Fakultät Mathematik und Naturwissenschaften, Fachbereich Informatik

Thema

Topology-based amino acid recognition

Report on the Research Internship: Frontiers in Applied Drug Design

Supervised by:
Herr Dr. Markus Zimmermann

Mykyta Borodin
mykyta.borodin@student.uni-tuebingen.de
Matrikelnummer: 7056316

Inhalt

1. Introduction

Proteins are responsible for most cellular functions such as catalyzing biochemical reactions, transmitting signals and providing structural support. However, when structures are processed through quantum mechanical calculations, homology modelling or file format conversions (for example from PDB to SDF) residue identifiers can be lost. This leaves only unlabelled atoms and bond connections, making it essential to recover the original amino acid sequence for downstream tasks like validating homology models, setting up molecular simulations or interpreting experimental data.

Each amino acid contains a conserved backbone motif NH Cα C=O while its unique side chain provides the chemical signature that distinguishes it from other residues. In practice, experimental structures often omit certain atoms such as hydrogens and may include variable protonation states or covalent modifications (for example phosphorylation or methylation). These factors can break simple pattern matching rules, so a robust algorithm must tolerate incomplete or noisy data. Ideally the method assigns a probability to each possible amino acid based on observed topology and selects the most likely candidate.

During this six week internship two software pipelines were developed to perform topology based amino acid recognition:

Version 1 uses a streamlined pattern matching approach. It locates the peptide backbone by detecting NH Cα single bonds, Cα C single bonds and C=O double bonds among consecutive atom indices. Non backbone atoms are grouped into side chain collections and simple bond count heuristics classify common residues such as glycine, alanine, valine, isoleucine and threonine. This version is optimized for speed and is well suited to rapid analyses of small or well resolved peptides.

Version 2 extends this methodology with recursive bond order aware scoring and full annotation. After backbone identification it traverses each side chain step by step, computing detailed counts of carbon, oxygen, nitrogen and sulfur bonds to distinguish all twenty canonical amino acids, including aromatic and long aliphatic residues. It then translates residue names into standard three letter codes (for example ALA or PHE) and produces an annotated SDF file by appending these codes to each atom record. This output can be directly used in simulation or analysis pipelines.

The goals of this project were to demonstrate that atom and bond topology alone is sufficient to reconstruct peptide sequences and to deliver two implementations tailored to different needs – one for rapid prototyping and another for comprehensive annotation.

2. Materials and Data

2.1. Software Environment and Libraries

The project was developed using Python 3.9, with a combination of standard library modules and external packages that support graph-based molecular analysis.

The following libraries were used:

- **`itertools`** — from the Python standard library, used for efficient iteration and slicing of dictionaries and lists (e.g. **`islice()`** for limiting output during debugging).
- **`networkx`** — used to represent the molecule as a labeled graph:
  - Each atom becomes a node, labeled by element type and atom index (e.g. `"C 12"`).
  - Each bond is modeled as an edge with a **`weight`** attribute indicating bond type.
  - Graph operations such as neighbor traversal, subgraph extraction (e.g. side chains), and backbone path detection are implemented using standard **`networkx`** methods.
- **`grakel`** — a graph kernel library used for structure comparison:
  - Each known amino acid side chain is represented as a graph template.
  - The side chain extracted from a molecule is compared to these templates using graph kernels such as Weisfeiler-Lehman or Shortest-Path Kernel.
  - The result is a similarity score for each amino acid, allowing probabilistic assignment based on topology.
- Built-in Python functions for file I/O and string operations — used to parse the V3000 SDF format, extract atoms and bonds, and generate outputs.

The scripts were developed and tested on a Windows 10 machine using Python 3.9.7.

2.2. File Format Details

The molecular structures used in this project are provided in SDF (Structure-Data File) format with V3000 annotations. This format defines atoms and bonds in clearly marked blocks:

- Atom block (**`M V30 BEGIN ATOM ... END ATOM`**) lists each atom with its index and element symbol.
- Bond block (**`M V30 BEGIN BOND ... END BOND`**) describes connections between atoms, including bond type (e.g. single or double) and the indices of the connected atoms.

During parsing, only the element type, atom index, and bond information are extracted. Each atom is linked to its neighbors and the corresponding bond types. Atoms without bonds (e.g. isolated solvent atoms) are excluded from further analysis.

This extracted topology is used as the input for all subsequent steps in the recognition pipeline, including backbone detection, side-chain grouping, and amino acid classification.

3. Methodology and Implementation

Two software pipelines were developed to perform topology-based amino acid recognition from molecular structure data in SDF format. The key challenge is to reconstruct amino acid sequences using only atomic connectivity and element types, despite often incomplete or noisy data caused by missing atoms, variable protonation states, or experimental limitations. Each pipeline employs a distinct approach: one uses heuristic pattern matching for fast residue classification, while the other applies graph-theoretical methods for detailed and robust identification of all standard amino acids.

3.1. Parsing the Molecular Structure

Both versions start by reading molecular structure data from SDF files in the V3000 format. The parsing logic is similar, but the way parsed data is stored differs significantly.

Version 1 reads the file line-by-line and extracts atoms and bonds into a dictionary `atoms_with_bonds`. The keys are strings combining the element symbol and atom index (e.g., `"C 12"`). Each key maps to a list of bonded neighbors represented as strings containing the bonded atom type, atom ID, and bond type. This structure supports direct neighbor lookups required for pattern matching.

Version 2 parses the atoms and bonds to build a graph `G` using the `NetworkX` library. Each atom is a node with an `atom` attribute indicating the element type. Bonds are added as edges with a `bond_type` attribute storing the bond order (e.g., `'1'` for single bond). This graph structure facilitates recursive traversal and graph-based isomorphism comparisons.

Both versions share a similar logic for identifying atom and bond sections in the SDF file. A simplified structure of this logic is shown below:

```
for line in raw_data:
    if line.startswith("M  V30 BEGIN ATOM"):
        atom_section = True
        continue
    if line.startswith("M  V30 END ATOM"):
        atom_section = False
        continue
    if line.startswith("M  V30 BEGIN BOND"):
        bond_section = True
        continue
    if line.startswith("M  V30 END BOND"):
        bond_section = False
        continue
    if atom_section and line.startswith("M  V30 "):
        # process atom line
    if bond_section and line.startswith("M  V30 "):
        # process bond line
```

3.2. Backbone Detection

The identification of the peptide backbone (N-Cα-C=O) is a crucial first step in recognizing amino acid residues within a molecular structure. Both versions of the software pipeline implement logic to detect this conserved motif, albeit with different approaches reflecting their overall design philosophy.

Version 1: Pattern Matching Approach

Version 1 employs a streamlined pattern matching technique that leverages the sequential nature of atom indices often found in standard molecular file formats like the input SDF. The **get_atoms_and_bond** function parses the SDF and stores the connectivity in a dictionary (**atoms_with_bonds**), where keys are atom identifiers (Element + ID) and values are lists of bonded neighbors with their bond types.

The core backbone detection logic resides in the **check_NCCO** and **all_NCCO** functions. **check_NCCO** attempts to identify an N-Cα-C=O sequence starting from a given atom. It checks for specific bond types (single N-C, single C-C, double C=O) and relies on the assumption that backbone atoms appear with consecutive or near-consecutive atom IDs in the input file. For instance, it looks for a Carbon atom with an ID one greater than the current Nitrogen atom's ID, and subsequently a Carbon atom with an ID two greater, and an Oxygen atom with an ID three greater. While efficient for well-ordered data, this dependency on atom ID sequence makes it potentially fragile to variations in file generation or processing that might reorder atoms. **all_NCCO** simply iterates through all atoms to find all occurrences of this pattern. The **last_atom** function helps in determining the end of the peptide chain by finding the last singly-bonded oxygen atom in the backbone.

```
# Simplified Version 1 Backbone Detection

def get_atoms_and_bond(raw_data):
    # Parses SDF to create a dictionary of atoms and their bonded neighbors
    # Key: "Element ID" (e.g., "C 12")
    # Value: List of strings describing neighbors and bond types (e.g., ["N id: 11 Type of bond: 1", ...])
    atoms_with_bonds = {}
    # ... parsing logic ...
    return atoms_with_bonds
```

```
def check_NCCO(atom, atoms_with_bonds):
    # Checks if a given atom is the start of an N-C-C=O backbone pattern
    # Relies on checking for specific bond types and potentially sequential atom IDs
    # Returns a list of atoms in the pattern if found, otherwise None
    # ... pattern checking logic ...
    return result # or None

def all_NCCO(atoms_with_bonds):
    # Iterates through all atoms to find all NCCO backbone patterns
    # Returns a list of all atoms identified as part of a backbone
    nccos = []
    # ... iteration and calling check_NCCO ...
    return nccos

def last_atom(atom_bonds, backbone):
    # Finds the last oxygen atom in the backbone with a single bond
    # Used to determine the end of the peptide chain
    # ... logic to find the last single-bonded oxygen in the backbone ...
    return last_oxygen_atom # or None
```

Version 2: Graph-Based Approach

Version 2 adopts a more robust graph-based approach using the NetworkX library. The **get_nodes_and_edges** function builds a graph representation of the molecule where atoms are nodes (with element type as an attribute) and bonds are edges (with bond type as an attribute).

The **find_all_backbones_and_C_alpha** function identifies the backbone by traversing this graph structure. It iterates through all nodes and, upon finding a Nitrogen atom, explores its neighbors. It then checks if a neighbor is a Carbon connected by a single bond (the potential Cα), and from there checks if that Carbon is connected to another Carbon by a single bond (the potential carbonyl Carbon), and finally if that carbonyl Carbon is connected to an Oxygen by a double bond. This method relies purely on the connectivity and bond types within the graph, making it independent of the atom ordering in the input file and thus more resilient to variations in the SDF structure. This function also explicitly identifies the Cα atoms, which are crucial for delineating the start of each amino acid's side chain.

```
def get_nodes_and_edges(data):
    # Parses SDF data to create a NetworkX graph
    # Nodes represent atoms with 'atom' attribute (element type)
    # Edges represent bonds with 'bond_type' attribute (bond order)
    G = nx.Graph()
    # ... parsing and graph building logic ...
    return G

def find_all_backbones_and_C_alpha(G):
    # Identifies peptide backbone atoms (N-C-C=O) and C-alpha atoms in the graph
    # Traverses the graph to find the specific bond pattern
    # Returns two lists: backbone atoms and C-alpha atoms
    backbones = []
    c_alphas = []
    # ... graph traversal and pattern matching logic ...
    return backbones, c_alphas
```

3.3. Side Chain Analysis and Amino acid Identification

Once the peptide backbone is identified, the remaining atoms are typically part of the amino acid side chains. The process of analyzing these side chains to determine the specific amino acid residue differs significantly between the two versions.

Version 1: Heuristic Pattern Matching

In Version 1, the `side_chains` function separates atoms not identified as part of the backbone into distinct groups, each corresponding to a potential side chain attached to a Cα atom. The identification of the amino acid type is then performed by the `side_chain_To_AC` function using a series of heuristic checks based on bond counts and types within each side chain group.

The `find_first_after_C_alpha` function locates the atom directly bonded to the Cα within the side chain. The `score_for_bonds` function calculates the number of bonds to Carbon, Oxygen, Nitrogen, and Sulfur atoms and records the types of these bonds for a given atom. The `one_step` function facilitates traversing the side chain one bond away from a given atom.

The `side_chain_To_AC` function uses a cascade of `if` and `elif` statements to identify common amino acids. For example, if the side chain consists of only one atom (the Cα) and has a single bond to a Carbon, it's identified as Alanine. If it has a single bond to a Carbon which is then singly bonded to an Oxygen, it might be Serine. This approach is essentially a decision tree based on simple topological features and works well for smaller, less complex side chains. However, as side chains become larger or more complex (e.g., aromatic rings, multiple branches), the number of specific patterns to check increases, and this heuristic method can become less comprehensive and more prone to errors if the exact pattern is not explicitly coded. The `translate_AC_to_name` function is used to convert the identified amino acid names to their standard three-letter codes.

```
def side_chains(atoms_with_bonds, backbone):
    # Separates non-backbone atoms into groups corresponding to side chains
    # Returns a dictionary where keys are group IDs and values are dictionaries of atoms and bonds in that group
    groups = {}
    # ... logic to group side chain atoms ...
    return groups

def translate_AC_to_name(AC):
    # Translates long amino acid names to three-letter codes
    # ... dictionary lookup ...
    return three_letter_code

def side_chain_To_AC(group, backbone):
    # Identifies the amino acid type based on the side chain structure using heuristics
    # Checks bond counts and types of atoms in the side chain
    # Returns the name of the identified amino acid (e.g., "Alanine") or "Unknown"
    # ... heuristic checks based on bond scores and structure ...
    return amino_acid_name
```

```
def find_first_after_C_alpha(bonds, backbone):
    # Finds the first atom in the side chain directly bonded to the C-alpha
    # ... logic to find the starting atom of the side chain ...
    return first_atom # or None

def score_for_bonds(atom, group):
    # Calculates the number and types of bonds for a given atom within its group
    # Returns a dictionary with counts and concatenated bond types for C, O, N, S
    # ... bond scoring logic ...
    return score_dictionary

def one_step(first_atom, prev_atom, group, backbone):
    # Finds the next atom in a chain from a previous atom within the side chain group
    # Used for traversing the side chain
    # ... logic to find the next connected atom ...
    return next_atom # or None
```

Version 2: Graph Isomorphism and Similarity

Version 2 takes a more generalized and powerful approach to side chain analysis using graph theory. The `extract_side_chain` function generates a subgraph for each identified Cα atom, representing its complete side chain by recursively including all connected atoms that are not part of the main peptide backbone.

A key component of Version 2 is the `amino_acids` dictionary, which stores pre-defined NetworkX graph structures for the side chains of all 20 standard amino acids (including some common isomers to account for potential structural variations or protonation states).

The `identify_ac` function attempts to find an exact match between an extracted side chain graph and the known amino acid side chain patterns using graph isomorphism. The `compare_graphs` function performs this comparison. It first checks basic properties like the number of nodes and edges. Then, it uses `networkx.algorithms.isomorphism.GraphMatcher` with custom node and edge matching functions (`node_match` checks if atom types are the same, `edge_match` checks if bond types are the same) to determine if the two graphs are structurally identical. It also includes a check to ensure the Cα nodes in both graphs correspond.

Recognizing that experimental data might not always perfectly match ideal structures (e.g., due to missing hydrogen atoms or minor structural distortions), Version 2 also incorporates a similarity measure using graph kernels from the `grakel` library. The `wl_similarity` function calculates the similarity between two graphs using the Weisfeiler-Lehman kernel, which provides a score between 0 and 1 indicating how alike the two graphs are based on their local neighborhood structures. The `similarity` function uses this score as a fallback: if an exact isomorphism is not found, it identifies the amino acid whose known side chain pattern has the highest similarity score above a predefined threshold (0.85 in the code). This makes Version 2 more robust to minor structural deviations in the input data.

The `map_atoms_to_amino_acids` function then assigns each atom in the original molecule to its corresponding amino acid residue based on the side chain analysis. Side chain atoms are assigned directly from the identified side chain group. Backbone atoms are assigned to the amino acid corresponding to the nearest Cα atom in the graph.

```
# Dictionary of pre-defined NetworkX graph patterns for amino acid side chains
amino_acids = {'ALA': nx.Graph(), 'GLY': nx.Graph(), ...} # ... full dictionary ...

def extract_side_chain(c_alpha, backbone, C_alphas, Graphh):
    # Extracts the subgraph representing the side chain for a given C-alpha atom
    # Recursively includes all connected atoms not in the backbone
    # Returns a NetworkX graph of the side chain
    Gr = nx.Graph()
    # ... recursive extraction logic ...
    return Gr

def compare_graphs(graph1, graph2):
    # Compares two graph structures for isomorphism (structural identity)
    # Checks node/edge counts, atom attributes, and bond types
    # Uses GraphMatcher to find isomorphisms, including checking C-alpha correspondence
    # Returns True if isomorphic, False otherwise
    # ... isomorphism comparison logic ...
    return True or False

def nx_to_grakel(nx_graph):
    # Converts a NetworkX graph to the format required by GraKeL
    # Returns a tuple of edges, node labels, and edge labels
    # ... conversion logic ...
    return (edges, node_labels, edge_labels)

def wl_similarity(graph1, graph2):
    # Calculates the similarity between two graphs using the Weisfeiler-Lehman kernel
    # Returns a similarity score between 0 and 1
    # ... kernel calculation logic ...
    return similarity_score

def similarity(side_ch, backbone_patterns):
    # Finds the amino acid pattern with the highest similarity score to a given side chain
    # Iterates through known patterns and calculates WL similarity
    # Returns the name of the best matching amino acid and its score
    # ... similarity comparison logic ...
    return best_match_name, best_score

def identify_ac(graph, amino_acids_patterns, c_alphas):
    # Identifies the amino acid by comparing its side chain graph to known patterns
    # First tries exact isomorphism, then falls back to similarity if no exact match
    # Returns the identified amino acid code (e.g., "ALA") or "UNK"
    # ... identification logic using compare_graphs and similarity ...
    return amino_acid_code

def map_atoms_to_amino_acids(data):
    # Maps each atom in the original molecule to its corresponding amino acid
    # Uses side chain analysis and assigns backbone atoms based on nearest C-alpha
    # Returns a dictionary mapping atom IDs to amino acid codes
    atom_to_aa = {}
    # ... mapping logic ...
    return atom_to_aa
```

3.4. Output and Annotation

Both versions produce an annotated SDF file as output, which is highly valuable for downstream bioinformatics tasks. The `create_annotated_sdf` function in both scripts takes the original SDF data and the mapping of atoms to amino acids and generates a new SDF file. For each atom record in the original file that is part of the identified protein structure, the corresponding three-letter amino acid code is appended to the line. Atoms not assigned to an amino acid (e.g., isolated molecules, ions) might be

labeled as "UNK" (Unknown) or left without an annotation depending on the implementation details. This annotated SDF file can then be directly used as input for molecular visualization software, simulation setups, or further analysis pipelines that require residue-level information.

Simplified code example for output and annotation (Version 2 shown, Version 1 is similar):

```
def create_annotated_sdf(input_data, output_file):
    # Get atom to amino acid mapping
    atom_to_aa = map_atoms_to_amino_acids(input_data)
    # Write the new SDF file
    with open(output_file, "w") as outfile:
        in_atom_section = False
        for line in input_data:
            # Check if we're in the atom section
            if line.startswith("M  V30 BEGIN ATOM"):
                in_atom_section = True
                outfile.write(line)
                continue
            if line.startswith("M  V30 END ATOM"):
                in_atom_section = False
                outfile.write(line)
                continue
            # If in atom section, append amino acid
            if in_atom_section and line.startswith("M  V30") and not line.startswith("M  V30 BEGIN") and not
line.startswith("M  V30 END"):
                parts = line.split()
                if len(parts) >= 3:
                    atom_id = parts[2]
                    if atom_id in atom_to_aa:
                        # Keep original line and append amino acid
                        outfile.write(f"{line.rstrip()} {atom_to_aa[atom_id]}\n")
                    else:
                        outfile.write(f"{line.rstrip()} UNK\n")
                else:
                    outfile.write(line)
            else:
                outfile.write(line)
```

4. Comparative Analysis and Application Scenarios

This section provides a detailed analysis comparing the two developed software pipelines and outlines the specific scenarios where each version is most appropriately applied based on their distinct characteristics, operational aspects, and execution methods.

4.1. Execution

Both Python scripts (`Project1.py` and `Project2.py`) are command-line applications. Ensure a Python interpreter is installed. Ensure the `file_path` variable in each script is updated to point to your specific input SDF file.

To run **Version 1** (`Project1.py`), execute the script using the Python interpreter. This will generate the output file. To run **Version 2** (`Project2.py`), first install required libraries: `pip install network grakel`. Then, execute the script using the Python interpreter. This will generate the output file.

4.2. Comparative Analysis

The two pipelines possess distinct advantages and disadvantages stemming from their core methodologies. Understanding these differences is key to selecting the appropriate tool for a given task.

Feature	Version 1 (Pattern Matching)	Version 2 (Graph-Based Recognition)
Core Methodology	Heuristic pattern matching based on atom indices and simple bond counts.	Graph representation, isomorphism, and similarity using graph kernels.
Backbone Detection	Relies on sequential atom IDs and specific bond patterns (potentially fragile to reordering).	Graph traversal based on connectivity and bond types (robust to atom ordering).
Side Chain Analysis	Heuristic rules based on bond counts and types (effective for simple side chains, less so for complex ones).	Graph comparison to pre-defined patterns using isomorphism and similarity (more comprehensive and robust).
Amino Acid Coverage	Primarily identifies common amino acids based on explicit heuristic rules.	Aims to identify all 20 standard amino acids, including complex and aromatic ones, using graph patterns.
Robustness to Noise	Less robust; sensitive to missing atoms or deviations from expected patterns.	More robust; graph kernels handle minor structural variations and can provide probabilistic assignments.
External Dependencies	Minimal (primarily standard Python library).	Requires `networkx` and `grakel`.
Computational Cost	Generally faster, especially for small molecules.	Can be more computationally intensive for large molecules due to graph operations.
Implementation Complexity	Simpler and easier to understand at a glance.	More complex due to the use of graph libraries and algorithms.
Output Annotation	Appends three-letter codes to atom lines, primarily for identified residues.	Appends three-letter codes, aiming for comprehensive labeling of all protein atoms.

4.3. Application scenarios

The choice between using Version 1 and Version 2 depends on the characteristics of your input data and the specific requirements of your task.

Use Cases for Version 1:

- Speed-critical tasks: Rapid processing of many small peptides.
- Clean, standardized data: Using SDFs with predictable atom ordering and minimal noise.
- Focus on common amino acids: Primarily identifying Gly, Ala, Val, Leu, Ile, Ser, Thr, Cys.
- Minimizing dependencies: Environments where external library installation is difficult.
- Initial exploration: Quick testing on simple, ideal datasets.

Use Cases for Version 2:

Version 2 is preferred for scenarios demanding higher accuracy, robustness, and comprehensive coverage:

- Noisy data: Processing experimental structures with imperfections.
- Identifying all 20 standard amino acids: Accurate identification of all canonical amino acids, including complex side chains.
- Comprehensive structural analysis: Applications requiring accurate residue assignment for all protein atoms (simulations, structural interpretation).
- Robustness to variations: Handling minor structural deviations using graph similarity.
- Graph-based workflows: Integration with subsequent analysis steps using graph methods.

In essence, Version 1 offers a fast, simple solution for ideal cases, while Version 2 provides a more powerful, accurate, and adaptable approach for real-world molecular structures.

5. Conclusion

This project successfully addressed the critical challenge of identifying amino acid residues within molecular structures solely based on their topology and elemental composition, a common necessity when explicit residue information is lost during data processing or conversion. The work unequivocally demonstrated the feasibility of this topology-based approach, providing a valuable method for recovering essential structural context.

Two distinct software pipelines were developed and implemented to tackle this problem, each tailored to different needs and computational environments. Version 1, characterized by its streamlined pattern matching methodology, offers a fast and straightforward solution. It proves highly effective for processing ideal, well-ordered structures and is particularly well-suited for rapid analysis and initial prototyping where speed and simplicity are prioritized.

Complementing this, Version 2 was developed as a more robust and comprehensive tool, leveraging the power of graph-based methods, including graph isomorphism and similarity kernels. This approach provides significantly greater resilience to the complexities and noise inherent in real-world biological data, such as missing atoms, structural distortions, and variations in atom ordering. Version 2's ability to accurately identify all 20 standard amino acids and handle structural variations makes it invaluable for detailed structural analysis and applications demanding high reliability.

A key outcome of both pipelines is the generation of annotated SDF files. By appending the identified three-letter amino acid codes to the atomic records, these files provide crucial residue-level information that seamlessly integrates with existing bioinformatics workflows, enabling downstream tasks like molecular simulations, structural validation, and detailed interpretation.

In summary, this project not only validated the principle of topology-based amino acid recognition but also delivered two practical and distinct tools. While Version 1 offers an efficient solution for well-behaved data and rapid assessments, Version 2 stands as a powerful and versatile instrument for comprehensive and reliable analysis of diverse protein structures encountered in research, making a significant contribution to the toolkit for structural bioinformatics.