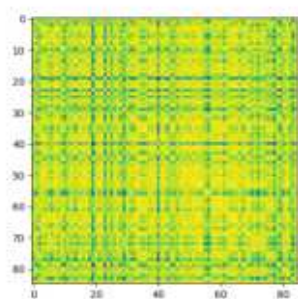


Figure 47: GS on VDW

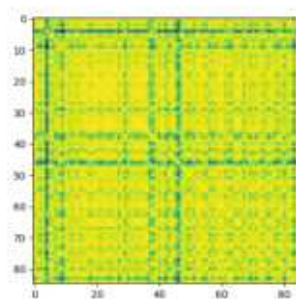


Figure 48: GS on ALL

4.2.6 Weisfeiler-Lehman

This approach has been performed with two different modalities: the first with only one iteration of the WL algorithm the second one with five iterations.

The obtained results clarify that the desired pattern is evident already in the first iteration and is maintained in the subsequent steps.

One iteration

Computing the WL graph kernel with only one iteration gives some important hints. In fact, in the first iteration only the vertices and their direct neighbours are taken into consideration. Hence, the results given by this kernel say that the vertices and their positions in the protein sequences as well as their edges are relevant for discriminating the pathogenic mutations. Note that vertices have already shown to be relevant and edges too in the Hydrogen bond case.

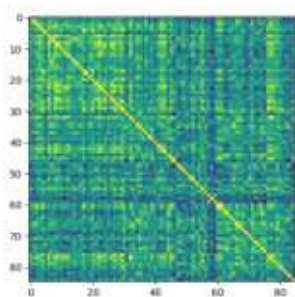


Figure 49: WL on $\pi - \pi$ stacking

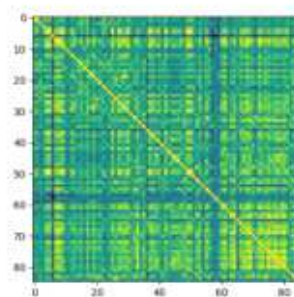


Figure 50: WL on π -cation

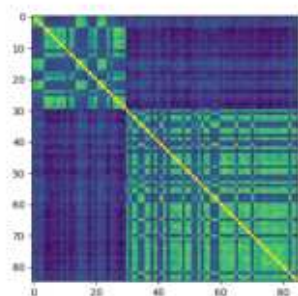


Figure 51: WL on IONIC

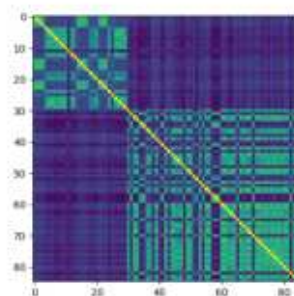


Figure 52: WL on HBOND

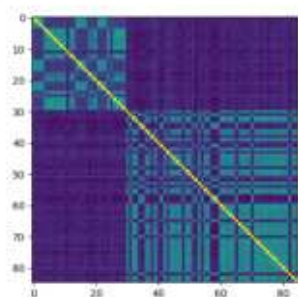


Figure 53: WL on VDW

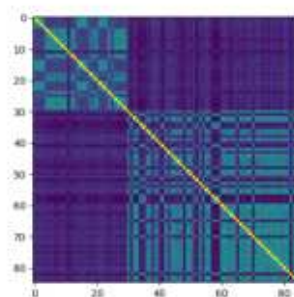


Figure 54: WL on ALL

Five iterations

The WL graph kernel with five iterations has been computed, this means: the depth of the vertex neighbourhood that has been analyzed is equal to five.

The results are very similar to the one given by the same algorithm with only one iteration, the graphs seems to be less similar because iterating it five times, it takes 5 levels of neighbors.

Code:

```

def computeKernel(graphs):
    print("— computing kernel")
    wl_kernel = GraphKernel(kernel=
4     [{"name": "weisfeiler_lehman", "niter": 1},
        {"name": "subtree_wl"}], normalize=True)
6     return wl_kernel.fit_transform(graphs)

```

Where n_iter is the variable that defines the number of iterations.

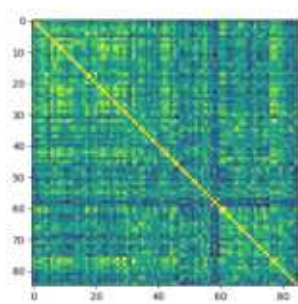
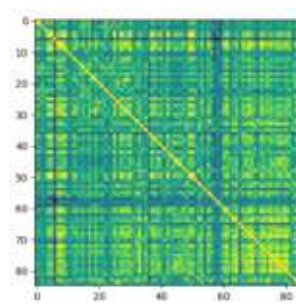
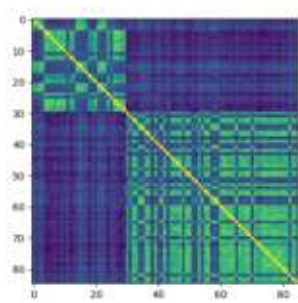
Figure 55: WL on $\pi - \pi$ stackingFigure 56: WL on π -cation

Figure 57: WL on IONIC

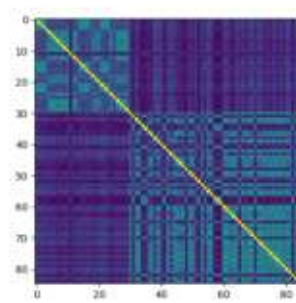


Figure 58: WL on HBOND

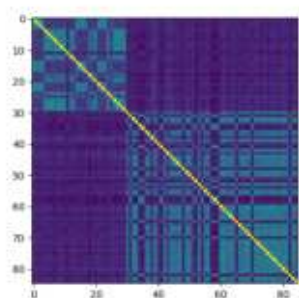


Figure 59: WL on VDW

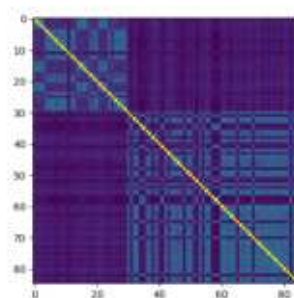


Figure 60: WL on ALL

4.2.7 SVM

As said in 3.3.4, support vector machines so called as SVM is a supervised learning algorithm which can be used for classification and regression problems. We recall that SVM is based on the idea of finding a hyperplane that best separates the features into different domains. When applied to Graph Kernels, their effectiveness depends upon the type of kernel, the kernel parameters and the soft margin.

Code:

```
def runSVM(K, labels):
    print("— computing scores with SVM")
    mod = svm.SVC(kernel='precomputed')
    scores = cross_val_score(mod, K, labels, cv=10)
    return np.mean(scores)
```

In the code shown above the kernel used in the algorithm is by default the rbf (Radial basis function). This kernel is used when the boundaries are hypothesized to be curve-shaped.

RBF kernel uses two main parameters, gamma (Current default is 'auto' which uses $1/n_{features}$) and C (set by default at 1.0) The gamma parameter defines how far the influence of a single training example reaches, with low values

meaning ‘far’ and high values meaning ‘close’. The C parameter trades off correct classification of training examples against maximization of the decision function’s margin. Here below we present the scores of the SVM on the dataset take into consideration.

	$\pi - \pi$ stacking	π - cation	IONIC	HBOND	VDW	ALL
VH	0.7875	0.7028	0.9889	0.9639	0.9889	0.7833
EH	0.6458	0.6458	0.6458	0.6458	0.6458	0.6458
RW	/	0.6458	0.6458	/	/	/
SP	0.8111	0.6903	0.9889	0.9889	0.9889	1
GS	0.6458	0.6458	0.6458	0.6458	0.6458	0.6458
WL1	0.7764	0.7028	0.9889	0.9778	0.9889	0.9889
WL5	0.7875	0.6917	0.9889	0.9889	0.9889	0.9889

Table 2: SVM results

As already shown in the graph kernel results, the best results are given by VH, SP and WL (with one or five iterations). It can be finally said that those are the best kernel methods that should be applied for this case study.

4.3 Conclusion

Thanks to Graph Kernel results, we had the idea of developing an application for the visualization of the three-dimensional structure of a protein that allows the user to:

- display every single non-covalent bond separately (or in couple)
- visually compare two different proteins with the possibility to show and give in output a file that represents the differences between of the two proteins.

The application is called SphereMole and it requires in input the pdb file, representing the three-dimensional structure of the protein and the XML file given in output by RING 2.0, which represents the RINs for every single non-covalent bond of the protein itself.

Chapter 5

SphereMole

In this chapter we illustrate SphereMole, a standalone application that provides a three-dimensional visualization of proteins, with the possibility to highlight or visualize the non-covalent bond. We start by motivating the technological choices and then we provide a functional description, all the application requirement of the application and a detailed illustration of the user interface. Finally, some tests and their results are presented.

5.1 Technological choices

In the literature there are many applications for the visualization of the three-dimensional structure of the proteins, see [9] for a brief survey.

However, our applications requires some features that distinguish it from all the other proposal such as:

- possibility to highlight the amino-acids involved in the considered non-covalent bonds;
- possibility to visualize two proteins at the same time using separate windows, allowing for separate zooming, rotation commands and show the differences between those two considered proteins.

The main criteria driving the technological choices for the development of SphereMole were the following:

- Graphical rendering: we needed a good quality graphical rendering for the three-dimensional structure of the protein
- Portability: we wanted the application to be portable with respect to the main operating system (macOs, Windows and Linux).
- Standalone vs web application: we preferred the stand-alone option.

Under this requirements, we chose the Unity [S7] platform to develop SphereMole. In fact Unity is mainly used for videogames developing and provides an excellent quality also for little details. Unity allows to build the application in various mode: stand-alone, webapp, mobile apps, etc. However, to the complexity of the target objects to be visualized, we expected an heavy application from the graphical/computational viewpoint. In this respect, we thought that the stand-alone option was a good choice.

Unity is a cross-platform graphics engine developed by Unity Technologies that allows the development of videogames and other interactive contents, such as architectural visualizations or 3D animations in real time.

One of its main features is that a standalone application, can be built with a unique project, consisting of scenes, objects and C# scripts, and can run on Windows, macOS and Linux.

On the other side, one weak point of an application developed with Unity could be its weight, since when it is compiled to be executed as standalone app, all the architectural part of Unity is included in the application directory. Moreover, for Unity applications it is recommended to have a device with a dedicated GPU, because the graphical quality guaranteed by Unity could require a lot of computational resources on the graphical side.

5.1.1 Unity scene structure

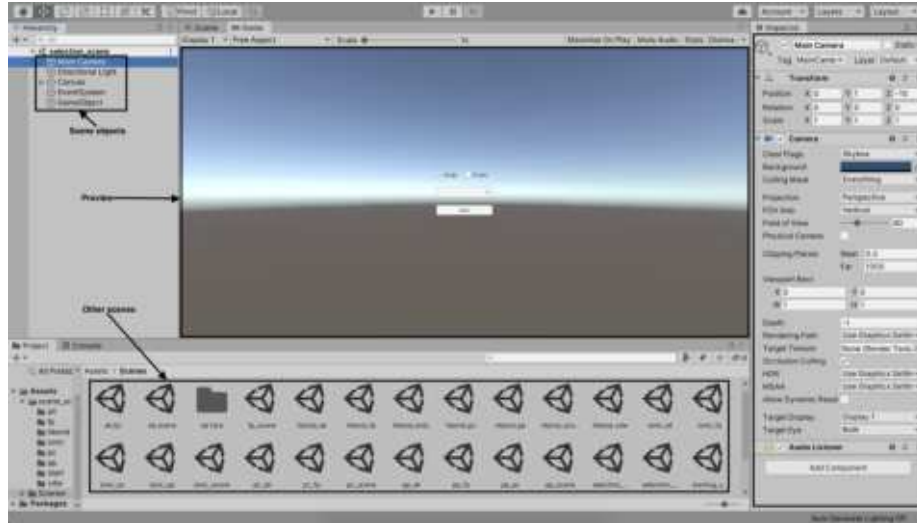


Figure 61: Unity scene example

A scene in Unity contains all the objects that take part of that specific scene. An object, more formally a `GameObject`, can assume different functionalities:

- Camera: it is the object that represents the user point of view.
- Directional light: it represents the position and direction of the light that hits the 3D object visualized.
- User Interface: those are all the interactive objects, such as buttons, toggles that allow the user to interact with the application. All objects must be included (or child) of an object (parent) called `Canvas`, that includes all those objects that belong to this category. Every `Canvas` is equipped with an `EventSystem` that allows to handle all the events that occur.
- Other scopes: A `GameObject` can also act as a controller, that manages all the other objects.

Every `GameObject` can have a script (or more than one) that describes its activity.

5.2 Functional description of the application

The following flow-chart represents the application flow, with all the possible scenarios.

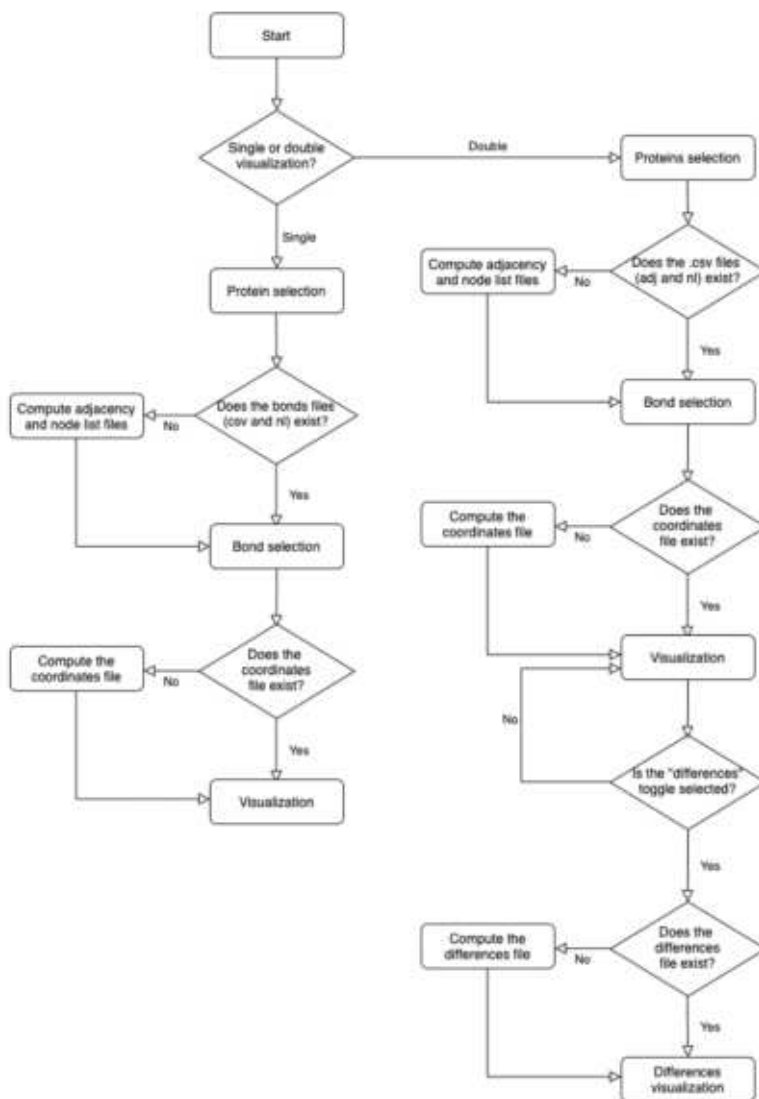


Figure 62: Application flow-chart

The application can visualize every kind of proteins, it requires in input only the PDB file and the XML file given in output by RING2.0.

The application first allows for choosing between single and double visualization. When the protein or the proteins are selected, the application checks if the files containing the RIN adjacency and the node list exist, if not, it parses the file XML given in output by RING2.0 [6] and computes such information creating those two files.

The node list file named `mutationName_nl_bondName` (e.i. `Y1537N_nl_VDW`), contains all the amino acids belonging to that mutation in that bond, identified with their names and their labels. The adjacency file, named `mutationName_adj_bondName` (e.i. `Y1537N_adj_VDW`), contains all the couple of amino acids involved in that mutation belonging to that bond.

After having parsed the XML file given in output by RING2.0 and having created the two files containing the adjacency and the node list, the application checks if the file containing the coordinates of the selected mutation and bond exists. If the file exists, it proceeds to render every single atom belonging to every single amino acid present PDB file, otherwise, the applications automatically creates it. In order to compute the file containing the coordinates, named `mutationName_bondName_coord` (e.i. `G1674A_IONIC_coord`), the application read the amino acid id from the node list file, and it searches for the id in the PDB file that contains all the atoms present in that given mutation, and writes all the coordinates belonging to that amino acid.

When the coordinates file is computed the visualization can start, the application reads the coordinates and render a sphere for every triplet (coordinate x; coordinate y; coordinate z).

The *difference* between two mutations is a feature present only in the double visualization and allows the user to visualize the differences between the two mutations. As for the other features, it requires a computational step if the file containing the differences does not already exist. The computational step consists in finding all the differences in the adjacency files of the two proteins that the user is visualizing. In order to find the different amino acids, the adjacency

files of the two proteins are analyzed for finding the differences. For instance, if in the first file there are the following interactions: $A \rightarrow B$; $B \rightarrow C$ and in the second file there is only the interaction $A \rightarrow B$, the differences file contains node C (because it is not present in the second file) and also node B (because the interaction $B \rightarrow C$ is not present). After computing the differences, the result is displayed.

5.3 Requirements

Two types of requirements are presented in this section: functional and non-functional. By functional requirement, we mean what the system must do, or, in the case of an application, the service that the application provides; instead by non-functional requirement, we mean a property of the system, such as safety or efficiency. For the representation of these requirements, we use a table scheme to make the explanation as clear and compact as possible. All the tables that describe the requirements follow the format of the template shown in Table 3.

ID	Identifier for the requirement
Service Name	Representative name for the requirement
Definition	Brief definition of the requirement
Reason	Why the requirement has been identified
Influences	Related and influenced requirement ID
Dependence	Requirement ID on which it depends

Table 3: Example of requirement table

5.3.1 Functional requirements

ID	SD_SELECTION
Service Name	Single or double representation menu
Definition	Selection of the single or double representation by the user.
Reason	Since there are two possible representation of the three-dimensional protein structure, this menu enable the user to choose between those two representation. This selection can change the menu structure and the representation of the protein. If the single representation is chosen, there is only one protein to choose and there will be only one protein represented otherwise there is a double choice for the proteins and the two proteins chosen will be represented one next to the other.
Influences	PROTEIN_SELECTION
Dependence	-

Table 4: Single or double representation selection

ID	PROTEIN_SELECTION
Service Name	Protein selection
Definition	Selection of the protein (or proteins) to be visualized
Reason	The selection of the protein is done in order to visualize the selected protein or proteins, in order to visualize the three-dimensional representation, since the one (or more) chosen will be visualized in their shape.
Influences	MOVEMENT, ROTATION, BOND_SELECTION, DIFFERENCES, BACK_TO_SELECTION
Dependence	SD_SELECTION

Table 5: Protein selection

ID	BOND_SELECTION
Service Name	Non-covalent bond selection
Definition	Selection of the non-covalent bond to be visualized
Reason	The bond selection is performed to visualize the selected bond on proteins. In the single visualization it is also possible to visualize two bonds at the same time, in order to see how they interact
Influence	MOVEMENT, ROTATION, DIFFERENCES
Dependence	SD_SELECTION, PROTEIN_SELECTION

Table 6: Non-covalent bond selection

ID	BACK_TO_SELECTION
Service Name	Back to protein selection menu
Definition	Button that allows to go back to the protein selection
Reason	Clicking this button allows the user to go back to the protein selection
Influence	MOVEMENT, ROTATION, DIFFERENCES
Dependence	SD_SELECTION, PROTEIN_SELECTION

Table 7: Return to the protein selection button

ID	MOVEMENT
Service Name	Movement commands in protein visualization
Definition	Movement commands that allows the user to explore the protein structure
Reason	Using the buttons WASD or the arrows of the keyboard the user is allowed to explore the whole protein structure and to turn the camera dragging the mouse
Influence	-
Dependence	SD_SELECTION, PROTEIN_SELECTION, BOND_SELECTION

Table 8: Set of commands for exploring the protein

ID	ROTATION
Service Name	Button set that allows the rotation of the protein
Definition	Button set that allows the user to rotate the protein in order to visualize the proteins from all possible angles
Reason	Using this functionality the user can easily compare two given proteins rotating them in all possible ways and it gives a complete visualization of the protein or of the selected bond. If the double visualization is chosen, there will be two button sets for the rotation, one for each protein
Influence	-
Dependence	SD_SELECTION, PROTEIN_SELECTION, BOND_SELECTION

Table 9: Protein rotation functionality

ID	DIFFERENCES
Service Name	Differences between protein
Definition	Functionality that allows to visualize the differences between two given proteins
Reason	Using this functionality the application highlights the difference between two proteins selected in the proteins selection phase. It is possible to highlight the single bond differences. This functionality is available only for the double visualization.
Influence	-
Dependence	SD_SELECTION, PROTEIN_SELECTION, BOND_SELECTION

Table 10: Difference between two proteins

5.3.2 Non-functional requirements

ID	DIR_ACC
Service Name	Directory access permission
Definition	The user has to give the permission to access the application directory, that contains the essential files.
Reason	Without this permission, the application will not work.
Influences	Correct execution of the application
Dependence	-

Table 11: User permission to access the application directory.

5.3.3 Device requirements

The recommended requirements for running SphereMole are the following:

- An operating system such as Windows, MacOS or Linux
- At least 8 GB of RAM (16 are recommended)
- A dedicated graphic card (optional)
- A modern processor, multi-core, that allows hyper-threading.

5.4 User interface

The application starts by allowing the user to choose the protein to visualize. This part of the application is characterized by two toggles that allows to pass from the single to the double choice, one dropdown menu which contains all the proteins IDs and a button that starts the visualization; if the user chooses to visualize and analyze two proteins together, there will be two dropdown menus that will allow to choose the two proteins to be considered.

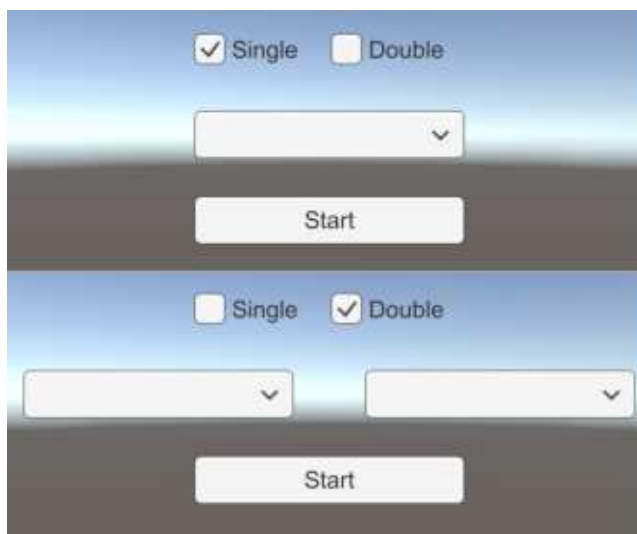


Figure 63: Protein selection phase

If the single protein visualization was chosen, the application passes to the bond selection phase in which, the user chooses the non-covalent bond that wants to visualize and analyze. The bonds that can be visualized are the following: hydrogen bond, Van der Waals interactions, ionic bond, $\pi - \pi$ stacking, π cation, all the non-covalent bonds together (under the name of ALL) and the full protein.

This phase is mostly the same for the single or double visualization, with the exception that in the double visualization and there will be the two proteins IDs visualized instead of one ID in the single visualization.

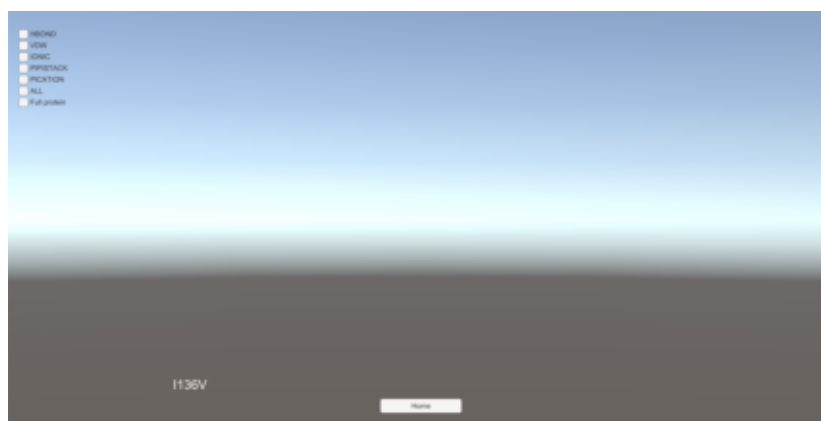


Figure 64: Bond selection phase

The next phase, after the bond selection is the visualization of the selected bond. Each bond has an assigned color in its three-dimensional visualization. In this phase, the user can explore the bond that is shown using the movement commands, that has been assigned to the arrows and the WASD keyboard buttons; the user can also move its point of view by dragging the mouse. Another action that can be performed by the user is turning the protein with the group of buttons that are placed at the bottom left of the screen. Those buttons allow to rotate the protein in the direction shown by every single button.

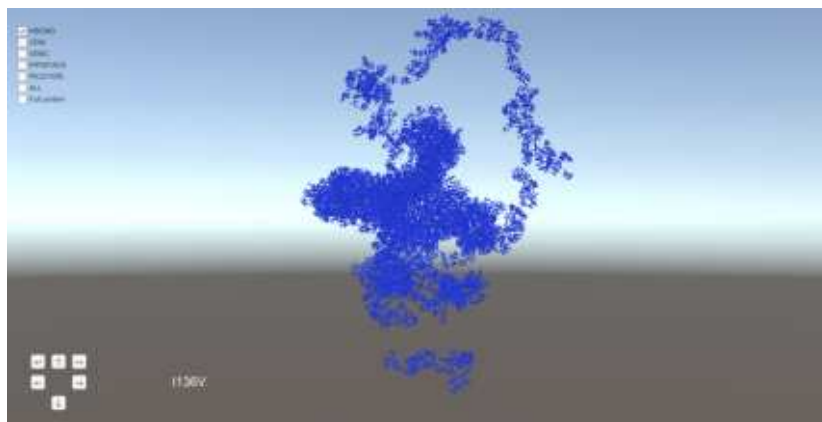


Figure 65: Single bond visualization

Starting from the bond visualization, the user can choose to see how two bonds coexist in the same protein, or how that bond is placed with respect to all the non-covalent bonds or with respect to the entire protein. This type of visualization can be performed by just clicking one of the other possible toggles present in the bond visualization phase.

All the actions previously described are possible also in this type of visualization.

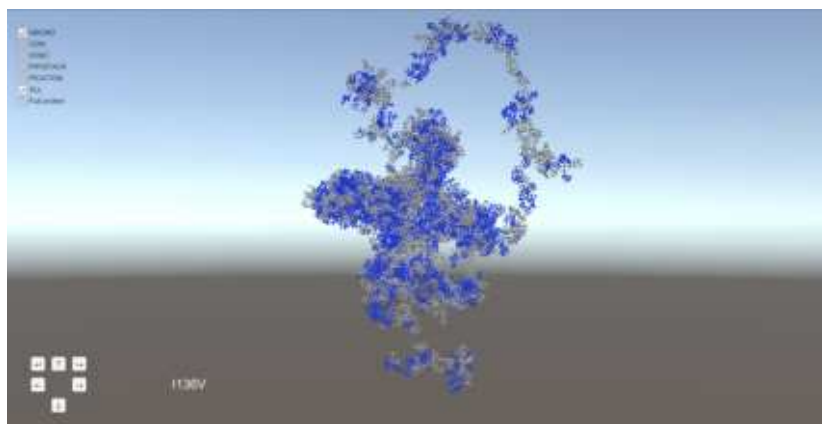


Figure 66: Two bonds visualization

If the user had previously chosen the double visualization, the application will display the bond selection phase for the two selected proteins.



Figure 67: Bond selection in double protein visualization

Once the bond is selected, the application provides its three-dimensional representation, for the two selected proteins. In this phase, there are two distinct group of buttons, one for each protein, that allows to rotate the single protein as the user wants.

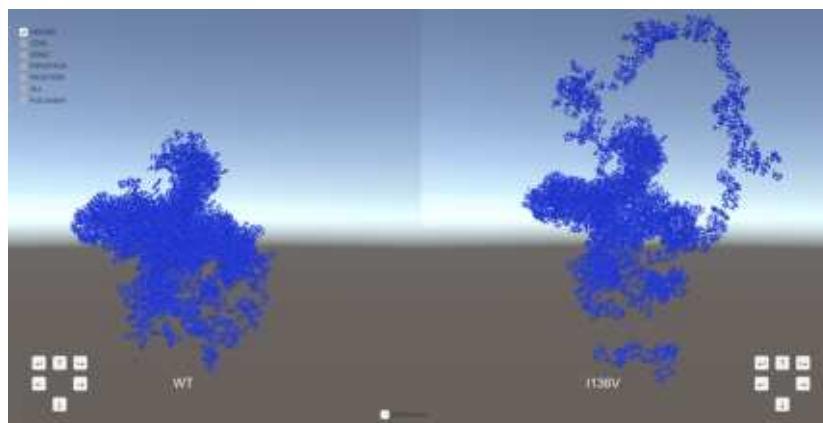


Figure 68: Double protein visualization

One additional functionality with respect to the single visualization, is the possibility to show the difference between the two proteins that the user is analyzing. By just clicking the difference toggle, the application provides the visualization of the difference between the two proteins, highlighting it in a

different color.

Figure 69 shows a comparison between the Hydrogen bonds of the two visualized proteins. On the left, the Hydrogen bonds of the protein that are not shared with the protein on the right are shown in yellow. Similarly, on the right, the Hydrogen bonds that are not shared with the protein on the left are shown in yellow.

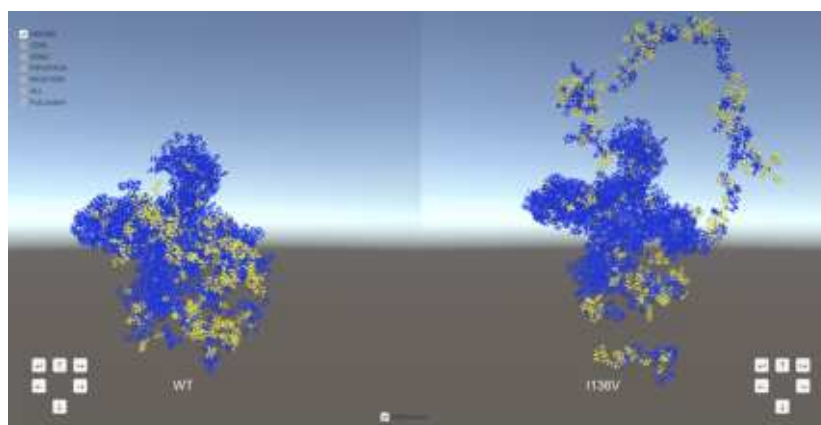


Figure 69: Double protein visualization with differences

5.5 Tests

Two types of tests have been performed on SphereMole, the first is a visual comparison with another well known program for three-dimensional protein viewer called UCSF Chimera [S6] while the second one are some timing tests on two different computer configurations.

5.5.1 Comparison with Chimera

We tested if SphereMole visualizes correctly the protein three-dimensional structure by visually comparing it with Chimera, a program that provides similar functionalities.

UCSF Chimera is a well known and widely used program for the interactive visualization and analysis of molecular structures and related data, including density maps, trajectories, and sequence alignments [S6].

Several proteins have been tried with both applications and all the results were visually almost identical. As shown in Figure 70, on the right it is represented the result with SphereMole and on the left the result with Chimera and the two images are clearly very similar.

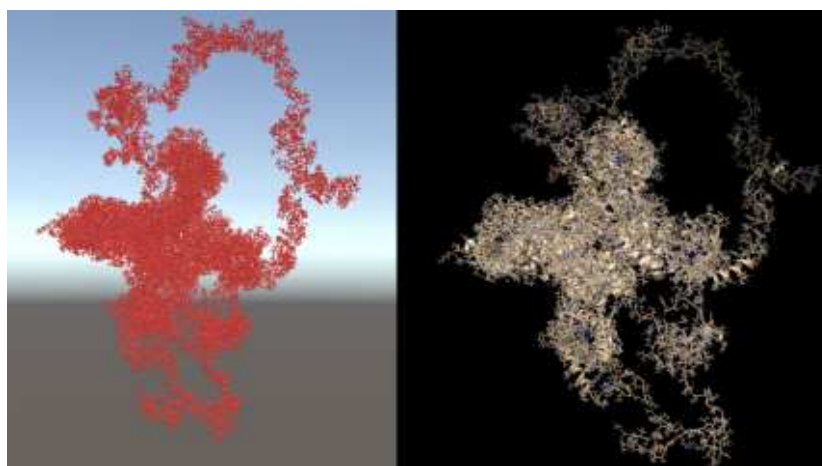


Figure 70: Comparison between SphereMole and Chimera: the same protein is visualized with the two tools

5.5.2 Rendering time tests

The other tests performed on SphereMole measures the rendering time of protein structure. This test has been performed for all the proteins structure of the case of study, with the following two computer configuration:

Computer configuration 1	
O.S	macOS Catalina
Processor	Intel Core i7 (dual-core@3,3 GHz)
RAM	16 GB DDR4
GPU	Intel Iris Graphics 550, 1536 MB (integrated)
Storage	512 GB SSD
Computer configuration 2	
O.S	Windows 10
Processor	Intel Core i7 (quad-core@3,5 GHz)
RAM	16 GB DDR4
GPU	Nvidia GeForce GTX 960, 4 GB (dedicated)
Storage	256 GB SSD + 1 TB HD

Table 12: Devices configuration

The obtained results by those tests are reported below with the average time and the standard deviation, obtained by 50 tests.

The standard deviation, identified with the letter σ , is a measure of how spread out numbers are. It can be calculated in with the following formula:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

where N represents the number of samples, x_i is the value of the sample and μ represents the mean value.

Results configuration 1		
Single visualization		
BOND	Mean (ms)	Standard deviation (ms)
HBOND	1086	157
VDW	1016	166
IONIC	615	47
$\pi - \pi$ stack	312	33
π -cation	279	31
ALL	1330	175
Full protein	1515	168
Double visualization		
BOND	Mean (ms)	Standard deviation (ms)
HBOND	3418	178
VDW	2489	183
IONIC	720	50
$\pi - \pi$ stack	390	29
π -cation	412	22
ALL	3476	199
Full protein	4523	153
Differences visualization		
BOND	Mean (ms)	Standard deviation (ms)
HBOND	2627	112
VDW	2290	89
IONIC	673	45
$\pi - \pi$ stack	276	40
π -cation	377	12
ALL	2899	88

Table 13: Rendering time results with the first configuration

Results configuration 2		
Single visualization		
BOND	Mean (ms)	Standard deviation (ms)
HBOND	989	112
VDW	947	137
IONIC	660	69
$\pi - \pi$ stack	355	17
π -cation	283	36
ALL	1217	132
Full protein	1279	155
Double visualization		
BOND	Mean (ms)	Standard deviation (ms)
HBOND	2889	182
VDW	2237	116
IONIC	733	27
$\pi - \pi$ stack	333	15
π -cation	374	12
ALL	3122	128
Full protein	3456	89
Differences visualization		
BOND	Mean (ms)	Standard deviation (ms)
HBOND	1745	97
VDW	1623	23
IONIC	559	37
$\pi - \pi$ stack	222	15
π -cation	210	17
ALL	2773	91

Table 14: Rendering time results with the second configuration

The analyzed proteins of the case of study are very complex, and we can say that the timing test analyzes the worst case, since the proteins have about 2000 amino acids and 30000 atoms.

In order to achieve better performances, it has been followed the guide provided by [S8], that suggests to build scenes as light as possible, obtaining a project that is fragmented in more subunits all linked together.

The different device configuration does not significantly influence the rendering time, that seems to be more stable (with a smaller standard deviation) in the second configuration.

Chapter 6

Conclusion and future work

This thesis, inspired by the pipeline built in [5], explores some Graph Kernels in order to better define the feature that discern pathogenic mutations from the neutral ones. The Graph Kernels have been tested over a set of 85 mutation, represented as RINs, in which 30 of them were recognized to be related to neuropathies, the other 55 to be neutral. Over this set of mutations we tested the following Graph Kernels: Vertex histogram, Edge histogram, Shortest path, Graphlet-sampling and Weisfeiler-Lehman. Among them, Vertex histogram, Shortest path and Weisfeiler-Lehman Graph Kernel methods were able to distinguish pathogenic mutation from neutral ones. The ability of Graph Kernels and, more in general, the importance of non-covalent bonds, suggested us the idea of a visualization tool able to show the three-dimensional structure of the protein and of its non-covalent bonds. Hence, we developed SphereMole, an application that can highlight the nodes of the RIN (i.e. the amino acids in non-covalent bonds) back to the three-dimensional protein structure.

SphereMole is a standalone application developed with Unity as development platform, that runs in the main operating systems systems such as macOS, Windows and Linux. The application allows the user to visualize the protein and the following non-covalent bonds: Hydrogen bond, Van der Waals interactions, Ionic bonds, $\pi - \pi$ stacking, π -cation and all the previous bonds together. The

application allows also the split view, comparing two different proteins and also to highlight the different amino acids between two visualized proteins.

Hence, the pipeline built in [5], can be now further extended with the protein visualization and comparison with SphereMole. We conclude by delineating some possible future developments of SphereMole. First of all, some graphical features can be introduced to improve the user experience. For instance, when an user hovers an atom with the pointer, the application could show the information about the atom.

Moreover, at the moment, the application is based on an atom view of the protein structure, but also the secondary structure view (in terms of α -helices, β -sheets and loops) could be proposed. We believe that this kind of visualization would greatly improve the application performance.

Finally, Unity allows for easily integrate the Virtual Reality of the visualized object. It would be interesting, for a team of experts, to explore proteins using the Virtual Reality.

Bibliography

- [1] B.Anderson, *Pattern Recognition: An introduction*, ED-Tech Press, Waltham Abbey 2019
- [2] C. B. Anfinsen, *Principles that govern the folding of protein chains*, “Science”, 181 ,1973, pp. 223-230.
- [3] J. Berg, J. L. Tymoczko, L. Stryer, *Biochemistry*, W. H. Freeman, New York 2006.
- [4] N. A. Campbell, J. B. Reece, *Principi di biologia*, Pearson Benjamin Cummings, London 2010.
- [5] A. Giacometti, M.Simeoni, A.Toffano, *A network topology approach to the relation between painful disorders and mutations in sodium channel proteins*, ECLT, Venice 2019.
- [6] K. Grewal, S. Roy, *Modeling Proteins as Residue Interaction Networks*, “Protein and Peptide Letters”, Number 10, 2015, pp. 923-933.
- [7] Lodish H, et al., *Molecular Cell Biology. 4th edition*, W. H. Freeman, New York 2000.
- [8] M. Marchi, *Genetic and pain*, Seminar slides, Fondazione IRCCS Istituto Neurologico “Carlo Besta”, Milan 2019.
- [9] N. P. Matrignon, *PDBjs, un tool per la visualizzazione 3D di proteine*, Tesi triennale in Informatica, Università Ca’ Foscari, Venezia 2020.

-
- [10] G. Nikolentzos, G. Siglidis, M. Vazirgiannis, *Graph Kernels: A Survey*, Cornell University, New York 2019.
 - [11] D. Piovesan, G. Minervini and S. C. E. Tosatto, *The RING 2.0 web server for high quality residue interaction networks*, “Nucleic Acids Research”, 44, 2016, pp. 367-374.
 - [12] S.V.N. Vishwanathan, et al., *Graph Kernels*. “The Journal of Machine Learning Research”, 11, 2010, pp. 1201–1242.

Sitography

- [S1] Berkeley, University of California. *Types of mutations and their impact on protein function*, 2019
http://mcb.berkeley.edu/courses/mcb142/lecture%20topics/Dernburg/Lecture6_Chapter8_forprinting.pdf [02/07/2020]
- [S2] R. Berwick, *An Idiot's guide to Support vector machines (SVMs)*, 2019
<http://web.mit.edu/6.034/wwwbob/svm-notes-long-08.pdf>
[02/07/2020]
- [S3] De Agostini, *La sintesi proteica*,
<http://www.sapere.it/sapere/strumenti/studiafacile/biologia/La-cellula/Il-linguaggio-della-cellula/La-sintesi-proteica.html>
[02/07/2020]
- [S4] G. Nikolentzos, G. Siglidis, M. Vazirgiannis, *GraKel*,
<https://ysig.github.io/GraKeL/dev/index.html> [02/07/2020]
- [S5] J. Hunter, [et. al], *Matplotlib*,
<https://matplotlib.org/> [02/07/2020]
- [S6] University of California, *Chimera*,
<https://www.cgl.ucsf.edu/chimera/> [02/07/2020]
- [S7] Unity, *Official Website*,
<https://unity.com/> [02/07/2020]

- [S8] Unity, *Optimizing graphics performance*,
<https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html>
[02/07/2020]
- [S9] wwPDB, *PDB*
<http://www.wwpdb.org/> [02/07/2020]
- [S10] G. van Rossum, *Python*,
<https://www.python.org/> [02/07/2020]
- [S11] D. Cournapeau, *Scikit*,
<https://scikitlearn.org> [02/07/2020]
- [S12] D. Cournapeau, *Scikit learn, Cross-validation: evaluating estimator performance*,
https://scikitlearn.org/stable/modules/cross_validation.html
[02/07/2020]