SANA software

SANA was developed in a Linux environment (Ubuntu 14.04) and it has only been tested in this environment.

The source code is "conscious" of the folder structure, so **it is important that files are in the expected locations and with the expected names**. While this approach is less flexible, it simplifies things for the user, since during a typical execution SANA may interact with many files, and it would not be practical to specify all of them as arguments.

We call the "root" folder of the project the one containing the Makefile. File paths within this document, such as `src/main.cpp`, are relative to the root folder of the project.

This document is not self-contained. Many concepts are described in the SANA Paper, the Thesis, or the Temperature Schedule Report.

Through this document, "SANA" might refer to two different things: the SANA method in particular, or the whole program, which includes many other things (and other methods).

Obtaining biological data

Sequence and GO term data should be downloaded and placed in the locations expected by SANA prior to running it.

GO terms

Evaluating GO term based measures, such as GO coverage, requires having Gene Ontology data for each specie, i.e., the list saying which GO terms each protein has. Different datasets use different formats:

- Noisy yeast dataset: The networks from this dataset do not have GO term data, as the dataset is only used to evaluate topology.

- BioGRID dataset: There is a single file containing the GO terms of every network in this dataset. It can be downloaded here: ftp.ncbi.nlm.nih.gov/gene/DATA/gene2go.gz (17MB). SANA expects it to be stored as `go/gene2go`.
- Yeast and Human dataset: each of these networks has its own file with the GO terms of their proteins. They are already in the project, as `networks/human/go/human_gene_association.txt` and `networks/yeast/go/yeast_gene_association.txt`, so they don't need to be downloaded. The original files are in http://www.geneontology.org/gene-associations/submission/gene_association.goa_human.gz (6.2MB) and http://www.geneontology.org/gene-associations/submission/gene_association.sgd.gz (1.2MB).

Sequence

There are two steps in order to be able to evaluate sequence similarity. First, the sequences of each protein must be downloaded. Second, BLAST must be executed to obtain the sequence alignment scores.

We have uploaded the sequence alignment scores directly. They can be downloaded from ~~https://goo.gl/yPzTsT~~ (400MB compressed, 1.3GB decompressed) (note: no longer online). The downloaded zip contains a folder `scores` that should be placed in the `sequence` folder, i.e., as `sequence/scores`.

Next, we explain the way to generate the scores yourself (not recommended; you can skip to next section). The way to download sequences depends on the dataset:

- Noisy yeast dataset: The networks from this dataset do not have sequence data, as the dataset is only used to evaluate topology.
- BioGRID dataset: Given a network file using the protein identifiers from the BioGRID database, the script `sequence/PPI_get_FASTA.py` queries the BioGRID database for the sequences of the proteins. It has two arguments: (i) the network file; (ii) the output file. The script `sequence/get_all_FASTA.sh` downloads the sequences of all the proteins at the same time, using the `PPI_get_FASTA.py` script. It should be run as `./get_all_FASTA.sh` from the `sequence` folder. SANA expects the output files to be stored at `networks/name/sequence/name.fasta`, where `name` is one of the names of the BioGRID dataset networks. The script `get_all_FASTA.sh` stores the output files this way, so you just need to run this script.
- Yeast and Human dataset: the files with the sequences already come with the project. They are `networks/yeast/sequence/yeast_curated.fasta`

and
`networks/human/sequence/human_curated.fasta`.

Once all the files with the sequences are in their place, BLAST must be run to create the sequence alignment scores. The script `sequence/run_blast.sh` executes BLAST with the 16 pairs of networks from the BioGRID database and the yeast and human networks. It should be run as `./run_blast.sh` from the `sequence` folder. For each pair of networks, the output of BLAST should be in `sequence/scores/name1_name2_blast.out`, where `name1` is the name of the first network and `name2` is the name of the second network. The script `run_blast.sh` stores the output files this way. Note: BLAST can take up to several hours to run for each pair of networks.

Folder structure

The project folder contains the following elements:

- `Makefile`: Makefile to build the project. It can be used as follows:
  - `make`: builds the SANA software. The output executable is called `sana`.
  - `make clear_cache`: removes files auto-generated by SANA.
  - `make clean`: removes files auto-generated by SANA as well as object (`.o`) files and the `sana` executable.
- `src/`: the source code of SANA.
- `networks/`: the networks from the datasets used in the paper. It should contain one folder for each network, with the network file in "GraphWin" (`.gw`) format. The name of the folder and the network should match. For instance, `networks/yeast/yeast.gw`. SANA will create a subfolder `networks/yeast/autogenerated/` to store data from this network, such as its distance matrix or its graphlet degree vectors.
- `NAPAbench/`: to be filled later.
- `alignmentDrawer/`: to be filled later.
- `plots/`: to be filled later.
- `scripts/`: to be filled later.
- `resnik/`: to be filled later.
- `wrappedAlgorithms/`: to be filled later.
- `experiments/`: to be filled later.

- `alignments/`: the alignments generated by SANA. SANA has an argument to specify the name of the output file containing the generated alignment. In absence of this argument, SANA gives a relevant name to the output file and stores it in this folder. For instance, `alignments/yeast_human/yeast_human_SANA_EC_480.txt`.
- `LGRAAL/`: the LGRAAL software.
- `HubAlign/`: the HubAlign software.
- `go/`: this folder contains Gene Ontology data. Moreover, it contains three scripts used internally by SANA to evaluate the Go Average measure: `itgom.py`, `pairwise.py`, and `protein_pair_sim.py`. Moreover, it should contain the file `gene2go` (see section [b:goterms]).
- `sequence/`: this folder contains sequence data. It contains the following files and folders:
  - `PPI_get_FASTA.py`: A python script to download proteins sequences (see section [b:sequence]).
  - `get_all_FASTA.sh`: A bash script that executes `PPI_get_FASTA.py` for all the network from the BioGRID dataset.
  - `run_blast.sh`: A bash script to run BLAST.
  - `blast/`: the BLAST software.
  - `scores/`: output of BLAST for each pair of networks.
  - `bitscores/`: folder auto-generated by SANA.
- `matrices/`: auxiliary folder that SANA creates to store similarity matrices among species.
- `tmp/`: auxiliary folder where SANA stores temporary files, such as intermediate inputs and outputs of the scripts.

Compiling and executing

To compile, do `make` from the root project directory. To execute, do `./sana` from the same folder, followed by the arguments (see section [b:args]). Note that prior to running SANA, you should obtain the necessary biological data (see section [b:obtainbio]). The first time you run SANA with a new pair of networks, there will be a big overhead in the beginning because SANA will have to compute a lot of data in order to be able to evaluate all the measures. This data is stored for posterior runs.

During the execution of simulated annealing, SANA gives feedback about the progress of the alignment. The log looks something like this:

```
...
12 (23.106s): score = 0.00913156 P = 0.00687286
13 (30.805s): score = 0.010701 P = 0.00407068
14 (38.722s): score = 0.012009 P = 0.00228178
15 (46.697s): score = 0.0143891 P = 0.00120349
...
```

It shows, from left to right: the iteration divided by the N parameter of the restart schedule (by default, 10^7), the execution time, the score of the objective function, and the current probability to accept a worse solution with a typical energy increment for this pair of networks and objective function.

SANA outputs the alignment in a file. The first row contains the alignment itself, in a format that SANA understands. This way, it can be used as starting alignment for a different execution of SANA. The remaining of the file contains a report: the details of the networks, the method used to create it, the parameters used, the scores of all the measures, and the largest connected components of the common subgraph.

SANA arguments

There are optional arguments and mandatory arguments. The mandatory arguments have default values, which can be seen (and edited) in `src/arguments/defaultArguments.cpp`. The basic syntax is `./sana arg1 value1 arg2 value2 ...` The order of the arguments is not important. If an argument appears more than once, the last value is the only one that counts. All the arguments start with `-` and are all in lowercase (e.g., `-method`). There are four kinds of arguments, according to the type of value that they expect.

- String arguments: the value is a string. For instance, `-g1 yeast`
- Double arguments: the value is a number. For instance, `-alpha 0.5`
- Bool arguments: they are always optional and they don't have a value. For instance, `-restart`.
- Vector arguments: the value is a sequence of numbers. The first number indicates the size of the sequence, and the following numbers are the elements. For instance, `-nodecweights 4 0.1 0.25 0.5 0.15`

Next, we show the list of arguments[1]. For each argument, if and only if they are mandatory, we show its default value in `src/arguments/defaultArguments.cpp`.

[1] Some arguments for advanced functions have been omitted. The full list can be found in `src/arguments/supportedArguments.cpp`

Networks

- `-g1 yeast` (string): Specifies the name of G_1. The value should be the name of one of the folders in `networks`.
- `-g2 human` (string): Specifies the name of G_2. The value should be the name of one of the folders in `networks`, and it should have more nodes than G_1.

Method

- `-method sana` (string): its possible values are `lgraal`, `hubalign`, `sana`, `wave`, `random`, `tabu`, `dijkstra`, `netal`, `mi-graal`, `ghost`, `piswap`, `optnetalign`, `spinal`, `great`, `netalie`, `gedevo`, `greedylccs`, `magna`, `waveSim`, `none`, and `hc`.
- `-t 1` (double): maximum execution time for SANA, in minutes. If during the execution of SANA you input `Control+C`, the program will ask if you want to save the alignment as it is.
- `-alpha 0` (double): balance between topology and sequence similarity in the objective function of L-GRAAL and HubAlign.

Temperature schedule

- `-k 1` (double): k parameter of the temperature schedule, scaled by $2.5 \cdot 10^4$. It can receive a special value: `auto`. In that case the parameter is set automatically.
- `-l 1` (double): λ parameter of the temperature schedule, scaled by 10^8. It can receive a special value: `auto`. In that case the parameter is set automatically.
- `-startalignment` (string): file containing the starting alignment (in the format outputted by SANA). Some methods allow this option, while the rest start with random alignments. It is not implemented in SANA, although it could be done.

Restart scheme

- `-restart` (bool): active the restart scheme in SANA.
- `-tnew 3` (double): parameter t_1 of the restart scheme, in minutes.
- `-iterperstep 10000000` (double): parameter N of the restart scheme.
- `-numcand 3` (double): parameter K of the restart scheme.

- `-tcand 1` (double): parameter t_2 of the restart scheme, in minutes.
- `-tfin 3` (double): parameter t_3 of the restart scheme, in minutes.

Objective function

In SANA, the objective function can be any weighted combination of measures among EC, S3, WEC, and any local measures. Each measure can have a weight. If the sum of the different weights is not 1, they are scaled accordingly.

- `-ec 1` (double): weight of EC.
- `-s3 0` (double): weight of S3.
- `-graphlet 0` (double): weight of Graphlet similarity.
- `-graphletlgraal 0` (double): weight of Graphlet similarity, as defined in L-GRAAL.
- `-nodec 0` (double): weight of Node density.
- `-edgec 0` (double): weight of Edge density.
- `-importance 0` (double): weight of Importance.
- `-wec 0` (double): weight of WEC.
- `-sequence 0` (double): weight of Sequence similarity (in SANA, the contribution of Sequence similarity is tuned through the `-alpha` argument, not this).
- `-nodecweights 4 0.1 0.25 0.5 0.15` (vector): weights w of the Node density measure. They are automatically scaled to 1.
- `-edgecweights 4 0.1 0.25 0.5 0.15` (vector): weights w of the Edge density measure. They are automatically scaled to 1.
- `-wecnodesim nodec` (string): local measure used to weight the edges in the WEC measure.

Objective Scoring Combinations

In SANA, the objective score is the result of the measures above, and a scoring combination choice. The default is sum.

The possible values thus far are the following.

- `-score sum` (string): The default score combination, where each weighted measure is summed together for the cumulative score.

- `-score product` (string): Score combination where each weighted measure is multiplied together for the final score.
- `-score inverse` (string): Score combination method is set to value1.
- `-score max` (string): Score combination where swaps are made if any of the measures will increase with the swap.
- `-score min` (string): Score combination where swaps are made if no measure will decrease as a result of it.
- `-score maxFactor` (string): Score combination where swaps are made if the potential gain of a measure is great than the potential loss of another measure.

Other measures

- `-goweights 1 1` (vector): all the GO_k measures are added into a single measure, with different weights. This vector specifies the maximum GO_k measure and the weight of each one. For instance, `-goweights 2 0.5 0.5` means evaluate GO_1 and GO_2 in equal parts. The weights are automatically scaled to 1.
- `-goavg` (bool): indicates whether the GO Average measure should be incorporated in the report (this measure is optional because it requires a lot of time.)

  Go Average can only be evaluated with the yeast and human networks, as IT-GOM does not understand the protein identifiers from the BioGRID dataset.
- `-truealignment` (string): alignment file containing the "true" alignment. This is used to evaluate the NC measure. In its absence, NC assumes that the true alignment is the identity (the node with index i in G_1 is mapped to the node with index i in G_2). In any case, NC is only evaluated when G_1 and G_2 have the same size.

Others

- `-o` (string): Specifies the output file, relative to the project folder. In its absence, SANA gives it a relevant name automatically and stores it in the `alignments` folder (see section [b:struct]).

Examples

Default arguments

```
./sana
```

Executes SANA, aligning the `yeast` and `human` networks, optimizing S3, for 5 minute.

Slower temperature schedule

```
./sana -l 0.7 -k 2
```

Restart scheme

```
./sana -restart -tnew 20 -iterperstep 30000000 -numcand 15 -tcand 2 -tfin 10
```

One hour total execution time

Optimize EC

```
./sana -ec 1 -s3 0
```

Optimize L-GRAAL's objective function and use $\alpha = 0.5$

```
./sana -ec 0 -wec 0.5 -sequence 0.5 -wecnodesim graphletlgraal
```

Custom output file

```
./sana -o x
```

Stores the output in `x`.

Different node density weights

```
./sana -nodecweights 3 0 0 1
```

Networks from the Noisy yeast dataset

```
./sana -g1 0syn -g2 5syn
```

The networks are named `0syn`, `5syn`, `10syn`, `15syn`, `20syn`, and `25syn`, where the numbers indicate the amount of noise.

Networks from the BioGRID dataset

```
./sana -g1 RNorvegicus -g2 SPombe
```

L-GRAAL with $\alpha = 0.5$

```
./sana -method lgraal -t 60 -alpha 0.5
```

HubAlign with $\alpha = 0.3$

```
./sana -method hubalign -alpha 0.3
```

Objective function combination

```
./sana -ec 1 -ics 0.5 -s3 3 -graphlet 1 -edgec 0.5
```

GO_5 measure as objective function

```
./sana -ec 0 -go 1 -goweights 5 0 0 0 0 1
```

Evaluate GO Average measure

```
./sana -g1 yeast -g2 human -goavg
```

Formats

Networks

Networks in the `networks/` folder are specified in *graphWin* format. Another network file format sometimes used by other software is what I call "edge list". The file has one line per edge, each line consisting of two node names.

Internally, nodes are identified by their index (starting with 0) in the graphWin file, and not by their names. However, the `Graph` class has functions to translate between node names and indices. The indices are stored as `ushort` (alias for `unsigned short int`), to save space in the adjacency matrices and other data structures. This limits the maximum size of networks to 65536 nodes.

Alignments

When SANA finishes it saves the resulting alignment in the file named with the value of the argument `-o` (or with an automatically generated name, if this argument is not found). The alignment is contained in the first line. It is a list of n_1 numbers between 0 and $n_2 - 1$. If the i-th number is j, it means that the node with index i in G_1 maps to the node with index j in G_2.

The remaining of the file is just a description of the networks, the properties of the alignment (such as scores of different measures), and details about the method used to generate it.

Coding conventions

- Classes have two files named the same as the class, the header (`.hpp`) and the source (`.cpp`).
- Class files start with uppercase, while other files start with lowercase.

Source files

In this section, `.*pp` means that there is both a header and a source file.

- `utils.*pp` Mix of auxiliary functions that didn't belong anywhere else.
- `templateUtils.cpp` Same as `utils.cpp`, but with template functions.
- `Timer.*pp` A timer.
- `ArgumentParser.*pp` An argument parser.

- `NormalDistribution.*pp` Statistical functions about a normal distribution.

- `computeGraphlets.*pp` An algorithm to compute the graphlet degree vectors of a graph. I did not create this code myself, I just adapted it. It is free software (the license is in the code) so it can't be used in a closed version of SANA. It is called from the method `loadGraphletDegreeVectors()` from the `Graph` class, so it shouldn't require direct manipulation.

  To avoid recomputing the GDVs, the `Graph` class stores them in `networks/`*name*`/autogenerated/`*name*`_gdv.bin` (in binary format), where *name* is the name of the graph. When `loadGraphletDegreeVectors()` is called, first it is checked if the binary file already exists. If it does, it is loaded directly. Otherwise, it is computed and stored.

- `Graph.*pp` Networks.

  The function `getDistanceMatrix` returns the distance matrix. The first time it is called, it is computed and stored in `networks/`*name*`/autogenerated/`*name*`_distMatrix.txt`. When `getDistanceMatrix` is called, it is first checked if this file exists. If it does, the matrix is loaded directly instead of recomputing it.

  A special function of this class is

  ```
  static void GeoGeneDuplicationModel(uint numNodes,
      uint numEdges, string outputFile)
  ```

  which builds a graph following the GEO-GD expansion model (from the paper "Geometric evolutionary dynamics of protein interaction networks") with a certain number of nodes and edges, and stores it in graphWin format in the specified file.

- `Alignment.*pp` Alignment between two graphs. It does not have the graphs as attributes, so it receives them as arguments in many functions.

  Internally, alignments are stored as a vector A of `ushort`. $A[i] = j$ indicates that the node of G_1 with index i is mapped to the node with index j of G_2. One of the useful functions of this class to catch bugs is

  ```
  bool isCorrectlyDefined(const Graph& G1, const Graph& G2);
  ```

  which returns whether the alignment is correctly defined between G_1 and G_2. It is recommendable to always call this function before doing any analysis on an alignment.

- `randomSeed.*pp` Random seed generator.

Measures

- `Measure.*pp` Abstract base class (i.e., cannot be instantiated) for all the measures. It requires that all subclasses implement the function `eval(const Alignment& A)`, which is supposed to evaluate the alignment A and return a score.

  Its attributes are a name and two graphs. While measures (e.g., EC) "exist" independently of any two particular graphs, sometimes the `eval` function uses some data structures which can be precomputed for any two particular graphs, such as the similarity matrix in case of Sequence similarity. Since we don't want to recompute this matrix every time we call `eval`, this type of data structure are initialized when creating the measure. This requires that the measure has the graphs as attributes.

- `EdgeCorrectness.*pp` Subclass of `Measure`. Also known as EC, or Edge Coverage.

- `SymmetricSubstructureScore.*pp` Subclass of `Measure`. Usually called S3.

- `InducedSubstructureScore.*pp` Subclass of `Measure`. Usually called ICS.

- `LargestCommonConnectedSubgraph.*pp` Subclass of `Measure`. Usually called LCCS. Recall that there are two versions of LCCS (see a comment in the code for details). A constant bool `USE_MAGNA_DEFINITION` choses between the two.

- `NodeCorrectness.*pp` Subclass of `Measure`. Usually called NC. The constructor receives the "true" alignment as argument.

- `ShortestPathConservation.*pp` Subclass of `Measure`. A measure based on shortest paths. It is the only measure that is not normalized, and in addition, you want to minimize it instead of maximizing.

- `InvalidMeasure.*pp` Subclass of `Measure`. It is a dummy class that does nothing. It is used in places where a Measure is technically required but not actually used.

- `LocalMeasure.*pp` Subclass of `Measure`. It is also an abstract base class (i.e., cannot be instantiated) for all the local measures. What all local measures have in common is that they have a similarity matrix, and that the score is the average similarity of the aligned nodes. The subclasses of `LocalMeasure` don't need to define `eval`, they only need to define `initSimMatrix()`, which creates the similarity matrix. Some similarity matrices may be costly to compute, so the first time they are computed they are stored in `matrices/autogenerated/` with a unique name and in binary format. When a local measure is instantiated, first it is checked if

the file exists, and if it does it is loaded instead of recomputed. Since this process is common for all local measures, `LocalMeasure` handles it.

- `GraphletSimilarity.*pp` Subclass of `LocalMeasure`. Graphlet similarity measure as defined in the GRAAL paper.
- `GraphletLGRAAL.*pp` Subclass of `LocalMeasure`. Graphlet similarity measure as defined in the L-GRAAL paper.
- `Importance.*pp` Subclass of `LocalMeasure`. Importance measure used in HubAlign.
- `EdgeCount.*pp` Subclass of `LocalMeasure`.
- `NodeCount.*pp` Subclass of `LocalMeasure`.
- `Sequence.*pp` Subclass of `LocalMeasure`. It uses BLAST bit-scores.

  In order to initialize its similarity matrix, the BLAST output is necessary. It is expected to be `sequence/scores/`*g1Name*`_`*g2Name*`_blast.out`. If it is not found, the program halts. The BLAST output is not generated automatically by SANA, as it can take hours. Instead, there are some scripts in `sequence/` to run BLAST and obtain these files (see Section [b:sequence] for details).

  In the case of the Yeast and Human networks, `Sequence` has to translate the names of the nodes in the network to the names that appear in the BLAST output files. This translation is used by looking at the `networks/yeast/sequence/yeast_curated.fasta` file (analogous for the Human network).

- `GoSimilarity.*pp` Subclass of `LocalMeasure`. This is a weighted combination of all the GO_k measures. The constructor receives a vector `countWeights` that determines the relative weight of each GO_k measure. For instance, if the vector is (1), then the corresponding measure is GO_1; if it is $(0, 1)$, the corresponding measure is GO_2; if it is $(1, 1)$, then it is the average of both.

  In order to initialize its similarity matrix, the GO terms of each node are necessary. As explained in Section [b:goterms], the corresponding data is expected to be found in different files for different datasets. In all cases, the corresponding file is identified (if the file is not found, the program halts), parsed to remove all the unnecessary data, and stored in `networks/`*name*`/autogenerated/`*name*`_go_simple.txt`. Next, this file is parsed again (as usual, if it doesn't exist yet it is first generated) and the data is transformed again to a different format that is more convenient (the GO terms are replaced by their numbers and grouped by proteins, and the proteins names are replaced by their indices). The result is stored in `networks/`*name*`/autogenerated/`*name*`_go_internal.txt`. This is the file loaded to compute the similarity matrix (if it doesn't exist yet it is first generated).

As defined in MAGNA, this local measure is special because instead of being the average similarity of the aligned nodes, it is the sum of the similarity of the aligned nodes divided by the number of proteins with at least one GO term in G_1 (or the number of proteins with at least one GO term in G_2, if this number is smaller). However, this is inconvenient because it is useful to combine several local measures by combining their similarity matrices, but that only works if they are all evaluated the same. Therefore, we use the traditional definition, the average similarity of the aligned nodes (the reimplementation of `eval` is in the code but it is commented).

- `GoCoverage.*pp` Subclass of `Measure`. Note: even though this measure can be implemented as a local measure, that has not been done yet (it was on the to-do list).

  GO Coverage measure. It uses (among others) the public function `loadGOTerms` from the sister class `GoSimilarity`, which returns the list of GO terms of every protein. In this way, only the `GoSimilarity` class has to interact directly with the auxiliary GO files.

- `GoAverage.*pp` Subclass of `Measure`. Note: even though this measure can be implemented as a local measure, that has not been done yet.

  GO Average measure. This measure was used in MAGNA. It calls the scripts `go/pairwise.py`, `go/itgom.py`, and `go/protein_pair_sim.py`, which were created by Vikram. These scripts query a GO-term semantic similarity analysis website.

  The scripts take very long to finish. The data could be stored to speed the process for subsequent executions, but this has not been implemented yet, as it is a measure that we have not used much and it only works with the Yeast and Human networks.

- `GenericLocalMeasure.*pp` Subclass of `LocalMeasure` This is a local measure that is not any particular one. Instead, it receives the similarity matrix directly in the constructor. It is used, for instance, to combine several local measures into one.

- `WeightedEdgeConservation.*pp` Subclass of `Measure`. Also known as WEC, or Weighted Edge Coverage. The constructor receives a local measure that is used to give weights to the edges.

- `MeasureCombination.*pp` This class is like a measure made of a combination of measures. It has a vector of pointers to measures and a corresponding vector of weights. It is used because sometimes it is necessary to iterate through all the measures (for instance, to write all the scores to the report). It is also useful to specify the objective function of SANA, which is a weighted combination of measures. SANA receives a `MeasureCombination` as argument in the constructor.

Methods

- `Method` Abstract base class (i.e., cannot be instantiated) for all the network alignment methods (such as SANA). The main method of this class is `run()`, which must be implemented by the subclasses. This function returns an alignment. The constructor is

  ```
  Method(Graph* G1, Graph* G2, string name);
  ```

  Even though the methods "exist" independently of the graphs G_1 and G_2, the constructor requires the graphs so that methods can initialize any data structures needed for `run()` (analogous to how for measures we want to do any preprocessing necessary for `eval()` just once, for methods we want to do any preprocessing necessary for `run()` just once).

  There are two other methods that subclasses must implement. The first is `fileNameSuffix(const Alignment& A)`. If when running SANA the argument `-o` to specify the output file is not supplied, a name is generated automatically:
  `alignments/`*G1Name_G2Name*`/`*G1Name_G2Name_method_suffix*`.txt`
  where *method* is the name of the method, and *suffix* is what `fileNameSuffix()` returns, and is supposed to give additional information about the details of the alignment. For instance:

  ```
  yeast_human_SANA_EC_048.txt
  yeast_human_LGRAAL_alpha_0.txt
  ```

  Note: if the automatically generated name already exists, a numerical suffix is added automatically.

  The other function subclasses of `Method` must implement is `describeParameters()`, which should describe the parameters used in this particular execution of the method (for instance, the values of k and λ in the case of SANA). This information is added to the report in the output file.

- `SANA.*pp` Subclass of `Method`. The code is quite complex because it has many features (incremental evaluation, combination of objective functions, restart scheme, automatic setting of temperature schedule parameters). However, I tried to comment it adequately.

  By default, SANA is initialized without restarting scheme. It can be enabled with `enableRestartingScheme(...)`. Similarly, automatic temperature schedule can be enabled with `setKAutomatically()` and `setLAutomatically()`. SANA sets values of k and λ when initialized, but these functions override them.

  Note: although in the paper it is explained that the α parameter controls the weight of Sequence similarity versus topology, in the code there is no

explicit α parameter. SANA has a `MeasureCombination` attribute which has the weight of each measure in the objective function. The effect of α can be emulated with these weights.

In order to know what is a "typical energy increment" for a particular pair of networks and objective function, SANA first runs itself with temperature fixed to 0, which is usually quick, and uses this execution to sample energy increments. Hence, when running SANA, it might seem like it runs twice.

- `HillClimbing.*pp` Subclass of `Method`. It is very similar to SANA, but it implements steepest ascent hill-climbing. In other words, it looks at every neighbor at each iteration, and chooses the best, until it can't improve anymore. It is a very slow method.

- `HubAlignWrapper.*pp` Subclass of `Method`. It runs HubAlign, which is expected to be in `HubAlign/HubAlign` (if it isn't, the program halts). The purpose of this class is to give a uniform interface for running SANA and HubAlign: HubAlign expects the networks and sequence data in a different format, outputs the alignment in a different format, and requires other several parameters, which the class fixes to the values specified in the paper.

  Moreover, in HubAlign the α parameter has the opposite meaning as usual (it is the weight of topology, not sequence), although this is accounted for in the `main.cpp` file.

- `LGraalWrapper.*pp` Subclass of `Method`. Analogous to `HubAlignWrapper` for L-GRAAL. In this case, the software is expected to be in `LGRAAL/L-GRAAL.exe`. The program `LGRAAL/ncount4.exe` is also executed, which they use to compute GDVs. The output of this program, which is part of the input of `L-GRAAL.exe`, is stored for quick access in `networks/`*name*`/autogenerated/`*name*`_lgraal_gdvs.ndump2`.

- `RandomAligner.*pp` Subclass of `Method`. It returns a random alignment.

- `GreedyLCCS.*pp` Subclass of `Method`. This method is bugged. It is supposed to optimize LCCS in a greedy fashion. It is in the to-do list to fix it (or reimplement/delete it).

- `NoneMethod.*pp` Subclass of `Method`. It is a dummy class that does nothing. It is used in places where a method is technically required but not actually used.

- `WeightedAlignmentVoter.*pp` Subclass of `Method`. An implementation of WAVE.

The files `Experiment.*pp`, `ParameterEstimation.*pp`, `AlphaEstimation.*pp`, and `ComplementaryProteins.*pp` are not described because they are complicated and serve very specific purposes.

Modes

`AnalysisMode.*pp`

`ClusterMode.*pp` Used in submission to cluster.

`DebugMode.*pp` Used for debugging purposes.

`NormalMode.*pp` Default mode used in running

`SimilarityMode.*pp` Used in creating similarity files.

Main program

The main program is in `main.cpp`. **The project folder is hard-coded in this file, and it is necessary to update it for the program to run correctly**.

The main program does the following:

1. The arguments are parsed.
2. If the bool argument `-qsub` is present, then the execution is sent to the cluster. A small bash script `tmp/submit.sh` (with a unique suffix if necessary) is created and executed. This script submits SANA to run in the cluster with exactly the same arguments (except `-qsub`). Moreover, the `-qcount` argument allows you to submit it more than once.
3. The arguments and their values are printed in the terminal, so that the user can inspect what is being executed.
4. If any of the arguments `-experiment`, `-paramestimation`, or `-alphaestimation` is present, then the corresponding experiment is executed.
5. Any folder that will be used by the program is created if it did not already exist, such as `tmp/`.
6. The graphs are read and their data structures initialized.
7. If the double argument `-rewire` is greater than 0, the corresponding fraction of edges in G_2 is randomly rewired. For instance, with `-rewire 0.25`, 25% of the edges will be rewired. This is useful when aligning a network vs a noisy version of itself.

   Note: there are also functions to add and remove edges randomly, but there is no argument for those.
8. If the bool argument `-dbg` is found, the execution flow is passed to the function `dbgMode`. This function does not do anything in particular, it is just used to try things out of the usual. It is not meant for anything permanent, and it is overwritten as needed.

9. The measures are initialized. The first time SANA is executed, this step takes a long time. The following times, it takes a fraction of the time because most of the data is stored.

   Some measures are only initialized if certain preconditions are met (for instance, for Importance there must at least exist one node with degree greater than 10 in each network).

10. The method that is going to be run is initialized.

11. The method is executed.

12. The correctness of the alignment is checked.

13. The report is generated and saved.

Extending the software

When adding measures or methods, it is useful to look at already existing examples.

Adding a new measure

This requires creating a new class that is a subclass of `Measure` or `LocalMeasure`.

In the first case, one must implement the constructor and `eval()`. The constructor should receive at least G_1 and G_2 as parameters. It should call the constructor of `Measure` with G_1, G_2 and the name (or acronym) of the measure. For instance:

```
EdgeCorrectness(Graph* G1, Graph* G2) : Measure(G1, G2, "ec") {}
```

In the second case, one must implement the constructor and `initSimMatrix()`.

The function `initSimMatrix()` should compute the similarity matrix of this measure and set it in the attribute `vector<vector<float>> sims` of the superclass. The constructor should do exactly three things:

1. Call the constructor of `LocalMeasure`, e.g.:

   ```
   Importance(Graph* G1, Graph* G2) : LocalMeasure(G1, G2, "importance")
   ```

2. Choose a name for the file to store the similarity matrix at; it is typically `matrices/autogenerated/G1Name_G2Name_suffix.bin`, where `suffix` could be the name of the measure.

3. Call `loadBinSimMatrix(file)`, where `file` is the name of the file of the similarity matrix. This function from the `LocalMeasure` superclass looks up the file, loads the matrix if it already exists, or calls `initSimMatrix()` (which is reimplemented by the subclass) and then stores the similarity matrix.

Once the new class has been created, in order for it to be evaluated and added to the final report, there must be some modifications made in the file `main.cpp`:

- Include the header file.
- In the function `initMeasures(...)`, initialize a pointer to an instance of the measure and add it to the `MeasureCombination` variable `M`, e.g.:

```
Measure *m;
m = new LargestCommonConnectedSubgraph(&G1, &G2);
M.addMeasure(m);
```

In addition, if you have modified SANA so that it can optimize that measure (more on that latter) or if it is a local measure (all local measures can be optimized by SANA by default), and you want SANA to be able to optimize it, you must do the following:

- Add a double argument to the list of arguments (it appears in the start of the file) with the name of the measure, e.g., `-ec`.
- In `initMeasures(...)`, add it to `M` with the value of the argument (i.e., the weight):

```
m = new EdgeCorrectness(&G1, &G2);
M.addMeasure(m, args.doubles["-ec"]);
```

Note: since G_1 and G_2 are already protected attributes of `Measure`, it is not necessary for the subclasses to have them as their own attributes.

Adding a new method

This requires creating a subclass of `Method`. It is necessary to implement the constructor, `run()`, `describeParameters()` and `fileNameSuffix()` (see Section [methods]). Similar to `Measure`, the constructor of `Method` expects the networks and a name for the new method.

Once the new class has been created, in order to be able to use it, there must be some modifications made in the file `methodSelector.cpp`:

- Include the header file.
- In the function `initMethod(...)`, add a new if case, which compares the variable `name` to the name of the method and returns a new instance of the method, e.g.:

```
if (name == "random") {
  return new RandomAligner(&G1, &G2);
}
```

Once this is done, you can execute the new method with `./sana -method name`.

Note: since G_1 and G_2 are already protected attributes of `Method`, it is not necessary for the subclasses to have them as their own attributes.

Adding a new wrapper method

This requires creating a subclass of `WrapperMethod`. It is necessary to implement the constructor, `generateAlignment()`, `loadDefaultParameters()` and `loadAlignment()` (see Section [methods]). Similar to `Measure`, the constructor of `Method` expects the networks and a name for the new method.

Once the new class has been created, in order to be able to use it, there must be some modifications made in the file `methodSelector.cpp`:

- Include the header file.
- In the function `initMethod(...)`, add a new if case, which compares the variable `name` to the name of the wrapper method and returns a new instance of the method, e.g.:

```
if (name == "random") {
  return new RandomAligner(&G1, &G2);
}
```

Once this is done, you can execute the new method with `./sana -method name`.

Note: since G_1 and G_2 are already protected attributes of `Method`, it is not necessary for the subclasses to have them as their own attributes.

Adding a new objective function to SANA

You don't need to modify the `SANA` class to add a new local measure. The changes described above to the `main.cpp` file suffice.

For other measures, many more changes are necessary to `SANA.hpp` and `SANA.cpp`. In the simplest case, you would need to do the following. Let the measure be x.

1. Add a private attribute `xScore` to the `SANA` class.

2. The function `initDataStructures(const Alignment& startA)`, which is called prior to starting the iterations of SA, prepares all the data structures necessary to run SANA starting with the alignment `startA`. Here, the value of `xScore` should be initialized, i.e., `xScore = x.eval(startA)`, where `x` is an instance of the measure x.

3. Modify `performChange()` and `performSwap()` to compute it incrementally. Both functions follow the same structure, and both require the same additions:

   (a) The new x score should be computed in a local variable `xNewScore`.

   (b) The new current score should be computed accounting for the new x score and the weight of x:

   ```
   double newCurrentScore = 0;
   newCurrentScore += ecWeight * ecNewScore;
   newCurrentScore += s3Weight * s3NewScore;
   ...
   newCurrentScore += xWeight * xNewScore;
   ```

   Note that the weight of x can be obtained as `MC->getWeight(x)`, where `x` is the name of the measure. MC is an attribute of SANA of type `MeasureCombination` which has the weight of each measure.

   (c) If (and only if) the new solution is accepted, update `xScore` and any data structures that are necessary to compute it incrementally. This has to be done inside the following `if` (it is the same in both `performChange()` and `performSwap()`):

   ```
   if (energyInc >= 0 or randomReal(gen) <= exp(energyInc/T)) {
     ...
     xScore = xNewScore;
     ...
   }
   ```

Analysis of complementary homolog proteins

The files `ComplementaryProteins.*pp` contain function to do this analysis. The file containing the raw data about which protein pairs are complementary or non-complementary is found in `sequence/complementProteins.txt`.

Note that this analysis can be done with the Yeast and Human networks or with the SCerevisiae and HSapiens networks (from the BioGRID dataset).

The first useful function is `printProteinPairCountInNetworks(bool BioGRIDNetworks)`, which prints how many of the protein from `sequence/complementProteins.txt`

actually appear in the networks. The bool `BioGRIDNetworks` distinguishes between the two possible pairs of networks. Its output (with `BioGRIDNetworks = false`) is

```
The yeast network contains 138 of the 200 complementary yeast proteins.
The yeast network contains 163 of the 224 non-complementary yeast proteins.
The human network contains 137 of the 200 complementary human proteins.
The human network contains 150 of the 224 non-complementary human proteins.

103 of the 200 pairs of complementary proteins appear in both networks.
119 of the 224 pairs of non-complementary proteins appear in both networks.
```

The second useful function is `printComplementaryProteinCounts(const Alignment& A, bool BioGRIDNetworks)`, which analyses how many complementary and non-complementary pairs the alignment A has aligned. An example of output (with `BioGRIDNetworks = false`) would be:

```
Complementary homolog proteins aligned: 0/103 (0%)
Non-complementary homolog proteins aligned: 0/119 (0%)
```

SANA does not currently have any argument to call these functions. They are called from the `dbgMode(...)` function using the `-dbg` argument. For instance, the following code analyses the alignment `alignment.txt` given in edge-list format.

```
void dbgMode(Graph& G1, Graph& G2, ArgumentParser& args) {
  printProteinPairCountInNetworks(false);
  Alignment A = Alignment::loadEdgeList(&G1, &G2, "alignment.txt");
  printComplementaryProteinCounts(A, false);
  exit(0);
}
```

SANA should be run as `./sana -dbg -g1 yeast -g2 human`. If `alignment.txt` were an alignment between SCerevisiae and HSapiens, then SANA should be run as `./sana -dbg -g1 SCerevisiae -g2 HSapiens`, and the code should be modified as:

```
void dbgMode(Graph& G1, Graph& G2, ArgumentParser& args) {
  printProteinPairCountInNetworks(true);
  Alignment A = Alignment::loadEdgeList(&G1, &G2, "alignment.txt");
  printComplementaryProteinCounts(A, true);
  exit(0);
}
```