A STUDY OF A REDUNDANT MEMORY REPAIR ALGORITHM

By

SONG GAO

Bachelor of Medicine

Beijing Medical University

Beijing, P. R. China

1999

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May 2003

A STUDY OF A REDUNDANT MEMORY REPAIR

ALGORITHM

Thesis Approved:

[illegible]

Thesis Advisor

J Chandler

[illegible]

[illegible]

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere appreciation to my major advisor, Dr. Nohpill Park for his intelligent supervision, constructive guidance, inspiration and friendship. My sincere appreciation extends to my other committee members Dr. J. P. Chandler and Dr. G. E. Hedrick, whose guidance, assistance, encouragement, and friendship are also invaluable.

More over, I wish to express my sincere gratitude to those who provided suggestions and assistance for this study: Ms. Nin Jing, Ms. Lingyan Li, Mr. LingFa Kong and Mr. Zou Zhen.

Finally, I would like to thank the Department of Computer Science for the excellent advanced education.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter I

Introduction

As VLSI technology advances, the number of devices per chip and the chance of having device failures on the chip increases dramatically. Including redundant rows and columns that can be used to replace defective rows or columns, so-called row/column deletion technique, is a standard practice to enhance memory yield substantially. However, the overhead of utilization of redundant elements and its cost-benefit is still an open problem due to its high computational complexity.

The problem, repairing reconfigurable memory array with optimal spare rows and spare columns, is NP-complete [7]. There have been extensive researches on the redundant memory repair algorithms, such as repair-most [12], polynomial approximation algorithm [7] and comprehensive approaches [2]. However, none of them can generate an optimal repair solution [7,12,2]. Due to time and space limitation of the testing equipment, the polynomial approximation algorithm and comprehensive algorithm are also not time-efficient [7].

In this thesis, we propose a new *two-dimensional array of linked list representation* of defective memory cells to implement an *approximation algorithm*, which will greatly decrease the space required to represent the memory cells. Also, we will modify the polynomial approximation algorithm [9] and use it for our memory-repair yield estimation. Comparing with Kuo and

Fuchs' algorithm [7], our proposed algorithm is easier to implement and saves computational spaces by about one half.

The objective of this thesis is to use our proposed repair algorithms to study the relationship among memory size, repair redundancy and fault rates. Also, we will study the impact of different fault models such as random distribution model and negative binomial distribution model on memory yield.

This proposal is organized as follows. In the following section, literature review related to this research work will be given. In section III and IV, preliminary results and conclusions are addressed. Section V summarizes the proposed work and points out its future application.

Chapter II

Literature Review

2.1 Memory redundancy architecture

Memory plays an important role in today's computer systems. With the advent of deep submicron technology and system-on-chip (SoC) design methodology, heterogeneous cores from different sources can be integrated in a single chip that contains multi-million gates [1]. Embedded memory is one of the most widely used cores for SoC, and memory cores usually dominate the silicon area and yield of the chip [1]. Increasing the memory on a SoC adds layers, complicates the manufacturing processes, and increases cell density [14,13]. In fact, because of their high cell density, embedded memories are more prone to defects than any other component on the chip [14].

One solution to minimizing the occurrences of faults is to improve the manufacturing and testing processes (*fault-avoidance technique*) [1]. However, this can't be considered as a viable alternative because it can be very costly and also quite difficult (or even impossible) to implement. On the other hand, we can now afford to put redundancies on the IC to make fault-tolerant design viable by setting aside some of the chip/wafer area to this purpose (*fault-tolerance technique*) [12].

There are several redundancy architectures existing in large memory cores to facilitate repair and maintain an acceptable manufacturing yield to date, such as spare rows, columns, and/or banks.

In addition to the traditional spare rows/columns configuration of memory arrays, Park and Lombardi [10] have proposed the laser physical cutting of spare rows/columns, thus increasing the spare units and the yield without increasing spare redundancy.

Moreover, multichip module technology [16] has also employed redundancy techniques. However, conventional redundancy methods cannot always generate acceptable repair solutions for multichip memories. For example, in order to decrease the current and reduce the access time by shortening the length of the bit and word lines [16], the large size of the memory array are often partitioned into several sub-arrays. Using the conventional redundancy methods, each sub-array will have its own redundant rows and columns, leading to situations where one sub-array has an insufficient number of spare lines to handle local defects while others still have several unused redundant lines. Also, the higher density of the new sub-micron memory ICs drastically decreases the yield loss due to chip-kill defects, e.g., defects in core circuits like sense ampliers and line drivers, while the conventional technique using spare rows and columns is incapable of dealing with such defects [17].

Koren et al [5] proposed a Hybrid defect-tolerance scheme for high-density memory ICs by using smaller sub-array redundancy containing modules. Kikuda et al [4] introduced the failure-related yield model, based on which they generated an optimized redundancy scheme for 64-Mb DRAM. It shows that memory with 1-MB or smaller subblocks containing more than two spare rows and two spare columns in each subblock can increase yield greatly.

2.2 Memory redundancy repair algorithms

The redundancy analysis algorithm also has been addressed extensively.

The algorithm proposed by Day [2] is an exhaustive search algorithm that generates the entire tree of all possible solutions. This approach is not acceptable when the array size is large.

The repair-most algorithm [12] proposed by Tarr et al. is a greedy method, which repetitively choose the row or column that has the most number of faulty cells. Though the repair-most algorithm is simple and easy to implement, its yield calculation is far more than satisfactory. For example, it may not generate a solution for a theoretically repairable defective array [2]; also the solution it generated may not be optimal [7].

Kuo and Fuchs [7] have stated that the problem is NP-complete and proposed a branch-and-bound algorithm which is actually a modified comprehensive algorithm and a heuristic polynomial approximation algorithm. The branch-and-bound approach is not efficient as the problem becomes large. The heuristic polynomial approximation algorithm [7] and its modified version [9] suffer from implementation complexity. However, they are the most accurate approximation algorithms for yield improvement of reconfigurable arrays to date.

2.3 Memory defect models

Not only the algorithms are important for yield estimation, the faulty memory cell distribution models also play an important role. In order to evaluate the manufacturing yield of fault-tolerant VLSI chips, different defect models have been proposed.

Because of the inherent fluctuations in an IC fabrication process, defects may be independently introduced during any of the many processing steps that a VLSI chip undergoes. Thus, chip yield is the product of the yields of the individual processing steps. The random defect model (The Flat(Uniform) Distribution. p(x) dx = {1 \over {b-a}} dx, if a <= x < b and 0 otherwise) [14] assumes that defects occur randomly on a wafer. This yield model observes the Poisson random variable distribution. However, simple random defect model is widely criticized as being too pessimistic for single chips [1], because the defects are often not randomly distributed across a wafer, but rather are clustered in certain regions.

Fault clusters in integrated circuits can be roughly categorized into four classes [1]. The first class is that the fault clusters must be larger than the chip size (large-size clustering); the second class is that the fault clusters must be smaller than the chip size (small-size clustering); the third class is that the fault clusters must be with the same dimension as that of the chip area (medium-size clustering); and the fourth class is that the clusters vary in dimension.

To account for nonrandom defect distributions, different models have been proposed for the first three classes of fault clusters. The unified negative binomial distribution model(p(k) = {\Gamma(n + k) \over \Gamma(k+1) \Gamma(n) } p^n (1-p)^k) proposed by Koren et al [6], the model of compound Poisson distribution with gamma function, is the best fit for the experimental data in the case of large-size fault clustering, medium-size fault clustering as well as small-size fault clustering [1]. It proposed that the number of faults in a block has the negative binomial distribution, while the defects in each block is distributed randomly. This block-sized negative binomial distribution model has three parameters: the average number of faults λ, the clustering parameter α, and the block size B.

Chapter III

Preliminaries

3.1 **Existing base algorithms**: There are three kinds of algorithms exist to date, however, none of them has a good performance when repairing a large size of redundancy memory.

3.1.1 **Repair-Most** [12, Figure 1, Figure 2]: Repetitively chooses and replaces the row or column that has the most number of faulty cells to cover.

1) Computational Time Complexity: O(M+N) where M is the number of rows that have defects and N is the number of columns that have defects. Proof: as each time, the process will repair one row or one column, there are at most (M+N) iterations, so the computational time complexity is O(M+N).
2) Computational Space Complexity: O(R*C) where R is the number of rows of the memory, and C is the number of columns of the memory. Proof: because the algorithm is using array [Figure 3b] to represent the defective memory, and the array has C columns and R rows, so the computational time complexity is O(R*C).

3) Yield Optimization: Not optimal. Proof: some defective memory patterns can't be repaired by using this algorithm but can be repaired by using optimal algorithm.

4) Implementation: Easy. Proof: the implementation is straightforward, and we only need to keep the number of defective cells in each row and in each column.

5) Repair Process: greedy method, repeatedly choose the row or column that has the most number of faulty cells.

6) Disadvantage: It may not generate a solution for a theoretically repairable defective memory array [2]; also the solution they generated may not be optimal [7]. Its yield is far more than satisfactory.

3.1.2 **Heuristic Approximation Algorithm** [7, Figure 3, Figure 5]: Optimally finds and replaces the defect that has only one defect in a particular row or column. If there is no single defect in a row or column, it finds and replaces the row or column that has the greatest repair effect.

1) Computational Time Complexity: O((SR+SC)*(M+N)) where M is the number of rows that have defects, and N is the number of columns that have defects. SR is the number of spare rows, and SC is the number of spare columns. Proof: as there are only SR spare rows and SC spare columns, there are at most O(SR+SC) iterations. For each iteration, the algorithm will search all the rows and columns that have defects to decide which one to be

replaced, and there are (M+N) rows and columns to be compared. So the total computational time complexity is O((SR+SC)*(M+N)).

2) Computational Space Complexity: O(R+C+2E), where R is the number of rows of the memory, C is the number of columns of the memory, and E is the number of defects in the memory. Proof: the algorithm is using bipartite graph [Figure 3a] to represent the defective memory, that is, it needs row array of linked list and column array of linked list. For row array of linked list, we need a row array (size R) and R linked lists. The total number of nodes of R linked lists is the total number of defective memory cells represented as edges. The column array of linked list is represented similarly. So the total computational space complexity is O(R+C+2E).

3) Yield Optimization: Not optimal, however, optimal solutions have been generated for most of cases [7]. Proof: in Fuchs' paper [7], there are comparisons between exhaustive algorithm and the approximation algorithm, and for most case, the approximation algorithm can generated optimal solutions.

4) Implementation: complex. Proof: as the algorithm uses set and graph theory, it is difficult to be understood and implemented.

5) Repair Process: greedy method, repetitively chooses the row or column that has the greatest repair effects.

6) Disadvantage: It may not generate a solution for a theoretically repairable defective memory array [2]; also the solution they generated may not be optimal [7].

3.1.3 Exhaustive algorithm [2]: Generates a tree of all possible solutions and finds the optimal repair solution.

1) Computational Time Complexity: NP-complete [7]. Proof: Fuchs has proved that the problem is NP-complete [7].
2) Computational Space Complexity: O(R+C+2E), where R is the number of rows of the memory, C is the number of columns of the memory, and E is the number of defects in the memory. Proof: the algorithm is using bipartite graph [Figure 3a] to represent the defective memory, that is, it needs row array of linked list and column array of linked list. For row array of linked list, we need a row array (size R) and R linked lists. The total number of nodes of R linked lists is the total number of defective memory cells represented as edges. The column array of linked list is represented similarly. So the total computational space complexity is O(R+C+2E).
3) Yield Optimization: Optimal. Proof: the algorithm generates all the repair solutions and finds the optimal.
4) Implementation: Hard. Proof: the algorithm uses set and graph theory, and generates all the possible combinations of spare rows and spare columns, it is difficult to be understood and implemented.

5) Repair Process: Exhaustively test all the possible spare row and column repair combinations to find the optimal one.

6) Disadvantage: Time inefficient (as the problem is NP-complete, it is not efficient for even moderate size of memory).

3.2 Proposed algorithm: We propose the *two-dimensional array of linked list representation* of the memory with defects. Our proposed algorithm searches the two-dimensional array of linked list represented memory repeatedly to repair the row or column that has the greatest repairing effects. The algorithm we propose in this thesis shows both computational space and time efficiency [Figure 7, Figure 8, Figure 11].

1) Computational Time Complexity: O((SR+SC)*(M+N)) where M is the number of rows that have defects, and N is the number of columns that have defects. SR is the number of spare rows, and SC is the number of spare columns. Proof: as there are only SR spare rows and SC spare columns, there are at most O(SR+SC) iterations. For each iteration, the algorithm will search all the rows and columns that have defects to decide which one to be replaced, and there are (M+N) rows and columns to be compared. So the total computational time complexity is O((SR+SC)*(M+N)).

2) Computational Space Complexity: O(R+C+E), where R is the number of rows of the memory, C is the number of columns of the memory, and E is the number of defects in the memory. Proof: the algorithm is using *Two-dimensional array of linked list* [Figure 6] to represent the defective memory, that is, each defective memory cell is only represented once. In addition, the algorithm needs one row

array of size R, and one column array of size C. As there are total E defective memory cells, the total computational space complexity is O(R+C+E).

3) Yield Optimization: Not optimal, however, optimal solutions can be generated for most of cases. Proof: this feature is tested and conformed by experiments.

4) Implementation: the proposed algorithm is not as simple as the repair-most algorithm to be implemented. However, it is easier to implement than comprehensive and heuristic approximation algorithms. Proof: the algorithm uses *two-dimensional array of linked list* to represent the defective memory, and this representation requires only constant time to access each defective memory cell's defective neighbors. For heuristic approximation algorithm and exhaustive algorithm, it will search all the corresponding linked lists to find and update its neighbors' cost and degree.

5) Repair Process: greedy method, repetitively chooses the row or column that has the greatest repairing effects.

6) Disadvantage: It may not generate a solution for a theoretically repairable defective memory array; also the solution they generated may not be optimal.

```
1. For i=0 to Row
     Save the number of Faults in Row i in RowCount[i]
   For j=0 to Column
     Save the number of Faults in Column j in ColCount[j]
2. Find the row I or column j that have the biggest number of faults.
3. If(SR>0 and RowCount[i] is the biggest) Then
           Repair the Memory with a Spare Row;
           Update RowCount[] and ColCount[];
           SR:=SR-1;
   Else Repair The Memory With a Spare Column
           Update RowCount[] and ColCount[];
           SC:=SC-1;
4. Repeat step one until no spares or faults remain.
5. If (SR=0 and SC=0 and faults remain), then this device cannot be
   repaired using this algorithm.
6. If no fault remains, then the device can be repaired.
```

Figure 1. Repair-Most algorithm [1].

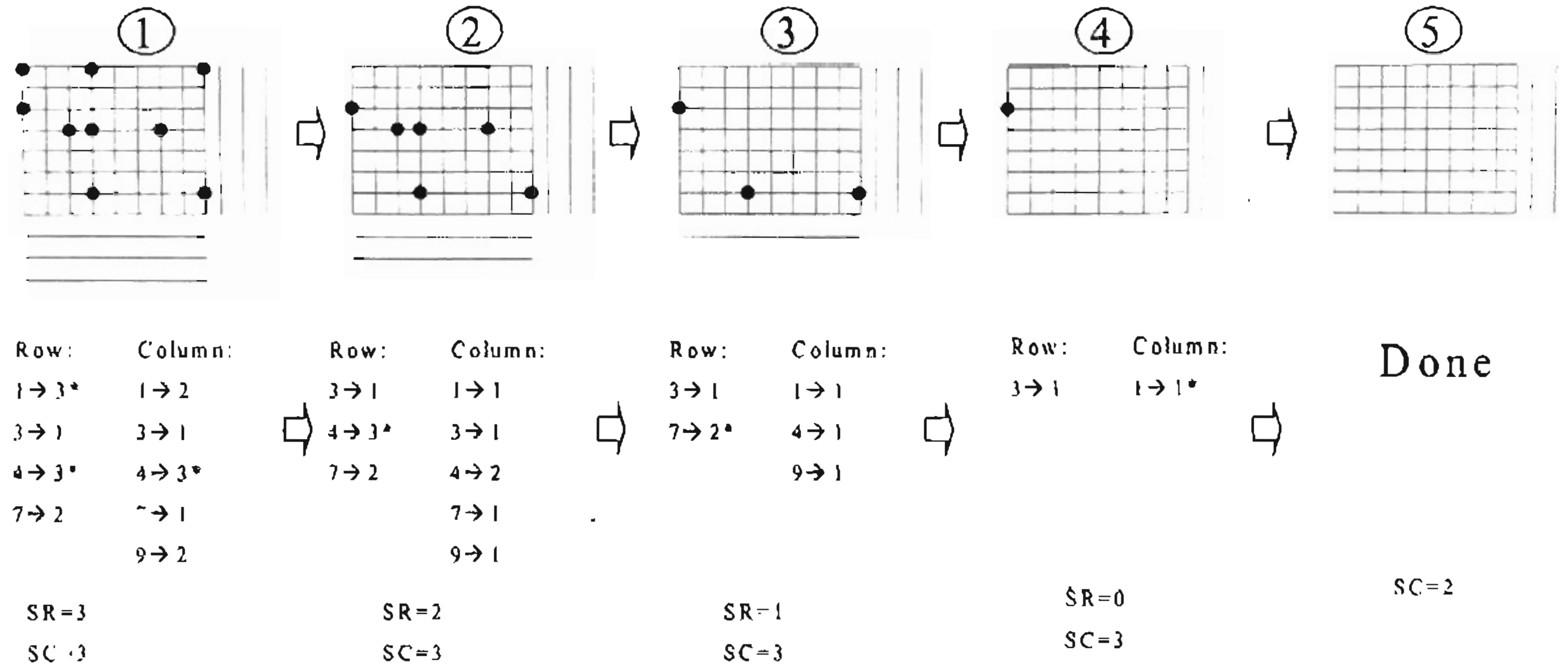
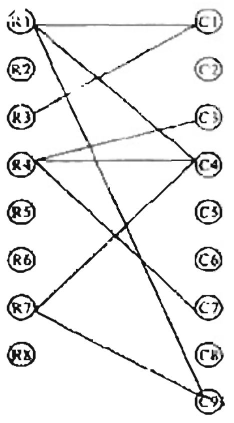


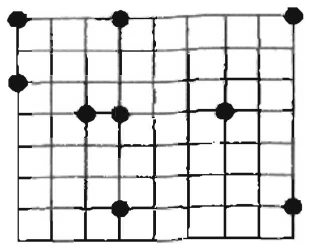
Figure 2. Repair Most Algorithm Repair Process

```
Begin
Begin
    For each vertex v in row vertices A and column vertices B
    Calculate the cost cc(v) and degree dc(v).
End
    Success := false;
While defects exist and (SR>0 or SC>0 ) do Begin
        If there are nodes with degree one and it is selectable, then
            Select the vertex v with the minimum cc(v)/dc(v);
        Else
            Select the selectable vertex v with minimum cc(v)/dc(v) over all
            vertices
        If v ∈ A and SR>0 then Begin
            Success := true;
            SR := SR-1;
            For each (u,v) ∈ E  do Begin
                    cc(u) := cc(u) – 1;
                    dc(u) := dc(u) –1;
                End;
        End;
        Else If v εB and SC>0 then Begin
                Success := true;
                SC := SC-1;
                For each (u,v) ∈ E do
                    Begin
                        cc(u) := cc(u) – 1;
                        dc(u) := dc(u) –1;
                    End;
            End;
        If Success then Begin
                cc(v) :=0, add  v to repair-solution Rh,
                delete v, all incident edges to v, and resulting isolated
                vertices.
                Success := false;
            End;
        Else if v ∈ A then
            Mark the remaining vertices in A unselectable.
            Else mark the remaining vertices in B unselectable
    End;
If there are still defects then
    Return fail;
Else return Rh;
End;
```

Figure 3. Heuristic approximation algorithm [8].

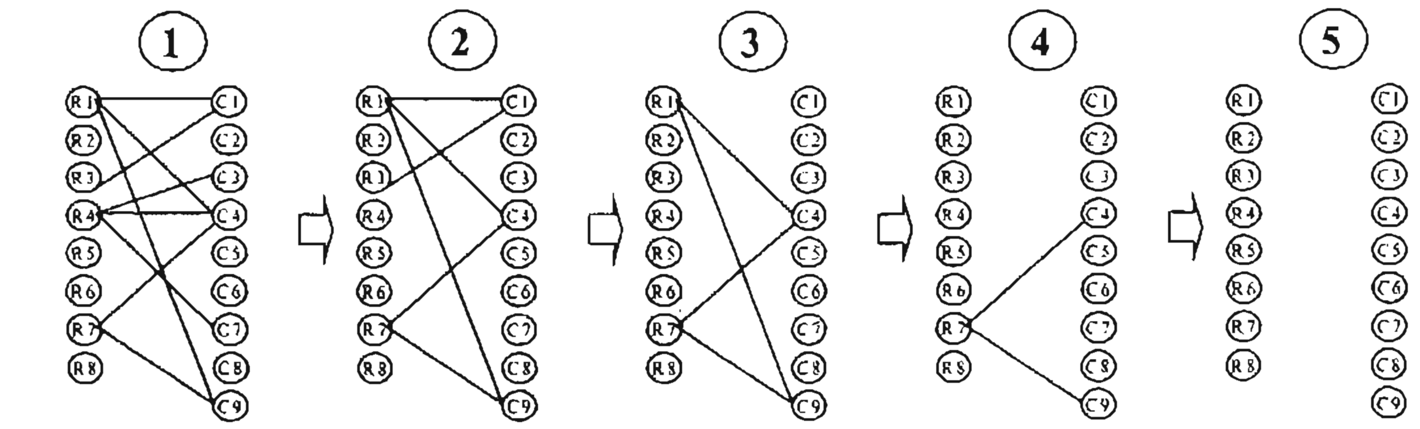


(a)



(b)

Figure 4. Bipartite graph (a) representation of the memory faulty pattern (b).



Row:	Column:	Row:	Column:	Row:	Column:	Row:	Column:	Done
1: 2$\rightarrow$2	1: 1$\rightarrow$1	1: 2$\rightarrow$3	1: 1$\rightarrow$1*	1: 2$\rightarrow$2*	4: 2$\rightarrow$2	7: 1$\rightarrow$2*	4: 1$\rightarrow$1	
3: 2$\rightarrow$1	3: 3$\rightarrow$1	3: 2$\rightarrow$1	4: 2$\rightarrow$1	7: 2$\rightarrow$2	9: 2$\rightarrow$2		9: 1$\rightarrow$1	
4: 1$\rightarrow$2*	4: 2$\rightarrow$1	7: 2$\rightarrow$2	9: 2$\rightarrow$1					
7: 2$\rightarrow$1	7: 3$\rightarrow$1							
	9: 2$\rightarrow$1							
SR=3		SR=2		SR=2		SR=1		SC=2
SC=3		SC=3		SC=2		SC=2		

Figure 5. Heuristic Approximation Algorithm Repair Process

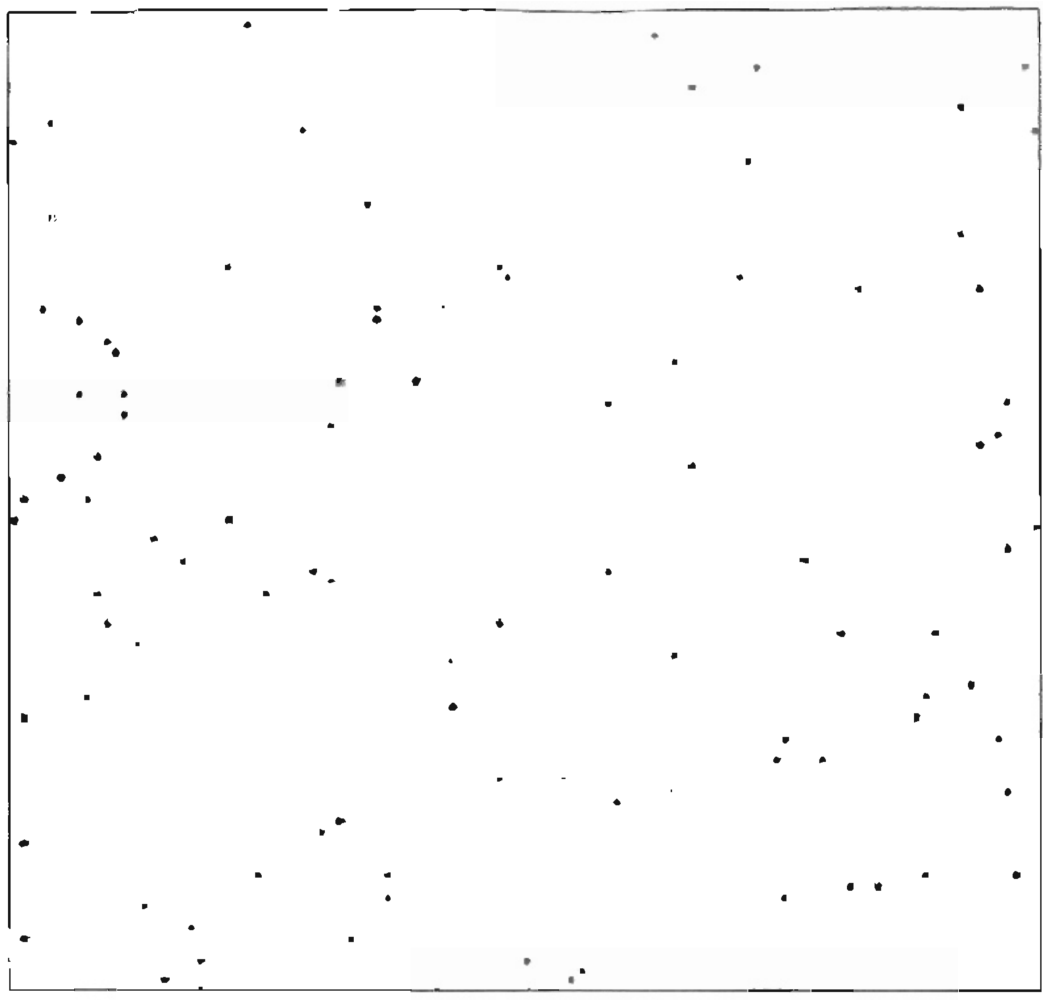


Fig. 6 Random Fault Distribution Map (Random Distribution)
with Row =Column=128, Faulty Rate=1%

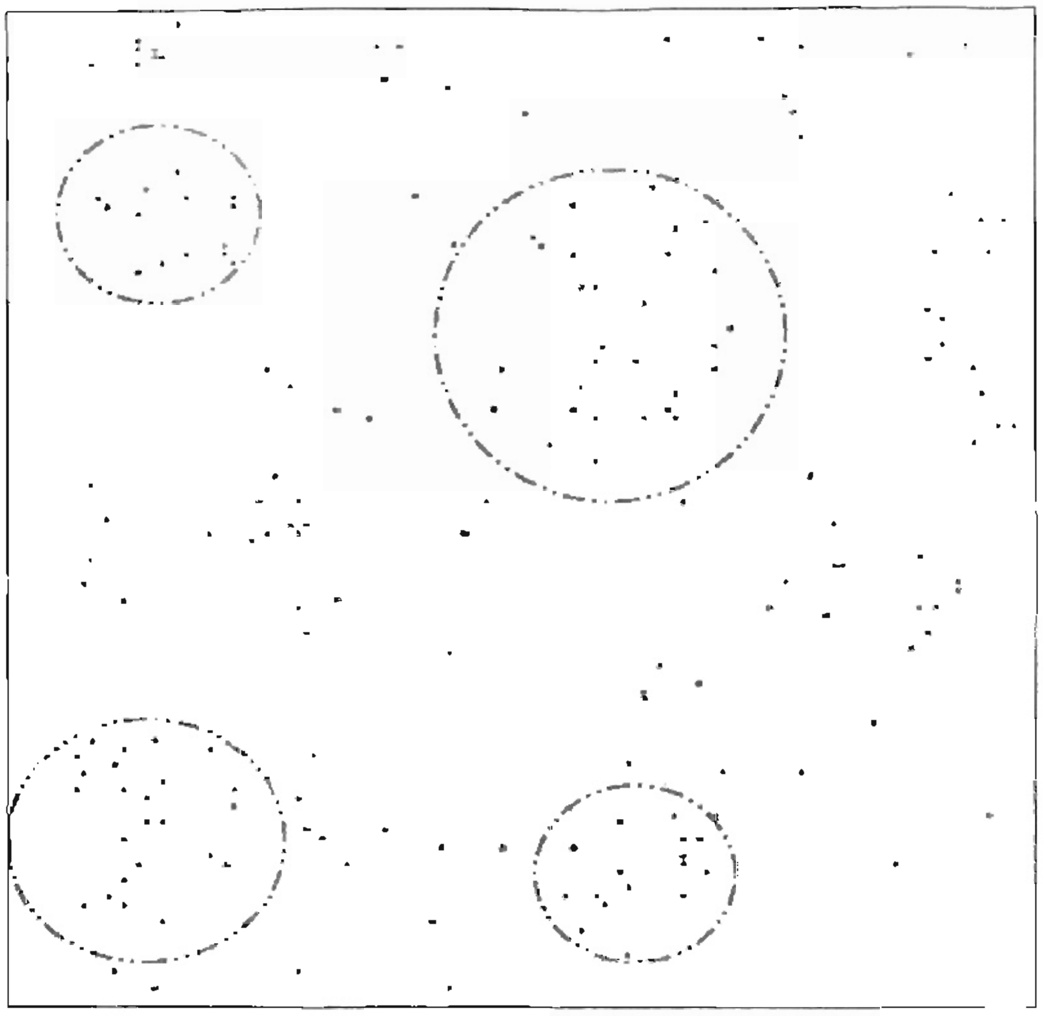


Fig. 7 Clustered Fault Distribution Map (Negative Binomial distribution) with Row=Column=128, Faulty Rate=1%, α=3.8274, λ=1.934.

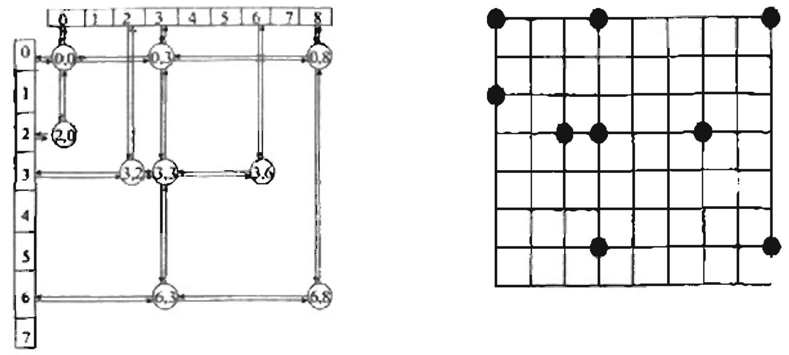


Figure 8. *Two-Dimensional Array of Linked List* (a)
Representation of the memory faulty pattern (b).

```
Begin
    For each row v in R or column v in C
    Calculate cost cc(v), degree dc(v), and counter n(v);
End;
Success := false;
While defects exist and (SR>0 or SC>0 ) do Begin
        If there is row or column v with cc(v)=1, then Begin
            For all rows and columns with cc(v)=1,
                Select the row/column v with maximum n(v);
            If more than one maximum n(v) exist, then
                Select v with the maximum dc(v);
        End;
        Else Begin
            Select the selectable row or column v with minimum cc(v);
            If more than one minimum cc(v) exist, then Begin
                Select the row/column v with maximum n(v) and minimum
            cc(v);
                If more than one maximum n(v) exist, then
                Select v with the maximum dc(v);
            End;
        End;
        If v ε R and SR>0 then Begin
            Success := true;
            SR := SR-1;
            For each u in v do Begin
                    dc(u) := dc(u) –1;
                    dc(v) :=0;
                    if( n(v) = cc(u) ) then Begin
                        n(u) := n(u) – 1;
                        Recalculate cc(u);
```

Figure 9. Modified Heuristic Approximation Algorithm for repairing large size of memory. (part 1)

```
                End;
            End;
        End;
        Else If v ε C and SC>0 then
        Begin
            Success := true;
            SC := SC-1;
            For each u in v do
            Begin
                    dc(u) := dc(u) –1;
                    dc(v) :=0;
                    if( n(v) = cc(u) ) then Begin
                        n(u) := n(u) – 1;
                        Recalculate cc(u);
                    End;
            End;
        End;
        If Success then
            Begin
                cc(v) :=0, add v to repair-solution Rh,
                delete v, all u in v
                Success := false;
            End;
        Else if v εR then
            Mark the remaining vertices in R unselectable.
            Else mark the remaining vertices In C unselectable
    End;
If there are still defects then
    Return fail;
Else return Rh;
```

Figure 10. Modified Heuristic Approximation Algorithm for repairing large size of memory. (part 2)

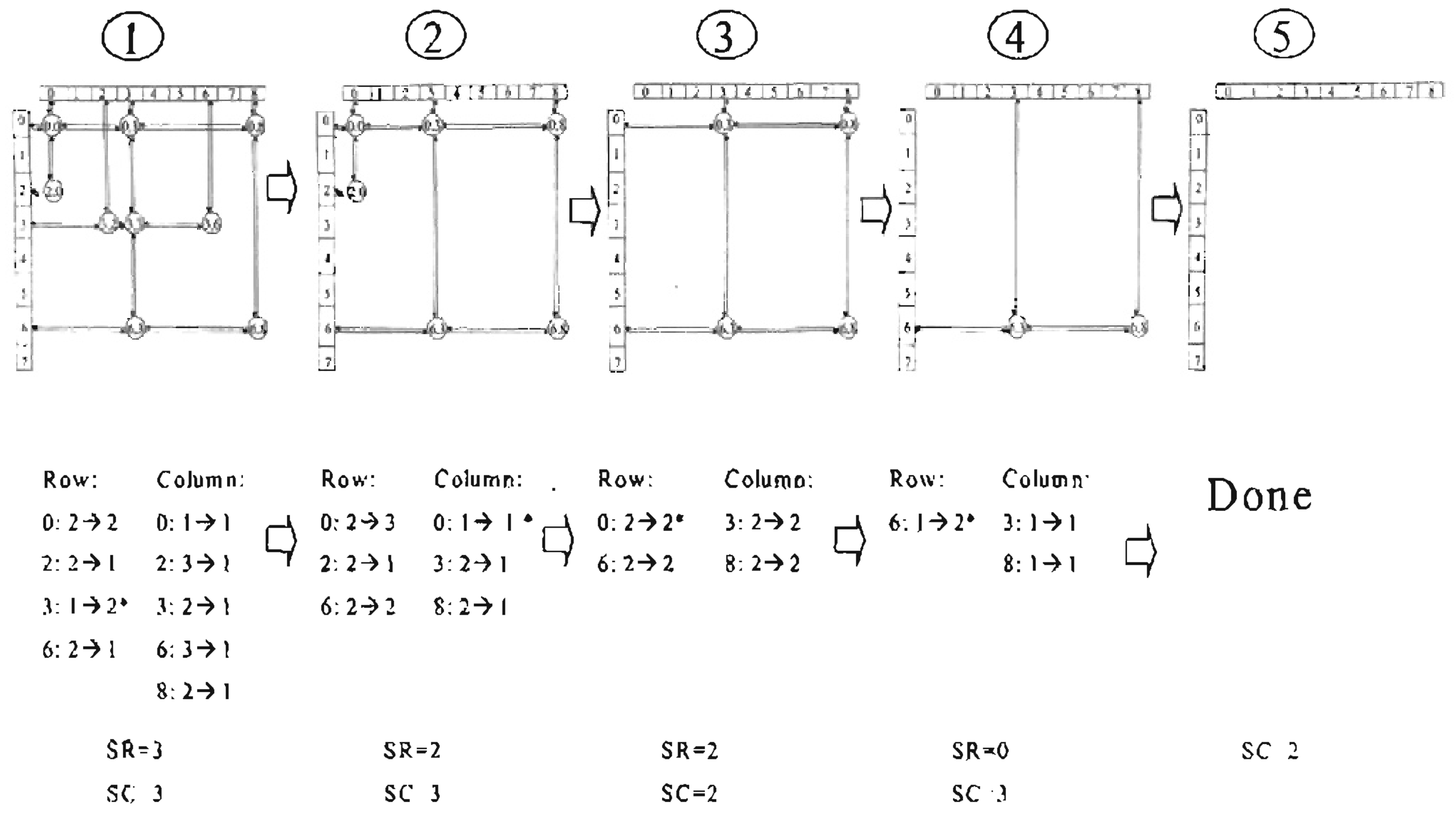


Figure 11. Proposed Algorithm Repair Process

Chapter IV

Preliminary Simulation Results

Theoretically, the algorithm we propose will reduce either time or space requirement to generate the repair solution. We will justify this by comparing it with the Repair-Most algorithm, which is still one of the widely used algorithms. The exhaustive search algorithm surely will be the best algorithm to generate the repair solutions, however, it is not practical for repairing even moderate size of memories. Moreover, it is difficult to implement. Our proposed algorithm is based on the same logic as the heuristic approximation algorithm but with different memory representations to generate repair solutions, thus, the yield of our proposed algorithm will be exactly the same as the yield generated by the heuristic approximation algorithm. While the Repair-Most algorithm is using different approaches to address this issue, it will be more practical to do comparison with Repair-Most algorithm.

1. Our proposed algorithm can repair the redundant memory effectively. [Figure 12, Figure 13, Figure 14, Figure 15].

   Figure 12 and 14 show that the yields of our proposed algorithm have nearly the same yields as the Repair Most algorithm no matter the faults are randomly distributed or clustered. Sometimes, Repair Most algorithm has higher yield than our proposed algorithm. This is because that the faults in memories are randomly generated, so the defect numbers and positions are not constant. The higher yield generated by Repair Most algorithm on some cases does not indicate that the same defect patterns repaired by our proposed algorithm will generate lower yield.

2. For the same fault pattern, the yield generated by proposed algorithm is nearly always higher than the one generated by Repair Most algorithm [Figure 16, Figure 17]. However, on some fault patterns, Repair Most algorithm will generate higher yield than our proposed algorithm. In our test cases of randomly distributed fault patterns, only 2% of memories will get higher yield when they are repaired by Repair Most algorithm rather than our proposed algorithm [Figure 16]. It is around 4% higher when the faults are clustered [Figure 17].
3. Statistically, our proposed algorithm will generate higher yield than Repair Most algorithm [Figure 13, Figure 15]. Figure 13 shows that for memories with randomly distributed faults, the yield repaired by our proposed algorithm is around 4% higher the yield repaired by Repair Most algorithm. For memories with clustered fault distribution, the yield increase is about 2.5% [Figure 15].
4. Our proposed algorithm is more time efficient than Repair Most algorithm when the memory size is large [Figure 18]. When the memory size is small, the Repair Most algorithm is more efficient than our proposed algorithm. However, when the memory size is large, the Repair Most algorithm is extremely inefficient. In our test cases, we randomly distribute 9000 faults in redundant memories with 100 spare rows and spare columns each. When the memory size is

bigger than 168Mb, it is terribly slow, as in this stage, the testing equipment has run out of real memory, and the slow accessing time of disk (as virtual memory) compared with the fast access time of real memory is the confounding factor that account for the slowness. When the memory size is bigger than 379Mb, it can't allocate enough memory on the testing equipment to generate the repair solution. Our proposed algorithm can efficiently generate solution for memories up to 4.31Gbs. Since the running time complexity of our proposed algorithm is O((M+N)*(SR+SC)), and in this test case, we fixed the SR and SC, the time complexity will be only affected by M and N, which is the number of rows and the number of columns that has defects in the memory respectively. As there are fixed 9000 defects in the simulated memory, and M and N will increase as the memory size increase. However, there are at most 9000 defective rows and 9000 defective columns, which means that the upper bound of M and N are 9000, this upper bound is corresponded to the stable stage in figure 18.

5. Our proposed algorithm repair process will use less memory than repair most algorithm [Figure 19]. Figure 19 shows the theoretical memory requirement of repair most process and proposed process.
6. The simulation results under different fault models and different conditions [Figure 20, Figure 21, Figure 22, Figure 23, Figure 24, Figure 25] show that the proposed algorithm nearly always has higher yield than repair most algorithm.

7. Theoretically the proposed algorithm will have the same yield as heuristic approximation algorithm, however, the space it required reduces about one half.
8. As the proposed algorithm is a polynomial approximation algorithm and its computational time complexity is O((SR+SC)*(M+N)), it is more efficient than the exhaustive algorithm whose computational time complexity is NP-complete.

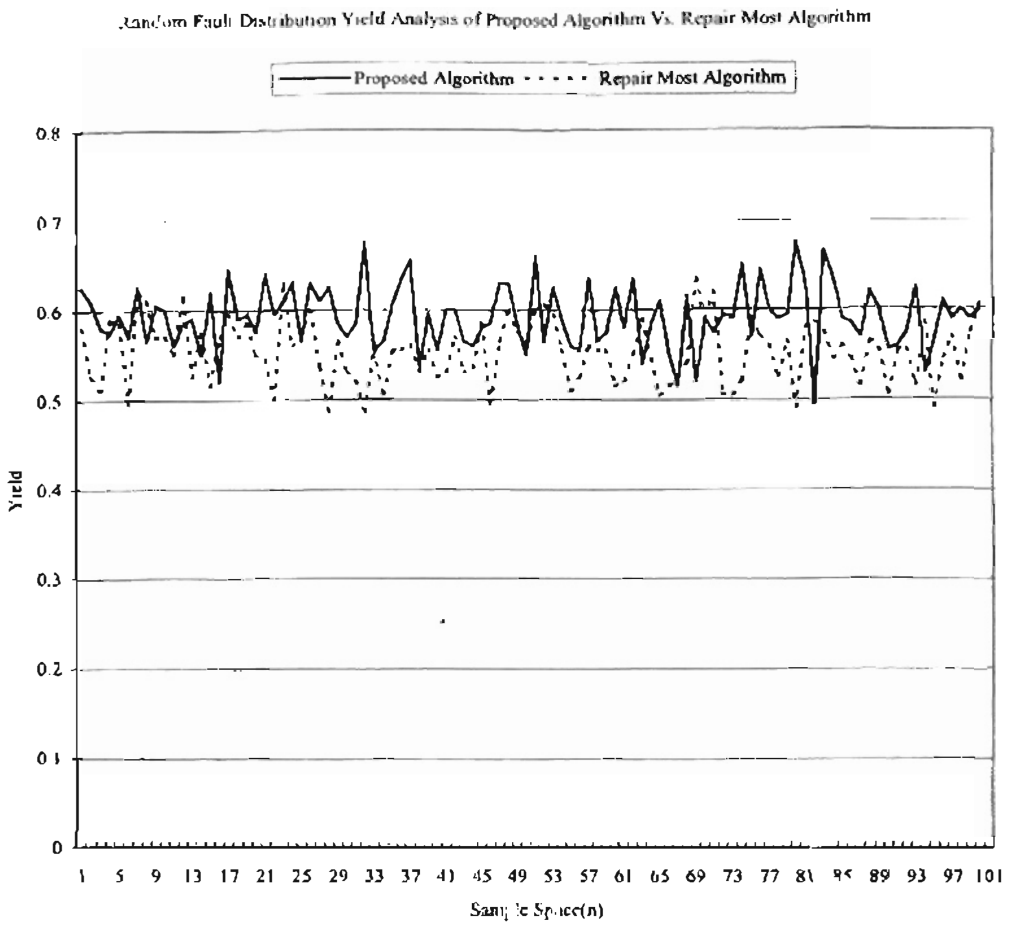


Figure 12. Yield Analysis of Repair Most Vs. Proposed Repair Algorithm on Redundant Memory with Random Fault Distribution (Row=Column=100, Spare Row=Spare Column=20, P=0.6%)

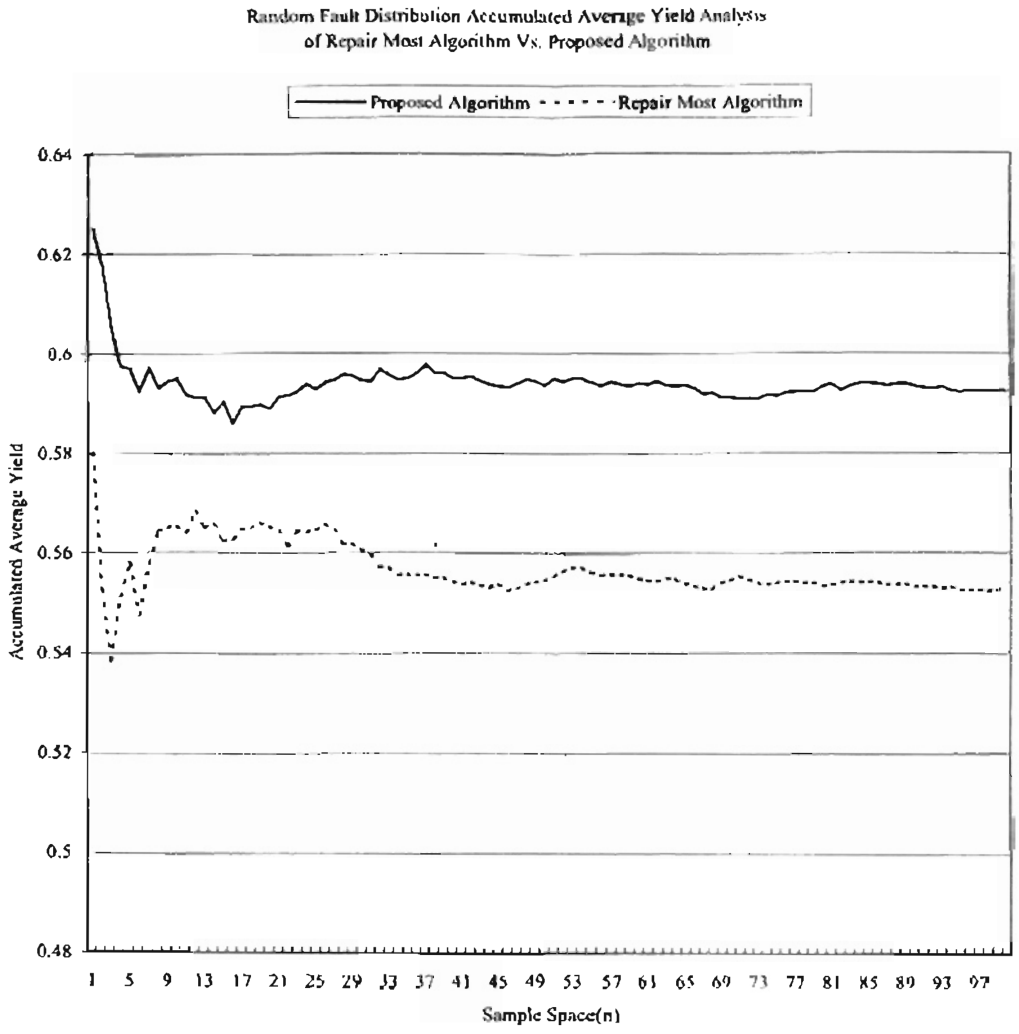


Figure 13. Accumulated Average Yield Analysis of Repair Most Vs. Proposed Repair Algorithm on Redundant Memory with Random Fault Distribution (Row=Column=100, Spare Row=Spare Column=20, P=0.6%)

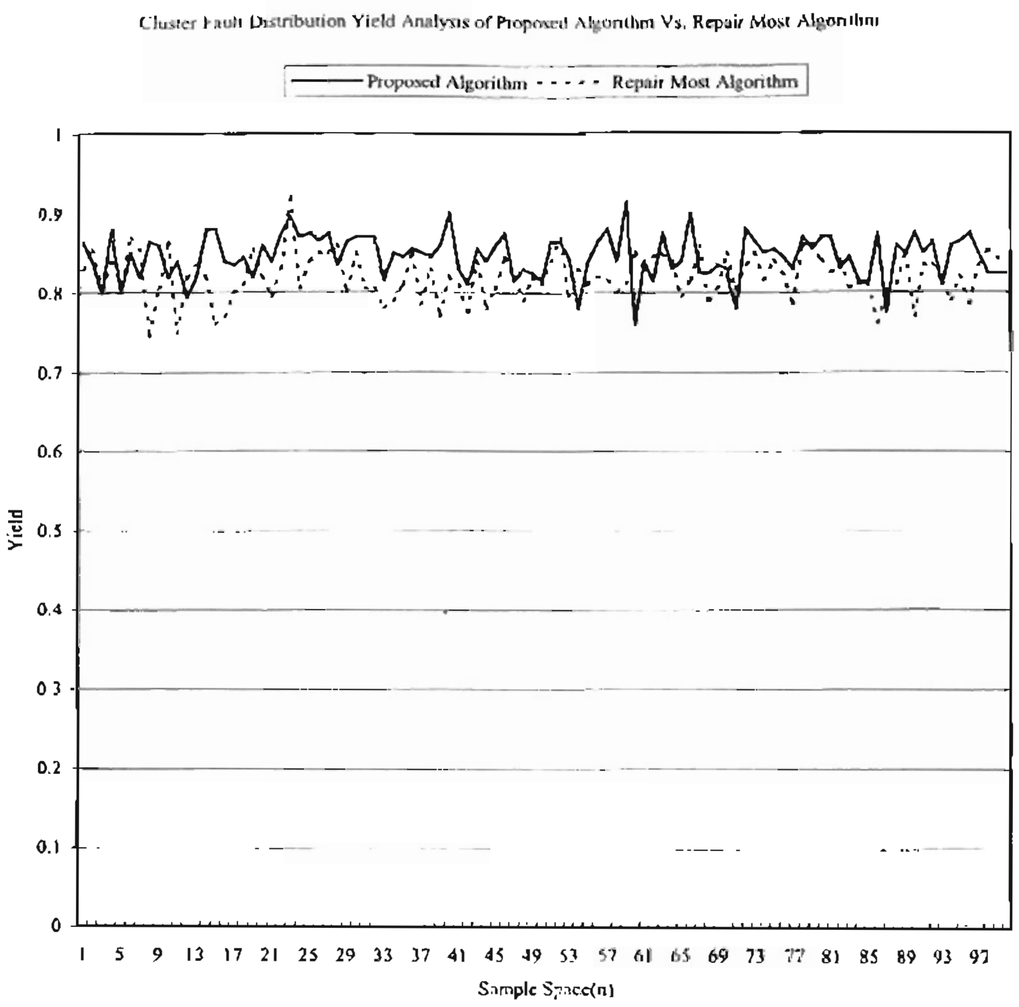


Figure 14. Yield Analysis of Repair Most Vs. Proposed Repair Algorithm on Redundant Memory with Clustered Fault Distribution (Row=Column=100, Spare Row=Spare Column=20, P=0.6%, α=3.8274, λ=1.934)

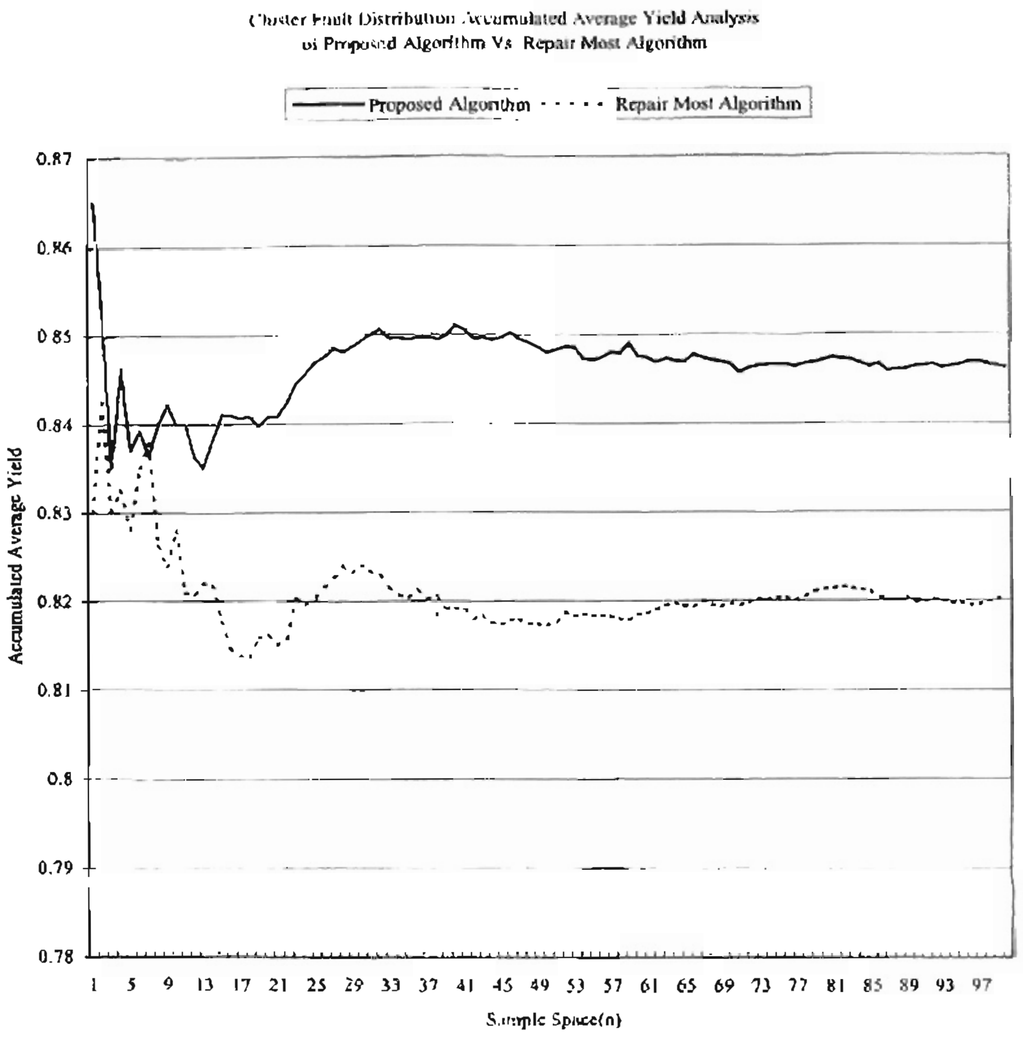


Figure 15. Accumulated Average Yield Analysis of Repair Most Vs. Proposed Repair Algorithm on Redundant Memory with Clustered Fault Distribution (Row=Column=100, Spare Row=Spare Column=20, P=0.6%, α=3.8274, λ=1.934)

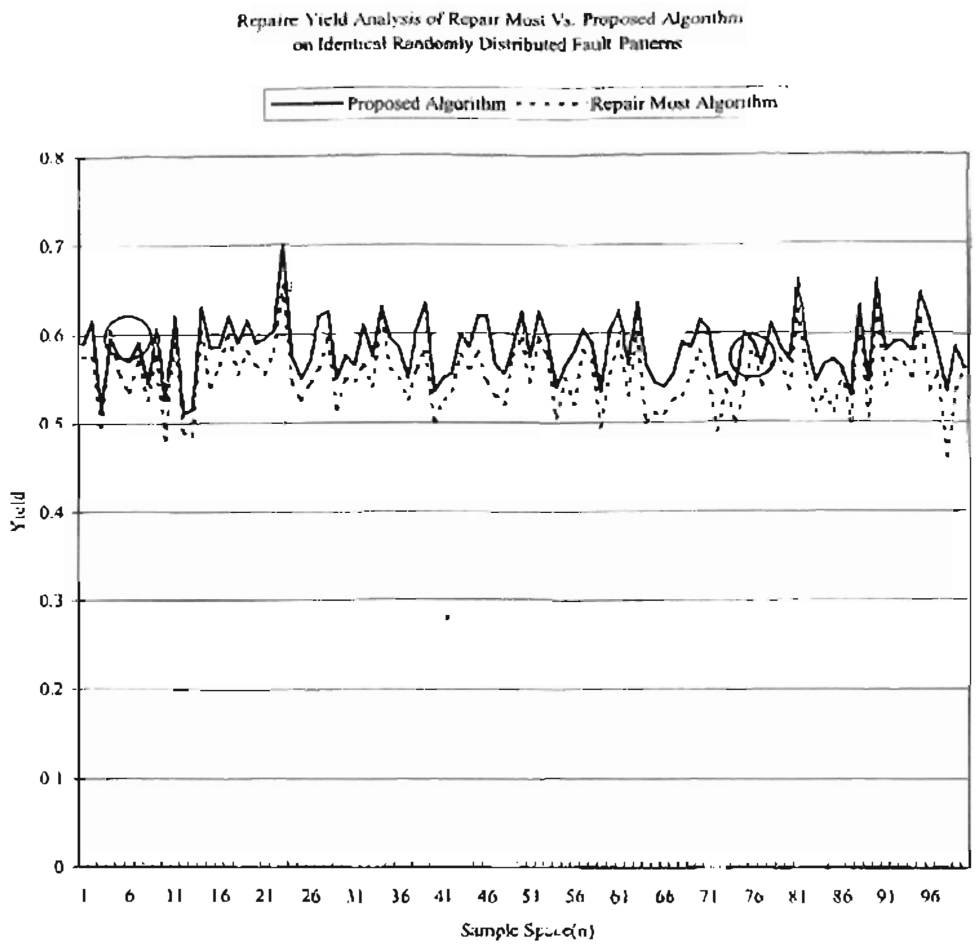


Figure 16. Yield Analysis of Repair Most Vs. Proposed Repair Algorithm on Redundant Memory with Identical Random Fault Distribution Patterns (Row=Column=100, Spare Row=Spare Column=20, P=0.6%)

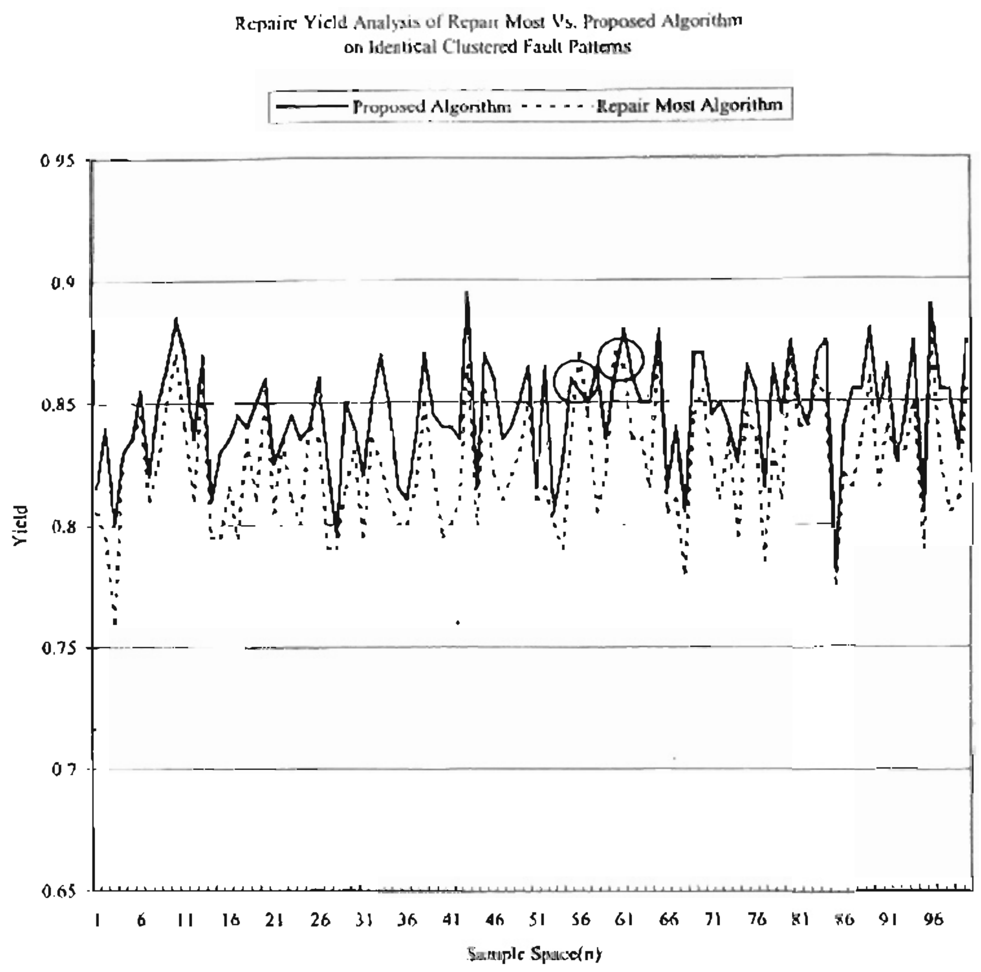


Figure 17. Yield Analysis of Repair Most Vs. Proposed Repair Algorithm on Redundant Memory with Identical Clustered Fault Distribution Patterns (Row=Column=100, Spare Row=Spare Column=20, P=0.6%, α=3.8274, λ=1.934)

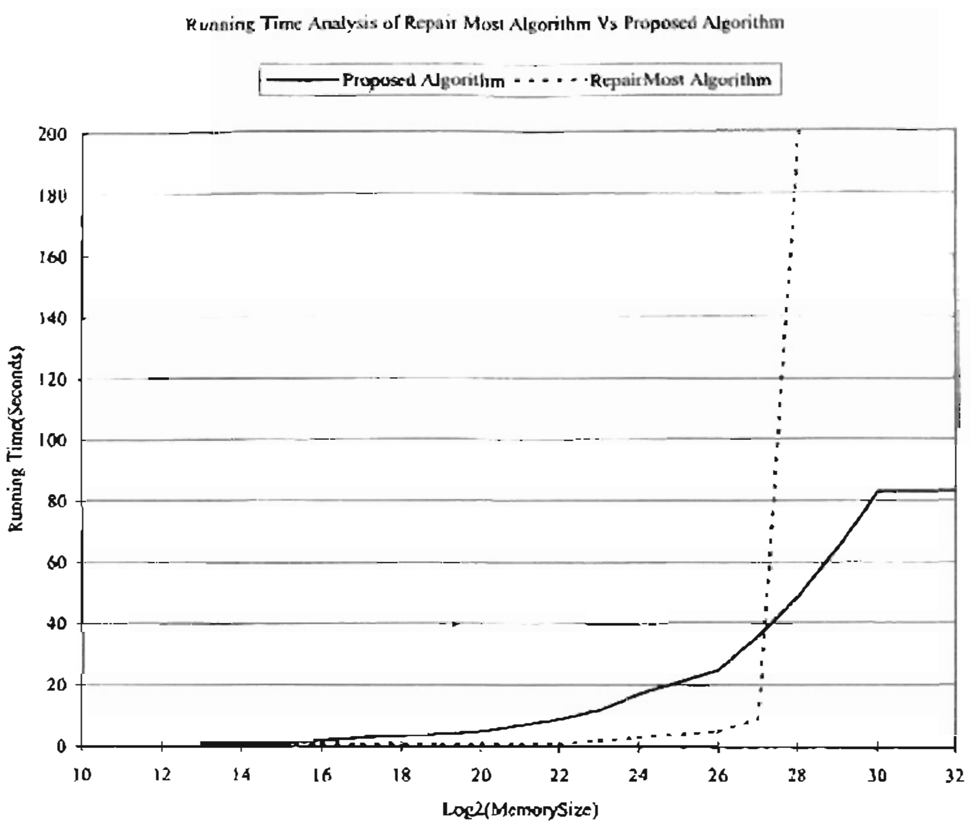


Figure 18. Running Time Analysis of Repair Most Vs. Proposed Repair Algorithm on Different sizes of Redundant Memories with Fixed Defects and Spare Lines. (Defects=9000, Spare Row=Spare Column=100)

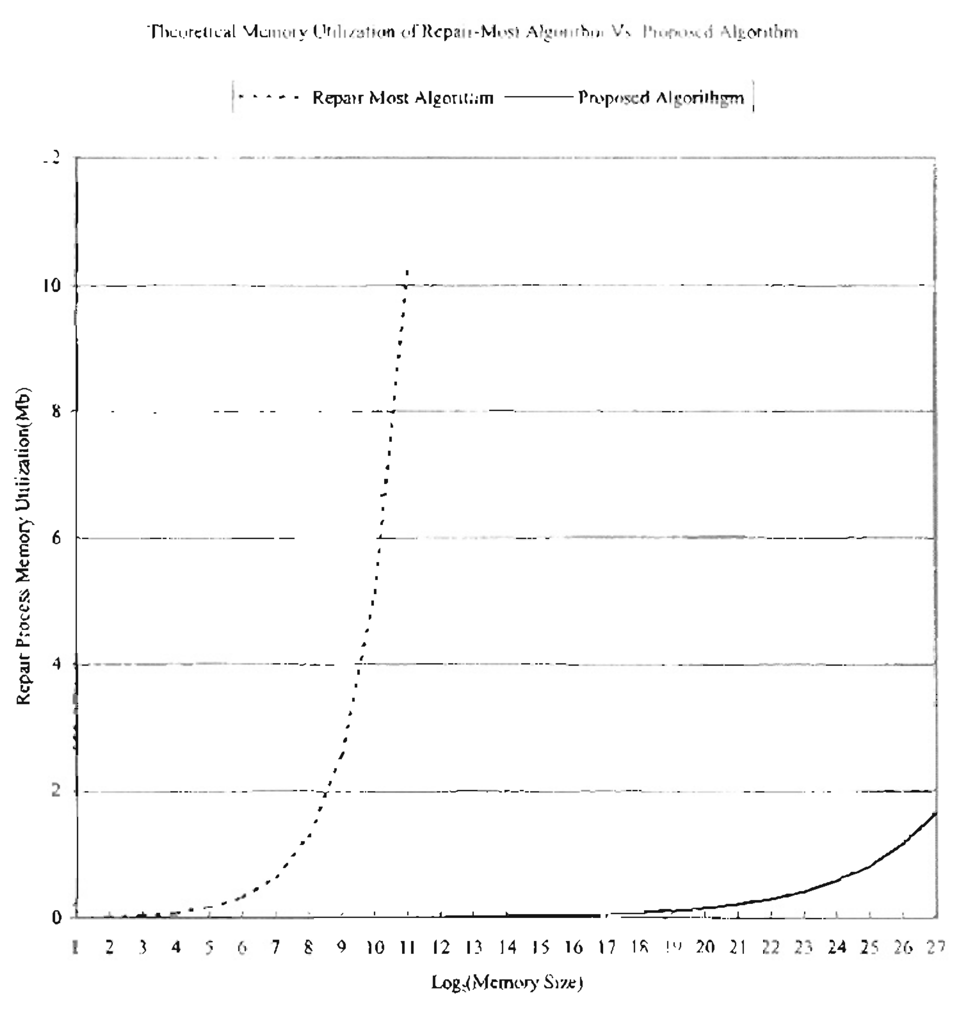


Figure 19. Theoretical Repair Process Memory Utilization Analysis of Repair Most Vs. Proposed Repair Algorithm on Different sizes of Redundant Memories with Fixed Defects and Spare Lines. (Defects=9000, Spare Row=Spare Column=100)

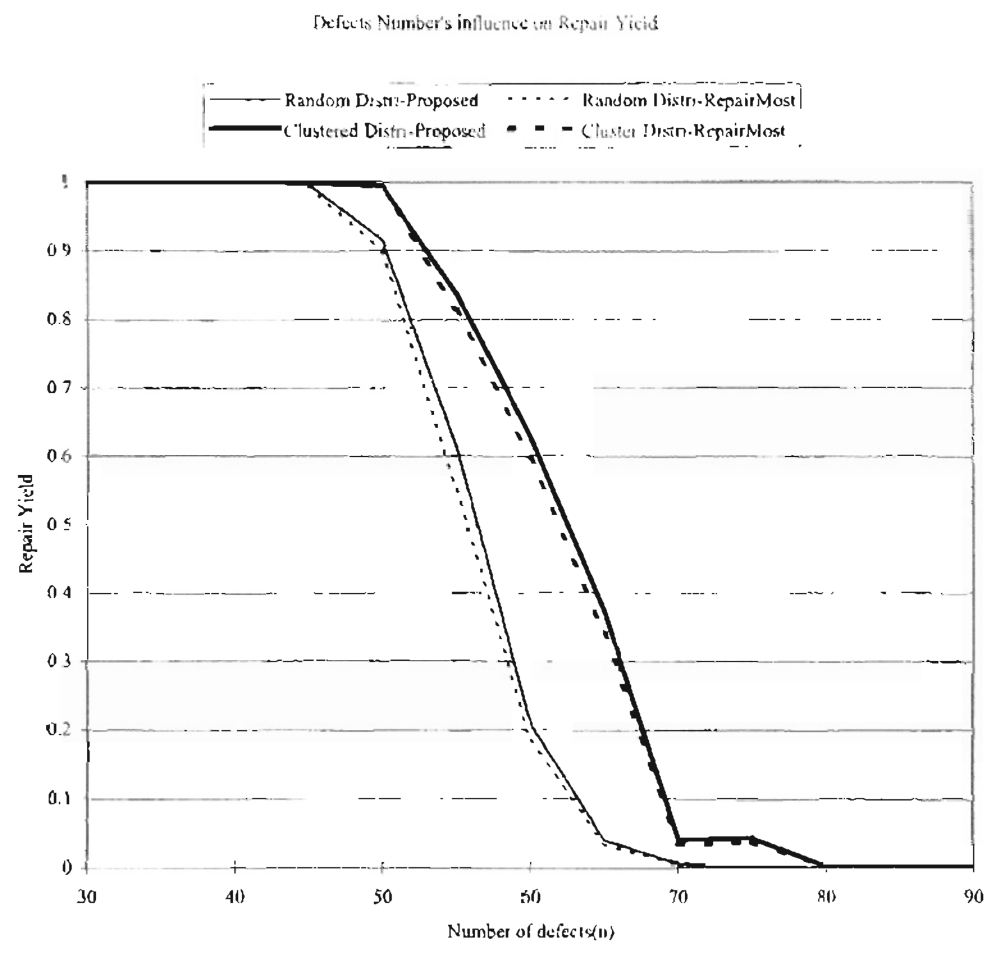


Figure 20. The influence of Defect Number's on Repair Yield. (Random Distribution Parameter: Row=Column=100, SR=SC=20; Clustered Distribution Parameter: Row=Column=100, SR=SC=20, α=3.8274, λ=1.934)

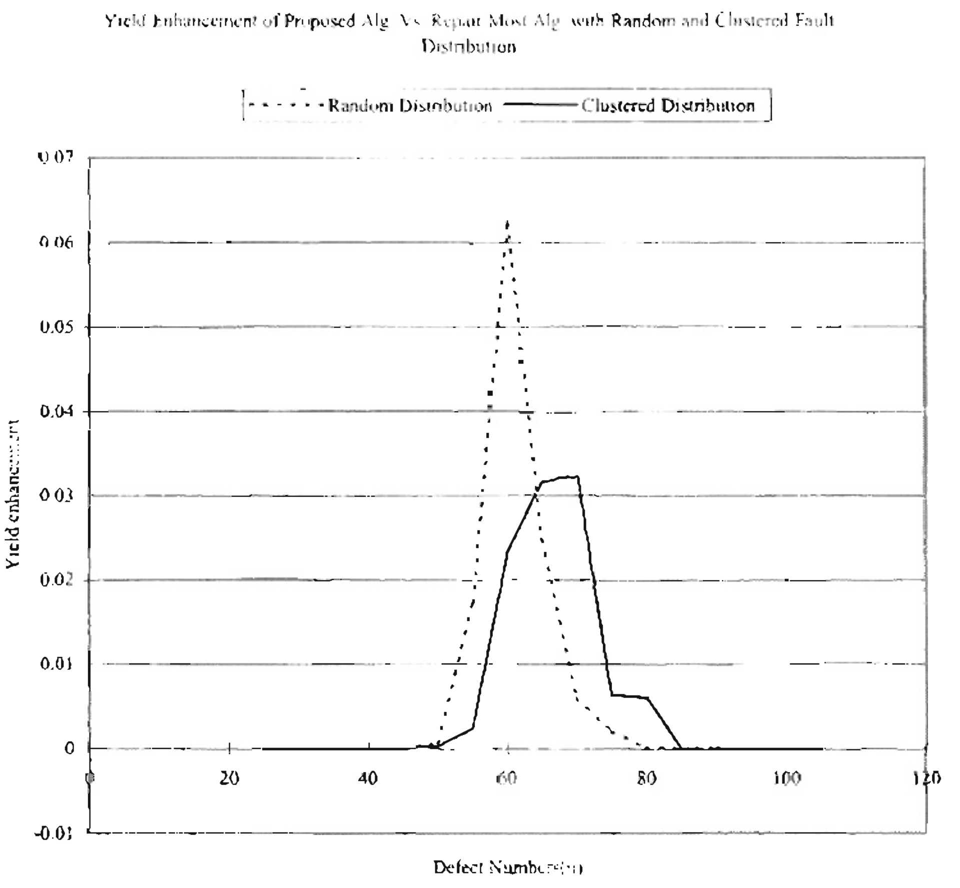


Figure 21. Yield enhancement of the proposed algorithm in contrast with repair-most algorithm (Random Distribution Parameter: Row=Column=100, SR=SC=20; Clustered Distribution Parameter: Row=Column=100, SR=SC=20, α=3.8274, λ=1.934)

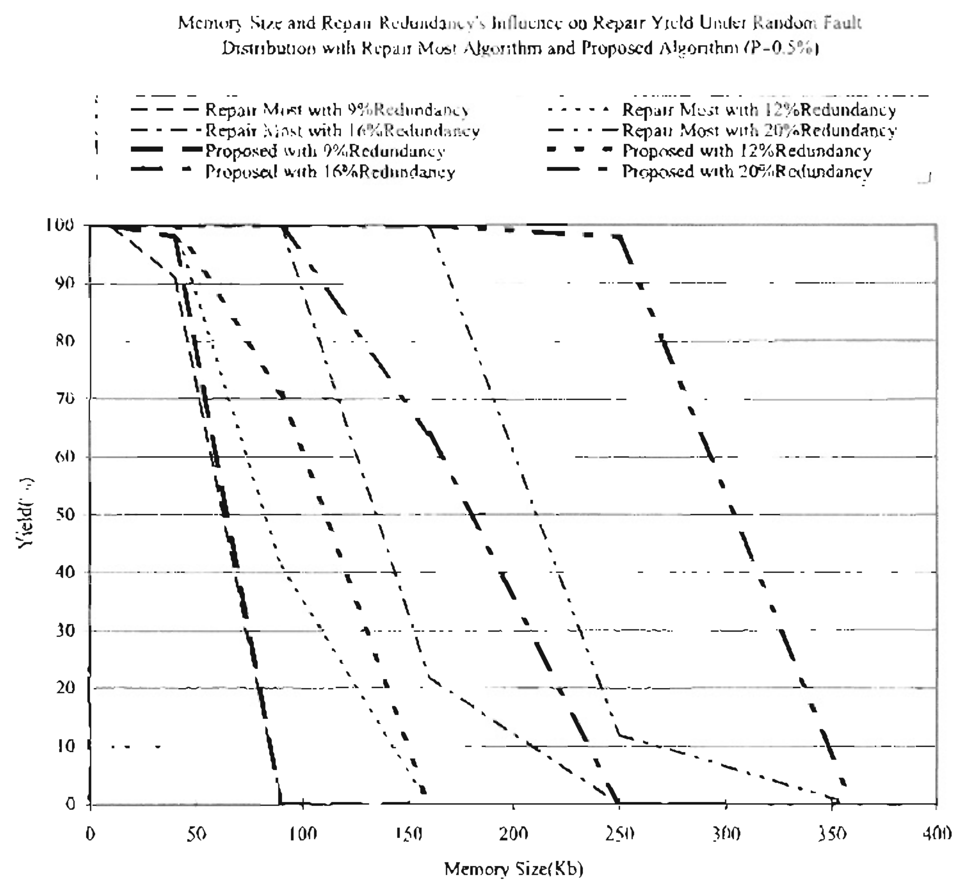


Figure 22. Yield analysis results of Repair-Most with different size of memory and different repair redundancy under random fault Distribution. (P · 0.5%)

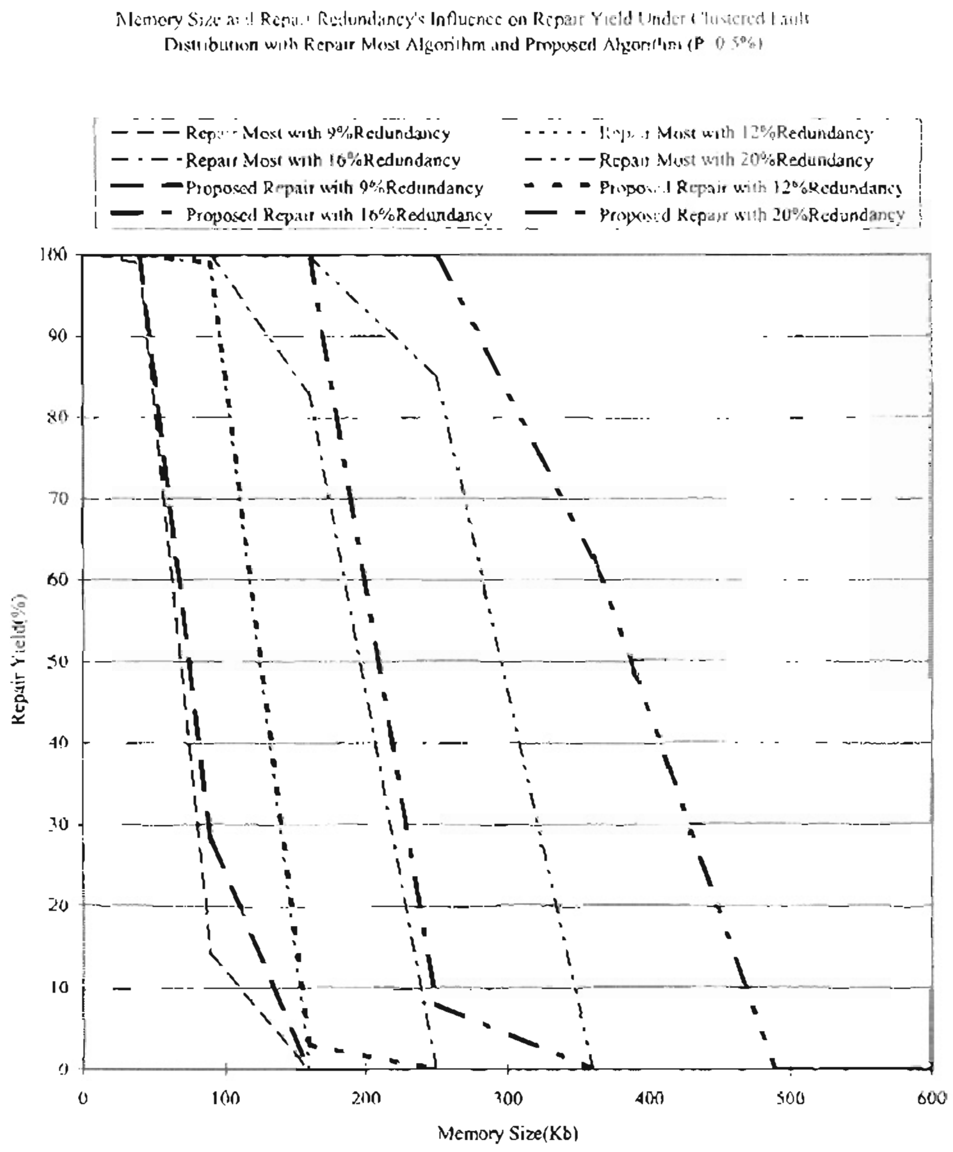


Figure 23. Yield analysis results of Repair-Most with different size of memory and different repair redundancy under clustered fault Distribution. (P=0.5%)

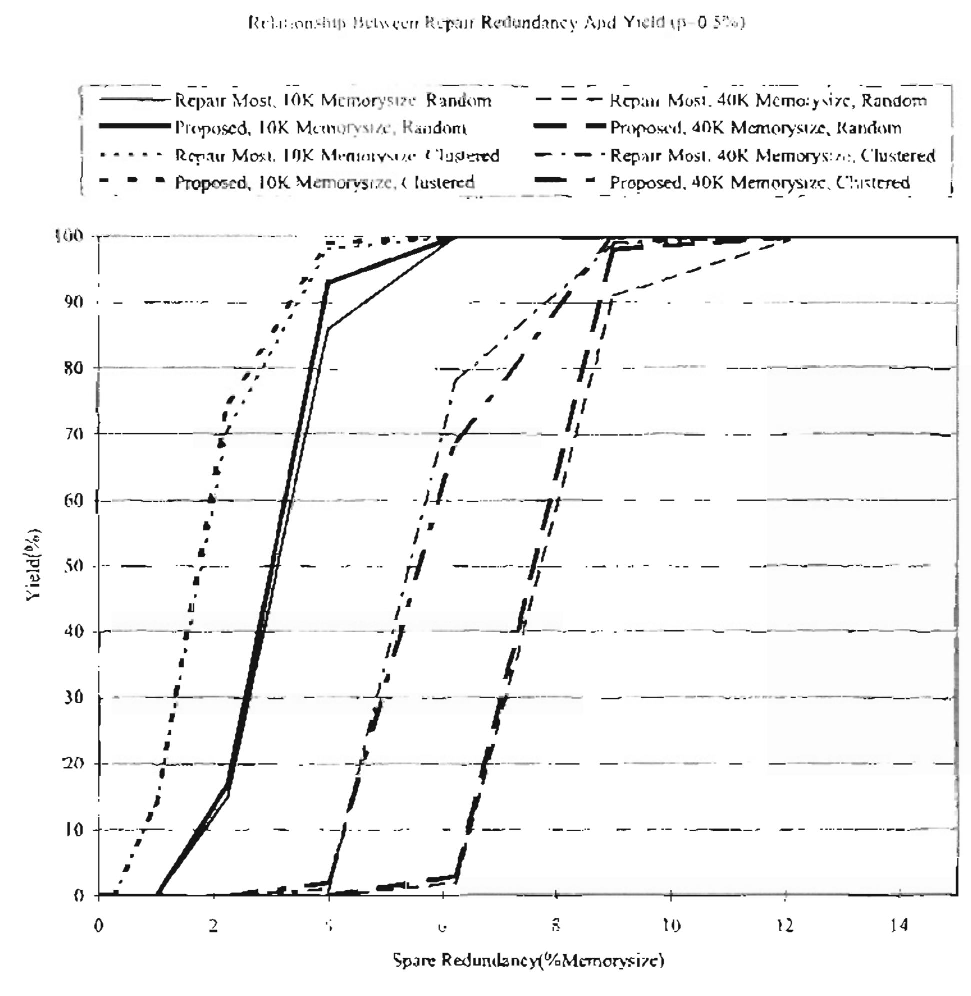


Figure 24. Relationship between Repair Redundancy and Repair Yield. (P=0.5%)

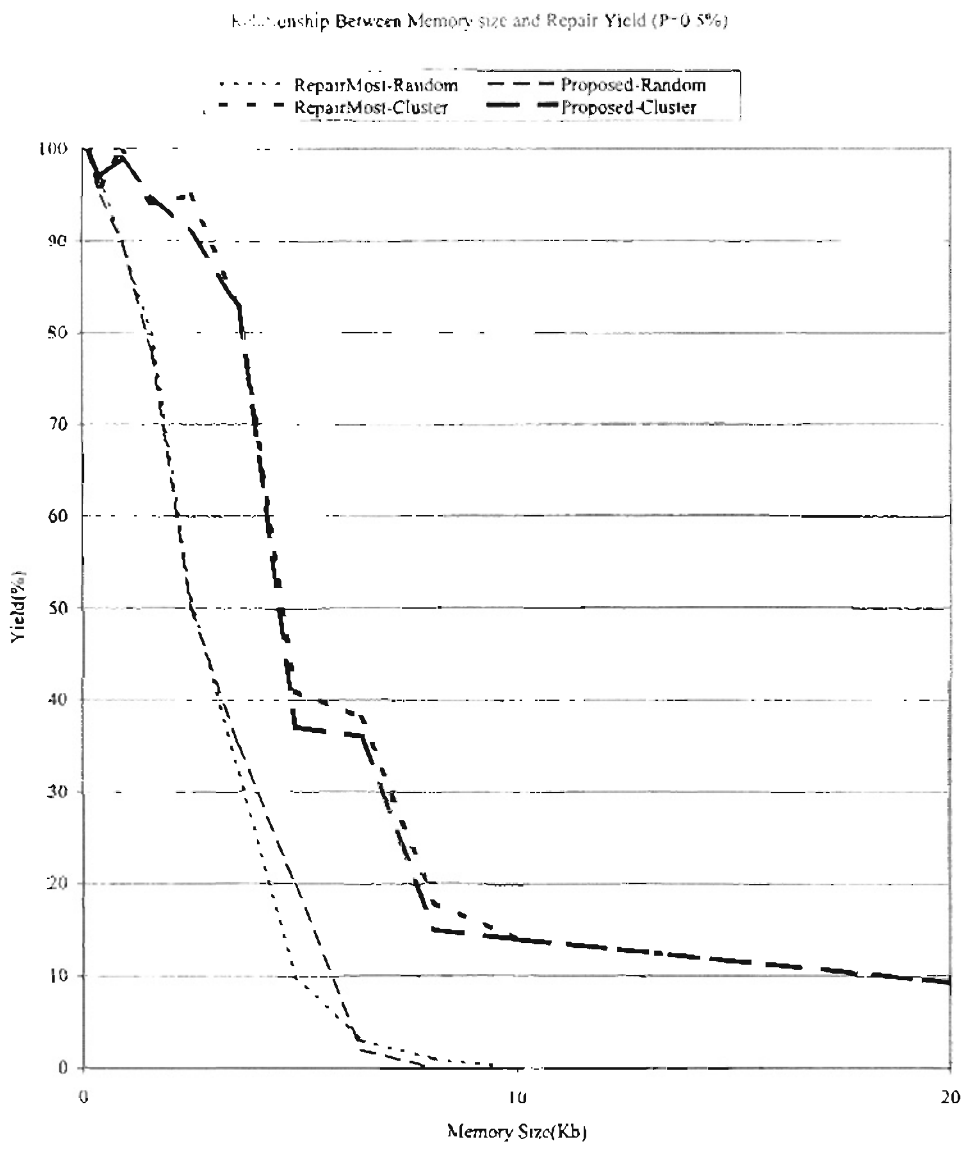


Figure 25. Relationship between Memory Size and Repair Yield. (P=0.5% SR=SC=1%MemorySize)

Chapter V

Conclusion

The algorithm presented in this thesis is efficient and effective by using *two-dimensional array of linked list* to represent the memory with defects. The algorithm also employs a *greedy approach* to repeatedly find and repair the row or column for the greatest yield. The computational space complexity of the proposed algorithm is O(R+C+E), (where R and C are the number of rows and columns of redundancy memory, respectively, and E is the number of nodes, or the number of edges in graph representation). This shows that the computational space is bounded either on the number of defects on the memory when the memory cell defective rate is not very small (i.e. E>>R+C); or is bounded on the sum of the number of rows and columns when the memory cell defective rate is small (R+C>>E). Even though the solution generated by the proposed algorithm is not always optimal, its computational time complexity is O((SR+SC)*(M+N)) (where SR or SC are the number of spare rows or spare columns respectively, and M or N are the number of rows or columns that have faulty memory cells, respectively). Hence, the proposed algorithm can compute the repair process in polynomial time, which is a great accomplishment compared with the conventional NP-complete exhaustive algorithms. The proposed algorithm has revealed a significant yield improvement by up to 5% compared with another polynomial approximation algorithm, the repair-most algorithm.

When there are spare rows or spare columns, and there are defective memory cells, the proposed algorithm greedily finds the rows or columns in polynomial time for the greatest repair yield without checking whether the solution is optimal or not.

Reference

[1] Ciciani, B., "Manufacturing yield evaluation of VLSI/WSI systems" IEEE Computer Society Press, Los Alamitos, CA, 1995.

[2] Day, J., "A Fault-Driven Comprehensive Redundancy Algorithm," Design & Test of Computers, IEEE, vol. 2, pp. 35-44, Jun. 1985.

[3] Ernst, R., P. Nowottnick, "Fault Tolerant VLSI Design With Functional Block Redundancy", Computer Design: VLSI in Computers and Processors, 1991. ICCD '91. Proceedings., 1991 IEEE International Conference on, Cambridge, MA, USA, pp 432 –436, 1991.

[4] Kikuda, S., et al., "Optimized Redundancy Selection Based on Failure-Related Yield Model for 64Mb DRAM and Beyond", Solid-State Circuits Conference, 1991. Digest of Technical Papers. 38th ISSCC., 1991 IEEE International, San Francisco, CA, USA, Feb. 1991.

[5] Koren I., and Z. Koren, "Analysis of a Hybrid Defect-Tolerance Scheme for High-Density Memory Ics", Proceedings of the 1997 Workshop on Defect and Fault-Tolerance in VLSI Systems, Mar. 1997.

[6] Koren I., et al., "A Unified Negative Binomial Distribution For Yield Analysis of Defect Tolerant Circuits." IEEE Trans. Computers, Vol. 42, No. 6, pp: 724-733, June 1993.

[7] Kuo, S.Y. and W. K. Fuchs, "Efficient Spare Allocation for Reconfigurable Arrays," Design & Test of Computers, IEEE, vol 4, pp.24-31, Jan. 1987.

[8] Kuo, S.-Y., W. K. Fuchs, "Modelling and Algorithms for Spare Allocation in Reconfigurable VLSI." Computers and Digital Techniques, IEE Proceedings, Vol.139, pp: 323 –328, July 1992.

[9] Kuo, S.-Y., and W. K. Fuchs, "Modelling and Algorithms for Spare Allocation in Reconfigurable VLSI," Computers and Digital Techniques, IEE Proceedings, Vol.139, pp.323-328, July 1992.

[10] Park, N., E. Lombardi, "Repair of Memory Arrays by Cutting," Memory Technology, Design and Testing, pp. 124 –130, Aug. 1998

[11] Park, N., F. Lombardi, V. Piuri, "Testing and Evaluating the Quality-level of Stratified Multichip Module Instrumentation," IEEE Transactions on Instrumentation and Measurement, Vol 50, pp. 1615 –1624, Dec. 2001.

[12] Tarr, M., D. Boudreau, and R., Murphy, "Defect Analysis System Speeds Test and Repair of Redundant Memories," Electronics, pp.175-179, Jan. 1984.

[13] Timothy M., et al., "A Discussion of Yield Modeling With Defect Clustering, Circuit Repair, and Circuit Redundancy." IEEE Trans. Semiconductor Manufacturing, vol. 3, pp.116-127, Aug. 1990

[14] Warner R. M., "Applying a Composite Model to The IC Yield Problem." IEEE J. Solid-State Circuits, vol. SC-9, pp.86-95, June 1974.

[15] Wey, C.L., and F. Lombarrdi,"On The Repair of Redundant RAMs," IEEE Trans. Computer-Aided Design, pp.222-231, Mar. 1987.

[16] Yamagata T. et al., "A Distributed Globally Replaceable Redundancy Scheme for Sub-Half-micron ULSI Memories and Beyond," Solid-State Circuits, IEEE Journal of, vol. 31, pp.195-201, Feb. 1996.

[17] Yoo J-H. et al., "A 32-Bank 1Gb Self-Strobing Synchronous DRAM with 1GB/s Band-width," Solid-State Circuits, IEEE Journal of, vol. 31, pp. 1635-1643, Nov. 1996.

Appendix

```
/*
All the programs are coded in C++ and can be compiled by Visual C++ 6.0
and Visual C++ in Visual studio.net. All the simulations in this thesis are
running under the following conditions. Platform: Command Prompt of
Windows 2000 Professional with SP3. When the program is running, no other
activities are performed until the test process is done.
*/

/ *
  *  Main Procedure "main.cpp"
  *  by Song Gao
  *  Graduate Student
  *  Computer Science Department
  *  Oklahoma State University
  *  Stillwater, OK, 74075
  */
#include "iostream.h"
//#include "fstream.h"
//ofstream output("output.txt",ios::out|ios::app);
#include "demo.h"
/* See the file README.txt for information on compiling this program */
#include "math.h"
#include "stdlib.h"
```

```
int main(int argc, char *argv[])
{
        unsigned long R,C,SR,SC,seed,count1,count2;
        float P;
        if(argc<5) { cout<<"Command Line Parameter Error!"<<endl;  exit(2);}
        R=strtoul(argv[1],NULL,10);
        C=strtoul(argv[2],NULL,10);
        SR=strtoul(argv[3],NULL,10);
        SC=strtoul(argv[4],NULL,10);
        P=atof(argv[5]);
        seed=strtoul(argv[6],NULL,10);

        ArrayOfLinkedList *Matrix1=new ArrayOfLinkedList(R,C,SR,SC,P);
        if((Matrix1==NULL))
        {
              cout<<"Out of Memory"<<endl;
              exit(2);
        }
        Matrix1->DefectGeneration(0,seed);
        Matrix1->DefectParamInitialization();
        /*Matrix->MemoryDefectDisplay();*/
        count1=Matrix1->ProposedRepairSolution();
        delete Matrix1;
        MemoryArray *Matrix2=new MemoryArray(R,C,SR,SC,P);
        if((Matrix2==NULL))
        {
            cout<<"Out of Memory"<<endl;
            exit(2);
        }
        RepairMost Solution;
        Matrix2->DefectGeneration(0,seed);
        /*Matrix->DensityMap();*/
        //Matrix2->MemoryDefectDisplay();
        Solution.Initialization(*Matrix2);
        count2=Solution.RepairMostSolution(*Matrix2);
        return count1*10+count2;
}

/*
*   Header File "header.h"
*   by Song Gao
*   Graduate Student
*   Computer Science Department
*   Oklahoma State University
*   Stillwater, OK, 74075
*/

#define NULL 0
#include <stdlib.h>
#include <stdio.h>
```

```
#include "iostream.h"

#include "fstream.h"

#include <list>

#include <vector>

using namespace std;


class IndexCount

{

public:

        IndexCount(void);

        ~IndexCount(void);

        unsigned long Index;

        unsigned long Count;

        unsigned long Sub;

};


class Node

{

public:

        Node(unsigned long x,unsigned long y);

        ~Node(void);

        Node* Left;

        Node* Right;

        Node* Up;

        Node* Down;
        unsigned long x;
        unsigned long y;
        Node(void);
};

class ArrayOfLinkedList
{
public:
        ArrayOfLinkedList(unsigned long x,unsigned long y,unsigned long
        sr,unsigned long sc, float p);
        ~ArrayOfLinkedList(void);
        void DefectGeneration(int mode, unsigned long seed);
        IndexCount* FindMinimalIndex(unsigned long index, int mode);
        int MatrixAddNode(unsigned long x, unsigned long y);
        int MatrixDelColNode(unsigned long Col, list<unsigned long> & DOR);
        int MatrixDelRowNode(unsigned long Row, list<unsigned long>&
            DOC);
        int MemoryDefectDisplay(void);
        unsigned long Row;
        unsigned long Col;
        unsigned long SR;
        unsigned long SC;
        float Rate;
        unsigned long*RowArray;
        unsigned long*ColArray;
        Node* RowList;
        Node* ColList;
        list<unsigned long> DOR;
        list<unsigned long> DOC;
        int DefectParamInitialization(void);
        // List Iterator
        list<unsigned long>::iterator cl;
        // Proposed Reapir Solution
        int ProposedRepairSolution(void);
};

class MemoryArray
{
public:
        MemoryArray(unsigned long R,unsigned long C,unsigned long
              SR,unsigned long SC,float P);
        ~MemoryArray(void);
public:
        // Memory Representation
        unsigned long**MemoryMatrix;
        // Number of Rows of the memory
        unsigned long Row;
        // Number of Columns of memory
        unsigned long Columns;
        // Row Array of defective cells counter
        unsigned long *RowArray;
        // Column array of defective cell counter
        unsigned long *ColArray;
        // Spare row redundancy
        unsigned long SpareRow;
        // Spare Column Redundancy
        unsigned long SpareColumn;
        // Defective rate
        float Rate;
public:
        // Generate the memory defect pattern.
        int DefectGeneration(int mode,unsigned long seed);
        void MemoryDefectDisplay(void);
        void Reset(void);
        void DensityMap(void);
};

class RepairMost
{
public:
        RepairMost(void);
        ~RepairMost(void);
protected:
        // Rows that have defects
        list<unsigned long> DOR;
        // Columns that have defects
        list<unsigned long> DOC;
        list <unsigned long>::iterator cl;
public:
        // Initialization of Row and Column defective array
        void Initialization(MemoryArray &Matrix);
        // Repair Most Soution of Defective Memory
        int RepairMostSolution(MemoryArray& Matrix);
};
/ *
*   Implementation file  "procedure.cpp"
*   by Song Gao
*   Graduate Student
*   Computer Science Department
*   Oklahoma State University
*   Stillwater, OK, 74075
*/
#include "iostream.h"
#include "fstream.h"
ofstream out("out.txt",ios::out|ios::app);
#include "demo.h"
#include <time.h>
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>
```

```
Node::Node(unsigned long a,unsigned long b)
: Left(NULL)
, Right(NULL)
, Up(NULL)
, Down(NULL)
, x(a)
, y(b)
{
}

Node::~Node(void)
{
}

Node::Node(void)
: Left(NULL)
, Right(NULL)
, Up(NULL)
, Down(NULL)
, x(0)
, y(0)
{
}
```

```
IndexCount::IndexCount(void)
: Index(0)
, Count(0)
, Sub(0)
{
}

IndexCount::~IndexCount(void)
{
}

ArrayOfLinkedList::ArrayOfLinkedList(unsigned long x,unsigned long
        y,unsigned long sr,unsigned long sc, float p)
: Row(x)
, Col(y)
, SR(sr)
, SC(sc)
, Rate(p)
{
        RowArray=new unsigned long [x];
        for(unsigned long i=0;i<x;i++)
              RowArray[i]=0;
        ColArray=new unsigned long [y];
        for(unsigned long j=0;j<y;j++)
              ColArray[j]=0;
        RowList=new Node [x];
        ColList=new Node [y];
        if(RowArray==NULL||ColArray==NULL||RowList==NULL||ColList==
            NULL)
        {
            cout<<"Memory Allocation Error!"<<endl;
            exit(2);
        }
}

ArrayOfLinkedList::~ArrayOfLinkedList(void)
{
        delete [] RowArray;
        delete [] ColArray;
        delete [] RowList;
        delete [] ColList;
}

void ArrayOfLinkedList::DefectGeneration(int mode, unsigned long seed)
{
        if(mode==0)//Rondom Distribution
        {
        //Sampling from a random number generator
        //Random: double gsl_rng  uniform (const gsl_rng * r)
```

```
//This function returns a double precision floating point number
//uniformly distributed in the range [0,1[. The range includes 0.0 but
//excludes 1.0. The value is typically obtained by dividing the result of
//gsl_rng_get(r) by gsl_rng_max(r) + 1.0 in double precision. Some
//generators compute this ratio internally so that they can provide
//floating point numbers with more than 32 bits
//of randomness (the maximum number of bits that can be portably
//represented in a single unsigned long int).
    /*srand(seed);
    for(unsigned long i=0;i<this->Row;i++)
        for(unsigned j=0;j<this->Col;j++)
            if(rand()%10000<this->Rate*10000)
                this->MatrixAddNode(i,j);*/
    const gsl_rng_type * T;
    gsl_rng * r;
    /* create a generator chosen by the environment variable
       GSL_RNG_TYPE */
    srand(seed);
    gsl_rng_env_setup();
    T = gsl_rng_default;
    r = gsl_rng_alloc (T);
    gsl_rng_set(r,rand());
    double u;

    for(unsigned long i=0;i<this->Row;i++)
```

```
                    for(unsigned long j=0;j<this->Col;j++)
                    {
                            u = gsl_rng_uniform (r);
                            if(u*10000<this->Rate*10000)
                                    this->MatrixAddNode(i,j);
                    }
            gsl_rng_free (r);
    }
else      /*Random Fault Cluster Distribution: unsigned int
    gsl_ran_negative_binomial (const gsl_rng * r,
     double p, double n) This function returns a random integer from the
    negative binomial distribution, the number of failures occurring before
    n successes in independent trials with probability p of success. The
    probability distribution for negative binomial variates is, p(k) =
    {\Gamma(n + k) \over \Gamma(k+1) \Gamma(n) } p^n (1-p)^k Note
    that n is not required to be an integer. This routine is from The GNU
    Scientific Library (GSL). Version 1.1, March 2000 Copyright ? 2000
    Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston,
    MA  02111-1307, USA */
    {
      const gsl_rng_type * T;
        gsl_rng * r;
        /* create a generator chosen by the environment variable
              GSL_RNG_TYPE */
        srand(seed);
```

```
gsl_rng_env_setup();
T = gsl_rng_default;
r = gsl_rng_alloc (T);
gsl_rng_set(r,rand());

double p,n,alpha,lamda;
lamda=1.2934;
alpha=3.8274;
int y;
unsigned long a,b;
p=alpha/(alpha+lamda);
a=(unsigned long)(floor(sqrt(lamda/this->Rate)));
b=(unsigned long)(ceil(sqrt(lamda/this->Rate)));
n=alpha;
unsigned long i,j;
for(i=0;i<(unsigned long)(this->Row/a);i++)
        for(j=0;j<(unsigned long)(this->Col/b);j++)
{
        y=gsl_ran_negative_binomial(r,p,n);
        int m=0,n=0;
        for(m=0;m<a;m++)
                for(n=0;n<b;n++)
                        if((gsl_rng_uniform(r)*(a*b))<y)
                                this->MatrixAddNode(a*i+m,b*j+n);
}
```

```
        }
}

IndexCount* ArrayOfLinkedList::FindMinimalIndex(unsigned long index, int
        mode)
{
        IndexCount *IdxCnt=new IndexCount();
        if(IdxCnt==NULL)
        {
            cout<<"Memory Allocation Error!"<<endl;
            exit(2);
        }
        IdxCnt->Sub=index;
        Node *p;
        if(mode==0)
        {
            IdxCnt->Index=this->ColArray[this->RowList[index].Right->y];
            IdxCnt->Count=1;
            p=this->RowList[index].Right;
            p=p->Right;
            while(p)
            {
                if(this->ColArray[p->y]<IdxCnt->Index)
                {
                    IdxCnt->Index=this->ColArray[p->y];
            IdxCnt->Count=1;
        }
        else if(this->ColArray[p->y]==IdxCnt->Index)
            IdxCnt->Count+=1;
        p=p->Right;
    }
    return IdxCnt;
}
else
{
    p=this->ColList[index].Down;
    IdxCnt->Index=this->RowArray[p->x];
    IdxCnt->Count=1;
    p=p->Down;
    while(p)
    {
        if(this->RowArray[p->x]<IdxCnt->Index)
        {
            IdxCnt->Index=this->RowArray[p->x];
            IdxCnt->Count=1;
        }
        else if(this->RowArray[p->x]<IdxCnt->Index)
            IdxCnt->Count+=1;
        p=p->Down;
    }
```

```
		return IdxCnt;
	}
}

int ArrayOfLinkedList::MatrixAddNode(unsigned long x, unsigned long y)
{
	Node *p=new Node(x,y);
	if(p==NULL)
	{
		cout<<"Memory Allocation Error!"<<endl;
		exit(2);
	}
	this->ColArray[y]+=1;
	this->RowArray[x]+=1;
	if(this->RowList[x].Right==NULL)
	{
		this->RowList[x].Right=p;
		p->Left=&this->RowList[x];
	}
	else
	{
		p->Right=this->RowList[x].Right;
		this->RowList[x].Right->Left=p;
		this->RowList[x].Right=p;
		p->Left=&this->RowList[x];
```

```
        }
        if(this->ColList[y].Down==NULL)
        {
            this->ColList[y].Down=p;
            p->Up=&this->ColList[y];
        }
        else
        {
            p->Down=this->ColList[y].Down;
            this->ColList[y].Down->Up=p;
            this->ColList[y].Down=p;
            p->Up=&this->ColList[y];
        }
        return 0;
}

int ArrayOfLinkedList::MatrixDelColNode(unsigned long Col, list<unsigned
        long> & DOR)
{
        this->ColArray[Col]=0;
        Node *N;
        Node *p=this->ColList[Col].Down;
        while(p)
        {
```

```
        if(p->Down==NULL)
        {
                this->ColList[Col].Down=NULL;
        }
        else
        {
                this->ColList[Col].Down=p->Down;
                p->Down->Up=&this->ColList[Col];
        }
        if(p->Right==NULL)
        {
                p->Left->Right=NULL;
        }
        else
        {
                p->Left->Right=p->Right;
                p->Right->Left=p->Left;
        }
        this->RowArray[p->x]-=1;
        if(this->RowArray[p->x]==0)
                DOR.remove(p->x);
        N=p;
        p=p->Down;
        delete N;
}
```

```
        return 0;
}

int ArrayOfLinkedList::MatrixDelRowNode(unsigned long Row, list<unsigned
        long>& DOC)
{
        this->RowArray[Row]=0;
        Node *p,*N;
        p=this->RowList[Row].Right;
        while(p!=NULL)
        {
              if(p->Right==NULL)
              {
                      this->RowList[Row].Right=NULL;
              }
              else
              {
                      this->RowList[Row].Right=p->Right;
                      p->Right->Left=&this->RowList[Row];
              }
              if(p->Down==NULL)
              {
                      p->Up->Down=NULL;
              }
              else
```

```
                {
                        p->Up->Down=p->Down;
                        p->Down->Up=p->Up;
                }

                this->ColArray[p->y]-=1;
                if(this->ColArray[p->y]==0)
                        DOC.remove(p->y);
                N=p;
                p=p->Right;
                delete N;

        }
        return 0;
}

int ArrayOfLinkedList::MemoryDefectDisplay(void)
{
        unsigned long count=0;
        unsigned long i;
        Node *p;
        for(i=0;i<this->Row;i++)
                if(this->RowArray[i])
                {
                        p=this->RowList[i].Right;
                while(p)
                {
                        out<<"("<<p->x<<","<<p->y<<") ";
                        count++;
                        p=p->Right;
                }
                out<<endl;
        }
        for(i=0;i<this->Row;i++)
                out<<this->RowArray[i]<<" ";
        out<<endl;
        for( i=0;i<this->Col;i++)
                out<<this->ColArray[i]<<" ";
        out<<endl<<endl;
    return 0;
}

int ArrayOfLinkedList::DefectParamInitialization(void)
{
    unsigned long i;
    for(i=0;i<this->Row;i++)
        if(this->RowArray[i]) this->DOR.push_back(i);
    for( i=0;i<this->Row;i++)
        if(this->ColArray[i]) this->DOC.push_back(i);
    return 0;
```

```
}

// Proposed Reapir Solution
int ArrayOfLinkedList::ProposedRepairSolution(void)
{
        IndexCount *RowCount, *ColCount, *Rtemp;
        while(this->DOR.size())
        {
              RowCount=this->FindMinimalIndex(*DOR.begin(),0);
              for(cl=DOR.begin();cl!=DOR.end();cl++)
              {
                      Rtemp=this->FindMinimalIndex(*cl,0);
                      if(Rtemp->Index<RowCount->Index)
                      {
                              RowCount->Index=Rtemp->Index;
                              RowCount->Count=Rtemp->Count;
                              RowCount->Sub=*cl;
                      }
                      else if(Rtemp->Index==RowCount->Index)
                      {
                              if(Rtemp->Count>RowCount->Count)
                              {
                                      RowCount->Index=Rtemp->Index;
                                      RowCount->Count=Rtemp->Count;
                                      RowCount->Sub=*cl;
```

```
                }
        }
}
ColCount=this->FindMinimalIndex(*DOC.begin(),1);
for(cl=DOC.begin();cl!=DOC.end();cl++)
{
        Rtemp=this->FindMinimalIndex(*cl,1);
        if(Rtemp->Index<ColCount->Index)
        {
                ColCount->Index=Rtemp->Index;
                ColCount->Count=Rtemp->Count;
                ColCount->Sub=*cl;
        }
        else if(Rtemp->Index==ColCount->Index)
        {
                if(Rtemp->Count>ColCount->Count)
                {
                        ColCount->Index=Rtemp->Index;
                        ColCount->Count=Rtemp->Count;
                        ColCount->Sub=*cl;
                }
        }
}
if((RowCount->Index<ColCount->Index)||(RowCount->
         Index==ColCount->Index)&&(RowCount-> Count>=
     ColCount->Count))
{
    if(this->SR>0)
    {
        //cout<<"Row:"<<RowCount->Sub<<endl;
        this->SR-=1;
        this->MatrixDelRowNode(RowCount->Sub,DOC);
        DOR.remove(RowCount->Sub);
    }
    else if(DOC.size()<=this->SC)
    {
        for(cl=DOC.begin();cl!=DOC.end();cl++)
            this->MatrixDelColNode(*cl,DOR);
        DOC.clear();
        DOR.clear();
        return 0;
    }
    else
    {
        for(cl=DOC.begin();cl!=DOC.end();cl++)
            this->MatrixDelColNode(*cl,DOR);
        DOC.clear();
        DOR.clear();
        return 1;
    }
```

```
}
else
{
        if(this->SC>0)
        {
                //cout<<"Col:"<<ColCount->Sub<<endl;
                this->SC-=1;
                this->MatrixDelColNode(ColCount->Sub,DOR);
                DOC.remove(ColCount->Sub);
        }
        else
                if(DOR.size()<=this->SR)
                {
                        for(cl=DOR.begin();cl!=DOR.end();cl++)
                                this->MatrixDelRowNode(*cl,DOC);
                        DOC.clear();
                        DOR.clear();
                        return 0;
                }
                else
                {
                        for(cl=DOR.begin();cl!=DOR.end();cl++)
                                this->MatrixDelRowNode(*cl,DOC);
                        DOC.clear();
                        DOR.clear();
                        return 1;
                    }
                }
            }
            return 0;
}

RepairMost::RepairMost(void)
{
}

RepairMost::~RepairMost(void)
{
}

// Initialization of Row and Column defective array
void RepairMost::Initialization(MemoryArray &Matrix)
{
        for(unsigned long i=0;i<Matrix.Row;i++)
              if(Matrix.RowArray[i]) DOR.push_back(i);
        //for(cl = DOR.begin();cl!=DOR.end();cl++)
        //      out<<"Row Defect array"<<*cl<<endl;
        for(unsigned long j=0;j<Matrix.Columns;j++)
              if(Matrix.ColArray[j]) DOC.push_back(j );
        //for(cl= DOC.begin();cl!=DOC.end();cl++)
```

```
	//	out<<"Column Defect Array"<<*cl<<endl;
}

// Repair Most Soution of Defective Memory
int RepairMost::RepairMostSolution(MemoryArray& Matrix)
{
	vector<unsigned long> R_V;
	//out<<"Repair Solution"<<endl;
	unsigned long i=0,j=0;
	while(DOR.size())
	{
		i=0;j=0;
		for(cl = DOR.begin();cl!=DOR.end();cl++)
		{
			if(Matrix.RowArray[*cl]>Matrix.RowArray[i]) i=*cl;
		}

		for(cl= DOC.begin();cl!=DOC.end():cl++)
		{
			if(Matrix.ColArray[*cl]>Matrix.ColArray[j]) j=*cl;
		}
		if(Matrix.RowArray[i]>=Matrix.ColArray[j])
		{
			if(Matrix.SpareRow>0)
			{ /* out<<endl<<"Row "<<i<<" ";*/
```

```
        Matrix.SpareRow-=1;
        for(cl= DOC.begin();cl!=DOC.end();cl++)
        {
            if(Matrix.MemoryMatrix[i][*cl])
            {
                Matrix.MemoryMatrix[i][*cl]=0;
                Matrix.ColArray[*cl]-=1;
                if(Matrix.ColArray[*cl]==0)
                {
                    R_V.push_back(*cl);
                }
            }
        }
        while(R_V.size())
        {
            DOC.remove(R_V.back());
            R_V.pop_back();
        }
        Matrix.RowArray[i]=0;
        DOR.remove(i);
    }
    else if(DOC.size()<=Matrix.SpareColumn)
        {
            Matrix.SpareColumn-=DOC.size();
            DOC.clear();
            DOR.clear();
            return 0;
        }
    else
    {
        DOR.clear();
        DOC.clear();
        return 1;
    }
}
else
{
    if(Matrix.SpareColumn>0)
    { /* out<<endl<<"Col "<<j<<" ";*/
        Matrix.SpareColumn-=1;
        for(cl= DOR.begin();cl!=DOR.end();cl++)
        {
            if(Matrix.MemoryMatrix[*cl][j])
            {
                Matrix.MemoryMatrix[*cl][j]=0;
                Matrix.RowArray[*cl]-=1;
                if(Matrix.RowArray[*cl]==0)
                {
                    R V.push back(*cl);
                }
                }
            }
            while(R_V.size())
            {
                DOR.remove(R_V.back());
                R V.pop_back();
            }
            //out<<endl;
            Matrix.ColArray[j]=0;
            DOC.remove(j);
        }
        else if(DOR.size()<=Matrix.SpareRow)
            {
                Matrix.SpareRow-=DOR.size();
                DOR.clear();
                DOC.clear();
                return 0;
            }
            else
            {
                DOR.clear();
                DOC.clear();
                return 1;
            }
    }
```

```
        }
    return 0;
}

MemoryArray::MemoryArray(unsigned long R,unsigned long C,unsigned
        long SR,unsigned long SC,float P)
:Row(R)
, Columns(C)
, SpareRow(SR)
, SpareColumn(SC)
, Rate(P)
{
        this->RowArray=new unsigned long [R];
        if(this->RowArray==NULL)
        {
              cout<<"Out of Memory"<<endl;
              exit(2);
        }
        for(unsigned long i=0;i<R;i++)
              this->RowArray[i]=0;
        this->ColArray=new unsigned long [C];
        if(this->ColArray==NULL)
        {
              cout<<"Out of Memory"<<endl;
              exit(2);
        }
        for(unsigned long j=0;j<C;j++)
            this->ColArray[j]=0;
        MemoryMatrix=new unsigned long * [R];
        if(MemoryMatrix==NULL)
        {
            cout<<"Out of Memory"<<endl;
            exit(2);
        }
        for(unsigned long k=0;k<R;k++)
        {
            this->MemoryMatrix[k]=new unsigned long [C];
            if(this->MemoryMatrix[k]==NULL)
            {
                cout<<"Out of Memory"<<endl;
                exit(2);
            }
        }
        for(unsigned long l=0;l<Row;l++)
            for(unsigned long p=0;p<Columns;p++)
                this->MemoryMatrix[l][p]=0;
}

MemoryArray::~MemoryArray(void)
{
        delete [] this->ColArray;
        delete [] this->RowArray;
        for(unsigned long i=0;i<this->Row;i++)
        delete [] this->MemoryMatrix[i];
        delete [] this->MemoryMatrix;
}

// Generate the memory defect pattern.
int MemoryArray::DefectGeneration(int mode,unsigned long seed)
{
        if(mode==0)//Rondom Distribution
        {
            //Sampling from a random number generator
            //Random: double gsl_rng_uniform (const gsl_rng * r)
            //This function returns a double precision floating point number
            //uniformly  distributed in the range [0,1). The range includes 0.0
            //but excludes 1.0. The value is typically obtained by dividing the
            //result of gsl_rng_get(r) by gsl_rng_max(r) + 1.0 in double
            //precision. Some generators compute this ratio internally so that
            //they can provide floating point numbers with more than 32 bits
            //of randomness (the maximum number of bits that can be
            //portably represented in a single unsigned long int).
            const gsl_rng_type * T;
            gsl_rng * r;
```

```
/* create a generator chosen by the
environment variable GSL_RNG_TYPE */
srand(seed);
gsl_rng_env_setup();
T = gsl_rng_default;
r = gsl_rng_alloc (T);
gsl_rng_set(r,rand());
double u;
//cout<<gsl_rng_uniform (r)<<endl;
for(unsigned long i=0;i<Row;i++)
        for(unsigned long j=0;j<Columns;j++)
        {
                u = gsl_rng_uniform (r);
                if(u*10000<this->Rate*10000)
                {
                        this->MemoryMatrix[i][j]=1;
                        this->ColArray[j]+=1;
                        this->RowArray[i]+=1;
                }
                else
                {
                        this->MemoryMatrix[i][j]=0;
                }
        }
gsl_rng_free (r);
```

```
                return 0;
        }
else        /*Random   Fault   Cluster   Distribution:   unsigned   int
        gsl_ran_negative_binomial (const gsl_rng * r, double p, double n)
        This function returns a random integer from the negative binomial
        distribution, the number of failures occurring before n successes in
        independent trials with probability p of success. The probability
        distribution for negative binomial variates is, p(k) = {\Gamma(n + k)
        \over \Gamma(k+1) \Gamma(n) } p^n (1-p)^k Note that n is not
        required to be an integer. This routine is from The GNU Scientific
        Library (GSL). Version 1.1, March 2000 Copyright ? 2000 Free
        Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA
        02111-1307, USA */
        {
            const gsl_rng_type * T;
            gsl_rng * r;

            /* create a generator chosen by the
            environment variable GSL_RNG_TYPE */
            srand(seed);
            gsl_rng_env_setup();
            T = gsl_rng_default;
            r = gsl_rng_alloc (T);
            gsl_rng_set(r,rand());
```

```
double p,n,alpha,lamda;
lamda=1.2934;
alpha=3.8274;
int y;
unsigned long a,b;
p=alpha/(alpha+lamda);
a=(unsigned long)(floor(sqrt(lamda/this->Rate)));
b=(unsigned long)(ceil(sqrt(lamda/this->Rate)));
n=alpha;
//cout<<gsl_ran_negative_binomial(r,p,n)<<endl;
unsigned long i,j;
for(i=0;i<(unsigned long)(this->Row/a);i++)
        for(j=0;j<(unsigned long)(this->Columns/b);j++)
{
        y=gsl_ran_negative_binomial(r,p,n);
        int m=0,n=0;
        for(m=0;m<a;m++)
                for(n=0;n<b;n++)
                {
                        if((gsl_rng_uniform(r)*(a*b))<y)
                        {
                                this->MemoryMatrix[a*i+m][b*j+n]=1;
                                this->ColArray[b*j+n]+=1;
                                this->RowArray[a*i+m]+=1;
                        }
```

```
                                        else this->MemoryMatrix[a*i+m][b*j+n]=0;

                                }

                        }

                        gsl_rng_free (r);

                        return 0;

                }

}


void MemoryArray::MemoryDefectDisplay(void)

{   unsigned long count=0;

        for(unsigned long i=0;i<this->Row;i++)

        {

                for(unsigned long j=0;j<this->Columns;j++)

                        if(this->MemoryMatrix[i][j]==1)

                        {

                                out<<"( "<<i<<","<<j<<" )"<<" ";

                                count++;

                        }

                        if(this->RowArray[i]) out<<endl;

        }

        /*for(unsigned long i=0;i<this->Row;i++)

                out<<this->RowArray[i]<<endl;

        out<<endl<<endl;

        for(unsigned long i=0;i<this->Columns;i++)

        out<<this->ColArray[i]<<endl;*/
        out<<count<<endl<<endl;
}

void MemoryArray::Reset(void)
{
        for(unsigned long i=0;i<this->Row;i++)
        this->RowArray[i]=0;
        for(unsigned long j=0;j<this->Columns;j++)
        this->ColArray[j]=0;
        for(unsigned long k=0;k<Row;k++)
              for(unsigned long l=0;l<Columns;l++)
                      this->MemoryMatrix[k][l]=0;
}

void MemoryArray::DensityMap(void)
{
        for(unsigned long i=0;i<this->Row;i++)
        {
              for(unsigned long j=0;j<this->Columns;j++)
                      if(this->MemoryMatrix[i][j]==1)
                              out<<"*";
                      else out<<" ";
              out<<endl;
        }
        out<<endl<<"End of One Map"<<endl;
```

```
  }
/ *
  *   One of the Test Procedure "test.cpp"
  *   by Song Gao
  *   Graduate Student
  *   Computer Science Department
  *   Oklahoma State University
  *   Stillwater, OK, 74075
  */
#include <process.h>
#include <stdlib.h>
#include <stdio.h>

#include "IOSTREAM.H"
#include "fstream.h"
#include "time.h"
ofstream output("Yield.txt",ios::out|ios::app);
void main(int argc, char* argv[])
{
  long sr,r=0,count,i,j;
  int a;
  srand(time(NULL));

        for(r=10;r<100;r+=10)
        {
              sr=r*0.1;
```

```
                for(i=0;i<10;i++)

                {

                        count=0;

                        for(j=0;j<100;j++)

                        {

               char buff[128];

              sprintf(buff,"%s %d %d %d %d 0.005 %d",argv[1],r,r,sr,sr,rand());

                    a=system(buff);

                    if(a==0) count++;

                    if(a==2)

                            {

                     cout<<"Error occured"<<endl;

                     exit(1);

                            }

                        }

                        output<<count<<endl;

                        sr+=r*0.05;

                }

        }

}
```

VITA

Song Gao

Candidate for the Degree of

Master of Science

Thesis: A STUDY OF A REDUNDANT MEMORY REPAIR ALGORITHM

Major Field: Computer Science

Biographical:

Education: Received Bachelor of Medicine degree in Basic Medicine from Beijing Medical University, Beijing, China in July 1999. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in May 2003.