FINAL PROJECT

DUE ON 1.5 (23:55)

"Normal people believe that if it ain't broke, don't fix it. Engineers believe that if it ain't broke, it doesn't have enough features yet."
-Scott Adams

INTRODUCTION

In assignment 3, you were asked to implement a content based image retrieval (CBIR) application. Our implementation was very basic and the complexity was horrible. For the final project, we will implement a CBIR application while practicing the programming principles that we learned in the third assignment, we will also try to improve the execution time.

NOTE: Make sure you read all the instructions before proceeding with the implementation

SPLOGGER

Printing mechanism is useful for many purposes. We will implement a logger which will be used to print all messages during the run of our program. Four types of messages will be considered:

1- Error – messages that are used to indicate an error during run-time. Usually after error messages the program should terminate.
2- Warning – messages that are used to indicate a warning during run-time. Usually if a warning occurred the program will continue running.
3- Info- informational messages used to indicate the state of the system (for example –they may be used to inform the user about the progress of the program). Info messages should only contain reasonably significant information that will make sense to end users and system administrators.
4- Debug- messages are used for debugging purposes. These messages may contain anything the programmer sees fit.

The logger has 4 active levels. At each level only certain messages will be printed. The levels are:

1- Error level – In this level only error messages will be printed.
2- Warning level – In this level only warning and error messages will be printed.
3- Info- In this level all messages **EXCEPT** debug messages will be printed (i.e., error, warning and info).
4- Debug- In this level all messages will be printed.

For example, if the logger is initialized with SP_LOGGER_WARNING_ERROR_LEVEL. After executing the following lines:

```
spLoggerPrintErrorMsg("Error msg","..\unit_tests\sp_logger_unit_test.c",
"basicDebugTest",72);
spLoggerPrintWarningMsg("Warning msg","..\unit_tests\sp_logger_unit_test.c",
"basicDebugTest",74);
spLoggerPrintInfoMsg("Info msg");
```

The resulting output should be:

```
---ERROR---
- file: ..\unit_tests\sp_logger_unit_test.c
- function: basicDebugTest
- line: 72
- message: Error msg
---WARNING---
- file: ..\unit_tests\sp_logger_unit_test.c
- function: basicDebugTest
- line: 74
- message: Warning msg
```

Notice that the info message wasn't printed. For further information, please review the header file.

SPECIAL MACROS

In order to pass the line number/function name/ filename to SPLoggerPrint functions, you can use special macros used by the preprocessor. These macros are as follow:

__FILE__ - The preprocessor replaces this macro with the filename in which it appeared.
__func__ - The preprocessor replaces this macro with the function name in which it appeared.
__LINE__ - The preprocessor replaces this macro with the line number in which it appeared.

SPLOGGER.C

Partial implementation of the logger is given. You will need to extend the implementation to support all functionalities provided in the header file SPLogger.h.

Notice that before using the logger you need to use spLoggerCreate. This function must be called once, at the beginning of the main function. After that you can call the printing functions anywhere in your code.

SP_LOGGER_UNIT_TEST.C

You need to extend the unit test given in sp_logger_unit_test.c in order for your implementation to work as defined by the documentation in the header file.

PRINTING

There are two types of messages, log messages and regular messages.

Log messages will be printed using the logger we implemented in previous sections (i.e. using spLogger). Make sure that the logger was set at the beginning of the main function. Make sure to print log messages to the logger depending on the error occurred. You are free to format your messages as you wish, as long as they do represent the occurred error.
Regular messages will be printed to stdout and throughout the instruction file we will mark regular messages with the symbol **[R]**.

CONFIGURATION

Before proceeding with the description of our system, we need to be able to initialize our application in an orderly fashion. As we did in assignment 3, we need to be able to tell where our images directory will reside along with the number of images and possibly other parameters.

The way we choose to do so in assignment 3 was by receiving several parameters from the user for the purpose of configuring our system. As you may guess, this isn't the right way to do so, therefore for the final project we will choose different direction in which the configuration of our system will be set by the use of a configuration file.

In software engineering, a configuration file (config file) is used to configure parameters and initial settings for the program. The configuration file is written in ASCII encoding and line-oriented, with lines terminated by a newline character. The configuration file has a special format, such that each line in the configuration file is either a system parameter configuration, a comment or an empty line. Notice that comment lines and empty lines (such that contain only white-spaces) are valid and will be discarded.

SYSTEM VARIABLE

There are several variables that **MUST** be initially set, these variables are:

- **spImagesDirectory** : **type** string - a parameter which contains the path of the images' directory. This directory will contain all images in our database, such that given a query image, we will look for similar images in the directory **spImagesDirectory**.
  Constraint: the string contains no spaces.

- **spImagesPrefix** : **type** string - the images prefix. As we saw in assignment 3, all images in **spImagesDirectory** will start with this name.
  Constraint: the string contains no spaces.

- **spImagesSuffix** : **type** string - the images file format. This string represents the images format in **spImagesDirecoty** (all images in **spImagesDirectory** has the same file extension).

Constraint: the string value is in the following set { .jpg , .png , .bmp , .gif }

- **spNumOfImages** : **type** int - the number of images in **spImagesDirectory**.
  Constraint: positive integer.

- **spPCADimension** : **type** int -we will use this value to reduce the dimension of the SIFT features from **128** to **spPCADimension**.
  Constraint: positive integer in the range [10 , 28].

- **spPCAFilename**: **type** string - the filename of the PCA file.
  Constraint: the string contains no spaces.

- **spNumOfFeatures**: **type** int - the number of features which will be extracted per image.
  Constraint: positive integer.

- **spExtractionMode** : **type** boolean - a boolean variable which indicates if preprocessing must be done first or if the features are extracted from files (more on this later).
  Constraint: the value is in the following set { true , false}

- **spNumOfSimilarImages** : **type** int - a positive integer which indicates the number of similar images which will be presented given a query image. That is if this parameter is set to 2, the most 2 similar images with respect to a query image will be presented.
  Constraint: **spNumOfSimilarImages**>0.

- **spKDTreeSplitMethod** : **type** enum -a parameter which represents the cut method when the kd-tree is build (more later).
  Constraint: an enum which takes one of the following values {RANDOM, MAX_SPREAD, INCREMENTAL }

- **spKNN**: **type** int - a positive integer which is used in the k nearest neighbor search algorithm (more on this later).
  Constraint: spKNN>0

- **spMinimalGUI** : **type** boolean - a boolean variable which indicates if minimal GUI is supported (more on this later).
  Constraint: the value is in the following set { true , false}

- **spLoggerLevel** : **type** integer - an integer which indicates the active level of the logger. The value 1 indicates error level, the value 2 indicates warning level, the value 3 indicates info level and 4 indicates debug level.
  Constraint: the value is in the following set { 1 , 2 , 3 , 4}

- **spLoggerFilename** : **type** string - the log file name. If the value is **stdout ,** then the standard output will be used for printing. Otherwise, a log will be created with the name given by the parameter spLoggerFilename.
  Constraint: the string contains no spaces.

Some of the above parameters have default values such that if the variable is not set in the configuration file a default value will be chosen for this parameter. Parameters that don't have a default value must be set in the configuration file, otherwise the program will terminate with an appropriate message. The following parameters have default values

- **spPCADimension** : **default value =** 20
- **spPCAFilename**: **default value =** pca.yml
- **spNumOfFeatures**: **default value =** 100
- **spExtractionMode** : **default value =** true
- **spMinimalGUI** : **default value =** false
- **spNumOfSimilarImages**: **default value=** 1
- **spKNN**: **default value=** 1
- **spKDTreeSplitMethod**: **default value=** MAX_SPREAD
- **spLoggerLevel**: **default value=** 3
- **spLoggerFilename**: **default value=** stdout

The format of a system-variable line is as follow:

<variable-name> = <value>

Notice that both the variable name <variable-name> and the value <value> must not contain any-spaces. For instance, the following lines are not valid:

```
sp   Images    Directory = ./images/
spImagesSuffix = values cannot contain whitespaces
```

Whitespaces other than in <variable-name> and <value> are allowed. That is the following are valid:

```
        spImagesDirectory=./images/
spImagesSuffix =    img
```

From now on, system variable will be **bolded and green colored.**

COMMENTS

Each comment line must start with # (may contain whitespaces at the beginning) and will be discarded. For example the following lines represents comment-lines:

```
#   This is a comment line
        #Another valid comment line
#####!!!! I can write anything I want!
```

However the following are not valid comment lines:

```
Oh no!!! ##
//This is not a comment-line!
```

ERROR MESSAGES

The following messages will be printed in case of errors:

1- **[R]** If a line is invalid (i.e. neither a comment/empty line nor system parameter configuration). then the following error message should be printed:
"File: <the configuration filename>
Line: <the number of the invalid line>
Message: Invalid configuration line"

2- **[R]** If any of the constraints on the system parameters are not met the following should be printed:
"File: <the configuration filename>
Line: <the number of the invalid value>
Message: Invalid value - constraint not met"

3- **[R]** If a parameter with no default value is not set then the following should be printed:
"File: <the configuration filename>
Line: <the number of lines in the configuration file>

Message: Parameter <parameter name> is not set"
where <parameter name> is the name of the system variable which was not set. Notice that if multiple variables are not set then you should only print an error message regarding the parameter which appeared first in the section System variable.

NOTES:

- Since the errors occur prior to the initializing of the logger, all messages are regular message.
- In case any of the above error occur, you should terminate your program properly (free all resources including the logger spLogger)
- If a configuration file contains multiple errors, you should print the first as error which appears first in the configuration file.

IMPLEMENTATION

You are given a header file SPConfig.h, which includes several functions that needs to be implemented. Make sure you implement the functions as given in the documentation, this is important because the class ImageProc uses the structure SPConfig in its implementation and assumes the interface documentation is met. Further, notice that the header file is not complete and you will probably need to extend it.

EXAMPLE OF A CONFIGURATION FILE

The following configuration file is valid.

```
#this is a valid configuration file
#The following variables don't have default values and must be set
spImagesDirectory = ./images/
spImagesPrefix=img
spImagesSuffix = .png
spNumOfImages = 17

#the following variables have default values, if not set the default value is choosen
# spPCADimension = 20 -> notice the default value is chosen since this is a comment
spPCAFilename = pca.yml
spNumOfFeatures = 100
spExtractionMode = true
        #spMinimalGUI = true -> the default value is chosen since this is a comment
    spNumOfSimilarImages = 5
spLoggerFilename = stdout
```

COMMAND LINE ARGUMENT

The configuration filename is received by the program either:

1- command line argument: The user enter the configuration file using the flag -c.
   For example, if the program name is **SPCBIR** and the configuration filename is myconfig.config then the user must run the program using the following command:
   `>> ./SPCBIR -c myconfig.config`
2- using default value: if not command line argument was entered, then the program will use the default file called **spcbir.config**

Error messages:

1- **[R]** If the user entered invalid command line arguments then the following should be printed:
   "Invalid command line : use -c <config_filename>"
2- **[R]** If the user entered **valid command line arguments** but the file couldn't be open for any reason then the following should be printed:
   "The configuration file <filename> couldn't be open"
3- **[R]** If the user didn't entered a configuration file as a command line argument and the default configuration file (spcbir.config) couldn't be open for any reason then the following should be printed:
   "The default configuration file spcbir.config couldn't be open"

Remark: all messages should be followed by a new line.

PREPROCESSING

Before we go to the image retrieving phase, we first need to extract all the features of the images which are in the directory given by **spImagesDirectory** (refer to System variable for more information about **spImagesDirectory**).

However, the extraction of all images features is a heavy process and it could consume a lot of time. Therefore we need to avoid the extraction processes whenever we load our program. In order to achieve this we support two operation modes, ExtractionMode and nonExtractionMode.

EXTRACTION MODE

In this mode, we first extract the features of each image and then store each of these features to a file which will be located in the directory given by **spImagesDirectory**.

For example, let us assume that the system parameters are the following:

- **spImagesDirectory** = ./images/
- **spImagesPrefix** = img
- **spImagesSuffix** = .png
- **spNumOfImages** = 17
- **spNumOfFeatures** = 100

We first extract 100 features for each of the 17 images in the directory **./images/** (i.e. we extract 100 features for the images **./images/img0.png**, **./images/img1.png**, etc….). Then for each of these images, we store the extracted features to a file (the filename is the same as the image name with the extension **.feats**), thus we store the features of **./images/img0.png** to the file **./images/img0.feats** and the features of **./images/img1.png** to the file **./images/img1.feats** and so on. Notice that you need to store the features in a special format so it will be easier for you to extract these features again from those files. There is no restriction on the way you choose to store the features, but we do recommend you store the actual number of features at the beginning of the features file alongside with the image index.

NON-EXTRACTION MODE

In this mode the features of the images are extracted from the features files that we generated in extraction mode.

For example, for the following system parameters:

- **spImagesDirectory** = ./images/
- **spImagesPrefix** = img
- **spNumOfImages** = 17

The features of the 17 images will be extracted from the features files located in **./images/** . Thus the features of the first image will be extracted from the file **"./images/img0.feats"**, and the features of the second image will be extracted from the file **"./images/img1.feats"** and so on.

SP IMAGEPROC CLASS

For the final project, you are given c++ class (SPImageProc.cpp) which supports different functionalities for the purpose of image processing. A brief manual is given in this section and you are expected to use this class in your project. Notice that all the functionalities needed for the final project are supported (i.e you will not need to use opencv library).

CONSTRUCTOR

```
ImageProc(const SPConfig config);
```

Creates a new ImageProcessing object based on the configuration parameters given by **config**.

GET IMAGE FEATURES

```
SPPoint** getImageFeatures(const char* imagePath,int index,
                           int* numOfFeats);
```

This method extracts approximately **spNumOfFeatures** features for the image given by the path **imagePath**. The image index (i.e the index of the returned features) is given by the argument **index** and the actual number of features extracted will be stored in **numOfFeatures**.

SHOW IMAGE

```
void showImage(const char* imagePath);
```

Given the image located by **imagePath** the method displays the image in a new window. This is a blocking function, that is, the execution of the program continues if the image window is closed. To close the image window, the user only needs to press any key.

Special NOTE: this method operates only if the parameter **spMinimalGUI** is set (i.e the parameter **spMinimalGUI = true**).

QUERY

After the preprocessing is done, the program will ask the user to enter an image path. The message that will be printed is the following **[R]:**

"Please enter an image path:\n"

After the image path is entered the program will find similar images as will be described in the section Search Similar images.

EXITING

The user may terminate the program, simply by entering the line string **"<>"**.

If the user enters "<>", then the program will terminate and free all resources and prints the following message **[R]**:

"Exiting...\n"

SEARCH SIMILAR IMAGES

Given an image path (say the path is given by the variable **imagePath**), we need to search **spNumOfSimilarImages** similar images in the directory **spImagesDirectory**.

Let us recall how we did this in assignment 3:

1. We stored all features of all the images in **spImagesDirectory** in a three-dimensional array.
2. For each feature f_i of the query image, we found the **5 closest features** with respect to f_i.
3. For each image img_i , we kept track of the number of times a feature of img_i was among the **5** closest features found in step 2.
4. Display the 5 most similar images to the query image (in this case the most similar images were the 5 highest ranks (calculated in step 3)).

In the final project, we will search for the **spNumOfSimilarImages** most similar images **(notice that spNumOfSimilarImages** in assignment 3 was 5) as follow:

1. We will store all features in a special data-structure called **KD-TREE (refer to the lecture slides for more information).** The construction time of the **KD-TREE** should be in $O(d \times nlog(n))$ where
   $d = spPCADimension$ and
   $n = the\ total\ number\ of\ features\ for\ all\ images\ in\ spImagesDirectory$
2. For each feature f_i of the query image, find the **k-nearest features** (**k = spKNN)** with respect to f_i using the k-nearest-neighbor search algorithm (more later).
3. For each image img_i in **spImagesDirectory** , keep track of the number of times a feature of img_i is among the **spKNN nearest features** with respect to some feature f_i in step 2.
4. Display the images in **spImagesDirectory** with the highest **spNumOfSimilarImages** ranks as calculated in step 3.

KD-ARRAY

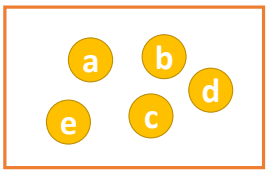
Before proceeding with the implementation of the KD-TREE, we first need to define a new data-structure called KD-ARRAY which supports the following operations (you may define new functionalities as long as you think they will help you in your solution):

1- Init(SPPoint** arr, int size) - Initializes the kd-array with the data given by arr. The complexity of this operation is $O(d \times nlog(n))$.
2- Split(SPKDArray kdArr, int coor) - Returns two kd-arrays **(kdLeft, kdRight)** such that the first $\lceil n/2 \rceil$ points with respect to the coordinate **coor** are in **kdLeft** , and the rest of the points are in **kdRight**.

An example is given below for the case in which the points in SPPoint are a 2D points.

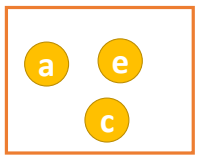
1 **P** = { **a** = (1,2), **b**= (123,70), **c** = (2,7), **d** = (9,11), **e** = (3,4) }

2 kdArr = Init(p, 5)

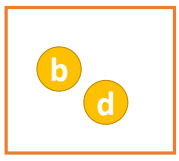


3 (kdLeft,kdRight) = Split(kdArr, 0)

kdLeft =



kdRight =



First (1) we define an array P of 5 points (a,b,c,d and e) in R^2, then (2) we create a new kd-array which contains all the 5 points in P. The operation in (2) should take $O(d \times nlog(n))$. In order to split the kd-array in half (3) with respect to the x-coordinate, we use the function split, which returns two kd-arrays

(kdLeft, kdRight). Notice that the left kd-array contains the points (a,c and e) since they are lowest $\lceil \boldsymbol{n/2} \rceil$ x-coordinate. The operation time in (3) should take $O(d \times n)$

Hint for the implementation:

1- Copy the array you received in the **init** function. Let us denote this array by $\boldsymbol{P}$

2- Create a $d \times n$ matrix $\boldsymbol{A} = (a_{i,j})$ of indexes such that each i^{th} row is the indexes of tshe points in **P** sorted according to their i^{th} dimension, That is:

$$a_{i,j} = the\ index\ of\ the\ j^{th}\ point\ with\ respect\ to\ the\ i^{th} coordinate$$

For the above example your resulting matrix should look like this:

Input array: { **a** = (1,2), **b**= (123,70), **c** = (2,7), **d** = (9,11), **e** = (3,4) }

	0	1	2	3	4
P	a	b	c	d	e

A =					
	0	**2**	**4**	**3**	**1**
	0	4	2	3	1

Sorting each row takes $O(nlog(n))$ time, and since there are d rows, the complexity of the initialization is $O(d \times nlog(n))$

To split your kd-array according to some dimension follow these steps:

1- If you want to split your kd-array according to the i^{th} dimension, then the middle point (i.e $\lceil n/2 \rceil$) splits the points in half according dimension. For the above example, if we want to split the array according to the x-coordinate then the elements marked in **orange** should be in the left kd-array and the **green** elements should be in the right kd-array.

	0	1	2	3	4
P	a	b	c	d	e

A =					
	0	**2**	**4**	**3**	**1**
	0	4	2	3	1

2- Let S be the set of points that are on the left half, and $\bar{S}$ be the set of points on the right. For simplicity create an array X such that $X[i] = 0$ if $P[i] \in S$ and $X[i] = 1$ if $P[i] \in \bar{S}$.

X=	0	1	2	3	4
	0	1	0	1	0

Notice that X can be calculated in $O(n)$ simply by going over the i^{th} row in A (the split dimension)

3- Create two arrays $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ which contain the elements in $\boldsymbol{S}$ and $\overline{\boldsymbol{S}}$ respectively

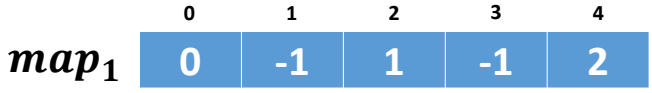
4- Define two arrays which maps the indexes of the elements in $\boldsymbol{P}$ to their indexes in $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$. That is:

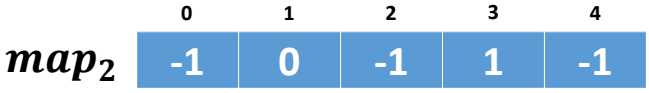
$$map_1[i] = \begin{cases} j & if\ P[i] = P_1[j] \\ -1 & otherwise \end{cases}$$

And

$$map_2[i] = \begin{cases} j & if\ P[i] = P_2[j] \\ -1 & otherwise \end{cases}$$

for the above example:





5- To build the sorted matrix (without sorting the matrix from scratch) scan the sorted matrix of the original kd-array and use the arrays X, map_1 and $map_2$to achieve complexity $O(d \times n)$. The resulting matrices for the above example should be:

NOTE: There is a solution in which map_1 and map_2 are not needed.

CONSTRUCTION OF A KD-TREE USING KD-ARRAY

We first need to define the kd-tree node (called **KDTreeNode**), the node must contain the following information:

1- Dim = The splitting dimension
2- Val = The median value of the splitting dimension
3- Left = Pointer to the left subtree
4- Right = Pointer to the right subtree
5- Data = Pointer to a point (only if the current node is a leaf) otherwise this field value is NULL

To initialize the kd-tree we first need to build a kd-array with all features stored in it (this takes $O(d \times nlog(n))$ time) and then we recursively apply the following steps:

1- If the size of the kd-array = 1 create and return the node:
   a. Dim = invalid
   b. Val = invalid
   c. Left = NULL
   d. Right = NULL
   e. Data = the only point in the kd-array
2- Split the kd-arrary according to the dimension given by the splitting criteria (ie. The criteria is set by the system parameter **spKDTreeSplitMethod)** this should take $O(d \times n)$:
   a. **MAX_SPREAD** - Define the spread of the i^{th} dimension to be the difference between the maximum and minimum of the i^{th} coordinate of all points. In the example given in the previous section the spread of the x-coordinate is 123 - 1 = 122 and for the y-coordinate is 70-2 =68. Split the kd-array according the dimension with the highest spread (if there are several candidates choose the lowest dimension)
   b. **RANDOM** - choose a random dimension
   c. **INCREMENTAL** - if the splitting dimension of the upper level was i, then the splitting dimension at this level is $(i + 1)\%d$
3- Create a new node with the following values:
   a. Dim = i_{split}
   b. Val = The median according the dimension i_{split}
   c. Left = Recursively build the kd-tree with respect to the left kd-array
   d. Right = Recursively build the kd-tree with respect to the right kd-array
   e. Data = NULL

Refer to the lecture on kd-trees for more information.

K-NEAREST NEIGHBOR SEARCH

A pseudo code for the nearest neighbor search is given below:

Init:

Maintain a BPQ of the candidate nearest neighbors, called 'bpq'
Set the maximum size of 'bpq' to spKNN

Starting at the root of the kd-tree, execute the following procedure in order to find the K nearest neighbors of the test point P:

kNearestNeighbors: **KDTreeNode** curr , **BPQ** bpq, **Point** P

```
if curr == NULL
   return

/* Add the current point to the BPQ.  Note that this is a no-op if the
 * point is not as good as the points we've seen so far.*/
if isLeaf(curr) then
      enqueue (index(current), distance(curr,p)) into bpq
      return

/* Recursively search the half of the tree that contains the test point. */
if(P[curr.dim] <= curr.val) then
         recursively search the left subtree
else
         recursively search the right subtree

/* If the candidate hypersphere crosses this splitting plane, look on the
 * other side of the plane by examining the other subtree*/

if:

  bpq isn't full

     -or-

  |curr.val - P[curr.dim]| is less than the priority of the
                          max-priority    element of bpq

then
      recursively search the other subtree on the next axis
```

See note next page ->

Special NOTE: in the search process, please note the highlighted lines.
When you **enqueue (index(current),distance(curr,p))**, if you enqueued the L2-**Squared** distance, then you need to check **(curr.val - P[curr.dim])^2** instead of **|curr.val - P[curr.dim]|.** If you enqueued the L2 distance, then you need to check **|curr.val - P[curr.dim]|**

SHOWING RESULTS

After the search is done, you should have **spNumOfSimilarImages** indexes of the best candidates for the given query image. To present the result, we support two mode MinimalGUI and non-MinimalGUI, this value is set by the system parameter **spMinimalGUI**.

MINIMAL GUI

In this mode, you need to display the images one after the other (use the method show image defined in SPImageProc class).

For example given that:

- the query image path is "./query.png"
- **spNumOfSimilarImages** = 2
- and the best candidates are ./images/img9.png and ./images/img4.png

You first need to show the image **./images/img9.png**. If the user presses any key, the next image should be displayed (i.e **./images/img4.png**)

NON-MINIMAL GUI

In this mode, you need to print the result in the following format **[R]**:

"**Best candidates for - <query image path> - are:\n**"
"**<best candidate image path>\n**"
"**<second best candidate image path>\n**"
and so on.

For example given that:

- the query image path is **"./query.png"**
- **spNumOfSimilarImages** = 2
- and the best candidates are **"./images/img9.png"** and **"./images/img4.png"**

then you need to print the following messages:
"Best candidates for - ./query.png - are:\n"
"./images/img9.png\n"
"./images/img4.png\n"

FLOWCHART

The flowchart of the application is given below.

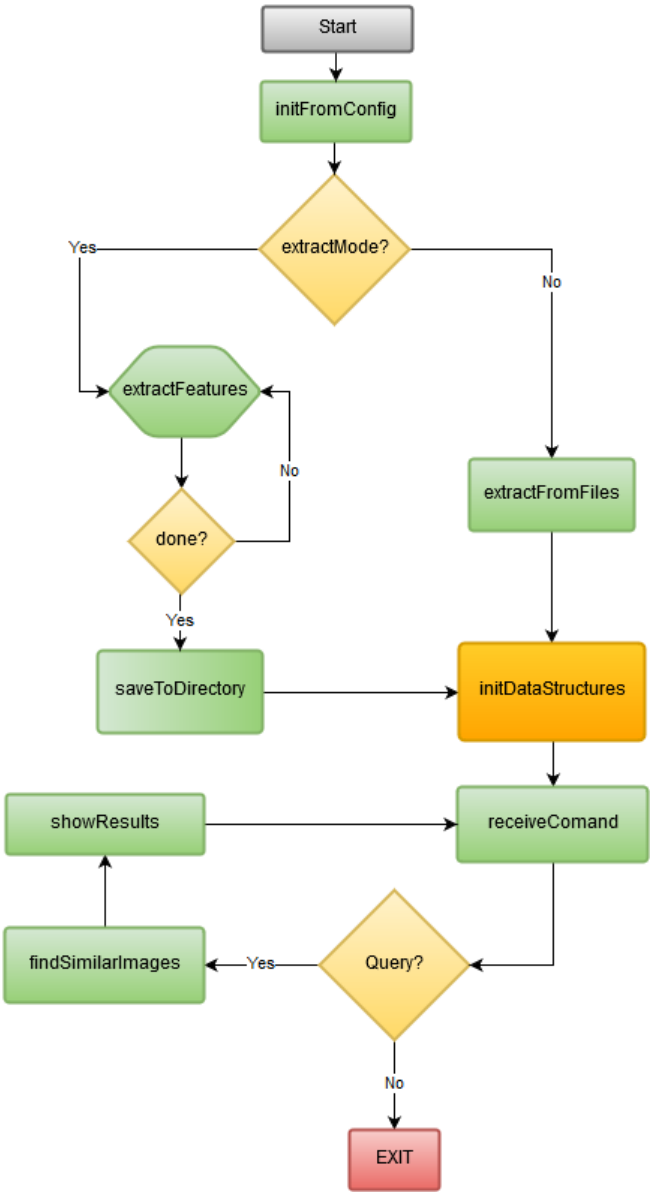


Figure 1 - Flowchart of the CBIR system. Initially, features of all images are extracted and saved in a data-structure which supports nearest neighbor search (i.e KD-Tree). Next, a query image is received and a nearest neighbor search algorithm is used to find similar images in the database. If no query image was received, the program will terminate.

MIXING C WITH C++

Although this is a C project, but in order to use the class SPImageProc.cpp, your main function should be in a c++ source file (i.e main.cpp). Thus your main function should only be in the source file main.cpp doing otherwise will cause an unexpected outcome. Carefully review and follow the guideline below

C++ programming language is far more advanced than C and it is relatively easy to work with. **FOR THIS REASON YOU SHOULD NOT (FOR ANY REASON) USE ANY OF THE C++ LIBRARIES. ANY SOLUTION MIXING C and C++ WILL NOT BE ACCEPTED.**

Important notes when you come to implement you main.cpp file:

- If you want to use C libraries in main.cpp just include the library name without the .h extension and with c at the beginning. For example if you want to use <stdlib.h> then you need to #include <cstdlib>
- If you want to use C source files that you implemented, then you need to use the special macro **extern**. For example if you want to use SPPoint and SPLogger then you need to write the following lines at the beginning of main.cpp:

```
#include <cstdlib> //include c library
extern "C"{
//include your own C source files
#include "SPPoint.h"
#include "SPLogger.h"
}
//include C++ source files
#include "SPImageProc.h"

int main(){
      //Your implementation of the main function
      return 0;
}
```

- The source file main.cpp must contain only one function (i.e the main function). Don't define any auxiliary functions in main.cpp. If you want to implement helper functions then create another **C++ source file (name it main_aux)** and include it in your main function as has been described in previous section.
- The only C++ source files your project must have are **main.cpp**, **SPImageProc.cpp**. and **main_aux**. If you use any other C++ files **your project will not be accepted!**
- In main_aux.cpp you may add helper functions as we did in assignment 3.

MAKEFILE

You will need to write your own **makefile** when submitting your code. The resulting program executable name should be **SPCBIR.** For your convenience a partial makefile is added, refer to the first tutorial on makefiles if needed. Make sure you add the makefile for the unit tests.

SPECIAL REMARKS

- You may assume that each line in the configuration file contains no more than 1024 characters (doesn't include the null character).
- You may assume that the path of any file (i.e image path, pca file path) contains no more than 1024 characters (doesn't include the null character).
- You must print informative errors/warning/info message to the log file. There will be no restriction on the format of the messages. Just make sure in case of **error messages** your program should terminate and free all memory resources.
- You also need to write a unit test for each module you implement. Such as KD-Tree or KD-Array and so on. A partial unit test for the logger is provided.

SUBMISSION

Please submit a zip file named **id1_id2_finalproject.zip** where id1 and id2 are the ids of the partners. The zipped file must contain the following files:

- All header/source files (main.cpp, SPImageProc.cpp, SPImageProc.h etc...)
- partners.txt – This file must contain the full name, id and moodle username for both partners. Please follow the pattern in the assignment files. **(do not change the pattern)**
- **unit_tests** – The directory which contains the all unit tests (including the looger unit test which you will need to extend)
    - Source files: sp_logger_unit_test.c etc...
    - Header files: unit_test_util.h
- **The makefiles for each unit test -** (including SPLoggerTest.make)
- makefile – the makefile provided in the assignment files that compiles the main program **(SPCBIR)**.

REMARKS

- For any question regarding the assignment, please don't hesitate to contact Moab Arar by mail: moabarar@mail.tau.ac.il.
- Borrowing from others' work is not acceptable and may bear severe consequences.

GOOD LUCK!