Bioinformatics Algorithms ICA

Command-Line Interface (CLI)

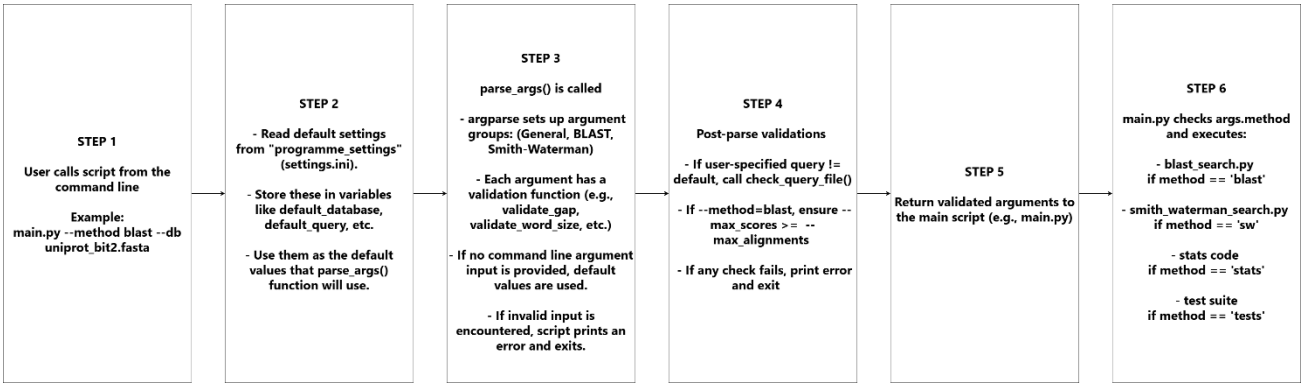

Figure 1: Flow chart of the command-line interface logic

1.1 Overview

To enable a user-friendly command-line interface (CLI) for Blast101, the Python argparse library is employed. Furthermore, all CLI functions and variables are stored in the cli.py script for better organization, reusability, and maintainability of the command-line tools, as well as a rationale that follows best practices while also acting as a filter for the arguments that will be subsequently used. Initially, as demonstrated by Figure 1, the default values for the arguments that the user can use are extracted from the settings.ini and thus incorporated into the CLI argument parser function parse_args(), which holds the main functionality of the CLI. Next, the CLI sets up a series of command-line arguments and validates both default and user-specific inputs before returning the arguments to any script that calls the argument parser function, ensuring that improper inputs are caught early. Overall, unlike the original versions of the scripts that pulled values from settings.ini at runtime, this CLI version allows direct user input to override the default values provided by settings.ini for any of the specific parameters that the argument parser function exposes.

1.2 Design Goals and Rationale

When working on the CLI for Blast101, there were three main design goals: keeping it simple, acting as a gatekeeper for bad inputs, and supporting future extensibility. Simplicity was crucial so that the user would not be overwhelmed and would have an interface that anyone in the field could use. Therefore, it was decided that the arguments available to the user should match those found on the NCBI Blast website including some advance parameters so that users maintain control over the process without risking breakage in the code. Additionally, a “--help” argument option is provided, which gives the user a rundown of the possible arguments, their functionality, and an example to follow, all while using industry-standard naming conventions for clarity. Every argument has a specific “type” parameter managed by a unique validation function, ensuring that it follows the correct specification for subsequent processing (for example, requiring integers or file paths). Finally, by having the code follow certain aspects of functional programming and modular organization, you can easily add or modify arguments, update validations, or change error-handling behaviour without affecting the rest of the script.

1.3 Essential CLI Features

As mentioned above, for simplicity's sake, the CLI tries to match the inputs provided by the website version of Blast. Because of this, there are nine inputs in total that the user can provide, encompassing three groups of arguments (Code Block 1). The first group consists of general arguments that most users would be familiar with, such as the method to run, the database path, and the query file location. The BLAST and Smith-Waterman specific sections consist mainly of the more advanced options available in NCBI BLAST, such as word size and the maximum number of scores to retrieve. To elaborate on the possible four methods, both "blast" and "sw" run the alignment scripts, while "stats" initiates the statistical analysis for the Gumbel distribution, and "tests" initializes all possible unit tests. The rest of the arguments that these methods use are supplied through the settings.ini file.

Code Block 1: Limited CLI Help Output

```
General Arguments:
  --gap <int>           Gap penalty (negative integer) (default: -5)
  --db <file>           Path to the primary database FASTA file (default: uniprot_bit2.fasta)
  --query <file>        Path to query FASTA file (default: pwnaaplhnfgedflq)
  --method <blast|sw|stats|tests>
                        Choose which search method to run (default: blast)
  --blosum <int>        Scoring matrix (BLOSUM) to use (45|50|62|80|90) (default: 62)
  -h, --help            Show this help message and exit

BLAST-Specific Arguments:
  --max_alignments <int>
                        Maximum number of alignments to output (default: 100)
  --max_scores <int>    This determines the maximum number of initial high-scoring hits (HSPs) that the
                        BLAST algorithm internally keeps during the search. (default: 200)
  --word_size <int>     Word size for BLAST heuristic (default: 4)

Smith-Waterman Arguments:
  --max_sw_scores <int>
                        This determines the maximum number of initial high-scoring hits (HSPs) that the SW
                        algorithm internally keeps during the search. (default: 1000)
```

In addition, as seen in the limited CLI's help output, each argument has a type that it needs to follow, "int" means an integer, and "file" means a path to a specific file. The problem is that the input that the user provides, for example if the user attempts to enter the number "three" for a parameter that must be an integer, it is recognised by the system as a string and thus the validation functions in the CLI need to act both as transformers (converting strings to integers) and checkers (ensuring that the values fall within acceptable thresholds). The input validations can be split into two types, first a typical numeric validation uses a try statement to verify that the input can be converted to an integer and then checks whether the value is within the allowed range, returning the transformed value with no errors raised (Code Block 2).

Code Block 2: Validation Function

```
    try:
        ivalue = int(value)
    except ValueError:
        raise argparse.ArgumentTypeError("Must be an integer.")

    max_allowed = 5000
    if ivalue <= 0 or ivalue > max_allowed:
        raise argparse.ArgumentTypeError(f"Value must be less than or equal to {max_allowed} but bigger than
0.")
    return ivalue
```

For the query and database files, the validation function checks that each file exists, and they are not empty. Additionally, using Biopython's SeqIO fasta parser package the type and structure of the files are verified and the sequences are extracted for a verification through regular expressions. First, the query is verified that it doesn't only consists of "ATGC" characters and then for both query and database sequences, if the only characters found are protein characters (Code Block 3). If an illegal character is detected, an error is raised, indicating the sequence ID and the problem character. There are also some post-validations, for instance ensuring that the maximum score is always less than or equal to the maximum number of alignments; it would not make sense for the opposite to be true.

Code Block 3: Regular Expression Validation

```
if not re.fullmatch(r"[ACDEFGHIKLMNPQRSTVWYBZXOJU]+", str(record.seq.upper())):
```

Another feature worth mentioning is the function named "current_arguments", which is not part of the argument parser itself but is still closely connected. This function is called in all the main methods (except for tests) to display the current arguments when the BLAST, Smith-Waterman, or stats scripts are run (Code Block 4).

Code Block 4: Current Argument Output

```
Running with arguments:
  blosum: 62
  db: uniprot_bit2.fasta
  gap: -5
  max_alignments: 100
  max_scores: 200
  max_sw_scores: 1000
  method: blast
  query: pwnaaplhnfgedflqpyvqlqqnfsasdlevnleatreshahfstpqalelflnysvtp
  word_size: 4
```

1.4 User Instructions

The final point to highlight (Code Block 5) is how all components integrate. The cli.py script does not run on its own; rather, it provides the logic that main.py uses to execute the correct script, depending on the user's command-line inputs. Once main.py is invoked, subsequent modules like blast_search.py also initialize the argument parser function to apply the user's customized settings for a successful run. Two common examples are:

Code Block 5: Example of CLI command to run

```
# 1) BLAST search using defaults from settings.ini:
python3 main.py

# 2) BLAST search overriding defaults:
python3 main.py --db custom_db.fasta --query custom_query.fasta --method blast
```

Testing the Blast101 Application

2.1 Testing Strategy & Framework

Following standard software development practices and the need for automated functionality when testing code, the chosen framework for unit testing the Blast101 program is unittest. This framework was selected over other testing packages largely because it integrates seamlessly with vanilla Python and does not introduce external dependencies. Moreover, functionalities such as the discover command automatically detect scripts whose names begin with “test,” and there is compatibility with additional tools like coverage (which provides reports on test coverage) and unittest.mock (a module that simplifies replacing system components with mock objects for detailed assertion checks). In addition, unittest complements the extensibility of the CLI by promoting modular test organization and offering a wide range of assertion methods that facilitate clear, easy-to-understand error messages when tests fail, thereby speeding up bug identification and resolution.

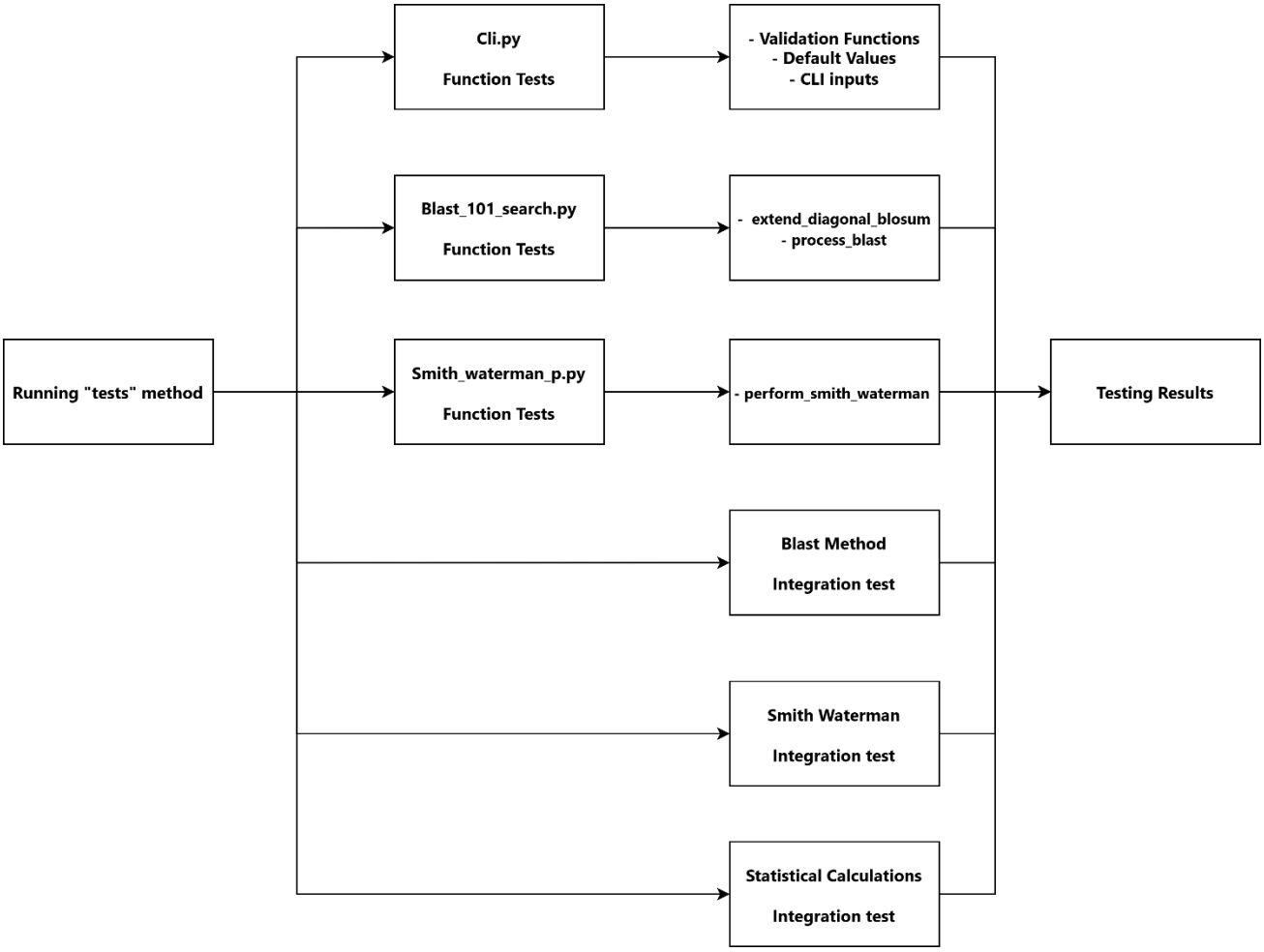

Figure 2: Flow chart of the Unit and integration tests

The main types of tests implemented through unittest include both unit tests for key functions and integration tests for end-to-end execution. In total, there are six test scripts containing twenty seven tests, some of which include sub-tests. Three of these test scripts are end-to-end integration tests

corresponding to the three possible methods, while the remaining three focus on important functions in BLAST, Smith-Waterman, and the CLI.

2.2 Test Implementation

As previously mentioned, the testing strategy for Blast101 was designed to ensure that both individual functions and the overall pipeline function reliably. For unit testing, each function was verified independently to confirm that it behaves as expected. These functions include alignment scoring for both BLAST and Smith-Waterman, as well as various input validation routines, each tested with sets of valid and invalid inputs. Test cases were constructed using Python's unittest framework, wherein functions such as validate_gap were verified by checking correct type conversions, appropriate scoring, the filling of any missing CLI arguments with defaults, and proper error handling when encountering out-of-range values. For a function like check_db_file, a temporary FASTA file was provided with different variations non-existent paths, incorrect file extensions, and invalid characters to cover all potential branches in the validation logic and ensure that all edge cases were tested. Similarly, scoring functions such as perform_smith_waterman and extend_diagonal_blosum were validated by comparing computed alignment scores against manually calculated expected values for small, well-defined input sequences across multiple BLOSUM matrices. For process_blast, the alignment score was tested in a similar manner, although the focus was on verifying scenarios that would return a score of zero.

In addition to these focused unit tests, integration tests were performed to ensure the seamless functioning of the entire pipeline from reading command-line arguments and generating final output to verifying that the output is non-empty and that the exit code does not indicate errors. Integration testing involved simulating complete user runs by invoking the main script with various parameter combinations, then confirming that the resulting outputs were generated as expected. For instance, a test might mimic a full BLAST search by providing a predefined set of command-line inputs, then verifying that subsequent modules receive validated parameters and produce the correct final output. This end-to-end approach confirms that the individual components remain logically consistent and robust when combined.

All the tests described above are designed to run automatically via the testing framework, and no external tools are used for calculations. This approach ensures that there are not dependencies when calculating alignment code. Scores for all BLOSUM tables used by perform_smith_waterman and extend_diagonal_blosum were obtained from NCBI's data repository and cross-referenced with the data provided by the internal BLOSUM package. Collectively, these automated testing efforts provide a comprehensive safety net that maintains the code's reliability and performance under production-like conditions.

2.3 Example Test Cases

This section presents examples of both unit tests and integration tests, highlighting the technical aspects of the implementation. Code Block 6 shows the general structure used for integration tests; it specifically illustrates how the end-to-end BLAST method is tested.

Code Block 6: Integration Test for Blast Method

```
class TestBlastIntegration(unittest.TestCase):
   def test_default_values(self):
       """
```

```
        """
        test_argv = [
            "progname",
            "--db", "uniprot_bit2.fasta",
            "--query", "logs/Blast_test.fasta",
            "--method", "blast"
        ]

        csv_path = os.path.join("logs", "BLsearch.csv")

        if os.path.exists(csv_path):
            os.remove(csv_path)

        with patch.object(sys, 'argv', test_argv):

            mock_stdout = io.StringIO()
            mock_stderr = io.StringIO()
            with patch('sys.stdout', new=mock_stdout), patch('sys.stderr', new=mock_stderr):
                with self.assertRaises(SystemExit) as cm:
                    main.main()

            exit_code = cm.exception.code

            # Check that the pipeline signaled success (exit code=0).
            with self.subTest("Return code check"):
                self.assertEqual(exit_code, 0, "Blast pipeline did not exit with code 0.")

            # Check that BLsearch.csv was created and has content.
            with self.subTest("CSV file existence check"):
                self.assertTrue(
                    os.path.exists(csv_path),
                    "Expected logs/BLsearch.csv to be created, but it does not exist."
                )

            if os.path.exists(csv_path):
                with open(csv_path, "r") as f:
                    lines = f.readlines()

                with self.subTest("CSV file content check"):
                    self.assertGreater(
                        len(lines), 3,
                        "Expected logs/BLsearch.csv to have some content, but it's empty or missing lines."
                    )

            if os.path.exists(csv_path):
                os.remove(csv_path)
```

Here, test arguments are assigned to a variable. The uniprot_bit2.fasta database is used along with a custom FASTA file containing a known sequence (“pwnaaplhnfgedflqpyvqlqqnfsasdlevnleatreshahfstpqalelflnysvtp”) that has expected hits. A path for the resulting CSV file is similarly defined and checked; if it exists, it is removed to avoid false positives. The test arguments are then patched into the command line (via sys.argv), and the main function is invoked, capturing the exit code. Assertions check whether the exit code is zero (indicating smooth execution), whether the file is created, and whether it contains more than three lines. Similar logic applies to the Smith-Waterman integration test, while the stats script test primarily checks for the proper exit status rather than file output.

Following the next piece of code block 7 combines and show cases two test scenarios for the two distinct validation function types in this case testing the database input and testing the smith waterman maximum score.

Code Block 7: Unit Test for CLI Functions

```
def test_database(self):

    test_cases = [
        (">test_sequence\n****\n", "Database file with invalid characters"),
        ("", "Empty database file"),
        (None, "Nonexistent database file")
    ]

    for content, description in test_cases:
        with self.subTest(description=description):
            if content is not None:
                tmp_file = tempfile.NamedTemporaryFile(mode="w", delete=False)
                try:
                    tmp_file.write(content)
                    tmp_file.close()
                    tmp_path = tmp_file.name
                except Exception:
                    tmp_file.close()
                    os.remove(tmp_file.name)
                    raise
            else:
                tmp_path = "nonexistent_file.fasta"

            test_argv = ["progname", "--db", tmp_path]
            with patch.object(sys, 'argv', test_argv):
                with patch('sys.stdout'):
                    with self.assertRaises(SystemExit):
                        cli.parse_args()

            if content is not None:
                os.remove(tmp_path)

def test_sw_max_score(self):

    test_argv = ["progname", "--max_sw_scores", "50000"]
    with patch.object(sys, 'argv', test_argv):
        with patch('sys.stderr'):
            with self.assertRaises(SystemExit, msg="Expected parse_args() to exit on invalid SW max score."):
                cli.parse_args()
```

First, the database input test focuses on three scenarios: a file containing invalid characters, an empty file, and a nonexistent file. For the two scenarios requiring a FASTA file, the tempfile package creates a temporary file to store the corresponding test content. The test then patches a mock object to simulate CLI behavior by providing the path to this file as the database argument. If the validation function in the CLI script correctly raises an error, the test is considered successful; if not, an error message indicates the mismatch between expected and observed behavior.

The second part of Code Block 8 illustrates a test for the --max_sw_scores argument. Here, the CLI argument parser is given an out-of-range value of 50000, which exceeds the permitted limit. An assertion confirms that the parser raises an exception, validating the function's proper handling of invalid numeric inputs. In practice, numerous test cases of this kind are used to cover the full range of potential user errors. The goal is to detect and correct any issues in the code before users encounter them.

Code Block 8 shows two additional functions tested in test_blast.py: extend_diagonal_blosum and process_blast. Because the CLI supports multiple BLOSUM matrices, tests for the extend_diagonal_blosum function include coverage of all possible alignment scores under different BLOSUM tables.

Code Block 8: Unit Test for Blast Functions

```
class TestExtendDiagonalBlosum62(unittest.TestCase):
   def test_extend_diagonal_blosum(self):
       """
       """
       test_cases = [
           (62, 19, "BLOSUM 62"),
           (45, 24, "BLOSUM 45"),
           (50, 26, "BLOSUM 50"),
           (90, 21, "BLOSUM 90"),
           (80, 20, "BLOSUM 80")
       ]

       for blosum, b_scores, expected in test_cases:
           with patch.object(blast_101_search, 'ddist', new=bl.BLOSUM(blosum)), \
                patch.object(SW, 'dist', new=bl.BLOSUM(blosum)), \
                patch.object(blast_101_search, 'word_size', new=3), \
                patch.dict(blast_101_search.programme_settings.settings["BLAST"], {
                    "min_extension_score": "0",
                    "max_extension_length": "10"
                }):
               # Sequences
               s0 = "ACD"
               s1 = "ACD"
               # The starting position (0,0)
               pos_s0_s1 = (0, 0)
               # Call extend_diagonal
               score = blast_101_search.extend_diagonal(pos_s0_s1, s0, s1)

               with self.subTest(f"Testing the function for expected score based on {expected}"):
                   with patch('sys.stderr'):
                       self.assertEqual(
                           score,
                           b_scores,
                           f"Expected a perfect diagonal match of {b_scores}, got {score}"

class TestProcessBlast(unittest.TestCase):

   def test_single_match_found(self):
       with patch.object(blast_101_search, 'word_size', new=3), \
            patch.object(blast_101_search, 'query_sequence', new=defaultdict(list, {"GGG": [0]})), \
            patch.object(blast_101_search, 'qsequence', new="GGG"), \
            patch.dict(blast_101_search.programme_settings.settings["BLAST"], {
                "min_extension_score": "0",
                "max_extension_length": "10",
            }):
           myline_database = "GGGXXXX"
           score = blast_101_search.process_blast(myline_database)
           with self.subTest("One match found"):
               self.assertEqual(score, 0, "Expected 0 if only one match was found (count_matches < 2).")
```

This function accepts three primary parameters: the starting position (e.g., (0, 0)) and two sequences to be aligned. It calculates the alignment score using the selected BLOSUM table, then attempts to extend beyond the word size in both directions. If the extended score fails to exceed a specified min_extension_score, alignment stops. The initial test checks whether the function's computed score matches a manually calculated score. Additional tests handle variations in extension length and min_extension_score logic.

The process_blast function, also shown in Code Block 8, splits the query into words of a predetermined size and does the same for each sequence in the database. A series of conditional checks then determines if it should call extend_diagonal_blosum to compute a final alignment score. For example, if fewer than two matches are found, the function returns a score of zero. Although only one scenario is shown, other tests examine different input conditions to confirm comprehensive coverage.

Code Block 9 presents a similar scoring example for the perform_smith_waterman function.

Code Block 9: Unit Test for Smith Waterman Functions

```
def test_insertion_sequence(self):
    """
    """

    test_cases = [
        (62, 6, "BLOSUM 62"),
        (45, 8, "BLOSUM 45"),
        (50, 8, "BLOSUM 50"),
        (90, 8, "BLOSUM 90"),
        (80, 8, "BLOSUM 80")
    ]

    for blosum, b_scores, expected in test_cases:
        smith_waterman_p.dist = bl.BLOSUM(blosum)

        seq1 = "AA"
        seq2 = "ACA"

        with patch.object(smith_waterman_p.args, 'gap', -2):
            score = smith_waterman_p.perform_smith_waterman(
                seq1, seq2, print_m=False, print_a=False
            )
            with self.subTest(f"Testing the function for gap insertion scenario on {expected}"):
                self.assertEqual(score, b_scores, f"Expected a score of {b_scores}, got {score}")
```

This test examines cases where inserting a gap maximizes the alignment score. In particular, the test controls the BLOSUM matrix, gap penalty, and sequence inputs. A manual score is calculated from the relevant BLOSUM table, and the result returned by perform_smith_waterman is compared against this known value. Other tests in the same suite explore additional conditions such as different gap penalties or edge cases with low-complexity sequences to ensure that alignment logic remains correct under a wide range of circumstances.

Overall, these tests collectively validate the handling of invalid file inputs, the correct enforcement of maximum score parameters, and the behaviour of core alignment functions. By covering edge cases and typical user interactions alike, the suite aims to detect and resolve potential bugs before they reach production.

Table 1: Alignment Matrix

		A	A
	00	00	00
A	00	04	04
C	00	02	04
A	00	04	06

2.3 Results of Testing

Coverage measurements were generated using coverage, yielding an average score of approximately 81% across all files. Although this number is encouraging, the way coverage tools operate can inflate results somewhat, particularly when integration tests call many lines of code. Nonetheless, it is believed that the most critical functions supporting the core operations of Blast101 are thoroughly tested, even if a few edge cases may remain uncovered.

During the testing process, three bugs were discovered. One was potentially serious, involving process_blast and its handling of identical words in a specific grouping mechanism. Additionally, an issue arose when BLAST and Smith-Waterman scripts were run on a Linux server, preventing output from being written to the logs folder, and another bug caused problems with the print_traceback feature. The table below (Table 2) summarizes the coverage for each file, illustrating that most modules possess high test coverage, with a few areas left for further attention.

Table 2: Coverage Report

File	Statements	Missing	Excluded	Coverage
blast_101_search.py	197	3	0	98%
calc_bit_and_evalues.py	72	14	0	81%
cli.py	130	10	0	92%
create_seq_word_dict.py	54	7	0	87%
main.py	41	18	0	56%
print_logger.py	14	0	0	100%
process_fasta_file.py	56	2	0	96%
programme_settings.py	8	2	0	75%
smith_waterman_p.py	146	21	0	86%
smith_waterman_search.py	70	0	0	100%

BLAST Questions

Question 1:

These extra residues typically represent non-standard or ambiguous amino acids that are not part of the twenty canonical amino acids listed in BLOSUM tables. While the BLOSUM62 matrix does not explicitly provide substitution scores for every possible non-standard residue, most alignment programs either treat them like a generic “X,” apply a default penalty, or omit them if they cannot be processed. In practice, these residues do not usually affect alignments unless they occur with high frequency, because their presence is often minimal and the alignment scores for unknown residues tend to be neutral or slightly penalizing. Completely removing these residues from a random alignment simulation can introduce bias by selectively discarding biological data and skewing the frequency distributions. Therefore, rather than removing them outright, it is common to let the alignment software handle them as placeholders, ensuring that the dataset remains representative and the impact on overall alignment quality is minimal.

Question 2:

Based on the code of BLAST101 and specifically the process_blast function, the two sequences are broken down into words of a specific size, with their locations on the sequence recorded. When we find a repeated word, for example “AAA,” the position on the sequence is saved in a dictionary and compared to the words found on the other sequence. For each position where we have a match, the location is saved in the res_store variable, and then we compare the locations to see where we have multiple matches that are assigned to the same group. Thus, identical repeats are grouped together, and the score of the alignment is only produced once for each group. A possible improvement to the code could be the size of the words used, by finding the ideal size that still performance comparative well to the golden standard of smith waterman we can minimize the redundant alignment score checks.

Question 3:

This approach of clustering the database and focusing on representative sequences can indeed speed up searches, because it minimizes redundant comparisons by grouping highly similar sequences. Essentially, if the representative sequence matches well, we assume that other members of the cluster are also relevant. However, it may cause issues if the chosen representative does not fully capture the variation within a cluster, resulting in missed or lower-scoring matches for slightly divergent sequences. Additionally, if the clustering is too coarse, some distant but important hits might be overlooked. Thus, while this strategy can significantly improve search efficiency, care should be taken to balance speed with the risk of missing potentially meaningful, less-represented matches.