Programs for signal path analysis - description

Mateusz Fortunka

December 28, 2021

1. Introduction

Presented scripts were written by me during work for my B.Sc. degree "Statistical physics methods to understand the role of YARS2-tRNA complex in MLASA disease" in SulkowskaLab.

The first set of programs were developed to change data structures. One of them is important to mention, because its output was useful in other created programs. The task of "list_residues.py" is to take a PDB file and give a table with all residues including their name and three types of numbering in the output. It is mainly intended for the translation of aa numbers to their names or reversely. For instance, VMD plugin and its "subopt" program for paths calculation use only numbers.[1][2] So their inputs and outputs were easier to create and analyse using "list_residues.py". Similarly, it was useful for the programs described in the next paragraph.

The second part of scripts I wrote to cover the last step of dynamical network analysis conducted in the presented thesis - processing the big amount of data generated from calculations of signal paths. The VMD plugin gives a set of optimal and suboptimal paths with a small amount of additional information. It includes a number of paths, the most often encountered nodes and edges with their count. A program named "paths.py" takes all the output files from the "subopt" program in a folder (so from one trajectory) and gives a file with a summary for every path. Apart from the number of all found pathways and their average length with standard deviation, it also provides a number of mutated residues present in the paths. Also, it rewrites the most common aas from the "subopt" output file. Then, having summary files from all MD repeats, program "paths_summary.py" list specified pathways from every "paths.py" output file present in the folder. So its main idea is to put the data for every path together, which is useful for further analysis. Then three programs: "in_sum.py", "in_mirror.py" and "cross_sum.py" compare the data. The first two takes "paths_summary.py" output file and calculate several values for every pathway. The output "in_sum.py" gives optimal path identity, average number of paths and length. It also provides an average count of mutated residues and standard deviation of all values. "In_mirror.py" comparison is similar, but it is not performed between data from MD repeats for one chosen pathway, but between "mirror" paths, which have the same starting nodes and targets, but their chains are changed. So for paths starting at the F185 in chain A and ending at E264 in chain B, the "mirror" path will be going from E264 in chain B to F185 in chain A. This example residues were taken from a wild variant of YARS2. The last program - "cross_sum.py" takes the same paths for two variants and compares them like in the "in_sum.py". It gives an output file with a difference of every value corresponding to the path between native and mutated YARS2.

The last group of programs (e.g. cp_paths, cp_sum.sh) was prepared to avoid much typing and accelerate executing programs, copying files and changing their names. As it is basic usage of bash and C++ languages, it will not be elaborated further.

2. Description of the software

In this section, I would like to describe comprehensively five main programs prepared by myself to process the data generated by calculating signal paths. All of them are available under the link `https://github.com/ilbsm/signal_path_analysis`. Of course, I created more scripts, but these five are the most important as they could not be replaced by other existing software and contain new ideas on how to analyse mechanical pathways in proteins.

First script named "paths.py" takes all files with "out.out" extension and the list of residues in the protein. After processing the output from "subopt" program, it gives a file with blocks of information for every input file with data for one type of path. It contains aas associated with starting and ending nodes, optimal pathway, number of all calculated paths in this case, average length in distance and number of aas and standard deviations of both values. At the end of each block, there is a list of most frequently appearing residues and additional ones specified earlier with numbers of occurrences. On the Fig. 1 and Fig. 2 there is a complete block diagram of the "paths.py" program which starts with taking the path to a folder and name of the file with a list of residues. The next step is to load the list into an array and define a dictionary with specified aas. There were written residues at the mutation sites. Then the function "ch_res" was defined. It basically takes the number of a residue and translates it to the format X_ABC_123 where X is the chain letter, ABC three-letter code of a residue and 123 is its number. Subsequently, "add_res" function is defined. Its task is to add occurrences of specified aas to the "mut" dictionary. The next line of code is a one-line loop reading all filenames ending with "out.out" in the folder selected by the user. After specifying the name of the output file and opening it, there is a big, main loop in the script. It takes consecutive files, reads them, processes and writes results to the output file. Then the program closes it and writes a final message on the screen. The main loop opens a file to read and proceeds if it is not empty. It skips three first lines and from fourth and fifth reads starting and ending node number. Then the program writes the line with these two and defines two lists for lengths of paths and resets values in the "mut" dictionary. In the next step, there is a loop over remaining lines in the file. These are subsequent paths starting with optimal and followed by many more suboptimal. Every path is written as a sequence of numbers representing nodes separated with commas and spaces. At the end there is a distance in brackets. It is written in slightly strange units - 0.1 Å , but the authors wanted to avoid calculation on floating points instead of integers. Loop goes over every line and counts nodes representing aas, finds residues at mutation sites increasing proper number in "mut" dictionary and adds lengths of paths to a list. However, in the first run it also writes the optimal path, using "ch_res" function, to the output. The loop ends when the program spots the line starting with "Number", which is written to the output, or when the file ends. If there is no line with "Number" in front, that means the calculation of paths is not complete. In that case, the program deletes the last read path which can be incomplete and writes a line with a number of paths and information that

the calculation may be unfinished. Thereafter, the script computes averages along with standard deviations of the distances and numbers of aas. All these values are rounded to two decimal places. Then the line with above results is written to the output. After that, the program reads and writes the most frequent and specific residues in a small loop. The numbers of nodes are also translated to the format described earlier. At the end of the main loop, the contents of "mut" dictionary are transferred to the "out" file. After writing two additional new line characters, the input file is closed and the loop continues with the next file.

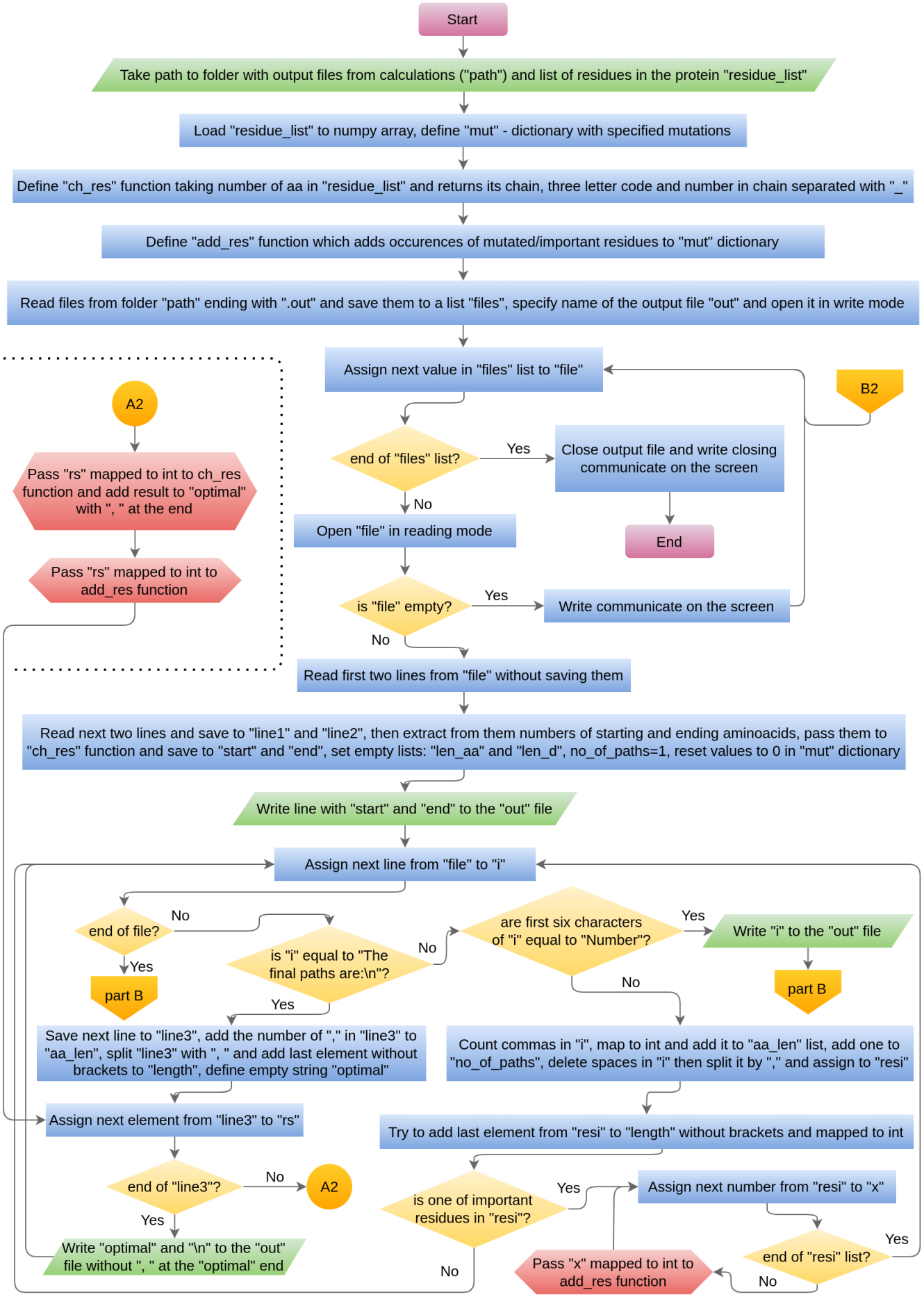

Fig. 1: Part A of the "paths" program block diagram.

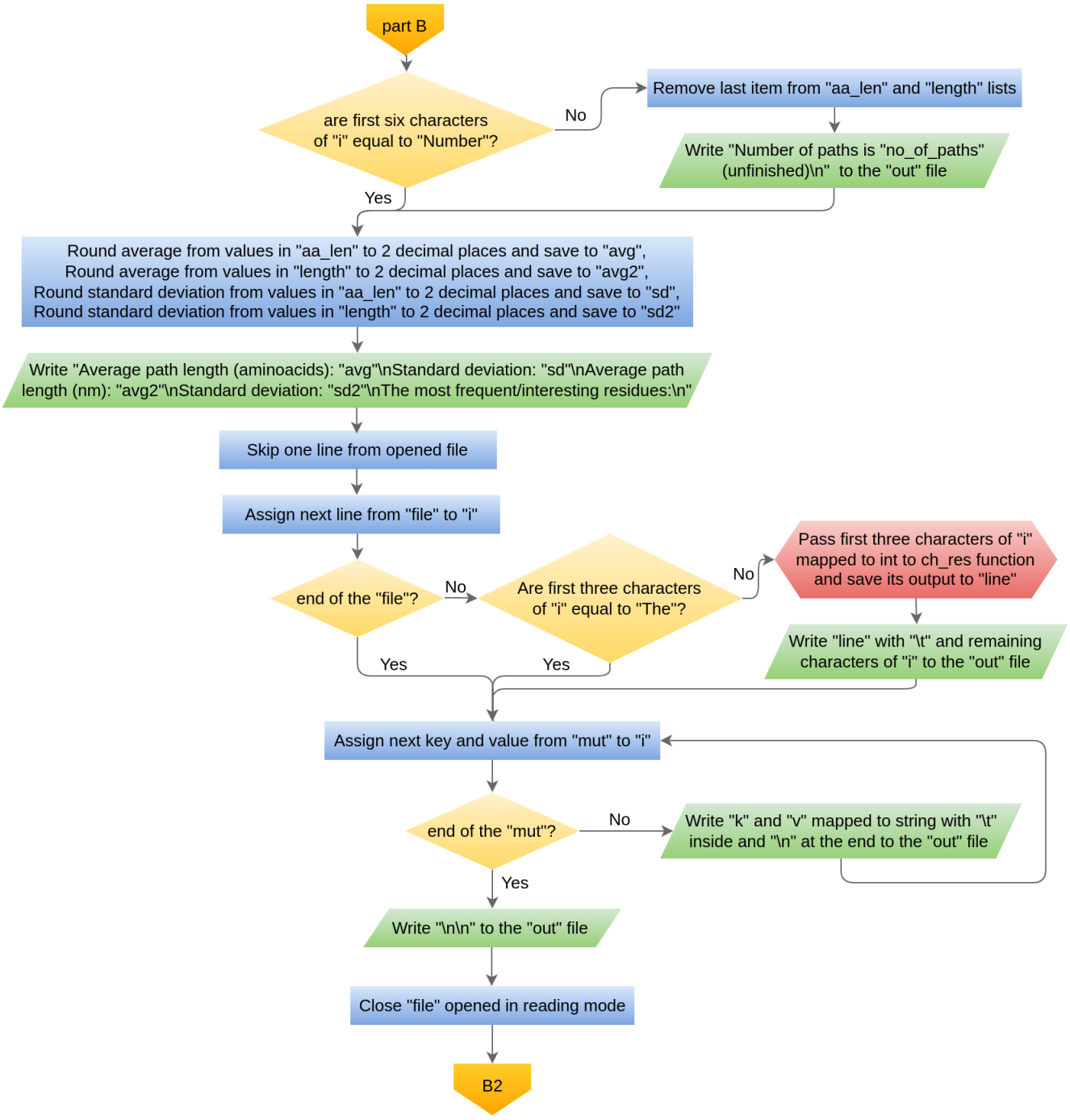

Fig. 2: Part B of the "paths" program block diagram.

The next program is "paths_summary.py" which takes the output files from "paths.py" generated for every MD repeat and gives summary for every type of pathway. Precisely, it writes each pair of starting and ending aas and values calculated for signal paths in every MD repeat in subsequent lines. These files created from independent MD simulations must be in one folder. The whole block diagram is presented on the Fig. 4 with function "calcnwrite" on Fig. 3. The latter takes three input values: list of lines from the file, number of the MD repeat and the number where the block with values for the currently analysed path starts. Then it just reads lines in the block and writes to the output loaded values in one line with the number of MD repeat. In my case, it is called five times in a loop for every kind of path to cover all performed MD repeats of one type in one program run. These values were discussed earlier with

description of the paths.py script. After defining "calcnwrite" function, "paths_summary.py" loads names of files with ".txt" extension and sorts them alphabetically. Then empty dictionary "summaries" is created and filled with contents of the files from the list with next numbers starting from 0 as keys. The next action of "paths_summary.py" is to define the name of the output file and to open it. The last part is the most important one. It contains a loop over the lines in the first file, which serves as a template to get the names of the paths which are found in the rest of the files. At the start of the loop, the program checks if the line has "Start:" at the front. If positive, it reads starting and ending aas and writes them in the output. Then it passes the template file to "calcnwrite" with index of the path block starting line. It means that the first line, with values from one MD repeat, is written to the output. Under this entry, summaries for the same path from next simulations should appear. It is done in the next few lines in the nested double loop - first over the remaining files and the second over the lines inside. When the block with summary for the specified type of path is found, the program passes the file, its key and the number of starting line to "calcnwrite" function. When the main loop ends, output is closed and the user gets a neat file containing a table with blocks of values in separate lines for every MD repeat sorted by the type of path which name is written above the corresponding block.

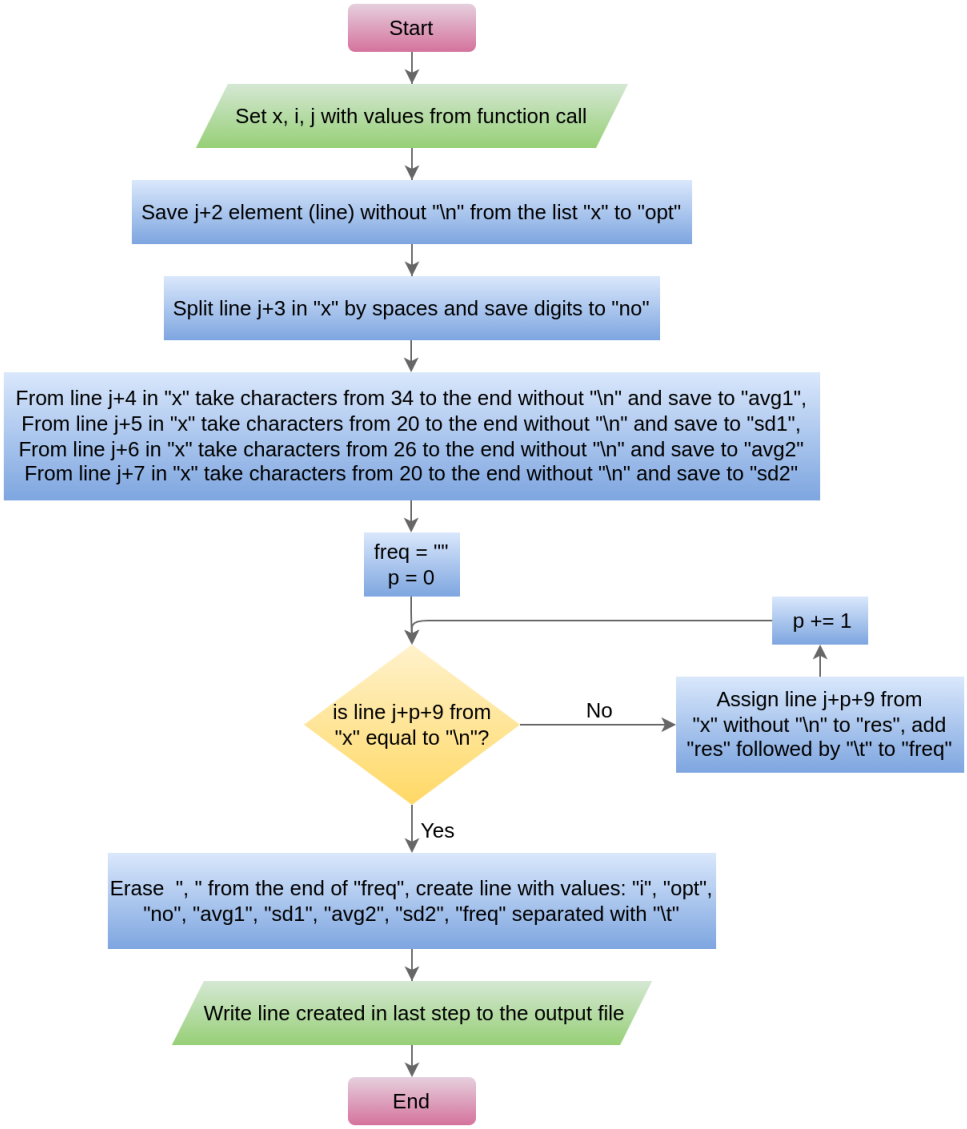

Fig. 3: Block diagram of the "calcnwrite" function used in the "paths_summary.py" program.

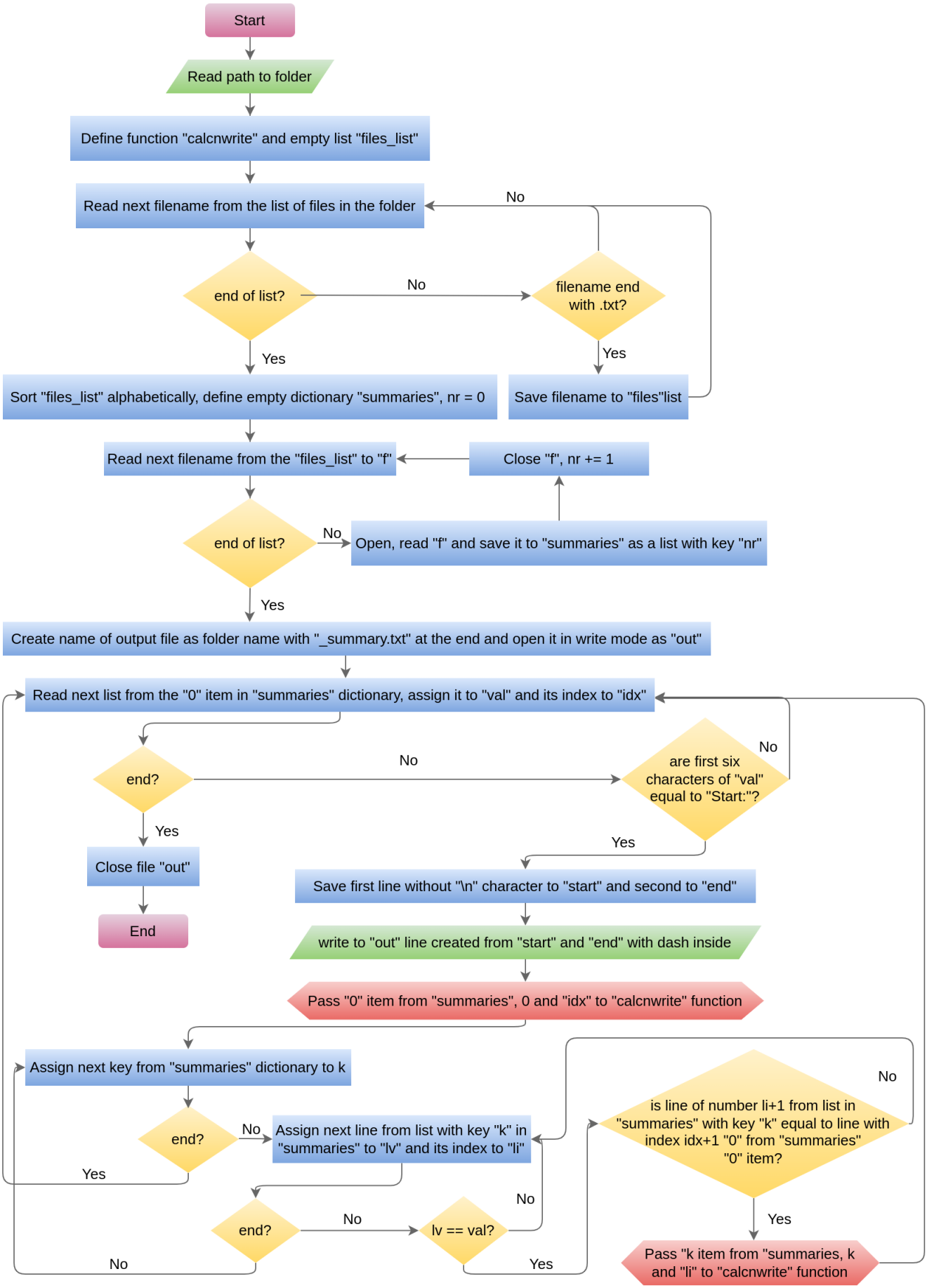

Fig. 4: Block diagram of the "paths_summary.py" program.

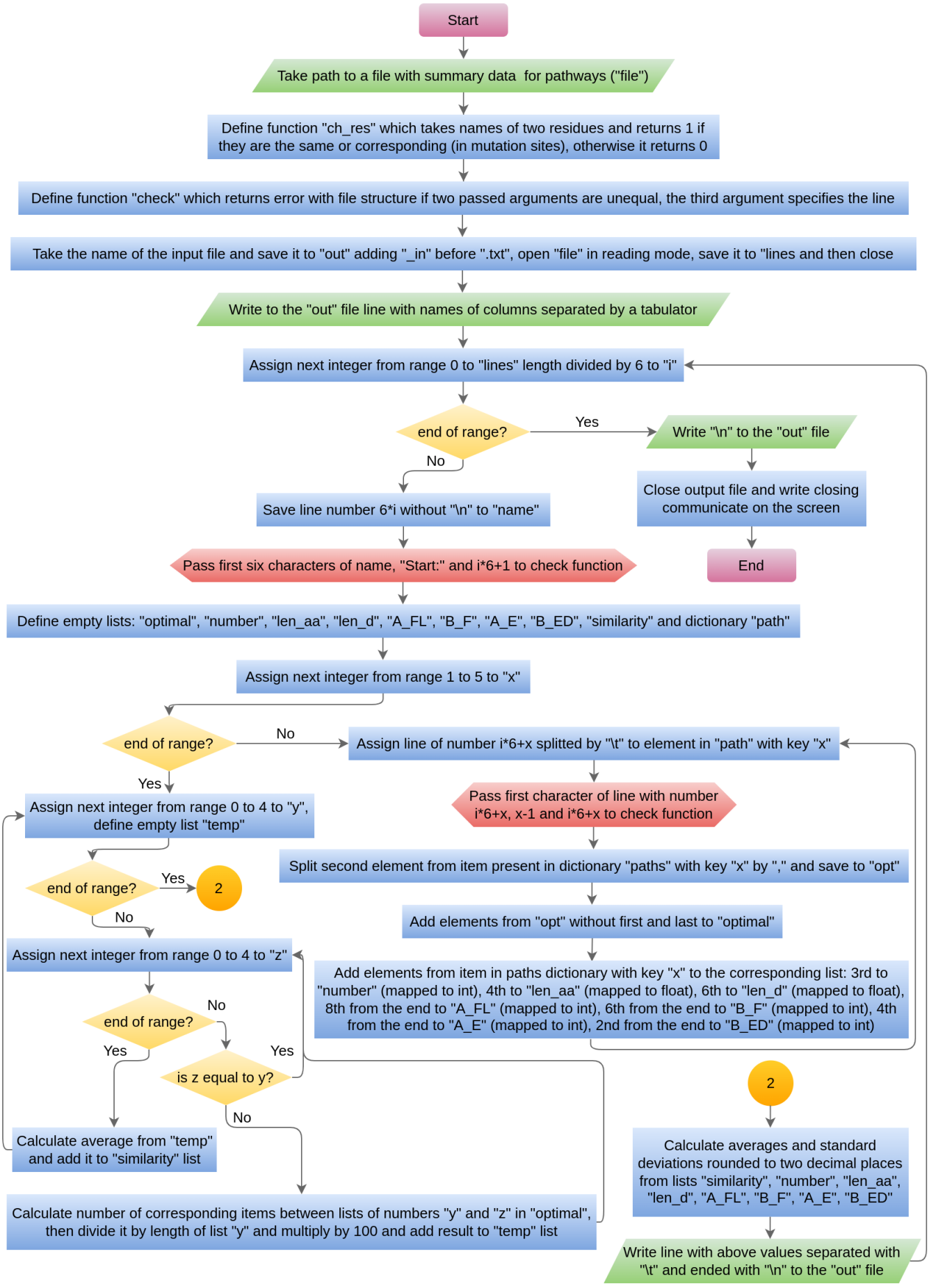

Fig. 5: Block diagram of the "in_sum.py" program.

Having data for every series of MD simulations in one file, it is possible to process it in various ways. First is to check every type of path within itself. The idea was to compare all repeats for a specific pathway to see if it is stable - has similar values in every calculation, and to have one line of average values to compare them with other types of paths. This functionality is covered by "in_sum.py" program. Its block diagram is presented on the Fig. 5. It starts with loading the path to the input file and defining functions "ch_res" and "check". The first one is the same as in "paths.py". The second returns communicate of the problem with file structure if passed values are not equal. The third input variable specifies the number of the line with the issue. This function is used to check if there is a proper starting line on each block and if sorted entries for next repeats have consecutive numbers. After this, the script specifies the name of the output file and opens it, writing the name of columns at the top. Now comes the main loop, iterating over the blocks in the input file. It takes the name of the path, checks its format and then defines dictionary "path" and lists: "optimal", "number", "len_aa", "len_d", "similarity", "A_FL", "A_E", "B_F", "B_ED". Thereafter, the program loads the values from entries into the specified lists. In the "path" dictionary, there are names which consist of starting and ending aas in the format X_ABC_123 discussed earlier. To the list "optimal" are added aas from the optimal path. Object "number" holds amounts of tracks calculated for the type of path. Lists "len_aa" and "len_d" have lengths inside - first is specified in terms of number of aas and second in distance with the unit of 0.1 Å . The last four lists ("A_FL", "A_E", "B_F", "B_ED") are for the amounts of residues at the mutation sites occurring in calculated paths. The next double loop iterates over aas in optimal paths and checks how many of them are the same, comparing all MD repeats. In this case, the number of intersecting aas between one entry and the other is divided by the amount of aas in this optimal path and multiplied by 100. After adding corresponding values obtained by comparing the same pathway to the remaining, the average is calculated. That way, I got the value which measures preserving the optimal pathway within MD repeats. When the loop ends, all lists of interesting values are used to calculate averages and standard deviations. Then they are written to the output in one line, starting with the name of the pathway. At the very end, when the main loop ends, the program writes a newline character, closes the output and writes the ending message.

In the thesis, there were also presented results from another two types of analysis - between "mirror" paths and also explicit "cross" comparison of mutated and native paths. Both mentioned scripts are very similar to the "in_sum.py", so I did not present their block diagrams. However, I am going to describe the most significant differences. "Cross_sum.py" was used to analyse relationships between the same path types in two YARS2 variants. Hence, it needs two input files generated using "paths_summary.py". The second thing is to include a loop after loading them which changes names of the mutated residues for the native ones to simplify the processing stage. The next difference is that values passed to the lists are the result of subtraction of numbers from native and mutated proteins. That means the output values are mean differences and their standard deviations. In addition, I have to make a remark about "similarity". In this case, the measure of preserving the optimal paths obviously had to change. It is no longer a comparison within repeats in the data for one YARS2 variant, but for two. Precisely, one path is taken from the first set e.g. native and the average is calculated from intersections found with each path in the second set - mutated one in this example. It is normalised like in the "in_sum.py" by the number of aas in the first path and variant, then multiplied by a hundred to get percentages in the result. There is also a second, reverse "similarity". It is counted likewise, but starting from

the second set and normalising by paths being in there. This value is implemented to check if the first one is not biased by the normalisation. I can imagine a case with short optimal path in the first variant and long in the second, but also containing most of the aas from the first one. In that example, calculations made in the one direction will show a very high "similarity factor" of the optimal paths, but it could lead to wrong conclusions. It is because the "reverse similarity factor" will be much lower owing to the division by a longer path in the process of normalisation. In my work, these two values were similar, so it was not further discussed.

The last program crucial in this study was called "in_mirror.py". It is similar to "cross_sum.py". The difference is that it takes only one file, but likely compares two types of pathways, which in this case are "mirror" paths. They were described comprehensively in the section **??** This is also a reason why the main loop was changed to the function. At first, the program loads every path name and then finds corresponding "mirror" pathways using a function which simply reverses chain symbols. Then all the same values as in "cross_sum.py" are calculated and written to the output, and the pair is removed from the list of path names. If a program does not find a "mirror" pathway (it may simply not exist) it writes, communicates and continues with the remaining ones in the list. It is vital to underline that the approach with "mirror" paths is only appropriate for the proteins with roughly identical chains.

3. Bibliography

[1] W. Humphrey, A. Dalke, and K. Schulten, "VMD – Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33–38, 1996.

[2] A. Sethi, J. Eargle, A. A. Black, and Z. Luthey-Schulten, "Dynamical networks in trna: protein complexes," *Proceedings of the National Academy of Sciences*, vol. 106, no. 16, pp. 6620–6625, 2009.