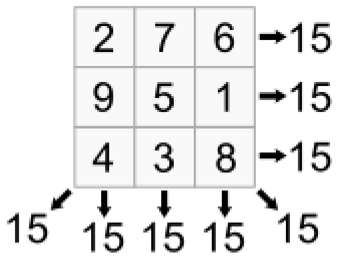
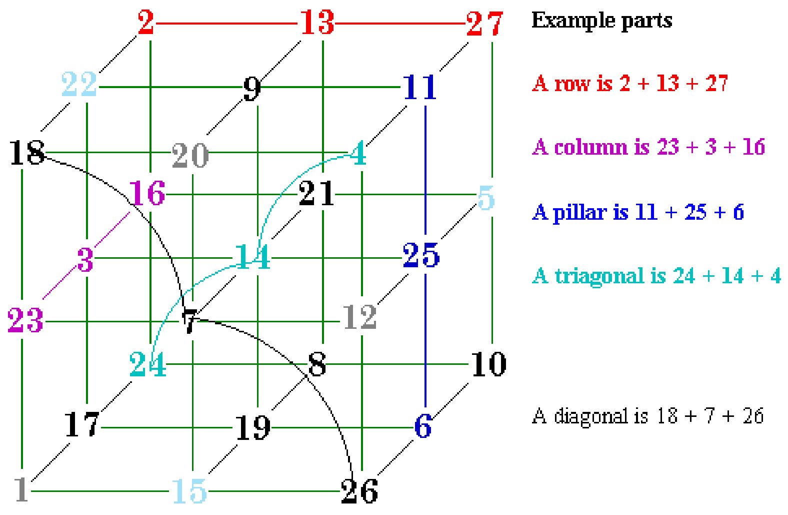
Natural Computing – Practical Assignment

Solving Magic Cube Problems with Nature-Inspired Algorithms

The Magic Square or Magic Cube is a very interesting combinatorial optimization problem. Taking the magic square for example, a magic square of order *n* is defined as an arrangement of distinct integers (from 1 to n^2) in a square grid, such that the sum of numbers along every row, column and diagonal is the same. The figure below shows an example of a magic square of order 3:



In a similar way, the perfect magic cube could be defined as the extension of the magic square, where in addition, the space diagonal and the pillars have to be taken into account. See the example below.



If the diagonals are not considered (diagonal lines on each square slice), then the resulting cube is called a semi-perfect magic cube, which is relatively easier to construct.

Interestingly, for a certain order, the sum of each row, column, etc., have been proven to be a constant, the so-called *magic constant.* For a magic square of order *n* it is:

$$\frac{n(n^2+1)}{2}$$

and for a magic cube of order *n*, it is:

$$\frac{n(n^3+1)}{2}$$

Using such a magic constant, we could consider the magic square (or cube) construction as a combinatorial optimization problem, where the error of each sum to the magic constant is aggregated as an objective function (to be minimized).

Assignment

This Assignment should be done in a team of two! Please form the team asap.

Based on skeleton files provided, implement the following (all-of-them) nature-inspired algorithms to construct a magic square and cube:

- A *Simulated Annealing* (SA) algorithm;
- A *Genetic Algorithm* (GA);

Write a report on your work, starting with a section in which per algorithm you provide a description of your implementation, including *pseudocode*. Under the Experiments section, list comparisons of algorithm convergence on each test problem (make a plot of the error of the best square/cube found so far against the number of used evaluations), and visualize the best result per optimizer for all test problems. Analyze the outcome of the experiments.

NOTE: Your report should contain enough detail for someone to implement your algorithms from the information in your report and reproduce your results.

Submission Guidelines

This project is to be implemented in Python. Each algorithm should be fully contained in one .py file and structured according to the skeleton provided on Blackboard.

Your report in PDF format should include one page per algorithm and two pages for the results. It should follow the template provided in this document.

Your complete submission consists of three files: a report in **PDF** format and <u>one</u> **.py** file per algorithm. These files can be submitted on blackboard.

Follow carefully the Python *Implementation Details* and the *Report Structure Template* provided below (not adhering to the requirements and conventions will affect your grade).

Deadline: before December 7th, 2018, 12:00 (noon) (submitting after this deadline results in 0.5 point subtracted per week)

Late submission deadline: before January 11th, 2019, 12:00 (noon)

NOTE: Submission is NOT possible after the late deadline and any assignments submitted after this date will NOT be graded

Python Implementation Details

Each algorithm implementation consists of **one** .py file named **lastname1_lastname2_sa.py**, **lastname1_lastname2_ga.py**

Each implementation should be structured as follows:

```
def lastname1_lastname2_sa(dim, eval_budget, fitness_func, do_plot=False, return_stats=False):

    ...

    if return_stats:
        return xopt, fopt, hist_best_f
    else:
        return xopt, fopt
```

- **(in)** *dim*: problem dimension, which should be either 2 (square) or 3 (cube)
- **(in)** *eval_budget*: the number of function evaluations that can be used. One *evaluation* means computing the error of one candidate solution and one *iteration* of the algorithm (generally) consists of several such evaluations;
- **(in)** *fitness_func*: the fitness function provided (See PA.zip).

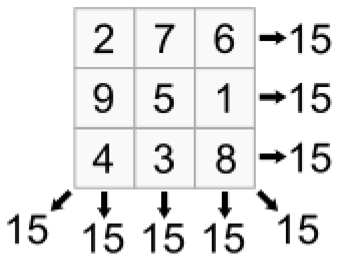
- **(in)** *do_plot:* whether or not plots should be shown during the run. Default value: False
- **(in)** *return_stats:* whether or not the fitness history of the optimization run should be returned too. Default value: False.

- **(out)** *xopt*: the best found solution vector, containing integers permutations of 1 to n^2 (magic square) or 1 to n^3 (magic cube).
- **(out)** *fopt*: the corresponding error of the best found solution vector.

NOTE: Make sure all output (e.g., plotting and debug print statements) is turned off in the submitted version of your work!

Solution Representation for Evaluation

All the evaluation code provided uses an integer vector representation as the input and returns the error of this vector as output. In order to evaluate the candidate solution, you have to convert the solution vector to the integer vector encoding. For the magic square problem, the integer vector representation is constructed by taking numbers from the square row-wisely. For the magic square shown in the following figure, its representation is `[2,7,6,9,5,1,4,3,8]`.



For the magic cube problem, the integer vector is constructed by taking square slices from top to bottom. And the numbers in each square slice are arranged in the same way as in the magic square.

Problem Instances

You are required to solve three magic square/cube problems:

1. **Search for a magic square of order 12** (12^2 = 144 parameters); Use `eval_square()` from `magic_square.py` for evaluation.
2. **Search for a semi-perfect magic square of order 7** (7^3 = 343 parameters); Note that in this case, the diagonals are not considered. Use `eval_semi_cube()` from `magic_square.py` for evaluation.
3. **Search for a perfect magic cube of order 7** (7^3 = 343 parameters); Note that in this case, the diagonals are considered. Use `eval_cube()` from `magic_square.py` for evaluation.

The function evaluation budget should be set according to the problem scale and the difficulties. You are advised to test your implementation using the following budget settings:

1. **magic square of order 12**: 10^4
2. **a semi-perfect magic square of order 7**: 10^4
3. **a perfect magic square of order 7**: 10^4

NOTE: Reaching the optimum (error = 0) is nice but not a must. As long as your algorithms make good progress in reducing the error it is OK.

Bonus Point

You can get up to 1.0 (one) bonus point by creating a nice visualization for the magic square optimization with your Python code. This visualization would ideally show the best found magic square each generation. In this way the user would see a sort of movie of the optimization process of the magic square.

Skeleton Files

Complete the following two skeleton files for your implementations:

- **sa_skeleton.py**
- **ga_skeleton.py**

Provided Functions

The evaluation functions in `magic_square.py` are documented here:

eval_square(): evaluates a candidate solution for the magic square problem.

```
from magic_square import eval_square
f = eval_square(square)
"""
Fitness function of the magic square: this code takes into
account the diagonal sums on each square slice

:param square: array, the solution vector that represents a magic cube.
:return:  double, the error value of the input solution vector.
          The mean squared error (MSE) of all each row, column, diagonal
          and space diagonal sum to the magic constant is computed
"""
```

eval_semi_cube(): evaluates a candidate solution for the semi-perfect magic cube problem.

```
from magic_square import eval_semi_cube
f = eval_semi_cube(cube)
```

```
"""
Fitness function of the perfect magic cube: this code takes into
account the diagonal sums on each square slice

:param cube: array, the solution vector that represents a magic cube.
:return:  double, the error value of the input solution vector.
          The mean squared error (MSE) of all each row, column, diagonal
          and space diagonal sum to the magic constant is computed
"""
```

eval_cube(): evaluates a candidate solution for the perfect magic cube problem.

```
from magic_square import eval_cube
f = eval_semi_cube(x)
"""
Fitness function of the perfect magic cube: this code takes into
account the diagonal sums on each square slice

:param cube: array, the solution vector that represents a magic cube.
:return:  double, the error value of the input solution vector.
          The mean squared error (MSE) of all each row, column, diagonal
          and space diagonal sum to the magic constant is computed
"""
```

compare_optimizers.py: use this script to generate the final plots for your report; the results and plots will be stored in a subdirectory named Results. By adhering to the naming convention your optimizers are automatically detected.

```
$ python3 compare_optimizers.py
```

This file contains its own documentation at the top of the file, which can also be displayed using

```
$ python3 compare_optimizers.py --help
```

Report Structure Template

Authors (names, email addresses, and student numbers)

Simulated Annealing

Pseudocode per algorithm including explanatory description, overview of algorithm parameters and optimal settings used for these parameters, and argumentation for design choices. Make sure that the algorithm and the results found are reproducible from your description.

Genetic Algorithm

Ditto.

Experiments

Use `compare_algorithms.py` to generate plots of your results. Analyze the outcome of the experiments.

Discussion and Conclusion

Summarize the results and conclude your report.