Assigment
Clustering algorithm *K-means*

1 K-means Clustering Algorithm

Clustering is a technique used to divide a dataset into groups or clusters and discover hidden patterns within the data. It involves grouping similar data objects into separate clusters. This technique presents wide applications in areas such as artificial intelligence, biology, data compression, and data mining, among others.

The *K-means* clustering algorithm is an unsupervised clustering method used to classify a dataset into K different clusters, where the number of clusters K is known beforehand.

Given a cloud of m points $P = \mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_m}$, where each point has n dimensions $\mathbf{x_i} = (x_{i1}, x_{i2}, \ldots, x_{in})$, the *K-means* algorithm assigns each point to a specific cluster $c_1, c_2, \ldots, c_K$, where $K < n$. The algorithm assigns each point to the cluster whose centroid is at the minimum Euclidean distance.

The algorithm works as follows:

1. Select K random centroids $(\mathbf{ce_1}, \mathbf{ce_2}, \ldots, \mathbf{ce_K})$, one for each cluster, from the point cloud P.

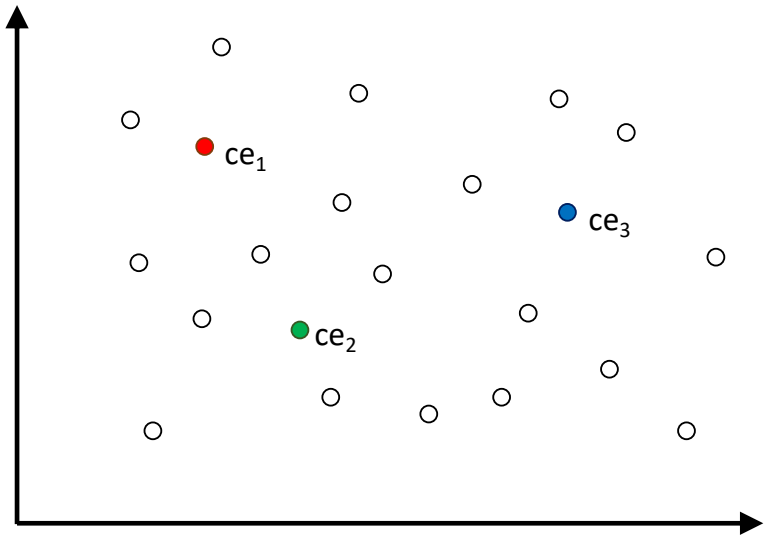

Figure 1: Random Selection of Centroids ($K = 3$).

2. Assign each point to the nearest centroid (smallest Euclidean distance).

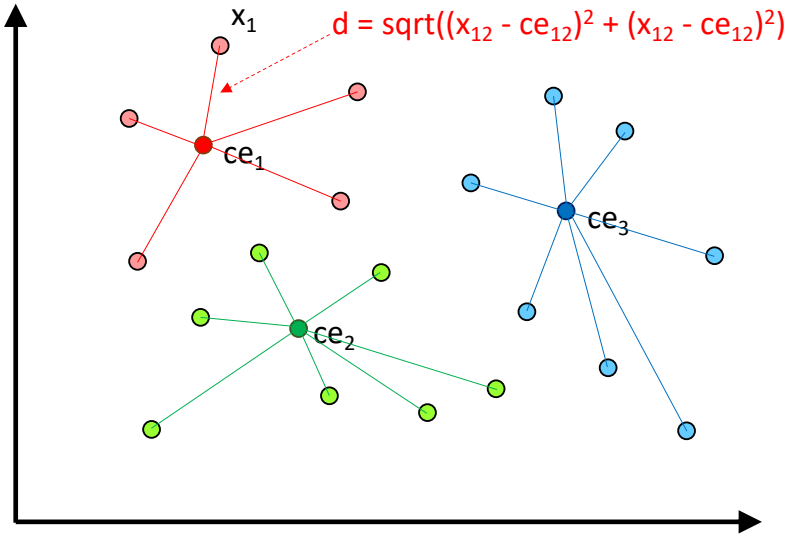

Figure 2: Assignment of points to the clusters.

3. Recalculate the centroids of each cluster as the mean of the data points assigned to that cluster.

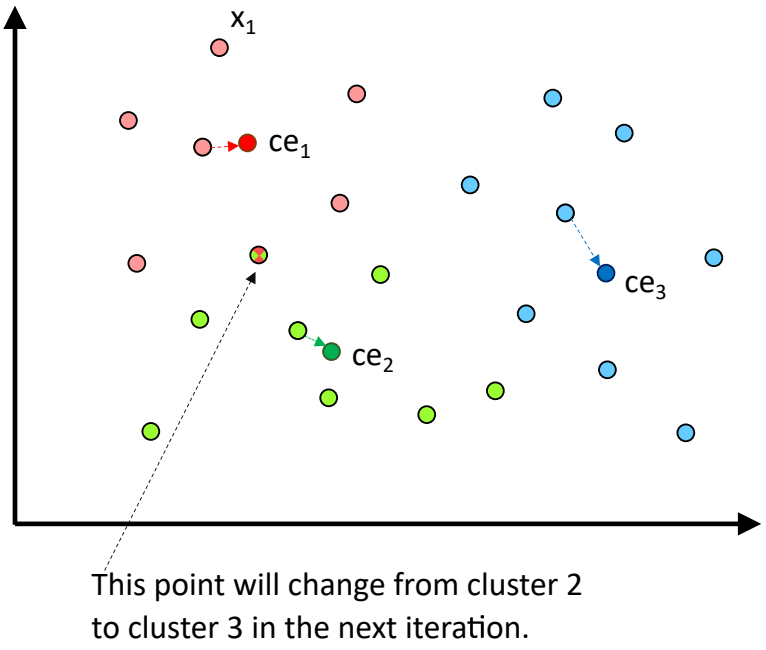

Figure 3: Calculation of the new centroids.

4. Repeat steps 2 and 3 until:
   (a) The centroids do not change significantly, i.e., between one iteration and the previous one, the maximum movement of all centroids within each cluster is less than a given threshold.
   (b) The number of data point changes from one cluster to another between one iteration and the previous one is less than a given threshold.

(c) A maximum number of iterations is reached.

1.1 Sequential Code

A C program implementing this version of the *K-means* algorithm will be provided to the students. This program receives the following command line arguments:

1. `argv[1]`: Input data file.
2. `argv[2]`: Number of clusters.
3. `argv[3]`: Maximum number of iterations.
4. `argv[4]`: Threshold for the number of data point changes from one cluster to another between one iteration and the previous one.
5. `argv[5]`: Threshold for the maximum movement of centroids between one iteration and the previous one.
6. `argv[6]`: Output file. Cluster assigned to each line of the file.

The input file has one line for each data point in the cloud of points to be classified, with each dimension of the points separated by tabulators. Test input files of different sizes and dimensions will be provided on the virtual campus.

1.2 Assignment

Each group must submit a parallel version of the provided C source file. The objective of the practice is to minimize the computation time as much as possible without **altering the results obtained in the sequential version**.

It is not necessary to parallelize the reading of the input file.

2 C Functions used in the sequential code

1. `strtok`: splits a string into a series of *tokens* using a delimiter.

```
char *strtok(char *str, const char *delim);
```

Parameters:

- `str`: The content of this string **is modified** and divided into smaller strings (*tokens*).
- `delim`: It is the string containing the delimiter. This may vary from one call to another.

Return:

- `char *`: This function returns a pointer to the next *token* found in the string. It returns a null pointer `NULL` if there are no more tokens to split.

It is usually used in a `while` loop that invokes `strtok` in each iteration until the function returns `NULL` (no more tokens). Remember, the string `str` is modified by the function, and its content can be lost.

Example:

```c
#include <string.h>
#include <stdio.h>

int main () {
   char str[100] = "This is a token;other token;and other token";
   const char delim[2] = ";";
   char *token; //no need to reserve memory

   //reading the first token
   token = strtok(str, delim);

   //Loop to search for tokens
   while( token != NULL ) {
      printf( "%s\n", token );
      token = strtok(NULL, delim);
   }
   printf("%s\n",str);

   return(0);
}
```

2. `memcpy`: copies `n` bytes from the memory area pointed by `src` to the memory area pointed by `dest`.

```c
void *memcpy(void *dest, const void *src, size_t n);
```

Parameters:

- `dest`: Pointer to the destination array where the content will be copied, converted to a pointer of type `void*`. The memory space must be previously allocated.
- `src`: Pointer to the source array of data to be copied, converted to a pointer of type `void*`.
- `n`: The number of bytes to copy.

Example Usage:

```c
#include <string.h>
#include <stdio.h>

int main () {
  int *vector, *vector2;

  vector = (int*) calloc(10,sizeof(int));
  vector2 = (int*) calloc(10,sizeof(int));
  if (vector==NULL || vector2==NULL) {
     printf("Error allocating memory\n");
     exit(-1);
  }

  for(i=0; i<10; i++) {
    vector[i]=rand()%5;
  }

  //Copies the 10 integers pointed by vector to the
  //memory area pointed by vector2
```

```
  memcpy(vector2, vector, 10*sizeof(int));
  return 0;
}
```

3 Using Linear Memory for Two-Dimensional Structures

When allocating memory with `malloc` or `calloc` in C for two-dimensional (or higher-order) structures, the data is stored in memory unidimensionally, and you can only use one index to access the elements.

For example, the following C code represents the allocation of a 3-row by 4-column matrix which is filled with random values.

```
int main()
{
    int rowx=3, cols=4, i;
    int *matrix = NULL;

    //Matrix allocation
    matrix = (int *)malloc(rows*cols*sizeof(int));
    if (matriz == NULL) {
        fprintf(stderr,"Error allocating memory\n");
        exit(-1);
    }

    //You can only use one index to access matrix positions
    for (i=0; i<rows*cols; i++) {
        matrix[i] = rand() % 10;
        printf("matrix[%d]: %d\n",i,matrix[i]);
    }
    free(matrix);
}
```

The data in memory is stored linearly, and by using the index, you can access each element, as shown on the left side of Figure 4.

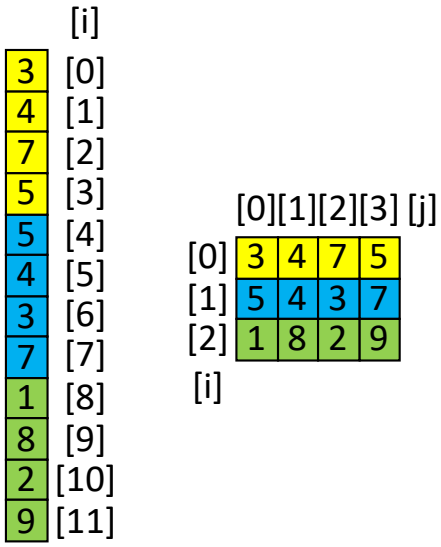

Figure 4: Representation of the matrix in memory.

If you want to access the elements of the matrix by rows and columns, using two indices, one to index the row and the other the column, and thus with two nested loops, you can resort to one of the following solutions:

1. Using an expression that operates with the row and column indices to obtain the linear index (depends on the number of columns):

```
int main()
{
    int rows=3, cols=4, i, j;
    int *matrix = NULL;
    //Matrix allocation
    matrix = (int *)malloc(rows*cols*sizeof(int));
    if (matrix == NULL) {
        fprintf(stderr,"Error allocating memory\n");
        exit(-1);
    }
    for (i=0; i<rows; i++) { //rows
        for (j=0; j<cols; j++) { //columns
            matrix[i*cols + j] = rand() % 10;
            printf("matrix[%d, %d]: %d\n",i,j,matrix[i*cols + j]);
        }
    }
    free(matrix);
}
```

2. Using a C macro that uses the previous expression.

```
//Macro for accessing bidimensional matrices
#define accesMat(m,i,j,col) m[i*col+j]
int main()
{
    int rows=3, cols=4, i, j;
    int *matriz = NULL;

    //Matrix allocation
```

```
    matriz = (int *)malloc(rows*cols*sizeof(int));
    if (matrix == NULL) {
        fprintf(stderr,"Error allocating memory\n");
        exit(-1);
    }

    for (i=0; i<rows; i++) { //rows
        for (j=0; j<cols; j++) { //columns
            accesMat(matrix,i,j,cols) = rand() % 10;
            printf("matrix[%d, %d]: %d\n",i,j,accesMat(matrix,i,j,cols));
        }
    }
    free(matrix);
}
```