

Organising a Colour Palette

“Welcome! You’re our new employee, right? You should already know that our company specialises in software for editing and managing large collections of photographs. Some of our valued customers want to be able to browse their photograph collection in a visually pleasing fashion with similar coloured photographs following each other.

We are not really sure what that means but you can interpret that as having some *ordering* on the dominant colour of consecutive photographs. You have a month to prototype some ideas and report your results.




Colour Models

Colours used in computer graphics are based on a particular model. The model you pick depends on the range of colours you need in a graphic and whether it is going to be output to print media or to screen. There are various colour models available. Some examples are: Black & White, Grayscale, RGB (Red, Green and Blue), CMYK (Cyan, Magenta, Yellow and Black), and HSB (Hue, Saturation & Brightness)

In this assignment, we will consider the [RGB \(Links to an external site.\)](#) model. Red, green, and blue can be combined in various proportions to obtain any colour in the visible spectrum. In our representation R, G, B can each range from 0 to 1. Where 0 indicates absence and 1 full intensity.

Dataset

You have given 3 datasets with increasing size to support your implementation and testing:

- 10 colours: [col10.txt](#) 
- 100 colours: [col100.txt](#) 
- 500 colours: [col500.txt](#) 

The files contain rows of three floating-point values. Each row represents a colour by its red, green, blue coordinates. The first uncommented line is the number of colours.

You are also given a Python notebook with useful code: [colour_startup.ipynb](#)

This notebook includes the code to read the datafile and visualise a sequence of colours. It also contains some other useful functions to support your assignment.

Solution Representation

To solve this problem, your implementation should encode a candidate solution as an ordering of indices, not as an ordering of the colours themselves.

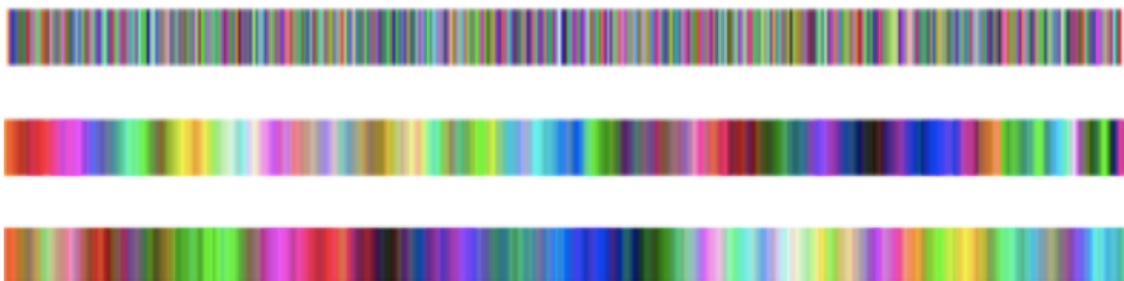
To clarify this, let us assume that your dataset has 5 colours (each colour represented with its RGB coordinates) and it is stored in a list called `c`. An ordering of the colours (i.e a candidate solution to your problem) should be encoded as a list of indices of length 5, where each element in the list is an index in the `c` list.

For example, a solution encoded as `e s = [1, 0, 2, 4, 3]`, represents the following ordering of colours `c[1] c[0] c[2] c[4] c[3]`.

What do you need to do?

Your task is to provide a more aesthetically pleasing ordering of a list of colours.

To illustrate what we mean, consider the colour bands in the figure below. The top band shows an unordered set of colours, while the 2- and 3- bands have been ordered by different algorithms. You can clearly notice the difference between an arbitrary ordering and an improved ordering.



To complete your task, you are asked to implement 3 different algorithms.

1. Multi-start hill-climbing algorithm
2. Clustering-based algorithm
3. Algorithm of your choice

Below more details for the implementation of each algorithm

1. Multi-start hill-climbing algorithm

How can we formulate this problem as an optimisation problem? One possibility is to search for an ordering of the given colours where adjacent colours are somewhat similar. This can be achieved by finding an ordering of colours that minimises the sum of the distances between adjacent colours, where the distance between two adjacent colours is computed with the Euclidean distance. This will be the objective function to evaluate solutions (called **evaluate**). To facilitate your work, we provide the implementation of this function, in the given Jupyter notebook: [colour_startup.ipynb](#)

You do not need to implement the function as it is provided. Here an explanation of what the function does: it computes the sum of the Euclidean distances, between all pairs of adjacent colours. For example, for a 5 colours list `c` and the solution `s =`

[1, 0, 2, 4, 3], and assuming $d()$ is the Euclidean distance, the evaluation function returns $d(\mathbf{c}[1], \mathbf{c}[0]) + d(\mathbf{c}[0], \mathbf{c}[2]) + d(\mathbf{c}[2], \mathbf{c}[4]) + d(\mathbf{c}[4], \mathbf{c}[3])$.

To implement multi-start hill-climbing in a modular way you should first implement a hill-climbing method as follows:

Hill-climbing

This is the algorithm we have discussed in class and implemented in practical 3 for the Knapsack problem. It starts from an initial random solution and tries to improve it iteratively using a mutation operator. Here you can reuse your Hill-climbing code for the Knapsack problem. Notice, however, that we cannot use the same bit-flip mutation operator used for the Knapsack problem as the representation in the colour ordering problem is not a list of binary numbers! Our representation is instead an ordering (also called a *permutation*) of integers. Notice that in a permutation, each integer or index can appear only once.

For a permutation representation, the simplest mutation operator is to swap two colour indices selected at random. For example, given a solution $\mathbf{s} = [1, 0, 2, 4, 3]$, the solution $\mathbf{s}' = [1, 3, 2, 4, 0]$ is a neighbouring solution that swaps the indices at position 1 and 4. This operator is called *swap*.

Other possible operators are:

- *inversion*: this operator works by Inverting (reversing) the ordering between any two colour indices selected at random. For example, given the solution $\mathbf{s} = [1, 0, 2, 4, 3]$, the solution $\mathbf{s}' = [1, 3, 4, 2, 0]$ reverses the indices between positions 1 and 4
- *scramble*: this operator works by randomly shuffling (scrambling) the ordering between any two colour indices selected at random. For example, given the solution $\mathbf{s} = [1, 0, 2, 4, 3]$, the solution $\mathbf{s}' = [1, 4, 0, 3, 2]$ scrambles the indices between positions 1 and 4.

The selection of which operator to implement and use is left to your own choice, you do not need necessarily to implement all of them!

An initial randomly generated solution for the colouring ordering problem is also different than that of the Knapsack problem. A random solution is a random permutation or reordering of the colour indexes. The source code provided illustrates how this can be done in Python (function **random_sol**).

Multi-start hill-climbing

Once you implemented the hill-climbing function, the multi-start hill-climbing simply Implements a loop that calls the **Hill-climbing** algorithm from different initial solutions. Your function should receive a parameter indicating the number of repetitions. Here again, you can reuse the code you implemented for the Knapsack problem.

2. Clustering-based algorithm

Here your solution should consider a clustering algorithm as implemented in the scikit-learn library. You can use any of the clustering methods available (k-means, hierarchical, etc).

The idea is to apply the clustering method to the colours dataset, so the colours will be assigned to a number of clusters. Let us assume your clustering algorithm produced K groups or clusters. Then you will assemble a solution by ordering the colours according to their cluster membership. That is, first add all the colours from cluster 1, then add the colours from cluster 2 and so on up to cluster K .

Notice that within each cluster, the colours will not be ordered, but this is OK. This algorithms only orders the colours at the level of clusters.

3. Algorithm variant of your choice

Here you're given the opportunity to design your own algorithm. The general goal is to try to improve the performance and quality of solutions obtained by Algorithms 1 and 2. This part is left open to your creativity. You can propose variations of Algorithms 1 and 2 by incorporating some of the algorithm ideas and metaheuristics discussed in the lectures, you can combine methods, you can do your own research or try your own ideas. Marks will be given for the effort in your research and implementation, and for the quality of your best-obtained solutions. By "quality" of solutions, I mean both the subjective appearance and the value obtained using evaluation function (sum of distances).

What do you need to submit?

You need to submit 3 Jupyter notebooks, one for each Algorithm. The notebooks will contain your code, some text descriptions and some plots illustrating your results. There is no page/length limit, but you are encouraged to be concise and clear in your descriptions.

There is no need to recapitulate the problem or have an introduction. Here a description of what to include in each of the Jupyter notebooks in addition to your code. You will also find an indication of the % marks for each part.

1.

1. **Multi-start hill-climbing algorithm (40 %)**

- Indicate which neighbourhood you implemented, briefly justify your choice.
- Record the trace of objective function values obtained across a single run of the Hill-climbing algorithm. With a line plot, visualise this trace.

- Run your multi-start hill-climbing algorithm for both colour sizes 100 and 500. You can conduct your own experimentation, there is no particular limit on the number of iterations, repetitions you want to run.
- Report (by assigning them to Python variables) the best solution found for each instance during your experimentation, call them **mhc_best100** and **mhc_best500**. Using the visualisation function, produce the colour band plots for your solutions, report also their objective function (evaluation) values.
- Describe briefly the experiments you conducted to find the best solutions (i.e. number of iterations, tries)

2. Clustering-based algorithm (30%)

- Indicate which clustering algorithm you used, briefly justify your choice.
- Indicate how many clusters K you used for each instance size (100, 500), briefly justify your answer.
- Run your clustering-based algorithm for both instances 100 and 500. You can conduct your own experimentation, there is no particular limit on the number of iterations, repetitions you want to run.
- Report (by assigning them to Python variables) the best solution found for each instance during your experimentation, call them **cl_best100** and **cl_best500**. Using the visualisation function, produce the colour band plots for your solutions.
- Here what you consider your **best** solutions are subject to your appreciation. Notice that the function **evaluate** considered in Algorithm 1, is not used here. You can still use the evaluate function if you would like to have an approximate assessment of the quality of the clustering-based solutions.
- Describe briefly the experiments you conducted to find the best solutions (i.e. number tries, clustering algorithm parameters)

3. Algorithm variant of your choice (30 %)

- Briefly describe in text the main idea and motivation behind your algorithm
- Run your algorithm for both dataset sizes 100 and 500. You can conduct your own experimentation, there is no particular limit on the number of iterations, repetitions you want to run.
- Report (by assigning them to Python variables) the best solution found for each instance during your experimentation, call them **my_best100** and **my_best500**. Using the visualisation function, produce the colour band plots for your solutions, report also their objective function (evaluation) values.
- Describe briefly the experiments you conducted to find the best solutions (i.e. number tries, iterations, parameters)