Algorithms for testing connectivity - 
Implementation on binary images

Tel-Aviv University
School of EE

M.Sc. project

Rotem Aviv
Supervisor: Prof. Dana Ron

May 2011
# Table of contents

1. Introduction .................................................................................................................................................. 4
2. Definitions and abbreviations .................................................................................................................... 6
3. About the project ........................................................................................................................................ 7
   3.1. Description ......................................................................................................................................... 7
   3.2. Project’s goal ..................................................................................................................................... 8
4. Algorithms for Connectivity Testing ........................................................................................................ 9
   4.1. Connectivity Testing algorithm for dense images ........................................................................ 9
   4.2. Connectivity Testing algorithm for sparse images ................................................................. 10
5. Creation of a binary image from real data ............................................................................................... 14
   5.1. Creation of dense binary images ............................................................................................... 14
   5.2. Creation of sparse binary images ............................................................................................ 15
6. An upper bound on Distance to Connectivity ....................................................................................... 17
   6.1. Motivation ....................................................................................................................................... 17
   6.2. Obtaining an upper bound on the DtC of an image .................................................................. 17
   6.3. Searching for a tighter bound on DtC ....................................................................................... 18
7. Implementation of Connectivity Testing Algorithms ............................................................................. 20
   7.1. Software Architecture .............................................................................................................. 20
   7.2. Dense and Sparse Connectivity Testing ................................................................................ 21
   7.3. Computing upper bound on Distance to Connectivity .......................................................... 22
8. Implementation results ............................................................................................................................. 23
   8.1. Implementation Preliminaries .................................................................................................. 23
   8.2. Results of Connectivity Testing implementation on Dense Images .................................. 24
   8.3. Analysis of algorithm implementation results on Dense images ....................................... 38
   8.4. Results of Connectivity Testing implementation on Sparse images ................................ 39
   8.5. Analysis of algorithm implementation results on sparse images ..................................... 54
9. Conclusions and future work .................................................................................................................. 55
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.</td>
<td>References</td>
<td>57</td>
</tr>
<tr>
<td>11.</td>
<td>Source Code</td>
<td>58</td>
</tr>
<tr>
<td>11.1.</td>
<td>Dense image creation and Connectivity Testing</td>
<td>58</td>
</tr>
<tr>
<td>11.2.</td>
<td>Sparse image creation and Connectivity Testing</td>
<td>67</td>
</tr>
<tr>
<td>11.3.</td>
<td>BFS function (C wrapped by MEX)</td>
<td>74</td>
</tr>
<tr>
<td>11.4.</td>
<td>DtC upper bound compute function</td>
<td>83</td>
</tr>
<tr>
<td>11.5.</td>
<td>Connectivity Elements graph function (C wrapped by MEX)</td>
<td>87</td>
</tr>
<tr>
<td>11.6.</td>
<td>Connectivity Elements distance function (C wrapped by MEX)</td>
<td>91</td>
</tr>
</tbody>
</table>
1. Introduction

Given an image defined by a 0/1-valued $n \times n$ matrix $M$, one may wish to know whether the image has a certain property. The most intuitive approach is to read all the pixels (bits) in the matrix, and run an appropriate algorithm on this data, thus obtaining the exact answer in at least linear time. Another approach is to run a much more efficient algorithm, while sacrificing some measure of precision. Several Property Testing algorithms were developed by Raskhodnikova [1] and by Ron and Tsur [2]. These algorithms can distinguish with high success probability between a matrix that has the specified property, and a matrix that does not or that is relatively far from having the property. One of the properties tested by Raskhodnikova and by Ron and Tsur’s algorithms is connectivity, which will be in the focus of this project.

Raskhodnikova [1] introduced a Property Testing model that defines distance between matrices by the Hamming weight of their symmetric differences, normalized by the matrices size. It also uses the straightforward form of accessing the matrix by probing its entries. This model is more suited for dense images, that is, images in which the number of 1-pixels is $\Omega(n^2)$. Ron and Tsur model, on the other hand, is more suitable for relatively sparse images, in which the number of 1-pixels is $O(n)$. In this model, the distance is normalized with respect to the Hamming weight of the matrix, denoted by $w(M)$, and also provides the algorithm with access to uniformly selected 1-pixels (in addition to query access to entries of its choice).

In this project we have implemented the operation of Ron and Tsur's algorithms on real binary images, and we will show how the theoretical complexity and precision measures are manifested on real data. The implementation also includes modules for creation of binary images from color or gray level images, generation of distance to connectivity bound and other functionalities required for the evaluation of the
algorithms. In most tested cases, Ron and Tsur’s algorithms performance on real images is better than the theoretical measure claimed in their article.
2. Definitions and abbreviations

For a given binary image $M$ with size of $n \times n$, we denote by $G(M)$ the underlying graph induced by the neighborhood relation between 1-pixels in $M$. We say that the image $M$ is connected if $G(M)$ is connected.

We also define the term *Distance to Connectivity* for a given image $M$, $DtC(M)$, as the minimal number of pixels in $M$ that must be modified so as to make $M$ connected. Since we do not know the exact value of $DtC(M)$, we define an upper bound on it, $DtC_{UB}(M)$, so that $DtC(M) \leq DtC_{UB}(M)$. An additional term in this context is $\epsilon$, which is a parameter provided to the Connectivity Testing algorithms, as will be described later. An image $M$ is $\epsilon$-far from being connected if more than $\epsilon \cdot w(M)$ pixels must be modified so as to make $M$ connected.

A one-sided error algorithm is defined as an algorithm which accepts an image with probability 1 if the image is connected (in the case of Connectivity Testing algorithm). Otherwise, it will be referred to as a two-sided error algorithm.
3. About the project

3.1. Description

Connectivity Testing algorithms are designed to check whether an input matrix has the connectivity property. As explained before, an image may be close to being connected or far from being connected (DtC closer to 0 or closer to 1, respectively). If one wishes to test the connectivity property of a given image, the first approach may be to read all the pixels of the input image and run the appropriate algorithm on that data, thus obtaining the exact answer in at least linear time. Another approach will be to use a randomized sampling technique on the input data, reducing the complexity and the overall running time of the algorithm to sub-linear. This approach will most likely increase efficiency, but will also reduce result’s accuracy, meaning the algorithm will not always provide the correct answer.

This project will focus on the Connectivity Testing algorithms designed by Ron and Tsur [2]. In their article, Ron and Tsur created Connectivity Testing algorithms with sub-linear complexity for sparse as well as for dense images. As will be explained later, the Connectivity Testing algorithm for sparse images is a two-sided error algorithm, while the algorithm for dense images is defined as a one-sided error algorithm. This means the result returned by the algorithm depends on the sampling pattern of the input image. The theoretical expressions describing the algorithms complexity and measure of accuracy are based on "worst-case scenario", and can not take into account practical cases. This leads us to the question: How well does Ron and Tsur's Connectivity Testing algorithms perform on real-images?
3.2. Project's goal

In this project we will implement Ron and Tsur [2] Connectivity Testing algorithms. The main purpose of the implementation is to evaluate the performance of these algorithms on real images, and to understand how close it is to the theoretical complexity and measure of accuracy. The implementation of Connectivity Testing algorithms on real images can shed light on aspects such as redundancy, efficiency and practicality.
4. Algorithms for Connectivity Testing

In general, Ron and Tsur [2] divide images into two categories: dense images and sparse images. If the Hamming weight of a given image $M$ with size $n \times n$ is less than or equal to $n^{4/3}$, it is considered sparse. Otherwise, the image is considered dense.

4.1. Connectivity Testing algorithm for dense images

Ron and Tsur designed two versions of the algorithm for testing connectivity in dense images. The first version is rather simple, and the second one includes an iterated approach based on a coarse to fine principle, therefore offers reduced complexity.

Let us start by describing the first version of the algorithm.

Input arguments to the algorithm are:

- Input image $M$ with size $n \times n$
- An estimation of $w(M)$, $\hat{w}$, that satisfies $w(M)/c < \hat{w} \leq w(M) \cdot c$ ($c$ is constant and known to the algorithm)
- $\varepsilon$ of the image $M$

To implement the algorithm, perform:

I. Take a sample of $\Theta(1/\varepsilon)$ 1-pixels.

II. From each sampled 1-pixel perform BFS in $G(M)$ until $16cn^2/(\varepsilon^2\hat{w})$ vertices in $G(M)$ are reached or the BFS gets "stuck" before reaching this number of vertices (a small connected component is found).

III. If a small connected component is found, reject, otherwise, accept.
This is a one-sided error algorithm for connectivity, with sampling complexity of \( O(1/\varepsilon) \) and query complexity and run time of \( O((n^2/w(M)) \cdot \varepsilon^{-3}) \). The algorithm total complexity is sub-linear.

Denote this as Dense I algorithm.

The advanced version of the Connectivity Testing algorithm for dense images is defined as follows:

Divide data processing into \( \log(1/\varepsilon) \) iterations. For each iteration \( i \):

I. Take a sample of \( \Theta((\log(1/\varepsilon))/(2^i \varepsilon)) \) 1-pixels.

II. From each sampled 1-pixel, perform BFS in \( G(M) \) until \( (32cn^22^i)/(\varepsilon w) \) vertices are reached, or the BFS gets "stuck" before reaching this number of vertices (a small connected component is found).

III. If a small connected component is found, reject, otherwise, accept and continue to the next iteration.

Denote this as Dense II algorithm.

4.2. Connectivity Testing algorithm for sparse images

Ron and Tsur also designed an algorithm for testing connectivity in sparse images.

The high level idea of the algorithm is as follows: the algorithm tries to find evidence that the tested matrix \( M \) is not connected, where the evidence comes in one of the following forms. "Hard" evidence, in the form of a small connected component in \( G(M) \); "Soft" ("statistical") evidence in the form of more than one connected component when viewing the matrix at a "coarser" resolution. Namely, if we partition the matrix into (equal-size) sub-matrices, and take a sample of 1-pixels,
then we can define a graph over those sub-matrices that contain at least one sample 1-pixel similarly to the way it was defined for single 1-pixels (i.e., \( G(M) \)).

The algorithm checks whether this 'backbone' graph is connected. Evidence against connectivity of this type is "soft", or "statistical" since it is possible that the matrix is connected but the sample missed some sub-matrix, causing the backbone graph to be disconnected. The fact that the algorithm decides based on the second type of evidence and not only on the first, makes it a two-sided error algorithm. Evidence of the "hard" form is obtained by performing several BFS cycles on \( G(M) \) (note that the neighbors of a vertex in \( G(M) \) that corresponds to an entry \((i; j)\) in \( M \) can be obtained by performing 8 queries to \( M \)).

Input arguments to the algorithm are:

- Input image \( M \) with size \( n \times n \)
- An estimation of \( w(M) \), \( \hat{w} \), that satisfies \( w(M)/c < \hat{w} \leq w(M) \cdot c \) (\( c \) is constant and known to the algorithm)
- \( \varepsilon \) of the image \( M \)

To implement the algorithm, perform:

I. Consider a fixed partition of \( M \) into equal-size sub-matrices of dimensions \( s \times s \) where \( s = \sqrt{\hat{w} / c} \) (recall that \( w(M)/c < \hat{w} \leq c \cdot w(M) \) and that the constant \( c \) is known to the algorithm).

II. Take a sample \( S_1 \) of \( t_1 = \Theta(\sqrt{\hat{w}} \cdot \log(\hat{w})) \) uniformly distributed 1-pixels in \( M \) and consider all non-empty sub-matrices in the above-mentioned partition (that is, all sub-matrices that contain a sample 1-pixel). Let \( B(S_1) \) be the ("backbone") graph whose vertices are the non-empty sub-matrices, and where there is an edge between two sub-matrices if they are adjacent (horizontally, vertically, or diagonally). If \( B(S_1) \) is not connected, then reject (otherwise, continue).
III. Select, uniformly at random, $t_2 = \Theta(\log(\hat{w})/\varepsilon)$ non-empty sub-matrices (vertices in $B(S_1)$). For each sub-matrix selected, consider the first sample 1-pixel that fell into the sub-matrix, and perform BFS in $G(M)$ starting from the vertex that correspond to this 1-pixel. Stop once the BFS reaches at least $8\sqrt{c \cdot \bar{w}}/\varepsilon$ vertices in $G(M)$ or the BFS gets "stuck" (a small connected component in $G(M)$ is detected). In the latter case reject (otherwise, continue).

IV. Take an additional sample, $S_3$, of $t_3 = \Theta(1/\varepsilon)$ 1-pixels. If any selected 1-pixel belongs to a sub-matrix that does not neighbor a sub-matrix in the backbone (a vertex of $B(S_1)$), then reject. Otherwise, perform a BFS starting from each sample 1-pixel in $S_3$ as described in the previous step. If a small connected component is found then reject.

V. If no step caused rejection, then accept.

Denote this as the Sparse algorithm.
For an illustration of a (successful) execution of the algorithm, refer to the following figure (from Ron and Tsur’s paper, “Testing properties of sparse images”):

The partition into sub-matrices is marked by a grid of dashed lines. The sampled 1-pixels (in either Step 3 or in Step 4) are marked by dark filled pixels, and the queried entries that are answered by 1 in the course of the BFS’s are marked by lighter filled pixels. The backbone is outlined by a bold line. Note that the marked pixels outside the backbone correspond to a BFS performed in Step 4.
5. Creation of a binary image from real data

Approaching the project at hand, the first task we encountered was to create test images that can be used as input to the implemented algorithms. Synthetic images, like those shown below, are useful for software verification, but our main interest was to understand how the Connectivity Testing algorithms perform on real data.

Images are usually captured and stored in a non-binary format. In almost every case, a binary image is the output of some image processing function or device. Therefore, we suggest a simple method for creating binary images from non-binary source.

5.1. Creation of dense binary images

Our method for creation of dense binary images is based on the following steps.

For a given image \( M \), perform:

I. If the number of data channels of \( M \) is larger than 1, convert to a single channel image. For a color image with 3 channels (Red, Green, and Blue, RGB), we used MATLAB’s RGB to gray conversion:

\[
\text{grayscale} = 0.2989 \times R + 0.5870 \times G + 0.1140 \times B
\]

II. Convolve the output of step I with a Gaussian kernel (size and sigma can vary according to image structure).

III. Apply a threshold on the convolved image, so that pixels above (or below) threshold are marked as ‘1’ and those below it are marked as ‘0’. Output is a binary image.
IV. Run a median filter on the binary image to eliminate noise.

Examples for binary images created using this method are shown the following in figure.

5.2. Creation of sparse binary images

Our method for creation of sparse binary images is based on the following steps.

For a given image M, perform:

I. If the number of data channels of M is larger than 1, convert to a single channel image. For a color image with 3 channels (Red, Green, and Blue, RGB), we used MATLAB’s RGB to gray conversion:

\[
\text{grayscale} = 0.2989 \times R + 0.5870 \times G + 0.1140 \times B
\]

II. Run a median filter on the single-channel image to eliminate noise and preserve edges

III. Apply a Sobel filter on image to find edges
IV. Apply threshold on the filtered image, so that pixels above threshold are marked as ‘1’ and those below it are marked as ‘0’. Output is a binary image.

V. Apply the following morphological filter:

In a block of size 5x5 around each 1-pixel, count the number of 1-pixels. If the number is less than 12, modify to 0-pixel.

VI. Apply the following Edge Thinning morphological filter:

Around each 1-pixel, check whether there exist four adjacent vertical and horizontal neighbors of this pixel. If yes, modify to a 0-pixel.

Examples for binary images created using this method are shown in the following figure.
6. An upper bound on Distance to Connectivity

6.1. Motivation

As described in the previous chapter, the Connectivity Testing algorithms operation is based on the parameter \( \epsilon \), provided as an input to the algorithms. The value of \( \epsilon \) is used in several calculations such as in determining how many vertices to sample from \( G(M) \), or how many vertices of \( G(M) \) should the BFS cover.

As explained in chapter 2, we distinguish between three different arguments: \( DTC(M) \), \( DTC_{UB}(M) \), and \( \epsilon \). Going back to the definition of \( DTC(M) \), we defined it as the minimal number of 1-pixels in the image \( M \) that needs to be modified so as to make \( G(M) \) connected. Since \( DTC(M) \) is not known, we will calculate an upper bound on its value, \( DTC_{UB}(M) \). We know that \( DTC_{UB}(M) \geq DTC(M) \), therefore we can use the value of \( DTC_{UB}(M) \) for \( \epsilon \), and provide it as an input for the Connectivity Testing algorithms. Providing the algorithm with a value of \( \epsilon \) that is larger than the true \( DTC(M) \) of the image \( M \) makes the algorithm operation more difficult in terms of accuracy.

The computation of \( DTC_{UB}(M) \) is not performed in sub-linear time, and it is done in order to provide the algorithms with the input data required for their operation.

6.2. Obtaining an upper bound on the DTC of an image

The following section describes how an upper bound on the DTC of a given image \( M \) can be calculated. The main principle in the basis of this method is to connect all the connectivity elements in a given image using straight lines, so that the total length of lines added to the image is minimal.

Given an image \( M \) with hamming weight \( w(M) \), generate the 1-pixels graph \( G(M) \) and follow the steps described hereafter:
I. By performing BFS on the entire graph $G(M)$, find all connectivity elements. Denote $NE$ to be the number of connectivity elements found.

II. If $NE < 2$, a single connectivity element was found, meaning the image is connected (another option is that there are no connectivity elements at all). Return $DtC = 0$. If $NE \geq 2$, continue.

III. Measure the distance between every two connectivity elements. Build a new weighted graph, $G_{MST}(M)$, so that every connectivity element is a node and there is an edge between every two nodes. The weight of an edge in $G_{MST}(M)$ graph is the distance measured between the two nodes in each side of the edge.

IV. Run a Minimum Spanning Tree (MST) algorithm on $G_{MST}(M)$. Denote by $SUM_{MST}$ the minimum weight of the spanning tree. Return $DtC_{UB}(M) = SUM_{MST}/w(M)$.

6.3. Searching for a tighter bound on $DtC$

In general, $DtC(M)$ is based on modification of pixels either from ‘1’ to ‘0’ or from ‘0’ to ‘1’. The method described previously for finding an upper bound on $DtC$ joins connectivity elements by adding straight lines connecting between the elements in the image. In other words, this method uses only addition of new 1-pixels into the image.

We would like to expand the method described previously, by suggesting a way to generate a tighter bound on $DtC$. In some cases, the cost of removing a connectivity element is lower than the cost of connecting it to other elements. Therefore, a tighter $DtC$ measure can be found by removing one or more connectivity elements from the initial $G_{MST}(M)$ graph, and running a Minimum Spanning Tree (MST)
algorithm on the remaining sub-graph. The total cost for each sub-graph will be the cost of the minimum weight of the spanning tree added to the cost of removing the specific connectivity elements. The final value of $\text{DtC}_{UB}(M)$ will be the minimum cost found among all sub-graphs.
7. Implementation of Connectivity Testing Algorithms

7.1. Software Architecture

Implementing the Connectivity Testing algorithms discussed in this project requires the creation of a complete development environment. The environment is responsible to the entire work flow, from creation of a binary image, through running the algorithms, and to gathering results and performing analysis.

The development environment is based on MATLAB and MATLAB-Executable (MEX-files). The main reason for using MATLAB was to shorten development time, as MATLAB supports many required features, such as image loading, saving, spatial filters, minimum spanning tree module, and more. Performing BFS and other functionalities that required serial processing of data were exported to C code, wrapped by MEX-file. MATLAB's MEX-file enables fast and simple interface between MATLAB code and C code, allowing the user to select the more suitable tool for implementation for each part of the application.

The software implementation is divided into several blocks according to the functionality and method of implementation. The following sections describe the different blocks and provide information about processing flow.
7.2. Dense and Sparse Connectivity Testing

The following figure presents the processing flow and the software blocks that were developed to enable implementation of dense and sparse Connectivity Testing.
As can be seen, most of the implementation is MATLAB-based. Since BFS is composed of serial operations, it is implemented in C/Mex code, which enables faster running-time for functionalities of this kind.

7.3. Computing upper bound on Distance to Connectivity

The following figure presents the processing flow and the software blocks that were developed to enable calculation of DtCUB of a binary image.

Similarly to Connectivity Testing implementation, DtCUB(M) calculation was also mostly MATLAB-based. The block of computing connectivity elements distance was implemented in C/Mex code.
8. Implementation results

8.1. Implementation Preliminaries

The implementation environment developed in this project is based on using specific values for the algorithm’s parameters, instead of the asymptotic bounds described in the original definition. For example, the sparse algorithm requires sampling of \( \Theta(\sqrt{\hat{w}} \cdot \log(\hat{w})) \) 1-pixels to create the Back-Bone graph, whereas the implementation samples the rounded value of \( \sqrt{w} \cdot \log(w) \) 1-pixels. Additionally, we do not use an estimation of the image’s Hamming weight, \( \hat{w}(M) \). Instead, we compute the exact \( w(M) \) value of the image. These adjustments are required by the implementation, and should not have a negative effect on the performance of any of the algorithms.

The module for \( \text{DtC}_{UB}(M) \) computation is based on the method described in Section 6. The module includes calculation of the \( \text{DtC}_{UB}(M) \) value based on Minimum Spanning Tree on the following Connectivity Elements graph and sub-graphs, and selecting the minimal value:

- Entire graph of Connectivity Elements
- Residual Connectivity Elements sub-graph after removing one element
- Residual Connectivity Elements sub-graph after removing a growing number of elements sorted by their size
8.2. Results of Connectivity Testing implementation on Dense Images

In the following section, we describe the results obtained from the implementation of Ron and Tsur’s Connectivity Testing algorithms on dense images. The implementation is based on the iterated version of the Dense algorithm, referred to as Dense II, and with various dense binary images. Additionally, we will present the implementation results of the Sparse Connectivity Testing algorithm on Dense images.

Due to the statistical nature of the algorithms, the results shown in this chapter are averaged on hundreds of repeated tests.
We start with a detailed description of the implementation results from the following dense binary image:

![Image](image.png)

The following table presents the image properties, along with several parameters computed during implementation:

<table>
<thead>
<tr>
<th>Image Properties and implementation Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Size</td>
</tr>
<tr>
<td>Hamming Weight</td>
</tr>
<tr>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>Number of Connectivity Elements</td>
</tr>
<tr>
<td>Largest Connectivity Elements size</td>
</tr>
<tr>
<td>Number of Sampled 1-pixels in 1st iteration</td>
</tr>
<tr>
<td>BFS limit</td>
</tr>
<tr>
<td>Maximum number of iterations</td>
</tr>
</tbody>
</table>

The results of Dense II algorithm implementation show the algorithm discovers the image is not connected at 100% of tests, while processing 16,865 1-pixels in average.
In the worst-case scenario, the maximum number of pixels the algorithm can process during its operation is the size of the largest connectivity element added with the number of sampled 1-pixels in the 1st iteration (no further iterations are required in our case). We notice that under worst-case scenario, the algorithm would process 28,985 1-pixels. The theoretic complexity of the algorithm is actually higher than the worst-case complexity we present.

We will now show the influence of reducing the number of sampled 1-pixels on the accuracy and complexity of the Dense algorithm.

The following graph describes the average complexity as function of a scalar divider, NodeSampleNorm, which determines the number of sampled 1-pixels according to the following expression:

\[ \text{Number of sampled 1-pixels} = \log(1/\varepsilon)/(2^i \cdot \varepsilon) / \text{NodeSampleNorm}, \text{ where } 1 < i < \log(1/\varepsilon) \]

As can be seen, there is no consistency in the effect of the number of sampled 1-pixels on the complexity of the algorithm. Additionally, our results show that the algorithm’s accuracy was not affected by varying the number of sampled 1-pixels.
The next two graphs describe the average complexity and accuracy as function of a scalar divider, BFS Limit Norm, which determines the limit value for BFS operation according to the following expression:

\[
\frac{(32cn^2 + 2)}{(\epsilon \omega \cdot \text{BFS Limit Norm})}, \text{ where } 1 < i < \log(1/\epsilon)
\]

Clearly, reducing the BFS limit value under a certain threshold lowers the algorithm's accuracy. This can be explained by looking at a case where a sampled 1-
pixel belongs to a Connectivity Element which is larger than the BFS limit. The algorithm, in this case, will not reject.

The graph of Complexity as function of BFSLimitNorm shows no consistency in the effect of the BFS limit value on the algorithm complexity.

As a final step in the implementation, we would like to test the performance of the Sparse Connectivity testing algorithm.

The results of the Sparse algorithm implementation show the algorithm discovered the image is not connected at 100% of the tests, while processing 13,294 1-pixels in average. In the worst-case scenario, the maximum number of pixels the Sparse algorithm can process during its operation is the size of the largest connectivity element added with the number of sampled 1-pixels in the back-bone stage. We notice that under worst-case scenario, the algorithm would process 31,952 1-pixels. The theoretic complexity of the Sparse algorithm is actually higher than the worst-case complexity we present.
In this next part, we present the implementation results obtained from additional dense binary images. The results and graphs in this part are presented in the same order as shown for the previous test image.

![Image](image.png)

**Image Properties and implementation Information**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Size</td>
<td>450 x 318</td>
</tr>
<tr>
<td>Hamming Weight</td>
<td>20,748</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.006</td>
</tr>
<tr>
<td>Number of Connectivity Elements</td>
<td>11</td>
</tr>
<tr>
<td>Largest Connectivity Elements size</td>
<td>14,473</td>
</tr>
<tr>
<td>Number of Sampled 1-pixels in 1st iteration</td>
<td>186</td>
</tr>
<tr>
<td>BFS limit</td>
<td>20,749</td>
</tr>
<tr>
<td>Maximum number of iterations</td>
<td>3</td>
</tr>
</tbody>
</table>
Complexity as function of the number of sampled 1-pixels:

Results accuracy rate is 100%.

Complexity as function of the BFS Limit value:
Accuracy as function of the BFS Limit value:

Sparse algorithm implementation results show 100% accuracy, average sampling of 13,383 1-pixels, compared to 15,814 1-pixels under worst-case scenario.
<table>
<thead>
<tr>
<th>Image Properties and implementation Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Size</td>
<td>911 x 683</td>
</tr>
<tr>
<td>Hamming Weight</td>
<td>59,039</td>
</tr>
<tr>
<td>ε</td>
<td>0.003</td>
</tr>
<tr>
<td>Number of Connectivity Elements</td>
<td>5</td>
</tr>
<tr>
<td>Largest Connectivity Elements size</td>
<td>36,645</td>
</tr>
<tr>
<td>Number of Sampled 1-pixels in 1st iteration</td>
<td>421</td>
</tr>
<tr>
<td>BFS limit</td>
<td>59,039</td>
</tr>
<tr>
<td>Maximum number of iterations</td>
<td>3</td>
</tr>
</tbody>
</table>
Complexity as function of the number of sampled 1-pixels:

Results accuracy rate is 100%.

Complexity as function of the BFS Limit value:
Accuracy as function of the BFS Limit value:

Sparse algorithm implementation results show 100% accuracy, average sampling of 21,986 1-pixels, compared to 39,045 1-pixels under worst-case scenario.
<table>
<thead>
<tr>
<th>Image Properties and implementation Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Image Size</strong></td>
</tr>
<tr>
<td><strong>Hamming Weight</strong></td>
</tr>
<tr>
<td><strong>$\varepsilon$</strong></td>
</tr>
<tr>
<td><strong>Number of Connectivity Elements</strong></td>
</tr>
<tr>
<td><strong>Largest Connectivity Elements size</strong></td>
</tr>
<tr>
<td><strong>Number of Sampled 1-pixels in 1\text{st} iteration</strong></td>
</tr>
<tr>
<td><strong>BFS limit</strong></td>
</tr>
</tbody>
</table>
Complexity as function of the number of sampled 1-pixels:

Results accuracy rate is 100%.

Complexity as function of the BFS Limit value:
Accuracy as function of the BFS Limit value:

Sparse algorithm implementation results show 100% accuracy, average sampling of 3,130 1-pixels, compared to 4,616 1-pixels under worst-case scenario.
8.3. Analysis of algorithm implementation results on Dense images

The implementation results of the dense images lead to several observations. The first and most clear observation is that the performance of the Dense and Sparse Connectivity Testing algorithms on dense images is better than the theoretic performance.

When modifying the algorithm parameters, we see that changing the number of sampled 1-pixels does not influence the result's accuracy or complexity. This can be explained by the fact that for images with few Connectivity Elements, such as the images we use in this project, the algorithm requires sampling of only few 1-pixels in order to get to the result. We also notice that reducing the BFS Limit value decreases the result accuracy and may also cause an increase in complexity. When analyzing the BFS Limit value, we can see that if it is smaller then the size of one or more Connectivity Elements, the dense algorithm will not be able to identify these Elements as separate Connectivity Elements. Therefore, the algorithm will process more pixels until rejecting, and in some cases may even provide a wrong answer.

When comparing the results of the dense and the sparse Connectivity Testing algorithms, we see that both algorithms provide highly accurate results with similar complexity. In our implementation, the sparse algorithm is not able to detect separate Connectivity Elements in the Back-Bone stage, thus it is required to perform BFS, similarly to the dense algorithm.
8.4. Results of Connectivity Testing implementation on Sparse images

In the following section, we describe the results obtained from implementation of Ron and Tsur’s Connectivity Testing algorithms on sparse images. The implementation is based on the sparse algorithm presented in Section 4, and with various sparse binary images. Additionally, we present the implementation results of the Dense Connectivity Testing algorithm on sparse images.

Due to the statistical nature of the Sparse Connectivity Testing algorithm, the results shown in this chapter are averaged on hundreds of repeated tests.

We start with a detailed description of the implementation results from the following sparse binary image:
The following table presents the image properties, along with several parameters computed during implementation:

<table>
<thead>
<tr>
<th>Image Properties and implementation Information</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming Weight</td>
<td>16,802</td>
</tr>
<tr>
<td>ε</td>
<td>0.025</td>
</tr>
<tr>
<td>Size of largest Connectivity Element</td>
<td>5,187</td>
</tr>
<tr>
<td>Number of Connectivity Elements</td>
<td>64</td>
</tr>
<tr>
<td>Sub-Matrix size</td>
<td>127 x 127</td>
</tr>
<tr>
<td>Number of sub-matrices</td>
<td>8 x 8</td>
</tr>
<tr>
<td>Number of sampled 1-pixels in Back-Bone stage</td>
<td>1,234</td>
</tr>
<tr>
<td>Number of sampled 1-pixels in BFS stage</td>
<td>388</td>
</tr>
<tr>
<td>BFS limit</td>
<td>16,802</td>
</tr>
</tbody>
</table>

The results of the Sparse algorithm implementation show the algorithm discovers the image is not connected at 100% of tests, while processing 1,432 1-pixels in average. In the worst-case scenario, the maximum number of pixels the Sparse algorithm can process during its operation is the size of the largest connectivity element added with the number of sampled 1-pixels in the back-bone stage. We notice that under worst-case scenario, the algorithm would process 6,421 1-pixels. The theoretic complexity of the Sparse algorithm is actually higher than the worst-case complexity we present.

We will now show the influence of modifying the size of the Back-Bone matrix on the algorithm's accuracy and complexity.
The following graph describes the average complexity as function of a scalar divider, BBSizeNorm, which determines the 1-D size of the Back-Bone matrix according to the following expression:

\[
\text{Back-Bone matrix 1-D size} = \frac{\sqrt{w(M)}}{\text{BBSizeNorm}}
\]

In addition to the average complexity as function of BBSizeNorm, the graph above shows the complexity of the "worst-case" scenario, as it is defined in the previous page. When modifying the size of the Back-Bone matrix, we also adjusted the number of sampled 1-pixels in the Back-Bone stage according to the following expression:

\[
\text{Number of sampled 1-pixels} = (\sqrt{w(M)} \cdot \log(w(M))) \cdot (\text{BBSizeNorm}^2)
\]

Our implementation result shows 100% accuracy measure of the algorithm for all tested values of BBSizeNorm.
The next result we present describes how the algorithm performs when modifying the BFS limit value. The following graph illustrates the average complexity as a function of a scalar divider, BFSLimitNorm, which determines the limit value for BFS operation according to the following expression:

\[ BFS \ Limit = 8 \cdot \sqrt{w(M)} / (\varepsilon \cdot BFSLimitNorm) \]

Our results show 100% accuracy measure of the algorithm for all tested values of BFSLimitNorm.
Another aspect we test is how the algorithm performs when reducing the number of sampled 1-pixels. The reduction is performed only for 1-pixels sampled in the BFS stage, and not in the Back-Bone stage. This is done to avoid partial coverage of the image by the Back-Bone graph.

The following graph describes the average complexity as function of a scalar divider, NodeSampleNorm, which determines the number of sampled 1-pixels according to the following expression:

\[
\text{Number of sampled 1-pixels} = \frac{\log(\hat{w})/\epsilon}{\text{NodeSampleNorm}}
\]

Our results show 100% accuracy measure of the algorithm for all tested values of NodeSampleNorm.

In the last stage of the implementation we present the performance of the Dense algorithm on the sparse binary image. Dense II algorithm implementation shows average accuracy of 100%, with complexity of 1845 1-pixels. In the worst-case scenario, the maximum number of pixels the Dense algorithm can process during its operation is the size of the largest connectivity element added with the number of sampled 1-pixels in the 1st iteration (no further iterations are required in this case).
We notice that under worst-case scenario, the algorithm would process 5,187 1-pixels. The theoretic complexity of the algorithm is actually higher than the worst-case complexity we present.
In this next part, we present the implementation results obtained from additional sparse binary images. The results and graphs are presented in the same order as shown for the previous test image.

<table>
<thead>
<tr>
<th>Image Properties and implementation Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming Weight</td>
</tr>
<tr>
<td>ε</td>
</tr>
<tr>
<td>Size of largest Connectivity Element</td>
</tr>
<tr>
<td>Number of Connectivity Elements</td>
</tr>
<tr>
<td>Sub-Matrix size</td>
</tr>
<tr>
<td>Number of sub-matrices</td>
</tr>
<tr>
<td>Number of sampled 1-pixels in Back-Bone stage</td>
</tr>
<tr>
<td>Number of sampled 1-pixels in BFS stage</td>
</tr>
<tr>
<td>BFS limit</td>
</tr>
</tbody>
</table>

The Sparse algorithm implementation results show 100% accuracy, average complexity of 904 1-pixels in average. In the worst-case scenario, the algorithm’s complexity would have been 2,086 1-pixels.
With implementation results accuracy of 100%:
With implementation results accuracy of 100%:

Dense II algorithm implementation results show 100% accuracy, average sampling of 639 1-pixels, compared to 1,335 1-pixels in the worst-case scenario.
Image Properties and implementation Information

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming Weight</td>
<td>8,132</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0.01</td>
</tr>
<tr>
<td>Size of largest Connectivity Element</td>
<td>4,702</td>
</tr>
<tr>
<td>Number of Connectivity Elements</td>
<td>32</td>
</tr>
<tr>
<td>Sub-Matrix size</td>
<td>90 x 90</td>
</tr>
<tr>
<td>Number of sub-matrices</td>
<td>12 x 12</td>
</tr>
<tr>
<td>Number of sampled 1-pixels in Back-Bone stage</td>
<td>812</td>
</tr>
<tr>
<td>Number of sampled 1-pixels in BFS stage</td>
<td>900</td>
</tr>
<tr>
<td>BFS limit</td>
<td>8,132</td>
</tr>
</tbody>
</table>
The Sparse algorithm implementation results show 100% accuracy, average complexity of 816 1-pixels in average. In the worst-case scenario, the algorithm’s complexity would have been 5,514 1-pixels.

With implementation results accuracy of 100%:
With implementation results accuracy of 100%:

Dense II algorithm implementation results show 100% accuracy, average sampling of 2,248 1-pixels, compared to 4,702 1-pixels in the worst-case scenario.
Image Properties and implementation Information

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming Weight</td>
<td>14,387</td>
</tr>
<tr>
<td>ε</td>
<td>0.002</td>
</tr>
<tr>
<td>Size of largest Connectivity Element</td>
<td>8,270</td>
</tr>
<tr>
<td>Number of Connectivity Elements</td>
<td>25</td>
</tr>
<tr>
<td>Sub-Matrix size</td>
<td>120 x 120</td>
</tr>
<tr>
<td>Number of sub-matrices</td>
<td>9 x 9</td>
</tr>
<tr>
<td>Number of sampled 1-pixels in Back-Bone stage</td>
<td>1,148</td>
</tr>
<tr>
<td>Number of sampled 1-pixels in BFS stage</td>
<td>4,787</td>
</tr>
<tr>
<td>BFS limit</td>
<td>14,387</td>
</tr>
</tbody>
</table>

The Sparse algorithm implementation results show 100% accuracy, average complexity of 1,566 1-pixels in average. In the worst-case scenario, the algorithm’s complexity is 9,418 1-pixels.
With implementation results accuracy of 100%:

![Graph 1](image1)

With implementation results accuracy of 100%:

![Graph 2](image2)
With implementation results accuracy of 100%:

Dense II algorithm implementation results show 100% accuracy, average sampling of 5,500 1-pixels, compared to 8,270 1-pixels in the worst-case scenario.
8.5. Analysis of algorithm implementation results on sparse images

The implementation of the sparse and dense Connectivity Testing algorithms on binary images provides interesting results.

First, we can clearly see that the sparse and dense Connectivity Testing algorithms outperform the theoretic performance for the sparse images given in this implementation. This is similar to the behavior of the algorithms on dense images. The results of the sparse and dense algorithms were 100% accurate for hundreds of tests. The improvement in efficiency varies from one image to another. This differentiation is probably due to the number of Connectivity Elements and their size, which is different from one image to another.

Increasing the size of the Back-Bone matrix improves the complexity of the sparse algorithm, without affecting the accuracy. Our results show that the maximal efficiency is achieved when the size of the Back-Bone matrix is about the size of the image, which means the contribution of the Back-Bone stage to the sparse algorithm result is minimal.

An additional observation from the results is that modifying the limit value of the BFS or the number of sampled 1-pixels (in the BFS stage) does not influence the results accuracy or the complexity. This may be explained by the fact that our images are structured in such a way that the algorithm rejects (meaning it decides the image is not connected) after processing a relatively small part of the image. The presence of small Connectivity Elements allows the algorithm to operate accurately even with a tight BFS limit.

As can be seen, the sparse and dense Connectivity Testing algorithms achieve accurate results with similar complexity when applied to the test images. We notice that the sparse algorithm is not able to reject in the Back-Bone stage, therefore its result is based on BFS processing of the image.
9. Conclusions and future work

Implementation of Connectivity Testing algorithms on real images is a useful method for performance evaluation. The environment developed in this project enables testing of the algorithms on various input images and analyzing their performance. In general, the implementation provides us with results that meet our expectations, which are derived from the theory, but also lead us to several interesting conclusions.

Before discussing our conclusions, we would like to emphasize the fact that this project focuses on the performance of the Connectivity Testing algorithm, without relating it to a specific application. The set of binary images we used as input data for the implementation was generated so that the binary image contains recognizable and meaningful data. We noticed that creating a connected binary image using the method demonstrated in Section 5 is usually not possible. We assume that real data generated by an external application will have similar properties to our binary images.

The main parameters, which the algorithms were evaluated by, were accuracy and complexity. Although the Sparse and Dense algorithms are defined as "two-sided" and "one-sided" error algorithms respectively, the results of both algorithms have exceeded the theoretic measure of accuracy. To evaluate the algorithms performance in terms of complexity, we can compare the implementation results to the measure defined as the “worst-case” scenario of the given image, as shown by several graphs in Section 8. Our results show the algorithms outperform this measure, when the extent of improvement varies from one image to another.

A central part of the implementation focused on the algorithms performance while modifying various parameters, such as the BFS limit value and the number of sampled 1-pixels. The purpose of these modifications was to check whether we can achieve better performance of an algorithm by tuning its parameters. Looking at the
results, we conclude that the BFS operation, as described in Ron and Tsur algorithms, is highly effective and significantly contributes to the sub-linear nature of the algorithms. The BFS operation and the BFS Limit value, as defined in the algorithms, achieve accurate results while processing a relatively small part of the image. This, of course, depends on the image structure, but our results are consistent throughout all the test images.

The results of the sparse algorithm implementation show that the Back-Bone stage is not effective in discovering different Connectivity Elements in the images we used. The two key factors enabling the Back-Bone sub-matrix process to truly identify different Connectivity Elements are the sub-matrix size and the offset location of the sub-matrix grid. We show that increasing the sub-matrix resolution by using smaller sub-matrices increases the overall complexity. Utilizing the second factor and applying the sub-matrix with variable offset location onto the image may contribute to a successful identification of the image as not-connected, and reduce the algorithm complexity.

The work done in this project reveals several matters that may be further developed in the future. First, one may develop a tighter bound on Distance to Connectivity, or may even design an algorithm for finding the exact Distance to Connectivity of an image. It is also possible to design an algorithm that answers whether an image contains up to a specific number of Connectivity Elements. In the aspect of efficient implementation, one may implement methods for image representation, pixel sampling and pixel querying, that will allow efficient memory utilization, along with fast and simple access to 1-pixels of an image.

Our project uses only a few test images and it does not provide an analysis based on a broad base of images. Therefore, the algorithms implemented in this project can be coupled with an application that will utilize its capabilities. This will enable a more profound analysis of the algorithms, and may lead to the creation of improved versions of the algorithms.
10. References


11. Source Code

The following section presents the source code developed during the implementation of the Connectivity Testing algorithms and their environment.

11.1. Dense image creation and Connectivity Testing

clear all
close all

%% Dense_main.m %%

% This application is an implementation of the connectivity testing algorithm for dense images published by Dana Ron and Gilad Tsur
% This program implements the second and more efficient version to the algorithm

clear all
close all

% Add Mex file DLL to path
path(path,'../MatlabDll_BFS/Debug');

ImConnected = 0; % Determines if Image is connected or not
nTests = 100;

Draw = 0; % enable/disable drawing
Verbose = 1; % verbosity level

fileName = 'cars1';
epsilon = 0.012;
showIm = 0;

thresh = 0.85; % threshold for generation of binary image
doGauss = 1; % enable/disable gaussian filtering
doMed = 1; % enable/disable median filtering

reverse = 0; % enable/disable reversing of binary image
filt_size = 21;
medsize = 10;

% Pre-process image
file=['../images/', fileName, '.jpg'];
Im = im2single(imread(char(file)));

if(size(Im,3) > 1)
    Im = Im(:,:,1);
end

if showIm == 1
    subplot(131)
    imshow(Im);
end

if doGauss == 1
    gauss = zeros(filt_size, filt_size);
    x = -(filt_size-1)/2:(filt_size-1)/2;
    y = x';

58
[x, y] = meshgrid(x, y);
gauss(:, :) = exp(-(x.^2 + y.^2)/2*filt_sigma^2);
gauss = gauss/sum(sum(gauss));
Im = imfilter(Im, gauss, 'symmetric');
end
Im(find(Im > thresh)) = 1;
Im(find(Im <= thresh)) = 0;
if reverse == 1
    Im = abs(1-Im);
end
if showIm == 1
    subplot(132)
    imshow(Im);
end
%Perform median filtering
if doMed == 1
    if(median_size > 0)
        Im = medfilt2(Im, [median_size median_size]);
    end
end
if showIm == 1
    subplot(133)
    imshow(Im);
end
Im = round(Im);
%Save Image
FileName=['../images/', name, '_preProc_thresh_', ...
    num2str(thresh)];
imwrite(Im, [FileName, '.jpg', 'jpeg', 'quality', 100);
%Test if image is dense
if(length(find(Im == 1)) > sqrt(size(Im,1)*size(Im,2)^(4/3))
    if(Verbose == 1)
        fprintf(['Image is dense, total of ',num2str(length(find...
            (Im == 1))'),' 1-pixels
']);
    else
        fprintf('Image is dense
');
    end
else
    fprintf('Image is sparse. Exiting
');
    return;
end
if showIm == 1
    imshow(Im);
    return;
end
% Testing connectivity property
%Configuration params
BFSStopNorm = [1 2 5 10:10:100 500 1000:1000:10000];
%BFSStopNorm = [1];
nBFSStopNorm = length(BFSStopNorm);
%NodeSampleNorm = [0.01 0.05 0.1 0.5 1 2 5 10:10:100];
NodeSampleNorm = [1];
nNodeSampleNorm = length(NodeSampleNorm);
CntOuter = 1;
CntInner = 1;
save('tmpFile', 'NodeSampleNorm', 'nNodeSampleNorm', 'Draw',
'Verbose');
Result = zeros(8,30);
TestResults = zeros(8,30, nNodeSampleNorm);
for K = 1:nBFSStopNorm
  save('tmpFile', 'NodeSampleNorm', 'TestResults', 'nNodeSampleNorm',
  'ImConnected', 'nTests', 'Draw', 'Verbose', 'K', 'BFSStopNorm',
  'nBFSStopNorm','CntOuter', 'CntInner', 'fileName', 'epsilon');
  Result = Dense_func(NodeSampleNorm(1), BFSStopNorm(K), ImConnected,...
  nTests, Draw, Verbose, fileName, epsilon);
  fprintf('Completed %d cases\n', K);
  load('tmpFile');
  load('tmpFile2');
  TestResults(:, :, K) = Result;
  clear all
  load('tmpFile');
end
figure(2)
plotRes = zeros(1, nBFSStopNorm);
plotRes(:) = TestResults(1,1,:)/(TestResults(1,1,:)+...
  TestResults(2,1,:));
semilogx(BFSStopNorm, plotRes(:),'-rs','LineWidth',2,...
  'MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize',5);
title('Accuracy as function of BFSLimitNorm');
xlabel('BFSLimitNorm');
ylabel('Accuracy');
figure(3)
plotRes = zeros(1, nBFSStopNorm);
plotRes(:) = TestResults(3,1,:);
loglog(BFSStopNorm, plotRes(:),'-rs','LineWidth',2,'MarkerEdgeColor',
  'k','MarkerFaceColor','g','MarkerSize',5);
title('Average complexity as function of BFSLimitNorm');
xlabel('BFSLimitNorm');
ylabel('Complexity');
delete('tmpFile.mat');
delete('tmpFile2.mat');
return;
function [Stats] = Dense_func(LocalNodeSampleNorm, LocalBFSStopNorm, ImConnected, nTests, Draw, Verbose, fileName, eps)
%LocalNodeSampleNorm: Normalization factor of number of sampled pixels
%LocalBFSNorm: Normalization factor of BFS size
%ImConnected: tells the program if the image is connected or not
%nTests: Number of tests to perform
%Draw: Enable/disable drawing
%Verbose: Verbosity level of function
%fileName: Name of the file the image is stored in
%eps: epsilon of the image

%%%%%%%%%%%%%%%%%%%%%%%%%%
%Stats description
%%%%%%%%%%%%%%%%%%%%%%%%%%
%Stats(1,1) = Number of successful tests
%Stats(2,1) = Number of failed tests
%Stats(3,1) = average number of Nodes covered per test
%Stats(4,:) = index of iteration number
%Stats(5,1:Niter) = average number of nodes covered in each iteration
%Stats(6,1:Niter) = Number of sampled 1-pixels in each iteration
%Stats(7,1:Niter) = BFSStopValue in each iteration
%Stats(8,1) = Average number of iterations per test

file=['../images/', fileName, '.jpg'];
Im = im2single(imread(char(file)));
if(size(Im,3) > 1)
    Im = Im(:,:,1);
end

% Measure w(M) and calculate epsilon
Hamm = length(find(Im == 1));
if(Verbose == 1)
    fprintf('Hammign distance = %d\n', Hamm);
end
Niter = ceil(log10(1/eps));
if(Verbose == 1)
    fprintf('Number of iterations = %d\n', Niter);
    fprintf('Number of 1-pixels to invert = %d\n', eps*Hamm);
end
pVect = find(Im == 1);
Stats = zeros(8,30);
if(ImConnected == 1)
    for k = 1:nTests
        NodesCoveredInIter = 0;
        NodesCoveredTotal = 0;
        for iter = 1:Niter
            pNum = ceil(log10(1/eps)/(2^iter)*eps); %Number of nodes to select
            Stats(6,iter) = pNum;
            if(Verbose > 0)
                fprintf('Number of sampled 1-pixels = %d\n', pNum);
            end
            if pNum < 1
                pNum = 1;
            end
            Stats(5,iter) = pNum;
            Stats(6,iter) = pNum;
            Stats(7,iter) = BFSStopValue in each iteration
            Stats(8,1) = Average number of iterations per test
        end
    end
end

if (Verbose > 0)
    fprintf('Randomly selecting %d nodes\n',pNum);
end
if (pNum > Hamm)
    fprintf('Error in pNum calculation');
end
%calculate stop condition for BFS
BFSMax = ceil(32 * (2^iter) * size(Im,1) * size(Im,2) /
    (eps * Hamm) / LocalBFSStopNorm);
if (Verbose > 0)
    fprintf('Calculated BFSStop = %d\n', BFSMax);
end
if BFSMax > Hamm
    BFSMax = Hamm;
    if (Verbose > 0)
        fprintf('BFS stop condition is too high, Stopping
    at hamming dist (%d).\n', BFSMax);
    end
end
Stats(7, iter) = BFSMax;
if (Verbose > 0)
    fprintf('BFSStop value is %d\n', BFSMax);
end
% Step1: randomly select 1/eps 1-pixels
randidx = randperm(length(pVect));
IdxSel = pVect(randidx(1:pNum));
MarkNodes = Im;
MarkNodes = double(MarkNodes);
SelNodes = zeros(ceil(pNum),2); %coordinates of sel. nodes
[row col] = ind2sub(size(Im),IdxSel(1:pNum));
SelNodes(:, :) = [row col];
%Step2: Perform BFS on each selected node
%Perform BFS on the first node in SelNodes
if k==1 && iter==1 && Draw==1
    figure(1)
end
for Nd = 1:pNum
    StartNd = SelNodes(Nd, :);
    if (Verbose > 1)
        [MarkNodes IsConn Nnodes] = 
        MatlabDll(StartNd(1),StartNd(2),MarkNodes,
        BFSMax, Nd, 1);
    else
        [MarkNodes IsConn Nnodes] = 
        MatlabDll(StartNd(1),StartNd(2),MarkNodes,
        BFSMax, Nd, 0);
    end
    NodesCoveredInIter = NodesCoveredInIter + Nnodes;
    if k==1 && Draw==1
        if Nd > 1
            DispLastMarkNodes = DispMarkNodes;
        end
        DispMarkNodes = MarkNodes;
        DispMarkNodes(find(DispMarkNodes > 1)) = 0;
    end
end
end
if (Verbose > 0)
    fprintf('Randomly selecting %d nodes\n',pNum);
end
if (pNum > Hamm)
    fprintf('Error in pNum calculation');
end
%calculate stop condition for BFS
BFSMax = ceil(32 * (2^iter) * size(Im,1) * size(Im,2) /
    (eps * Hamm) / LocalBFSStopNorm);
if (Verbose > 0)
    fprintf('Calculated BFSStop = %d\n', BFSMax);
end
if BFSMax > Hamm
    BFSMax = Hamm;
    if (Verbose > 0)
        fprintf('BFS stop condition is too high, Stopping
    at hamming dist (%d).\n', BFSMax);
    end
end
Stats(7, iter) = BFSMax;
if (Verbose > 0)
    fprintf('BFSStop value is %d\n', BFSMax);
end
% Step1: randomly select 1/eps 1-pixels
randidx = randperm(length(pVect));
IdxSel = pVect(randidx(1:pNum));
MarkNodes = Im;
MarkNodes = double(MarkNodes);
SelNodes = zeros(ceil(pNum),2); %coordinates of sel. nodes
[row col] = ind2sub(size(Im),IdxSel(1:pNum));
SelNodes(:, :) = [row col];
%Step2: Perform BFS on each selected node
%Perform BFS on the first node in SelNodes
if k==1 && iter==1 && Draw==1
    figure(1)
end
for Nd = 1:pNum
    StartNd = SelNodes(Nd, :);
    if (Verbose > 1)
        [MarkNodes IsConn Nnodes] = 
        MatlabDll(StartNd(1),StartNd(2),MarkNodes,
        BFSMax, Nd, 1);
    else
        [Mark Nodes IsConn Nnodes] = 
        MatlabDll(StartNd(1),StartNd(2),MarkNodes,
        BFSMax, Nd, 0);
    end
    NodesCoveredInIter = NodesCoveredInIter + Nnodes;
    if k==1 && Draw==1
        if Nd > 1
            DispLastMarkNodes = DispMarkNodes;
        end
        DispMarkNodes = MarkNodes;
        DispMarkNodes(find(DispMarkNodes > 1)) = 0;
    end
end
 subplot(131)
 imshow(Im)
 subplot(133)
 imshow(DispMarkNodes)
 if Nd > 1
 subplot(132)
 imshow(DispLastMarkNodes)
 end
drawnow
 end
 if IsConn == 0
 break;
 end
 if(Verbose > 0)
 fprintf('NodesCovered in iteration = %d 

', NodesCoveredInIter);
 end
 Stats(4, iter) = iter;
 Stats(5, iter) = Stats(5, iter)*((k-1)/k) +
 NodesCoveredInIter/k;
 %calculate average number of iterations
 Stats(8,1) = (Stats(8,1)*(k-1) + iter)/k;
 NodesCoveredTotal = NodesCoveredTotal + NodesCoveredInIter;
 NodesCoveredInIter = 0;
 if IsConn == 0
 break;
 end
 end
 %End of Test, gather statistics
 %calculate average of Nodes covered in all tests
 Stats(3,1) = Stats(3,1)*((k-1)/k) + NodesCoveredTotal/k;
 NodesCoveredTotal = 0;
 if IsConn == 0
 if(Verbose > 0)
 fprintf('Graph is NOT connected
');
 end
 %algorithm provided a wrong result
 Stats(2,1) = Stats(2,1) + 1;
 else
 if(Verbose > 0)
 fprintf('Graph is connected\n');
 end
 %algorithm provided a correct result
 Stats(1,1) = Stats(1,1) + 1;
 end
 fprintf('Success rate is %.2f\n',Stats(1,1)/
 (Stats(1,1)+Stats(2,1)));
clear MarkNodes;
clear SelNodes;
clear randidx;
clear IdxSel;
clear DispMarkNodes;
clear DispLastMarkNodes;
clear row;
clear col;
clear Nd;
clear StartNd;
clear IsConn;
fprintf('Completed %d tests\n', k);
save('tmpFile2', 'k', 'pVect', 'pNum', 'BFSMax', 'Hamm', 'Im', 'eps', 'LocalNodeSampleNorm', 'nTests', 'StopIter', 'ImConnected', 'Niter', 'Stats', 'Draw', 'Verbose', 'LocalBFSStopNorm');
clear all
load('tmpFile2');
end
else %ImConnected == 0
if (Verbose > 0)
    fprintf('LocalNodeSampleNorm = %d\n', LocalNodeSampleNorm);
end
for k = 1:nTests
    NodesCoveredInIter = 0;
    NodesCoveredTotal = 0;
    for iter = 1:Niter
        pNum = ceil(log10(1/eps)/(2^iter)*eps) / LocalNodeSampleNorm; %Number of nodes to select
        Stats(6, iter) = pNum;
        if (Verbose > 0)
            fprintf('Number of sampled 1-pixels = %d\n', pNum);
        end
        if pNum < 1
            pNum = 1;
        end
        if (Verbose > 0)
            fprintf('Randomly selecting %d nodes\n', pNum);
        end
        if (pNum > Hamm)
            fprintf('Error in pNum calculation');
        end
        % calculate stop condition for BFS
        BFSMax = ceil(32 * (2^iter) * size(Im,1) * size(Im,2) / (eps * Hamm) / LocalBFSStopNorm);
        if (Verbose > 0)
            fprintf('Calculated BFSStop = %d\n', BFSMax);
        end
        if BFSMax > Hamm
            BFSMax = Hamm;
            if (Verbose > 0)
                fprintf('BFS stop condition is too high, Stopping at hamming dist (%d).\n', BFSMax);
            end
        end
        Stats(7, iter) = BFSMax;
        if (Verbose > 0)
            fprintf('BFSStop value is %d\n', BFSMax);
        end
        % Step1: randomly select 1/eps '1' pixels
        randidx = randperm(length(pVect));
        IdxSel = pVect(randidx(1:pNum));
        MarkNodes = Im;
        MarkNodes = double(MarkNodes);
%array of coordinates of selected nodes
SelNodes = zeros(ceil(pNum),2);
[rw col] = ind2sub(size(Im),IdxSel(1:pNum));
SelNodes(:,:,1) = [row col];
%Step2: Perform BFS on each selected node
%Perform BFS on the first node in SelNodes
if k==1 && iter==1 && Draw==1
    figure(1)
end
for Nd = 1:pNum
    if mod(Nd,500) == 0
        fprintf(['Processing node: ', num2str(Nd), 'out of ', num2str(pNum), '\n']);
    end
    StartNd = SelNodes(Nd,:);
    if(Verbose > 1)
        [MarkNodes IsConn Nnodes] = MatlabDll(StartNd(1),StartNd(2),MarkNodes,
            BFSMax, Nd, 1);
    else
        [MarkNodes IsConn Nnodes] = MatlabDll(StartNd(1),StartNd(2),MarkNodes,
            BFSMax, Nd, 0);
    end
    NodesCoveredInIter = NodesCoveredInIter + Nnodes;
    if k==1 && Draw==1
        if Nd > 1
            DispLastMarkNodes = DispMarkNodes;
        end
        DispMarkNodes = MarkNodes;
        DispMarkNodes(find(DispMarkNodes > 1)) = 0;
        subplot(131)
        imshow(Im)
        subplot(133)
        imshow(DispMarkNodes)
        if Nd > 1
            subplot(132)
            imshow(DispLastMarkNodes)
        end
        drawnow
    end
    if IsConn == 0
        break;
    end
end
if(Verbose > 0)
    fprintf('NodesCovered in iteration = %d 

', NodesCoveredInIter);
end
Stats(4, iter) = iter;
Stats(5, iter) = Stats(5, iter)*((k-1)/k) + NodesCoveredInIter/k;
%calculate average number of iterations
Stats(8,1) = (Stats(8,1)*(k-1) + iter)/k;
NodesCoveredTotal = NodesCoveredTotal + NodesCoveredInIter;
NodesCoveredInIter = 0;
if IsConn == 0
    fprintf('Graph is NOT connected\n');
    break;
end

%End of Test, gather statistics
%calculate average of Nodes covered in all tests
Stats(3,1) = Stats(3,1)*((k-1)/k) + NodesCoveredTotal/k;
NodesCoveredTotal = 0;
if IsConn == 0
    if (Verbose > 0)
        fprintf('Graph is NOT connected\n');
    end
    % algorithm provided a correct result
    Stats(1,1) = Stats(1,1) + 1;
else
    if (Verbose > 0)
        fprintf('Graph is connected\n');
    end
    % algorithm provided a wrong result
    Stats(2,1) = Stats(2,1) + 1;
end
fprintf('Success rate is %.2f\n', Stats(1,1)/
    (Stats(1,1)+Stats(2,1)));
clear MarkNodes;
clear SelNodes;
clear randidx;
clear IdxSel;
clear DispMarkNodes;
clear DispLastMarkNodes;
clear row;
clear col;
clear Nd;
clear StartNd;
clear IsConn;
fprintf('Completed %d tests. Success percentage %.3f \n', k,
    Stats(1,1)/Stats(1,1)+Stats(2,1));
save('tmpFile2', 'k', 'Stats', 'pVect', 'pNum', 'BFSMax',
    'Hamm', 'Im', 'eps', 'LocalNodeSampleNorm',...
    'nTests', 'StopIter', 'ImConnected', 'Niter', 'Draw',
    'Verbose', 'LocalBFSStopNorm');
clear all
load('tmpFile2');
end
end
11.2. Sparse image creation and Connectivity Testing

clear all
close all
% Sparse_main.m %

% This application is an implementation of the connectivity testing algorithm for sparse images published by Dana Ron and Gilad Tsur

clear all
close all
% Add Mex file DLL to path
path(path,'../MatlabDll_BFS/Debug');
ImConnected = 0; % Determines if Image is connected or not
nTests = 100;
Draw = 1; % enable/disable drawing
Verbose = 0; % verbosity level
eps = 0.012; % epsilon of image
LargestCE = 1335; % size of largest connectivity element
fileName = 'cars1'; % image file name
showIm = 0;
SaveImage = 0; % enable/disable image saving
thresh = 0.1;
EdgeCutoff = 10;
doMed = 0; % perform median filtering before applying threshold
doMedSecond = 0; % perform median filtering after applying threshold
doFilterSmallParticles = 1; % enable/disable filtering of small particles
PartFiltSize = 5;
PartFiltThresh = 12;
doThinning = 1; % enable/disable thinning
median_size = 9;

% Pre-process image
file=['../images/', fileName, '.jpg'];
Im = im2single(imread(char(file)));
if(size(Im,3) > 1)
    Im = Im(:,:,1);
end
if showIm == 1
    subplot(221)
    imshow(Im);
end
% Perform median filtering
if doMed == 1
    if(median_size > 0)
        Im = medfilt2(Im, [median_size median_size]);
    end
end
if showIm == 1
    subplot(222)
    imshow(Im);
end
%Sobel filtering
ImGradX = imfilter(Im, [-1 0 1; -2 0 2; -1 0 1], 'symmetric');
ImGradY = imfilter(Im, [1 2 1; 0 0 0; -1 -2 -1], 'symmetric');
Im = sqrt(ImGradX.^2 + ImGradY.^2);
if showIm == 1
    subplot(223)
    imshow(Im);
end
Im(find(Im > thresh)) = 1;
Im(find(Im <= thresh)) = 0;
%Perform median filtering again
if doMedSecond == 1
    median_size = 3;
    if(median_size > 0)
        Im = medfilt2(Im, [median_size median_size]);
    end
end
if doFilterSmallParticles == 1
    halfFiltsize = (PartFiltSize-1)/2;
    for row = PartFiltSize:size(Im,2) - PartFiltSize
        for col = PartFiltSize:size(Im,1) - PartFiltSize
            box = Im(col-halfFiltsize:col+halfFiltsize, row-halfFiltsize:row+halfFiltsize);
            if length(find(box == 1)) < PartFiltThresh
                Im(col,row) = 0;
            end
        end
    end
end
if doThinning == 1
    ImCleaned = Im;
    for row = 2:size(Im,2) - 1
        for col = 2:size(Im,1) - 1
            if (Im(col+1,row) + Im(col-1,row) + Im(col,row+1) + Im(col,row-1)) == 4
                ImCleaned(col,row) = 0;
            end
        end
    end
    Im = ImCleaned;
end
if EdgeCutoff > 0
    Im = Im(EdgeCutoff:end-EdgeCutoff, :);
    Im = Im(:, EdgeCutoff:end - EdgeCutoff);
end
if showIm == 1
    subplot(224)
    imshow(Im);
end
Im = round(Im);
%Save Image
if SaveImage == 1
    fileName=['../images/', fileName, '_preProc_sparse_thresh_', num2str(thresh)];
    imwrite(Im, [fileName, '.jpg'], 'jpeg', 'quality', 100);
end
%Test if image is sparse
SparsityBorder = ((size(Im,1) + size(Im,2)) / 2) ^ (4/3);
if length(find(Im == 1)) < SparsityBorder
    if (Verbose == 1)
        fprintf(['Image is sparse, total of ', num2str(length(find(Im == 1))), ' 1-pixels
']);
    else
        fprintf(['Image is sparse
']);
    end
else
    if (Verbose == 1)
        fprintf(['Image is dense ', num2str(length(find(Im == 1))), ' pixels. For Sparse requires up to ', num2str(SparsityBorder), '. Exiting
']);
    else
        fprintf('Image is dense. Exiting
');
    end
end
if showIm == 1
    return;
end
%Test if image is connected
%BBsizeNorm = [0.1:0.1:1 1.2:0.2:10];
BBsizeNorm = 1;
nBBsizeNorm = length(BBsizeNorm);
%BFSNorm = [1:10 20:10:100 200 500];
BFSNorm = [1];
nBFSNorm = length(BFSNorm);
NodeSampleNorm = [1:10 20:10:100];
%nNodeSampleNorm = length(NodeSampleNorm);
SimResult = zeros(6,30);
%for k=1:nBBsizeNorm
%    for k=1:nBFSNorm
        for k=1:nNodeSampleNorm
            SimResult(:,k) = Sparse_func(ImConnected, nTests, Draw, Verbose, Im, eps, BBsizeNorm(1), BFSNorm(1), NodeSampleNorm (k), LargestCE);
        end
end
figure(2)
plot(NodeSampleNorm, SimResult(1,1:nNodeSampleNorm)/nTests, '-rs','LineWidth',2,'MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize',4);
title('Success rate as function of node sample norm');
figure(3)
semilogx(NodeSampleNorm, SimResult(3,1:nNodeSampleNorm), '-rs','LineWidth',2,'MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize',5);
hold on
semilogx(NodeSampleNorm, SimResult(6,1:nNodeSampleNorm), ':rs','LineWidth',2,'MarkerEdgeColor','k','MarkerFaceColor','b','MarkerSize',5);
hold off
legend('Real image', 'Worst-case');
title('Complexity as function of NodeSampleNorm');
xlabel('NodeSampleNorm');
ylabel('Complexity');
function [Stats] = Sparse_func(ImConnected, nTests, Draw, Verbose, Im, eps, BBNorm, BFSNorm, NodeSampleNorm, LargestCE)

% ImConnected: tells the program if the image is connected or not
% nTests: Number of tests to perform
% Draw: Enable/disable drawing
% Verbose: Verbosity level of function
% Im: Input image
% eps: Image epsilon
% BBNorm: Normalization factor of back-bone matrix size
% BFSNorm: Normalization factor of BFS size
% NodeSampleNorm: Normalization factor of number of sampled pixels
% LargestCE: Size of largest connectivity element

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Stats description
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Stats = zeros(6,1);
Stats(1,1) = Number of successful tests
Stats(2,1) = Number of failed tests
Stats(3,1) = average number of Nodes covered per test
Stats(4,1) = number of tests completed in BB phase
Stats(5,1) = number of tests completed in 2nd phase
Stats(6,1) = Algorithm complexity in worst-case

if ImConnected == 1
    fprintf('Program does not support connected images\n');
    return;
end

size(Im);
% Measure w(M) and calculate epsilon
Hamm = length(find(Im == 1));
if (Verbose == 1)
    fprintf('Hamming distance = %d\n', Hamm);
end

%build backbone graph
BBsize = round(sqrt(Hamm)/BBNorm)  %width of each backbone sub-matrix
if BBsize > min(size(Im,1), size(Im,2))
    BBsize = min(size(Im,1), size(Im,2))
end
if BBsize == 0
    BBsize = 1;
end
%number of BB submatrices along the X dimension
BBdimX = ceil((size(Im,2) / BBsize))
%number of BB submatrices along the Y dimension
BBdimY = ceil((size(Im,2) / BBsize))
if (Verbose == 1)
    fprintf('backbone is %d x %d submatrices, each of size %d\n',
        BBdimY, BBdimX, BBsize);
end

71
BBsampleNum = round(sqrt(Hamm)*log(Hamm)*BBNorm*BBNorm); % increase number of sampled pixels by BBNorm^2 since there are BBNorm^2 more sub-matrices
if BBsampleNum > Hamm
    BBsampleNum = Hamm;
end
if BBsampleNum == 0
    BBsampleNum = 1;
end
if (Verbose == 1)
    fprintf('Sampling %d one-pixels of matrix\n', BBsampleNum);
end
Stats(6,1) = BBsampleNum + LargestCE;

for Test = 1:nTests
    BBgraph = zeros(BBdimY, BBdimX);
    % Randomly select BBsampleNum 1-pixels from image
    NodesInTest = BBsampleNum;
    pVect = find(Im == 1);
    randidx = randperm(length(pVect));
    IdxSel = pVect(randidx(1:BBsampleNum));
    MarkNodes = Im;
    MarkNodes = double(MarkNodes);
    % array of coordinates of selected nodes
    SelNodes = zeros(ceil(BBsampleNum),2);
    [row col] = ind2sub(size(Im),IdxSel(1:BBsampleNum));
    SelNodes(:,:) = [row col];

    % fill backbone graph according to sampled 1-pixels
    % save the coordinates of every 1-pixel that is the first to fill a sub-matrix
    FirstPixinBB = [0 0];
    for k=1:BBsampleNum
        CoorY = ceil(SelNodes(k,1)/BBsize);
        CoorX = ceil(SelNodes(k,2)/BBsize);
        if BBgraph(CoorY, CoorX) == 0
            FirstPixinBB = [FirstPixinBB; SelNodes(k,1) SelNodes(k,2)];
            BBgraph(CoorY, CoorX) = 1;
        end
    end
    FirstPixinBB = FirstPixinBB(2:end, :);

    % check connectivity of BBgraph
    StartNd = [ceil(SelNodes(1,1)/BBsize) ceil(SelNodes(1,2)/BBsize)];
    [BBgraphRemain IsConn BBgraphNnodeNum] = MatlabDll(StartNd(1),StartNd(2),BBgraph, BBdimX*BBdimY, 1, Verbose);
    if BBgraphNnodeNum ~= length(find(BBgraph == 1))
        if Verbose == 1
            fprintf('Graph is not connected\n');
        end
        Stats(1,1) = Stats(1,1) + 1;
        Stats(3,1) = Stats(3,1)*((Test-1)/Test) + NodesInTest/Test;
        Stats(4,1) = Stats(4,1) + 1;
    end
end
continue;
end

%Perform pixel level BFS
FineSampleNum = round(((log(Hamm)/eps)*BBNorm*BBNorm/
    NodeSampleNorm);
if FineSampleNum > length(find(BBgraph == 1))
    FineSampleNum = length(find(BBgraph == 1));
end
if FineSampleNum < 1
    FineSampleNum = 1;
end

%Randomly select BBSampleNum 1-pixels from FirstPixinBB
randidx = randperm(size(FirstPixinBB, 1));
randidx = randidx(1:FineSampleNum);
SelNodes = FirstPixinBB(randidx, :);

%Perfrom BFS from every sampled 1-pixel
BFSStop = 8*sqrt(Hamm)/eps/BFSNorm;
if BFSStop > Hamm
    BFSStop = Hamm;
end
if BFSStop < 1
    BFSStop = 1;
end

MarkNodes = Im;
MarkNodes = double(MarkNodes);
IsConn = 1;
for Nd=1:length(SelNodes)
    [MarkNodes IsConn Nnodes] = MatlabDll(SelNodes(Nd, 1),
        SelNodes(Nd, 2), MarkNodes, BFSStop, Nd, Verbose);
    NodesInTest = NodesInTest + Nnodes;
    if IsConn == 0
        Stats(1,1) = Stats(1,1) + 1;
        Stats(3,1) = Stats(3,1)*((Test-1)/Test) + NodesInTest/Test;
        Stats(5,1) = Stats(5,1) + 1;
        ifVerbose == 1
            fprintf('IsConn is zero\n');
        end
        break;
    end
end
if IsConn == 1
    Stats(2,1) = Stats(2,1) + 1;
    Stats(3,1) = Stats(3,1)*((Test-1)/Test) + NodesInTest/Test;
    Stats(5,1) = Stats(5,1) + 1;
end
end
11.3. BFS function (C wrapped by MEX)

```c
#include "mex.h"
#include "engine.h"
#include "mat.h"
#include "matrix.h"

struct nodeStruct {
    double X;
    double Y;
};

double BFS(double* BFSMat, mwSize ncols, mwSize nrows, double StartX, double StartY, double nMaxNodes, double nIteration, double verbose, double* NodesCovered) {
    nodeStruct * pnode;
    int nStartInd = 0;
    int nEndInd = 0;
    int nFullness = 0;
    int nCntNodes = 0;
    int nLength = 2 * (ncols + nrows);
    double IsConn = 0.0f;
    nodeStruct CurrNode;
    pnode = (nodeStruct*)malloc(sizeof(nodeStruct) * nLength);
    pnode[nEndInd].X = StartX;
    pnode[nEndInd].Y = StartY;
    nFullness++;
    nCntNodes++;
    nEndInd++;

    if (verbose > 0)
        mexPrintf("BFS (start):Current node value is %.1f\n", BFSMat[(int)((pnode[0].X-1)*nrows+pnode[0].Y-1)]);
    //Number of nodes covered is at least one
    *NodesCovered = 1.0;
    if (BFSMat[(int)((pnode[0].X-1)*nrows+pnode[0].Y-1)] > 1.0f) {
        //Start Node was already visited and Node must be part of a
        //large connectivity element
        if (verbose > 0)
            mexPrintf("BFS: Node (%.0f,%.0f) was already
visited\n", StartY, StartX);
        IsConn = 1.0f;
        return IsConn;
    } else {
        if (verbose > 0)
            mexPrintf("Marking node (1) (%.0f,%.0f) in BFS\n", StartY, StartX);
    }
}
```

74
while(nFullness > 0) {
  //Get node from fifo
  CurrNode.X = pnode[nStartInd].X;
  CurrNode.Y = pnode[nStartInd].Y;
  if(Verbose > 0) {
    mexPrintf("Processing node (%.0f,%.0f) in BFS
loop\n", CurrNode.Y, CurrNode.X);
  }
  //explore current node neighbors
  if(Verbose > 0)
    mexPrintf("Evaluating direction up\n");
  //search up
  if((CurrNode.Y > 1) && (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-2)] == 1.0f)) {
    if(Verbose > 0)
      mexPrintf("Marking node (up) (%.0f,%.0f) in
BFS\n", CurrNode.Y-1, CurrNode.X);
    mexPrintf("BFSMat node (%.0f,%.0f) = %.0f\n", CurrNode.Y-1, CurrNode.X,
BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-2)]);
    BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-2)] =
nIteration + 1.0;
    pnode[nEndInd].X = CurrNode.X;
    pnode[nEndInd].Y = CurrNode.Y-1;
    nFullness++;
    nCntNodes++;
    *NodesCovered += 1.0;
    nEndInd = (++nEndInd) % nLength;
    if(nCntNodes >= nMaxNodes) {
      IsConn = 1.0f;
      return IsConn;
    }
  }
  else if((CurrNode.Y > 1) && (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-2)] > 1.0f) &&
    (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-2)]) <=
nIteration)) {
    if(Verbose > 0)
      mexPrintf("Found already visited node (up)
(%.0f,%.0f) in BFS\n", CurrNode.Y-1, CurrNode.X);
    IsConn = 1.0f;
  }
  if(Verbose > 0)
    mexPrintf("Evaluating direction down\n");
// search down
if((CurrNode.Y < nrows) && (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y)] == 1.0f))
{
    if(verbose > 0)
        mexPrintf("Marking node (down) (%.0f,%.0f) in BFS\n", CurrNode.Y+1,CurrNode.X);
    BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y)] = nIteration + 1.0;
    pnode[nEndInd].X = CurrNode.X;
pnode[nEndInd].Y = CurrNode.Y+1;
nFullness++;
    nCntNodes++;
    *NodesCovered += 1.0;
    nEndInd++;
nEndInd = nEndInd % nLength;
    if(nCntNodes >= nMaxNodes)
    {
        IsConn = 1.0f;
        return IsConn;
    }
    //mexPrintf("Added new node to fifo (%f,%f), fullness = %d\n", CurrNode.Y+1,CurrNode.X,nFullness);
}
else if((CurrNode.Y < nrows) && (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y)] > 1.0f) &&
        (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y)] <= nIteration))
{
    if(verbose > 0)
        mexPrintf("Found already visited node (down) (%.0f,%.0f) in BFS\n", CurrNode.Y+1,CurrNode.X);
    IsConn = 1.0f;
}
if(verbose > 0)
mexPrintf("Evaluating direction left\n");
// search left
if((CurrNode.X > 1) && (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-1)] == 1.0f))
{
    if(verbose > 0)
        mexPrintf("Marking node (left) (%.0f,%.0f) in BFS\n", CurrNode.Y,CurrNode.X-1);
    BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y-1)] = nIteration + 1.0;
pnode[nEndInd].X = CurrNode.X-1;
pnode[nEndInd].Y = CurrNode.Y;
nFullness++;
    nCntNodes++;
    *NodesCovered += 1.0;
    nEndInd++;
nEndInd = nEndInd % nLength;
    if(nCntNodes >= nMaxNodes)
    {
        IsConn = 1.0f;
    }
}
return IsConn;

// mexPrintf("Added new node to fifo (%f,%f), fullness = %d\n", CurrNode.Y, CurrNode.X-1, nFullness);
}
else if((CurrNode.X > 1) && (BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y-1)]) > 1.0f) &&
        (BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y-1)]) <= nIteration)
{
    if(verbose > 0)
        mexPrintf("Found already visited node (left) (%.0f,%.0f) in BFS\n", CurrNode.Y, CurrNode.X-1);
    IsConn = 1.0f;
}
if(verbose > 0)
    mexPrintf("Evaluating direction right\n");
// search right
if((CurrNode.X < ncols) &&
    (BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-1)] == 1.0f))
{
    if(verbose > 0)
        mexPrintf("Marking node (right) (%.0f,%.0f) in BFS\n", CurrNode.Y, CurrNode.X+1);
    BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-1)] = nIteration + 1.0;
    pnode[nEndInd].X = CurrNode.X+1;
    pnode[nEndInd].Y = CurrNode.Y;
    nFullness++;
    nCntNodes++;
    *NodesCovered += 1.0;
    nEndInd++;
    nEndInd = nEndInd % nLength;
    if(nCntNodes >= nMaxNodes)
    {
        IsConn = 1.0f;
        return IsConn;
    }
    // mexPrintf("Added new node to fifo (%f,%f), fullness = %d\n", CurrNode.Y, CurrNode.X+1, nFullness);
}
else if((CurrNode.X < ncols) &&
        (BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-1)] > 1.0f) &&
        (BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-1)]) <= nIteration)
{
    if(verbose > 0)
        mexPrintf("Found already visited node (right) (%.0f,%.0f) in BFS\n", CurrNode.Y, CurrNode.X+1);
    IsConn = 1.0f;
}
if(verbose > 0)
    mexPrintf("Evaluating direction up+left\n");
// search diag up and left
if((CurrNode.Y > 1) && (CurrNode.X > 1) &&
(BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y-2)] == 1.0f))
{
    if(verbose > 0)
        mexPrintf("Marking node (up+left) (%.0f,%.0f)
in BFS\n", CurrNode.Y-1,CurrNode.X-1);
    BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y-2)] =
nIteration + 1.0;
pnode[nEndInd].X = CurrNode.X-1;
pnode[nEndInd].Y = CurrNode.Y-1;
nFullness++;
nCtNodes++;*NodesCovered += 1.0;
nEndInd++;
nEndInd = nEndInd % nLength;
if(nCtNodes >= nMaxNodes)
    {
        IsConn = 1.0f;
        return IsConn;
    }
}
else if((CurrNode.Y > 1) && (CurrNode.X > 1)&&
(BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y-2)] > 1.0f) &&
(BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y-2)] <=
nIteration))
{
    if(verbose > 0)
        mexPrintf("Found already visited node (up+left)
(%.0f,%.0f) in BFS\n", CurrNode.Y-1,CurrNode.X-1);
    IsConn = 1.0f;
}
if(verbose > 0)
mexPrintf("Evaluating direction down+left\n");
//search diag down and left
if((CurrNode.Y < nrows) && (CurrNode.X > 1) &&
(BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y)] == 1.0f))
{
    if(verbose > 0)
        mexPrintf("Marking node (down+left) (%.0f,%.0f)
in BFS\n", CurrNode.Y+1,CurrNode.X-1);
    BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y)] =
nIteration + 1.0;
pnode[nEndInd].X = CurrNode.X-1;
pnode[nEndInd].Y = CurrNode.Y+1;
nFullness++;
nCtNodes++;*NodesCovered += 1.0;
nEndInd++;
nEndInd = nEndInd % nLength;
if(nCtNodes >= nMaxNodes)
    {
        IsConn = 1.0f;
        return IsConn;
    }
//mexPrintf("Added new node to fifo (%f,%f),fullness
if((CurrNode.Y < nrows) && (CurrNode.X > 1) &&
(BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y)] >
1.0f) && (BFSMat[(int)((CurrNode.X-2)*nrows+CurrNode.Y)] <= nIteration))
{
    if(verbose > 0)
        mexPrintf("Found already visited node
down+left) (%.0f,%.0f) in BFS\n", CurrNode.Y+1,CurrNode.X-1);
    IsConn = 1.0f;
}
if(verbose > 0)
    mexPrintf("Evaluating direction up+right\n");
// search diag up and right
if((CurrNode.Y > 1) && (CurrNode.X < ncols) &&
(BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-2)] == 1.0f))
{
    if(verbose > 0)
        mexPrintf("Marking node (up+right) (%.0f,%.0f)
in BFS\n", CurrNode.Y-1,CurrNode.X+1);
    BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-2)] = nIteration + 1.0;
    pnode[nEndInd].X = CurrNode.X+1;
    pnode[nEndInd].Y = CurrNode.Y-1;
    nFullness++;
    nCntNodes++;
    *NodesCovered += 1.0;
    nEndInd++;
    nEndInd = nEndInd % nLength;
    if(nCntNodes >= nMaxNodes)
    {
        IsConn = 1.0f;
        return IsConn;
    }
}
else if((CurrNode.Y > 1) && (CurrNode.X < ncols) &&
(BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-2)] > 1.0f) &&
(BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-2)] <= nIteration))
{
    if(verbose > 0)
        mexPrintf("Found already visited node
up+right) (%.0f,%.0f) in BFS\n", CurrNode.Y-1,CurrNode.X+1);
    IsConn = 1.0f;
}
if(verbose > 0)
    mexPrintf("Evaluating direction down+right\n");
// search diag down and right
if((CurrNode.Y < nrows) && (CurrNode.X < ncols) &&
(BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y)] == 1.0f))
{
    if(verbose > 0)
        mexPrintf("Marking node (down+right)
BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y)] = nIteration + 1.0;
pnode[nEndInd].X = CurrNode.X+1;
pnode[nEndInd].Y = CurrNode.Y+1;
nFullness++;
NodesCovered += 1.0;
EndInd++;
EndInd = EndInd % nLength;
if(nCntNodes >= nMaxNodes)
{
    IsConn = 1.0f;
    return IsConn;
}
else if((CurrNode.Y < nrows) && (CurrNode.X < ncols) &&
    (BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y)] > 1.0f) &&
    (BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y)] <=
    nIteration))
{
    if(Verbose > 0)
        mexPrintf("Found already visited node (down+right) (%.0f,%.0f) in BFS\n", CurrNode.Y+1,CurrNode.X+1);
    IsConn = 1.0f;
}
//remove current node from fifo
nFullness--;
StartInd++;
StartInd = StartInd % nLength;
//mexPrintf("start = %d, end = %d\n", StartInd,EndInd);
if((nCntNodes == 1) && (verbose > 0))
{
    mexPrintf("BFS (end):Current node value is %.1f\n", BFSMat[(int)((pnode[0].X-1)*nrows+pnode[0].Y-1])]);
}
free(pnode);
if(Verbose > 0)
    mexPrintf("BFS: Processed %d Nodes\n", nCntNodes);
return IsConn;
/* The gateway function */
void mexFunction( int nlhs, mxArray *plhs[],
    int nrhs, const mxArray *prhs[])
mwSize ncols;  /* num of matrix columns*/
mwSize nrows;  /* num of matrix rows*/
double verbose;  /* Verbosity level*/
double *outMatrix;  /* output matrix */
double *IsConn;  /* output variable - Is matrix connected */
double *Nnodes;  /* Number of nodes visited during BFS*/
mwSize cnt;
double nMaxNodes;
double NodesCovered;

/* check for proper number of arguments */
/* Get verbosity level */
verbose = mxGetScalar(prhs[5]);
/* get the value of the scalar input */
StartY = mxGetScalar(prhs[0]);
StartX = mxGetScalar(prhs[1]);
if (verbose > 0)
    mexPrintf("Starting BFS from (%.0f,%.0f)\n", StartY, StartX);
/* create a pointer to the real data in the input matrix */
inMatrix = mxGetPr(prhs[2]);

/*Read value of maximum number of nodes to be processed during BFS */
nMaxNodes = mxGetScalar(prhs[3]);
if (verbose > 0)
    mexPrintf("Max nodes to be processed = %.1f\n", nMaxNodes);
nIteration = mxGetScalar(prhs[4]);
if (verbose > 0)
    mexPrintf("Iteration number is %.1f\n", nIteration);

/* get dimensions of the input matrix */
ncols = mxGetN(prhs[2]);
nrows = mxGetM(prhs[2]);
if (verbose > 0)
    mexPrintf("Input image is of size %d x %d\n", ncols, nrows);
/* create the output matrix */
plhs[0] = mxCreateDoubleMatrix(nrows, ncols, mxREAL);
plhs[1] = mxCreateDoubleMatrix(1, 1, mxREAL);
plhs[2] = mxCreateDoubleMatrix(1, 1, mxREAL);
if (verbose > 0)
    mexPrintf("Created output variables\n");

/* get a pointer to the real data in the output matrix */
outMatrix = mxGetPr(plhs[0]);
IsConn = mxGetPr(plhs[1]);
Nnodes = mxGetPr(plhs[2]);

if (verbose > 0)
    mexPrintf("Created pointers to output variables\n");

for (cnt = 0; cnt < (ncols * nrows); cnt++)
{
    outMatrix[cnt] = inMatrix[cnt];
    //mexPrintf("inMatrix[%d] = %f\n", cnt, inMatrix[cnt]);
if (verbose > 0)
    mexPrintf("Copied input to output matrix\n");

    // Elements are organized in Matrix in column first order
    /* call the computational routine */
    *IsConn = BFS(outMatrix, ncols, nrows, StartX, StartY, nMaxNodes,
                 nIteration, verbose, (double*)&NodesCovered);
    *Nnodes = NodesCovered;
    return;
}
11.4. DtC upper bound compute function

clear all
close all

% This application finds distance to connectivity by running minimum
spanning tree algorithm on connectivity elements of a binary image

clear all
close all

% Add Mex file DLL to path
path(path,'../MatlabDll_MST_DtC/Debug');

Draw = 0; % enable/disable drawing
Verbose = 1; % verbosity level

fileName = 'Coke_preProc_sparse_thresh_0.1_eps_0.002';

% load image
file = '../images/' filename '.jpg';
Im = im2double(imread(char(file)));

% send image to MatlabDll to find graph nodes and edges
[Im Distances Nnodes Sizes] = MatlabDll(Im, 1, EdgeDeletion); % 3rd
% parameter enables/disables deletion of edges that cross other
% connectivity elements, 0 means delete, 1 means save

% create sparse vector
Graph_Weights = zeros(3,1);
for i=1:size(Distances,1)
    if length(find(Distances(i,:) > 0)) == 0
        Distances(i, i) = size(Im,1)*size(Im,2); % this is a trick to
        allow DG to be a square matrix
    end
end

for i=1:Nnodes
    for j=1:Nnodes
        if Distances(j, i) ~= 0
            Graph_Weights = [Graph_Weights zeros(3,1)];
            Graph_Weights(:, end) = [i; j; Distances(j, i)];
        end
    end
end
Graph_Weights = Graph_Weights(:, 2:end);

DG = sparse(Graph_Weights(1,:), Graph_Weights(2,:),
Graph_Weights(3,:));

[ST, pred] = graphminspantree(tril(DG+DG'));
DtC = zeros(Nnodes+1, 1);
DtC(1) = sum(sum(full(ST)))/Num1Pixels;

fprintf(['Distance to connectivity based on all elements is ',
    num2str(DtC(1)), '\n']);
%Sort connectivity elements by size
%SizesOrder holds the index value of the sorted elements, e.g.
SizesOrder(1) is index of largest element, SizesOrder(2) is index of
2nd largest element and so
[SortedSizes SizesOrder] = sort(Sizes, 1, 'descend');

%Iterate and remove connectivity elements by size from large to small
%and check DtC:
for ElemCounter=1:Nnodes
  i = SizesOrder(ElemCounter);
  RemovedSize = Sizes(i);
  if i==1
    ActiveDistances = Distances(2:end, :);
    ActiveDistances = ActiveDistances(:, 2:end);
  elseif i == Nnodes
    ActiveDistances = Distances(1:end-1, :);
    ActiveDistances = ActiveDistances(:, 1:end-1);
  else
    ActiveDistances = [Distances(1:i-1, :); Distances(i+1:end, :)];
    ActiveDistances = [ActiveDistances(:, 1:i-1) ActiveDistances(:, i+1:end)];
  end
  ActiveNodes = Nnodes - 1;
  %create sparse vector
  Graph_Weights = zeros(3,1);
  for i=1:size(ActiveDistances,1)
    if length(find(ActiveDistances(i,:) > 0)) == 0
      %this is a trick to allow DG to be a square matrix
      ActiveDistances(i, i) = size(Im,1)*size(Im,2);
    end
  end
  for i=1:ActiveNodes
    for j=1:ActiveNodes
      if ActiveDistances(j, i) ~= 0
        Graph_Weights = [Graph_Weights zeros(3,1)];
        Graph_Weights(:, end) = [i; j; ActiveDistances(j, i)];
      end
    end
  end
  Graph_Weights = Graph_Weights(:, 2:end);
  DG = sparse(Graph_Weights(1,:), Graph_Weights(2,:),
               Graph_Weights(3,:));
  [ST, pred] = graphminspantree(tril(DG+DG'));
  DtC(1+ ElemCounter) = (sum(sum(full(ST))) +
                         RemovedSize)/Num1Pixels;
end
fprintf(['Minimal DtC for removing one element is ',
            num2str(min(DtC(2:end))), '
']);
figure(2)
plot(1:length(DtC(2:end)), flipud(DtC(2:end)'), '--rs',
     'LineWidth',2,'MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize',
     10);
title('DtC for removing a single element, smaller element first');
% Try removing more than one element and more, those with size in the lower part of the sorted list
DtC_partial = zeros(Nnodes-2,1);
for LowerQSize = 1:(Nnodes-2)
    % Sort indices of elements to be removed in descending order
    IndicesToRemove = SizesOrder(end - LowerQSize+1 : end);
    IndicesToRemove = sort(IndicesToRemove, 1, 'descend');
    RemovedSize = 0;
    ActiveNodes = Nnodes;
    ActiveDistances = Distances;
    for ElemCounter=1:LowerQSize
        i = IndicesToRemove(ElemCounter);
        RemovedSize = RemovedSize + Sizes(i);
        if i==1
            ActiveDistances = ActiveDistances(2:end, :);
        else if i == Nnodes
            ActiveDistances = ActiveDistances(1:end-1, :);
        else
            ActiveDistances = [ActiveDistances(1:i-1, :); ActiveDistances(i+1:end, :)];
        end
        ActiveNodes = ActiveNodes - 1;
    end

    % Create sparse vector
    Graph_Weights = zeros(3,1);
    for i=1:size(ActiveDistances,1)
        if length(find(ActiveDistances(i,:) > 0)) == 0
            % This is a trick to allow DG to be a square matrix
            ActiveDistances(i, i) = size(Im,1)*size(Im,2);
        end
    end
    for i=1:ActiveNodes
        for j=1:ActiveNodes
            if ActiveDistances(j, i) ~= 0
                Graph_Weights = [Graph_Weights zeros(3,1)];
                Graph_Weights(:, end) = [i; j; ActiveDistances(j, i)];
            end
        end
    end
    Graph_Weights = Graph_Weights(:, 2:end);
    DG = sparse(Graph_Weights(1,:), Graph_Weights(2,:),
                 Graph_Weights(3,:));
    [ST, pred] = graphminsptree(tril(DG+DG'));
    DtC_PartialElements = (sum(sum(full(ST))) + RemovedSize)/Num1Pixels;
    DtC_partial(LowerQSize) = DtC_PartialElements;
end
figure(3)
plot(1:Nnodes-1, DtC_partial, '--rs',
     'LineWidth',2,'MarkerEdgeColor','k','MarkerFaceColor','g','MarkerSize', 10);
title('DtC for removing several element, smaller first');
fprintf(['Minimal DtC for removing ',num2str(find(DtC_partial ==
min(DtC_partial))),' elements is ', num2str(min(DtC_partial)), '
']);
11.5. Connectivity Elements graph function (C wrapped by MEX)

```c
void mexFunction( int nlhs, mxArray *plhs[],
                   int nrhs, const mxArray *prhs[])
{
    double *inMatrix;    /* MxN input matrix */
    mwSize ncols;        /* num of matrix columns*/
    mwSize nrows;        /* num of matrix rows*/
    double verbose;      /* Verbosity level */
    double *outMatrix;   /* output matrix */
    double *Distances;   /*Distances matrix*/
    double *ElemSizes;   /*Size of each connectivity element*/
    double *Nnodes;      /*Number of nodes visited during BFS*/
    mwSize cnt;
    //number of 1-pixels processed during scan
    double NodesCovered = 0.0;
    int nNumElements = 0;
    int i;
    double DelEdges;
    Conn_Element * pCE = NULL;
    Conn_Element * CurrpCE;
    pCE = (Conn_Element *)malloc(sizeof(Conn_Element));
    if(pCE == NULL)
    {
        mexPrintf("Error in allocation of pCE\n");
        return;
    }
    /* Get verbosity level */
    verbose = mxGetScalar(prhs[1]);
    if(verbose > 0)
        mexPrintf("Starting Analysis\n");
    /* create a pointer to the real data in the input matrix */
    inMatrix = mxGetPr(prhs[0]);
    /* Get definition of algorithm operative mode, delete/save edges that cross other connectivity elements */
    DelEdges = mxGetScalar(prhs[2]);
    if(DelEdges == 0.0)
        mexPrintf("Edges deletion is off\n");
    else if(DelEdges == 1.0)
        mexPrintf("Edges deletion is on\n");
    /* get dimensions of the input matrix */
    ncols = mxGetN(prhs[0]);
    nrows = mxGetM(prhs[0]);
    if(Verbose > 0)
        mexPrintf("input matrix is of size %d x %d\n", ncols,
                 nrows);
```
/* create the output matrix */
plhs[0] = mxCreateDoubleMatrix(nrows, ncols, mxREAL);
plhs[2] = mxCreateDoubleMatrix(1, 1, mxREAL);
mexPrintf("created plhs[0] and plhs[1]\n");

/* get a pointer to the real data in the output matrix */
outMatrix = mxGetPr(plhs[0]);
Nnodes = mxGetPr(plhs[2]);
mexPrintf("received pointers for plhs[0] and plhs[1]\n");
for(cnt = 0; cnt < (ncols*nrows); cnt++)
{
    outMatrix[cnt] = inMatrix[cnt];
}
mexPrintf("After loop\n");
pCE->nIdx = 0;
pCE->pNextCE = NULL;

Scan_Image(outMatrix, ncols, nrows, verbose,
            (double*)&NodesCovered, (int*)&nNumElements, pCE);
mexPrintf("Number of connectivity elements = %d\n", nNumElements);

//Create matrix for storing distances between different
//connectivity elements
if(nNumElements < 1000)
{
    plhs[1] = mxCreateDoubleMatrix(nNumElements, nNumElements, mxREAL);
    Distances = mxGetPr(plhs[1]);
    plhs[3] = mxCreateDoubleMatrix(nNumElements, 1, mxREAL);
    ElemSizes = mxGetPr(plhs[3]);
}
else
{
    plhs[1] = mxCreateDoubleMatrix(1, 1, mxREAL);
    Distances = mxGetPr(plhs[1]);
    mexPrintf("Too many connectivity elements. Aborting...\n");
    return;
}

//Read element sizes into vector so it can be sent to caller
CurrpCE = pCE;
for(i=0; i<nNumElements; i++)
{
    ElemSizes[i] = (double)CurrpCE->size;
    if(CurrpCE->pNextCE != NULL)
    {
        CurrpCE = CurrpCE->pNextCE;
    }
    else
    {
        mexPrintf("Error while traveling on connectivity
elements list\n");
        break;
    }
Zero distances matrix

```c
for (i=0; i<(nNumElements*nNumElements); i++)
{
    Distances[i] = 0.0f;
}
```

Go through all combinations of different connectivity elements and find distance between them

```c
Find_Distances(outMatrix, ncols, nrows, verbose, nNumElements,
    pCE, Distances, DelEdges);
```

```c
*Nnodes = nNumElements;
```

```c
mexPrintf("Freeing up memory\n");
Finish(pCE);
mexPrintf("End of MatlabDll\n");
return;
```

Scan Image

```c
void Scan_Image(double* BFSMat, mwSize ncols, mwSize nrows,
    double verbose, double* NodesCovered, int* nNumElements,
    Conn_Element* pCE)
{
    Conn_Element* CurrpCE = pCE;
    unsigned int i, j;
    int nCurrElem = 0;
    verbose = 0.0;
    int nTmpCnt = 0;
    int ElemSize;
    for (i=0; i<ncols; i++)
    {
        for (j=0; j<nrows; j++)
        {
            if (BFSMat[(int)(i*nrows+j)] == 1)
            {
                ElemSize = BFS_func(BFSMat, ncols, nrows, i, j,
                    nCurrElem + 1, verbose, NodesCovered);
                nCurrElem++;
                (*nNumElements)++;
                CurrpCE->nIdx = nCurrElem;
                CurrpCE->nXcoor = i;
                CurrpCE->nYcoor = j;
                CurrpCE->size = ElemSize;
                CurrpCE->pNextCE = (Conn_Element*)malloc(sizeof(Conn_Element));
                CurrpCE = CurrpCE->pNextCE;
            }
        }
    }
}
```

return;
void Finish (Conn_Element * CE)
{
    Conn_Element* pNextCE = NULL;
    if (CE->pNextCE == NULL)
    {
        free(CE);
        return;
    }
    while (CE->pNextCE != NULL)
    {
        pNextCE = CE->pNextCE;
        free(CE);
        CE = pNextCE;
    }
}
11.6. Connectivity Elements distance function (C wrapped by MEX)

```c
void Find_Distances(double* BFSMat, mwSize ncols, mwSize nrows, double verbose, int nNumElements, Conn_Element * pCE, double * distances, double DelEdges)
{
    int i, j;
    mexPrintf("Inside Find_Distances. nNumElements = %d\n", nNumElements); //removeme
    PerimNode * PerimNodeList =
        (PerimNode*)malloc(nNumElements*sizeof(PerimNode));
    Conn_Element * CurrpCE = pCE;
    double * BFSMatCopied =
        (double*)malloc(ncols*nrows*sizeof(double));
    double MaxDist = sqrt((double)(ncols*nrows));
    for(i=0; i<(int)(ncols*nrows); i++)
    {
        BFSMatCopied[i] = BFSMat[i];
    }

    //Compute a linked list of perimeter pixels per each connectivity
    //element
    for(i=0; i<nNumElements; i++)
    {   PerimNodeList[i].pNext = NULL;
        Find_Perimeter(BFSMatCopied, BFSMat, ncols, nrows, verbose, CurrpCE, (PerimNode *) &PerimNodeList[i]);
        if(CurrpCE->pNextCE != NULL)
        {
            CurrpCE = CurrpCE->pNextCE;
        } else
        {
            mexPrintf("Error in reading list of connectivity
            elements\n");
            return;
        }
    }

    for(i=1; i<nNumElements; i++)
    {
        for(j=0; j<i; j++)
        {
            mexPrintf("Finding minimal distance between (%d, %d)
            to (%d, %d)\n", PerimNodeList[i].nYcoor, PerimNodeList[i].nXcoor, PerimNodeList[j].nYcoor, PerimNodeList[j].nXcoor);
            Distances[i*nNumElements + j] = Shortest_Dist((PerimNode *) &PerimNodeList[i], (PerimNode *) &PerimNodeList[j], MaxDist, BFSMat, ncols, nrows, verbose, DelEdges);
        }
    }
}
```
double Shortest_Dist(PerimNode * PerimNodeList1, PerimNode * PerimNodeList2, double MinDist, double* BFSMat, mwSize ncols, mwSize nrows, double verbose, double DelEdges)
{
    // go through pixels on edge of both elements and find shortest distance
    PerimNode * CurrNode1 = NULL;
    PerimNode * CurrNode2 = NULL;
    PerimNode * ShorestNode1 = PerimNodeList1;
    PerimNode * ShorestNode2 = PerimNodeList2;
    double dist;
    CurrNode1 = PerimNodeList1;

    while(CurrNode1 != NULL)
    {
        CurrNode2 = PerimNodeList2;
        while(CurrNode2 != NULL)
        {
            // Compute distance between two nodes
            if(dist == 0.0)
            {
                mexPrintf("distance from (%d, %d) to (%d, %d) = %f\n", CurrNode1->nYcoor, CurrNode1->nXcoor, CurrNode2->nYcoor, CurrNode2->nXcoor, dist);
            }
            else
            {
                if(dist < MinDist)
                {
                    MinDist = dist;
                    ShorestNode1 = CurrNode1;
                    ShorestNode2 = CurrNode2;
                }
                CurrNode2 = CurrNode2->pNext;
            }
        }
        CurrNode1 = CurrNode1->pNext;
    }

    if(MinDist <= 1.0f)
    {
        mexPrintf("Error: Distance between connectivity elements is %f\n", MinDist);
        return 0.0f;
    }

    MinDist = (double)Bresenham(BFSMat, ncols, nrows, ShorestNode1, ShorestNode2, DelEdges);
}
if( verbose > 0.0 )
    mexPrintf( "Min distance is \%f from (\%d, \%d) to (\%d, \%d)\n", 
        MinDist, ShorestNode1->nYcoor, ShorestNode1->nXcoor, 
        ShorestNode2->nYcoor, ShorestNode2->nXcoor );
    return MinDist;
}

int Bresenham(double* BFSMat, mwSize ncols, mwSize nrows, PerimNode * 
        ShorestNode1, PerimNode * ShorestNode2, double DelEdges)
{
    int nNodesCounter = 0;
    int nStartX = ShorestNode1->nXcoor;
    int nStartY = ShorestNode1->nYcoor;
    int nEndX = ShorestNode2->nXcoor;
    int nEndY = ShorestNode2->nYcoor;

    bool steep = abs(nEndY - nStartY) > abs(nEndX - nStartX);
    //mexPrintf("Steep  = \%d\n", steep);
    if (steep)
{
        swap(&nStartX, &nStartY);
        swap(&nEndX, &nEndY);
}
    if (nStartX > nEndX)
{
        swap(&nStartX,&nEndX);
        swap(&nStartY,&nEndY);
}
    float deltax = (float)(nEndX - nStartX);
    float deltay = (float)abs(nEndY - nStartY);
    float error = 0.0f;
    float deltaerr = deltay/deltax;
    int ystep;
    int nCurrLocX;
    int nCurrLocY = nStartY;
    if(nStartY < nEndY)
        ystep = 1;
    else
        ystep = -1;
    for(nCurrLocX = nStartX; nCurrLocX <= nEndX; nCurrLocX++)
{
        if(steep)
        {
            if((nCurrLocX > nStartX) && (nCurrLocX < nEndX) &&
                (BFSMat[nCurrLocY*(int)nrows+nCurrLocX] > 0.0f) &&
                (DelEdges == 0.0))
            {
                mexPrintf("shortest path is going through 
other connectivity element\n");
                return 0;
            }
        }
    }
}
else
{
    if((nCurrLocX > nStartX) && (nCurrLocX < nEndX) &&
        (BFSMat[nCurrLocX*(int)nrows+nCurrLocY] > 0.0f) &&
        (DelEdges == 0.0))
    {
        mexPrintf("shortest path is going through
          another connectivity element\n");
        return 0;
    }
}

if(error >= 0.5f)
{
    nCurrLocY += ystep;
    error -= 1.0f;
    nNodesCounter++;
}

else

void swap(int * pint1, int * pint2)
{
    int tmp = *pint2;
    *pint2 = *pint1;
    *pint1 = tmp;
    return;
}
void free_PerimNodeList(PerimNode * PerimNodeList, int nNumElements)
{
    int i;
    PerimNode * PerimNodeCurr;
    PerimNode * PerimNodeNext;
    for(i=0; i<nNumElements; i++)
    {
        if(PerimNodeList[i].pNext != NULL)
        {
            PerimNodeCurr = PerimNodeList[i].pNext;
            while(PerimNodeCurr->pNext != NULL)
            {
                PerimNodeNext = PerimNodeCurr->pNext;
                free(PerimNodeCurr);
                PerimNodeCurr = PerimNodeNext;
            }
            free(PerimNodeCurr);
        }
    }
    free(PerimNodeList);
    return;
}

void Find_Perimeter(double* BFSMat, double* BFSMatOrig, mwSize ncols, mwSize nrows, double verbose, Conn_Element * pCE, PerimNode * PerimListStart)
{
    //Start scanning image at given pixel of connectivity element, 
    //perform BFS and add each pixel on perimeter to linked list
    nodeStruct * pnode;
    PerimNode * PerimNodeCurr = &PerimListStart[0];
    int nStartInd = 0;
    int nEndInd = 0;
    int nFullness = 0;
    int nCntNodes = 0;
    int nLength = 2 * (ncols + nrows);
    bool PerimRes = false;
    bool firstPerimNode = true;
    nodeStruct CurrNode;
    pnode = (nodeStruct*)malloc(sizeof(nodeStruct) * nLength);
    pnode[nEndInd].X = (*pCE).nXcoor;
    pnode[nEndInd].Y = (*pCE).nYcoor;
    nFullness++;
    nCntNodes++;
    nEndInd++;
    if(verbose > 1)
        mexPrintf("Find_Perimeter: starting at (%d, %d)\n", pnode[0].Y, pnode[0].X);
    if(BFSMat[(int)(pnode[0].X*nrows+pnode[0].Y)] == 0.0)
    {
        mexPrintf("Error in Find_Perimeter. Starting at node that was already visited\n");
        return;
    }
}
while(nFullness > 0)
{
  //Get node from fifo
  CurrNode.X = pnode[nStartInd].X;
  CurrNode.Y = pnode[nStartInd].Y;
  if(verbose > 1)
  {
    mexPrintf("Processing node (%d,%d) in BFS loop\n", CurrNode.Y, CurrNode.X);
  }
  PerimRes = CheckIfPerimeter(CurrNode.Y, CurrNode.X, BFSMatOrig, ncols, nrows);
  if(PerimRes == true)
  {
    //first node found on perimeter and to be entered into list
    if(firstPerimNode == true)
    {
      BFSMat[(int)(CurrNode.X*nrows+CurrNode.Y)] = 0.2f;
      PerimNodeCurr->nXcoor = CurrNode.X;
      PerimNodeCurr->nYcoor = CurrNode.Y;
      PerimNodeCurr->pNext = NULL;
      firstPerimNode = false;
    }
    else
    {
      //allocate new PrimNode for storing node's coordinates
      PerimNodeCurr->pNext = (PerimNode*)malloc(sizeof(PerimNode));
      if(NULL != PerimNodeCurr->pNext)
      {
        PerimNodeCurr = PerimNodeCurr->pNext;
        PerimNodeCurr->pNext = NULL;
      }
      else
      {
        mexPrintf("Error in allocating memory for PerimNode\n");
        return;
      }
      PerimNodeCurr->nXcoor = CurrNode.X;
      PerimNodeCurr->nYcoor = CurrNode.Y;
    }
  }
  //explore current node neighbors
  if( verbose > 1)
    mexPrintf("Evaluating direction up\n");
  //search up
  if((CurrNode.Y > 0) && (BFSMat[(int)(CurrNode.X*nrows+CurrNode.Y-1)] >= 1.0f))
  {
    if( verbose > 1)
{
    mexPrintf("Marking node (up) (%d,%d) in BFS\n", CurrNode.Y-1,CurrNode.X);
    mexPrintf("BFSMat node (%d,%d) = %.0f\n", CurrNode.Y-1,CurrNode.X,
        BFSMat[(int)(CurrNode.X*nrows+CurrNode.Y-1)]);
}
BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-1)] = 0.5f;
pnode[nEndInd].X = CurrNode.X;
pnode[nEndInd].Y = CurrNode.Y-1;
nFullness++;
CntNodes++;
EndInd = (++EndInd) % nLength;
}
else if((CurrNode.Y > 0) &&
        (BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y-1)] < 1.0f))
{
    if(Verbose > 1)
        mexPrintf("Found already visited node (up) (%d,%d) in BFS\n", CurrNode.Y-1,CurrNode.X);
    if(Verbose > 1)
        mexPrintf("Evaluating direction down\n");
    //search down
    if((CurrNode.Y < (nrows-1)) &&
        (BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y+1)] >= 1.0f))
    {
        if(Verbose > 1)
            mexPrintf("Marking node (down) (%d,%d) in BFS\n", CurrNode.Y+1,CurrNode.X);
        BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y+1)] = 0.5f;
pnode[nEndInd].X = CurrNode.X;
pnode[nEndInd].Y = CurrNode.Y+1;
nFullness++;
CntNodes++;
EndInd++;
EndInd = nEndInd % nLength;
    }
else if((CurrNode.Y < (nrows-1)) &&
        (BFSMat[(int)((CurrNode.X)*nrows+CurrNode.Y+1)] < 1.0f))
{
    if(Verbose > 1)
        mexPrintf("Found already visited node (down) (%d,%d) in BFS\n", CurrNode.Y+1,CurrNode.X);
}
if(Verbose > 1)
    mexPrintf("Evaluating direction left\n");
    //search left
    if((CurrNode.X > 0) && (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y)]) >= 1.0f))
{
    if(Verbose > 1)
        mexPrintf("Marking node (left) (%d,%d) in BFS\n", CurrNode.X-1,CurrNode.Y)
BFS

BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y)] = 0.5f;
pnode[nEndInd].X = CurrNode.X-1;
pnode[nEndInd].Y = CurrNode.Y;
nFullness++; nCntNodes++;
nEndInd++;
nEndInd = nEndInd % nLength;
}
else if((CurrNode.X > 0) && (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y)] < 1.0f))
{
    if(verbose > 1)
        mexPrintf("Found already visited node (left) (%d,%d) in BFS\n", CurrNode.Y,CurrNode.X-1);
}
if(verbose > 1)
    mexPrintf("Evaluating direction right\n");
//search right
if((CurrNode.X < (ncols-1)) && (BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y)] >= 1.0f))
{
    if(verbose > 1)
        mexPrintf("Marking node (right) (%d,%d) in BFS\n", CurrNode.Y,CurrNode.X+1);
    BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y)] = 0.5f;
pnode[nEndInd].X = CurrNode.X+1;
pnode[nEndInd].Y = CurrNode.Y;
nFullness++; nCntNodes++;
nEndInd++;
nEndInd = nEndInd % nLength;
}
else if((CurrNode.X < (ncols-1)) && (BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y)] < 1.0f))
{
    if(verbose > 1)
        mexPrintf("Found already visited node (right) (%d,%d) in BFS\n", CurrNode.Y,CurrNode.X+1);
}
if(verbose > 1)
    mexPrintf("Evaluating direction up+left\n");
//search diag up and left
if((CurrNode.Y > 0) && (CurrNode.Y > 0) && (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-1)] >= 1.0f))
{
    if(verbose > 1)
        mexPrintf("Marking node (up+left) (%d,%d) in BFS\n", CurrNode.Y,CurrNode.X-1);
    BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-1)] = 0.5f;
pnode[nEndInd].X = CurrNode.X-1;
pnode[nEndInd].Y = CurrNode.Y-1;
nFullness++;
nCntNodes++;
  nEndInd++;
  nEndInd = nEndInd % nLength;
}
else if((CurrNode.Y > 0) && (CurrNode.X > 0) &&
    (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y-1)] < 1.0f))
{
  if( verbose > 1)
    mexPrintf("Found already visited node (up+left) (%d,%d) in BFS\n", CurrNode.Y-1, CurrNode.X-1);
}
else if((CurrNode.Y < (nrows-1)) && (CurrNode.X > 0) &&
    (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y+1)] >= 1.0f))
{
  if( verbose > 1)
    mexPrintf("Found already visited node (down+left) (%d,%d) in BFS\n", CurrNode.Y+1, CurrNode.X-1);
  BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y+1)] = 0.5f;
  pnode[nEndInd].X = CurrNode.X-1;
  pnode[nEndInd].Y = CurrNode.Y+1;
  nFullness++;
  nCntNodes++;
  nEndInd++;
  nEndInd = nEndInd % nLength;
}
else if((CurrNode.Y < (nrows-1)) && (CurrNode.X > 0) &&
    (BFSMat[(int)((CurrNode.X-1)*nrows+CurrNode.Y+1)] < 1.0f))
{
  if( verbose > 1)
    mexPrintf("Found already visited node (down+left) (%d,%d) in BFS\n", CurrNode.Y+1, CurrNode.X-1);
}
else if((CurrNode.Y > 0) && (CurrNode.X < (ncols-1)) &&
    (BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y-1)] >= 1.0f))
{
  if( verbose > 1)
    mexPrintf("Found already visited node (up+right) (%d,%d) in BFS\n", CurrNode.Y-1, CurrNode.X+1);
  BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y-1)] = 0.5f;
  pnode[nEndInd].X = CurrNode.X+1;
  pnode[nEndInd].Y = CurrNode.Y-1;
  nFullness++;
  nCntNodes++;
  nEndInd++;
  nEndInd = nEndInd % nLength;
}
else if((CurrNode.Y > 0) && (CurrNode.X < (ncols-1)) &&
if((BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y-1)] < 1.0f))
{
    if(Verbose > 1)
        mexPrintf("Found already visited node (up+right) (%d,%d) in BFS\n", CurrNode.Y-1,CurrNode.X+1);
}
if(Verbose > 1)
    mexPrintf("Evaluating direction down+right\n");
//search diag down and right
if((CurrNode.Y < (nrows-1)) && (CurrNode.X < (ncols-1)) &&
    (BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y+1)] >= 1.0f))
{
    if(Verbose > 1)
        mexPrintf("Marking node (down+right) (%d,%d) in BFS\n", CurrNode.Y+1,CurrNode.X+1);
    BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y+1)] = 0.5f;
    pnode[nEndInd].X = CurrNode.X+1;
    pnode[nEndInd].Y = CurrNode.Y+1;
    nFullness++;
    nCntNodes++;
    nEndInd++;
    nEndInd = nEndInd % nLength;
}
else if((CurrNode.Y < (nrows-1)) && (CurrNode.X < (ncols-1)) &&
    (BFSMat[(int)((CurrNode.X+1)*nrows+CurrNode.Y+1)] < 1.0f))
{
    if(Verbose > 1)
        mexPrintf("Found already visited node (down+right) (%d,%d) in BFS\n", CurrNode.Y+1,CurrNode.X+1);
}
//remove current node from fifo
nFullness--;
nStartInd++;
if(Verbose > 1)
{
    mexPrintf("Completed processing neighbors of pixel (%d, %d)\n", CurrNode.Y, CurrNode.X);
}
if((nCntNodes == 1) && (Verbose > 0))
{
    mexPrintf("BFS (end):Current node value is %.1f\n", BFSMat[(int)((pnode[0].X)*nrows+pnode[0].Y)]);
}
if(Verbose > 1)
    mexPrintf("BFS (end):Current node value is %.1f\n", BFSMat[(int)((pnode[0].X)*nrows+pnode[0].Y)]);
free(pnode);
if (verbose > 1)
    mexPrintf("BFS: Processed %d Nodes\n", nCntNodes);

return;
}

bool CheckIfPerimeter(int Y, int X, double * BFSMat, mwSize ncols, mwSize nrows)
{
    unsigned int PerimRes = 0;
    if ((X==0) || (X == (ncols-1)) || (Y==0) || (Y == (nrows-1)))
    {
        return true;
    }

    //pixel is in middle of image
    //check direction left
    if ((BFSMat[(X-1)*nrows+Y] >= 1.0f))
    {
        PerimRes++;
    }
    //check direction right
    if ((BFSMat[(X+1)*nrows+Y] >= 1.0f))
    {
        PerimRes++;
    }
    //check direction up
    if ((BFSMat[X*nrows+Y-1] >= 1.0f))
    {
        PerimRes++;
    }
    //check direction down
    if ((BFSMat[X*nrows+Y+1] >= 1.0f))
    {
        PerimRes++;
    }

    if(PerimRes == 4)
    {
        return false;
    }
    else
    {
        return true;
    }
}

struct PerimNode
{
    int nXcoor;
    int nYcoor;
    PerimNode *pNext;
};