Enhanced Gap Sequencing Shell Sort

Agam Verma¹, Jitender Kumar Sharma
M.Tech(CSE)
United College of Engg. and Research
Allahabad, India
agam_alpine@yahoo.com¹, jitenderkumkum@yahoo.com²

Abstract: Sorting algorithms are used for arranging a list of numbers or characters in an ascending or descending order. Many sorting algorithms have been proposed for sorting a given sequence, some important algorithms of them are Bubble Sort, Insertion Sort, Selection Sort, Shell sort and many more. Sorting algorithms put elements in a certain order such as numerical or lexicographical order. Shell sort is the fastest algorithm in comparison to bubble, insertion and selection. Shell sort is an enhanced version of insertion sort. It reduces the number of swaps of the elements being sorted to minimize the complexity and time compared to insertion sort. Shell sorting algorithm sorts the elements according to the gap sequences. The operations depend on the used gap sequences (many gap sequences have been proposed). In this paper, we analyze the algorithm by using the following gap sequence: \(N_1; N_2; N_3; \ldots; 1\), where

\[
N_1 = \text{floor} \left( \frac{3n}{4} \right);
N_2 = \text{floor} \left( \frac{3N_1}{4} \right);
N_3 = \text{floor} \left( \frac{3N_2}{4} \right);
\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \}
B. Shell Sort:

Shell sort sorting algorithm is introduced by the Donald L. Shell in 1959[1,2]. Shell sort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared. Shellsort uses a gap sequence \( g_0 \); \( g_1 \); \( \ldots \); called the increment sequence. Any increment sequence is fine as long as \( g_1 = 1 \) and some other choices are better than others. Shellsort makes multiple phases through a list and sorts a number of equally sized sub lists using the insertion sort. Shellsort is also known as diminishing increment sort[2].

The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared. After each phase and some increment \( g_0 \), for every \( i \), we have \( a[i] \leq a[i+g_0] \) all elements spaced \( g_0 \) apart are sorted. The file is said to be \( g_0 \) -sorted. The size of the sub-lists, which are to be sorted gets larger with each phase through the list, until the sub-list consists of the entire list. (Note that as the size of the sub-list increases, the number of sub-lists to be sorted decreases.) This arrangement makes the insertion sort to run for an almost-best case with a complexity that approaches \( O(n) \).

The elements contained in each sub-list are not contiguous, rather, if there are \( i \) sub-lists then asub-list is composed of every \( i \)-th element. For example, if there are 4 sub-lists then the first sub-list would contain the elements located at positions 1, 5, 9 and so on. The second sub-list would contain the elements located at positions 2, 6, 10, and so on; while the fourth sub-list would contain the items located at positions 4, 8, 12, and so on.[3]

The efficiency of the algorithm depends on the size of the sub-lists used for each iteration. Along with the benefit of being robust, Shellsort is a complex algorithm and not nearly as efficient as the merge, heap, and quicksort. The Shell sort is still significantly slower than the merge, heap, and quick sorts. Enhanced Gap Sequencing Shell Sort introduces a new way to find the gap sequences to sort the large list of elements rapidly. Enhanced Gap Sequencing Shell Sort algorithm also works in same fashion as the previous existing versions of the Shell sort algorithm. The only difference is the way to choose the more efficient gap sequence, which is a key step for the algorithm to be more effective and better. Calculating the value of the gap sequence, in conventional Shell sort is a key step in the execution of the algorithm.

In conventional Shell sort, given by Knuth, the value of \( h_i \) is found by the following formula:

\[
h_1 = 1; \quad h_{i+1} = 3h_i + 1, \text{ and stop with } h_{i+1} \geq N.
\]

By using this existing formula Shell sort reduces the number of swaps up to 50% as compared to that of Insertion sort.

Enhanced Gap Sequencing Shell Sort mainly focuses to improve the efficiency of the previous existing shell sort algorithms. It can be achieved by choosing the appropriate values of gap sequences, which can reduce the number of swaps. In conventional Shell sort, the gap sequences are small, so they divide the list into large number of sub-steps, for example, if \( n = 100 \) then the gap sequence is (13, 4, 1), which means to sort the elements, it first divide the list into sub-lists of size 13, then 25-sublists of size 4. It makes algorithm to swap many elements many times to place them into right place. So, if we increase the size of gap sequence, it divides the list into less number of sub-lists of larger size and elements are not needed to swap many numbers of times.

Enhanced Gap Sequencing Shell Sort introduces a new mechanism to calculate the value of gap sequence. The formula for calculating gap sequence is as follows:

The gap sequence is of the form \( N_1, N_2, N_3, \ldots, 1 \), where \( N_1 = \text{floor}(3n/4) \);

\[
N_2 = \text{floor}(3N_1/4); \quad N_3 = \text{floor}(3N_2/4); \quad \ldots
\]

Now we apply the shell sort for calculating the number of swaps to sort the same problem (as discussed in Insertion sort) by using Knuth’s gap sequences.

1. **Unsorted list:**

\[
22,12,53,94,27,59,50,39,14,88,35,3,4,230,29,84,6,2,102,14,54,5,3,87,67,16,43,73,19,27,64,16,4,2,85,51,34.
\]

After applying the shell sort using Knuth’s gap sequences on this array of numbers, the number of swaps calculated for sorting it, are 170.

C. Enhanced Gap Sequencing Shell Sort:

Enhanced Gap Sequencing Shell Sort introduced a new way to find the gap sequences to sort the large list of elements rapidly. Enhanced Gap Sequencing Shell Sort algorithm also works in same fashion as the previous existing versions of the Shell sort algorithm. The only difference is the way to choose the more efficient gap sequence, which is a key step for the algorithm to be more effective and better. Calculating the value of the gap sequence, in conventional Shell sort is a key step in the execution of the algorithm.

In conventional Shell sort, given by Knuth, the value of \( h_i \) is found by the following formula:

\[
h_1 = 1; \quad h_{i+1} = 3h_i + 1, \text{ and stop with } h_{i+1} \geq N.
\]

By using this existing formula Shell sort reduces the number of swaps up to 50% as compared to that of Insertion sort.

Enhanced Gap Sequencing Shell sort mainly focuses to improve the efficiency of the previous existing shell sort algorithms. It can be achieved by choosing the appropriate values of gap sequences, which can reduce the number of swaps. In conventional Shell sort, the gap sequences are small, so they divide the list into large number of sub-steps, for example, if \( n = 100 \) then the gap sequence is (13, 4, 1), which means to sort the elements, it first divide the list into sub-lists of size 13, then 25-sublists of size 4. It makes algorithm to swap many elements many times to place them into right place. So, if we increase the size of gap sequence, it divides the list into less number of sub-lists of larger size and elements are not needed to swap many numbers of times.

Enhanced Gap Sequencing Shell Sort introduces a new way to calculate the value of gap sequence. The formula for calculating gap sequence is as follows:

The gap sequence is of the form \( N_1, N_2, N_3, \ldots, 1 \), where \( N_1 = \text{floor}(3n/4) \);

\[
N_2 = \text{floor}(3N_1/4); \quad N_3 = \text{floor}(3N_2/4); \quad \ldots
\]

To sort 125 items we first find an ‘\( h_i \)’ such that \( h_i \geq 125 \).

For 125 items, \( h_i \) is selected. The final value of \( h_i \) is two steps lower, or \( h_i \). Therefore, sequence for the values of ‘\( h \)’ will be 40, 13, 4, 1. Once the initial ‘\( h \)’ value has been determined, subsequent values may be calculated using the formula

\[
h_{i+1} = \text{floor}(h_i/3).
\]
\[ N_1 = \text{floor} \left( \frac{3 \times 6}{4} \right) = 4 \]
\[ N_2 = \text{floor} \left( \frac{3 \times 4}{4} \right) = 3 \]
\[ N_3 = \text{floor} \left( \frac{3 \times 3}{4} \right) = 2 \]
\[ N_4 = \text{floor} \left( \frac{3 \times 2}{4} \right) = 1 \]

So the gap sequence for sorting 125 elements is 
\[ (93, 69, 51, 38, 28, 21, 15, 11, 8, 6, 4, 3, 2, 1). \]

But in conventional Shell sort the gap sequence for 125 elements is 
\[ (40, 13, 4, 1). \]

Now we apply the Enhanced Gap Sequencing shell sort for calculating the number of swaps to sort the same problem as discussed in Insertion sort and Shell Sort).

\[ a. \quad \text{Unsorted list:} \]
\[ 22, 12, 53, 94, 27, 59, 50, 39, 14, 88, 35, 3, 115, 3, 4, 230, 29, 84, 6, 2, 102, 14, 54, 5, 3, 87, 67, 16, 43, 73, 19, 27, 64, 16, 4, 85, 51, 34. \]

Here \( n = 38 \), so
\[ N_1 = \text{floor} \left( \frac{3 \times 38}{4} \right) = 28 \]
\[ N_2 = \text{floor} \left( \frac{3 \times 28}{4} \right) = 21 \]
\[ N_3 = \text{floor} \left( \frac{3 \times 21}{4} \right) = 15 \]
\[ N_4 = \text{floor} \left( \frac{3 \times 15}{4} \right) = 11 \]
\[ N_5 = \text{floor} \left( \frac{3 \times 11}{4} \right) = 8 \]
\[ N_6 = \text{floor} \left( \frac{3 \times 8}{4} \right) = 6 \]
\[ N_7 = \text{floor} \left( \frac{3 \times 6}{4} \right) = 4 \]
\[ N_8 = \text{floor} \left( \frac{3 \times 4}{4} \right) = 3 \]
\[ N_9 = \text{floor} \left( \frac{3 \times 3}{4} \right) = 2 \]
\[ N_{10} = \text{floor} \left( \frac{3 \times 2}{4} \right) = 1 \]

So the gap sequence for 38 elements is 
\[ (28, 21, 15, 11, 8, 6, 4, 3, 2, 1). \]

After applying the Enhanced Gap Sequencing shell sort on this array of numbers, the number of swaps calculated for sorting it are 67.

II. COMPARISON OF ABOVE DISCUSSED THREE TECHNIQUES

Now the comparison for the three techniques is made here for the same problem.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Insertion Sort</th>
<th>Shell Sort</th>
<th>E.G.S. Shell Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>367</td>
<td>170</td>
<td>67</td>
</tr>
</tbody>
</table>

It is apparent that Shell sort reduces the number of swaps up to half as compared to the number of swaps in Insertion sort and Enhanced Gap Sequencing Shell Sort reduces the number of swaps further up to less than half as compared to the number of swaps in Shell Sort, thus improving the efficiency of the algorithm.

III. DETAIL DISCUSSION OF RESULTS FOR MORE PROBLEMS

It is necessary to execute a detailed comparison of all three algorithms by applying the algorithms on a wider variety of data to answer and establish results concretely.

A. For 25 Elements:

- Number of swaps in Insertion Sort: 83
- Number of swaps in Shell Sort: 47
- Number of swaps in E.G.S. Shell Sort: 22

B. For 50 Elements:

- Number of swaps in Insertion Sort: 464
- Number of swaps in Shell Sort: 127
- Number of swaps in E.G.S. Shell Sort: 63

C. For 100 Elements:

- Number of swaps in Insertion Sort: 2154
- Number of swaps in Shell Sort: 419
- Number of swaps in E.G.S. Shell Sort: 179

D. For 200 Elements:

- Number of swaps in Insertion Sort: 10180
- Number of swaps in Shell Sort: 886
- Number of swaps in E.G.S. Shell Sort: 544

E. For 350 Elements:

- Number of swaps in Insertion Sort: 10180
- Number of swaps in Shell Sort: 886
- Number of swaps in E.G.S. Shell Sort: 544
36,232,234,223,208,267,137,245,112,272,233,113,102,137,97,199,208,180,156,193,173,152,169,125,180,154,152,4,167,184,188,166,196,197,190,198,175,158,194,204,154,1
1,187,188,229,239,194,111,250,200,258,154,170,252,1,26,30,232,234,208,267,137,245,112,272,233,113,102,137,126,107,160,223,85,163,150,166,147,222,162,246,73,297,29

Number of swaps in Insertion Sort: 29068.
Number of swaps in Shell Sort: 2232.
Number of swaps in E.G.S. Shell Sort: 1079.

F. For 500 Elements:
Number of swaps in Insertion Sort: 243764
Number of swaps in Shell Sort: 8744
Number of swaps in E.G.S. Shell Sort: 3934

Table 2: Comparison of all three sorts

<table>
<thead>
<tr>
<th>Cases</th>
<th>Insertion Sort</th>
<th>Shell Sort</th>
<th>E. G. S. Shell Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>83</td>
<td>47</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>464</td>
<td>127</td>
<td>63</td>
</tr>
<tr>
<td>3.3</td>
<td>2154</td>
<td>419</td>
<td>179</td>
</tr>
<tr>
<td>3.4</td>
<td>10180</td>
<td>886</td>
<td>544</td>
</tr>
<tr>
<td>3.5</td>
<td>29068</td>
<td>2232</td>
<td>1079</td>
</tr>
<tr>
<td>3.6</td>
<td>62679</td>
<td>3456</td>
<td>1613</td>
</tr>
<tr>
<td>3.7</td>
<td>243764</td>
<td>8744</td>
<td>3934</td>
</tr>
</tbody>
</table>

Figure 2: Comparison graph based on cases

The Enhanced Gap Sequencing Shell Sorting algorithm is a good approach towards achieving the excellence in the algorithms to provide the more efficient solutions. This has been achieved by decreasing the number of swaps required to sort the list of elements.

The results of above solved problems show that the new algorithm provides a much efficient way to sort the elements and hence cause to save the computational resources. It has been observed that the Enhanced Gap Sequencing Shell Sorting algorithm can solve the problem in almost 60 times less swaps as compared to insertion sort and in almost half swaps as compared to the traditional shell sorting algorithm. Figure 3 shows the detailed overview of the number of swaps required to sort different numbers of elements.

Figure 3: Graph showing elements-swap ratio

IV. ANALYSIS

In the paper “Analysis of Shellsort and Related Algorithms”, Robert Sedgewick [4] has described an open problem, “Are there increment sequences that perform better than known ones in practice?”, for performance issues and claims that finding a sequence that leads to running times 25% lower than the best known certainly would be of practical interest, we can reduce the running time by reducing the number of comparisons for the algorithm. In our proposed algorithm, we have reduced the number of comparison up-to 40% to 50% in some ideal cases but in many cases up-to 20% to 30%. We compare our algorithm with insertion sort and shell sort and get some interesting results as can be seen in the above given graphs and tables. Marcin Ciura [5] in his paper “Best Increments for the Average Case of Shell sort”, shows the result for 128 elements where data get sorted in 535(approx.) swaps but in our case 200 elements takes 544 swaps to get sorted. Thus the proposed algorithm is better for sorting.

V. CONCLUSION

This research paper focuses on an enhancement and improvement in existing sorting algorithms. The traditional Shell sort algorithm results an average number of comparisons of elements but it does not give minimum number of swaps. The older approaches of the Shell Sort algorithm have stated that the number of swaps produced by Shell Sort can be further reduced by choosing an efficient gap sequence.

The main purpose of reducing the number of swaps is to use the computational resources that are available in terms of processor speed, memory and storage.

Enhanced Gap Sequencing shell sort algorithm provides an efficient and better approach to decrease the number of comparisons as well as number of swaps. Enhanced Gap Sequencing shell sort algorithm results least number of swaps on any size of data. This algorithm works more efficiently as the size of data grows.

This algorithm has described a simple and easy formula that calculates the values of \( N_1 \) by the formula \( N_1 = \text{floor} \left( \frac{3n}{4} \right) \), where ‘n’ is the number of elements to be sorted, and then find values of \( N_2; N_3; \ldots \ldots \); \( N_k \) by placing values of \( N_1; N_2; N_3; \ldots \ldots; N_{k-1} \) in place of ‘n’ in the formula. This algorithm improves the performance of the existing algorithms up to 60% in some cases.

VI. ACKNOWLEDGMENT

A number of people have made this paper possible. In particular I wish to thank my friend, Mr. Jitender Kumar, for providing much needed help and support and my teachers for their guidance and displaying inconceivable patience. My family and friends for constantly asking me when I would finish my thesis. And lastly, my mother for teaching me never to give up on any endeavor.

VII. REFERENCES

