Burrows-Wheeler Transform server and client

Overview

The project consists of an implementation of a server (along with a corresponding client) that accepts a DNA sequence and returns its Burrows-Wheeler Transform (BWT). The server should also accept a BWT and return the corresponding original DNA sequence.

Dependencies

- Python 3.10+
- NumPy 2.1.3

Usage

Running the server

To start the server, run on the terminal:

```
python server.py -H <host> -p <port> --processes <n_processes>
```

The `host`, `port`, and `n_processes` arguments are optional. If not specified, defaults are `localhost` (host), `12345` (port), and `number of CPU cores - 1` (at least 1). Advanced users can specify a number of processes ranging from 1 to double the number of available CPU cores.

Running the client

To connect the client to the server, run on the terminal:

```
python client.py -H <host> -p <port> -o <operation> -f <input_file>
```

The `host` and `port` arguments are optional. If not specified, defaults are `localhost` (host) and `12345` (port). The `operation` parameter is mandatory and must be either `"BWT"` (or "bwt") to perform burrows-Wheeler Transform or `"REVERT"` (or "revert") to revert into the original sequence. The decision to let the user specify the operation via the command line, rather than including it in the input file, aims to minimize potential errors. This approach reduces the risk of incorrect formatting, invalid commands, or typing mistakes within the file, which may disrupt the process and waste resources. The `input_file` parameter is also mandatory and must be a `.txt` of `.fasta` file. The file must contain exactly one header line, starting with `>`, followed by a single sequence. A `.txt` file example is provided in the project folder (`sequence_example.txt`)

Project files

server.py

The `server.py` file handles client connections and processes requests from users. It performs the following tasks:

1. Socket creation and binding Once the host, port, and n_processes arguments are validated (see the `Validation` section), it creates the socket and binds it to the address (host and port). If no errors occur, the server starts listening for incoming connections.
2. Multiprocessing Uses the `multiprocessing` module to efficiently handle multiple client connections simultaneously. The default number of processes is set to the number of CPU cores minus one. Creating a pool of processes with this default allows the server to optimize resource allocation and ensure efficient task execution without overloading the system. This is particular important in this context, where the operations performed by the server are mainly CPU-bound. However, advanced users can specify a number of processes between 1 and twice the number of CPU cores to guarantee flexibility.
3. Handling Client requests
    - Accepts incoming connections.
    - Receives data from the client, including the operation to perform (BWT or REVERT) and the DNA sequence.
    - Processes the request using the functions provided in the `conversion_functions.py` file, depending on the operation, and generates the result.
    - Sends the result back to the client.
4. Error handling and Logging Handles various types of errors, including validation errors and socket-related errors, using logging and exit codes. All server activities and errors are recorded in the `server_activity.log` file for debugging.
5. Connection Closure Ensure that client connections are closed, wether the request is succesfully completed or results in an error.

client.py

The `client.py` file allows users to send requests to the server and receive the results. It performs the following tasks:

1. Parameters Validation
    - Validates the host and port arguments.
    - Validates the input file, ensuring it exists and contain a valid DNA sequence. For further information, see the `Validation` section under `Additional Information`.
2. Handling the connection with the server
    - Creates a socket and connects to the server using the specified or default host and port.
    - Sends the `operation` (BWT or REVERT) and the DNA sequence (`seq`) taken from the input file to the server.
    - Receives the processed data and decodes it.
    - Close the connection with the server, wether the request is succesfully completed or results in an error.
3. Generating the output file
    - Creates an output file named based on the input file, the operation performed, date, and time.
    - Writes the sequence header, the operation performed, and the output sequence into the file.
4. Error handling and Logging Handles various types of errors, including validation errors, socket-related errors, and writing errors (output file), using logging and exit codes. All client activities and errors are recorded in the `client_activity.log` file for debugging.

conversion_functions.py

The `conversion_functions.py` file contains the implemented functions necessary for the Burrows-Wheeler Transform and its inverse.

1. burrows_wheeler_conversion The `burrows_wheeler_conversion` function converts a DNA sequence into its Burrows-Wheeler Transform (BWT), a reversible permutation of its characters (bases) used for data compression and indexing. This transformation is performed by generating a `suffix array` using the `build_suffix_array` function. A suffix array is the array of the starting indices of the sequence's suffixes based on lexicographic order. Firstly, the `$` terminator character (lexicographic smaller than all the other characters) is appended at the end of the sequence to distinguish all the suffixes. Then, the BWT is constructed by taking the character that preceeds the start of each suffix in the original sequence and joining these characters into a single string[1]. The construction of the suffix array is supported by the `calculate_ranks` function, which recalculates the ranks of the suffixes iteratively. The process involves reordering and reassigning ranks by comparing tuples of two values: the current rank and the rank at a k distance ahead. During the iteration, progressively larger portions of the suffixes are considered as k increases.

   The suffix array is a more efficient choice than constructing the permutation matrix, which is generally used to obtain the BWT. While the permutation matrix considers all rotations of the sequence, the suffix array focuses only on its suffixes, making it more efficient in both time and space complexity. The permutation matrix requires O(n^2) memory and has a computational complexity of O(n^2 logn), whereas the suffix array implemented here requires `O(n) memory` and has a `O(n $log^2$n) computational complexity`. The efficiency of the suffix array is further improved by the computation of ranks, which reduces the number of direct comparisons between suffixes, making the computation faster.
   Consider the string `"BANANA$"`, with the `$` character already added by the function, and its suffixes. By ordering them lexicographically, the suffix array can be obtained:

```
6: $       # index 6
5: A$      # index 5
3: ANA$    # index 3
1: ANANA$  # index 1
0: BANANA$ # index 0
4: NA$     # index 4
2: NANA$   # index 2

suffix_array = [6, 5, 3, 1, 0, 4, 2]
```

   To obtain the `BWT`, we need to take the character that preceeds the start of each suffix in the original sequence and make a join into a single string.

```
$:        A
A$:       N
ANA$:     N
ANANA$:   B
```

```
ANANA$:   $
NA$:      A
NANA$:    A

bwt = ANNB$AA
```

2. revert_burrows_wheeler The `revert_burrows_wheeler` function reverses the BWT encoded string into the original DNA sequence. This reversion is performed relying on the `LF Mapping` (Last to First mapping), a property of the BWT. The LF Mapping is implemented through the `map_last_to_first` function. The function first takes the BWT string (Last Column) and sorts it lexicographically (First Column). After that, it determines the rank of each character in the last column, where the `rank` represents the number of times a character is met in the last column up to a specific position. By summing the rank of a character determined from the last column with the index of its first occurrence in the First Column, the function calculates the corresponding position of the character in the first column. This is possible due to the LF mapping property: the rank of a specific character is the same in both the last and first columns[1]. The function then returns the final array with the indices representing this mapping. Once the mapping is performed, the revert_burrows_wheeler function iterates through the BWT array, starting from the position of the `$` terminator character, and uses the resulting indices to retrieve the characters in reverse order. Finally, the characters are joined to form the original DNA sequence, with the terminator character removed. This approach requires `O(n) memory` and has a `O(n logn) computational complexity`.
Consider the string `"ANNB$AA"`, which is the BWT of the string `"BANANA$"`. The First Column is obtained through a lexicographically sorting.

```
Last Column (BWT):     A  N  N  B  $  A  A
First Column:          $  A  A  A  B  N  N
```

Once determined the First Column, the next step is determining the LF mapping as explained above.

```
Index (Last):          0  1  2  3  4  5  6
Character (Last):      A  N  N  B  $  A  A
Rank:                  0  0  1  0  0  1  2
First occurrence:      1  5  5  4  0  1  1
LF Mapping (Indices):  1  5  6  4  0  2  3
```

The original sequence is reconstructed in reverse order using the LF mapping, starting from the `$` terminator character. The `$` is located at index 4 in the Last Column and its corresponding LF Mapping index is 0. At index 0 in the Last Column, the character `A` is found and put before the `$` in the reconstructed string. The character `A` at index 0 corresponds to LF Mapping index 1. At index 1 in the Last Column, the character `N` is found and put before `A`. This iterative process continues until the entire original sequence is reconstructed. Finally, the `$` terminator character is removed from the resulting sequence.

```
original_seq = BANANA$  # before the terminator character removal.
```

Additional Information

Validation

The validation is performed through the `validation_client` and `validation_server` functions. They ensure that the inputs provided by users meet the requirements of the client-server application. Both functions share the same logic for the host and port validation, as reported below.

- Host and Port validation
  - The `host` parameter must resolve to a valid hostname.
  - The `port` parameter must be an integer number between 1 and 65535.

The `validation_client` function has also a logic to check for the input file.

- Input file validation The `input_file` must have a .txt or a .fasta extension. It must contain a header starting with `>`, and a DNA sequence with valid DNA bases (`C`, `G`, `T`, `A`, `R`, `Y`, `S`, `W`, `K`, `M`, `B`, `D`, `H`, `V`, `N`, `$`), either uppercase or lowercase, and must be non-empty. The program automatically converts all bases to uppercase and then removes the newline characters in the sequence (normally present in fasta files). The inclusion of all the DNA bases is thought to represent the biological complexity, allowing the use of the program in scenarios where the sequences contain ambiguity. However, it is the users' responsibility to correctly interpret the results in presence of these ambiguities. If the operation to perform is `"BWT"`, the sequence must not have the `$` terminator character. If the operation to perform is `"REVERT"` instead, the sequence must also contain the terminator character `$`, which is required for the reverse transformation and must be present only once.

The `validation_server` has also a logic to check for the number of processes provided.

- Number of processes validation The `n_processes` must be a number between 1 and twice the number of CPU cores (included). This approach is needed to avoid performance degradation due to increased memory and resources consumption, and reduced efficiency.

Errors during validation are managed using log files for debugging (`client_activity.log` and `server_activity.log`) and specific exit codes (see the `Error codes` section below).

Tests

To guarantee the robustness of the system, a series of tests was implemented using the python module `unittest`. To perform the tests, navigate to the project's directory and run the following command:

```
PYTHONPATH=$(pwd) python -m unittest discover -s tests -t .
```

1. test_validation_functions These tests focus on validating the correctness of the inputs provided and simulate scenarios with possible input errors, verifying the resulting exit code. These tests verify:
   - valid inputs for client
   - invalid port
   - invalid host

- file not found
- invalid file extension
- invalid operation (due to incorrect input sequence)
- invalid header
- invalid sequence
- valid inputs for server
- invalid number of processes The simulated errors will be recorded in the log files, as if they occurred during normal operation.

2. test_conversion_functions These test focus on ensuring the accuracy of the `burrows_wheeler_conversion` and `revert_burrows_wheeler` functions.

Error codes

Exit Code	Description
0	Successful execution
1	Unexpected error
2	File not found
3	Validation error
4	Invalid host
5	Socket timeout error
6	Broken pipe error
7	Connection refused error
8	Connection reset error
9	Socket error
10	File writing error

References

[1]: Langmead, Ben. Introduction to the Burrows-Wheeler Transform and FM Index. Department of Computer Science, Johns Hopkins University, 24 Nov. 2013.

Contact

For any additional questions or feedback, please contact Luca Lepore