Telecommun Syst (2013) 52:1491–1501
DOI 10.1007/s11235-011-9626-x

An alternative clustering approach for reconstructing cross cut shredded text documents

Azzam Sleit · Yacoub Massad · Mohammed Musaddaq

Published online: 3 September 2011

Abstract In this paper, we propose a clustering approach for solving the problem of reconstructing cross-cut shredded documents. This problem is important in the field of forensic science. Unlike other clustering approaches which are applied as a preprocessing step before the actual reconstruction algorithms, our clustering approach is part of the reconstruction process itself. We define a new cost function which mainly relies on black pixels to measure the cost of pairing two shreds together. The reconstruction algorithm creates multiple clusters which grow by adding additional shreds based on the cost function. Adding a shred may result in merging two or more clusters to produce a larger cluster. We, also, propose a way to involve the user in the reconstruction process. We compare our approach with a recent proposal and conclude that our approach gives better solutions in less time.

Keywords Cross cut · Shred · Reconstruction · Cost function · Document

1 Introduction and related work

Recent years have witnessed a growing interest in the field of document analysis [5, 8, 9, 12]. One interesting problem in this field is the problem of reconstructing shredded documents, which is very important in the field of forensic science [16]. Documents can be shredded by hand [7] or using a shredding machine [13]. One type of shredding machines shreds the document into strips, in which case the document is called strip-cut document. Another type of machines shreds documents both horizontally and vertically. This type of machines produces cross-cut shredded documents. The problem of reconstructing these documents can be considered as a special case of the jigsaw puzzle [16]. Proposals for solving the jigsaw puzzle can be found in [3, 6, 15]. Given N shreds, each of which is presented as a binary bitmap of size W × H pixels, and assuming that the shreds are placed in the correct orientation, the problem of reconstructing shredded document is to find the correct positioning of these shreds such that they compose the original document. Few researches have worked on solving the problem of reconstructing strip-cut documents. Prandtstetter and Raidl proposed a method that used a specific Variable Neighborhood Search [11] approach to reconstruct documents [13], which involved the user in the construction process to enhance the results. Ukovich et al. proposed an algorithm for the reconstruction of strip-cut shredded documents, paying particular attention to the possibility of using MPEG-7 descriptors [16]. Marques and Freitas used features such as boundary color and utilized the Nearest Neighbor Algorithm to calculate the Euclidean distance between the feature vectors corresponding to the concerned strips. The winner takes all approach was used to decide which strips would fit together [10].

In [14], Prandtstetter and Raidl introduced an algorithm for reconstructing a cross-cut document consisting of N rectangle-shaped equal sized shreds in the correct orientation. The algorithm utilized a cost function to measure the alignment incompatibility error arising due to setting two

A. Sleit (✉)
University of Jordan, P.O. Box 13898, Amman 11942, Jordan
e-mail: azzam.sleit@ju.edu.jo

Y. Massad · M. Musaddaq
University of Jordan, Amman, Jordan

Y. Massad
e-mail: yacoub_m@yahoo.com

M. Musaddaq
e-mail: sarurim@yahoo.com

specific shreds next to each other. The cost function counts the number of pixels not having the same values on both borders of the two shreds.[1] A smaller cost means better possibility that the alignment is correct. It is worth noting that in this case the cost of aligning two shreds with white borders is 0, while most correct alignments will have cost > 0. This is because the cost function treats white and black pixels similarly. An initial solution can be obtained in polynomial time, which is followed by a search method such as Variable Neighborhood Search [11] or Ant Colony Optimization [4] to enhance the initial solution.

This article proposes a new clustering approach for reconstructing cross-cut shredded text documents. Unlike other clustering approaches used as a preprocessing step, such as in [17], our clustering approach is part of the actual reconstruction process. Our algorithm utilizes a cost function that only considers black pixels; i.e., when two shreds have matching white pixels on their edges, the algorithm does not consider this as a sign that these shreds should be paired together. We argue that black pixels carry more meaningful information in comparison with white pixels. Given N shreds in the correct orientation, we assume that a preprocessing algorithm such as the ones in [1, 2] runs before our algorithm starts. Each shred is presented as a binary bitmap of size $W \times H$ pixels. The problem is to find the correct positioning of these shreds such that they compose the original document. A human is the final judge for the correctness of the solution.

Section 2 describes our proposed cost functions. Next, Sect. 3 proposes a reconstruction algorithm and gives an example of the reconstruction process. Then, Sect. 4 shows the results of the experiments conducted to evaluate the performance of our proposed algorithm and to compare it with the algorithm in [14]. Section 5 describes the process of user interaction that we implemented to obtain enhanced results and Sect. 6 summarizes the article.

2 Proposed cost functions

This section proposes two cost functions; namely, CostX(A, B) and CostY(A, B). CostX(A, B) returns the error obtained due to placing shred B right next to shred A, whereas CostY(A, B) returns the error obtained due to placing shred A on top of shred B. Both cost functions consider black pixels only. To show why this is important, assume that the right border of shred A is white except for some black pixels on the top and the left border of shred B is white except for some pixels on the bottom. In this example, it is most probably that shred B should not be aligned on the right of shred A, but considering white pixels may erroneously consider this alignment to be valid.

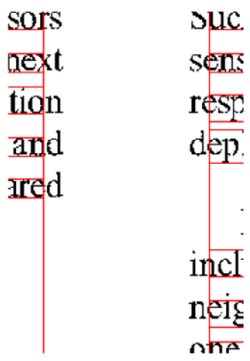


Fig. 1 The meaning of RightLines and LeftLines functions in a sample shred

For each shred A, we define:

– RightLines(A) = the array of text lines that appear in the t% right part of the shred.
– LeftLines(A) = the array of text lines that appear in the t% left part of the shred.

The threshold value t is determined experimentally. Figure 1 shows a sample shred. When t is chosen to be 15, the vertical red lines show the 15% left part and the 15% right part of the shred. For the left part, there are 5 text lines while for the right part there are 7 text lines. The horizontal red lines show the start and end of text lines.

RightLines and LeftLines can be calculated in many ways. We start by scanning the first row of the shred for black pixels. When such pixels are found, we mark this row as the start of a text line in the shred. We continue to the next row until we find a row with no black pixels. When we reach such a row, we mark this row as the end of the text line. Then, we search for the next row with black pixels to mark it as a start of a new line. The process is repeated until the last row has been reached. In order to avoid considering noise or a small number of pixels as a line, we ignore lines with height less than some threshold value. However, in case that the first line starts at row 0 or that the last line ends at row $H - 1$, we accept first and last lines to be of any height. While this threshold value may vary, a heuristic algorithm can easily determine the average height of text lines in a shred, and this threshold value is a percentage of the height of text lines. By experimenting, we found that this threshold should be 25% of the average line height.

Before we define the horizontal cost function CostX, we need to define some functions that are utilized by CostX function. The NumberOfCompatibileLinesX(A, B) function (defined in Algorithm 1) counts the number of compatible lines between the right part of A and the left part of B. A line in A is considered to be compatible with a line in B if they lie approximately at the same vertical position. RightLines(A) is the array of lines in the right side of A. Each element of this array has a property named Location which specifies the vertical position of the line (as per Fig. 2). The vertical

[1]They actually use a slightly different cost function, see [13] for more information.

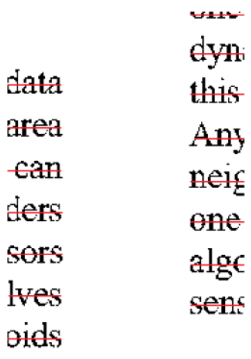


Fig. 2 The meaning of the Location property in RightLines and LeftLines arrays in a sample shred, each red line corresponds to the Location property of each text line

Algorithm 1 NumberOfCompatibileLinesX(A, B)

```
$i \leftarrow 0$
$j \leftarrow 0$
$numLines \leftarrow 0$
while $i < RightLines(A).Length$
and $j < LeftLines(B).Length$ do
  $posDifference \leftarrow |RightLines(A)(i).Location - LeftLines(B)(j).Location|$
  if $posDifference < Pixelth$ then
    $numLines \leftarrow numLines + 1$
    $i \leftarrow i + 1$
    $j \leftarrow j + 1$
  else
    if $RightLines(A)(i).Location > LeftLines(B)(j).Location$ then
      $j \leftarrow j + 1$
    else
      $i \leftarrow i + 1$
    end if
  end if
end while
return $numLines$
```

position of a line is the center of gravity of black pixels between the vertical position of the beginning of the line and the vertical position of the end of the line. We assume that this array is sorted according to the vertical position in an increasing manner. LeftLines(B) is defined in the same way with respect to the left side of the shred.

Experimentally, we found that *Pixelth* should be of value 33% of the average text line height, which may differ depending on the font sizes. A simple heuristic function can determine the average line height and hence determine this value.

Next, we define an important function LinesCompatibilityX(A , B) that returns whether the right lines of A are compatible with the left lines of B (Algorithm 2).

In our implementation for the LinesCompatibilityX algorithm, we use $Lines_{th}$ of value 1. The following PixelCompatibilityX(A, B, y) Algorithm 3 determines the compatibil-

Algorithm 2 LinesCompatibilityX(A, B)

```
if $RightLines(A).Length = 0$ or
$LeftLines(B).Length = 0$ then
  return $false$
end if
$linesDifference \leftarrow |RightLines(A).Length - LeftLines(B).Length|$
if $linesDifference > Linesth$ then
  return $false$
end if
$numCompatibileLines \leftarrow NumberOfCompatibileLinesX(A, B)$
if $numCompatibileLines = 0$ then
  return $false$
end if
return $true$
```

Algorithm 3 PixelCompatibilityX(A, B, y)

```
return $PixelCompatibilityXLeftSide(A, B, y) + PixelCompatibilityXRightSide(A, B, y)$
```

ity between the pixels at height y in the right border of A and the left border of B.

The pixels of the right border of shred A can be obtained by the function RightBorderPixel(A, y) where y is the row number which may have a value from 0 to H − 1. Likewise, the pixels of the left border of shred B can be accessed via the function LeftBorderPixel(B, y). The weights of the pixels that we use (0.75, 1.5, 4 and 5) are determined experimentally. The PixelCompatibilityXLeftSide (Algorithm 4) and PixelCompatibilityXRightSide (Algorithm 5) functions are outlined as follows. Please note also that these functions are considered as an extension of the cost function [14].

The AllPixelsCompatibilityX(A, B) Algorithm 6 determines the overall compatibility between the border pixels of shred A and B.

The following CostX(A, B) function (Algorithm 7) is the cost function which characterizes the cost of horizontally aligning shred A next to shred B. CostX utilizes the two functions BlackPixelsRight(A) and BlackPixelsLeft(B) which return the number of black pixels on the right border of A and the number of black pixels on the left border of B, respectively. If any shred has less black pixels on its border than *Blackth*, then the information obtained by this border is unreliable and thus we need to ignore it. In our implementation, we use *Blackth* of value 2.

The CostY(A, B) function (Algorithm 8) characterizes the cost of aligning shred A on top of shred B. It utilizes the BlackPixelsBottom(A) and BlackPixelsTop(B) functions which return the number of black pixels in row H − 1 in shred A and the number of black pixels in row 0 in shred B, respectively.

Algorithm 4 PixelCompatibilityXLeftSide(A, B, y)

```
if $RightBorderPixel(A, y) = White$ then
    return 0
end if
$Sum \leftarrow 0$
if $LeftBorderPixel(B, y-1) = Black$ then
    $Sum \leftarrow Sum + 1.5$
else
    $Sum \leftarrow Sum - 0.75$
end if
if $LeftBorderPixel(B, y) = Black$ then
    $Sum \leftarrow Sum + 4$
else
    $Sum \leftarrow Sum - 5$
end if
if $LeftBorderPixel(B, y+1) = Black$ then
    $Sum \leftarrow Sum + 1.5$
else
    $Sum \leftarrow Sum - 0.75$
end if
return $Sum$
```

Algorithm 5 PixelCompatibilityXRightSide(A, B, y)

```
if $LeftBorderPixel(B, y) = White$ then
    return 0
end if
$Sum \leftarrow 0$
if $RightBorderPixel(A, y-1) = Black$ then
    $Sum \leftarrow Sum + 1.5$
else
    $Sum \leftarrow Sum - 0.75$
end if
if $RightBorderPixel(A, y) = Black$ then
    $Sum \leftarrow Sum + 4$
else
    $Sum \leftarrow Sum - 5$
end if
if $RightBorderPixel(A, y+1) = Black$ then
    $Sum \leftarrow Sum + 1.5$
else
    $Sum \leftarrow Sum - 0.75$
end if
return $Sum$
```

Algorithm 6 AllPixelsCompatibilityX(A, B)

```
return $\sum_{y=1}^{H-2} PixelCompatibilityX(A, B, y)$
```

The AllPixelsCompatibilityY(A, B) function is the same as AllPixelsCompatibilityX(A, B), except that AllPixelsCompatibilityY(A, B) considers the last row of A and the first row of B. Please note that both functions return -1**x** their respective pixel compatibility function because cost is inversely proportional to compatibility.

Algorithm 7 CostX(A, B)

```
if $BlackPixelsRight(A) < Blackth$ or
$BlackPixelsLeft(B) < Blackth$ then
    return $\infty$
end if
if $LinesCompatibilityX(A, B)$ then
    return $-1$x $AllPixelsCompatibilityX(A, B)$
else
    return $\infty$
end if
```

Algorithm 8 CostY(A, B)

```
if $BlackPixelsBottom(A) < Blackth$ or
$BlackPixelsTop(B) < Blackth$ then
    return $\infty$
end if
return $-1\times AllPixelsCompatibilityY(A, B)$
```

3 The reconstruction algorithm

The reconstruction algorithm starts by creating an array called the CostArray of 2×N ×(N −1) items. Each item has four fields: Shred1, Shred2, Direction and Cost. Shred1 and Shred2 can have values from 0 to N −1 and Direction can have one of two values: Vertical or Horizontal. The fourth field Cost captures the cost of aligning Shred1 and Shred2 according to the value stored in the Direction field. The CostArray represents all possible pairs of shreds on different directions and the cost of such pairing. This array is created by the following CreateCostArray Algorithm 9. The CostArray array is sorted after it is created.

The SolutionStructure is a data structure that holds the solution or part of the solution. It is a 2-dimensional repository of ShredNode objects. A ShredNode object holds information about a single shred. Each ShredNode can be accessed by the position which is relative to the position of the first shred inserted in the structure. For example, if the structure is initialized with shred A, then shred A will have position (0, 0). If shred B is inserted to the right of shred A, then shred B will have position (1, 0). If shred C is inserted to the left of shred A then shred C will have position (−1, 0), as per Fig. 3. Each ShredNode maintains the number(index) of the shred it represents and its position. This information is captured by the fields ShredNode.ShredIndex, ShredNode.xPos and ShredNode.yPos. The SolutionStructure maintains four important fields which are MinimumX, MaximumX, MinimumY and MaximumY that define the borders of the solution.

Algorithm 9 CreateCostArray

```
$arrayIndex \leftarrow 0$
for $i = 0$ to $N-1$ do
  for $j = 0$ to $N-1$ do
    if $i \neq j$ then
      $CostArray[arrayIndex] \leftarrow (i, j, Horizontal, CostX(i, j))$
      $arrayIndex \leftarrow arrayIndex + 1$
      $CostArray[arrayIndex] \leftarrow (i, j, Vertical, CostY(i, j))$
      $arrayIndex \leftarrow arrayIndex + 1$
    end if
  end for
end for
sort array $CostArray$ by $Cost$ in an increasing manner
return $CostArray$
```

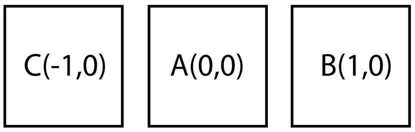


Fig. 3 The shred indexing method

The InsertShredNode (SolStruct, ShredIndex, xPos, yPos) function inserts a shred into the solution at the specified position. The MergeTwoSolutionStructures (SolStruct1, SolStruct2, xPos1, yPos1, xPos2, yPos2) (Algorithm 10) takes two solution structures and merges them together. This function adds the contents of SolStruct2 into SolStruct1 and deletes SolStruct2, the function places the nodes of SolStruct2 into SolStruct1 such that their new relative position to (xPos1, yPos1) is the same as their original relative position to (xPos2, yPos2). This function fails if it tries to place a shred from SolStruct2 in a non-empty position of SolStruct1; i.e., if the merging process results in having two shreds in the same position in SolStruct1, the whole merging process will be canceled. The MergeTwoSolutionStructures function utilizes the InsertShredNode function as well as the FindShredNodeByShredIndex (SolStruct, ShredIndex) and FindShredNodeByPosition (SolStruct, xPos, yPos) functions which retrieve a ShredNode by index or relative position, respectively.

Now, we are ready to define the reconstruction algorithm ReconstructShreds. For simplicity, we use MTSS as an abbreviation for MergeTwoSolutionStructures.

The ReconstructShreds Algorithm 11 is responsible for discovering the positions of the N shreds using the CostArray array. Each time it reads an item from CostArray, it performs an action depending on whether the shreds in this item have already been placed into some cluster. Clusters are of type SolutionStructure. When the algorithm retrieves the first item in CostArray array (which is the pair of shreds that have the smallest alignment cost), it creates a new cluster and inserts the two paired shreds into this cluster (Algorithm 12). On the second read, the array returns the pair of shreds with the next least cost. If none of the shreds of this item are already placed in a previous cluster, then the algorithm will create a new cluster for the two shreds, see the CreateNewCluster (Algorithm 13) sub-function. If only one of these shreds is already placed in any of the existing clusters, the algorithm inserts the other shred in that cluster in the appropriate position unless that position is already occupied, in which case the algorithm skips this item, see the InsertShred2IntoCluster1 (Algorithm 14) and the InsertShred1IntoCluster2 (Algorithm 15) sub-functions. If both shreds are already placed in two different clusters, the reconstruction algorithm tries to merge the two clusters

Algorithm 10 MergeTwoSolutionStructures (SolStruct1, SolStruct2, xPos1, yPos1, xPos2, yPos2)

```
for $i = SolStruct2.MinimumX$ to $SolStruct2.MaximumX$ do
  for j = $SolStruct2.MinimumY$ to $SolStruct2.MaximumY$ do
    $sNode \leftarrow FindShredNodeByPosition(SolStruct2, i, j)$
    if $sNode \neq NULL$ then
      $newPosX \leftarrow i - xPos2 + xPos1$
      $newPosY \leftarrow j - yPos2 + yPos1$
      $dNode \leftarrow FindShredNodeByPosition(SolStruct1, newPosX, newPosY)$
      if $dNode \neq NULL$ then
        return $false$
      end if
    end if
  end for
end for
for $i = SolStruct2.MinimumX$ to $SolStruct2.MaximumX$ do
  for $j = SolStruct2.MinimumY$ to $SolStruct2.MaximumY$ do
    $sNode \leftarrow FindShredNodeByPosition(SolStruct2, i, j)$
    if $sNode \neq NULL$ then
      $newPosX \leftarrow i - xPos2 + xPos1$
      $newPosY \leftarrow j - yPos2 + yPos1$
      $InsertShredNode(SolStruct1, sNode.ShredIndex, newPosX, newPosY)$
    end if
  end for
end for
remove all nodes from $SolStruct2$
return $true$
```

Algorithm 11 ReconstructShreds

```
The following variables (up to numClusters) are
global variables
$numberOfShredsPut \leftarrow 0$
for $index = 0$ to $N - 1$ do
  $shredPut[index] \leftarrow false$
end for
$CostArray \leftarrow CreateCostArray()$
$CostArrayIndex \leftarrow 0$
$Clusters \leftarrow \varphi$
$numClusters \leftarrow 0$
while $numberOfShredsPut < N$ do
  $S1 \leftarrow CostArray[CostArrayIndex].Shred1$
  $S2 \leftarrow CostArray[CostArrayIndex].Shred2$
  $shredsDirection \leftarrow$
  $CostArray[CostArrayIndex].Direction$
  if $shredPut[S1]$ and $shredPut[S2]$ then
    $ReconstructShreds.MergeClusters$
    $(shredsDirection, S1, S2)$
  else
    if not $shredPut[S1]$ and not $shredPut[S2]$ then
      $ReconstructShreds.CreateNewCluster$
      $(shredsDirection, S1, S2)$
    else
      if $shredPut[S1]$ then
        $ReconstructShreds.InsertShred2IntoCluster1$
        $(shredsDirection, S1, S2)$
      else
        $ReconstructShreds.InsertShred1IntoCluster2$
        $(shredsDirection, S1, S2)$
      end if
    end if
  end if
$CostArrayIndex \leftarrow CostArrayIndex + 1$
end while
```

using the position of these two shreds and the direction specified by CostArray to select the merge positions of these clusters, see the MergeClusters sub-function. The algorithm stops when all shreds are placed into the solution. The ReconstructShreds algorithm is detailed as follows.

In order to explain the ReconstructShreds algorithm, we provide an example for reconstructing 8 shreds using the computed and sorted CostArray in Table 1.

The reconstruction algorithm starts with the first item. It checks it to see that both shred 0 and shred 1 have not yet been placed into any clusters. The algorithm creates a new cluster for them as shown in Fig. 4.

Inspecting the second item of CostArray, the algorithm finds that both shred 10 and 11 have not yet been placed into a cluster. Therefore, it creates a new cluster for them as shown in Fig. 5.

Algorithm 12 ReconstructShreds.MergeClusters (shredsDirection, S1, S2)

```
$C1 \leftarrow ShredCluster[S1]$
$C2 \leftarrow ShredCluster[S2]$
if $C1 \neq C2$ then
  $S1Node \leftarrow$
  $FindShredNodeByShredIndex(Clusters[C1], S1)$
  $S2Node \leftarrow$
  $FindShredNodeByShredIndex(Clusters[C2], S2)$
  $xPos2 \leftarrow S2Node.xPos$
  $yPos2 \leftarrow S2Node.yPos$
  if $shredsDirection = Horizontal$ then
    $xPos1 \leftarrow S1Node.xPos + 1$
    $yPos1 \leftarrow S1Node.yPos$
    if $MTSS(Clusters[C1], Clusters[C2], xPos1,$
    $yPos1, xPos2, yPos2)$ then
      for each shreds s in $Clusters$[C2] do
        $ShredCluster[s] \leftarrow C1$
      end for
    end if
  else
  $xPos1 \leftarrow S1Node.xPos$
  $yPos1 \leftarrow S1Node.yPos + 1$
  if $MTSS(Clusters[C1], Clusters[C2], xPos1,$
  $yPos1, xPos2, yPos2)$ then
  for each shreds s in $Clusters$[C2] do
        $ShredCluster[s] \leftarrow C1$
      end for
    end if
  end if
end if
```

Algorithm 13 ReconstructShreds.CreateNewCluster (shredsDirection, S1, S2)

```
$Clusters[numClusters] \leftarrow$ new $SolutionStructure$
$InsertShredNode(Clusters[numClusters], S1, 0, 0)$
$ShredCluster[S1] \leftarrow numClusters$
if $shredsDirection = Horizontal$ then
  $InsertShredNode(Clusters[numClusters], S2, 1, 0)$
else
  $InsertShredNode(Clusters[numClusters], S2, 0, 1)$
end if
$shredPut[S1] \leftarrow true$
$shredPut[S2] \leftarrow true$
$ShredCluster[S2] \leftarrow numClusters$
$numClusters \leftarrow numClusters + 1$
$numberOfShredsPut \leftarrow numberOfShredsPut + 2$
```

Then, the algorithm inspects the third item in the array to find that shred 1 is already placed in cluster 1 but shred 2 has not yet been placed into any cluster. Consequently, it

Algorithm 14 ReconstructShreds.InsertShred2IntoCluster1 (shredsDirection, S1, S2)

```
$C1 \leftarrow ShredCluster[S1]$
$S1Node \leftarrow FindShredNodeByShredIndex(Clusters[C1], S1)$
if $shredsDirection = Horizontal$ then
    $xPos2 \leftarrow S1Node.xPos + 1$
    $yPos2 \leftarrow S1Node.yPos$
else
    $xPos2 \leftarrow S1Node.xPos$
    $yPos2 \leftarrow S1Node.yPos + 1$
end if
$SomeNode \leftarrow FindShredNodeByPosition(Clusters[C1], xPos2, yPos2)$
if $SomeNode = NULL$ then
    $InsertShredNode(Clusters[C1], S2, xPos2, yPos2)$
    $ShredCluster[S2] \leftarrow C1$
    $numberOfShredsPut \leftarrow numberOfShredsPut + 1$
    $shredPut[S2] \leftarrow true$
end if
```

Algorithm 15 ReconstructShreds.InsertShred1IntoCluster2 (shredsDirection, S1, S2)

```
$C2 \leftarrow ShredCluster[S2]$
$S2Node \leftarrow FindShredNodeByShredIndex(Clusters[C2], S2)$
if $shredsDirection = Horizontal$ then
$xPos1 \leftarrow S2Node.xPos - 1$
$yPos1 \leftarrow S2Node.yPos$
else
$xPos1 \leftarrow S2Node.xPos$
$yPos1 \leftarrow S2Node.yPos - 1$
end if
$SomeNode \leftarrow FindShredNodeByPosition(Clusters[C2], xPos1, yPos1)$
if $SomeNode = NULL$ then
$InsertShredNode(Clusters[C2], S1, xPos1, yPos1)$
$ShredCluster[S1] \leftarrow C2$
$numberOfShredsPut \leftarrow numberOfShredsPut + 1$
$shredPut[S1] \leftarrow true$
end if
```

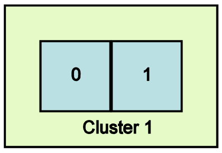


Fig. 4 Cluster 1 is created after processing the first item of the example CostArray

adds shred 2 into cluster 1 right next to shred 1 preserving the direction specified in the CostArray as shown in Fig. 6.

Inspecting the fourth item of the CostArray, the algorithm finds that both shred 1 and shred 10 have been assigned to two different clusters. The algorithm tries to merge the two clusters. However, the algorithm finds that merging the two clusters will make two shreds (shred 10 and shred 2) have the same position in cluster 1. Therefore, the algorithm cancels the merge operation and the clusters keep the original state as per Fig. 7.

Inspecting the fifth item of the CostArray, the algorithm finds that both shred 2 and shred 11 have been assigned to two different clusters. The algorithm uses this information to merge the two clusters as shown in Fig. 8.

Then, the algorithm will inspect the sixth item in the array to find that both shred 18 and shred 19 have not been placed yet into any clusters. The algorithm creates a new cluster for these two shreds as shown in Fig. 9.

Table 1 The CostArray array used in the example

Shred1	Shred2	Direction	Cost
0	1	Horizontal	–300
10	11	Horizontal	–290
1	2	Horizontal	–280
1	10	Horizontal	–277
2	11	Vertical	–275
18	19	Horizontal	–270
19	28	Vertical	–260
10	19	Vertical	–250
...	...	...	...
...	...	...	...

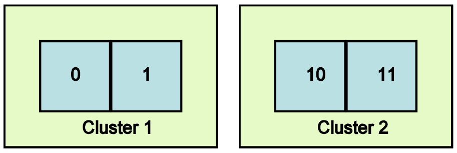


Fig. 5 Cluster 2 is created after processing the second item of the example CostArray

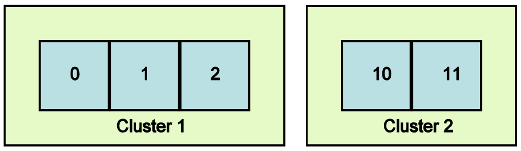


Fig. 6 Processing the third item of the example CostArray

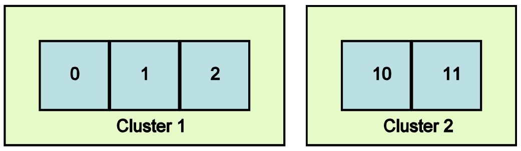


Fig. 7 Processing the fourth item of the example CostArray

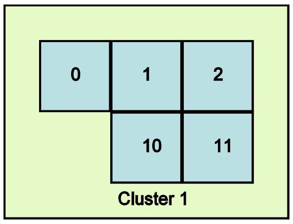


Fig. 8 Merging cluster 1 and 2 due to processing the fifth item of the example CostArray

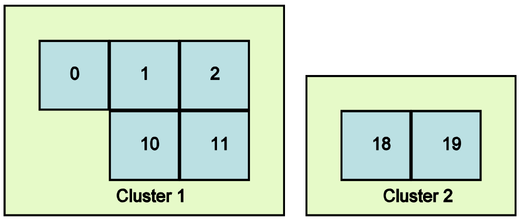


Fig. 9 Cluster 2 is created due to processing the sixth item of the example CostArray

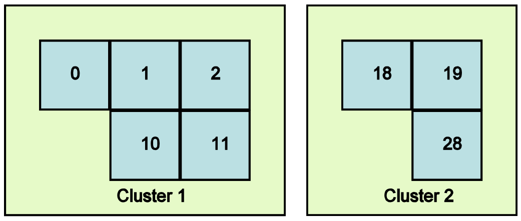


Fig. 10 Processing the seventh item of the example CostArray

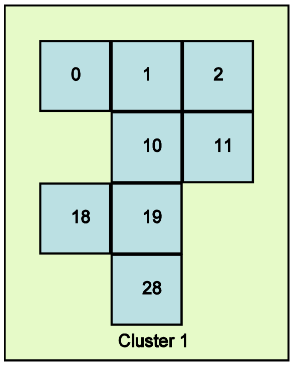


Fig. 11 Merging cluster 1 and 2 due to processing the eighth item of the example CostArray

Inspecting the seventh item in the array, the algorithm finds that shred 19 is already assigned to cluster 2 but shred 28 has not yet been assigned to any cluster. The algorithm adds shred 28 into cluster 2 right under shred 19 as shown in Fig. 10.

Finally, the algorithm inspects the eighth item of the CostArray to find that both shred 10 and shred 19 have been assigned to two different clusters. The algorithm will use this information to merge the two clusters as shown in Fig. 11.

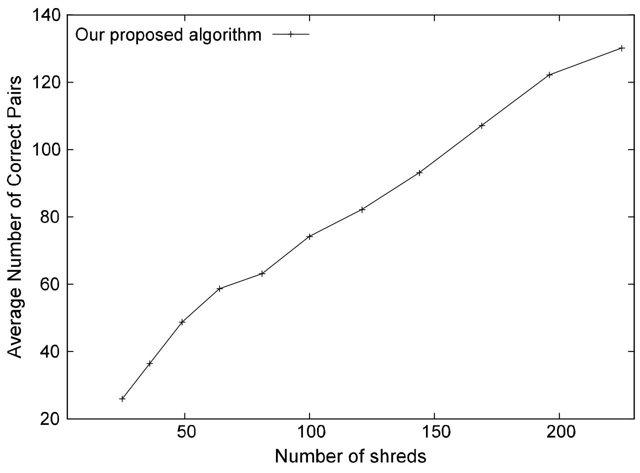


Fig. 12 Average number of correct pairs for different documents and different sizes

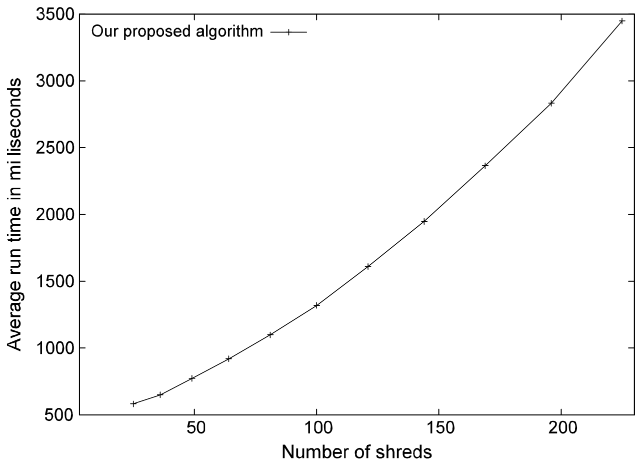


Fig. 13 Average run time for different documents and different sizes

4 Experimental results

In order to evaluate the performance of the proposed algorithm, we considered two main performance metrics; namely, the number of generated correct pairs and run time requirement. We used a single core of a Core2 duo 2.00 GHz with 3 Gigabytes of RAM. Please note that a document cut into $Y \times Y$ shreds has $2 \times Y \times (Y - 1)$ correct pairs; $Y \times (Y - 1)$ horizontal pairs and $Y \times (Y - 1)$ vertical pair.

We considered a set of 100 documents shredded into 5×5, 6×6, 7×7, 8×8, 9×9, 10×10, 11×11, 12×12, 13×13, 14×14, and 15×15 shreds by the computer. Figures 12 and 13 characterize the performance of the proposed reconstruction algorithm in terms of the average number of correct pairs and average time needed to execute the algorithm, respectively. The displayed run times include reading the shred bitmaps from disk, calculating the CostArray array and running the reconstruction algorithm.

Table 2 A comparison between our algorithm and the one in [14]

# Shreds	Correct Pairs		Time in milliseconds	
	Our proposal	[14]	Our proposal	[14]
9×9	63.08 (δ : 26.48)	54.78	1097.92	62022.25
12×12	93.03 (δ : 27.42)	66.03	1948.30	280782.58

Table 2 compares our algorithm and the algorithm proposed in [14] with respect to another dataset of 4 documents shredded into 9×9 and 12×12 shreds, where the average number of correct pairs and run times were obtained for both algorithms. The algorithm in [14] was run using our dataset by the authors of [14] upon our request. The table shows that our proposed algorithm is superior in terms of the average number of obtained correct pairs and runtime. The massive difference in run times between the two algorithms is attributed to the fact that although the algorithm in [14] finds an initial solution in a reasonable time, it goes through several iterations using meta-heuristic optimization algorithms to find a better solution which ends up requiring massive run time.

5 User interactions

In this section, we describe our implementation for user interaction with the reconstruction algorithm. The idea of user interactivity has been used to enhance the results of many solutions to problems related to documents and image processing [13, 18–20]. We run the reconstruction algorithm to a point where the current item being read from CostArray has cost ∞. At this point there will be several clusters. Each cluster will be shown to the user, who will mark the correct pairs with green and incorrect pairs with red. He/she will also mark pairs that he/she is not sure about with blue. After the user gives his/her feedback, the CostArray array will be modified to set the cost of the green pairs to $-\infty$ and the cost of the red pairs to ∞. Then, the algorithm will start again with the new CostArray array.

The process continues until the user is satisfied with the results. We also allow the user to pair two shreds together by selecting the borders of the two shreds that he/she thinks should be paired.

Figure 14 shows a window that is given to the user, which displays one cluster. The user uses the mouse and keyboard to mark the areas between the shreds with green, red or blue. The user can also move around the cluster since sometimes the size of the window is smaller than the size of the cluster.

Figure 15 shows another cluster shown to the user, who has marked some pairs with red and some with green. Figure 16 shows the same cluster but in the second iteration (i.e. after the CostArray array was updated and the reconstruction algorithm ran again using the new CostArray array).

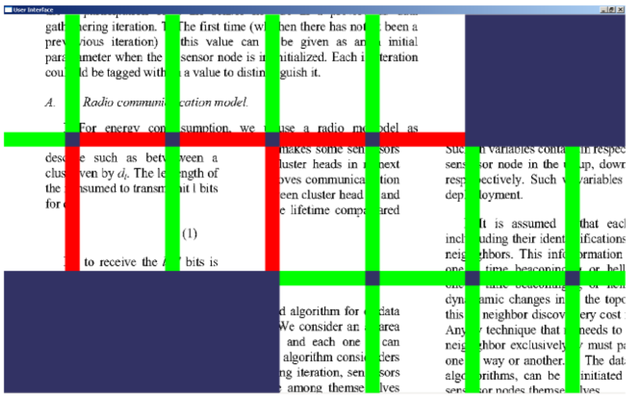


Fig. 14 A part of a cluster as given to the user, the user has marked some pairs with *green* and some with *red*

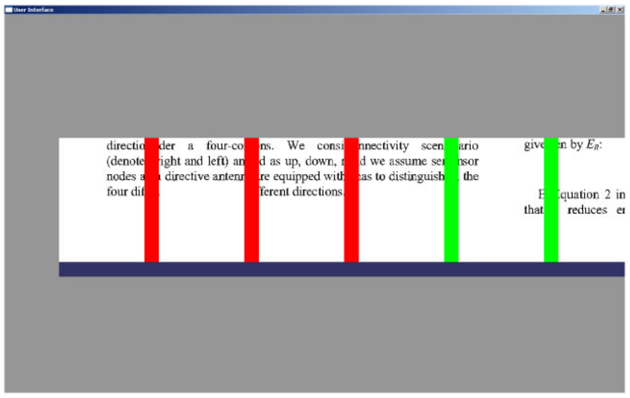


Fig. 15 A part of a cluster as given to the user, the user has marked some pairs with *green* and some with *red*

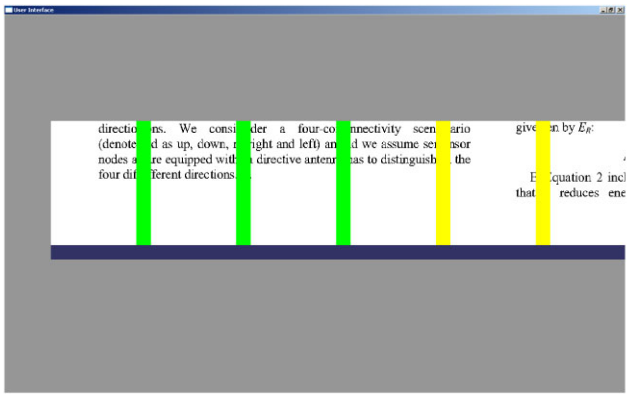
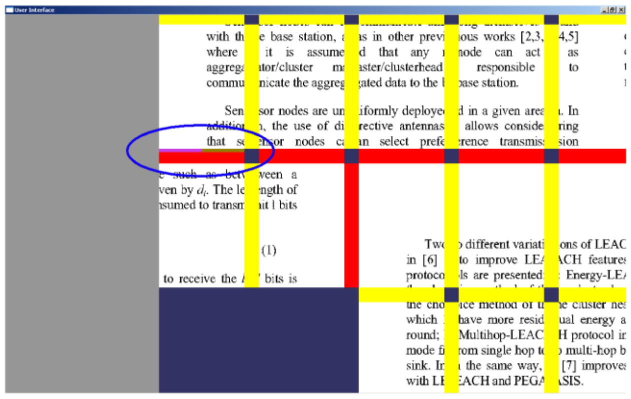


Fig. 16 A part of the same cluster shown in Fig. 15 as given to the user, the *yellow color* means that the user has already marked these pairs *green* in a previous iteration

The yellow pairs indicate that the user has already marked these pairs with green in a previous iteration. Note that the second and third shreds from the left have been swapped in the second iteration as a result of the feedback given by the user.


with th e base station, s in other previ ous works [2,3, 4,5]
where it is assume d that any node can act as
aggreg ator/cluster m aster/clusterhead responsible to
commu nicate the aggreg ated data to the base station.

Sen sor nodes are un iformly deploye d in a given area . In
additio n, the use of di rective antennas allows conside ring
that s ensor nodes ca n select prefe rence transmis sion

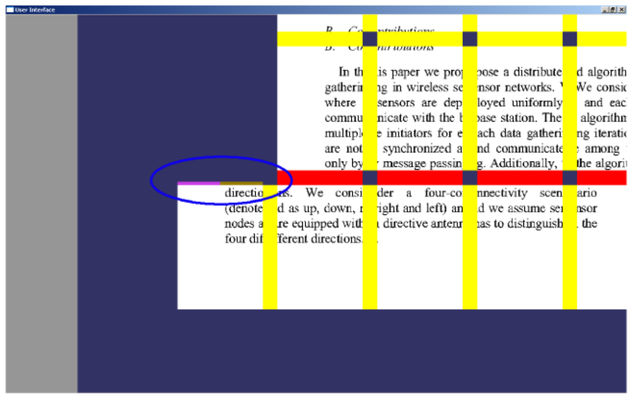
such as betw een a
ven by d_i. The le ngth of
ssumed to transm it l bits

(1)

to receive the l bits is

Two different variati ons of LEACH
in [6] to improve LE ACH features
protoco ls are presented: Energy-LE
the cho ice method of th e cluster hea
which have more resid ual energy a
round; Multihop-LEAC H protocol in
mode fr om single hop to multi-hop b
sink. In the same way, [7] improves
with LE ACH and PEGA SIS.

Fig. 17 A part of a cluster as given to the user, the user wants to pair two shreds together, in this figure the user marked the lower border of one shred


B. Co ntributions

In th is paper we pro pose a distribute d algorith
gatheri ng in wireless se nsor networks. We consid
where sensors are dep loyed uniformly and eac
commu nicate with the b ase station. The algorithm
multipl e initiators for e ach data gatheri ng iteratio
are not synchronized a nd communicate among
only by message passin g. Additionally, the algori

directio ns. We consi der a four-co nnectivity scen ario
(denote d as up, down, r ight and left) an d we assume se nsor
nodes a re equipped wit h directive antenn as to distinguish the
four dif ferent directions.

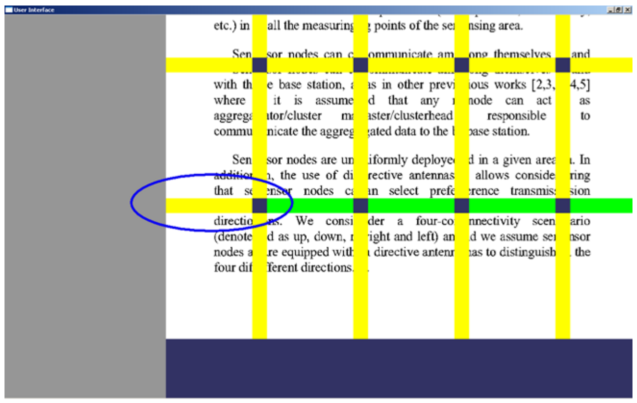
Fig. 18 A part of the same cluster in Fig. 17 in the same iteration, the user used the mouse and keyboard to navigate to this part of the cluster, in this figure the user marked the upper border of the shred that he thinks should be put under the shred marked in Fig. 17

Figures 17 and 18 show two different parts of the same cluster in the same iteration. The user wishes to pair the shreds marked in these figures together. He/She can achieve this objective by marking the corresponding borders of the two shreds.

Figure 19 shows the same cluster in Figs. 17 and 18 in the next iteration. The two shreds were paired together as a result of the user feedback. Please note that the borders of these two shreds are white. There is no simple and automatic method to figure out that these two shreds should be paired together. The user have read the readable parts of the document to decide that these shreds should be paired.

6 Conclusions

In this paper, we proposed a new clustering approach for reconstructing crosscut shredded documents. The algorithm applies clustering as part of the reconstruction process and not as a preprocessing step. We have demonstrated the superiority of our algorithm by comparing it with another algorithm [14]. During our research, we observed that the cost function has the most influential impact on the reconstruction process. As a future research avenue, it will be recommended to further research enhancements for the proposed cost function in order to increase the percentage of correct pairs. Also, we would like to test the algorithm using real data; i.e., using documents shredded by shredding machines.


etc.) in all the measuring points of the se nsing area.

Sen sor nodes can c ommunicate am ong themselves and
with th e base station, s in other previ ous works [2,3, 4,5]
where it is assume d that any node can act as
aggreg ator/cluster m aster/clusterhead responsible to
commu nicate the aggreg ated data to the base station.

Sen sor nodes are un iformly deploye d in a given area . In
additio n, the use of di rective antennas allows conside ring
that s ensor nodes ca n select prefe rence transmis sion
directio ns. We consi der a four-co nnectivity scen ario
(denote d as up, down, r ight and left) an d we assume se nsor
nodes a re equipped wit h directive antenn as to distinguish the
four dif ferent directions.

Fig. 19 A part of the same cluster in Figs. 17 and 18 but in the next iteration, the two shreds were paired together as a result of the user feedback

Acknowledgements The authors of this article would like to thank Dr. Matthias Prandtstetter who is a co-author of [14] for running their algorithm on our data set and providing us with the numbers which we used in our experimental analysis. Furthermore, the authors thank the editors of this special issue and the anonymous reviewers for their valuable comments which enhanced the quality of this article.

References

1. Aradhye, H. B. (2005). A generic method for determining up/down orientation of text in roman and non-roman scripts. *Pattern Recognition*, *38*(11), 2114–2131.
2. Bose, P. & Kilani Ghoudi, J.D.C. (1998). Detection of text-line orientation. In *Canadian conference on computational geometry*.
3. Chung, M. G., Fleck, M., & Forsyth, D. (1998). Jigsaw puzzle solver using shape and color. In *ICSP '98. Fourth international conference on signal processing proceedings* (Vol. 2, pp. 877–880).
4. Dorigo, M., & Blum, C. (2005). Ant colony optimization theory: a survey. *Theoretical Computer Science*, *344*(2–3), 243–278.
5. Faure, C., & Vincent, N. (2007). Document image analysis for active reading. In *SADPI '07: proceedings of the 2007 international workshop on semantically aware document processing and indexing* (pp. 7–14). New York: ACM Press.
6. Goldberg, D., Malon, C., & Bern, M. (2004). A global approach to automatic solution of jigsaw puzzles. *Computational Geometry*, *28*(2–3), 165–174.
7. Justino, E., Oliveira, L. S., & Freitas, C. (2006). Reconstructing shredded documents through feature matching. *Forensic Science International*, *160*(2–3), 140–147.
8. Likforman-Sulem, L., Zahour, A., & Taconet, B. (2007). Text line segmentation of historical documents: a survey. *International Journal on Document Analysis and Recognition*, *9*(2), 123–138.
9. Lu, X., Kataria, S., Brouwer, W. J., Wang, J. Z., Mitra, P., & Giles, C. L. (2009). Automated analysis of images in documents for

intelligent document search. *International Journal on Document Analysis and Recognition*, *12*(2), 65–81.

10. Marques, M. A. O., & Freitas, C. O. A. (2009). Reconstructing strip-shredded documents using color as feature matching. In *SAC '09: Proceedings of the 2009 ACM symposium on applied computing* (pp. 893–894). New York: ACM Press.
11. Mladenoviá, N., & Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, *24*(11), 1097–1100.
12. Ogier, J. M., & Tombre, K. (2006). Madonne: document image analysis techniques for cultural heritage documents. In *International conference on digital cultural heritage*, Vienna, Austria.
13. Prandtstetter, M., & Raidl, G. R. (2008). Combining forces to reconstruct strip shredded text documents. In *HM '08: proceedings of the 5th international workshop on hybrid metaheuristics* (pp. 175–189). Berlin: Springer.
14. Prandtstetter, M., & Raidl, G. R. (2009). Meta-heuristics for reconstructing cross cut shredded text documents. In *GECCO '09: proceedings of the 11th annual conference on genetic and evolutionary computation* (pp. 349–356). New York: ACM Press.
15. Tybon, R., & Kerr, D. (2009). Automated solutions to incomplete jigsaw puzzles. *Artificial Intelligence Review*, *32*(1–4), 77–99.
16. Ukovich, A., Ramponi, G., Doulaverakis, H., Kompatsiaris, Y., & Strintzis, M. (2004). Shredded document reconstruction using mpeg-7 standard descriptors. In *Signal processing and information technology. Proceedings of the fourth IEEE international symposium* (pp. 334–337).
17. Ukovich, A., Zacchigna, A., Ramponi, G., & Schoier, G. (2006). Using clustering for document reconstruction. In E. R. Dougherty, J. T. Astola, K. O. Egiazarian, N. M. Nasrabadi, & S. A. Rizvi (Eds.), *Society of photo-optical instrumentation engineers (SPIE) conference series* (pp. 168–179). Bellingham: SPIE Press.
18. Wang, Y., & Wahl, F. M. (1996). Interactive multiobjective decision-making approach to image reconstruction from projections. *Signal Processing*, *48*(1), 67–75.
19. Wenyin, L., Zhang, W., & Yan, L. (2007). An interactive example-driven approach to graphics recognition in engineering drawings. *International Journal on Document Analysis and Recognition*, *9*(1), 13–29.
20. Zhang, S., Li, B., & Xue, X. (2010). Semi-automatic dynamic auxiliary-tag-aided image annotation. *Pattern Recognition*, *43*(2), 470–477.

Azzam Sleit is an Associate Professor with the Computer Science Department, University of Jordan, where he also functioned as the Director of the Computer Center from 2007 through 2009. Before joining the University of Jordan in 2005, Dr. Sleit was the Chief Information Officer at Hamad Medical/ Ministry of Public Health, where he was responsible for developing and overseeing the execution of the information technology strategy of healthcare for State of Qatar. Dr. Sleit has over twenty years of experience and leadership in the information technology field including work at all levels of government, private and international sectors. Before joining Hamad Medical, Dr. Sleit was the Vice President of Strategic Group & Director of Professional Services of Triada, USA. Dr. Sleit also served as the Director of Professional Services at Information Builders, USA. His research interest includes imaging databases, information retrieval, and distributed systems. Dr. Sleit holds B.Sc., M.Sc. and Ph.D. in Computer Science. He received his Ph.D. in 1995 from Wayne State University, Michigan.

Yacoub Massad was born in Beit Jala, Palestine. He received his M.Sc. in computer science from the University of Jordan, Amman in 2010, B.Sc. in computer information systems from the University of Bethlehem in 2007. His research interest includes wireless ad hoc networks, wireless sensor networks and image processing. He holds MCP and MCAD certificates from Microsoft.

Mohammed Musaddaq has received his Bachelor Degree in Computer Science from Yarmouk University, Jordan, in 2007, and Master Degree in Computer Science from the University of Jordan, Jordan, in 2011. He has practical certificates in networks which are MCP, MCSA, CCNA, CCNP. His research interests include the area of quality of service over wireless networks, especially in WiMAX networks.