Chapter 4

Zone Boundary Identification : Use of Dynamic Programming

THE previous chapter illustrates the application of simple concept like slope of a line for zone boundary detection. In this chapter we will show an application of dynamic programming to overcome limitations of the method shown in the previous chapter. First section in this chapter introduces basics of dynamic programming and later sections are devoted to list the limitation of the algorithm discribed in the previous chapter for zone boundary identification and proposed approach to overcome it.

4.1 Mathematical Preliminaries

4.1.1 Dynamic Programming [39]

Dynamic Programming

Dynamic programming (DP) determines the optimum solution of a multivariate problem by decomposing it into stages, each stage comprising a single variable subproblem. The advantage of the decomposition is that the optimization process at each stage involves one variable only, a computationally simpler task than dealing with all the variables simultaneously. A DP model is basically a recursive equation linking different stages of the problem in a manner that guarantees that at each stage an optimal feasible solution is obtained for any given state at that stage, which in turn yields an optimal and feasible solution for the entire problem for the given initial state.

4.1.1.1 Recursive Nature of Computation In DP

Computations in DP are done recursively, so that the optimum solution of one subproblem is used as an input to the next subproblem. By the time the last subproblem is solved the optimum solution for the entire problem is at hand. The manner in which the recursive computations are carried out depends on how we decompose the original problem. In particular, the subproblems are normally linked by common constraints : as we move from the one subproblem to the next, the feasibility of these common constraints must be maintained . Let us try to understand the concepts in dynamic programming by an example.

Example 4.1.1 (Shortest Path) *Consider a case wherein we have to find a shortest highway route between two cities. The available highway network, say for example, is as shown in the Fig. 4.1. Here, we have to find route between the starting city at node 1 and the destination city at node 7. The routes pass through intermediate cities designated by node 2 to 6.*

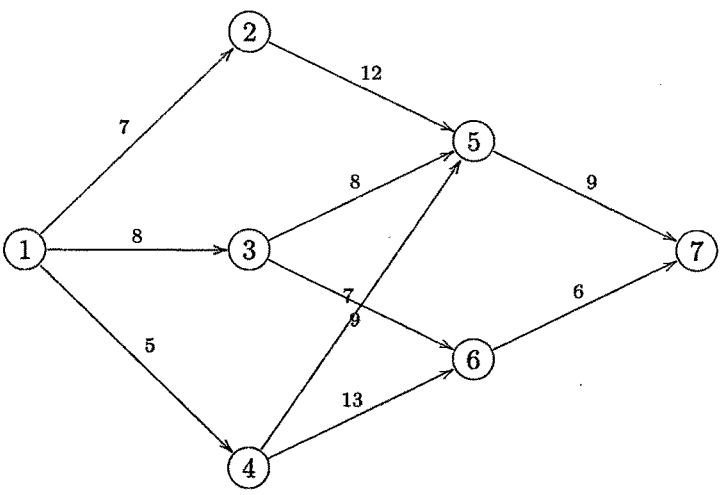

Figure 4.1: Route Network

We can solve this problem by exhaustively enumerating all the routes between node 1 and 7. However, in a large network, exhaustive enumeration may be intractable computationally.

To solve this problem by DP, we first decompose it into stages as shown in Fig. 4.2. Next, we carry out the computations for each stage separately.

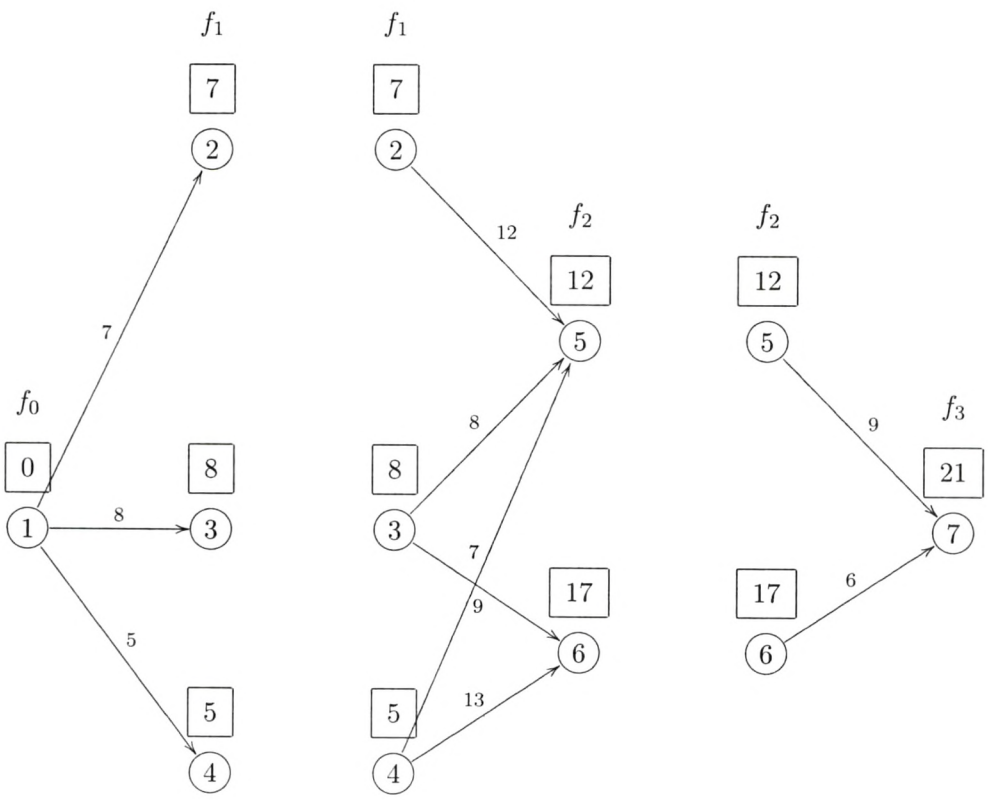

Figure 4.2: Decomposition of the Shortest Route Problem into Stages

The general idea for determining the shortest route is to compute the shortest(cumulative) distances to all the terminal nodes of a stage and then use these distances as input data to the immediately succeeding stage. Starting from node1, stage1 includes three end nodes (2, 3 and 4) and its computations can be shown as follows:

Stage 1 Summery :

Shortest distance from node 1 to node 2 = 7
Shortest distance from node 1 to node 3 = 8
Shortest distance from node 1 to node 4 = 5

Next, stage 2 has two end nodes, 5 and 6. Considering node 5 first, we see from Fig. 4.2 that a node 5 can be reached from three nodes 2,3 and 4 by three different routes : (2, 5), (3, 5), and (4, 5). This information together with the shortest distances to node 2,3 and 4, determines the shortest (cumulative)

distance to node 5 as

$$\begin{pmatrix}\text{Shortest Distance}\\ \text{to node 5}\end{pmatrix} = \min_{i=2,3,4}\left\{\begin{pmatrix}\text{Shortest}\\ \text{Distance}\\ \text{to node } i\end{pmatrix} + \begin{pmatrix}\text{Distance from}\\ \text{node } i\\ \text{to node 5}\end{pmatrix}\right\}$$

$$= min\left\{\begin{matrix}7+12=19\\ 8+8=16\\ 5+7=12\end{matrix}\right\} = 12 \quad (\textit{from node } 3)$$

Node 6 can be reached from nodes 3 and 4 only. Thus

$$\begin{pmatrix}\text{Shortest Distance}\\ \text{to node 6}\end{pmatrix} = \min_{i=3,4}\left\{\begin{pmatrix}\text{Shortest}\\ \text{Distance}\\ \text{to node } i\end{pmatrix} + \begin{pmatrix}\text{Distance from}\\ \text{node } i\\ \text{to node 6}\end{pmatrix}\right\}$$

$$= min\left\{\begin{matrix}8+9=17\\ 5+13=18\end{matrix}\right\} = 17 \quad (\textit{from node } 3)$$

Stage 2 Summery :

Shortest distance from node 1 to node 5 = 12 (*from node 4*)
Shortest distance from node 1 to node 6 = 17 (*from node 3*)

The last step is to consider stage 3. The destination node 7 can be reached from either node 5 or 6. Using the summary results from stage 2 and the distances from nodes 5 and 6 to node 7, we get

$$\begin{pmatrix}\text{Shortest Distance}\\ \text{to node 7}\end{pmatrix} = \min_{i=5,6}\left\{\begin{pmatrix}\text{Shortest}\\ \text{Distance}\\ \text{to node } i\end{pmatrix} + \begin{pmatrix}\text{Distance from}\\ \text{node } i\\ \text{to node 7}\end{pmatrix}\right\}$$

$$= min\left\{\begin{matrix}12+9=21\\ 17+6=23\end{matrix}\right\} = 21 \quad (\textit{from node } 5)$$

Stage 3 Summery :

Shortest distance from node 1 to node 7 = 21 (*from node 5*)

Stage 3 summery shows that the shortest distance between nodes 1 and 7 is 21. To determine the optimal route, stage 3 summery links node 7 to node 5, stage 2 summery inks node 4 to node 5 and stage 1 summery links node 4 to node 1. Thus, the shortest route is $1 \rightarrow 4 \rightarrow 5 \rightarrow 7$. ■

The example reveals the basic properties of computations in DP:

1. The computations at each stage involves the feasible routes of that stage, and that stage alone.

2. A current stage is linked to the *immediately preceding* stage only without regard to earlier stages. The linkage is in the form of the shortest-distance summery that represents the output of the immediately preceding stage.

4.1.2 General DP Model

We now show how the recursive computations in Example 4.1.1 can be expressed in general DP framework. Conventionally, Dynamic Programming Technique is applied to Multistage Optimization problems as described below. Consider a system consisting of n stages as shown in the Fig. 4.3

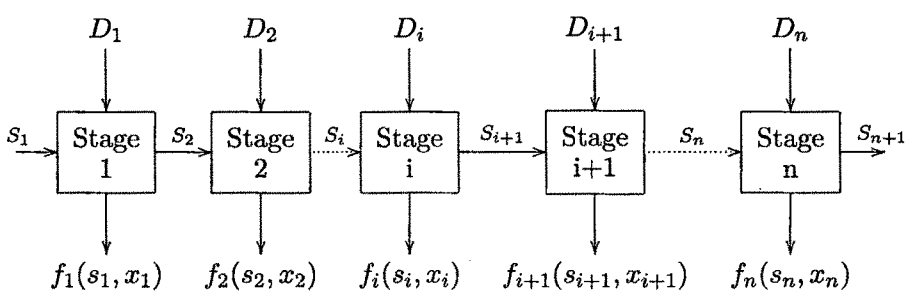

Figure 4.3: DP Stages

At each stage the system may be in some state and it is required to take certain decisions depending on which the system yields an output at that stage. The overall outcome of the system is a simple function (say, sum or product) of all the stage-wise outputs. The state at any particular stage and the decision taken at that stage changes the state of the system in the next stage. This conversion is governed by some state conversion law. It is required to take appropriate decisions at all the stages so as to optimize overall outcome of the system.

Let S_i and D_i denote the sets of state and decision variables respectively at stage-i. Suppose outcome at state-i is given by function $f_i(s_i, x_i)$ and the sate conversion law is given as $s_{i+1} = S(s_i, x_i)$. Then the Multistage Optimization problem is

$$\max/\min_{x_i \in D_i,\, 1 \le i \le n} F(s_1, x_1, x_2, \ldots, x_n) = \sum_{i=1}^{n} f_i(s_i, x_i) \text{ for any initial state } s_1 \in S_1 \tag{4.1}$$

The DP model not only optimizes a specific Optimization problem but decides the Optimum Policy which states that at any stage what decision should be taken with respect to any state resulting from the previous decisions, so that the over all outcome of the system is optimized. This is achieved by splitting this multistage n variable problem into n one variable parametric optimization problems and using the following Principle.

Bellman's Principle of Optimality : Future decision for the remaining stages will constitute an optimal policy regardless of the policy adopted in previous stages.

Bellman's Principle of Optimality

These n problems correspond to the n stages of the system and are solved recursively either from first stage onwards to the n^{th} stage (*forward recursion*) or starting from n^{th} stage and moving backwards to the first stage (*backward recursion*). At the i^{th} stage the optimum policy (optimum decisions with respect to any possible input state $s_i \in S_i$ at that stage) is decided on the basis of the cumulative outcome up to the previous stage plus the outcome of the current stage.

Recursion

In forward pass, we compute the cumulative outcome $F_i(s_{i+1})$ up to the i^{th} stage for reaching any possible state $s_{i+1} \in S_{i+1}$, by solving the recursive equations,

$$F_i(s_{i+1}) = \max/\min_{x_i \in D_i, S_{i+1}=S(s_i,x_i)} F_{i-1}(s_i) + f_i(s_i, x_i), \text{ for } i = 1, 2, \ldots, n \quad (4.2)$$

with an assumption that $F_0(s_1) = 0$ for any $s_1 \in S_1$. In this case, optimum decision is denoted by $x_i^*(s_{i+1})$ and corresponding initial state at the stage-i is denoted by $s_i^*(s_{i+1})$. $F_n(s_{n+1})$ gives the optimum overall outcome of the system and optimum decisions can be computed in reverse order, first finding $x_n^*(s_{n+1}), x_{n-1}^*(s_n^*(s_{n+1})), \ldots, x_1^*(s_2^*(\ldots))$.

In backward pass, we compute the cumulative outcome $F_i(s_i)$ up to the i^{th} stage for any possible initial state $s_i \in S_i$, by solving the recursive equations

$$F_i(s_i) = \max/\min_{x_i \in D_i, S_{i+1}=S(s_i,x_i)} F_{i+1}(s_{i+1}) + f_i(s_i, x_i), \text{ for } i = n, n-1, \ldots, 1 \quad (4.3)$$

with an assumption that $F_{n+1}(s_{n+1}) = 0$ for any s_{n+1}. In this case, optimum decision is denoted by $x_i^*(s_i)$and corresponding final state at stage-i is denoted by $s_{i+1}^*(s_i)$. $F_1(s_1)$ gives the optimum overall outcome of the sys-

tem and optimum decisions can be computed in forward order, first finding $x_1^*(s_1), x_2^*(s_2^*(s_1)), \ldots, x_n^*(s_n^*(\ldots))$.

Example 4.1.1 uses *forward recursion* in which the computation proceed from stage 1 to stage 3. The state of the system is described by the city from which the journey can begin at that state and the decision to be taken is to which immediate next city we should reach. Thus, in this example $S_1 = \{1\}$, $S_2 = \{2,3,4\}$, $S_3 = \{5,6\}$ and $S_4 = \{7\}$. $D_1 = \{2,3,4\}$, $D_2 = \{5,6\}$ and $D_3 = \{7\}$. The state conversion law is $s_{i+1} = x_i$. $f_i(s_i, x_i) = d(s_i, x_i)$.

Forward Recursion

The same example can be solved by *Backward recursion*, starting at stage 3 and ending at stage 1.Both the forward and backward recursions yield the same solution. Although the forward procedure appears more logical, DP literature invariably uses backward recursion. The reason for this preference is that, in general, backward recursion may be more efficient computationally.

Example 4.1.2 (Backward Recursion for Shortest Path Problem)

The backward recursive equation for Example 4.1.1 is

$$F_i(s_i) = \min_{x_i \in D, s_{i+1} = x_i} \{F_{i+1}(s_{i+1}) + d(s_i, x_i)\}, \; i = 1, 2, 3. \qquad (4.4)$$

where $F_4(s_4) = 0$ for $s_4 = 7$. The associated order of computation is $F_3 \rightarrow F_2 \rightarrow F_1$.

Stage 3. Because node 7 ($x_3 = s_4 = 7$) is connected to nodes 5 and 6 ($s_3 = 5$ and 6) with exactly one route each, there are no alternatives to choose from, and stage 3 results can be summarized as shown in the following tableau:

	$F_4(s_4) + d(s_3, x_3)$		
s_3	x_3=7	$F_3(s_3)$	$x_3^*(s_3)$
5	0+9	9	7
6	0+6	6	7

Stage 2. Route (2,6) is blocked because it does not exist. Given $F_3(s_3)$ from stage 3, we can compare the feasible alternatives as shown in the following tableau:

	$F_3(s_3)+d(s_2,s_2)$			
s_2	x_2=5	$x_2=6$	$F_2(s_2)$	$x_2^*(s_2)$
2	9+12=21	–	21	5
3	9+8=17	6+9=15	15	6
4	9+7=16	6+13=19	16	5

The optimum solution of stage 2 reads as follows: if you are in cities 2 or 4, the shortest route passes through city 5, and if in city 3, the shortest route passes through city 6.

Stage 1. From node 1. we have three alternative routes : (1,2),(1,3), and (1,4). Using $F_2(s_2)$ from stage 2, we compare these outcomes as shown in the following tableau.

	$F_2(s_2)+d(s_1,x_1)$				
s_1	x_2=2	$x_2=3$	x_2=4	$F_1(s_1)$	$x_1^*(s_1)$
1	7+21=28	8+15=23	5+16=21	21	4

Optimum solution at stage 1 shows that city 1 linked to city 4. Next, the optimum solution at stage 2 links city 4 to city 5. Finally, the optimum solution at stage 3 connects city 5 to city 7. Thus, the complete route is given as $1 \rightarrow 4 \rightarrow 5 \rightarrow 7$ and the associated optimum distance is 21. ■

4.2 The Need

We now come back to the problem of zone boundary identification. The algorithm presented in previous chapter gives reasonably good results, however it has some limitations as follows:

1. It assumes the connected components of a word to be vertically aligned.
2. It assumes the zone separation path to be a straight line.

These assumptions together lead to problems when the connected components of a word are not aligned, as shown in Fig.4.4 below:

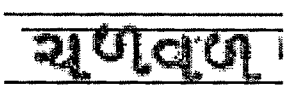

Figure 4.4: Problem Due to Straight Line Zone Separator

It can easily be seen that for robust and reliable zone segmentation, the assumptions made in previous method need to be relaxed. i.e. any general path

can be zone separator and the selection of the path should depend only on the connected component under consideration and it should not refer to any other connected component in the word or line. An obvious question that anyone can have in mind is, how to find this general path which can be a reliable zone separator?

4.3 Proposed Approach

In search of an appropriate method, we have gone through many papers wherein researchers have tried various approaches for different kinds of segmentations, like in [4], [8], [32], [34], [31]. Here we provide details about the method that we have selected to find a path as mentioned above with justifications wherever needed.

A careful analysis of the Gujarati Script revealed the following facts :

1. We can assume upper zone separation boundary just above a horizontal stroke (run of horizontally oriented line / pixel).
2. There is a Zone constraint - viz. the zone boundary falls in the region from 15% to 40% of the line height below the top of the text line.

Similar assumptions can be made for lower zone also. That is,

1. We can assume lower zone separation boundary just below a horizontal stroke.
2. There is a Zone constraint - viz. the zone boundary falls in the region from 15% to 40% of the line height above the bottom of the text line.

Now, finding the location of a zone separator is dependent on the existence of a horizontally oriented portion at the top and bottom of the middle zone component. This assumption is true for most of the Gujarati symbols that falls into middle zone. The exceptions are listed in Fig.4.5(a). Zone boundary in such cases can be identified by joining the ends of neighboring zone separators by a straight line, without loss of precision(Fig.4.5(b)).

(a) Illustrations (b) Exception Handling

Figure 4.5: Exception

We know that in conventional setup the term *touching glyphs* means the glyphs which are part of two different characters are touching horizontally. If we refer to this as a *horizontal touching* then on the same lines we can look at the intersection between glyphs in two different zones as *vertical touching* (Fig.4.6).

Touching Glyphs

Horizontal Touching

Vertical Touching

Figure 4.6: Touching Example

The advantage of defining this concept is that now we can use the rich knowledge of touching character segmentation for zone separation.

In [32] authors have given size and style independent algorithm for touching numeral segmentation which gives good results. However, this segmentation algorithm assumes a touching component, detected using another algorithm given in the same work, as input. But, in our case, as there are no touching characters, the vertically touching glyphs do not satisfy criteria given in [32] which are used to find the touching characters. Therefore, this vertically touching glyphs cannot be identified as touching characters by the algorithm and hence we cannot apply the algorithm in [32] for our purpose.

Breuel [8] has described an approach to segment hand printed characters, which allows the cuts to be curved (*Curved Pre-Stroke Cuts* (CPSC)). From the observations made above and the analysis of the script, it is clear that this approach can be adapted to find a zone separating path. We assume our image to be of the size $w \times h$.

The adapted version of the algorithm is as follows:

As in [8] we define a path P as a sequence of of pixels (x_i, y_i), $i = 0, \ldots, w$ $x_i, y_i \in N$ in an image of a connected component. Now out of all possible path we have to find the path which is optimum in some sense and can be the zone separator. The cost C assigned to a path can be modified as

$$C = \sum_{i=1}^{w} c_s(y_i - y_{i-1}) + c_i(x_i, y_i; I) \tag{4.5}$$

where,

$$c_s(\Delta y) = \begin{cases} 0, \text{ for } |\Delta y| = 0 \ . \\ 1, \text{ for } |\Delta y| = 1 \ . \\ \infty, \text{ for } |\Delta y| \geq 1 \ . \end{cases} \tag{4.6}$$

This limits the set of paths with finite cost to paths that are contained inside a cone within an angle of $\frac{\pi}{4}$ of the horizontal. Further, for finding upper zone the cost C should be minimum at just above the horizontal edge and for lower zone it should be minimum at just below the horizontal stroke.

It is clear from the eq. 4.6 that c_s depends only on step size. Therefore, for finding upper zone separator, c_i should be defined in such a way that it is smallest just above the horizontal stroke, largest at points inside the a stroke (to discourage cuts from going through the strokes) and it should be intermediate for background pixels. We have chosen these values as follows,

$$c_i(x_i, y_i; I) = \begin{cases} -5, & \text{if } (x_i, y_i) \text{ is just above a horizontal stroke} \ . \\ 2, & \text{if } (x_i, y_i) \text{ is in a stroke} \ . \\ 1, & \text{if } (x_i, y_i) \text{ is a background pixel} \ . \end{cases} \tag{4.7}$$

For finding lower zone separator, c_i should be defined in such a way that it is smallest just below the horizontal stroke, largest at points inside the stroke (to discourage cuts from going through the strokes) and it should be intermediate for background pixels. c_s for lower zone can be defined as

$$c_i(x_i, y_i; I) = \begin{cases} -5, & \text{if } (x_i, y_i) \text{ is just below a horizontal stroke} \ . \\ 2, & \text{if } (x_i, y_i) \text{ is in a stroke} \ . \\ 1, & \text{if } (x_i, y_i) \text{ is a background pixel} \ . \end{cases} \tag{4.8}$$

It is also important to note that we do not want path to progress in vertical direction. Therefore, the selection of points on the path will be constrained by $x_i = i$. We need to enforce some more constraints on selection of the point that would be the member of the segmenting path due to the observation we made regarding the location of the zone separator.

- Point (x, y) can be member of upper zone separator of a line with line height L_h if and only if $0.15L_h \leq y \leq 0.4L_h$.
- Similarly, point (x, y) can be member of lower zone separator of a line with line height L_h, if and only if $(L_h - 0.40L_h) \leq y \leq (L_h - 0.15L_h)$.

Application of these constraints will prevent the algorithm from producing

the zone separator in case of exceptional glyphs listed in Table 4.5(a). Such cases can be handled by producing path by joining end points of the neighboring zone separators (right end point of the left neighbor and left end of the right neighbor). If these characters occur as the first (last) character of the line then the segmenting path from the right (left) is extended up to the left (right) boundary of the text line. In other words, if for any part of the line, if any of the zone separators are not identified after executing this algorithm, then that region is assumed to have a straight line joining the ends of neighboring paths as its zone separator. (Fig.4.5(b))

The process starts by finding the connected component of a text line and for each component we try to find cut locations. Here, the basic assumption is that the connected region(potential connection point between symbols in middle and upper / lower zone) is having dense population of the black pixels and hence the separating path passes through the centroid.

Our goal is to find these two zone boundaries. Therefore, we divide the connected component array in two parts and process upper and lower half of the connected component separately. First step is to compute the centroid of one of the halves of the connected component say (x_c , y_c) and then we calculate $c(y)$ for all the paths passing through point (x_c, y), for all y.

In order to compute $c(y)$ efficiently, we use dynamic programming in two stages: first by finding optimal path passing through each point (x_c, y) between pixels of row number 0 and row number x_c and then finding it between pixels of column number w and number x_c. The procedure to compute optimum path starting at each (x_c, y) and w is described as **Algorithm** 4.1

Algorithm 4.1 To Find Optimum path starting at each (x_c, y) and w

Input: Binarized image, I_{ij} of a connected component.

Output: Arrays *cost* and *source* having same dimension as I. Where *source* is used to trace the path and corresponding cost is stored in *cost* array. **Process:**

for $i = 1$ to w **do**
  for $j = 1$ to h **do**
    $cost[i][j] \leftarrow \infty$
    $source[i][j] \leftarrow undefined$
  end for
end for

```
for i = 0 to h do
    add point (w, i) to queue
    cost[w, i] ← 0
end for
while queue ≠ empty do
    take point (i, j) from queue
    if i ≥ x_c then
        for delta = −1 to 1 do
            newCost ← cs(delta) + ci(i − 1, j + delta, image)
            if newCost < cost[i − 1, j + delta] then
                source[i − 1, j + delta] ← (i, j)
                cost[i − 1, j + delta] ← newCost
                add point (i − 1, j + delta) to queue
            end if
        end for
    end if
end while
for i = 0 to h do
    add point (0, i) to queue
    cost[0, i] ← 0
end for
while queue ≠ empty do
    take point (i, j) from queue
    if i ≤ x_c then
        for delta = −1 to 1 do
            newCost ← cs(delta) + ci(i − 1, j + delta, image)
            if newCost < cost[i − 1, j + delta] then
                source[i − 1, j + delta] ← (i, j)
                cost[i − 1, j + delta] ← newCost
                add point (i − 1, j + delta) to queue
            end if
        end for
    end if
end while
```

The algorithm is computed for upper and lower half separately.

Algorithm begins by initializing the *cost* array and *source* array, both with the same size as the image, padded by one pixel in the x direction. Initially

all costs are set to ∞ and all source point are set to *undefined.*

During the execution of the algorithm, the value of the cost array at point (i, j) is either the special value ∞ if the point has not been reached yet, or the cost of the best path to pixel (i, j) found so far. Similarly, the value of the source array is either the special value *undefined* if the pixel has not been reached yet, or the immediate predecessor of the pixel (i, j) on the best path to (i, j) from some point with $x = w$.

Next, the cost for the pixels on the line $\overline{(w, 0)(w, end)}$ is set to zero and those pixels are added to a FIFO queue. Until the queue is empty, pixel coordinates are taken from its front. The set of neighboring pixels that could be part of a valid path and their corresponding costs are computed. If the cost of reaching one of those neighboring pixels via a path through the current pixel is lower than any previously known path to that neighboring pixel, the cost and source arrays are updated, and the coordinates of the neighboring pixel are added to the back of the queue.

It may be noted that the only difference between the algorithm presented in [8] and **Algorithm** 4.1 is that the row operations in the first one is replaced by column operations in our case. After calculating the cost we find out local optima at the line $x = x_c$. Then the optimal cut though the point (x_c, y) is obtained by following the content of the *source* array.

Fig. 4.7 shows result of applying this algorithm to a connected component.

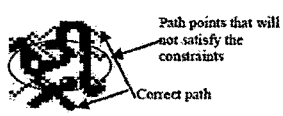

Figure 4.7: Result of Applying DP Based Zone Boundary Idnetification

4.4 Conclusions

This approach gives the zone separator which is not a straight line and hence prevents over-segmentation. However, it is very time consuming and work is needed to improve it further. The basic problem with this algorithm is that it assumes, centroid to be located in the touching area [8], which need not be true in all the cases for Gujarati script and hence, in the case where it is out side touching area, the algorithm might fail to identify a correct zone separator.