**ΕΡΓΑΣΙΑ ΣΤΟ ΜΑΘΗΜΑ ΑΡΧΕΣ ΚΑΙ ΓΛΩΣΣΕΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ**

**ΕΙΣΑΓΩΓΗ ΣΤΗΝ C**

# ΔΗΜΙΟΥΡΓΙΑ ΠΡΟΓΡΑΜΜΑΤΟΣ ΓΙΑ ΕΠΕΞΕΡΓΑΣΙΑ GRAYSCALE ΕΙΚΟΝΩΝ ΜΕ ΧΡΗΣΗ ΜΑΣΚΩΝ ΚΑΘΩΣ ΚΑΙ ΕΦΑΡΜΟΓΗ CONNECTED COMPONENTS ΣΕ ΔΥΑΔΙΚΕΣ ΕΙΚΟΝΕΣ

**Παπαντώνης Χρήστος**

**ΜΔΕ Ηλεκτρονικού Αυτοματισμού**

**ΑΜ : 2012 515**

**Επιβλέπων : Δ . Μαρούλης , Ε. Κωστοπούλου**

## Introduction

Αρχικά για την δημιουργία ενός προγράμματος επεξεργασίας εικόνας , πρέπει να δημιουργήσουμε συναρτήσεις οι οποίες θα εκτελούν το διάβασμα της εικόνας και την εγγραφή της επεξεργασμένης εικόνας σε ένα καινούργιο αρχείο. Έτσι αφού διαβάσουμε την εικόνα θα πρέπει να την αποθηκεύσουμε προσωρινά σε ένα δισδιάστατο πίνακα για να μπορέσουμε να την επεξεργαστούμε. Επίσης λόγω του ότι χρειαζόμασταν και κάποιες επιπλέον πληροφορίες για την εικόνα , δημιουργήσαμε το παρακάτω **struct** (Image).

```
struct image{
    unsigned short **imageMatrix;
    int width;
    int height;
};
```

Το παραπάνω **struct** αποτελείται από τον δισδιάστατο πίνακα (**imageMatrix**) που περιέχει τις τιμές του εκάστοτε Pixel της εικόνας και τις τιμές του πλάτους και ύψους της εικόνας. Οι τιμές του πίνακα είναι τύπου **unsigned short** για να δεσμεύουμε μικρό χώρο στην μνήμη για το εκάστοτε pixel καθώς η εικόνες μας είναι gray scale και Binary (16,8 bit depth).

Το image i/o (input/output) υλοποιήθηκε με την χρήση των παρακάτω συναρτήσεων :

```
struct image *scanSize(struct image* the_image , char *fileName);
struct image* readImage(struct image* the_image,char *fileName , int*binary );
struct image *initOut_image(struct image*out_image , struct image *the_image);
void writeImage(struct image*out_image , char *fileName);
struct image *importStaff(struct image*out_image , struct image *the_image , int*binary);
struct image*copyImage(struct image* the_image, struct image* out_image );
#endif // !I_Oheader
```

Η **scanSize** παίρνει σαν είσοδο το Path της εικόνας και το malloc-ed image struct (the_image) ανοίγει το αρχείο και κάνει parse την εικόνα. Ανάλογα με το πόσα "\n" πόσα "\t" θα διαβάσει ορίζει ανάλογα το ύψος και το πλάτος της εικόνας αντίστοιχα. Οι τιμές των i , j (temp vars για το διάβασμα της εικόνας) γίνονται Initialize στην μονάδα καθώς η C ξεκινάει να μετράει από το 0 οπότε θα έβγαιναν N-1 οι τιμές μας.

```
parser = getc(fp);
printf("Successfully Opening the image...\n");
do{
    parser = fgetc(fp);
        if(i ==1 && parser == '\t' ){j +=1;}
        if(parser == '\n'){i +=1;}
}while (parser !=EOF );
```

Η **readImage** παίρνει πάλι σαν είσοδο το path της εικόνας, το struct της εικόνας καθώς και μια Boolean μεταβλητή (0 ή 1) την οποία την περνάμε by reference από την main για να μπορεί η συνάρτηση να αλλάξει την τιμή της. Η συνάρτηση τώρα, διαβάζει μια προ μία τις τιμές από το αρχείο, κάνει dymamic malloc για το εκάστοτε pixel και του αναθέτει την τιμή που διάβασε. Αν έχει διαβάσει τιμές από 1 εώς 255 και μεγαλύτερες από 65.535 η binary var γίνεται 1 αλλιώς 0. Δηλαδή από τις τιμές των pixel καταλαβαίνουμε αν έχουμε grayscale ή binary εικόνα. Έτσι ο χρήστης δεν μπορεί να μας δώσει καταλάθως binary εικόνα αντί για graysale και ούτε το αντίστροφο.

```
the_image->imageMatrix  = (unsigned short**)malloc(the_image->height*sizeof(unsigned short*));
for (h = 0; h < the_image->height ; h++)
{
    *(the_image->imageMatrix + h) = (unsigned short*)malloc(the_image->width*sizeof(unsigned short));
    for (w = 0; w < the_image->width; w++){
        //data = getc(fp);
        fscanf(fp, "%d", &data);
        if ((data < 255 && data > 1)||  data > 255 ){
            *binary = 1;
        }
        *(*(the_image->imageMatrix + h) + w) = data;
    }
}
```

Η **initOut** παίρνει την malloc –ed και ως προ έξοδο εικόνα , κάνει δυναμική ανάθεση μνήμης σε κάθε pixel και του δίνει την τιμή 0. Η **writeImage** παίρνει πάλι ως είσοδο την επεξεργασμένη πλέον εικόνα καθώς και το όνομα με το οποίο θέλουμε αν την αποθηκεύσουμε και την γράφει σε ένα αρχείο .txt.

```
FILE *fp;
if(out_image != NULL){
    fp = fopen(fileNameOut,"w");
        for(i = 0 ; i < out_image->height; i++){
            for(j =0 ; j < out_image->width; j++){
                fprintf(fp,"%d\t",*(*(out_image->imageMatrix + i) + j));
            }
            fprintf(fp,"\n");
        }
        printf("%s image has been successfully produced \n" , fileNameOut);
```

Η **copyImage** αντιγράφει την Output εικόνα στην input εικόνα (out_image to the_image). Την αντιγραφή εικόνας την χρειαζόμαστε όταν επεξεργαστούμε μία φορά την εικόνα και μετά θέλουμε να συνεχίσουμε την επεξεργασία της ήδη επεξεργασμένης εικόνας. Έτσι δεν χρειάζεται να τρέχουμε το πρόγραμμα πάλι από την αρχή.

Η **importStaff** καλείται από την main και ουσιαστικά είναι το I/o menu το οποίο καλεί όλες τις παραπάνω συναρτήσεις.

```
    // I/O Menu
struct image* importStaff(struct image*the_image , struct image *out_image , int*binary){
    char fileName[32];
    do{
        printf("-> Write name of the file you want to open . . . \n");
        printf("-> Some available choices for connencted components are : \n1.image.txt (bin)\n2.letters.txt(bin)\n3.shapes.txt(bin
        printf("-> Or for grayscale images some available choices are :\n1.test2.txt (gray)\n2.big1.txt\n3.mySynth.txt \n");
        printf("-> Or give me another name that exist in the directory\n");
        gets(fileName);
        the_image = scanSize(the_image, fileName);//scans size of image
        if (the_image->height != 0){
            the_image = readImage(the_image , fileName , binary);//scans the image
        }
    }while (the_image ->height == 0);
    return the_image;
    }
```

## Mask Creation – Filtering Options

Προκειμένου να φιλτράρουμε μία grayscale εικόνα πρέπει πρώτα να δημιουργήσουμε τις αντίστοιχες μάσκες. Αφού πρώτα διαβάσουμε την εικόνα και η binary var == 1 τότε το πρόγραμμα μας πετάει στο menu δημιουργίας μάσκας(**maskMenu**). Την μάσκα την αποθηκεύουμε σε ένα struct που έχει την παρακάτω δομή :

```
struct mask
{
    int ** maskArray; // array of the filter mask
    int O; //Order of growth from center
    int centerX;
    int centerY;
};
```

Το κάθε struct γίνεται malloc μόλις κληθεί η αντίστοιχη συνάρτηση για την υλοποίηση της μάσκας. Οι μάσκες που δημιουργήσαμε είναι τετράγωνη , κυκλική και ρόμβος. Οι αντίστοιχε συναρτήσεις που τις υλοποιούν είναι οι παρακάτω:

```
struct mask *maskMenu();
struct mask *squareMaskConstructor(struct mask* squareMask, int aktina);
struct mask *circleMaskConstructor (struct mask* circleMask , int aktina);
struct mask *diamondMaskConstructor (struct mask* diamondMask , int aktina );
```

Οι συναρτήσεις παίρνουν ως είσοδο ένα mask struct από το menu επιλογής καθώς και την ακτίνα η οποία δείχνει πόσο μεγάλη να είναι η μάσκα μας ξεκινώντας από το κέντρο (1,1).

```
Your mask will be looking  like  the following . . .
1    1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1    1
1    1    1    1    1    1    1    1    1
```

**Τετράγωνη μάσκα ακτίνας 4**

```
Your mask will be looking  like  the following . . .
0    0    0    0    1    0    0    0    0
0    0    1    1    1    1    1    0    0
0    1    1    1    1    1    1    1    0
0    1    1    1    1    1    1    1    0
1    1    1    1    1    1    1    1    1
0    1    1    1    1    1    1    1    0
0    1    1    1    1    1    1    1    0
0    0    1    1    1    1    1    0    0
0    0    0    0    1    0    0    0    0
```

**Κυκλική μάσκα ακτίνας 4**

```
Your mask will be looking  like  the following . . .
0    0    0    0    1    0    0    0    0
0    0    0    1    1    1    0    0    0
0    0    1    1    1    1    1    0    0
0    1    1    1    1    1    1    1    0
1    1    1    1    1    1    1    1    1
0    1    1    1    1    1    1    1    0
0    0    1    1    1    1    1    0    0
0    0    0    1    1    1    0    0    0
0    0    0    0    1    0    0    0    0
```

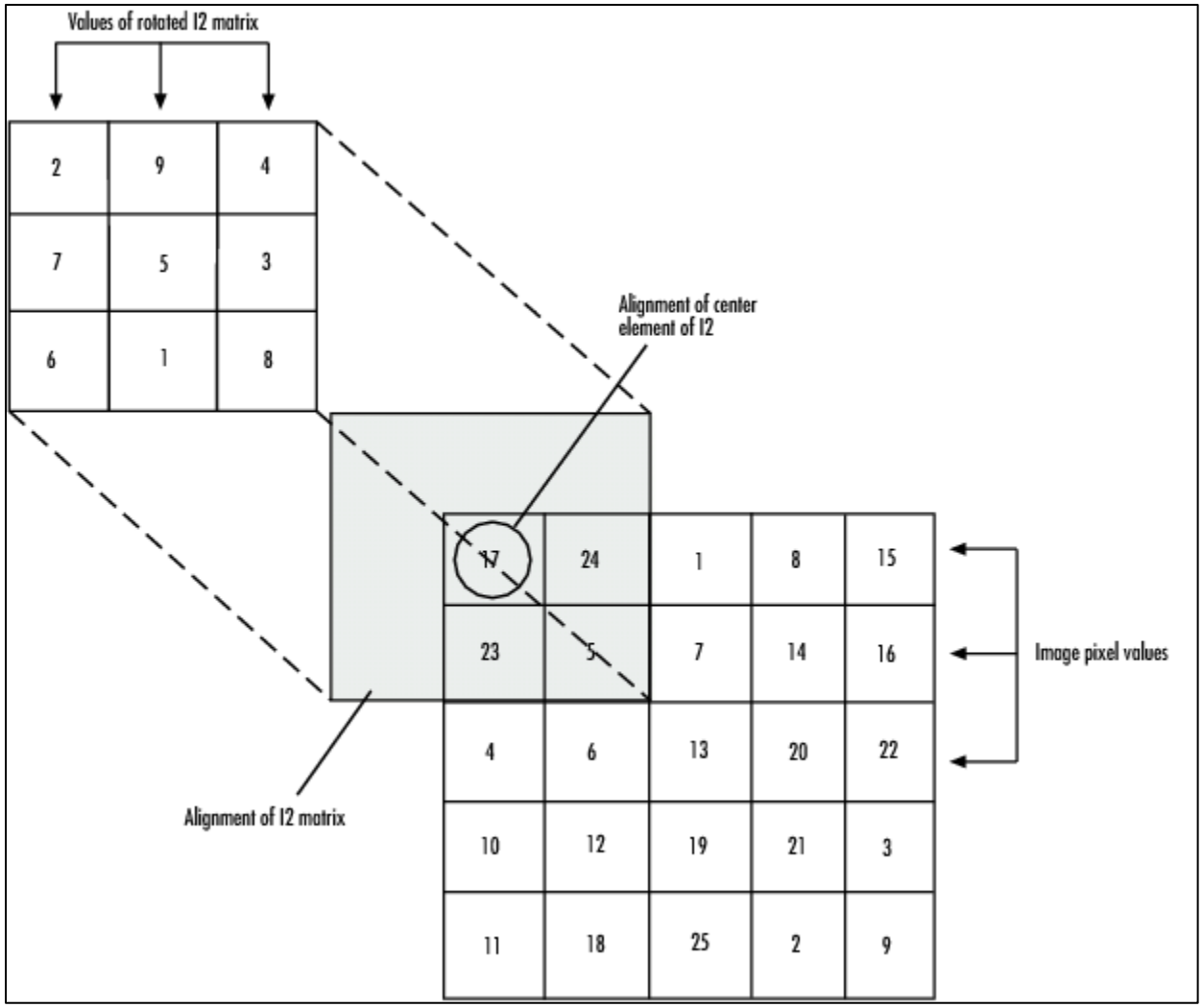
**Μάσκα ρόμβος ακτίνας 4**

Οι παραπάνω εικόνες προέρχονται από το terminal του προγράμματος κατά την διάρκεια λειτουργίας του. Η μάσκα πρέπει να έχει πάντα μονές γραμμές και στήλες προκειμένου να υπάρχει πάντα κεντρικό pixel το οποίο μας δείχνει πάντα σε ποιο pixel της εικόνας θα κάνουμε την αλλαγή/επεξεργασία.

Μετά την επιλογή μάσκας το πρόγραμμα μας πετάει σε μια επιλογή να επιλέξουμε αλγόριθμο φιλτραρίσματος. Σε αυτό το πρόγραμμα πρακτικά υλοποιήσαμε 4 αλγορίθμους : Erosion , Dilation ,Open ,Close . Στην ουσία όμως οι τελευταίοι είναι η εφαρμογή των πρώτων με διαδοχική σειρά (Open = Erosion -> Dilation , Close = Dilation -> Erosion). Οι Συναρτήσεις υλοποίησης είναι οι ακόλουθες :

```
struct image * filterMenu (struct image *the_image,struct image *out_image, struct mask *mainMask);
struct image*Erosion (struct image *the_image, struct image* out_image, struct mask*squareMask);
struct image* Dilation(struct image *the_image, struct image* out_image, struct mask*squareMask);
```

Σαν είσοδο οι συναρτήσεις δέχονται τα 2 Image struct Input , output καθώς και την μάσκα την οποία θα εφαρμόσουν πάνω στην εικόνα. Η συναρτήσεις κάνουν parse την εικόνα με την μάσκα , με βάση το κεντρικό pixel της μάσκας και κάνει αντιπαραβολή τις αντίστοιχες θέσεις πάνω στην εικόνα. Αν κάποιο Pixel της μάσκας έχει τιμή 1 παίρνει την τιμή του αντίστοιχου Pixel της εικόνας προς επεξεργασία , αλλιώς αν έχει την τιμή 0 δεν λαμβάνει υπ' όψιν το αντίστοιχο Pixel της εικόνας. Το πώς εφαρμόζεται η μάσκα πάνω στην εικόνα μπορούμε να το δούμε και την παρακάτω εικόνα.




Συγκεκριμένα τώρα, η **Erosion** εφαρμόζει την μάσκα πάνω στην εικόνα. Κοιτάζει αρχικά ώστε τα στοιχεία της μάσκας να είναι μέσα στο όρια της εικόνας της ως προς επεξεργασία εικόνας. Όποια από αυτά ανήκουν μέσα στην εικόνα και έχουν την τιμή 1 για να παίρνει και τις αντίστοιχες τιμές των pixel της εικόνας. Μόλις βρει ποια Pixel θα επεξεργαστεί ψάχνει να βρει το μικρότερο και το βάζει στην Output εικόνα στην θέση που βλέπει το κεντρικό Pixel της μάσκας. Αυτό συμβαίνει διαδοχικά μέχρι το κεντρικό Pixel της μάσκας να έχει περάσει πάνω από όλα τα Pixel της εικόνας.

Η **Dilation** από την άλλη βρίσκει το μέγιστο Pixel από αυτά που βλέπει η μάσκα και το βάζει στην θέση που "βλέπει" το κεντρικό της μάσκας.

Παρακάτω φαίνεται ο αλγόριθμος και οι συνθήκες για την υλοποίηση του Erosion.

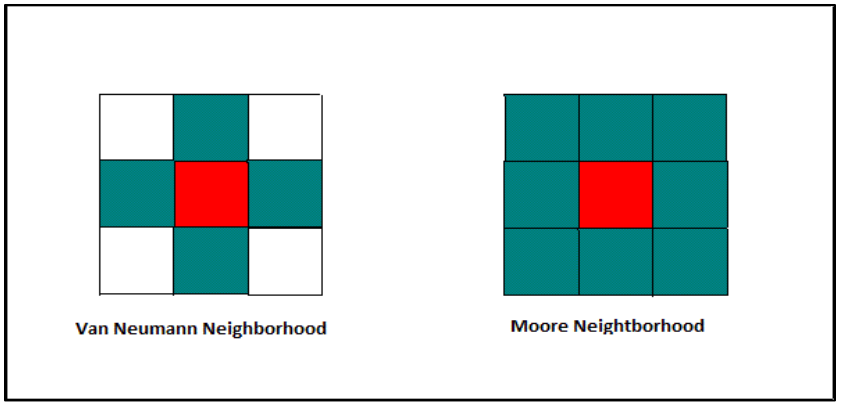
```
for (i = 0; i < the_image->height ; i++){//2 loops for parsing image
    for (j = 0; j < the_image->width; j++){
        min = 1000000;
        for (a = -mainMask->O; a <= mainMask->O; a++){//2 loops for mask
            for (b =  -mainMask->O; b <=  mainMask->O; b++){
                if( (i +a) >= 0 && (i + a) <  the_image->height &&  ( j + b ) >= 0 && (j +b) < the_image->width){///checks boundaries with mask
                    if (mainMask->maskArray[a +  mainMask->O][b +  mainMask->O] == 1){  //checks whether mask's element value is 1
                        if (*(*(the_image->imageMatrix + i + a)+ j + b) < min){ //compares the element value with the minimum value of the mask
                            min = *(*(the_image->imageMatrix + i + a)+ j + b);
                        }
                    }
                } /* ends if mask == 1 */
            } /* ends loop over b */
        } /* ends loop over a */
        //printf("%d\n",min);
        out_image->imageMatrix[i][j] = min;
    } /* ends loop over j */
} /* ends loop over i */
return out_image;
```

```
for (i = 0; i < the_image->height ; i++){//2 loops for parsing image
    for (j = 0; j < the_image->width; j++){
        max = 0;
        for (a = -mainMask->O; a <= mainMask->O; a++){//2 loops for mask
            for (b =  -mainMask->O; b <=  mainMask->O; b++){
                if( (i +a) >= 0 && (i + a) <  the_image->height &&  ( j + b ) >= 0 && (j +b) < the_image->width){///checks boundaries with mask
                    if (mainMask->maskArray[a +  mainMask->O][b +  mainMask->O] == 1){  //checks whether mask's element value is 1
                        if (*(*(the_image->imageMatrix + i + a)+ j + b) > max){ //compares the element value with the minimum value of the mask
                            max = *(*(the_image->imageMatrix + i + a)+ j + b);
                        }
                    }
                } /* ends if mask == 1 */
            } /* ends loop over b */
        } /* ends loop over a */
        //printf("%d\n",min);
        out_image->imageMatrix[i][j] = max;
    } /* ends loop over j */
} /* ends loop over i */
return out_image;
```

**Erosion (up) – Dilation(down)**

## Connected Components

Σκοπός του συγκεκριμένου αλγόριθμου είναι να βρίσκει και να μαρκάρει γειτονικές περιοχές σε μια εικόνα. Στο πρόγραμμα μας αυτό γίνεται για δυαδικές εικόνες(Binary) δηλαδή ασπρόμαυρες. Για τον υπολογισμό των CC (connected components) αρχικά πρέπει να πούμε το πώς ορίζουμε μία γειτονιά. Σε αυτή την εργασία συγκεκριμένα θα ασχοληθούμε με τα 2 γειτονιές την Moore και Van Neumann.




Η γειτονιά Moore θεωρεί γειτονικά Pixel όλα τα pixel που ουσιαστικά ακουμπάνε πάνω στο κεντρικό Pixel (το προς εξέταση pixel). Από την άλλη η Van Neumann γειτονία θεωρεί γειτονικά Pixel μόνο τα πάνω ,κάτω , δεξιά και αριστερά από το κεντρικό σχηματίζοντας ουσιαστικά ένα σταυρό γύρω από το κεντρικό pixel. Η γειτονιά Moore είναι ποιο αποδοτική καθώς καλύπτει περισσότερη περιοχή σαν γειτονιά, για την υλοποίηση της απαιτούνται περισσότερα resources.

Η υλοποίηση του μπορεί να γίνει με δύο τρόπους. Μία είναι με διπλό raster scan στην εικόνα όπου στο πρώτο scan μαρκάρεις κάθε pixel του foreground ανάλογα με ένα counter που κρατάει τις περιοχές και ενημερώνει τα γύρω pixel τους για το ποιος είναι ο πατέρας τους. Οπότε σε κάθε pixel αναθέτουμε 2 τιμές. Έτσι στο επόμενο raster scan όταν υπάρχει conflict των 2 τιμών στο label του pixel αναθέτουμε την μικρότερη τιμή από τις 2.

Η άλλη υλοποίηση του αλγορίθμου (αφορά την δική μου υλοποίηση) είναι με την χρήση αναδρομής και έγινε με τις παρακάτω συναρτήσεις:

```
struct image *ConnectedComponentsMenu(struct image *the_image,struct image *out_image);
void Moore_CC(struct image *the_image,struct image *out_image);
void VanNeumann_CC (struct image *the_image,struct image* out_image );
void VN_pixelFlood(int i , int j , struct image *the_image, struct image *out_image , int newNeighbor);
void Moore_pixelFlood(int i , int j , struct image *the_image, struct image *out_image , int newNeighbor);
```

Ας πούμε 2 πράγματα για αυτή την υλοποίηση και πως δουλεύει. Για την υλοποίηση χρησιμοποιήσαμε την μέθοδο την πλημμύρας (**Flood Fill** [1]) . Επιλέξαμε αυτή την τεχνική προκειμένου να έχουμε καλύτερη κατανομή μνήμης, δηλαδή δεν χρειάστηκε να δημιουργήσουμε ένα struct για κάθε Pixel προκειμένου να κρατάει τις τιμές label και parent . Έτσι αρχικά έχουμε δημιουργήσει 2 image struct the_image και out_image(initialized all values to 0) τα οποία αποτελούν την διαβασμένη εικόνα και την ως προ εγγραφή επεξεργασμένη εικόνα από τις οποίες παίρνουμε πληροφορίες συνεχώς .

Με το που καλέσουμε την συνάρτηση **Moore_CC** ή **VanNeumann_CC** περνώντας τους τις εικόνες (i/o image structs), ξεκινάμε να σκανάρουμε την εικόνα. Όταν βρούμε Pixel που να μην ανήκει στο background , δηλαδή να έχει τιμή 1 , η συνάρτηση καλεί την pixelFlood (VN ή Moore) περνώντας της , τις συντεταγμένες του ευρεθέντος pixel . Ακόμη της περνάει τους pointers των εικόνων για να μπορεί να διαβάσει και να επεξεργαστεί τις εικόνες καθώς και τον αριθμό την συγκεκριμένης γειτονιάς.

Η **Flood Fill** τώρα πραγματοποιεί 2 ελέγχους αν ισχύει κάποια από αυτές κάνει **return** αλλιώς θέτει το τρέχον label στο αντίστοιχο (i , j)pixel της εικόνα εξόδου :

```
if (*(*(the_image->imageMatrix + i )+ j) == 0 ){
    return;} //this condition checks if the neightbor pixel is
            // 0 or 1..We use this to avoid seg_fault on recursion
if  ( *(*(out_image->imageMatrix + i )+ j) != 0 ){
    return;} //if this condition meets
            //means that a node has been visites before..

    *(*(out_image -> imageMatrix + i ) +  j) = newNeighbor;
```

Η παραπάνω συνάρτηση είναι αναδρομική, δηλαδή θα καλέσει τον εαυτό της με διαφορετικά ορίσματα στο i , j.

```
///Recursive Part for neighbot pixels
if (j -1 > 0 ){ // pixel left
    Moore_pixelFlood (i , j-1 , the_image , out_image , newNeighbor);
}
if ( i -1 > 0 ){//pixel up
    Moore_pixelFlood (i -1 , j , the_image , out_image , newNeighbor);
}
if (i +1 <  the_image->height  ){ // pixel down
    Moore_pixelFlood ( i+1 , j , the_image , out_image , newNeighbor);
```

[1] http://en.wikipedia.org/wiki/Flood_fill

Άρα δίνοντας διαφορετικά ορίσματα καταφέρνουμε να καλέσουμε την συνάρτηση για όλα τα γειτονικά Pixel και αυτά με την σειρά τους για τα δικά τους γειτονικά. Και επειδή καλούμε την συνάρτηση για όλα τα γειτονικά Pixel χωρίς να ξέρουμε κάτι για αυτά γι' αυτό το λόγω χρησιμοποιούμε τους αρχικούς ελέγχους.

Στον 1$^{ο}$ έλεγχο αν ο γείτονας ενός pixel με συντεταγμένες π.χ (i -1, j -1) για το οποίο έχει κληθεί η pixelflood ελέγχει αν το (i -1 , j-1)pixel της Input εικόνας είναι μηδενικό. Αν είναι τότε κάνει return αφού το συγκεκριμένο pixel ανήκει στο background και όχι σε γειτονιά.

Στον 2$^{ο}$ έλεγχο, ελέγχει αν το pixel (i-1 , j-1) της output εικόνας είναι διάφορο του μηδέν. Αν ισχύει κάνει πάλι return αφού αν το συγκεκριμένο pixel έχει αλλάξει τιμή σημαίνει ότι κάποιος το επισκέφτηκε ποιο πριν (γείτονας άλλου pixel ) και δεν χρειάζεται η περαιτέρω διερεύνηση του.

Επομένως όταν θα φτάσει εντέλει να τελειώσει η pixelFlood από το deep recursion ανεβαίνει στην συνάρτηση που την κάλεσε. Αυτή μέσω διπλής **for** ψάχνει το επόμενο Pixel που να μην ανήκει στο background , αλλά και το αντίστοιχο (i , j) pixel της output image να είναι μηδέν (να μην έχει αλλάξει από τη Pixelflood λόγο γειτονικότητας) και τότε και μόνο τότε αυξάνει το neighborhood_counter και καλή την Pixelflood με τις συντεταγμένες του (i , j) pixel και τον incremented neighborhood_counter.

```
void Moore_CC (struct image *the_image,struct image* out_image ){
    int i , j , neighborhood = 0;
    for (i = 0 ; i < the_image -> height ; i++ ){
        for ( j = 0 ; j < the_image ->width ; j++){
            if (*(*(the_image->imageMatrix + i ) +j) == 1 || *(*(the_image->imageMatrix + i ) +j) == 255){
                if (*(*(out_image->imageMatrix + i ) +j) == 0 ){
                    neighborhood ++;
                    Moore_pixelFlood ( i , j , the_image , out_image , neighborhood);
```

Ακολουθούν στο τέλος εικόνες που επιβεβαιώνουν την ορθή λειτουργία του αλγόριθμου..

## Δομή προγράμματος

Με το που τρέξουμε το πρόγραμμα (binaryImageProc.exe) το command line μας εμφανίζει ένα μενού το οποίο μας ζητάει να επιλέξουμε εικόνα προτείνοντας μας μερικές που υπάρχουν ήδη στο path (gray scale / binary) ή μπόρει ο χρήστης να βάλει μία δική του, βάζοντας το path που βρίσκεται η συγκεκριμένη εικόνα. ( Όλες οι γραμμές στο cmd που έχουν μπροστά ένα βέλος ζητάνε εισαγωγή από το χρήστη ).

```
********************************************************************************
WELCOME TO  IMAGE PROCESSING  !!!!
THIS PROGRAM FILTERS GRAYSCALE IMAGES AND APPLIES CONNECTED ON BINARY
O_O
********************************************************************************
-> Write name of the file you want to open . . .
-> Some available choices for connencted components are :
1.image.txt (bin)
2.letters.txt(bin)
3.shapes.txt(bin)
4.crosses.txt

-> Or for grayscale images some available choices are :
1.test2.txt (gray)
2.big1.txt
3.mySynth.txt
-> Or give me another name that exist in the directory
```

Έπειτα αφού δώσουμε ένα σωστό όνομα (e.g MySynth.txt) ο file pointer ανοίγει, διαβάζει το αρχείο, τυπώνει στο cmd τις πληροφορίες της εικόνας και μας ρωτάει αν θέλουμε να συνεχίσουμε. Αν όχι το πρόγραμμα τερματίζει

```
Successfully Opening the image...
End of image reached.. All image has been parsed
the scanned has a size of 1901 x 1400
Importing Image . . .

-> Image is ready for processing... Continue [1] or quit [0]
```

Πατώντας ‘1’ μας εμφανίζει το μενού επιλογής μάσκας καθώς και την ακτίνα από το κεντρικό pixel που θέλουμε να έχει. Μετά μας κάνει μια προεπισκόπηση της μάσκας και μας ρωτάει αν θέλουμε να συνεχίσουμε (π.χ κυκλική μάσκα ακτίνας 3).

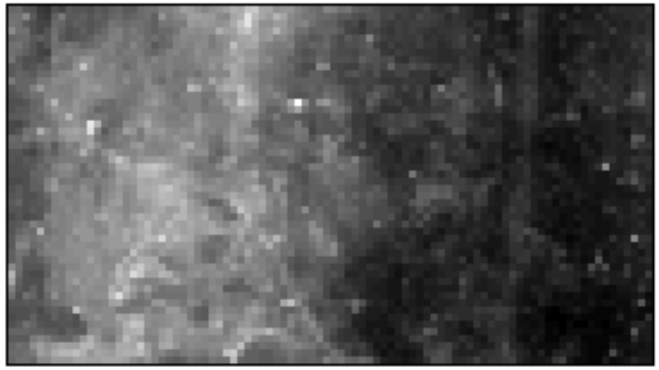
```
-> Choose the type of mask you want to apply:
-> Press (1) squareMask , (2) circleMask , (3) diamondMask
2
You have selected a Circle mask
Give the distance from center
3
Your mask will be looking  like  the following . . .
0    0    0    1    0    0    0
0    1    1    1    1    1    0
0    1    1    1    1    1    0
1    1    1    1    1    1    1
0    1    1    1    1    1    0
0    1    1    1    1    1    0
0    0    0    1    0    0    0
-> Do you want to use another mask [1] or apply the selected  [0] ??
```

Επιλέγοντας την 2η επιλογή καταλήγουμε στο μενού εφαρμογής φίλτρου (π.χ Erosion) και αφού τελειώσει ο αλγόριθμος μας ζητάει να γράψουμε το όνομα του αρχείου που θέλουμε να κάνουμε αποθήκευση. Γράφοντας τώρα το όνομα και πατώντας enter μας εμφανίζει ένα τελικό μενού που μας δίνει κάποιες επιλογές όπως αν θέλουμε περαιτέρω επεξεργασία της ίδιας είτε άλλης εικόνας είτε αν επιθυμούμε τον τερματισμό του προγράμματος.

```
0
Creating  the selected mask . . .
Choose the filtering process you want to apply :
Erosion [1] , Dilation [2] , Open [3] , Close [4]
1
You have chosen Erosion . . .
Working. . .
Done . . .
Write the name of the file you want to write the processed image . . .
out1.txt
out1.txt image has been successfully produced

-> How to you want to continue ??
-> Process another image [1] , Process again the created image [2] , Process again the initial image [3]  or quit [0]
```

## Επεξεργασία εικόνας 50x90 (test2.txt) με χρήση μασκών



**Initial image**

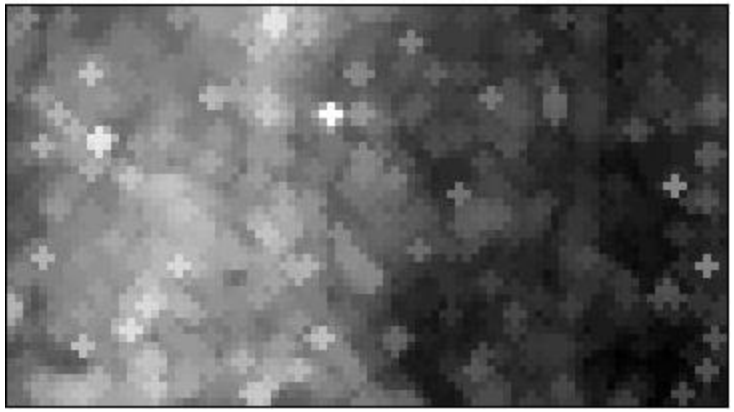


**Image after dilation with 3x3 circle mask**

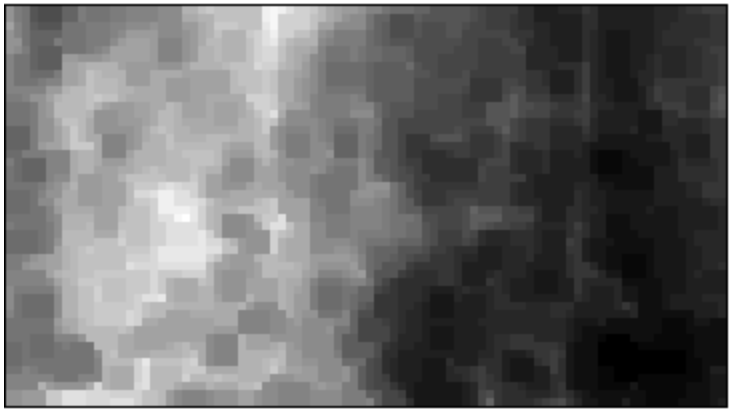


**Image after erosion with 3x3 square mask**

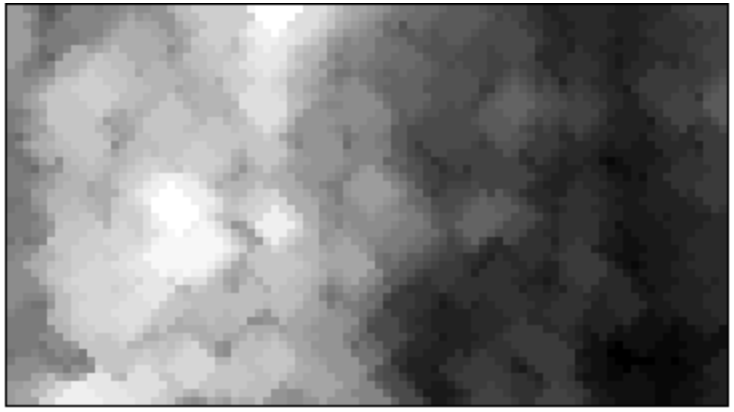
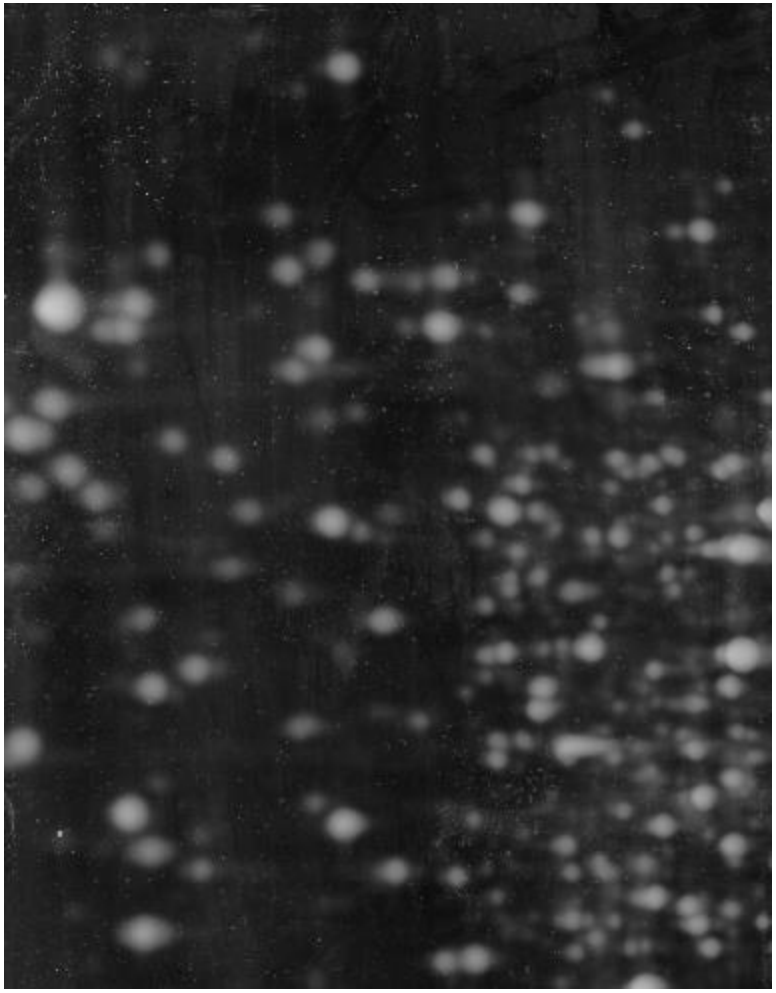
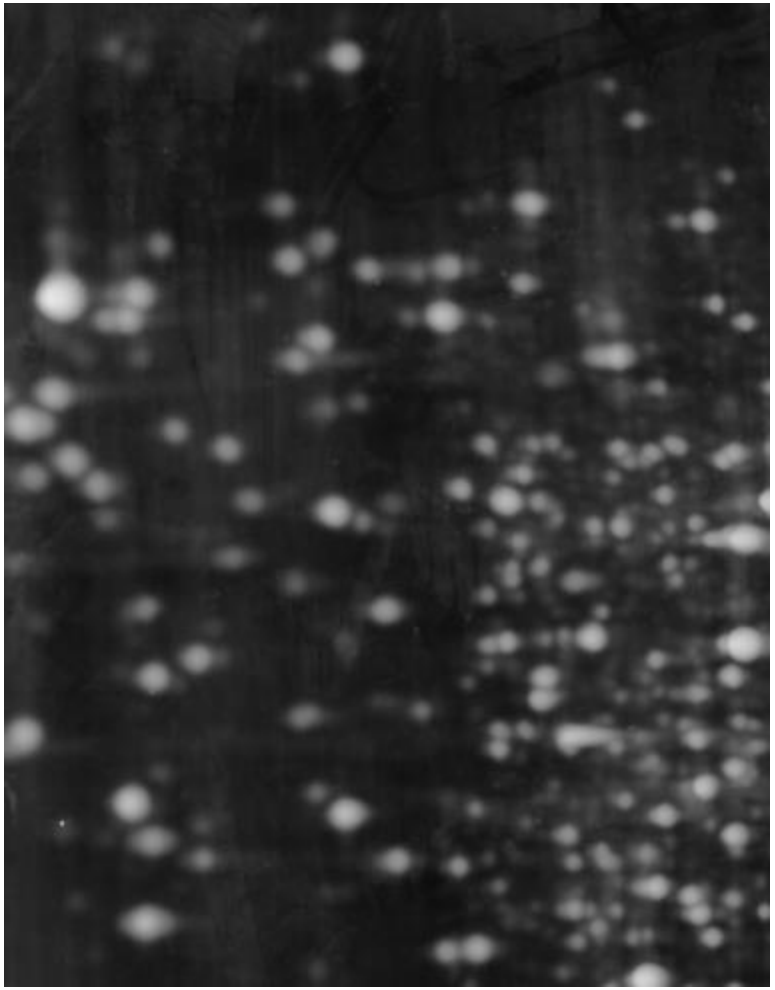


**Image after open with 5x5 diamond mask**

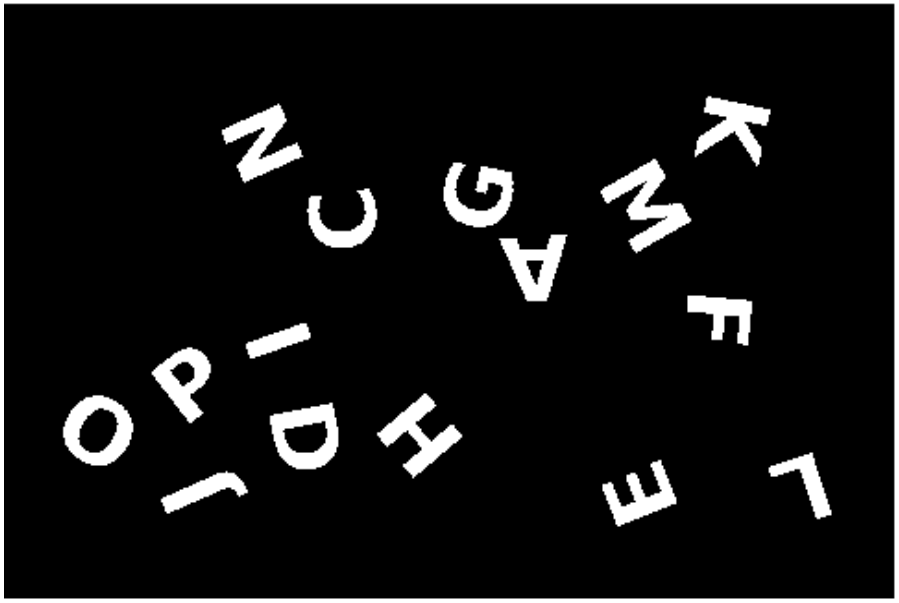


**Same Part of the initial image**



**Part of Big image Image (2501x1822) after erosion with 5x5 circle mask**

## Εύρεση γειτονιών με χρήση CC algorithm



**Initial Image**

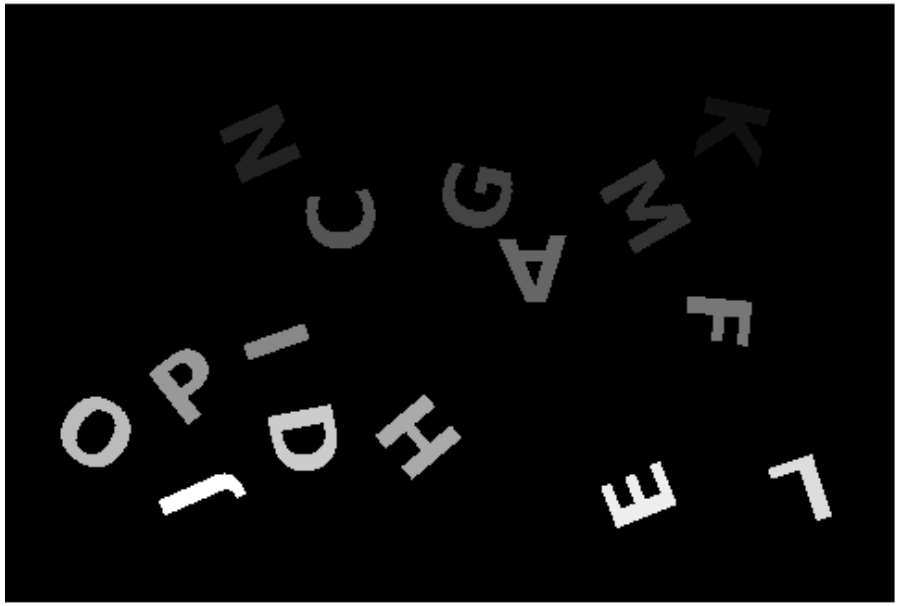
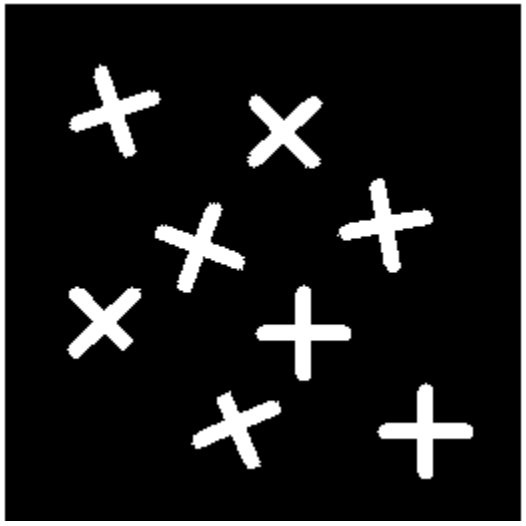


**Image after CC**

**Initial image**

**image after CC**



**Initial Image**

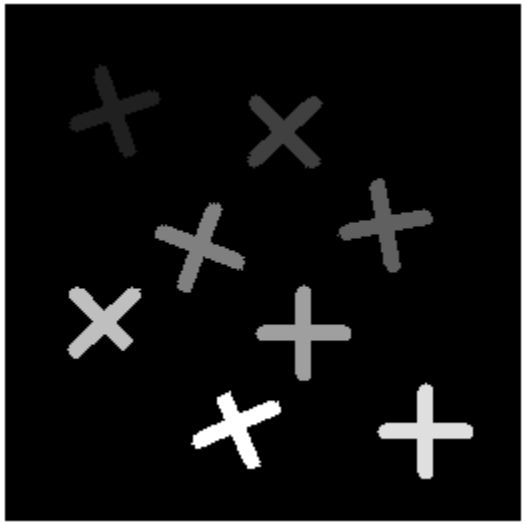


**image after CC**