Operations Research. Laboratory Session

Heuristic Optimization with `metaheuR`.

(Experimenting with different neighborhoods in Local Search)

(The MIS problem)

The aim of this laboratory session is to continue learning how to implement heuristic algorithms for solving combinatorial optimization problems using the `metaheuR` package in R. Be careful, in R there exists a package called `metaheur`, but that's not the one we'll use. In addition, the `igraph` package is also used for plotting the solutions.

1. Installing `metaheuR`

You can install it directly from RStudio. Download the file `metaheuR_0.3.tar.gz` from the eGela platform to a working directory in your computer. I saved it here:

/Users/JosuC/Desktop

To install the package, write the path that corresponds to the working directory where you saved the file `metaheuR_0.3.tar.gz` and execute the following commands:

```
setwd("/Users/JosuC/Desktop")  # write yours
install.packages("ggplot2", dependencies=TRUE)
install.packages("igraph", dependencies=TRUE)
install.packages("metaheuR_0.3.tar.gz", repos = NULL, type="source")
library(igraph)
library(metaheuR)
```

Once the package is installed and loaded, you can go to RStudio "Packages" and click on the name of the package to see the help pages of all the functions defined in it.

For more extensive documentation, the book "*Bilaketa Heuristikoak: Teoria eta Adibideak R lengoaian*" published by the UPV/EHU is suggested. It is written in Basque and freely accessible in:

https://addi.ehu.es/handle/10810/25757

2. The Maximum Independent Set (MIS)

With illustrative purposes, in the current session, the Maximum Independent Set (MIS) problem will be considered. The MIS problem is stated as follows: given a graph $G = (V, E)$, the aim is to find the largest set of vertices S of the graph G such that any pair of vertices in S does not have a connecting edge. Any solution that does not comply with the previous restriction is considered an unfeasible solution, and need to be corrected.

Formulating the problem in RStudio

First of all, we need to define the problem, to that end, first we will create a random graph with 100 vertices.

```r
# Create a random graph that is going to be used as structure of the Maximum Independent Set,
# and create the object of the problem.

# set number of nodes in the graph.
number_nodes <- 100

# create a random graph using "igraph" library.
rnd.graph <- random.graph.game(number_nodes, p.or.m=0.05)

#create the problem object.
# IMPLEMENT HERE
```

Propose a suitable representation to codify the solutions of the MIS, then create a random solution, and evaluate it. Try to visualize the proposed solution using the `plot` function to gain intuition on the problem.

```r
# Create an object, and evaluate it using the object in the previous code chunk
# IMPLEMENT HERE
```

Question

- What is the most appropriate codification to represent the solutions for the MIS problem? [ANSWER] ...
- How many solutions are induced by the codification? Are all of them valid solutions of the search space of solutions? [ANSWER] ...

As stated previously, any two vertices in the independent set cannot be linked with an edge. If that happens, we will say that particular solution is unfeasible. A common procedure for such cases is to correct the solutions using the utilities contained in the problem object. In the following code chunk, check whether the solution generated previously is feasible, and then use the correction method to make it feasible. Once the solution is corrected, evaluate again its objective value.

```r
# IMPLEMENT HERE
```

Question

- Did the solutions (before and after correction) change? How many nodes are in the independent sets in each case?

[ANSWER] ...

3. Local search

Local search is a heuristic method that is based on the intuition about the searching process: given a solution, the idea is to find better solutions in its neighborhood. At each iteration, the algorithm keeps a current solution and substitutes it with another one in the neighborhood. We already worked with it in the previous laboratory session.

The efficiency of the algorithm is highly related to neighborhood operator selected, and also to the criterion employed to choose the neighboring solution. In this lab session, we are going to experiment with one neighborhood function: the bit-flit neighborhood.

```r
# Initialize a random initial valid solution, and create run neighborhood.
# IMPLEMENT HERE
```

Having the initial solution and the neighborhood defined, now we'll apply the `basicLocalSearch` function, as we did in the previous lab session. You can have a look at the help pages to see how to use it. It requires quite a lot of parameters.

As we have seen in the theory, there are different strategies to select a solution among the ones in the neighborhood during the searching process. In the previous lab, we applied a greedy strategy to select a solution in the neighborhood with `greedySelector`. This time we will also experiment with another option and select the first neighbor that improves the current solution, `firstImprovementSelector`. We will consider the two options and compare the results. The aim is to estimate the number of local optima with each of the criteria, and to select the most efficient one.

According to the resources available for the searching process, this time we will limit them as follows: the execution time (10 seconds), the number of evaluations performed ($100n^2$) and the number of iterations to be carried out ($100n$), being n the size of the problem.

Once all the parameters are ready, the searching process can start. Try with the different options for the parameters mentioned.

```
# IMPLEMENT HERE
# Set up local search and execute

# WAIT... It takes time...

# Extract the approximated solution found and the objective value
```

Note that in order to perform the basic local search efficiently, unlike in permutation problems, in the MIS problem, it is mandatory to provide the functions that will help the algorithm to identify if the solution that is being considered in the neighborhood is feasible or not, and if it is not, then, it is likely to improve it. To that end, pay attention to the arguments `non.valid`, `valid` and `correct`.

Questions:

- Which would you say that is the purpose of the `non.valid`, `valid` and `correct` functions in the code above? (Use the help of the package if needed.)

[ANSWER] ...

- What are the approximate solutions obtained for the different strategies to select a solution among the ones in the neighborhood? What's their objective value? Compare them and say which one is the best (do not forget that `metaheuR` tries to minimize the function).

1) Apply a greedy strategy to select a solution in the bit flip neighborhood:

[ANSWER] ...

2) Select the first neighbor that improves the current solution in the bit flip neighborhood:

[ANSWERS] ...

```
# CODE FOR THE QUESTIONS ABOVE IMPLEMENT HERE.
```

- Now, you can try to increment the resources for the searching process. Do you obtain better results?

[ANSWER] ...

Estimating the number of local optima

Our intuition suggests that if there is a small number of local optimum in the neighborhood, the algorithm has more chances to find the global optimum; its shape is said to be "smooth" or "flat". On the contrary, having a neighborhood with a lot of local optimum makes it more difficult to find the global optimum, since the algorithm will very easily get stack in a local optimum; its shape is said to be "wrinkled" or "rugged".

In addition to the neighborhood structure, another element that influences how the fitness landscape look like is the selection criteria. Having a "greedy" or "first improvement" criteria, can make a real difference. So, how can we know which of the two is better? To that end, we will estimate the number of local optima we obtain with each of the selection criteria.

There is a very easy way to estimate the number of local optimum in a neighborhood. It is possible to generate k initial solutions at random and apply the local search algorithm to each of them to obtain k local optima. They are not all necessarily different, of course, because of the "basins of attraction". Let's say that for a particular selection and starting at k initial solutions LO_{diff} different local optima are obtained. So, the percentage of different local optima obtained can be computed like this:

$$100 * \frac{LO_{diff}}{k}.$$

In the following you are asked to estimate the number of local optimum for the two selection criteria we created at the beginning of the lab session: the "greedy" neighborhood and the "firstImprovement" criteria. We will generate ($k = 5$) initial solutions to start. According to the resources, we will limit the search to $1000n^2$ evaluations.

```
# IMPLEMENT HERE

different.local.optima<-unique(local.optima)
cat("Number of local optimum: ", length(different.local.optima))
cat("\nPercentage of local optimum in the search space: ",
    100*as.double(length(different.local.optima))/k, "%")
```

Questions:

- What is the number of local optimum estimated for the "greedy" selector? And, for the "firstImprovement" selector?

– "greedy" selector, bitflipNeighborhood, k=5, evaluations=$1000 \times n^2$

[ANSWER] ...

– "firstImprovement" selector, bitflipNeighborhood, k=5, evaluations=$1000 \times n^2$

[ANSWER] ...

We should increase the number of initial solutions and see if it makes any difference...

- Which one would you say is more appropriate for the MIS problem?

[ANSWER] ...

- Repeat the experiment for different number of initial solutions, $k = 10, 15, 20$. Do not consider very large values for k, because it takes time to do the estimations... Can you observe any difference?

[ANSWER] ...

5. Advanced local search-based algorithms

Local search-based algorithms stop their execution whenever local optima solution have been found (unless assigned resources run out before). This implies that no matter how much we increase the availability of execution resources, the algorithm will remain stucked in the same solution. In response to such weaknesses, the community of heuristic optimization proposed a number of strategies that permit the algorithm to scape being stucked, and enable to continue optimizing. An obvious strategy, is to run another local search algorithm, however, since the current solution is a local optimum, then, no improvement will take place. In this sense, an alternative is to perturbate the current solution (5% of the numbers that compound the solution), and then apply again the local search algorithm. This general procedure is known Iterated Local Search (ILS). The algorithm repeats until the available resources run out.

`metaheuR` already includes an implementation of the ILS, but I do not want you to use it. Instead, I want you to implement your own design. You have almost every puzzle piece:

- Local search algorithm (using the bit flip neighborhood found so far).
- The `getEvaluation` function to obtain the objective value of the best solution found by the local search algorithm.
- The non-consumed evaluations can be known with the function `getConsumedEvaluations`.
- Stop the ILS algorithm after 10^5 function evaluations. If it takes too much time, test first with 10^4 evaluations.

The perturbation function is given below. The `ratio` parameter describes the percentage of the solution that will be shuffled.

```
perturbShuffle<-function(sol,ratio,...){
return (binaryMutation(solution= sol,ratio=ratio))
}
```

The implementation of the ILS:

```
# IMPLEMENT HERE
```

Questions

- Return the solution and objective value of the solution found so far. Is this solution better than the one calculated by the local search algorithm in the previous sections?

[ANSWER] ...