Goals

What a scripting language is?

A **scripting language** is a programmable language that supports scripts, namely programs written for a special **run-time environment** that **automate** the execution of tasks that could alternatively be executed **one-by-one** by a human operator.

Which run-time environment will we use?

Bash, the most common Unix textual shell that also provide a **bash programming** language. Bash is different from sh.

What will we learn?

We will learn how to **automate extensive computational tasks** (i.e. running programs, dealing with their outputs and making pipelines).
We will learn **batch processing:** the execution of a series of jobs in a program on a computer without manual intervention.

How will we do it?

We will make use of bash programming and python(3) scripts in order to build (bioinformatics) **pipelines** that transform **raw data**, execute **programs**, and present **results**.

We will focus on **real applications**, such as testing a given algorithm, creating and managing input raw data for testing it, and handle its results.

Materials

Website
Http://….

Documentation?
Course books
Man pages: `man <command>`
Bash reference manual: `http://www.gnu.org/software/bash/manual/html_node/`
Bash how-to: `http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`
Python3 reference page: `https://docs.python.org/3/reference/`
Python3 official tutorial: `https://docs.python.org/3/tutorial/`
Every other resource on the web (knowledge is infinite)

Suggested IDEs (Integrate Development Environment) and environments
Ubuntu/Mac OSX
Any kind of editor (i.e. gedit, nano, vi, vim or Komodo edit free version)

Python3 (the python3 environment) and ipython3 (an interactive shell for programing in python)
How to install under Ubuntu:
- `sudo apt-get install python3`
- `sudo apt-get install ipython3`

Bash: hello World

Hello World

Create a file named hello.sh

```
#!/bin/bash
# This is a comment
echo "Hello World"  # this is another comment
```

Digit `chmod -rx hello.sh` in order to set the file to be executable
Run the script `./hello.sh` or bash `hello.sh`
If `./hello.sh` is called, the first line of the script indicates which program (in our case bash) has to be used to interpret and run the script.

```
This is another hello word example
#!/bin/bash
# This is a comment
echo "Hello World"  #this is another comment
echo "Hello    World"
echo hello World
echo "hello" World
echo 'hello' World
echo -n "Hello World" # does not make a new line
echo ": you\'r beautiful" # use \ for character escaping
```

Stay careful, space characters are important!!!

Bash: variables

Setting and printing variables

```
#!/bin/bash
say=Hello
location="World"
echo $say ${location}
echo "${say}" "$location"
```

Reading from standard input

```
#!/bin/bash
echo "What's your name?"
read name
echo "Welcome $name"   "
```

Bash: variables

Numeric variables and integer expressions

```
#!/bin/bash
number=1   # this can be interpreted as a numeric value
expr $number + 1 # prints on screen the result of the expression, but does not modify the variable
echo $number # in fact, this prints 1 instead of 2
number=`expr $number + 1` # prints nothing but assigns the new value to the variable
echo $number # well, now it prints 2
```

Arithmetic expansion construct

```
(( number = number++ ))
echo $number
```

This construct **evaluate and/or test** numerical expressions

Floating point numbers and the arbitrary precision calculator

The `expr` command can be used for integer expressions. For dialing with floating point number you need to use the `bc` command. For example:

```
echo "20 / 30 * 100" | bc -l
echo "$(echo '3.4 + 2.2' | bc)"
```

Bash: indexed arrays

Setting and accessing one-dimensional indexed arrays

Indexed arrays can be accessed only through numeric indexes, namely positive numbers, and only one-dimensional arrays are allowed.

```
#!/bin/bash
my_array[0]=Hello
my_array[1]="World"
echo ${my_array[0]} $my_array[1]

my_array=(1 2 3 4)
echo "the length of the array is: ${#my_array[@]}"
echo "the second and third elements are: ${my_array[@]:1:2}"

unset my_array[0] # leaves position 0 empty
echo "the new length of the array is: ${#my_array[@]}"
echo "the first element is: ${my_array[0]}"
echo "the second element is: ${my_array[1]}"
```

Bash: associative arrays

Setting and accessing associative arrays

An associative arrays can be indexed by any kind of “key”. It is a data structure that associate keys to values. It can not contains duplicated keys.

```
#!/bin/bash
declare -A my_array

my_array[first]="first element"
my_array["second"]="second element"
my_array[2]="third element"

echo ${my_array[first]}
echo ${my_array["first"]}

echo ${my_array[2]}
echo ${my_array["2"]}

echo ${my_array[0]}
```

Keys are strings, thus `my_array[2]` is equal to `my_array["2"]`. On the contrary, associative arrays have not positional indexes, thus `my_array[0]` is not the first element but it is the element having `"0"` as key.

Variables can be used in order to specify **keys**.

```
my_key="second"
echo ${my_array[${my_key}]}
```

The list of **keys** can be retrieved by `${!array[@]}`

Bash: if-elif-else-fi statement

Conditional statements and operators

```
if [ condition ]
then
    echo "do something"
elif [ condition ]
then
    echo "do something else"
else
    : # do nothing
fi
```

The `elif-then` and `else` statements are optional, and multiple `elif-then` can be specified.

In `[ condition ]`, boolean expressions can be evaluated by means of a variety of available operators. Actually, `[ condition ]` corresponds to the command `test`. The arithmetic expansion operator can be used too.

```
i="abc"
j="abc"
[ $i == $j ]  # string comparator
echo $? # recall the last evaluation
if test $i == $j; then
    echo "equals"
fi
```

```
i=0
j=1
[ $i -eq $j ]  # numeric comparator
echo $?
if (( $i < $j )); then
    echo "less than"
fi
```

Bash: comparison operators

Test operators

String comparison

`if [ "$a" == "$b" ]` two string are equal
`if [ "$a" != "$b" ]` two string are different
`if [[ "$a" < "$b" ]]` or `[ "$a" \< "$b"]` for testing ASCII alphabetical ordering
`if [ -z "$a" ]` the string is null

Numeric comparison

`if [ "$a" -eq "$b" ]` test for number equality, `$a` is equal to `$b`
`if [ "$a" -ne "$b" ]` is not equal
`if [ "$a" -gt "$b" ]` is greater than
`if [ "$a" -ge "$b" ]` is greater than or equal to
`if [ "$a" -lt "$b" ]` is less then
`if [ "$a" -le "$b" ]` is less than or equal to

```
...
if (("$a" < "$b"))
...
```

Bash: comparison operators

Test operators

Boolean composition

`[[ condition1 && condition2 ]]` logical `AND`
`[ condition1 -a condition2 ]` logical `AND`
`[[ condition1 || condition2 ]]` logical `OR`
`[ condition1 -o condition2 ]` logical `OR`

File test operators

`if [ -e "$myfile" ]` test if file exists
`-f` the file is a regular file (not a directory or a device)
`-s` the file is not zero size
`-d` the file is a directory
…

Testing regular expressions

```
if grep -q "regex" my file; then
```

The `-q` parameter is mandatory if `grep` is used within `if` conditions.

Bash: loops

For-statement

```
for variable [in list]
do
    # do something
done
```

```
for ((expr1; expr2; expr3))
do
    # do something
done
```

Examples

```
list="a b c d"
for e in $list
do
    echo $e
done
```

```
for f in `ls *`
do
    if [ -f $f ]; then
        echo "$f is a regular file"
    fi
done
```

```
for i in "${!array[@]}"
do
  echo "key :" $i
  echo "value:" ${array[$i]}
done
```

```
for ((i=0; i<10; i++))
do
    echo $i
done
```

Bash: loops

While-statement

```
while [positive condition]
do
    # do something
done
```

Examples

The following pieces of code are equivalent

```
i=0
while (( i< 10 ))
do
    echo $i
    (( i = i + 1 ))
done
```

Until-statement

```
until [negative condition]
do
    # do something
done
```

```
i=0
until (( i >= 10 ))
do
    echo $i
    (( i = i + 1 ))
done
```

Break keyword

```
break
```

```
for ((i=0; ; i++))
do
    echo $i
    if ((i == 9)); then
        break
    fi
done
```

Bash: pipes and I/O redirection

Redirection means capturing output from a file, command, program, script, or even code block within a script and sending it as input to another file, command, program, or script.

Input/output redirection from /to files

Redirect standard streams (input, output and error).

```
command > file          write the output of the command into file.
                        If file exists then it is completely overwritten
command >& file         redirect stdout/stderr to file
command 1> file         redirect stdout to file
command 2> file         redirect stderr to file
command >> file         redirect stdout to file, but append to it
command [parameters] < file   get stdin from file
command 2>&1          redirect stderr to stdout
```

Pipes

Similar to ">" but more general. It is useful for chaining commands, scripts, files and programs together.

```
cat *.txt | sort | uniq > result-file
```

Sorts the output of all the .txt files, deletes duplicate lines, and save the result to "result-file".

```
cat *.txt | sort | uniq | tee result-file
```

Bash: redirection within scripts

Read a file line by line

```
#!/bin/bash
for line in `cat $1` # equivalent to  for line in $(cat $1)
do
    echo $line
done
```

```
cat filename | while read line
do
    echo $line
done
```

```
while read line
do
    echo $line
done < "$1"
```

`$1,…,$n` are used to access the arguments that have been given to the execution of the script, i.e. `./my_script.sh arg1 arg2 arg3`
Moreover
`$0` is the file name of the running script
`$#` is the total number of given input arguments

Bash: functions

Functions

```
function name {
    # commands, also called code block of the function
}
```

Function parameters are available by `$1,….$n`, like it is a single script that is independent from the other part of the current script file.
The return value may be returned by the key `return`.

Including external scripts

The construct `source [list of files]` allows for including external sources.
It reads the listed files and “execute” them.

Variable scope

A variable that is **explicitly declared as local is visible only within the block** of code in which it appears, **otherwise every variable is global** (even if the source keywords is used to include external files).
Example by calling (see next slide) `./script1.sh`

Bash: functions

```
#!/bin/bash
source script2.sh
# prints a,,,3,  because arguments of script1.sh are visible here
echo "$1,$glo_var1,$glo_var_2,$glo_var_3,$int_var"
# prints a,,,3,

glo_var_1=1
function foo {
    glo_var_2=2
    local int_var=3
    echo "$1,$glo_var1,$glo_var_2,$glo_var_3,$int_var"
    # prints p,,2,3,3
}
foo "p" # calling foo with parameter "p"
echo "$1,$glo_var1,$glo_var_2,$glo_var_3,$int_var"
# prints a,,2,3,
foo2 "e"
# prints e,,2,3,
```

```
#!/bin/bash      # file script2.sh #
glo_var_3=3
echo "$1,$glo_var1,$glo_var_2,$glo_var_3,$int_var"
function foo2 {
    echo "$1,$glo_var1,$glo_var_2,$glo_var_3,$int_var"
}
```

Bash: regular expression

Bash provides a set of rules for creating searching patterns, for example useful in finding occurrences of strings in files. Formally, a **regular expression** is a pattern that describes a set of strings.

Some simple rules

- `.` identifies a single character
- `^` line start
- `$` line end
- `\` for escaping meta-characters, es `\^`
- `[]` everything inside the brackets, i.e. `[0-9]` is any digit counted once
- `[^]` everything except what inside the brackets, i.e. `[0-9]` everything is not a digit

Modifiers:

- `?` zero times or once
- `+` at least once
- `*` zero or more times
- `|` logic OR

Bash: useful commands

Grep

Search a patterns within a give list of files. It prints to stdout the rows containing the pattern.

```
grep [options] pattern [file list]
```

`grep message myfile` or `grep "message" myfile` or `grep "^[0-9]+$" myfile`

Some options

- `-c` counts the number of rows where the patter occurs
- `-i` case insensitive
- `-l` print only the name of the files containing the pattern
- `-v` logical NOT
- `-q` quite, do not print something but return with a specific exit code depending on the fact that the pattern have been found or not

Sed (awk)

Search and replace a pattern.

`sed s/patter/replacement/g file` in a **very simple form!**

It prints to stdout the rows containing the pattern after the replacement has been applied. If the `-i` option is used then it directly modify the given file.

```
sed s/my\ message/your\ message/g my_file.txt
sed 's/$myvar/something/g'
```

Encapsulate the patter-replacement between two ' for using bash variables

Bash: useful commands

Cut

Split rows of a file into columns by searching for a given delimiter character. By default the searched character is a tab.

```
cut [options] -f[field indexes] file
```

It is useful for example for splitting CSV files into columns. The first column has index equal to 1.

The option `-d` specify the delimiter character used for splitting.

```
cut -d" " -f1,3,4 my_file
```

Splits rows by making use of a single space character as delimiter and outputs the first, third and fourth columns.

Bash: useful commands

Sort

Sort rows in a given set of files or coming from stdin.

```
sort [options] file1 file2 filen …
```

Some options

- `-b` ignore spaces
- `-d` alphabetical mode, take into account only letters, digits and spaces
- `-f` case insensitive
- `-o` file print to the given file
- `-r` reverse ordering
- `-t` character use a given character as delimiter
- `-k S1,S2,Sn` use the specified fields of the row (similar to `cut`)

Uniq

Remove **contiguous** duplicate rows in a file or from stdin. Contiguous means that two identical rows that are not adjacent are not considered duplicate.

```
uniq [-u] [-c] file
```

- `-u` show only non duplicate row
- `-c` show a counter of removed rows, too

Use the following pipe for **removing any duplicate row**, even not contiguous

```
sort myfile | uniq
```

Bash: useful commands

Head and tail

These two commands print a specified number of leading/trailing rows of a file or of the stdin.

```
head -n10 myfile.txt | tail -n5
```

`tail -f myfile.txt` wait for new data from the file

Cat

Concatenate files and print on the standard output. It can also be used for printing a file on the stdout.

```
cat file1 file2 >> file3
```

Wc

Counts rows, words or bytes of a give list of files.

```
wc [-l,-m,-w,-c] myfile # counts lines, chars, words or bytes
nof_lines=`wc -l myfile | cut -d" " -f1`
```

Touch

Designed for changing file timestamps, it is useful for ensuring the existence of a file. In fact , if a file does not exists, `touch` will create a new one. In this way we can avoid file existence test in bash scripts

```
#!/bin/bash
touch myfile
# if [ -f myfile ] # grep can only be applied to existent files
grep "pattern" myfile
```

Bash: useful commands

Miscellaneous

List file with having a specific name suffix but more than one possible prefix.

```
ls {A,B}*.txt  # list all A*.txt or B.txt files
```

```
more myfile
```

Navigate a file only forwards, but print current read percentage.

```
less myfile
```

Allows for forward and backward navigation, but do no print percentage.

```
mkdir -p /dir1/dir2/dir3/…/dirn
```

Create, if they do not exist, all the directories of the given path.

```
timeout [options] [duration] command [arg]
```

Run a command with a time limit.

```
/usr/bin/time [options and output format] command
```

Measure running times, memory usage, and other things, used by the command. Arguments can not be passed to the command. **The reported running time are not so precise, and there are some well-known bugs regarding the reported memory usage.**

`diff` and `comm` for file comparison

Bash: useful commands

Miscellaneous

`mktemp` for generating temporary files, usually stored in `/tmp/`

`ssconvert` a command line spreadsheet format converter
Available by `sudo apt-get install gnumeric`

`xls2csv` a converter from xls (microsoft excel 2010 format) to CSV
Available by `sudo apt-get install catdoc`

`wget` a non-interactive internet downloader

`basename` get the file name and exclude the complete path

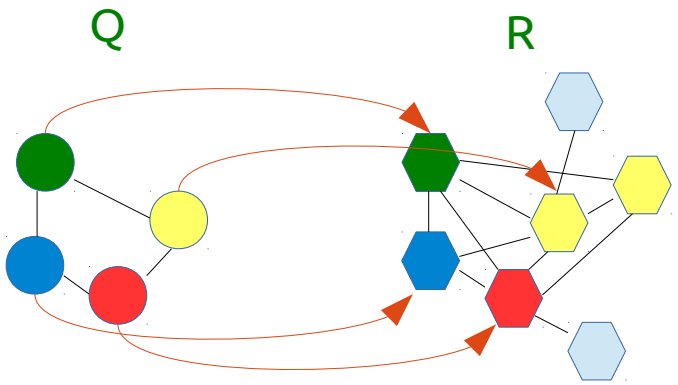
Our real case

SubGraph Isomorphism (SubGI)

A **Graph** is a quadruple **(V, E, α, β)** where
- **V** is the set of vertices
- **E : V**x**V** is the set edges (arcs)
- **α** a function which maps vertices to labels
- **β** a function which maps edges to labels

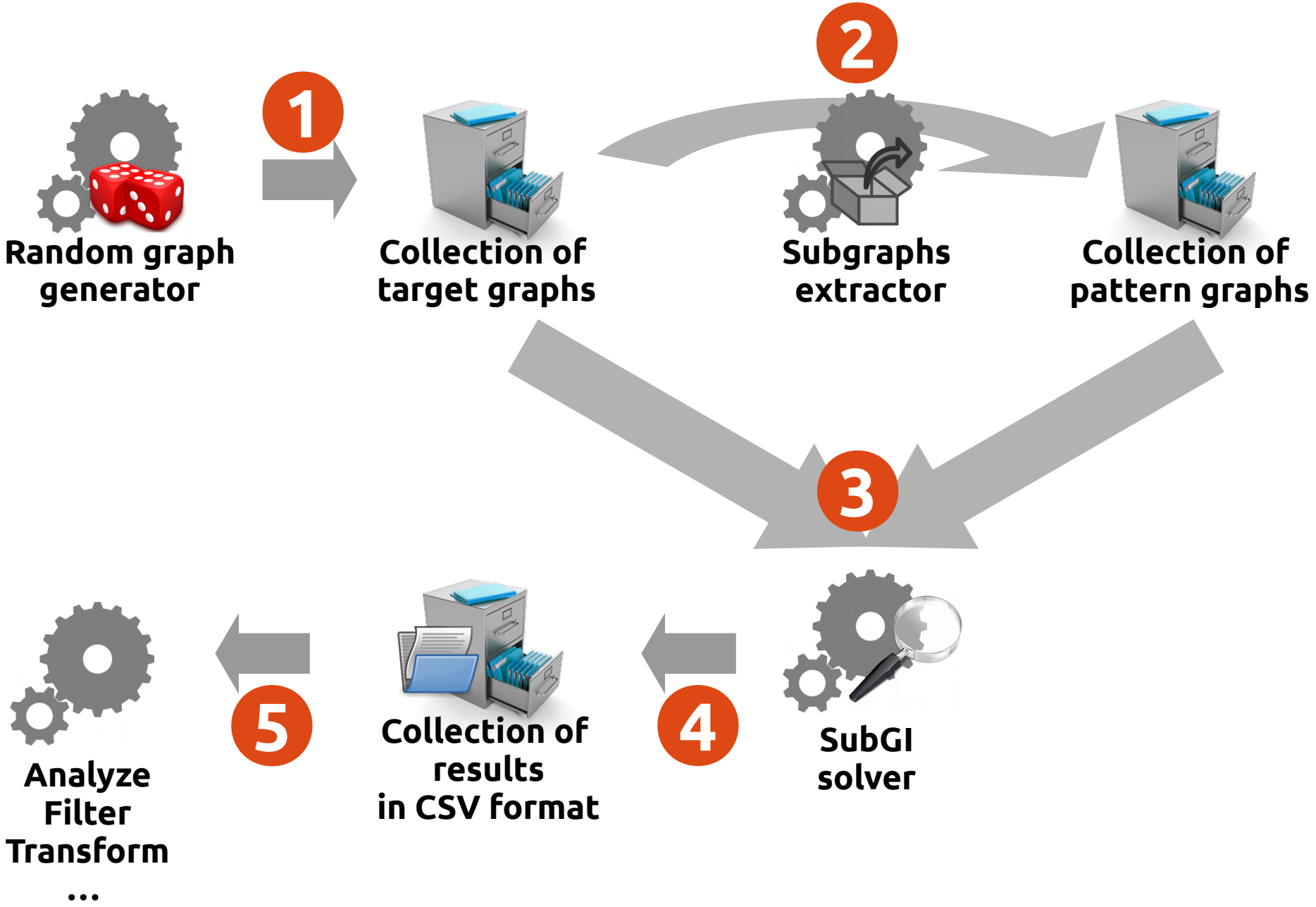
Given a **pattern (query)** graph **Q=(V,E)**
and a **target (reference)** graph **R=(V',E')**
- Graph **isomorphism** problem:
  - Find a function **M : V → V'** such that, for u,v ϵ V
    - α(u) = α(M(u))
    - for each (u,v) ϵ E than (M(u), M(v)) ϵ E' and β((u,v)) = β((M(u), M(v)))
    - M is bijective
- **Induced subgraph isomorphism**:
  - M is injective
  - for each (u,v) ÏE than (M(u), M(v)) ÏE'
- **Subgraph isomorphism** (**monomorphism**):
  - M is injective

- **NP-complete (general case)**



Why a computer science problem?
There are **plenty of applications** of it in any science field, some of them are in **computational biology** (i.e. searching for **network modules** and **sub-pathways** or as building block of **motif searching**)
Remember that we are **bioinformaticians**, we build software for the Biology, we analyze biological data, and sometimes we arise biological questions.

A SubGI testing pipeline



Materials

Pre-written software

generate_graph.py : a python script for generating random graphs.

```
python3 generate_graph.py <nof_nodes> <edge_percentage> <nof_labels> <o_file>
```

Generate a uniformly random graph with a specified number of nodes and save it into `<o_file>`. The percentage of edges is referred to the maximum number of edges within a graph of n nodes, that is `nof_nodes * nof_nodes`. Graph node are labeled with a total number of distinct `<nof_labels>` labels, the parameter cannot exceed 25. Labels are single characters in `[A-Z]`.

extract_queries.py : a python script for extracting subgraphs to be used as queries.

```
python3 extract_queries.py <i_graph_file> true <nof_queries> <nof_nodes>
<edge_percentage> <output_prefix>
```

Extract from the graph stored in the `<i_graph_file>` file a total of `<nof_queries>` subgraphs. Each subgraph has `<nof_nodes>` nodes and the given edge percentage, with respect to the number of nodes of the subgraph. Extracted subgraphs are stored in a st of files, one for each subgraph, into the path specified by `<output_prefix>`.

Materials

Pre-written software

RI3.6 : a C implementation of an algorithm to solve (sub)graph isomorphism

How to compile under Ubuntu:

- `install make: sudo apt-get install make`
- `install g++: sudo apt-get install g++`
- enter RI3.6-release-verbose directory
- compile: `make -B`
- the `ri36` executable will be created/overwritten

Usage:

```
./ri36 <isomorphism_type> <input_format> <reference_graphs_file>
<query_graph_file>
```

`<isomorphism_type>` can be iso (isomorphism), ind (induced sugraph isomorphism), mono (monomorphism). We will use mono.
As input format we will use gfd,the one for directed graphs having labels on nodes.
The `<reference_graphs_file>` contains one or more reference graphs.
The `<query_graph_file>` contains only one query graph that is searched within all the input reference graphs.

Materials

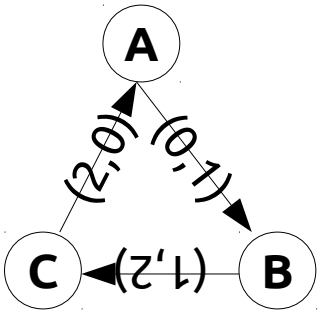
The GFD input format

GFD is a format used to describe directed (ordered) graphs having labels inly on nodes.
The schema is the following:

```
graph_name
number_of_nodes
label_node_0
label_node_1
…
label_node_n
number_of_edges
source_node_id[blank]target_node_id
source_node_id[blank]target_node_id
…
source_node_id[blank]target_node_id
```

An example

```
#my_graph
3
A
B
C
3
0 1
1 2
2 0
```



Every token (graph name, node label, etc..) can not contain blank characters, except the one used between node ids for describing edges.
The first node has ID equal to 0, and the last node has ID equal to number_of_nodes – 1.
Labels are implicitly listed from node 0 to the last node ID.

Exercises

B.1 Generate a collection of random graphs by making use of the pre-written software generate_graph.py . The step must generate batches of networks having 100,200,500,1000,2000 and 5000 nodes, two edges percentages that are 0.2 and 0.1, and a number of labels ranging in 2,4,8,16,25.
For each combination of number of nodes, edge percentage and number of labels, two graph must be generate. Thus, every generated graph has a unique path described as
`/path_to_mydir/collection/NOF_NODES/EDGE_PERC/LABELS/graph_[0,1].gfd`
but commas must not be contained in directory names, ex

```
./collection/100/0_2/2/graph_0.gfd
./collection/100/0_2/2/graph_1.gfd
```

B.2 Generate a collection of query graph by making use of the pre-written software extract_subgraph.py. Query graph must be extracted from previously generate target graph (in exercise B.1) and stored in a sub-directory named queries and located into the target graph collection. Extracted subgraphs must have 4,8,16,24,32,64 and 128 nodes, and an edge percentage of 0.1 and 0.2. For each combination of number of nodes, edge percentage and target graph, ten queries must be extracted.
Example:

```
./collection/100/0_2/2/queries/4/0_1/graph_0_0.gfd
...
./collection/100/0_2/2/queries/4/0_1/graph_0_9.gfd
./collection/100/0_2/2/queries/4/0_1/graph_1_0.gfd
...
```

Exercises

B.3 Write a bash script for extracting information regarding GFD files. The script must inform about the number of nodes and the number of edges.

The script must output:

```
file path X
graph name X
number of nodes X
number of edges X
```

Example:

```
file path ./collection/100/0_2/2/graph_0.gfd
graph name graph_0
number of nodes 100
number of edges 2000
```

Exercises

B.4 Build a TSV (Tab-Separated Values) file reporting information about the previously generated collection of target graphs, and by making use of the script of exercise B.3. For each row, the TSV file must report the complete file path, the graph name, the number of nodes and edges.

B.5 Looking at the TSV file, answer the following questions:

- How many graphs are in the collection?
- How many graphs have 200 nodes?
- How many graphs are in the collection, for each different number of nodes?
- How many graphs have 2000 edges?
- How many graphs have 2 distinct labels and 100 nodes?
- How many graphs have a 0.2 edge percentage?

B.6 Write a bash script that, given an input graph in GFD format, reports for each distinct label the number of nodes having such label.

B.7 Write a bash script that takes a GFD file as input and a remapping label file. The remapping label file reports a configuration for changing node label. Each row of the file is composed by two columns, the first one reports the original label within the input graph, the second row reports the new label that replaces the original one. The transformed graph is saved into a file specified by a third parameter.

B.7.1 Write the B.7 script such that it uses the command `sed -i`

Exercises

B.8 Write a bash script that execute subisomorphism testing on the previously generated collection of target and query graphs.
Queries must be searched within target graphs having the properties of the graph from which they have been extracted. For example, if a query have been extracted from a graph having 100 nodes, 2 labels and 0.2 edge percentage, then the query must be searched within every target graph of the same category.
The RI software must be used to accomplish the goal.

Results mu be written in a TSV file reporting:

- Complete Target graph path
- Complete query graph path
- Total execution time
- Matching time
- Number of found occurrences (matches)

NB: take the advantages of the RI output format for extracting the required information.

Exercises

B.9 Extend the previous exercise such that:

- The RI command is executed with a given timeout
- The TSV file reports only execution that finished within the timeout
- Extend the TSV output format such that any row also reports the memory usage `/usr/bin/time -f"%M" …`
- Aborted execution must be reported in a separated file
- The script can be stopped by CTRL+C (at the end, every script can be stopped in such a way), but if the user restarts it then already tested combinations of query and target graphs must not be performed again.