B4 - Functional Programming

B-FUN-400

Image Compressor

a nice application to parallel k-means

1.13

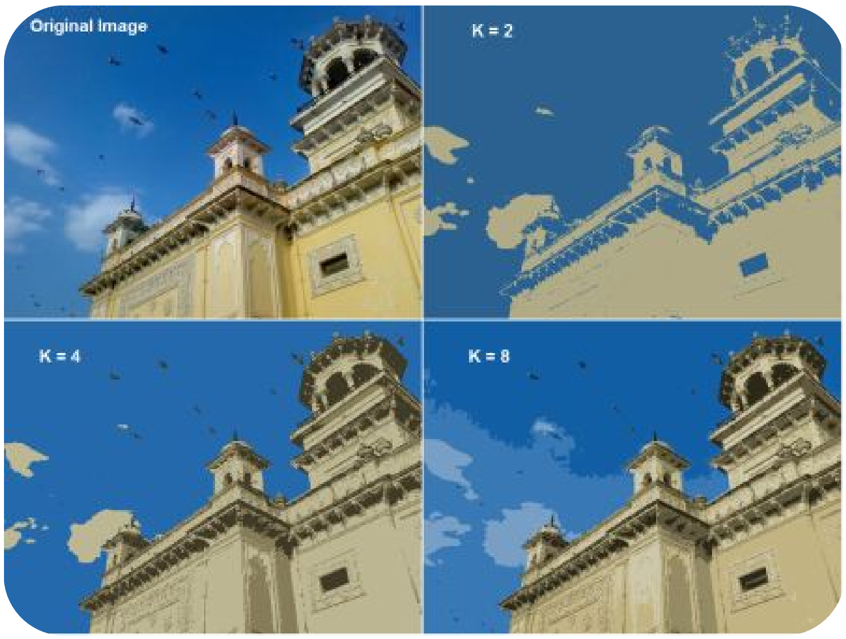
Image Compressor

binary name: imageCompressor
language: Haskell
compilation: stack wrapped in a Makefile *(see below)*

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

A pretty basic way to compress image consists in reducing the number of colors it contains.
3 steps are needed to do so:

1. **read** the image and **extract** the colors of each pixel,
2. **cluster** these colors, and **replace** each color of a given cluster by the mean color of this cluster,
3. **index** the means of the cluster, and **create** the compressed image.

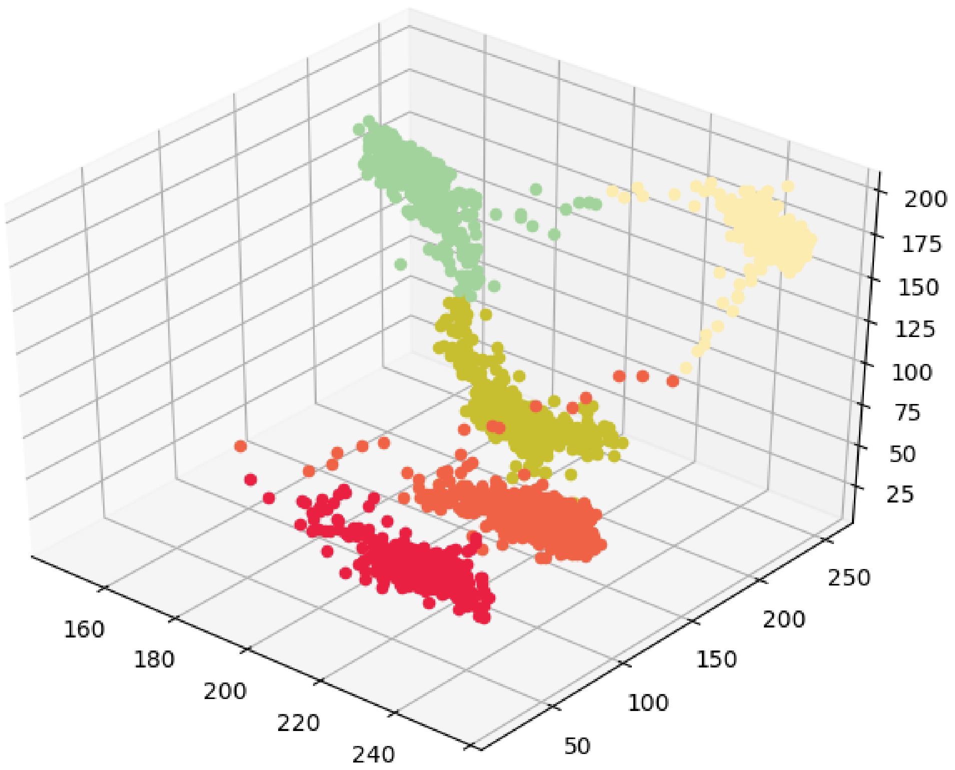


In this project, the first and third steps, reading from and writing into images, are considered as bonus. You are to focus on the second part of the process: **clustering**.

From a file listing all the pixels of the image with their position and color, regroup them into a given number of clusters, according to their colors.

You HAVE to use the **k-means** algorithm for clustering.

Remember how k-means work? At every step, compute a new bunch of k centroids and add data to its nearest centroid, as in a Voronoi space, until the algorithm converges.



k-means work with a distance on the data space. You can simply use the euclidean distance:

$$d = \sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$$

(r_1, g_1, b_1) being the red, green and blue components of the first point,
(r_2, g_2, b_2) being the red, green and blue components of the second point.

To test the convergence of the algorithm, use the convergence limit given as parameter. convergence is considered reached when all the clusters have moved less than the convergence parameter since the last iteration.

Terminal

```
~/B-FUN-400> ./imageCompressor
USAGE: ./imageCompressor -n N -l L -f F

      N       number of colors in the final image
      L       convergence limit
      F       path to the file containing the colors of the pixels
```

INPUT

You should read the list of pixels from a file passed as argument, according to the following grammar:

```
IN       ::= POINT ' ' COLOR ('\n' POINT ' ' COLOR)*
POINT    ::= '(' int ',' int ')'
COLOR    ::= '(' SHORT ',' SHORT ',' SHORT ')'
SHORT    ::= '0'..'255'
```

For example,

Terminal

```
~/B-FUN-400> head exampleIn
(0,0) (33,18,109)
(0,1) (33,18,109)
(0,2) (33,21,109)
(0,3) (33,21,112)
(0,4) (33,25,112)
(0,5) (33,32,112)
(1,0) (33,18,109)
(1,1) (35,18,109)
(1,2) (35,21,109)
(1,3) (38,21,112)
```

OUTPUT

You should print the list of clustered colors on the standard output, according to the following grammar:

```
OUT     ::= CLUSTER*
CLUSTER ::= '--\n' COLOR '\n-\n' (POINT ' ' COLOR '\n')*
POINT   ::= '(' int ',' int ')'
COLOR   ::= '(' SHORT ',' SHORT ',' SHORT ')'
SHORT   ::= '0'..'255'
```

For example,

Terminal

```
~/B-FUN-400> ./imageCompressor -n 2 -l 0.8 -f in
--
(77,63,204)
-
(0,1) (98,99,233)
(2,0) (88,77,211)
(0,2) (45,12,167)
--
(35,36,45)
-
(1,0) (33,16,94)
(1,1) (78,8,9)
(1,2) (20,27,67)
(2,1) (1,56,37)
(0,0) (66,20,26)
(2,2) (15,89,40)
```

The color in the header of the cluster is the mean color.

You might have a slightly different result due to randomness...

Bonus

Reading and writing to real image formats is an obvious bonus. **JuicyPixels** is a nice library to accomplish such a task.

Also, real-time display of the evolution of compression would be nice. **Gloss** permits creating graphical and interactive applications in Haskell with ease.

Image compression is an obvious application of the K-Means algorithm, but it can be applied to arbitrary datasets with arbitrary dimensions. Writing the core algorithm in a generic fashion and demonstrating how it can be applied to different problems is also a valuable bonus to have.

Stack

As in the first project, you must use:

- Stack to build your project, with **version 2.1.3 at least**,
- **hpack** build tool (package.yaml file in your project, autogenerated .cabal file),
- **stackage** with version **LTS 20** (`resolver: 'lts-20.11'` in stack.yaml).

In `stack.yaml`, extra-dependencies cannot be used.

base, **random**, **parallel**, **optparse-applicative** and **containers** are the only dependencies allowed in the `lib` and `executable` sections of your project (package.yaml).
There is no restriction on the dependencies of the `tests` sections.

If you work on the image part, **JuicyPixels** is allowed too.
Obviously, the palettize functions are forbidden.

You must provide a **Makefile** that builds your stack project (i.e. it should at some point call 'stack build').

'stack build' puts your executable in a directory that is **system-dependent**, which you may want to copy.
A useful command to learn this path, in a system-**independent** way, is:
`stack path --local-install-root`.